Proxedo Network Security Suite 2 Reference Guide

Publication date February 29, 2024

Abstract
This document is a detailed reference guide for Proxedo Network Security Suite administrators.

Table of Contents

List of Examples

List of Procedures

Preface

Welcome to the Application-level Gateway Reference Guide. This book contains reference documentation on the available PNS proxies and their working environment, the Python framework.

This book contains information about the low-level proxy attributes available to customize proxy behavior and the low-level classes comprising ALG's access control and service framework. Basic introduction to the various protocols is also provided for reference, but the detailed discussion of the protocols is beyond the scope of this book.

Note
The name of the application effectively serving as the Application-level Gateway component of Proxedo Network Security Suite is PNS, commands, paths and internal references will relate to that naming.

1. Summary of contents

Chapter 1, How PNS works (p. 1) provides an overview of the internal working of ALG, for example, how a connection is received.

Chapter 2, Configuring PNS proxies (p. 4) describes the general concepts of configuring ALG proxies.

Chapter 3, The PNS SSL framework (p. 9) explains how to handle SSL-encrypted connections with ALG.

Chapter 4, Proxies (p. 34) is a complete reference of the ALG proxies, including their special features and options.

Chapter 5, Core (p. 169) is the reference of ALG core modules which are directly used by gateway administrators, forming the access control and authentication framework.

Appendix C, PNS manual pages (p. 335) is a collection of the command-line PNS utilities.

Appendix B, Global options of PNS (p. 330) is a reference the global options of PNS.

2. Terminology

The following terms used throughout this documentation might require a brief explanation:

- *class*: A class is a set of attribute and method definitions performing certain specific functionality. Classes can inherit methods and attributes from one or more parent classes. Classes do not contain actual values for attributes; they only describe them.
- *instance*: An instance is a set of attribute values (as described by the class) and associated methods. Instances are also called objects. Instances are created from classes by "calling" the class, with arguments required by the constructor. For example, to create an instance of a class named "class" one would write `class(arg1, arg2 [,.. argN])` where arg1 and arg2 are arguments of the constructor.

- *method*: A function working in the context of an instance. It automatically receives a "self" argument which can be used to fetch or set attributes stored in the associated instance.
- *type*: Variables in Python are not strongly typed, meaning that it is possible to assign any kind of values to a variable; typing is assigned to the value.
- *attribute*: An attribute of an object is a variable holding some value, interpreted and manipulated by object methods. Although Python is not strongly typed, types were assigned to the variables in PNS to indicate what kind of values they are supposed to hold.
- *actiontuple*: A tuple is a simple Python type defined as a list of values. An actiontuple is a special tuple defined by PNS where the first value must be a value specifying what action to take, and trailing items specify arguments to the action. For example (HTTP_REQ_REJECT, "We don't like this request") is a tuple for rejecting HTTP requests and returning the message specified in the second value.

3. Target audience and prerequisites

This guide is intended for use by system administrators and consultants responsible for network security and whose task is the configuration and maintenance of PNS firewalls. PNS gives them a powerful and versatile tool to create full control over their network traffic and enables them to protect their clients against Internet-delinquency.

This guide is also useful for IT decision makers evaluating different firewall products because apart from the practical side of everyday PNS administration, it introduces the philosophy behind PNS without the marketing side of the issue.

The following skills and knowledge are necessary for a successful PNS administrator.

Skill	Level/Description
Linux	At least a power user's knowledge is required.
Experience in system administration	Experience in system administration is certainly an advantage, but not absolutely necessary.
Programming language knowledge	It is not an explicit requirement to know any programming languages though being familiar with the basics of Python may be an advantage, especially in evaluating advanced firewall configurations or in troubleshooting misconfigured firewalls.
General knowledge on firewalls	A general understanding of firewalls, their roles in the enterprise IT infrastructure and the main concepts and tasks associated with firewall administration is essential. To fulfill this requirement a significant part of *Chapter 3, Architectural overview* in the *PNS Administrator's Guide* is devoted to the introduction to general firewall concepts.
Knowledge on Netfilter concepts and IPTables	In-depth knowledge is strongly recommended; while it is not strictly required definitely helps understanding

Skill	Level/Description
	the underlying operations and also helps in shortening the learning curve.
Knowledge on TCP/IP protocol	High level knowledge of the TCP/IP protocol suite is a must, no successful firewall administration is possible without this knowledge.

Table 1. Prerequisites

4. Products covered in this guide

The PNS Distribution DVD-ROM contains the following software packages:

- Current version of PNS 2 packages.
- Current version of Management Server (MS) 2.
- Current version of Management Console (MC) 2 (GUI) for both Linux and Windows operating systems, and all the necessary software packages.
- Current version of Authentication Server (AS) 2.
- Current version of the Authentication Agent (AA) 2, the AS client for both Linux and Windows operating systems.

For a detailed description of hardware requirements of PNS, see *Chapter 1, System requirements* in *Proxedo Network Security Suite 2 Installation Guide*.

For additional information on PNS and its components visit the *PNS website* containing white papers, tutorials, and online documentations on the above products.

5. Contact and support information

This product is developed and maintained by BalaSys IT Ltd..

Contact:

BalaSys IT Ltd.
4 Alíz Street
H-1117 Budapest, Hungary
Tel: +36 1 646 4740
E-mail: `<info@balasys.hu>`
Web: *http://balasys.hu/*

5.1. Sales contact

You can directly contact us with sales related topics at the e-mail address `<sales@balasys.hu>`, or *leave us your contact information and we call you back*.

5.2. Support contact

To access the BalaSys Support System, sign up for an account at *the BalaSys Support System page*. Online support is available 24 hours a day.

BalaSys Support System is available only for registered users with a valid support package.

Support e-mail address: `<support@balasys.hu>`.

5.3. Training

BalaSys IT Ltd. holds courses on using its products for new and experienced users. For dates, details, and application forms, visit the *https://www.balasys.hu/en/services#training* webpage.

6. About this document

This guide is a work-in-progress document with new versions appearing periodically.

The latest version of this document can be downloaded from the *Documentation Page*.

6.1. Feedback

Any feedback is greatly appreciated, especially on what else this document should cover, including protocols and network setups. General comments, errors found in the text, and any suggestions about how to improve the documentation is welcome at `<support@balasys.hu>`.

Chapter 1. How PNS works

This chapter describes how PNS works, and provides information about the core PNS modules, explaining how they interoperate. For a detailed reference of the core modules, see the description of the particular in *Chapter 5, Core (p. 169)*.

- *PNS startup and initialization*: The main PNS thread is started, and the rules listening for incoming connections are initialized.
- *Handling incoming connections*: The client-side connection is established and the service to proxy the connection is selected.
- *Proxy startup and server-side connections*: The proxy instance inspecting the traffic is created and connection to the server is established.

1.1. Procedure – PNS startup and initialization

Step 1. The `velactl` utility loads the `instances.conf` file and starts the main `PNS` program. The `instances.conf` file stores the parameters of the configured PNS instances.

Step 2. `PNS` performs the following initialization steps:

- Sets the stack limit.
- Creates its PID file.
- Changes the running user to the user and group specified for the instance.
- Initializes the handling of dynamic capabilities and sets the chroot directory.
- Loads the firewall policy from the `policy.py` file.

Step 3. The `init()` of PNS initializes the ruleset defined for the PNS instance.

Step 4. The `kvela` kernel module uploads packet filtering services, rules, and zones into the kernel.

Note
PNS creates four sockets (one for each type of traffic: TCP IPv6, TCP IPv4, UDP IPv6, UDP IPv4); the kvela module directs the incoming connections to the appropriate socket.

1.2. Handling incoming connections

Incoming connections are first received by the kvela kernel module, which is actually a netfilter table. The kvela module determines the source and destination zones of the connection, and then tries to find a suitable firewall rule. If the rule points to a packet filtering service, the connection is processed according to *Procedure 1.2.1, Handling packet filtering services (p. 2)*; if it points to an application-level service, the connection is processed according to *Procedure 1.2.2, Handling application-level services (p. 2)*. If no suitable rule is found, the connection is rejected.

1.2.1. Procedure – Handling packet filtering services

Step 1. PNS generates a session ID and creates a CONNTRACK entry for the connection. This ID is based on all relevant information about the connection, including the protocol (TCP/UDP) and the client's address.
The session ID uniquely identifies the connection and is included in every log message related to this particular connection.

Step 2. Based on the parameters of the connection, the Rule selects the service that will inspect the connection.

Step 3. The Router defined in the service determines the destination address of the server.
The Router performs the following actions:

- Determines the destination address of the server.
- Sets the source address of the server-side connection, according to the `forge_address` settings of the router.

Step 4. If the client is permitted to access the selected service, the packet filter is instructed to let the connection pass PNS.

Step 5. The kvela module performs network address translation (NAT) on the connection, if needed.

1.2.2. Procedure – Handling application-level services

Step 1. For incoming connection requests that are processed on the application level, the main PNS thread establishes the connection with the client. The connection is further processed in a separate thread; the main thread is listening for new connections.

Step 2. The *Dispatcher* creates the *MasterSession* object of the connection and generates the base session ID. This object stores all relevant information of the connection, including the protocol (TCP/UDP) and the client's address.
The session ID uniquely identifies the connection and is included in every log message related to this particular connection. Other components of PNS add further digits to the session ID.

Step 3. For TCP-based connections, PNS copies the Type of Service (ToS) value of the client-PNS connection in the PNS-client connection.

Step 4. The Rule selects the service that will inspect the connection.

Step 5. The Router defined in the service determines the destination address of the server. The result is stored in the Session object, where the Chainer can access it later.
The Router performs the following actions:

- Determines the destination address of the server.
- Sets the source address of the server-side connection (according to the `forge_port`, `forge_address` settings of the router).
- Sets the ToS value of the server-side connection.

Step 6. If the client is permitted to access the selected service, the `startInstance()` method of the service is started. The `startInstance()` method performs the following actions:

- Verifies that the new instance does not exceed the number of instances permitted for the service (*max_instances* parameter).
- Creates the final session ID.
- Creates an instance of the proxy class associated with the service. This proxy instance is associated with a *StackedSession* object. The startup of the proxy is detailed in *Procedure 1.3, Proxy startup and the server-side connection (p. 3)*.

1.3. Procedure – Proxy startup and the server-side connection

Step 1. To create an instance of the application-level proxy, the `__init__` constructor of the proxy class calls the `Proxy.__init__` function of the *Proxy* module. The proxy instance is created into a new thread from the PNSProxy ancestor class.

Step 2. From the new thread, the proxy loads its configuration.

Step 3. The proxy initiates connection to the server.

> **Note**
> Some proxies connect the server only after receiving the first client request.

Step 4. The *Proxy.connectServer()* method creates the server-side connection using the *Chainer* assigned to the service. The Chainer performs the following actions:

- Reads the parameters related to the server-side connection from the *Session* object. These parameters were set by the *Router* and the Proxy.
- Performs source and destination network address translation. This may modify the addresses set by the Router and the Proxy.
- Verifies that access to the server is permitted.
- Establishes the connection using the Attach subsystem, and passes to the proxy the stream that represents the connection.

> **Note**
> The *Proxy.connectServer()* method connects stacked proxies with their parent proxies.

Chapter 2. Configuring PNS proxies

This chapter describes how PNS proxies work in general, and how to configure them.

- For the details on configuring TLS/SSL connections, see *Chapter 2, Configuring PNS proxies (p. 4)*.
- For a complete reference of the available PNS proxies, see *Chapter 4, Proxies (p. 34)*.

2.1. Policies for requests and responses

PNS offers great flexibility in proxy customization. Requests and commands, responses, headers, etc. can be managed individually in PNS. This means that it is not only possible to enable/disable them one-by-one, but custom actions can be assigned to them as well. The available options are listed in the description of each proxy, but the general guidelines are discussed here.

All important events of a protocol have an associated policy hash: usually there is one for the requests or commands and one for the responses. Where applicable for a protocol, there are other policy hashes defined as well (e.g., for controlling the capabilities available in the IMAP protocol, etc.). The entries of the hash are the possible events of the protocol (e.g., the request hash of the FTP protocol contains the possible commands - RMD, DELE, etc.) and an action associated with the event - what PNS should do when this event occurs. The available actions may slightly vary depending on the exact protocol and the hash, but usually they are the following:

Action	Description
ACCEPT	Enable the event; the command/response/etc. can be used and is allowed through the firewall.
REJECT	Reject the event and send an error message. The event is blocked and the client notified. The communication can continue, the connection is not closed.
DROP	Reject the event without sending an error message. The event is blocked but the client is not notified. The communication can continue, the connection is not closed. In some cases (depending on the protocol) this action is able to remove only a part of the message (e.g., a particular header in HTTP traffic) without rejecting the entire message.
ABORT	Reject the event and terminate the connection.
POLICY	Call a Python function to make a decision about the event. The final decision must be one of the above actions (i.e. POLICY is not allowed). The parameters received by the function are listed in the module

Action	Description
	descriptions. See the examples below and in the module descriptions for details.

Table 2.1. Action codes for protocol events

The use of the policy hashes and the action codes is illustrated in the following examples.

Example 2.1. Customizing FTP commands
In this example the 'RMD' command is rejected, and the connection is terminated if the user attempts to delete a file.

```
class MyFtp(FtpProxy):
 def config(self):
  self.request["RMD"] = (FTP_REQ_REJECT)
  self.request["DELE"] = (FTP_REQ_ABORT)
```

Example 2.2. Using the POLICY action
This example calls a function called pUser (defined in the example) whenever a USER command is received within an FTP session. All other commands are accepted. The parameter of the USER command (i.e. the username) is examined: if it is 'user1' or 'user2', the connection is accepted, otherwise it is rejected.

```
class MyFtp(FtpProxy):
 def config(self):
  self.request["USER"] = (FTP_REQ_POLICY, self.pUser)
  self.request["*"] = (FTP_REQ_ACCEPT)

 def pUser(self,command):
  if self.request_parameter == "user1" or self.request_parameter == "user2":
   return FTP_REQ_ACCEPT
  return FTP_REQ_REJECT
```

It must be noted that there is a difference between how PNS processes the POLICY actions and all the other ones (e.g., ACCEPT, DROP, etc.). POLICY actions are evaluated on the policy (or Python) level of PNS, while the other ones on the proxy (or C) level. Since the proxies of PNS are written in C, and operate on the proxy level, the evaluation of POLICY actions is slightly slower, but this can be an issue only in very high-throughput environments with complex policy settings.

2.1.1. Default actions

Default actions for all events of a hash (e.g., all requests) can be set using the '*' wildcard as the event. (Most hashes have default actions configured by default, these can be found in the description of the proxy classes.) It is important to note that setting the action using the '*' wildcard does NOT override an action explicitly defined for an event, even if the explicit setting precedes the general one in the Python code. This feature is illustrated in the example below.

Example 2.3. Default and explicit actions
The following two proxy classes have the same effect, even though the order of the code lines is switched. The 'APPE' command is rejected, while all other commands are accepted.

```
class MyFtp1(FtpProxy):
 def config(self):
  self.request["APPE"] = (FTP_REQ_REJECT)
  self.request["*"] = (FTP_REQ_ACCEPT)
```

```
class MyFtp2(FtpProxy):
 def config(self):
  self.request["*"] = (FTP_REQ_ACCEPT)
  self.request["APPE"] = (FTP_REQ_REJECT)
```

Warning
If the relevant hash does not contain a received request or response, the '*' entry is used which matches to every request/response. If there is no '*' entry in the given hash, the request/response is denied.

2.1.2. Response codes

Responses in certain protocols include numeric response codes, e.g., in the FTP protocol responses start with a three-digit code. In PNS it is possible to filter these codes as well, furthermore, to filter them based on the command to which the response arrives to. In these cases the hash contains both the command and the answer, and an action as well. The '*' wildcard character can be used to match for every command or response code.

Example 2.4. Customizing response codes
The following example accepts the response '250' only to the 'DELE' command, but allows any response code to the 'LIST' command.

```
class MyFtp1(FtpProxy):
 def config(self):
  self.response["DELE", "250"] = (FTP_RSP_ACCEPT)
  self.response["*", "250"] = (FTP_RSP_REJECT)
  self.response["LIST", "*"] = (FTP_RSP_ACCEPT)
```

It is not necessary to specify the full response code, it is also possible to specify only the first, or the first two digits.

For example, all three response codes presented below are valid, but have different effects:

- "PWD","200"
  Match exactly the answer 200 coming in a reply to a PWD command.

- "PWD","2"
  Match every answer starting with '2' in a reply to a PWD command.

- "*","20"
  Match every answer between 200 and 209 in a reply to any command.

This kind of response code lookup is available in the following proxies: FTP, HTTP, and SMTP. The precedence how the hash table entries are processed is the following:

1. Exact match. ("PWD","200")
2. Exact command match, partial response matches ("PWD","20"; "PWD","2"; "PWD","*")
3. Wildcard command, with answer codes repeated as above. ("*","200"; "*","20"; "*","2")
4. Wildcard for both indexes. ("*","*")

2.2. Secondary sessions

Certain proxies support the use of secondary sessions, i.e. several sessions using the same proxy instance (the same thread), effectively reusing proxy instances. As new sessions enter the proxy via a fastpath, using secondary sessions can significantly decrease the load on the firewall.

When a new connection is accepted, PNS looks for the appropriate proxy instance which is willing to accept secondary sessions. If there is none, a new proxy instance is started. An already running proxy instance is appropriate if it is willing to accept secondary channels and the criteria about secondary sessions are met. (The criteria can be specified in the configuration of the proxy class.)

The criteria are set via the *secondary_mask* attribute, while the number of secondary sessions allowed within the same instance is controlled by the *secondary_sessions* attribute. The *secondary_mask* attribute is an integer specifying which properties of an established session are considered to be important. If all important properties match, the connection can be handled as a secondary session by a proxy instance accepting secondary sessions, provided the new session does not exceed the limit set in *secondary_sessions*. The *secondary_mask* attribute is actually a bitfield interpreted as follows: bit 0 means source address; bit 1 means source port; bit 2 means destination address; bit 3 means destination port.

Currently the Plug, RADIUS, and Sip proxies support the use of secondary sessions.

Example 2.5. Example PlugProxy allowing secondary sessions
This example allows 100 parallel sessions in one proxy thread if the IP address and Port of the targets are the same.

```
class MyPlugProxy(PlugProxy):
 def config(self):
  PlugProxy.config(self)
  self.secondary_mask = 0xC
  self.secondary_sessions = 100
```

2.3. Embedded protocol analysis

Each protocol proxy available in PNS inspects the traffic for conformance to the given protocol. Often further analysis of the data transferred via the protocol is required, this can be accomplished via stacking. Stacking is a method when the data transferred in the protocol is passed to another proxy or program. After performing the inspection, the stacked proxy or program returns the data to the original proxy, which resumes its transmission.

2.3.1. Proxy stacking

Proxy stacking is mainly used to inspect embedded protocols, or perform virus filtering: e.g., to inspect the parts of e-mail messages, the mail transport protocol is examined with an Smtp proxy, and then a MIME proxy is stacked to inspect MIME-formatted mail messages. It is possible to stack several layers of proxies into each other if needed, e.g., in the above example, a further virus filtering solution (like a CF module) could be stacked into the MIME proxy.

Note
Every proxy is able to handle SSL/TLS-encypted connection on its own. This feature greatly decreases the need of proxy stacking, making it needed only in special cases, for example, to inspect HTTP traffic tunneled in SSH.

Stacking a proxy to inspect the embedded protocol is possible via the *`self.request_stack`* attribute; if another attribute has to be used, it is noted in the description of the given proxy. The HTTP proxy is special in the sense that it is possible to stack different proxies into the requests and the responses.

The parameters of the stack attribute has to specify the following:

- The protocol elements for which embedded inspection is required. This parameter can be used to specify if all received data should be passed to the stacked proxy ("*"), or only the data related (sent or received) to specific protocol elements (e.g., only the data received with a GET request in HTTP).
- The mode how the data is passed to the stacked proxy. This parameter governs if only the data part should be passed to the stacked proxy (XXXX_STK_DATA, where XXXX depends on the protocol), or (if applicable) MIME header information should be included as well (XXXX_STK_MIME) to make it possible to process the data body as a MIME envelope. Please note that while it is possible to change the data part in the stacked proxy, it is not possible to change the MIME headers - they can be modified only by the upper level proxy. The available constants are listed in the respective protocol descriptions. The default value for this argument is XXXX_STK_NONE, meaning that no data is transferred to the stacked proxy. In some proxies it is also possible to call a function (using the XXXX_STK_POLICY action) to decide which part (if any) of the traffic should be passed to the stacked proxy.
- The proxy class that will perform inspection of the embedded protocol.

For additional information on proxy stacking, see *Section 6.6.3, Analyzing embedded traffic* in *Proxedo Network Security Suite 2 Administrator Guide*, and the various tutorials available at the BalaSys Documentation Page.

2.3.2. Program stacking

When stacking a program, the data received by a proxy within a protocol is directed to the standard input. Arbitrary commands (including command line scripts, or applications) working from the standard input can be run on this data stream. The original proxy obtains the processed data back from the standard output. When stacking a command, the command to be called has to be included in the proper stack attribute of the proxy between double-quotes. This is illustrated in the following example.

Example 2.6. Program stacking in HTTP
In this example a simple 'sed' (stream editor) command is stacked into the HTTP proxy to replace all occurrences of 'http' to 'https', thus securing the HTTP connections on one side of the firewall.

```
class MyHttp(HttpProxy):
 def config(self):
  HttpProxy.config(self)
  self.response_stack["GET"] = /
  (HTTP_STK_DATA, (V_STACK_PROGRAM, "/bin/sed '/http:/s//https:/g'"))
```

Chapter 3. The PNS SSL framework

This chapter describes the SSL protocol and the SSL framework available for every Application-level Gateway proxy.

3.1. The SSL and TLS protocols

Secure Socket Layer v3 (SSL) and Transport Layer Security v1 (TLS) are widely used crypto protocols guaranteeing data integrity and confidentiality in many PKI and e-commerce systems. They allow both the client and the server to authenticate each other. SSL/TLS use a reliable TCP connection for data transmission and cooperate with any application-level protocol. SSL/TLS guarantee that:

- Communication in the channel is private, only the other communicating party can decrypt the messages.
- The channel is authenticated, so the client can make sure that it communicates with the right server. Optionally, the server can also authenticate the client. Authentication is performed via certificates issued by a Certificate Authority (CA). Certificates identify the owner of an encryption keypair used in encrypted communication.
- The channel is reliable, which is ensured by message integrity verification using MAC.

SSL/TLS is almost never used in itself: it is used as a secure channel to transfer other, less secure protocols. The protocols most commonly embedded into SSL/TLS are HTTP and POP3 (i.e. these are the HTTPS and POP3S protocols).

3.1.1. Procedure – The SSL handshake

As an initial step, both the client and the server collect information to start the encrypted communication.

Step 1. The client sends a CLIENT-HELLO message.

Step 2. The server answers with a SERVER-HELLO message containing the certificate of the server. At this point the parties determine if a new master key is needed.

> **Note**
> The server stores information (including the session ID and other parameters) about past SSL/TLS sessions in its session cache. Clients that have contacted a particular server previously can request to continue a session (by identifying its session ID); this can be used to accelerate the initialization of the connection. Application-level Gateway currently does not support this feature, but this does not cause any noticeable difference to the clients.

Step 3. The client verifies the server's certificate. If the certificate is invalid the client sends an ERROR message to the server.

> **Note**
> If a new master key is needed the client gets the server certificate from the SERVER-HELLO message and generates a master key, sending it to the server in a CLIENT-MASTER-KEY message.

Step 4. The server sends a SERVER-VERIFY message, which authenticates the server itself.

Step 5. Optionally, the server can also authenticate the client by requesting the client's certificate with a REQUEST-CERTIFICATE message.

Step 6. The server verifies the certificate received from the client and finishes the handshake with a SERVER-FINISH message.

Note
In SSL two separate session keys are used, one for outgoing communication (which is of course incoming at the other end), and another key for incoming communication. These are known as SERVER/CLIENT-READ-KEY and SERVER/CLIENT-WRITE-KEY.

3.2. Handling TLS and SSL connections in Application-level Gateway

PNS has a common framework that allows every Application-level Gateway proxy to use SSL/TLS encryption, and - in some cases - also supports STARTTLS.

Note
Currently, the following proxies support STARTTLS: Ftp proxy (to start FTPS sessions), Smtp proxy.

3.2.1. Behavior of the SSL framework

The SSL framework inspects SSL/TLS connections, and also any other connections embedded into the encrypted SSL/TLS channel. SSL/TLS connections initiated from the client are terminated on the firewall, and two separate SSL/TLS connections are built: one between the client and the firewall, and one between the firewall and the server. If both connections match the configuration settings of Application-level Gateway (for example, the certificates are valid, and only the allowed encryption algorithms are used), Application-level Gateway inspects the protocol embedded into the secure channel as well. Note that the configuration settings can be different for the two connections, for example, it is possible to permit different protocol versions and encryption settings.

When a firewall rule matches an incoming connection, Application-level Gateway starts the Service specified in the firewall rule to inspect the connection. The Encryption policy set in the Service determines the encryption settings used in the connection.

- For the details of the attributes related to the SSL framework, see *Section 5.5, Module Encryption (p. 195)*.
- Several configuration examples and considerations are discussed in the Technical White Paper and Tutorial *Proxying secure channels - the Secure Socket Layer*, available at the BalaSys Documentation Page.

Depending on the scenario (TwoSidedEncryption, ClientOnlyEncryption, and so on) set in the Encryption policy, the SSL framework selects the first peer to perform the SSL handshake with.

As part of the handshake process, Application-level Gateway checks if encryption is required on the given side. It is not necessary for SSL to be enabled on both sides - Application-level Gateway can handle one-sided SSL connections as well (for example, the firewall communicates in an unencrypted channel with the client, but in a secure channel with the server). If SSL is not enabled, the handshake is skipped for that side.

When SSL is needed, the Service collects the required parameters (keys, certificates, and so on) from the Encryption policy.

The SSL handshake is slightly different for the client (in this case Application-level Gateway behaves as an SSL server) and the server (when Application-level Gateway behaves as an SSL client):

- **Client-side (SSL server) behavior.** In the client-side connection Application-level Gateway acts as an SSL server, and shows the client a certificate.
  If peer authentication is enabled (that is, the *`required`* and *`trust_level`* attributes of the verifier used in the Encryption policy is properly set), Application-level Gateway sends a list of trusted CAs to the client. If the client returns a certificate, Application-level Gateway verifies it against the trusted CA list and their associated revocation lists, and also checks the validity of the certificate.

- **Server-side (SSL client) behavior.** The server-side handshake is similar to the client-side handshake only the order of certificate verification is different. On the server side, Application-level Gateway verifies the server's certificate first, and then sends its own certificate for verification.

3.2.2. Session reuse in SSL and TLS connections

Starting with version 6.0, PNS supports session reuse in SSL and TLS connections. PNS supports both session identifiers (*RFC 8446*) and session tickets (*RFC 8446*). Note that session tickets can be used only in TLS connections. Unless explicitly disabled in the configuration of the Encryption policy (for details, see *Section 5.5, Module Encryption (p. 195)*), PNS attempts to use session tickets, and automatically falls back to using session identifiers if needed.

3.2.3. Understanding Encryption policies

This section describes the configuration blocks of Encryption policies and objects used in Encryption policies. Encryption policies were designed to be flexible, and make encryption settings easy to reuse in different services.

An **Encryption policy** is an object that has a unique name, and references a fully-configured **encryption scenario**.

Encryption scenarios are actually Python classes that describe how encryption is used in a particular connection, for example, both the server-side and the client-side connection is encrypted, or the connection uses a one-sided SSL connection, and so on. Encryption scenarios also reference other classes that contain the actual settings for the scenario. Depending on the scenario, the following classes can be set for the client-side, the server-side, or both.

- **Certificate generator**: It creates or loads an X.509 certificate that Application-level Gateway shows to the peer. The certificate can be a simple certificate (*Section 5.5.23, Class StaticCertificate (p. 233)*), a dynamically generated certificate (for example, used in a keybridging scenario, *Section 5.5.12, Class DynamicCertificate (p. 215)*), or a list of certificates to support Server Name Indication (SNI, *Section 5.5.18, Class SNIBasedCertificate (p. 225)*).
  The related parameters are: *`client_certificate_generator`*, *`server_certificate_generator`*

- **Certificate verifier**: The settings in this class determine if Application-level Gateway requests a certificate of the peer and the way to verify it. Application-level Gateway has separate built-in classes for the client-side and the server-side verification settings: *Section 5.5.6, Class ClientCertificateVerifier (p. 204)* and *Section 5.5.19, Class ServerCertificateVerifier (p. 226)*. For details and examples, see *Section 3.2.5, Certificate verification options (p. 23)*.
  The related parameters are: `client_verify`, `server_verify`

- **Protocol settings**: The settings in this class determine the protocol-level settings of the SSL/TLS connection, for example, the permitted ciphers and protocol versions, session-reuse settings, and so on. Application-level Gateway has separate built-in classes for the client-side and the server-side SSL/TLS settings: *Section 5.5.10, Class ClientTLSOptions (p. 210)* and *Section 5.5.22, Class ServerTLSOptions (p. 230)*. For details and examples, see *Section 3.2.6, Protocol-level TLS settings (p. 24)*.
  The related parameters are: `client_tls_options`, `server_tls_options`

Application-level Gateway provides the following built-in encryption scenarios:

- **TwoSidedEncryption**: Both the client-Application-level Gateway and the Application-level Gateway-server connections are encrypted. For details, see *Section 5.5.25, Class TwoSidedEncryption (p. 237)*.
- **ClientOnlyEncryption**: Only the client-Application-level Gateway connection is encrypted, the Application-level Gateway-server connection is not. For details, see *Section 5.5.8, Class ClientOnlyEncryption (p. 207)*.
- **ServerOnlyEncryption**: Only the Application-level Gateway-server connection is encrypted, the client-Application-level Gateway connection is not. For details, see *Section 5.5.21, Class ServerOnlyEncryption (p. 229)*.
- **ForwardStartTLSEncryption**: The client can optionally request STARTTLS encryption. For details, see *Section 5.5.16, Class ForwardStartTLSEncryption (p. 221)*.
- **ClientOnlyStartTLSEncryption**: The client can optionally request STARTTLS encryption, but the server-side connection is always unencrypted. For details, see *Section 5.5.9, Class ClientOnlyStartTLSEncryption (p. 208)*.
- **FakeStartTLSEncryption**: The client can optionally request STARTTLS encryption, but the server-side connection is always encrypted. For details, see *Section 5.5.15, Class FakeStartTLSEncryption (p. 219)*.

For example, on configuring Encryption policies, see *How to configure TLS proxying in PNS 2*. For details on HTTPS-specific problems and the related solutions, see *How to configure HTTPS proxying in PNS 2*.

3.2.4. Configuring Encryption policies

To configure Encryption policies, you have to create an Encryption policy, and derive and configure your own scenario from the available built-in scenarios. To configure a scenario, you have to derive and configure your own certificate generator, certificate verifier, and protocol settings classes. (Do not change the built-in classes directly, because that changes the default behavior of Application-level Gateway, and can have unexpected and unwanted effects on the configuration of Application-level Gateway.)

Note
If the built-in scenarios do not cover your particular use-case, derive an own class from TwoSidedEncryption, and configure it to suit your needs.

For a details on configuring Encryption Policies, see the following procedure, or the *How to configure TLS proxying in PNS 2* tutorial.

3.2.4.1. Procedure – Enabling TLS-encryption in the connection

Purpose:

To proxy HTTPS connections, configure an Encryption Policy to handle TLS connections, and use this Encryption Policy in your Service. The policy will be configured to:

- Require the client and the server to use strong encryption algorithms, the use of weak algorithms will not be permitted.
- Enable connections only to servers with certificates signed by CAs that are in the trusted CAs list of the PNS firewall node. (For details on managing trusted CA groups, see *Section 11.3.7.3, Managing trusted groups* in *Proxedo Network Security Suite 2 Administrator Guide*.)
- The clients will only see the certificate of PNS. To allow the clients to access the certificate information of the server, see *Procedure 2.2, Configuring keybridging* in *How to configure TLS proxying in PNS 2*.

Steps:

Step 1. Generate a certificate for your firewall. The Application-level Gateway component requires its own certificate and keypair to perform TLS proxying.
MC: Create a certificate, set the firewall as the owner host of the certificate, then distribute it to the firewall host. For details, see *Chapter 11, Key and certificate management in PNS* in *Proxedo Network Security Suite 2 Administrator Guide*.

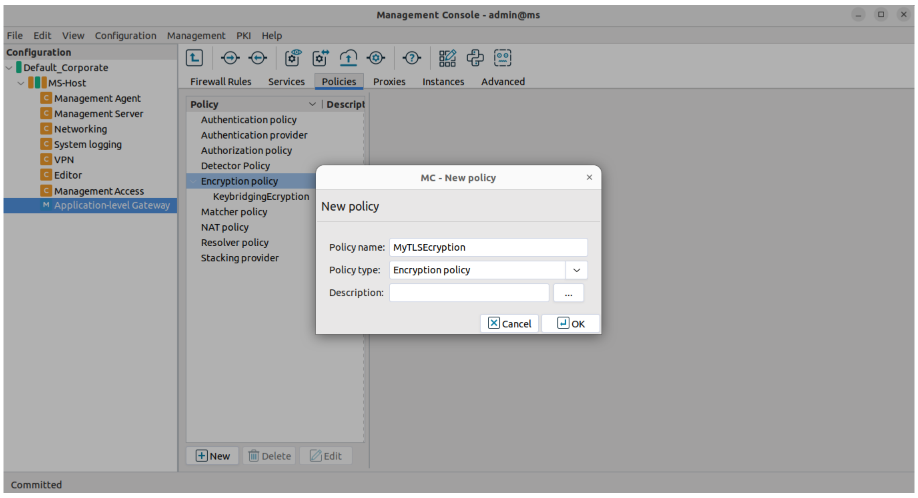
Python: In configurations managed manually from python, create an X.509 certificate (with its related keypair) using a suitable software and deploy it to the PNS firewall host (for example, copy it to the `/etc/key.d/mycert` folder).

Step 2. Create and configure an Encryption Policy. Complete the following steps.

Step a. Navigate to the **Application-level Gateway** MC component of the firewall host.

Step b. Select **Policies** > **New**.

Step c. Enter a name into the **Policy name** field, for example, `MyTLSEncryption`.



MC - New policy

New policy

Policy name: MyTLSEcryption

Policy type: Encryption policy

Description:

Cancel OK

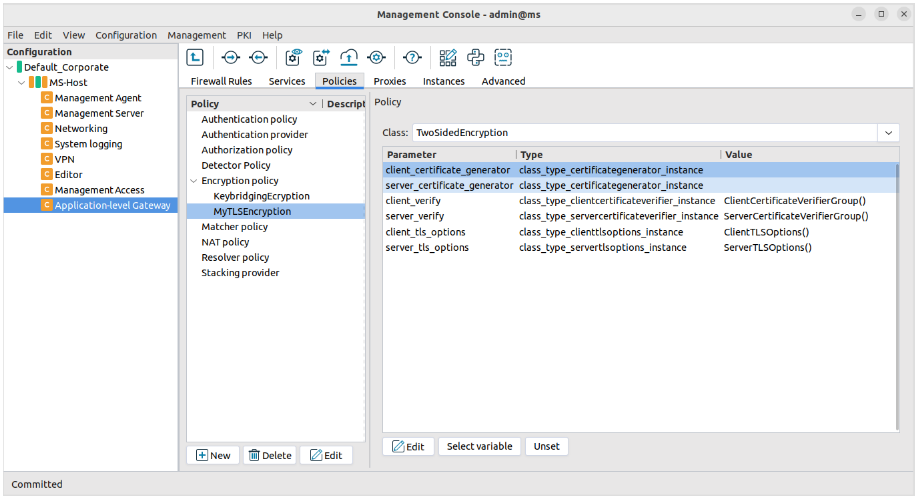
Figure 3.1. Creating a new Encryption policy

Step d. Select **Policy type** > **Encryption Policy**, then click OK.

Step e. Select **Class** > **TwoSidedEncryption**.
Python:

```
EncryptionPolicy(
    name="MyTLSEncryption",
    encryption=TwoSidedEncryption()
    )
```

Step f. Double-click **client_certificate_generator**, then select **Class** > **StaticCertificate**.



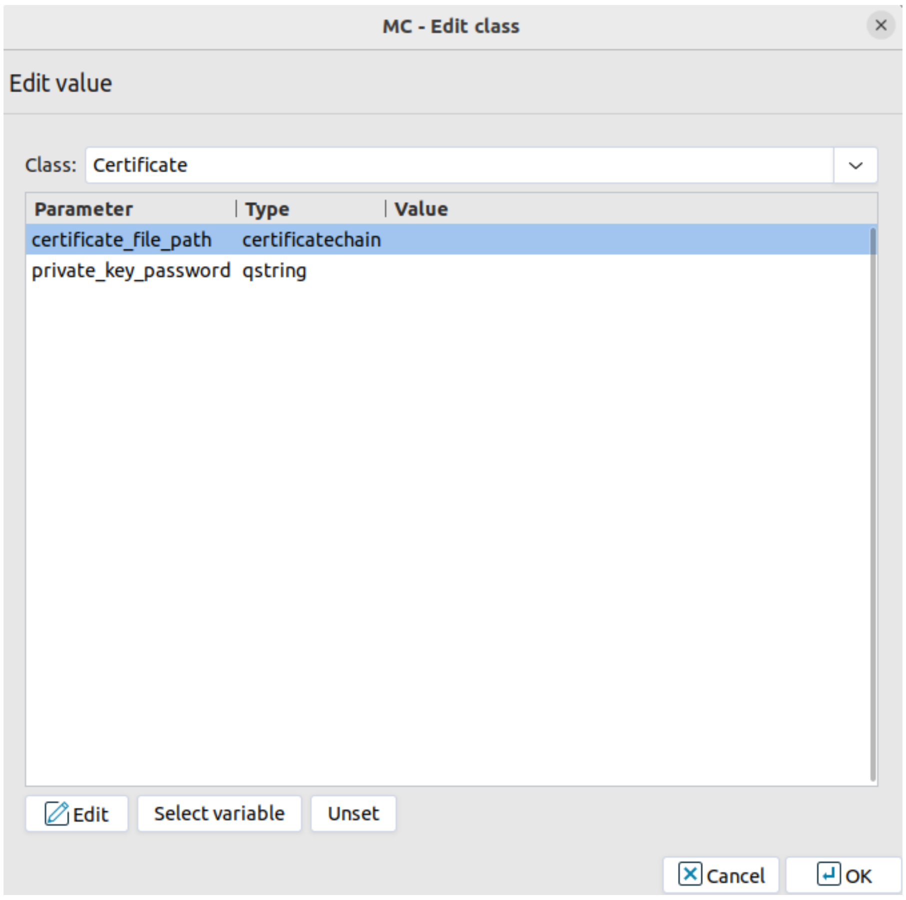
Policy

Class: TwoSidedEncryption

Parameter	Type	Value
client_certificate_generator	class_type_certificategenerator_instance	
server_certificate_generator	class_type_certificategenerator_instance	
client_verify	class_type_clientcertificateverifier_instance	ClientCertificateVerifierGroup()
server_verify	class_type_servercertificateverifier_instance	ServerCertificateVerifierGroup()
client_tls_options	class_type_clienttlsoptions_instance	ClientTLSOptions()
server_tls_options	class_type_servertlsoptions_instance	ServerTLSOptions()

Figure 3.2. Selecting Encryption policy class

Step g. Double-click the **certificates** and click **New** to add a certificate entry to a list of certificates.

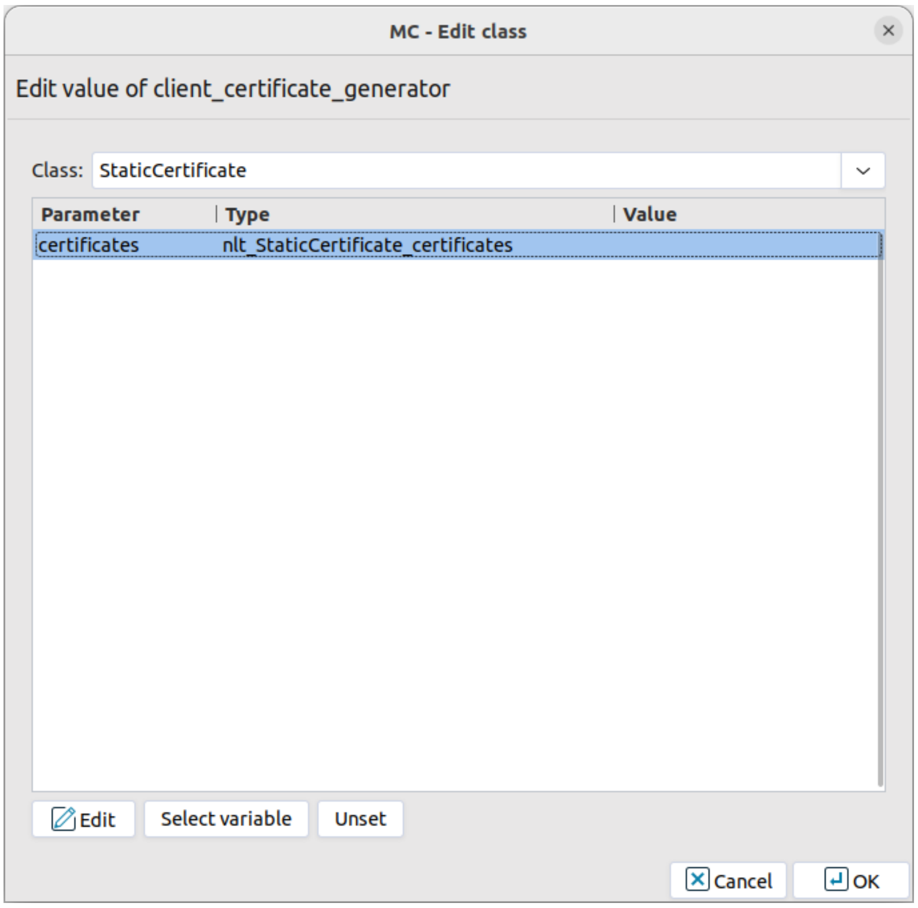


Class: Certificate

Parameter	Type	Value
certificate_file_path	certificatechain	
private_key_password	qstring	

Figure 3.3. Creating a new certificate entry

Step h. Double-click the **certificate_file_path**. A window displaying the certificates owned by the host will open up. The lower section of the window shows the information contained in the certificate. Select the list of certificates Application-level Gateway is required to show to the clients (for example, the certificate created in Step 1), then click **Select**.


MC - Edit class

Edit value of client_certificate_generator

Class: StaticCertificate

Parameter	Type	Value
certificates	nlt_StaticCertificate_certificates	

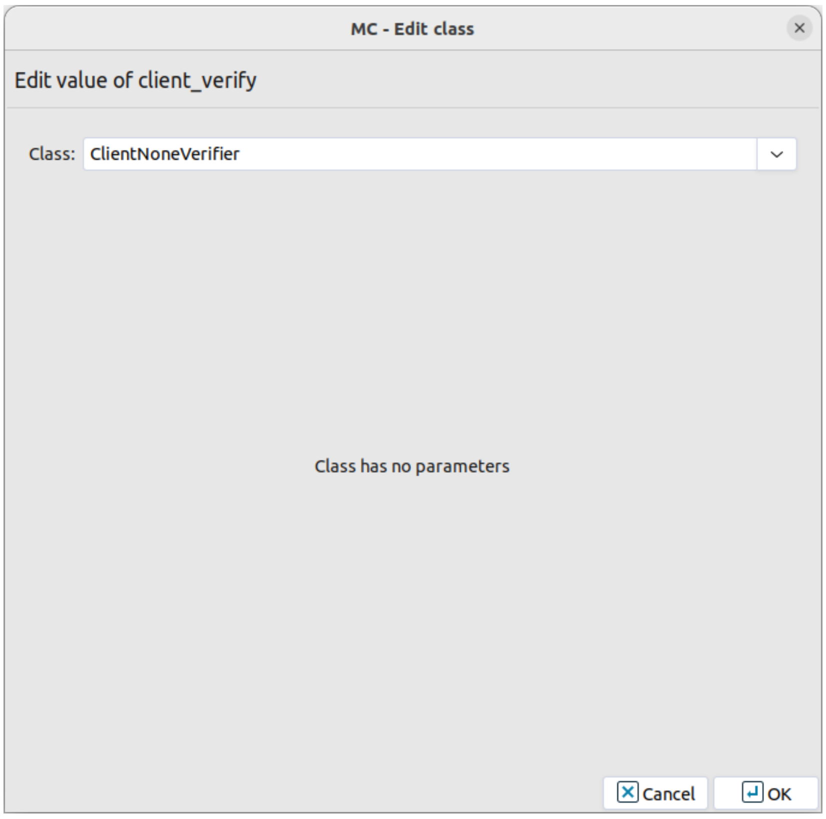
Figure 3.4. Creating a new Encryption policy

Python:

```
    encryption=TwoSidedEncryption(
        client_certificate_generator=StaticCertificate(
            certificates=(
                Certificate.fromFile(

certificate_file_path="/etc/key.d/VMS_Engine/cert.pem",
                    private_key=PrivateKey.fromFile(
                        "/etc/key.d/VMS_Engine/key.pem")
                ),
            )
        )
    )
```

Step i. If the private key of the certificate is password-protected, double-click **private_key_password**, type the password, then click OK. Otherwise, click OK.

Step j. Disable mutual authentication. That way, Application-level Gateway will not request a certificate from the clients.
Double-click **client_verify**, select **Class** > **ClientNoneVerifier**, then click OK.



Edit value of client_verify

Class: ClientNoneVerifier

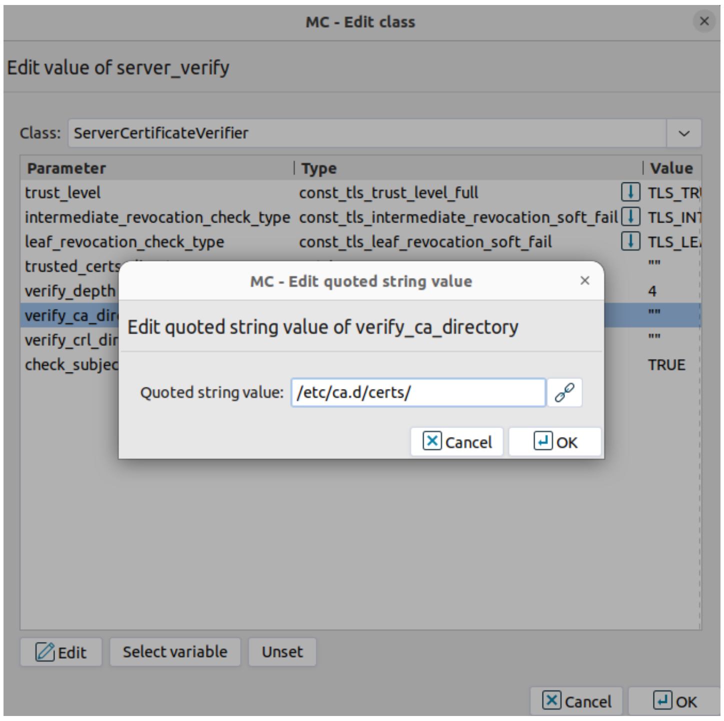
Class has no parameters

Figure 3.5. Disabling mutual authentication

Python:

```
    encryption=TwoSidedEncryption(
        client_verify=None
    )
```

Step k. Specify the directory containing the certificates of the trusted CAs. These settings determine which servers can the clients access: the clients will be able to connect only those servers via TLS which have certificate signed by one of these CAs (or a lower level CA in the CA chain).
Double-click **server_verify**, double-click **verify_ca_directory**, then type the path and name to the directory that stores the trusted CA certificates, for example, `/etc/ca.d/certs/`. Click OK.


MC - Edit class

Edit value of server_verify

Class: ServerCertificateVerifier

Parameter	Type	Value
trust_level	const_tls_trust_level_full	TLS_TR
intermediate_revocation_check_type	const_tls_intermediate_revocation_soft_fail	TLS_INT
leaf_revocation_check_type	const_tls_leaf_revocation_soft_fail	TLS_LE
trusted_certs		""
verify_depth		4
verify_ca_dir		""
verify_crl_dir		""
check_subjec		TRUE

MC - Edit quoted string value

Edit quoted string value of verify_ca_directory

Quoted string value: /etc/ca.d/certs/

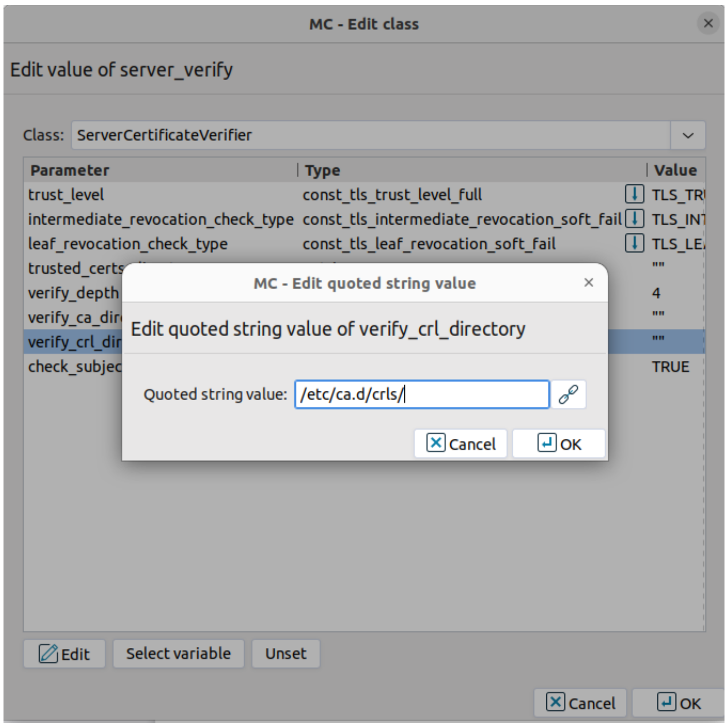
Figure 3.6. Specifying trusted CAs

Python:

```
    encryption=TwoSidedEncryption(
        server_verify=ServerCertificateVerifier(
            verify_ca_directory="/etc/ca.d/certs/"
        )
    )
```

Note
CAs cannot be referenced directly, only the trusted group containing them. For details on managing trusted groups, see Section 11.3.7.3, Managing trusted groups in *Proxedo Network Security Suite 2 Administrator Guide*.

Step l. Specify the directory containing the CRLs of the trusted CAs.
Double-click **verify_crl_directory**, then type the path and name to the directory that stores the CRLs of the trusted CA certificates, for example, `/etc/ca.d/crls/`. Click OK.


MC - Edit class
Edit value of server_verify
Class: ServerCertificateVerifier
MC - Edit quoted string value
Edit quoted string value of verify_crl_directory
Quoted string value: /etc/ca.d/crls/
Cancel
OK
Edit
Select variable
Unset
Cancel
OK

Parameter	Type	Value
trust_level	const_tls_trust_level_full	TLS_TR
intermediate_revocation_check_type	const_tls_intermediate_revocation_soft_fail	TLS_INT
leaf_revocation_check_type	const_tls_leaf_revocation_soft_fail	TLS_LE
trusted_certs		""
verify_depth		4
verify_ca_dir		""
verify_crl_dir		""
check_subjec		TRUE

Figure 3.7. Specifying CRLs

Python:

```
    encryption=TwoSidedEncryption(
        server_verify=ServerCertificateVerifier(
            verify_ca_directory="/etc/ca.d/certs/",
            verify_crl_directory="/etc/ca.d/crls/"
        )verify_
    )
```

Step m. *Optional Step*: The Common Name in the certificate of a server or webpage is usually its domain name or URL. By default, Application-level Gateway compares this Common Name to the actual domain name it receives from the server, and rejects the connection if they do not match. That way it is possible to detect several types of false certificates and prevent a number of phishing attacks. If this mode of operation interferes with your environment, and you cannot use certificates that have proper Common Names, disable this option.

Double-click **server_verify** > **check_subject**, select *FALSE*, then click OK.

Python:

```
    encryption=TwoSidedEncryption(
        server_verify=ServerCertificateVerifier(
            verify_ca_directory="/etc/ca.d/certs/",
            verify_crl_directory="/etc/ca.d/crls/",
            check_subject=FALSE
        )
    )
```

Step n. *Optional Step*: Forbid the use of weak encryption algorithms to increase security. The related parameters can be set separately for the client and the server-side of Application-level Gateway, using the **client_tls_options** and **server_tls_options** parameters of the Encryption Policy. Disabling weak algorithms also eliminates the risk of downgrade attacks, where the attacker modifies the TLS session-initiation messages to force using weak encryption that can be easily decrypted by a third party.

> **Note**
> Certain outdated operating systems, or old browser applications do not properly support strong encryption algorithms. If your clients use such systems or applications, it might be required to permit weak encryption algorithms.

Step o. *Optional Step*: Enable untrusted certificates. Since a significant number of servers use self-signed certificates (with unverifiable trustworthiness), in certain situations it might be needed to permit access to servers that have untrusted certificates.

> **Note**
> When an untrusted certificate is accepted, the generated certificates will be signed with the untrusted CA during keybridge scenarios. For details on configuring keybridging, see *Procedure 2.2, Configuring keybridging* in *How to configure TLS proxying in PNS 2*

Double-click **server_verifier** > **trust_level**, click the drop-down menu and select `UNTRUSTED`, then click OK.

> **Note**
> When the **trust_level** value is *`NONE`*, even the invalid certificates are accepted and at the client side there is no client certificate request sent to the client.

Python:

```
    encryption=TwoSidedEncryption(
        server_verify=ServerCertificateVerifier(
            trust_level=TLS_TRUST_LEVEL_UNTRUSTED
        )
    )
```

Python:

The Encryption Policy configured in the previous steps is summarized in the following code snippet.

```
EncryptionPolicy(
    name="MyTLSEncryption",
    encryption=TwoSidedEncryption(
        client_verify=ClientNoneVerifier(),
        client_tls_options=ClientTLSOptions(),
        server_verify=ServerCertificateVerifier(
            trust_level=TLS_TRUST_LEVEL_FULL,
            intermediate_revocation_check_type =
                TLS_INTERMEDIATE_REVOCATION_SOFT_FAIL,
            leaf_revocation_check_type =
                TLS_LEAF_REVOCATION_SOFT_FAIL,
            trusted_certs_directory="",
            verify_depth=4,
            verify_ca_directory="/etc/ca.d/certs/",
            verify_crl_directory="/etc/ca.d/crls/",
            check_subject=TRUE
            ),
        server_tls_options=ServerTLSOptions(),
        client_certificate_generator=StaticCertificate(
            certificates=(
                Certificate.fromFile(
                    certificate_file_path=
                        "/etc/key.d/VMS_Engine/cert.chain.pem",
                    private_key=PrivateKey.fromFile(
                        "/etc/key.d/VMS_Engine/key.pem")),
            ))
    ))
```

Step 3. Select **PKI** > **Distribute Certificates**.
Note when managing PNS without MC, copy the certificates and CRLs to their respective directories. They are not updated automatically as in configurations managed by MC.

By performing the above steps, the proxy has been configured to use the specified certificate and its private key, and also the directory has been set that will store the certificates of the trusted CAs and their CRLs. Client authentication has also been disabled.

Step 4. Create a service that clients can use to access the Internet in a secure channel. This service will use the `MyTLSEncryption` Encryption Policy.

Step a. Select **Services** > **New**, enter a name for the service (for example, *`intra_HTTPS_inter`*), then click OK.

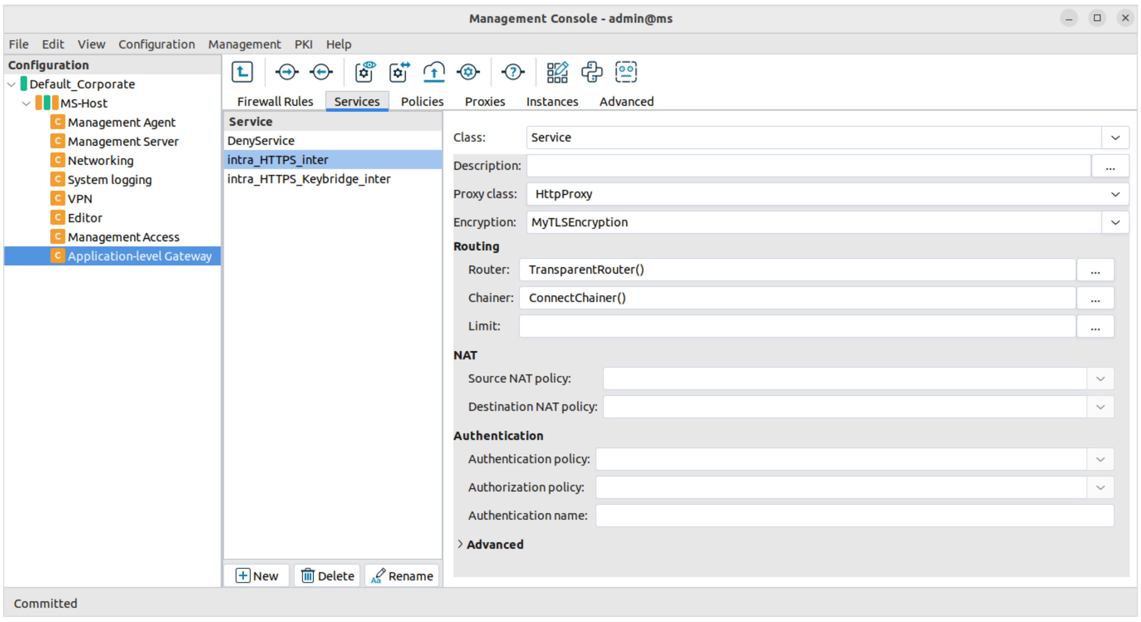
Step b. Select **Proxy class** > **Http** > **HttpProxy**.

Step c. Select **Encryption** > **MyTLSEncryption**.

Step d. Configure the other parameters of the service as neecessary for the environment, then click **OK**.

Step e. Select **Firewall Rules** > **New** > **Service**, and select the service created in the previous step. For more details on creating firewall rules, see Section 6.5, Configuring firewall rules in *Proxedo Network Security Suite 2 Administrator Guide*.

Step f. Configure the other parameters of the rule as necessary for the environment, then click **OK**.


Management Console - admin@ms

File Edit View Configuration Management PKI Help

Configuration
- Default_Corporate
  - MS-Host
    - Management Agent
    - Management Server
    - Networking
    - System logging
    - VPN
    - Editor
    - Management Access
    - Application-level Gateway

Firewall Rules | Services | Policies | Proxies | Instances | Advanced

Service
- DenyService
- intra_HTTPS_inter
- intra_HTTPS_Keybridge_inter

New | Delete | Rename

Class: Service
Description:
Proxy class: HttpProxy
Encryption: MyTLSEncryption

Routing
Router: TransparentRouter()
Chainer: ConnectChainer()
Limit:

NAT
Source NAT policy:
Destination NAT policy:

Authentication
Authentication policy:
Authorization policy:
Authentication name:

› Advanced

Committed

Figure 3.8. Creating a Service

Python:

```
def demo() :
    Service(
        name='demo/intra_HTTPS_inter',
        router=TransparentRouter(),
        chainer=ConnectChainer(),
        proxy_class=HttpProxy,
        max_instances=0,
        max_sessions=0,
        keepalive=V_KEEPALIVE_NONE,
        encryption_policy="MyTLSEncryption"
    )

    Rule(
        rule_id=300,
        src_subnet=('192.168.1.1/32', ),
        dst_zone=('internet', ),
        proto=6,
        service='demo/intra_HTTPS_inter'
    )
```

Step 5. Commit and upload the changes, then restart Application-level Gateway.
Expected result:

Every time a client connects to a server, Application-level Gateway checks the certificate of the server. If the signer CA is trusted, Application-level Gateway shows a trusted certificate to the client (browser or other application). If the certificate of the server is untrusted, Application-level Gateway shows an untrusted certificate to the client, giving a warning to the user. The user can then decide whether the certificate can be accepted or not.

3.2.5. Certificate verification options

Application-level Gateway is able to automatically verify the certificates received.

When checking revocation state for certificate chains Application-level Gateway offers two options:

- leaf verification: When enabled, leaf revocation check performs both CRL and OCSP staple checking.
- non-leaf verification: Non-leaf revocation check performs CRL checking.

To support both maximum security verification and the more common use-cases with less strict scenarios, the level of strictness for the verification can be configured separately for leaf and for non-leaf certificates. The configuration is done in encryption policies using options *`intermediate_revocation_check_type`* and *`leaf_revocation_check_type`*.

For both *`intermediate_revocation_check_type`* and *`leaf_revocation_check_type`* options 3 values are available:

- NONE: no revocation check is done
- HARD_FAIL: revocation check is performed with strict rules. No uncertainty is tolerated, the certificate must have up-to-date and authentic information about a status not being revoked. Otherwise the certificate is rejected by the policy.
- SOFT_FAIL: revocation check is performed, however, the information is not verified, and the failure to get revocation information is tolerated. If any available information states that the certificate is revoked, the certificate is rejected, otherwise the connection can be established.

`intermediate_revocation_check_type` also controls the chain verifier behavior of OCSP stapling. If no CRL list from the intermediate CA is available, the OCSP stapling signer's 'certification revoked' state cannot be determined. If *`intermediate_revocation_check_type`* has the HARD_FAIL value, the OCSP stapling message is not accepted as a valid revocation information. However, if the OCSP stapling message shows 'revoked state' the message is considered regardless of its chain verification.

The following tables describe how the values, that is NONE, HARD_FAIL or SOFT_FAIL influence the result of the revocation check type:

CRL result	**OCSP stapling result**	**SOFT_FAIL evaluates the results as:**	**HARD_FAIL evaluates the results as:**
GOOD	GOOD	ACCEPT	ACCEPT
GOOD	UNKNOWN	ACCEPT	ACCEPT
UNKNOWN	GOOD	ACCEPT	ACCEPT
UNKNOWN	UNKNOWN	ACCEPT	DENY
GOOD	REVOKED	DENY	DENY
REVOKED	GOOD	DENY	DENY
UNKNOWN	REVOKED	DENY	DENY
REVOKED	UNKNOWN	DENY	DENY

CRL result	OCSP stapling result	SOFT_FAIL evaluates the results as:	HARD_FAIL evaluates the results as:
REVOKED	REVOKED	DENY	DENY

Table 3.1. Evaluation logic for leaf revocation check type

CRL result	SOFT_FAIL evaluates the results as:	HARD_FAIL evaluates the results as:
GOOD	ACCEPT	ACCEPT
UNKNOWN	ACCEPT	DENY
REVOKED	DENY	DENY

Table 3.2. Evaluation logic for non-leaf revocation check type

The types of accepted certificates can be controlled separately on the client and the server side using the attributes of the *ClientCertificateVerifier* and *ServerCertificateVerifier* classes (or your own classes derived from these), respectively.

By default (if the *check_subject* parameter is set to *TRUE* in the verifier), Application-level Gateway compares the domain name provided in the *Subject* field of the server certificate to application-level information about the server (that is, the domain name of the URL in HTTP and HTTPS connections).

Example 3.1. Accepting invalid certificates
The following example configures a simple Encryption Policy that permits invalid certificated, and does not check the subject of the server's certificate.

```
EncryptionPolicy(
    name="MyTLSEncryption",
    encryption=TwoSidedEncryption(
        client_verify=ClientNoneVerifier(),
        server_verify=ServerCertificateVerifier(
            trust_level=TLS_TRUST_LEVEL_NONE, intermediate_revocation_check_type =
            TLS_INTERMEDIATE_REVOCATION_NONE, leaf_revocation_check_type =
            TLS_LEAF_REVOCATION_NONE,
            trusted_certs_directory="",
            verify_depth=4,
            verify_ca_directory="/etc/ca.d/certs/",
            verify_crl_directory="/etc/ca.d/crls/",
            check_subject=FALSE
            )
```

3.2.6. Protocol-level TLS settings

The following sections describe and show examples to common protocol-level TLS settings.

Cipher selection

The cipher algorithms used for key exchange and mass symmetric encryption are specified by the *cipher* attribute of the class referred in the *client_tls_options* or *server_tls_options* of the Encryption policy. These attributes contain a cipher specification as specified by the OpenSSL manuals, see the manual page `ciphers(ssl)` for further details.

The default set of ciphers can be set by using the following predefined variables.

Name	Value
TLS_CIPHERS_DEFAULT	n/a
TLS_CIPHERS_OLD	n/a
TLS_CIPHERS_CUSTOM	n/a

Table 3.3. Constants for cipher selection

Cipher specifications as defined above are sorted by key length. The cipher providing the best key length will be the most preferred.

3.2.7. Enabling STARTTLS

Application-level Gateway supports the STARTTLS method for encrypting connections. STARTTLS support can be configured separately for the client- and server side. Currently, the following proxies support STARTTLS: Ftp proxy (to start FTPS sessions), Smtp proxy.

STARTTLS is enabled by default in the following encryption scenarios:

- *ClientOnlyStartTLSEncryption*: STARTTLS is enabled on the client-side, but the server-side connection will not be encrypted.
- *FakeStartTLSEncryption*: STARTTLS is enabled on the client-side, the server-side connection is always encrypted.
- *ForwardStartTLSEncryption*: STARTTLS is enabled on the client-side, and Application-level Gateway forwards the request to the server.

Example 3.2. Configuring FTPS support
This example is a standard FtpProxy with FTPS support enabled.

```
class FtpsProxy(FtpProxy):
    def config(self):
        FtpProxy.config(self)
        self.max_password_length=64

    EncryptionPolicy(
      name="ForwardSTARTTLS",
      encryption=ForwardStartTLSEncryption(
        client_verify=ClientCertificateVerifier(),
        client_tls_options=ClientTLSOptions(),
        server_verify=ServerCertificateVerifier(),
        server_tls _options=ServerTLSOptions(),
        client_certificate_generator=DynamicCertificate(
          private_key=PrivateKey.fromFile(key_file_path="/etc/key.d/VMS_Engine/key.pem"),

trusted_ca=Certificate.fromFile(certificate_file_path="/etc/ca.d/certs/my-trusted-ca-cert.pem",
private_key=PrivateKey.fromFile("/etc/ca.d/keys/my-trusted-ca-cert.pem")),

untrusted_ca=Certificate.fromFile(certificate_file_path="/etc/ca.d/certs/my-untrusted-ca-cert.pem",
private_key=PrivateKey.fromFile("/etc/ca.d/keys/my-untrusted-ca-cert.pem")))))

    def demo() :
        Service(name='demo/MyFTPSService', router=TransparentRouter(), chainer=ConnectChainer(),
proxy_class=FtpsProxy, max_instances=0, max_sessions=0, keepalive=V_KEEPALIVE_NONE,
encryption_policy="ForwardSTARTTLS")

    Rule(rule_id=2,
    proto=6,
    service='demo/MyFTPSService'
    )
```

3.2.8. Procedure – Configuring keybridging

Purpose:

Keybridging is a method to let the client see a copy of the server's certificate (or vice versa). That way the client can inspect the certificate of the server, and decide about its trustworthiness. If the PNS firewall is proxying the TLS connection, the client cannot inspect the certificate of the server directly, but you can configure Application-level Gateway to generate a new certificate on-the-fly, using the data in the server's certificate. Application-level Gateway sends this generated certificate to the client. To configure to perform keybridging, complete the following steps.

Steps:

Step 1. Create the required keys and CA certificates.

Step a. Generate two local CA certificates. Application-level Gateway will use one of them to sign the generated certificate for servers having trusted certificates, the other one for servers with untrusted or self-signed certificates. Make this difference visible somewhere in the CA's certificates, for example, in their common name (`CA_for_Untrusted_certs`; `CA_for_Trusted_certs`). These CA certificates can be self-signed, or signed by your local root CA.
IMPORTANT: Do NOT set a password for these CAs, as Application-level Gateway must be able to access them automatically.

Step b. Import the certificate of the CA signing the trusted certificates to your clients to make the generated certificates 'trusted'.
IMPORTANT: Do NOT import the other CA certificate.

Step c. Generate a new certificate. The private key of this keypair will be used in the on-the-fly generated certificates, the public part (DN and similar information) will not be used.

Step d. In MC, set the PNS firewall host to be the owner of this certificate, then select **PKI > Distribute Certificates**.
Python:

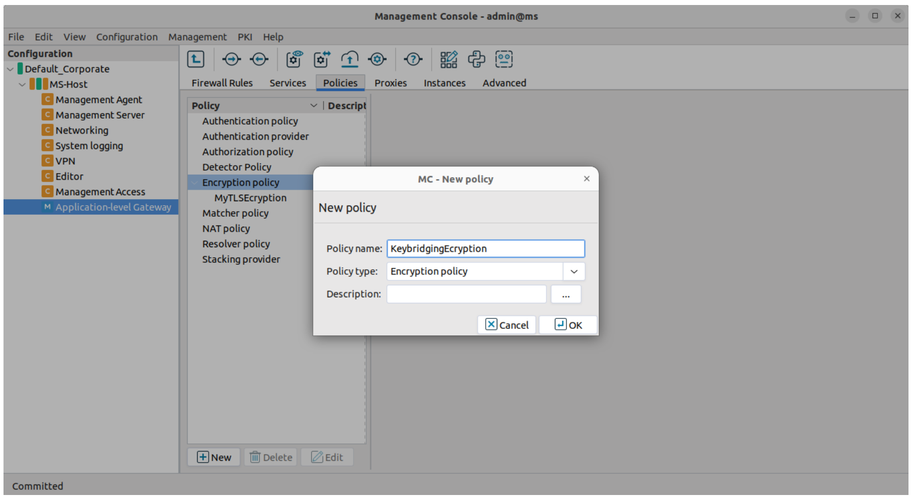
Copy the certificates and CRLs to their respective directories (for example, into `/etc/vela/tls-bridge/`). Note that they are not updated automatically as in configurations managed by MC.

Step 2. Create and configure an Encryption Policy. Complete the following steps.

Step a. Navigate to the **Application-level Gateway** MC component of the firewall host.

Step b. Select **Policies > New**.

Step c. Enter a name into the **Policy name** field, for example, `KeybridgingEncryption`.

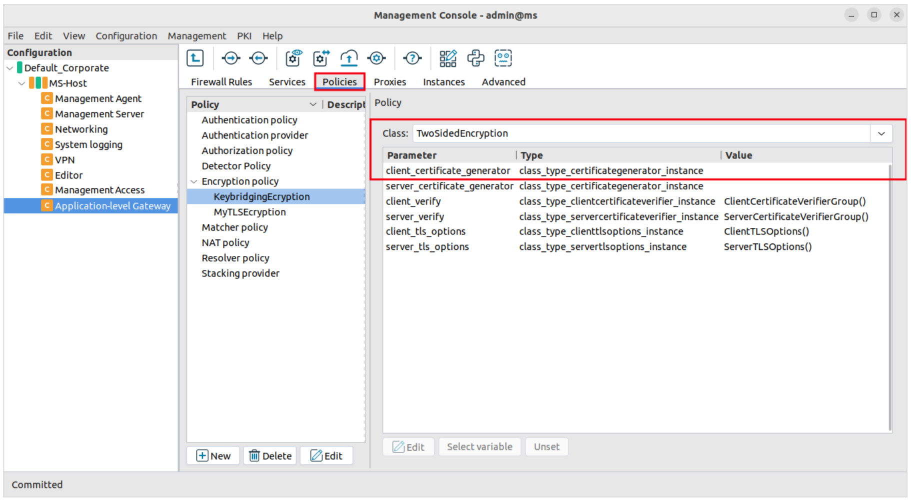


Policy name: KeybridgingEcryption
Policy type: Encryption policy
Description:

Figure 3.9. Creating an Encryption policy

Step d. Select **Policy type** > **Encryption Policy**, then click OK.

Step e. Select **Class** > **TwoSidedEncryption**.



Class: TwoSidedEncryption

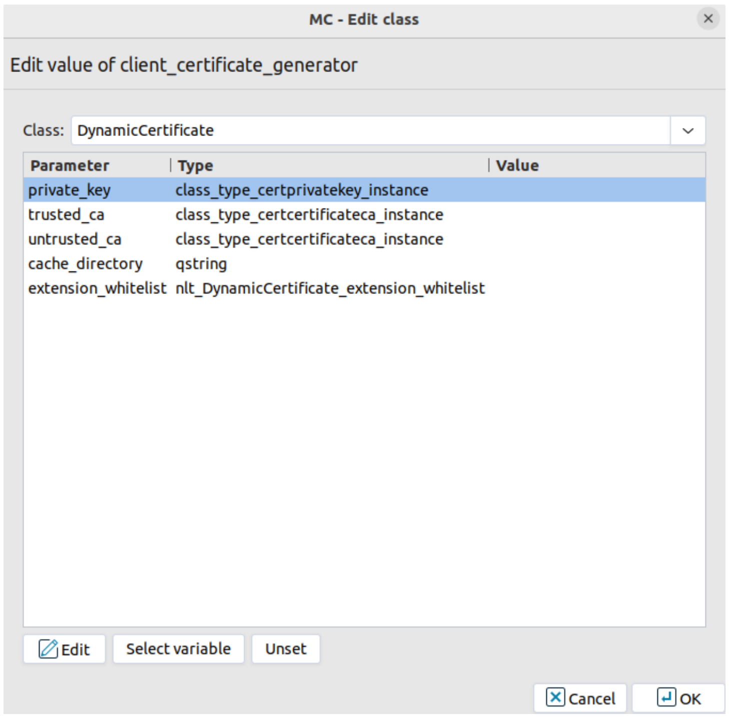
Parameter	Type	Value
client_certificate_generator	class_type_certificategenerator_instance	
server_certificate_generator	class_type_certificategenerator_instance	
client_verify	class_type_clientcertificateverifier_instance	ClientCertificateVerifierGroup()
server_verify	class_type_servercertificateverifier_instance	ServerCertificateVerifierGroup()
client_tls_options	class_type_clienttlsoptions_instance	ClientTLSOptions()
server_tls_options	class_type_servertlsoptions_instance	ServerTLSOptions()

Figure 3.10. Selecting the encryption class

Python:

```
EncryptionPolicy(
    name="KeybridgingEncryption",
    encryption=TwoSidedEncryption()
)
```

Step f. Double-click **client_certificate_generator**, then select **Class** > **DynamicCertificate**.


MC - Edit class

Edit value of client_certificate_generator

Class: DynamicCertificate

Parameter	Type	Value
private_key	class_type_certprivatekey_instance	
trusted_ca	class_type_certcertificateca_instance	
untrusted_ca	class_type_certcertificateca_instance	
cache_directory	qstring	
extension_whitelist	nlt_DynamicCertificate_extension_whitelist	

Figure 3.11. Selecting the certificate

Python:

```
     encryption=TwoSidedEncryption(
        client_certificate_generator=DynamicCertificate()
        )
```

Step g. Double-click **private_key** > **key_file_path**. The certificates owned by the host will be displayed. Select the one you created in Step 1c, then click OK. MC will automatically fill the value of the parameter to point to the location of the private key file of the certificate.
If the private key of the certificate is password-protected, double-click *passphrase*, then enter the passphrase for the private key.

Python:

```
     encryption=TwoSidedEncryption(
         client_certificate_generator=DynamicCertificate(

private_key=PrivateKey.fromFile(key_file_path="/etc/key.d/TLS-bridge/key.pem")

            )
        )
```

Step h. Double-click *trusted_ca_files* > *certificate_file_path*, select CA that will be used to sign the generated certificates for trusted peers (for example, `CA_for_Trusted_certs`), then click OK.

If the private key of the certificate is password-protected, double-click *`private_key_password`*, then enter the passphrase for the private key.

Python:

```
        client_certificate_generator=DynamicCertificate(

private_key=PrivateKey.fromFile(key_file_path="/etc/key.d/TLS-bridge/key.pem"),

            trusted_ca=Certificate.fromFile(

certificate_file_path="/etc/ca.d/certs/CA_for_Trusted_certs.pem",


private_key=PrivateKey.fromFile("/etc/ca.d/keys/CA_for_Trusted_certs.pem"))

        )
```

Step i. Double-click *`untrusted_ca_files`*, then select CA that will be used to sign the generated certificates for untrusted peers (for example, `CA_for_Untrusted_certs`). If the private key of the certificate is password-protected, double-click *`private_key_password`*, then enter the passphrase for the private key.

Python:

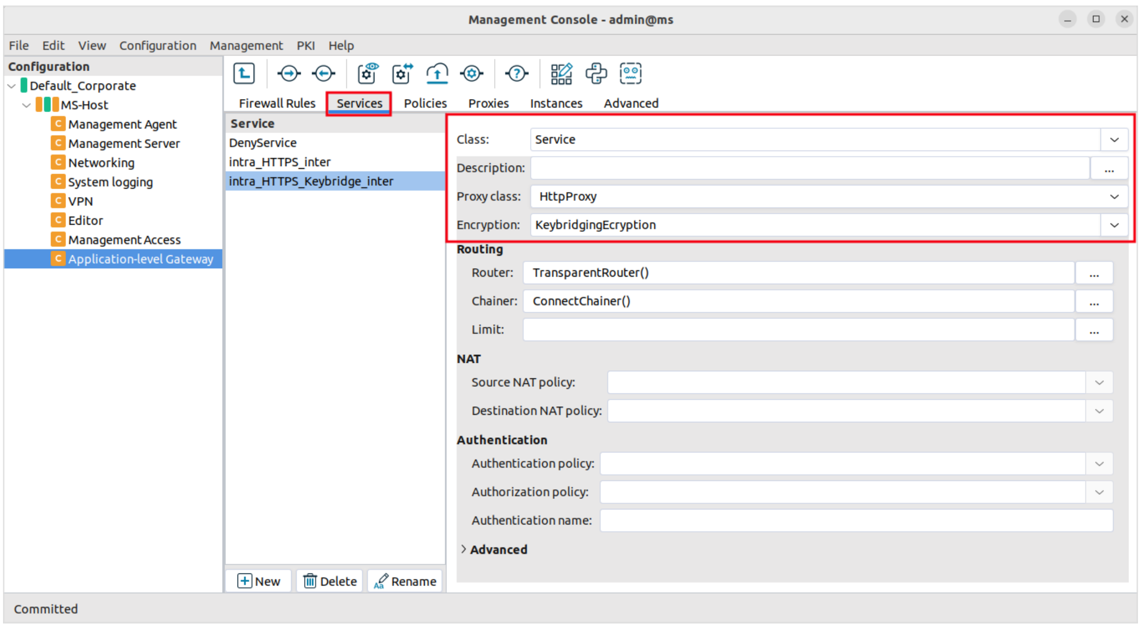
```
        client_certificate_generator=DynamicCertificate(

private_key=PrivateKey.fromFile(key_file_path="/etc/key.d/TLS-bridge/key.pem"),

            trusted_ca=Certificate.fromFile(

certificate_file_path="/etc/ca.d/certs/CA_for_Trusted_certs.pem",


private_key=PrivateKey.fromFile("/etc/ca.d/keys/CA_for_Trusted_certs.pem")),

             untrusted_ca=Certificate.fromFile(

certificate_file_path="/etc/ca.d/certs/CA_for_Untrusted_certs.pem",


private_key=PrivateKey.fromFile("/etc/ca.d/keys/CA_for_Untrusted_certs.pem"))

        )
```

Python:

The Encryption Policy configured in the previous steps is summarized in the following code snippet.

```
EncryptionPolicy(
    name="KeybridgingEncryption",
    encryption=TwoSidedEncryption(
        client_verify=ClientNoneVerifier(),
        client_tls_options=ClientTLSOptions(),
        server_verify=ServerCertificateVerifier(),
        server_tls_options=ServerTLSOptions(),
        client_certificate_generator=DynamicCertificate(

private_key=PrivateKey.fromFile(key_file_path="/etc/key.d/TLS-bridge/key.pem"),

            trusted_ca=Certificate.fromFile(

certificate_file_path="/etc/ca.d/certs/CA_for_Trusted_certs.pem",


private_key=PrivateKey.fromFile("/etc/ca.d/keys/CA_for_Trusted_certs.pem")),

            untrusted_ca=Certificate.fromFile(

certificate_file_path="/etc/ca.d/certs/CA_for_Untrusted_certs.pem",


private_key=PrivateKey.fromFile("/etc/ca.d/keys/CA_for_Untrusted_certs.pem")

            )
        )
    ))
```

Step 3. Create a service that uses the Encryption Policy created in the previous step.



Class: Service
Description:
Proxy class: HttpProxy
Encryption: KeybridgingEcryption
Routing
Router: TransparentRouter()
Chainer: ConnectChainer()
Limit:
NAT
Source NAT policy:
Destination NAT policy:
Authentication
Authentication policy:
Authorization policy:
Authentication name:
Advanced

Figure 3.12. Creating a service

Python:

```
def demo_instance() :
        Service(name='demo/intra_HTTPS_Keybridge_inter',
router=TransparentRouter(), chainer=ConnectChainer(), proxy_class=HttpProxy,
 max_instances=0, max_sessions=0, keepalive=V_KEEPALIVE_NONE,
encryption_policy="KeybridgingEncryption")

    Rule(rule_id=20,
    src_zone=('intra', ),
    dst_zone=('internet', ),
    proto=6,
    service='demo_instance/intra_HTTPS_Keybridge_inter'
    )
```

Step 4. Configure other parameters of the Encryption Policy, service, and firewall rule as needed by your environment.

