How to configure TLS proxying in PNS 2

February 29, 2024

Abstract
This tutorial describes how to configure PNS to proxy TLS traffic

Table of Contents

1. Preface

This tutorial provides guidelines for PNS administrators on how to enable proxying secure TLS connections. Knowledge in TCP/IP and PNS administration is required to fully comprehend the contents of this paper. The procedures and concepts described here are applicable to version 2 of PNS. Detailed information is provided to configure PNS both from Management Console and using Python scripts.

Note that explaining the concepts of the different aspects of TLS proxying is beyond the scope of this tutorial. For background information, see the following documents:

- For details on deriving and modifying proxies, see *Section 6.6, Proxy classes* in *Proxedo Network Security Suite 2 Administrator Guide*.
- For an overview on certificates and related topics in connection with PNS, see *Chapter 11, Key and certificate management in PNS* in *Proxedo Network Security Suite 2 Administrator Guide*.
- For details on the available TLS-related attributes of the Application-level Gateway proxies, see *Section 3.2, Handling TLS and SSL connections in Application-level Gateway* in *Proxedo Network Security Suite 2 Reference Guide*

You can download the above documents at the *Documentation Page*.

1.1. Configuring Application-level Gateway: MC versus Python

Application-level Gateway can be fully configured using either the graphical Management Console (MC) or manually by editing plain text Python configuration files. The suggestions and the background information provided in this tutorial are equally applicable to both methods. Step-by-step explanation with screenshots are given for MC-based configuration, while sample Python code lines can be found at the end of each step. After replacing the sample parameters (for example, IP addresses) with the proper ones for the actual configuration, add these lines to the policy file of Application-level Gateway (usually found under `/etc/vela/policy.py`). Also pay attention to the proper indentation of Python code lines. For more details, see *Chapter 10, Local firewall administration* in *Proxedo Network Security Suite 2 Administrator Guide*.

If Management Console is used and the Python code generated by MC needs to be displayed, select a host, then select **Configuration** > **View** from the main menu.

2. Configuring TLS proxying

For proxying connections embedded in TLS, a properly configured proxy for the embedded protocol is required — the best way is to derive an own proxy class and modify its parameters, but you can use a built-in proxy (for example, the `HttpProxy` for HTTPS traffic) if its default behavior is acceptable for you. The TLS framework validates the certificate of the server, decrypts the secure channel, then passes the data to the proxy. To transfer traffic that does not have a native proxy, or to inspect only the TLS connection without analyzing the embedded protocol, use *`PlugProxy`*.

For details on deriving and modifying proxies, see *Section 6.6, Proxy classes* in *Proxedo Network Security Suite 2 Administrator Guide*.

The following procedure describes how to configure Application-level Gateway proxies to handle TLS connections. For the configuration examples, an Http proxy will be used to inspect HTTPS connections — you can use other proxies similarly to inspect IMAPS, POP3S, and other types of traffic.

2.1. 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 (p. 14)*.

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*.

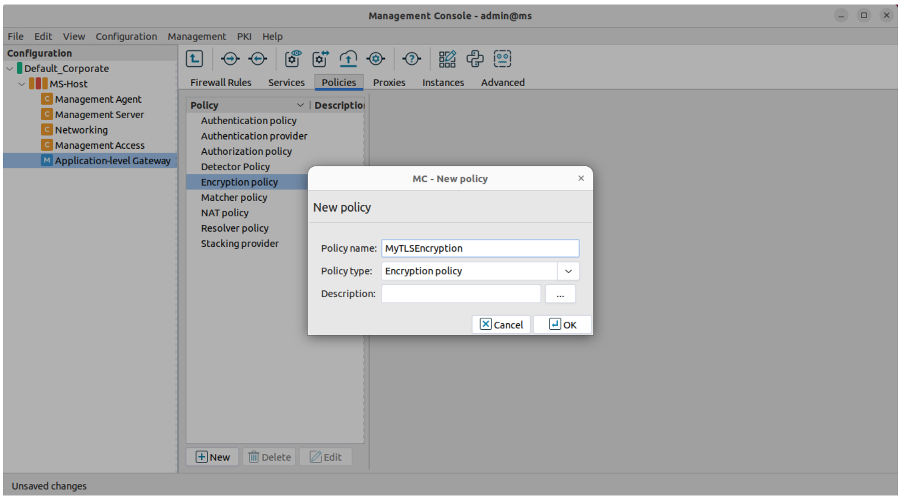
Python: In configurations managed manually from python, create an X.509 certificate (with its related keypair) using a suitable software and deploy it to the PNS firewall host (for example, copy it to the `/etc/key.d/mycert` folder).

