Red Hat Enterprise Linux 10

Security hardening

Enhancing security of Red Hat Enterprise Linux 10 systems

Last Updated: 2026-01-27

Red Hat Enterprise Linux 10 Security hardening

Enhancing security of Red Hat Enterprise Linux 10 systems

Legal Notice

Abstract

Learn the processes and practices for securing Red Hat Enterprise Linux servers and workstations against local and remote intrusion, exploitation, and malicious activity. By using these approaches and tools, you can create a more secure computing environment for the data center, workplace, and home.

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation. Let us know how we can improve it.

Submitting feedback through Jira (account required)

1. Log in to the Jira website.

2. Click **Create** in the top navigation bar.

3. Enter a descriptive title in the **Summary** field.

4. Enter your suggestion for improvement in the **Description** field. Include links to the relevant parts of the documentation.

5. Click **Create** at the bottom of the dialogue.

CHAPTER 1. SWITCHING RHEL TO FIPS MODE

To enable the cryptographic module self-checks mandated by the Federal Information Processing Standard (FIPS) 140-3, you must operate RHEL 10 in FIPS mode. The only correct way to switch the system to FIPS mode is to enable it during the RHEL installation.

Switching a system to FIPS mode after installation is not supported in Red Hat Enterprise Linux 10. In particular, setting the **FIPS** system-wide cryptographic policy is not sufficient to enable FIPS mode and guarantee compliance with the FIPS 140 standard. The **fips-mode-setup** tool, which used the **FIPS** policy for enabling FIPS mode, has been removed from RHEL.

To turn off FIPS mode, you must reinstall the system without enabling FIPS mode during the installation process.

1.1. FEDERAL INFORMATION PROCESSING STANDARDS 140 AND FIPS MODE

Understand the Federal Information Processing Standard (FIPS) 140, a government standard specifying cryptographic module security. FIPS mode enables required self-checks for compliance.

The FIPS Publication 140 is a series of computer security standards developed by the National Institute of Standards and Technology (NIST) to ensure the quality of cryptographic modules. The FIPS 140 standard ensures that cryptographic tools implement their algorithms correctly. Runtime cryptographic algorithms and integrity self-tests are some of the mechanisms to ensure a system uses cryptography that meets the requirements of the standard.

1.1.1. RHEL in FIPS mode

To ensure that your RHEL system generates and uses all cryptographic keys only with FIPS-approved algorithms, you must switch RHEL to FIPS mode.

To enable FIPS mode, start the installation in FIPS mode. This avoids cryptographic key material regeneration and reevaluation of the compliance of the resulting system associated with converting already deployed systems. Additionally, components that change their algorithm choices based on whether FIPS mode is enabled choose the correct algorithms. For example, LUKS disk encryption uses the PBKDF2 key derivation function (KDF) during installation in FIPS mode, but it chooses the non-FIPS-compliant Argon2 KDF otherwise. Therefore, a non-FIPS installation with disk encryption is either not compliant or potentially unbootable when switched to FIPS mode after the installation.

To operate a FIPS-compliant system, create all cryptographic key material in FIPS mode. Furthermore, the cryptographic key material must never leave the FIPS environment unless it is securely wrapped and never unwrapped in non-FIPS environments.

1.1.2. FIPS mode status

Whether FIPS mode is enabled is determined by the **fips=1** boot option on the kernel command line. The system-wide cryptographic policy automatically follows this setting if it is not explicitly set by using the **update-crypto-policies --set FIPS** command. Systems with a separate partition for **/boot** additionally require a **boot=UUID=<uuid-of-boot-disk>** kernel command line argument. The installation program performs the required changes when started in FIPS mode.

Enforcement of restrictions required in FIPS mode depends on the contents of the **/proc/sys/crypto/fips_enabled** file. If the file contains **1**, RHEL core cryptographic components switch to mode, in which they use only FIPS-approved implementations of cryptographic algorithms. If

/proc/sys/crypto/fips_enabled contains **0**, the cryptographic components do not enable their FIPS mode.

1.1.3. FIPS in crypto-policies

The **FIPS** system-wide cryptographic policy helps to configure higher-level restrictions. Therefore, communication protocols supporting cryptographic agility do not announce ciphers that the system refuses when selected. For example, the ChaCha20 algorithm is not FIPS-approved, and the **FIPS** cryptographic policy ensures that TLS servers and clients do not announce the TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 TLS cipher suite, because any attempt to use such a cipher fails.

If you operate RHEL in FIPS mode and use an application providing its own FIPS-mode-related configuration options, ignore these options and the corresponding application guidance. The system runs in FIPS mode and the system-wide cryptographic policies enforce only FIPS-compliant cryptography. For example, the Node.js configuration option **--enable-fips** is ignored if the system runs in FIPS mode. If you use the **--enable-fips** option on a system not running in FIPS mode, you do not meet the FIPS-140 compliance requirements.

1.1.4. RHEL 10.0 OpenSSL FIPS indicators

Because RHEL introduced OpenSSL FIPS indicators before the OpenSSL upstream did, and both designs differ, the indicators might change in a future minor version of RHEL 10. After the potential adoption of the upstream API, the RHEL 10.0 indicators might return an error message "unsupported" instead of a result. See the OpenSSL FIPS Indicators GitHub document for details.

> NOTE
>
> For the certification status of RHEL 10 cryptographic modules according to the FIPS 140-3 requirements by the National Institute of Standards and Technology (NIST) Cryptographic Module Validation Program (CMVP), see the FIPS - Federal Information Processing Standards section on the Product compliance Red Hat Customer Portal page.

Additional resources

- FIPS - Federal Information Processing Standards (Red Hat Customer Portal)
- RHEL system-wide cryptographic policies
- FIPS publications at NIST Computer Security Resource Center
- Federal Information Processing Standards Publication: FIPS 140-3 (NIST)

1.2. INSTALLING THE SYSTEM WITH FIPS MODE ENABLED

To enable the cryptographic module self-checks mandated by the Federal Information Processing Standard (FIPS) 140, enable FIPS mode during the Red Hat Enterprise Linux system installation.

WARNING

After you complete the setup of FIPS mode, you cannot disable FIPS mode without putting the system into an inconsistent state. If your scenario requires this change, the only correct way is a complete re-installation of the system.

Procedure

1. Add the **fips=1** option to the kernel command line at the start of the system installation when the Red Hat Enterprise Linux boot window opens and displays available boot options.
    a. On UEFI systems, press the **e** key, move the cursor to the end of the **linuxefi** kernel command line, and add **fips=1** to the end of this line, for example:

    ```
    linuxefi /images/pxeboot/vmlinuz inst.stage2=hd:LABEL=RHEL-10-0-BaseOS-x86_64
    rd.live.\
    check quiet fips=1
    ```

    b. On BIOS systems, press the **Tab** key, move the cursor to the end of the kernel command line, and add **fips=1** to the end of this line, for example:

    ```
    > vmlinuz initrd=initrd.img inst.stage2=hd:LABEL=RHEL-10-0-BaseOS-x86_64
    rd.live.check quiet fips=1
    ```

2. During the software selection stage, do not install any third-party software.
3. After the installation, the system starts in FIPS mode automatically.

Verification

- After the system starts, check that FIPS mode is enabled:

```
$ cat /proc/sys/crypto/fips_enabled
1
```

Additional resources

- Customizing boot options

1.3. ENABLING FIPS MODE WITH RHEL IMAGE BUILDER

You can create a customized image and boot a FIPS-enabled RHEL image. Before you compose the image, you must change the value of the **fips** directive in your blueprint.

Prerequisites

- You are logged in as the root user or a user who is a member of the **weldr** group.

Procedure

1. Create a plain text file in the Tom's Obvious, Minimal Language (TOML) format with the following content:

```
name = "system-fips-mode-enabled"
description = "blueprint with FIPS enabled"
version = "0.0.1"

[customizations]
fips = true

[[customizations.user]]
name = "admin"
password = "admin"
groups = ["users", "wheel"]
```

2. Build the customized RHEL image:

```
# image-builder build image-type --blueprint blueprint-name
```

Verification

1. Log in to the system image with the username and password that you configured in your blueprint.

2. Check if FIPS mode is enabled:

```
$ cat /proc/sys/crypto/fips_enabled
1
```

1.4. CREATING A BOOTABLE DISK IMAGE FOR A FIPS-ENABLED SYSTEM

You can create a disk image and enable FIPS mode when performing an Anaconda installation. You must add the **fips=1** kernel argument when booting the disk image.

Prerequisites

- You have Podman installed on your host machine.
- You have **virt-install** installed on your host machine.
- You have root access to run the **bootc-image-builder** tool, and run the containers in **--privileged** mode, to build the images.

Procedure

1. Create a **01-fips.toml** to configure FIPS enablement, for example:

```
# Enable FIPS
kargs = ["fips=1"]
```

2. Create a Containerfile with the following instructions to enable the **fips=1** kernel argument and adjust the cryptographic policies:

   ```
   FROM registry.redhat.io/rhel10/rhel-bootc:latest
   # Enable fips=1 kernel argument: https://bootc-dev.github.io/bootc/building/kernel-arguments.html
   COPY 01-fips.toml /usr/lib/bootc/kargs.d/
   # Install and enable the FIPS crypto policy
   RUN dnf install -y crypto-policies-scripts && update-crypto-policies --no-reload --set FIPS
   ```

3. Create your bootc ***<image>*** compatible base disk image by using **Containerfile** in the current directory:

   ```
   $ sudo podman run \
     --rm \
     -it \
     --privileged \
     --pull=newer \
     --security-opt label=type:unconfined_t \
     -v ./config.toml:/config.toml:ro \
     -v ./output:/output \
     -v /var/lib/containers/storage:/var/lib/containers/storage \
     registry.redhat.io/rhel10/bootc-image-builder:latest \
     --type iso \
     quay.io/<namespace>/<image>:<tag>
   ```

4. Enable FIPS mode during the system installation:

   a. When booting the RHEL Anaconda installer, on the installation screen, press the TAB key and add the **fips=1** kernel argument.
      After the installation, the system starts in FIPS mode automatically.

Verification

- After login in to the system, check that FIPS mode is enabled:

  ```
  $ cat /proc/sys/crypto/fips_enabled
  1
  $ update-crypto-policies --show
  FIPS
  ```

Additional resources

- Installing the system with FIPS mode enabled

1.5. LIST OF RHEL APPLICATIONS USING CRYPTOGRAPHY THAT IS NOT COMPLIANT WITH FIPS 140-3

Review the RHEL applications that use cryptography that is not compliant with FIPS 140-3. To pass all relevant cryptographic certifications, such as FIPS 140-3, use libraries from the core cryptographic components set.

These libraries, except for **libgcrypt**, also follow the RHEL system-wide cryptographic policies.

See the RHEL core cryptographic components Red Hat Knowledgebase article for information about the core cryptographic components, how they are selected, how they integrate with the operating system, how they support hardware security modules and smart cards, and how cryptographic certifications apply to them.

The following RHEL 10 applications use cryptography that is not compliant with FIPS 140-3:

Bacula

Implements the CRAM-MD5 authentication protocol.

Cyrus SASL

Uses the SCRAM-SHA-1 authentication method.

curl

Supports NTLM authentication which uses MD4 and MD5.

Dovecot

Uses SCRAM-SHA-1.

Emacs

Uses SCRAM-SHA-1.

FreeRADIUS

Uses MD5 and SHA-1 for authentication protocols.

Ghostscript

Custom cryptography implementation (MD5, RC4, SHA-2, AES) to encrypt and decrypt documents.

GnuPG

The package uses the **libgcrypt** module, which is not validated.

GRUB2

Supports legacy firmware protocols requiring SHA-1 and includes the **libgcrypt** library.

iPXE

Implements TLS stack.

Kerberos

Preserves support for SHA-1 (interoperability with Windows).

Lasso

The **lasso_wsse_username_token_derive_key()** key derivation function (KDF) uses SHA-1.

libgcrypt

The module is deprecated. It is no longer validated since RHEL 10.0.

MariaDB, MariaDB Connector

The **mysql_native_password** authentication plugin uses SHA-1.

MySQL

mysql_native_password uses SHA-1.

OpenIPMI

The RAKP-HMAC-MD5 authentication method is not approved for FIPS usage and does not work in FIPS mode.

Ovmf (UEFI firmware), Edk2, shim

Full cryptographic stack (an embedded copy of the OpenSSL library).

Perl

Uses HMAC, HMAC-SHA1, HMAC-MD5, SHA-1, and SHA-224.

Pidgin

Implements DES and RC4 ciphers.

Poppler

Can save PDFs with signatures, passwords, and encryption based on non-allowed algorithms if they are present in the original PDF (for example, MD5, RC4, and SHA-1).

PostgreSQL

Implements Blowfish, DES, and MD5. A KDF uses SHA-1.

QAT Engine

Uses a mix of hardware and software implementation of cryptographic primitives (RSA, EC, DH, AES, and others).

Ruby

Provides insecure MD5 and SHA-1 library functions.

Samba

Preserves support for RC4 and DES (interoperability with Windows).

Sequoia

Uses the deprecated OpenSSL API, which does not work in FIPS mode.

Syslinux

Firmware passwords use SHA-1.

SWTPM

Explicitly disables FIPS mode in its OpenSSL usage.

Unbound

DNS specification requires that DNSSEC resolvers use a SHA-1-based algorithm in DNSKEY records for validation.

Valgrind

AES, SHA hashes.[1]

zip

Custom cryptography implementation (insecure PKWARE encryption algorithm) to encrypt and decrypt archives by using a password.

Additional resources

- FIPS - Federal Information Processing Standards (Red Hat Customer Portal)
- RHEL core cryptographic components (Red Hat Knowledgebase)
- FIPS 140-3 changes for PKCS #12 (Red Hat Blog)

[1] Re-implements in software hardware-offload operations, such as AES-NI or SHA-1 and SHA-2 on ARM.

CHAPTER 2. USING SYSTEM-WIDE CRYPTOGRAPHIC POLICIES

The system-wide cryptographic policies component configures the core cryptographic subsystems, which cover the TLS, IPsec, SSH, DNSSEC, and Kerberos protocols. As an administrator, you can select one of the provided cryptographic policies for your system.

2.1. SYSTEM-WIDE CRYPTOGRAPHIC POLICIES

The RHEL system-wide cryptographic policies configure core subsystems, such as TLS and SSH, which ensures that applications reject weak algorithms by default. The four predefined policies are **DEFAULT**, **LEGACY**, **FUTURE**, and **FIPS**.

When a system-wide policy is set up, applications in RHEL follow it and deny using algorithms and protocols that do not meet the policy, unless you explicitly request the application to do so. That is, the policy applies to the default behavior of applications when running with the system-provided configuration but you can override it if required.

2.1.1. Predefined cryptographic policies

RHEL 10 contains the following predefined policies:

DEFAULT

The default system-wide cryptographic policy level offers secure settings for current threat models. It allows the TLS 1.2 and 1.3 protocols, as well as the IKEv2 and SSH2 protocols. The RSA keys and Diffie-Hellman parameters are accepted if they are at least 2048 bits long. TLS ciphers that use the RSA key exchange are rejected.

LEGACY

Ensures maximum compatibility with Red Hat Enterprise Linux 6 and earlier; it is less secure due to an increased attack surface. CBC-mode ciphers are allowed to be used with SSH. It allows the TLS 1.2 and 1.3 protocols, as well as the IKEv2 and SSH2 protocols. The RSA keys and Diffie-Hellman parameters are accepted if they are at least 2048 bits long. SHA-1 signatures are allowed outside TLS. Ciphers that use the RSA key exchange are accepted.

FUTURE

A stricter forward-looking security level intended for testing a possible future policy. This policy does not allow the use of SHA-1 in DNSSEC or as an HMAC. SHA2-224 and SHA3-224 hashes are rejected. 128-bit ciphers are disabled. CBC-mode ciphers are disabled except in Kerberos. It allows the TLS 1.2 and 1.3 protocols, as well as the IKEv2 and SSH2 protocols. The RSA keys and Diffie-Hellman parameters are accepted if they are at least 3072 bits long. If your system communicates on the public internet, you might face interoperability problems.

> IMPORTANT
>
> Because a cryptographic key used by a certificate on the Customer Portal API does not meet the requirements by the **FUTURE** system-wide cryptographic policy, the **redhat-support-tool** utility does not work with this policy level at the moment.
>
> To work around this problem, use the **DEFAULT** cryptographic policy while connecting to the Customer Portal API.

FIPS

Conforms with the FIPS 140 requirements. Red Hat Enterprise Linux systems in FIPS mode use this policy.

NOTE

Your system is not FIPS-compliant after you set the **FIPS** cryptographic policy. The only correct way to make your RHEL system compliant with the FIPS 140 standards is by installing it in FIPS mode.