Step 5. Commit and upload the changes, then restart Application-level Gateway.
Expected result:

Every time a client connects to a previously unknown server, Application-level Gateway will generate a new certificate, sign it with one of the specified CAs, and send it to the client. This new certificate will be stored under `/var/lib/vela/tls-bridge` under a filename based on the original server certificate. If the signer CA is trusted, the client (browser or other application) will accept the connection. If the certificate is signed by the CA for untrusted certificates, the application will not recognize the issuer CA (since its certificate has not been imported to the client) and give a warning to the user. The user can then decide whether the certificate can be accepted or not.

(Actually, two files are stored on the firewall for each certificate: the original certificate received from the server, and the generated certificate. When a client connects to the server, the certificate provided by the server is compared to the stored one: if they do not match, a new certificate is generated. For example, this happens when the server certificate has been expired and refreshed.)

3.3. Related standards

- The SSL protocol, the TLS protocol, as well as the session tickets and session identifiers as methods for SSL session reuse are described in RFC 8446 in details.
- The SSL protocol is described in RFC 6101 in details.

3.4. Encryption options reference

The available encryption-related classes and options are described in *Section 5.5, Module Encryption (p. 195)*.

3.5. X.509 Certificates

An X.509 certificate is a public key with a subject name specified as an X.500 DN (distinguished name) signed by a certificate issuing authority (CA). X.509 certificates are represented as Python policy objects having the following attributes:

`subject`	Subject of the certificate.

`issuer`	Issuer of the certificate (i.e. the CA that signed it).
`serial`	Serial number of the certificate.
`blob`	The certificate itself as a string in PEM format.

PNS uses X.509 certificates to provide a convenient and efficient way to manage and distribute certificates and keys used by the various components and proxies of the managed firewall hosts. It is mainly aimed at providing certificates required for the secure communication between the different parts of the firewall system, e.g. firewall hosts and MS engine (the actual communication is realized by agents).

Certificates of trusted CAs (and their accompanying CRLs) are used in Application-level Gateway to validate the certificates of servers accessed by the clients. The hashes and structures below are used by the various certificate-related attributes of the Application-level Gateway Encryption Policies, particularly the ones of *`certificate`* type.

3.5.1. X.509 Certificate Names

A certificate name behaves as a string, and contains a DN in the following format (also known as one-line format):

/RDN=value/RDN=value/.../RDN=value/

The word RDN stands for relative distinguished name. For example, the DN *`cn=Root CA, ou=CA Group, o=Foo Ltd, l=Bar, st=Foobar State, c=US`* becomes *`/C=US/ST=Foobar State/L=Bar/O=Foo Ltd/OU=CA Group/CN=Root CA/`*

3.5.2. X.509 Certificate Revocation List

A certifying authority may revoke the issued certificates. A revocation means that the serial number and the revocation date is added to the list of revoked certificates. Revocations are published on a regular basis. This list is called the Certificate Revocation List, also known as CRL. A CRL always has an issuer, a date when the list was published, and the expected date of its next update.

3.5.3. X.509 Online Certificate Status Protocol (OCSP) stapling

Online Certificate Status Protocol (OCSP) stapling is an alternative to Certificate Revocation Lists (CRL) in verifying the validity of certificates. The protocol is described in details in IETF RFC 6960. It is now also possible to define to what level of strictness the encryption policies shall check the revocation status of the certificates. OCSP stapling provides a potentially faster revocation state with less traffic.

3.5.4. X.509 Certificate hash

The proxy stores trusted CA certificates in a Certificate hash. This hash can be indexed by two different types. If an integer index is used, the slot specified by this value is looked up; if a string index is used, it is interpreted as a one-line DN value, and the appropriate certificate is looked up. Each slot in this hash contains an X.509 certificate.

3.5.5. X.509 CRL hash

Similarly to the certificate hash, a separate hash for storing Certificate Revocation Lists was defined. A CRL contains revocation lists associated to CAs.

Chapter 4. Proxies

This chapter contains reference information on all the available PNS proxies.

4.1. General information on the proxy modules

The sections discussing the available proxies are organized as follows. Overall introduction is followed by proxy class descriptions. Each module has an abstract class which is an interface between the policy and the proxy itself. Abstract classes are the point where the low-level attributes implemented by the proxy appear.

Each Python module contains an abstract proxy class (e.g., AbstractFtpProxy) and one or more preconfigured proxy classes derived from the abstract class (e.g., FtpProxy, FtpProxyRO, etc.). These abstract proxies are very low level classes which always require customization to operate at all, thus they are not directly usable. The preconfigured classes customize the base abstract proxy to perform actually useful functionality. These derived classes inherit all their attributes from the class they were derived from, but have some of their parameters set to default values. Consequently, they can be used for certain tasks without any (or only minimal) modification. Most default classes were derived directly from the abstract classes, but it is possible to derive a class from another derived class. In this case this new class inherits the attributes from its parent class and the abstract class as well. Abstract classes should not be used directly for configuring services in PNS, always derive an own class and modify its attributes to suit the requirements.

4.2. Attribute values

The description of each abstract class includes a detailed list and definition of the attributes of the proxy class. The type and default value of the attribute is also provided. Most types of the attributes (e.g., integer, string, boolean, etc.) are self-explanatory; more complicated attributes (listed as complex type) are explained in their respective description or in the general proxy behavior section of the module.

Proxy attributes can be available and modified during configuration time, run time, or both. Configuration time attributes are set and modified when the proxy is configured, that is, when the session starts. Run time attributes are available when the connection is active, for example, information about the HTTP header being processed is available only when the header is processed. Access to the attributes is indicated in the header of the description of the attribute in the following format: *`availability during configuration time : availability during run time`*. The type of availability can be read (*`r`*) access, write (*`w`*) access, both, or not available (*`n/a`*). An attribute that is available for reading and writing during both configuration and run time is indicated as *`rw:rw`*, an attribute that is available only for reading during run time is indicated as *`n/a:r`*.

Note
Unless noted otherwise, default values related to lengths (e.g., line length, etc.) are in bytes.

Timeout values are always given in milliseconds. Setting a timeout to *`-1`* disables the timeout (i.e. it becomes unlimited).

The description of every proxy class includes a list or textual description of the attributes modified relative to their parent class. The values of the other attributes are inherited from the parent class.

4.3. Examples

A number of Python code samples is provided for the proxies to illustrate both their general operation and their capabilities. Most of the proxy configurations shown in the examples can be easily reproduced using the MC graphical interface. However, some of them utilize the advanced flexibility of PNS and therefore require the use of configuration scripts written in Python. From MC these can be implemented, maintained and edited using the Class editor. (The Class editor is available under the **Proxies** tab of the **PNS** MC component. When creating a new class, click on the **Class editor** button under the list of available classes.)

4.4. Module AnyPy

This module defines an interface to the AnyPy proxy implementation. AnyPy is basically a Python proxy which means that the proxy behaviour is defined in Python by the administrator.

4.4.1. Related standards

4.4.2. Classes in the AnyPy module

Class	**Description**
AbstractAnyPyProxy	Class encapsulating an AnyPy proxy.
AnyPyProxy	Class encapsulating the default AnyPy proxy.

Table 4.1. Classes of the AnyPy module

4.4.3. Class AbstractAnyPyProxy

This class encapsulates AnyPy, a proxy module calling a Python function to do all of its work. It can be used for defining proxies for protocols not directly supported.

Warning
This proxy class is a basis for creating a custom proxy, and cannot be used on its own. Create a new proxy class using the AnyPyProxy as its parent, and implement the proxyThread method to handle the traffic.

Your code will be running as the proxy to transmit protocol elements. When writing your code, take care and be security conscious: do not make security vulnerabilities.

4.4.3.1. Attributes of AbstractAnyPyProxy

client_max_line_length (integer)
Default: 4096
Size of the line buffer in the client stream in bytes. Default value: 4096

server_max_line_length (integer)
Default: 4096

server_max_line_length (integer)
Size of the line buffer in the server stream in bytes. Default value: 4096

4.4.3.2. AbstractAnyPyProxy methods

Method	Description
__init__(self, session)	Constructor to initialize an AnyPy instance.
proxyThread(self)	Function called by the low-level proxy core to transfer requests.

Table 4.2. Method summary

Method __init__(self, session)

This constructor initializes a new AnyPy instance based on its arguments, and calls the inherited constructor.

Arguments of __init__

session (unknown)
Default: n/a
The session to be inspected with the proxy instance.

Method proxyThread(self)

This function is called by the proxy module to transfer requests. It can use the 'self.session.client_stream' and 'self.session.server_stream' streams to read data from and write data to.

4.4.4. Class AnyPyProxy

This class encapsulates AnyPy, a proxy module calling a Python function to do all of its work. It can be used for defining proxies for protocols not directly supported.

4.4.4.1. Note

This proxy class can only be used as a basis for creating a custom proxy and cannot be used on its own. Please create a new proxy class with the AnyPyProxy as its parent and implement the proxyThread method for handling traffic.

Your code will be running as the proxy to transmit protocol elements, you'll have to take care and be security conscious not to make security vulnerabilities.

4.5. Module Ftp

The Ftp module defines the classes constituting the proxy for the File Transfer Protocol (FTP).

4.5.1. The FTP protocol

File Transfer Protocol (FTP) is a protocol to transport files via a reliable TCP connection between a client and a server. FTP uses two reliable TCP connections to transfer files: a simple TCP connection (usually referred to as the Control Channel) to transfer control information and a secondary TCP connection (usually referred to as the Data Channel) to perform the data transfer. It uses a command/response based approach, i.e. the client issues a command and the server responds with a 3-digit status code and associated status information in text format. The Data Channel can be initiated either from the client or the server; the Control Channel is always started from the client.

The client is required to authenticate itself before other commands can be issued. This is performed using the USER and PASS commands specifying username and password, respectively.

4.5.1.1. Protocol elements

The basic protocol is as follows: the client issues a request (also called command in FTP terminology) and the server responds with the result. Both commands and responses are line based: commands are sent as complete lines starting with a keyword identifying the operation to be performed. A response spans one or more lines, each specifying the same 3-digit status code and possible explanation.

4.5.1.2. Data transfer

Certain commands (for example RETR, STOR or LIST) also have a data attachment which is transferred to the peer. Data attachments are transferred in a separate TCP connection. This connection is established on-demand on a random, unprivileged port when a data transfer command is issued.

Endpoint information of this data channel is exchanged via the PASV and PORT commands, or their newer equivalents (EPSV and EPRT).

The data connection can either be initiated by the client (passive mode) or the server (active mode). In passive mode (PASV or EPSV command) the server opens a listening socket and sends back the endpoint information in the PASV response. In active mode (PORT or EPRT command) the client opens a listening socket and sends its endpoint information as the argument of the PORT command. The source port of the server is usually either 20, or the port number of the Command Channel minus one.

Example 4.1. FTP protocol sample

```
220 FTP server ready
USER account
331 Password required.
PASS password
230 User logged in.
SYST
215 UNIX Type: L8
PASV
227 Entering passive mode (192,168,1,1,4,0)
LIST
150 Opening ASCII mode data connection for file list
226-Transferring data in separate connection complete.
226 Quotas off
QUIT
221 Goodbye
```

4.5.2. Proxy behavior

FtpProxy is a module built for parsing commands of the Control Channel in the FTP protocol. It reads the REQUEST at the client side, parses it and - if the local security policy permits - sends it to the server. The proxy parses the arriving RESPONSES and sends them to the client if the policy permits that. FtpProxy uses a PlugProxy to transfer the data arriving in the Data Channel. The proxy is capable of manipulating commands and stacking further proxies (for example, *MimeProxy*) into the Data Channel. Both transparent and non-transparent modes are supported.

The default low-level proxy implementation (AbstractFtpProxy) denies all requests by default. Different commands and/or responses can be enabled by using one of the several predefined proxy classes which are suitable for most tasks. Alternatively, use of the commands can be permitted individually using different attributes. This is detailed in the following two sections.

4.5.2.1. Configuring policies for FTP commands and responses

Changing the default behavior of commands can be done by using the hash attribute *`request`*, indexed by the command name (e.g.: USER or PWD). There is a similar attribute for responses called *`response`*, indexed by the command name and the response code. The possible values of these hashes are shown in the tables below. See *Section 2.1, Policies for requests and responses (p. 4)* for details. When looking up entries of the *`response`* attribute hash, the lookup precedence described in *Section 2.1.2, Response codes (p. 6)* is used.

Action	Description
FTP_REQ_ACCEPT	Allow the request to pass.
FTP_REQ_REJECT	Reject the request with the error message specified in the second optional parameter.
FTP_REQ_ABORT	Terminate the connection.

Table 4.3. Action codes for commands in FTP

Action	Description
FTP_RSP_ACCEPT	Allow the response to pass.
FTP_RSP_REJECT	Modify the response to a general failure with error message specified in the optional second parameter.
FTP_RSP_ABORT	Terminate the connection.

Table 4.4. Action codes for responses in FTP

Example 4.2. Customizing FTP to allow only anonymous sessions
This example calls a function called pUser (defined in the example) whenever a USER command is received. All other commands are accepted. The parameter of the USER command (i.e. the username) is examined: if it is 'anonymous' or 'Anonymous', the connection is accepted, otherwise it is rejected.

```
class AnonFtp(FtpProxy):
        def config(self):
                self.request["USER"] = (FTP_REQ_POLICY, self.pUser)
                self.request["*"] = (FTP_REQ_ACCEPT)

        def pUser(self,command):
                if self.request_parameter == "anonymous" or self.request_parameter == "Anonymous":
                        return FTP_REQ_ACCEPT
                return FTP_REQ_REJECT
```

4.5.2.2. Configuring policies for FTP features and FTPS support

FTP servers send the list of supported features to the clients. For example, ProFTPD supports the following features: *LANG en, MDTM, UTF8, AUTH TLS, PBSZ, PROT, REST STREAM, SIZE*. The default behavior of FTP features can be changed using the hash attribute *features*, indexed by the name of the feature (e.g.: UTF8 or AUTH TLS). The possible actions are shown in the table below. See *Section 2.1, Policies for requests and responses (p. 4)* for details.

The built-in FTP proxies permit the use of every feature by default.

Action	Description
FTP_FEATURE_ACCEPT	Forward the availability of the feature from the server to the client.
FTP_FEATURE_DROP	Remove the feature from the feature list sent by the server.
FTP_FEATURE_INSERT	Add the feature into the list of available features.

Table 4.5. Policy about enabling FTP features.

Enabling FTPS connections

For FTPS connections to operate correctly, the FTP server and client applications must comply to the *FTP Security Extensions (RFC 2228)* and *Securing FTP with TLS (RFC 4217)* RFCs.

For FTPS connections, the *AUTH TLS, PBSZ, PROT* features must be accepted. Also, STARTTLS support must be properly configured. See *Section 3.2, Handling TLS and SSL connections in Application-level Gateway (p. 10)* for details.

If the proxy is configured to disable encryption between PNS and the client, the proxy automatically removes the *AUTH TLS, PBSZ, PROT* features from the list sent by the server.

If STARTTLS connections are accepted on the client side (*self.tls.client_security=TLS_ACCEPT_STARTTLS*), but TLS-forwarding is disabled on the server side, the proxy automatically inserts the *AUTH TLS, PBSZ, PROT* features into the list sent by the server. These features are inserted even if encryption is explicitly disabled on the server side or the server does not support the *FEAT* command, making one-sided STARTTLS support feasible.

Warning
When using inband routing with the FTPS protocol, the server's certificate is compared to its hostname. The subject_alt_name parameter (or the Common Name parameter if the subject_alt_name parameter is empty) of the server's certificate must contain the hostname or the IP address (as resolved from the PNS host) of the server (e.g., *ftp.example.com*).

Alternatively, the Common Name or the *subject_alt_name* parameter can contain a generic hostname, e.g., **.example.com*.

Note that if the Common Name of the certificate contains a generic hostname, do not specify a specific hostname or an IP address in the *subject_alt_name parameter*.

Note

- The FTP proxy does not support the following FTPS-related commands: *`REIN, CCC, CDC`*.
- STARTTLS is supported in nontransparent scenarios as well.

Example 4.3. Configuring FTPS support
This example is a standard FtpProxy with FTPS support enabled.

```
class FtpsProxy(FtpProxy):
    def config(self):
        FtpProxy.config(self)
        self.max_password_length=64

    EncryptionPolicy(
      name="ForwardSTARTTLS",
      encryption=ForwardStartTLSEncryption(
        client_verify=ClientCertificateVerifier(),
        client_tls_options=ClientTLSOptions(),
        server_verify=ServerCertificateVerifier(),
        server_tls _options=ServerTLSOptions(),
        client_certificate_generator=DynamicCertificate(
          private_key=PrivateKey.fromFile(key_file_path="/etc/key.d/VMS_Engine/key.pem"),

trusted_ca=Certificate.fromFile(certificate_file_path="/etc/ca.d/certs/my-trusted-ca-cert.pem",
private_key=PrivateKey.fromFile("/etc/ca.d/keys/my-trusted-ca-cert.pem")),

untrusted_ca=Certificate.fromFile(certificate_file_path="/etc/ca.d/certs/my-untrusted-ca-cert.pem",
private_key=PrivateKey.fromFile("/etc/ca.d/keys/my-untrusted-ca-cert.pem")))))

    def demo() :
        Service(name='demo/MyFTPSService', router=TransparentRouter(), chainer=ConnectChainer(),
proxy_class=FtpsProxy, max_instances=0, max_sessions=0, keepalive=V_KEEPALIVE_NONE,
encryption_policy="ForwardSTARTTLS")

    Rule(rule_id=2,
    proto=6,
    service='demo/MyFTPSService'
    )
```

4.5.2.3. Stacking

The available stacking modes for this proxy module are listed in the following table. For additional information on stacking, see *Section 2.3.1, Proxy stacking (p. 7)*.

Action	Description
FTP_STK_DATA	Pass the data to the stacked proxy or program.
FTP_STK_NONE	No proxy stacked.

Table 4.6. Stacking policy.

4.5.2.4. Configuring inband authentication

The Ftp proxy supports *inband authentication* as well to use the built-in authentication method of the FTP and FTPS protocols to authenticate the client. The authentication itself is performed by the ASbackend configured for the service.

If the client uses different usernames on AS and the remote server (e.g., he uses his own username to authenticate to AS, but anonymous on the target FTP server), the client must specify the usernames and passwords in the following format:

Username:

```
<ftp user>@<proxy user>@<remote site>[:<port>]
```

Password:

```
<ftp password>@<proxy password>
```

Alternatively, all the above information can be specified as the username:

```
<ftp user>@<proxy user>@<remote site>[:<port>]:<ftp password>@<proxy password>
```

Warning
When using *inband routing* with the FTPS protocol, the server's certificate is compared to its hostname. The subject_alt_name parameter (or the Common Name parameter if the subject_alt_name parameter is empty) of the server's certificate must contain the hostname or the IP address (as resolved from the PNS host) of the server (e.g., `ftp.example.com`).

Alternatively, the Common Name or the `subject_alt_name` parameter can contain a generic hostname, e.g., `*.example.com`.

Note that if the Common Name of the certificate contains a generic hostname, do not specify a specific hostname or an IP address in the `subject_alt_name parameter`.

4.5.3. Related standards

- The File Transfer Protocol is described in RFC 959.
- FTP Security Extensions including the FTPS protocol and securing FTP with TLS are described in RFC 2228 and RFC 4217.

4.5.4. Classes in the Ftp module

Class	Description
AbstractFtpProxy	Class encapsulating the abstract FTP proxy.
FtpProxy	Default Ftp proxy based on AbstractFtpProxy.
FtpProxyAnonRO	FTP proxy based on AbstractFtpProxy, only allowing read-only access to anonymous users.
FtpProxyAnonRW	FTP proxy based on AbstractFtpProxy, allowing full read-write access, but only to anonymous users.
FtpProxyRO	FTP proxy based on AbstractFtpProxy, allowing read-only access to any user.

Class	Description
FtpProxyRW	FTP proxy based on AbstractFtpProxy, allowing full read-write access to any user.

Table 4.7. Classes of the Ftp module

4.5.5. Class AbstractFtpProxy

This proxy implements the FTP protocol as specified in RFC 959. All traffic and commands are denied by default. Consequently, either customized Ftp proxy classes derived from the abstract class should be used, or one of the predefined classes (e.g.: *FtpProxy*, *FtpProxyRO*, etc.).

4.5.5.1. Attributes of AbstractFtpProxy

active_connection_mode (enum, rw:r)
Default: FTP_ACTIVE_MINUSONE
In active mode the server connects the client. By default this must be from Command Channel port minus one (FTP_ACTIVE_MINUSONE). Alternatively, connection can also be performed either from port number 20 (FTP_ACTIVE_TWENTY) or from a random port (FTP_ACTIVE_RANDOM).

auth_tls_ok_client (boolean, n/a:r)
Default: ""
Shows whether the client-side authentication was performed over a secure channel.

auth_tls_ok_server (boolean, n/a:r)
Default: ""
Shows whether the server-side authentication was performed over a secure channel.

buffer_size (integer, rw:r)
Default: 4096
Buffer size for data transfer in bytes.

data_mode (enum, rw:r)
Default: FTP_DATA_KEEP
The type of the FTP connection on the server side can be manipulated: leave it as the client requested (FTP_DATA_KEEP), or force passive (FTP_DATA_PASSIVE) or active (FTP_DATA_ACTIVE) connection.

data_port_max (integer, rw:r)
Default: 41000

data_port_max (integer, rw:r)
On the proxy side, ports equal to or below the value of `data_port_max` can be allocated as the data channel.

data_port_min (integer, rw:r)
Default: 40000
On the proxy side, ports equal to or above the value of `data_port_min` can be allocated as the data channel.

data_protection_enabled_client (boolean, n/a:r)
Default: ""
Shows whether the data channel is encrypted or not on the client-side.

data_protection_enabled_server (boolean, n/a:r)
Default: ""
Shows whether the data channel is encrypted or not on the server-side.

features (complex, rw:rw)
Default:
Hash containing the filtering policy for FTP features.

hostname (string, n/a:rw)
Default:
The hostname of the FTP server to connect to, when inband routing is used.

hostport (integer, n/a:rw)
Default:
The port of the FTP server to connect to, when inband routing is used.

masq_address_client (string, rw:r)
Default: ""
IP address of the firewall appearing on the client side. If its value is set, this IP is sent regardless of its true IP (where it is binded). This attribute may be used when network address translation is performed before Vela.

masq_address_server (string, rw:r)
Default: ""

masq_address_server (string, rw:r)
IP address of the firewall appearing on the server side. If its value is set, this IP is sent regardless of its true IP (where it is binded). This attribute may be used when network address translation is performed before Vela.

max_continuous_line (integer, rw:r)
Default: 100
Maximum number of answer lines for a command.

max_hostname_length (integer, rw:r)
Default: 128
Maximum length of hostname. Used only in non-transparent mode.

max_line_length (integer, rw:r)
Default: 255
Maximum length of a line that the proxy is allowed to transfer. Requests/responses exceeding this limit are dropped.

max_password_length (integer, rw:r)
Default: 64
Maximum length of the password.

max_username_length (integer, rw:r)
Default: 32
Maximum length of the username.

password (string, n/a:rw)
Default:
The password to be sent to the server.

permit_client_bounce_attack (boolean, rw:rw)
Default: FALSE
If enabled the IP addresses of data channels will not need to match with the IP address of the control channel, permitting the use of FXP while increasing the security risks.

permit_empty_command (boolean, rw:r)
Default: TRUE
Enable transmission of lines without commands.

permit_server_bounce_attack (boolean, rw:rw)
Default: FALSE
If enabled the IP addresses of data channels will not need to match with the IP address of the control channel, permitting the use of FXP while increasing the security risks.

permit_unknown_command (boolean, rw:r)
Default: FALSE
Enable the transmission of unknown commands.

proxy_password (string, n/a:rw)
Default:
The password to be used for proxy authentication given by the user, when inband authentication is used.

proxy_username (string, n/a:rw)
Default:
The username to be used for proxy authentication given by the user, when inband authentication is used.

request (complex, rw:rw)
Default:
Normative policy hash for FTP requests indexed by command name (e.g.: "USER", "PWD" etc.). See also *Section 2.1, Policies for requests and responses (p. 4)*.

request_command (string, n/a:rw)
Default: n/a
When a request is evaluated on the policy level, this variable contains the requested command.

request_parameter (string, n/a:rw)
Default: n/a
When a request is evaluated on the policy level, this variable contains the parameters of the requested command.

request_stack (complex, rw:rw)
Default:

request_stack (complex, rw:rw)
Hash containing the stacking policy for the FTP commands. The hash is indexed by the FTP command (e.g. RETR, STOR). See also *Section 2.3.1, Proxy stacking (p. 7)*.

response (complex, rw:rw)
Default:
Normative policy hash for FTP responses indexed by command name and answer code (e.g.: "USER","331"; "PWD","200" etc.). See also *Section 2.1, Policies for requests and responses (p. 4)*.

response_parameter (string, n/a:rw)
Default:
When a response is evaluated on the policy level, this variable contains answer parameters.

response_status (string, n/a:rw)
Default:
When a response is evaluated on the policy level, this variable contains the answer code.

response_strip_msg (boolean, rw:r)
Default: FALSE
Strip the response message and only send the response code.

strict_port_checking (boolean, rw:rw)
Default: TRUE
If enabled the foreign port is strictly checked: in active mode the server must be connected on port 20, while in any other situation the foreign port must be above 1023.

target_port_range (string, rw:r)
Default: "21"
The port where the client can connect through a non-transparent FtpProxy.

timeout (integer, rw:r)
Default: 300000
General I/O timeout in milliseconds. When there is no specific timeout for a given operation, this value is used.

transparent_mode (boolean, rw:r)
Default: TRUE
Specifies if the proxy works in transparent (TRUE) or non-transparent (FALSE) mode.

username (string, n/a:rw)
Default:
The username authenticated to the server.

valid_chars_username (string, rw:r)
Default: "a-zA-Z0-9._@"
List of the characters accepted in usernames.

4.5.6. Class FtpProxy

A permitting Ftp proxy based on the AbstractFtpProxy, allowing all commands, responses, and features, including unknown ones. The connection is terminated if a response with the answer code `421` is received.

4.5.7. Class FtpProxyAnonRO

FTP proxy based on AbstractFtpProxy, enabling read-only access (i.e. only downloading) to anonymous users (uploads and usernames other than 'anonymous' or 'ftp' are disabled). Commands and return codes are strictly checked, unknown commands and responses are rejected. Every feature is accepted.

The ABOR; ACCT; AUTH; CDUP; CWD; EPRT; EPSV; FEAT; LIST; MODE; MDTM; NLST; NOOP; OPTS; PASV; PASS; PORT; PWD; QUIT; REST; RETR; SIZE; STAT; STRU; SYST; TYPE; and USER commands are permitted, the CLNT; XPWD; MACB commands are rejected.

4.5.8. Class FtpProxyAnonRW

FTP proxy based on AbstractFtpProxy, enabling full read-write access to anonymous users (the 'anonymous' and 'ftp' usernames are permitted). Commands and return codes are strictly checked, unknown commands and responses are rejected. Every feature is accepted.

The ABOR; ACCT; APPE; CDUP; CWD; DELE; EPRT; EPSV; LIST; MKD; MODE; MDTM; NLST; NOOP; OPTS; PASV; PASS; PORT; PWD; QUIT; RMD; RNFR; RNTO; REST; RETR; SIZE; STAT; STOR; STOU; STRU; SYST; TYPE; USER and FEAT commands are permitted, the AUTH; CLNT; XPWD; MACB commands are rejected.

4.5.9. Class FtpProxyRO

FTP proxy based on AbstractFtpProxy, enabling read-only access to any user. Commands and return codes are strictly checked, unknown commands and responses are rejected. Every feature is accepted.

The ABOR; ACCT; AUTH; CDUP; CWD; EPRT; EPSV; FEAT; LIST; MODE; MDTM; NLST; NOOP; OPTS; PASV; PASS; PORT; PWD; QUIT; REST; RETR; SIZE; STAT; STRU; SYST; TYPE; and USER commands are permitted, the CLNT; XPWD; MACB commands are rejected.

4.5.10. Class FtpProxyRW

FTP proxy based on AbstractFtpProxy, enabling full read-write access to any user. Commands and return codes are strictly checked, unknown commands and responses are rejected. Every feature is accepted.

The ABOR; ACCT; AUTH; CDUP; CWD; EPRT; EPSV; FEAT; LIST; MODE; MDTM; NLST; NOOP; OPTS; PASV; PASS; PORT; PWD; QUIT; REST; RETR; SIZE; STAT; STRU; SYST; TYPE; and USER commands are permitted, the CLNT; XPWD; MACB commands are rejected.

4.6. Module Http

The Http module defines the classes constituting the proxy for the HyperText Transfer Protocol (HTTP). HTTP is the protocol the Web is based on, therefore it is the most frequently used protocol on the Internet. It is used to access different kinds of content from the Web. The type of content retrieved via HTTP is not restricted, it can range from simple text files to hypertext files and multimedia formats like pictures, videos or audio files.

4.6.1. The HTTP protocol

HTTP is an open application layer protocol for hypermedia information systems. It basically allows an open-ended set of methods to be applied to resources identified by Uniform Resource Identifiers (URIs).

4.6.1.1. Protocol elements

HTTP is a text based protocol where a client sends a request comprising of a METHOD, an URI and associated meta information represented as MIME-like headers, and possibly a data attachment. The server responds with a status code, a set of headers, and possibly a data attachment. Earlier protocol versions perform a single transaction in a single TCP connection, HTTP/1.1 introduces persistency where a single TCP connection can be reused to perform multiple transactions.

An HTTP method is a single word - usually spelled in capitals - instructing the server to apply a function to the resource specified by the URI. Commonly used HTTP methods are "GET", "POST" and "HEAD". HTTP method names are not restricted in any way, other HTTP based protocols (such as WebDAV) add new methods to the protocol while keeping the general syntax intact.

Headers are part of both the requests and the responses. Each header consists of a name followed by a colon (':') and a field value. These headers are used to specify content-specific and protocol control information.

The response to an HTTP request starts with an HTTP status line informing the client about the result of the operation and an associated message. The result is represented by three decimal digits, the possible values are defined in the HTTP RFCs.

4.6.1.2. Protocol versions

The protocol has three variants, differentiated by their version number. Version 0.9 is a very simple protocol which allows a simple octet-stream to be transferred without any meta information (e.g.: no headers are associated with requests or responses).

Version 1.0 introduces MIME-like headers in both requests and responses; headers are used to control both the protocol (e.g.: the "Connection" header) and to give information about the content being transferred (e.g.: the "Content-Type" header). This version has also introduced the concept of name-based virtual hosts.

Building on the success of HTTP/1.0, version 1.1 of the protocol adds persistent connections (also referred to as "connection keep-alive") and improved proxy control.

4.6.1.3. Bulk transfer

Both requests and responses might have an associated data blob, also called an entity in HTTP terminology. The size of the entity is determined using one of three different methods:

1. The complete size of the entity is sent as a header (the Content-Length header).
2. The transport layer connection is terminated when transfer of the blob is completed (used by HTTP/0.9 and might be used in HTTP/1.1 in non-persistent mode).
3. Instead of specifying the complete length, smaller chunks of the complete blob are transferred, and each chunk is prefixed with the size of that specific chunk. The end of the stream is denoted by a zero-length chunk. This mode is also called chunked encoding and is specified by the Transfer-Encoding header.

Example 4.4. Example HTTP transaction

```
GET /index.html HTTP/1.1
Host: www.example.com
Connection: keep-alive
User-Agent: My-Browser-Type 6.0

HTTP/1.1 200 OK
Connection: close
Content-Length: 14

<html>
</html>
```

4.6.2. Proxy behavior

The default low-level proxy implementation (*AbstractHttpProxy*) denies all requests by default. Different requests and/or responses can be enabled by using one of the several predefined proxy classes which are suitable for most tasks. Alternatively, a custom proxy class can be derived from AbstractHttpProxy and the requests and responses enabled individually using different attributes.

Several examples and considerations on how to enable virus filtering in the HTTP traffic are discussed in the Technical White Paper and Tutorial *Virus filtering in HTTP*, available at the BalaSys *Documentation Page*.

4.6.2.1. Transparent and non-transparent modes

HttpProxy is able to operate both in transparent and non-transparent mode. In transparent mode, the client does not notice (or even know) that it is communicating through a proxy. The client communicates using normal server-style requests.

In non-transparent mode, the address and the port of the proxy server must be set on the client. In this case the client sends proxy-style requests to the proxy.

Example 4.5. Proxy style HTTP query

```
GET http://www.example.com/index.html HTTP/1.1
Host: www.example.com
Connection: keep-alive
User-Agent: My-Browser-Type 6.0

HTTP/1.1 200 OK
Connection: close
Content-Length: 14

<html>
</html>
```

In non-transparent mode it is possible to request the use of the SSL protocol through the proxy, which means the client communicates with the proxy using the HTTP protocol, but the proxy uses HTTPS to communicate with the server. This technique is called data tunneling.

Example 4.6. Data tunneling with connect method

```
CONNECT www.example.com:443 HTTP/1.1
Host: www.example.com
User-agent: My-Browser-Type 6.0

HTTP/1.0 200 Connection established
Proxy-agent: My-Proxy/1.1
```

4.6.2.2. Configuring policies for HTTP requests and responses

Changing the default behavior of requests is possible using the *request* attribute. This hash is indexed by the HTTP method names (e.g.: GET or POST). The *response* attribute (indexed by the request method and the response code) enables the control of HTTP responses. The possible actions are described in the following tables. See also *Section 2.1, Policies for requests and responses (p. 4)*. When looking up entries of the *response* attribute hash, the lookup precedence described in *Section 2.1.2, Response codes (p. 6)* is used.

Action	Description
HTTP_REQ_ACCEPT	Allow the request to pass.
HTTP_REQ_REJECT	Reject the request. The reason for the rejection can be specified in the optional second argument.
HTTP_REQ_ABORT	Terminate the connection.

Action	Description
HTTP_REQ_POLICY	Call the function specified to make a decision about the event. The function receives four arguments: self, method, url, version. See *Section 2.1, Policies for requests and responses (p. 4)* for details.

Table 4.8. Action codes for HTTP requests

Action	Description
HTTP_RSP_ACCEPT	Allow the response to pass.
HTTP_RSP_DENY	Reject the response and return a policy violation page to the client.
HTTP_RSP_REJECT	Reject the response and return a policy violation page to the client, with error information optionally specified as the second argument.
HTTP_RSP_POLICY	Call the function specified to make a decision about the event. The function receives five parameters: self, method, url, version, response. See *Section 2.1, Policies for requests and responses (p. 4)* for details.

Table 4.9. Action codes for HTTP responses

Example 4.7. Implementing URL filtering in the HTTP proxy
This example calls the filterURL function (defined in the example) whenever a HTTP GET request is received. If the requested URL is 'http://www.disallowedsite.com', the request is rejected and an error message is sent to the client.

```
class DmzHTTP(HttpProxy):
        def config(self):
                HttpProxy.config(self)
                self.request["GET"] = (HTTP_REQ_POLICY, self.filterURL)

        def filterURL(self, method, url, version):
                if (url == "http://www.disallowedsite.com"):
                        self.error_info = 'Access of this content is denied by the local policy.'
                        return HTTP_REQ_REJECT
                return HTTP_REQ_ACCECT
```

Example 4.8. 404 response filtering in HTTP
In this example the 404 response code to GET requests is rejected, and a custom error message is returned to the clients instead.

```
class DmzHTTP(HttpProxy):
        def config(self):
                HttpProxy.config(self)
                self.response["GET", "404"] = (HTTP_RSP_POLICY, self.filter404)

        def filter404(self, method, url, version, response):
                self.error_status = 404
                self.error_info = "Requested page was not accessible."
                return HTTP_RSP_REJECT
```

4.6.2.3. Configuring policies for HTTP headers

Both request and response headers can be modified by the proxy during the transfer. New header lines can be inserted, entries can be modified or deleted. To change headers in the requests and responses use the *request_header* hash or the *response_header* hash, respectively.

Similarly to the request hash, these hashes are indexed by the header name (like "User-Agent") and contain an actiontuple describing the action to take.

By default, the proxy modifies only the "Host", "Connection", "Proxy-Connection" and "Transfer-Encoding" headers. "Host" headers need to be changed when the proxy modifies the URL; "(Proxy-)Connection" is changed when the proxy turns connection keep-alive on/off; "Transfer-Enconding" is changed to enable chunked encoding.

Action	Description
HTTP_HDR_ABORT	Terminate the connection.
HTTP_HDR_ACCEPT	Accept the header.
HTTP_HDR_DROP	Remove the header.
HTTP_HDR_POLICY	Call the function specified to make a decision about the event. The function receives three parameters: self, hdr_name, and hdr_value.
HTTP_HDR_CHANGE_NAME	Rename the header to the name specified in the second argument.
HTTP_HDR_CHANGE_VALUE	Change the value of the header to the value specified in the second argument.
HTTP_HDR_CHANGE_BOTH	Change both the name and value of the header to the values specified in the second and third arguments, respectively.
HTTP_HDR_INSERT	Insert a new header defined in the second argument.
HTTP_HDR_REPLACE	Remove all existing occurrences of a header and replace them with the one specified in the second argument.

Table 4.10. Action codes for HTTP headers

Example 4.9. Header filtering in HTTP
The following example hides the browser used by the client by replacing the value of the User-Agent header to Lynx in all requests. The use of cookies is disabled as well.

```
class MyHttp(HttpProxy):
        def config(self):
                HttpProxy.config(self)
                self.request_header["User-Agent"] = (HTTP_HDR_CHANGE_VALUE, "Lynx 2.4.1")
                self.request_header["Cookie"] = (HTTP_HDR_POLICY, self.processCookies)
                self.response_header["Set-Cookie"] = (HTTP_HDR_DROP,)

        def processCookies(self, name, value):
                # You could change the current header in self.current_header_name
                # or self.current_header_value, the current request url is
                # in self.request_url
                return HTTP_HDR_DROP
```

4.6.2.4. Redirecting URLs

URLs or sets of URLs can be easily rejected or redirected to a local mirror by modifying some attributes during request processing.

When an HTTP request is received, normative policy chains are processed (*self.request*, *self.request_header*). Policy callbacks for certain events can be configured with the HTTP_REQ_POLICY or HTTP_HDR_POLICY directives. Any of these callbacks may change the *request_url* attribute, instructing the proxy to fetch a page different from the one specified by the browser. Please note that this is transparent to the user and does not change the URL in the browser.

Example 4.10. URL redirection in HTTP proxy
This example redirects all HTTP GET requests to the 'http://www.example.com/' URL by modifying the value of the requested URL.

```
class MyHttp(HttpProxy):
        def config(self):
                HttpProxy.config(self)
                self.request["GET"] = (HTTP_REQ_POLICY, self.filterURL)

        def filterURL(self, method, url, version):
                self.request_url = "http://www.example.com/"
                return HTTP_REQ_ACCEPT
```

Example 4.11. Redirecting HTTP to HTTPS
This example redirects all incoming HTTP connections to an HTTPS URL.

```
class HttpProxyHttpsredirect(HttpProxy):
        def config(self):
                HttpProxy.config(self)
                self.error_silent = TRUE
                self.request["GET"] = (HTTP_REQ_POLICY, self.reqRedirect)

        def reqRedirect(self, method, url, version):
                self.error_status = 301
                #self.error_info = 'HTTP/1.0 301 Moved Permanently'
                self.error_headers="Location: https://%s/" % self.request_url_host
                return HTTP_REQ_REJECT
```

4.6.2.5. Request types

PNS differentiates between two request types: server requests and proxy request.

- Server requests are sent by browsers directly communicating with HTTP servers. These requests include an URL relative to the server root (e.g.: /index.html), and a 'Host' header indicating which virtual server to use.
- Proxy requests are used when the browser communicates with an HTTP proxy. These requests include a fully specified URL (e.g.: http://www.example.com/index.html).

The type of the incoming request is determined from the request URL, even if the Proxy-connection header exists. As there is no clear distinction between the two request types, the type of the request cannot always be accurately detected automatically, though all common cases are covered.

Requests are handled differently in transparent and non-transparent modes.

- A transparent HTTP proxy (*transparent_mode* attribute is TRUE) is meant to be installed in front of a network where clients do not know about the presence of the firewall. In this case the proxy expects to see server type requests only. If clients communicate with a real HTTP proxy through the firewall, proxy type requests must be explicitly enabled using the *permit_proxy_requests* attribute, or transparent mode has to be used.
- The use of non-transparent HTTP proxies (*transparent_mode* attribute is FALSE) must be configured in web browsers behind the firewall. In this case only proxy requests are expected, and server requests are emitted (assuming *parent_proxy* is not set).

4.6.2.6. Using parent proxies

Parent proxies are non-transparent HTTP proxies used behind PNS. Two things have to be set in order to use parent proxies. First, select a router which makes the proxy connect to the parent proxy, this can be either InbandRouter() or DirectedRouter(). Second, set the *parent_proxy* and *parent_proxy_port* attributes in the HttpProxy class. Setting these attributes results in proxy requests to be emitted to the target server both in transparent and non-transparent mode.

The parent proxy attributes can be set both in the configuration phase (e.g.: config() event), or later on a per-request basis. This is possible because the proxy re-connects.

Example 4.12. Using parent proxies in HTTP
In this example the MyHttp proxy class uses a parent proxy. For this the domain name and address of the parent proxy is specified, and a service using an InbandRouter is created.

```
class MyHttp(HttpProxy):
        def config(self):
                HttpProxy.config(self)
                self.parent_proxy = "proxy.example.com"
                self.parent_proxy_port = 3128

def instance():
        Service("http", MyHttp, router=InbandRouter())
        Listener(SockAddrInet('10.0.0.1', 80), "http")
```

4.6.2.7. FTP over HTTP

In non-transparent mode it is possible to let PNS process ftp:// URLs, effectively translating HTTP requests to FTP requests on the fly. This behaviour can be enabled by setting *permit_ftp_over_http* parameter to TRUE and adding port 21 to *target_port_range*. Currently only passive mode transfers are supported.

4.6.2.8. Error messages

There are cases when the HTTP proxy must return an error page to the client to indicate certain error conditions. These error messages are stored as files in the directory specified by the *error_files_directory* attribute, and can be customized by changing the contents of the files in this directory.

Each file contains plain HTML text, but some special macros are provided to dynamically add information to the error page. The following macros can be used:

- *@INFO@* -- further error information as provided by the proxy

- *@VERSION@* -- PNS version number
- *@DATE@* -- current date
- *@HOST@* -- hostname of PNS

It is generally recommended not to display error messages to untrusted clients, as they may leak confidential information. To turn error messages off, set the *`error_silent`* attribute to TRUE, or strip error files down to a minimum.

Note
The language of the messages can be set using the *`config.options.language`* global option, or individually for every Http proxy using the *`language`* parameter. See *Appendix B, Global options of PNS (p. 330)* for details.

4.6.2.9. Stacking

HTTP supports stacking proxies for both request and response entities (e.g.: data bodies). This is controlled by the *`request_stack`* and *`response_stack`* attribute hashes. See also *Section 2.3.1, Proxy stacking (p. 7)*.

There are two stacking modes available: HTTP_STK_DATA sends only the data portion to the downstream proxy, while HTTP_STK_MIME also sends all header information to make it possible to process the data body as a MIME envelope. Please note that while it is possible to change the data part in the stacked proxy, it is not possible to change the MIME headers - they can be modified only by the HTTP proxy. The possible parameters are listed in the following tables.

Action	Description
HTTP_STK_NONE	No additional proxy is stacked into the HTTP proxy.
HTTP_STK_DATA	The data part of the HTTP traffic is passed to the specified stacked proxy.
HTTP_STK_MIME	The data part including header information of the HTTP traffic is passed to the specified stacked proxy.

Table 4.11. Constants for proxy stacking

Please note that stacking is skipped altogether if there is no body in the message.

4.6.2.10. Webservers returning data in 205 responses

Certain webserver applications may return data entities in 205 responses. This is explicitly prohibited by the RFCs, but such responses are permitted for interoperability reasons.

4.6.2.11. Session persistence in load balancing

HTTP proxy offers the ‘session persistence in load balancing’ feature, further enhancing load balancing capabilities by that.

With the help of this feature, the Round Robin chainer can identify connections by their session IDs and make sure that every connection with the same session ID is always addressed to the same server, so that the session persists.

For using the ‘session persistence in load balancing’ feature, the administrator has to configure the following three attributes for the HTTP proxy:

- *Enable_session_persistence*
  You can switch on or off the ‘Session persistence in load balancing’ feature with that parameter.

- *Session_persistence_cookie_name*
  This parameter can only be configured if *enable_session_persistence* is set to TRUE. The administrator can provide the name of the cookie here: All incoming requests are directed to a web server and each web server sends a session ID back. The name of this session ID, that is, the cookie name, can be provided here to ensure that requests with the same session ID are directed to the same web server.

- *Session_persistence_cookie_salt*
  This parameter can only be configured if enable_session_persistence is set to TRUE. The administrator can provide the salt here, with which the IP address of the web server can be hashed before the session ID. With the help of this hashed information PNS can next time identify to which server the next connection attempt of this session has to be directed.

4.6.2.12. URL filtering in HTTP

The integrated category-based URL filtering solution maintains a permanently updated database. During the installation of the URL filtering database the administrator can choose the size of the URL filtering database. The database can be a smaller-sized, *optimized database* (the recommended version) for usual scenarios, which requires 1 GB storage space and 300 MB daily update traffic, or a *normal database* for more extensive scenarios, which requires 6 GB storage space and 2 GB daily update traffic. Note, that during the upgrade, the 6 GB storage space usage will temporarily increase to 18-20 GB.

- For an overview on how URL-filtering works, see *Section How URL filtering works (p. 56)*.
- For information on installing and selecting the size of the URL filter database, see *Procedure 4.1.2, Configuring the pns-common package* in *Proxedo Network Security Suite 2 Installation Guide*.
- To configure URL filtering, see *Section Configuring URL filtering in HTTP (p. 57)*.
- To customize or expand the URL-database, see *Section Customizing the URL database (p. 66)*.
- For the list of categories available by default, see *Section List of URL-filtering categories (p. 61)*.

How URL filtering works

The PNS URL filter checks the URL of the HTTP requests and compares them to a categorized database. The URLs and the domains in the database are organized into thematic categories like `adult`, `health`, `business`, and so on. If the requested URL is listed in the database, the categories matching the URL are assigned to the request. PNS can accept, redirect, or reject the HTTP request based on the category it belongs to.

Typically, accessing a webpage involves several HTTP requests. URL filtering is applied to every single HTTP request, meaning that different parts of the webpage can belong to different categories. PNS can remove the parts belonging to unwanted categories, and permit access only to the remaining part. For example, this can be used to prohibit access to advertisements or videos, but to permit access to other parts of the website.

Note
URL filtering is handled by the PNS Http proxy, without the need of using CF. The URL filtering capability of PNS is available only after purchasing the *`url-filter`* license option.

Updates to the URL database are automatically downloaded daily from the BalaSys website using the `vavupdate` utility.

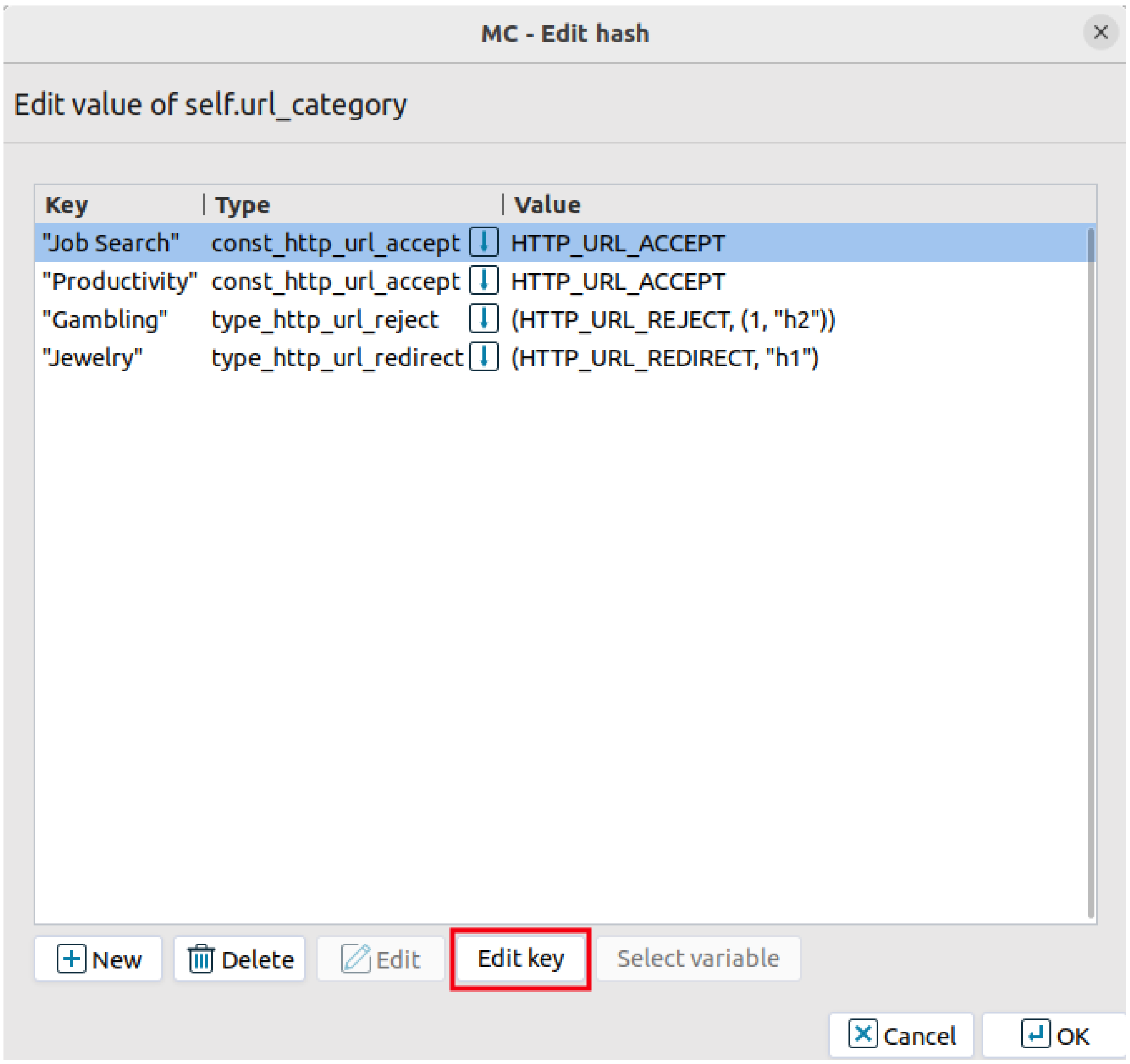
Configuring URL filtering in HTTP

Prior to the update to the new URL filter, it is necessary to check whether at least 1.7 GB free space is avaialble on the partition including the `/var/lib/vela/urlfilter/` folder. Following the update, the partition will utilize close to 850 MB space.

To enable url-filtering, follow the forthcoming steps:

Step 1. Choose the proper URL filter proxy under *PNS component* → *Proxies* menu item.

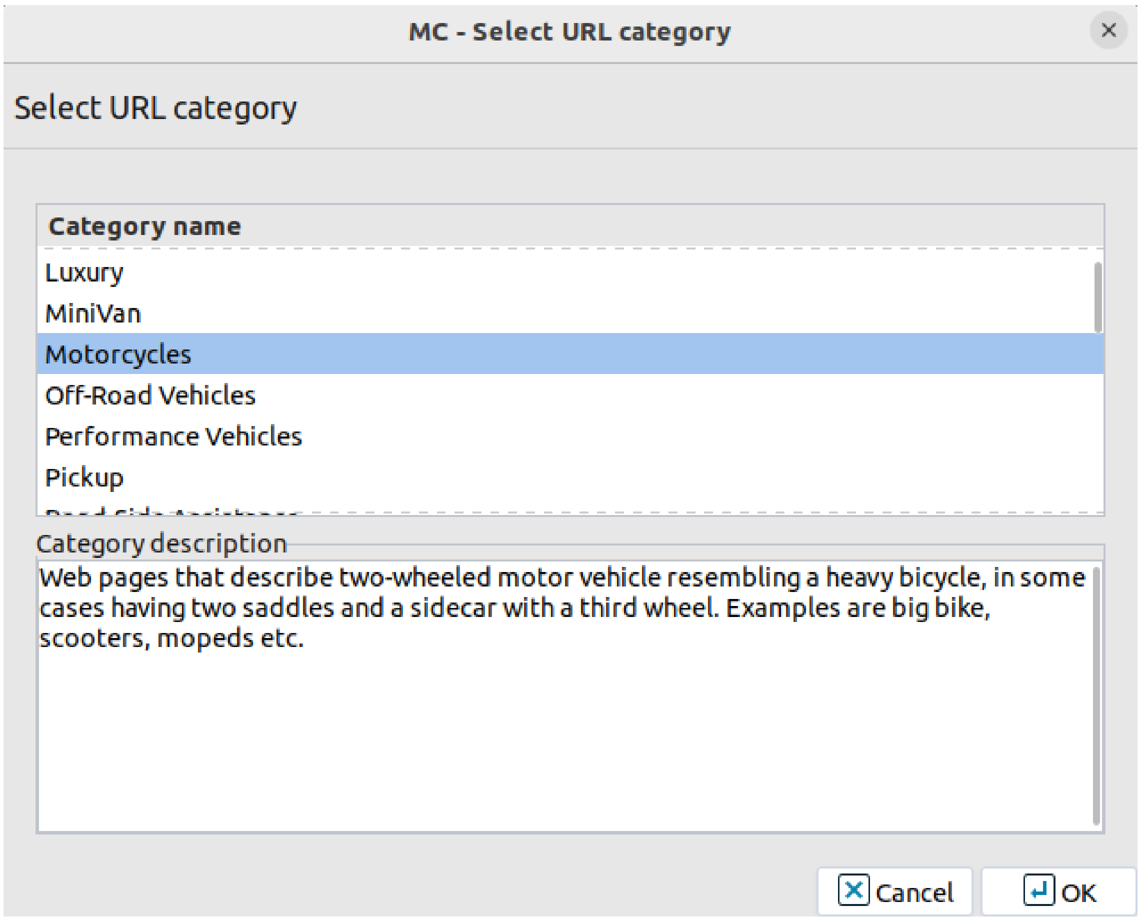
Step 2. Edit the *self.url_category* under the *Attribute* menu item.
You can find the categories from the previous version of the URL filter under the *Key* menu item. These items now belong to the 'uncategorized' group category.



Key	Type	Value
"Job Search"	const_http_url_accept	HTTP_URL_ACCEPT
"Productivity"	const_http_url_accept	HTTP_URL_ACCEPT
"Gambling"	type_http_url_reject	(HTTP_URL_REJECT, (1, "h2"))
"Jewelry"	type_http_url_redirect	(HTTP_URL_REDIRECT, "h1")

Figure 4.1. self.url_category attribute with the old categories yet

Step 3. Click the *Edit key* button and the selection of the new categories will be displayed:



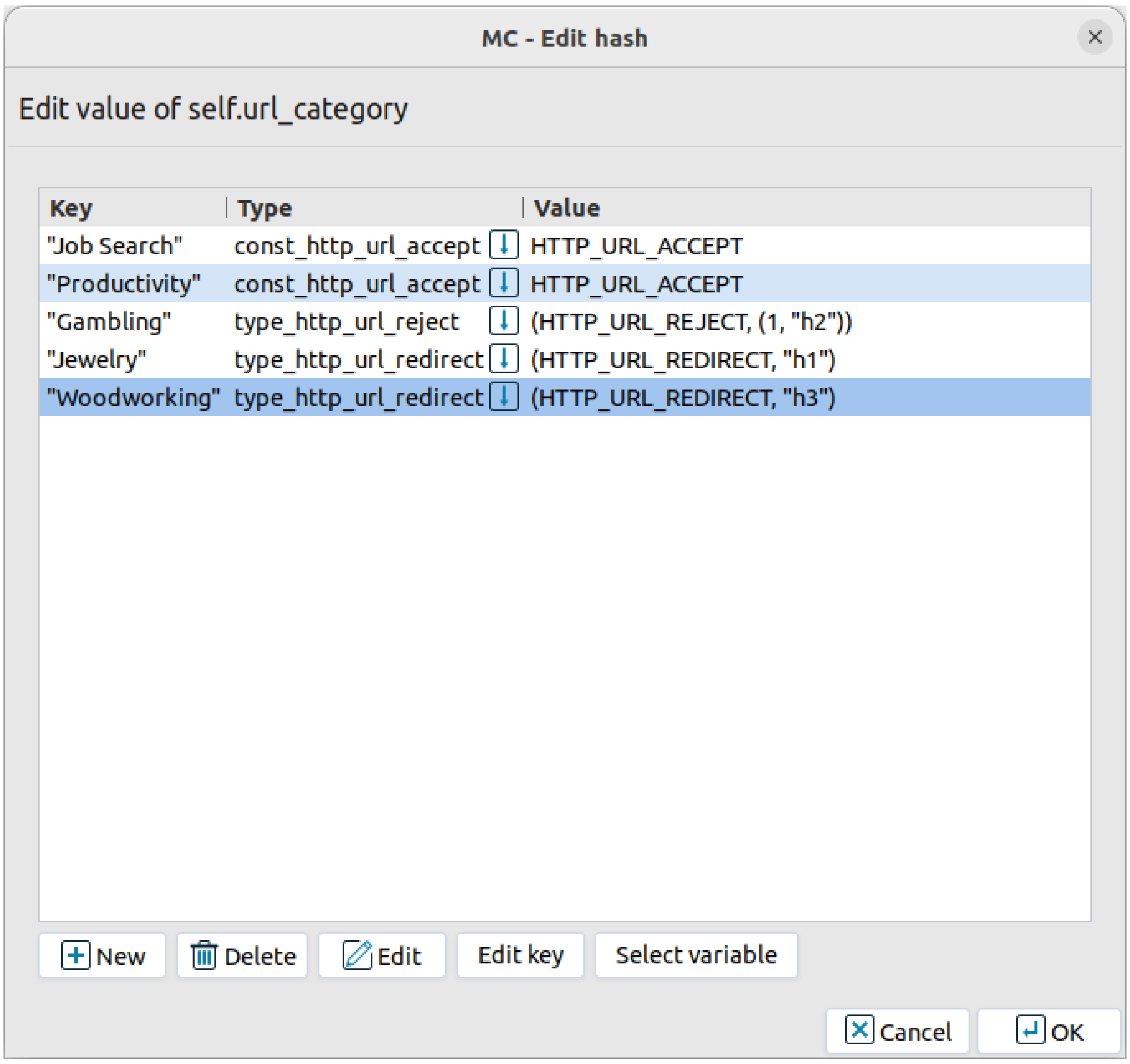
Web pages that describe two-wheeled motor vehicle resembling a heavy bicycle, in some cases having two saddles and a sidecar with a third wheel. Examples are big bike, scooters, mopeds etc.

Figure 4.2. The menu of the new category selection

Step 4. Choose the new category corresponding to the old one, from the list.

For the list of categories available by default, see *Section List of URL-filtering categories (p. 61)*. It is not necessary to change all the categories at once. The remaining elements will still remain as 'uncategorized' in the background.

While searching for the categories, it might be helpful to use the *Ctrl* + *F* key combination. Alternatively, when starting typing, the findings matching the first keystrokes will pop up in a list of highlights. Among these highlighted matches we can choose with the 'up' and 'down' arrows.



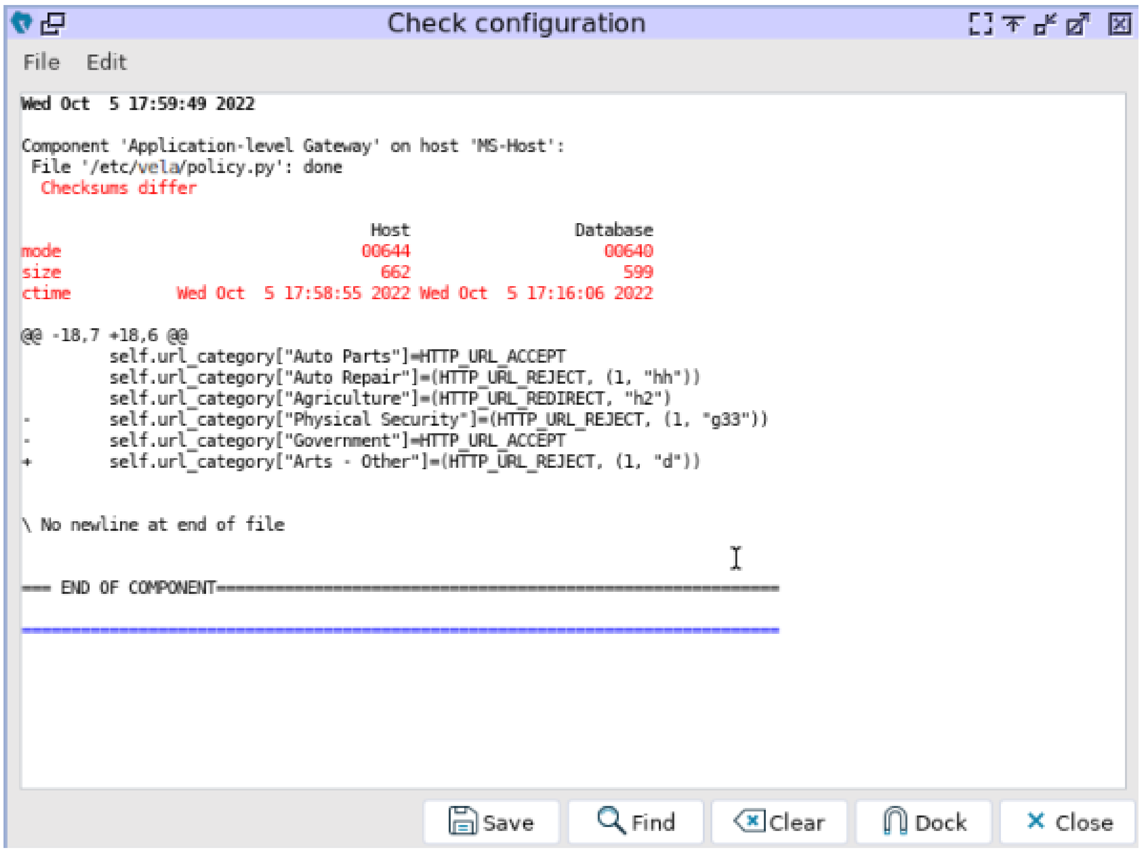
MC - Edit hash

Edit value of self.url_category

Key	Type	Value
"Job Search"	const_http_url_accept	HTTP_URL_ACCEPT
"Productivity"	const_http_url_accept	HTTP_URL_ACCEPT
"Gambling"	type_http_url_reject	(HTTP_URL_REJECT, (1, "h2"))
"Jewelry"	type_http_url_redirect	(HTTP_URL_REDIRECT, "h1")
"Woodworking"	type_http_url_redirect	(HTTP_URL_REDIRECT, "h3")

Figure 4.3. self.url_category attribute with partly new categories

Step 5. After pressing the *commit* button, validate the changes with the *check configuration* option.



```
Wed Oct  5 17:59:49 2022

Component 'Application-level Gateway' on host 'MS-Host':
 File '/etc/vela/policy.py': done
  Checksums differ

                                      Host                 Database
mode                                 00644                    00640
size                                   662                      599
ctime            Wed Oct  5 17:58:55 2022 Wed Oct  5 17:16:06 2022

@@ -18,7 +18,6 @@
         self.url_category["Auto Parts"]=HTTP_URL_ACCEPT
         self.url_category["Auto Repair"]=(HTTP_URL_REJECT, (1, "hh"))
         self.url_category["Agriculture"]=(HTTP_URL_REDIRECT, "h2")
-        self.url_category["Physical Security"]=(HTTP_URL_REJECT, (1, "g33"))
-        self.url_category["Government"]=HTTP_URL_ACCEPT
+        self.url_category["Arts - Other"]=(HTTP_URL_REJECT, (1, "d"))


\ No newline at end of file

=== END OF COMPONENT=============================================================

=================================================================================
```

Figure 4.4. Validating configuration

The update of the URL filter database is performed by the `vavupdate` command, run by the *cron* job, every day at 11 pm by default. Use the login name and password necessary for accessing the *apt repo* as well. All these ensure a solution that can be updated more easily.

List of URL-filtering categories

The PNS URL database contains the following thematic categories by default.

Main category	Subcategory
Adult	Abortion, Abortion Pro Choice, Abortion Pro Life, Child Inappropriate, Gambling, Gay, Lesbian, Bisexual, Lingerie, Suggestive and Pinap, Nudity, Pornography, R-Rated, Sex and Erotic, Sex Education and Pregnancy, Tobacco
Aggressive	Military, Violence, Weapons, Aggressive - Other
Arts	Fine Art, Arts - Other

Main category	Subcategory
Automotive	Auto Parts, Auto Repair, Buying/Selling Cars, Car Culture, Certified Pre-Owned, Convertible, Coupe, Crossover, Diesel, Electric Vehicle, Hatchback, Hybrid, Luxury, MiniVan, Motorcycles, Off-Road Vehicles, Performance Vehicles, Pickup, Road-Side Assistance, Sedan, Trucks & Accessories, Vintage Cars, Wagon, Automotive - Other
Business	Agriculture, Biotechnology, Business Software, Construction, Forestry, Government, Green Solutions & Conservation, Home & Office Furnishings, Human Resources, Manufacturing, Marketing Services, Metals, Physical Security, Productivity, Retirement Homes & Assisted Living, Shipping & Logistics,Business - Other
Careers	Career Advice, Career Planning, College, Financial Aid, Job Fairs, Job Search, Nursing, Resume Writing/Advice, Scholarships, Telecommuting, U.S. Military, Careers - Other
Criminal Activities	Child Abuse Images, Criminal Skills, Hacking, Hate Speech, Illegal Drugs, Marijuana, Piracy & Copyright Theft, School Cheating, Self Harm, Torrent Repository, Criminal Activities - Other
Dynamic	Anonymizer, Chat, Community Forums, Instant Messenger, Login Screens, Personal Pages & Blogs, Photo Sharing, Professional Networking, Redirect, Social Networking, Text Messaging & SMS, Translator, Web-based Email, Web-based Greeting Card
Education	7-12 Education, Adult Education, Art History, College Administration, College Life, Distance Learning, Educational Institutions, Educational Materials & Studies, English as a 2nd Language, Graduate School, Homeschooling, Homework/Study Tips, K-6 Educators, Language Learning, Literature & Books, Private School, Reference Materials & Maps, Special Education, Studying Business, Tutoring, Wikis, Education - Other

Main category	Subcategory
Entertainment	Entertainment News & Celebrity Sites, Entertainment Venues & Events, Humor, Movies, Music, Streaming & Downloadable Audio, Streaming & Downloadable Video, Television, Entertainment - Other
Family and Parenting	Adoption, Babies and Toddlers, Daycare/Pre School, Eldercare, Family Internet, Parenting - K-6 Kids, Parenting Teens, Pregnancy, Special Needs Kids, Family & Parenting - Other
Fashion	Accessories, Beauty, Body Art, Clothing, Fashion, Jewelry, Swimsuits, Fashion - Other
Finance	Accounting, Banking, Beginning Investing, Credit/Debt & Loans, Financial News, Financial Planning, Hedge Fund, Insurance, Investing, Mutual Funds, Online Financial Tools & Quotes, Options, Retirement Planning, Stocks, Tax Planning, Finance - Other
Food and Drink	American Cuisine, Barbecues & Grilling, Cajun/Creole, Chinese Cuisine, Cocktails/Beer, Coffee/Tea, Cuisine-Specific, Desserts & Baking, Dining Out, Food Allergies, French Cuisine, Health/Low fat Cooking, Italian Cuisine, Japanese Cuisine, Mexican Cuisine, Vegan, Vegetarian, Winer, Food & Drink - Other
Health	A.D.D., AIDS/HIV, Allergies, Alternative Medicine, Arthritis, Asthma, Autism/PDD, Bipolar Disorder, Brain Tumor, Cancer, Children's Health, Cholesterol, Chronic Fatigue, Chronic Pain, Cold & Flu, Cosmetic Surgery, Deafness, Dental Care, Depression, Dermatology, Diabetes, Disorders, Epilepsy, Exercise, GERD/Acid Reflux, Headaches/Migraines, Heart Disease, Herbs for Health, Holistic Healing, IBS/Crohn’s Disease, Incest/Abuse Support, Incontinence, Infertility, Men’s Health, Nutrition & Diet, Orthopedics, Panic/Anxiety, Pediatrics, Pharmaceuticals, Physical Therapy, Psychology/PsychiatrySelf-help & Addiction, Senior Health, Sexuality, Sleep Disorders, Smoking Cessation, Supplements & Compounds, Syndrome, Thyroid Disease, Weight Loss, Women’s Health, Health - Other

Main category	Subcategory
Hobbies and Interests	Art/Technology, Arts & Crafts, Beadwork, Birdwatching, Board Games/Puzzles, Candle & Soap Making, Card Games, Cartoons, Anime & Comic Books, Chess, Cigars, Collecting, Comic Books, Drawing/Sketching, Freelance Writing, Genealogy, Getting Published, Guitar, Home Recording,Investors & Patents, Jewelry Making, Magic & Illusion, Needlework, Painting, Photography, Radio, Roleplaying Games, Sci-Fi & Fantasy, Scrapbooking, Screenwriting, Stamps & Coins, Themes, Video & Computer Games, Woodworking, Hobbies & Interests - Other
House and Garden	Appliances, Entertaining, Environmental Safety, Gardening, Home Repair., Home Theater, Interior Decorating, Landscaping, Remodeling & Construction, Home & Garden - Other
Kids	Games, Kid's Pag, Toys, Kids - Other
Lifestyle	Dating & Relationships, Divorce Support, Ethnic Specific, Marriage, Parks, Rec Facilities & Gyms, Senior Living, Teens, Weddings, Lifestyle - Other
Malicious	Ad Fraud, Botnet, Command and Control Centers, Compromised & Links To Malware, Malware Call-Home, Malware Distribution Point, Phishing/Fraud, Spam URLs, Spyware & Questionable Software
Miscellaneous	Content Server, No Content Found, Parked & For Sale Domains, Private IP Address, Unreachable, Miscellaneous - Other
News, Portal and Search	Image Search, International News, Local News, Magazines, National News, Portal Sites, Search Engines, News, Portal & Search - Other
Online Ads	Pay-to-Surf, Online Ads - Other
Pets	Aquariums, Birds, Cats, Dogs, Large Animals, Reptiles, Veterinary Medicine, Pets - Other

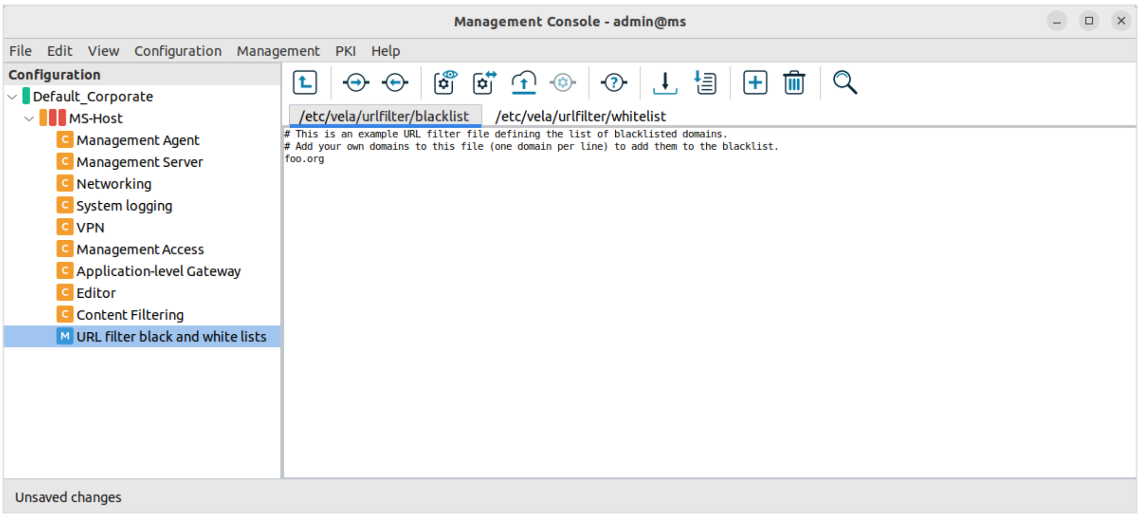
Main category	Subcategory
Public, Government and Law	Advocacy Groups & Trade Associations, Commentary, Government Sponsored, Immigration, Legal Issues, Philanthropic Organizations, Politics, Social & Affiliation Organizations, U.S. Government Resources, Public, Government & Law - Other
Real Estate	Apartments, Architects, Buying/Selling Homes, Real Estate - Other
Religion	Alternative Religions, Atheism & Agnosticism, Buddhism, Catholicism, Christianity, Hinduism, Islam, Judaism, Latter-Day Saints, Non-traditional Religion and Occult, Pagan/Wiccan, Religion - Other
Science	Anatomy, Astrology and Horoscopes, Biology, Botany, Chemistry, Weather, Geography, Geology, Paranormal Phenomena, Physics, Space/Astronomy, Science - Other
Shopping	Auctions & Marketplaces, Catalogs, Contests & Surveys, Shopping - Online, Engines, Product Reviews & Price Comparisons, Coupons, Shopping - Other
Sports	Auto Racing, Baseball, Bicycling, Bodybuilding, Boxing, Canoeing/Kayaking, Cheerleading, Climbing, Cricket, Figure Skating, Fly Fishing, Football, Freshwater Fishing, Game & Fish, Golf, Horse Racing, Horses, Inline Skating, Martial Arts, Mountain Biking, NASCAR Racing, Olympics, Sports - Other, Paintball, Power & Motorcycles, Pro Basketball, Pro Ice Hockey, Rodeo, Rugby, Running/Jogging, Sailing, Saltwater Fishing, Scuba Diving, Skateboarding, Skiing, Snowboarding, Sport Hunting, Surfing/Bodyboarding, Swimming, Table Tennis/Ping-Pong, Tennis, Volleyball, Walking, Waterski/Wakeboard, World Soccer
Technology	3-D Graphics, Animation, Antivirus Software, C/C++, Cameras & Camcorders, Computer Certification, Computer Networking, Computer Peripherals, Computer Reviews, Databases, Desktop Publishing, Desktop Video, File Repositories, Graphics Software, Home Video/DVD, Information Security, Internet Phone & VOIP, Internet Technology, Java, Javascript,

Main category	Subcategory
	Linux, Mac OS, Technology - Other, Mac Support, Mobile Phones, MP3/MIDI, Net Conferencing, Net for Beginners, Network Security, Palmtops/PDAs, PC Support, Peer-to-Peer, Personal Storage, Portable, Entertainment, Remote Access, Shareware/Freeware, Unix, Utilities, Visual Basic, Web Clip Art, Web Design/HTML, Web Hosting, ISP & Telco, Windows, Online Information Management
Travel	Adventure Travel, Africa, Air Travel, Australia & New Zealand, Bed & Breakfast, Budget Travel, Business Travel, By US Locale, Camping, Canada, Caribbean, Cruises, Eastern Europe, Europe, Travel - Other, France, Greece, Honeymoons/Getaways, Hotels, Italy, Japan, Mexico & Central America, National Parks, Navigation, South America, Spas, Theme Parks, Traveling with Kids, United Kingdom

Table 4.12. URL filter categories

Customizing the URL database

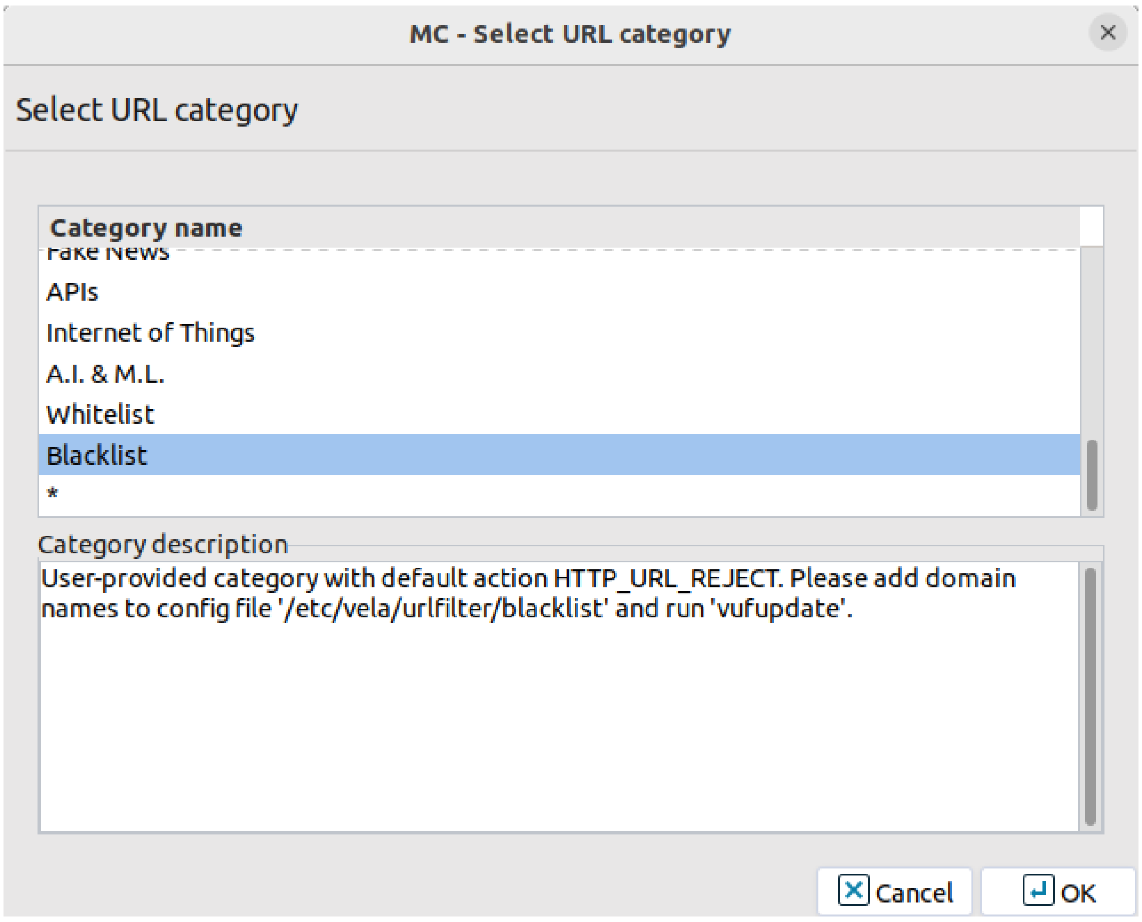
Create blacklist and whitelist if necessary. This can be achieved with the help of free text editor, on the `PNS Node` → `New` → `Text editor` → `URL filter black and whitelists` path.



```
# This is an example URL filter file defining the list of blacklisted domains.
# Add your own domains to this file (one domain per line) to add them to the blacklist.
foo.org
```

Figure 4.5. Editor for URL filter blacklist and whitelist

The choices for blacklist, whitelist or * options are also put into the category list:



User-provided category with default action HTTP_URL_REJECT. Please add domain names to config file '/etc/vela/urlfilter/blacklist' and run 'vufupdate'.

Figure 4.6. The 'whitelist' and 'blacklist' options are at the end of the category list

4.6.3. Related standards

- The Hypertext Transfer Protocol -- HTTP/1.1 protocol is described in RFC 2616.
- The Hypertext Transfer Protocol -- HTTP/1.0 protocol is described in RFC 1945.

4.6.4. Classes in the Http module

Class	Description
AbstractHttpProxy	Class encapsulating the abstract HTTP proxy.
HttpProxy	Default HTTP proxy based on AbstractHttpProxy.
HttpProxyNonTransparent	HTTP proxy based on HttpProxy, operating in non-transparent mode.
HttpProxyURIFilter	HTTP proxy based on HttpProxy, with URI filtering capability.

Class	Description
HttpProxyURIFilterNonTransparent	HTTP proxy based on HttpProxyURIFilter, with URI filtering capability and permitting non-transparent requests.
HttpProxyURLCategoryFilter	HTTP proxy based on HttpProxy, with URL filtering capability based on categories.
HttpWebdavProxy	HTTP proxy based on HttpProxy, allowing WebDAV extensions.
NontransHttpWebdavProxy	HTTP proxy based on HttpProxyNonTransparent, allowing WebDAV extension in non-transparent requests.

Table 4.13. Classes of the Http module

4.6.5. Class AbstractHttpProxy

This class implements an abstract HTTP proxy - it serves as a starting point for customized proxy classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractHttpProxy, or one of the predefined proxy classes, such as *HttpProxy* or *HttpProxyNonTransparent*. AbstractHttpProxy denies all requests by default.