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: MyTLSEncryption

Policy type: Encryption policy

Description:

☒ Cancel ☐ OK

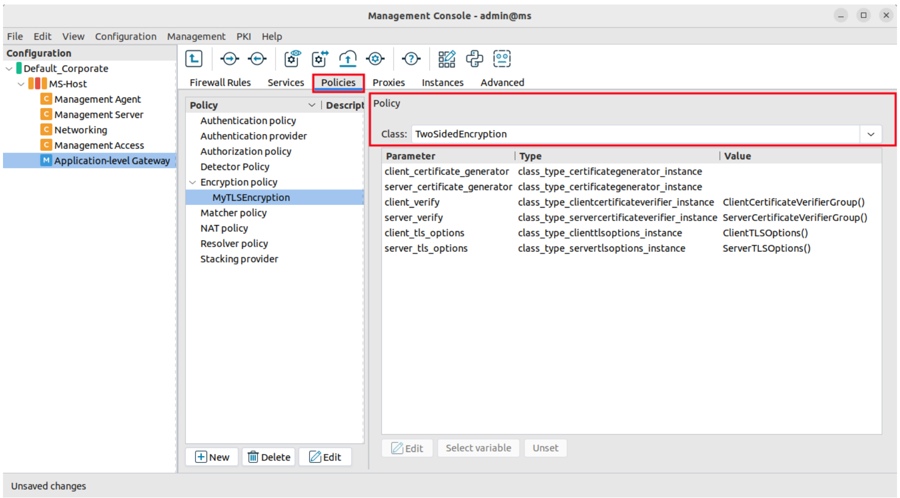
Figure 1. Creating a new Encryption policy

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**.



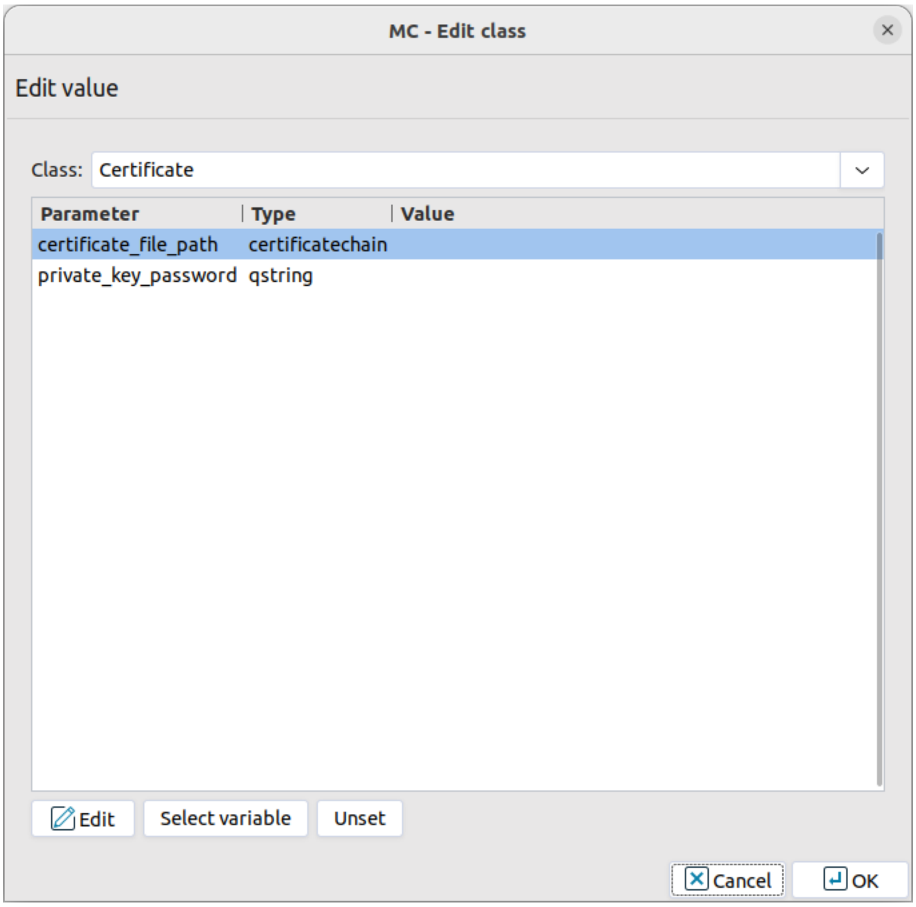
Policy

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 2. Selecting Encryption policy class