RHEL also provides the **FIPS:OSPP** system-wide subpolicy, which contains further restrictions for cryptographic algorithms required by the Common Criteria (CC) certification. The system becomes less interoperable after you set this subpolicy. For example, you cannot use RSA and DH keys shorter than 3072 bits, additional SSH algorithms, and several TLS groups. Setting **FIPS:OSPP** also prevents connecting to Red Hat Content Delivery Network (CDN) structure. Furthermore, you cannot integrate Active Directory (AD) into the IdM deployments that use **FIPS:OSPP**, communication between RHEL hosts that use **FIPS:OSPP** and AD domains might not work, or some AD accounts might not be able to authenticate.

NOTE

Your system is not CC-compliant after you set the **FIPS:OSPP** cryptographic subpolicy. The only correct way to make your RHEL system compliant with the CC standard is by following the guidance provided in the **cc-config** package. See the Common Criteria section on the Product compliance Red Hat Customer Portal page for a list of certified RHEL versions, validation reports, and links to CC guides.

Red Hat continuously adjusts all policy levels so that all libraries provide secure defaults, except when using the **LEGACY** policy. Even though the **LEGACY** profile does not provide secure defaults, it does not include any algorithms that are easily exploitable. As such, the set of enabled algorithms or acceptable key sizes in any provided policy might change during the lifetime of Red Hat Enterprise Linux.

Such changes reflect new security standards and new security research. If you must ensure interoperability with a specific system for the whole lifetime of Red Hat Enterprise Linux, opt out from the system-wide cryptographic policies for components that interact with that system or re-enable specific algorithms by using custom cryptographic policies.

The specific algorithms and ciphers described as allowed in the policy levels are available only if an application supports them:

Table 2.1. Cipher suites and protocols enabled in the cryptographic policies

	LEGACY	DEFAULT	FIPS	FUTURE
IKEv1	no	no	no	no
3DES	no	no	no	no
RC4	no	no	no	no
DH	min. 2048-bit	min. 2048-bit	min. 2048-bit	min. 3072-bit

	LEGACY	DEFAULT	FIPS	FUTURE
RSA	min. 2048-bit	min. 2048-bit	min. 2048-bit	min. 3072-bit
DSA	no	no	no	no
TLS v1.1 and older	no	no	no	no
TLS v1.2 and newer	yes	yes	yes	yes
SHA-1 in digital signatures and certificates	yes[a]	no	no	no
CBC mode ciphers	yes	no[b]	no[c]	no[d]
Symmetric ciphers with keys < 256 bits	yes	yes	yes	no

[a] SHA-1 signatures are disabled in TLS contexts

[b] CBC ciphers are disabled for SSH

[c] CBC ciphers are disabled for all protocols except Kerberos

[d] CBC ciphers are disabled for all protocols except Kerberos

You can find further details about cryptographic policies and covered applications in the **crypto-policies(7)** man page on your system.

2.1.2. Post-quantum cryptography

RHEL 10.0 introduced post-quantum cryptography (PQC) in its core cryptographic subsystems. The purpose of PQC is to safeguard the confidentiality, integrity, and authenticity of digital communication and stored data against future attacks that use cryptographically relevant quantum computers (CRQCs).

While CRQCs are not currently operational, continued advances in quantum computing might render classic cryptographic algorithms such as RSA and elliptic curve cryptography computationally feasible to break. PQC algorithms mitigate this anticipated long-term security risk by providing quantum-resistant equivalents. You can prevent the current risk of "harvest now, decrypt later" attacks by configuring RHEL with PQC.

Starting with RHEL 10.1, post-quantum cryptography (PQC) algorithms are enabled by default in all predefined policy levels. To turn off support for post-quantum cryptography, apply the **NO-PQ** subpolicy to your selected cryptographic policy.

Additional resources

- Product compliance (Red Hat Customer Portal)
- Enable SHA-1 signatures in OpenSSL on Red Hat Enterprise Linux 10 (Red Hat Knowledgebase)
- Post-quantum cryptography in Red Hat Enterprise Linux 10 (Red Hat Blog)
- Interoperability of RHEL 10 post-quantum cryptography (Red Hat Knowledgebase)
- Red Hat’s path to post-quantum cryptography (Red Hat Blog)
- How Red Hat is integrating post-quantum cryptography into our products (Red Hat Blog)

2.2. CHANGING THE SYSTEM-WIDE CRYPTOGRAPHIC POLICY

You can change the system-wide cryptographic policy on your system by using the **update-crypto-policies** tool and restarting your system. With **update-crypto-policies**, you can quickly switch between predefined policies such as DEFAULT, LEGACY, or FUTURE.

The **update-crypto-policies(8)** man page on your system provides the reference of all options, corresponding files, and per-application details.

Prerequisites

- You have root privileges on the system.

Procedure

1. Optional: Display the current cryptographic policy:

   ```
   $ update-crypto-policies --show
   DEFAULT
   ```

2. Set the new cryptographic policy:

   ```
   # update-crypto-policies --set <POLICY>
   <POLICY>
   ```

   Replace ***<POLICY>*** with the policy or subpolicy you want to set, for example, **FUTURE**, **LEGACY**, or **FIPS:OSPP**.

3. Restart the system:

   ```
   # reboot
   ```

Verification

- Display the current cryptographic policy:

  ```
  $ update-crypto-policies --show
  <POLICY>
  ```

Additional resources

- System-wide cryptographic policies

2.3. SWITCHING THE SYSTEM-WIDE CRYPTOGRAPHIC POLICY TO MODE COMPATIBLE WITH EARLIER RELEASES

Switch to the **LEGACY** cryptographic policy if environments require compatibility with systems such as RHEL 6 or earlier. This action sacrifices security for interoperability with insecure protocols.

> **WARNING**
>
> Switching to the **LEGACY** policy results in a less secure system and applications.

Prerequisites

- Commands that start with the **#** command prompt require administrative privileges provided by **sudo** or root user access. For information on how to configure **sudo** access, see Enabling unprivileged users to run certain commands.

Procedure

1. To switch the system-wide cryptographic policy to **LEGACY**, enter:

   ```
   # update-crypto-policies --set LEGACY
   Setting system policy to LEGACY
   ```

   For the list of available cryptographic policies, see the **update-crypto-policies(8)** man page on your system.

2. To make your cryptographic settings effective for already running services and applications, restart the system:

   ```
   $ reboot
   ```

Verification

- After the restart, verify the current policy is set to **LEGACY**:

  ```
  $ update-crypto-policies --show
  LEGACY
  ```

Next steps

- For defining custom cryptographic policies, see the **Custom Policies** section in the **update-crypto-policies(8)** man page and the **Crypto Policy Definition Format** section in the **crypto-policies(7)** man page on your system.

2.4. SETTING UP SYSTEM-WIDE CRYPTOGRAPHIC POLICIES IN THE WEB CONSOLE

You can set up system-wide cryptographic policies in the RHEL web console. This provides a graphical interface for changing policies such as DEFAULT, LEGACY, or FUTURE.

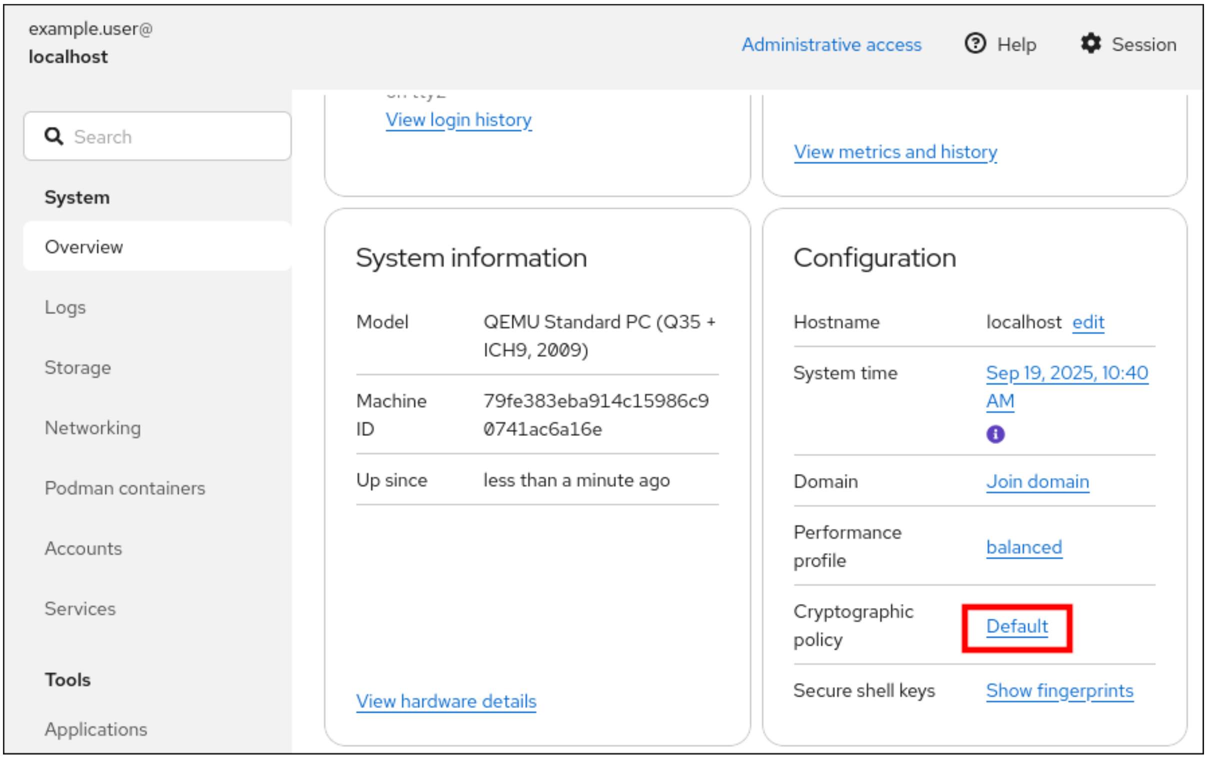
Besides the three predefined system-wide cryptographic policies, you can also apply a combination of the **LEGACY** policy and the **AD-SUPPORT** subpolicy through the graphical interface. The **LEGACY:AD-SUPPORT** policy is the **LEGACY** policy with less secure settings that improve interoperability for Active Directory services.

Prerequisites

- You have installed the RHEL 10 web console.
  For instructions, see Installing and enabling the web console .
- Commands that start with the **#** command prompt require administrative privileges provided by **sudo** or root user access. For information on how to configure **sudo** access, see Enabling unprivileged users to run certain commands.

Procedure

1. Log in to the RHEL 10 web console.
2. In the **Configuration** card of the **Overview** page, click your current policy value next to **Crypto policy**.



System information

Model QEMU Standard PC (Q35 + ICH9, 2009)

Machine ID 79fe383eba914c15986c90741ac6a16e

Up since less than a minute ago

Configuration

Hostname localhost edit

System time Sep 19, 2025, 10:40 AM

Domain Join domain

Performance profile balanced

Cryptographic policy Default

Secure shell keys Show fingerprints

3. In the **Change crypto policy** dialog window, click the policy you want to start using on your system.
4. Click the **Apply and reboot** button.

Verification

- After the restart, log back in to web console, and check that the **Crypto policy** value corresponds to the one you selected.
  Alternatively, you can enter the **update-crypto-policies --show** command to display the current system-wide cryptographic policy in your terminal.

2.5. EXAMPLES OF OPTING OUT OF SYSTEM-WIDE CRYPTOGRAPHIC POLICIES

Review examples demonstrating how to opt out of system-wide crypto policies for applications such as OpenSSH and wget. Use these methods when specific applications require legacy ciphers or protocols.

You can customize cryptographic settings used by your application by configuring supported cipher suites and protocols directly in the application.

You can also remove a symlink related to your application from the **/etc/crypto-policies/back-ends** directory and replace it with your customized cryptographic settings. This configuration prevents system-wide cryptographic policies from being applied to applications that use the excluded back end. Furthermore, this modification is not supported by Red Hat.

curl

To specify ciphers used by the **curl** tool, use the **--ciphers** option and provide a colon-separated list of ciphers as a value. For example:

```
$ curl <https://example.com> --ciphers '@SECLEVEL=0:DES-CBC3-SHA:RSA-DES-CBC3-SHA'
```

See the **curl(1)** man page for more information.

Libreswan

See the Enabling legacy ciphers and algorithms in Libreswan section in the *Securing networks* document for detailed information.

Mozilla Firefox

Even though you cannot opt out of system-wide cryptographic policies in the Mozilla Firefox web browser, you can further restrict supported ciphers and TLS versions in the Firefox's Configuration Editor. Type **about:config** in the address bar and change the value of the **security.tls.version.min** option as required. Setting **security.tls.version.min** to **1** allows TLS 1.0 as the minimum required, **security.tls.version.min 2** enables TLS 1.1, and so on.

OpenSSH server

To opt out of the system-wide cryptographic policies for your OpenSSH server, specify the cryptographic policy in a drop-in configuration file located in the **/etc/ssh/sshd_config.d/** directory. Use a two-digit number prefix smaller than 50, so that it lexicographically precedes the **50-redhat.conf** file, and a **.conf** suffix, for example, **49-crypto-policy-override.conf**.
See the **sshd_config(5)** man page for more information.

OpenSSH client

To opt out of system-wide cryptographic policies for your OpenSSH client, perform one of the following tasks:

- For a given user, override the global **ssh_config** with a user-specific configuration in the **~/.ssh/config** file.
- For the entire system, specify the cryptographic policy in a drop-in configuration file located

in the **/etc/ssh/ssh_config.d/** directory, with a two-digit number prefix smaller than 50, so that it lexicographically precedes the **50-redhat.conf** file, and with a **.conf** suffix, for example, **49-crypto-policy-override.conf**.

See the **ssh_config(5)** man page for more information.

wget

To customize cryptographic settings used by the wget network downloader, use the **--secure-protocol** and **--ciphers** options. For example:

```
$ wget --secure-protocol=TLSv1_1 --ciphers="SECURE128" <https://example.com>
```

See the HTTPS (SSL/TLS) Options section of the **wget(1)** man page for more information.

2.6. CUSTOMIZING SYSTEM-WIDE CRYPTOGRAPHIC POLICIES WITH SUBPOLICIES

Adjust the set of enabled cryptographic algorithms or protocols on the system. In RHEL, you can either apply custom subpolicies on top of an existing system-wide cryptographic policy or define such a policy from scratch.

The concept of scoped policies enables different sets of algorithms for different back ends. You can limit each configuration directive to specific protocols, libraries, or services.

Furthermore, you can use wildcard characters in directives, for example, an asterisk to specify multiple values. For the complete syntax reference, see the **Custom Policies** section in the **update-crypto-policies(8)** man page and the **Crypto Policy Definition Format** section in the **crypto-policies(7)** man page on your system.

- The **/etc/crypto-policies/state/CURRENT.pol** file lists all settings in the currently applied system-wide cryptographic policy after wildcard expansion.
- To make your cryptographic policy more strict, consider using values listed in the **/usr/share/crypto-policies/policies/FUTURE.pol** file.
- You can find example subpolicies in the **/usr/share/crypto-policies/policies/modules/** directory.

Procedure

1. Checkout to the **/etc/crypto-policies/policies/modules/** directory:

   ```
   # cd /etc/crypto-policies/policies/modules/
   ```

2. Create subpolicies for your adjustments, for example:

   ```
   # touch <MYCRYPTO-1>.pmod
   # touch <SCOPES-AND-WILDCARDS>.pmod
   ```

IMPORTANT

Use upper-case letters in file names of policy modules.

3. Open the policy modules in a text editor of your choice and insert options that modify the system-wide cryptographic policy, for example:

```
# vi <MYCRYPTO-1>.pmod
```

```
min_rsa_size = 3072
hash = SHA2-384 SHA2-512 SHA3-384 SHA3-512
```

```
# vi <SCOPES-AND-WILDCARDS>.pmod
```

```
# Disable the AES-128 cipher, all modes
cipher = -AES-128-*

# Disable CHACHA20-POLY1305 for the TLS protocol (OpenSSL, GnuTLS, NSS, and
OpenJDK)
cipher@TLS = -CHACHA20-POLY1305

# Allow using the FFDHE-1024 group with the SSH protocol (libssh and OpenSSH)
group@SSH = FFDHE-1024+

# Disable all CBC mode ciphers for the SSH protocol (libssh and OpenSSH)
cipher@SSH = -*-CBC

# Allow the AES-256-CBC cipher in applications using libssh
cipher@libssh = AES-256-CBC+
```

4. Save the changes in the module files.

5. Apply your policy adjustments to the **DEFAULT** system-wide cryptographic policy level:

```
# update-crypto-policies --set DEFAULT:<MYCRYPTO-1>:<SCOPES-AND-WILDCARDS>
```

6. To make your cryptographic settings effective for already running services and applications, restart the system:

```
# reboot
```

Verification

- Check that the **/etc/crypto-policies/state/CURRENT.pol** file contains your changes, for example:

```
$ cat /etc/crypto-policies/state/CURRENT.pol | grep rsa_size
min_rsa_size = 3072
```

Additional resources

- How to customize cryptographic policies in RHEL 8.2 (Red Hat blog)

2.7. CREATING AND SETTING A CUSTOM SYSTEM-WIDE CRYPTOGRAPHIC POLICY

Apply custom subpolicies on top of an existing system-wide policy to modify enabled cryptographic algorithms or protocols. By using this granular customization, you can ensure that your system adheres to your specific security requirements.