4.6.5.1. Attributes of AbstractHttpProxy

auth_by_cookie (boolean, rw:r)
Default: FALSE
Authentication informations for one-time-password mode is organized by a cookie not the address of the client.

auth_by_form (boolean, rw:r)
Default: FALSE
When enabled, and the client tries to access an URL that requires authentication, a webpage where users can enter their authentication information is displayed. If the authentication is successful, the result is cached in a cookie.

auth_cache_time (integer, rw:r)
Default: 0
Caching authentication information this amount of seconds.

auth_cache_update (boolean, rw:r)
Default: FALSE
Update authentication cache by every connection.

auth_forward (boolean, rw:rw)
Default: FALSE
Controls whether inband authentication information (username and password) should be forwarded to the upstream server. When a parent proxy is present, the incoming authentication request is put into a 'Proxy-Authorization' header. In other cases the 'WWW-Authorization' header is used.

auth_realm (string, w:r)
Default: "Vela HTTP auth"
The name of the authentication realm to be presented to the user in the dialog window during inband authentication.

buffer_size (integer, rw:r)
Default: 1500
Size of the I/O buffer used to transfer entity bodies.

connect_proxy (class, rw:rw)
Default: PlugProxy
For CONNECT requests the HTTP proxy starts an independent proxy to control the internal protocol. The connect_proxy attribute specifies which proxy class is used for this purpose.

connection_mode (enum, n/a:rw)
Default: n/a
This value reflects the state of the session. If the value equals to 'HTTP_CONNECTION_CLOSE', the session will be closed after serving the current request. Otherwise, if the value is 'HTTP_CONNECTION_KEEPALIVE' another request will be fetched from the client. This attribute can be used to forcibly close a keep-alive connection.

current_header_name (string, n/a:rw)
Default: n/a
Name of the header. It is defined when the header is processed, and can be modified by the proxy to actually change a header in the request or response.

current_header_value (string, n/a:rw)
Default: n/a
Value of the header. It is defined when the header is processed, and can be modified by the proxy to actually change the value of the header in the request or response.

default_port (integer, rw:rw)
Default: 80
This value is used in non-transparent mode when the requested URL does not contain a port number. The default should be 80, otherwise the proxy may not function properly.

enable_session_persistence (boolean, rw:rw)
Default: FALSE
Allow persistent load balanced connections when accessing session-aware application servers.

enable_url_filter (boolean, rw:r)
Default: FALSE
Enables URL filtering in HTTP requests. See *Section 4.6.2.12, URL filtering in HTTP (p. 56)* for details.

enable_url_filter_dns (boolean, rw:r)
Default: FALSE
Enables DNS- and reverse-DNS resolution to ensure that a domain or URL is correctly categorized even when it is listed in the database using its domain name, but the client tries to access it with its IP address (or vice-versa). See *Section 4.6.2.12, URL filtering in HTTP (p. 56)* for details.

error_files_directory (string, rw:rw)
Default: "/usr/share/vela/http"
Location of HTTP error messages.

error_headers (string, n/a:rw)
Default: n/a
A string included as a header in the error response. The string must be a valid header and must end with a " " sequence.

error_info (string, n/a:rw)
Default: n/a
A string to be included in error messages.

error_msg (string, n/a:rw)
Default: n/a
A string used as an error message in the HTTP status line.

error_silent (boolean, rw:rw)
Default: FALSE
Turns off verbose error reporting to the HTTP client (makes firewall fingerprinting more difficult).

error_status (integer, rw:rw)
Default: 500
If an error occurs, this value will be used as the status code of the HTTP response it generates.

keep_persistent (boolean, rw:r)
Default: FALSE
Try to keep the connection to the client persistent even if the server does not support it.

language (string, rw:r)
Default: "en"
Specifies the language of the HTTP error pages displayed to the client. English (*en*) is the default. Other supported languages: *de* (German); *hu* (Hungarian).

max_auth_time (integer, rw:rw)
Default: 0
Request password authentication from the client, invalidating cached one-time-passwords. If the time specified (in seconds) in this attribute expires, a new authentication from the client browser is requested even if it still has a password cached.

max_body_length (integer, rw:rw)
Default: 0
Maximum allowed length of an HTTP request or response body. The default "0" value means that the length of the body is not limited.

max_chunk_length (integer, rw:rw)
Default: 0
Maximum allowed length of a single chunk when using chunked transfer-encoding. The default "0" value means that the length of the chunk is not limited.

max_header_lines (integer, rw:rw)
Default: 50
Maximum number of header lines allowed in a request or response.

max_hostname_length (integer, rw:rw)
Default: 256
Maximum allowed length of the hostname field in URLs.

max_keepalive_requests (integer, rw:rw)
Default: 0
Maximum number of requests allowed in a single session. If the number of requests in the session the reaches this limit, the connection is terminated. The default "0" value allows unlimited number of requests.

max_line_length (integer, rw:r)
Default: 4096
Maximum allowed length of lines in requests and responses. This value does not affect data transfer, as data is transmitted in binary mode.

max_url_length (integer, rw:rw)
Default: 4096
Maximum allowed length of an URL in a request. Note that this directly affects forms using the 'GET' method to pass data to CGI scripts.

parent_proxy (string, rw:rw)
Default: ""
The address or hostname of the parent proxy to be connected. Either DirectedRouter or InbandRouter has to be used when using parent proxy.

parent_proxy_port (integer, rw:rw)
Default: 3128
The port of the parent proxy to be connected.

permit_ftp_over_http (boolean, rw:r)
Default: FALSE
Allow processing FTP URLs in non-transparent mode.

permit_http09_responses (boolean, rw:r)
Default: TRUE
Allow server responses to use the limited HTTP/0.9 protocol. As these responses carry no control information, verifying the validity of the protocol stream is impossible. This does not pose a threat to web clients, but

permit_http09_responses (boolean, rw:r)
exploits might pass undetected if this option is enabled for servers. It is recommended to turn this option off for protecting servers and only enable it when Vela is used in front of users.

permit_invalid_hex_escape (boolean, rw:r)
Default: FALSE
Allow invalid hexadecimal escaping in URLs (% must be followed by two hexadecimal digits).

permit_null_response (boolean, rw:r)
Default: TRUE
Permit RFC incompliant responses with headers not terminated by CRLF and not containing entity body.

permit_proxy_requests (boolean, rw:r)
Default: FALSE
Allow proxy-type requests in transparent mode.

permit_server_requests (boolean, rw:r)
Default: TRUE
Allow server-type requests in non-transparent mode.

permit_unicode_url (boolean, rw:r)
Default: FALSE
Allow unicode characters in URLs encoded as %u. This is an IIS extension to HTTP, UNICODE (UTF-7, UTF-8 etc.) URLs are forbidden by the RFC as default.

request (complex, rw:rw)
Default: empty
Normative policy hash for HTTP requests indexed by the HTTP method (e.g.: "GET", "PUT" etc.). See also *Section 4.6.2.2, Configuring policies for HTTP requests and responses (p. 50)*.

request_count (integer, n/a:r)
Default: 0
The number of keepalive requests within the session.

request_header (complex, rw:rw)
Default: empty

request_header (complex, rw:rw)
Normative policy hash for HTTP header requests indexed by the header names (e.g.: "Set-cookie"). See also *Section 4.6.2.3, Configuring policies for HTTP headers (p. 52)*.

request_method (string, n/a:r)
Default: n/a
Request method (GET, POST, etc.) sent by the client.

request_mime_type (string, n/a:r)
Default: n/a
The MIME type of the request entity. Its value is only defined when the request is processed.

request_stack (complex, rw:rw)
Default: n/a
Attribute containing the request stacking policy: the hash is indexed based on method names (e.g.: GET). See *Section 4.6.2.9, Stacking (p. 55)*.

request_url (string, n/a:rw)
Default: n/a
The URL requested by the client. It can be modified to redirect the current request.

request_url_file (string, n/a:r)
Default: n/a
Filename specified in the URL.

request_url_host (string, n/a:r)
Default: n/a
Remote hostname in the URL.

request_url_passwd (string, n/a:r)
Default: n/a
Password in the URL (if specified).

request_url_port (integer, n/a:r)
Default: n/a
Port number as specified in the URL.

request_url_proto (string, n/a:r)
Default: n/a
Protocol specifier of the URL. This attribute is an alias for *`request_url_scheme`*.

request_url_scheme (string, n/a:r)
Default: n/a
Protocol specifier of the URL (http://, ftp://, etc.).

request_url_username (string, n/a:r)
Default: n/a
Username in the URL (if specified).

request_version (string, n/a:r)
Default: n/a
Request version (1.0, 1.1, etc.) used by the client.

require_host_header (boolean, rw:r)
Default: TRUE
Require the presence of the Host header. If set to FALSE, the real URL cannot be recovered from certain requests, which might cause problems with URL filtering.

rerequest_attempts (integer, rw:rw)
Default: 0
Controls the number of attempts the proxy takes to send the request to the server. In case of server failure, a reconnection is made and the complete request is repeated along with POST data.

reset_on_close (boolean, rw:rw)
Default: FALSE
Whenever the connection is terminated without a proxy generated error message, send an RST instead of a normal close. Causes some clients to automatically reconnect.

response (complex, rw:rw)
Default: empty
Normative policy hash for HTTP responses indexed by the HTTP method and the response code (e.g.: "PWD", "209" etc.). See also *Section 4.6.2.2, Configuring policies for HTTP requests and responses (p. 50)*.

response_header (complex, rw:rw)
Default: empty
Normative policy hash for HTTP header responses indexed by the header names (e.g.: "Set-cookie"). See also *Section 4.6.2.3, Configuring policies for HTTP headers (p. 52)*.

response_mime_type (string, n/a:r)
Default: n/a
The MIME type of the response entity. Its value is only defined when the response is processed.

response_stack (complex, rw:rw)
Default: n/a
Attribute containing the response stacking policy: the hash is indexed based on method names (e.g.: GET). See *Section 4.6.2.9, Stacking (p. 55)*.

rewrite_host_header (boolean, rw:rw)
Default: TRUE
Rewrite the Host header in requests when URL redirection is performed.

server_response_time (integer, n/a:r)
Default: n/a
This value stores the time difference between sending the request to the server and receiving the response, in milliseconds.

session_persistence_cookie_name (string, rw:rw)
Default: "JSESSIONID"
The name of the cookie which will be used to persist load balanced connections when accessing session-aware application servers.

session_persistence_cookie_salt (string, rw:rw)
Default: n/a
The salt to use when hashing the target server addresses in persistent load balanced connections. If session persistence is enabled, this parameter must be set.

strict_header_checking (boolean, rw:r)
Default: FALSE
Require RFC conformant HTTP headers.

strict_header_checking_action (enum, rw:r)
Default: HTTP_HDR_DROP
This attribute controls what should happen if a non-rfc conform or unknown header found in the communication. Only the HTTP_HDR_ACCEPT, HTTP_HDR_DROP and HTTP_HDR_ABORT can be used.

target_port_range (string, rw:rw)
Default: "80,443"
List of ports that non-transparent requests are allowed to use. The default is to allow port 80 and 443 to permit HTTP and HTTPS traffic. (The latter also requires the CONNECT method to be enabled).

timeout (integer, rw:rw)
Default: 300000
General I/O timeout in milliseconds. If there is no timeout specified for a given operation, this value is used.

timeout_request (integer, rw:rw)
Default: 10000
Time to wait for a request to arrive from the client.

timeout_response (integer, rw:rw)
Default: 300000
Time to wait for the HTTP status line to arrive from the server.

transparent_mode (boolean, rw:r)
Default: TRUE
Set the operation mode of the proxy to transparent (TRUE) or non-transparent (FALSE).

url_category (complex, rw:rw)
Default: empty
Normative policy hash for category-based URL-filtering. The hash is indexed by the name of the category.

url_filter_uncategorized_action (enum, rw:rw)
Default: HTTP_URL_ACCEPT
The action applied to uncategorized (unknown) URLs when URL filtering is used. By default, uncategorized URLs are accepted: *`self.url_filter_uncategorized_action=(HTTP_URL_ACCEPT,)`*. Note that if you set this option to *`HTTP_URL_REJECT`*, you must add every URL on your intranet to a category and set an *`HTTP_URL_ACCEPT`* rule to this category, otherwise your clients will not able to access your intranet sites. For details, see *Section Configuring URL filtering in HTTP (p. 57)*.

use_canonicalized_urls (boolean, rw:rw)
Default: TRUE
This attribute enables URL canonicalization, which means to automatically convert URLs to their canonical form. This enhances security but might cause interoperability problems with some applications. It is recommended to disable this setting on a per-destination basis. URL filtering still sees the canonicalized URL, but at the end the proxy sends the original URL to the server.

use_default_port_in_transparent_mode (boolean, rw:rw)
Default: TRUE
Set the target port to the value of `default_port` in transparent mode. This ensures that only the ports specified in `target_port_range` can be used by the clients, even if InbandRouter is used.

4.6.5.2. AbstractHttpProxy methods

Method	**Description**
getRequestHeader(self, header)	Function returning the value of a request header.
getResponseHeader(self, header)	Function returning the value of a response header.
setRequestHeader(self, header, new_value)	Function changing the value of a request header.
setResponseHeader(self, header, new_value)	Function changing the value of a response header.

Table 4.14. Method summary

Method getRequestHeader(self, header)

This function looks up and returns the value of a header associated with the current request.

Arguments of getRequestHeader

header (unknown, n/a:n/a)
Default: n/a
Name of the header to look up.

Method getResponseHeader(self, header)

This function looks up and returns the value of a header associated with the current response.

Arguments of getResponseHeader

header (unknown, n/a:n/a)
Default: n/a
Name of the header to look up.

Method setRequestHeader(self, header, new_value)

This function looks up and changes the value of a header associated with the current request.

Arguments of setRequestHeader

header (unknown, n/a:n/a)
Default: n/a
Name of the header to change.

new_value (unknown, n/a:n/a)
Default: n/a
Change the header to this value.

Method setResponseHeader(self, header, new_value)

This function looks up and changes the value of a header associated with the current response.

Arguments of setResponseHeader

header (unknown, n/a:n/a)
Default: n/a
Name of the header to change.

new_value (unknown, n/a:n/a)
Default: n/a
Change the header to this value.

4.6.6. Class HttpProxy

HttpProxy is a default HTTP proxy based on AbstractHttpProxy. It is transparent, and enables the most commonly used HTTP methods: "GET", "POST" and "HEAD".

4.6.7. Class HttpProxyNonTransparent

HTTP proxy based on HttpProxy. This class is identical to *HttpProxy* with the only difference being that it is non-transparent (`transparent_mode = FALSE`). Consequently, clients must be explicitly configured to connect to this proxy instead of the target server and issue proxy requests. On the server side this proxy connects transparently to the target server.

For the correct operation the proxy must be able to set the server address on its own. This can be accomplished by using *InbandRouter*.

4.6.8. Class HttpProxyURIFilter

HTTP proxy based on HttpProxy, having URL filtering capability. The matcher attribute should be initialized to refer to a Matcher object. The initialization should be done in the class body as shown in the next example.

Example 4.13. URL filtering HTTP proxy

```
class MyHttp(HttpProxyURIFilter):
    matcher = RegexpFileMatcher('/etc/vela/blacklist.txt',
'/etc/vela/whitelist.txt')
```

4.6.8.1. Attributes of HttpProxyURIFilter

matcher (class, rw:rw)
Default: None
Matcher determining whether access to an URL is permitted or not.

4.6.9. Class HttpProxyURIFilterNonTransparent

HTTP proxy based on HttpProxyURIFilter, but operating in non-transparent mode (*`transparent_mode = FALSE`*).

4.6.10. Class HttpProxyURLCategoryFilter

HTTP proxy based on HttpProxy with enabled URL filtering (with DNS and reverse-DNS resolution) and preconfigured default category actions.

The following categories have policy action *`HTTP_URL_REJECT`*:

- Ad Fraud
- Blacklist
- Botnet
- Child Abuse Images
- Command and Control Centers
- Compromised & Links To Malware
- Criminal Skills
- Gambling
- Illegal Drugs
- Malware Call-Home
- Malware Distribution Point
- Phishing/Fraud
- Pornography

- Self Harm
- Sex & Erotic
- Spam URLs
- Spyware & Questionable Software
- Violence

The following categories have policy action *`HTTP_URL_ACCEPT`*:

- Whitelist

4.6.11. Class HttpWebdavProxy

HTTP proxy based on HttpProxy, also capable of inspecting WebDAV extensions of the HTTP protocol.

The following requests are permitted: PROPFIND; PROPPATCH; MKCOL; COPY; MOVE; LOCK; UNLOCK.

4.6.12. Class NontransHttpWebdavProxy

HTTP proxy based on HttpProxyNonTransparent, operating in non-transparent mode (*`transparent_mode = FALSE`*) and capable of inspecting WebDAV extensions of the HTTP protocol.

The following requests are permitted: PROPFIND; PROPPATCH; MKCOL; COPY; MOVE; LOCK; UNLOCK.

4.7. Module Plug

This module defines an interface to the Plug proxy. Plug is a simple TCP or UDP circuit, which means that transmission takes place without protocol verification.

4.7.1. Proxy behavior

This class implements a general plug proxy, and is capable of optionally disabling data transfer in either direction. Plug proxy reads connection on the client side, then creates another connection at the server side. Arriving responses are sent back to the client. However, it is not a protocol proxy, therefore PlugProxy does not implement any protocol analysis. It offers protection to clients and servers from lower level (e.g.: IP) attacks. It is mainly used to allow traffic pass the firewall for which there is no protocol proxy available.

By default plug copies all data in both directions. To change this behavior, set the *`copy_to_client`* or *`copy_to_server`* attribute to FALSE.

Plug supports the use of secondary sessions. For details, see *Section 2.2, Secondary sessions (p. 7)*.

Note
Copying of out-of-band data is not supported.

4.7.2. Related standards

Plug proxy is not a protocol specific proxy module, therefore it is not specified in standards.

4.7.3. Classes in the Plug module

Class	Description
AbstractPlugProxy	Class encapsulating the abstract Plug proxy.
PlugProxy	Class encapsulating the default Plug proxy.

Table 4.15. Classes of the Plug module

4.7.4. Class AbstractPlugProxy

An abstract proxy class for transferring data.

4.7.4.1. Attributes of AbstractPlugProxy

bandwidth_to_client (integer, n/a:r)
Default: n/a
Read-only variable containing the bandwidth currently used in server->client direction.

bandwidth_to_server (integer, n/a:r)
Default: n/a
Read-only variable containing the bandwidth currently used in client->server direction.

buffer_size (integer, w:r)
Default: 1500
Size of the buffer used for copying data.

copy_to_client (boolean, w:r)
Default: TRUE
Allow data transfer in the server->client direction.

copy_to_server (boolean, w:r)
Default: TRUE
Allow data transfer in the client->server direction.

packet_stats_interval_packet (integer, w:r)
Default: 0
The number of passing packages between two successive packetStats() events. It can be useful when the Quality of Service for the connection is influenced dynamically. Set to 0 to turn packetStats() off.

packet_stats_interval_time (integer, w:r)
Default: 0
The time in milliseconds between two successive packetStats() events. It can be useful when the Quality of Service for the connection is influenced dynamically. Set to 0 to turn packetStats() off.

secondary_mask (secondary_mask, rw:r)
Default: 0xf
Specifies which connections can be handled by the same proxy instance. See *Section 2.2, Secondary sessions (p. 7)* for details.

secondary_sessions (integer, rw:r)
Default: 10
Maximum number of allowed secondary sessions within a single proxy instance. See *Section 2.2, Secondary sessions (p. 7)* for details.

shutdown_soft (boolean, w:r)
Default: FALSE
If enabled, the two sides of a connection are closed separately. (E.g.: if the server closes the connection the client side connection is held until it is verified that no further data arrives, for example from a stacked proxy.) It is automatically enabled when proxies are stacked into the connection.

stack_proxy (enum, w:r)
Default: n/a
Proxy class to stack into the connection. All data is passed to the specified proxy.

timeout (integer, w:r)
Default: 600000
I/O timeout in milliseconds.

4.7.4.2. AbstractPlugProxy methods

Method	Description
packetStats(self, client_bytes, client_pkts, server_bytes, server_pkts)	Function called when the packet_stats_interval is elapsed.

Table 4.16. Method summary

Method packetStats(self, client_bytes, client_pkts, server_bytes, server_pkts)

This function is called whenever the time interval set in packet_stats_interval elapses, or a given number of packets were transmitted. This event receives packet statistics as parameters. It can be used in managing the Quality of Service of the connections; e.g.: to terminate connections with excessive bandwidth requirements (for instance to limit the impact of a covert channel opened when using plug instead of a protocol specific proxy).

Arguments of packetStats

client_bytes (unknown, n/a:n/a)
Default: n/a
Number of bytes transmitted to the client.

client_pkts (unknown, n/a:n/a)
Default: n/a
Number of packets transmitted to the client.

server_bytes (unknown, n/a:n/a)
Default: n/a
Number of bytes transmitted to the server.

server_pkts (unknown, n/a:n/a)
Default: n/a
Number of packets transmitted to the server.

4.7.5. Class PlugProxy

A default PlugProxy based on AbstractPlugProxy.

4.8. Module Pop3

The Pop3 module defines the classes constituting the proxy for the POP3 protocol.

4.8.1. The POP3 protocol

Post Office Protocol version 3 (POP3) is usually used by mail user agents (MUAs) to download messages from a remote mailbox. POP3 supports a single mailbox only, it does not support advanced multi-mailbox operations offered by alternatives such as IMAP.

The POP3 protocol uses a single TCP connection to give access to a single mailbox. It uses a simple command/response based approach, the client issues a command and a server can respond either positively or negatively.

4.8.1.1. Protocol elements

The basic protocol is the following: the client issues a request (also called command in POP3 terminology) and the server responds with the result. Both commands and responses are line based, each command is sent as a complete line, a response is either a single line or - in case of mail transfer commands - multiple lines.

Commands begin with a case-insensitive keyword possibly followed by one or more arguments (such as RETR or DELE).

Responses begin with a status indicator ("+OK" or "-ERR") and a possible explanation of the status code (e.g.: "-ERR Permission denied.").

Responses to certain commands (usually mail transfer commands) also contain a data attachment, such as the mail body. See the *Section 4.8.1.3, Bulk transfers (p. 85)* for further details.

4.8.1.2. POP3 states

The protocol begins with the server displaying a greeting message, usually containing information about the server.

After the greeting message the client takes control and the protocol enters the AUTHORIZATION state where the user has to pass credentials proving his/her identity.

After successful authentication the protocol enters TRANSACTION state where mail access commands can be issued.

When the client has finished processing, it issues a QUIT command and the connection is closed.

4.8.1.3. Bulk transfers

Responses to certain commands (such as LIST or RETR) contain a long data stream. This is transferred as a series of lines, terminated by a "CRLF '.' CRLF" sequence, just like in SMTP.

Example 4.14. POP3 protocol sample

```
+OK POP3 server ready
USER account
+OK User name is ok
PASS password
+OK Authentication successful
LIST
+OK Listing follows
1 5758
```

```
2 232323
3 3434
.
RETR 1
+OK Mail body follows
From: sender@sender.com
To: account@receiver.com
Subject: sample mail

This is a sample mail message. Lines beginning with
..are escaped, another '.' character is perpended which
is removed when the mail is stored by the client.
.
DELE 1
+OK Mail deleted
QUIT
+OK Good bye
```

4.8.2. Proxy behavior

Pop3Proxy is a module built for parsing messages of the POP3 protocol. It reads and parses COMMANDs on the client side, and sends them to the server if the local security policy permits. Arriving RESPONSEs are parsed as well, and sent to the client if the local security policy permits. It is possible to manipulate both the requests and the responses.

4.8.2.1. Default policy for commands

By default, the proxy accepts all commands recommended in RFC 1939. Additionally, the following optional commands are also accepted: USER, PASS, AUTH. The proxy understands all the commands specified in RFC 1939 and the AUTH command. These additional commands can be enabled manually.

4.8.2.2. Configuring policies for POP3 commands

Changing the default behavior of commands can be done using the hash named `request`. The hash is indexed by the command name (e.g.: USER or AUTH). See *Section 2.1, Policies for requests and responses (p. 4)* for details.

Action	Description
POP3_REQ_ACCEPT	Accept the request without any modification.
POP3_REQ_ACCEPT_MLINE	Accept multiline requests without modification. Use it only if unknown commands has to be enabled (i.e. commands not specified in RFC 1939 or RFC 1734).
POP3_REQ_REJECT	Reject the request. The second parameter contains a string that is sent back to the client.
POP3_REQ_POLICY	Call the function specified to make a decision about the event. See *Section 2.1, Policies for requests and responses (p. 4)* for details. This action uses two additional tuple items, which must be callable Python

Action	Description
	functions. The first function receives two parameters: self and command. The second one is called with an answer, (if the answer is multiline, it is called with every line) and receives two parameters: self and response_param.
POP3_REQ_ABORT	Reject the request and terminate the connection.

Table 4.17. Action codes for POP3 requests

Example 4.15. Example for allowing only APOP authentication in POP3
This sample proxy class rejects the USER authentication requests, but allows APOP requests.

```
class APop3(Pop3Proxy):
      def config(self):
              Pop3Proxy.config(self)
              self.request["USER"] = (POP3_REQ_REJECT)
              self.request["APOP"] = (POP3_REQ_ACCEPT)
```

Example 4.16. Example for converting simple USER/PASS authentication to APOP in POP3
The above example simply rejected USER/PASS authentication, this one converts USER/PASS authentication to APOP authentication messages.

```
class UToAPop3(Pop3Proxy):
      def config(self):
              Pop3Proxy.config(self)
              self.request["USER"] = (POP3_REQ_POLICY,self.DropUSER)
              self.request["PASS"] = (POP3_REQ_POLICY,self.UToA)

      def DropUSER(self,command):
              self.response_value = "+OK"
              self.response_param = "User ok Send Password"
              return POP3_REQ_REJECT

      def UToA(self,command):
              # Username is stored in self->username,
              # password in self->request_param,
              # and the server timestamp in self->timestamp,
              # consequently the digest can be calculated.
              # NOTE: This is only an example, calcdigest must be
              # implemented separately
              digest = calcdigest(self->timestamp+self->request_param)
              self->request_command = "APOP"
              self->request_param = name + " " + digest
              return POP3_REQ_ACCEPT
```

4.8.2.3. Rewriting the banner

As in many other protocols, POP3 also starts with a server banner. This banner contains the protocol version the server uses, the possible protocol extensions that it supports and, in many situations, the vendor and exact version number of the POP3 server.

This information is useful only if the clients connecting to the POP3 server can be trusted, as it might make bug hunting somewhat easier. On the other hand, this information is also useful for attackers when targeting this service.

To prevent this, the banner can be replaced with a neutral one. Use the *`request`* hash with the 'GREETING' keyword as shown in the following example.

Example 4.17. Rewriting the banner in POP3

```
class NeutralPop3(Pop3Proxy):
      def config(self):
      Pop3Proxy.config(self)
      self.request["GREETING"] = (POP3_REQ_POLICY, None, self.rewriteBanner)

      def rewriteBanner(self, response)
              self.response_param = "Pop3 server ready"
              return POP3_RSP_ACCEPT
```

Note
Some protocol extensions (most notably APOP) use random characters in the greeting message as salt in the authentication process, so changing the banner when APOP is used effectively prevents APOP from working properly.

4.8.2.4. Stacking

The available stacking modes for this proxy module are listed in the following table. For additional information on stacking, see *Section 2.3.1, Proxy stacking (p. 7)*.

Action	Description
POP3_STK_POLICY	Call the function specified to decide which part (if any) of the traffic should be passed to the stacked proxy.
POP3_STK_NONE	No additional proxy is stacked into the POP3 proxy.
POP3_STK_MIME	The data part of the traffic including the MIME headers is passed to the specified stacked proxy.
POP3_STK_DATA	Only the data part of the traffic is passed to the specified stacked proxy.

Table 4.18. Action codes for proxy stacking

4.8.2.5. Rejecting viruses and spam

When filtering messages for viruses or spam, the content vectoring modules reject infected and spam e-mails. In such cases the POP3 proxy notifies the client about the rejected message in a special e-mail.

To reject e-mail messages using the `ERR` protocol element, set the `reject_by_mail` attribute to `FALSE`. However, this is not recommended, because several client applications handle `ERR` responses incorrectly.

Note
Infected e-mails are put into the quarantine and deleted from the server.

4.8.3. Related standards

- Post Office Protocol Version 3 is described in RFC 1939.
- The POP3 AUTHentication command is described in RFC 1734.

4.8.4. Classes in the Pop3 module

Class	Description
AbstractPop3Proxy	Class encapsulating the abstract POP3 proxy.
Pop3Proxy	Default POP3 proxy based on AbstractPop3Proxy.
Pop3STLSProxy	POP3 proxy based on Pop3Proxy allowing Start TLS.

Table 4.19. Classes of the Pop3 module

4.8.5. Class AbstractPop3Proxy

This class implements an abstract POP3 proxy - it serves as a starting point for customized proxy classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractPop3Proxy, or a predefined Pop3Proxy proxy class. AbstractPop3Proxy denies all requests by default.

4.8.5.1. Attributes of AbstractPop3Proxy

max_authline_count (integer, rw:r)
Default: 4
Maximum number of lines that can be sent during the authentication conversation. The default value is enough for password authentication, but might have to be increased for other types of authentication.

max_password_length (integer, rw:r)
Default: 16
Maximum allowed length of passwords.

max_request_line_length (integer, rw:r)
Default: 90
Maximum allowed line length for client requests, without the CR-LF line ending characters.

max_response_line_length (integer, rw:r)
Default: 512
Maximum allowed line length for server responses, without the CR-LF line ending characters.

max_username_length (integer, rw:r)
Default: 8
Maximum allowed length of usernames.

password (string, n/a:r)
Default:
Password sent to the server (if any).

permit_longline (boolean, rw:r)
Default: FALSE
In multiline answer (especially in downloaded messages) sometimes very long lines can appear. Enabling this option allows the unlimited long lines in multiline answers.

permit_unknown_command (boolean, rw:r)
Default: FALSE
Enable unknown commands.

reject_by_mail (boolean, rw:r)
Default: TRUE
If the stacked proxy or content vectoring module rejects an e-mail message, reply with a special e-mail message instead of an *ERR* response. See *Section 4.8.2.5, Rejecting viruses and spam (p. 88)* for details.

request (complex, rw:rw)
Default:
Normative policy hash for POP3 requests indexed by the command name (e.g.: "USER", "UIDL", etc.). See also *Section 4.8.2.2, Configuring policies for POP3 commands (p. 86)*.

request_command (string, n/a:rw)
Default: n/a
When a command is passed to the policy level, its value can be changed to this value.

request_param (string, n/a:rw)
Default: n/a
When a command is passed to the policy level, the value of its parameters can be changed to this value.

response_multiline (boolean, n/a:rw)
Default: n/a
Enable multiline responses.

response_param (string, n/a:rw)
Default: n/a
When a command or response is passed to the policy level, the value its parameters can be changed to this value. (It has effect only if the return value is not POP3_*_ACCEPT).

response_stack (complex, rw:rw)
Default:
Hash containing the stacking policy for multiline POP3 responses. The hash is indexed by the POP3 response. See also *Section 4.8.2.4, Stacking (p. 88)*.

response_value (string, n/a:rw)
Default: n/a
When a command or response is passed to the policy level, its value can be changed to this value. (It has effect only if the return value is not POP3_*_ACCEPT).

session_timestamp (string, n/a:r)
Default: n/a
If the POP3 server implements the APOP command, with the greeting message it sends a timestamp, which is stored in this parameter.

timeout (integer, rw:r)
Default: 600000
Timeout in milliseconds. If no packet arrives within this interval, connection is dropped.

username (string, n/a:r)
Default: n/a
Username as specified by the client.

4.8.6. Class Pop3Proxy

Pop3Proxy is the default POP3 proxy based on AbstractPop3Proxy, allowing the most commonly used requests.

The following requests are permitted: APOP; DELE; LIST; LAST; NOOP; PASS; QUIT; RETR; RSET; STAT; TOP; UIDL; USER; GREETING. All other requests (including CAPA) are rejected.

4.8.7. Class Pop3STLSProxy

Pop3STLSProxy is based on Pop3Proxy, allowing the most commonly used requests.

The following requests are permitted: APOP; DELE; LIST; LAST; NOOP; PASS; QUIT; RETR; RSET; STAT; TOP; UIDL; USER; CAPA; STLS; GREETING. All other requests are rejected. The self.max_request_line_length is set to 253.

4.9. Module Smtp

Simple Mail Transport Protocol (SMTP) is a protocol for transferring electronic mail messages from Mail User Agents (MUAs) to Mail Transfer Agents (MTAs). It is also used for exchanging mails between MTAs.

4.9.1. The SMTP protocol

The main goal of SMTP is to reliably transfer mail objects from the client to the server. A mail transaction involves exchanging the sender and recipient information and the mail body itself.

4.9.1.1. Protocol elements

SMTP is a traditional command based Internet protocol; the client issues command verbs with one or more arguments, and the server responds with a 3 digit status code and additional information. The response can span one or multiple lines, the continuation is indicated by an '-' character between the status code and text.

The communication itself is stateful, the client first specifies the sender via the "MAIL" command, then the recipients using multiple "RCPT" commands. Finally it sends the mail body using the "DATA" command. After a transaction finishes the client either closes the connection using the "QUIT" command, or starts a new transaction with another "MAIL" command.

Example 4.18. SMTP protocol sample

```
220 mail.example.com ESMTP Postfix (Debian/GNU)
EHLO client.host.name
250-mail.example.com
250-PIPELINING
250-SIZE 50000000
250-VRFY
250-ETRN
250-XVERP
250 8BITMIME
MAIL From: <sender@sender.com>
250 Sender ok
RCPT To: <account@recipient.com>
250 Recipient ok
RCPT To: <account2@recipient.com>
250 Recipient ok
DATA
```

```
354 Send mail body
From: sender@sender.com
To: account@receiver.com
Subject: sample mail

This is a sample mail message. Lines beginning with
..are escaped, another '.' character is perpended which
is removed when the mail is stored by the client.
.
250 Ok: queued as BF47618170
QUIT
221 Farewell
```

4.9.1.2. Extensions

Originally SMTP had a very limited set of commands (HELO, MAIL, RCPT, DATA, RSET, QUIT, NOOP) but as of RFC 1869, an extension mechanism was introduced. The initial HELO command was replaced by an EHLO command and the response to an EHLO command contains all the extensions the server supports. These extensions are identified by an IANA assigned name.

Extensions are used for example to implement inband authentication (AUTH), explicit message size limitation (SIZE) and explicit queue run initiation (ETRN). Each extension might add new command verbs, but might also add new arguments to various SMTP commands. The SMTP proxy has built in support for the most important SMTP extensions, further extensions can be added through customization.

4.9.1.3. Bulk transfer

The mail object is transferred as a series of lines, terminated by the character sequence "CRLF '.' CRLF". When the '.' character occurs as the first character of a line, an escaping '.' character is prepended to the line which is automatically removed by the peer.

4.9.2. Proxy behavior

The Smtp module implements the SMTP protocol as specified in RFC 2821. The proxy supports the basic SMTP protocol plus five extensions, namely: PIPELINING, SIZE, ETRN, 8BITMIME, and STARTTLS. All other ESMTP extensions are filtered by dropping the associated token from the EHLO response. If no connection can be established to the server, the request is rejected with an error message. In this case the proxy tries to connect the next mail exchange server.

4.9.2.1. Default policy for commands

The abstract SMTP proxy rejects all commands and responses by default. Less restrictive proxies are available as derived classes (e.g.: SmtpProxy), or can be customized as required.

4.9.2.2. Configuring policies for SMTP commands and responses

Changing the default behavior of commands can be done by using the hash attribute *`request`*. These hashes are indexed by the command name (e.g.: MAIL or DATA). Policies for responses can be configured using the *`response`* attribute, which is indexed by the command name and the response code. The possible actions are shown in the tables below. See *Section 2.1*, *Policies for requests and responses (p. 4)* for details. When looking up entries of the *`response`* attribute hash, the lookup precedence described in *Section 2.1.2, Response codes (p. 6)* is used.

Action	Description
SMTP_REQ_ACCEPT	Accept the request without any modification.
SMTP_REQ_REJECT	Reject the request. The second parameter contains an SMTP status code, the third one an associated parameter which will be sent back to the client.
SMTP_REQ_ABORT	Reject the request and terminate the connection.

Table 4.20. Action codes for SMTP requests

Action	Description
SMTP_RSP_ACCEPT	Accept the response without any modification.
SMTP_RSP_REJECT	Reject the response. The second parameter contains an SMTP status code, the third one an associated parameter which will be sent back to the client.
SMTP_RSP_ABORT	Reject the response and terminate the connection.

Table 4.21. Action codes for SMTP responses

SMTP extensions can be controlled using the *`extension`* hash, which is indexed by the extension name. The supported extensions (SMTP_EXT_PIPELINING; SMTP_EXT_SIZE; SMTP_EXT_ETRN; SMTP_EXT_8BITMIME) can be accepted or dropped (SMTP_EXT_ACCEPT or SMTP_EXT_DROP) individually or all at once using the SMTP_EXT_ALL index value.

4.9.2.3. Stacking

The available stacking modes for this proxy module are listed in the following table. For additional information on stacking, see *Section 2.3.1, Proxy stacking (p. 7)*.

Action	Description
SMTP_STK_NONE	No additional proxy is stacked into the SMTP proxy.
SMTP_STK_MIME	The data part including header information of the traffic is passed to the specified stacked proxy.

Table 4.22. Stacking options for SMTP

4.9.3. Related standards

- Simple Mail Transfer Protocol is described in RFC 2821.
- SMTP Service Extensions are described in the obsoleted RFC 1869.
- The STARTTLS extension is described in RFC 3207.

4.9.4. Classes in the Smtp module

Class	Description
AbstractSmtpProxy	Class encapsulating the abstract SMTP proxy.
SmtpProxy	Default SMTP proxy based on AbstractSmtpProxy.

Table 4.23. Classes of the Smtp module

4.9.5. Class AbstractSmtpProxy

This class implements an abstract SMTP proxy - it serves as a starting point for customized proxy classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractSmtpProxy, or one of the predefined proxy classes.

The following requests are permitted: HELO; MAIL; RCPT; DATA; RSET; QUIT; NOOP; EHLO; AUTH; ETRN. The following extensions are permitted: PIPELINING; SIZE; ETRN; 8BITMIME; STARTTLS.

4.9.5.1. Attributes of AbstractSmtpProxy

active_extensions (integer, n/a:r)
Default: n/a
Active extension bitmask, contains bits defined by the constants 'SMTP_EXT_*'

add_received_header (boolean, rw:rw)
Default: FALSE
Add a Received: header into the email messages transferred by the proxy.

append_domain (string, rw:rw)
Default:
Domain to append to email addresses which do not specify domain name. An address is rejected if it does not contain a domain and append_domain is empty.

autodetect_domain_from (enum, rw:rw)
Default:
If you want to autodetect the domain name of the firewall and write it to the Received line, then set this. This attribute either set the method how the mailname should be detected. Only takes effect if add_received_header is TRUE.

domain_name (string, rw:rw)
Default:

domain_name (string, rw:rw)
If you want to set a fix domain name into the added Receive line, set this. Only takes effect if add_received_header is TRUE.

extensions (complex, rw:rw)
Default:
Normative policy hash for ESMTP extension policy, indexed by the extension verb (e.g. ETRN). It contains an action tuple with the SMTP_EXT_* values as possible actions.

interval_transfer_noop (integer, rw:rw)
Default: 600000
The interval between two NOOP commands sent to the server while waiting for the results of stacked proxies.

max_auth_request_length (integer, rw:r)
Default: 256
Maximum allowed length of a request during SASL style authentication.

max_request_length (integer, rw:r)
Default: 256
Maximum allowed line length of client requests.

max_response_length (integer, rw:r)
Default: 512
Maximum allowed line length of a server response.

permit_long_responses (boolean, rw:r)
Default: FALSE
Permit overly long responses, as some MTAs include variable parts in responses which might get very long. If enabled, responses longer than *`max_response_length`* are segmented into separate messages. If disabled, such responses are rejected.

permit_omission_of_angle_brackets (boolean, rw:r)
Default: FALSE
Permit MAIL From and RCPT To parameters without the normally required angle brackets around them. They will be added when the message leaves the proxy anyway.

permit_unknown_command (boolean, rw:r)
Default: FALSE
Enable unknown commands.

request (complex, rw:rw)
Default:
Normative policy hash for SMTP requests indexed by the command name (e.g.: "USER", "UIDL", etc.). See also *Section 4.9.2.2, Configuring policies for SMTP commands and responses (p. 93)*.

request_command (string, n/a:rw)
Default: n/a
When a command is passed to the policy level, its value can be changed to this value.

request_param (string, n/a:rw)
Default: n/a
When a command is passed to the policy level, the value of its parameter can be changed to this value.

request_stack (complex, rw:rw)
Default:
Attribute containing the stacking policy for SMTP commands. See *Section 4.9.2.3, Stacking (p. 94)*.

require_crlf (boolean, rw:r)
Default: TRUE
Specifies whether the proxy should enforce valid CRLF line terminations.

resolve_host (boolean, rw:rw)
Default: FALSE
Resolve the client host from the IP address and add it to the Received line. Only takes effect if add_received_header is TRUE.

response (complex, rw:rw)
Default:
Normative policy hash for SMTP responses indexed by the command name and the response code. See also *Section 4.9.2.2, Configuring policies for SMTP commands and responses (p. 93)*.

response_param (string, n/a:rw)
Default: n/a
When a response is passed to the policy level, the value of its parameter can be changed to this value. (It has effect only when the return value is not SMTP_*_ACCEPT.)

response_value (string, n/a:rw)
Default: n/a
When a response is passed to the policy level, its value can be changed to this value. (It has effect only when the return value is not SMTP_*_ACCEPT.)

timeout (integer, rw:r)
Default: 600000
Timeout in milliseconds. If no packet arrives within this in interval, the connection is dropped.

tls_passthrough (boolean, rw:r)
Default: FALSE
Change to passthrough mode after a successful STARTTLS request. The encrypted traffic is not processed or changed in any way, it is transported intact between the client and server.

unconnected_response_code (integer, rw:rw)
Default: 451
Error code sent to the client if connecting to the server fails.

4.9.6. Class SmtpProxy

SmtpProxy implements a basic SMTP Proxy based on AbstractSmtpProxy, with relay checking and sender/recipient check restrictions. (Exclamation marks and percent signs are not allowed in the e-mail addresses.)

4.9.6.1. Attributes of SmtpProxy

error_soft (boolean, rw:rw)
Default: FALSE
Return a soft error condition when recipient filter does not match. If enabled, the proxy will try to re-validate the recipient and send the mail again. This option is useful when the server used for the recipient matching is down.

permit_exclamation_mark (boolean, rw:rw)
Default: FALSE

permit_exclamation_mark (boolean, rw:rw)
Allow the '!' sign in the local part of e-mail addresses.

permit_percent_hack (boolean, rw:rw)
Default: FALSE
Allow the '%' sign in the local part of e-mail addresses.

recipient_matcher (class, rw:rw)
Default:
Matcher class (e.g.: SmtpInvalidRecipientMatcher) used to check and filter recipient e-mail addresses.

relay_check (boolean, rw:rw)
Default: TRUE
Enable/disable relay checking.

relay_domains (complex, rw:r)
Default:
Domains mails are accepted for. Use Postfix style lists. (E.g.: '.example.com' allows every subdomain of example.com, but not example.com. To match example.com use 'example.com'.)

relay_domains_matcher (class, rw:r)
Default:
Domains mails are accepted for based on a matcher (e.g.: RegexpFileMatcher).

relay_zones (complex, rw:r)
Default:
Zones that are relayed. The administrative hierarchy of the zone is also used.

sender_matcher (class, rw:rw)
Default:
Matcher class (e.g.: SmtpInvalidRecipientMatcher) used to check and filter sender e-mail addresses.

4.10. Module Telnet

The Telnet module defines the classes constituting the proxy for the TELNET protocol.

4.10.1. The Telnet protocol

The Telnet protocol was designed to remotely login to computers via the network. Although its main purpose is to access a remote standard terminal, it can be used for many other functions as well.

The protocol follows a simple scenario. The client opens a TCP connection to the server at the port 23. The server authenticates the client and opens a terminal. At the end of the session the server closes the connection. All data is sent in plain text format whithout any encryption.

4.10.1.1. The network virtual terminal

The communication is based on the network virtual terminal (NVT). Its goal is to map a character terminal so neither the "server" nor "user" hosts need to keep information about the characteristics of each other's terminals and terminal handling conventions. NVT uses 7 bit code ASCII characters as the display device. An end of line is transmitted as a CRLF (carriage return followed by a line feed). NVT ASCII is used by many other protocols as well.

NVT defines three mandatory control codes which must be understood by the participants: NULL, CR (Carriage Return), which moves the printer to the left margin of the current line and LF (Line Feed), which moves the printer to the next line keeping the current horizontal position.

NVT also contains some optional commands which are useful. These are the following:

- *BELL* is an audible or visual sign.
- *BS* (Back Space) moves the printer back one position and deletes a character.
- *HT* (Horizontal Tab) moves the printer to the next horizontal tabular stop.
- *VT* Vertical Tab moves the printer to the next vertical tabular stop.
- *FF* (Form Feed) moves the printer to the top of the next page.

4.10.1.2. Protocol elements

The protocol uses several commands that control the method and various details of the interaction between the client and the server. These commands can be either mandatory commands or extensions. During the session initialization the client and the server negotiates the connection parameters with these commands. Sub-negotiation is a process during the protocol which is for exchanging extra parameters of a command (e.g.: sending the window size). The commands of the protocol are:

Request/Response	Description
SE	End of sub-negotiation parameters.
NOP	No operation.
DM	Data mark - Indicates the position of Sync event within the data stream.
BRK	Break - Indicates that a break or attention key was hit.
IP	Suspend, interrupt or abort the process.

Request/Response	Description
AO	Abort output - Run a command without sending the output back to the client.
AYT	Are you there - Request a visible evidence that the AYT command has been received.
EC	Erase character - Delete the character last received from the stream.
EL	Erase line - Erase a line without a CRLF.
GA	Go Ahead - Instruct the other machine to start the transmission.
SB	Sub-negotiation starts here.
WILL	Will (option code) - Indicates the desire to begin performing the indicated option, or confirms that it is being performed.
WONT	Will not (option code) - Indicates the refusal to perform, or continue performing, the indicated option.
DO	Do (option code) - Indicates the request that the other party perform, or confirmation that the other party is expected to perform, the indicated option.
DONT	Do not (option code) - Indicates the request that the other party stop performing the indicated option, or confirmation that its performing is no longer expected.
IAC	Interpret as command.

Table 4.24. Telnet protocol commands

4.10.2. Proxy behavior

TelnetProxy is a module built for parsing TELNET protocol commands and the negotiation process. It reads and parses COMMANDs on the client side, and sends them to the server if the local security policy permits. Arriving RESPONSEs are parsed as well and sent to the client if the local security policy permits. It is possible to manipulate options by using TELNET_OPT_POLICY. It is also possible to accept or deny certain options and suboptions.

The Telnet shell itself cannot be controlled, thus the commands issued by the users cannot be monitored or modified.

4.10.2.1. Default policy

The low level abstract Telnet proxy denies every option and suboption negotiation sequences by default. The different options can be enabled either manually in a derived proxy class, or the predefined TelnetProxy class can be used.

4.10.2.2. Configuring policies for the TELNET protocol

The Telnet proxy can enable/disable the use of the options and their suboptions within the session. Changing the default policy can be done using the `option` multi-dimensional hash, indexed by the option and the suboption (optional). If the suboption is specified, the lookup precedence described in *Section 2.1.2, Response codes (p. 6)* is used. The possible action codes are listed in the table below.

Action	Description
TELNET_OPT_ACCEPT	Allow the option.
TELNET_OPT_DROP	Reject the option.
TELNET_OPT_ABORT	Reject the option and terminate the Telnet session.
TELNET_OPT_POLICY	Call the function specified to make a decision about the event. The function receives two parameters: self, and option (an integer). See *Section 2.1, Policies for requests and responses (p. 4)* for details.

Table 4.25. Action codes for Telnet options

Example 4.19. Example for disabling the Telnet X Display Location option

```
class MyTelnetProxy(TelnetProxy):
      def config(self):
          TelnetProxy.config(self)
          self.option[TELNET_X_DISPLAY_LOCATION] = (TELNET_OPT_REJECT)
```

Constants have been defined for the easier use of TELNET options and suboptions. These are listed in *Table A.1, TELNET options and suboptions (p. 305)*.

Policy callback functions

Policy callback functions can be used to make decisions based on the content of the suboption negotiation sequence. For example, the suboption negotiation sequences of the Telnet Environment option transfer environment variables. The low level proxy implementation parses these variables, and passes their name and value to the callback function one-by-one. These values can also be manipulated during transfer, by changing the *`current_var_name`* and *`current_var_value`* attributes of the proxy class.

Example 4.20. Rewriting the DISPLAY environment variable

```
class MyRewritingTelnetProxy(TelnetProxy):
      def config(self):
          TelnetProxy.config()
          self.option[TELNET_ENVIRONMENT, TELNET_SB_IS] = (TELNET_OPTION_POLICY, self.rewriteVar)

      def rewriteVar(self, option, name, value):
          if name == "DISPLAY":
                  self.current_var_value = "rewritten_value:0"
          return TELNET_OPTION_ACCEPT
```

Option negotiation

In the Telnet protocol, options and the actual commands are represented on one byte. In order to be able to use a command in a session, the option (and its suboptions if there are any) corresponding to the command has to be negotiated between the client and the server. Usually the command and the option is represented by the same value, e.g.: the *`TELNET_STATUS`* command and option are both represented by the value "5". However, this is not always the case. The *`negotiation`* hash is indexed by the code of the command, and contains the code of the option to be negotiated for the given command (or the *`TELNET_NEG_NONE`* when no negotation is needed).

Currently the only command where the code of the command differs from the related option is *`self.negotiation["239"] = int(TELNET_EOR)`*.

4.10.3. Related standards

The Telnet protocol is described in RFC 854. The different options of the protocol are described in various other RFCs, listed in *Table A.1, TELNET options and suboptions (p. 305)*.

4.10.4. Classes in the Telnet module

Class	Description
AbstractTelnetProxy	Class encapsulating the abstract Telnet proxy.
TelnetProxy	Default Telnet proxy based on AbstractTelnetProxy.
TelnetProxyStrict	Telnet proxy based on AbstractTelnetProxy, allowing only the minimal command set.

Table 4.26. Classes of the Telnet module

4.10.5. Class AbstractTelnetProxy

This class implements the Telnet protocol (as described in RFC 854) and its most common extensions. Although not all possible options are checked by the low level proxy, it is possible to filter any option and suboption negotiation sequences using policy callbacks. AbstractTelnetProxy serves as a starting point for customized proxy classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractTelnetProxy, or one of the predefined TelnetProxy proxy classes. AbstractTelnetProxy denies all options by default.

4.10.5.1. Attributes of AbstractTelnetProxy

current_var_name (string, n/a:rw)
Default: n/a
Name of the variable being negotiated.

current_var_value (string, n/a:rw)
Default: n/a

current_var_value (string, n/a:rw)
Value of the variable being negotiated (e.g.: value of an environment variable, an X display location value, etc.).

enable_audit (boolean, w:r)
Default: FALSE
Enable session auditing.

negotiation (complex, rw:rw)
Default:
Normative hash listing which options must be negotiated for a given command. See *Section Option negotiation (p. 103)* for details.

option (complex, rw:rw)
Default: n/a
Normative policy hash for Telnet options indexed by the option and (optionally) the suboption. See also *Section 4.10.2.2, Configuring policies for the TELNET protocol (p. 102)*.

timeout (integer, rw:r)
Default: 600000
I/O timeout in milliseconds.

4.10.6. Class TelnetProxy

TelnetProxy is a proxy class based on AbstractTelnetProxy, allowing the use of all Telnet options.

4.10.7. Class TelnetProxyStrict

TelnetProxyStrict is a proxy class based on AbstractTelnetProxy, allowing the use of the options minimally required for a useful Telnet session.

The following options are permitted: ECHO; SUPPRESS_GO_AHEAD; TERMINAL_TYPE; NAWS; EOR; TERMINAL_SPEED; X_DISPLAY_LOCATION; ENVIRONMENT. All other options are rejected.

4.11. Module Imap

Internet Message Access Protocol (IMAP) is a protocol to access electronic mailboxes via a reliable TCP connection between the client and the server.

4.11.1. The IMAP protocol

IMAP is a standard IETF protocol to access mail folders stored on a remote mail server. Unlike POP3 which gives only limited access to a single INBOX, IMAP permits manipulation of a remote mail store in a way that is functionally equivalent to local mailboxes.

Unlike many common IETF protocols, IMAP is not a one-request/one-response protocol. The client might issue one or more actions to be performed in parallel, thus responses to those commands can arrive in an order independent from the order they were issued. Requests and the appropriate responses are paired by a unique request identifier called 'tag'. There is one exception to this rule: the server might return untagged responses, when more than a single response is associated with a single command. In this case the server responds with one or more untagged responses and at the end a tagged response to indicate the end of the processing.

4.11.1.1. Protocol elements

The syntax of the IMAP protocol is strictly defined, both the client and the server is either reading a complete line or a sequence of octets prefixed with the length of the sequence.

Request lines start with the tag, followed by a command verb identifying the operation. Each command might have one or more arguments separated by spaces. Each argument has an associated type, one of: ATOM, LITERAL, STRING, LIST. The type further specifies the syntax how these arguments are represented.

A response from the server might be sent directly in response to a request, or unilaterally whenever the server implementation feels it appropriate. The response includes a response verb with zero or more arguments. Note that there might be more response verbs returned for a single command and the response verbs have no direct relationship with the request verb.

Content (e.g.: mail bodies) are transferred as literals embedded in commands and responses. There is no separate bulk transfer mode in the protocol like in POP3 or SMTP. This results in extremely large request/response sizes.

Each message might have one or more associated message flags like '\Deleted' or '\Seen'.

4.11.1.2. Protocol states

IMAP defines four protocol states. Most commands are valid only in certain states. IMAP has the following states:

- Non-Authenticated State: This state is at the beginning of the protocol flow before the client authenticates him/herself.
- Authenticated State: In this state the client is authenticated and MUST select a mailbox to access before commands that affect messages are be permitted.
- Selected State: In this state, a mailbox is selected for access. The protocol enters this state when a mailbox has been successfully selected.
- Logout State: In this state the connection is being terminated and the server will close the connection.

IMAP is similar to other protocols in the sense that a connection is authenticated once, at the beginning of the communication. Before authentication is performed only a limited set of commands can be issued, for example AUTHENTICATE and LOGIN.

Each IMAP operation requires a current mailbox which is similar to the current working directory on UNIX systems. Without a selected mailbox, only a limited set of commands can be issued, for example SELECT, CREATE or REMOVE.

Once a mailbox is selected using the SELECT command, further operations become available, like FETCH or STORE.

Example 4.21. IMAP protocol sample

```
* OK newmail IMAP server ready
A001 CAPABILITY
* CAPABILITY IMAP4 IMAP4rev1 ACL QUOTA LITERAL+\
        MAILBOX-REFERRALS NAMESPACE UIDPLUS ID\
        NO_ATOMIC_RENAME UNSELECT CHILDREN\
        MULTIAPPEND SORT THREAD=ORDEREDSUBJECT\
        THREAD=REFERENCES IDLE STARTTLS LISTEXT\
        LIST-SUBSCRIBED ANNOTATEMORE
A001 OK Completed
A002 LOGIN user password
A002 OK User logged in
A003 SELECT INBOX
* FLAGS (\Answered \Flagged \Draft \Deleted \Seen)
* OK [PERMANENTFLAGS (\Answered \Flagged \Draft\
        \Deleted \Seen \*)]
* 1094 EXISTS
* 3 RECENT
* OK [UNSEEN 1092]
* OK [UIDVALIDITY 1047554575]
* OK [UIDNEXT 36885]
A003 OK [READ-WRITE] Completed
A004 FETCH 1 RFC822
* 1 FETCH (RFC822 {12}
123456789012
)
A004 OK Completed
A005 LOGOUT
* BYE LOGOUT received
A005 OK Completed
```

Responses to IMAP requests come in two types: tagged and untagged. When a client issues a request, the server responds with a single tagged response, which may be preceeded by a number of untagged response lines. In the example above, the client issues a tagged A001 CAPABILITY command to ask the server for the supported capabilities. The server replies with the untagged * CAPABILITY IMAP4 ... line, listing the capabilities, and the tagged A001 OK Completed line, indicating that the request was successfully completed.

4.11.2. Proxy behavior

ImapProxy is a module built for parsing requests and responses of the IMAP protocol. It reads all the REQUESTs at the client side, parses them and - if the local security policy permits - sends them to the server one-by-one. When the RESPONSEs arrive they are parsed by the proxy and sent to the client one by one if the local security policy permits it. Simple greeting rewriting is supported to hide the version of the server. ImapProxy also implements the NAMESPACE, RLIST and RLSUB commands and the LOGIN authentication method. Other authentication methods are not supported and are denied (the proxy does not send them to the policy level).

4.11.2.1. Configuring policies for IMAP requests and responses

Changing the default behaviour of requests is possible using the *`request`* attribute. This hash is indexed by the IMAP command name.

The *`response`* attribute is indexed as follows: The *`response`* attribute hash is a three-dimensional hash, indexed by the command name for which the response is sent; the type of the response (TAGGED or UNTAGGED); and the response name. Untagged responses are accepted when there is a command in the pending queue (i.e. no tagged response arrived to it yet). The following constants are defined for the response types:

Name	Value
IMAP_TAG_UNTAGGED	Untagged responses.
IMAP_TAG_ALL	Both types of responses.
IMAP_TAG_TAGGED	Tagged responses.

Table 4.27. Constants for IMAP response types

The proxy looks up the hash value corresponding to the IMAP command name as the key. If the hash contains no entry for a command, the "*" entry is used. If there is no "*" entry in the hash, the command is denied.

The possible actions are described in the following tables.

Action	Description
IMAP_REQ_ACCEPT	Allow the command to pass.
IMAP_REQ_REJECT	Reject the command and send an error message to the client.
IMAP_REQ_DROP	Silently drop the command - reject the command without sending an error message.
IMAP_REQ_ABORT	Terminate the connection.
IMAP_REQ_POLICY	Call the function specified in the argument to make a decision about the event. See *Section 4.11.2.1, Configuring policies for IMAP requests and responses (p. 107)* for details.
IMAP_REQ_REWRITE	Replace the request with a predefined one. See the example below.
IMAP_REQ_RESPOND	Respond to the request instead of the server. The request is not sent to the server. This action requires two arguments: a string containing a tagged response

Action	Description
	for the request, and a string list containing the optional untagged responses.

Table 4.28. Action codes for IMAP requests

Action	Description
IMAP_RSP_ACCEPT	Allow the response to pass.
IMAP_RSP_REJECT	Reject the response and send an error message to the client.
IMAP_RSP_DROP	Silently drop the response.
IMAP_RSP_ABORT	Terminate the connection.
IMAP_RSP_POLICY	Call the function specified to make a decision about the event. See *Section 4.11.2.1, Configuring policies for IMAP requests and responses (p. 107)* for details.
IMAP_RSP_REWRITE	Replace the response containing the greeting string with a predefined one. See the example below.

Table 4.29. Action codes for IMAP responses

4.11.2.2. Calling methods

For calling a method, the hash must contain a tuple containing two values. The first value is IMAP_REQ_POLICY and the second is the function to call. The function must return with one of the IMAP_REQ_* values (excluding IMAP_*_POLICY), displayed in the table above.

The function is called with three arguments (apart from 'self'): the command tag, the command name, and its arguments. The representation of arguments used by IMAP is described in *Section 4.11.2.4, The IMAP command structure in policies (p. 110)*.

If the proxy is to answer instead of the server, the action tuples must contain the following three items: The value IMAP_REQ_RESPOND; the STRING to be sent back followed by a command tag, and a LIST containing untagged lines to be sent back to the client.

For example, to reply to every CAPABILITY request on behalf of the server:

Example 4.22. Rewriting IMAP capability response

```
self.request["CAPABILITY"] = (IMAP_REQ_RESPOND, "OK CAPABILITY completed", ("[IMAP4rev1]", ))
```

There are other methods to control which CAPABILITYs are known by the client. There is a separate capability hash for this, indexed by the name of the capabilities. The valid values are listed below.

Action	Description
IMAP_CAP_ACCEPT	Allow use of the capability.

Action	Description
IMAP_CAP_DROP	Reject the capability.

Table 4.30. Action codes for IMAP capabilities

This *hash* has nothing to do with capabilities known by the proxy; it defines which answers can arrive to the client for a CAPABILITY command.

Modifying the IMAP greeting string

The IMAP greeting string can be modified (rewritten) by the proxy to hide sensitive information about the server. This can be realized as a rule defined as a tuple containing the following three items:

- The value IMAP_REQ_REWRITE;
- a default return value (e.g.: IMAP_REQ_ACCEPT);
- and a string.

Example 4.23. Changing the greeting string in IMAP

```
def config(self):
        ...
        self.response["GREETING", "UNTAGGED", "OK"] = /
        (IMAP_REQ_REWRITE, IMAP_REQ_ACCEPT, "Welcome to IMAP proxy")
        ...
```

IMAP states

In IMAP there are some defined states, and some commands are allowed only in certain states. On the policy level these states may be examined and modified if necessary. This can be accomplished by setting two attributes, *imap_state_old* and *imap_state_new*. The possible values for these variables are listed in the following table.

Name	Value
IMAP_IS_INITIAL	Before any command arrived.
IMAP_IS_NONAUTH	Before authentication.
IMAP_IS_AUTHENTICATING	Authentication is in progress.
IMAP_IS_AUTH	Authentication performed.
IMAP_IS_SELECTED	A mailbox is selected.
IMAP_IS_QUIT	Logged out.

Table 4.31. IMAP states

4.11.2.3. Configuring acceptable flags

In the IMAP protocol the user can assign flags to mails (or other objects). For example, a flag is assigned to a message to indicate that it has been read (\Seen), it can be marked as important mail or it can be indicated that it has already been answered (\Answered). The usable flags are not predefined in the protocol, IMAP clients can assign any flags they desire.

Flags can be controlled similarly to requests and responses using the *`flag`* hash. It is a normative hash indexed by the name of the flag (case sensitive). The common practice is to accept any flags by default and explicitly drop unneeded flags. The possible actions related to flags are shown in the table below.

Action	Description
IMAP_FLAG_ACCEPT	Accept the flag.
IMAP_FLAG_REJECT	Reject the flag, including the entire command or response.
IMAP_FLAG_DROP	Drop the flag silently, but accept the rest of the command. If the command contains only the flag that is dropped, the entire command is dropped.

Table 4.32. Action codes for IMAP flags

4.11.2.4. The IMAP command structure in policies

When using functions in policies to evaluate IMAP commands, the commands are represented as a recursive tuple of tuples having the following structure. Every command is a tuple of length 3, containing the tag of the command, the name of the command and a tuple containing the arguments.

The following values are possible as arguments (IMAP command structure in the policy layer):

- (int, string) -- Integer
- (int, int) -- Range
- <LITERAL> -- Literal Literals (the actual messages) in the requests/responses are represented by a string having the 'Literal' value. The reason for this is that literals can be very large, therefore they are not sent to (thus not available) the policy level.
- string -- A string or an atom
- (",", a1, a2...) -- Comma-separated list
- ("[", a1, a2...) -- Bracketed list
- ("(", a1, a2...) -- Parenthesized list

Of course, lists can contain other lists recursively.

When processing IMAP responses where a number argument precedes the response name (e.g.: 1094 EXISTS), the number counts as the first argument.

Below are some examples how the different argument types are used in the IMAP protocol.

Example 4.24. IMAP arguments in use

```
Issued command: a0001 FETCH 1,2 RFC822
The command as processed by the IMAP proxy:
tag: "a0001"
command: "FETCH"
arguments: ((',', (1, '1'), (2, '2')), 'RFC822');
```

```
where (',', (1, '1'), (2, '2') is a comma separated list,
(1, '1') is an integer, and RFC822 is a string.
```

```
Issued command: a0002 FETCH 1:2 RFC822
The command as processed by the IMAP proxy:
tag: a0002
command: FETCH
arguments: ((1, 2), 'RFC822');
where (1, 2) is a range.
```

```
Received response: * 1 FETCH (RFC822 <literal>)
The command as processed by the IMAP proxy:
command: FETCH
arguments: ((1, '1'), ('(', 'RFC822', '<LITERAL>'));
where <LITERAL> is a literal represented by a string.
```

4.11.2.5. Stacking

IMAP supports stacking proxies into different levels of the IMAP communication. Stacking is controlled by the `stack` attribute hash. See also *Section 2.3.1, Proxy stacking (p. 7)*. There are three stacking modes available, described in the table below.

Name	Value
IMAP_BODY_FULL	Pass the complete IMAP messages to the stacked proxy or program.
IMAP_BODY_PART	Pass only the body part of IMAP messages to the stacked proxy or program.
IMAP_BODY_TEXT	Pass only the text part of IMAP messages to the stacked proxy or program.

Table 4.33. Body part selection for stacking

4.11.3. Related standards

- Internet Message Access Protocol (v4rev1) is described in RFC 3501.
- IMAP4 Binary Content Extension is described in RFC 3516.
- The IMAP UNSELECT command is described in RFC 3691.
- The IMAP MULTIAPPEND Extension is described in RFC 3502.
- The IMAP4 ID Extension is described in RFC 2971.
- IMAP4 Namespace is described in RFC 2342.
- IMAP4 Login Referrals are described in RFC 2221.
- IMAP4 Mailbox Referrals are described in RFC 2193.
- The IMAP4 QUOTA Extension is described in RFC 2087.
- The IMAP4 ACL Extension is described in RFC 2086.
- IMAP/POP AUTHorize Extension for Simple Challenge/Response is described in RFC 2195.

4.11.4. Classes in the Imap module

Class	Description
AbstractImapProxy	Class encapsulating the abstract IMAP proxy.
ImapProxy	Default IMAP proxy based on AbstractImapProxy.
ImapProxyStrict	IMAP proxy based on AbstractImapProxy, allowing only the minimal command set.

Table 4.34. Classes of the Imap module

4.11.5. Class AbstractImapProxy

This class implements an abstract IMAP proxy - it serves as a starting point for customized proxy classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractImapProxy, or one of the predefined proxy classes, such as *ImapProxy* or *ImapProxyStrict*. AbstractImapProxy denies every command, response, etc. by default.

4.11.5.1. Attributes of AbstractImapProxy

capability (complex, rw:rw)
Default:
Normative hash defining the capabilities accepted by the proxy. See *Section 4.11.2.2, Calling methods (p. 108)*.

flag (complex, rw:rw)
Default:
Normative hash controlling flag values accepted by the proxy. See *Section 4.11.2.3, Configuring acceptable flags (p. 109)*.

imap_state_new (enum, n/a:rw)
Default: n/a
Protocol state after processing the line, one of the IMAP_IS_* constants. See *Section 4.11.2.2, Calling methods (p. 108)*.

imap_state_old (enum, n/a:rw)
Default: n/a
Protocol state before the command arrived, one of the IMAP_IS_* constants. See *Section 4.11.2.2, Calling methods (p. 108)*.

max_line_length (integer, rw:rw)
Default: 2048

max_line_length (integer, rw:rw)
Maximum allowed line length.

max_literal_count (integer, rw:rw)
Default: 32
Maximum number of literals allowed in one command or answer.

max_literal_length (integer, rw:rw)
Default: 65536
Maximum allowed literal length (e.g.: e-mail bodies are sent as literals).

max_password_length (integer, rw:rw)
Default: 32
Maximum allowed length of passwords.

max_pending_count (integer, rw:rw)
Default: 4
Maximum number of pending IMAP commands.

max_respond_lines (integer, rw:rw)
Default: 2
Maximum number of untagged lines that may be sent back to the client from policy.

max_username_length (integer, rw:rw)
Default: 32
Maximum allowed length of usernames.

password (string, n/a:r)
Default: n/a
Password sent to the remote server.

permit_alternative_login_challenges (boolean, rw:r)
Default: FALSE
Permit the use of "Username:" and "Password:" challenge strings durring LOGIN Authentication SASL mechanism, described in section 2.2 of draft-murchison-sasl-login.

request (complex, rw:rw)
Default:
Normative policy hash for IMAP requests indexed by command name. See also *Section 2.1, Policies for requests and responses (p. 4)*.

response (complex, rw:rw)
Default:
Normative policy hash for IMAP responses, indexed by command name, response type and response name. See *Section 4.11.2.1, Configuring policies for IMAP requests and responses (p. 107)*.

stack (complex, rw:rw)
Default:
Attribute containing the stacking policy for IMAP messages. See *Section 4.11.2.5, Stacking (p. 111)* for details.

timeout (integer, rw:rw)
Default: 600000
Timeout value in milliseconds.

username (string, n/a:r)
Default: n/a
Username sent to the remote server.

4.11.6. Class ImapProxy

ImapProxy is the default proxy for the IMAP protocol, based on AbstractImapProxy.

All requests, responses and flags are permitted, as well as the following capabilities: IMAP4; IMAP4rev1; ACL; QUOTA; NAMESPACE; X-NON-HIERARCHICAL-RENAME; NO_ATOMIC_RENAME; UNSELECT; MAILBOX-REFERRALS; LOGIN-REFERRALS; AUTH=LOGIN; ID; CHILDREN; MULTIAPPEND; SORT; THREAD=ORDEREDSUBJECT; THREAD=REFERENCES; LISTEXT; LIST-SUBSCRIBED; ANNOTATEMORE.

4.11.7. Class ImapProxyStrict

IMAP proxy based on AbstractImapProxy, allowing only the minimal command set.

All flags are accepted. The following commands are permitted: AUTHENTICATE; CAPABILITY; CHECK; CLOSE; EXAMINE; FETCH; FIND; GETACL; LIST; LOGIN; LOGOUT; LSUB; NAMESPACE; NOOP; RLIST; RLSUB; SELECT; STATUS; UID; EXPUNGE; STORE.

The permitted capabilities are the following IMAP4; IMAP4rev1; ACL; QUOTA; NAMESPACE; X-NON-HIERARCHICAL-RENAME; NO_ATOMIC_RENAME; UNSELECT; MAILBOX-REFERRALS; LOGIN-REFERRALS; AUTH=LOGIN.

4.12. Module Ldap

The Ldap module defines the classes constituting the proxy for the LDAP protocol.

4.12.1. The LDAP protocol

Lightweight Directory Access Protocol (LDAP) is designed to provide access to X.500 directory services (i.e. to maintain directory databases). It is frequently used to distribute public key certificates, address book information, and user authentication information. Clients can be controlled by individuals (via an application, called LDAP browser) or an agent (e.g.: authentication module or any other application).

X.500 represents information in a hierarchical directory structure. Every entry in the tree is identified with a unique distinguished name (DN) and contains several attributes. A DN looks like the following:

```
uid=username,ou=administrators,ou=some-department,ou=some-part-of-the-company,dc=company,dc=net
```

A schema defines sets of attribute entries in an ObjectClass. Every container can have different ObjectClasses, with each ObjectClass having mandatory and optional entries. The following example defines a user with several attributes from five ObjectClasses.

Example 4.25. Example Ldap entry

```
dn: uid=username,ou=departnent,dc=company,dc=hu
uid: username
cn: username
sn: username
uidNumber: 1234
gidNumber: 1234
mail: username@company.hu
displayName: Dr. UserName
homeDirectory: /home/username
objectClass: top
objectClass: posixAccount
objectClass: inetOrgPerson
objectClass: inetLocalMailRecipient
objectClass: sambaSamAccount
sambaSID: 1234
loginShell: /bin/bash
userPassword: {SMD5}fdsfhiz234dsadsad
telephoneNumber: 1234
street: Foo
postOfficeBox: 1234
roomNumber: 107
```

4.12.1.1. Protocol elements

LDAP is a request/response based binary protocol. The client can connect to the server on a channel at TCP/389 port and send REQUESTs. The client can request several operations in parallel. The following operations can be performed:

- Bind: Identify the client and optionally perform authentication.

- Unbind: Terminate a protocol session.
- Search: Search for entries using filters.
- Modify: Modify tree entries and attributes.
- Add: Request the addition of an entry into the directory.
- Delete: Request the deletion of an entry from the directory.
- Modify DN: Change the leftmost component of the name of an entry in the directory, or to move a subtree of entries to a new location in the directory.
- Compare: Compare an assertion provided with an entry in the directory.
- Abandon: Request the server to cancel an outstanding operation.
- Extended: This operation is for additional operations to be defined for services not available elsewhere in the protocol.

The protocol operates according to the following general scheme:

1. The client opens a connection at TCP/389 and binds to an object in the directory tree. The server authenticates the client to this object. If authentication is not required, the client can use the given tree anonymously.
2. If the authentication process is successful the client can perform requests (i.e. the above mentioned operations: modify, add, delete etc.).
3. Finally the client unbinds and closes the connection.

The LDAP protocol is described using ASN.1 (Abstract Syntax Notation), and is typically transferred using the Basic Encoding Rules, a subset of ASN.1.

4.12.2. Proxy behavior

LdapProxy is a module built for parsing the LDAP protocol version v2 and v3. It reads and parses the REQUESTs at the client side and - if the local security policy permits - sends them to the server. It parses the arriving RESPONSE and - if the local security policy permits - forwards it to the client. LdapProxy can parse the following requests and responses, consequently, these requests can be accepted or denied:

Request/Response	**Description**
BindRequest	Request for binding as an object.
BindResponse	Response to BindRequests.
UnbindRequest	Request for unbinding.
SearchRequest	Request for submitting an LDAP query.
SearchResultEntry	Response to SearchRequests.
SearchResultDone	Response indicating the SearchRequest was performed.
ModifyRequest	Request to modify an entry.
ModifyResponse	Response to ModifyRequests.
AddRequest	Request to add a new entry.

Request/Response	Description
AddResponse	Response to AddRequests.
DelRequest	Request to delete an LDAP entry.
DelResponse	Response to DelRequests.
ModifyDNRequest	Request to modify a DN object.
ModifyDNResponse	Response to ModifyDNRequests.
CompareRequest	Request to compare the provided assertion with an entry in the directory.
CompareResponse	Response to CompareRequests.
AbandonRequest	Request to cancel a request.
SearchResultReference	Response referring to another LDAP server.
ExtendedRequest	Request reserved for further queries.
ExtendedResponse	Response to ExtendedRequests.

Table 4.35. Parsed LDAP operations

4.12.3. Configuring policies for LDAP requests

Changing the default behavior of requests can be done using the hash attribute `request`. The hash is indexed by the request name. The possible values of these hashes are shown in the tables below. See *Section 2.1, Policies for requests and responses (p. 4)* for details.

Action	Description
LDAP_REQ_ACCEPT	Allow the request to pass.
LDAP_REQ_REJECT	Reject the request.
LDAP_REQ_ABORT	Terminate the connection.

Table 4.36. Action codes for LP requests

Example 4.26. Example of the commands usage
In the following example the Ldap proxy allows only BindRequest, UnbindRequest, SearchRequest and CompareRequest requests.

```
def config(self):
      AbstractLdapProxy.config(self)
      self.request["BindRequest"]     = LDAP_REQ_ACCEPT
      self.request["UnbindRequest"]   = LDAP_REQ_ACCEPT
      self.request["SearchRequest"]   = LDAP_REQ_ACCEPT
      self.request["CompareRequest"]  = LDAP_REQ_ACCEPT
      self.request["*"]               = LDAP_REQ_REJECT
```

4.12.4. Simple Authentication and Security Layer (SASL) on LDAP messages

Simple Authentication and Security Layer (SASL) is a framework for authentication and data security in Internet protocols. It is also used by Microsoft Active directory. Please note that support for the SASL security layer is a work in progress - currently LDAP protocol analysis is effectively disabled for SASL wrapped requests.

4.12.5. Related standards

- Lightweight Directory Access Protocol (v3) is described in RFC 2251.
- The LDAP URL Format is described in RFC 2255.
- Using Domains in LDAP/X.500 Distinguished Names is described in RFC 2247.
- Lightweight Directory Access Protocol (v3): Technical Specification is in RFC 3377.

4.12.6. Classes in the Ldap module

Class	Description
AbstractLdapProxy	Class encapsulating the abstract Ldap proxy.
LdapProxy	Default Ldap proxy based on AbstractLdapProxy.
LdapProxyRO	Ldap proxy enabling only read-only access.

Table 4.37. Classes of the Ldap module

4.12.7. Class AbstractLdapProxy

This class implements an abstract LDAP proxy - it serves as a starting point for customized proxy classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractLdapProxy, or one of the predefined proxy classes. AbstractLdapProxy denies all requests by default.

4.12.7.1. Attributes of AbstractLdapProxy

max_message_size (integer, rw:r)
Default: 65535
Maximum allowed size of requests and responses.

max_pending_request (integer, rw:r)
Default: 32
Maximum number of pending requests.

max_search_response_number (integer, w:r)
Default: 2147483648

max_search_response_number (integer, w:r)
Determines the maximal number of LDAP search results. The action to perform on results over this limit can be set in the *response_overrun_action* attribute.

permit_sasl_transport (boolean, rw:r)
Default: FALSE
Permit the use of the Simple Authentication and Security Layer (SASL) on LDAP messages. See *Section 4.12.4, Simple Authentication and Security Layer (SASL) on LDAP messages (p. 118)* for details.

request (complex, rw:r)
Default: n/a
Normative policy hash for LDAP requests indexed by the request. See also *Section 4.12.3, Configuring policies for LDAP requests (p. 117)*.

response_overrun_action (enum, w:r)
Default: LDAP_RSP_DROP
The action to perform on search results over the limit set in the *max_search_response_number* attribute.

timeout (integer, rw:r)
Default: 600000
I/O timeout in milliseconds.

4.12.8. Class LdapProxy

LdapProxy is a default proxy for the LDAP protocol based on AbstractLdapProxy. All syntactically correct operation is permitted.

4.12.9. Class LdapProxyRO

LDAP proxy based on AbstractLdapProxy, with read-only access. This proxy does not allow clients to write or delete on the Ldap server, i.e. the Add, Modify, Delete operations are disabled.

4.13. Module Mime

This module defines the classes representing the MIME proxy.

4.13.1. The MIME protocol

Multipurpose Internet Mail Extensions (MIME) is a complex representation of multiple type of message bodies, and refers to an official Internet standard that defines how messages must be formatted. It makes possible for different types of e-mail systems to exchange messages successfully. MIME is a flexible format which allows to include different type of messages in a single e-mail message. It redefines message format to allow:

- text messages in different character sets;
- extensible set of non-text format messages;
- multiple message types in one message body;
- text header information.

The content of the message is shown by the MIME header, which indicates the type and number of parts the message contains. The header also contains encoding system and version information. MIME supports the following body-types:

Body-type	Description
text	The primary type of MIME content. The main subtype is plain.
multipart	The message contains different types of data.
message	Indicates an encapsulated text message.
image	Indicates that the message contains image file.
audio	Indicates that the message contains audio data.
video	Indicates that the message contains video data.

Table 4.38. MIME body-types

To make sure message contents arrive without corruption, non-text messages must be encoded to printable ASCII characters. Older UNIX systems use uuencode/uudecode transformation. MIME encoding provides base64 to encode any attachment as text.

MIME indicates the parameters of the message in the header field, which can be the following:

MIME header	Description
MIME-Version	Indicates the exact version of the MIME message.
Content-Type	Indicates the type of the data contained in the body. The default content type is 'text/plain; charset=us-ascii'.
Content-Transfer-Encoding	Indicates the encoding used in the message part. It is also possible to create private transfer encoding, which can be indicated by X-My-Private-Transfer-Encoding.
Content-ID	Unique identifier of the MIME object.
Content-Description	Extra comments added to the message by the user.

MIME header	Description
Additional MIME Header Fields	Extra fields to be used by the developers in the future.

Table 4.39. MIME headers

Note
MIME headers do not guarantee that the message really contains the type of content indicated in the header.

Example 4.27. Example mail header containing MIME message
A simple e-mail message containing text message.

```
From: Sender User <sender@balasys.hu>
To: Receiver User <receiver@balasys.com>
Message-Id: <asdfghjkl@balasys.internal.server>
Content-Type: text/plain
Mime-Version: 1.0
Date: Thu, 01 Jul 2004 11:34:30 +0200
Content-Transfer-Encoding: 7bit
```

Example 4.28. Example PNG format picture attachment
A message containing an image attachment in base64 encoding.

```
Mime-Version: 1.0
Content-Type: image/jpeg; name="image.png"
Content-Transfer-Encoding: base64
Content-Disposition: inline; filename="image.png"
```

Example 4.29. Example multipart message
A multipart type message containing a simple text message with a postscript attachment.

```
This is a multi-part message in MIME format.
--------------080709090505030904090905
Content-Type: text/plain; charset=us-ascii; format=flowed
Content-Transfer-Encoding: 7bit

us-ascii message comes here...

--------------080709090505030904090905
Content-Type: application/postscript; name="pns-reference-guide-3.0.ps"
Content-Transfer-Encoding: base64
Content-Disposition: inline; filename="pns-reference-guide-3.0.ps"

base64 message comes here...
```

4.13.2. Proxy behavior

MimeProxy is a module built for parsing MIME messages. Since MIME is not a communication protocol in itself, the MIME proxy cannot be used on its own. It can only inspect data received from a protocol proxy (e.g.: a HTTP proxy, POP3 proxy, etc. that stacks the MimeProxy). MimeProxy reads the data received from the other proxy and handles message headers and bodies if there are any. If the message conforms to the RFC standard it is accepted, otherwise the content is rejected. It is also possible to stack a further proxy into the Mime module (e.g.: a virus filtering module).