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

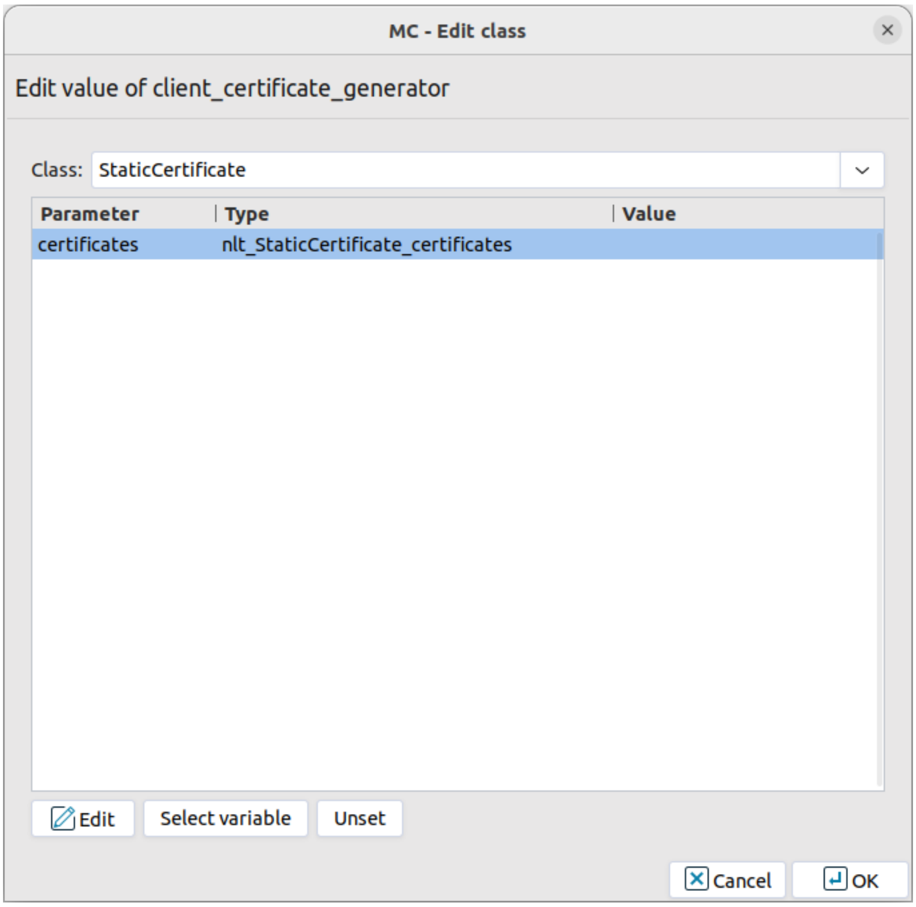


Class: Certificate

Parameter	Type	Value
certificate_file_path	certificatechain	
private_key_password	qstring	