Procedure

1. Create a policy file for your customizations:

   ```
   # cd /etc/crypto-policies/policies/
   # touch <MYPOLICY>.pol
   ```

   Alternatively, start by copying one of the four predefined policy levels:

   ```
   # cp /usr/share/crypto-policies/policies/DEFAULT.pol /etc/crypto-
   policies/policies/<MYPOLICY>.pol
   ```

2. Edit the file with your custom cryptographic policy in a text editor of your choice to fit your requirements, for example:

   ```
   # vi /etc/crypto-policies/policies/<MYPOLICY>.pol
   ```

   See the **Custom Policies** section in the **update-crypto-policies(8)** man page and the **Crypto Policy Definition Format** section in the **crypto-policies(7)** man page on your system for the complete syntax reference.

3. Switch the system-wide cryptographic policy to your custom level:

   ```
   # update-crypto-policies --set <MYPOLICY>
   ```

4. To make your cryptographic settings effective for already running services and applications, restart the system:

   ```
   # reboot
   ```

Additional resources

- How to customize cryptographic policies in RHEL 8.2 (Red Hat blog)

2.8. DISABLING POST-QUANTUM ALGORITHMS SYSTEM-WIDE

From RHEL 10.1, post-quantum cryptography (PQC) algorithms are enabled in all predefined policies by default. You can turn them off by applying the **NO-PQ** subpolicy.

Prerequisites

- Commands that start with the **#** command prompt require administrative privileges provided by **sudo** or root user access. For information on how to configure **sudo** access, see Enabling unprivileged users to run certain commands.

Procedure

1. Apply the **NO-PQ** cryptographic subpolicy on top of your current system-wide policy, for example:

   ```
   # update-crypto-policies --show
   DEFAULT
   # update-crypto-policies --set DEFAULT:NO-PQ
   ```

2. To make your cryptographic settings effective for already running services and applications, restart the system:

   ```
   # reboot
   ```

Verification

- Check that the **/etc/crypto-policies/state/CURRENT.pol** file does not contain PQC algorithm strings **MLKEM**, **MLDSA**, and **KEM-ECDH**, for example:

  ```
  $ cat /etc/crypto-policies/state/CURRENT.pol | grep MLKEM
  ```

  The output of the previous command must be empty.

2.9. ENHANCING SECURITY WITH THE FUTURE CRYPTOGRAPHIC POLICY USING THE CRYPTO_POLICIES RHEL SYSTEM ROLE

You can use the **crypto_policies** RHEL system role to configure the **FUTURE** cryptographic policy on your managed nodes.

The **FUTURE** policy helps to achieve, for example:

Future-proofing against emerging threats

Anticipates advancements in computational power.

Enhanced security

Stronger encryption standards require longer key lengths and more secure algorithms.

Compliance with high-security standards

In some industries, for example, in healthcare, telco, and finance the data sensitivity is high, and availability of strong cryptography is critical.

Typically, **FUTURE** is suitable for environments handling highly sensitive data, preparing for future regulations, or adopting long-term security strategies.

WARNING

Legacy systems and software do not have to support the more modern and stricter algorithms and protocols enforced by the **FUTURE** policy. For example, older systems might not support TLS 1.3 or larger key sizes. This could lead to interoperability problems.

Also, using strong algorithms usually increases the computational workload, which could negatively affect your system performance.

Prerequisites

- You have prepared the control node and the managed nodes .
- You are logged in to the control node as a user who can run playbooks on the managed nodes.
- The account you use to connect to the managed nodes has **sudo** permissions for these nodes.

Procedure

1. Create a playbook file, for example, **~/playbook.yml**, with the following content:

```
---
- name: Configure cryptographic policies
  hosts: managed-node-01.example.com
  tasks:
    - name: Configure the FUTURE cryptographic security policy on the managed node
      ansible.builtin.include_role:
        name: redhat.rhel_system_roles.crypto_policies
      vars:
        - crypto_policies_policy: FUTURE
        - crypto_policies_reboot_ok: true
```

The settings specified in the example playbook include the following:

crypto_policies_policy: *FUTURE*

Configures the required cryptographic policy (**FUTURE**) on the managed node. It can be either the base policy or a base policy with some subpolicies. The specified base policy and subpolicies have to be available on the managed node. The default value is **null**, which means that the configuration is not changed and the **crypto_policies** RHEL system role only collects the Ansible facts.

crypto_policies_reboot_ok: *true*

Causes the system to reboot after the cryptographic policy change to make sure all of the services and applications will read the new configuration files. The default value is **false**.

For details about the role variables and the cryptographic configuration options, see the **/usr/share/ansible/roles/rhel-system-roles.crypto_policies/README.md** file and the **update-crypto-policies(8)** and **crypto-policies(7)** manual pages on the control node.

2. Validate the playbook syntax:

```
$ ansible-playbook --syntax-check ~/playbook.yml
```

Note that this command only validates the syntax and does not protect against a wrong but valid configuration.

3. Run the playbook:

```
$ ansible-playbook ~/playbook.yml
```

Verification

1. On the control node, create another playbook named, for example, **verify_playbook.yml**:

```
---
- name: Verification
  hosts: managed-node-01.example.com
  tasks:
    - name: Verify active cryptographic policy
      ansible.builtin.include_role:
        name: redhat.rhel_system_roles.crypto_policies
    - name: Display the currently active cryptographic policy
      ansible.builtin.debug:
        var: crypto_policies_active
```

The settings specified in the example playbook include the following:

crypto_policies_active

An exported Ansible fact that contains the currently active policy name in the format as accepted by the **crypto_policies_policy** variable.

2. Validate the playbook syntax:

```
$ ansible-playbook --syntax-check ~/verify_playbook.yml
```

3. Run the playbook:

```
$ ansible-playbook ~/verify_playbook.yml
TASK [debug] **************************
ok: [host] => {
    "crypto_policies_active": "FUTURE"
}
```

The **crypto_policies_active** variable shows the active policy on the managed node.

CHAPTER 3. CONFIGURING APPLICATIONS TO USE CRYPTOGRAPHIC HARDWARE THROUGH PKCS #11

Configure applications to use cryptographic hardware such as smart cards and HSMs through the consistent PKCS #11 API. Isolating secrets on hardware helps provide an additional layer of security.

3.1. CRYPTOGRAPHIC HARDWARE SUPPORT THROUGH PKCS #11

In RHEL, you can use the PKCS #11 API standard and the **p11-kit** tool to access cryptographic tokens such as smart cards and hardware security modules (HSMs). By isolating keys on these dedicated devices, you ensure your sensitive data is managed securely.

Public-Key Cryptography Standard (PKCS) #11 defines an application programming interface (API) to cryptographic devices that hold cryptographic information and perform cryptographic functions.

PKCS #11 introduces the *cryptographic token*, an object that presents each hardware or software device to applications in a unified manner. Therefore, applications view devices such as smart cards, which are typically used by persons, and hardware security modules, which are typically used by computers, as PKCS #11 cryptographic tokens.

A PKCS #11 token can store various object types including a certificate; a data object; and a public, private, or secret key. These objects are uniquely identifiable through the PKCS #11 Uniform Resource Identifier (URI) scheme.

A PKCS #11 URI is a standard way to identify a specific object in a PKCS #11 module according to the object attributes. This enables you to configure all libraries and applications with the same configuration string in the form of a URI.

RHEL provides the OpenSC PKCS #11 driver for smart cards by default. However, hardware tokens and HSMs can have their own PKCS #11 modules that do not have their counterpart in the system. You can register such PKCS #11 modules with the **p11-kit** tool, which acts as a wrapper over the registered smart-card drivers in the system.

To make your own PKCS #11 module work on the system, add a new text file to the **/etc/pkcs11/modules/** directory.

You can add your own PKCS #11 module into the system by creating a new text file in the **/etc/pkcs11/modules/** directory. For example, the OpenSC configuration file in **p11-kit** looks as follows:

```
$ cat /usr/share/p11-kit/modules/opensc.module
module: opensc-pkcs11.so
```

Additional resources

- The PKCS #11 URI Scheme (IETF.org)
- Controlling access to smart cards (Red Hat Blog)

3.2. AUTHENTICATING BY SSH KEYS STORED ON A SMART CARD

Use SSH keys stored on a smart card for authentication to add a physical layer of protection to your credentials. This method provides enhanced security against unauthorized access.

You can create and store ECDSA and RSA keys on a smart card and authenticate by the smart card on an OpenSSH client. Smart-card authentication replaces the default password authentication.

Prerequisites

- On the client side, the **opensc** package is installed and the **pcscd** service is running.

Procedure

1. List all keys provided by the OpenSC PKCS #11 module including their PKCS #11 URIs and save the output to the **keys.pub** file:

   ```
   $ ssh-keygen -D pkcs11: > keys.pub
   ```

2. Transfer the public key to the remote server. Use the **ssh-copy-id** command with the **keys.pub** file created in the previous step:

   ```
   $ ssh-copy-id -f -i keys.pub <username@ssh-server-example.com>
   ```

3. Connect to ***<ssh-server-example.com>*** by using the ECDSA key. You can use just a subset of the URI, which uniquely references your key, for example:

   ```
   $ ssh -i "pkcs11:id=%01?module-path=/usr/lib64/pkcs11/opensc-pkcs11.so" <ssh-server-example.com>
   Enter PIN for 'SSH key':
   [ssh-server-example.com] $
   ```

   Because OpenSSH uses the **p11-kit-proxy** wrapper and the OpenSC PKCS #11 module is registered to the **p11-kit** tool, you can simplify the previous command:

   ```
   $ ssh -i "pkcs11:id=%01" <ssh-server-example.com>
   Enter PIN for 'SSH key':
   [ssh-server-example.com] $
   ```

   If you skip the **id=** part of a PKCS #11 URI, OpenSSH loads all keys that are available in the proxy module. This can reduce the amount of typing required:

   ```
   $ ssh -i pkcs11: <ssh-server-example.com>
   Enter PIN for 'SSH key':
   [ssh-server-example.com] $
   ```

4. Optional: You can use the same URI string in the **~/.ssh/config** file to make the configuration permanent:

   ```
   $ cat ~/.ssh/config
   IdentityFile "pkcs11:id=%01?module-path=/usr/lib64/pkcs11/opensc-pkcs11.so"
   $ ssh <ssh-server-example.com>
   Enter PIN for 'SSH key':
   [ssh-server-example.com] $
   ```

   The **ssh** client utility now automatically uses this URI and the key from the smart card.

3.3. APPLICATION SETTINGS FOR AUTHENTICATION WITH CERTIFICATES ON SMART CARDS

You can configure applications such as **curl** to authenticate using certificates stored on a smart card. This robust method helps protect your system with physical security credentials.

Authentication by using smart cards in applications might increase security and simplify automation. You can integrate the Public Key Cryptography Standard (PKCS) #11 URIs into your application by using the following methods:

- The **Firefox** web browser automatically loads the **p11-kit-proxy** PKCS #11 module. This means that every supported smart card in the system is automatically detected. For using TLS client authentication, no additional setup is required and keys and certificates from a smart card are automatically used when a server requests them.

- If your application uses the **GnuTLS** or **NSS** library, it already supports PKCS #11 URIs. Also, applications that rely on the **OpenSSL** library can access cryptographic hardware modules, including smart cards, through the PKCS #11 provider installed by the **pkcs11-provider** package.

- Applications that require working with private keys on smart cards and that do not use **NSS**, **GnuTLS**, nor **OpenSSL** can use the **p11-kit** API directly to work with cryptographic hardware modules, including smart cards, rather than using the PKCS #11 API of specific PKCS #11 modules.

- With the **wget** network downloader, you can specify PKCS #11 URIs instead of paths to locally stored private keys and certificates. This might simplify creation of scripts for tasks that require safely stored private keys and certificates. For example:

  ```
  $ wget --private-key 'pkcs11:token=softhsm;id=%01;type=private?pin-value=111111' --
  certificate 'pkcs11:token=softhsm;id=%01;type=cert' https://example.com/
  ```

- You can also specify PKCS #11 URI when using the **curl** tool:

  ```
  $ curl --key 'pkcs11:token=softhsm;id=%01;type=private?pin-value=111111' --cert
  'pkcs11:token=softhsm;id=%01;type=cert' https://example.com/
  ```

NOTE

Because a PIN is a security measure that controls access to keys stored on a smart card and the configuration file contains the PIN in the plain text form, consider additional protection to prevent an attacker from reading the PIN. For example, you can use the **pin-source** attribute and provide a **file:** URI for reading the PIN from a file. See RFC 7512: PKCS #11 URI Scheme Query Attribute Semantics for more information. Note that using a command path as a value of the **pin-source** attribute is not supported.

See the **curl(1)**, **wget(1)**, **p11-kit(8)**, and **provider-pkcs11(7)** man pages on your system for details and additional examples.

3.4. USING HSMS PROTECTING PRIVATE KEYS IN APACHE

The Apache HTTP server can work with private keys stored on hardware security modules (HSMs), which helps to prevent the keys' disclosure and man-in-the-middle attacks. Note that this usually requires high-performance HSMs for busy servers.

For secure communication in the form of the HTTPS protocol, the Apache HTTP server (**httpd**) uses the OpenSSL library. OpenSSL does not support PKCS #11 natively. To use HSMs, you must install the **pkcs11-provider** package, which provides access to PKCS #11 modules. You can use a PKCS #11 URI instead of a regular file name to specify a server key and a certificate in the **/etc/httpd/conf.d/ssl.conf** configuration file, for example:

```
SSLCertificateFile    "pkcs11:id=%01;token=softhsm;type=cert"
SSLCertificateKeyFile "pkcs11:id=%01;token=softhsm;type=private?pin-value=111111"
```

Install the **httpd-manual** package to obtain complete documentation for the Apache HTTP Server, including TLS configuration. The directives available in the **/etc/httpd/conf.d/ssl.conf** configuration file are described in detail in the **/usr/share/httpd/manual/mod/mod_ssl.html** file.

CHAPTER 4. CONTROLLING ACCESS TO SMART CARDS BY USING POLKIT

Configure the **polkit** framework in RHEL to control access to smart cards. This provides fine-grained control and mitigates threats that built-in mechanisms, such as PINs, PIN pads, and biometrics, cannot prevent.

System administrators can configure **polkit** to fit specific scenarios, such as smart-card access for non-privileged or non-local users or services.

4.1. SMART-CARD ACCESS CONTROL THROUGH POLKIT

With the **polkit** authorization manager, you can define precise policies for securing smart cards. Administrators can use this framework to control which users can perform specific privileged operations with the card.

The Personal Computer/Smart Card (PC/SC) protocol specifies a standard for integrating smart cards and their readers into computing systems. In RHEL, the **pcsc-lite** package provides middleware to access smart cards that use the PC/SC API. A part of this package, the **pcscd** (PC/SC Smart Card) daemon, ensures that the system can access a smart card by using the PC/SC protocol.

Because access-control mechanisms built into smart cards, such as PINs, PIN pads, and biometrics, do not cover all possible threats, RHEL uses the **polkit** framework for more robust access control. The **polkit** authorization manager can grant access to privileged operations. In addition to granting access to disks, you can use **polkit** also to specify policies for securing smart cards. For example, you can define which users can perform which operations with a smart card.

After installing the **pcsc-lite** package and starting the **pcscd** daemon, the system enforces policies defined in the **/usr/share/polkit-1/actions/** directory. The default system-wide policy is in the **/usr/share/polkit-1/actions/org.debian.pcsc-lite.policy** file. Polkit policy files use the XML format and the syntax is described in the **polkit(8)** man page.

The **polkitd** service monitors the **/etc/polkit-1/rules.d/** and **/usr/share/polkit-1/rules.d/** directories for any changes in rule files stored in these directories. The files contain authorization rules in JavaScript format. System administrators can add custom rule files in both directories, and **polkitd** reads them in lexical order based on their file name. If two files have the same names, then the file in **/etc/polkit-1/rules.d/** is read first.

4.2. TROUBLESHOOTING PROBLEMS RELATED TO PC/SC AND POLKIT

You can troubleshoot smart-card authentication failures by checking the Journal log for messages related to the **pcscd** daemon and the **polkit** authorization framework. This helps identify policy or access denials.

Polkit policies that are automatically enforced after you install the **pcsc-lite** package and start the **pcscd** daemon might ask for authentication in the user’s session even if the user does not directly interact with a smart card. In GNOME, you can see the following error message:

```
Authentication is required to access the PC/SC daemon
```

Note that the system can install the **pcsc-lite** package as a dependency when you install other packages related to smart cards such as **opensc**.

If your scenario does not require any interaction with smart cards and you want to prevent displaying authorization requests for the PC/SC daemon, you can remove the **pcsc-lite** package. Keeping the minimum of necessary packages is a good security practice anyway.