4.13.2.1. Configuring policies for MIME headers and content types

Configuring the default behavior for MIME objects is possible using the *header* and *body_type* attributes.

MimeProxy parses MIME headers first. See *Table 4.39, MIME headers (p. 120)* and *Table 4.38, MIME body-types (p. 120)* for the available headers and body-types. The following table shows the possible actions on MIME headers. Headers may be accepted or dropped, or the entire object can be rejected. Subobjects (i.e. MIME objects embedded into other MIME objects) cannot be dropped or rejected individually, the entire object must be rejected/dropped.

Action	Description
MIME_HDR_ACCEPT	Accept header.
MIME_HDR_DROP	Drop the header, but do not reject the entire MIME object.
MIME_HDR_ABORT	Reject the entire connection.
MIME_HDR_POLICY	Call the function specified to make a decision about the header. See *Section 4.13.2.1, Configuring policies for MIME headers and content types (p. 122)* for details. Put header line into policy level.

Table 4.40. Action codes for MIME headers

Second, MimeProxy parses MIME content (or body) types. The following table shows the possible actions on MIME types (*body_type*). Stacking another module is possible using the MIME_TPE_STACK action.

Action	Description
MIME_TPE_ACCEPT	Accept the MIME type.
MIME_TPE_DROP	Drop the entire MIME object.
MIME_TPE_DROP_ONE	Drop the MIME object. This does not affect other objects in the object.
MIME_TPE_CHANGE	Modify the type of the object to the one specified in the second argument.
MIME_TPE_ABORT	Abort the connection and reject the entire MIME object.
MIME_TPE_STACK	Pass the content to be inspected by another proxy.
MIME_TPE_POLICY	Call the function specified to make a decision about the event. See *Section 4.13.2.1, Configuring policies for MIME headers and content types (p. 122)* for details.

Table 4.41. Action codes for MIME content types

If all contents and headers are acceptable by the local security policy, MimeProxy rebuilds the MIME message and passes it back to the parent proxy.

Example 4.30. Example usage of MimeProxy module, denying applications
Removes all applications from the messages. An error message is sent to the client (*`silent_drop = FALSE`*; the directory where the error messages are stored is specified in the *`mime_message_path`* attribute).

```
class MyMimeProxy(MimeProxy):
      def config(self):
              MimeProxy.config(self)
              self.body_type["application" "*"] = (MIME_TPE_DROP)
              self.silent_drop = FALSE
              self.mime_message_path="/usr/share/vela/mime"
```

4.13.3. Related standards

- RFC 2045: MIME Part One: Format of Internet Message Bodies
- RFC 2046: MIME Part Two: Media Types
- RFC 2047: MIME Part Three: Message Header Extensions for Non-ASCII Text
- RFC 2048: MIME Part Four: Registration Procedures
- RFC 2049: MIME Part Five: Conformance Criteria and Example

4.13.4. Classes in the Mime module

Class	Description
AbstractMimeProxy	Class encapsulating the abstract MIME proxy.
MimeProxy	Default MIME proxy based on AbstractMimeProxy.

Table 4.42. Classes of the Mime module

4.13.5. Class AbstractMimeProxy

This class implements an abstract MIME proxy - it serves as a starting point for customized proxy classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractMimeProxy, or the predefined MimeProxy proxy class. AbstractMimeProxy rejects all headers and body-types by default.

4.13.5.1. Attributes of AbstractMimeProxy

append_object (string, rw:r)
Default: ""
Appends the specified file (e.g.: *`/tmp/attachment`*) as a new attachment. Requires the *`permit_empty_headers`* parameter to be set to *`TRUE`*.

body_type (complex, rw:r)
Default:
Multi-dimensional policy hash for body-types, indexed by body-type name (major and minor parts of the body type). See *Section 4.13.2.1, Configuring policies for MIME headers and content types (p. 122)*.

drop_bad_header (boolean, rw:r)
Default: FALSE
Reject the (sub)object or silently drop the header if it is syntactically or semantically incorrect. If the header is essential for MIME parsing, this option is ignored and the message will be dropped.

error (complex, rw:rw)
Default: n/a
An alias of the `error_action` parameter. Obsolete, use `error_action` instead.

error_action (complex, rw:rw)
Default: n/a
With this normative hash you can control the action taken when some error occurs. For compatibility reasons, the `error` parameter refers to the same hash.

header (complex, rw:r)
Default:
Normative policy hash for MIME header types, indexed by the header type. See *Section 4.13.2.1, Configuring policies for MIME headers and content types (p. 122)*.

keep_header_comments (boolean, rw:r)
Default: TRUE
Keep or remove header comments. The syntax of MIME headers is very complex and it is possible to confuse a parser with a specially crafted comment. To prevent this, it is possible to remove all comments. (NOTE: This option will be header-specific in future releases.)

max_header_length (integer, rw:r)
Default: 4096
The maximum length of a single header.

max_header_line_length (integer, rw:r)
Default: 1000
The maximum length of a single header line. A header may be split into multiple lines, this value limits the length of a single line.

max_header_lines (integer, rw:r)
Default: 1024

max_header_lines (integer, rw:r)
Maximum number of headers in a (sub)object. Different objects are counted separately even when these objects are subobjects of the same object. If drop_bad_header turned on all headers above this number will dropped. If not, the conversation will aborted.

max_multipart_level (integer, rw:r)
Default: 10
The maximum recursion level the proxy should check. If the number of levels in an object exceeds the allowed limit, the object is rejected.

max_multipart_number (integer, rw:r)
Default: 100
Maximum number of subobjects that an object is allowed to contain. The default is 100. The counter is not restarted when checking a new subobject. (i.e.: this limits the global number of objects)

mime_message_path (string, rw:r)
Default: "/usr/share/vela/mime"
Path to the directory where the custom error messages are stored.

permit_bad_continuous_line (boolean, rw:r)
Default: FALSE
Parse bad headers as continuous lines.

permit_empty_headers (boolean, rw:r)
Default: FALSE
If enabled (TRUE) and the first line of a MIME object (or subobject) is not parseable as a MIME header, it is handled as a MIME body without a header.

silent_drop (boolean, rw:r)
Default: FALSE
If disabled (FALSE), dropped objects are replaced with an object containing an error message. If enabled (TRUE), objects and headers are dropped without notification.

timeout (integer, rw:r)
Default: -1
I/O timeout in milliseconds. The default value (-1) means unlimited.

4.13.6. Class MimeProxy

MimeProxy is a default MIME proxy based on the AbstractMimeProxy. All headers and body-types are accepted.

4.14. Module Modbus

4.14.1. Classes in the Modbus module

Class	Description
AbstractModbusProxy	Class encapsulating the abstract MODBUS proxy.
ModbusProxy	Default Modbus proxy based on AbstractModbusProxy.

Table 4.43. Classes of the Modbus module

4.14.2. Class AbstractModbusProxy

This class implements an abstract MODBUS proxy - it serves as a starting point for customized proxy classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractModbusProxy, or one of the predefined proxy classes. AbstractModbusProxy denies all requests by default.

4.14.2.1. Attributes of AbstractModbusProxy

allow_user_defined (boolean)
Default: FALSE
Allow user-defined function requests. User-defined functions are not specified by the MODBUS specification, thus will not be intepreted by ModbusProxy, but passed through. Set value to FALSE to reject user-defined functions, unless it is specified in the request policy expicitly.

buffer_size (integer)
Default: 8192
Size of the request/response buffer in bytes. Must be larger than the maximum size of a MODBUS ADU (260 bytes)

default_accept (boolean)
Default: FALSE
Allow all funcion requests by default, which are not specified in the request policy.

request (complex)
Default:

request (complex)
Normative policy hash for MODBUS function requests indexed by the request.

timeout (integer)
Default: 10000
Initial timeout in milliseconds.

transaction_limit (integer)
Default: 64
Maximum number of pending tranasctions.

transaction_timeout (integer)
Default: 60
Transaction timeout in seconds.

4.14.3. Class ModbusProxy

ModbusProxy is a default proxy for the MODBUS protocol based on AbstractModbusProxy. All syntactically correct operation is permitted.

4.15. Module MSRpc

Remote Procedure Call (RPC) is a protocol for calling procedures on remote machines.

4.15.1. The RPC protocol

The RPC protocol consists of two phases: negotiating an access point to a service and communicating with the service itself. On the server side the negotiation is performed by a special service called 'Endpoint Mapper' (EPM), which listens on the TCP/UDP port 135. The protocol of the communication is specified in the DCE RPC Specification. If the client is allowed to use the requested service, the EPM passes its address and IP in its response, and the client may connect to it and make any data transfer it wishes. The protocol format varies from service to service, so with maintained transparent forwarding facilities between the client and the service, only the communication between the client and the EPM is filtered.

The filtering of the traffic between the client and the EPM means that requests can be approved or rejected for services specified by their UUID. The denial of a service is implemented as if the EPM had refused it, the approval is transparent in a way that the resulting service access point has the same IP as in the original EPM request: only the port is altered to point to the dedicated forwarder facility.

The timing parameters of the communication may also be limited by specifying the maximal allowed duration of the requests/responses; the idle timeout between requests/responses and the maximal delay between the service approval and the connection to the approved service.

4.15.2. Proxy behavior

The MSRpc proxy is a module supporting version 2 of the MSRPC protocol.

4.15.2.1. Setting policies for services

Changing the default behavior for services can be accomplished via the *self.interface* hash attribute. This hash is indexed by the service UUID, and each item in this hash is an action code defining proxy behavior for the given command. The available action codes are shown in the following table:

Name	Value
MSRPC_UUID_ACCEPT	Allow access to the requested service.
MSRPC_UUID_REJECT	Reject access to the requested service.
MSRPC_UUID_DROP	Drop the request without further notice.

Table 4.44. Action codes for MSRpc requests.

Example 4.31. Customising RPC to allow connection to service "11223344-5566-7788-99aa-bbccddeeff00"

```
class MyRpcProxy(MSRpcProxy):
        def config(self):
                self.interface["11223344-5566-7788-99aa-bbccddeeff00"] = MSRPC_UUID_ACCEPT
```

4.15.2.2. Restrictions

Currently the proxy handles only TCP connections, and tracks/filters only the traffic toward the EPM service. Since this does not cover the protocols used by either the standardized or the proprietary DCOM services, some applications may not work properly through this proxy. Some remote management applications that use the ISystemActivator service and the notification feature of Exchange are known to have issues with the MSRpc proxy.

4.15.2.3. Global options

The following global options apply to all classes of the the MSRpc proxy:

`config.msrpc.forwarder_data_timeout`	Timeout value (in milliseconds) for forwarded traffic. Default value: *60000* (60 sec)

4.15.3. Classes in the MSRpc module

Class	Description
AbstractMSRpcProxy	Class encapsulating the abstract MSRpc proxy.

Class	Description
MSRpcProxy	Default MSRpc proxy based on AbstractMSRpcProxy.

Table 4.45. Classes of the MSRpc module

4.15.4. Class AbstractMSRpcProxy

This class implements an abstract MSRpc proxy, denying access to all services by default.

4.15.4.1. Attributes of AbstractMSRpcProxy

command_timeout (integer, rw:r)
Default: 600000
Command timeout in milliseconds. If a packet cannot be transmitted during this interval, the connection is dropped.

forwarder_timeout (integer, rw:r)
Default: 20000
Forwarder timeout in milliseconds. If no connection is established to a forwarder facility during this period (measured from service approval), the forwarder will be cancelled.

interface (complex, rw:rw)
Default: empty
Normative policy hash indexed by the UUID of the services, specifying the security policy about the service. See *Section 4.15.2.1, Setting policies for services (p. 128)* for details.

secondary_port_max (integer, rw:r)
Default: 0
The upper limit of the port range allocated for forwarders. (Zero means no restriction.)

secondary_port_min (integer, rw:r)
Default: 0
The lower limit of the port range allocated for forwarders. (Zero means no restriction.)

timeout (integer, rw:r)
Default: 600000
Idle timeout in milliseconds. If no packet arrives during this period, the connection is dropped.

4.15.5. Class MSRpcProxy

This proxy allows access only to the most necessary EPM services for RPC to function. These services are *`99fcfec4-5260-101b-bbcb-00aa0021347a`* and *`8a885d04-1ceb-11c9-9fe8-08002b104860`*.

4.16. Module Radius

The Radius module defines the classes constituting the proxy for the RADIUS protocol.

4.16.1. The RADIUS protocol

Remote Authentication Dial In User Service (RADIUS) is a client-server protocol for user authentication between the Network Access Server (NAS) and the authenticator server. The protocol has three participants:

- The user requesting network access the service (e.g.: PPP, PLIP etc.).
- The access point (modem pools or NAS servers), which delivers the service. The access point acts as the client in the protocol.
- The server which authenticates the user.

The RadiusProxy is installed between the server and client (i.e. the access point).

4.16.1.1. Protocol elements

During the authentication process the participants use the following protocol elements:

- REQUEST: When a new connection attempt arrives to the NAS, it sends a message towards the RADIUS server requesting the authentication of the user; or it sends an accounting related message.
- RESPONSE: The RADIUS server attempts to authenticate the user when an authentication REQUEST is received. The server returns the result of the process to the NAS in a RESPONSE message.
- ATTRIBUTE: Both the REQUEST and RESPONSE packets contain a set of structured attribute-value pairs containing information like username, password or the type of service requested by the user. Attributes are identified by a number ranging from 0 to 255. Each attribute has an associated type specified in the RADIUS RFCs which define the range of valid values.

Note
There are also some vendor-specific RADIUS dictionaries, where certain attributes are used for internal purposes. Obviously, these are not discussed in the RFCs.

4.16.1.2. RADIUS states

The user initiates the authentication process when attempting to use a NAS service. When the user request arrives, the NAS sends an Access-Request message containing the attributes username, MD5-hashed password, the user IP and the port ID. The message is sent to port UDP/1812; if no response is received within a period of time, the request is re-sent a number of times.

If RADIUS is configured to use username/password based authentication, the server consults the database and if all the terms match, the server replies with an Access-Accept message. When the challenge/response method is used the server generates a challenge and sends it to the client in an Access-Challenge message. The client displays it to the user who calculates the response which is resubmitted by the NAS client in another Access-Request message with a new request ID, encrypted User-Password attribute and the State Attribute. If the response is correct the server allows the connection request in an Access-Accept message and the NAS starts to deliver the service. If the authentication process fails the server sends an Access-Reject message and the NAS denies the delivery of the service.

The user and the NAS server (technically the radius client) are authenticated separately. The user is authenticated only after the NAS has been verified via the Radius secret (i.e. password). Users can be authenticated by username/password or challenge/response methods.

Username/password authentication is a traditional authentication method where the user id identifies the user and the password authenticates him/her. During the challenge/response the user ID identifies the user itself and the client is authenticated by a one time password. The server sends an unpredictable number to the client. The user calculates it with a hardware or software tool and sends the result back. If the answer is correct, it validates the client's identity and this is the response which authenticates the user.

The Access-Accept message might deliver additional parameters to the service, such as IP address. These additional parameters are delivered as values of various attributes.

RADIUS can also be used to send Service-Start and Service-End messages for accounting purposes. While the protocol is the same as the one described above, it uses a separate port and a separate set of attributes. When the client is configured to use RADIUS Accounting, it sends an Accounting-Start message describing the type of the service and the user using it. RADIUS accounting uses the port UDP/1813. The RADIUS server returns an acknowledgment. The client repeats sending the request until it receives the acknowledgment. At the end of delivering the service, the client sends an Accounting-Stop message to the server describing the type and optionally the statistics about the connection. The server acknowledges the stop messages as well.

Note
Earlier UDP/1645 was also used by RADIUS servers, and accounting messages were sent using port UDP/1646.

4.16.2. Proxy behavior

RadiusProxy is a module built for parsing the messages of the RADIUS protocol. It reads the REQUESTs at the client side and decrypts the user password with the given shared secret (known by both the client and the server). If the REQUEST and all the ATTRIBUTEs are permitted by the local security policy, it sends the message to the RADIUS server. It parses the arriving RESPONSE and validates the authenticator signature. The authenticator signature is an MD5 hash included in the RADIUS message, generated from various message parameters, including the shared secret. It is used to ensure that the response is genuine and was indeed sent by the server. If the RESPONSE is permitted by the local security policy and is authentic, the message encrypted with the secret is returned to the NAS. It is possible to keep different secrets on the two sides of the proxy (i.e. password translation is possible). RadiusProxy is able to parse both authentication and accounting messages, and it can also manipulate RESPONSEs if the secret is available. If the secret is not available, authenticator signatures cannot be validated, thus it is not possible to verify that the received response was sent to a proper request. Both the client and server side secrets are required for modifying the messages; for validating the authenticator signature, the server side secret is sufficient.

4.16.2.1. Configuring policies for RADIUS commands and responses

Changing the default behavior of commands can be done by using the hash attribute `request`. There is a similar attribute for responses called `response`. These hashes are indexed by the type of the request/response. The possible values of these hashes are shown in the tables below. See *Section 2.1, Policies for requests and responses (p. 4)* for details.

Action	Description
RADIUS_REQ_ACCEPT	Allow the request to pass.
RADIUS_REQ_REJECT	Block the request and report it to the client.
RADIUS_REQ_ABORT	Terminate the connection.
RADIUS_REQ_DROP	Block the request without further action.
RADIUS_REQ_POLICY	Call the function specified to make a decision about the event. See *Section 2.1, Policies for requests and responses (p. 4)* for details.

Table 4.46. Action codes for RADIUS requests

Action	Description
RADIUS_RSP_ACCEPT	Allow the response to pass.
RADIUS_RSP_REJECT	Block the response and report it to the client.
RADIUS_RSP_ABORT	Terminate the connection.
RADIUS_RSP_DROP	Block the response without further action.
RADIUS_RSP_POLICY	Call the function specified to make a decision about the event. See *Section 2.1, Policies for requests and responses (p. 4)* for details.

Table 4.47. Action codes for RADIUS responses

Similar policies can be defined for RADIUS attributes. For easier use, predefined constants are available for the different attributes. The possible actions on the attributes are listed in the following table. The attribute constants are listed in *Table A.2, RADIUS Protocol Attribute types described in RFC 2865. (p. 311)*.

Action	Description
RADIUS_ATR_ACCEPT	Allow the attribute to pass.
RADIUS_ATR_REJECT	Block the attribute and report it to the client.
RADIUS_ATR_ABORT	Terminate the connection.
RADIUS_ATR_DROP	Reject the entire message if it contains the specified attribute.
RADIUS_ATR_POLICY	Call the function specified to make a decision about the event. See *Section 2.1, Policies for requests and responses (p. 4)* for details.

Action	Description
RADIUS_ATR_ZERO	An alias of *RADIUS_ATR_DROP* the action code.
RADIUS_ATR_ACCEPT_MAXONE	The message can contain zero or one of the specified attribute.
RADIUS_ATR_ACCEPT_ONE	Accept exactly one attribute in the message. The message is rejected if it does not contain the specified attribute. This action can be used to check the existance of mandatory attributes.
RADIUS_ATR_DROP_ONE	Drop the attribute from the message; the message itself is not rejected.

Table 4.48. Action codes for RADIUS attributes

4.16.2.2. Binding secondary sessions

The RADIUS protocol does not guarantee the delivery of the messages (since it uses UDP), consequently packages are dropped if the system is overburden. Clients and servers attempt to send messages several times; allowing secondary sessions increases reliability and decreases server load. See *Section 2.2, Secondary sessions (p. 7)* for further information.

Example 4.32. Example RadiusProxy config
The following example defines a RADIUS proxy which serves 1000 parallel requests in one thread. Packet rebuilding is turned on as well, therefore the server and client side secrets are also specified.

```
class MyRadiusProxy(RadiusProxy):
      def config(self):
              RadiusProxy.config(self)
              self.client_secret="secret"
              self.server_secret="secret"
              self.rebuild_packets='TRUE'
              self.secondary_mask = 0xC
              self.secondary_sessions = 1000
```

4.16.3. Related standards

- The RADIUS protocol is defined in RFC 2865.
- The RADIUS Accounting protocol is defined in RFC 2866.

4.16.4. Classes in the Radius module

Class	Description
AbstractRadiusProxy	Class encapsulating the abstract RADIUS proxy.
RadiusProxy	Default RADIUS proxy based on AbstractRadiusProxy.

Class	Description
RadiusProxyStrict	RADIUS proxy based on AbstractRadiusProxy, allowing only a minimal command set.

Table 4.49. Classes of the Radius module

4.16.5. Class AbstractRadiusProxy

This class implements the RADIUS protocol as described by RFC 2865.

4.16.5.1. Attributes of AbstractRadiusProxy

attribute_desc (complex, rw:rw)
Default: n/a
Attribute descriptors, this hash is indexed by the attribute type and the value contains a tuple of (type, min, max). The min and max values are interpreted depending on the RADIUS type. For integers it means the minimum and maximum integer values, for strings it is applied to the string length.

attribute_usage (complex, rw:rw)
Default:
Describes attribute usage, the hash is indexed by the tuple of (packet type, attribute id). The value is a singleton tuple containing one of the RADIUS_ATR values.

client_secret (string, rw:r)
Default:
Secret string (password) shared between the client (probably NAS) and Vela. Setting this value is not mandatory, but some of the proxy functions will not be available (see *Section 4.16.2, Proxy behavior (p. 131)* for details).

max_packet_length (integer, rw:r)
Default: 4096
Maximum allowed length of packets.

permit_trailing_zeroes (boolean, rw:rw)
Default: FALSE
Workaround for a Cisco bug (the router sometimes pads the packets with NUL bytes).

rebuild_packets (boolean, rw:rw)
Default: FALSE

rebuild_packets (boolean, rw:rw)
Specifies whether to rebuild packets (requires both shared secrets to be available, see *Section 4.16.2, Proxy behavior (p. 131)* for details).

request (complex, rw:rw)
Default:
Normative policy hash for RADIUS request types indexed by the type of the request. See also *Section 4.16.2.1, Configuring policies for RADIUS commands and responses (p. 132)*.

response (complex, rw:rw)
Default:
Normative policy hash for RADIUS response types indexed by the type of the response. See also *Section 4.16.2.1, Configuring policies for RADIUS commands and responses (p. 132)*.

secondary_mask (secondary_mask, rw:r)
Default: 0xf
Specifies which connections can be handled by the same proxy instance (the same connection is enabled as secondary session by default). See *Section 2.2, Secondary sessions (p. 7)* for details.

secondary_sessions (integer, rw:r)
Default: 10
Maximum number of allowed secondary sessions within a single proxy instance. See *Section 2.2, Secondary sessions (p. 7)* for details.

server_secret (string, rw:r)
Default:
Secret string (password) shared between the server and Vela. Setting this value is not mandatory, but some of the proxy functions will not be available (see *Section 4.16.2, Proxy behavior (p. 131)* for details).

timeout (integer, rw:r)
Default: 60000
Timeout in milliseconds.

4.16.6. Class RadiusProxy

Default RADIUS proxy based on AbstractRadiusProxy allowing all well-formed RADIUS packets (all requests, responses, and attributes) through the firewall. Secondary sessions are enabled for the same target (`secondary_mask=0xC`) (maximum 10). For a stricter default configuration use the RadiusProxyStrict class.

4.16.7. Class RadiusProxyStrict

Radius proxy strictly checking RFC compliance of the passing packets. Packets containing attributes that are not defined in the RFC are dropped.

The following requests and responses are permitted: radius_access_request; radius_access_challenge; radius_access_reject; radius_access_accept; radius_accounting_request; radius_accounting_response. All other requests and responses are rejected. The policy used for the attributes is listed in the Radius Appendix.

4.17. Module Sip

The Sip module defines the classes constituting the proxy for the Session Initiation Protocol (SIP).

4.17.1. The SIP protocol

SIP is a peer-to-peer protocol providing call processing functions and features similar to public switched telephone networks. The SIP protocol (or protocol family rather) is not a conventional Internet protocol, because it is not based on the traditional client-server model. Although there are prioritized servers for performing certain tasks, in most cases SIP phones function as both clients and servers on the network. Consequently, the protocol does not use the usual request/response based communication, and that has important consequences in perimeter defense.

4.17.1.1. Protocol elements

The devices involved in SIP communication can have several different roles, but a single device can play the part of different roles at the same time. The most important roles are briefly summarized below:

- *User-agent*: The phone itself. In the traditional model, this would be called client.
- *Registrar*: The registration service. The address where a particular user-agent is accessible is registered here. It acts as a sort of a name service for the protocol.
- *Proxy*: This device transmits the requests of the user-agents. It has nothing to do, and is not to be confused with a proxy firewall or with a web cache proxy.
- *Presence server*: Similar to the registrar; this device stores information about the availability of the user-agents. Users can monitor if the VoIP devices of their contacts (friends, business partners, etc.) are active (i.e. on-line) via the presence server.
- *Back2back user-agent*: This is a special proxy implementing the functions of two user-agents. On one side of a connection it acts as the caller, on the other side as the called party.

SIP is only involved in the signaling part of a communication session, and relies on other protocols to perform the actual data transfer. SIP communication takes place in multiple channels: one is the signaling channel, the other one the actual data channel used to transmit the voice and/or video data. This latter channel is opened dynamically according to parameters negotiated in the signaling channel. The negotiation uses a separate - embedded - protocol called Session Description Protocol (SDP) used to describe the channel and the type of media used in a session (i.e. the IP ports, codecs, etc.). It is essential for the firewall to understand and inspect the SDP protocol, since it contains all the information required to allow the VoIP traffic pass the firewall. The SDP traffic also has to be modified in case network address translation is performed. To transfer the actual voice, video, or other data, SIP uses the Real-time Transport Protocol (RTP). RTP defines a standardized packet format for delivering audio and video over the Internet, and is frequently used in audio/video streaming and conferencing solutions.

From the signaling point of view, it is important to note that there is no client/server hierarchy between the user-agents, only caller/called party. The signaling traffic is usually not transmitted directly between the user-agents, generally proxies and back2back user-agents are also involved. Consequently, signaling messages (for example a request and a corresponding answer) can take very different routes between two user-agents, greatly complicating the secure transmission of the protocol. On the other hand, the RTP session is built directly between the user-agents without the interaction of proxies, though back2back user-agents may still be involved in the transmission of the audio/video data. Therefore a special care must be taken when creating the access control rules of the SIP signaling and data traffic.

4.17.1.2. Proxy behavior

The SIP proxy allows SIP signaling (accepting SIP messages on the TCP port 5060) and the dynamic RTP traffic through the firewall without compromising the security of the firewall and the defended network. Ports are dynamically opened through the firewall based on information received in the signaling traffic. The signaling part of the protocol is inspected on the application level for protocol conformance: SIP proxy enforces the standards, protecting the network from attacks violating the protocol. This is especially important since SIP clients and even servers are rarely designed with security in mind and many of them have issues from a security point of view. As an application level gateway, PNS parses, checks, and rebuilds every passing signaling request and response. The actual (audio, video, etc.) communication is not inspected, it is forwarded through PNS on the kernel level using stateful package filtering. These connections are handled as related UDP connections. Furthermore, it is possible to perform NATing and connection marking (see the description of the SIP proxy classes for details).

When packets arrive to the port the SIP proxy is listening on, basic access control is performed based on the source IP address of the packets. Each and every request and response is inspected on the application level (Layer 7 in the OSI model). The requests and responses - including protocol elements like headers - are parsed and strictly checked for conformance with the SIP standards. The SIP proxy understands and enforces the SIP protocol as described in RFC 3261. The syntax and length of the various protocol elements (e.g.: length of lines, headers, requests, etc.) is checked in order to repel various attack forms based on malformed messages, such as buffer overflow attacks. The relation of the arriving packets relative to other packets and previous communication information is also inspected. Packets not conforming to the logic and workflow of the protocol (e.g.: responses without requests, etc.) are rejected. This step is important because SIP uses random ports for transferring the actual communication data (the RTP stream, e.g.: voice, video), and otherwise it would be possible to open covert channels through the firewall between machines, not only the intended VoIP communication between the two SIP endpoints (i.e. the caller and the receiver).

The payload (SDP) part of the communication is parsed as well and modified if network address translation (NAT) is used. In this case, the addresses and dynamic ports used by the RTP traffic stream have to be modified accordingly. After all these sanity checks the policy settings of the firewall are consulted. Address, and media type filtering is performed (e.g.: to allow only voice traffic to/from specific addresses). Network address translation is also performed at this step if required.

Access control on the RTP stream part of the protocol is performed separately. This is important because RTP and signaling streams can have different access control settings. If SIP servers or a SIP proxy is used on some part of the network, the signaling and the RTP streams originate from different sources. (In such situation, the signaling is originating from the proxy, but the RTP stream arrives directly from the actual client. However, such a situation could also be used to initiate covert channels.)

The proxy supports the use of secondary sessions as described in *Section 2.2, Secondary sessions (p. 7)*.

4.17.1.3. Keepalive messages in SIP

Keepalive messages in SIP are not originally part of the RFC. However, many SIP implementations actually use them, sending UDP packets (containing only whitespaces) to maintain the connection. These packets are accepted if they are not longer than a preset value (see the *max_keepalive_size* attribute of the *AbstractSipProxy* proxy class) and interprets them as keepalive messages. Such packets are uniformly replaced with UDP packets containing only a single line-feed.

4.17.1.4. Configuring SIP policies

The SIP proxy is capable of filtering the different media types in the SIP traffic based on their SDP headers using the *media* hash attribute. The possible actions for the different media types are shown in the table below. See *Section 2.1, Policies for requests and responses (p. 4)* for details.

Action	Description
SIP_MEDIA_ACCEPT	Accept the media type.
SIP_MEDIA_DROP	Drop the media from the list of proposed media channels but forward the message to the peer.
SIP_MEDIA_ABORT	Drop the SIP message containing the corresponding media type.
SIP_MEDIA_POLICY	Call the function specified to make a decision about the media type. The function receives two parameters: self, and the media type string. See *Section 2.1, Policies for requests and responses (p. 4)* for details.

Table 4.50. Action codes for SIP media types.

Media types are the strings in SDP headers that identify the type of media sent in the channel (e.g.: *audio*, *video*, *** for all types, etc.). There are no predefined constants for the media types, as they are not defined in any RFCs or other standards. Typically, *audio* and *video* are used for voice and video streams, respectively.

Example 4.33. Disabling video traffic in SIP
This example class accepts only voice traffic, denying video streams and aborting on all other types of media streams.

```
class AudioSip(SipProxy)
      def config(self):
        self.media["audio"]=[SIP_MEDIA_ACCEPT]
        self.media["video"]=[SIP_MEDIA_DROP]
        self.media["*"]=[SIP_MEDIA_ABORT]
```

4.17.2. Related standards

- The Session Initiation Protocol is described in RFC 3261.

- The Session Description Protocol is described in RFC 2327.
- RTP: A Transport Protocol for Real-Time Applications is described in RFC 3550.

4.17.3. Classes in the Sip module

Class	Description
AbstractSipProxy	Class encapsulating the abstract SIP proxy.
SipProxy	Default SIP proxy class based on AbstractSipProxy.

Table 4.51. Classes of the Sip module

4.17.4. Class AbstractSipProxy

This proxy implements the SIP protocol as specified in RFC 3261. Service definitions should refer to a customized class derived from AbstractSipProxy, or a predefined proxy class.

4.17.4.1. Attributes of AbstractSipProxy

max_keepalive_size (integer, w:r)
Default: 128
Maximum size for SIP signaling keepalive messages in bytes. See *Section 4.17.1.3, Keepalive messages in SIP (p. 138)* for details.

max_message_size (integer, w:r)
Default: 65536
Maximum allowed size of a SIP signaling message in bytes.

media_connection_mark (integer, w:rw)
Default: 0
Connection mark value that is set on all on media connections. That way media connections can be easily identified and handled by specific packet filtering rules.

secondary_mask (secondary_mask, rw:r)
Default: 0xf
Specifies which connections can be handled by the same proxy instance. See *Section 2.2, Secondary sessions (p. 7)* for details.

secondary_sessions (integer, rw:r)
Default: 10

secondary_sessions (integer, rw:r)
Maximum number of allowed secondary sessions within a single proxy instance. See *Section 2.2, Secondary sessions (p. 7)* for details.

timeout (integer, w:r)
Default: 600000
I/O timeout in milliseconds.

4.17.5. Class SipProxy

This class encapsulates the default SIP proxy.

4.17.5.1. Attributes of SipProxy

media (complex, rw:r)
Default:
Policy hash implementing media type filtering, indexed by the media type (as a string, e.g.: `audio`). See *Section 4.17.1.4, Configuring SIP policies (p. 138)* for details.

permit_rtp_zones (complex, rw:r)
Default:
A comma-separated list of zone pairs that are permitted to exchange voice or video streams (e.g.: `(("internet", "intranet"),)`). This option replaces the DAC decision (which is unavailable here, since RTP streams are forwarded on the kernel level). NOTE: this is a two-way connection between the zones.

rtp_endpoint_rewrite_nat_policy (unknown, rw:r)
Default:
Reference to an existing NAT policy that rewrites RTP endpoints from internal to external addresses. The policy is called for all messages containing an SDP part, since those may also contain addresses of the endpoints.

4.18. Module Socks

The Socks module defines the classes for the proxy to inspect Socks communication.

4.18.1. The SOCKS protocol

4.18.2. Proxy behaviour

SOCKS is a network protocol for routing packets using a proxy server between the clients and the servers. SOCKS performs at Layer 5 of the OSI model. SOCKS is typically used to proxy other, Layer 7 protocols, most often HTTP.

Example 4.34. SOCKS and HTTP traffic
The following configuration example embeds an HTTP proxy into a Socks proxy and can be used to inspect HTTP traffic that uses a SOCKS proxy to access the servers. Client authentication is disabled.

```
class MySocksProxy(SocksProxy):
    def config(self):
        SocksProxy.config(self)
        self.enable_socks_v4 = TRUE;
        self.require_auth_v5 = FALSE

    def requestStack(self, ip, port):
        return MyHttpProxy
```

4.18.2.1. Authenticating clients

The proxy can authenticate the clients using passwords. GSS-API and other authentication methods supported by the SOCKSv5 protocol are not supported. The process of negotiating the authentication between the client and the Socks proxy is the following:

1. The client sends the list of authentication methods is supports to the SOCKS server.
2. The Socks proxy replies to the client on behalf of the SOCKS server, depending on the configuration of the Socks proxy:
   - If the client selected password-based authentication and the *disable_auth_v5* option is set to *FALSE* and the *require_auth_v5* is set to *TRUE* (which are the defaults), PNS replies that password authentication is supported.
   - If the *require_auth_v5* is set to *FALSE*, and the client supports the *none* authentication method, the connection is accepted without authentication.
   - In other cases, the client receives an authentication error.

The Socks proxy supports inband authentication as well. For details on inband authentication, see *Section 5.1.10, Class InbandAuthentication (p. 176)*.

4.18.3. Related standards

- The SOCKS 5 protocol is defined in *RFC 1928*.

4.18.4. Classes in the Socks module

Class	Description
AbstractSocksProxy	Class encapsulating the Socks Proxy.

Class	Description
SocksProxy	Default Socks proxy class based on AbstractSocksProxy.

Table 4.52. Classes of the Socks module

4.18.5. Class AbstractSocksProxy

This proxy validates SOCKS traffic. It serves as a starting point for customized proxy classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractSocksProxy, or the predefined SocksProxy proxy class.

4.18.5.1. Attributes of AbstractSocksProxy

auth (class)
Default:
The authentication provider object used in the authentication process, set in the *`authentication_policy()`* parameter of the service. See *Section 5.1.1, Authentication and authorization basics (p. 169)* for details.

auth_server (boolean)
Default:
The address of the VAS server used to authenticate the connection. Note that this option cannot be modified by the proxy, it is set in the AuthenticationPolicy used by the Service definition.

connect_server (boolean)
Default: TRUE
Set to *`TRUE`*, if the Socks proxy is connecting directly to the SOCKS server. Set to *`FALSE`*, if the Socks proxy is an embedded proxy and another proxy is performing the actual connection.

disable_auth_v5 (boolean)
Default: FALSE
Disable authentication in the SOCKSv5 protocol. If this option is enabled, the proxy sends only the *`none`* authentication method to the client.

enable_socks_v4 (boolean)
Default: FALSE
Accept SOCKSv4 connections as well. If the client is using an unsupported protocol version, or the client is using SOCKSv4 but the *`enable_socks_v4()`* option is set to FALSE, the *`Unsupported protocol version='4'`* log message is sent to the system logs.

require_auth_v5 (boolean)
Default: TRUE
Require authentication in the SOCKSv5 protocol. If this option is enabled, the proxy sends only the *password* authentication method to the client. Note that using this option requires a properly configured VAS AuthenticationPolicy and an authentication backend in the definition of the service that uses the Socks proxy.

timeout (integer)
Default: 600000
Timeout in milliseconds. The -1 value disables the timeout.

4.18.5.2. AbstractSocksProxy methods

Method	Description
requestForward(self, ip, port)	Called when the SOCKS protocol reaches forward state.

Table 4.53. Method summary

Method requestForward(self, ip, port)

This method must determine whether to stack another proxy class into the traffic, or simply forward the traffic without analyzing. The method can raise an exception which will result in denying any traffic. The default behavior is to forward traffic without analyzing.

Arguments of requestForward

IP (string)
Default: n/a
The IP address of the target host.

port (integer)
Default: n/a
The port number to connect to.

return (complex)
Default: n/a
Tuple of *SOCKS_STK_** and a class. *SOCKS_STK_NONE* will result in simple forwarding, while *SOCKS_STK_DATA* will start a stacked proxy instance of the returned class.

4.18.6. Class SocksProxy

A default proxy for the SOCKS protocol based on AbstractSocksProxy. It serves as a starting point for customized proxy classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractSocksProxy, or the predefined SocksProxy proxy class. By default, the proxy rejects SOCKSv4 connections, and requires authentication from the clients.

4.19. Module SQLNet

4.19.1. The SQL*Net protocol

This class implements parts of Oracle TNS (Transparent Network Substrate) to enable clients to communicate with Oracle servers behind firewalls using port TCP/1521. This module is especially needed when tnslsnr (the TNS listener) is in Multi-threaded Server (MTS) mode.

The SQL*Net proxy does not analyze the whole protocol stream, as the data protocol of Oracle operates on top of TNS.

An example for the SQL*Net connection string is provided in *Example A.1, An example for the SQL*Net connection string (p. 327)*.

4.19.2. Proxy behavior

SQLNetProxy is a module built for parsing messages of the SQL*Net protocol. It reads and parses QUERYs on the client side, and sends them to the server if the local security policy permits.

In MTS mode Oracle returns a redirect packet specifying where the client should connect to. The proxy processes this packet and initiates a new connection to the address specified; all packets sent by the client will be automatically redirected to this new address. This functionality is completely transparent to the clients. To accomplish this, either InbandRouter has to be used, or the overridable option has to be set for DirectedRouter and TransparentRouter.

SQLNet proxy is able to parse *connect_string* and *connection_data* containing the address and port of the target server and information on the database.

When the connection is established the SQLNetProxy inspects TNS headers, but does not inspect the layers above TNS.

4.19.3. Related standards

SQL*Net is a not specified in any public standards.

4.19.4. Classes in the SQLNet module

Class	Description
AbstractSQLNetProxy	Class encapsulating the abstract SQLNet proxy.

Class	Description
SQLNetProxy	Default SQLNet proxy class based on AbstractSQLNetProxy.

Table 4.54. Classes of the SQLNet module

4.19.5. Class AbstractSQLNetProxy

AbstractSQLNetProxy is a default proxy for the SQL*Net protocol - it serves as a starting point for customized proxy classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractSQLNetProxy, or the predefined proxy class.

4.19.5.1. Attributes of AbstractSQLNetProxy

connect_data (string, n/a:rw)
Default: n/a
The TNS connect string as sent by the client, or as modified by the policy.

server_address (string, rw:rw)
Default: "n/a"
Name of the Oracle server to connect to. This value is only used together with InbandRouter, or if the overridable option is set for DirectedRouter or TransparentRouter.

server_port (integer, rw:rw)
Default: "n/a"
Port of the Oracle listener to connect to.

split_connect_threshold (integer, rw:rw)
Default: 231
CONNECT data that is larger than this value will be split into smaller DATA packets.

strict_redirect_parsing (boolean, rw:rw)
Default: TRUE
Disabling this option allows improperly formed packets to pass the firewall.

timeout (integer, rw:r)
Default: 600000
Timeout in milliseconds.

4.19.5.2. AbstractSQLNetProxy methods

Method	Description
connectRequest(self, connect_data)	Function called when the client issues a CONNECT request.

Table 4.55. Method summary

Method connectRequest(self, connect_data)

This function is called when the client issues a CONNECT request, to have a chance to validate and change the CONNECT string sent by the client. The connect string can be found in the parameter *connect_data*. The function has to return a logical *TRUE* or *FALSE* value, i.e. *SQLNET_ACCEPT* or *SQLNET_ABORT*.

Arguments of connectRequest

connect_data (unknown, n/a:n/a)
Default: n/a
The connect string as sent by the client.

4.19.6. Class SQLNetProxy

A transparent SQL*Net proxy based on AbstractSQLNetProxy.

In transparent mode the client addresses directly the server, so the target address is readily available; while in nontransparent mode the client connects directly to Vela, and Vela receives the address of the server within the protocol.

4.19.6.1. Attributes of SQLNetProxy

transparent_mode (boolean, rw:rw)
Default: TRUE
Enable/disable transparent mode operation.

4.20. Module Ssh

4.20.1. The Secure Shell protocol

Secure Shell (SSH) is a protocol designed to remotely access (login and execute commands) on a computer connected to the network. SSH was aimed to replace the earlier unencrypted protocols (e.g.: rlogin, TELNET and rsh), and provides secure encrypted communication between two hosts over an insecure network. Users of SSH can also use it for tunneling, forwarding arbitrary TCP ports and X11 connections over the resultant secure channel; and can transfer files using the embedded SFTP or SCP protocols.

4.20.1.1. Protocol elements

One of the main features of the SSH protocol is that almost the entire communication between the client and the server is encrypted - including the authentication of the user. (Naturally, the negotiation of the encryption method to be used is in plain text). During the initialization of the session server authentication is performed and parameters for encryption, data compression and integrity verification of the data transferred are negotiated. The protocol enforces user authentication and is capable of authenticating the user via various methods: password, RSA key, Challenge/Response schemes like S/Key and OPIE, etc.

The typical uses of SSH include the following:

Remote shell	Remotely administer a computer via an interactive terminal console. This is one of the most widespread uses of SSH.
Remote command execution	Execute commands on the remote machine. Remote command execution can also result in significant data transfer, for example when performing scheduled or manual tasks such as file copying (scp), data or file synchronization (rsync), creating archive backups (tar), etc.
TCP IP forwarding (also known as port forwarding)	It is possible to tunnel any TCP/IP connection from the client or from the server into the encrypted SSH channel. It can also be used to forward communication otherwise not allowed, such as the access of ports banned by the security policy. This allows to secure any - normally unencrypted - data transfer and is frequently used as an easy way to secure connections between the hosts without the need to set up full VPN connections.
File transfer	Securely transfer files using SFTP.
X11 forwarding	Applications running on the server and requiring graphical interface (X Window) appear on the client's monitor, but run on the server in all other respect, thus it is possible to work with them remotely.
Agent forwarding:	Transfer authentication requests to the client machine.

4.20.1.2. Protocol versions

The original version of the protocol (SSH-1, dated 1995) has been revised in 1996, and SSH-2 was created offering improved security and new features. The two versions of the protocol are incompatible with each other. Since SSH-1 has inherent design flaws and is vulnerable to attacks, it is now generally considered obsolete and its use should not be permitted. Practically all server and client applications today support SSH-2, however, software not supporting SSH-2 may still be in use by some organizations, posing a considerable security vulnerability to them.

The SSH proxy supports only the SSHv2 protocol (SECSH).

4.20.2. Proxy behavior

SSH proxy uses man-in-the-middle technique to decrypt and terminate the SSH connections on the firewall. It separates the connections into two parts and inspects all traffic, so that no data can be directly transferred between the server and the client. Only the SSH-2 protocol is supported exclusively, but owing to the widespread use and availability of SSH-2 implementations, this does not mean any hindrance. The general capabilities of SSH proxy are summarized below.

- *Protocol inspection* : All traffic is inspected and only permitted across the firewall if it fully complies to the SSH-2 protocol. This feature provides effective protection against a great number of attacks exploiting vulnerabilities of server and client applications, including buffer overflow vulnerabilities.
- *Verify encryption method* : The internal parameters of the connections can also be controlled, allowing the proxy to enforce the use of selected encryption methods (cipher type, key length, etc.), thus provide protection against downgrade attacks.
- *Control user authentication* : The different authentication methods can be separately enabled or disabled, e.g.: it is possible to enforce the use of strong authentication methods by completely disabling password based authentication. User-level filtering and access control can also be performed. Although this can obviously be done on the servers themselves, PNS as an external device provides these features reliably even if the server or the client machines get compromised.
- *Control of SSH channels* : There is full control over the SSH channels, i.e. it can be specified which channels are allowed to and from a given server or in a given connection. For instance, file transfer, port forwarding, or X forwarding can be separately enabled/disabled based on various criteria.
- *Disable agent forwarding* : Agent forwarding can be disabled, thus prevent that the keys used in the internal network become accessible on external machines.
- *Control remote command execution* : The SSH protocol can be fully inspected, thus it can be specified which commands are allowed, which ones are disabled. More sophisticated decisions can also be made based on the parameters of the session, e.g.: to allow the execution of a command only to certain users, etc.

4.20.2.1. Configuring policies for SSH channels

The opening of SSH channels from the server and the client side is possible using the `server_channel` and `client_channel` hashes. These hashes are indexed by the channel type (e.g.: `session`). The available channel types are listed in the following table.

Name	Value
session	Channels for terminal shells, remote execution requests (e.g.: scp), and SFTP.
direct-tcpip	Channels for client-to-server forwarded connections.
forwarded-tcpip	Channels for server-to-client forwarded connections.
auth-agent	Channels for forwarding authentication agents.
auth-agent@openssh.com	Channels for forwarding authentication agents, as implemented in OpenSSH.

Name	Value
x11	Channels for forwarding graphical interfaces.

Table 4.56. The list of available channel types.

The possible actions are described in the following table. See also *Section 2.1, Policies for requests and responses (p. 4)*.

Action	Description
SSH_CHAN_ACCEPT	Accept the request without any modification.
SSH_CHAN_REJECT	Reject the channel opening request.
SSH_CHAN_POLICY	Call the function specified to make a decision about the channel opening request.
SSH_CHAN_ABORT	Reject the channel opening request and terminate the connection.

Table 4.57. Action codes for SSH channel open requests.

Example 4.35. Enabling and disabling SSH channels

The following proxy class accepts only terminal session (shell) connections, and rejects all other channel types.

```
class ShellonlySshProxy(SshProxy):
        def config(self):
                SshProxy.config(self)
                self.client_channel["session"] = (SSH_CHAN_ACCEPT)
                self.client_channel["session-shell"] = (SSH_CHAN_ACCEPT)
                self.client_request["session-exec"] = (SSH_REQ_REJECT)
                self.client_request["session-subsystem"] = (SSH_REQ_REJECT)
```

4.20.2.2. Configuring policies for SSH requests

Changing the default behavior of requests arriving from the server and the client side is possible using the *server_request* and *client_request* attributes. All requests specified in the RFCs are supported. The index of these hashes is composed of the channel type (e.g.: *session*, see *Section 4.20.2.1, Configuring policies for SSH channels (p. 148)* for a detailed list), a single hyphen, and the request name as defined by the SSH protocol specification. E.g.: *session-x11-req*. The possible actions are described in the following table. See also *Section 2.1, Policies for requests and responses (p. 4)*.

Action	Description
SSH_REQ_ACCEPT	Accept the request without any modification.
SSH_REQ_REJECT	Reject the request.
SSH_REQ_POLICY	Call the function specified to make a decision about the request.

Action	Description
SSH_REQ_ABORT	Reject the request and terminate the connection.

Table 4.58. Action codes for SSH channel and global requests.

For complex decisions that are based on the parameters of the requests, you have to use the *SSH_REQ_POLICY* parameter and create a function within the proxy class that examines and optionally modifies the parameters.

This custom function can receive the following four attributes:

self	
side	The side of the connection relative to PNS: *0* for the client side, *1* for the server side.
index	The name of the request, e.g., *x11*, *subsystem*, etc.
request	A structure that has fields containing the parameters of the request. See *Section 4.20.2.3, Parameters of the SSH requests (p. 150)* for details on the different request parameters.

See the following example.

Example 4.36. Enabling only SFTP connections
The following proxy class accepts SFTP connections. SFTP is a subsystem of SSH, therefore the parameters of the *session-subsystem* request must be examined. (This is for example only, for SFTP only configuration use SshProxySftpOnly predefined class)

```
class SFtponlySshProxy(SshProxy):
        def config(self):
                SshProxy.config(self)
                self.client_channel["session"] = (SSH_CHAN_ACCEPT)
                self.client_request["session-subsystem"] = (SSH_REQ_POLICY, self.permitSFTPOnly)
                self.client_request["session-pty-req"] = (SSH_REQ_REJECT)
                self.client_request["session-shell"] = (SSH_REQ_REJECT)
                self.client_request["session-exec"] = (SSH_REQ_REJECT)
        def permitSFTPOnly(self, side, index, request):
                if request.subsystem == "sftp":
                    return SSH_REQ_ACCEPT
                return SSH_REQ_REJECT
```

4.20.2.3. Parameters of the SSH requests

SSH requests can be controlled using the *server_request* and *client_request* hashes. These hashes are indexed by the channel type (e.g.: *session*). Some requests have additional parameters that are also listed. Some channels (e.g., the X11 channel) require two request messages to open, the first message requests the channel, while the second message actually opens the requested channel. The following requests are available from the client side. For examples on local and remote forwarding, see *Section 4.20.2.4, Configuring local and remote forwarding (p. 153)*.

window-change	
When the window (terminal) size changes on the client side a message may be sent to inform the server of the new window dimensions. Parameters of the request:	
width_cols	Width of the terminal window in characters.
height_rows	Height of the terminal window in characters.
width_px	Width of the terminal window in pixels.

window-change	
height_px	Height of the terminal window in pixels.

pty-req	
Request a pseudo-terminal for the session. Parameters of the request:	
term	Requests a pseudo-terminal.
width_cols	Width of the terminal window in characters.
height_rows	Height of the terminal window in characters.
width_px	Width of the terminal window in pixels.
height_px	Height of the terminal window in pixels.

x11-req	
Request X11 forwarding for the session. Parameters of the request:	
x11_auth_proto	The name of the X11 authentication method used, e.g., *MIT-MAGIC-COOKIE-1*.
x11_auth_cookie	
screen_number	
single_connection	If set to *TRUE*, the server forwards only a single connection.

x11	
Open an X11 channel. Parameters of the request:	
originator_host	IP address of the host.
originator_port	Port number of the host.

auth-agent-req
Request the forwarding of the authentication requests. This request has no additional parameters.

auth-agent-req@openssh.com
Request the forwarding of the authentication requests, as implemented in OpenSSH. This request has no additional parameters.

env	
Pass an environment variable and its value in the message. Parameters of the request:	
name	The name of environment variable.
value	The value of environment variable.

shell
Request a shell be started on the server side. This request has no additional parameters.

exec	
Request the server to start the execution of the command sent in the message. Parameters of the request:	
command	The command to be executed. The command may include a path.

subsystem	
Request the server to execute a predefined subsystem. (Subsystems usually include a general file transfer mechanism, and possibly other features as well.) Parameters of the request:	
subsystem	Name of the subsystem to be executed.

signal	
A signal delivered to the remote process or service. Parameters of the request:	
signal	Name of the signal to be sent.

The following requests are available from the server side. Some requests have additional parameters that are also listed.

exit-status	
When the command running on the server terminates, an *exit-status* message can be sent to return the exit status of the command.	
exit_status	

exit-signal	
A message indicating that the remote command was terminated violently due to a signal. A zero usually means that the command terminated successfully.	
signal_name	Name of the signal. One of: *`ABRT`*, *`ALRM`*, *`FPE`*, *`HUP`*, *`ILL`*, *`INT`*, *`KILL`*, *`PIPE`*, *`QUIT`*, *`SEGV`*, *`TERM`*, *`USR1`*, *`USR2`*, or a custom signal consisting of two strings and the *`@`* character (e.g., *`signal@ example`*).
core_dumped	
error	The text of the error message. The message may consist of multiple lines separated by CRLF (Carriage Return - Line Feed) pairs.
lang	Language tag confirming to RFC3066.

xon-xoff
A message informing the client when it can or cannot perform flow control.

xon-xoff	
client_can_do	*TRUE* if the client can perform flow control.

4.20.2.4. Configuring local and remote forwarding

Remote port-forwarding transfers connections arriving to a port of the server to the client. The client sends a *global-tcpip-forward* request to the server. The parameters of this request tell the server which address and port it should listen on for incoming connections (*bind_address*, *bind_port*). When the server receives a connection to this address/port pair, it opens a *forwarded-tcpip* towards the client. The parameters of these requests are summarized in the following tables.

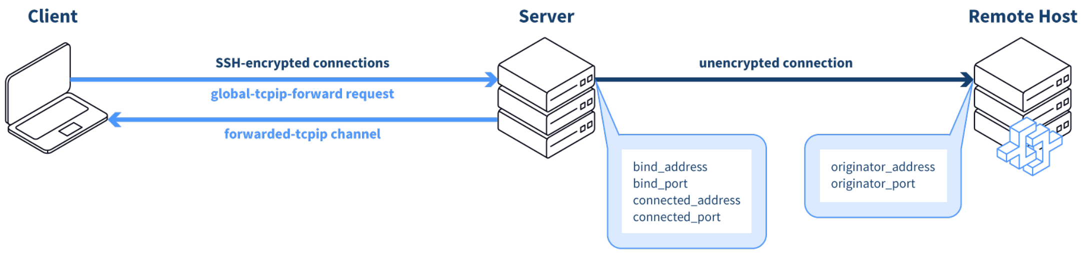


Figure 4.7. Remote TCP forwarding

global-tcpip-forward	
Connections arriving to the specified IP address and port of the server are forwarded to the client.	
bind_address	The server forwards connections received on this address to the client. The following special addresses may be used: ■ The *""* parameter means that connections are to be accepted on all protocol families supported by the SSH implementation. ■ The *0.0.0.0* parameter means to listen on all IPv4 addresses. ■ The *::* parameter means to listen on all IPv6 addresses. ■ The *localhost* parameter means to listen on all protocol families supported by the SSH implementation on loopback addresses only ([RFC3330] and [RFC3513]). ■ The *127.0.0.1* and *::1* parameters indicate listening on the loopback interfaces for IPv4 and IPv6, respectively.
bind_port	The server forwards connections received on this port to the client.

forwarded-tcpip	
Opens a channel used to forward remote connections to the client.	
connected_addr	The IP address of the server that received the connection.
connected_port	The port of the server that received the connection.

forwarded-tcpip	
originator_addr	The IP address of the remote host whose connection is forwarded to the client.
originator_port	The port of the remote host whose connection is forwarded to the client

Local port-forwarding transfers connections arriving to the client from a host to a remote host via the SSH server. For local port-forwarding, the client sends a *`direct-tcpip`* channel opening request to the server. The parameters of this request tell the server which host it should forward the connection, as well as the address of the host that connects to the client (usually localhost). This request has the following parameters.

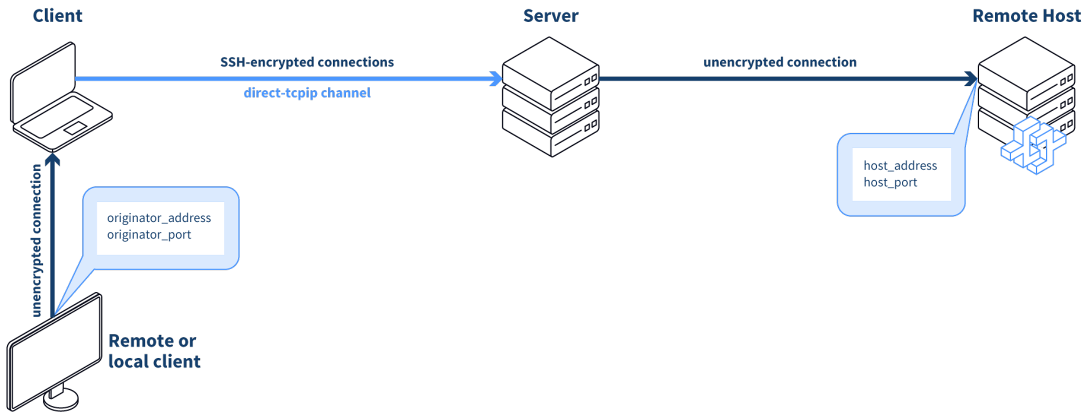


Figure 4.8. Local TCP forwarding

direct-tcpip	
Opens a channel used to forward remote connections to the client.	
originator_addr	The IP address of the host whose connection is forwarded to the remote host.
originator_port	The port of the host whose connection is forwarded to the remote host.
host_addr	The IP address of the remote host that is the destination of the forwarded connection.
host_port	The port of the remote host that is the destination of the forwarded connection.

Example 4.37. Restricting local forwarding

The following proxy class permits local forwading only to port *80* of the *192.168.1.1* remote host. Only shell and local forwarding channels are permitted.

```
class RestrictedlocalforwardSshProxy(SshProxy):
        def config(self):
                SshProxy.config(self)
                self.client_channel["session"] = (SSH_CHAN_ACCEPT)
                self.client_channel["direct-tcpip"] = (SSH_CHAN_ACCEPT)
                self.client_request["direct-tcpip"] = (SSH_REQ_POLICY, self.controllocalforward)
                self.client_request["session-exec"] = (SSH_REQ_REJECT)
                self.client_request["session-subsystem"] = (SSH_REQ_REJECT)
        def controllocalforward(self, side, index, request):
                if request.host_address == "192.168.1.1" and request.host_port == "80":
                    return SSH_REQ_ACCEPT
                return SSH_REQ_REJECT
```

4.20.2.5. Configuring encryption parameters

The SSH proxy is able to enforce policies on the various elements of the encrypted SSH communication, such as the MAC, key-exchange, etc. algorithms that are permitted to be used. The parameters can be set separately for the client and for the server side. The attributes are represented as comma-separated strings listing the enabled methods/algorithms, in the order of preference.

Key exchange algorithms

The permitted key exchange algorithms can be specified via the `client_kex_algos` and `server_kex_algos` attributes. The SSH proxy supports the `diffie-hellman-group16-sha512` and `diffie-hellman-group18-sha512` and `diffie-hellman-group14-sha256` and `diffie-hellman-group14-sha1` and `diffie-hellman-group1-sha1` algorithms.

Host key algorithms

The permitted host key algorithms can be specified via the `client_hostkey_algos` and `server_hostkey_algos` attributes. The supported algorithms are `ssh-rsa, rsa-sha2-256, rsa-sha2-512` and `ssh-dss`.

Note
For a hostkey algorithm to work for the clients the corresponding private key has to be set in the `host_key_rsa` or the `host_key_dss` attribute. The supported algorithms are `ssh-rsa, rsa-sha2-256, rsa-sha2-512` and `ssh-dss`.

Public key algorithms

The permitted public key algorithms can be specified via the `client_pubkey_algos` and `server_pubkey_algos` attributes. The supported algorithms are `ssh-rsa, rsa-sha2-256, rsa-sha2-512` and `ssh-dss`.

Symmetric cipher algorithms

The permitted symmetric cipher algorithms can be specified via the `client_cipher_algos` and `server_cipher_algos` attributes. The following algorithms are supported: `aes128-cbc`, `3des-cbc`, `blowfish-cbc`, `cast128-cbc`, `arcfour`, `aes192-cbc`, `aes256-cbc`, `aes128-ctr`, `aes192-ctr`, `aes256-ctr`, `aes128-gcm@openssh.com`, `aes256-gcm@openssh.com`.

MAC algorithms

The permitted MAC algorithms can be specified via the `client_mac_algos` and `server_mac_algos` attributes. The supported algorithms are: `hmac-sha2-256` and `hmac-sha2-512` and `hmac-sha1` and `hmac-md5`.

4.20.2.6. Host key verification

To successfully build the required SSH connections both towards the client and the server, PNS has to show the appropriate keys to the client (otherwise the client will reject the connection as the key does not match the server it intends to connect). This problem can be easily overcome if PNS is used to protect the servers: the server key has to be deployed on PNS as well. However, this is not possible when protecting clients, because the private keys of all servers that will be contacted is rarely available. In this case, SSH proxy can be configured to automatically verify the identity of the server using the *`server_hostkeys_verify`* attribute. This is similar to certificate verification in SSL connections, but in SSH there is no certificate or other identity information attached to the host keys.

The methods supported for host key verification are shown in the following table.

Name	Value
SSH_HKV_ACCEPT_ANY	Accept any host key.
SSH_HKV_ACCEPT_ONCE	Accept unknown host keys only on the first occassion. The IP address-port pair of unknown host keys is registered, later on that key is used to verify connections from that address.
SSH_HKV_ACCEPT_KNOWN	Accept only known host keys. Public keys can be configured for each IP address or port pair (like in case of the `known_hosts` file). For any unknown IP address-port pair the connection is terminated.

Table 4.59. SSH host key verification mode.

4.20.2.7. Auditing SSH channels

The SSH proxy supports the general auditing framework of PNS. The SSH proxy can even be configured to audit only certain types of channels, it is not necessary to fully audit all sessions (e.g.: the auditing of large file transfers such as backups is rarely needed). The channels to be audited can be set via the *`audit_trails`* attribute. The available channel types are described in *Section 4.20.2.1, Configuring policies for SSH channels (p. 148)*.

4.20.2.8. Manipulating the keys of public-key authentication

The SSH proxy can use different keys in the server-side connection and the client-side connection. To use this feature, you have to derive a custom proxy class from the SshProxy class, and override the `mapUserKey` function. In the `mapUserKey` function, you can check the public key of the client, and return the private key that will be used in the server-side connection. Using this function you can set every connection to use a single key on the server side, change the type of the key from RSA to DSA, or restrict access of certain channels only to the selected users.

The `mapUserKey` function receives the *`blob_type`* and *`blob`* parameters that contain the type of the key (*`ssh-dss`* for DSA keys, *`ssh-rss`* for RSA keys) and the public key of the client. The function can return *`None`* to reject the connection, or a key type and a private key that will be used to authenticate on the target server.

Example 4.38. Modifying the keypair used in public-key authentication
The following proxy class accepts only connections that use a specific DSA public key, and uses a different RSA key-pair on the server side.

```
class KeymappingSshProxy(SshProxy):
        def config(self):
                SshProxy.config(self)
        def mapUserKey(self, blob_type, blob):
                if blob_type != 'ssh-dss' or blob != """ssh-dss
                AAAAB3NzaC1kc3MAAACBANhSxBWzv4kLvnBEV9sJX4rQkNtTxARJUP4lOu71Nu..."""
                        return None
                return ('ssh-rss', """-----BEGIN RSA PRIVATE KEY-----
                MIIEogIBAAKCAQEAz/U9WbGjeQfEj4nUoqSImQpKIPoNPIPQG2IPGTRC/ROc+VeQ
                D/ax8n7wB3PF/1DBOWpHK5jO75yJ6TPCPqFDYLOWOM41sBhyHsGCiGyDuNCOaRal
                ....
                -----END RSA PRIVATE KEY-----""")
```

4.20.3. Related standards

The Secure Shell (SSH) Protocol is described in the following RFCs:Architecture is described in RFC 4251.

- The Secure Shell (SSH) Protocol Architecture is described in RFC 4251.
- The Secure Shell (SSH) Authentication Protocol is described in RFC 4252.
- The Secure Shell (SSH) Transport Layer Protocol is described in RFC 4253.
- The Secure Shell (SSH) Connection Protocol is described in RFC 4254.

4.20.4. Classes in the Ssh module

Class	Description
AbstractSshProxy	Class encapsulating the abstract SSH proxy.
SshProxy	Class encapsulating the abstract SSH proxy.
SshProxySftpOnly	Ssh proxy based on SshProxy, allowing SFTP access only.
SshSFtpProxy	Class encapsulating an SFTP proxy.
SshScpProxy	Class encapsulating an SCP proxy.

Table 4.60. Classes of the Ssh module

4.20.5. Class AbstractSshProxy

This class implements an abstract SSH proxy for the SSH2 protocol - it serves as a starting point for customized proxy classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractSshProxy, or one of the predefined proxy classes.

4.20.5.1. Attributes of AbstractSshProxy

audit_channels (string, rw:r)
Default: ""
A comma separated list of channel types to be audited. See also *Section 4.20.2.7, Auditing SSH channels (p. 156)*.

auth_agent_forward (boolean, w:r)
Default: FALSE

auth_agent_forward (boolean, w:r)
Authenticate using the data received from the agent during agent-forwarding.

auth_methods (string, rw:rw)
Default: "password,keyboard-interactive,none"
A comma separated list of permitted authentication methods as defined in the SSH protocol specification. The proxy currently supports the following authentication methods: *`publickey`*, *`keyboard-interactive`*, *`password`* and *`none`*. The *`none`* method is only used to determine which authentication methods does the server support.

check_insane_settings (boolean, w:r)
Default: TRUE
Reject unrealistic terminal and screen settings. The number of columns and rows of the terminal must be lower than 512; the size of the screen cannot be greater than 8192 pixels in either directions.

client_channel (complex, r:r)
Default:
A normative policy hash defining the action to take when a specific channel type is opened on the client side. See *Section 4.20.2.1, Configuring policies for SSH channels (p. 148)* for details.

client_cipher_algos (string, rw:r)
Default: "aes128-gcm@openssh.com,aes256-gcm@openssh.com,aes128-ctr,aes192-ctr,aes256-ctr,aes128-cbc,blowfish-cbc,cast128-cbc,aes192-cbc,aes256-cbc,3des-cbc,arcfour"
A comma separated list of symmetric cipher algorithms permitted on the client side, in the order of preference. See *Section 4.20.2.5, Configuring encryption parameters (p. 155)* for details.

client_comp_algos (string, rw:r)
Default:
A comma separated list of compression algorithms, in the order of preference. Currently no compression algorithm is supported.

client_hostkey_algos (string, rw:r)
Default: "rsa-sha2-512,rsa-sha2-256,ssh-rsa,ssh-dss"
A comma separated list of hostkey algorithms permitted on the client side, in the order of preference. See *Section 4.20.2.5, Configuring encryption parameters (p. 155)* for details.

client_kex_algos (string, rw:r)
Default: "diffie-hellman-group16-sha512,diffie-hellman-group18-sha512,diffie-hellman-group14-sha256,diffie-hellman-group14-sha1,diffie-hellman-group1-sha1"
A comma separated list of allowed key exchange algorithms permitted on the client side, in the order of preference. See *Section 4.20.2.5, Configuring encryption parameters (p. 155)* for details.

client_mac_algos (string, rw:r)
Default: "hmac-sha2-256,hmac-sha2-512,hmac-sha1,hmac-md5"
A comma separated list of MAC algorithms, in the order of preference. See *Section 4.20.2.5, Configuring encryption parameters (p. 155)* for details.

client_pubkey_algos (string, rw:r)
Default: "rsa-sha2-512,rsa-sha2-256,ssh-rsa,ssh-dss"
A comma separated list of public key algorithms permitted on the client side, in the order of preference. See *Section 4.20.2.5, Configuring encryption parameters (p. 155)* for details.

client_request (complex, r:r)
Default:
A normative policy hash defining the action to take when a specific channel request is received from the client side. See *Section 4.20.2.2, Configuring policies for SSH requests (p. 149)* for details.

connection_start (enum, rw:r)
Default: SSH_CONN_START_IMMEDIATELY
Specifies when is the server-side connection started. When using agent authentication, set it to *`SSH_CONN_START_AFTER_PROXY_AUTH`*.

greeting (string, rw:r)
Default:
The content of this attribute is sent to the SSH client before sending the protocol header, e.g.: before performing key exchange or authentication. It is usually displayed to the user or sent to the system log.

host_key_x509_dss (string, rw:r)
Default:
The DSS host key in openssl PEM format used when communicating with SSH clients. Either `host_key_rsa` or `host_key_dss` is required.

host_key_x509_dss_certificate (string, rw:r)
Default:
The DSS host key in openssl PEM format used when communicating with SSH clients. Either *`host_key_rsa`* or *`host_key_dss`* is required.

host_key_x509_dss_files (certificate, rw:r)
Default:
A tuple of two file names containing the certificate and key files for the DSS host key in PEM format.

host_key_x509_rsa (string, rw:r)
Default:
The RSA host key in openssl PEM format used when communicating with SSH clients. Either *`host_key_rsa`* or *`host_key_dss`* is required.

host_key_x509_rsa_certificate (string, rw:r)
Default:
The RSA host key in openssl PEM format used when communicating with SSH clients. Either *`host_key_rsa`* or *`host_key_dss`* is required.

host_key_x509_rsa_files (certificate, rw:r)
Default:
A tuple of two file names containing the certificate and key files for the RSA host key in PEM format.

id_comment (string, rw:r)
Default:
Specifies the comment field in the SSH protocol header.

max_kbdint_prompt_len (integer, rw:r)
Default: 128
Specifies the maximum length of a prompt in the keyboard-interactive authentication method.

max_kbdint_prompts (integer, rw:r)
Default: 10
Specifies the maximum number of prompts in the keyboard-interactive authentication method.

max_kbdint_response_len (integer, rw:r)
Default: 128
Specifies the maximum length of a response in the keyboard-interactive authentication method.

server_channel (complex, r:r)
Default:
A normative policy hash defining the action to take when a specific channel type is opened on the server side. See *Section 4.20.2.1, Configuring policies for SSH channels (p. 148)* for details.

server_cipher_algos (string, rw:r)
Default: "aes128-gcm@openssh.com,aes256-gcm@openssh.com,aes128-ctr,aes192-ctr,aes256-ctr,aes128-cbc,blowfish-cbc,cast128-cbc,aes192-cbc,aes256-cbc,3des-cbc,arcfour"
A comma separated list of symmetric cipher algorithms permitted on the server side, in the order of preference.

server_comp_algos (string, rw:r)
Default:
A comma separated list of compression algorithms permitted on the server side, in the order of preference. Currently no compression algorithm is supported.

server_hostkey_algos (string, rw:r)
Default: "rsa-sha2-512,rsa-sha2-256,ssh-rsa,ssh-dss"
A comma separated list of hostkey algorithms permitted on the server side, in the order of preference. See *Section 4.20.2.5, Configuring encryption parameters (p. 155)* for details.

server_kex_algos (string, rw:r)
Default: "diffie-hellman-group16-sha512,diffie-hellman-group18-sha512,diffie-hellman-group14-sha256,diffie-hellman-group14-sha1,diffie-hellman-group1-sha1"
A comma separated list of key exchange algorithms permitted on the server side, in the order of preference. See *Section 4.20.2.5, Configuring encryption parameters (p. 155)* for details.

server_mac_algos (string, rw:r)
Default: "hmac-sha2-256,hmac-sha2-512,hmac-sha1,hmac-md5"
A comma separated list of MAC algorithms permitted on the server side, in the order of preference. See *Section 4.20.2.5, Configuring encryption parameters (p. 155)* for details.

server_pubkey_algos (string, rw:r)
Default: "rsa-sha2-512,rsa-sha2-256,ssh-rsa,ssh-dss"

server_pubkey_algos (string, rw:r)
A comma separated list of public key algorithms permitted on the server side, in the order of preference. See *Section 4.20.2.5, Configuring encryption parameters (p. 155)* for details.

server_request (complex, r:r)
Default:
A normative policy hash defining the action to take when a specific channel request is received from the server side. See *Section 4.20.2.2, Configuring policies for SSH requests (p. 149)* for details.

software_version (string, rw:r)
Default: "SSH"
The string sent to the SSH peers as the version of the software. Before changing the default, please note that peers enable or disable various protocol workarounds based on the value of this attribute.

timeout (integer, rw:r)
Default: 600000
I/O timeout in milliseconds. If no activity is detected within this period interval, the connection is terminated.

transparent_mode (boolean, rw:r)
Default: TRUE
Specifies whether the proxy is in transparent or non-transparent mode. In non-transparent mode the name of destination server is extracted from the username, which should be in the format (user@host:port). The set of characters accepted as username/hostname separators is '@' and '%'. The set of characters that separates hostname from port number is ':', '+' and '/'.

userauth_banner (string, rw:r)
Default:
The content of this attribute is sent to the SSH client at the start of the SSH userauth protocol. It is usually displayed by clients as a text message.

4.20.6. Class SshProxy

This proxy implements a default SSH proxy based on *AbstractSshProxy*. A number of higher-level attributes have been defined that allow easy configuration of the various services offered by SSH (e.g.: port-forwarding, etc.). Port-forwarding, X11-forwarding, and agent-forwarding are disabled by default, the clients may open only `session` channels. The following client requests are accepted in the channel: `window-change`, `pty-req`, `shell`, `exec`, `subsystem`, `signal`, `exit-status`, `exit-signal`, and `xon-xoff`. The `env` request is not permitted. Only known host keys are accepted on the server side.

4.20.6.1. Attributes of SshProxy

enable_agent_forward (boolean, rw:r)
Default: FALSE
Enable SSH agent forwarding specific requests and channels. NOTE: this is a high level interface for changing the low level attributes, thus using this setting while changing the low level policy hashes manually might lead to conflicts.

enable_port_forward (boolean, rw:r)
Default: FALSE
Enable port forwarding (both client and server initiated) specific requests and channels. NOTE: this is a high level interface for changing the low level attributes, thus using this setting while changing the low level policy hashes manually might lead to conflicts.

enable_x11_forward (boolean, rw:r)
Default: FALSE
Enable X11 display forwarding specific requests and channels. NOTE: this is a high level interface for changing the low level attributes, thus using this setting while changing the low level policy hashes manually might lead to conflicts.

host_key_dss_file (certificate, rw:r)
Default: ""
Read the DSS hostkey from the file specified. This must be DSA, not RSA.

host_key_rsa_file (certificate, rw:r)
Default: ""
Read the RSA hostkey from the file specified. This must be RSA, not DSA.

server_hostkeys_dir (trustedkeydir, rw:r)
Default:
The directory containing known SSH host keys.

server_hostkeys_verify (enum, rw:r)
Default: SSH_HKV_ACCEPT_KNOWN
The verification mode for SSH host keys. See *Section 4.20.2.6, Host key verification (p. 155)*.

4.20.6.2. SshProxy methods

Method	Description
checkUserKey(self, blob_type, blob)	None
mapUserKey(self, blob_type, blob)	None

Table 4.61. Method summary

Method checkUserKey(self, blob_type, blob)

This method is called by the proxy to check the publickey. It returns FALSE if it cannot be accepted, TRUE otherwise.

Method mapUserKey(self, blob_type, blob)

This method is called by the proxy to map the publickey of a user to a keypair.

4.20.7. Class SshProxySftpOnly

Ssh proxy based on SshProxy, allowing SFTP access only. Commands other than 'sftp' subsystem request are rejected.

4.20.8. Class SshSFtpProxy

This class implements an SFTP helper to be stacked into an SSH proxy parent.

4.20.8.1. Attributes of SshSFtpProxy

timeout (integer, rw:r)
Default: 600000
I/O timeout in milliseconds. If no activity is detected within this period interval, the connection is terminated.

4.20.9. Class SshScpProxy

This class implements an SCP helper to be stacked into an SSH proxy parent.

4.21. Module TFtp

The TFtp module defines the classes constituting the proxy for the TFTP protocol.

4.21.1. The TFtp protocol

Trivial File Transfer Protocol (TFTP) is a very simple protocol used to transfer files over the UDP transport protocol. It is commonly used for bootstrapping diskless systems (normally workstations or routers).

The protocol follows a very simple procedure. The client sends a request to read (RRQ) or write (WRQ) a file to the server's UDP/69 port. If the server grants the request a connection is opened and the file server starts sending the file in fixed length blocks of 512 bytes. TFTP transports data in *netascii* encoding format (ASCII text with each line terminated by the 2-character sequence of a carriage return followed by a linefeed called CR/LF) or *octet* (data as 8-bit bytes with no interpretation) which is set by the mode indicator at the end of the RRQ/WRQ message. The DATA packet also contains a block number which is used later for acknowledgment. Every packet sent must be acknowledged by the receiver, which guarantees that the previous packet has been received. If a packet is lost the receiver sends a request after a timeout. The server keeps just one packet in store for retransmission until the acknowledgment arrives. A packet shorter than 512 bytes indicates the end of the transmission.

Most errors cause termination of the transfer process and are signaled by the sending of an error packet. This is neither acknowledged nor retransmitted. If an error occurred, then an ERROR packet is sent. If a network error occurred then even the ERROR packet might get lost, therefore timeout is also used to detect errors.

Normal transmission termination is started by a packet smaller than 512 bytes. The packet is acknowledged by a normal ACK packet like all the previous packet. Then the host sends the final ACK and waits for a while before it terminates the transmission. If the final ACK is not acknowledged or the the connection timed out the final ACK packet is retransmitted.

4.21.1.1. Protocol elements

TFTP supports five types of packets, all of which have been mentioned above:

- 1 - Read request (RRQ)
- 2 - Write request (WRQ)
- 3 - Data (DATA)
- 4 - Acknowledgment (ACK)
- 5 - Error (ERROR), which can contain the following error messages:
  - 0 - Not defined, see error message (if any).
  - 1 - File not found.
  - 2 - Access violation.
  - 3 - Disk full or allocation exceeded.
  - 4 - Illegal TFTP operation.
  - 5 - Unknown transfer ID.
  - 6 - File already exists.
  - 7 - No such user.

4.21.2. Proxy behavior

TFtpProxy is a module built for parsing messages of the TFTP protocol. It reads and parses REQUESTs on the client side, and sends them to the server if the local security policy permits. The answers are similarly parsed and returned to the client if the local security policy permits. Rewriting the requested filename and encoding is supported (although transcoding is not).

One proxy instance is able to handle more than one session, if the Router and Chainer classes support fast path operation (currently this is supported in DirectedRouter). This functionality is similar to, but different from the secondary session handling used in PlugProxy and RadiusProxy. In TftpProxy the parameters of secondary sessions cannot be set, they are managed automatically based on the logic of the protocol.

4.21.2.1. Configuring policies for TFTP commands

Changing the default behaviour of requests is possible using the `request` attribute. This hash is indexed by the request method ("read" or "write"), and the requested filename. If the hash contains no entry for a given combination, the "*" entry is used. If there is no matching entry in the hash, the command is rejected. The possible actions are described in the following table. See also *Section 2.1, Policies for requests and responses (p. 4)*.

Action	Description
TFTP_REQ_ACCEPT	Allow the request to pass.
TFTP_REQ_REJECT	Reject the request and send an error message. Message code and text can be specified as second and third elements of the tuple.
TFTP_REQ_DROP	Drop the packet.
TFTP_REQ_POLICY	Call the function specified to make a decision about the event. The function receives four parameters: self, the method ("read"/"write"), the file name and the encoding used in the request. See *Section 2.1, Policies for requests and responses (p. 4)* for details.
TFTP_REQ_REWRITE	Rewrite filename and/or encoding and accept the packet. See *Section Rewriting the request (p. 166)* for details.

Table 4.62. Action codes on TFTP requests

Rewriting the request

To rewrite and accept a request, the hash value must be a tuple containing TFTP_REQ_REWRITE as the first value, and the filename and encoding to be sent to the server as the second and third values.

Responding with a custom error

To respond with a user-defined error code and message, the hash value must be a tuple containing TFTP_REQ_ERROR as the first value, the error code (an integer as defined by the TFTP RFC) as the second one, and the error message as the third. The session is (obviously) terminated; the TFTP server is not notified.

4.21.3. Related standards

Trivial File Transfer Protocol is described in RFC 1350.

4.21.4. Classes in the TFtp module

Class	Description
AbstractTFtpProxy	Class encapsulating the abstract TFtp proxy.
TFtpProxy	Default TFtp proxy class based on AbstractTFtpProxy.

Table 4.63. Classes of the TFtp module

4.21.5. Class AbstractTFtpProxy

This class implements the TFTP protocol as described in RFC 1350. It serves as a starting point for customized proxy classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractTFtpProxy, or the predefined TFtpProxy proxy class.