Figure 3. 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	

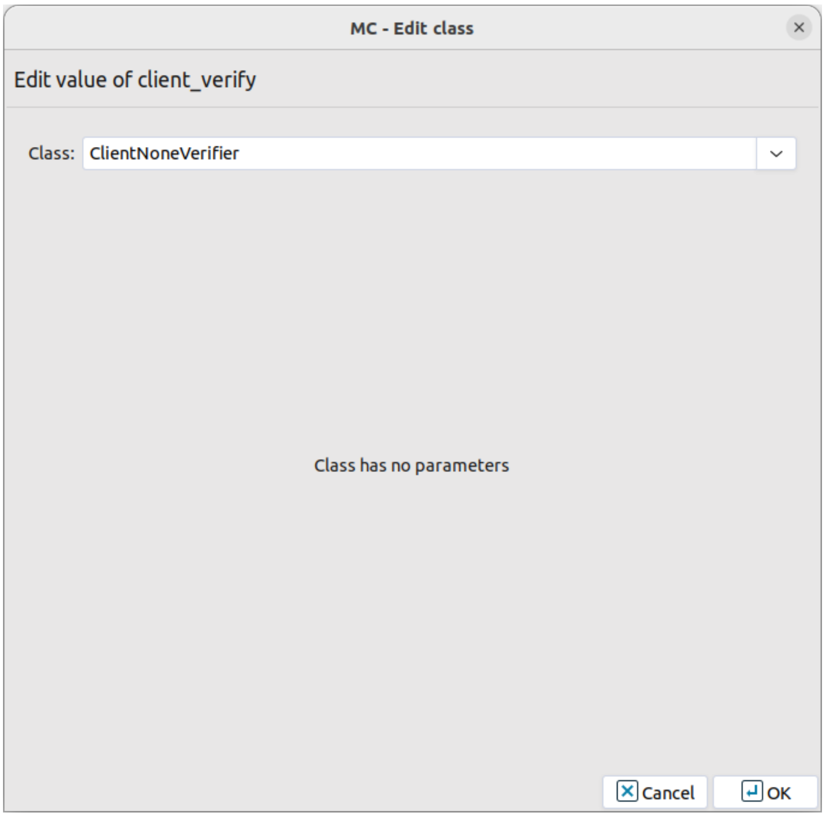
Figure 4. Creating a new Encryption policy

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.



MC - Edit class

Edit value of client_verify

Class: ClientNoneVerifier

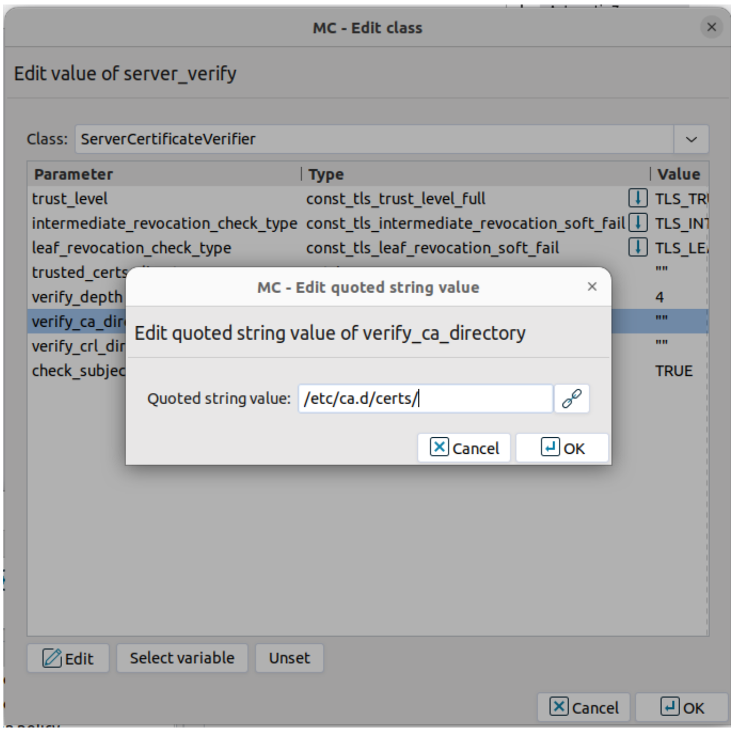
Class has no parameters

Figure 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_TRU
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 6. Specifying trusted CAs

Python:

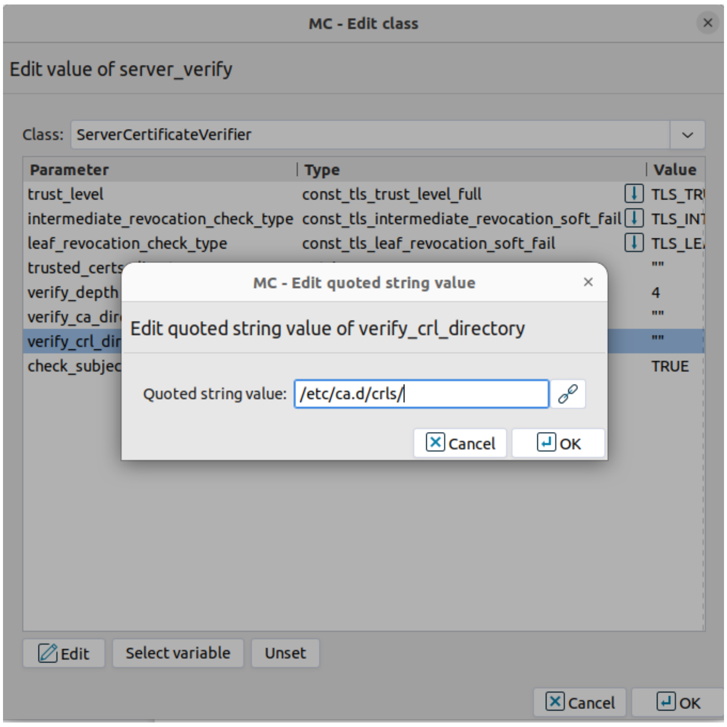
```
    encryption=TwoSidedEncryption(
        server_verify=ServerCertificateVerifier(
            verify_ca_directory="/etc/ca.d/certs/"
        )
    )
```

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

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_crl_directory

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

Figure 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 (p. 14)*

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.

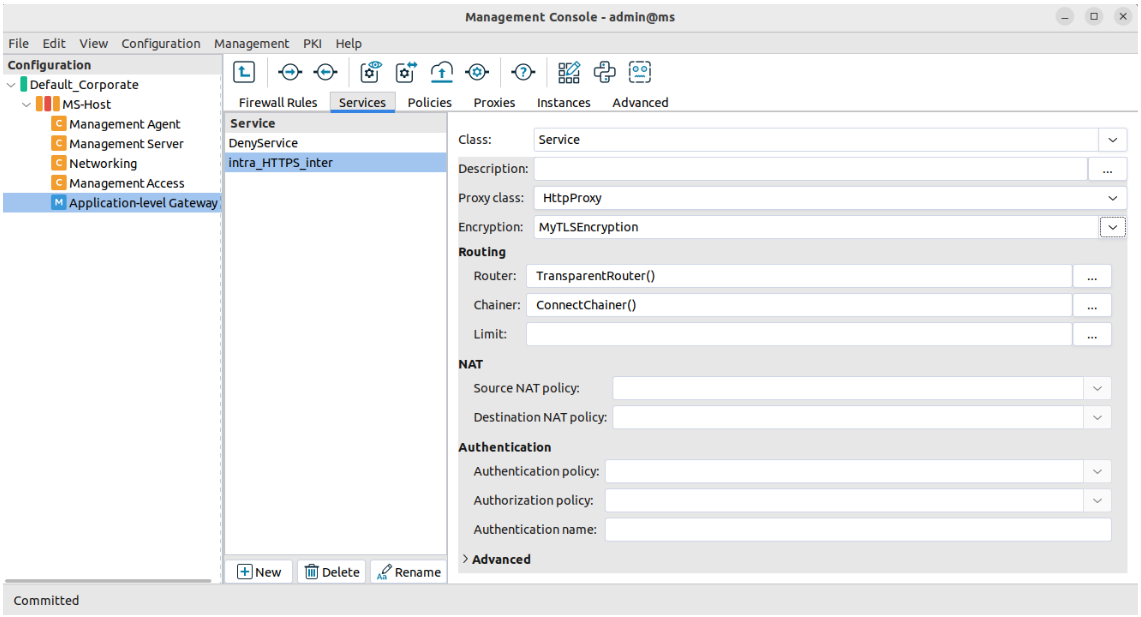
Step b. Select **Proxy class** > **Http** > **HttpProxy**.

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**.



Default_Corporate
MS-Host
Management Agent
Management Server
Networking
Management Access
Application-level Gateway

Service
DenyService
intra_HTTPS_inter

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 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.