If you use smart cards, start troubleshooting by checking the rules in the system-provided policy file at **/usr/share/polkit-1/actions/org.debian.pcsc-lite.policy**. You can add your custom rule files to the policy in the **/etc/polkit-1/rules.d/** directory, for example, **03-allow-pcscd.rules**. Note that the rule files use the JavaScript syntax, the policy file is in the XML format.

To understand what authorization requests the system displays, check the Journal log, for example:

```
$ journalctl -b | grep pcsc
…
Process 3087 (user: 1001) is NOT authorized for action: access_pcsc
…
```

The previous log entry means that the user is not authorized to perform an action by the policy. You can solve this denial by adding a corresponding rule to **/etc/polkit-1/rules.d/**.

You can search also for log entries related to the **polkitd** unit, for example:

```
$ journalctl -u polkit
…
polkitd[NNN]: Error compiling script /etc/polkit-1/rules.d/00-debug-pcscd.rules
…
polkitd[NNN]: Operator of unix-session:c2 FAILED to authenticate to gain authorization for action
org.debian.pcsc-lite.access_pcsc for unix-process:4800:14441 [/usr/libexec/gsd-smartcard] (owned
by unix-user:group)
…
```

In the previous output, the first entry means that the rule file contains some syntax error. The second entry means that the user failed to gain the access to **pcscd**.

You can also list all applications that use the PC/SC protocol by a short script. Create an executable file, for example, **pcsc-apps.sh**, and insert the following code:

```
#!/bin/bash

cd /proc

for p in [0-9]*
do
 if grep libpcsclite.so.1.0.0 $p/maps &> /dev/null
 then
  echo -n "process: "
  cat $p/cmdline
  echo " ($p)"
 fi
done
```

Run the script as **root**:

```
# ./pcsc-apps.sh
process: /usr/libexec/gsd-smartcard (3048)
enable-sync --auto-ssl-client-auth --enable-crashpad (4828)
…
```

4.3. DISPLAYING MORE DETAILED INFORMATION ABOUT POLKIT AUTHORIZATION TO PC/SC

In the default configuration, the **polkit** authorization framework sends only limited information to the Journal log. You can extend **polkit** log entries related to the PC/SC protocol by adding new rules.

Prerequisites

- You have installed the **pcsc-lite** package on your system.
- The **pcscd** daemon is running.

Procedure

1. Create a new file in the **/etc/polkit-1/rules.d/** directory:

   ```
   # touch /etc/polkit-1/rules.d/00-test.rules
   ```

2. Edit the file in an editor of your choice, for example:

   ```
   # vi /etc/polkit-1/rules.d/00-test.rules
   ```

3. Insert the following lines:

   ```
   polkit.addRule(function(action, subject) {
     if (action.id == "org.debian.pcsc-lite.access_pcsc" ||
      action.id == "org.debian.pcsc-lite.access_card") {
    polkit.log("action=" + action);
    polkit.log("subject=" + subject);
     }
   });
   ```

   Save the file, and exit the editor.

4. Restart the **pcscd** and **polkit** services:

   ```
   # systemctl restart pcscd.service pcscd.socket polkit.service
   ```

Verification

1. Make an authorization request for **pcscd**. For example, open the Mozilla Firefox web browser or use the **pkcs11-tool -L** command provided by the **opensc** package.

2. Display the extended log entries, for example:

   ```
   # journalctl -u polkit --since "1 hour ago"
   polkitd[1224]: <no filename>:4: action=[Action id='org.debian.pcsc-lite.access_pcsc']
   ```

```
polkitd[1224]: <no filename>:5: subject=[Subject pid=2020481 user=user'
groups=user,wheel,mock,wireshark seat=null session=null local=true active=true]
```

4.4. ADDITIONAL RESOURCES

- Controlling access to smart cards (Red Hat Blog)

CHAPTER 5. SCANNING THE SYSTEM FOR CONFIGURATION COMPLIANCE

Scan your RHEL configuration against specific rules defined in a compliance policy. With these checklists, you can ensure that your environment adheres to strict security requirements.

Security professionals define the compliance policy by specifying the required settings for a computing environment, often in the form of a checklist.

Compliance policies can vary substantially across organizations and even across different systems within the same organization. Differences among these policies are based on the purpose of each system and its importance for the organization. Custom software settings and deployment characteristics also raise a need for custom policy checklists.

5.1. CONFIGURATION COMPLIANCE TOOLS IN RHEL

You can use tools such as OpenSCAP and the SCAP Security Guide (SSG) in RHEL to audit system security and maintain compliance with established security baselines.

RHEL 10 provides a set of configuration-compliance tools for performing a fully automated compliance audit. These tools are based on the Security Content Automation Protocol (SCAP) standard and are designed for automated tailoring of compliance policies.

OpenSCAP

The **OpenSCAP** library, with the accompanying **oscap** command-line utility, is designed to perform configuration scans on a local system, to validate configuration compliance content, and to generate reports and guides based on these scans and evaluations. With **oscap**, you can scan systems to assess their alignment with security policies contained in **scap-security-guide**. You can also perform an automated remediation that configures the system into a state that is aligned with a selected policy.

IMPORTANT

You can experience memory-consumption problems while using **OpenSCAP**, which can cause stopping the program prematurely and prevent generating any result files. See the OpenSCAP memory-consumption problems Knowledgebase article for details.

SCAP Security Guide (SSG)

The **scap-security-guide** package provides collections of security policies for Linux systems. The guidance consists of a catalog of practical hardening advice, linked to government requirements where applicable. The project bridges the gap between generalized policy requirements and specific implementation guidelines.

Script Check Engine (SCE)

With SCE, which is an extension to the SCAP protocol, administrators can write their security content by using a scripting language, such as Bash, Python, and Ruby. The SCE extension is provided in the **openscap-engine-sce** package. The SCE itself is not part of the SCAP standard.

Alternatively, you can perform automated compliance audits on multiple systems remotely by using the OpenSCAP solution for Red Hat Satellite.

Additional resources

- Red Hat Security Demos: Creating Customized Security Policy Content to Automate Security Compliance
- Red Hat Security Demos: Defend Yourself with RHEL Security Technologies
- Managing security compliance in Red Hat Satellite

5.2. CONFIGURATION COMPLIANCE SCANNING

Verify if your Red Hat Enterprise Linux systems adhere to security baselines, such as industry standards or internal policies, by performing a configuration compliance scan. You can scan local and remote systems, containers, and container images using OpenSCAP and the SCAP Security Guide.

5.2.1. Configuration compliance in RHEL

Use configuration compliance scanning to conform to a baseline defined by a specific organization. For example, if you are a payment processor, you can align your systems with the Payment Card Industry Data Security Standard (PCI-DSS). You can also perform scanning to harden your system security.

Follow the Security Content Automation Protocol (SCAP) content provided in the SCAP Security Guide package because it is in line with Red Hat best practices for affected components.

The SCAP Security Guide package provides content which conforms to the SCAP 1.2 and SCAP 1.3 standards. The **openscap scanner** utility is compatible with both SCAP 1.2 and SCAP 1.3 content provided in the SCAP Security Guide package.

IMPORTANT

Performing a configuration compliance scanning does not guarantee the system is compliant.

The SCAP Security Guide suite provides profiles for several platforms in the form of data stream documents. A data stream is a file that contains definitions, benchmarks, profiles, and individual rules. Each rule specifies the applicability and requirements for compliance. RHEL provides several profiles for compliance with security policies. In addition to the industry standard, Red Hat data streams also contain information for remediation of failed rules. The data streams use the following structure of compliance scanning resources:

```
Data stream
 ├── xccdf
|      ├── benchmark
|          ├── profile
|          |    ├──rule reference
|          |    └──variable
|          ├── rule
|               ├── human readable data
|               ├── ocil reference
 ├── ocil          ├── cpe reference
 └── cpe            └── remediation
```

A profile is a set of rules based on a security policy, such as PCI-DSS and Health Insurance Portability and Accountability Act (HIPAA). After you select a profile, you can then perform an automated audit of the system for compliance with that profile.

You can also modify, or *tailor*, a profile to customize certain rules, for example, password length. For more information about profile tailoring, see Customizing a security profile with autotailor .

5.2.2. Possible results of an OpenSCAP scan

Understand the possible results, such as Pass, Fail, or Not Applicable, generated when running an OpenSCAP scan. This helps you interpret compliance reports accurately.

Depending on the data stream and profile applied to an OpenSCAP scan and the various properties of your system, each rule produces a specific result:

Pass

The scan did not find any conflicts with this rule.

Fail

The scan found a conflict with this rule.

Not checked

OpenSCAP does not perform an automatic evaluation of this rule. Check whether your system conforms to this rule manually.

Not applicable

This rule does not apply to the current configuration.

Not selected

This rule is not part of the profile. OpenSCAP does not evaluate this rule and does not display these rules in the results.

Error

The scan encountered an error. For additional information, you can enter the **oscap** command with the **--verbose DEVEL** option. File a support case on the Red Hat Customer Portal or open a ticket in the RHEL project in Red Hat Jira .

Unknown

The scan encountered an unexpected situation. For additional information, you can enter the **oscap** command with the **--verbose DEVEL** option. File a support case on the Red Hat Customer Portal or open a ticket in the RHEL project in Red Hat Jira .

5.2.3. Viewing profiles for configuration compliance

Before you decide to use profiles for scanning or remediation, you can list them and check their detailed descriptions by using the **oscap info** subcommand.

Prerequisites

- The **openscap-scanner** and **scap-security-guide** packages are installed.

Procedure

1. List all available files with security compliance profiles provided by the SCAP Security Guide project:

```
$ ls /usr/share/xml/scap/ssg/content/
ssg-rhel10-ds.xml
```

2. Display detailed information about a selected data stream by using the **oscap info** subcommand. XML files containing data streams are indicated by the **-ds** string in their names. In the **Profiles** section, you can find a list of available profiles and their IDs:

```
$ oscap info /usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml
Profiles:
…
  Title: Australian Cyber Security Centre (ACSC) ISM Official - Top Secret
  Id: xccdf_org.ssgproject.content_profile_ism_o_top_secret
 Title: PCI-DSS v4.0.1 Control Baseline for Red Hat Enterprise Linux 10
  Id: xccdf_org.ssgproject.content_profile_pci-dss
 Title: Red Hat STIG for Red Hat Enterprise Linux 10
  Id: xccdf_org.ssgproject.content_profile_stig

…
```

3. Select a profile from the data stream file and display additional details about the selected profile. To do so, use **oscap info** with the **--profile** option followed by the last section of the ID displayed in the output of the previous command. For example, the ID of the HIPPA profile is **xccdf_org.ssgproject.content_profile_hipaa**, and the value for the **--profile** option is **hipaa**:

```
$ oscap info --profile hipaa /usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml
…
Profile
 Title: Health Insurance Portability and Accountability Act (HIPAA)

 Description: The HIPAA Security Rule establishes U.S. national standards to protect
individuals' electronic personal health information that is created, received, used, or
maintained by a covered entity.
…
```

Additional resources

- OpenSCAP memory consumption problems

5.2.4. Assessing configuration compliance with a specific baseline

You can determine whether your system or a remote system conforms to a specific baseline, and save the results in a report by using the **oscap** command-line tool.

Prerequisites

- The **openscap-scanner** and **scap-security-guide** packages are installed.
- You know the ID of the profile within the baseline with which the system should comply. To find the ID, see the Viewing profiles for configuration compliance section.

Procedure

1. Scan the local system for compliance with the selected profile and save the scan results to a file:

```
$ oscap xccdf eval --report <scan_report.html> --profile <profile_ID>
/usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml
```

Replace:

- ***<scan_report.html>*** with the file name where **oscap** saves the scan results.
- ***<profile_ID>*** with the profile ID with which the system should comply, for example, **hipaa**.

2. Optional: Scan a remote system for compliance with the selected profile and save the scan results to a file:

```
$ oscap-ssh <username>@<hostname> <port> xccdf eval --report <scan_report.html> --
profile <profile_ID> /usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml
```

Replace:

- ***<username>@<hostname>*** with the user name and hostname of the remote system.
- ***<port>*** with the port number through which you can access the remote system.
- ***<scan_report.html>*** with the file name where **oscap** saves the scan results.
- ***<profile_ID>*** with the profile ID with which the system should comply, for example, **hipaa**.

Additional resources

- OpenSCAP memory consumption problems

5.2.5. Assessing security compliance of a container or a container image with a specific baseline

Assess the compliance of a container or container image in RHEL with security baselines, such as Payment Card Industry Data Security Standard (PCI-DSS) and Health Insurance Portability and Accountability Act (HIPAA), to identify vulnerabilities and ensure adherence to security standards.

Prerequisites

- The **openscap-utils** and **scap-security-guide** packages are installed.
- You have root access to the system.

Procedure

1. Find the ID of a container or a container image:

   a. To find the ID of a container:

   ```
   # podman ps -a
   ```

   b. To find the ID of a container image:

   ```
   # podman images
   ```

2. Evaluate the compliance of the container or container image with a profile and save the scan results into a file:

```
# oscap-podman <ID> xccdf eval --report <scan_report.html> --profile <profile_ID>
/usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml
```

Replace:

- ***<ID>*** with the ID of your container or container image
- ***<scan_report.html>*** with the file name where **oscap** saves the scan results
- ***<profile_ID>*** with the profile ID with which the system should comply, for example, **hipaa** or **pci-dss**

Verification

- Check the results in a browser of your choice, for example:

```
$ firefox <scan_report.html> &
```

NOTE

The rules marked as **notapplicable** apply only to bare metal and virtual systems and not to containers or container images.

5.3. CONFIGURATION COMPLIANCE REMEDIATION

To automatically align your system with a specific profile, you can perform a remediation. You can remediate the system to align with any profile provided by the SCAP Security Guide.

5.3.1. Remediating the system to align with a specific baseline

Remediate your RHEL system to align with a specific security baseline by using the **oscap xccdf eval --remediate** command. This automatically fixes configuration rules defined in the SCAP Security Guide.

For details on listing available profiles, see the Viewing profiles for configuration compliance section.

WARNING

Remediations are supported on RHEL systems in the default configuration. Remediating a system that has been altered after installation might render the system nonfunctional or noncompliant with the required security profile. Red Hat does not provide any automated method to revert changes made by security-hardening remediations.

Test the effects of the remediation before applying it on production systems.

Prerequisites

- The **openscap-scanner** and **scap-security-guide** packages are installed.

Procedure

1. Remediate the system:

   ```
   # oscap xccdf eval --profile <profile_ID> --remediate /usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml
   ```

   Replace ***<profile_ID>*** with the profile ID with which the system should comply, for example, **hipaa**.

2. Restart your system.

Verification

1. Evaluate compliance of the system with the profile, and save the scan results to a file:

   ```
   $ oscap xccdf eval --report <scan_report.html> --profile <profile_ID> /usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml
   ```

   Replace:

   - ***<scan_report.html>*** with the file name where **oscap** saves the scan results.
   - ***<profile_ID>*** with the profile ID with which the system should comply, for example, **hipaa**.

5.3.2. Remediating the system to align with a specific baseline by using an SSG Ansible Playbook

Use an Ansible Playbook provided by the SCAP Security Guide project to remediate your system against a specific security baseline. This helps ensure consistency and automation across multiple systems.

> **WARNING**
>
> Remediations are supported on RHEL systems in the default configuration. Remediating a system that has been altered after installation might render the system nonfunctional or noncompliant with the required security profile. Red Hat does not provide any automated method to revert changes made by security-hardening remediations.
>
> Test the effects of the remediation before applying it on production systems.

Prerequisites

- The **scap-security-guide** package is installed.
- The **ansible-core** package is installed. See the Ansible Installation Guide for more information.
- The **rhc-worker-playbook** package is installed.

- You know the ID of the profile according to which you want to remediate your system. For details, see Viewing profiles for configuration compliance.

Procedure

1. Remediate your system to align with a selected profile by using Ansible:

   ```
   # ANSIBLE_COLLECTIONS_PATH=/usr/share/rhc-worker-
   playbook/ansible/collections/ansible_collections/ ansible-playbook -i "localhost," -c local
   /usr/share/scap-security-guide/ansible/rhel10-playbook-<profile_ID>.yml
   ```

   The **ANSIBLE_COLLECTIONS_PATH** environment variable is necessary for the command to run the playbook.

   Replace ***<profile_ID>*** with the profile ID of the selected profile.

2. Restart the system.

Verification

- Evaluate the compliance of the system with the selected profile, and save the scan results to a file:

  ```
  # oscap xccdf eval --profile <profile_ID> --report <scan_report.html>
  /usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml
  ```

  Replace ***<scan_report.html>*** with the file name where **oscap** saves the scan results.

Additional resources

- Ansible Documentation

5.3.3. Creating a remediation Ansible Playbook to align the system with a specific baseline

You can create an Ansible Playbook containing only the remediations required to align your system with a specific baseline. This playbook is smaller because it does not cover requirements that are already satisfied.

Creating the playbook does not modify your system in any way, because you only prepare a file for later application.