4.21.5.1. Attributes of AbstractTFtpProxy

encoding (string, n/a:r)
Default: n/a
Encoding used in the current transfer.

filename (string, n/a:r)
Default: n/a
Name of the file being transferred.

request (complex, rw:rw)
Default:
Normative policy hash for TFTP requests indexed by the request method and the filename. See also *Section 4.21.2.1, Configuring policies for TFTP commands (p. 166)*.

timeout (integer, rw:r)
Default: -1
Timeout in milliseconds. The -1 value disables the timeout.

4.21.6. Class TFtpProxy

A default proxy for the TFTP protocol based on AbstractTFtpProxy, allowing only read-only access.

4.22. Module Vnc

VNC protocol is for accessing the desktop of remote computers.

4.22.1. Classes in the Vnc module

Class	Description
AbstractVncProxy	Class encapsulating the abstract Vnc proxy.
VncProxy	Default Vnc proxy based on AbstractVncProxy.

Table 4.64. Classes of the Vnc module

4.22.2. Class AbstractVncProxy

This class implements the VNC protocol. AbstractVncProxy serves as a starting point for customized proxy classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractVncProxy, or one of the predefined VncProxy proxy classes.

4.22.2.1. Attributes of AbstractVncProxy

readonly (boolean)
Default: FALSE
Decides whether to block client activities or not.

4.22.3. Class VncProxy

VncProxy is a proxy class based on AbstractVncProxy, allowing the use of all Vnc options.

Chapter 5. Core

This chapter provides detailed description for the core modules of PNS.

5.1. Module Auth

This module contains classes related to authentication and authorization. Together with the *AuthDB* module it implements the Authentication and Authorization framework.

User authentication verifies the identity of the user trying to access a particular network service. When performed on the connection level, that enables the full auditing of the network traffic. Authentication is often used in conjunction with authorization, allowing access to a service only to clients who have the right to do so.

5.1.1. Authentication and authorization basics

Authentication is a method to ensure that certain services (access to a server, etc.) can be used only by the clients allowed to access the service. The process generally called as authentication actually consists of three distinct steps:

- *Identification*: Determining the clients identity (e.g.: requesting a username).
- *Authentication*: Verifying the clients identity (e.g.: requesting a password that only the real client knows).
- *Authorization*: Granting access to the service (e.g.: verifying that the authenticated client is allowed to access the service).

Note
It is important to note that although authentication and authorization are usually used together, they can also be used independently. Authentication verifies the identity of the client. There are situations where authentication is sufficient, because all users are allowed to access the services, only the event and the user's identity has to be logged. On the other hand, authorization is also possible without authentication, for example if access to a service is time-limited (e.g.: it can only be accessed outside the normal work-hours, etc.). In such situations authentication is not needed.

5.1.2. Authentication and authorization in PNS

PNS can authenticate and authorize access to the services. The aim of authentication is to identify the user and the associated group memberships. When the client initiates a connection, it actually tries to use a service. PNS checks if an *authentication policy* is associated to the service. If an authentication policy is present, PNS contacts the *authentication provider* specified in the authentication policy. The type of authentication (the authentication class used, e.g., InbandAuthentication) is also specified in the authentication policy. The authentication provider connects to an *authentication backend* (e.g., a user database) to perform the authentication of the client - PNS itself does not directly communicate with the database.

If the authentication is successful, the client is verified if it is allowed to access the service (by evaluating the *authorization policy* and the identity and group memberships of the client). If the client is authorized to access the service, the server-side connection is built. The client is automatically authorized if no authorization policy is assigned to the service.

Currently only one authentication provider, the Authentication Server (AS) is available via the *VAS2AuthenticationBackend* class. Authentication providers are actually configured instances of the authentication backends, and it is independent from the database that the backend connects to. The authentication backend is that ties the authentication provider to the server storing the user data. For details on using AS, see the *Connection authentication and authorization* chapter of the *PNS Administrator's Guide*.

The aim of authentication is to identify the user and resolve group memberships. The results are stored in the in the `auth_user` and `auth_groups` attributes of the *session* object. Note that apart from the information required for authentication, PNS also sends session information (e.g., the IP address of the client) to the authentication provider.

PNS provides the following authentication classes:

- *InbandAuthentication*: Use the built-in authentication of the protocol to authenticate the client on the PNS.
- *ServerAuthentication*: Enable the client to connect to the target server, and extract its authentication information from the protocol.
- *VAAuthentication*: Outband authentication using the Authentication Agent.

If the authentication is successful, PNS verifies that the client is allowed to access the service (by evaluating the authorization policy). If the client is authorized to access the service, the server-side connection is built. The client is automatically authorized if no authorization policy is assigned to the service.

Each service can use an authorization policy to determine whether a client is allowed to access the service. If the authorization is based on the identity of the client, it takes place only after a successful authentication - identity-based authorization can be performed only if the client's identity is known and has been verified. The actual authorization is performed by PNS, based on the authentication information received from AS or extracted from the protocol.

PNS provides the following authorization classes:

- *PermitUser*: Authorize listed users.
- *PermitGroup*: Authorize users belonging to the specified groups.
- *PermitTime*: Authorize connections in a specified time interval.
- *BasicAccessList*: Combine other authorization policies into a single rule.
- *PairAuthorization*: Authorize only user pairs.
- *NEyesAuthorization*: Have another client authorize every connection.

5.1.3. Classes in the Auth module

Class	Description
AbstractAuthentication	Class encapsulating the abstract authentication interface.

Class	Description
AbstractAuthorization	Class encapsulating the authorization interface.
AuthCache	Class encapsulating the authentication cache.
AuthenticationPolicy	A policy determining how the user is authenticated to access the service.
AuthorizationPolicy	A policy determining how the user is authorized to access the service.
BasicAccessList	Class encapsulating the authorization by access list.
InbandAuthentication	Class encapsulating the inband authentication interface.
NEyesAuthorization	Class encapsulating N eyes authorization.
PairAuthorization	Class encapsulating pair-based 4 eyes authorization.
PermitGroup	Class encapsulating the group membership based authorization.
PermitTime	Class encapsulating time based authorization.
PermitUser	Class encapsulating the user-name based authorization.
ServerAuthentication	Class encapsulating the server authentication interface.
VAAuthentication	Class encapsulating the outband authentication interface using the Vela Authentication Agent.

Table 5.1. Classes of the Auth module

5.1.4. Class AbstractAuthentication

This class encapsulates interfaces for inband and outband authentication procedures. Service definitions should refer to a customized class derived from AbstractAuthentication, or one of the predefined authentication classes, such as *InbandAuthentication* or *VAAuthentication*.

5.1.4.1. AbstractAuthentication methods

Method	Description
__init__(self)	Constructor to initialize an AbstractAuthentication instance.

Table 5.2. Method summary

Method __init__(self)

This constructor initializes an instance of the AbstractAuthentication class.

5.1.5. Class AbstractAuthorization

This class encapsulates an authorization interface. Authorization determines whether the authenticated entity is in fact allowed to access a specific service. Service definitions should refer to a customized class derived from AbstractAuthorization, or one of the predefined authorization classes, such as *PermitUser* or *PermitGroup*.

5.1.6. Class AuthCache

This class encapsulates an authentication cache which associates usernames with client IP addresses. The association between a username and an IP address is valid only until the specified timeout. Caching the authentication results means that the users do not need to authenticate themselves for every request: it is assumed that the same user is using the computer within the timeout. E.g.: once authenticated for an HTTP service, the client can browse the web for **Timeout** period, but has to authenticate again to use FTP.

To use a single authorization cache for every service request of a client, set the *`service_equiv`* attribute to *`TRUE`*. That way Vela does not make difference between the different services (protocols) used by the client: after a successful authentication the user can use all available services without having to perform another authentication. E.g.: if this option is enabled in the example above, the client does not have to re-authenticate for starting an FTP connection.

5.1.6.1. AuthCache methods

Method	Description
__init__(self, timeout, update_stamp, service_equiv, cleanup_threshold)	Constructor to initialize an instance of the AuthCache class.

Table 5.3. Method summary

Method __init__(self, timeout, update_stamp, service_equiv, cleanup_threshold)

This constructor initializes and registers an AuthCache instance that can be referenced in authentication policies.

Arguments of __init__

cleanup_threshold (integer)
Default: 100
When the number of entries in the cache reaches the value of *`cleanup_threshold`*, old entries are automatically deleted.

service_equiv (boolean)
Default: FALSE
If enabled, then a single authentication of a user applies to every service from that client.

timeout (integer)
Default: 600
Timeout while an authentication is assumed to be valid.

update_stamp (boolean)
Default: TRUE
If set to *TRUE*, then cached authentications increase the validity period of the authentication cache. Otherwise, the authentication cache expires according to the timeout value set in *attribute timeout (p. 173)*.

5.1.7. Class AuthenticationPolicy

Authentication policies determine how the user is authenticated to access the service. The *authentication_policy* attribute of a service can reference an instance of the AuthenticationPolicy class.

Example 5.1. A simple authentication policy
The following example defines an authentication policy that can be referenced in service definitions. This policy uses inband authentication and references an *authentication provider*.

```
AuthenticationPolicy(name="demo_authentication_policy", cache=None,
authentication=InbandAuthentication(), provider="demo_authentication_provider")
```

To use the authentication policy, include it in the definition of the service:

```
Service(name="office_http_inter", proxy_class=HttpProxy,
authentication_policy="demo_authentication_policy", authorization_policy="demo_authorization_policy")
```

Example 5.2. Caching authentication decisions
The following example defines an authentication policy that caches the authentication decisions for ten minutes (600 seconds). For details on authentication caching, see *Section 5.1.6, Class AuthCache (p. 172)*).

```
AuthenticationPolicy(name="demo_authentication_policy", cache=AuthCache(timeout=600, update_stamp=TRUE,
 service_equiv=TRUE, cleanup_threshold=100), authentication=InbandAuthentication(),
provider="demo_authentication_provider")
```

5.1.7.1. AuthenticationPolicy methods

Method	Description
__init__(self, name, provider, authentication, cache)	Constructor to initialize an instance of the AuthenticationPolicy class.

Table 5.4. Method summary

Method __init__(self, name, provider, authentication, cache)

Arguments of __init__

authentication (class)
Default: None
The authentication method used in the authentication process. See *Section 5.1.1, Authentication and authorization basics (p. 169)* for details.

cache (class)
Default: None
Caching method used to store authentication results.

name (string)
Default: n/a
Name identifying the AuthenticationPolicy instance.

provider (class)
Default: n/a
The authentication provider object used in the authentication process. See *Section 5.1.1, Authentication and authorization basics (p. 169)* for details.

5.1.8. Class AuthorizationPolicy

Authorization policies determine how the user is authorized to access the service. The `authorization_policy` attribute of a service can reference an instance of the AuthorizationPolicy class.

Example 5.3. A simple authorization policy
The following example defines an authotization policy that can be referenced in a service definition and permits only the members of the `admin` or `system` groups to access the service.

```
AuthorizationPolicy(name="demo_authorization_policy", authorization=PermitGroup(grouplist=("admin",
"system")))
```

To use the authorization policy, include it in the definition of the service:

```
Service(name="office_http_inter", proxy_class=HttpProxy,
authentication_policy="demo_authentication_policy", authorization_policy="demo_authorization_policy")
```

5.1.8.1. AuthorizationPolicy methods

Method	Description
__init__(self, name, authorization)	

Table 5.5. Method summary

Method __init__(self, name, authorization)

Arguments of __init__

authorization (class)
Default: n/a
The authorization method (e.g., *PermitGroup*) used in the instance. See *Section 5.1.8, Class AuthorizationPolicy (p. 174)* for examples.

name (string)
Default: n/a
Name of the AuthorizationPolicy instance. This name can be referenced in service definitions.

5.1.9. Class BasicAccessList

This class encapsulates an access list that uses any class derived from the AbstractAuthorization class. BasicAccessList allows to combine multiple access control requirements into a single decision.

BasicAccessList uses a list of rules. The rules are evaluated sequentially. Each rule can specify whether matching the current rule is *Sufficient* or *Required*. A connection is authorized if a *Sufficient* rule matches the connection, or all *Required* rules are fulfilled. If a *Required* rule is not met, the connection is refused.

Rules are represented as a list of Python tuples as the following example shows:

Example 5.4. BasicAccessList example
When referenced in a service definition, the following users can access the service:

- members of the *development* group;
- anyone with the *user1* username;
- anyone with the *user2* username.

```
AuthorizationPolicy(name='intra',
      authorization=BasicAccessList(
            ((V_BACL_SUFFICIENT, PermitUser('user1')),
             (V_BACL_SUFFICIENT, PermitUser('user2')),
             (V_BACL_REQUIRED, PermitGroup('development')))))
```

5.1.9.1. BasicAccessList methods

Method	**Description**
__init__(self, acl)	Constructor to initialize a BasicAccessList instance.

Table 5.6. Method summary

Method __init__(self, acl)

This constructor creates a new BasicAccessList instance which can be referenced in an authentication policy.

Arguments of __init__

acl (complex)
Default: n/a
Access control rules represented as a list of tuple.

5.1.10. Class InbandAuthentication

This class encapsulates inband authentication. Inband authentication is performed by the proxy using the rules of the application-level protocol. Only the authentication methods supported by the particular protocol can be used during inband authentication. *Authentication policies* can refer to instances of the InbandAuthentication class using the `auth` parameter.

Warning
Inband authentication is currently supported only for the Http, Ftp, and Socks proxy classes.

5.1.10.1. InbandAuthentication methods

Method	**Description**
__init__(self)	Constructor to initialize an InbandAuthentication instance.

Table 5.7. Method summary

Method __init__(self)

This constructor initializes an instance of the InbandAuthentication class.

5.1.11. Class NEyesAuthorization

This class encapsulates an N-eyes based authorization method, which means that connections are authorized if other administrators authenticate themselves within the defined timelimits.

When *NEyesAuthorization* is used, the client trying to access the service has to be authorized by another (already authorized) client (this authorization chain can be expanded to multiple levels). *NEyesAuthorization* can only be used in conjunction with another *NEyesAuthorization* policy. One of them is the *authorizer* set to authorize the *authorized* policy.

In a simple 4-eyes scenario the *authorizer* policy points to the authorized policy in its *Authorization policy* parameter, and has its *wait_authorization* parameter disabled. The *authorized* policy has an empty *Authorization policy* parameter (meaning that it is at lower the end of an N-eyes chain), and has its *wait_authorization* parameter enabled, meaning that it has to be authorized by another policy.

For examples on using the NEyesAuthorization class, see the *Proxying secure channels - SSH tutorial* available from the BalaSys Documentation Page at *http://www.balasys.hu/documentation/*.

5.1.11.1. NEyesAuthorization methods

Method	**Description**
__init__(self, authorize_policy, wait_authorization, wait_timeout)	Constructor to initialize a NEyesAuthorization instance.

Table 5.8. Method summary

Method __init__(self, authorize_policy, wait_authorization, wait_timeout)

This constructor initializes an NEyesAuthorization instance.

Arguments of __init__

authorize_policy (class)
Default: None
The authorization policy authorized by the current *NEyesAuthorization* policy.

wait_authorization (boolean)
Default: FALSE
Specifies whether the current authorization policy must wait for other authorization policies to finish. If this parameter is set, the client has to be authorized by another client. If set to *FALSE*, the current client is at the top of an authorizing chain.

wait_timeout (integer)
Default: 60000
The time (in milliseconds) Vela will wait for the authorizing user to authorize the one accessing the service. If the other authorizations are not completed in time, the current authorization will fail.

5.1.12. Class PairAuthorization

This class encapsulates pair-based authorization method. Only two users simultaneously accessing the service are authorized, single users are not permitted to access the service. Set the time (in milliseconds) Vela will wait for the second user to access the service using the *`wait_timeout`* parameter.

Example 5.5. A simple PairAuthorization policy
The following example permits access to the service only if two users having different usernames authenticate successfully within one minute.

```
AuthorizationPolicy(name="demo_pairauthorization_policy",
authorization=PairAuthorization(wait_timeout=60000))
```

For more detailed examples, see the *Proxying secure channels - SSH tutorial* available from the BalaSys Documentation Page at *http://www.balasys.hu/documentation/*.

5.1.12.1. PairAuthorization methods

Method	Description
__init__(self, wait_timeout)	Constructor to initialize a PairAuthorization instance.

Table 5.9. Method summary

Method __init__(self, wait_timeout)

This constructor initializes a PairAuthorization instance.

Arguments of __init__

wait_timeout (integer)
Default: 60000
The time (in milliseconds) Vela will wait for the pair to complete the authorization. If the authorizations are not completed in time, the current authorization will fail.

5.1.13. Class PermitGroup

This class encapsulates an authorization decision based on group membership. Users who authenticate as a member of a usergroup specified in the policy receive access to the service. Otherwise access is denied.

Example 5.6. A simple PermitGroup policy
The following example permits only the members of the *`admin`* or *`system`* groups to access the service.

```
AuthorizationPolicy(name="demo_authorization_policy", authorization=PermitGroup(grouplist=("admin",
"system")))
```

5.1.13.1. PermitGroup methods

Method	Description
__init__(self, grouplist)	Constructor to initialize a PermitGroup instance.

Table 5.10. Method summary

Method __init__(self, grouplist)

This constructor initilizes a PermitGroup instance.

Arguments of __init__

grouplist (complex)
Default: n/a
The list of authorized groups, represented as group names.

5.1.14. Class PermitTime

This class encapsulates an authorization decision based on the time when the connection is started. The connection is permitted if it is started in one of the permitted time periods (according to the system time of the host running Vela).

Specify the permitted time intervals as a comma-separated list, where each element contains the beginning and ending time of the permitted interval in `HH:MM` format.

Example 5.7. PermitTime example
When used in the `intervals` attribute of a PermitTime instance, the following example permits access only from 07:00 to 09:00 and from 17:00 to 19:00.

```
(("7:00", "9:00"), ("17:00", "19:00"))
```

The following is a complete authorization policy using the above intervals:

```
AuthorizationPolicy(name="demo_permittime_policy", authorization=PermitTime(intervals=(("7:00",
"9:00"), ("17:00", "19:00"))))
```

5.1.14.1. PermitTime methods

Method	Description
__init__(self, intervals)	Constructor to initialize a PermitTime instance.

Table 5.11. Method summary

Method __init__(self, intervals)

This constructor initilizes a PermitTime instance.

Arguments of __init__

intervals (complex)
Default: n/a
List of time intervals when connections are permitted (in *HH:MM, HH:MM* format).

5.1.15. Class PermitUser

This class encapsulates an authorization decision based on usernames. Users who authenticate using one of the usernames specified in the policy receive access to the service. Otherwise access is denied.

Example 5.8. A simple PermitUser policy
The following example permits only the *admin* and *root* users to access the service.

```
AuthorizationPolicy(name="demo_permituser", authorization=PermitUser(userlist=("admin", "root")))
```

5.1.15.1. PermitUser methods

Method	Description
__init__(self, userlist)	Constructor to initialize a PermitUser instance.

Table 5.12. Method summary

Method __init__(self, userlist)

This constructor initilizes a PermitUser instance.

Arguments of __init__

userlist (complex)
Default: n/a
Comma-separated list of authorized usernames.

5.1.16. Class ServerAuthentication

This class encapsulates server authentication: Vela authenticates the user based on the response of the server to the user's authentication request. Server authentication is a kind of inband authentication, it is performed within the application protocol, but the target server checks the credentials of the user instead of Vela. This authentication method is useful when the server can be trusted for authentication purposes, but you need to include an authorization decision in the service definition.

5.1.16.1. ServerAuthentication methods

Method	Description
__init__(self)	Constructor to initialize a ServerAuthentication instance.

Table 5.13. Method summary

Method __init__(self)

This constructor initializes an instance of the ServerAuthentication class.

5.1.17. Class VAAuthentication

This class encapsulates outband authentication using the Vela Authentication Agent (VAA). The Vela Authentication Agent is an application that runs on the client computers and provides an interface for the users to authenticate themselves when Vela requests authentication for accessing a service. This way any protocol, even those not supporting authentication can be securely authenticated. All communication between Vela and VAA is SSL-encrypted.

Example 5.9. Outband authentication example
The following authentication policy defines a class that uses outband authentication.

```
AuthenticationPolicy(name="demo_outbandauthentication_policy", cache=None,
authentication=VAAuthentication(port=1316, timeout=60000, connect_timeout=60000,
pki=("/etc/key.d/Vela_certificate/cert.pem", "/etc/key.d/Vela_certificate/key.pem")),
provider="demo_authentication_provider")
```

5.1.17.1. VAAuthentication methods

Method	Description
__init__(self, pki, port, timeout, connect_timeout)	Constructor to initialize an instance of the VAAuthentication class.

Table 5.14. Method summary

Method __init__(self, pki, port, timeout, connect_timeout)

This constructor initializes an instance of the VAAuthentication authentication class that can be referenced in authentication policies to perform outband authentication.

Arguments of __init__

connect_timeout (integer)
Default: 60000
Connection timeout (in milliseconds) to the Vela Authentication Agent.

pki (certificate)
Default: None
A tuple containing the name of a certificate and a key file. Vela uses this certificate to encrypt the communication with the Authentication Agents.

port (integer)
Default: 1316
The port number where the Vela Authentication Agent is listening. Default value: *`1316`*.

timeout (integer)
Default: 60000
Authentication timeout in milliseconds.

5.2. Module AuthDB

This module contains classes related to authentication databases. Together with the *Auth* module it implements the Authentication and Authorization framework. See *Section 5.1.1, Authentication and authorization basics (p. 169)* and *Section 5.1.2, Authentication and authorization in PNS (p. 169)* for details.

5.2.1. Classes in the AuthDB module

Class	Description
AbstractAuthenticationBackend	Class encapsulating the abstract authentication backend like VAS.
AuthenticationProvider	A database-independent class used by Vela to connect to an authentication backend.
VAS2AuthenticationBackend	Class encapsulating the VAS authentication backend.

Table 5.15. Classes of the AuthDB module

5.2.2. Class AbstractAuthenticationBackend

This is an abstract class to encapsulate an authentication backend, which is responsible for checking authentication credentials against a backend database. In actual configurations, use one of the derived classes like *VAS2AuthenticationBackend*.

The interface defined here is used by various authentication methods like *VAAuthentication* and *InbandAuthentication*.

5.2.3. Class AuthenticationProvider

The authentication provider is an intermediate layer that mediates between Vela and the *authentication backend* (e.g., a user database) during connection authentication - Vela itself does not directly communicate with the database.

Example 5.10. A sample authentication provider
The following example defines an authentication provider that uses the *VAS2AuthenticationBackend* backend.

```
AuthenticationProvider(name="demo_authentication_provider",
backend=VAS2AuthenticationBackend(serveraddr=SockAddrInet('192.168.10.10', 1317), use_ssl=TRUE,
ssl_verify_depth=3, pki_cert=("/etc/key.d/VAS_certificate/cert.pem",
"/etc/key.d/VAS_certificate/key.pem"), pki_ca=("/etc/ca.d/groups/demo_trusted_group/certs/",
"/etc/ca.d/groups/demo_trusted_group/crls/")))
```

5.2.3.1. AuthenticationProvider methods

Method	Description
__init__(self, name, backend)	Constructor to initialize an AbstractAuthorizationBackend instance.

Table 5.16. Method summary

Method __init__(self, name, backend)

This constructor initializes an AbstractAuthorizationBackend instance.

Arguments of __init__

backend (class)
Default: n/a
Type of the database backend used by the VAS instance.

name (string)
Default: n/a
Name of the VAS instance.

5.2.4. Class VAS2AuthenticationBackend

This class encapsulates a Vela Authentication Server database and provides interface to other authentication classes to verify against users managed through VAS. See *Section 5.2.3, Class AuthenticationProvider (p. 183)* for examples on using the VAS2AuthenticationBackend class.

5.2.4.1. VAS2AuthenticationBackend methods

Method	Description
__init__(self, serveraddr, use_ssl, pki_cert, pki_ca, ssl_verify_depth)	Constructor to initialize a VAS2AuthenticationProvider instance.

Table 5.17. Method summary

Method __init__(self, serveraddr, use_ssl, pki_cert, pki_ca, ssl_verify_depth)

This constructor creates a new VAS2AuthenticationProvider instance that can be used in authentication policies.

Arguments of __init__

pki_ca (cagroup)
Default: None
The name of a trusted CA group. When using SSL, VAS must show a certificate signed by a CA that belongs to this group.

pki_cert (certificate)
Default: None
A tuple containing the name of a certificate and a key file. Vela shows this certificate to VAS when using SSL.

serveraddr (sockaddr)
Default: n/a
The IP address of this VAS instance. VAS accepts connections on this address.

ssl_verify_depth (integer)
Default: 3
Specifies the maximum number of CAs in the trust chain when verifying the certificate of Vela.

use_ssl (boolean)
Default: FALSE
Enable this option if Vela communicates with VAS using SSL.

5.3. Module Chainer

Chainers establish a TCP or UDP connection between a proxy and a selected destination. The destination is usually a server, but the *SideStackChainer* connects an additional proxy before connecting the server.

5.3.1. Selecting the network protocol

The client-side and the server-side connections can use different networking protocols if needed. The *`protocol`* attribute of the chainer classes determines the network protocol used in the server-side connection. By default, the same protocol is used in both connections. The following options are available:

Name	Description
ZD_PROTO_AUTO	Use the protocol that is used on the client side.
ZD_PROTO_TCP	Use the TCP protocol on the server side.
ZD_PROTO_UDP	Use the UDP protocol on the server side.

Table 5.18. The network protocol used in the server-side connection

5.3.2. Classes in the Chainer module

Class	Description
AbstractChainer	Class encapsulating the abstract chainer.
ConnectChainer	Class to establish the server-side TCP/IP connection.
FailoverChainer	Class encapsulating the connection establishment with multiple target addresses and keeping down state between connects. FailoverChainer prefers connecting to target hosts in the order they were specified.
MultiTargetChainer	Class encapsulating connection establishment with multiple target addresses.
RoundRobinChainer	Class encapsulating the connection establishment with multiple target addresses and keeping down state between connects.
SideStackChainer	Class to pass the traffic to another proxy.
StateBasedChainer	Class encapsulating connection establishment with multiple target addresses and keeping down state between connects.

Table 5.19. Classes of the Chainer module

5.3.3. Class AbstractChainer

AbstractChainer implements an abstract chainer that establishes a connection between the parent proxy and the selected destination. This class serves as a starting point for customized chainer classes, but is itself not directly usable. Service definitions should refer to a customized class derived from AbstractChainer, or one of the predefined chainer classes, such as *ConnectChainer* or *FailoverChainer*.

5.3.4. Class ConnectChainer

ConnectChainer is the default chainer class based on AbstractChainer. This class establishes a TCP or UDP connection between the proxy and the selected destination address.

ConnectChainer is used by default if no other chainer class is specified in the service definition.

ConnectChainer attempts to connect only a single destination address: if the connection establishment procedure selects multiple target servers (e.g., a *DNSResolver* with the `multi=TRUE` parameter or a *DirectedRouter* with multiple addresses), ConnectChainer will use the first address and ignore all other addresses. Use *FailoverChainer* to select from the destination from multiple addresses in a failover fashion, and *RoundRobinChainer* to distribute connections in a roundrobin fashion.

Example 5.11. A sample ConnectChainer
The following service uses a ConnectChainer that uses the UDP protocol on the server side.

```
Service(name="demo_service", proxy_class=HttpProxy, chainer=ConnectChainer(protocol=VD_PROTO_UDP),
router=TransparentRouter(overrideable=FALSE, forge_addr=FALSE))
```

5.3.4.1. ConnectChainer methods

Method	Description
__init__(self, protocol, timeout_connect)	Constructor to initialize an instance of the ConnectChainer class.

Table 5.20. Method summary

Method __init__(self, protocol, timeout_connect)

This constructor creates a new ConnectChainer instance which can be associated with a *Service*.

Arguments of __init__

protocol (enum)
Default: VD_PROTO_AUTO
Optional parameter that specifies the network protocol used in the connection protocol. By default, the server-side communication uses the same protocol that is used on the client side. See *Section 5.3.1, Selecting the network protocol (p. 185)* for details.

timeout_connect (integer)
Default: 30000
Specifies connection timeout to be used when connecting to the target server.

5.3.5. Class FailoverChainer

This class is based on the *StateBasedChainer* class and encapsulates a real TCP/IP connection establishment, and is used when a top-level proxy wants to perform chaining. In addition to ConnectChainer this class adds the capability to perform stateful, failover HA functionality across a set of IP addresses.

Note
Use FailoverChainer if you want to connect to the servers in a predefined order: i.e., connect to the first server, and only connect to the second if the first server is unavailable.

If you want to distribute connections between the servers (i.e., direct every new connection to a different server to balance the load) use *RoundRobinChainer* .

Example 5.12. A DirectedRouter using FailoverChainer
The following service definition uses a DirectedRouter class with two possible destination addresses. These destinations are used in a failover fashion, targeting the second address only if the first one is unaccessible.

```
Service(name="intra_HTTP_inter", router=DirectedRouter(dest_addr=(SockAddrInet('192.168.55.55', 8080),
 SockAddrInet('192.168.55.56', 8080)), forge_addr=FALSE, forge_port=V_PORT_ANY, overrideable=FALSE),
 chainer=FailoverChainer(protocol=VD_PROTO_AUTO, timeout_state=60000, timeout_connect=30000),
max_instances=0, proxy_class=HttpProxy,)
```

5.3.5.1. FailoverChainer methods

Method	Description
__init__(self, protocol, timeout_state, timeout_connect)	Constructor to initialize a FailoverChainer instance.

Table 5.21. Method summary

Method __init__(self, protocol, timeout_state, timeout_connect)

This constructor initializes a FailoverChainer class by filling arguments with appropriate values and calling the inherited constructor.

Arguments of __init__

protocol (enum)
Default: VD_PROTO_AUTO
Optional, specifies connection protocol (*VD_PROTO_TCP* or *VD_PROTO_UDP*), when not specified it defaults to the protocol used on the client side.

timeout_connect (integer)
Default: 30000
Specifies connection timeout to be used when connecting to the target server.

timeout_state (integer)
Default: 60000
The down state of remote hosts is kept for this interval in milliseconds.

5.3.6. Class MultiTargetChainer

This class encapsulates a real TCP/IP connection establishment, and is used when a top-level proxy wants to perform chaining. In addition to ConnectChainer, this class adds the capability to perform stateless, simple load balance server connections among a set of IP addresses.

The same mechanism is used to set multiple server addresses as with a single destination address: the Router class sets a list of IP addresses in the `session.target_address` attribute.

5.3.6.1. MultiTargetChainer methods

Method	Description
__init__(self, protocol, timeout_connect)	Constructor to initialize a MultiTargetChainer instance.

Table 5.22. Method summary

Method __init__(self, protocol, timeout_connect)

This constructor initializes a MultiTargetChainer class by filling arguments with appropriate values and calling the inherited constructor.

Arguments of __init__

protocol (enum)
Default: VD_PROTO_AUTO
Optional, specifies connection protocol (either VD_PROTO_TCP or VD_PROTO_UDP), when not specified defaults to the same protocol as was used on the client side.

self (class)
Default: n/a
this instance

timeout_connect (integer)
Default: 30000
Specifies connection timeout to be used when connecting to the target server.

5.3.7. Class RoundRobinChainer

This class is based on the *StateBasedChainer* class and encapsulates a real TCP/IP connection establishment, and is used when a top-level proxy wants to perform chaining. In addition to ConnectChainer this class adds the capability to perform stateful, load balance server connections among a set of IP addresses.

Example 5.13. A DirectedRouter using RoundRobinChainer
The following service definition uses a RoundRobinChainer class with two possible destination addresses. These destinations are used in a roundrobin fashion, alternating between the two destinations.

```
Service(name="intra_HTTP_inter", router=DirectedRouter(dest_addr=(SockAddrInet('192.168.55.55', 8080),
 SockAddrInet('192.168.55.56', 8080)), forge_addr=FALSE, forge_port=V_PORT_ANY, overrideable=FALSE),
 chainer=RoundRobinChainer(protocol=VD_PROTO_AUTO, timeout_state=60000, timeout_connect=30000),
max_instances=0, proxy_class=HttpProxy)
```

5.3.8. Class SideStackChainer

This class encapsulates a special chainer. Instead of establishing a connection to a server, it creates a new proxy instance and connects the server side of the current (parent) proxy to the client side of the new (child) proxy. The *right_class* parameter specifies the child proxy.

It is possible to stack multiple proxies side-by-side. The final step of sidestacking is always to specify a regular chainer via the *right_chainer* parameter that connects the last proxy to the destination server.

Tip
Proxy sidestacking is useful for example to create one-sided SSL connections. See the tutorials of the BalaSys Documentation Page available at *http://www.balasys.hu/documentation/* for details.

5.3.8.1. Attributes of SideStackChainer

right_chainer (unknown)
Default: n/a
The chainer used to connect to the destination of the side-stacked proxy class set in the *right_class* attribute.

right_class (unknown)
Default: n/a
The proxy class to connect to the parent proxy. Both built-in and customized classes can be used.

5.3.8.2. SideStackChainer methods

Method	Description
__init__(self, right_class, right_chainer)	Constructor to initialize an instance of the SideStackChainer class.

Table 5.23. Method summary

Method __init__(self, right_class, right_chainer)

This constructor creates a new FailoverChainer instance which can be associated with a *Service*.

Arguments of __init__

right_chainer (class)
Default: None
The chainer used to connect to the destionation of the side-stacked proxy class set in the `right_class` attribute.

right_class (class)
Default: n/a
The proxy class to connect to the parent proxy. Both built-in or customized classes can be used.

5.3.9. Class StateBasedChainer

This class encapsulates a real TCP/IP connection establishment, and is used when a top-level proxy wants to perform chaining. In addition to ConnectChainer, this class adds the capability to perform stateful, load balance server connections among a set of IP addresses.

Note
Both the *FailoverChainer* and *RoundRobinChainer* classes are derived from StateBasedChainer.

5.3.9.1. StateBasedChainer methods

Method	Description
__init__(self, protocol, timeout_connect, timeout_state)	Constructor to initialize a StateBasedChainer instance.

Table 5.24. Method summary

Method __init__(self, protocol, timeout_connect, timeout_state)

This constructor initializes a StateBasedChainer class by filling arguments with appropriate values and calling the inherited constructor.

Arguments of __init__

protocol (enum)
Default: VD_PROTO_AUTO
Optional, specifies connection protocol (`VD_PROTO_TCP` or `VD_PROTO_UDP`), when not specified it defaults to the same protocol used on the client side.

timeout_connect (integer)
Default: 30000
Specifies connection timeout to be used when connecting to the target server.

timeout_state (integer)
Default: 60000
The down state of remote hosts is kept for this interval in miliseconds.

5.4. Module Detector

Detectors can be used to determine if the traffic in the incoming connection uses a particular protocol (for example, HTTP, SSH), or if it has other specific characteristics (for example, it uses SSL encryption with a specific certificate). Such characteristics of the traffic can be detected, and start a specific service to inspect the traffic (for example, start a specific HttpProxy for HTTP traffic, and so on).

5.4.1. Classes in the Detector module

Class	Description
AbstractDetector	Class encapsulating the abstract detector.
CertDetector	Class encapsulating a Detector that determines if an SSL/TLS-encrypted connection uses the specified certificate
DetectorPolicy	Class encapsulating a Detector which can be used by a name.
HttpDetector	Class encapsulating a Detector that determines if the traffic uses the HTTP protocol
SniDetector	Class encapsulating a Detector that determines whether a client targets a specific host in a SSL/TLS-encrypted connection.

Class	Description
SshDetector	Class encapsulating a Detector that determines if the traffic uses the SSHv2 protocol

Table 5.25. Classes of the Detector module

5.4.2. Class AbstractDetector

This abstract class encapsulates a detector that determines whether the traffic in a connection belongs to a particular protocol.

5.4.3. Class CertDetector

This Detector determines if an SSL/TLS-encrypted connection uses the specified certificate, and rejects any other protocols and certificates.

Example 5.14. CertDetector example

The following example defines a DetectorPolicy that detects if the traffic is SSL/TLS-encrypted, and uses the certificate specified.

```
          mycertificate="-----BEGIN CERTIFICATE-----
MIIEdjCCA16gAwIBAgIIQ7Xu3Mwnk+4wDQYJKoZIhvcNAQEFBQAwSTELMAkGA1UE
BhMCVVMxEzARBgNVBAoTCkdvb2dsZSBJbmMxJTAjBgNVBAMTHEdvb2dsZSBJbnRl
cm5ldCBBdXRob3JpdHkgRzIwHhcNMTQwMTI5MTQwNTM3WhcNMTQwNTI5MDAwMDAw
WjBoMQswCQYDVQQGEwJVUzETMBEGA1UECAwKQ2FsaWZvcm5pYTEWMBQGA1UEBwwN
TW91bnRhaW4gVmlldzETMBEGA1UECgwKR29vZ2xlIEluYzEXMBUGA1UEAwwOd3d3
Lmdvb2dsZS5jb20wggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCkeHmm
eYY7uMMRxKg14NPx8zFtD/VmUI2b4FdQYgD8AuRifA+fqvxicEki7Td1SrZ4zldn
AjbAS+fC0eQji8foJTosrkXgQgv5ds0+8lU3dooVXoqemeJKUihzI/h+7cf1287/
7EbMI5RaDBUPTHmZHeDtk38XUYsBrS93nICq4VDUAxy2BKsGSS2l9wRvl4fhdDDm
guQ5cRDKn/pqdYEqAqxFVEjamwjcUWSBsWlqSn37fI9s/MZDCzfMwz6AheFMrRNL
OoJ2Y3cVdBxiDVdqjGS+AG5qIUz/AsvHNL3JEsa55OSrMFubCPCzYDMAVLKziqZX
5G25cOe/qhObSK4/AgMBAAGjggFBMIIBPTAdBgNVHSUEFjAUBggrBgEFBQcDAQYI
KwYBBQUHAwIwGQYDVR0RBBIwEIIOd3d3Lmdvb2dsZS5jb20waAYIKwYBBQUHAQEE
XDBaMCsGCCsGAQUFBzAChh9odHRwOi8vcGtpLmdvb2dsZS5jb20vR0lBRzIuY3J0
MCsGCCsGAQUFBzABhh9odHRwOi8vY2xpZW50czEuZ29vZ2xlLmNvbS9vY3NwMB0G
A1UdDgQWBBR1IOrR+bm3NNXp5DWKruhkxnMrpDAMBgNVHRMBAf8EAjAAMB8GA1Ud
IwQYMBaAFErdBhYbvPZotXb1gba7Yhq6WoEvMBcGA1UdIAQQMA4wDAYKKwYBBAHW
eQIFATAwBgNVHR8EKTAnMCWgI6Ahhh9odHRwOi8vcGtpLmdvb2dsZS5jb20vR0lB
RzIuY3JsMA0GCSqGSIb3DQEBBQUAA4IBAQA6j9oPKE5k/FX5sbLY4p7xsnltndHD
N1oyzmb8+cmke6W/eFHsYOg+zUeUBW3zbOEMBnNXWNTCB1aVIcRGe8GUDDAnAzSX
MQBeBisNb69kn2untS7RblL83+8H787RsLeXucahr3kCoc61oTemIOHEI43ODtVI
uFEDNJDE1wqsHkdZecnNS29IZySpK2skr3rH7qUkbP1lkzbFvsnFUyp3AJS4ib9+
4xPr65GQfUi/8vgoSVvOy5Y3rT/U3CtI9tPoDSZTYGTl64LDxJa8dEGYmTKHgjyJ
HmbKzes13N/BN18XUlvTnjEaifQXvJj9ypqcMHUFPjkqwI1HSyb1iRth
-----END CERTIFICATE-----"
          DetectorPolicy(name="MyCertDetector", detector=CertDetector(certificate=mycertificate)
```

5.4.3.1. Attributes of CertDetector

certificate (unknown)
Default: n/a
The certificate to detect in PEM format. You can use the certificate directly, or store it in a file and reference the file with full path, for example, *`DetectorPolicy(name="MyCertDetector", detector=CertDetector(certificate=("/etc/key.d/mysite/cert.pem", )))`*

5.4.3.2. CertDetector methods

Method	Description
__init__(self, certificate)	Constructor to initialize a CertDetector instance.

Table 5.26. Method summary

Method __init__(self, certificate)

This constructor initializes a CertDetector instance

Arguments of __init__

certificate (certificate)
Default: n/a
The certificate in PEM format. This must contain either the certificate as a string, or a full pathname to a file containing the certificate.

5.4.4. Class DetectorPolicy

DetectorPolicy instances are reusable detectors that contain configured instances of the detector classes (for example, HttpDetector, SshDetector) that detect if the traffic uses a particular protocol, or a particular certificate in an SSL/TLS connection. DetectorPolicy instances can be used in the *`detect`* option of firewall rules. For examples, see the specific detector classes.

5.4.5. Class HttpDetector

This Detector determines if the traffic uses the HTTP protocol, and rejects any other protocol.

Example 5.15. HttpDetector example
The following example defines a DetectorPolicy that detects HTTP traffic.

```
DetectorPolicy(name="http", detector=HttpDetector()
```

5.4.5.1. Attributes of HttpDetector

ignore (unknown)
Default: n/a
A list of compiled regular expressions which should be ignored when detecting the traffic type. By default, this list is empty.

match (unknown)
Default: n/a

match (unknown)
A list of compiled regular expressions which result in a positive match. If the traffic matches this regular expression, it is regarded as HTTP traffic. Default value: *`[OPTIONS|GET|HEAD|POST|PUT|DELETE|TRACE|CONNECT] + ".*HTTP/1."`*

5.4.5.2. HttpDetector methods

Method	Description
*__init__(self, **kw)*	Constructor to initialize a HttpDetector instance.

Table 5.27. Method summary

Method __init__(self, **kw)

This constructor initializes a HttpDetector instance

5.4.6. Class SniDetector

Class encapsulating a Detector that determines whether a client targets a specific host in a SSL/TLS-encrypted connection and rejects any other protocols and hostnames.

Example 5.16. SNIDetector example
The following example defines a DetectorPolicy that detects if the traffic is SSL/TLS-encrypted, and uses targets the host www.example.com.

```
        DetectorPolicy(name="MySniDetector",
detector=SniDetector(RegexpMatcher(match_list=("www.example.com",))))
```

5.4.6.1. Attributes of SniDetector

server_name_matcher (class)
Default: n/a
Matcher class (e.g.: RegexpMatcher) used to check and filter hostnames in Server Name Indication TLS extension, for example, *`DetectorPolicy(name="MySniDetector", detector=SniDetector(RegexpMatcher(match_list=("www.example.com",))))`*

5.4.6.2. SniDetector methods

Method	Description
__init__(self, server_name_matcher)	Constructor to initialize a SNIDetector instance.

Table 5.28. Method summary

Method __init__(self, server_name_matcher)

This constructor initializes a SNIDetector instance

Arguments of __init__

server_name_matcher (class)
Default: n/a
Matcher class (e.g.: RegexpMatcher) used to check and filter hostnames in Server Name Indication TLS extension.

5.4.7. Class SshDetector

This Detector determines if the traffic uses the SSHv2 protocol, and rejects any other protocol.

Example 5.17. SshDetector example
The following example defines a DetectorPolicy that detects SSH traffic.

```
DetectorPolicy(name="ssh", detector=SshDetector()
```

5.5. Module Encryption

The TLS framework of the proxies is in a separate entity called Encryption policy. That way, you can easily share and reuse encryption settings between different services: you have to configure the Encryption policy once, and you can use it in multiple services. The TLS framework is described in *Chapter 3, The PNS SSL framework (p. 9)*.

Note
STARTTLS support is currently available only for the Ftp proxy to support FTPS sessions and for the SMTP and the Pop3 proxies.

5.5.1. TLS parameter constants

Name	Value
TLS_CIPHERS_DEFAULT	n/a
TLS_CIPHERS_OLD	n/a

Name	Value
TLS_CIPHERS_CUSTOM	n/a

Table 5.29. Constants for cipher selection

Name	Value
TLSV1_3_CIPHERS_DEFAULT	n/a
TLSV1_3_CIPHERS_CUSTOM	n/a

Table 5.30. Constants for TLSv1.3 cipher selection

Name	Value
TLS_SHARED_GROUPS_DEFAULT	n/a
TLS_SHARED_GROUPS_CUSTOM	n/a

Table 5.31. Constants for shared group selection

Name	Value
TLS_HSO_CLIENT_SERVER	Perform the TLS-handshake with the client first.
TLS_HSO_SERVER_CLIENT	Perform the TLS-handshake with the server first.

Table 5.32. Handshake order.

Name	Value
TLS_NONE	Disable encryption between Vela and the peer.
TLS_FORCE_TLS	Require encrypted communication between Vela and the peer.
TLS_ACCEPT_STARTTLS	Permit STARTTLS sessions. Currently supported only in the Ftp, Smtp and Pop3 proxies.

Table 5.33. Client connection security type.

Name	Value
TLS_NONE	Disable encryption between Vela and the peer.
TLS_FORCE_TLS	Require encrypted communication between Vela and the peer.
TLS_FORWARD_STARTTLS	Forward STARTTLS requests to the server. Currently supported only in the Ftp, Smtp and Pop3 proxies.

Table 5.34. Server connection security type.

Name	Value
TLS_TRUST_LEVEL_NONE	Accept invalid for example, expired certificates.
TLS_TRUST_LEVEL_UNTRUSTED	Both trusted and untrusted certificates are accepted.

Name	Value
TLS_TRUST_LEVEL_FULL	Only valid certificates signed by a trusted CA are accepted.

Table 5.35. Constants for trust level selection.

Name	Value
TLS_INTERMEDIATE_REVOCATION_NONE	Ignore result of CA certificate revocation status check.
TLS_INTERMEDIATE_REVOCATION_SOFT_FAIL	Check every CA certificate revocation state in the certificate chain. Uncertainty is tolerated.
TLS_INTERMEDIATE_REVOCATION_HARD_FAIL	Check every CA certificate revocation state in the certificate chain. Uncertainty is not tolerated.

Table 5.36. Constants for intermediate certificates revocation check type.

Name	Value
TLS_LEAF_REVOCATION_NONE	Ignore result of leaf certificate revocation status check.
TLS_LEAF_REVOCATION_SOFT_FAIL	Check the revocation state of the leaf certificate. Uncertainty is tolerated.
TLS_LEAF_REVOCATION_HARD_FAIL	Check the revocation state of the leaf certificate. Uncertainty is not tolerated.

Table 5.37. Constants for leaf certificate revocation check type.

Name	Value
TLS_ERROR	n/a
TLS_DEBUG	n/a

Table 5.38. Verbosity level of the log messages

Name	Value
TLS_HS_ACCEPT	0
TLS_HS_REJECT	1
TLS_HS_POLICY	6
TLS_HS_VERIFIED	10

Table 5.39. Handshake policy decisions

5.5.2. Classes in the Encryption module

Class	Description
AbstractVerifier	Class encapsulating the abstract Certificate verifier.

Class	Description
Certificate	Class encapsulating a certificate and its private key, and optionally the passphrase for the private key.
CertificateCA	Class encapsulating the certificate of a Certificate Authority (CA certificate) and its private key, and optionally the passphrase for the private key.
ClientCertificateVerifier	Class that can be used to verify the certificate of the client-side connection.
ClientNoneVerifier	Disables certificate verification in client-side connection.
ClientOnlyEncryption	The ClientOnlyEncryption class handles scenarios when only the client-Vela connection is encrypted, the Vela-server connection is not
ClientOnlyStartTLSEncryption	The client can optionally request STARTTLS encryption, but the server-side connection is always unencrypted.
ClientTLSOptions	Class encapsulating a set of TLS options used in the client-side connection.
DHParam	Class encapsulating DH parameters.
DynamicCertificate	Class to perform TLS keybridging.
DynamicServerEncryption	The DynamicServerEncryption class handles scenarios when both the client-firewall and the firewall-server connections could be encrypted but the server side encryption parameters set dynamically from proxies.
EncryptionPolicy	Class encapsulating a named set of encryption settings.
FakeStartTLSEncryption	The client can optionally request STARTTLS encryption, but the server-side connection is always encrypted.
ForwardStartTLSEncryption	The ForwardStartTLSEncryption class handles scenarios when the client can optionally request STARTTLS encryption.
PrivateKey	Class encapsulating a private key.
SNIBasedCertificate	Class to be used for Server Name Indication (SNI)
ServerCertificateVerifier	Class that can be used to verify the certificate of the server-side connection.
ServerNoneVerifier	Disables certificate verification in server-side connection.

Class	Description
ServerOnlyEncryption	The ServerOnlyEncryption class handles scenarios when only the Vela-server connection is encrypted, the client-Vela connection is not
ServerTLSOptions	Class encapsulating a set of TLS options used in the server-side connection.
StaticCertificate	Class encapsulating a static Certificate object.
TLSOptions	Class encapsulating the abstract TLS options.
TwoSidedEncryption	The TwoSidedEncryption class handles scenarios when both the client-Vela and the Vela-server connections are encrypted.

Table 5.40. Classes of the Encryption module

5.5.3. Class AbstractVerifier

This class includes the settings and options used to verify the certificates of the peers in TLS connections. Note that you cannot use this class directly, use an appropriate derived class, for example, *ClientCertificateVerifier* or *ServerCertificateVerifier* instead.

5.5.3.1. Attributes of AbstractVerifier

intermediate_revocation_check_type (enum)
Default: TLS_INTERMEDIATE_REVOCATION_SOFT_FAIL
Specify how intermediate certificates revocation status check should work.

leaf_revocation_check_type (enum)
Default: TLS_LEAF_REVOCATION_SOFT_FAIL
Specify how leaf certificate revocation status check should work.

required (boolean)
Default: trusted
If the `required` is TRUE, a certificate is required from the peer.

trust_level (enum)
Default: TLS_TRUST_LEVEL_FULL
Specify which certificate should be accepted as trusted.

trusted_certs_directory (string)
Default: ""
A directory where trusted IP address - certificate assignments are stored. When a peer from a specific IP address shows the certificate stored in this directory, it is accepted regardless of its expiration or issuer CA. Each file in the directory should contain a certificate in PEM format. The filename must bethe IP address.

verify_ca_directory (string)
Default: ""
Directory where the trusted CA certificates are stored. CA certificates are loaded on-demand from this directory when the certificate of the peer is verified.

verify_crl_directory (string)
Default: ""
Directory where the CRLs (Certificate Revocation Lists) associated with trusted CAs are stored. CRLs are loaded on-demand from this directory when the certificate of the peer is verified.

verify_depth (integer)
Default: 4
The length of the longest accepted CA verification chain. Longer CA chains are automatically rejected.

5.5.3.2. AbstractVerifier methods

Method	Description
__init__(self, trust_level, intermediate_revocation_check_type, leaf_revocation_check_type, trusted_certs_directory, required, verify_depth, verify_ca_directory, verify_crl_directory)	Constructor to initialize an AbstractVerifier instance.

Table 5.41. Method summary

Method __init__(self, trust_level, intermediate_revocation_check_type, leaf_revocation_check_type, trusted_certs_directory, required, verify_depth, verify_ca_directory, verify_crl_directory)

This constructor defines an AbstractVerifier with the specified parameters.

Arguments of __init__

intermediate_revocation_check_type (enum)
Default: TLS_INTERMEDIATE_REVOCATION_SOFT_FAIL
Specify how intermediate certificates revocation status check should work.

leaf_revocation_check_type (enum)
Default: TLS_LEAF_REVOCATION_SOFT_FAIL
Specify how leaf certificate revocation status check should work.

required (boolean)
Default: TRUE
If the *`required`* is TRUE, a certificate is required from the peer.

trust_level (enum)
Default: TLS_TRUST_LEVEL_FULL
Specify which certificate should be accepted as trusted.

trusted_certs_directory (string)
Default: ""
A directory where trusted IP address - certificate assignments are stored. When a peer from a specific IP address shows the certificate stored in this directory, it is accepted regardless of its expiration or issuer CA. Each file in the directory should contain a certificate in PEM format. The filename must bethe IP address.

verify_ca_directory (string)
Default: ""
Directory where the trusted CA certificates are stored. CA certificates are loaded on-demand from this directory when the certificate of the peer is verified.

verify_crl_directory (string)
Default: ""
Directory where the CRLs (Certificate Revocation Lists) associated with trusted CAs are stored. CRLs are loaded on-demand from this directory when the certificate of the peer is verified.

verify_depth (integer)
Default: 4
The length of the longest accepted CA verification chain. Longer CA chains are automatically rejected.

5.5.4. Class Certificate

The Certificate class stores a certificate, its private key, and optionally a passphrase for the private key. The certificate must be in PEM format.

When configuring Vela manually using its configuration file, use the regular constructor of the Certificate class to load a certificate from a string. To load a certificate from a file, use the *Certificate.fromFile* method.

Example 5.18. Loading a certificate
The following example loads a certificate from the configuration file.

```
my_certificate = "-----BEGIN CERTIFICATE-----
MIICUTCCAfugAwIBAgIBADANBgkqhkiG9w0BAQQFADBXMQswCQYDVQQGEwJDTjEL
MAkGA1UECBMCUE4xCzAJBgNVBAcTAkNOMQswCQYDVQQKEwJPTjELMAkGA1UECxMC
VU4xFDASBgNVBAMTC0hlcm9uZyBZYW5nMB4XDTA1MDcxNTIxMTk0N1oXDTA1MDgx
NDIxMTk0N1owVzELMAkGA1UEBhMCQ04xCzAJBgNVBAgTAlBOMQswCQYDVQQHEwJD
TjELMAkGA1UEChMCT04xCzAJBgNVBAsTAlVOMRQwEgYDVQQDEwtIZXJvbmcgWWFu
ZzBcMA0GCSqGSIb3DQEBAQUAA0sAMEgCQQCp5hnG7ogBhtlynpOS21cBewKE/B7j
V14qeyslnr26xZUsSVko36ZnhiaO/zbMOoRcKK9vEcgMtcLFuQTWDl3RAgMBAAGj
gbEwga4wHQYDVR0OBBYEFFXI70krXeQDxZgbaCQoR4jUDncEMH8GA1UdIwR4MHaA
FFXI70krXeQDxZgbaCQoR4jUDncEoVukWTBXMQswCQYDVQQGEwJDTjELMAkGA1UE
CBMCUE4xCzAJBgNVBAcTAkNOMQswCQYDVQQKEwJPTjELMAkGA1UECxMCVU4xFDAS
BgNVBAMTC0hlcm9uZyBZYW5nggEAMAwGA1UdEwQFMAMBAf8wDQYJKoZIhvcNAQEE
BQADQQA/ugzBrjjK9jcWnDVfGHlk3icNRq0oV7Ri32z/+HQX67aRfgZu7KWdI+Ju
Wm7DCfrPNGVwFWUQOmsPue9rZBgO
-----END CERTIFICATE-----"
                         my_certificate_object = Certificate(my_certificate, 'mypassphrase')
```

The following example loads a certificate from an external file.

```
my_certificate_object = Certificate.fromFile("/tmp/my_certificate.pem", 'mypassphrase')
```

5.5.4.1. Attributes of Certificate

certificate_file_path (certificatechain)
Default: n/a
The path and filename to the certificate file. The certificate must be in PEM format.

private_key_password (string)
Default: None
Passphrase used to access the private key of the certificate specified in `certificate_file_path`.

5.5.4.2. Certificate methods

Method	Description
__init__(self, certificate, private_key)	Load a certificate from a string, and access it using its passphrase
fromFile(certificate_file_path, private_key)	Load a certificate from a file, and access it using its passphrase

Table 5.42. Method summary

Method __init__(self, certificate, private_key)

Initializes a Certificate instance by loading a certificate from a string, and accesses it using its passphrase. To load a certificate from a file, use the *Certificate.fromFile* method.

Arguments of __init__

certificate_file_path (certificate)
Default: n/a
The path and filename to the certificate file. The certificate must be in PEM format.

private_key_password (string)
Default: None
Passphrase used to access the private key of the certificate specified in *`certificate_file_path`*.

Method fromFile(certificate_file_path, private_key)

Initializes a Certificate instance by loading a certificate from a file, and accesses it using its passphrase.

Arguments of fromFile

certificate_file_path (certificate)
Default: n/a
The path and filename to the certificate file. The certificate must be in PEM format.

passphrase (string)
Default: None
Passphrase used to access the private key specified in *`certificate_file_path`*.

5.5.5. Class CertificateCA

The CertificateCA class stores a CA certificate, its private key, and optionally a passphrase for the private key. The certificate must be in PEM format.

5.5.5.1. Attributes of CertificateCA

certificate_file_path (certificate)
Default: n/a
The path and filename to the certificate file. The certificate must be in PEM format.

private_key_password (string)
Default: None
Passphrase used to access the private key of the certificate specified in *`certificate_file_path`*.

5.5.5.2. CertificateCA methods

Method	Description
__init__(self, certificate, private_key)	Load a CAcertificate from a string, and access it using its passphrase

Table 5.43. Method summary

Method __init__(self, certificate, private_key)

Initializes a CertificateCA instance by loading a CA certificate, and accesses it using its passphrase.

Arguments of __init__

certificate_file_path (certificate)
Default: n/a
The path and filename to the CA certificate file. The certificate must be in PEM format.

private_key_password (string)
Default: None
Passphrase used to access the private key specified in *`certificate_file_path`*.

5.5.6. Class ClientCertificateVerifier

This class includes the settings and options used to verify the certificates of the peers in client-side TLS connections.

5.5.6.1. Attributes of ClientCertificateVerifier

ca_hint_directory (string)
Default: ""
Set directory containing certificates to provide the client the list of CA certificates (subject names) that are used for verifying the client certificate.

intermediate_revocation_check_type (enum)
Default: TLS_INTERMEDIATE_REVOCATION_SOFT_FAIL
Specify how intermediate certificates revocation status check should work.

leaf_revocation_check_type (enum)
Default: TLS_LEAF_REVOCATION_SOFT_FAIL
Specify how leaf certificate revocation status check should work.

required (boolean)
Default: TRUE
If the `required` is TRUE, a certificate is required from the peer.

trust_level (enum)
Default: TLS_TRUST_LEVEL_FULL
Specify which certificate should be accepted as trusted.

trusted_certs_directory (string)
Default: ""
A directory where trusted IP address - certificate assignments are stored. When a peer from a specific IP address shows the certificate stored in this directory, it is accepted regardless of its expiration or issuer CA. Each file in the directory should contain a certificate in PEM format. The filename must bethe IP address.

verify_ca_directory (string)
Default: ""
Directory where the trusted CA certificates are stored. CA certificates are loaded on-demand from this directory when the certificate of the peer is verified.

verify_crl_directory (string)
Default: ""
Directory where the CRLs (Certificate Revocation Lists) associated with trusted CAs are stored. CRLs are loaded on-demand from this directory when the certificate of the peer is verified.

verify_depth (integer)
Default: 4
The length of the longest accepted CA verification chain. Longer CA chains are automatically rejected.

5.5.6.2. ClientCertificateVerifier methods

Method	Description
__init__(self, trust_level, intermediate_revocation_check_type, leaf_revocation_check_type, trusted_certs_directory, required, verify_depth, verify_ca_directory, verify_crl_directory, ca_hint_directory)	Constructor to initialize a ClientCertificateVerifier instance.

Table 5.44. Method summary

Method __init__(self, trust_level, intermediate_revocation_check_type, leaf_revocation_check_type, trusted_certs_directory, required, verify_depth, verify_ca_directory, verify_crl_directory, ca_hint_directory)

This constructor defines a ClientCertificateVerifier with the specified parameters.

Arguments of __init__

ca_hint_directory (string)
Default: ""
Set directory containing certificates to provide the client the list of CA certificates (subject names) that are used for verifying the client certificate.

intermediate_revocation_check_type (enum)
Default: TLS_INTERMEDIATE_REVOCATION_SOFT_FAIL
Specify how intermediate certificates revocation status check should work.

leaf_revocation_check_type (enum)
Default: TLS_LEAF_REVOCATION_SOFT_FAIL
Specify how leaf certificate revocation status check should work.

required (boolean)
Default: TRUE
If the `required` is TRUE, a certificate is required from the peer.

trust_level (enum)
Default: TLS_TRUST_LEVEL_FULL
Specify which certificate should be accepted as trusted.

trusted_certs_directory (string)
Default: ""
A directory where trusted IP address - certificate assignments are stored. When a peer from a specific IP address shows the certificate stored in this directory, it is accepted regardless of its expiration or issuer CA. Each file in the directory should contain a certificate in PEM format. The filename must bethe IP address.

verify_ca_directory (string)
Default: ""
Directory where the trusted CA certificates are stored. CA certificates are loaded on-demand from this directory when the certificate of the peer is verified.

verify_crl_directory (string)
Default: ""
Directory where the CRLs (Certificate Revocation Lists) associated with trusted CAs are stored. CRLs are loaded on-demand from this directory when the certificate of the peer is verified.

verify_depth (integer)
Default: 4
The length of the longest accepted CA verification chain. Longer CA chains are automatically rejected.

5.5.7. Class ClientNoneVerifier

This class disables every certificate verification in client-side TLS connections.

5.5.8. Class ClientOnlyEncryption

The ClientOnlyEncryption class handles scenarios when only the client-Vela connection is encrypted, the Vela-server connection is not.

5.5.8.1. Attributes of ClientOnlyEncryption

client_certificate_generator (class)
Default: n/a
The class that will generate the certificate that will be showed to the client. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

client_tls_options (class)
Default: ClientTLSOptions()
The protocol-level encryption settings used on the client side. This must be a *ClientTLSOptions* instance.

client_verify (class)
Default: ClientCertificateVerifierGroup()
The settings used to verify the certificate of the client. This must be a *ClientCertificateVerifier* instance.

5.5.8.2. ClientOnlyEncryption methods

Method	Description
__init__(self, client_certificate_generator, client_verify, client_tls_options)	Initializes TLS connection on the client side.

Table 5.45. Method summary

Method __init__(self, client_certificate_generator, client_verify, client_tls_options)

The ClientOnlyEncryption class handles scenarios when only the client-Vela connection is encrypted, the Vela-server connection is not.

Arguments of __init__

client_certificate_generator (class)
Default: n/a
The class that will generate the certificate that will be showed to the client. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

client_tls_options (class)
Default: ClientTLSOptions()
The protocol-level encryption settings used on the client side. This must be a *ClientTLSOptions* instance.

client_verify (class)
Default: ClientCertificateVerifierGroup()
The settings used to verify the certificate of the client. This must be a *ClientCertificateVerifier* instance.

5.5.9. Class ClientOnlyStartTLSEncryption

The ClientOnlyStartTLSEncryption class handles scenarios when the client can optionally request STARTTLS encryption. If the client sends a STARTTLS request, the client-side connection will use STARTTLS. The server-side connection will not be encrypted.

Warning
If the client does not send a STARTTLS request, the client-side communication will not be encrypted at all. The server-side connection will never be encrypted.

5.5.9.1. Attributes of ClientOnlyStartTLSEncryption

client_certificate_generator (class)
Default: n/a
The class that will generate the certificate that will be showed to the client. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

client_tls_options (class)
Default: ClientTLSOptions()
The protocol-level encryption settings used on the client side. This must be a *ClientTLSOptions* instance.

client_verify (class)
Default: ClientCertificateVerifierGroup()
The settings used to verify the certificate of the client. This must be a *ClientCertificateVerifier* instance.

5.5.9.2. ClientOnlyStartTLSEncryption methods

Method	**Description**
__init__(self, client_certificate_generator, client_verify, client_tls_options)	The client can optionally request STARTTLS encryption, but the server-side connection is always unencrypted.

Table 5.46. Method summary

Method __init__(self, client_certificate_generator, client_verify, client_tls_options)

The ClientOnlyStartTLSEncryption class handles scenarios when the client can optionally request STARTTLS encryption. If the client sends a STARTTLS request, the client-side connection will use STARTTLS. The server-side connection will not be encrypted.

Warning
If the client does not send a STARTTLS request, the client-side communication will not be encrypted at all. The server-side connection will never be encrypted.

Arguments of __init__

client_certificate_generator (class)
Default: n/a
The class that will generate the certificate that will be showed to the client. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

client_tls_options (class)
Default: ClientTLSOptions()
The protocol-level encryption settings used on the client side. This must be a *ClientTLSOptions* instance.

client_verify (class)
Default: ClientCertificateVerifier()
The settings used to verify the certificate of the client. This must be a *ClientCertificateVerifier* instance.

5.5.10. Class ClientTLSOptions

This class (based on the TLSOptions class) collects the TLS settings directly related to encryption, for example, the permitted protocol versions, ciphers, session reuse settings, and so on.

5.5.10.1. Attributes of ClientTLSOptions

cipher_server_preference (boolean)
Default: FALSE
Use server and not client preference order when determining which cipher suite, signature algorithm or elliptic curve to use for an incoming connection.

ciphers (enum)
Default: n/a
Specifies the allowed ciphers. For details, see *Table 5.29, Constants for cipher selection (p. 195)*.

ciphers_tlsv1_3 (enum)
Default: n/a
Specifies the allowed ciphers for TLSv1.3 connections. For details, see *Table 5.30, Constants for TLSv1.3 cipher selection (p. 196)*.

dh_params (dhparams)
Default: None
The DH parameter used by ephemeral DH key generarion. Please be mind that this option is ignored in TLSv1.3 as it does not support custom DH parameters.

disable_compression (boolean)
Default: TRUE
Set this to FALSE to support TLS compression. Please be mind that this option is ignored in TLSv1.3 as it does not support compression.

disable_renegotiation (boolean)
Default: TRUE
Set this to TRUE to disable client initiated renegotiation. Please be mind that this option is ignored in TLSv1.3 as it does not support renegotiation.

disable_send_root_ca (boolean)
Default: FALSE
Inhibit sending Root CA to client, even if present in local certificate chain.

disable_session_cache (boolean)
Default: TRUE
Do not store session information in the session cache. Set this option to FALSE to enable TLS session reuse. Please be mind that this option is ignored in TLSv1.3 as it does not support session IDs.

disable_ticket (boolean)
Default: TRUE
Session tickets are a method for TLS session reuse, described in RFC 5077. Set this option to FALSE to enable TLS session reuse using session tickets.

prioritize_chacha (boolean)
Default: FALSE
When cipher_server_preference is TRUE, reprioritize ChaCha20-Poly1305 cipher if it is at the top of the client cipher list.

session_cache_size (integer)
Default: 20480
The number of sessions stored in the session cache for TLS session reuse. Please be mind that this option is ignored in TLSv1.3 as it does not support session IDs.

shared_groups (enum)
Default: n/a
Specifies the allowed shared groups. For details, see *Table 5.31, Constants for shared group selection (p. 196)*.

timeout (integer)
Default: 300
Drop idle connection if the timeout value (in seconds) expires.

tls_max_version (enum)
Default: TLS_VERSION_1_3
Specify the maximum supported TLS protocol version.

tls_min_version (enum)
Default: TLS_VERSION_1_2
Specify the minimum supported TLS protocol version.

5.5.10.2. ClientTLSOptions methods

Method	Description
__init__(self, tls_min_version, tls_max_version, ciphers, ciphers_tlsv1_3, shared_groups, timeout, session_cache_size, disable_session_cache, disable_ticket, disable_compression, cipher_server_preference, prioritize_chacha, dh_params, disable_renegotiation, disable_send_root_ca)	Constructor to initialize a ClientTLSOptions instance.

Table 5.47. Method summary

Method __init__(self, tls_min_version, tls_max_version, ciphers, ciphers_tlsv1_3, shared_groups, timeout, session_cache_size, disable_session_cache, disable_ticket, disable_compression, cipher_server_preference, prioritize_chacha, dh_params, disable_renegotiation, disable_send_root_ca)

This constructor defines a ClientTLSOptions with the specified parameters.

Arguments of __init__

cipher_server_preference (boolean)
Default: FALSE
Use server and not client preference order when determining which cipher suite, signature algorithm or elliptic curve to use for an incoming connection.

ciphers (enum)
Default: n/a
Specifies the allowed ciphers. For details, see *Table 5.29, Constants for cipher selection (p. 195).*

ciphers_tlsv1_3 (enum)
Default: n/a

ciphers_tlsv1_3 (enum)
Specifies the allowed ciphers for TLSv1.3 connections. For details, see *Table 5.30, Constants for TLSv1.3 cipher selection (p. 196)*.

dh_param_file_path (string)
Default: None
The path and filename to the DH parameter file. The DH parameter file must be in PEM format. Please be mind that this option is ignored in TLSv1.3 as it does not support custom DH parameters.

disable_renegotiation (boolean)
Default: TRUE
Set this to TRUE to disable client initiated renegotiation. Please be mind that this option is ignored in TLSv1.3 as it does not support renegotiation.

disable_send_root_ca (boolean)
Default: FALSE
Set this to TRUE to inhibit sending root ca to client, even if present in local chain.

disable_session_cache (boolean)
Default: TRUE
Do not store session information in the session cache. Set this option to FALSE to enable TLS session reuse. Please be mind that this option is ignored in TLSv1.3 as it does not support session IDs.

disable_ticket (boolean)
Default: TRUE
Session tickets are a method for TLS session reuse, described in RFC 5077. Set this option to FALSE to enable TLS session reuse using session tickets.

prioritize_chacha (boolean)
Default: FALSE
When cipher_server_preference is TRUE, reprioritize ChaCha20-Poly1305 cipher if it is at the top of the client cipher list.

session_cache_size (integer)
Default: 20480
The number of sessions stored in the session cache for TLS session reuse. Please be mind that this option is ignored in TLSv1.3 as it does not support session IDs.

shared_groups (enum)
Default: n/a
Specifies the allowed shared groups. For details, see *Table 5.31, Constants for shared group selection (p. 196)*.

timeout (integer)
Default: 300
Drop idle connection if the timeout value (in seconds) expires.

tls_max_version (enum)
Default: TLS_VERSION_1_3
Specify the maximum supported TLS protocol version.

tls_min_version (enum)
Default: TLS_VERSION_1_2
Specify the minimum supported TLS protocol version.

5.5.11. Class DHParam

The DHParam class stores DH parameters. The DH parameters must be in PEM format.

When configuring Vela manually using its configuration file, use the regular constructor of the DHParam class to load DH parameters key from a string. To load DH parameters key from a file, use the *DHParam.fromFile* method.

Example 5.19. Loading DH parameters
The following example loads DH parameters from the configuration file.

```
my_dh_params = "-----BEGIN DH PARAMETERS-----
MIIBCAKCAQEAvvO8WguTNtkDs33qe5u1T7IjllmTrRnwFV4z7W4AODu9j+prdRdD
UAblHYBrQn3OFsfg/6WDVTmUj8Lvgn9aFjWYTe6U3Ey7CQt4MBw2BhCO3Rl9KDw7
Im8UdBBhxuekuqZGifMkEEFzAcbiQepvBXiGMucDWgbLaaTY/FrKqb5O9DvoenSV
Aj/VNFnsefQTHXGo1Urg8ixaWj7kTNhM3x7kj7BhK4ALfBuv/93aet2SQjU2O7C6
Oj3mku8CD93Xsbng6rIzmRd6pCANEFHORgo1OX7+vMwwG5h5YDsF8cVAcRroZkxR
dyPdVNzYlz1X3Jxln3It/6F2yyx/FOXAGwIBAg==
-----END DH PARAMETERS-----"
                    my_dh_params_object = DHParam(my_dh_params)
```

The following example loads DH parameters key from an external file.

```
my_dh_params_object = DHParam.fromFile("/tmp/my_dh_params.pem")
```

5.5.11.1. Attributes of DHParam

params (string)
Default: ""

params (string)
The path and filename to the DH parameters file. The DH parameters must be in PEM format.

5.5.11.2. DHParam methods

Method	Description
__init__(self, params)	Load DH parameters key from a string
fromFile(file_path)	Load a DH parameters from a file

Table 5.48. Method summary

Method __init__(self, params)

Initializes a DHParam instance by loading DH parameters key from a string. To load a DH parameters from a file, use the *DHParam.fromFile* method.

Arguments of __init__

params (certificate)
Default: n/a
The path and filename to the DH parameters file. The DH parameters must be in PEM format.

Method fromFile(file_path)

Initializes a DHParam instance by loading a DH parameters from a file.

Arguments of fromFile

file_path (dhparam)
Default: n/a
The path and filename to the DH parameters file. The DH parameters must be in PEM format.

5.5.12. Class DynamicCertificate

This class is able to generate certificates mimicking another certificate, primarily used to transfer the information of a server's certificate to the client in keybridging. Can be used only in *TwoSidedEncryption*. For details on configuring keybridging, see *Procedure 3.2.8, Configuring keybridging (p. 26)*.

5.5.12.1. DynamicCertificate methods

Method	Description
__init__(self, private_key, trusted_ca, untrusted_ca, cache_directory, extension_whitelist)	Initializes a DynamicCertificate instance to use for keybridging

Table 5.49. Method summary

Method __init__(self, private_key, trusted_ca, untrusted_ca, cache_directory, extension_whitelist)

Arguments of __init__

cache_directory (string)
Default: None
The cache directory to store the keybridged generated certificates, for example, `/var/lib/vela/tlsbridge/`. The *`vela`* user must have write privileges for this directory.

extension_whitelist (complex)
Default: None

private_key (class)
Default: n/a
The private key of the CA certificate set in *`trusted_ca`*

trusted_ca (class)
Default: n/a
The CA certificate that will used to sign the keybridged certificate of trusted peers.

untrusted_ca (class)
Default: n/a
The CA certificate that will used to sign the keybridged certificate of untrusted peers.

5.5.13. Class DynamicServerEncryption

The DynamicServerEncryption class handles scenarios when both the client-firewall and the firewall-server connections could be encrypted but the server side encryption parameters set dynamically from proxies.

5.5.13.1. Attributes of DynamicServerEncryption

client_certificate_generator (class)
Default: n/a
The class that will generate the certificate that will be showed to the client. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

client_security (enum)
Default: n/a
Set security mode.

client_tls_options (class)
Default: ClientTLSOptions()
The protocol-level encryption settings used on the client side. This must be a *ClientTLSOptions* instance.

client_verify (class)
Default: ClientCertificateVerifierGroup()
The settings used to verify the certificate of the client. This must be a *ClientCertificateVerifier* instance.

5.5.13.2. DynamicServerEncryption methods

Method	**Description**
__init__(self, client_security, client_certificate_generator, client_verify, client_tls_options)	Initializes TLS connection on the client side.

Table 5.50. Method summary

Method __init__(self, client_security, client_certificate_generator, client_verify, client_tls_options)

The DynamicServerEncryption class handles scenarios when both the client-firewall and the firewall-server connections could be encrypted but the server side encryption parameters set dynamically from proxies.

Arguments of __init__

client_certificate_generator (class)
Default: n/a
The class that will generate the certificate that will be showed to the client. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

client_tls_options (class)
Default: ClientTLSOptions()
The protocol-level encryption settings used on the client side. This must be a *ClientTLSOptions* instance.

client_verify (class)
Default: ClientCertificateVerifierGroup()
The settings used to verify the certificate of the client. This must be a *ClientCertificateVerifier* instance.

5.5.14. Class EncryptionPolicy

This class encapsulates a named set of encryption settings and an associated Encryption policy instance. Encryption policies provide a way to re-use encryption settings without having to define encryption settings for each service individually.

5.5.14.1. Attributes of EncryptionPolicy

encryption (class)
Default: n/a
An encryption scenario instance that will be used in the Encryption Policy. This describes the scenario and the settings how encryption is used in the scenario, for example: ■ Both the client-side and the server-side connections are encrypted (*TwoSidedEncryption*) ■ Only the client-side connection is encrypted (*ClientOnlyEncryption*) ■ Only the server-side connection is encrypted (*ServerOnlyEncryption*) ■ STARTTLS is enabled (*ClientOnlyStartTLSEncryption*, *FakeStartTLSEncryption*, or *ForwardStartTLSEncryption*) To customize the settings of a scenario (for example, to set the used certificates), derive a class from the selected scenario, set its parameters as needed for your environment, and use the customized class.

name (string)
Default: n/a
Name identifying the Encryption policy.

5.5.14.2. EncryptionPolicy methods

Method	Description
__init__(self, name, encryption)	Constructor to create an Encryption policy.

Table 5.51. Method summary

Method __init__(self, name, encryption)

This constructor initializes an Encryption policy, based on the settings of the `encryption` parameter. This describes the scenario and the settings how encryption is used in the scenario, for example:

- Both the client-side and the server-side connections are encrypted (*TwoSidedEncryption*)
- Only the client-side connection is encrypted (*ClientOnlyEncryption*)
- Only the server-side connection is encrypted (*ServerOnlyEncryption*)
- STARTTLS is enabled (*ClientOnlyStartTLSEncryption*, *FakeStartTLSEncryption*, or *ForwardStartTLSEncryption*)

To customize the settings of a scenario (for example, to set the used certificates), derive a class from the selected scenario, set its parameters as needed for your environment, and use the customized class.

Arguments of __init__

encryption (class)
Default: n/a
An encryption scenario instance that will be used in the Encryption Policy.

name (string)
Default: n/a
Name identifying the Encryption policy.

5.5.15. Class FakeStartTLSEncryption

The FakeStartTLSEncryption class handles scenarios when the client can optionally request STARTTLS encryption. If the client sends a STARTTLS request, the client-side connection will use STARTTLS. The server-side connection will always be encrypted.

Warning
If the client does not send a STARTTLS request, the client-side communication will not be encrypted at all. The server-side connection will always be encrypted.

5.5.15.1. Attributes of FakeStartTLSEncryption

client_certificate_generator (class)
Default: n/a
The class that will generate the certificate that will be showed to the client. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

client_tls_options (class)
Default: ClientTLSOptions()
The protocol-level encryption settings used on the client side. This must be a *ClientTLSOptions* instance.

client_verify (class)
Default: ClientCertificateVerifierGroup()
The settings used to verify the certificate of the client. This must be a *ClientCertificateVerifier* instance.

server_tls_options (class)
Default: ServerTLSOptions()
The protocol-level encryption settings used on the server side. This must be a *ServerTLSOptions* instance.

server_verify (class)
Default: ServerCertificateVerifierGroup()
The settings used to verify the certificate of the server. This must be a *ServerCertificateVerifier* instance.

5.5.15.2. FakeStartTLSEncryption methods

Method	**Description**
__init__(self, client_certificate_generator, client_verify, server_verify, client_tls_options, server_tls_options)	Initializes a FakeStartTLSEncryption instance to handle scenarios when the client can optionally request STARTTLS encryption.

Table 5.52. Method summary

Method __init__(self, client_certificate_generator, client_verify, server_verify, client_tls_options, server_tls_options)

The FakeStartTLSEncryption class handles scenarios when the client can optionally request STARTTLS encryption. If the client sends a STARTTLS request, the client-side connection will use STARTTLS. The server-side connection will always be encrypted.

Warning
If the client does not send a STARTTLS request, the client-side communication will not be encrypted at all. The server-side connection will always be encrypted.

Arguments of __init__

client_certificate_generator (class)
Default: n/a
The class that will generate the certificate that will be showed to the client. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

client_tls_options (class)
Default: ClientTLSOptions()
The protocol-level encryption settings used on the client side. This must be a *ClientTLSOptions* instance.

client_verify (class)
Default: ClientCertificateVerifierGroup()
The settings used to verify the certificate of the client. This must be a *ClientCertificateVerifier* instance.

server_tls_options (class)
Default: ServerTLSOptions()
The protocol-level encryption settings used on the server side. This must be a *ServerTLSOptions* instance.

server_verify (class)
Default: ServerCertificateVerifierGroup()
The settings used to verify the certificate of the server. This must be a *ServerCertificateVerifier* instance.

5.5.16. Class ForwardStartTLSEncryption

The ForwardStartTLSEncryption class handles scenarios when the client can optionally request STARTTLS encryption. If the client sends a STARTTLS request, the client-side connection will use STARTTLS, and the request will be forwarded to the server. If the server supports STARTTLS, the server-side connection will also use STARTTLS.

Warning
If the client does not send a STARTTLS request, the communication will not be encrypted at all. Both the client-Vela and the Vela-server connections will be unencrypted.

5.5.16.1. Attributes of ForwardStartTLSEncryption

client_certificate_generator (class)
Default: n/a
The class that will generate the certificate that will be showed to the client. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

client_tls_options (class)
Default: ClientTLSOptions()
The protocol-level encryption settings used on the client side. This must be a *ClientTLSOptions* instance.

client_verify (class)
Default: ClientCertificateVerifierGroup()
The settings used to verify the certificate of the client. This must be a *ClientCertificateVerifier* instance.

server_tls_options (class)
Default: ServerTLSOptions()
The protocol-level encryption settings used on the server side. This must be a *ServerTLSOptions* instance.

server_verify (class)
Default: ServerCertificateVerifierGroup()
The settings used to verify the certificate of the server. This must be a *ServerCertificateVerifier* instance.

5.5.16.2. ForwardStartTLSEncryption methods

Method	**Description**
__init__(self, client_certificate_generator, client_verify, server_verify, client_tls_options, server_tls_options)	Initializes a ForwardStartTLSEncryption instance to handle scenarios when the client can optionally request STARTTLS encryption.