2.2. 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:

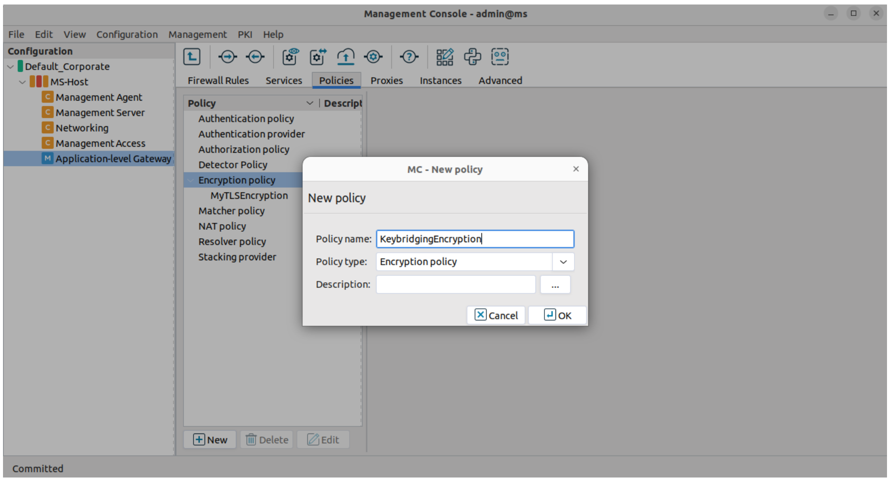
Copy the certificates and CRLs to their respective directories (for example, into `/etc/vela/tls-bridge/`). Note that they are not updated automatically as in configurations managed by MC.

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`.



MC - New policy

New policy

Policy name: KeybridgingEncryption

Policy type: Encryption policy

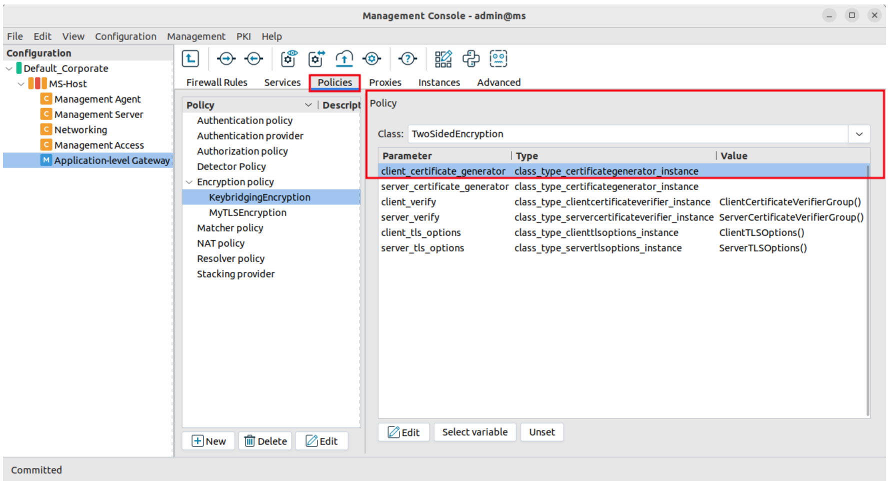
Description:

☒ Cancel ↵ OK

Figure 9. Creating an Encryption policy

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

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



Policy

Class: TwoSidedEncryption

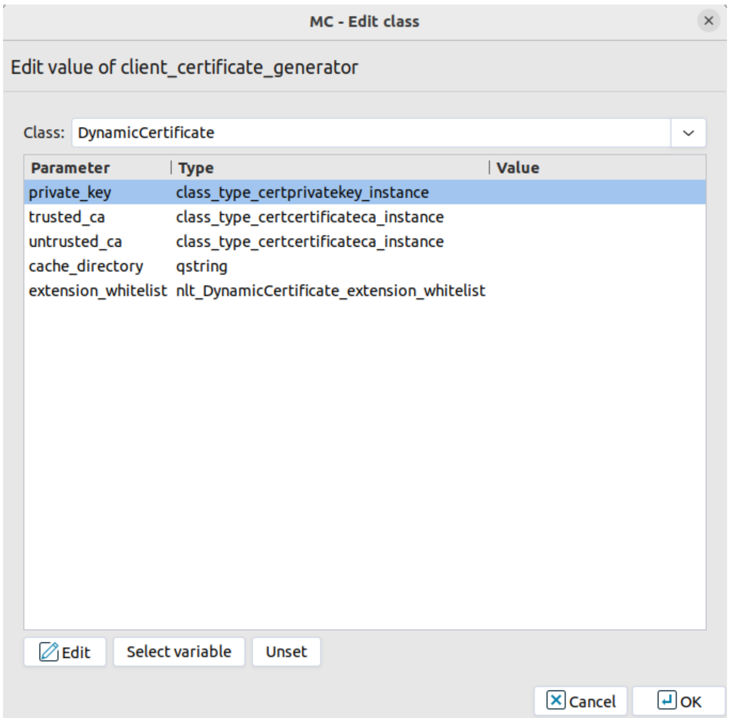
Parameter	Type	Value
client_certificate_generator	class_type_certificategenerator_instance	
server_certificate_generator	class_type_certificategenerator_instance	
client_verify	class_type_clientcertificateverifier_instance	ClientCertificateVerifierGroup()
server_verify	class_type_servercertificateverifier_instance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