Prerequisites

- The **scap-security-guide** package is installed.
- The **ansible-core** package is installed. See the Ansible Installation Guide for more information.
- The **rhc-worker-playbook** package is installed.
- You know the ID of the profile according to which you want to remediate your system. For details, see Viewing profiles for configuration compliance.

Procedure

1. Scan the system and save the results:

   ```
   # oscap xccdf eval --profile <profile_ID> --results <profile_results.xml>
   /usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml
   ```

   Replace: * ***<profile_ID>*** with the profile ID with which the system should comply, for example, **hipaa** * ***<profile_results.xml>*** with the path to the file where **oscap** saves the results

2. Find the value of the result ID in the file with the results:

   ```
   # oscap info <profile_results.xml>
   ```

3. Generate an Ansible Playbook based on the file generated in step 1:

   ```
   # oscap xccdf generate fix --fix-type ansible --result-id xccdf_org.open-
   scap_testresult_xccdf_org.ssgproject.content_profile_<profile_ID> --output
   <profile_remediations.yml> <profile_results.xml>
   ```

   Replace ***<profile_remediations.yml>*** with the path to the file where **oscap** saves rules that failed the scan.

4. Review the generated ***<profile_remediations.yml>*** file.

5. Remediate your system to align with a selected profile by using Ansible:

   ```
   # ANSIBLE_COLLECTIONS_PATH=/usr/share/rhc-worker-
   playbook/ansible/collections/ansible_collections/ ansible-playbook -i "localhost," -c local
   <profile_remediations.yml>`
   ```

   The **ANSIBLE_COLLECTIONS_PATH** environment variable is necessary for the command to run the playbook.

   > **WARNING**
   >
   > Remediations are supported on RHEL systems in the default configuration. Remediating a system that has been altered after installation might render the system nonfunctional or noncompliant with the required security profile. Red Hat does not provide any automated method to revert changes made by security-hardening remediations.
   >
   > Test the effects of the remediation before applying it on production systems.

Verification

- Evaluate the compliance of the system with the selected profile, and save the scan results to a file:

  ```
  # oscap xccdf eval --profile <profile_ID> --report <scan_report.html>
  /usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml
  ```

- Replace ***<scan_report.html>*** with the file name where **oscap** saves the scan results.

Additional resources

- Ansible Documentation

5.4. PERFORMING A HARDENED INSTALLATION OF RHEL WITH KICKSTART

To make your system compliant with a specific security profile, such as DISA STIG, CIS, or ANSSI, you can prepare a Kickstart file that defines the hardened configuration, customize it with a tailoring file, and run an automated installation of the hardened system.

Prerequisites

- The **openscap-scanner** is installed on your system.
- The **scap-security-guide** package is installed on your system and the package version corresponds to the version of RHEL that you want to install. For more information, see Supported versions of the SCAP Security Guide in RHEL . Using a different version can cause conflicts.

> **NOTE**
>
> If your system has the same version of RHEL as the version you want to install, you can install the **scap-security-guide** package directly.

Procedure

1. Find the ID of the security profile from the data stream file:

```
$ oscap info /usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml
Profiles:
...
  Title: Australian Cyber Security Centre (ACSC) Essential Eight
 Id: xccdf_org.ssgproject.content_profile_e8
  Title: Health Insurance Portability and Accountability Act (HIPAA)
 Id: xccdf_org.ssgproject.content_profile_hipaa
  Title: PCI-DSS v3.2.1 Control Baseline for Red Hat Enterprise Linux 10
 Id: xccdf_org.ssgproject.content_profile_pci-dss
...
```

2. Optional: If you want to customize your hardening with XCCDF Tailoring file you can use the **autotailor** command provided in the **openscap-utils** package. For more information, see Customizing a security profile with autotailor .

3. Generate the Kickstart file from the SCAP source data stream:

```
$ oscap xccdf generate fix --profile <profile_ID> --output <kickstart_file>.cfg --fix-type
kickstart /usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml
```

+ Replace ***<profile_ID>*** with the profile ID with which the system should comply, for example, **hipaa**.

+ If you are using a tailoring file, embed the tailoring file into the generated Kickstart file by using the **--tailoring-file tailoring.xml** option and your custom profile ID, for example:

+

```
$ *oscap xccdf generate fix --tailoring-file tailoring.xml --profile _<custom_profile_ID>_ --output
_<kickstart_file>_.cfg --fix-type kickstart ./ssg-rhel10-ds.xml*
```

1. Review and, if necessary, manually modify the generated ***<kickstart_file>.cfg*** to fit the needs of your deployment. Follow the instructions in the comments in the file.

   NOTE

   Some changes might affect the compliance of the systems installed by the Kickstart file. For example, some security policies require defined partitions or specific packages and services.

2. Use the Kickstart file for your installation. For the installation program to use the Kickstart, the Kickstart can be served through a web server, provided in PXE, or embedded into the ISO image. For detailed steps, see the Semi-automated installations: Making Kickstart files available to the RHEL installer chapter in the Automatically installing RHEL document.

3. After the installation finishes, the system reboots automatically. After the reboot, log in and review the installation SCAP report saved in the **/root** directory.

Verification

- Scan the system for compliance and save the report in a HTML file for review:
  - With the original profile:

    ```
    # oscap xccdf eval --report report.html --profile <profile_ID>
    /usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml
    ```

  - With the tailored profile:

    ```
    # oscap xccdf eval --report report.html --tailoring-file tailoring.xml --profile
    <custom_profile_ID> /usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml
    ```

5.5. CUSTOMIZING A SECURITY PROFILE WITH AUTOTAILOR

To align with internal security standards, you can customize a RHEL security profile by selecting or removing existing rules. While you can modify parameters such as password length, you cannot define entirely new rules during customization.

By using the **autotailor** utility, you create an XCCDF tailoring file that contains all of the modifications of the original profile. Then, when you are scanning, remediating, or installing a system in accordance with a SCAP profile, you pass this tailoring file to the **oscap** command-line utility.

Note that you cannot define new rules when customizing a profile.

Prerequisites

- The **openscap-utils** package is installed on your system.
- You know the ID of the profile within the baseline which you want to customize. To find the ID, see the Viewing profiles for configuration compliance section.

Procedure

1. Create a tailoring file for your profile by using the **autotailor** command, for example:

   ```
   $ autotailor \ --select=<rule_ID_1> \ --select=<rule_ID_2> \ --unselect=<rule_ID_3> \ --var-
   value=<value_ID_1>=<value_1> \ --var-value=<value_ID_2>=<value_2> \ --
   output=<tailoring.xml> \ --tailored-profile-id=<custom_profile_ID> \
   /usr/share/xml/scap/ssg/content/ssg-rhel10-ds.xml <profile_ID>
   ```

   Where:

   - ***<customization_options>*** are the modifications of the profile. Use one or more of the following options:

     --select=*<rule_ID>*

     Add an existing rule to the profile.

     --unselect=*<rule_ID>*

     Remove a rule from the profile.

     --var-value=*<value_ID>*=*<value>*

     Override a pre-set value. For example, to set **var_sshd_max_sessions** to **10**, use **--var-value=var_sshd_max_sessions=10**.

   - ***<tailoring.xml>*** is the file name where **autotailor** saves the tailoring.
   - ***<custom_profile_ID>*** is the profile ID within which the **autotailor** saves customizations, for example, **custom_cis**.
   - ***<profile_ID>*** is the profile ID with which the system should comply, for example, **cis**.

   > NOTE
   >
   > For all profile, rule, and variable XCCDF IDs, you can use either a full namespaced identifier or a shortened ID that **autotailor** automatically augments with the namespace prefix. For example, **cis** is equivalent to **xccdf_org.ssgproject.content_profile_cis**.
   >
   > You can override the default namespace **org.ssgproject.content** by using the **--id-namespace** option.

2. Optional: Create a tailoring file based on the customizations defined in the JSON Tailoring format:

   ```
   $ autotailor --output=<tailoring.xml> --json-tailoring=<json_tailoring.json>
   ```

   Replace ***<json_tailoring.json>*** with the file name with JSON Tailoring definitions.

NOTE

You can mix **--json-tailoring** with **--select**, **--unselect**, and **--var-value** command-line customizations. In that case, command-line customizations have priority over JSON Tailoring.

5.6. SCAP SECURITY GUIDE PROFILES SUPPORTED IN RHEL 10

Review the SCAP Security Guide profiles supported in RHEL 10, such as HIPAA, STIG, and CIS. To accommodate hardening components updated with new capabilities, use only the SCAP content provided in your specific minor release.

SCAP content changes to reflect these updates, but it is not always compatible with earlier versions.

NOTE

You can get the information relevant for the version of **scap-security-guide** RPM installed on your system by using the **oscap info** command. For more information, see Viewing profiles for configuration compliance .

Table 5.1. SCAP Security Guide profiles supported in RHEL 10.1

Profile name	Profile ID	Policy version
French National Agency for the Security of Information Systems (ANSSI) BP-028 Enhanced Level	**xccdf_org.ssgproject.content_profile_anssi_bp28_enhanced**	2.0
French National Agency for the Security of Information Systems (ANSSI) BP-028 High Level	**xccdf_org.ssgproject.content_profile_anssi_bp28_high**	2.0
French National Agency for the Security of Information Systems (ANSSI) BP-028 Intermediary Level	**xccdf_org.ssgproject.content_profile_anssi_bp28_intermediary**	2.0
French National Agency for the Security of Information Systems (ANSSI) BP-028 Minimal Level	**xccdf_org.ssgproject.content_profile_anssi_bp28_minimal**	2.0
CIS Red Hat Enterprise Linux 10 Benchmark for Level 2 - Server	**xccdf_org.ssgproject.content_profile_cis**	1.0.1
CIS Red Hat Enterprise Linux 10 Benchmark for Level 1 - Server	**xccdf_org.ssgproject.content_profile_cis_server_l1**	1.0.1
CIS Red Hat Enterprise Linux 10 Benchmark for Level 1 - Workstation	**xccdf_org.ssgproject.content_profile_cis_workstation_l1**	1.0.1

Profile name	Profile ID	Policy version
CIS Red Hat Enterprise Linux 10 Benchmark for Level 2 - Workstation	**xccdf_org.ssgproject.content_profile_cis_workstation_l2**	1.0.1
Australian Cyber Security Centre (ACSC) Essential Eight	**xccdf_org.ssgproject.content_profile_e8**	not versioned
Health Insurance Portability and Accountability Act (HIPAA)	**xccdf_org.ssgproject.content_profile_hipaa**	not versioned
Australian Cyber Security Centre (ACSC) ISM Official - Base	**xccdf_org.ssgproject.content_profile_ism_o**	not versioned
Australian Cyber Security Centre (ACSC) ISM Official - Secret	**xccdf_org.ssgproject.content_profile_ism_o_secret**	not versioned
Australian Cyber Security Centre (ACSC) ISM Official - Top Secret	**xccdf_org.ssgproject.content_profile_ism_o_top_secret**	not versioned
PCI-DSS v4.0.1 Control Baseline for Red Hat Enterprise Linux 10	**xccdf_org.ssgproject.content_profile_pci-dss**	4.0.1
The Defense Information Systems Agency Security Technical Implementation Guide (DISA STIG) for Red Hat Enterprise Linux 10	**xccdf_org.ssgproject.content_profile_stig**	vendor
The Defense Information Systems Agency Security Technical Implementation Guide (DISA STIG) with GUI for Red Hat Enterprise Linux 10	**xccdf_org.ssgproject.content_profile_stig_gui**	vendor

Table 5.2. SCAP Security Guide profiles supported in RHEL 10.0

Profile name	Profile ID	Policy version
French National Agency for the Security of Information Systems (ANSSI) BP-028 Enhanced Level	**xccdf_org.ssgproject.content_profile_anssi_bp28_enhanced**	2.0
French National Agency for the Security of Information Systems (ANSSI) BP-028 High Level	**xccdf_org.ssgproject.content_profile_anssi_bp28_high**	2.0

Profile name	Profile ID	Policy version
French National Agency for the Security of Information Systems (ANSSI) BP-028 Intermediary Level	**xccdf_org.ssgproject.content_profile_anssi_bp28_intermediary**	2.0
French National Agency for the Security of Information Systems (ANSSI) BP-028 Minimal Level	**xccdf_org.ssgproject.content_profile_anssi_bp28_minimal**	2.0
CIS Red Hat Enterprise Linux 10 Benchmark for Level 2 - Server	**xccdf_org.ssgproject.content_profile_cis**	1.0.1
CIS Red Hat Enterprise Linux 10 Benchmark for Level 1 - Server	**xccdf_org.ssgproject.content_profile_cis_server_l1**	1.0.1
CIS Red Hat Enterprise Linux 10 Benchmark for Level 1 - Workstation	**xccdf_org.ssgproject.content_profile_cis_workstation_l1**	1.0.1
CIS Red Hat Enterprise Linux 10 Benchmark for Level 2 - Workstation	**xccdf_org.ssgproject.content_profile_cis_workstation_l2**	1.0.1
Australian Cyber Security Centre (ACSC) Essential Eight	**xccdf_org.ssgproject.content_profile_e8**	not versioned
Health Insurance Portability and Accountability Act (HIPAA)	**xccdf_org.ssgproject.content_profile_hipaa**	not versioned
Australian Cyber Security Centre (ACSC) ISM Official - Base	**xccdf_org.ssgproject.content_profile_ism_o**	not versioned
Australian Cyber Security Centre (ACSC) ISM Official - Secret	**xccdf_org.ssgproject.content_profile_ism_o_secret**	not versioned
Australian Cyber Security Centre (ACSC) ISM Official - Top Secret	**xccdf_org.ssgproject.content_profile_ism_o_top_secret**	not versioned
PCI-DSS v4.0.1 Control Baseline for Red Hat Enterprise Linux 10	**xccdf_org.ssgproject.content_profile_pci-dss**	4.0.1
The Defense Information Systems Agency Security Technical Implementation Guide (DISA STIG) for Red Hat Enterprise Linux 10	**xccdf_org.ssgproject.content_profile_stig**	vendor

Profile name	Profile ID	Policy version
The Defense Information Systems Agency Security Technical Implementation Guide (DISA STIG) with GUI for Red Hat Enterprise Linux 10	**xccdf_org.ssgproject.content_profile_stig_gui**	vendor

5.7. ADDITIONAL RESOURCES

- Supported versions of the SCAP Security Guide in RHEL
- The OpenSCAP project page
- The SCAP Security Guide (SSG) project page
- Using OpenSCAP for security compliance and vulnerability scanning - aA hands-on lab
- Red Hat Security Demos: Creating Customized Security Policy Content to Automate Security Compliance
- Red Hat Security Demos: Defend Yourself with RHEL Security Technologies
- National Institute of Standards and Technology (NIST) SCAP page
- Managing security compliance in Red Hat Satellite

CHAPTER 6. ENSURING SYSTEM INTEGRITY WITH KEYLIME

With Keylime, you can continuously monitor the integrity of remote systems and verify the state of systems at boot. You can also send encrypted files to the monitored systems, and specify automated actions triggered whenever a monitored system fails the integrity test.

6.1. HOW KEYLIME WORKS

Keylime verifies system integrity by comparing measurements of boot components and runtime files against a known-good manifest. With Trusted Platform Module (TPM) technology, you can ensure that your remote infrastructure remains secure and untampered.

You can configure Keylime agents to perform one or more of the following actions:

Runtime integrity monitoring

Keylime runtime integrity monitoring continuously monitors the system on which the agent is deployed and measures the integrity of the files included in the allowlist and not included in the excludelist.

Measured boot

Keylime measured boot verifies the system state at boot.

Keylime’s concept of trust is based on the Trusted Platform Module (TPM) technology. A TPM is a hardware, firmware, or virtual component with integrated cryptographic keys. By polling TPM quotes and comparing the hashes of objects, Keylime provides initial and runtime monitoring of remote systems.

IMPORTANT

Keylime running in a virtual machine or using a virtual TPM depends upon the integrity of the underlying host. Ensure you trust the host environment before relying upon Keylime measurements in a virtual environment.

Keylime consists of three main components:

Verifier

Initially and continuously verifies the integrity of the systems that run the agent. You can deploy the verifier from a package, as a container, or by using the **keylime_server** RHEL system role.