Table 5.53. Method summary

Method __init__(self, client_certificate_generator, client_verify, server_verify, client_tls_options, server_tls_options)

Initializes a ForwardStartTLSEncryption instance to handle scenarios when the client can optionally request STARTTLS encryption. If the client sends a STARTTLS request, the client-side connection will use STARTTLS, and the request will be forwarded to the server. If the server supports STARTTLS, the server-side connection will also use STARTTLS.

Warning
If the client does not send a STARTTLS request, the communication will not be encrypted at all. Both the client-Vela and the Vela-server connections will be unencrypted.

Arguments of __init__

client_certificate_generator (class)
Default: n/a
The class that will generate the certificate that will be showed to the client. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

client_tls_options (class)
Default: ClientTLSOptions()
The protocol-level encryption settings used on the client side. This must be a *ClientTLSOptions* instance.

client_verify (class)
Default: ClientCertificateVerifierGroup()
The settings used to verify the certificate of the client. This must be a *ClientCertificateVerifier* instance.

server_tls_options (class)
Default: ServerTLSOptions()
The protocol-level encryption settings used on the server side. This must be a *ServerTLSOptions* instance.

server_verify (class)
Default: ServerCertificateVerifierGroup()
The settings used to verify the certificate of the server. This must be a *ServerCertificateVerifier* instance.

5.5.17. Class PrivateKey

The PrivateKey class stores a private key and optionally a passphrase for the private key. The private key must be in PEM format.

When configuring Vela manually using its configuration file, use the regular constructor of the PrivateKey class to load a private key from a string. To load a private key from a file, use the *PrivateKey.fromFile* method.

Example 5.20. Loading a private key
The following example loads a private key from the configuration file.

```
my_private_key = "-----BEGIN RSA PRIVATE KEY-----
MIIEpgIBAAKCAQEA9rbxqq+Zi7OnRFAZe7SCTB6VgzP1PhkiUmOPmbwFmROSlSSy
yMPSyIzaQqwELyOSQTZtsT3jhd6MCFPBZntym63/GwDuethGSjE9y8rt/9yr+T3I
zz+6ABnZXHJ38tdGYataF1Ndi3CsY5NXGszVFv1Is17P5mbYWQgJ7QzI/a5mPKa+
9pVXsDQthEV3BVUawIEJJnSOTHD5XVQJ/MX6F4RPn+2MC9i/RbcAORVnLPmt2eiy
```

```
NV3+55sKdd7GpdMmEbRv9HZyW2xJNyu1xYbwU9YIP88dHCgvqoOgkAX2HLxCJOy6
2gvsS8J7HEbohD98dxPJX7P8w9juORi6HpsqOwIDAQABAoIBAQDXStIdJtuRC+GG
RXfXca/6iP3j3qV2KSzATRe+CkvAROo1CC9T7z6zb+bPI5kLIblxWvPiJaWOnn4I
jj5JFhTvMalagTeaz7yW5d2NR2rlSkZwW7Au2uePSv9ZIzL1IVLzzDnz/PW2xv5I
brOmT/Tr+N9GV8iIwNqu5sryp6OFasKB/55LhCcKVYrkdy2WhJc8Y8TXUjF4n8Jn
Xuyd44N6uu5RUiEgN7bPszO1F1T8ujCICwDNnYUw9lwSVvEC2EbTg84lu2UcnE4k
grB7rCKLooDpYlKjXx/1o9Dj9Uv3hwLpSTw2dYRoZSOkOFIKYACP1QcininrTGeL
cOPXyK6BAoGBAPvnBd7/U94Krp9Bp3jjxUEnlFrgf+B7QgRKpG7tN3RDRJmIVL8Z
mnxvbW6o4hsq4TzF/ratnRjqp+79Tw5wUz36G98ftWlTUs62OBznIkwImDGo+ysv
3QK8XUZ4Wg3EcnE5bG8AmOKoDRazcOg7UxopbHC+SNLRMZA/2dBvVh4zAoGBAPq6
UWIfcSnLyFYy7EPh3P7qmotBNPORgcX6aKdwR7pzk6MqTADHxKvIP+eeDEWpF58T
RYBW7KxN4h6cNMglRZBbhED3hONJkpYMGSqOhyczN4OSIHHrf3iBO7p35v7Eee82
2H/rT6BNrQF1fPIbz5spgT+eV5BuTAB7bsbWiuDhAoGBALVAgeT26y21mfhVkV9W
5LQA+qp5JworJlFYNADtBx3M2StwASqQDazDsIYTVr4dmHvWK3TebO9iaPt5oMzO
3daWhD+D3VCv98FtM+r4FKGI/Zmd8Twd8HTrfGIcbw/A7mex3efxEhDkwqY28Rhk
N2N3suNcx6GJjJQynVNxCRIpAoGBAOJyIEqUxynOiPOBLm3osiXxUP7wN5i8FA7w
qFCBUecNt4uoCdiyk+fqBf1OevT3UQQO7ZKJ71t3RAANaIZTUO6buQjMBFMbAa9O
4fP19BLtaQCaHH+HCCuX3I/+9rumS9JHIKX3qoTHYrdsmxo3D/u9MqR4p/EkDLRq
xpQC9I9BAoGBAPZtxtEKcOxhYeuor4qIQbt1edrO+cfEzaXyUvjleLdg8rU3Yeh3
JLbYgcSNr4rMvEwhuvwbwgWJjed7TvqjKKEYYSWW2ESwcmAjNIhDBVzX9oh1cY34
Ae/P63OHt89sWbb5oG2+fcb7xCwH3kYmVgT4/xPvOFQRspwpErKYlCWg
-----END RSA PRIVATE KEY-----"
                    my_private_key_object = PrivateKey(my_private_key, 'mypassphrase')
```

The following example loads a private key from an external file.

```
my_private_key_object = PrivateKey.fromFile("/tmp/my_private.key", 'mypassphrase')
```

5.5.17.1. Attributes of PrivateKey

key_file_path (string)
Default: ""
The path and filename to the private key file. The private key must be in PEM format.

passphrase (string)
Default: None
Passphrase used to access the private key specified in *`key_file_path`*.

5.5.17.2. PrivateKey methods

Method	**Description**
__init__(self, key, passphrase)	Load a private key from a string, and access it using its passphrase
fromFile(key_file_path, passphrase)	Load a private key from a file, and access it using its passphrase

Table 5.54. Method summary

Method __init__(self, key, passphrase)

Initializes a PrivateKey instance by loading a private key from a string, and accesses it using its passphrase. To load a private key from a file, use the *PrivateKey.fromFile* method.

Arguments of __init__

key_file_path (certificate)
Default: n/a
The path and filename to the private key file. The private key must be in PEM format.

passphrase (string)
Default: None
Passphrase used to access the private key specified in *`key_file_path`*.

Method fromFile(key_file_path, passphrase)

Initializes a PrivateKey instance by loading a private key from a file, and accesses it using its passphrase.

Arguments of fromFile

key_file_path (certificate)
Default: n/a
The path and filename to the private key file. The private key must be in PEM format.

passphrase (string)
Default: None
Passphrase used to access the private key specified in *`key_file_path`*.

5.5.18. Class SNIBasedCertificate

This class adds support for the Server Name Indication (SNI) TLS extension, as described in RFC 6066. It stores a mapping between hostnames and certificates, and automatically selects the certificate to show to the peer if the peer has sent an SNI request.

5.5.18.1. Attributes of SNIBasedCertificate

default (complex)
Default: None
The certificate to show to the peer if no matching hostname is found in *`hostname_certificate_map`*.

hostname_certificate_map (complex)
Default: n/a

hostname_certificate_map (complex)
A hash containing a matcher-certificate map. Each element of the hash contains a matcher and a certificate: if a matcher matches the hostname in the SNI request, the certificate is showed to the peer. You can use any matcher policy, though in most cases, RegexpMatcher will be adequate. Different elements of the hash can use different types of matchers, for example, RegexpMatcher and RegexpFileMatcher. For details on matcher policies, see *Section 5.8, Module Matcher (p. 243)*.

```
hostname_certificate_map={
                RegexpMatcher(
                 match_list=("myfirstdomain.example.com", )): StaticCertificate(

                        certificates=(Certificate.fromFile(

certificate_file_path="/etc/key.d/myfirstdomain/cert.pem",
                            private_key=PrivateKey.fromFile(
                                "/etc/key.d/myfirstdomain/key.pem")),)),}
```

5.5.18.2. SNIBasedCertificate methods

Method	Description
__init__(self, hostname_certificate_map, default)	

Table 5.55. Method summary

Method __init__(self, hostname_certificate_map, default)

Arguments of __init__

default (complex)
Default: None
The certificate to show to the peer if no matching hostname is found in `hostname_certificate_map`.

hostname_certificate_map (complex)
Default: n/a
A matcher-certificate map that describes which certificate will be showed to the peer if the matcher part matches the hostname in the SNI request. For details on matcher policies, see *Section 5.8, Module Matcher (p. 243)*.

5.5.19. Class ServerCertificateVerifier

This class includes the settings and options used to verify the certificates of the peers in server-side TLS connections. Note that the ServerCertificateVerifier class always requests a certificate from the server.

5.5.19.1. Attributes of ServerCertificateVerifier

check_subject (boolean)
Default: TRUE
If the `check_subject` parameter is TRUE, the Subject of the server-side certificate is compared with application-layer information (for example, it checks whether the Subject matches the hostname in the URL). For details, see *Section 3.2.5, Certificate verification options (p. 23)*.

intermediate_revocation_check_type (enum)
Default: TLS_INTERMEDIATE_REVOCATION_SOFT_FAIL
Specify how intermediate certificates revocation status check should work.

leaf_revocation_check_type (enum)
Default: TLS_LEAF_REVOCATION_SOFT_FAIL
Specify how leaf certificate revocation status check should work.

trust_level (enum)
Default: TLS_TRUST_LEVEL_FULL
Specify which certificate should be accepted as trusted.

trusted_certs_directory (string)
Default: ""
A directory where trusted IP address - certificate assignments are stored. When a peer from a specific IP address shows the certificate stored in this directory, it is accepted regardless of its expiration or issuer CA. Each file in the directory should contain a certificate in PEM format. The filename must bethe IP address.

verify_ca_directory (string)
Default: ""
Directory where the trusted CA certificates are stored. CA certificates are loaded on-demand from this directory when the certificate of the peer is verified.

verify_crl_directory (string)
Default: ""
Directory where the CRLs (Certificate Revocation Lists) associated with trusted CAs are stored. CRLs are loaded on-demand from this directory when the certificate of the peer is verified.

verify_depth (integer)
Default: 4

verify_depth (integer)
The length of the longest accepted CA verification chain. Longer CA chains are automatically rejected.

5.5.19.2. ServerCertificateVerifier methods

Method	Description
__init__(self, trust_level, intermediate_revocation_check_type, leaf_revocation_check_type, trusted_certs_directory, verify_depth, verify_ca_directory, verify_crl_directory, check_subject)	Constructor to initialize a ServerCertificateVerifier instance.

Table 5.56. Method summary

Method __init__(self, trust_level, intermediate_revocation_check_type, leaf_revocation_check_type, trusted_certs_directory, verify_depth, verify_ca_directory, verify_crl_directory, check_subject)

This constructor defines a ServerCertificateVerifier with the specified parameters.

Arguments of __init__

check_subject (boolean)
Default: TRUE
If the *check_subject* parameter is TRUE, the Subject of the server-side certificate is compared with application-layer information (for example, it checks whether the Subject matches the hostname in the URL). For details, see *Section 3.2.5, Certificate verification options (p. 23)*.

intermediate_revocation_check_type (enum)
Default: TLS_INTERMEDIATE_REVOCATION_SOFT_FAIL
Specify how intermediate certificates revocation status check should work.

leaf_revocation_check_type (enum)
Default: TLS_LEAF_REVOCATION_SOFT_FAIL
Specify how leaf certificate revocation status check should work.

trust_level (enum)
Default: TLS_TRUST_LEVEL_FULL
Specify which certificate should be accepted as trusted.

trusted_certs_directory (string)
Default: ""
A directory where trusted IP address - certificate assignments are stored. When a peer from a specific IP address shows the certificate stored in this directory, it is accepted regardless of its expiration or issuer CA. Each file in the directory should contain a certificate in PEM format. The filename must bethe IP address.

verify_ca_directory (string)
Default: ""
Directory where the trusted CA certificates are stored. CA certificates are loaded on-demand from this directory when the certificate of the peer is verified.

verify_crl_directory (string)
Default: ""
Directory where the CRLs (Certificate Revocation Lists) associated with trusted CAs are stored. CRLs are loaded on-demand from this directory when the certificate of the peer is verified.

verify_depth (integer)
Default: 4
The length of the longest accepted CA verification chain. Longer CA chains are automatically rejected.

5.5.20. Class ServerNoneVerifier

This class disables every certificate verification in server-side TLS connections.

5.5.21. Class ServerOnlyEncryption

The ServerOnlyEncryption class handles scenarios when only the Vela-server connection is encrypted, the client-Vela connection is not.

5.5.21.1. Attributes of ServerOnlyEncryption

server_certificate_generator (class)
Default: None
The class that will generate the certificate that will be showed to the server. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

server_tls_options (class)
Default: ServerTLSOptions()
The protocol-level encryption settings used on the server side. This must be a *ServerTLSOptions* instance.

server_verify (class)
Default: ServerCertificateVerifierGroup()
The settings used to verify the certificate of the server. This must be a *ServerCertificateVerifier* instance.

5.5.21.2. ServerOnlyEncryption methods

Method	Description
__init__(self, server_certificate_generator, server_verify, server_tls_options)	Initializes TLS connection on the server side.

Table 5.57. Method summary

Method __init__(self, server_certificate_generator, server_verify, server_tls_options)

The ServerOnlyEncryption class handles scenarios when only the Vela-server connection is encrypted, the client-Vela connection is not.

Arguments of __init__

server_certificate_generator (class)
Default: None
The class that will generate the certificate that will be showed to the server. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

server_tls_options (class)
Default: ServerTLSOptions()
The protocol-level encryption settings used on the server side. This must be a *ServerTLSOptions* instance.

server_verify (class)
Default: ServerCertificateVerifierGroup()
The settings used to verify the certificate of the server. This must be a *ServerCertificateVerifier* instance.

5.5.22. Class ServerTLSOptions

This class (based on the TLSOptions class) collects the TLS settings directly related to encryption, for example, the permitted protocol versions, ciphers, session reuse settings, and so on.

5.5.22.1. Attributes of ServerTLSOptions

ciphers (enum)
Default: n/a

ciphers (enum)
Specifies the allowed ciphers. For details, see *Table 5.29, Constants for cipher selection (p. 195)*.

ciphers_tlsv1_3 (enum)
Default: n/a
Specifies the allowed ciphers for TLSv1.3 connections. For details, see *Table 5.30, Constants for TLSv1.3 cipher selection (p. 196)*.

disable_compression (boolean)
Default: TRUE
Set this to FALSE to support TLS compression. Please be mind that this option is ignored in TLSv1.3 as it does not support compression.

disable_session_cache (boolean)
Default: TRUE
Do not store session information in the session cache. Set this option to FALSE to enable TLS session reuse. Please be mind that this option is ignored in TLSv1.3 as it does not support session IDs.

disable_ticket (boolean)
Default: TRUE
Session tickets are a method for TLS session reuse, described in RFC 5077. Set this option to FALSE to enable TLS session reuse using session tickets.

session_cache_size (integer)
Default: 20480
The number of sessions stored in the session cache for TLS session reuse. Please be mind that this option is ignored in TLSv1.3 as it does not support session IDs.

shared_groups (enum)
Default: n/a
Specifies the allowed shared groups. For details, see *Table 5.31, Constants for shared group selection (p. 196)*.

timeout (integer)
Default: 300
Drop idle connection if the timeout value (in seconds) expires.

tls_max_version (enum)
Default: TLS_VERSION_1_3
Specify the maximum supported TLS protocol version.

tls_min_version (enum)
Default: TLS_VERSION_1_2
Specify the minimum supported TLS protocol version.

5.5.22.2. ServerTLSOptions methods

Method	**Description**
__init__(self, tls_min_version, tls_max_version, ciphers, ciphers_tlsv1_3, shared_groups, timeout, session_cache_size, disable_session_cache, disable_ticket, disable_compression)	Constructor to initialize a ServerTLSOptions instance.

Table 5.58. Method summary

Method __init__(self, tls_min_version, tls_max_version, ciphers, ciphers_tlsv1_3, shared_groups, timeout, session_cache_size, disable_session_cache, disable_ticket, disable_compression)

This constructor defines a ServerTLSOptions with the specified parameters.

Arguments of __init__

ciphers (enum)
Default: n/a
Specifies the allowed ciphers. For details, see *Table 5.29, Constants for cipher selection (p. 195)*.

ciphers_tlsv1_3 (enum)
Default: n/a
Specifies the allowed ciphers for TLSv1.3 connections. For details, see *Table 5.30, Constants for TLSv1.3 cipher selection (p. 196)*.

disable_session_cache (boolean)
Default: TRUE
Do not store session information in the session cache. Set this option to FALSE to enable TLS session reuse. Please be mind that this option is ignored in TLSv1.3 as it does not support session IDs.

disable_ticket (boolean)
Default: TRUE
Session tickets are a method for TLS session reuse, described in RFC 5077. Set this option to FALSE to enable TLS session reuse using session tickets.

session_cache_size (integer)
Default: 20480
The number of sessions stored in the session cache for TLS session reuse. Please be mind that this option is ignored in TLSv1.3 as it does not support session IDs.

shared_groups (enum)
Default: n/a
Specifies the allowed shared groups. For details, see *Table 5.31, Constants for shared group selection (p. 196)*.

timeout (integer)
Default: 300
Drop idle connection if the timeout value (in seconds) expires.

tls_max_version (enum)
Default: TLS_VERSION_1_3
Specify the maximum supported TLS protocol version.

tls_min_version (enum)
Default: TLS_VERSION_1_2
Specify the minimum supported TLS protocol version.

5.5.23. Class StaticCertificate

This class encapsulates a static Certificate that can be used in TLS connections.

5.5.23.1. Attributes of StaticCertificate

certificates (complex)
Default: n/a
List of certificate instances to show to the peer.

5.5.23.2. StaticCertificate methods

Method	Description
__init__(self, certificates)	Initializes a static Certificate object.

Table 5.59. Method summary

Method __init__(self, certificates)

A static Certificate that can be used in TLS connections.

Arguments of __init__

certificates (complex)
Default: n/a
List of certificate instances to show to the peer.

5.5.24. Class TLSOptions

This class collects the TLS settings directly related to encryption, for example, the permitted protocol versions, ciphers, session reuse settings, and so on. Note that you cannot use this class directly, use an appropriate derived class, for example, *ClientTLSOptions* or *ServerTLSOptions* instead.

5.5.24.1. Attributes of TLSOptions

ciphers (complex)
Default: n/a
Specifies the allowed ciphers. For details, see *Table 5.29, Constants for cipher selection (p. 195)*.

ciphers_tlsv1_3 (complex)
Default: n/a
Specifies the allowed ciphers for TLSv1.3 connections. For details, see *Table 5.30, Constants for TLSv1.3 cipher selection (p. 196)*.

disable_compression (boolean)
Default: TRUE
Set this to FALSE to support TLS compression. Please be mind that this option is ignored in TLSv1.3 as it does not support compression.

disable_session_cache (boolean)
Default: TRUE

disable_session_cache (boolean)
Do not store session information in the session cache. Set this option to FALSE to enable TLS session reuse. Please be mind that this option is ignored in TLSv1.3 as it does not support session IDs.

disable_ticket (boolean)
Default: TRUE
Session tickets are a method for TLS session reuse, described in RFC 5077. Set this option to FALSE to enable TLS session reuse using session tickets.

session_cache_size (integer)
Default: 20480
The number of sessions stored in the session cache for TLS session reuse. Please be mind that this option is ignored in TLSv1.3 as it does not support session IDs.

shared_groups (complex)
Default: n/a
Specifies the allowed shared groups. For details, see *Table 5.31, Constants for shared group selection (p. 196)*.

timeout (integer)
Default: 300
Drop idle connection if the timeout value (in seconds) expires.

tls_max_version (enum)
Default: TLS_VERSION_1_3
Specify the maximum supported TLS protocol version.

tls_min_version (enum)
Default: TLS_VERSION_1_2
Specify the minimum supported TLS protocol version.

5.5.24.2. TLSOptions methods

Method	Description
__init__(self, tls_min_version, tls_max_version, ciphers, ciphers_tlsv1_3, shared_groups, timeout, session_cache_size, disable_session_cache, disable_ticket, disable_compression)	Constructor to initialize an TLSOptions instance.

Table 5.60. Method summary

Method __init__(self, tls_min_version, tls_max_version, ciphers, ciphers_tlsv1_3, shared_groups, timeout, session_cache_size, disable_session_cache, disable_ticket, disable_compression)

This constructor defines an TLSOptions with the specified parameters.

Arguments of __init__

ciphers (enum)
Default: n/a
Specifies the allowed ciphers. For details, see *Table 5.29, Constants for cipher selection (p. 195)*.

ciphers_tlsv1_3 (enum)
Default: n/a
Specifies the allowed ciphers for TLSv1.3 connections. For details, see *Table 5.30, Constants for TLSv1.3 cipher selection (p. 196)*.

disable_session_cache (boolean)
Default: TRUE
Do not store session information in the session cache. Set this option to FALSE to enable TLS session reuse. Please be mind that this option is ignored in TLSv1.3 as it does not support session IDs.

disable_ticket (boolean)
Default: TRUE
Session tickets are a method for TLS session reuse, described in RFC 5077. Set this option to FALSE to enable TLS session reuse using session tickets.

session_cache_size (integer)
Default: 20480
The number of sessions stored in the session cache for TLS session reuse. Please be mind that this option is ignored in TLSv1.3 as it does not support session IDs.

shared_groups (enum)
Default: n/a
Specifies the allowed shared groups. For details, see *Table 5.31, Constants for shared group selection (p. 196)*.

timeout (integer)
Default: 300
Drop idle connection if the timeout value (in seconds) expires.

tls_max_version (enum)
Default: TLS_VERSION_1_3
Specify the maximum supported TLS protocol version.

tls_min_version (enum)
Default: TLS_VERSION_1_2
Specify the minimum supported TLS protocol version.

5.5.25. Class TwoSidedEncryption

The TwoSidedEncryption class handles scenarios when both the client-Vela and the Vela-server connections are encrypted. If you do not need encryption on the client- or the server-side, use the *ServerOnlyEncryption* or *ClientOnlyEncryption* classes, respectively. For a detailed example on keybridging, see *Procedure 3.2.8, Configuring keybridging (p. 26)*.

5.5.25.1. Attributes of TwoSidedEncryption

client_certificate_generator (class)
Default: n/a
The class that will generate the certificate that will be showed to the client. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

client_tls_options (class)
Default: ClientTLSOptions()
The protocol-level encryption settings used on the client side. This must be a *ClientTLSOptions* instance.

client_verify (class)
Default: ClientCertificateVerifierGroup()
The settings used to verify the certificate of the client. This must be a *ClientCertificateVerifier* instance.

server_certificate_generator (class)
Default: None
The class that will generate the certificate that will be showed to the server. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

server_tls_options (class)
Default: ServerTLSOptions()
The protocol-level encryption settings used on the server side. This must be a *ServerTLSOptions* instance.

server_verify (class)
Default: ServerCertificateVerifierGroup()
The settings used to verify the certificate of the server. This must be a *ServerCertificateVerifier* instance.

5.5.25.2. TwoSidedEncryption methods

Method	**Description**
__init__(self, client_certificate_generator, server_certificate_generator, client_verify, server_verify, client_tls_options, server_tls_options)	Initializes TLS connection with both peers.

Table 5.61. Method summary

Method __init__(self, client_certificate_generator, server_certificate_generator, client_verify, server_verify, client_tls_options, server_tls_options)

The TwoSidedEncryption class handles scenarios when both the client-Vela and the Vela-server connections are encrypted.

Arguments of __init__

client_certificate_generator (class)
Default: n/a
The class that will generate the certificate that will be showed to the client. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

client_tls_options (class)
Default: ClientTLSOptions()
The protocol-level encryption settings used on the client side. This must be a *ClientTLSOptions* instance.

client_verify (class)
Default: ClientCertificateVerifierGroup()
The settings used to verify the certificate of the client. This must be a *ClientCertificateVerifier* instance.

server_certificate_generator (class)
Default: None
The class that will generate the certificate that will be showed to the server. You can use an instance of the *StaticCertificate*, *DynamicCertificate*, or *SNIBasedCertificate* classes.

server_tls_options (class)
Default: ServerTLSOptions()
The protocol-level encryption settings used on the server side. This must be a *ServerTLSOptions* instance.

server_verify (class)
Default: ServerCertificateVerifierGroup()
The settings used to verify the certificate of the server. This must be a *ServerCertificateVerifier* instance.

5.6. Module Ids

The IDS settings of the proxies is in a separate entity called Ids policy. That way, you can easily share and reuse encryption settings between different services: you have to configure the Ids policy once, and you can use it in multiple services.

5.6.1. Classes in the Ids module

Class	Description
Ids	Settings needed to send data towards an IDS.
IdsPolicy	Class encapsulating a named set of ids settings.

Table 5.62. Classes of the Ids module

5.6.2. Class Ids

5.6.2.1. Attributes of Ids

interface_name (string)
Default: n/a
The network interface name on which the traffic is sent.

mac_address (string)
Default: n/a
The MAC address of the ids.

5.6.2.2. Ids methods

Method	**Description**
__init__(self, interface_name, mac_address)	Settings needed to send data towards an IDS.

Table 5.63. Method summary

Method __init__(self, interface_name, mac_address)

Arguments of __init__

interface_name (string)
Default: n/a
The network interface name on which the traffic is sent.

mac_address (string)
Default: n/a
The MAC address of the IDS.

5.6.3. Class IdsPolicy

This class encapsulates a named set of ids settings and an associated Ids policy instance. Ids policies provide a way to re-use ids settings without having to define ids settings for each service individually.

5.6.3.1. Attributes of IdsPolicy

ids (class)
Default: n/a
An ids configuration instance that will be used in the Ids Policy. This describes the settings to connect to the IDS.

name (string)
Default: n/a
Name identifying the Ids policy.

5.6.3.2. IdsPolicy methods

Method	Description
__init__(self, name, ids)	Constructor to create an Ids policy.

Table 5.64. Method summary

Method __init__(self, name, ids)

This constructor initializes an Ids policy, based on the settings of the `ids` parameter. This describes the settings to connect to the IDS

Arguments of __init__

ids (class)
Default: n/a
An ids settings instance that will be used in the Ids Policy.

name (string)
Default: n/a
Name identifying the Ids policy.

5.7. Module Keybridge

Keybridging is a method to let the client see a copy of the server's certificate (or vice versa), allowing it to inspect it and decide about its trustworthiness. Because of proxying the SSL/TLS connection, the client is not able to inspect the certificate of the server directly, therefore a certificate based on the server's certificate is generated on-the-fly. This generated certificate is presented to the client.

For details on configuring keybridging, see *Procedure 3.2.8, Configuring keybridging (p. 26)*.

5.7.1. Classes in the Keybridge module

Class	Description
X509KeyBridge	Class to perform SSL keybridging.

Table 5.65. Classes of the Keybridge module

5.7.2. Class X509KeyBridge

This class is able to generate certificates mimicking another certificate, primarily used to transfer the information of a server's certificate to the client in keybridging. For details on configuring keybridging, see *Procedure 3.2.8, Configuring keybridging (p. 26)*.

5.7.2.1. Attributes of X509KeyBridge

cache_directory (string)
Default: ""
The directory where all automatically generated certificates are cached.

key_file (string)
Default: ""
Name of the private key to be used for the newly generated certificates.

key_passphrase (string)
Default: ""
Passphrase required to access the private key stored in *`key_file`*.

trusted_ca_files (certificate)
Default: None
A tuple of *`cert_file`*, *`key_file`*, *`passphrase`*) for the CA used for keybridging trusted certificates.

untrusted_ca_files (certificate)
Default: None
A tuple of *`cert_file`*, *`key_file`*, *`passphrase`*) for the CA used for keybridging untrusted certificates.

5.7.2.2. X509KeyBridge methods

Method	**Description**
_old_init(self, key_file, cache_directory, trusted_ca_files, untrusted_ca_files, key_passphrase, extension_whitelist)	None

Table 5.66. Method summary

Method _old_init(self, key_file, cache_directory, trusted_ca_files, untrusted_ca_files, key_passphrase, extension_whitelist)

n/a

Arguments of _old_init

cache_directory (string)
Default: "/var/lib/vela/keybridge-cache"

cache_directory (string)
The directory where all automatically generated certificates are cached.

extension_whitelist (complex)
Default: None
The following certificate extensions are transfered to the client side: *Key Usage*, *Subject Alternative Name*, *Extended Key Usage*. Other extensions will be automatically deleted during keybridging. This is needed because some certificate extensions contain references to the Issuer CA, which references become invalid for keybridged certificates. To transfer other extensions, list them in the *extension_whitelist* parameter. Note that modifying this parameter replaces the default values, so to extend the list of transferred extensions, include the *'keyUsage', 'subjectAltName', 'extendedKeyUsage'* list as well. For example: `self.extension_whitelist = ('keyUsage', 'subjectAltName', 'extendedKeyUsage', 'customExtension')`

key_file (certificate)
Default: n/a
Name of the private key to be used for the newly generated certificates.

key_passphrase (string)
Default: ""
Passphrase required to access the private key stored in *key_file*.

trusted_ca_files (certificate)
Default: n/a
A tuple of *cert_file*, *key_file*, *passphrase*) for the CA used for keybridging trusted certificates.

untrusted_ca_files (certificate)
Default: None
A tuple of *cert_file*, *key_file*, *passphrase*) for the CA used for keybridging untrusted certificates.

5.8. Module Matcher

In general, matcher policies can be used to find out if a parameter is included in a list (or which elements of a list correspond to a certain parameter), and influence the behavior of the proxy class based on the results. Matchers can be used for a wide range of tasks, for example, to determine if the particular IP address or URL that a client is trying to access is on a black or whitelist, or to verify that a particular e-mail address is valid.

5.8.1. Classes in the Matcher module

Class	Description
AbstractMatcher	Class encapsulating the abstract string matcher.
CombineMatcher	Matcher for implementing logical expressions based on other matchers.
DNSMatcher	DNS matcher
MatcherPolicy	Class encapsulating a Matcher which can be used by a name.
RegexpFileMatcher	Class encapsulating Matcher which uses regular expressions stored in files for string matching.
RegexpMatcher	Class encapsulating a Matcher which uses regular expressions for string matching.
SmtpInvalidRecipientMatcher	Class verifying the validity of the recipient addresses in E-mails.
WindowsUpdateMatcher	Windows Update matcher

Table 5.67. Classes of the Matcher module

5.8.2. Class AbstractMatcher

This abstract class encapsulates a string matcher that determines whether a given string is found in a backend database.

Specialized subclasses of AbstractMatcher exist such as 'RegexpFileMatcher' which use regular expressions stored in flat files to find matches.

5.8.3. Class CombineMatcher

This matcher makes it possible to combine the results of several matchers using logical operations. CombineMatcher uses prefix-notation in its expressions and uses the following format: the operand, a comma, first argument, a comma, second argument. For example, an AND expression should be formatted the following way: `(V_AND, matcher1, matcher2)`. Expressions using more than one operands should be bracketed, e.g., `(V_OR (V_AND, matcher1, matcher2), matcher3)`. The following oprations are available:

- *V_AND* : Logical AND operation.
- *V_OR* : Logical OR operation.
- *V_XOR* : Logical XOR operation.
- *V_NOT* : Logical negation.
- *V_EQ* : Logical equation.

Example 5.21. Whitelisting e-mail recipients
A simple use for CombineMatcher is to filter the recipients of e-mail addresses using the following process:

1. An SmtpInvalidMatcher (called *`SmtpCheckrecipient`*) verifies that the recipient exists.
2. A RegexpMatcher (called *`SmtpWhitelist`*) or RegexpFileMatcher is used to check if the address is on a predefined list (list of permitted addresses).
3. A CombineMatcher (called *`SmtpCombineMatcher`*) sums up the results of the matchers with a logical AND operation.
4. An SmtpProxy (called *`SmtpRecipientMatcherProxy`*) references *`SmtpCombineMatcher`* in its *`recipient_matcher`* attribute.

```
Python:
class SmtpRecipientMatcherProxy(SmtpProxy):
recipient_matcher="SmtpCombineMatcher"
def config(self):
super(SmtpRecipientMatcherProxy, self).config()

MatcherPolicy(name="SmtpCombineMatcher", matcher=CombineMatcher (expr=(V_AND, "SmtpCheckrecipient",
"SmtpWhitelist")))
MatcherPolicy(name="SmtpWhitelist", matcher=RegexpMatcher (match_list=("info@example.com",),
ignore_list=None))
MatcherPolicy(name="SmtpCheckrecipient", matcher=SmtpInvalidRecipientMatcher (server_port=25,
cache_timeout=60, force_delivery_attempt=FALSE, server_name="recipientcheck.example.com"))
```

5.8.4. Class DNSMatcher

DNSMatcher retrieves the IP addresses of domain names. This can be used in domain name based policy decisions, for example to allow encrypted connections only to trusted e-banking sites.

DNSMatcher operates as follows: it resolves the IP addresses stored in the list of domain names using the specified Domain Name Server, and compares the results to the IP address of the connection (i.e., the IP address of the server or the client). The matcher returns a true value if the IP addresses resolved from the list of domain names include the IP address of the connection.

Example 5.22. DNSMatcher example
The following DNSMatcher class uses the *`dns.example.com`* name server to resolve the *`example2.com`* and *`example3.com`* domain names.

```
MatcherPolicy(name="ExampleDomainMatcher", matcher=DNSMatcher(server="dns.example.com",
hosts=("example2.com", "example3.com")))
```

5.8.4.1. DNSMatcher methods

Method	Description
__init__(self, hosts, server, resolve_on_init)	Constructor to initialize an instance of the DNSMatcher class.

Table 5.68. Method summary

Method __init__(self, hosts, server, resolve_on_init)

This constructor initializes an instance of the DNSMatcher class.

Arguments of __init__

hosts (complex)
Default: n/a
Hostnames to resolve.

resolve_on_init (boolean)
Default: FALSE
Resolve all hostnames on startup time. Otherwise, names will be resolved on-demand.

server (string)
Default: None
IP address of the DNS server to query. Defaults to the servers set in the `resolv.conf` file.

5.8.5. Class MatcherPolicy

Matcher policies can be used to find out if a parameter is included in a list, or which elements of a list correspond to a certain parameter), and influence the behavior of the proxy class based on the results. Matchers can be used for a wide range of tasks, for example, to determine if the particular IP address or URL that a client is trying to access is on a black or whitelist, or to verify that a particular e-mail address is valid.

MatcherPolicy instances are reusable matchers that contain configured instances of the matcher classes (e.g., DNSMatcher, RegexpMatcher). For examples, see the specific matcher classes.

5.8.6. Class RegexpFileMatcher

This class is similar to *RegexpMatcher*, but stores the regular expressions to match and ignore in files. For example, this class can be used for URL filtering. The matcher itself stores only the paths and the filenames to the lists. The file is automatically monitored and reloaded when it is modified. Searches are case-insensitive.

Example 5.23. RegexpFileMatcher example

```
MatcherPolicy(name="demo_regexpfilematcher",
matcher=RegexpFileMatcher(match_fname="/tmp/match_list.txt", ignore_fname="/tmp/ignore_list.txt"))
```

5.8.6.1. Attributes of RegexpFileMatcher

ignore_date (unknown)
Default: n/a
Date (in unix timestamp format) when the *`ignore_file`* was loaded.

ignore_file (unknown)
Default: n/a
Name of the file storing the patterns to ignore.

match_date (unknown)
Default: n/a
Date (in unix timestamp format) when the *`match_file`* was loaded.

match_file (unknown)
Default: n/a
Name of the file storing the patterns for positive matches.

5.8.6.2. RegexpFileMatcher methods

Method	Description
__init__(self, match_fname, ignore_fname)	Constructor to initialize a RegexpFileMatcher instance.

Table 5.69. Method summary

Method __init__(self, match_fname, ignore_fname)

This constructor initializes an instance of the RegexpFileMatcher class.

Arguments of __init__

ignore_fname (filename)
Default: None
Name of the file storing the patterns to ignore.

match_fname (filename)
Default: None
Name of the file storing the patterns for positive matches.

5.8.7. Class RegexpMatcher

A simple regular expression based matcher with a match and an ignore list. Searches are case-insensitive.

Example 5.24. RegexpMatcher example
The following RegexpMatcher matches only the *`smtp.example.com`* string.

```
MatcherPolicy(name="Smtpdomains", matcher=RegexpMatcher (match_list=("smtp.example.com",),
ignore_list=None))
```

5.8.7.1. Attributes of RegexpMatcher

ignore (unknown)
Default: n/a
A list of compiled regular expressions defining the strings to be ignored even if `match` resulted in a positive match.

match (unknown)
Default: n/a
A list of compiled regular expressions which result in a positive match.

5.8.7.2. RegexpMatcher methods

Method	**Description**
__init__(self, match_list, ignore_list, ignore_case)	Constructor to initialize a RegexpMatcher instance.

Table 5.70. Method summary

Method __init__(self, match_list, ignore_list, ignore_case)

This constructor initializes a RegexpMatcher instance by setting the `match` and `ignore` attributes to an empty list.

Arguments of __init__

ignore_list (filename)
Default: None
The list of regular expressions to ignore.

match_list (filename)
Default: None
The list of regular expressions to match.

5.8.8. Class SmtpInvalidRecipientMatcher

This class encapsulates a VRFY/RCPT based validity checker to transparently verify the existance of E-mail addresses. Instead of immediately sending the e-mail to the recipient SMTP server, an independent SMTP server is queuried about the existance of the recipient e-mail address.

Instances of this class can be referred to in the `recipient_matcher` attribute of the *SmtpProxy* class. The SmtpProxy will automatically reject unknown recipients even if the recipient SMTP server would accept them.

Example 5.25. SmtpInvalidMatcher example

```
Python:
class SmtpRecipientMatcherProxy(SmtpProxy):
recipient_matcher="SmtpCheckrecipient"
def config(self):
super(SmtpRecipientMatcherProxy, self).config()

MatcherPolicy(name="SmtpCheckrecipient", matcher=SmtpInvalidRecipientMatcher (server_port=25,
cache_timeout=60, force_delivery_attempt=FALSE, server_name="recipientcheck.example.com"))
```

5.8.8.1. SmtpInvalidRecipientMatcher methods

Method	Description
__init__(self, server_name, server_port, cache_timeout, force_delivery_attempt, sender_address, bind_name)	

Table 5.71. Method summary

Method __init__(self, server_name, server_port, cache_timeout, force_delivery_attempt, sender_address, bind_name)

Arguments of __init__

bind_name (string)
Default: ""
Specifies the hostname to bind to before initiating the connection to the SMTP server.

cache_timeout (integer)
Default: 60
How long will the result of an address verification be retained (in seconds).

force_delivery_attempt (boolean)
Default: FALSE
Force a delivery attempt even if the autodetection code otherwise would use VRFY. Useful if the server always returns success for VRFY.

sender_address (string)
Default: "<>"
This value will be used as the mail sender for the attempted mail delivery. Mail delivery is attempted if the *force_delivery_attempt* is TRUE, or the recipient server does not support the VRFY command.

server_name (string)
Default: n/a
Domain name of the SMTP server that will verify the addresses.

server_port (integer)
Default: 25
Port of the target server.

5.8.9. Class WindowsUpdateMatcher

WindowsUpdateMatcher is actually a DNSMatcher used to retrieve the IP addresses currently associated with the `v5.windowsupdate.microsoft.nsatc.net`, `v4.windowsupdate.microsoft.nsatc.net`, and `update.microsoft.nsatc.net` domain names from the specified name server. Windows Update is running on a distributed server farm, using the DNS round robin method and a short TTL to constantly change the set of servers currently visible, consequently the IP addresses of the servers are constantly changing.

Example 5.26. WindowsUpdateMatcher example

```
MatcherPolicy(name="demo_windowsupdatematcher", matcher=WindowsUpdateMatcher())
```

5.8.9.1. WindowsUpdateMatcher methods

Method	Description
__init__(self, server)	Constructor to initialize an instance of the WindowsUpdateMatcher class.

Table 5.72. Method summary

Method __init__(self, server)

This constructor initializes an instance of the WindowsUpdateMatcher class.

Arguments of __init__

server (string)
Default: None
The IP address of the name server to query.

5.9. Module NAT

Network Address Translation (NAT) is a technology that can be used to change source or destination addresses in a connection from one IP address to another one. This module defines the classes performing the translation for IP addresses.

Several different NAT methods are supported using different NAT classes, like *GeneralNAT* or *StaticNAT*. To actually perform network address translation in a service, you have to use a *NATPolicy* instance that contains a configured NAT class. NAT policies provide a way to re-use NAT instances whithout having to define NAT mappings for each service individually.

5.9.1. Classes in the NAT module

Class	Description
AbstractNAT	Class encapsulating the abstract NAT interface.
FWMark	Helper class to create packet mark value from gateway index number
GeneralNAT	Class encapsulating a general subnet-to-subnet NAT.
HashNAT	Class which sets the address from a hash table.
LinkAvailabilityPFNat	Class encapsulating a general subnet-to-subnet NAT.
NAT46	Class that performs translation from IPv4 to IPv6 addresses (NAT46)
NAT64	Class that performs translation from IPv6 to IPv4 addresses (NAT64)
NATPolicy	Class encapsulating named NAT instances.
RandomNAT	Class generating a random IP address.
StaticNAT	Class that replaces the source or destination address with a predefined address.

Table 5.73. Classes of the NAT module

5.9.2. Class AbstractNAT

This class encapsulates an interface for application level network address translation (NAT). This NAT is different from the NAT used by packet filters: it modifies the outgoing source/destination addresses just before Vela connects to the server.

Source and destination NATs can be specified when a *Service* is created.

The NAT settings are used by the *ConnectChainer* class just before connecting to the server.

5.9.2.1. AbstractNAT methods

Method	Description
__init__(self)	Constructor to initialize an AbstractNAT instance.

Method	**Description**
performTranslation(self, session, addrs, nat_type)	Function that performs the address translation.

Table 5.74. Method summary

Method __init__(self)

This constructor initializes an AbstractNAT instance. Currently it does nothing, but serves as a placeholder for future extensions.

Method performTranslation(self, session, addrs, nat_type)

This function is called before connecting a session to the destination server. The function returns the address (a *SockAddr* instance) to bind to before establishing the connection.

Arguments of performTranslation

addrs (unknown)
Default: n/a
tuple of (source, destination) address, any of them can be none in case of the other translation

nat_type (unknown)
Default: n/a
translation type, either NAT_SNAT or NAT_DNAT

session (unknown)
Default: n/a
Session which is about to connect the server.

5.9.3. Class FWMark

5.9.3.1. FWMark methods

Method	Description
__init__(self, gw_mark)	Constructor to initialize an FWMark instance.

Table 5.75. Method summary

Method __init__(self, gw_mark)

Arguments of __init__

gw_mark (integer)
Default: n/a
Index of gateway where packets routed to in advanced routing mode. 0 means default gateway in main routing table.

5.9.4. Class GeneralNAT

This class encapsulates a general subnet-to-subnet NAT. It requires a list of `from, to, translated to` parameters:

- *from*: the source address of the connection.
- *to*: the destination address of the connection.
- *translated to*: the translated address.

If the NAT policy is used as SNAT, the translated address is used to translate the source address of the connection; if the NAT policy is used as DNAT, the translated address is used to translate the destination address of the connection. The translation occurs according to the first matching rule.

Example 5.27. GeneralNat example

The following example defines a simple GeneralNAT policy that maps connections coming from the *192.168.1.0/24* subnet and targeting the *192.168.10.0/24* subnet into the *10.70.0.0/24* subnet.

```
NATPolicy(name="Demo_GeneralNAT", nat=GeneralNAT(mapping=((InetSubnet("192.168.1.0/24"),
InetSubnet("192.168.10.0/24"), InetSubnet("10.70.0.0/24")),)))
```

If the policy is used as SNAT, the *192.168.1.0/24* subnet is translated into the *10.70.0.0/24* subnet and used as the source address of the connection. If the policy is used as DNAT, the *192.168.10.0/24* subnet is translated into the *10.70.0.0/24* subnet and used as the target address of the connection.

5.9.4.1. GeneralNAT methods

Method	Description
__init__(self, mapping)	Constructor to initialize a GeneralNAT instance.

Table 5.76. Method summary

Method __init__(self, mapping)

This constructor initializes a GeneralNAT instance.

Arguments of __init__

mapping (complex)
Default: n/a
List of tuples of InetSubnets in (source domain, destination domain, mapped domain) format.

5.9.5. Class HashNAT

HashNAT statically maps an IP address to another using a hash table. The table is indexed by the source IP address, and the value is the translated IP address. Both IP addresses are stored in string format.

5.9.5.1. HashNAT methods

Method	Description
__init__(self, ip_hash, default_reject)	Constructor to initialize a HashNAT instance.

Table 5.77. Method summary

Method __init__(self, ip_hash, default_reject)

This constructor initializes a HashNAT instance.

Arguments of __init__

default_reject (boolean)
Default: TRUE
Enable this parameter to reject all connections outside the specific source range.

ip_hash (complex)
Default: n/a
The hash storing the IP address.

5.9.6. Class LinkAvailabilityPFNat

This class encapsulates a subnet-to-subnet NAT, which is usable in PFService only, and limited to SNAT. It requires a list of `from, to, translated to, fwmark` parameters:

- *from*: the source address of the connection.
- *to*: the destination address of the connection.
- *translated to*: the translated address.
- *fwmark*: mark the packets of the traffic.

The NAT policy could be only used as SNAT, the translated address is used to translate the source address of the connection. The translation occurs according to the first matching rule. The translation happens in POSTROUTING mangle chain. This NAT adds an extra 4 bit FWMARK to the traffic, additionally to the PNS MARK bits. This can be used for advanced routing by FWMARK.

5.9.6.1. LinkAvailabilityPFNat methods

Method	Description
__init__(self, mapping)	Constructor to initialize a LinkAvailabilityPFNat instance.

Table 5.78. Method summary

Method __init__(self, mapping)

This constructor initializes a LinkAvailabilityPFNat instance.

Arguments of __init__

mapping (complex)
Default: n/a
List of tuples in (source domain, destination domain, mapped domain, fwmark) format.

5.9.7. Class NAT46

NAT46 embeds and IPv4 address into a specific portion of the IPv6 address according to the NAT46 specification as described in RFC6052 (http://tools.ietf.org/html/rfc6052#section-2.2).

5.9.7.1. NAT46 methods

Method	Description
__init__(self, prefix, prefix_mask, suffix)	Constructor to initialize a NAT46 instance.

Table 5.79. Method summary

Method __init__(self, prefix, prefix_mask, suffix)

This constructor initializes a NAT46 instance.

Arguments of __init__

prefix (string)
Default: "64:ff9b::"
This parameter specifies the common leading part of the IPv6 address that the IPv4 address should map into. Bits that exceed the mask will be overwritten by the mapping.

prefix_mask (integer)
Default: 96
This parameter specifies the position to embed the IPv4 address to and must be one of 32, 40, 48, 56, 64, or 96.

suffix (string)
Default: "::"
This parameter specifies the common trailing part of the IPv6 address that the IPv4 address should map into. The length of the suffix must not exceed the empty bit count determined by the configured prefix mask.

5.9.8. Class NAT64

NAT64 maps specific bits of the IPv6 address to IPv4 addresses according to the NAT64 specification as described in RFC6052 (http://tools.ietf.org/html/rfc6052#section-2.2).

5.9.8.1. NAT64 methods

Method	Description
__init__(self, prefix_mask)	Constructor to initialize a NAT64 instance.

Table 5.80. Method summary

Method __init__(self, prefix_mask)

This constructor initializes a NAT64 instance.

Arguments of __init__

prefix_mask (integer)
Default: 96
This parameter specifies the length of the IPv6 address to consider and must be one of 32, 40, 48, 56, 64, or 96.

5.9.9. Class NATPolicy

This class encapsulates a name and an associated NAT instance. NAT policies provide a way to re-use NAT instances whithout having to define NAT mappings for each service individually.

Example 5.28. Using Natpolicies
The following example defines a simple NAT policy, and uses this policy for SNAT in a service.

```
NATPolicy(name="demo_natpolicy", nat=GeneralNAT(mapping=((InetSubnet(addr="10.0.1.0/24"),
InetSubnet(addr="192.168.1.0/24")),)))

Service(name="office_http_inter", proxy_class=HttpProxy, snat_policy="demo_natpolicy")
```

5.9.9.1. NATPolicy methods

Method	**Description**
__init__(self, name, nat, cacheable)	Constructor to initialize a NAT policy.

Table 5.81. Method summary

Method __init__(self, name, nat, cacheable)

This contructor initializes a NAT policy.

Arguments of __init__

cacheable (boolean)
Default: TRUE
Enable this parameter to cache the NAT decisions.

name (string)
Default: n/a
Name identifying the NAT policy.

nat (class)
Default: n/a

nat (class)
NAT object which performs address translation.

5.9.10. Class RandomNAT

This class randomly selects an address from a list of IP addresses. This can be used for load-balancing several lines by binding each session to a different interface.

5.9.10.1. RandomNAT methods

Method	Description
__init__(self, addresses)	Constructor to initialize a RandomNAT instance.

Table 5.82. Method summary

Method __init__(self, addresses)

This constructor initializes a RandomNAT instance.

Arguments of __init__

addresses (complex)
Default: n/a
List of the available interfaces. Each item of the list must be am instance of the `SockAddr` (or a derived) class.

5.9.11. Class StaticNAT

This class assigns a predefined value to the address of the connection.

5.9.11.1. StaticNAT methods

Method	Description
__init__(self, addr)	Constructor to initialize a StaticNAT instance.

Table 5.83. Method summary

Method __init__(self, addr)

This constructor initializes a StaticNAT instance.

Arguments of __init__

addr (sockaddr)
Default: n/a
The address that replaces all addresses.

5.10. Module Notification

5.10.1. Classes in the Notification module

Class	Description
AbstractNotificationMethod	Class encapsulating the abstract notification method.
EmailNotificationMethod	Class sending out notifications in e-mail.
NotificationPolicy	Class encapsulating a NotificationPolicy which describes how to send out notifications.

Table 5.84. Classes of the Notification module

5.10.2. Class AbstractNotificationMethod

This abstract class encapsulates a notification that is performed when a certain event occurs.

Specialized classes can be derived from AbstractNotification, such as the *EmailNotificationMethod* class.

5.10.3. Class EmailNotificationMethod

This class encapsulates a notification handler that sends an e-mail with the given mail properties.

5.10.3.1. Attributes of EmailNotificationMethod

recipient (string)
Default: n/a
The e-mail address of the recipient.

5.10.3.2. EmailNotificationMethod methods

Method	Description
__init__(self, recipient)	Constructor to initialize an EmailNotification instance.

Table 5.85. Method summary

Method __init__(self, recipient)

This constructor initializes an EmailNotification instance and sets the attributes of the outgoing e-mail.

Arguments of __init__

recipient (string)
Default: n/a
The e-mail address of the recipient.

5.10.4. Class NotificationPolicy

5.11. Module Proxy

This module encapsulates the Proxy component. The Proxy module provides a common framework for protocol-specific proxies, implementing the functions that are used by all proxies. Protocol-specific proxy modules are derived from the Proxy module, and are described in *Chapter 4, Proxies (p. 34)*.

5.11.1. Functions in module Proxy

Function	Description
proxyLog	Function to send a proxy-specific message to the system log.

Table 5.86. Function summary

5.11.2. Classes in the Proxy module

Class	Description
Proxy	Class encapsulating the abstract proxy.

Table 5.87. Classes of the Proxy module

5.11.3. Functions

5.11.3.1. Function proxyLog(self, type, level, msg, args)

This function sends a message into the system log. All messages start with the `session_id` that uniquely identifies the connection.

Arguments of proxyLog

level (integer)
Default: n/a
Verbosity level of the log message.

msg (string)
Default: n/a
The text of the log message.

type (string)
Default: n/a
The class of the log message.

5.11.4. Class Proxy

This class serves as the abstact base class for all proxies. When an instance of the Proxy class is created, it loads and starts a protocol-specific proxy. Proxies operate in their own threads, so this constructor returns immediately.

5.11.4.1. Attributes of Proxy

encryption_policy (class)
Default: None
Name of the Encryption policy instance used to encrypt the sessions and verify the certificates used. For details, see *Section 5.5, Module Encryption (p. 195)*.

ids_policy (class)
Default: None
Name of the Ids policy instance used to send traffic to Intrusion Detection Systems. For details, see *Section 5.6, Module Ids (p. 239)*.

language (string)
Default: "en"
Determines the language used for user-visible error messages. Supported languages: *en* - English; *de* - German; *hu* - Hungarian.

5.11.4.2. Proxy methods

Method	**Description**
closedByAbort(self)	Function called by the proxy core when an abort has been occured.
config(self)	Function called by the proxy core to initialize the proxy instance.
connectServer(self)	Function called by the proxy instance to establish the server-side connection.

Method	Description
getCredentials(self, method, username, domain, target, port)	Function called when proxy requires credentials for server side authentication.
invalidPolicyCall(self)	Invalid policy function called.
setServerAddress(self, host, port)	Function called by the proxy instance to set the address of the destination server.
setServerSideEncryption(self)	Function called by the proxy instance to set up the server side encryption parameters dynamically.
userAuthenticated(self, entity, groups, auth_info)	Function called when inband authentication is successful.

Table 5.88. Method summary

Method closedByAbort(self)

This function is called when a callback gives abort or no result. It simply sets a flag that will be used for logging the reason of the proxy's ending.

Method config(self)

This function is called during proxy startup. It sets the attributes of the proxy instance according to the configuration of the proxy.

Method connectServer(self)

This function is called to establish the server-side connection. The function either connects a proxy to the destination server, or an embedded proxy to its parent proxy. The proxy may set the address of the destination server using the `setServerAddress` function.

The `connectServer` function calls the chainer specified in the service definition to connect to the remote server using the host name and port parameters.

The `connectServer` function returns the descriptor of the server-side data stream.

Method getCredentials(self, method, username, domain, target, port)

The proxy instance calls this function to retrieve authentication credentials for authentication method *`method`* and the target user *`username`*.

Arguments of getCredentials

domain (string)
Default: n/a
Domain the user name belongs to.

method (string)
Default: n/a
Method that will be used for authentication on target server.

port (integer)
Default: n/a
Target server port.

target (string)
Default: n/a
Target server hostname.

username (string)
Default: n/a
Username that will be used for authentication on target server.

Method invalidPolicyCall(self)

This function is called when invalid policy function has been called.

Method setServerAddress(self, host, port)

The proxy instance calls this function to set the address of the destination server. This function attempts to resolve the hostname of the server using the DNS; the result is stored in the *`session.server_address`* parameter. The address of the server may be modified later by the router of the service. See *Section 5.13, Module Router (p. 266)* for details.

Note
The *`setServerAddress`* function has effect only when *InbandRouter* is used.

Arguments of setServerAddress

host (string)
Default: n/a
The host name of the server.

port (integer)
Default: n/a

port (integer)
The Port number of the server.

Method setServerSideEncryption(self)

Function called by the proxy instance when the encryption scenario is dynamic (eg.: DynamicServerEncryption) to set up the server side encryption parameters. It should return with a DynamicServerEncryptionServerParams if DynamicServerEncryption scenario used otherwise with None.

This method unconditionally raises a NotImplementedError exception to indicate that it must be overridden by descendant classes like 'Proxy'.

Method userAuthenticated(self, entity, groups, auth_info)

The proxy instance calls this function to indicate that the inband authentication was successfully performed. The name of the client is stored in the `entity` parameter.

Arguments of userAuthenticated

entity (unknown)
Default: n/a
Username of the authenticated client.

5.12. Module Resolver

This module defines the AbstractResolver interface and various derived classes to perform name lookups.

5.12.1. Classes in the Resolver module

Class	Description
AbstractResolver	Class encapsulating the abstract Resolver interface.
DNSResolver	Class encapsulating DNS-based name resolution.
HashResolver	Class encapsulating hash-based name resolution.

Table 5.89. Classes of the Resolver module

5.12.2. Class AbstractResolver

This class encapsulates an interface for application level name resolution.

5.12.3. Class DNSResolver

DNSResolver policies query the domain name server to resolve domain names.

Example 5.29. A simple DNSResolver policy
Below is a simple DNSResolver policy enabled to return multiple 'A' and 'AAAA' records from the nameserver 1.1.1.1 with 2s timeout.

```
ResolverPolicy(name="Mailservers", resolver=DNSResolver(name_server='1.1.1.1', timeout=2))
```

5.12.3.1. DNSResolver methods

Method	Description
__init__(self, name_server, timeout, use_search_domain)	Constructor to initialize a DNSResolver instance.

Table 5.90. Method summary

Method __init__(self, name_server, timeout, use_search_domain)

This constructor initializes a DNSResolver instance.

Arguments of __init__

name_server (string)
Default: None
IP address of the DNS server to query. Defaults to the servers set in the `resolv.conf` file.

timeout (integer)
Default: 2
Seconds to wait a response from a server.

use_search_domain (boolean)
Default: FALSE
Append the host's search domain to the query.

5.12.4. Class HashResolver

HashResolver policies are used to locally store the IP addresses belonging to a domain name. A domain name (Hostname) and one or more corresponding IP addresses (Addresses) can be stored in a hash. If the domain name to be resolved is not included in the hash, the name resolution will fail. The HashResolver can be used to direct incoming connections to specific servers based on the target domain name.

Example 5.30. A simple HashResolver policy
The resolver policy below associates the IP addresses *`192.168.1.12`* and *`192.168.1.13`* with the *`mail.example.com`* domain name.

```
ResolverPolicy(name="DMZ",    resolver=HashResolver(mapping={"mail.example.com":   ("192.168.1.12",
 "192.168.1.13")}))
```

5.12.4.1. HashResolver methods

Method	Description
__init__(self, mapping)	Constructor to initialize a HashResolver instance.

Table 5.91. Method summary

Method __init__(self, mapping)

This constructor initializes a HashResolver instance.

Arguments of __init__

mapping (complex)
Default: n/a
Mapping that describes hostname->IP address pairs.

5.13. Module Router

Routers define the target IP address and port of the destination server, based on information that is available before started. The simplest router (*DirectedRouter*) selects a preset destination as the server address, while the most commonly used *TransparentRouter* connects to the IP address requested by the client. Other routers may make more complex decisions. The destination address selected by the router may be overridden by the proxy and the DNAT classes used in the service.

5.13.1. The source address used in the server-side connection

Routers also define source address and port of the server-side connection. This is the IP address that is used to connect the server. The server sees that the connection originates from this address. The following two parameters determine the source address used in the server-side connection:

`forge_addr`: If set to *`TRUE`*, the client's source address is used as the source of the server-side connection. Otherwise, the IP address of the interface connected to the server is used.

`forge_port`: This parameter defines the source port that is used in the server-side connection. Specify a port number as an integer value, or use one of the following options:

Name	Description
V_PORT_ANY	Selected a random port between *`1024`* and *`65535`*. This is the default behavior of every router.
V_PORT_GROUP	Select a random port in the same group as the port used by the client. The following groups are defined: *`0-513`*, *`514-1024`*, *`1025-`*.
V_PORT_EXACT	Use the same port as the client.

Name	Description
V_PORT_RANDOM	Select a random port using a cryptographically secure function.

Table 5.92. Options defining the source port of the server-side connection

5.13.2. Classes in the Router module

Class	Description
AbstractRouter	Class encapsulating the abstract router.
DirectedRouter	Class encapsulating a Router which explicitly defines the target address.
InbandRouter	Class encapsulating the Router which extracts the destination address from the application-level protocol.
TransparentRouter	Class encapsulating a Router which provides transparent services.

Table 5.93. Classes of the Router module

5.13.3. Class AbstractRouter

AbstractRouter implements an abstract router that determines the destination address of the server-side connection. Service definitions should refer to a customized class derived from AbstractRouter, or one of the predefined router classes, such as *TransparentRouter* or *DirectedRouter*. Different implementations of this interface perform Transparent routing (directing the client to its original destination), and Directed routing (directing the client to a given destination).

A proxy can override the destination selected by the router using the the *setServerAddress* method.

5.13.3.1. Attributes of AbstractRouter

forge_addr (boolean)
Default: n/a
If set to *TRUE*, the client's source address is used as the source of the server-side connection.

forge_port (unknown)
Default: n/a
Defines the source port that is used in the server-side connection. See *Section 5.13.1, The source address used in the server-side connection (p. 266)* for details.

5.13.4. Class DirectedRouter

This class implements directed routing, which means that the destination address is a preset address for each session.

Example 5.31. DirectedRouter example

The following service uses a DirectedRouter that redirects every connection to the `/var/sample.socket` Unix domain socket.

```
Service(name="demo_service", proxy_class=HttpProxy,
router=DirectedRouter(dest_addr=SockAddrUnix('/var/sample.socket'), overrideable=FALSE,
forge_addr=FALSE))
```

The following service uses a DirectedRouter that redirects every connection to the *`192.168.2.24:8080`* IP address.

```
Service(name="demo_service", proxy_class=HttpProxy,
router=DirectedRouter(dest_addr=SockAddrInet('192.168.2.24', 8080), overrideable=FALSE,
forge_addr=FALSE))
```

5.13.4.1. Attributes of DirectedRouter

dest_addr (unknown)
Default: n/a
The destination address to connect to.

5.13.4.2. DirectedRouter methods

Method	Description
__init__(self, dest_addr, forge_addr, overrideable, forge_port)	Constructor to initialize a DirectedRouter.

Table 5.94. Method summary

Method __init__(self, dest_addr, forge_addr, overrideable, forge_port)

This constructor initializes an instance of the DirectedRouter class.

Arguments of __init__

dest_addr (complex)
Default: n/a
The destination address to connect to.

forge_addr (boolean)
Default: FALSE
If set to *`TRUE`*, the client's source address is used as the source of the server-side connection.

forge_port (complex)
Default: V_PORT_ANY
Defines the source port that is used in the server-side connection. See *Section 5.13.1, The source address used in the server-side connection (p. 266)* for details.

overrideable (boolean)
Default: FALSE
If set to *TRUE*, the proxy may override the selected destination. Enable this option when the proxy builds multiple connections to the destination server, and the proxy knows the address of the destination server, for example, because it receives a redirect request. This situation is typical for the SQLNet proxy.

5.13.5. Class InbandRouter

This class implements inband routing, which means that the destination address will be determined by the protocol. Inband routing works only for protocols that can send routing information within the protocol, and is mainly used for non-transparent proxying. The InbandRouter class currently supports only the HTTP and FTP protocols.

Example 5.32. InbandRouter example
The following service uses an InbandRouter to extract the destination from the protocol.

```
Service(name="demo_service", proxy_class=HttpProxy, router=InbandRouter(forge_addr=FALSE))
```

5.13.5.1. InbandRouter methods

Method	Description
__init__(self, forge_addr, forge_port)	Constructor to initialize a InbandRouter.

Table 5.95. Method summary

Method __init__(self, forge_addr, forge_port)

This constructor initializes an instance of the InbandRouter class.

Arguments of __init__

forge_addr (boolean)
Default: FALSE
If set to *TRUE*, the client's source address is used as the source of the server-side connection.

forge_port (complex)
Default: V_PORT_ANY
Defines the source port that is used in the server-side connection. See *Section 5.13.1, The source address used in the server-side connection (p. 266)* for details.

5.13.6. Class TransparentRouter

This class implements transparent routing, which means that the destination server is the original destination requested by the client.

Example 5.33. TransparentRouter example
The following service uses a TransparentRouter that connects to the *8080* port of the server and uses the client's IP address as the source of the server-side connection.

```
Service(name="demo_service", proxy_class=HttpProxy, router=TransparentRouter(forced_port=8080,
overrideable=FALSE, forge_addr=TRUE))
```

5.13.6.1. Attributes of TransparentRouter

forced_port (unknown)
Default: n/a
Defines the source port that is used in the server-side connection. See *Section 5.13.1, The source address used in the server-side connection (p. 266)* for details.

forge_addr (unknown)
Default: n/a
If set to *TRUE*, the client's source address is used as the source of the server-side connection.

5.13.6.2. TransparentRouter methods

Method	Description
__init__(self, forced_port, forge_addr, overrideable, forge_port)	Constructor to initialize an instance of the TransparentRouter class.

Table 5.96. Method summary

Method __init__(self, forced_port, forge_addr, overrideable, forge_port)

This constructor creates a new TransparentRouter instance which can be associated with a *Service*.

Arguments of __init__

forced_port (integer)
Default: 0
Modify the destination port to this value. Default value: 0 (do not modify the target port)

forge_addr (boolean)
Default: FALSE
If set to *TRUE*, the client's source address is used as the source of the server-side connection.

forge_port (complex)
Default: V_PORT_ANY
Defines the source port that is used in the server-side connection. See *Section 5.13.1, The source address used in the server-side connection (p. 266)* for details.

overrideable (boolean)
Default: FALSE
If set to *TRUE*, the proxy may override the selected destination. Enable this option when the proxy builds multiple connections to the destination server, and the proxy knows the address of the destination server, for example, because it receives a redirect request. This situation is typical for the SQLNet proxy.

5.14. Module Rule

The Rule module defines the classes needed to create firewall rules.

5.14.1. Evaluating firewall rules

When Application-level Gateway receives a connection request from a client, it tries to select a rule matching the parameters of the connection. The following parameters are considered.

Name in MC	Name in policy.py
VPN	*reqid*
Source Interface	*src_iface*
Source Interface Group	*src_ifgroup*
Protocol	*proto*
Source Port	*src_port*
Destination Port	*dst_port*
Source Subnet	*src_subnet*

Name in MC	Name in policy.py
Source Zone	*src_zone*
Destination Subnet	*dst_subnet*
Destination Interface	*dst_iface*
Destination Interface Group	*dst_ifgroup*
Destination Zone	*dst_zone*

Table 5.97. Evaluated Rule parameters

Application-level Gateway selects the rule that most specifically matches the connection. Selecting the most specific rule is based on the following method.

- The order of the rules is not important.
- The parameters of the connection act as filters: if you do not set any parameters, the rule will match any connection.
- If multiple connections would match a connection, the rule with the most-specific match is selected. For example, you have configured two rules: the first has the *Source Zone* parameter set as the *office* (which is a zone covering all of your client IP addresses), the second has the *Source Subnet* parameter set as *192.168.15.15/32*. The other parameters of the rules are the same. If a connection request arrives from the *192.168.15.15/32* address, Application-level Gateway will select the second rule. The first rule will match every other client request.
- Application-level Gateway considers the parameters of a connection in groups. The first group is the least-specific, the last one is the most-specific. The parameter groups are listed below.
- The parameter groups are linked with a logical AND operator: if parameters of multiple groups are set in a rule, the connection request must match a parameter of every group. For example, if both the *Source Interface* and *Destination Port* are set, the connection must match both parameters.
- Parameters within the same group are linked with a logical OR operator: if multiple parameters of a group are set for a rule, the connection must match any one of the parameters. If there are multiple similar rules, the rule with the most specific parameter match for the connection will be selected.

Note
In general, avoid using multiple parameters of the same group in one rule, as it may lead to undesired side-effects. Use only the most specific parameter matching your requirements.

For example, suppose that you have a rule with the *Destination Zone* parameter set, and you want to create a similar rule for a specific subnet of this zone. In this case, create a new rule with the *Destination Subnet* parameter set, do not set the *Destination Zone* parameter in both rules. Setting the *Destination Zone* parameter in both rules and setting the *Destination Subnet* parameter in the second rule would work for connections targeting the specified subnet, but it would cause Application-level Gateway to reject the connections that target other subnets of the specified destination zone, because both rules would match for the connection.

- The parameter groups are the following from the least specific to the most specific ones. Parameters within each group are listed from left to right from the least specific to the most specific ones.

1. *`Destination Zone`* > *`Destination Interface Group`* > *`Destination Interface`* > *`Destination Subnet`*
2. *`Source Zone`* > *`Source Subnet`*
3. *`Destination Port`* (Note that port is more specific than port range.)
4. *`Source Port`* (Note that port is more specific than port range.)
5. *`Protocol`*
6. *`Source Interface Group`* > *`Source Interface`* > *`VPN`*

- If no matching rule is found, Application-level Gateway rejects the connection.

Note
It is possible to create rules that are very similar, making debugging difficult.

5.14.2. Sample rules

Example 5.34. Sample rule definitions
The following rule starts the service called *`MyPFService`* for every incoming TCP connection (*`proto=6`*).

```
Rule(proto=6,
    service='MyPFService'
    )
```

The following rule starts a service for TCP or UDP connections from the *`office`* zone.

```
Rule(proto=(6,17),
    src_zone='office',
    service='MyService'
    )
```

The following rule permits connections from the *`192.168.0.0/16`* IPv4 and the *`2001:db8:c001:ba80::/58`* IPv6 subnets. Note that since the *`src_subnet`* parameter has two values, they are specified as a Python tuple: *`('value1','value2')`*.

```
Rule(proto=6,
    src_subnet=('192.168.0.0/16', '2001:db8:c001:ba80::/58'),
    service='MyService'
    )
```

The following rule has almost every parameter set:

```
Rule(src_iface=('eth0', ),
    proto=6,
    dst_port=443,
    src_subnet=('192.168.10.0/24', ),
    src_zone=('office', ),
    dst_subnet=('192.168.50.50/32', ),
    dst_zone=('finance', ),
    service='MyHttpsService'
    )
```

5.14.3. Adding metadata to rules: tags and description

To make the configuration file more readable and informative, you can add descriptions and tags to the rules. Descriptions can be longer texts, while tags are simple labels, for example, to identify rules that belong to the same type of traffic. Adding metadata to rules is not necessary, but can be a great help when maintaining large configurations.

- To add a description to a rule, add the text of the description before the rule, enclosed between three double-quotes:

  ```
  """This rule is ..."""
  ```

- To tag a rule, add a comment line before the rule that contains the list of tags applicable to the rule, separated with commas.

  ```
  #Tags: tag1, tag2
  ```

Example 5.35. Tagging rules
The following rule has two tags, marking the traffic type and the source zone: `http` and `office`.

```
#Tags: http, office
    """Description"""
    Rule(proto=(6),
    src_zone='office',
    service='MyHttpService'
    )
```

5.14.4. Classes in the Rule module

Class	Description
PortRange	Specifies a port range for a rule
Rule	This class implements firewall rules

Table 5.98. Classes of the Rule module

5.14.5. Class PortRange

This class specifies a port range for a firewall rule. It can be used in the *src_port* and *dst_port* parameters of a rule. For example: *src_port=PortRange(2000, 2100)*, or *src_port=(PortRange(2000, 2100), PortRange(2500, 2600))*. When listing multiple elements, ports and port ranges can be mixed, for example: *src_port=(4433, PortRange(2000, 2100), PortRange(2500, 2600))*

5.14.5.1. Attributes of PortRange

high (integer)
Default: n/a
The higher value of the port range.

low (integer)
Default: n/a

low (integer)
The lower value of the port range.

5.14.6. Class Rule

This class implements firewall rules. For details, see *Section 5.14, Module Rule (p. 271)*.

5.14.6.1. Rule methods

Method	Description
*__init__(self, **kw)*	Initializes a rule

Table 5.99. Method summary

Method __init__(self, **kw)

Initializes a rule

Arguments of __init__

dst_iface (interface)
Default: n/a
Permit traffic only for connections that target a configured IP address of the listed interfaces. This parameter can be used to provide nontransparent service on an interface that received its IP address dynamically. For example, `dst_iface='eth0',` or `dst_iface=('eth0', 'tun1'),`.

dst_port (integer)
Default: n/a
Permit traffic only if the client targets the listed port. For example, `dst_port=80`, or `dst_port=(80, 443)`. To specify port ranges, use the *PortRange* class, for example, `dst_port=PortRange(2000, 2100)`.

dst_subnet (subnet)
Default: n/a
Permit traffic only for connections targeting a listed IP address, or an address belonging to the listed subnet. The subnet can be IPv4 or IPv6 subnet. When listing multiple subnets, you can list both IPv4 and IPv6 subnets. IP addresses are treated as subnets with a /32 (IPv4) or /128 (IPv6) netmask. If no netmask is set for a subnet, it is treated as a specific IP address. For example, `dst_subnet='192.168.10.16'` or `dst_subnet=('192.168.0.0/16', '2001:db8:c001:ba80::/58')`.

dst_zone (zone)
Default: n/a

dst_zone (zone)
Permit traffic only for connections targeting an address belonging to the listed zones. For example, `dst_zone='office'` or `dst_zone=('office', 'finance')`. Note that this applies to destination address of the client-side connection request: the actual address of the server-side connection can be different (for example, if a DirectedRouter is used in the service).

proto (integer)
Default: n/a
Permit only connections using the specified transport protocol. This is the transport layer (Layer 4) protocol of the OSI model, for example, TCP, UDP, ICMP, and so on. The protocol must be specified using a number: the decimal value of the "protocol" field of the IP header. This value is 6 for the TCP and 17 for the UDP protocol. For a list of protocol numbers, see the *Assigned Internet Protocol Numbers page of IANA*. For example: `proto=(6,17)`. To permit any protocol, do not add the `proto` parameter to the rule.

rule_id (integer)
Default: n/a
A unique ID number for the rule. This parameter is optional, an ID number is automatically generated for the rule during startup.

service (service)
Default: n/a
The name of the service to start for matching connections. This is the only required parameter for the rule, everything else is optional. For example, `service='MyService'`

src_iface (interface)
Default: n/a
Permit traffic only for connections received on the listed interface. For example, `src_iface='eth0',` or `src_iface=('eth0', 'tun1'),`.

src_port (integer)
Default: n/a
Permit traffic only if the client sends the connection request from the listed port. For example, `src_port=4455`. To specify port ranges, use the *PortRange* class, for example, `src_port=PortRange(2000, 2100)`.

src_subnet (subnet)
Default: n/a

src_subnet (subnet)
Permit traffic only for the clients of the listed subnet or IP addresses. The subnet can be IPv4 or IPv6 subnet. When listing multiple subnets, you can list both IPv4 and IPv6 subnets. IP addresses are treated as subnets with a /32 (IPv4) or /128 (IPv6) netmask. If no netmask is set for a subnet, it is treated as a specific IP address. For example, *src_subnet='192.168.10.16'* or *src_subnet=('192.168.0.0/16', '2001:db8:c001:ba80::/58')*.

src_zone (zone)
Default: n/a
Permit traffic only for the clients of the listed zones. For example, *src_zone='office'* or *src_zone=('office', 'finance')*.

5.15. Module Service

This module defines classes encapsulating service descriptions. The services define how the incoming connection requests are handled. When a connection is accepted by a *Rule*, the service specified in the Rule creates an instance of itself. This instance handles the connection, and proxies the traffic between the client and the server. It also handles TLS and SSL encryption of the traffic if needed, as configured in the *encryption_policy* parameter of the service. The instance of the selected service is created using the *'startInstance()'* method.

A service is not usable on its own, it needs a *Rule* to bind the service to a network interface of the firewall and activate it when a matching connection request is received. New instances of the service are started as the Rule accepts new connections.

5.15.1. Naming services

The name of the service must be a unique identifier; rules refer to this unique ID.

Use clear, informative, and consistent service names. Include the following information in the service name:

- Source zones, indicating which clients may use the service (e.g., *intranet*).
- The protocol permitted in the traffic (e.g., *HTTP*).
- Destination zones, indicating which servers may be accessed using the service (e.g., *Internet*).

Tip
Name the service that allows internal users to browse the Web *intra_HTTP_internet*. Use dots to indicate child zones, e.g., *intra.marketing_HTTP_inter*.

5.15.2. Classes in the Service module

Class	Description
AbstractService	Class encapsulating the abstract Service properties.

Class	Description
DenyService	DenyService prohibits access to certain services
PFService	Class encapsulating a packet-filter service definition.
Service	Class encapsulating a service definition.

Table 5.100. Classes of the Service module

5.15.3. Class AbstractService

AbstractService implements an abstract service. Service definitions should be based on a customized class derived from AbstractService, or on the predefined *Service* class.

5.15.3.1. Attributes of AbstractService

name (string)
Default: n/a
The name of the service.

5.15.3.2. AbstractService methods

Method	Description
__init__(self, name)	Constructor to initialize an instance of the AbstractService class.

Table 5.101. Method summary

Method __init__(self, name)

This constructor creates an AbstractService instance and sets the attributes of the instance according to the received arguments. It also registers the Service to the `services` hash so that rules can find the service instance.

Arguments of __init__

name (string)
Default: n/a
The name of the service.

5.15.4. Class DenyService

The DenyService class is a type of service that rejects connections with a predefined error code. DenyServices can be specified in the `service` parameter of *Rules*. If the rule referencing the DenyService matches a connection request, the connection is rejected. DenyService is a replacement for the obsolete Umbrella zone concept.

Example 5.36. A simple DenyService
The following defines a DenyService and a rule to reject all traffic that targets port 5555.

```
def demo() :
    DenyService(name='DenyService', ipv4_setting=DenyIPv4.DROP, ipv6_setting=DenyIPv6.DROP)
    Rule(dst_port=5555,
    service='DenyService'
    )
```

5.15.4.1. Attributes of DenyService

ipv4_setting (complex)
Default: n/a
Specifies how to reject IPv4 traffic. By default, the traffic is simply dropped without notifying the client (*`DenyIPv4.DROP`*). The following values are available: *`DenyIPv4.DROP`*, *`DenyIPv4.TCP_RESET`*, *`DenyIPv4.ICMP_NET_UNREACHABLE`*, *`DenyIPv4.ICMP_HOST_UNREACHABLE`*, *`DenyIPv4.ICMP_PROTO_UNREACHABLE`*, *`DenyIPv4.ICMP_PORT_UNREACHABLE`*, *`DenyIPv4.ICMP_NET_PROHIBITED`*, *`DenyIPv4.ICMP_HOST_PROHIBITED`*, *`DenyIPv4.ICMP_ADMIN_PROHIBITED`* **Note** When the *`DenyIPv4.TCP_RESET`* option is used, the TCP RESET packet is sent as if it was sent by the target server. When using an ICMP option, the appropriate ICMP packet is sent, just like a router would.

ipv6_setting (complex)
Default: n/a
Specifies how to reject IPv6 traffic. By default, the traffic is dropped without notifying the client (*`DenyIPv6.DROP`*). The following values are available: *`DenyIPv6.DROP`*, *`DenyIPv6.TCP_RESET`*, *`DenyIPv6.ICMP_NO_ROUTE`*, *`DenyIPv6.ICMP_ADMIN_PROHIBITED`*, *`DenyIPv6.ICMP_ADDR_UNREACHABLE`*, *`DenyIPv6.ICMP_PORT_UNREACHABLE`*

limit_policy (class)
Default: None
Name of the LimitPolicy instance used to rate limit the sessions.

name (string)
Default: n/a
The name of the service.

5.15.4.2. DenyService methods

Method	Description
__init__(self, name, logging, ipv4_setting, ipv6_setting, log_verbose, log_spec, limit_policy)	Constructor to initialize a DenyService instance.

Table 5.102. Method summary

Method __init__(self, name, logging, ipv4_setting, ipv6_setting, log_verbose, log_spec, limit_policy)

This constructor defines a DenyService with the specified parameters.

Arguments of __init__

limit_policy (class)
Default: None
Name of the LimitPolicy instance used to rate limit the sessions.

log_spec (string)
Default: None
Message filter expression.

log_verbose (integer)
Default: None
Default log verbosity level.

name (string)
Default: n/a
The name identifying the service.

5.15.5. Class PFService

PFServices allow you to replace the FORWARD rules of iptables, and configure application-level and packet-filter rules from Vela.

Note
The PFService class transfers packet-filter level services.

- To transfer connections on the packet-filter level, use the *PFService* class.
- To transfer connections on the application-level, use the *Service* class.

Example 5.37. PFService example

The following packet-filtering service transfers TCP connections that arrive to port *5555*.

```
PFService(name="intranet_PF5555_internet", router=TransparentRouter())
```

The following example defines a few classes: the client and server zones, a simple services, and a rule that starts the service.

```
Zone('internet', ['0.0.0.0/0'])
Zone('intranet', ['192.168.0.0/16'])

def demo() :
PFService(name="intranet_PF5555_internet", router=TransparentRouter())
Rule(dst_port=5555,
    src_zone='intranet',
    dst_zone='internet',
    service='PFService'
    )
```

5.15.5.1. Attributes of PFService

dnat_policy (class)
Default: n/a
Name of the NAT policy instance used to translate the destination addresses of the sessions. See *Section 5.9, Module NAT (p. 250)* for details.

limit_policy (class)
Default: None
Name of the LimitPolicy instance used to rate limit the sessions.

router (class)
Default: n/a
A router instance used to determine the destination address of the server. See *Section 5.13, Module Router (p. 266)* for details.

snat_policy (class)
Default: n/a
Name of the NAT policy instance used to translate the source addresses of the sessions. See *Section 5.9, Module NAT (p. 250)* for details.