Figure 10. Selecting the encryption class

Python:

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

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



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 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:

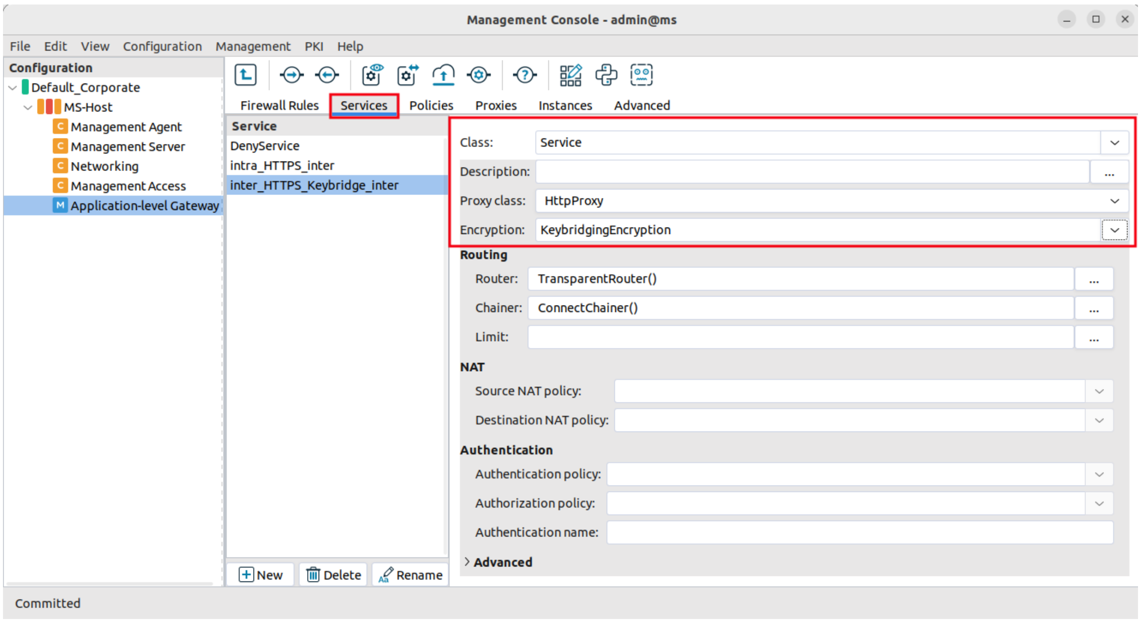
```
        client_certificate_generator=DynamicCertificate(
private_key=PrivateKey.fromFile(key_file_path="/etc/key.d/TLS-bridge/key.pem"),
            trusted_ca=Certificate.fromFile(
certificate_file_path="/etc/ca.d/certs/CA_for_Trusted_certs.pem",

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: KeybridgingEncryption

Routing

Router: TransparentRouter()

Chainer: ConnectChainer()

Limit:

NAT

Source NAT policy:

Destination NAT policy:

Authentication

Authentication policy:

Authorization policy:

Authentication name:

Advanced

Figure 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.)

2.3. Exemption sites and whitelists

Certain situations require client-side or mutual authentication that might not be proxied appropriately, therefore you have to allow them to pass the firewall using a plug proxy. This situation arises most commonly in secure banking and online ordering sites that use HTTPS, or with dedicated client-server applications (such as WindowsUpdate). To maintain a list of such sites, use one of the following methods:

- If the IP address of the affected servers is static, add them to a separate zone. For details, see *Procedure 2.3.1, IP filtering using a zone (p. 20)*.
- Use a MatcherPolicy. Matcher policies can compare the IP address of the target server to a predefined list, and can be configured to behave differently if a match is found or not. Another example will use a domain-name-matcher policy to resolve domain names that have dynamic IPs, or change their addresses periodically (for example, they use DNS round-robin method). Matcher policies are a bit more resource intensive, but easier to use and maintain after the initial configuration. If nothing restricts it, use a matcher policy.
- Use a DetectorService to select which service to start based on the traffic parameters. For details, see *Section 6.7.2, Detector policies* in *Proxedo Network Security Suite 2 Administrator Guide* and

Procedure 6.4.4, Creating a new DetectorService in *Proxedo Network Security Suite 2 Administrator Guide*.

More sophisticated configurations using both types of whitelisting can be also implemented based on the following examples.

2.3.1. Procedure – IP filtering using a zone

Purpose:

Firewall rules can select the service to be started based on the zone that the server and the client belongs to. Zones are organized based on IP addresses, the rule specifies which clients can use which services to access the specified servers. To create whitelists using a zone, complete the following steps.

Steps:

Step 1. Create a zone containing only the servers that require client-side authentication (or that have other reasons why the connection cannot be proxied).

Step a. Select the site that contains the firewall host, then click **New**, and create a new zone (for example, `banks`).

Step b. Select the newly created zone, and add a network to the zone, that is, specify the **Network address** and **Netmask** of the target servers. Include only the servers that you want to be available via encrypted but not proxied channels.

Note
You will also need a client zone. You can use the entire intranet or a smaller zone, as required. Create a new zone for the clients if you want to make these servers available only to certain clients.

Step 2. Create a new service.

Step a. Select your firewall host, navigate to **Application-level Gateway** > **Services**, then click **New**.

Step b. Create a new service, for example, `intra_PLUG_banks`.

Step c. Select **Proxy class** > **PlugProxy**.

Step d. Specify other service parameters as required for your environment.

Step 3. Create a firewall rule that uses the zones and the service created in the previous steps.

Step a. Select **Firewall Rules** > **New** > **Service** > **Class**, 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 b. Select **Conditions** > **Source** > **Add** > **Zone**, and select the zone from which the clients will connect to the servers (for example, `intranet`).

Step c. Select **Destination** > **Add** > **Zone**, and select the zone that contains the whitelisted servers (for example, `banks`).

Step d. Configure the other parameters of the rule as needed for your environment, then click **OK**.

Step 4. Commit and upload your changes, then restart Application-level Gateway.

3. Python code summary

When configured according to this tutorial, the policy.py file of Application-level Gateway should look something like this:

Configuring TLS proxying:

```
EncryptionPolicy(
    name="MyTLSEncryption",
    encryption=TwoSidedEncryption(
        client_verify=ClientNoneVerifier(),
        client_tls_options=ClientTLSOptions(),
        server_verify=ServerCertificateVerifier(
            verify_ca_directory="/etc/ca.d/certs/",
            verify_crl_directory="/etc/ca.d/crls/",
            trusted=TRUE, verify_depth=4,
            permit_invalid_certificates=FALSE,
            permit_missing_crl=FALSE,
            check_subject=TRUE
            ),
        server_tls_options=ServerTLSOptions(
            ciphers=TLS_CIPHERS_DEFAULT, timeout=300,
            tls_min_version=TLS_VERSION_1_3,
            ),
        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")),
            ))
        ))


    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'
    )
```

If keybridging is performed:

```
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")),
            cache_directory="/var/lib/vela/tls-bridge")
    ))


    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'
    )
```

4. Summary

This tutorial has shown how to configure PNS to proxy encrypted TLS traffic, including scenarios where keybridging is used. Although these examples are relatively simple, they provide a solid base from which more complex configurations can be built — just as the security policy of your organization requires it.

All questions, comments or inquiries should be directed to `<info@balasys.hu>` or by post to the following address: BalaSys IT Ltd. 1117 Budapest, Alíz Str. 4 Phone: +36 1 646 4740 Web: *https://www.balasys.hu/*

The latest version is always available at the BalaSys *Documentation Page*.