Registrar

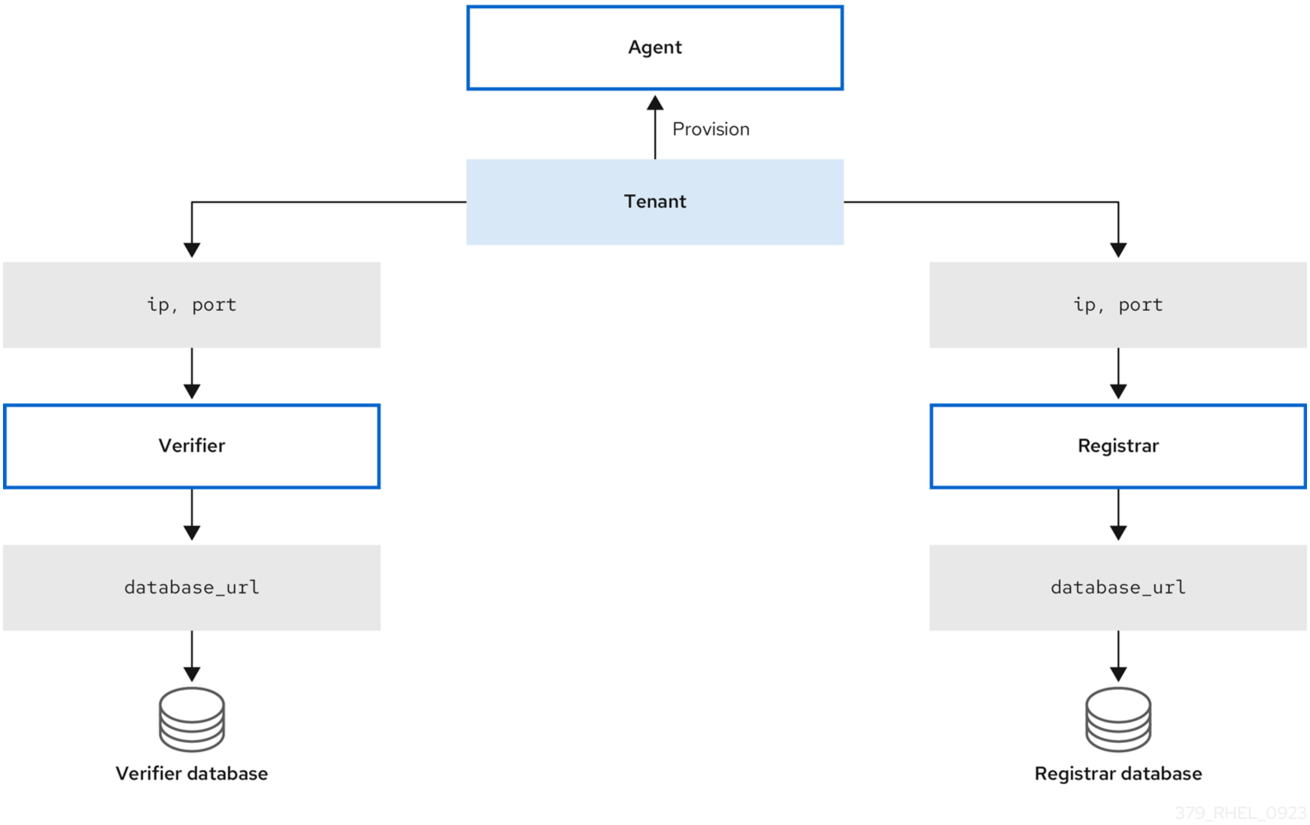
Contains a database of all agents and it hosts the public keys of the TPM vendors. You can deploy the registrar from a package, as a container, or by using the **keylime_server** RHEL system role.

Agent

Deployed to remote systems measured by the verifier.

In addition, Keylime uses the **keylime_tenant** utility for many functions, including provisioning the agents on the target systems.

Figure 6.1. Connections between Keylime components through configurations

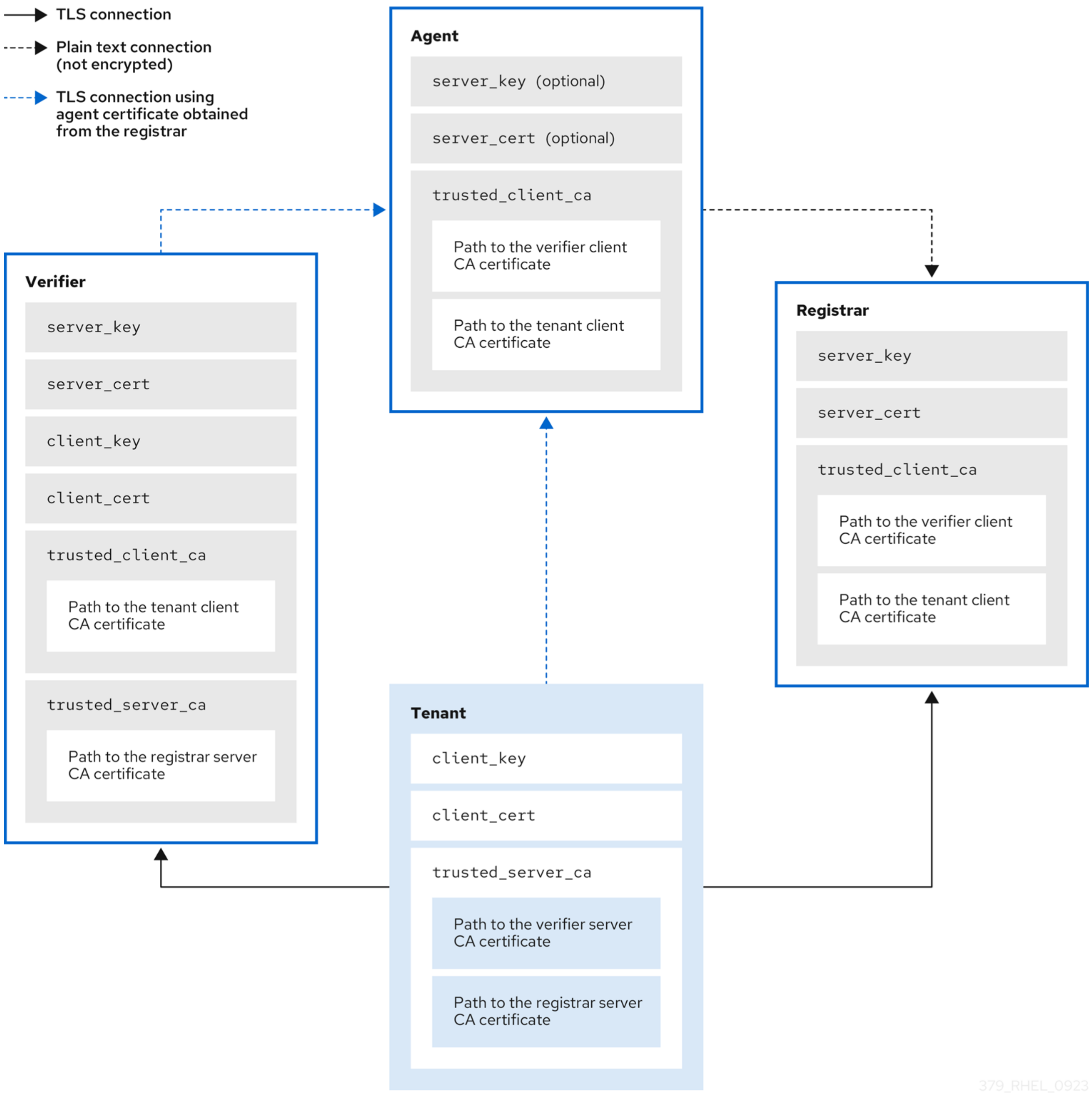


Keylime ensures the integrity of the monitored systems in a chain of trust by using keys and certificates exchanged between the components and the tenant. For a secure foundation of this chain, use a certificate authority (CA) that you can trust.

NOTE

If the agent receives no key and certificate, it generates a key and a self-signed certificate with no involvement from the CA.

Figure 6.2. Connections between Keylime components certificates and keys


TLS connection
Plain text connection (not encrypted)
TLS connection using agent certificate obtained from the registrar
Agent
server_key (optional)
server_cert (optional)
trusted_client_ca
Path to the verifier client CA certificate
Path to the tenant client CA certificate
Verifier
server_key
server_cert
client_key
client_cert
trusted_client_ca
Path to the tenant client CA certificate
trusted_server_ca
Path to the registrar server CA certificate
Registrar
server_key
server_cert
trusted_client_ca
Path to the verifier client CA certificate
Path to the tenant client CA certificate
Tenant
client_key
client_cert
trusted_server_ca
Path to the verifier server CA certificate
Path to the registrar server CA certificate
379_RHEL_0923

6.2. DEPLOYING KEYLIME VERIFIER FROM A PACKAGE

Deploy the Keylime verifier from a package to establish the core integrity monitoring component. The verifier acts as the verification authority for target systems – agents.

The verifier is the most critical component in Keylime. It performs initial and periodic checks of system integrity and supports bootstrapping a cryptographic key securely with the agent. The verifier uses mutual TLS encryption for its control interface.

IMPORTANT

To maintain the chain of trust, keep the system that runs the verifier secure and under your control.

You can install the verifier on a separate system or on the same system as the Keylime registrar, depending on your requirements. Running the verifier and registrar on separate systems provides better performance.

NOTE

To keep the configuration files organized within the drop-in directories, use file names with a two-digit number prefix, for example **/etc/keylime/verifier.conf.d/00-verifier-ip.conf**. The configuration processing reads the files inside the drop-in directory in lexicographic order and sets each option to the last value it reads.

Prerequisites

- You have **root** permissions and network connection to the system or systems on which you want to install Keylime components.
- You have valid keys and certificates from your certificate authority.
- Optional: You have access to the databases where Keylime saves data from the verifier. You can use any of the following database management systems:
  - SQLite (default)
  - PostgreSQL
  - MySQL
  - MariaDB

Procedure

1. Install the Keylime verifier:

   ```
   # dnf install keylime-verifier
   ```

2. Define the IP address and port of verifier by creating a new **.conf** file in the **/etc/keylime/verifier.conf.d/** directory, for example, **/etc/keylime/verifier.conf.d/00-verifier-ip.conf**, with the following content:

   ```
   [verifier]
   ip = <verifier_IP_address>
   ```

   - Replace ***<verifier_IP_address>*** with the verifier’s IP address. Alternatively, use **ip = *** or **ip = 0.0.0.0** to bind the verifier to all available IP addresses.
   - Optionally, you can also change the verifier’s port from the default value **8881** by using the **port** option.

3. Optional: Configure the verifier’s database for the list of agents. The default configuration uses an SQLite database in the verifier’s **/var/lib/keylime/cv_data.sqlite/** directory. You can define a different database by creating a new **.conf** file in the **/etc/keylime/verifier.conf.d/** directory, for example, **/etc/keylime/verifier.conf.d/00-db-url.conf**, with the following content:

```
[verifier]
database_url = <protocol>://<name>:<password>@<ip_address_or_hostname>/<properties>
```

Replace ***<protocol>://<name>:<password>@<ip_address_or_hostname>/<properties>*** with the URL of the database, for example, **postgresql://verifier:UQ?nRNY9g7GZzN7@198.51.100.1/verifierdb**.

Ensure that the credentials you use provide the permissions for Keylime to create the database structure.

4. Add certificates and keys to the verifier. You can either let Keylime generate them, or use existing keys and certificates:

   - With the default **tls_dir = generate** option, Keylime generates new certificates for the verifier, registrar, and tenant in the **/var/lib/keylime/cv_ca/** directory.

   - To load existing keys and certificates in the configuration, define their location in the verifier configuration. The certificates must be accessible by the **keylime** user, under which the Keylime services are running.
     Create a new **.conf** file in the **/etc/keylime/verifier.conf.d/** directory, for example, **/etc/keylime/verifier.conf.d/00-keys-and-certs.conf**, with the following content:

     ```
     [verifier]
     tls_dir = /var/lib/keylime/cv_ca
     server_key = </path/to/server_key>
     server_key_password = <passphrase1>
     server_cert = </path/to/server_cert>
     trusted_client_ca = ['</path/to/ca/cert1>', '</path/to/ca/cert2>']
     client_key = </path/to/client_key>
     client_key_password = <passphrase2>
     client_cert = </path/to/client_cert>
     trusted_server_ca = ['</path/to/ca/cert3>', '</path/to/ca/cert4>']
     ```

     NOTE

     Use absolute paths to define key and certificate locations. Alternatively, relative paths are resolved from the directory defined in the **tls_dir** option.

5. Open the port in firewall:

   ```
   # firewall-cmd --add-port 8881/tcp
   # firewall-cmd --runtime-to-permanent
   ```

   If you use a different port, replace **8881** with the port number defined in the **.conf** file.

6. Start the verifier service:

   ```
   # systemctl enable --now keylime_verifier
   ```

NOTE

In the default configuration, start the **keylime_verifier** before starting the **keylime_registrar** service because the verifier creates the CA and certificates for the other Keylime components. This order is not necessary when you use custom certificates.

Verification

- Check that the **keylime_verifier** service is active and running:

```
# systemctl status keylime_verifier
● keylime_verifier.service - The Keylime verifier
    Loaded: loaded (/usr/lib/systemd/system/keylime_verifier.service; disabled; vendor preset:
disabled)
    Active: active (running) since Wed 2022-11-09 10:10:08 EST; 1min 45s ago
```

Next steps

- Deploying the Keylime registrar from a package .

6.3. DEPLOYING KEYLIME VERIFIER AS A CONTAINER

Configure the Keylime verifier as a container instead of the RPM method, without any binaries or packages on the host. The container deployment provides better isolation, modularity, and reproducibility of Keylime components.

The Keylime verifier performs initial and periodic checks of system integrity and supports bootstrapping a cryptographic key securely with the agent.

After you start the container, the Keylime verifier is deployed with default configuration files. You can customize the configuration by using one or more of following methods:

- Mounting the host’s directories that contain the configuration files to the container.
- Modifying the environment variables directly on the container. Modifying the environment variables overrides the values from the configuration files.

Prerequisites

- The **podman** package and its dependencies are installed on the system.
- Optional: You have access to a database where Keylime saves data from the verifier. You can use any of the following database management systems:
    - SQLite (default)
    - PostgreSQL
    - MySQL
    - MariaDB
- You have valid keys and certificates from your certificate authority.

Procedure

1. Optional: Install the **keylime-verifier** package to access the configuration files. You can configure the container without this package, but it might be easier to modify the configuration files provided with the package.

   ```
   # dnf install keylime-verifier
   ```

2. Bind the verifier to all available IP addresses by creating a new **.conf** file in the **/etc/keylime/verifier.conf.d/** directory, for example, **/etc/keylime/verifier.conf.d/00-verifier-ip.conf**, with the following content:

   ```
   [verifier]
   ip = *
   ```

   - Optionally, you can also change the verifier's port from the default value **8881** by using the **port** option.

3. Optional: Configure the verifier's database for the list of agents. The default configuration uses an SQLite database in the verifier's **/var/lib/keylime/cv_data.sqlite/** directory. You can define a different database by creating a new **.conf** file in the **/etc/keylime/verifier.conf.d/** directory, for example, **/etc/keylime/verifier.conf.d/00-db-url.conf**, with the following content:

   ```
   [verifier]
   database_url = <protocol>://<name>:<password>@<ip_address_or_hostname>/<properties>
   ```

   Replace ***<protocol>://<name>:<password>@<ip_address_or_hostname>/<properties>*** with the URL of the database, for example, **postgresql://verifier:UQ?nRNY9g7GZzN7@198.51.100.1/verifierdb**.

   Ensure that the credentials you use have the permissions for Keylime to create the database structure.

4. Add certificates and keys to the verifier. You can either let Keylime generate them, or use existing keys and certificates:

   - With the default **tls_dir = generate** option, Keylime generates new certificates for the verifier, registrar, and tenant in the **/var/lib/keylime/cv_ca/** directory.
   - To load existing keys and certificates in the configuration, define their location in the verifier configuration. The certificates must be accessible by the **keylime** user, under which the Keylime processes are running.
     Create a new **.conf** file in the **/etc/keylime/verifier.conf.d/** directory, for example, **/etc/keylime/verifier.conf.d/00-keys-and-certs.conf**, with the following content:

     ```
     [verifier]
     tls_dir = /var/lib/keylime/cv_ca
     server_key = </path/to/server_key>
     server_cert = </path/to/server_cert>
     trusted_client_ca = ['</path/to/ca/cert1>', '</path/to/ca/cert2>']
     client_key = </path/to/client_key>
     client_cert = </path/to/client_cert>
     trusted_server_ca = ['</path/to/ca/cert3>', '</path/to/ca/cert4>']
     ```

> **NOTE**
>
> Use absolute paths to define key and certificate locations. Alternatively, relative paths are resolved from the directory defined in the **tls_dir** option.

5. Open the port in firewall:

```
# firewall-cmd --add-port 8881/tcp
# firewall-cmd --runtime-to-permanent
```

   If you use a different port, replace **8881** with the port number defined in the **.conf** file.

6. Run the container:

```
$ podman run --name keylime-verifier \
 -p 8881:8881 \
 -v /etc/keylime/verifier.conf.d:/etc/keylime/verifier.conf.d:Z \
 -v /var/lib/keylime/cv_ca:/var/lib/keylime/cv_ca:Z \
 -d \
 -e KEYLIME_VERIFIER_SERVER_KEY_PASSWORD=<passphrase1> \
 -e KEYLIME_VERIFIER_CLIENT_KEY_PASSWORD=<passphrase2> \
 registry.access.redhat.com/rhel9/keylime-verifier
```

   - The **-p** option opens the default port **8881** on the host and on the container.
   - The **-v** option creates a bind mount for the directory to the container.
     - With the **Z** option, Podman marks the content with a private unshared label. This means only the current container can use the private volume.
   - The **-d** option runs the container detached and in the background.
   - The option **-e KEYLIME_VERIFIER_SERVER_KEY_PASSWORD=*<passphrase1>*** defines the server key passphrase.
   - The option **-e KEYLIME_VERIFIER_CLIENT_KEY_PASSWORD=*<passphrase2>*** defines the client key passphrase.
   - You can override configuration options with environment variables by using the option **-e KEYLIME_VERIFIER_*<ENVIRONMENT_VARIABLE>*=*<value>***. To modify additional options, insert the **-e** option separately for each environment variable. For a complete list of environment variables and their default values, see Keylime environment variables.