5.15.5.2. PFService methods

Method	Description
__init__(self, name, router, snat_policy, dnat_policy, log_verbose, log_spec, limit_policy)	Constructor to initialize a PFService instance.

Table 5.103. Method summary

Method __init__(self, name, router, snat_policy, dnat_policy, log_verbose, log_spec, limit_policy)

This constructor defines a packetfilter-service with the specified parameters.

Arguments of __init__

limit_policy (class)
Default: None
Name of the LimitPolicy instance used to rate limit the sessions.

log_spec (string)
Default: None
Message filter expression.

log_verbose (integer)
Default: None
Default log verbosity level.

5.15.6. Class Service

A service is one of the fundamental objects. It stores the names of proxy-related parameters, and is also used for access control purposes to decide what kind of traffic is permitted.

Note
The Service class transfers application-level (proxy) services.

- To transfer connections on the packet-filter level, use the *PFService* class.
- To transfer connections on the application-level, use the *Service* class.

Example 5.38. Service example
The following service transfers HTTP connections. Every parameter is left at its default.

```
Service(name="demo_http, proxy_class=HttpProxy, router=TransparentRouter())
```

The following service handles HTTP connections. This service uses authentication and authorization, and network address translation on the client addresses (SNAT).

```
Service(name="demo_http", proxy_class=HttpProxy, authentication_policy="demo_authentication_policy",
 authorization_policy="demo_permituser", snat_policy="demo_natpolicy", router=TransparentRouter())
```

The following example defines a few classes: the client and server zones, a simple services, and a rule that starts the service.

```
Zone('internet', ['0.0.0.0/0'])
Zone('office', ['192.168.1.0/32', '192.168.2.0/32'])

def demo_instance() :
Service(name="office_http_inter", proxy_class=HttpProxy, router=TransparentRouter())
Rule(src_zone='office',
    proto=6,
    dst_zone='internet',
    service='office_http_inter'
    )
```

5.15.6.1. Attributes of Service

auth_name (string)
Default: n/a
Authentication name of the service. This string informs the users of the Vela Authentication Agent about which service they are authenticating for. Default value: the name of the service.

authentication_policy (class)
Default: n/a
Name of the AuthenticationPolicy instance used to authenticate the clients. See *Section 5.1, Module Auth (p. 169)* for details.

authorization_policy (class)
Default: n/a
Name of the AuthorizationPolicy instance used to authorize the clients. See *Section 5.1, Module Auth (p. 169)* for details.

chainer (class)
Default: n/a
A chainer instance used to connect to the destination server. See *Section 5.3, Module Chainer (p. 184)* for details.

dnat_policy (class)
Default: n/a
Name of the NAT policy instance used to translate the destination addresses of the sessions. See *Section 5.9, Module NAT (p. 250)* for details.

encryption_policy (class)
Default: None

encryption_policy (class)
Name of the Encryption policy instance used to encrypt the sessions and verify the certificates used. For details, see *Section 5.5, Module Encryption (p. 195)*.

instance_id (integer)
Default: n/a
The sequence number of the last session started

keepalive (integer)
Default: V_KEEPALIVE_NONE
The TCP keepalive option, one of the V_KEEPALIVE_NONE, V_KEEPALIVE_CLIENT, V_KEEPALIVE_SERVER, V_KEEPALIVE_BOTH values.

limit_policy (class)
Default: None
Name of the LimitPolicy instance used to rate limit the sessions.

max_instances (integer)
Default: n/a
Permitted number of concurrent instances of this service. Usually each service instance handles one connection. The default value is `0`, which allows unlimited number of instances.

max_sessions (integer)
Default: n/a
Maximum number of concurrent sessions handled by one thread.

num_instances (integer)
Default: n/a
The current number of running instances of this service.

proxy_class (class)
Default: n/a
Name of the proxy class instance used to analyze the traffic transferred in the session. See *Section 5.11, Module Proxy (p. 260)* for details.

resolver_policy (unknown)
Default: n/a
Name of the ResolvePolicy instance used to resolve the destination domain names. See *Section 5.12, Module Resolver (p. 264)* for details. Default value: `DNSResolver`

router (class)
Default: n/a
A router instance used to determine the destination address of the server. See *Section 5.13, Module Router (p. 266)* for details.

snat_policy (class)
Default: n/a
Name of the NAT policy instance used to translate the source addresses of the sessions. See *Section 5.9, Module NAT (p. 250)* for details.

5.15.6.2. Service methods

Method	**Description**
__init__(self, name, proxy_class, router, chainer, snat_policy, dnat_policy, authentication_policy, authorization_policy, max_instances, max_sessions, auth_name, resolver_policy, keepalive, encryption_policy, limit_target_zones_to, detector_config, detector_default_service_name, session_counting, limit_policy)	Constructor to initialize a Service instance.
startInstance(self, session)	Start a service instance.

Table 5.104. Method summary

Method __init__(self, name, proxy_class, router, chainer, snat_policy, dnat_policy, authentication_policy, authorization_policy, max_instances, max_sessions, auth_name, resolver_policy, keepalive, encryption_policy, limit_target_zones_to, detector_config, detector_default_service_name, session_counting, limit_policy)

This contructor defines a Service with the specified parameters.

Arguments of __init__

auth_name (string)
Default: None
Authentication name of the service. This string informs the users of the Vela Authentication Agent about which service they are authenticating for. Default value: the name of the service.

authentication_policy (class)
Default: None
Name of the AuthenticationPolicy instance used to authenticate the clients. See *Section 5.1, Module Auth (p. 169)* for details.

authorization_policy (class)
Default: None
Name of the AuthorizationPolicy instance used to authorize the clients. See *Section 5.1, Module Auth (p. 169)* for details.

chainer (class)
Default: None
Name of the chainer instance used to connect to the destination server. Defaults to *ConnectChainer* if no other chainer is specified.

dnat_policy (class)
Default: None
Name of the NAT policy instance used to translate the destination addresses of the sessions. See *Section 5.9, Module NAT (p. 250)* for details.

encryption_policy (class)
Default: None
Name of the Encryption policy instance used to encrypt the sessions and verify the certificates used. For details, see *Section 5.5, Module Encryption (p. 195)*.

keepalive (integer)
Default: V_KEEPALIVE_NONE
The TCP keepalive option, one of the V_KEEPALIVE_NONE, V_KEEPALIVE_CLIENT, V_KEEPALIVE_SERVER, V_KEEPALIVE_BOTH values.

limit_policy (class)
Default: None
Name of the LimitPolicy instance used to rate limit the sessions.

limit_target_zones_to (complex)
Default: None

limit_target_zones_to (complex)
A comma-separated list of zone names permitted as the target of the service. No restrictions are applied if the list is empty.

max_instances (integer)
Default: 0
Permitted number of concurrent instances of this service. Usually each service instance handles one connection. Default value: `0` (unlimited).

max_sessions (integer)
Default: 0
Maximum number of concurrent sessions handled by one thread.

name (string)
Default: n/a
The name identifying the service.

proxy_class (class)
Default: n/a
Name of the proxy class instance used to analyze the traffic transferred in the session. See *Section 5.11, Module Proxy (p. 260)* for details.

resolver_policy (class)
Default: None
Name of the ResolvePolicy instance used to resolve the destination domain names. See *Section 5.12, Module Resolver (p. 264)* for details. Default value: `DNSResolver`.

router (class)
Default: None
Name of the router instance used to determine the destination address of the server. Defaults to *TransparentRouter* if no other router is specified.

snat_policy (class)
Default: None
Name of the NAT policy instance used to translate the source addresses of the sessions. See *Section 5.9, Module NAT (p. 250)* for details.

Method startInstance(self, session)

Called by the Rule to create an instance of this service.

Arguments of startInstance

session (unknown)
Default: n/a
The session object

5.16. Module Session

This module defines the abstract session interface in a class named *`AbstractSession`*, and two descendants *`MasterSession`* and *`StackedSession`*.

Sessions are hierarchically stacked into each other just like proxies. All sessions except the master session have a parent session from which child sessions inherit variables. Child sessions are stacked into their master sessions, so stacked sessions can inherit data from the encapsulating proxy instances. (Inheritance is implemented using a simple `getattr` wrapper.)

Instances of the Session classes store the parameters of the client-side and server-side connections in a session object (for example, the IP addresses and zone of the server and the client, and the username and group memberships of the user when authentication is used). Other components refer to this data when making various policy-based decisions.

5.16.1. Classes in the Session module

Class	Description
StackedSession	Class encapsulating a subsession.

Table 5.105. Classes of the Session module

5.16.2. Class StackedSession

This class represents a stacked session, e.g., a session within the session hierarchy. Every subsession inherits session-wide parameters from its parent.

5.16.2.1. Attributes of StackedSession

chainer (class)
Default: n/a
The chainer used to connect to the parent proxy. If unset, the *`server_stream`* parameter must be set.

owner (class)
Default: n/a
The parent session of the current session.

server_address (class)
Default: n/a
The IP address to connect. Most often this is the IP address requested by the client, but the client requests can be redirected to different IPs.

server_local (class)
Default: n/a
The server is connected from this IP address. This is either the IP address of Vela's external interface, or the IP address of the client (if Forge Port is enabled). The client's original IP address may be modified if SNAT policies are used.

server_stream (class)
Default: n/a
Server-side stream.

server_zone (class)
Default: n/a
Zone of the server.

target_address (class)
Default: n/a
The IP address to connect. Most often this is the IP address requested by the client, but the client requests can be redirected to different IPs.

target_local (class)
Default: n/a
The server is connected from this IP address. This is either the IP address of Vela's external interface, or the IP address of the client (if Forge Port is enabled). The client's original IP address may be modified if SNAT policies are used.

target_zone (class)
Default: n/a
Zone of the server.

5.16.2.2. StackedSession methods

Method	Description
setTargetAddress(self, addr)	Set the target server address.

Table 5.106. Method summary

Method setTargetAddress(self, addr)

This is a compatibility function for proxies that override the routed target.

Arguments of setTargetAddress

addr (unknown)
Default: n/a
Server address

5.17. Module SockAddr

This module implements *inet_ntoa* and *inet_aton*. The module also provides an interface to the SockAddr services of the Vela core. SockAddr is used for example to define the address of the VAS server in *AuthenticationProvider* policies.

5.17.1. Classes in the SockAddr module

Class	Description
SockAddrInet	Class encapsulating an IPv4 address:port pair.
SockAddrInet6	Class encapsulating an IPv6 address:port pair.
SockAddrInetHostname	Class encapsulating a hostname:port pair.
SockAddrInetRange	Class encapsulating an IPv4 address and a port range.
SockAddrUnix	Class encapsulating a UNIX domain socket.

Table 5.107. Classes of the SockAddr module

5.17.2. Class SockAddrInet

This class encapsulates an IPv4 address:port pair, similarly to the *sockaddr_in* struct in C. The class is implemented and exported by the Vela core. The *SockAddrInet* Python class serves only documentation purposes, and has no real connection to the behavior implemented in C.

Example 5.39. SockAddrInet example
The following example defines an IPv4 address:port pair.

```
SockAddrInet('192.168.10.10', 80)
```

The following example uses SockAddrInet in a dispatcher.

```
Dispatcher(transparent=TRUE, bindto=DBSockAddr(protocol=VD_PROTO_TCP, sa=SockAddrInet('192.168.11.11',
 50080)), service="intra_HTTP_inter", backlog=255, rule_port="50080")
```

5.17.2.1. Attributes of SockAddrInet

ip (unknown)
Default: n/a
IP address (network byte order).

ip_s (unknown)
Default: n/a
IP address in string representation.

port (unknown)
Default: n/a
Port number (network byte order).

type (string)
Default: n/a
The *inet* value that indicates an address in the AF_INET domain.

5.17.3. Class SockAddrInet6

This class encapsulates an IPv6 address:port pair, similarly to the *sockaddr_in* struct in C. The class is implemented and exported by the Vela core. The *SockAddrInet* Python class serves only documentation purposes, and has no real connection to the behavior implemented in C.

Example 5.40. SockAddrInet example
The following example defines an IPv6 address:port pair.

```
SockAddrInet('fec0::1', 80)
```

The following example uses SockAddrInet in a dispatcher.

```
Dispatcher(transparent=TRUE, bindto=DBSockAddr(protocol=VD_PROTO_TCP, sa=SockAddrInet('fec0::1',
50080)), service="intra_HTTP_inter", backlog=255, rule_port="50080")
```

5.17.3.1. Attributes of SockAddrInet6

ip (unknown)
Default: n/a
IP address (network byte order).

ip_s (unknown)
Default: n/a
IP address in string representation.

port (unknown)
Default: n/a
Port number (network byte order).

type (string)
Default: n/a
The *`inet`* value that indicates an address in the AF_INET domain.

5.17.4. Class SockAddrInetHostname

This class encapsulates a hostname:port or IPv4 address:port pair. Name resolution is only performed when creating the SockAddrInetHostname object (that is, during startup and reload). The class is implemented and exported by the Vela core. The *`SockAddrInetHostname`* Python class serves only documentation purposes, and has no real connection to the behavior implemented in C.

Example 5.41. SockAddrInetHostname example
The following example defines a hostname:port or IPv4 address:port pair.

```
SockAddrInetHostname('www.example.com', 80)
```

```
SockAddrInetHostname('192.168.10.10', 80)
```

The following example uses SockAddrInetHostname in a dispatcher.

```
Dispatcher(transparent=TRUE, bindto=DBSockAddr(protocol=VD_PROTO_TCP,
sa=SockAddrInetHostname('www.example.com', 50080)), service="intra_HTTP_inter", backlog=255,
rule_port="50080")
```

5.17.4.1. Attributes of SockAddrInetHostname

ip (unknown)
Default: n/a
IP address (network byte order).

ip_s (unknown)
Default: n/a
IP address in string representation.

port (unknown)
Default: n/a
Port number (network byte order).

type (string)
Default: n/a
The *`inet`* value that indicates an address in the AF_INET domain.

5.17.5. Class SockAddrInetRange

A specialized SockAddrInet class which allocates a new port within the given range of ports when a dispatcher bounds to it. The class is implemented and exported by the Vela core. The *`SockAddrInetRange`* Python class serves only documentation purposes, and has no real connection to the behavior implemented in C.

5.17.5.1. Attributes of SockAddrInetRange

ip (unknown)
Default: n/a
IP address (network byte order).

ip_s (unknown)
Default: n/a
IP address in string representation.

port (unknown)
Default: n/a
Port number (network byte order).

type (string)
Default: n/a
The *`inet`* value that indicates an address in the AF_INET domain.

5.17.6. Class SockAddrUnix

This class encapsulates a UNIX domain socket endpoint. The socket is represented by a filename. The *`SockAddrUnix`* Python class serves only documentation purposes, and has no real connection to the behavior implemented in C.

Example 5.42. SockAddrUnix example
The following example defines a Unix domain socket.

```
SockAddrUnix('/var/sample.socket')
```

The following example uses SockAddrUnix in a DirectedRouter.

```
Service(name="demo_service", proxy_class=HttpProxy,
router=DirectedRouter(dest_addr=SockAddrUnix('/var/sample.socket'), overrideable=FALSE,
forge_addr=FALSE))
```

5.17.6.1. Attributes of SockAddrUnix

type (string)
Default: n/a
The *`unix`* value that indicates an address in the UNIX domain.

5.18. Module Stack

Vela is capable of stacking, that is, handing over parts of the traffic to other modules for further inspection (e.g., to other proxies to inspect embedded protocols, to content vectoring modules for virus filtering, etc.). The Stack module defines the classes required for this functionality.

Stacking in services is performed using *StackingProvider policies*, which reference the host that performs the stacked operations using the *RemoteStackingBackend* class.

5.18.1. Classes in the Stack module

Class	Description
AbstractStackingBackend	This is an abstract class, currently without any functionality.
RemoteStackingBackend	Constructor to initialize an instance of the RemoteStackingBackend class.
StackingProvider	This is a policy class that is used to reference a configured stacking provider in service definitions.

Table 5.108. Classes of the Stack module

5.18.2. Class AbstractStackingBackend

This is an abstract class, currently without any functionality.

5.18.3. Class RemoteStackingBackend

This class contains the address of the host that performs the stacked operations. It is typically used to access the Vela Content Vectoring Server (VCF) to perform virus filtering in the traffic. The remote backend can be accessed using the TCP protocol or a local socket, e.g., *RemoteStackingBackend(addrs=(SockAddrInet('192.168.2.3', 1318),))* or *RemoteStackingBackend(addrs=(SockAddrUnix('/var/run/vcf/vcf.sock'),)).* .

5.18.3.1. RemoteStackingBackend methods

Method	Description
__init__(self, addrs)	

Table 5.109. Method summary

Method __init__(self, addrs)

Arguments of __init__

addrs (complex)
Default: n/a
The address of the remote backend in *SockAddrInet* or *SockAddrUnix* format. Separate addresses with commas to list more than one address for a backend. Vela will connect to these addresses in a failover fashion.

5.18.4. Class StackingProvider

Instances of the StackingProvider class are policies that define which remote stacking backend a particular service uses to inspect the contents of the traffic.

Example 5.43. A simple StackingProvider class
The following class creates a simple stacking provider that can be referenced in service definitions. The remote host that provides the stacking services is located under the *192.168.12.12* IP address.

```
StackingProvider(name="demo_stackingprovider",
backend=RemoteStackingBackend(addrs=(SockAddrInet('192.168.12.12', 1318),)))
```

Example 5.44. Using a StackingProvider in an FTP proxy
The following classes define a stacking provider that can be accesses a local VCF instance using a domain socket. This service provider is then used to filter FTP traffic. The configuration of the VCF (i.e., what modules it uses to filter the traffic is not discussed here).

```
class StackingFtpProxy(FtpProxy):
def config(self):
    super(StackingFtpProxy, self).config()
    self.request_stack["RETR"]=(FTP_STK_DATA, (V_STACK_PROVIDER, "demo_stackingprovider",
"default_rulegroup"))

StackingProvider(name="demo_stackingprovider_socket",
backend=RemoteStackingBackend(addrs=(SockAddrUnix('/var/run/vcf/vcf.sock'),)))
```

5.18.4.1. StackingProvider methods

Method	Description
__init__(self, name, backend)	Constructor to initialize an instance of the StackingProvider class.

Table 5.110. Method summary

Method __init__(self, name, backend)

This constructor creates a StackingProvider instance and sets the attributes of the instance according to the received arguments.

Arguments of __init__

backend (class)
Default: n/a
A configured *RemoteStackingBackend* class containing the address of the remote stacking backend, e.g., `RemoteStackingBackend(addrs=(SockAddrInet('192.168.2.3', 1318),))` or `RemoteStackingBackend(addrs=(SockAddrUnix('/var/run/vcf/vcf.sock'),)).` .

name (string)
Default: n/a
Name of the Stacking provider policy. This name can be referenced in the service definitions.

5.19. Module Zone

This module defines the *Zone* class.

Zones are the basis of access control. A zone consists of a set of IP addresses, address ranges, or subnet. For example, a zone can contain an IPv4 or IPv6 subnet.

Zones are organized into a hierarchy created by the administrator. Child zones inherit the security attributes (set of permitted services etc.) from their parents. The administrative hierarchy often reflects the organization of the company, with zones assigned to the different departments.

When it has to be determined what zone a client belongs to, the most specific zone containing the searched IP address is selected. If an IP address belongs to two different zones, the most specific zone is selected.

Example 5.45. Finding IP networks
Suppose there are three zones configured: *Zone_A* containing the `10.0.0.0/8` network, *Zone_B* containing the `10.0.0.0/16` network, and *Zone_C* containing the `10.0.0.25` IP address. Searching for the `10.0.44.0` network returns *Zone_B*, because that is the most specific zone matching the searched IP address. Similarly, searching for `10.0.0.25` returns only *Zone_C*.

This approach is used in the service definitions as well: when a client sends a connection request, the most specific zone containing the IP address of the client is looked up. Suppose that the clients in *Zone_A* are allowed to use HTTP. If a client with IP `10.0.0.50` (thus belonging to *Zone_B*) can only use HTTP if *Zone_B* is the child of *Zone_A*, or if a service definition explicitly permits *Zone_B* to use HTTP.

Example 5.46. Zone examples
The following example defines a simple zone hierarchy. The following zones are defined:

- *internet*: This zone contains every possible IP addresses, if an IP address does not belong to another zone, than it belongs to the *internet* zone.
- *office*: This zone contains the `192.168.1.0/32` and `192.168.2.0/32` networks.
- *management*: This zone is separated from the *office* zone, because it contans an independent subnet `192.168.3.0/32`. But from the administrator's view, it is the child zone of the *office* zone, meaning that it can use (and accept) the same services as the *office* zone.
- *DMZ*: This is a separate zone.

```
Zone('internet', ['0.0.0.0/0', '::0/0'])
Zone('office', ['192.168.1.0/32', '192.168.2.0/32'])
Zone('management', ['192.168.3.0/32'])
Zone('DMZ', ['10.50.0.0/32'])
```

5.19.1. Classes in the Zone module

Class	Description
Zone	Class encapsulating IP zones.

Table 5.111. Classes of the Zone module

5.19.2. Class Zone

This class encapsulates IPv4 and IPv6 zones.

Example 5.47. Determining the zone of an IP address
An IP address always belongs to the most specific zone. Suppose that `Zone A` includes the IP network `10.0.0.0/8` and `Zone B` includes the network `10.0.1.0/24`. In this case, a client machine with the `10.0.1.100/32` IP address belongs to both zones from an IP addressing point of view. But `Zone B` is more specific (in CIDR terms), so the client machine belongs to `Zone B`.

5.19.2.1. Zone methods

Method	Description
__init__(self, name, addrs, hostnames, admin_parent)	Constructor to initialize a Zone instance

Table 5.112. Method summary

Method __init__(self, name, addrs, hostnames, admin_parent)

This constructor initializes a Zone object.

Arguments of __init__

addr (complex)
Default: n/a

addr (complex)
A string representing an address range interpreted by the domain class (last argument), *or* a list of strings representing multiple address ranges.

admin_parent (string)
Default: n/a
Name of the administrative parent zone. If set, the current zone inherits the lists of permitted inbound and outbound services from its administrative parent zone.

hostnames (complex)
Default: n/a
A string representing a domain name, the addresses of its A and AAAA records are placed into the zone hierarchy *or* a list of domain names representing multiple domain names

name (string)
Default: n/a
Name of the zone.

5.20. Module Vela

This module defines global constants (e.g., *TRUE* and *FALSE*) used by other components, and interface entry points to the core.

Chapter 6. Core-internal

This chapter provides information about some of the internal PNS modules.

6.1. Module Cache

Caching is used throughout the policy layer to improve performance. This module includes a couple of general caching classes used by various parts of the policy code.

6.2. Module Core

This module imports all public interfaces and makes it easy to use those from the user policy file by simply importing all symbols from Vela.Core.

6.3. Module Dispatch

Dispatchers bind to a specific IP address and port of the Vela firewall and wait for incoming connection requests. For each accepted connection, the Dispatcher creates a new service instance to handle the traffic arriving in the connection.

Note
Earlier product versions used different classes to handle TCP and UDP connections (Dispatchers, respectively). These classes have been merged into the Dispatcher module.

For each accepted connection, the Dispatcher creates a new service instance to handle the traffic arriving in the connection. The service started by the dispatcher depends on the type of the dispatcher:

- *Dispatchers* start the same service for every connection.
- *CSZoneDispatchers* start different services based on the zones the client and the destination server belong to.

Note
Only one dispatcher can bind to an IP address/port pair.

6.3.1. Zone-based service selection

Dispatchers can start only a predefined service. Use CSZonedDispatchers to start different services for different connections. CSZoneDispatchers assign different services to different client-server zone pairs. Define the zones and the related services in the `services` parameter. The `*` wildcard matches all client or server zones.

Note
The server zone may be modified by the proxy, the router, the chainer, or the NAT policy used in the service. To select the service, CSZoneDispatcher determines the server zone from the original destination IP address of the incoming client request. Similarly, the client zone is determined from the source IP address of the original client request.

To accept connections from the child zones of the selected client zones, set the *`follow_parent`* attribute to *`TRUE`*. Otherwise, the dispatcher accepts traffic only from the client zones explicitly listed in the *`services`* attribute of the dispatcher.

6.3.2. Classes in the Dispatch module

Class	Description
CSZoneDispatcher	Class encapsulating the Dispatcher which starts a service by the client and server zone.
Dispatcher	Class encapsulating the Dispatcher which starts a service by the client and server zone.

Table 6.1. Classes of the Dispatch module

6.3.3. Class CSZoneDispatcher

This class is similar to a simple Dispatcher, but instead of starting a fixed service, it chooses one based on the client and the destined server zone.

It takes a mapping of services indexed by a client and the server zone name, with an exception of the '*' zone, which matches anything.

NOTE: the server zone might change during proxy and NAT processing, therefore the server zone used here only matches the real destination if those phases leave the server address intact.

Example 6.1. CSZoneDispatcher example
The following example defines a CSZoneDispatcher that starts the service called *`internet_HTTP_DMZ`* for connections received on the *`192.168.2.1`* IP address, but only if the connection comes from the *`internet`* zone and the destination is in the *`DMZ`* zone.

```
CSZoneDispatcher(bindto=SockAddrInet('192.168.2.1', 50080), services={("internet",
"DMZ"):"internet_HTTP_DMZ"}, transparent=TRUE, backlog=255, threaded=FALSE, follow_parent=FALSE)
```

6.3.3.1. Attributes of CSZoneDispatcher

services (unknown)
Default: n/a
services mapping indexed by zone names

6.3.3.2. CSZoneDispatcher methods

Method	Description
*__init__(self, bindto, services, **kw)*	Constructor to initialize a CSZoneDispatcher instance.

Table 6.2. Method summary

Method __init__(self, bindto, services, **kw)

This constructor initializes a CSZoneDispatcher instance and sets its initial attributes based on arguments.

Arguments of __init__

bindto (sockaddr)
Default: n/a
An existing *socket address* containing the IP address and port number where the Dispatcher accepts connections.

follow_parent (boolean)
Default: n/a
Set this parameter to `TRUE` if the dispatcher handles also the connections coming from the child zones of the selected client zones. Otherwise, the dispatcher accepts traffic only from the explicitly listed client zones.

services (complex)
Default: n/a
Client zone - server zone - service name pairs using the `(("client_zone","server_zone"):"service")` format; specifying the service to start when the dispatcher accepts a connection from the given client zone that targets the server zone.

6.3.4. Class Dispatcher

This class is the starting point of services. It listens on the given port, and when a connection is accepted it starts a session and the given service.

Example 6.2. Dispatcher example

The following example defines a transparent dispatcher that starts the service called `demo_http_service` for connections received on the `192.168.2.1` IP address.

```
Dispatcher(bindto=SockAddrInet('192.168.2.1', 50080), service="demo_http_service", transparent=TRUE,
 backlog=255, threaded=FALSE)
```

6.3.4.1. Attributes of Dispatcher

backlog (integer)
Default: n/a
Applies only to TCP connections. This parameter sets the queue size (maximum number) of TCP connections that are established by the kernel, but not yet accepted by Vela. This queue stores the connections that successfully performed the three-way TCP handshake with the Vela host, until the dispatcher sends the *Accept* package.

bindto (sockaddr)
Default: n/a
An existing *socket address* containing the IP address and port number where the Dispatcher accepts connections.

protocol (unknown)
Default: n/a
the protocol we were bound to

service (service)
Default: n/a
Name of the service to start.

threaded (boolean)
Default: n/a
Set this parameter to *TRUE* to start a new thread for every client request. The proxy threads started by the dispatcher will start from the dispatcher's thread instead of the main thread. Incoming connections are accepted faster and optimizes queuing if this option is enabled. This improves user experience, but significantly increases the memory consumption of Vela. Use it only if a very high number of concurrent connections have to be transfered.

6.3.4.2. Dispatcher methods

Method	**Description**
*__init__(self, bindto, service, **kw)*	Constructor to initialize a Dispatcher instance.

Table 6.3. Method summary

Method __init__(self, bindto, service, **kw)

This constructor creates a new Dispatcher instance which can be associated with a *Service*.

Arguments of __init__

bindto (sockaddr)
Default: n/a
An existing *socket address* containing the IP address and port number where the Dispatcher accepts connections.

service (service)
Default: n/a
Name of the service to start.

transparent (boolean)
Default: n/a
Set this parameter to `TRUE` if the dispatcher starts a transparent service.

6.4. Module Globals

Global variables used by the policy layer.

6.5. Module Stream

This module defines the Stream class, encapsulating file descriptors and related functions.

6.5.1. Classes in the Stream module

Class	Description
Stream	Class encapsulating the file descriptor and related functions.

Table 6.4. Classes of the Stream module

6.5.2. Class Stream

This class encapsulates a full-duplex data tunnel, represented by a UNIX file descriptor. Proxies communicate with its peers through instances of this class. The `client_stream` and `server_stream` attributes of the *Session* class contain a Stream instance.

6.5.2.1. Attributes of Stream

bytes_recvd (integer)
Default: n/a
The number of bytes received in the stream.

bytes_sent (integer)
Default: n/a
The number of bytes sent in the stream.

fd (integer)
Default: n/a
The file descriptor associated to the stream.

name (string)
Default: n/a
The name of the stream.

6.5.2.2. Stream methods

Method	**Description**
__init__(self, fd, name)	Constructor to initialize a stream.

Table 6.5. Method summary

Method __init__(self, fd, name)

This constructor initializes a Stream instance setting its attributes according to arguments.

Arguments of __init__

fd (integer)
Default: n/a
The file descriptor associated to the stream.

name (string)
Default: n/a
The name of the stream.

Appendix A. Additional proxy information

A.1. TELNET appendix

The constants defined for the easier use of TELNET options and suboptions are listed in the table below. Suboptions are listed directly under the option they refer to. All suboptions have the TELNET_SB prefix. The RFC describing the given option is also shown in the table.

Name	Constant value of option/suboption	Detailed in RFC #
TELNET_BINARY	0	856
TELNET_ECHO	1	857
TELNET_SUPPRESS_GO_AHEAD	3	858
TELNET_STATUS	5	859
TELNET_SB_STATUS_SB_IS	0	
TELNET_SB_STATUS_SB_SEND	1	
TELNET_TIMING_MARK	6	860
TELNET_RCTE	7	726
TELNET_NAOCRD	10	652
TELNET_SB_NAOCRD_DR	0	
TELNET_SB_NAOCRD_DS	1	
TELNET_NAOHTS	11	653
TELNET_SB_NAOHTS_DR	0	
TELNET_SB_NAOHTS_DS	1	
TELNET_NAOHTD	12	654
TELNET_SB_NAOHTD_DR	0	
TELNET_SB_NAOHTD_DS	1	
TELNET_NAOFFD	13	655
TELNET_SB_NAOFFD_DR	0	
TELNET_SB_NAOFFD_DS	1	
TELNET_NAOVTS	14	656
TELNET_SB_NAOVTS_DR	0	
TELNET_SB_NAOVTS_DS	1	

Name	Constant value of option/suboption	Detailed in RFC #
TELNET_NAOVTD	15	657
TELNET_SB_NAOVTD_DR	0	
TELNET_SB_NAOVTD_DS	1	
TELNET_NAOLFD	16	658
TELNET_SB_NAOLFD_DR	0	
TELNET_SB_NAOLFD_DS	1	
TELNET_EXTEND_ASCII	17	698
TELNET_LOGOUT	18	727
TELNET_BM	19	735
TELNET_SB_BM_DEFINE	1	
TELNET_SB_BM_ACCEPT	2	
TELNET_SB_BM_REFUSE	3	
TELNET_SB_BM_LITERAL	4	
TELNET_SB_BM_CANCEL	5	
TELNET_DET	20	1043, 732
TELNET_SB_DET_DEFINE	1	
TELNET_SB_DET_ERASE	2	
TELNET_SB_DET_TRANSMIT	3	
TELNET_SB_DET_FORMAT	4	
TELNET_SB_DET_MOVE_CURSOR	5	
TELNET_SB_DET_SKIP_TO_LINE	6	
TELNET_SB_DET_SKIP_TO_CHAR	7	
TELNET_SB_DET_UP	8	
TELNET_SB_DET_DOWN	9	
TELNET_SB_DET_LEFT	10	
TELNET_SB_DET_RIGHT	11	
TELNET_SB_DET_HOME	12	
TELNET_SB_DET_LINE_INSERT	13	
TELNET_SB_DET_LINE_DELETE	14	
TELNET_SB_DET_CHAR_INSERT	15	

Name	Constant value of option/suboption	Detailed in RFC #
TELNET_SB_DET_CHAR_DELETE	16	
TELNET_SB_DET_READ_CURSOR	17	
TELNET_SB_DET_CURSOR_POSITION	18	
TELNET_SB_DET_REVERSE_TAB	19	
TELNET_SB_DET_TRANSMIT_SCREEN	20	
TELNET_SB_DET_TRANSMIT_UNPROTECTED	21	
TELNET_SB_DET_TRANSMIT_LINE	22	
TELNET_SB_DET_TRANSMIT_FIELD	23	
TELNET_SB_DET_TRANSMIT_REST_SCREEN	24	
TELNET_SB_DET_TRANSMIT_REST_LINE	25	
TELNET_SB_DET_TRANSMIT_REST_FIELD	26	
TELNET_SB_DET_TRANSMIT_MODIFIED	27	
TELNET_SB_DET_DATA_TRANSMIT	28	
TELNET_SB_DET_ERASE_SCREEN	29	
TELNET_SB_DET_ERASE_LINE	30	
TELNET_SB_DET_ERASE_FIELD	31	
TELNET_SB_DET_ERASE_REST_SCREEN	32	
TELNET_SB_DET_ERASE_REST_LINE	33	
TELNET_SB_DET_ERASE_REST_FIELD	34	
TELNET_SB_DET_ERASE_UNPROTECTED	35	
TELNET_SB_DET_FORMAT_DATA	36	
TELNET_SB_DET_REPEAT	37	
TELNET_SB_DET_SUPPRESS_PROTECTION	38	
TELNET_SB_DET_FIELD_SEPARATOR	39	
TELNET_SB_DET_FN	40	
TELNET_SB_DET_ERROR	41	
TELNET_SUPDUP	21	736, 734
TELNET_SUPDUP_OUTPUT	22	749
TELNET_SEND_LOCATION	23	779
TELNET_TERMINAL_TYPE	24	1091

Name	Constant value of option/suboption	Detailed in RFC #
TELNET_SB_TERMINAL_TYPE_IS	0	
TELNET_SB_TERMINAL_TYPE_SEND	1	
TELNET_EOR	25	885
TELNET_TUID	26	927
TELNET_OUTMRK	27	933
TELNET_TTYLOC	28 946	
TELNET_3270_REGIME	29	1041
TELNET_SB_3270_REGIME_IS	0	
TELNET_SB_3270_REGIME_ARE	1	
TELNET_X3_PAD	30	1053
TELNET_SB_X3_PAD_SET	0	
TELNET_SB_X3_PAD_RESPONSE_SET	1	
TELNET_SB_X3_PAD_IS	2	
TELNET_SB_X3_PAD_RESPONSE_IS	3	
TELNET_SB_X3_PAD_SEND	4	
TELNET_NAWS	31	1073
TELNET_TERMINAL_SPEED	32	1079
TELNET_SB_TERMINAL_SPEED_IS	0	
TELNET_SB_TERMINAL_SPEED_SEND	1	
TELNET_TOGGLE_FLOW_CONTROL	33	1372
TELNET_SB_TOGGLE_FLOW_CONTROL_OFF	0	
TELNET_SB_TOGGLE_FLOW_CONTROL_ON	1	
TELNET_SB_TOGGLE_FLOW_CONTROL_RESTART_ANY	2	
TELNET_SB_TOGGLE_FLOW_CONTROL_RESTART_XON	3	
TELNET_LINEMODE	34	1184
TELNET_SB_LINEMODE_MODE	1	
TELNET_SB_LINEMODE_FORWARDMASK	2	
TELNET_SB_LINEMODE_SLC	3	
TELNET_X_DISPLAY_LOCATION	35	1096
TELNET_SB_X_DISPLAY_LOCATION_IS	0	

Name	Constant value of option/suboption	Detailed in RFC #
TELNET_SB_X_DISPLAY_LOCATION_SEND	1	
TELNET_OLD_ENVIRONMENT	36	1408
TELNET_SB_OLD_ENVIRONMENT_IS	0	
TELNET_SB_OLD_ENVIRONMENT_SEND	1	
TELNET_SB_OLD_ENVIRONMENT_INFO	2	
TELNET_AUTHENTICATION	37	2941
TELNET_SB_AUTHENTICATION_IS	0	
TELNET_SB_AUTHENTICATION_SEND	1	
TELNET_SB_AUTHENTICATION_REPLY	2	
TELNET_SB_AUTHENTICATION_NAME	3	
TELNET_ENCRYPT	38	2946
TELNET_SB_ENCRYPT_IS	0	
TELNET_SB_ENCRYPT_SUPPORT	1	
TELNET_SB_ENCRYPT_REPLY	2	
TELNET_SB_ENCRYPT_START	3	
TELNET_SB_ENCRYPT_END	4	
TELNET_SB_ENCRYPT_REQUEST_START	5	
TELNET_SB_ENCRYPT_REQUEST_END	6	
TELNET_SB_ENCRYPT_ENC_KEYID	7	
TELNET_SB_ENCRYPT_DEC_KEYID	8	
TELNET_ENVIRONMENT	39	1572
TELNET_SB_ENVIRONMENT_IS	0	
TELNET_SB_ENVIRONMENT_SEND	1	
TELNET_SB_ENVIRONMENT_INFO	2	
TELNET_TN3270E	40	1647
TELNET_SB_TN3270E_ASSOCIATE	0	
TELNET_SB_TN3270E_CONNECT	1	
TELNET_SB_TN3270E_DEVICE_TYPE	2	
TELNET_SB_TN3270E_FUNCTIONS	3	
TELNET_SB_TN3270E_IS	4	

Name	Constant value of option/suboption	Detailed in RFC #
TELNET_SB_TN3270E_REASON	5	
TELNET_SB_TN3270E_REJECT	6	
TELNET_SB_TN3270E_REQUEST	7	
TELNET_SB_TN3270E_SEND	8	
TELNET_CHARSET	42	2066
TELNET_SB_CHARSET_REQUEST	1	
TELNET_SB_CHARSET_ACCEPTED	2	
TELNET_SB_CHARSET_REJECTED	3	
TELNET_SB_CHARSET_TTABLE_IS	4	
TELNET_SB_CHARSET_TTABLE_REJECTED	5	
TELNET_SB_CHARSET_TTABLE_ACK	6	
TELNET_SB_CHARSET_TTABLE_NAK	7	
TELNET_COM_PORT	44	2217
TELNET_SB_COM_PORT_CLI_SET_BAUDRATE	1	
TELNET_SB_COM_PORT_CLI_SET_DATASIZE	2	
TELNET_SB_COM_PORT_CLI_SET_PARITY	3	
TELNET_SB_COM_PORT_CLI_SET_STOPSIZE	4	
TELNET_SB_COM_PORT_CLI_SET_CONTROL	5	
TELNET_SB_COM_PORT_CLI_NOTIFY_LINESTATE	6	
TELNET_SB_COM_PORT_CLI_NOTIFY_MODEMSTATE	7	
TELNET_SB_COM_PORT_CLI_FLOWCONTROL_SUSPEND	8	
TELNET_SB_COM_PORT_CLI_FLOWCONTROL_RESUME	9	
TELNET_SB_COM_PORT_CLI_SET_LINESTATE_MASK	10	
TELNET_SB_COM_PORT_CLI_SET_MODEMSTATE_MASK	11	
TELNET_SB_COM_PORT_CLI_PURGE_DATA	12	
TELNET_SB_COM_PORT_SVR_SET_BAUDRATE	101	
TELNET_SB_COM_PORT_SVR_SET_DATASIZE	102	
TELNET_SB_COM_PORT_SVR_SET_PARITY	103	
TELNET_SB_COM_PORT_SVR_SET_STOPSIZE	104	
TELNET_SB_COM_PORT_SVR_SET_CONTROL	105	

Name	Constant value of option/suboption	Detailed in RFC #
TELNET_SB_COM_PORT_SVR_NOTIFY_LINESTATE	106	
TELNET_SB_COM_PORT_SVR_NOTIFY_MODEMSTATE	107	
TELNET_SB_COM_PORT_SVR_FLOWCONTROL_SUSPEND	108	
TELNET_SB_COM_PORT_SVR_FLOWCONTROL_RESUME	109	
TELNET_SB_COM_PORT_SVR_SET_LINESTATE_MASK	110	
TELNET_SB_COM_PORT_SVR_SET_MODEMSTATE_MASK	111	
TELNET_SB_COM_PORT_SVR_PURGE_DATA	112	
TELNET_KERMIT	47	2840
TELNET_SB_KERMIT_START_SERVER	0	
TELNET_SB_KERMIT_STOP_SERVER	1	
TELNET_SB_KERMIT_REQ_START_SERVER	2	
TELNET_SB_KERMIT_REQ_STOP_SERVER	3	
TELNET_SB_KERMIT_SOP	4	
TELNET_SB_KERMIT_RESP_START_SERVER	8	
TELNET_SB_KERMIT_RESP_STOP_SERVER	9	
TELNET_EXOPL	255	861
TELNET_SUBLIMINAL_MSG	257	1097

Table A.1. TELNET options and suboptions

A.2. RADIUS appendix

The list of RADIUS attributes as defined by the RADIUS RFC with their symbolic names:

Name	Value
RADIUS_USER_name	"1"
RADIUS_USER_PASSWORD	"2"
RADIUS_CHAP_PASSWORD	"3"
RADIUS_NAS_IP_ADDRESS	"4"
RADIUS_NAS_PORT	"5"
RADIUS_SERVICE_TYPE	"6"
RADIUS_FRAMED_PROTOCOL	"7"
RADIUS_FRAMED_IP_ADDRESS	"8"

Name	Value
RADIUS_FRAMED_IP_NETMASK	"9"
RADIUS_FRAMED_ROUTING	"10"
RADIUS_FILTER_ID	"11"
RADIUS_FRAMED_MTU	"12"
RADIUS_FRAMED_COMPRESSION	"13"
RADIUS_LOGIN_IP_HOST	"14"
RADIUS_LOGIN_SERVICE	"15"
RADIUS_LOGIN_TCP_PORT	"16"
RADIUS_REPLY_MESSAGE	"18"
RADIUS_CALLBACK_NUMBER	"19"
RADIUS_CALLBACK_ID	"20"
RADIUS_FRAMED_ROUTE	"22"
RADIUS_FRAMED_IPX_NETWORK	"23"
RADIUS_STATE	"24"
RADIUS_CLASS	"25"
RADIUS_VENDOR_SPECIFIC	"26"
RADIUS_SESSION_TIMEOUT	"27"
RADIUS_IDLE_TIMEOUT	"28"
RADIUS_TERMINATION_ACTION	"29"
RADIUS_CALLED_STATION_ID	"30"
RADIUS_CALLING_STATION_ID	"31"
RADIUS_NAS_IDENTIFIER	"32"
RADIUS_PROXY_STATE	"33"
RADIUS_LOGIN_LAT_SERVICE	"34"
RADIUS_LOGIN_LAT_NODE	"35"
RADIUS_LOGIN_LAT_GROUP	"36"
RADIUS_FRAMED_APPLETALK_LINK	"37"
RADIUS_FRAMED_APPLETALK_NETWORK	"38"
RADIUS_FRAMED_APPLETALK_ZONE	"39"
RADIUS_ACCT_STATUS_TYPE	"40"
RADIUS_ACCT_DELAY_TIME	"41"

Name	Value
RADIUS_ACCT_INPUT_OCTETS	"42"
RADIUS_ACCT_OUTPUT_OCTETS	"43"
RADIUS_ACCT_SESSION_ID	"44"
RADIUS_ACCT_AUTHENTIC	"45"
RADIUS_ACCT_SESSION_TIME	"46"
RADIUS_ACCT_INPUT_PACKETS	"47"
RADIUS_ACCT_OUTPUT_PACKETS	"48"
RADIUS_ACCT_TERMINATE_CAUSE	"49"
RADIUS_ACCT_MULTI_SESSION_ID	"50"
RADIUS_ACCT_LINK_COUNT	"51"
RADIUS_ACCT_INPUT_GIGAWORDS	"52"
RADIUS_ACCT_OUTPUT_GIGAWORDS	"53"
RADIUS_EVENT_TIMESTAMP	"55"
RADIUS_CHAP_CHALLENGE	"60"
RADIUS_NAS_PORT_TYPE	"61"
RADIUS_PORT_LIMIT	"62"
RADIUS_LOGIN_LAT_PORT	"63"
RADIUS_TUNNEL_TYPE	"64"
RADIUS_TUNNEL_MEDIUM_TYPE	"65"
RADIUS_TUNNEL_CLIENT_ENDPOINT	"66"
RADIUS_TUNNEL_SERVER_ENDPOINT	"67"
RADIUS_ACCT_TUNNEL_CONNECTION	"68"
RADIUS_TUNNEL_PASSWORD	"69"
RADIUS_ARAP_PASSWORD	"70"
RADIUS_ARAP_FEATURES	"71"
RADIUS_ARAP_ZONE_ACCESS	"72"
RADIUS_ARAP_SECURITY	"73"
RADIUS_ARAP_SECURITY_DATA	"74"
RADIUS_PASSWORD_RETRY	"75"
RADIUS_PROMPT	"76"
RADIUS_CONNECT_INFO	"77"

Name	Value
RADIUS_CONFIGURATION_TOKEN	"78"
RADIUS_EAP_MESSAGE	"79"
RADIUS_MESSAGE_AUTHENTICATOR	"80"
RADIUS_TUNNEL_PRIVATE_GROUP_ID	"81"
RADIUS_TUNNEL_ASSIGNMENT_ID	"82"
RADIUS_TUNNEL_PREFERENCE	"83"
RADIUS_ARAP_CHALLENGE_RESPONSE	"84"
RADIUS_ACCT_INTERIM_INTERVAL	"85"
RADIUS_ACCT_TUNNEL_PACKETS_LOST	"86"
RADIUS_NAS_PORT_ID	"87"
RADIUS_FRAMED_POOL	"88"
RADIUS_TUNNEL_CLIENT_AUTH_ID	"90"
RADIUS_TUNNEL_SERVER_AUTH_ID	"91"

Table A.2. RADIUS Protocol Attribute types described in RFC 2865.

Action	Attribute	Attribute policy
radius_access_request	radius_user_name	RADIUS_ATR_MAXONE
radius_access_request	radius_user_password	RADIUS_ATR_MAXONE
radius_access_request	radius_chap_password	RADIUS_ATR_MAXONE
radius_access_request	radius_nas_ip_address	RADIUS_ATR_MAXONE
radius_access_request	radius_nas_port	RADIUS_ATR_MAXONE
radius_access_request	radius_service_type	RADIUS_ATR_MAXONE
radius_access_request	radius_framed_protocol	RADIUS_ATR_MAXONE
radius_access_request	radius_framed_ip_address	RADIUS_ATR_MAXONE
radius_access_request	radius_framed_ip_netmask	RADIUS_ATR_MAXONE
radius_access_request	radius_framed_routing	RADIUS_ATR_ZERO
radius_access_request	radius_filter_id	RADIUS_ATR_ZERO
radius_access_request	radius_framed_mtu	RADIUS_ATR_MAXONE
radius_access_request	radius_framed_compression	RADIUS_ATR_MANY
radius_access_request	radius_login_ip_host	RADIUS_ATR_MANY
radius_access_request	radius_login_service	RADIUS_ATR_ZERO
radius_access_request	radius_login_tcp_port	RADIUS_ATR_ZERO

Action	Attribute	Attribute policy
radius_access_request	radius_reply_message	RADIUS_ATR_ZERO
radius_access_request	radius_callback_number	RADIUS_ATR_MAXONE
radius_access_request	radius_callback_id	RADIUS_ATR_ZERO
radius_access_request	radius_framed_route	RADIUS_ATR_ZERO
radius_access_request	radius_framed_ipx_network	RADIUS_ATR_ZERO
radius_access_request	radius_state	RADIUS_ATR_MAXONE
radius_access_request	radius_class	RADIUS_ATR_ZERO
radius_access_request	radius_vendor_specific	RADIUS_ATR_MANY
radius_access_request	radius_session_timeout	RADIUS_ATR_ZERO
radius_access_request	radius_idle_timeout	RADIUS_ATR_ZERO
radius_access_request	radius_termination_action	RADIUS_ATR_ZERO
radius_access_request	radius_called_station_id	RADIUS_ATR_MAXONE
radius_access_request	radius_calling_station_id	RADIUS_ATR_MAXONE
radius_access_request	radius_nas_identifier	RADIUS_ATR_MAXONE
radius_access_request	radius_proxy_state	RADIUS_ATR_MANY
radius_access_request	radius_login_lat_service	RADIUS_ATR_MAXONE
radius_access_request	radius_login_lat_node	RADIUS_ATR_MAXONE
radius_access_request	radius_login_lat_group	RADIUS_ATR_MAXONE
radius_access_request	radius_framed_appletalk_link	RADIUS_ATR_ZERO
radius_access_request	radius_framed_appletalk_network	RADIUS_ATR_ZERO
radius_access_request	radius_framed_appletalk_zone	RADIUS_ATR_ZERO
radius_access_request	radius_chap_challenge	RADIUS_ATR_MAXONE
radius_access_request	radius_nas_port_type	RADIUS_ATR_MAXONE
radius_access_request	radius_port_limit	RADIUS_ATR_MAXONE
radius_access_request	radius_login_lat_port	RADIUS_ATR_MAXONE
radius_access_request	radius_tunnel_type	RADIUS_ATR_MANY
radius_access_request	radius_tunnel_medium_type	RADIUS_ATR_MANY
radius_access_request	radius_tunnel_client_endpoint	RADIUS_ATR_MANY
radius_access_request	radius_tunnel_server_endpoint	RADIUS_ATR_MANY
radius_access_request	radius_tunnel_password	RADIUS_ATR_ZERO
radius_access_request	radius_arap_password	RADIUS_ATR_MAXONE

Action	Attribute	Attribute policy
radius_access_request	radius_arap_features	RADIUS_ATR_ZERO
radius_access_request	radius_arap_zone_access	RADIUS_ATR_ZERO
radius_access_request	radius_arap_security	RADIUS_ATR_MAXONE
radius_access_request	radius_arap_security_data	RADIUS_ATR_MANY
radius_access_request	radius_password_retry	RADIUS_ATR_ZERO
radius_access_request	radius_prompt	RADIUS_ATR_ZERO
radius_access_request	radius_connect_info	RADIUS_ATR_MAXONE
radius_access_request	radius_configuration_token	RADIUS_ATR_ZERO
radius_access_request	radius_eap_message	RADIUS_ATR_MANY
radius_access_request	radius_message_authenticator	RADIUS_ATR_MAXONE
radius_access_request	radius_tunnel_private_group_id	RADIUS_ATR_MANY
radius_access_request	radius_tunnel_assignment_id	RADIUS_ATR_ZERO
radius_access_request	radius_tunnel_preference	RADIUS_ATR_MANY
radius_access_request	radius_arap_challenge_response	RADIUS_ATR_ZERO
radius_access_request	radius_acct_interim_interval	RADIUS_ATR_ZERO
radius_access_request	radius_nas_port_id	RADIUS_ATR_MAXONE
radius_access_request	radius_framed_pool	RADIUS_ATR_ZERO
radius_access_request	radius_tunnel_client_auth_id	RADIUS_ATR_MANY
radius_access_request	radius_tunnel_server_auth_id	RADIUS_ATR_MANY
radius_access_accept	radius_user_name	RADIUS_ATR_MAXONE
radius_access_accept	radius_user_password	RADIUS_ATR_ZERO
radius_access_accept	radius_chap_password	RADIUS_ATR_ZERO
radius_access_accept	radius_nas_ip_address	RADIUS_ATR_ZERO
radius_access_accept	radius_nas_port	RADIUS_ATR_ZERO
radius_access_accept	radius_service_type	RADIUS_ATR_MAXONE
radius_access_accept	radius_framed_protocol	RADIUS_ATR_MAXONE
radius_access_accept	radius_framed_ip_address	RADIUS_ATR_MAXONE
radius_access_accept	radius_framed_ip_netmask	RADIUS_ATR_MAXONE
radius_access_accept	radius_framed_routing	RADIUS_ATR_MAXONE
radius_access_accept	radius_filter_id	RADIUS_ATR_MANY
radius_access_accept	radius_framed_mtu	RADIUS_ATR_MAXONE

Action	**Attribute**	**Attribute policy**
radius_access_accept	radius_framed_compression	RADIUS_ATR_MANY
radius_access_accept	radius_login_ip_host	RADIUS_ATR_MANY
radius_access_accept	radius_login_service	RADIUS_ATR_MAXONE
radius_access_accept	radius_login_tcp_port	RADIUS_ATR_MAXONE
radius_access_accept	radius_reply_message	RADIUS_ATR_MANY
radius_access_accept	radius_callback_number	RADIUS_ATR_MAXONE
radius_access_accept	radius_callback_id	RADIUS_ATR_MAXONE
radius_access_accept	radius_framed_route	RADIUS_ATR_MANY
radius_access_accept	radius_framed_ipx_network	RADIUS_ATR_MAXONE
radius_access_accept	radius_state	RADIUS_ATR_MAXONE
radius_access_accept	radius_class	RADIUS_ATR_MANY
radius_access_accept	radius_vendor_specific	RADIUS_ATR_MANY
radius_access_accept	radius_session_timeout	RADIUS_ATR_MAXONE
radius_access_accept	radius_idle_timeout	RADIUS_ATR_MAXONE
radius_access_accept	radius_termination_action	RADIUS_ATR_MAXONE
radius_access_accept	radius_called_station_id	RADIUS_ATR_ZERO
radius_access_accept	radius_calling_station_id	RADIUS_ATR_ZERO
radius_access_accept	radius_nas_identifier	RADIUS_ATR_ZERO
radius_access_accept	radius_proxy_state	RADIUS_ATR_MANY
radius_access_accept	radius_login_lat_service	RADIUS_ATR_MAXONE
radius_access_accept	radius_login_lat_node	RADIUS_ATR_MAXONE
radius_access_accept	radius_login_lat_group	RADIUS_ATR_MAXONE
radius_access_accept	radius_framed_appletalk_link	RADIUS_ATR_MAXONE
radius_access_accept	radius_framed_appletalk_network	RADIUS_ATR_MANY
radius_access_accept	radius_framed_appletalk_zone	RADIUS_ATR_MAXONE
radius_access_accept	radius_chap_challenge	RADIUS_ATR_ZERO
radius_access_accept	radius_nas_port_type	RADIUS_ATR_ZERO
radius_access_accept	radius_port_limit	RADIUS_ATR_MAXONE
radius_access_accept	radius_login_lat_port	RADIUS_ATR_MAXONE
radius_access_accept	radius_tunnel_type	RADIUS_ATR_MANY
radius_access_accept	radius_tunnel_medium_type	RADIUS_ATR_MANY

Action	Attribute	Attribute policy
radius_access_accept	radius_tunnel_client_endpoint	RADIUS_ATR_MANY
radius_access_accept	radius_tunnel_server_endpoint	RADIUS_ATR_MANY
radius_access_accept	radius_tunnel_password	RADIUS_ATR_MANY
radius_access_accept	radius_arap_password	RADIUS_ATR_ZERO
radius_access_accept	radius_arap_features	RADIUS_ATR_MAXONE
radius_access_accept	radius_arap_zone_access	RADIUS_ATR_MAXONE
radius_access_accept	radius_arap_security	RADIUS_ATR_ZERO
radius_access_accept	radius_arap_security_data	RADIUS_ATR_ZERO
radius_access_accept	radius_password_retry	RADIUS_ATR_ZERO
radius_access_accept	radius_prompt	RADIUS_ATR_ZERO
radius_access_accept	radius_connect_info	RADIUS_ATR_ZERO
radius_access_accept	radius_configuration_token	RADIUS_ATR_MANY
radius_access_accept	radius_eap_message	RADIUS_ATR_MANY
radius_access_accept	radius_message_authenticator	RADIUS_ATR_MAXONE
radius_access_accept	radius_tunnel_private_group_id	RADIUS_ATR_MANY
radius_access_accept	radius_tunnel_assignment_id	RADIUS_ATR_MANY
radius_access_accept	radius_tunnel_preference	RADIUS_ATR_MANY
radius_access_accept	radius_arap_challenge_response	RADIUS_ATR_MAXONE
radius_access_accept	radius_acct_interim_interval	RADIUS_ATR_MAXONE
radius_access_accept	radius_nas_port_id	RADIUS_ATR_ZERO
radius_access_accept	radius_framed_pool	RADIUS_ATR_MAXONE
radius_access_accept	radius_tunnel_client_auth_id	RADIUS_ATR_MANY
radius_access_accept	radius_tunnel_server_auth_id	RADIUS_ATR_MANY
radius_access_reject	radius_user_name	RADIUS_ATR_ZERO
radius_access_reject	radius_user_password	RADIUS_ATR_ZERO
radius_access_reject	radius_chap_password	RADIUS_ATR_ZERO
radius_access_reject	radius_nas_ip_address	RADIUS_ATR_ZERO
radius_access_reject	radius_nas_port	RADIUS_ATR_ZERO
radius_access_reject	radius_service_type	RADIUS_ATR_ZERO
radius_access_reject	radius_framed_protocol	RADIUS_ATR_ZERO
radius_access_reject	radius_framed_ip_address	RADIUS_ATR_ZERO

Action	Attribute	Attribute policy
radius_access_reject	radius_framed_ip_netmask	RADIUS_ATR_ZERO
radius_access_reject	radius_framed_routing	RADIUS_ATR_ZERO
radius_access_reject	radius_filter_id	RADIUS_ATR_ZERO
radius_access_reject	radius_framed_mtu	RADIUS_ATR_ZERO
radius_access_reject	radius_framed_compression	RADIUS_ATR_ZERO
radius_access_reject	radius_login_ip_host	RADIUS_ATR_ZERO
radius_access_reject	radius_login_service	RADIUS_ATR_ZERO
radius_access_reject	radius_login_tcp_port	RADIUS_ATR_ZERO
radius_access_reject	radius_reply_message	RADIUS_ATR_MANY
radius_access_reject	radius_callback_number	RADIUS_ATR_ZERO
radius_access_reject	radius_callback_id	RADIUS_ATR_ZERO
radius_access_reject	radius_framed_route	RADIUS_ATR_ZERO
radius_access_reject	radius_framed_ipx_network	RADIUS_ATR_ZERO
radius_access_reject	radius_state	RADIUS_ATR_ZERO
radius_access_reject	radius_class	RADIUS_ATR_ZERO
radius_access_reject	radius_vendor_specific	RADIUS_ATR_ZERO
radius_access_reject	radius_session_timeout	RADIUS_ATR_ZERO
radius_access_reject	radius_idle_timeout	RADIUS_ATR_ZERO
radius_access_reject	radius_termination_action	RADIUS_ATR_ZERO
radius_access_reject	radius_called_station_id	RADIUS_ATR_ZERO
radius_access_reject	radius_calling_station_id	RADIUS_ATR_ZERO
radius_access_reject	radius_nas_identifier	RADIUS_ATR_ZERO
radius_access_reject	radius_proxy_state	RADIUS_ATR_MANY
radius_access_reject	radius_login_lat_service	RADIUS_ATR_ZERO
radius_access_reject	radius_login_lat_node	RADIUS_ATR_ZERO
radius_access_reject	radius_login_lat_group	RADIUS_ATR_ZERO
radius_access_reject	radius_framed_appletalk_link	RADIUS_ATR_ZERO
radius_access_reject	radius_framed_appletalk_network	RADIUS_ATR_ZERO
radius_access_reject	radius_framed_appletalk_zone	RADIUS_ATR_ZERO
radius_access_reject	radius_chap_challenge	RADIUS_ATR_ZERO
radius_access_reject	radius_nas_port_type	RADIUS_ATR_ZERO

Action	Attribute	Attribute policy
radius_access_reject	radius_port_limit	RADIUS_ATR_ZERO
radius_access_reject	radius_login_lat_port	RADIUS_ATR_ZERO
radius_access_reject	radius_tunnel_type	RADIUS_ATR_ZERO
radius_access_reject	radius_tunnel_medium_type	RADIUS_ATR_ZERO
radius_access_reject	radius_tunnel_client_endpoint	RADIUS_ATR_ZERO
radius_access_reject	radius_tunnel_server_endpoint	RADIUS_ATR_ZERO
radius_access_reject	radius_tunnel_password	RADIUS_ATR_ZERO
radius_access_reject	radius_arap_password	RADIUS_ATR_ZERO
radius_access_reject	radius_arap_features	RADIUS_ATR_ZERO
radius_access_reject	radius_arap_zone_access	RADIUS_ATR_ZERO
radius_access_reject	radius_arap_security	RADIUS_ATR_ZERO
radius_access_reject	radius_arap_security_data	RADIUS_ATR_ZERO
radius_access_reject	radius_password_retry	RADIUS_ATR_MAXONE
radius_access_reject	radius_prompt	RADIUS_ATR_ZERO
radius_access_reject	radius_connect_info	RADIUS_ATR_ZERO
radius_access_reject	radius_configuration_token	RADIUS_ATR_ZERO
radius_access_reject	radius_eap_message	RADIUS_ATR_MANY
radius_access_reject	radius_message_authenticator	RADIUS_ATR_MAXONE
radius_access_reject	radius_tunnel_private_group_id	RADIUS_ATR_ZERO
radius_access_reject	radius_tunnel_assignment_id	RADIUS_ATR_ZERO
radius_access_reject	radius_tunnel_preference	RADIUS_ATR_ZERO
radius_access_reject	radius_arap_challenge_response	RADIUS_ATR_ZERO
radius_access_reject	radius_acct_interim_interval	RADIUS_ATR_ZERO
radius_access_reject	radius_nas_port_id	RADIUS_ATR_ZERO
radius_access_reject	radius_framed_pool	RADIUS_ATR_ZERO
radius_access_reject	radius_tunnel_client_auth_id	RADIUS_ATR_ZERO
radius_access_reject	radius_tunnel_server_auth_id	RADIUS_ATR_ZERO
radius_accounting_request	radius_user_name	RADIUS_ATR_MAXONE
radius_accounting_request	radius_user_password	RADIUS_ATR_ZERO
radius_accounting_request	radius_chap_password	RADIUS_ATR_ZERO
radius_accounting_request	radius_nas_ip_address	RADIUS_ATR_MAXONE

Action	Attribute	Attribute policy
radius_accounting_request	radius_nas_port	RADIUS_ATR_MAXONE
radius_accounting_request	radius_service_type	RADIUS_ATR_MAXONE
radius_accounting_request	radius_framed_protocol	RADIUS_ATR_MAXONE
radius_accounting_request	radius_framed_ip_address	RADIUS_ATR_MAXONE
radius_accounting_request	radius_framed_ip_netmask	RADIUS_ATR_MAXONE
radius_accounting_request	radius_framed_routing	RADIUS_ATR_MAXONE
radius_accounting_request	radius_filter_id	RADIUS_ATR_MANY
radius_accounting_request	radius_framed_mtu	RADIUS_ATR_MAXONE
radius_accounting_request	radius_framed_compression	RADIUS_ATR_MANY
radius_accounting_request	radius_login_ip_host	RADIUS_ATR_MANY
radius_accounting_request	radius_login_service	RADIUS_ATR_MAXONE
radius_accounting_request	radius_login_tcp_port	RADIUS_ATR_MAXONE
radius_accounting_request	radius_reply_message	RADIUS_ATR_ZERO
radius_accounting_request	radius_callback_number	RADIUS_ATR_MAXONE
radius_accounting_request	radius_callback_id	RADIUS_ATR_MAXONE
radius_accounting_request	radius_framed_route	RADIUS_ATR_MANY
radius_accounting_request	radius_framed_ipx_network	RADIUS_ATR_MAXONE
radius_accounting_request	radius_state	RADIUS_ATR_ZERO
radius_accounting_request	radius_class	RADIUS_ATR_MANY
radius_accounting_request	radius_vendor_specific	RADIUS_ATR_MANY
radius_accounting_request	radius_session_timeout	RADIUS_ATR_MAXONE
radius_accounting_request	radius_idle_timeout	RADIUS_ATR_MAXONE
radius_accounting_request	radius_termination_action	RADIUS_ATR_MAXONE
radius_accounting_request	radius_called_station_id	RADIUS_ATR_MAXONE
radius_accounting_request	radius_calling_station_id	RADIUS_ATR_MAXONE
radius_accounting_request	radius_nas_identifier	RADIUS_ATR_MAXONE
radius_accounting_request	radius_proxy_state	RADIUS_ATR_MANY
radius_accounting_request	radius_login_lat_service	RADIUS_ATR_MAXONE
radius_accounting_request	radius_login_lat_node	RADIUS_ATR_MAXONE
radius_accounting_request	radius_login_lat_group	RADIUS_ATR_MAXONE
radius_accounting_request	radius_framed_appletalk_link	RADIUS_ATR_MAXONE

Action	Attribute	Attribute policy
radius_accounting_request	radius_framed_appletalk_network	RADIUS_ATR_MAXONE
radius_accounting_request	radius_framed_appletalk_zone	RADIUS_ATR_MAXONE
radius_accounting_request	radius_acct_status_type	RADIUS_ATR_ONE
radius_accounting_request	radius_acct_delay_time	RADIUS_ATR_MAXONE
radius_accounting_request	radius_acct_input_octets	RADIUS_ATR_MAXONE
radius_accounting_request	radius_acct_output_octets	RADIUS_ATR_MAXONE
radius_accounting_request	radius_acct_session_id	RADIUS_ATR_ONE
radius_accounting_request	radius_acct_authentic	RADIUS_ATR_MAXONE
radius_accounting_request	radius_acct_session_time	RADIUS_ATR_MAXONE
radius_accounting_request	radius_acct_input_packets	RADIUS_ATR_MAXONE
radius_accounting_request	radius_acct_output_packets	RADIUS_ATR_MAXONE
radius_accounting_request	radius_acct_terminate_cause	RADIUS_ATR_MAXONE
radius_accounting_request	radius_acct_multi_session_id	RADIUS_ATR_MANY
radius_accounting_request	radius_acct_link_count	RADIUS_ATR_MANY
radius_accounting_request	radius_acct_input_gigawords	RADIUS_ATR_MAXONE
radius_accounting_request	radius_acct_output_gigawords	RADIUS_ATR_MAXONE
radius_accounting_request	radius_event_timestamp	RADIUS_ATR_MAXONE
radius_accounting_request	radius_chap_challenge	RADIUS_ATR_ZERO
radius_accounting_request	radius_nas_port_type	RADIUS_ATR_MAXONE
radius_accounting_request	radius_port_limit	RADIUS_ATR_MAXONE
radius_accounting_request	radius_login_lat_port	RADIUS_ATR_MAXONE
radius_accounting_request	radius_tunnel_type	RADIUS_ATR_MAXONE
radius_accounting_request	radius_tunnel_medium_type	RADIUS_ATR_MAXONE
radius_accounting_request	radius_tunnel_client_endpoint	RADIUS_ATR_MAXONE
radius_accounting_request	radius_tunnel_server_endpoint	RADIUS_ATR_MAXONE
radius_accounting_request	radius_acct_tunnel_connection	RADIUS_ATR_MAXONE
radius_accounting_request	radius_tunnel_password	RADIUS_ATR_ZERO
radius_accounting_request	radius_arap_password	RADIUS_ATR_ZERO
radius_accounting_request	radius_arap_features	RADIUS_ATR_ZERO
radius_accounting_request	radius_arap_zone_access	RADIUS_ATR_ZERO
radius_accounting_request	radius_arap_security	RADIUS_ATR_ZERO

Action	Attribute	Attribute policy
radius_accounting_request	radius_arap_security_data	RADIUS_ATR_ZERO
radius_accounting_request	radius_password_retry	RADIUS_ATR_ZERO
radius_accounting_request	radius_prompt	RADIUS_ATR_ZERO
radius_accounting_request	radius_connect_info	RADIUS_ATR_MANY
radius_accounting_request	radius_configuration_token	RADIUS_ATR_ZERO
radius_accounting_request	radius_eap_message	RADIUS_ATR_ZERO
radius_accounting_request	radius_message_authenticator	RADIUS_ATR_ZERO
radius_accounting_request	radius_tunnel_private_group_id	RADIUS_ATR_MAXONE
radius_accounting_request	radius_tunnel_assignment_id	RADIUS_ATR_MAXONE
radius_accounting_request	radius_tunnel_preference	RADIUS_ATR_ZERO
radius_accounting_request	radius_arap_challenge_response	RADIUS_ATR_ZERO
radius_accounting_request	radius_acct_interim_interval	RADIUS_ATR_ZERO
radius_accounting_request	radius_acct_tunnel_packets_lost	RADIUS_ATR_MAXONE
radius_accounting_request	radius_nas_port_id	RADIUS_ATR_MAXONE
radius_accounting_request	radius_framed_pool	RADIUS_ATR_ZERO
radius_accounting_request	radius_tunnel_client_auth_id	RADIUS_ATR_MAXONE
radius_accounting_request	radius_tunnel_server_auth_id	RADIUS_ATR_MAXONE
radius_accounting_response	radius_user_name	RADIUS_ATR_ZERO
radius_accounting_response	radius_user_password	RADIUS_ATR_ZERO
radius_accounting_response	radius_chap_password	RADIUS_ATR_ZERO
radius_accounting_response	radius_nas_ip_address	RADIUS_ATR_ZERO
radius_accounting_response	radius_nas_port	RADIUS_ATR_ZERO
radius_accounting_response	radius_service_type	RADIUS_ATR_ZERO
radius_accounting_response	radius_framed_protocol	RADIUS_ATR_ZERO
radius_accounting_response	radius_framed_ip_address	RADIUS_ATR_ZERO
radius_accounting_response	radius_framed_ip_netmask	RADIUS_ATR_ZERO
radius_accounting_response	radius_framed_routing	RADIUS_ATR_ZERO
radius_accounting_response	radius_filter_id	RADIUS_ATR_ZERO
radius_accounting_response	radius_framed_mtu	RADIUS_ATR_ZERO
radius_accounting_response	radius_framed_compression	RADIUS_ATR_ZERO
radius_accounting_response	radius_login_ip_host	RADIUS_ATR_ZERO

Action	Attribute	Attribute policy
radius_accounting_response	radius_login_service	RADIUS_ATR_ZERO
radius_accounting_response	radius_login_tcp_port	RADIUS_ATR_ZERO
radius_accounting_response	radius_reply_message	RADIUS_ATR_ZERO
radius_accounting_response	radius_callback_number	RADIUS_ATR_ZERO
radius_accounting_response	radius_callback_id	RADIUS_ATR_ZERO
radius_accounting_response	radius_framed_route	RADIUS_ATR_ZERO
radius_accounting_response	radius_framed_ipx_network	RADIUS_ATR_ZERO
radius_accounting_response	radius_state	RADIUS_ATR_ZERO
radius_accounting_response	radius_class	RADIUS_ATR_ZERO
radius_accounting_response	radius_vendor_specific	RADIUS_ATR_MANY
radius_accounting_response	radius_session_timeout	RADIUS_ATR_ZERO
radius_accounting_response	radius_idle_timeout	RADIUS_ATR_ZERO
radius_accounting_response	radius_termination_action	RADIUS_ATR_ZERO
radius_accounting_response	radius_called_station_id	RADIUS_ATR_ZERO
radius_accounting_response	radius_calling_station_id	RADIUS_ATR_ZERO
radius_accounting_response	radius_nas_identifier	RADIUS_ATR_ZERO
radius_accounting_response	radius_proxy_state	RADIUS_ATR_MANY
radius_accounting_response	radius_login_lat_service	RADIUS_ATR_ZERO
radius_accounting_response	radius_login_lat_node	RADIUS_ATR_ZERO
radius_accounting_response	radius_login_lat_group	RADIUS_ATR_ZERO
radius_accounting_response	radius_framed_appletalk_link	RADIUS_ATR_ZERO
radius_accounting_response	radius_framed_appletalk_network	RADIUS_ATR_ZERO
radius_accounting_response	radius_framed_appletalk_zone	RADIUS_ATR_ZERO
radius_accounting_response	radius_acct_status_type	RADIUS_ATR_ZERO
radius_accounting_response	radius_acct_delay_time	RADIUS_ATR_ZERO
radius_accounting_response	radius_acct_input_octets	RADIUS_ATR_ZERO
radius_accounting_response	radius_acct_output_octets	RADIUS_ATR_ZERO
radius_accounting_response	radius_acct_session_id	RADIUS_ATR_ZERO
radius_accounting_response	radius_acct_authentic	RADIUS_ATR_ZERO
radius_accounting_response	radius_acct_session_time	RADIUS_ATR_ZERO
radius_accounting_response	radius_acct_input_packets	RADIUS_ATR_ZERO

Action	Attribute	Attribute policy
radius_accounting_response	radius_acct_output_packets	RADIUS_ATR_ZERO
radius_accounting_response	radius_acct_terminate_cause	RADIUS_ATR_ZERO
radius_accounting_response	radius_acct_multi_session_id	RADIUS_ATR_ZERO
radius_accounting_response	radius_acct_link_count	RADIUS_ATR_ZERO
radius_accounting_response	radius_chap_challenge	RADIUS_ATR_ZERO
radius_accounting_response	radius_nas_port_type	RADIUS_ATR_ZERO
radius_accounting_response	radius_port_limit	RADIUS_ATR_ZERO
radius_accounting_response	radius_login_lat_port	RADIUS_ATR_ZERO
radius_accounting_response	radius_tunnel_type	RADIUS_ATR_ZERO
radius_accounting_response	radius_tunnel_medium_type	RADIUS_ATR_ZERO
radius_accounting_response	radius_tunnel_client_endpoint	RADIUS_ATR_ZERO
radius_accounting_response	radius_tunnel_server_endpoint	RADIUS_ATR_ZERO
radius_accounting_response	radius_acct_tunnel_connection	RADIUS_ATR_ZERO
radius_accounting_response	radius_tunnel_password	RADIUS_ATR_ZERO
radius_accounting_response	radius_tunnel_private_group_id	RADIUS_ATR_ZERO
radius_accounting_response	radius_tunnel_assignment_id	RADIUS_ATR_ZERO
radius_accounting_response	radius_tunnel_preference	RADIUS_ATR_ZERO
radius_accounting_response	radius_acct_tunnel_packets_lost	RADIUS_ATR_ZERO
radius_access_challenge	radius_user_name	RADIUS_ATR_ZERO
radius_access_challenge	radius_user_password	RADIUS_ATR_ZERO
radius_access_challenge	radius_chap_password	RADIUS_ATR_ZERO
radius_access_challenge	radius_nas_ip_address	RADIUS_ATR_ZERO
radius_access_challenge	radius_nas_port	RADIUS_ATR_ZERO
radius_access_challenge	radius_service_type	RADIUS_ATR_ZERO
radius_access_challenge	radius_framed_protocol	RADIUS_ATR_ZERO
radius_access_challenge	radius_framed_ip_address	RADIUS_ATR_ZERO
radius_access_challenge	radius_framed_ip_netmask	RADIUS_ATR_ZERO
radius_access_challenge	radius_framed_routing	RADIUS_ATR_ZERO
radius_access_challenge	radius_filter_id	RADIUS_ATR_ZERO
radius_access_challenge	radius_framed_mtu	RADIUS_ATR_ZERO
radius_access_challenge	radius_framed_compression	RADIUS_ATR_ZERO

Action	Attribute	Attribute policy
radius_access_challenge	radius_login_ip_host	RADIUS_ATR_ZERO
radius_access_challenge	radius_login_service	RADIUS_ATR_ZERO
radius_access_challenge	radius_login_tcp_port	RADIUS_ATR_ZERO
radius_access_challenge	radius_reply_message	RADIUS_ATR_MANY
radius_access_challenge	radius_callback_number	RADIUS_ATR_ZERO
radius_access_challenge	radius_callback_id	RADIUS_ATR_ZERO
radius_access_challenge	radius_framed_route	RADIUS_ATR_ZERO
radius_access_challenge	radius_framed_ipx_network	RADIUS_ATR_ZERO
radius_access_challenge	radius_state	RADIUS_ATR_MAXONE
radius_access_challenge	radius_class	RADIUS_ATR_ZERO
radius_access_challenge	radius_vendor_specific	RADIUS_ATR_MANY
radius_access_challenge	radius_session_timeout	RADIUS_ATR_MAXONE
radius_access_challenge	radius_idle_timeout	RADIUS_ATR_MAXONE
radius_access_challenge	radius_termination_action	RADIUS_ATR_ZERO
radius_access_challenge	radius_called_station_id	RADIUS_ATR_ZERO
radius_access_challenge	radius_calling_station_id	RADIUS_ATR_ZERO
radius_access_challenge	radius_nas_identifier	RADIUS_ATR_ZERO
radius_access_challenge	radius_proxy_state	RADIUS_ATR_MANY
radius_access_challenge	radius_login_lat_service	RADIUS_ATR_ZERO
radius_access_challenge	radius_login_lat_node	RADIUS_ATR_ZERO
radius_access_challenge	radius_login_lat_group	RADIUS_ATR_ZERO
radius_access_challenge	radius_framed_appletalk_link	RADIUS_ATR_ZERO
radius_access_challenge	radius_framed_appletalk_network	RADIUS_ATR_ZERO
radius_access_challenge	radius_framed_appletalk_zone	RADIUS_ATR_ZERO
radius_access_challenge	radius_chap_challenge	RADIUS_ATR_ZERO
radius_access_challenge	radius_nas_port_type	RADIUS_ATR_ZERO
radius_access_challenge	radius_port_limit	RADIUS_ATR_ZERO
radius_access_challenge	radius_login_lat_port	RADIUS_ATR_ZERO
radius_access_challenge	radius_tunnel_type	RADIUS_ATR_ZERO
radius_access_challenge	radius_tunnel_medium_type	RADIUS_ATR_ZERO
radius_access_challenge	radius_tunnel_client_endpoint	RADIUS_ATR_ZERO

Action	Attribute	Attribute policy
radius_access_challenge	radius_tunnel_server_endpoint	RADIUS_ATR_ZERO
radius_access_challenge	radius_tunnel_password	RADIUS_ATR_ZERO
radius_access_challenge	radius_arap_password	RADIUS_ATR_ZERO
radius_access_challenge	radius_arap_features	RADIUS_ATR_MAXONE
radius_access_challenge	radius_arap_zone_access	RADIUS_ATR_ZERO
radius_access_challenge	radius_arap_security	RADIUS_ATR_MAXONE
radius_access_challenge	radius_arap_security_data	RADIUS_ATR_MANY
radius_access_challenge	radius_password_retry	RADIUS_ATR_ZERO
radius_access_challenge	radius_prompt	RADIUS_ATR_MAXONE
radius_access_challenge	radius_connect_info	RADIUS_ATR_ZERO
radius_access_challenge	radius_configuration_token	RADIUS_ATR_ZERO
radius_access_challenge	radius_eap_message	RADIUS_ATR_MANY
radius_access_challenge	radius_message_authenticator	RADIUS_ATR_MAXONE
radius_access_challenge	radius_tunnel_private_group_id	RADIUS_ATR_ZERO
radius_access_challenge	radius_tunnel_assignment_id	RADIUS_ATR_ZERO
radius_access_challenge	radius_tunnel_preference	RADIUS_ATR_ZERO
radius_access_challenge	radius_arap_challenge_response	RADIUS_ATR_MAXONE
radius_access_challenge	radius_acct_interim_interval	RADIUS_ATR_ZERO
radius_access_challenge	radius_nas_port_id	RADIUS_ATR_ZERO
radius_access_challenge	radius_framed_pool	RADIUS_ATR_ZERO
radius_access_challenge	radius_tunnel_client_auth_id	RADIUS_ATR_ZERO
radius_access_challenge	radius_tunnel_server_auth_id	RADIUS_ATR_ZERO

Table A.3. Default attribute policy in RadiusProxyStrict

A.3. SQL*Net appendix

Example A.1. An example for the SQL*Net connection string

```
No.     Time        Source                Destination           Protocol Info
344 48.463276   127.0.0.1             127.0.0.1             TNS      Request, Connect (1), Connect

Frame 344 (269 bytes on wire, 269 bytes captured)
    Arrival Time: Dec 20, 2005 11:10:58.166023000
    Time delta from previous packet: 0.001255000 seconds
    Time since reference or first frame: 48.463276000 seconds
    Frame Number: 344
    Packet Length: 269 bytes
    Capture Length: 269 bytes
    Protocols in frame: eth:ip:tcp:tns
```

```
Ethernet II, Src: 00:00:00:00:00:00, Dst: 00:00:00:00:00:00
    Destination: 00:00:00:00:00:00 (00:00:00_00:00:00)
    Source: 00:00:00:00:00:00 (00:00:00_00:00:00)
    Type: IP (0x0800)
Internet Protocol, Src Addr: 127.0.0.1 (127.0.0.1), Dst Addr: 127.0.0.1 (127.0.0.1)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 255
    Identification: 0x86bc (34492)
    Flags: 0x04 (Don't Fragment)
        0... = Reserved bit: Not set
        .1.. = Don't fragment: Set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (0x06)
    Header checksum: 0xb53a (correct)
    Source: 127.0.0.1 (127.0.0.1)
    Destination: 127.0.0.1 (127.0.0.1)
Transmission Control Protocol, Src Port: 44404 (44404), Dst Port: 1521 (1521), Seq: 1, Ack: 1, Len:
203
    Source port: 44404 (44404)
    Destination port: 1521 (1521)
    Sequence number: 1    (relative sequence number)
    Next sequence number: 204    (relative sequence number)
    Acknowledgement number: 1    (relative ack number)
    Header length: 32 bytes
    Flags: 0x0018 (PSH, ACK)
        0... .... = Congestion Window Reduced (CWR): Not set
        .0.. .... = ECN-Echo: Not set
        ..0. .... = Urgent: Not set
        ...1 .... = Acknowledgment: Set
        .... 1... = Push: Set
        .... .0.. = Reset: Not set
        .... ..0. = Syn: Not set
        .... ...0 = Fin: Not set
    Window size: 32767
    Checksum: 0xfef3 (incorrect, should be 0x5c50)
    Options: (12 bytes)
        NOP
        NOP
        Time stamp: tsval 35686106, tsecr 35686106
Transparent Network Substrate Protocol
    Packet Length: 203
    Packet Checksum: 0x0000
    Packet Type: Connect (1)
    Reserved Byte: 00
    Header Checksum: 0x0000
    Connect
        Version: 312
        Version (Compatible): 300
        Service Options: 0x0c01
            ..0. .... .... .... = Broken Connect Notify: False
            ...0 .... .... .... = Packet Checksum: False
            .... 1... .... .... = Header Checksum: True
            .... .1.. .... .... = Full Duplex: True
            .... ..0. .... .... = Half Duplex: False
            .... ...0 .... .... = Don't Care: False
            .... .... 0... .... = Don't Care: False
            .... .... ...0 .... = Direct IO to Transport: False
            .... .... .... 0... = Attention Processing: False
            .... .... .... .0.. = Can Receive Attention: False
            .... .... .... ..0. = Can Send Attention: False
        Session Data Unit Size: 2048
        Maximum Transmission Data Unit Size: 32767
        NT Protocol Characteristics: 0x7f08
            0... .... .... .... = Hangon to listener connect: False
            .1.. .... .... .... = Confirmed release: True
            ..1. .... .... .... = TDU based IO: True
            ...1 .... .... .... = Spawner running: True
            .... 1... .... .... = Data test: True
            .... .1.. .... .... = Callback IO supported: True
            .... ..1. .... .... = ASync IO Supported: True
            .... ...1 .... .... = Packet oriented IO: True
            .... .... 0... .... = Can grant connection to another: False
            .... .... .0.. .... = Can handoff connection to another: False
            .... .... ..0. .... = Generate SIGIO signal: False
            .... .... ...0 .... = Generate SIGPIPE signal: False
            .... .... .... 1... = Generate SIGURG signal: True
            .... .... .... .0.. = Urgent IO supported: False
            .... .... .... ..0. = Full duplex IO supported: False
            .... .... .... ...0 = Test operation: False
        Line Turnaround Value: 0
        Value of 1 in Hardware: 0100
        Length of Connect Data: 145
        Offset to Connect Data: 58
        Maximum Receivable Connect Data: 512
        Connect Flags 0: 0x41
            ...0 .... = NA services required: False
            .... 0... = NA services linked in: False
            .... .0.. = NA services enabled: False
            .... ..0. = Interchange is involved: False
            .... ...1 = NA services wanted: True
        Connect Flags 1: 0x41
            ...0 .... = NA services required: False
            .... 0... = NA services linked in: False
            .... .0.. = NA services enabled: False
            .... ..0. = Interchange is involved: False
            .... ...1 = NA services wanted: True
        Trace Cross Facility Item 1: 0x00000000
        Trace Cross Facility Item 2: 0x00000000
        Trace Unique Connection ID: 0x0000660c007e4a09
        Connect Data:
(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=127.0.0.1)(PORT=1521))(CONNECT_DATA=(SERVICE_NAME=OPT.BALASYS)(CID=(PROGRAM=)(HOST=crm)(USER=oracle))))
```

Appendix B. Global options of PNS

PNS has a number of global options and variables that are used during the initialization of the engine, before any proxies or services are started. These options control the swapping of large data chunks (blobs) to disk, the handling of audit trails, and other miscellaneous parameters. To set these options, complete the following steps:

B.1. Procedure – Setting global options of PNS

Step 1. Select the PNS MC component, then select **Variables** > **New**.