Verification

- Check that the container is running:

```
$ podman ps -a
CONTAINER ID  IMAGE                                                COMMAND         CREATED
STATUS        PORTS                 NAMES
80b6b9dbf57c  registry.access.redhat.com/rhel9/keylime-verifier:latest  keylime_verifier  14
seconds ago  Up 14 seconds  0.0.0.0:8881->8881/tcp  keylime-verifier
```

Next steps

- Deploy the Keylime registrar as a container

Additional resources

- How Keylime works
- Deploying Keylime verifier from a package

6.4. DEPLOYING KEYLIME REGISTRAR FROM A PACKAGE

Deploy the Keylime registrar from a package to securely store and manage identity and key information about agents. Keeping the registrar secure helps maintain the chain of trust.

The registrar is the Keylime component that contains a database of all agents, and it hosts the public keys of the TPM vendors. After the registrar's HTTPS service accepts trusted platform module (TPM) public keys, it presents an interface to obtain these public keys for checking quotes.

IMPORTANT

To maintain the chain of trust, keep the system that runs the registrar secure and under your control.

You can install the registrar on a separate system or on the same system as the Keylime verifier, depending on your requirements. Running the verifier and registrar on separate systems provides better performance.

NOTE

To keep the configuration files organized within the drop-in directories, use file names with a two-digit number prefix, for example **/etc/keylime/registrar.conf.d/00-registrar-ip.conf**. The configuration processing reads the files inside the drop-in directory in lexicographic order and sets each option to the last value it reads.

Prerequisites

- You have network access to the systems where the Keylime verifier is installed and running. For more information, see

Section 6.2, “Deploying Keylime verifier from a package” .

- You have **root** permissions and network connection to the system or systems on which you want to install Keylime components.
- You have access to the database where Keylime saves data from the registrar. You can use any of the following database management systems:
  - SQLite (default)
  - PostgreSQL
  - MySQL
  - MariaDB
- You have valid keys and certificates from your certificate authority.

Procedure

1. Install the Keylime registrar:

   ```
   # dnf install keylime-registrar
   ```

2. Define the IP address and port of the registrar by creating a new **.conf** file in the **/etc/keylime/registrar.conf.d/** directory, for example, **/etc/keylime/registrar.conf.d/00-registrar-ip.conf**, with the following content:

   ```
   [registrar]
   ip = <registrar_IP_address>
   ```

   - Replace ***<registrar_IP_address>*** with the registrar's IP address. Alternatively, use **ip = *** or **ip = 0.0.0.0** to bind the registrar to all available IP addresses.

   - Optionally, change the port to which the Keylime agents connect by using the **port** option. The default value is **8890**.

   - Optionally, change the TLS port to which the Keylime verifier and tenant connect by using the **tls_port** option. The default value is **8891**.

3. Optional: Configure the registrar's database for the list of agents. The default configuration uses an SQLite database in the registrar's **/var/lib/keylime/reg_data.sqlite** directory. You can create a new **.conf** file in the **/etc/keylime/registrar.conf.d/** directory, for example, **/etc/keylime/registrar.conf.d/00-db-url.conf**, with the following content:

   ```
   [registrar]
   database_url = <protocol>://<name>:<password>@<ip_address_or_hostname>/<properties>
   ```

   Replace ***<protocol>://<name>:<password>@<ip_address_or_hostname>/<properties>*** with the URL of the database, for example, **postgresql://registrar:EKYYX-bqY2?#raXm@198.51.100.1/registrardb**.

   Ensure that the credentials you use have the permissions for Keylime to create the database structure.

4. Add certificates and keys to the registrar:

   - You can use the default configuration and load the keys and certificates to the **/var/lib/keylime/reg_ca/** directory.

   - Alternatively, you can define the location of the keys and certificates in the configuration. Create a new **.conf** file in the **/etc/keylime/registrar.conf.d/** directory, for example, **/etc/keylime/registrar.conf.d/00-keys-and-certs.conf**, with the following content:

     ```
     [registrar]
     tls_dir = /var/lib/keylime/reg_ca
     server_key = </path/to/server_key>
     server_key_password = <passphrase1>
     server_cert = </path/to/server_cert>
     trusted_client_ca = ['</path/to/ca/cert1>', '</path/to/ca/cert2>']
     ```

NOTE

Use absolute paths to define key and certificate locations. Alternatively, you can define a directory in the **tls_dir** option and use paths relative to that directory.

5. Open the ports in firewall:

   ```
   # firewall-cmd --add-port 8890/tcp --add-port 8891/tcp
   # firewall-cmd --runtime-to-permanent
   ```

   If you use a different port, replace **8890** or **8891** with the port number defined in the **.conf** file.

6. Start the **keylime_registrar** service:

   ```
   # systemctl enable --now keylime_registrar
   ```

NOTE

In the default configuration, start the **keylime_verifier** before starting the **keylime_registrar** service because the verifier creates the CA and certificates for the other Keylime components. This order is not necessary when you use custom certificates.

Verification

- Check that the **keylime_registrar** service is active and running:

  ```
  # systemctl status keylime_registrar
  ● keylime_registrar.service - The Keylime registrar service
      Loaded: loaded (/usr/lib/systemd/system/keylime_registrar.service; disabled; vendor
  preset: disabled)
      Active: active (running) since Wed 2022-11-09 10:10:17 EST; 1min 42s ago
  ...
  ```

Next steps

- Deploying Keylime tenant from a package

6.5. DEPLOYING KEYLIME REGISTRAR AS A CONTAINER

Configure the Keylime registrar as a container instead of the RPM method, without any binaries or packages on the host. The container deployment provides better isolation, modularity, and reproducibility of Keylime components.

The registrar is the Keylime component that contains a database of all agents, and it hosts the public keys of the trusted platform module (TPM) vendors. After the registrar’s HTTPS service accepts TPM public keys, it presents an interface to obtain these public keys for checking quotes.

After you start the container, the Keylime registrar is deployed with default configuration files. You can customize the configuration by using one or more of following methods:

- Mounting the host’s directories that contain the configuration files to the container. This is available in all versions of RHEL 9.
- Modifying the environment variables directly on the container. This is available in RHEL 9.3 and later versions. Modifying the environment variables overrides the values from the configuration files.

Prerequisites

- The **podman** package and its dependencies are installed on the system.
- Optional: You have access to a database where Keylime saves data from the registrar. You can use any of the following database management systems:
  - SQLite (default)
  - PostgreSQL
  - MySQL
  - MariaDB
- You have valid keys and certificates from your certificate authority.

Procedure

1. Optional: Install the **keylime-registrar** package to access the configuration files. You can configure the container without this package, but it might be easier to modify the configuration files provided with the package.

   ```
   # dnf install keylime-registrar
   ```

2. Bind the registrar to all available IP addresses by creating a new **.conf** file in the **/etc/keylime/registrar.conf.d/** directory, for example, **/etc/keylime/registrar.conf.d/00-registrar-ip.conf**, with the following content:

   ```
   [registrar]
   ip = *
   ```

   - Optionally, change the port to which the Keylime agents connect by using the **port** option. The default value is **8890**.
   - Optionally, change the TLS port to which the Keylime tenant connects by using the **tls_port** option. The default value is **8891**.

3. Optional: Configure the registrar’s database for the list of agents. The default configuration uses an SQLite database in the registrar’s **/var/lib/keylime/reg_data.sqlite** directory. You can create a new **.conf** file in the **/etc/keylime/registrar.conf.d/** directory, for example, **/etc/keylime/registrar.conf.d/00-db-url.conf**, with the following content:

   ```
   [registrar]
   database_url = <protocol>://<name>:<password>@<ip_address_or_hostname>/<properties>
   ```

Replace ***<protocol>://<name>:<password>@<ip_address_or_hostname>/<properties>*** with the URL of the database, for example, **postgresql://registrar:EKYYX-bqY2?#raXm@198.51.100.1/registrardb**.

Ensure that the credentials you use have the permissions for Keylime to create the database structure.

4. Add certificates and keys to the registrar:

    - You can use the default configuration and load the keys and certificates to the **/var/lib/keylime/reg_ca/** directory.

    - Alternatively, you can define the location of the keys and certificates in the configuration. Create a new **.conf** file in the **/etc/keylime/registrar.conf.d/** directory, for example, **/etc/keylime/registrar.conf.d/00-keys-and-certs.conf**, with the following content:

      ```
      [registrar]
      tls_dir = /var/lib/keylime/reg_ca
      server_key = </path/to/server_key>
      server_cert = </path/to/server_cert>
      trusted_client_ca = ['</path/to/ca/cert1>', '</path/to/ca/cert2>']
      ```

      NOTE

      Use absolute paths to define key and certificate locations. Alternatively, you can define a directory in the **tls_dir** option and use paths relative to that directory.

5. Open the ports in firewall:

    ```
    # firewall-cmd --add-port 8890/tcp --add-port 8891/tcp
    # firewall-cmd --runtime-to-permanent
    ```

    If you use a different port, replace **8890** or **8891** with the port number defined in the **.conf** file.

6. Run the container:

    ```
    $ podman run --name keylime-registrar \
      -p 8890:8890 \
      -p 8891:8891 \
      -v /etc/keylime/registrar.conf.d:/etc/keylime/registrar.conf.d:Z \
      -v /var/lib/keylime/reg_ca:/var/lib/keylime/reg_ca:Z \
      -d \
      -e KEYLIME_REGISTRAR_SERVER_KEY_PASSWORD=<passphrase1> \
      registry.access.redhat.com/rhel9/keylime-registrar
    ```

    - The **-p** option opens the default ports **8890** and **8881** on the host and on the container.

    - The **-v** option creates a bind mount for the directory to the container.

        - With the **Z** option, Podman marks the content with a private unshared label. This means only the current container can use the private volume.

    - The **-d** option runs the container detached and in the background.

- The option **-e KEYLIME_VERIFIER_SERVER_KEY_PASSWORD=*<passphrase1>*** defines the server key passphrase.

- You can override configuration options with environment variables by using the option **-e KEYLIME_REGISTRAR__*<ENVIRONMENT_VARIABLE>*=*<value>***. To modify additional options, insert the **-e** option separately for each environment variable. For a complete list of environment variables and their default values, see Section 6.12, "Keylime environment variables".

Verification

- Check that the container is running:

```
$ podman ps -a
CONTAINER ID  IMAGE                              COMMAND           CREATED            STATUS
PORTS                                    NAMES
07d4b4bff1b6  localhost/keylime-registrar:latest  keylime_registrar  12 seconds ago     Up 12
seconds      0.0.0.0:8881->8881/tcp, 0.0.0.0:8891->8891/tcp  keylime-registrar
```

Next steps

- Deploying Keylime tenant from a package .

Additional resources

- How Keylime works

- Deploying Keylime registrar from a package

6.6. DEPLOYING A KEYLIME SERVER BY USING RHEL SYSTEM ROLES

You can set up the verifier and registrar, which are the Keylime server components, by using the **keylime_server** RHEL system role. The **keylime_server** role installs and configures both the verifier and registrar components together on each node.

For more information about Keylime, see How Keylime works.

Prerequisites

- You have prepared the control node and the managed nodes .

- You are logged in to the control node as a user who can run playbooks on the managed nodes.

- The account you use to connect to the managed nodes has **sudo** permissions for these nodes.

Procedure

1. Create a playbook that defines the required role:

   a. Create a new YAML file and open it in a text editor, for example:

   ```
   # vi keylime-playbook.yml
   ```

   b. Insert the following content:

```
---
- name: Manage keylime servers
  hosts: all
  vars:
    keylime_server_verifier_ip: "{{ ansible_host }}"
    keylime_server_registrar_ip: "{{ ansible_host }}"
    keylime_server_verifier_tls_dir: <ver_tls_directory>
    keylime_server_verifier_server_cert: <ver_server_certfile>
    keylime_server_verifier_server_key: <ver_server_key>
    keylime_server_verifier_server_key_passphrase: <ver_server_key_passphrase>
    keylime_server_verifier_trusted_client_ca: <ver_trusted_client_ca_list>
    keylime_server_verifier_client_cert: <ver_client_certfile>
    keylime_server_verifier_client_key: <ver_client_key>
    keylime_server_verifier_client_key_passphrase: <ver_client_key_passphrase>
    keylime_server_verifier_trusted_server_ca: <ver_trusted_server_ca_list>
    keylime_server_registrar_tls_dir: <reg_tls_directory>
    keylime_server_registrar_server_cert: <reg_server_certfile>
    keylime_server_registrar_server_key: <reg_server_key>
    keylime_server_registrar_server_key_passphrase: <reg_server_key_passphrase>
    keylime_server_registrar_trusted_client_ca: <reg_trusted_client_ca_list>
  roles:
    - rhel-system-roles.keylime_server
```

You can find out more about the variables in Variables for the keylime_server RHEL system role.

2. Run the playbook:

```
$ ansible-playbook <keylime-playbook.yml>
```

Verification

1. Check that the **keylime_verifier** service is active and running on the managed host:

```
# systemctl status keylime_verifier
● keylime_verifier.service - The Keylime verifier
     Loaded: loaded (/usr/lib/systemd/system/keylime_verifier.service; disabled; vendor preset: disabled)
     Active: active (running) since Wed 2022-11-09 10:10:08 EST; 1min 45s ago
```

2. Check that the **keylime_registrar** service is active and running:

```
# systemctl status keylime_registrar
● keylime_registrar.service - The Keylime registrar service
     Loaded: loaded (/usr/lib/systemd/system/keylime_registrar.service; disabled; vendor preset: disabled)
     Active: active (running) since Wed 2022-11-09 10:10:17 EST; 1min 42s ago
...
```

Next steps

Install the Keylime tenant

6.7. VARIABLES FOR THE KEYLIME_SERVER RHEL SYSTEM ROLE

When setting up a Keylime server by using the **keylime_server** RHEL system role, you can customize the following variables for registrar and verifier.

List of **keylime_server** RHEL system role variables for configuring the Keylime verifier:

keylime_server_verifier_ip

Defines the IP address of the verifier.

keylime_server_verifier_tls_dir

Specifies the directory where the keys and certificates are stored. If set to default, the verifier uses the **/var/lib/keylime/cv_ca** directory.

keylime_server_verifier_server_key_passphrase

Specifies a passphrase to decrypt the server private key. If the value is empty, the private key is not encrypted.

keylime_server_verifier_server_cert: Specifies the Keylime verifier server certificate file.

keylime_server_verifier_trusted_client_ca

Defines the list of trusted client CA certificates. You must store the files in the directory set in the **keylime_server_verifier_tls_dir** option.

keylime_server_verifier_client_key

Defines the file containing the Keylime verifier private client key.

keylime_server_verifier_client_key_passphrase

Defines the passphrase to decrypt the client private key file. If the value is empty, the private key is not encrypted.

keylime_server_verifier_client_cert

Defines the Keylime verifier client certificate file.

keylime_server_verifier_trusted_server_ca

Defines the list of trusted server CA certificates. You must store the files in the directory set in the **keylime_server_verifier_tls_dir** option.

List of registrar variables for setting up keylime_server RHEL system role:

keylime_server_registrar_ip

Defines the IP address of the registrar.

keylime_server_registrar_tls_dir

Specifies the directory where you store the keys and certificates for the registrar. If you set it to default, the registrar uses the **/var/lib/keylime/reg_ca** directory.

keylime_server_registrar_server_key

Defines the Keylime registrar private server key file.

keylime_server_registrar_server_key_passphrase

Specifies the passphrase to decrypt the server private key of the registrar. If the value is empty, the private key is not encrypted.

keylime_server_registrar_server_cert

Specifies the Keylime registrar server certificate file.

keylime_server_registrar_trusted_client_ca

Defines the list of trusted client CA certificates. You must store the files in the directory set in the **keylime_server_registrar_tls_dir** option.

6.8. DEPLOYING KEYLIME TENANT FROM A PACKAGE

Keylime uses the **keylime_tenant** utility for many functions, including provisioning the agents on the target systems. You can install **keylime_tenant** on any system, including the systems that run other Keylime components, or on a separate system, depending on your requirements.

For additional advanced options for the **keylime_tenant** utility, enter the **keylime_tenant -h** command.

Prerequisites

- You have **root** permissions and network connection to the system or systems on which you want to install Keylime components.
- You have network access to the systems where the other Keylime components are configured:

  Verifier

  For more information, see Section 6.2, "Deploying Keylime verifier from a package" .

  Registrar

  For more information, see Section 6.4, "Deploying Keylime registrar from a package" .