Step 2. Enter the name of the global option into the **Name** field, and select the type of the option in the **Type** field.

Step 3. Select **OK**, then **Edit**.

Step 4. Enter the desired value of the option, then select **OK**.

Note
Global options can be also set at the beginning of the `Config.py` file if managing the configuration of PNS manually.

blob

blob

Description

These options control the handling of large data chunks (blobs), determine when the are swapped to disk, and also how much disk space and memory can be used by PNS.

Blob options

`config.blob.temp_directory`	The directory where the blobs are swapped to. Default value: */var/lib/vela/tmp/*
`config.blob.hiwat`	PNS tries to store everything in the memory if possible. If the memory usage of PNS reaches hiwat, it starts to swap the data onto the hard disk, until the memory usage reaches lowat. Default value: *128*0x100000* (128 MB)
`config.blob.lowat`	Global options can be also set at the beginning of the `Config.py` file if managing the configuration of PNS manually. Lower threshold of data swapping. Default value: *96*0x100000* (96 MB)
`config.blob.max_disk_usage`	The maximum amount of hard disk space that PNS is allowed to use. Default value: *1024*0x100000* (1 GB)
`config.blob.max_mem_usage`	The maximum amount of memory that PNS is allowed to use. Default value: *256*0x100000* (256 MB)
`config.blob.noswap_max`	Objects smaller than this value (in bytes) are never swapped to hard disk. Default value: *16384*

audit

audit

Description

These options control the handling of audit trails in PNS.

Audit options

`config.audit.compress`	Enable the compression of audit trail files. The level of compression can be set via the *`config.audit.compress_level`* parameter. Default value: *`TRUE`*
`config.audit.compress_level`	The level of compression ranging from 1 (lowest, default) to 9 (highest). Please note that higher compression levels use significantly more CPU, therefore it is usually not recommended to set it to higher than 4. Default value: *`1`*
`config.audit.encrypt`	Encrypt the audit trail files using the key provided in the *`config.audit.encrypt_certificate`* parameter. Default value: *`FALSE`*

`config.audit.encrypt_certificate`	The X.509 PEM certificate used to encrypt the audit trail files. Default value: empty.
	The certificate should be placed in the following format:
	`-----BEGIN CERTIFICATE-----`
	`insert key here`
	`-----END CERTIFICATE-----`

`config.audit.encrypt_certificate_file`	Name and path of the file containing the X.509 PEM certificate used to encrypt the audit trail files. If this parameter is set, it overrides the settings of *`config.audit.encrypt_certificate`*. Default value: empty.
`config.audit.reopen_size_threshold`	The maximum size of a single audit trail file in bytes. Default value: *`2000000000L`* (2 GB)
`config.audit.per_session`	Store each session in its own audit file. Default value: *`FALSE`*
`config.audit.reopen_time_threshold`	The maximum time frame of a single audit file in seconds. Default value: *`28800`* (8 hours)
`config.audit.rate_limit`	PNS considers it abnormal if the size of an audit trail is increasing faster than this value in byte/second. Default value: *`2097152`* (2 MB)
`config.audit.rate_notification_interval`	Time in seconds before repeating the notification about abnormally growing audit trails. Default value: *`300`* (5 minutes)

`config.audit.write_size_max`	Maximum size of an audit trail file in bytes. Default value: *`52428800`* (50 MB)
`config.audit.terminate_on_max_size`	If set to *`TRUE`*, PNS terminates the connection if the corresponding audit trail file reaches the size limit set in *`config.audit.write_size_max`*. Default value: *`FALSE`*

options

options

Description

These options control various behavior of PNS.

Options

`config.options.dscp_prio_mapping`	Priority mapping for transferring Differentiated Services Code Point (DSCP, also known as Type of Service or ToS). The low (*0*), normal (*1*), high (*2*), and urgent (*3*) priorities can be assigned to the DSCP classes. The assigned priority determines the priority of the PNS thread that handles the connection. The mapping is actually a hash table consisting of the DSCP class ID, a colon (`:`), the priority of the class (*0-3*), and a comma (`,`) except for the last row. For example:

```
config.options.dscp_mapping =  { 1: 3,
                                 2: 2,
                                 3: 2,
                                 4: 0 }
```

`config.options.language`	The default language used to display user-visible messages, e.g., HTTP error pages. Default value: *en* (English). Other supported languages: *de* (German); *hu* (Hungarian).
`config.options.timeout_server_connect`	The timeout (in milliseconds) used when establishing server-side connections. Default value: *30000* (30 sec)

Cache options

PNS caches certain data (e.g., to which zone a particular IP address belongs to) to decrease the time required to process a connection. The following parameters determine the size of these caches (the number of decisions stored in the cache). Adjusting these parameters is required only in environments having very complex zone structure and a large number of services. The following log message indicates that a cache is full: *`Cache over shift-threshold, shifting`*

`config.zone_cache_shift_threshold`	Stores IP addresses and the zone they belong to. Default value: *1000*
`config.inbound_service_cache_threshold`	Stores service-zone pairs, and if the service is permitted to enter the zone. Default value: *1000*
`config.outbound_service_cache_threshold`	Stores service-zone pairs, and if the service is permitted to leave the zone. Default value: *1000*

Appendix C. PNS manual pages

vas

vas — Vela Authentication Server

Synopsis

`vas` [options]

Description

VAS is an authentication server providing authentication services to a Vela based firewall. Its behaviour is controlled by *`vas.cfg(5)`* and `router.cfg`.

Options

`--foreground` or `-F`	Do not daemonize, run in the foreground.
`--no-syslog` or `-l`	Send log messages to the standard output instead of syslog. This option implies foreground mode, overriding the contradicting process options if present.
`--verbose <num>` or `-v <num>`	Set verbosity level to <num>. Valid values are 0-10; default value is 3.
`--log-tags` or `-T`	Enable logging of message tags.
`--log-spec <spec>` or `-s <spec>`	Set verbosity mask on a per category basis. The format of this value is described in *vela(8)*.
`--config <file>` or `-c <file>`	Use <file> as configuration file instead of the default `/etc/vas/vas.cfg`.
`--help` or `-h`	Display a brief help message.

Files

`/etc/vas/`

`/etc/vas/vas.cfg`

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

vas.cfg

vas.cfg — *vas(8)* configuration file.

Description

The *vas.cfg* file controls the operation of Vela Authentication Server.

Structure

The file uses an XML-like format to describe various configuration settings. It uses a configuration/section/<setting> structure where the "name" attribute of the configuration block identifies the VAS subsystem described by the nested tags. The example below sets the global options used by VAS, broken down to three different sections: "log" for log related settings, "router" to set the path to the `router.cfg` file and "ssl" for SSL related settings.

```
<configuration name="vas">
  <section name="log">
    <loglevel>3</loglevel>
    <use_syslog>1</use_syslog>
    <logtags>1</logtags>
  </section>
  <section name="router">
    <router>/etc/vas/router.cfg</router>
  </section>
  <section name="ssl">
    <use_ssl>0</use_ssl>
    <key>/etc/vas/vas.key</key>
    <cert>/etc/vas/vas.crt</cert>
    <verify_mode>0</verify_mode>
  </section>
 </configuration>
```

The VAS plugins (backends) have a slightly different structure. The name attribute in the configuration tag of the VAS plugin and the section name identifies an instance of that plugin. Each instance can be run with a different parameter set. The example below shows a complete configuration block for the *PAM* backend with two instances: `intra` and `internet`:

```
<configuration name="pam">
   <section name="intra">
    <service>vas_intra</service>
    <sleep_time>0</sleep_time>
    <fake_user>0</fake_user>
  </section>
  <section name="internet">
```

```
    <service>vas_internet</service>
    <sleep_time>10</sleep_time>
    <fake_user>1</fake_user>
  </section>
</configuration>
```

The router.cfg file

The `router.cfg` file controls the backend instance selection in VAS. When a new authentication request is initiated by vela(8), VAS selects an authentication backend and an instance based on the meta-information that Vela supplies. Each line in `router.cfg` comprises from a *`condition`* and an *`action`*, separated by whitespace. When an incoming request matches a *`condition`*, the corresponding the *`action`* identifies the authentication backend and its instance to be used.

The *`condition`* is a comma separated list of constraints, each constraint identifying an authentication header and an expected value in the `header=match,header=match,...` format. Wildcard characters like '*' and '?' can be included in the matches.The following headers are currently defined:

`Client-Zone`	The name of the zone the client belongs to.
`Client-IP`	The original IP address of the client initiating the connection to be authenticated.
`Service`	The name of the service the client is authenticating for.

The *`action`* identifies the VAS backend to use (e.g.: *`vas_db`*, *`pam`*, etc.) and the specific instance of that backend. The backend and instance names are separeated by colon (:). Instances are identified by simple names and are used distinguish between various setups of the same backend.

The example below selects the *`intra`* instance of the *`vas_db`* backend. If the configuration block for this backend is not found, or the condition does not match, the *`vas_db:default`* instance is used.

```
Client-Zone=intra     vas_db:intra
vas_db:default
```

Global VAS options

The global configuration options of VAS are described in the *`vas`* configuration block. The related options are grouped into sections. The following options are available:

Section `log`

`use_syslog`	Use syslog for logging.
`logtags`	Enable the logging of message tags.
`loglevel`	Level of verbosity for logging messages. Default value: 3.

Section `bind`

`ip`	IP address to which VAS binds. Default value: *`0.0.0.0`*.
`port`	Port to which VAS binds. Default value: 1317.

Section `ssl`

`use_ssl`	Enable SSL encryption.
`cert`	The certificate file used to authenticate VAS.
`key`	The private key file of the certificate used to authenticate VAS.
`ca_dir`	Path to the directory where the certificates of the trusted CAs are stored.
`crl_dir`	Path to the directory where the certificate revocation lists are stored.
`verify_depth`	The maximum length of the verification chain. Default value: 3.
`verify_mode`	Method how the certificates of the connections incoming to VAS are verified.

0	No certificate is needed.
1	Certificate is optional, but has to be valid if present.
2	A valid certificate is required, untrusted (but valid) certificates are also accepted.
3	A valid, trusted certificate is required.

Section `router`

`router`	Path to the `router.cfg` file.

Section `misc`

`trust_connection`	Permit password-based authentication methods even for unencrypted connections. Default value: *`0`* (false).

Backends

VAS operates using several authentication backends, each with its own set of parameters. Currently the following backends are available:

`vas_db`	Database based backend which currently provides the most features. It has a backing database (called "storage") and a set of authentication methods (called "methods"). The name of the configuration block is *vas_db*
`pam`	Authenticates users against the local PAM libraries on the host running VAS itself. The name of the configuration block is *pam*.
`htpass`	Authenticates users against an Apache htpasswd style password file. The name of the configuration block is *htpass*.
`radius`	Authenticates users against a RADIUS server. The name of the configuration block is *radius*.

`tacacs`	Authenticates users against a TACACS+ server. The name of the configuration block is *tacacs*.

All backends are capable of authentication faking. This is a method to hide the valid usernames, so that they cannot be guessed (for example using brute-force methods). If somebody tries to authenticate with a non-existing username, the attempt is not immediately rejected: the full authentication process is simulated (e.g.: password is requested, etc.), and rejected only at the end of the process. That way it is not possible to determine if the username itself was valid or not.

The Vas_db backend

The vas_db backend interprets the following parameters in its configuration block.

`storage`	Specifies the database plugin to use. Currently only the *ldap* database is supported.
`methods`	Specifies a space separated list of enabled authentication methods. The following authentication plugins are available: *passwd*, *skey*, *rb1*, *x509*, *ldapbind*, and *none*.
`fake_user`	Enables authentication faking.
`fake_user_name`	Specifies a user name which is used for faking authentication. This has to be an existing user name, used exclusively for this purpose.
`sleep_time`	Wait at least that many seconds after a failed authentication attempt.

Storage plugins

The *vas_db* backend authenticates against an abstract database, the actual implementation is specified using the *storage* parameter. The only storage plugin currently supported is *ldap*.

`ldap`	The *ldap* storage plugin uses the Lightweight Directory Access Protocol (LDAP) to access a directory based database. It has a separate configuration block identified by the name *vas_db_storage_ldap*.

The LDAP storage plugin

The LDAP storage plugin connects to an LDAP server, authenticates using a user-independent, service account and runs queries against the database to provide a *vas_db* dependent view on the directory. It uses a VAS specific LDAP scheme available in the vas package.

`use_ssl`	Enables SSL/TLS when connecting to the LDAP server.
`hostname`	Specifies the LDAP host to use.
`port`	Specifies port of the LDAP server to use.
`bind_dn`	Bind to this DN before accessing the database.
`bind_pw`	Use this password when binding to LDAP.

`base_dn`	Perform queries using this base DN.
`filter`	Search for an account using this filter expression. Defaults to '(uid=%u)'; %u is expanded to the username being searched for.
`scope`	Specifies the scope of the search. *`base`*, *`sub`*, and *`one`* are acceptable values, specifying LDAP_SCOPE_BASE, LDAP_SCOPE_SUB, and LDAP_SCOPE_ONE, respectively.
`user_is_dn`	Specify that the incoming username is a fully qualified DN.
`scheme`	Specify LDAP scheme to use: *`posix`* for POSIX, *`ad`* for ActiveDirectory, or *`nds`* for Novell eDirectory/NDS style directory layout.
`ldapbind_description`	When the *`ldapbind`* authentication method is used for authentication, the value of this string is returned as method description to the user. NOTE: This parameter is OBSOLETE, it must be set in the *`ldapbind`* authentication method.
`usercert_description`	When the directory contains user keys in the userCertificate attribute and it is used for X.509 based authentication, the value of this string will be returned as method description to the user. OBSOLETE. Set it in x509 authentication method.
`follow_referral`	If this option is set, VAS will respect the referral response from the LDAP server when looking up a user.

Authentication method plugins

The *vas_db* backend is general enough to allow the use of several different authentication methods. The set of permitted authentication methods is defined using the *`methods`* configuration option as described in the previous section. All pligins have a *`priority`* attribute. This attribute is used by the Authentication Agent client: the authentication methods available to the user are displayed in the order of the priority (starting with the highest value).

The following method plugins are available:

`passwd`	Implements password authentication. Password authentication is available only if the connection between Vela and VAS is secure. The name of the configuration block is *vas_db_method_passwd*. The password authentication method has the following parameters: `priority` — Priority of the authentication type.
`skey`	Implements S/Key authentication. The name of the configuration block is *vas_db_method_skey*. The S/Key authentication method has the following parameters: `priority` — Priority of the authentication type.

`rb1`	Implements CryptoCard RB1 hardware token based authentication. The name of the configuration block is *vas_db_method_rb1*. The RB1 authentication method has the following parameters:	
	`priority`	Priority of the authentication type.
`x509`	Implements X.509 certificate based authentication. The name of the configuration block is *vas_db_method_x509*. The X.509 authentication method has the following parameters:	
	`compare_cert`	Compare the stored certificate bit-by-bit to the certificate supplied by the client. The authentication will fail when the certificates do not match, even if the new certificate is trusted by the CA. Default value: 1 (TRUE).
	`trusted_ca_list`	Send a list of trusted certificates to the client to choose from to narrow the list of available certificates. Default value: 1 (TRUE).
	`verify_cert`	Verify the validity of the certificate (i.e. the certificate has to be issued by one of the trusted CAs and not revoked). This is verification is independent from the *`compare_cert`* setting, so if both parameters are set, both conditions must be fulfilled to accept the certificate. Default value: 1 (TRUE).
	`ca_locations`	A list of space separated URLs to the trusted CAs. The *`file://`* and *`ldap://`* URLs are supported.

`crl_locations`	A list of space separated URLs to the CRLs issued by the trusted CAs. The `file://` and `ldap://` URLs are supported.
`verify_depth`	The maximum length of the verification chain.
`priority`	Priority of the authentication type.

`ldapbind`

Implements authentication against the target LDAP server. Only password authentication is supported by this method, therefore it is only available if the connection between VAS and Vela is secured with SSL. The name of the configuration block is *vas_db_method_ldapbind*.
The LDAP authentication method has the following parameters:

`priority`	Priority of the authentication type.
`description`	The value of this string is returned as method description to the user.

`none`

Implements NO authentication. This method accept every authentication request if the user is exists in the database. The main advantage of this method is when the authentication is done somwhere outside of this program but the groups information is needed. The name of the configuration block is *vas_db_method_none*.
The None authentication method has the following parameters:

`priority`	Priority of the authentication type.
`description`	The value of this string is returned as method description to the user.

`gssapi`

Implements GSSAPI based authentication. NOTE: The Kerberos5 keytab file to be used can be specified via the standard *`KRB5_KTNAME`* environment variable. The name of the configuration block is *vas_db_method_gssapi*.
The gssapi authentication method has the following parameters:

`priority`	Priority of the authentication type.
`description`	The value of this string is returned as method description to the user.

`principal_name`	Specifies the GSSAPI principal name which this authentication service represents. Make sure that the keys associated with this principal are present in */etc/krb5.keytab*. Changing the keytab location is currently not possible.

The PAM backend

The PAM backend implements authentication based on the local authentication settings of the host running VAS. It basically authenticates the users against the local PAM installation and/or using GSSAPI/krb5. The PAM backend has the following parameters:

`use_local_accounts`	Use the local passwd/group database to query group membership of a given account. The Name Service Switch can also be used, so integrating other naming services is possible. Defaults value: 0 (FALSE).
`enable_pam_auth`	Enable PAM authentication. Default value: 1 (TRUE).
`pam_service`	Specifies the PAM service to use for authentication. This option is an alias for the now deprecated *service* option. Defaults value: *vas*.
`enable_gssapi_auth`	Enable GSSAPI/krb5 authentication in this backend. Defaults value: 0 (FALSE). NOTE: The Kerberos5 keytab file to be used can be specified via the standard *KRB5_KTNAME* environment variable.
`gssapi_princ_name`	Specifies the GSSAPI principal name which this authentication service represents. Make sure that the keys associated with this principal are present in `/etc/krb5.keytab`. Changing the keytab location is currently not possible.
`description`	The value of this string is returned as method description to the user.
`fake_user`	Enables authentication faking.
`sleep_time`	Wait at least that many seconds after a failed authentication attempt.

The Htpass backend

The htpass backend has the following parameters:

`filename`	The file to be read as password file. The file should contain two columns separated by colon (':'), with the first column containing

	the username, the second the password encrypted by *crypt(3)* function. This file can be created/maintained by the Apache *htpasswd(1)* utility.
`fake_user`	Enables authentication faking.
`sleep_time`	Wait at least that many seconds after a failed authentication attempt.

The Radius backend

The Radius backend has the following parameters:

`hostname`	The hostname of the RADIUS server.
`hostport`	The port of the RADIUS server.
`secret`	The shared secret between the authentication server and VAS.
`description`	The value of this string is returned as method description to the user.
`fake_user`	Enables authentication faking.
`sleep_time`	Wait at least that many seconds after a failed authentication attempt.

The TACACS+ backend

The TACACS backend has the following parameters:

`hostname`	The hostname of the TACACS+ server.
`hostport`	The port of the TACACS+ server, defaults to 49.
`secret`	The shared secret between the authentication server and VAS.
`description`	The value of this string is returned as method description to the user.
`fake_user`	Enables authentication faking.
`sleep_time`	Wait at least that many seconds after a failed authentication attempt.

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

vcf

vcf — Vela Content Vectoring Server

Synopsis

`vcf` [options]

Description

The Vela Content Vectoring Server (VCF) is a content scanning framework providing stream and file scanning services for *vela(8)*. VCF runs as a separate application and can be accessed over TCP, UNIX domain sockets and standard input and output file handles. The behaviour of VCF can be controlled via the *`vcf.cfg(5)`* configuration file.

Options

`--verbose <verbosity>` or `-v <verbosity>`	Set verbosity level to <verbosity>, or if <verbosity> is omitted increment it by one. Default the verbosity level is 3; possible values are 0-10.
`--no-syslog` or `-l`	Send log messages to the standard output instead of syslog. This option implies foreground mode, overriding the contradicting process options if present.
`--log-spec <spec>` or `-s <spec>`	Set verbosity mask on a per category basis. The format of this value is described in *vela(8)*.
`--log-tags` or `-T`	Enable logging of message tags.
`--foreground` or `-F`	Do not daemonize, run in the foreground.
`--help` or `-h`	Display a brief help message.
`--vela-mode <ctrl-fd>` or `-z <ctrl-fd>`	Start in Vela mode using the <ctrl-fd> file descriptor and remain in the foreground. In this mode only a single scan is performed on the data on the standard input. Results are sent to the standard output. (Naturally, log messages are not sent to the standard output in this mode, as this would interfere with the scanning results.) This mode is used mainly for testing purposes.
`--rule-group <rule-group>` or `-R <rule-group>`	The value for the vcf_rule_group routing variable in Vela mode.
`--config <file>` or `-c <file>`	Use the configuration file <file> instead of the default `/etc/vcf/vcf.cfg` file.
`--pidfile <file>` or `-P <file>`	Use <file> as pid file instead of the default `/var/run/vcf/vcf.pid` file.

Operation

VCF scans the contents of incoming streams. VCF has multiple channels, each performing a possibly different set of actions on the incoming stream. These channels are called "scanpaths", i.e. a scanpath is an ordered set of modules and their associated settings. The scanpath to be used is selected based on meta information provided by Vela and meta information gathered about the stream by VCF itself. This scanpath selection mechanism is called "routing decision" and is controlled by the router rules.

To summarize, VCF operates as follows: A connection is established between Vela and VCF. VCF selects a scanpath (i.e. makes the routing decision) based the collected information, the router rules and information received from Vela. The scanpath determines the modules to use and their associated settings. After the modules process the data received in the stream, the result of the scanning operation is sent back to Vela.

Files

`/etc/vcf/`

The routing and configuration file formats are described in `/etc/vcf/vcf.cfg` and `/etc/vcf/router.cfg`.

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

vcf.cfg

vcf.cfg — *vcf(8)* configuration file format

Description

The `vcf.cfg` file controls the operation of VCF, the Vela Content Vectoring Server.

Structure

`vcf.cfg` uses an XML-like format to describe various configuration settings. The exact structure is *`configuration/section/<setting>`*, where the "name" attribute of the configuration block identifies the VCF subsystem described by the nested tags.

The main configuration blocks of the file are the following:

`vcf`	Global options of `vcf`.
`scanpaths`	Definitions and settings of the scanpaths.
`nod32, html, etc.`	Definitions and instance-specific settings of the modules.
`module-options`	Global settings of the modules that apply to every instance of the module.

The example below sets the global options used by VCF, broken down to three different sections: *log* for log related settings, *router* for setting the path to the `router.cfg` file and *misc* for miscellaneous parameters.

```
<configuration name="vcf">
  <section name="log">
  <loglevel>3</loglevel>
  <use_syslog>true</use_syslog>
  <logtags>true</logtags>
</section>
<section name="router">
  <router>/etc/vcf/router.cfg</router>
</section>
<section name="misc">
  <magic_length>2048</magic_length>
</section>
</configuration>
```

The VCF modules have a slightly different structure. The name attribute in the configuration tag of the VCF module and the section name identifies an instance of that module. Each instance can be run with a different parameter set. The example below shows a complete configuration block for the *clamav* module, with an instance named *intranet* having normal, and another named *internet* having paranoid sensitivity.

```
<configuration name="clamav">
  <section name="internet">
```

```
    <mode>file</mode>
    <scan_packed>1</scan_packed>
    <disinfect>0</disinfect>
    <scan_suspicious>1</scan_suspicious>
    <heuristic_level>normal</heuristic_level>
  </section>
  <section name="intranet">
    <mode>file</mode>
    <scan_packed>1</scan_packed>
    <disinfect>1</disinfect>
    <scan_suspicious>0</scan_suspicious>
    <heuristic_level>normal</heuristic_level>
  </section>
</configuration>
```

The router.cfg file

The `router.cfg` file controls the scanpath selection in VCF. VCF selects the scanpath based on the meta-information that Vela supplies. Each line in `router.cfg` comprises from a *`condition`* and an *`action`*, separated by whitespace. When an incoming request matches a *`condition`*, the corresponding the *`action`* identifies the scanpath and its instance to be used.

The *`condition`* is a comma separated list of constraints, each constraint identifying a variable and an expected value in the `header=match,header=match,...` format. Wildcard characters like '*' and '?' can be included in the matches.The following variables are currently defined:

`vcf_rule_group`	The name of the rule group that the peer requests. Its value is specified the -R command line option in Vela mode, or is supplied by the peer during the handshake.
`content_type_detected`	MIME type detected based on the first bytes of the file.
`content_type_uncompressed`	MIME type detected based on the first bytes of the file looking into a compressed file header and decompressing it if necessary.
`content_type`	MIME type as specified by the peer.
`file_name`	File name or URL.
`file_extension`	File extension. Please note that this information might not be accurate as some URLs do not contain file extension in which case this variable is empty. For example it is common to reference directories in HTTP which implicitly map to a server defined content and the URL does not contain a filename extension as in http://domain.com/directory/. It is better to use content_type or content_type_detected for content specific scanning.
`file_xfer_direction`	File transfer direction, either "upload" or "download".
`vela_protocol`	Protocol that was used to transfer the checked file.

`vela_session_id`	Vela session id that requested content vectoring.
`vela_proxy_class`	The name of the proxy class that requested content vectoring.
`vela_auth_user`	The authenticated username.
`vela_client_address`	Client address in AF_INET(>ipaddr<, >port<) format.
`vela_client_address.ip`	Client IP address.
`vela_client_address.port`	Client TCP/UDP port.
`vela_client_zone`	The name of the client zone.
`vela_server_address`	Server address in AF_INET(>ipaddr<, >port<) format.
`vela_server_address.ip`	Server IP address.
`vela_server_address.port`	Server TCP/UDP port.
`vela_server_zone`	The name of the server zone.
`smtp_envelope_sender`	The envelope sender address in SMTP.
`smtp_envelope_recipients`	Space separated list of envelope recipient addresses in SMTP.
`http_request_method`	The type of the HTTP request.
`http_request_url`	The HTTP request URL.
`http_request_version`	The version of the HTTP request (e.g. 1.1).
`http_request_host`	The Host header included in the HTTP request.

Furthermore, virtually all defined Vela variables can be used as variables with the 'vela.' prefix, which denotes the 'session' object of the stacking proxy. For example: > *`vela.session_id`*, *`vela.client_address.ip_s`*, etc.

The *`action`* identifies the VCF scanpath to use.

The example below selects the *`html`* scanpath for all files which are recognized as "text/html" files, and rejects everything else. An object is scanned only by the scanpath of the first matching condition.

```
content_type="text/html" html
content_type_detected="text/html" html
REJECT
```

Global Options

Global options are stored in the configuration block named *vcf*. Related options are grouped into sections.

Section `log`

`use_syslog`	Use syslog for logging.
`logtags`	Enable the logging of message tags.
`loglevel`	Level of verbosity for logging messages. Default value: 3.

`logspec`	Set verbosity mask on a per category basis. The format of this value is described in *vela(8)*.

Section `misc`

`magic_length`	This parameter determines the amount of data (in bytes) read from MIME objects to detect their MIME-type. Higher value increases the precision of MIME-type detection. Default value: 0.
`tempdir`	Location of the temporal directory (used for swap files, etc.). Default value: `/var/lib/vela/tmp`

Section `router`

`router`	Location of the `router.cfg` file. Default value: `/etc/vcf/router.cfg`

Section `bind`

`ip`	IP address to which VCF binds. Default value: *`0.0.0.0`*.
`port`	Port to which VCF binds. Default value: *`1318`*.
`unix`	Bind to a unix domain socket. If only the empty tag is present, the default socket (`/var/run/vcf/vcf.sock`) is used.

When binding to a unix domain socket, the owner and the permissions of the socket can be set using the following parameters:

`owner`	The owner of the socket. By default its value is *`NULL`*, meaning that the owner of the socket is the user running VCF.
`group`	The owner group of the socket. Default value: *`velastate`*.
`perm`	The permission settings of the socket in Unix-style. Default value: *`770`*.

Section `blob`

`hiwat`	VCF tries to store everything in the memory if possible. If the memory usage of VCF reaches *`hiwat`*, it starts to swap the data onto the hard disk, until the memory usage reaches *`lowat`*. Default value: 960 *1024 *1024 (960 MB).
`lowat`	Lower threshold of data swapping. Default value: 640 *1024 *1024 (640 MB).
`max_disk_usage`	The maximum amount of hard disk space that VCF is allowed to use. Default value: 64 * 1024 * 1024 * 1024 (64 GB).
`max_mem_usage`	The maximum amount of memory that VCF is allowed to use. Default value: 1024 * 1024 * 1024 (1 GB).
`noswap_max`	Objects smaller than this value (in bytes) are never swapped to hard disk. Default value: 16384.
`deadlock_check_period`	The period of deadlock check in seconds, to resolve deadlock, when storage is full. Default value: 5.

`allocation_timeout`	The time in seconds of waiting for blob allocation, if storage is full. Default value: 10.

Scanpath Options

The scanpath options are stored in the configuration block named *scanpaths*. Each section in this block has the name of a scanpath and contains settings specific for the given scanpath.

Settings to control *trickling* can also be configured here. Content filtering cannot be performed on partial files: the entire file has to be available on the firewall. Sending of the file to the client is started only if no virus was found (or the file was successfully disinfected). Instead of receiving the data in a continuous stream, as when connecting to the server “regularly”, the client does not receive any data for a while, then “suddenly” it starts to flow. This phenomena is not a problem for small files, since these are transmitted and checked fast, probably without the user ever noticing the delay, but can be an issue for larger files when the client application might time out. Another source of annoyance can be when the bandwidth of the network on the client and server side of the firewall is significantly different. In order to avoid time outs, a solution called *trickling* is used. This means that the firewall starts to send small pieces of data to the client so it feels that it is receiving something and does not time out. For further information on trickling, see the *Virus filtering and HTTP Technical White Paper* available at the BalaSys Documentation Page at http://www.balasys.hu/en/documentation/

The following options are available for each scanpath:

`plugins`	Comma-separated list of colon separated pairs listing the modules to be executed in the scanpath. The colon-separated pairs specify the module and its instance (e.g.: *`html:filterscripts, nod32:paranoid`*).	
`quarantine_mode`	Quarantine mode to be used. Always the original file is quarantined.	
	always	Quarantine all objects rejected for any reason.
	rejected	Quarantine objects that could not be disinfected.
	modified+rejected	Quarantine only the original version of the files which were successfully disinfected. E.g.: if an infected object is found but it is successfully disinfected, the original (infected) object is quarantined. That way, the object is retained even if the disinfection eliminates some important information.
	never	Disable quarantining, objects rejected for any reason are dropped.
`threshold_oversize`	Objects larger than *`threshold_oversize`* (in bytes) are not scanned, because of performance/resource reasons (i.e. large archives, ISO files, etc.).	
`trickle_mode`	Mode of trickling to be used. Default: NONE.	
	none	Trickling is disabled.
	percent	Determine the amount of data to be trickled based on the size of the object. Data is sent to the client only when VCF

		receives new data; the size of the data trickled is the set percentage of the total data received so far. This is the recommended method to use.
	steady	Trickle fixed amount of data in fixed time intervals.
`trickle_percent`	Amount of data to be trickled (percentage). Defailt value: 10.	
`trickle_steady_initial_delay`	When an object is downloaded, trickling is started after this period (in seconds). Default value: 10.	
`trickle_steady_delay`	Period (in seconds) between trickling data chunks.	
`trickle_steady_bytes`	Amount of data (in bytes) that is sent to the client in a chunk during trickling. Default value: 128 bytes.	

Modules

The following modules are available in VCF:

sed	Filters and rewrites the input in stream similarly to the operation of the UNIX 'sed' command.
nod32	Performs virus scanning on the incoming data with the NOD32 engine. The data is processed in file mode.
clamav	Performs virus scanning on the incoming data with the Clam AntiVirus engine. The data is processed in file mode.
html	Performs JavaScript/Java/ActiveX filtering of HTML data in stream mode.
spamassassin	Performs spam filtering on the incoming e-mails with the SpamAssassin engine. The data is processed in file mode.
mail-hdr	Performs filtering and manipulation on the headers of e-mail messages. The data can be processed both in file and stream mode.
modsecurity	ModSecurity is a platform-independent web application level security gateway module (Web Application Firewall (WAF)), that can be integrated to Vela Gateway. ModSecurity's WAF solution can look into the HTTP(S) traffic and provides a powerful policy definition language and an API to achieve advanced security.
`program`	Performs filtering and/or manipulation of the data with an external 3rd-party application. The data can be processed either in file and stream mode.

The Sed module

The configuration name of the sed module is *sed*. This module has the following instance-specific options:

`filter`	The stucture of this string is the following: a slash (/), the string to be replaced, a slash (/), the replacement string, and the options.

Slashes in the string have to be escaped with backslashes.The folowing options are available:

-g	Replace all occurances of the string.
-i	Run in case insensitive mode.

For example, the `/example/sample/-g` filter replaces all occurances of 'example' to 'sample'.

The NOD32 module

The nod32 module has the following instance-specific options:

`scan_packed`	Perform virus scanning on archived files. Default value: YES.
`scan_suspicious`	Perform virus scanning on suspicious files (e.g.: suspicious files are often new variants of known viruses). Default value: NO.
`heuristic_level`	Level of heuristic sensitivity. The available levels are OFF, NORMAL, and HIGH. Default value: OFF.
`archive_max_size`	Archives larger than the specified value (in megabytes) are not scanned. Zero means unlimited. Default value: 10.

The clamav module

The configuration name of the Clam AntiVirus module is *clamav*. The module has the following module options:

`daemon_socket`	The domain socket used to communicate with the clamav engine. Default value: `/var/run/clamav/clamd.ctl`

The clamav module has the following instance-specific options:

`scan_packed`	Perform virus scanning on archived files. Default value: YES.

The SpamAssassin module

The configuration name of the SpamAssassin module is *spamassassin*. The module has the following instance-specific options::

`check_only`	Only check the e-mails, but do not make any modification to the e-mail. The result of the spam filtering is returned to VCF separately. Default value: FALSE.
`host` and `port`	The hostname and port number of the machine SpamAssassin is running on, if different from the VCF host.
`socketpath`	The domain socket used to communicate with SpamAssassin if it is running on the VCF host. Default value: `/var/run/spamassassin.sock`
`username`	The user under which SpamAssassin should filter e-mails. Default value: not set, the user running SpamAssassin is used (*`nobody`*).

`timeout`	Timeout value for the scanning requests in seconds. Default value: *`60`*. **Note** If the timeout is set to -1 (unlimited), then no timeout is used for the connection if SpamAssassin is running on a remote host.
`threshold`	By default, VCF rejects all e-mails SpamAssassin detects as spam. However, to minimize the impacts of false positives, if the spam status of an e-mail (as calculated by SpamAssassin) is over the *`required_score`* (default value: *`5`*), but below the value set in *`threshold`*, VCF only marks the e-mail as spam, but does not reject it. If the spam status of an e-mail is above the *`threshold`*, it is automatically rejected. Default value: *`10.0`*.

The HTML module

The configuration name of the html module is *html*.

The html module has the following instance-specific options:

`filter_javascript`	Remove javascript from HTML pages. Default value: NO. Enabling this option removes all *`javascript`* and *`script`* tags, and the conditional value prefixes (e.g.: *`onclick`*, *`onreset`*, etc.).
`filter_activex`	Remove ActiveX components from HTML pages. Default value: NO. Enabling this option removes the *`applet`* tags and the *`classid`* value prefix.
`filter_java`	Remove java from HTML pages. Default value: NO. Enabling this option removes the *`java:`* and *`application/java-archive`* inclusions, as well as the *`applet`* tags.
`filter_css`	Remove CSS (cascading style sheets) from HTML pages. Default value: NO. Enabling this option removes the single *`link`* tags, the *`style`* tags and options, as well as the *`class`* options.
`filter_custom`	A whitespace-separated value of colon separated pairs, specifying the headers, tags, etc. to be removed based on their names or their values. The following HTML elements can be filtered: *Tags*: Remove everything between the specified tag and its closing tag. Embedded structures are also handled. E.g.: *`closed-tag:ul`* *Single tags*: Remove all occurrences of the specified single tag (*`img`*, *`hr`*, etc.). E.g.: *`tag:hr`*

	Options: Remove options (e.g.: `width`, etc.) and their values. E.g.: `option:width` *Prefixes*: Remove all options starting with the set prefix. E.g.: `prefix:on` will remove all options like `onclick`, etc.
`buffer_size`	This attribute control the size of the internal buffer of this module

The mail header module

The configuration name of the mail header module is *mail-hdr*. A filter contains a pattern (i.e. the header line to be found) enclosed within backslashes (`/`), a whitespace, the action to be performed on the header line, and an optional argument. The pattern and the argument can be regular expressions. To search for the pattern in case insensitive mode, add an `i` character after the closing backslash of the pattern. The following actions can be performed on the mail headers:

- *Append*: Add the argument of the filter as a new header line after the match.
- *Discard*: Discard the entire e-mail message. The argument is returned to the mail server sending the message as an error message.
- *Ignore*: Remove the matching header line from the message.
- *Pass*: Accept the matching header line. This action can be used to create exceptions from other filter rules.
- *Prepend*: Add the argument of the filter as a new header line before the match.
- *Reject*: Reject the entire e-mail message. The argument is returned to the sender of the message as an error message.
- *Replace*: Replace the mathing header line to the argument of the filter.

The module has the following instance-specific options::

`filter` — The list of filters to be applied on the mail headers. For example:

```
<filter>
  /^Subject: hello$/i       DISCARD
  /^Date: (.*)/              APPEND "X-Date: \1 \1"
<filter>
```

`header_wrap_length`	If a manipulated header line is longer than this value (in bytes), is will be broken into a new lines. These new lines will not be longer then `header_wrap_length`. Default value: .
`max_line_length`	This attribute control the maximum length of a header line

The Modsecurity module

The configuration name of the Modsecurity module is *modsecurity*. The module has the following module options:

`config_file`	Ruleset configuration file for ModSecurity. Default value: not set.
`transaction_timeout`	Timeout value for transactions, in seconds. Default value: 60.
`process_request_body`	Request bodies will be buffered and processed by ModSecurity. Default value: YES.
`process_response_body`	Response bodies will be buffered by ModSecurity. Default value: YES.
`request_body_limit`	The maximum request body size ModSecurity will accept for buffering, in KBytes. Anything over the limit will be rejected with status code 413 (Request Entity Too Large). Default value: 10000.
`request_body_no_files_limit`	The maximum request body size ModSecurity will accept for buffering in KBytes, excluding the size of any files being transported in the request. Default value: 1000.
`response_body_limit`	The maximum response body size that will be accepted for buffering, in KBytes. Anything over this limit will be rejected with status code 500 (Internal Server Error). Default value: 1000.

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

vms

vms — Vela Management Server engine

Synopsis

`vms` [options]

Description

vms is the server component of Vela Management System, which is a distributed management system for Vela firewalls. The server component is responsible for storing configuration data, distributing keys and certificates, and accepting administrator changes via the VMC graphical user interface.

Options

`--version` or `-V`	Display version information.
`--foreground` or `-F`	Do not daemonize, run in the foreground.
`--no-syslog` or `-l`	Send log messages to the standard output instead of syslog. This option implies foreground mode, overriding the contradicting process options if present.
`--verbose <level>` or `-v <level>`	Set the verbosity level of logging to <level>. Default value: 3.
`--tags` or `-t`	Enable tag logging.
`--config <file>` or `-c <file>`	Use the configuration file <file> instead of the default file.
`--log-spec <spec>` or `-s <spec>`	Set verbosity mask on a per category basis. The format of this value is described in *vela(8)*.
`--bootstrap` or `-b`	Bootstrap the engine.
`--help` or `-h`	Display a brief help message.

Files

Configuration information for `vms(8)` is stored in `/etc/vms/vms.conf`.

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

Copyright © 1996-2024 Balasys IT Zrt. All rights reserved.

vms.conf

vms.conf — Configuration file format for the Vela Management Server (*vms(8)*).

Description

The `vms.conf` file controls the operation of the Vela Management Server *vms(8)*. It is rarely needed to modify the configuration manually.

WARNING: The settings stored in `vms.conf` are managed by the VMS engine via VMC. Do not modify the settings manually unless you know exactly what you are doing.

Structure

`vms.conf` uses an XML-like format to describe various configuration settings. The exact structure is *`configuration/section/<setting>`*, where the "name" attribute of the configuration block identifies the subsystem described by the nested tags.

Main configuration blocks are found in the *`default`* configuration block, with related options grouped into sections such as *`global`*, *`log`*, and *`ssl`*.

WARNING: The settings stored in the `vms.conf` file are used internally within Vela; the structure of the file and the individual options may change between the different Vela releases.

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

vms-integrity

vms-integrity — VMS Database Integrity Checker

Synopsis

`vms-integrity` [options]

Description

`vms-integrity` is a tool for checking the integrity of the VMS database. It can also be used for recovery if the database contains errors.

Options

`--datadir <dir>` or `-d <dir>`	Database directory. Default value: `/var/lib/vms`.
`--recover` or `-r`	Try to recover database if integrity check fails. If not set, the integrity check will only report the errors found.
`--syslog` or `-l`	Use syslog for logging.
`--verbose <level>` or `-v <level>`	Set the verbosity level of logging to <level>. Default value: 3.
`--quiet` or `-q`	No logging, only exit status is set.
`--help` or `-h`	Display a brief help message.

Example

When restoring an earlier VMS database and the process fails, the `vms-integrity` can fix the database backup archive file. To recover a `vms-backup-<timestamp>.tar.gz` formatted file, complete the following steps.

Create a temporary working directory	mkdir /tmp/vms-backup
Unpack the archive file	tar -zxf <backup-file-to-restore> -C /tmp/vms-backup
Try to recover the database	/usr/sbin/vms-integrity -r -d /tmp/vms-backup
Check the recovered database	/usr/sbin/vms-integrity -d /tmp/vms-backup
Pack the database	tar -C /tmp/vms-backup -czf <fixed-backup-file>
Delete the working directory	rm -rf /tmp/vms-backup

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

instances.conf

instances.conf — *vela(8)* instances database

Description

The `instances.conf` file describes the _*`vela(8)`*_ instances to be run on the system. It is processed by *`velactl(8)`*_ line by line, each line having the structure described below. Empty lines and lines beginning with '#' are comments ignored by `velactl`.

Structure

```
instance-name parameters [-- velactl-options]
```

instance-name is the name of the Vela instance to be started; it is passed to `vela` with its `--as` parameter. Instance names may consist of the characters [a-zA-Z0-9_] and must begin with a letter.

parameters are space separated parameters entered into the vela command-line. For details on these command-line parameters see *vela(8)*.

velactl-options are space separated parameters control startup specific options. They are processed by `velactl` itself. The following `velactl` options are available:

`--auto-restart` or `-A`	Enable the automatic restart feature of `velactl`. When an instance is in auto-restart mode, it is restarted automatically in case the instance exits.
`--no-auto-restart` or `-a`	Disable automatic restart for this instance.
`--fd-limit <number>` or `-f <number>`	Set the file descriptor limit to <number>. The file descriptor limit defaults to the number of threads (specified by the --*threads* parameter of *vela(8)*) multiplied by 4.
`--num-of-processes <number>` or `-P <number>`	Run <number> of processes for the instance. `velactl` starts exactly one Vela process in master mode and <number> of slave Vela processes. This mode of operation is incompatible with old-style dispatchers, you must use the new rule-based policy with this option.

Examples

```
vela_ftp --policy /etc/vela/policy.py --verbose 5
```

The line above describes a Vela instance named *vela_ftp* using policy file */etc/vela/policy.py*, and having verbosity level 5.

```
vela_intra -v4 -p /etc/vela/policy.py --threads 500 --no-auto-restart --fd-limit
1024 --process-limit 512
```

This line describes a vela instance named *vela_intra* using the policy file `/etc/vela/policy.py`, verbosity level 4. The maximum number of threads is set to 500, file descriptor limit to 1024, process limit to 512.

Files

The default location of `instances.conf` is `/etc/vela/instances.conf`. Defaults for velactl tunables can be specified in `/etc/vela/velactl`.

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

policy.py

policy.py — *vela(8)* policy file.

Description

The `policy.py` file is a Python module containing the zone and service definitions and other policy related settings used by *`vela(8)`*. Empty lines and lines beginning with '#' are comments and are ignored.

The `policy.py` file is generated automatically by VMC, the Vela Management Console, or it can be edited manually.

IMPORTANT: Do not edit manually a file generated by VMC, because the manual changes will not be retained by VMC and will be lost when re-generating the file.

Files

The default location of `policy.py` is `/etc/vela/policy.py`.

See Also

For further information on `policy.py` refer to the following sources:

A tutorial on manually editing the `policy.py` file can be found at *http://www.balasys.hu/documentation/*.

Additional information can also be found in the *Vela Administrator's Guide*, the *Vela Reference Guide*, and in the various tutorials available at the BalaSys Documentation Page at *http://www.balasys.hu/documentation*.

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

vela

vela — Vela Firewall Suite

Synopsis

`vela` [options]

Description

The `vela` command is the main entry point for a Vela instance, and as such it is generally called by *velactl(8)* with command line parameters specified in *`instances.conf(5)`*.

Options

`--version` or `-V`	Display version number and compilation information.
`--as <name>` or `-a <name>`	Set instance name to <name>. Instance names may consist of the characters [a-zA-Z0-9_] and must begin with a letter. Log messages of this instance are prefixed with this name.
`--no-syslog` or `-l`	Send log messages to the standard output instead of syslog. This option implies foreground mode, overriding the contradicting process options if present.
`--log-tags` or `-T`	Prepend log category and log level to each message.
`--log-escape`	Escape non-printable characters to avoid binary log files. Each character less than 0x20 and greater than 0x7F are escaped in the form <XX>.
`--log-spec <spec>` or `-s <spec>`	Set verbosity mask on a per category basis. Each log message has an assigned multi-level category, where levels are separated by a dot. For example, HTTP requests are logged under *http.request*. `<spec>` is a comma separated list of log specifications. A single log specification consists of a wildcard matching log category, a colon, and a number specifying the verbosity level of that given category. Categories match from left to right. E.g.: `--logspec 'http.*:5,core:3'`. The last matching entry will be used as the verbosity of the given category. If no match is found the default verbosity specified with `--verbose` is used.
`--threads <num>` or `-t <num>`	Set the maximum number of threads that can be used in parallel by this Vela instance.
`--idle-threads <num>` or `-I`	Set the maximum number of idle threads; this option has effect only if threadpools are enabled (see the option `--threadpools`).
`--threadpools` or `-O`	Enable the use of threadpools, which means that threads associated with sessions are not automatically freed, only if the maximum number of idle threads is exceeded.
`--user <user>` or `-u <user>`	Switch to the supplied user after starting up.

`--group <group>` or `-g <group>`	Switch to the supplied group after starting up.
`--chroot <dir>` or `-R <dir>`	Change root to the specified directory before reading the configuration file. The directory must be set up accordingly.
`--caps <caps>` or `-C <caps>`	Switch to the supplied set of capabilities after starting up. This should contain the required capabilities in the permitted set. For the syntax of capability description see the man page of cap_from_text(3).
`--no-caps` or `-N`	Do not change capabilities at all.
`--crypto-engine <engine>` or `-E <engine>`	Set the OpenSSL crypto engine to be used for hardware accelerated crypto support.

Files

`/etc/vela/`

`/etc/vela/policy.py`

`/etc/vela/instances.conf`

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

velactl

velactl — Start and stop vela instances.

Synopsis

`velactl command` [options [instances/@instance-list-file]]

Description

`velactl` starts and stops *vela(8)* instances based on the contents of the *`instances.conf(5)`* file. Multiple instance names can be specified in the command-line or in a file to start or stop several instances. If an error occurs while stopping or starting an instance, an exclamation mark is appended to the instance name as `velactl` processes the request, and a summary is printed when the program exits. If no instance is specified, the command is executed on all instances. The instances to be controlled can be specified in a file instead of listing them in the command line, e.g.: `velactl command options instances.txt`. The `instances.txt` should contain every instance name in a new line.

Commands

`start`	Starts the specified Vela instance(s).	
`force-start`	Starts the specified Vela instance(s) even if they are disabled.	
`stop`	Stops the specified Vela instance(s).	
`force-stop`	Forces the specified Vela instance(s) to stop using the KILL signal.	
`restart`	Restart the specified Vela instance(s).	
`force-restart`	Forces the specified Vela instance(s) to restart by stopping them using the KILL signal.	
`reload`	Reload the specified Vela instance(s).	
`status`	Display the status of the specified Vela instance(s).	
	`--verbose` or `-v`	Display detailed status information.
`gui-status`	Display the status of the specified Vela instance(s) in an internal format easily parsable by VMC. NOTE: This command is mainly used internally within Vela, and the structure of its output may change.	
`version`	Display version information on Vela.	
`inclog`	Raise the verbosity (log) level of the specified Vela instance(s) by one.	
`declog`	Decrease the verbosity (log) level of the specified Vela instance(s) by one.	
`log`	Change various log related settings in the specified Vela instance(s) using the following options:	

	`--vinc` or `-i`	Increase verbosity level by one.
	`--vdec` or `-d`	Decrease verbosity level by one.
	`--vset <verbosity>` or `-s <verbosity>`	Set verbosity level to <verbosity>.
	`--log-spec <spec>` or `-S <spec>`	Set verbosity mask on a per category basis. The format of this value is described in *vela(8)*.
	`--help` or `-h`	Display this help screen on the options of the `log` command.
`szig`	Display internal information from the specified Vela instance(s). The information to be disblayed can be specified with the following options:	
	`--walk` or `-w`	Walk the specified tree.
	`--root [node]` or `-r [node]`	Set the root node of the walk operation to [node].
	`--help` or `-h`	Display a brief help on the options of the `szig` command.
`help`	Display a brief help message.	

Examples

```
velactl start vela_ftp
```

The command above starts the vela instance named *vela-ftp* with parameters described in the `instances.conf` file.

Files

The default location for `instances.conf` is `/etc/vela/instances.conf`.

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

Copyright © 1996-2024 Balasys IT Zrt. All rights reserved.

velactl.conf

velactl.conf — *velactl(8)* configuration file.

Description

The `velactl.conf` file describes various global options ifluencing the behavior of *`velactl(8)`* . *`velactl(8)`* processes the file line by line, each line having the structure described below. Empty lines and lines beginning with '#' are comments and are ignored.

Structure

```
variable name = variable value
```

Each non-empty line specifies a variable name and its value separated by the equal sign ('='). The following variables are available:

`AUTO_RESTART`	Enable the automatic restart feature of `velactl`. Instances in auto-restart mode are restarted automatically when they exit. Default value: 1 (TRUE).
`STOP_CHECK_TIMEOUT`	The number of seconds to wait for a stopping Vela instance. Default value: 3.
`START_CHECK_TIMEOUT`	In *auto-restart* mode there is no real way to detect whether Vela failed to load or not. Velactl waits *START_CHECK_TIMEOUT* seconds and assumes that Vela loaded successfully if it did not exit within this interval. Default value: 5 seconds.
`START_WAIT_TIMEOUT`	In *no-auto-restart* mode the successful loading of a Vela instance can be verified by instructing Vela to daemonize itself and waiting for the parent to exit. This parameter specifies the number of seconds to wait for Vela to daemonize itself. Default value: 60 seconds.
`VELA_APPEND_ARGS`	Vela-specific arguments to be appended to the command line of each Vela instance. Also recognised as *APPEND_ARGS* (deprecated). Default value: *""*.
`VELACTL_APPEND_ARGS`	Velactl-specific arguments to be appended to the command line of each instance. Default value: *""*.
`CHECK_PERMS`	Specifies whether to check the permissions of the Vela configuration directory. If set, Vela refuses to run if the `/etc/vela` directory can be written by user other then *`vela`* Default value: 1 (TRUE).
`CONFIG_DIR`	The path to the Vela configuration directory to check if CHECK_PERMS is enabled. NOTE: it does not change the Vela policy file argument, this parameter is only used by the permission validating code. Default value: `${prefix}/etc/vela` .

`CONFIG_DIR_OWNER`, `CONFIG_DIR_GROUP`, `CONFIG_DIR_MODE`	The owner/group/permissions values considered valid for the configuration directory. `velactl` fails if the actual owner/group/permissions values conflict the ones set here. Default values: *root.vela, 0750* .
`PIDFILE_DIR`	The path to the Vela pid file directory. The directory is created automatically prior to starting Vela if it does not already exist.It is created if it does not exist, before NOTE: No --*pidfile* argument is passed to Vela, only texistance of the directory is verified. Default value: `/var/run/vela`.
`PIDFILE_DIR_OWNER`, `PIDFILE_DIR_GROUP`, `PIDFILE_DIR_MODE`	The owner/group/permission values the pidfile directory is created with if it does not exist. Default values: *root.root, 0700*.

Files

The default location for `velactl.conf` is `/etc/vela/velactl.conf`.

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

vela-zone-helper

vela-zone-helper — Zone helper daemon

Synopsis

`vela-zone-helper` [options]

Description

`vela-zone-helper(8)` is a daemon responsible for maintaining zone address information in vela-nfqueue-helper and also for updating dynamic address information in hostname-based zones. Its behaviour is controlled by *`vela-zone-helper.conf(5)`* or command-line options. Command-line options take precedence over configuration files.

Options

`--verbose <num>` or `-v <num>`	Set verbosity level to <num>. The valid values are 0-10; the default value is 3.
`--log-spec <spec>` or `-s <spec>`	Set verbosity mask on a per category basis. The format of this value is described in *vela(8)*.
`--no-syslog` or `-l`	Send log messages to the standard output instead of syslog.
`--dns-filter-level <num>` or `-f`	Set bogus DNS response filtering level to <num>. The valid values are 0-3; the default value is 3. For more information see *vela-zone-helper.conf.5*.
`--resolver-threads` or `-t`	Set the maximum number of DNS resolver threads. The default value is 8.
`--reload` or `-r`	Reload running `vela-zone-helper(8)` daemon.
`--config-file <filename>` or `-c <filename>`	Use <filename> as configuration file instead of the default `/etc/vela/vela-zone-helper.conf`.
`--help` or `-h`	Display a brief help message.

Files

`/etc/vela/vela-zone-helper.conf`

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

vela-zone-helper.conf

vela-zone-helper.conf — *vela-zone-helper(8)* configuration file

Description

The `vela-zone-helper.conf` file describes various global options controlling the behavior of *`vela-zone-helper(8)`*.

`vela-zone-helper(8)` processes the lines of the file under [vela-zone-helper] group, each line having the structure described below. Empty lines and lines beginning with '#' or ';' are comments and are ignored.

Structure

```
variable name = variable value
```

Each non-empty line specifies a variable name and its value separated by the equal sign ('='). The following variables are available:

`VERBOSE`	Set logging verbosity level. The default value is: 3.
`LOG_SPEC`	Set verbosity mask on a per category basis. The format of this value is described in *vela-zone-helper(8)*. The default value is: core.accounting:4.
`NO_SYSLOG`	Send log messages to the standard output instead of syslog. The default value is: 0 (False).
`DNS_FILTER_LEVEL`	Set bogus DNS response filtering level. The default value is: 3. Filtered addresses will not be used in hostname-based zones. DNS_FILTER_LEVEL value is interpreted as: ■ 0 = No filtering takes place ■ 1 = Filter invalid host addresses: unspecified addresses ('0.0.0.0/32', '::/128'). ■ 2 = Filter loopback address ranges ('127.0.0.0/8', '::1/128') ■ 3 = Filtering of private address ranges ('192.168.0.0/16', '10.0.0.0/8', '172.16. 0.0/12', 'fc00::/7'), link-local address ranges ('169.254.0.0/16', 'fe80::/10') and multicast ranges (224.0.0.0/4 , 'ff00::/8')
`RESOLVER_THREADS`	Set the maximum number of DNS resolver threads. To perform non-threaded resolving operations, set the value to 0. The default value is 8.

Files

The default location for `vela-zone-helper.conf` is `/etc/vela/vela-zone-helper.conf`.

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

vavupdate

vavupdate — Updates the various AntiVirus engine's databases.

Synopsis

`vavupdate` [options]

Description

`vavupdate` updates the databases of the various AntiVirus engines (Clamav, NOD32) used by Vela and VCF to filter the contents of the incoming and outgoing traffic.

Options

`-v <verbosity_level>`	The verbosity level of the program. Default value: 3. ■ *0*: No messages. ■ *1*: Show only error messages. ■ *2*: Report successful database updates. ■ *3*: Show also progress indicator messages. ■ *4*: Show all messages. (NOTE: The output can be very large.)
`-f`	Force the execution of vavupdate, with this option the HRS settings in `/etc/vela/*.options` can be overridden.
`-V`	Display the version number of `vavupdate`.
`-s`	Use syslog for logging (otherwise `vavupdate` logs into the file `/var/log/vavupdate.log`).
`-h`	Display a brief help message.

Files

`/etc/vela/vavupdate.conf`

`/var/log/vavupdate.log`

See Also

vavupdate.conf(5)

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

vavupdate.options

vavupdate.conf, clamav.options, nod32.options — *vavupdate(8)* configuration files.

Description

`vavupdate` reads its configuration from the `/etc/vela/vavupdate.conf` file and the various `.options` files in the `/etc/vela` directory. `vavupdate` was designed to run regularly as a cron job.

Options

`ADMINEMAIL`	The e-mail address(es) of the administrator(s). Leaving this field blank suppresses the sending of notification e-mails.
`VERBOSE`	The verbosity level of the program. ■ *0*: no messages; ■ *1*: show only error messages; ■ *2*: report successful database upgrades; ■ *3*: show also progress indicator messages; ■ *4*: show all messages (NOTE: The output can be huge.).
`SYSLOG`	When set to 1, vavupdate use syslog for logging (otherwise `vavupdate` logs into the file `/var/log/vavupdate.log`).
`FTPPROXY`	If access to FTP servers has to go through a proxy and the individual AV engine's package do not handle proxy server settings, the following setting has to be used: `FTPPROXY="http://proxyhost:proxyport/"` . If the proxy requires authentication, specify the username and the password as well: `FTPPROXY="http://username:password@proxyhost:proxyport/`.
`HTTPPROXY`	Access HTTP servers via proxy. The syntax is the same as the `FTPPROXY`'s.
`SUBJPREFIX`	An optional prefix which will be written to the subject line of the e-mail messages sent by the program. When using `vavupdate` on multiple hosts, this setting can be used to differentiate between the hosts. It is recommended to set this parameter to the hostname of the host `vavupdate` is running on.
`HRS`	The hours when `vavupdate` will run the database update for the specied AV engine. Example: *HRS="5 11 17 23"*. If the HRS parameter is left blank, `vavupdate` will updates the database every time it is invoked. It has to be specified in the per-engine `.options` files.
`DOENGINEUPGRADE`	DEPRECATED

Engine specific configuration files

Engine specific settings for the different AV engines are specified in the various `.options` files under `/etc/vela`. This files contain per-engine settings for `vavupdate`, most notably the HRS setting.

Files

`/etc/vela/vavupdate.conf`

`/etc/vela/nod32.options`

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

vqc

vqc — Vela Quarantine Checker

Synopsis

`vqc` [options]

Description

Check a quarantine directory and manage the files it contains.

There are no mandatory arguments; the settings may be specified as command-line options.

Options

`-h`	Display a brief help message.
`--quarantine <quarantine>` or `-q <quarantine>`	Directory used as quarantine. Default value: `/var/lib/vela/quarantine`.
`--format <format>` or `-t <format>`	Chooses output format (txt or xml). Default value: xml.
`--verbose` or `-v`	Makes the output more verbose.

Selection options

`--expr <expression>` or `-e <expression>`	Select only the objects matching <expression>. NOTE: This option may not be used together with `--id`.
`--id <id0, id1, ...>` or `-i <id0, id1, ...>`	Select only the objects having the specified ids. NOTE: This option may not be used together with `--expr`.

Restrictive options

`--older-than <days>`, or `-o <days>`	Select files older than <*days*> days.
`--bigger-than <Bytes>`, or `-b <Bytes>`	Select only files larger than <Bytes> bytes. The size is specified in bytes by default, the 'k' suffix means kilobytes, etc.
`--more-than <Number>`, or `-m <Number>`	Select only objects above the total count limit <Number>.

Action options

`--list_str <field0, field1, ...>`, or `-l <field0, field1, ...>`	List the selected objects. Only the specified fields (<field0, field1, ...>) are displayed.
`--preview <Bytes>`, or `-p <Bytes>`	Dump the selected objects, or only the first <Bytes> bytes of each object if <Bytes> is specified.

`--delete`, or `-d`	Delete the selected objects.
`--attachment-to <email address>`, or `-a <email address>`	Send the selected objects as attachment to <email address>,
`--subject <subject text>`, or `-s <subject text>`	Subject for email sent by `--attachment-to`, empty by default.
`--forward-to <email address>`, or `-f <email address>`	Forward the selected objects to <email address>.

See also

instances.conf(5), *vela(8)*, *velactl(8)*, *velactl.conf(5)*

Author

This manual page was written by the BalaSys Documentation Team <documentation@balasys.hu>.

Copyright

Appendix D. Proxedo Network Security Suite End-User License Agreement

D.1. 1. SUBJECT OF THE LICENSE CONTRACT

1.1 This License Contract is entered into by and between BalaSys and Licensee and sets out the terms and conditions under which Licensee and/or Licensee's Authorized Subsidiaries may use the Proxedo Network Security Suite under this License Contract.

D.2. 2. DEFINITIONS

In this License Contract, the following words shall have the following meanings:

2.1 BalaSys

Company name: BalaSys IT Ltd.

Registered office: H-1117 Budapest, Alíz Str. 4.

Company registration number: 01-09-687127

Tax number: HU11996468-2-43

2.2. Words and expressions

Annexed Software

Any third party software that is a not a BalaSys Product contained in the install media of the BalaSys Product.

Authorized Subsidiary

Any subsidiary organization: (i) in which Licensee possesses more than fifty percent (50%) of the voting power and (ii) which is located within the Territory.

BalaSys Product

Any software, hardware or service licensed, sold, or provided by BalaSys including any installation, education, support and warranty services, with the exception of the Annexed Software.

License Contract

The present Proxedo Network Security Suite License Contract.

Product Documentation

Any documentation referring to the Proxedo Network Security Suite or any module thereof, with special regard to the reference guide, the administration guide, the product description, the installation guide, user guides and manuals.

Protected Hosts

Host computers located in the zones protected by Proxedo Network Security Suite, that means any computer bounded to network and capable to establish IP connections through the firewall.

Protected Objects

The entire Proxedo Network Security Suite including all of its modules, all the related Product Documentation; the source code, the structure of the databases, all registered information reflecting the structure of the Proxedo Network Security Suite and all the adaptation and copies of the Protected Objects that presently exist or that are to be developed in the future, or any product falling under the copyright of BalaSys.

Proxedo Network Security Suite

Application software BalaSys Product designed for securing computer networks as defined by the Product Description.

Warranty Period

The period of twelve (12) months from the date of delivery of the Proxedo Network Security Suite to Licensee.

Territory

The countries or areas specified above in respect of which Licensee shall be entitled to install and/or use Proxedo Network Security Suite.

Take Over Protocol

The document signed by the parties which contains

a) identification data of Licensee;

b) ordered options of Proxedo Network Security Suite, number of Protected Hosts and designation of licensed modules thereof;

c) designation of the Territory;

d) declaration of the parties on accepting the terms and conditions of this License Contract; and

e) declaration of Licensee that is in receipt of the install media.

D.3. 3. LICENSE GRANTS AND RESTRICTIONS

3.1. For the Proxedo Network Security Suite licensed under this License Contract, BalaSys grants to Licensee a non-exclusive,

non-transferable, perpetual license to use such BalaSys Product under the terms and conditions of this License Contract and the applicable Take Over Protocol.

3.2. Licensee shall use the Proxedo Network Security Suite in the in the configuration and in the quantities specified in the Take Over Protocol within the Territory.

3.3. On the install media all modules of the Proxedo Network Security Suite will be presented, however, Licensee shall not be entitled to use any module which was not licensed to it. Access rights to modules and IP connections are controlled by an "electronic key" accompanying the Proxedo Network Security Suite.

3.4. Licensee shall be entitled to make one back-up copy of the install media containing the Proxedo Network Security Suite.

3.5. Licensee shall make available the Protected Objects at its disposal solely to its own employees and those of the Authorized Subsidiaries.

3.6. Licensee shall take all reasonable steps to protect BalaSys's rights with respect to the Protected Objects with special regard and care to protecting it from any unauthorized access.

3.7. Licensee shall, in 5 working days, properly answer the queries of BalaSys referring to the actual usage conditions of the

Proxedo Network Security Suite, that may differ or allegedly differs from the license conditions.

3.8. Licensee shall not modify the Proxedo Network Security Suite in any way, with special regard to the functions inspecting the usage of the software. Licensee shall install the code permitting the usage of the Proxedo Network Security Suite according to the provisions defined for it by BalaSys. Licensee may not modify or cancel such codes. Configuration settings of the Proxedo Network Security Suite in accordance with the possibilities offered by the system shall not be construed as modification of the software.

3.9. Licensee shall only be entitled to analize the structure of the BalaSys Products (decompilation or reverse-engineering) if concurrent operation with a software developed by a third party is necessary, and upon request to supply the information required for concurrent operation BalaSys does not provide such information within 60 days from the receipt of such a request. These user actions are limited to parts of the BalaSys Product which are necessary for concurrent operation.

3.10. Any information obtained as a result of applying the previous Section

(i) cannot be used for purposes other than concurrent operation with the BalaSys Product;

(ii) cannot be disclosed to third parties unless it is necessary for concurrent operation with the BalaSys Product;

(iii) cannot be used for the development, production or distribution of a different software which is similar to the BalaSys Product

in its form of expression, or for any other act violating copyright.

3.11. For any Annexed Software contained by the same install media as the BalaSys Product, the terms and conditions defined by its copyright owner shall be properly applied. BalaSys does not grant any license rights to any Annexed Software.

3.12. Any usage of the Proxedo Network Security Suite exceeding the limits and restrictions defined in this License Contract shall qualify as material breach of the License Contract.

3.13. The Number of Protected Hosts shall not exceed the amount defined in the Take Over Protocol.

3.14. Licensee shall have the right to obtain and use content updates only if Licensee concludes a maintenance contract that includes such content updates, or if Licensee has otherwise separately acquired the right to obtain and use such content updates. This License Contract does not otherwise permit Licensee to obtain and use content updates.

D.4. 4. SUBSIDIARIES

4.1 Authorized Subsidiaries may also utilize the services of the Proxedo Network Security Suite under the terms and conditions of this License Contract. Any Authorized Subsidiary utilising any service of the Proxedo Network Security Suite will be deemed to have accepted the terms and conditions of this License Contract.

D.5. 5. INTELLECTUAL PROPERTY RIGHTS

5.1. Licensee agrees that BalaSys owns all rights, titles, and interests related to the Proxedo Network Security Suite and all of BalaSys's patents, trademarks, trade names, inventions, copyrights, know-how, and trade secrets relating to the design, manufacture, operation or service of the BalaSys Products.

5.2. The use by Licensee of any of these intellectual property rights is authorized only for the purposes set forth herein, and upon termination of this License Contract for any reason, such authorization shall cease.

5.3. The BalaSys Products are licensed only for internal business purposes in every case, under the condition that such license does not convey any license, expressly or by implication, to manufacture, duplicate or otherwise copy or reproduce any of the BalaSys Products.

No other rights than expressly stated herein are granted to Licensee.

5.4. Licensee will take appropriate steps with its Authorized Subsidiaries, as BalaSys may request, to inform them of and assure compliance with the restrictions contained in the License Contract.

D.6. 6. TRADE MARKS

6.1. BalaSys hereby grants to Licensee the non-exclusive right to use the trade marks of the BalaSys Products in the Territory in accordance with the terms and for the duration of this License Contract.

6.2. BalaSys makes no representation or warranty as to the validity or enforceability of the trade marks, nor as to whether these infringe any intellectual property rights of third parties in the Territory.

D.7. 7. NEGLIGENT INFRINGEMENT

7.1. In case of negligent infringement of BalaSys's rights with respect to the Proxedo Network Security Suite, committed by violating the restrictions and limitations defined by this License Contract, Licensee shall pay liquidated damages to BalaSys. The amount of the liquidated damages shall be twice as much as the price of the BalaSys Product concerned, on BalaSys's current Price List.

D.8. 8. INTELLECTUAL PROPERTY INDEMNIFICATION

8.1. BalaSys shall pay all damages, costs and reasonable attorney's fees awarded against Licensee in connection with any claim brought against Licensee to the extent that such claim is based on a claim that Licensee's authorized use of the BalaSys Product infringes a patent, copyright, trademark or trade secret. Licensee shall notify BalaSys in writing of any such claim as soon as Licensee learns of it and shall cooperate fully with BalaSys in connection with the defense of that claim. BalaSys shall have sole control of that defense (including without limitation the right to settle the claim).

8.2. If Licensee is prohibited from using any BalaSys Product due to an infringement claim, or if BalaSys believes that any BalaSys Product is likely to become the subject of an infringement claim, BalaSys shall at its sole option, either: (i) obtain the right for Licensee to continue to use such BalaSys Product, (ii) replace or modify the BalaSys Product so as to make such BalaSys Product non-infringing and substantially comparable in functionality or (iii) refund to Licensee the amount paid for such infringing BalaSys Product and provide a pro-rated refund of any unused, prepaid maintenance fees paid by Licensee, in exchange for Licensee's return of such BalaSys Product to BalaSys.

8.3. Notwithstanding the above, BalaSys will have no liability for any infringement claim to the extent that it is based upon:

(i) modification of the BalaSys Product other than by BalaSys,

(ii) use of the BalaSys Product in combination with any product not specifically authorized by BalaSys to be combined with the BalaSys Product or

(iii) use of the BalaSys Product in an unauthorized manner for which it was not designed.

D.9. 9. LICENSE FEE

9.1. The number of the Protected Hosts (including the server as one host), the configuration and the modules licensed shall serve as the calculation base of the license fee.

9.2. Licensee acknowlegdes that payment of the license fees is a condition of lawful usage.

9.3. License fees do not contain any installation or post charges.

D.10. 10. WARRANTIES

10.1. BalaSys warrants that during the Warranty Period, the optical media upon which the BalaSys Product is recorded will not be defective under normal use. BalaSys will replace any defective media returned to it, accompanied by a dated proof of purchase, within the Warranty Period at no charge to Licensee. Upon receipt of the allegedly defective BalaSys Product, BalaSys will at its option, deliver a replacement BalaSys Product or BalaSys's current equivalent to Licensee at no additional cost. BalaSys will bear the delivery charges to Licensee for the replacement Product.

10.2. In case of installation by BalaSys, BalaSys warrants that during the Warranty Period, the Proxedo Network Security Suite, under normal use in the operating environment defined by BalaSys, and without unauthorized modification, will perform in substantial compliance with the Product Documentation accompanying the BalaSys Product, when used on that hardware for which it was installed, in compliance with the provisions of the user manuals and the recommendations of BalaSys. The date of the notification sent to BalaSys shall qualify as the date of the failure. Licensee shall do its best to mitigate the consequences of that failure. If, during the Warranty Period, the BalaSys Product fails to comply with this warranty, and such failure is reported by Licensee to BalaSys within the Warranty Period, BalaSys's sole obligation and liability for breach of this warranty is, at BalaSys's sole option, either:

(i) to correct such failure,

(ii) to replace the defective BalaSys Product or

(iii) to refund the license fees paid by Licensee for the applicable BalaSys Product.

D.11. 11. DISCLAIMER OF WARRANTIES

11.1. EXCEPT AS SET OUT IN THIS LICENSE CONTRACT, BALASYS MAKES NO WARRANTIES OF ANY KIND WITH RESPECT TO THE Proxedo Network Security Suite. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, BALASYS EXCLUDES ANY OTHER WARRANTIES, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF SATISFACTORY QUALITY, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS.

D.12. 12. LIMITATION OF LIABILITY

12.1. SOME STATES AND COUNTRIES, INCLUDING MEMBER COUNTRIES OF THE EUROPEAN UNION, DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES AND, THEREFORE, THE FOLLOWING LIMITATION OR EXCLUSION MAY NOT APPLY TO THIS LICENSE CONTRACT IN THOSE STATES AND COUNTRIES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW AND REGARDLESS OF WHETHER ANY REMEDY SET OUT IN THIS LICENSE CONTRACT FAILS OF ITS ESSENTIAL PURPOSE, IN NO EVENT SHALL BALASYS BE LIABLE TO LICENSEE FOR ANY SPECIAL, CONSEQUENTIAL, INDIRECT OR SIMILAR DAMAGES OR LOST PROFITS OR LOST DATA ARISING OUT OF THE USE OR INABILITY TO USE THE Proxedo Network Security Suite EVEN IF BALASYS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

12.2. IN NO CASE SHALL BALASYS'S TOTAL LIABILITY UNDER THIS LICENSE CONTRACT EXCEED THE FEES PAID BY LICENSEE FOR THE Proxedo Network Security Suite LICENSED UNDER THIS LICENSE CONTRACT.

D.13. 13.DURATION AND TERMINATION

13.1. This License Contract shall come into effect on the date of signature of the Take Over Protocol by the duly authorized

representatives of the parties.

13.2. Licensee may terminate the License Contract at any time by written notice sent to BalaSys and by simultaneously destroying all copies of the Proxedo Network Security Suite licensed under this License Contract.

13.3. BalaSys may terminate this License Contract with immediate effect by written notice to Licensee, if Licensee is in material or persistent breach of the License Contract and either that breach is incapable of remedy or Licensee shall have failed to remedy that breach within 30 days after receiving written notice requiring it to remedy that breach.

D.14. 14. AMENDMENTS

14.1. Save as expressly provided in this License Contract, no amendment or variation of this License Contract shall be effective unless in writing and signed by a duly authorised representative of the parties to it.

D.15. 15. WAIVER

15.1. The failure of a party to exercise or enforce any right under this License Contract shall not be deemed to be a waiver of that right nor operate to bar the exercise or enforcement of it at any time or times thereafter.

D.16. 16. SEVERABILITY

16.1. If any part of this License Contract becomes invalid, illegal or unenforceable, the parties shall in such an event negotiate in good faith in order to agree on the terms of a mutually satisfactory provision to be substituted for the invalid, illegal or unenforceable

provision which as nearly as possible validly gives effect to their intentions as expressed in this License Contract.

D.17. 17. NOTICES

17.1. Any notice required to be given pursuant to this License Contract shall be in writing and shall be given by delivering the notice by hand, or by sending the same by prepaid first class post (airmail if to an address outside the country of posting) to the address of the relevant party set out in this License Contract or such other address as either party notifies to the other from time to time. Any notice given according to the above procedure shall be deemed to have been given at the time of delivery (if delivered by hand) and when received (if sent by post).

D.18. 18. MISCELLANEOUS

18.1. Headings are for convenience only and shall be ignored in interpreting this License Contract.

18.2. This License Contract and the rights granted in this License Contract may not be assigned, sublicensed or otherwise transferred in whole or in part by Licensee without BalaSys's prior written consent. This consent shall not be unreasonably withheld or delayed.

18.3. An independent third party auditor, reasonably acceptable to BalaSys and Licensee, may upon reasonable notice to Licensee and during normal business hours, but not more often than once each year, inspect Licensee's relevant records in order to confirm that usage of the Proxedo Network Security Suite complies with the terms and conditions of this License Contract. BalaSys shall bear the costs of such audit. All audits shall be subject to the reasonable safety and security policies and procedures of Licensee.

18.4. This License Contract constitutes the entire agreement between the parties with regard to the subject matter hereof. Any modification of this License Contract must be in writing and signed by both parties.

Appendix E. Creative Commons Attribution Non-commercial No Derivatives (by-nc-nd) License

f. "Work" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

g. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

h. "Publicly Perform" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

i. "Reproduce" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. *Fair Dealing Rights.* Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. *License Grant.* Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

   a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections; and,

   b. to Distribute and Publicly Perform the Work including as incorporated in Collections.

   The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats, but otherwise you have no rights to make Adaptations. Subject to 8(f), all rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights set forth in Section 4(d).

4. *Restrictions.* The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested.

b. You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.

c. If You Distribute, or Publicly Perform the Work or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (for example a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work. The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Collection, at a minimum such credit will appear, if a credit for all contributing authors of Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

d. For the avoidance of doubt:

   i. Non-waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

   ii. Waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License if Your exercise of such rights is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(b) and otherwise waives the right to collect royalties through any statutory or compulsory licensing scheme; and,

iii. Voluntary License Schemes. The Licensor reserves the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License that is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(b).

e. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation.

5. *Representations, Warranties and Disclaimer* UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. *Limitation on Liability.* EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. *Termination*

   a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

   b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. *Miscellaneous*

   a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   b. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

c. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

d. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

e. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

Index of Proxy attributes

Index of Core attributes

Index of all attributes