Procedure

1. Install the Keylime tenant:

   ```
   # dnf install keylime-tenant
   ```

2. Define the tenant's connection to the Keylime verifier by editing the **/etc/keylime/tenant.conf.d/00-verifier-ip.conf** file:

   ```
   [tenant]
   verifier_ip = <verifier_ip>
   ```

   - Replace ***<verifier_ip>*** with the IP address to the verifier's system.
   - If the verifier uses a different port than the default value **8881**, add the **verifier_port = *<verifier_port>*** setting.

3. Define the tenant's connection to the Keylime registrar by editing the **/etc/keylime/tenant.conf.d/00-registrar-ip.conf** file:

   ```
   [tenant]
   registrar_ip = <registrar_ip>
   ```

   - Replace ***<registrar_ip>*** with the IP address to the registrar's system.
   - If the registrar uses a different port than the default value **8891**, add the **registrar_port = *<registrar_port>*** setting.

4. Add certificates and keys to the tenant:

   a. You can use the default configuration and load the keys and certificates to the

a. You can use the default configuration and load the keys and certificates to the **/var/lib/keylime/cv_ca** directory.

b. Alternatively, you can define the location of the keys and certificates in the configuration. Create a new **.conf** file in the **/etc/keylime/tenant.conf.d/** directory, for example, **/etc/keylime/tenant.conf.d/00-keys-and-certs.conf**, with the following content:

```
[tenant]
tls_dir = /var/lib/keylime/cv_ca
client_key = tenant-key.pem
client_key_password = <passphrase1>
client_cert = tenant-cert.pem
trusted_server_ca = ['</path/to/ca/cert>']
```

The **trusted_server_ca** parameter accepts paths to the verifier and registrar server CA certificate. You can provide multiple comma-separated paths, for example if the verifier and registrar use different CAs.

NOTE

Use absolute paths to define key and certificate locations. Alternatively, you can define a directory in the **tls_dir** option and use paths relative to that directory.

5. Optional: If the trusted platform module (TPM) endorsement key (EK) cannot be verified by using certificates in the **/var/lib/keylime/tpm_cert_store** directory, add the certificate to that directory. This can occur particularly when using virtual machines with emulated TPMs.

Verification

1. Check the status of the verifier:

```
# keylime_tenant -c cvstatus
Reading configuration from ['/etc/keylime/logging.conf']
2022-10-14 12:56:08.155 - keylime.tpm - INFO - TPM2-TOOLS Version: 5.2
Reading configuration from ['/etc/keylime/tenant.conf']
2022-10-14 12:56:08.157 - keylime.tenant - INFO - Setting up client TLS...
2022-10-14 12:56:08.158 - keylime.tenant - INFO - Using default client_cert option for tenant
2022-10-14 12:56:08.158 - keylime.tenant - INFO - Using default client_key option for tenant
2022-10-14 12:56:08.178 - keylime.tenant - INFO - TLS is enabled.
2022-10-14 12:56:08.178 - keylime.tenant - WARNING - Using default UUID d432fbb3-d2f1-
4a97-9ef7-75bd81c00000
2022-10-14 12:56:08.221 - keylime.tenant - INFO - Verifier at 127.0.0.1 with Port 8881 does
not have agent d432fbb3-d2f1-4a97-9ef7-75bd81c00000.
```

If correctly set up, and if no agent is configured, the verifier responds that it does not recognize the default agent UUID.

2. Check the status of the registrar:

```
# keylime_tenant -c regstatus
Reading configuration from ['/etc/keylime/logging.conf']
2022-10-14 12:56:02.114 - keylime.tpm - INFO - TPM2-TOOLS Version: 5.2
Reading configuration from ['/etc/keylime/tenant.conf']
2022-10-14 12:56:02.116 - keylime.tenant - INFO - Setting up client TLS...
```

```
2022-10-14 12:56:02.116 - keylime.tenant - INFO - Using default client_cert option for tenant
2022-10-14 12:56:02.116 - keylime.tenant - INFO - Using default client_key option for tenant
2022-10-14 12:56:02.137 - keylime.tenant - INFO - TLS is enabled.
2022-10-14 12:56:02.137 - keylime.tenant - WARNING - Using default UUID d432fbb3-d2f1-
4a97-9ef7-75bd81c00000
2022-10-14 12:56:02.171 - keylime.registrar_client - CRITICAL - Error: could not get agent
d432fbb3-d2f1-4a97-9ef7-75bd81c00000 data from Registrar Server: 404
2022-10-14 12:56:02.172 - keylime.registrar_client - CRITICAL - Response code 404: agent
d432fbb3-d2f1-4a97-9ef7-75bd81c00000 not found
2022-10-14 12:56:02.172 - keylime.tenant - INFO - Agent d432fbb3-d2f1-4a97-9ef7-
75bd81c00000 does not exist on the registrar. Please register the agent with the registrar.
2022-10-14 12:56:02.172 - keylime.tenant - INFO - {"code": 404, "status": "Agent d432fbb3-
d2f1-4a97-9ef7-75bd81c00000 does not exist on registrar 127.0.0.1 port 8891.", "results": {}}
```

If correctly set up, and if no agent is configured, the registrar responds that it does not recognize the default agent UUID.

Next steps

- Deploying Keylime agent from a package

6.9. DEPLOYING KEYLIME AGENT FROM A PACKAGE

Deploy the Keylime agent from a package onto the target system that requires continuous integrity monitoring. The agent is the monitored endpoint that reports its security state.

By default, the Keylime agent stores all its data in the **/var/lib/keylime/** directory of the monitored system.

> **NOTE**
>
> To keep the configuration files organized within the drop-in directories, use file names with a two-digit number prefix, for example **/etc/keylime/agent.conf.d/00-registrar-ip.conf**. The configuration processing reads the files inside the drop-in directory in lexicographic order and sets each option to the last value it reads.

Prerequisites

- You have **root** permissions to the monitored system.
- The monitored system has a Trusted Platform Module (TPM). To verify, enter the **tpm2_pcrread** command. If the output returns several hashes, a TPM is available.
- You have network access to the systems where the other Keylime components are configured:

  Verifier

  For more information, see Deploying Keylime verifier from a package .

  Registrar

  For more information, see Deploying Keylime registrar from a package .

  Tenant

  For more information, see Deploying Keylime tenant from a package .

- Integrity measurement architecture (IMA) is enabled on the monitored system. For more information, see Enabling integrity measurement architecture and extended verification module .

Procedure

1. Install the Keylime agent:

   ```
   # dnf install keylime-agent
   ```

   This command installs the **keylime-agent-rust** package.

2. Define the agent's IP address and port in the configuration files. Create a new **.conf** file in the **/etc/keylime/agent.conf.d/** directory, for example, **/etc/keylime/agent.conf.d/00-agent-ip.conf**, with the following content:

   ```
   [agent]
   ip = '<agent_ip>'
   ```

   NOTE

   The Keylime agent configuration uses the TOML format, which is different from the INI format used for configuration of the other components. Therefore, enter values in valid TOML syntax, for example, paths in single quotation marks and arrays of multiple paths in square brackets.

   - Replace ***<agent_IP_address>*** with the agent's IP address. Alternatively, use **ip = '*'** or **ip = '0.0.0.0'** to bind the agent to all available IP addresses.
   - Optionally, you can also change the agent's port from the default value **9002** by using the **port = '*<agent_port>*'** option.

3. Define the registrar's IP address and port in the configuration files. Create a new **.conf** file in the **/etc/keylime/agent.conf.d/** directory, for example, **/etc/keylime/agent.conf.d/00-registrar-ip.conf**, with the following content:

   ```
   [agent]
   registrar_ip = '<registrar_IP_address>'
   ```

   - Replace ***<registrar_IP_address>*** with the registrar's IP address.
   - Optionally, you can also change the registrar's port from the default value **8890** by using the **registrar_port = '*<registrar_port>*'** option.

4. Optional: Define the agent's universally unique identifier (UUID). If it is not defined, the default UUID is used. Create a new **.conf** file in the **/etc/keylime/agent.conf.d/** directory, for example, **/etc/keylime/agent.conf.d/00-agent-uuid.conf**, with the following content:

   ```
   [agent]
   uuid = '<agent_UUID>'
   ```

   - Replace ***<agent_UUID>*** with the agent's UUID, for example **d432fbb3-d2f1-4a97-9ef7-abcdef012345**. You can use the **uuidgen** utility to generate a UUID.

5. Optional: Load existing keys and certificates for the agent. If the agent receives no **server_key** and **server_cert**, it generates its own key and a self-signed certificate.
   Define the location of the keys and certificates in the configuration. Create a new **.conf** file in the **/etc/keylime/agent.conf.d/** directory, for example, **/etc/keylime/agent.conf.d/00-keys-and-certs.conf**, with the following content:

   ```
   [agent]
   server_key = '</path/to/server_key>'
   server_key_password = '<passphrase1>'
   server_cert = '</path/to/server_cert>'
   trusted_client_ca = '[</path/to/ca/cert3>, </path/to/ca/cert4>]'
   ```

   NOTE

   Use absolute paths to define key and certificate locations. The Keylime agent does not accept relative paths.

6. Open the port in firewall:

   ```
   # firewall-cmd --add-port 9002/tcp
   # firewall-cmd --runtime-to-permanent
   ```

   If you use a different port, replace **9002** with the port number defined in the **.conf** file.

7. Enable and start the **keylime_agent** service:

   ```
   # systemctl enable --now keylime_agent
   ```

8. Optional: From the system where the Keylime tenant is configured, verify that the agent is correctly configured and can connect to the registrar.

   ```
   # keylime_tenant -c regstatus --uuid <agent_uuid>
   Reading configuration from ['/etc/keylime/logging.conf']
   ...
   ==\n-----END CERTIFICATE-----\n", "ip": "127.0.0.1", "port": 9002, "regcount": 1,
   "operational_state": "Registered"}}}
   ```

   - Replace ***<agent_uuid>*** with the agent's UUID.
     If the registrar and agent are correctly configured, the output displays the agent's IP address and port, followed by **"operational_state": "Registered"**.

9. Create a new IMA policy by entering the following content into the **/etc/ima/ima-policy** file:

   ```
   # PROC_SUPER_MAGIC = 0x9fa0
   dont_measure fsmagic=0x9fa0
   # SYSFS_MAGIC = 0x62656572
   dont_measure fsmagic=0x62656572
   # DEBUGFS_MAGIC = 0x64626720
   dont_measure fsmagic=0x64626720
   # TMPFS_MAGIC = 0x01021994
   dont_measure fsmagic=0x1021994
   # RAMFS_MAGIC
   dont_measure fsmagic=0x858458f6
   ```

```
# DEVPTS_SUPER_MAGIC=0x1cd1
dont_measure fsmagic=0x1cd1
# BINFMTFS_MAGIC=0x42494e4d
dont_measure fsmagic=0x42494e4d
# SECURITYFS_MAGIC=0x73636673
dont_measure fsmagic=0x73636673
# SELINUX_MAGIC=0xf97cff8c
dont_measure fsmagic=0xf97cff8c
# SMACK_MAGIC=0x43415d53
dont_measure fsmagic=0x43415d53
# NSFS_MAGIC=0x6e736673
dont_measure fsmagic=0x6e736673
# EFIVARFS_MAGIC
dont_measure fsmagic=0xde5e81e4
# CGROUP_SUPER_MAGIC=0x27e0eb
dont_measure fsmagic=0x27e0eb
# CGROUP2_SUPER_MAGIC=0x63677270
dont_measure fsmagic=0x63677270
# OVERLAYFS_MAGIC
# when containers are used we almost always want to ignore them
dont_measure fsmagic=0x794c7630
# MEASUREMENTS
measure func=BPRM_CHECK
measure func=FILE_MMAP mask=MAY_EXEC
measure func=MODULE_CHECK uid=0
```

This policy targets runtime monitoring of executed applications. You can adjust this policy according to your scenario. You can find the MAGIC constants in the **statfs(2)** man page.

10. Update kernel parameters:

```
# grubby --update-kernel DEFAULT --args 'ima_appraise=log ima_canonical_fmt
ima_policy=tcb ima_template=ima-ng'
```

11. Reboot the system to apply the new IMA policy.

Verification

1. Verify that the agent is running:

```
# systemctl status keylime_agent
● keylime_agent.service - The Keylime compute agent
   Loaded: loaded (/usr/lib/systemd/system/keylime_agent.service; enabled; preset:
disabled)
   Active: active (running) since ...
```

Next steps

After the agent is configured on all systems you want to monitor, you can deploy Keylime to perform one or both of the following functions:

- Configuring Keylime for runtime monitoring
- Configuring Keylime for measured boot attestation

Additional resources

- Integrity Measurement Architecture (IMA) Wiki

6.10. CONFIGURING KEYLIME FOR RUNTIME MONITORING

To verify that the state of monitored systems is correct, the Keylime agent must be running on the monitored systems.

> **IMPORTANT**
>
> Because Keylime runtime monitoring uses integrity measurement architecture (IMA) to measure large numbers of files, it might have a significant impact on the performance of your system.

When provisioning the agent, you can also define a file that Keylime sends to the monitored system. Keylime encrypts the file sent to the agent, and decrypts it only if the agent's system complies with the TPM policy and with the IMA allowlist.

You can make Keylime ignore changes of specific files or within specific directories by configuring a Keylime excludelist. The excluded files are still measured by IMA.

The allowlist and excludelist are combined into the Keylime runtime policy.

Prerequisites

- You have network access to the systems where the Keylime components are configured:

  Verifier

  For more information, see Deploying Keylime verifier from a package .

  Registrar

  For more information, see Deploying Keylime registrar from a package .

  Tenant

  For more information, see Deploying Keylime tenant from a package .

  Agent

  For more information, see Deploying Keylime agent from a package .

Procedure

1. On the monitored system where the Keylime agent is configured and running, install the **python3-keylime** package, which contains the **keylime-policy** tool:

   ```
   # dnf -y install python3-keylime
   ```

2. Create a runtime policy from the current state of the agent system:

   ```
   # keylime-policy create runtime --ima-measurement --rootfs '/' --ramdisk-dir '/boot/' --output
   <policy.json>
   ```

   In this command,

   - Replace ***<policy.json>*** with the file name of the runtime policy.

- The following directories are automatically excluded from measurement:
  - **/sys**
  - **/run**
  - **/proc**
  - **/lost+found**
  - **/dev**
  - **/media**
  - **/snap**
  - **/mnt**
  - **/var**
  - **/tmp**
- Optionally, you can exclude additional specific paths from measurement by adding a **--excludelist *<excludelist.txt>*** option. The excludelist accepts Python regular expressions with one regular expression per line. See Regular expression operations at docs.python.org for the complete list of special characters.

3. Copy the generated runtime policy to the system where the **keylime_tenant** utility is configured, for example:

```
# scp <policy.json> root@<tenant.ip>:/root/<policy.json>
```

4. On the system where the Keylime tenant is configured, provision the agent by using the **keylime_tenant** utility:

```
# keylime_tenant --command add --targethost <agent_ip> --uuid <agent_uuid> --runtime-
policy <policy.json> --cert default
```

- Replace ***<agent_ip>*** with the agent's IP address.
- Replace ***<agent_uuid>*** with the agent's UUID.
- Replace ***<policy.json>*** with the path to the Keylime runtime policy file.
- With the **--cert** option, the tenant generates and signs a certificate for the agent by using the CA certificates and keys located in the specified directory, or the default **/var/lib/keylime/ca/** directory. If the directory contains no CA certificates and keys, the tenant will generate them automatically according to the configuration in the **/etc/keylime/ca.conf** file and save them to the specified directory. The tenant then sends these keys and certificates to the agent.
  When generating CA certificates or signing agent certificates, you might be prompted for the password to access the CA private key: **Please enter the password to decrypt your keystore:**.

NOTE

Keylime encrypts the file sent to the agent, and decrypts it only if the agent's system complies with the TPM policy and the IMA allowlist. By default, Keylime decompresses sent **.zip** files.

As an example, with the following command, **keylime_tenant** provisions a new Keylime agent at **127.0.0.1** with UUID **d432fbb3-d2f1-4a97-9ef7-75bd81c00000** and loads a runtime policy **policy.json**. It also generates a certificate in the default directory and sends the certificate file to the agent. Keylime decrypts the file only if the TPM policy configured in **/etc/keylime/verifier.conf** is satisfied:

```
# keylime_tenant --command add --targethost 127.0.0.1 --uuid d432fbb3-d2f1-4a97-9ef7-
75bd81c00000 --runtime-policy policy.json --cert default
```

NOTE

You can stop Keylime from monitoring a node by using the **# keylime_tenant --command delete --uuid *<agent_uuid>*** command.

You can modify the configuration of an already registered agent by using the **keylime_tenant --command update** command.

Verification

1. Optional: Reboot the monitored system to verify that the settings are persistent.

2. Verify a successful attestation of the agent:

```
# keylime_tenant --command cvstatus --uuid <agent_uuid>
...
{"<agent_uuid>": {"operational_state": "Get Quote"..."attestation_count": 5
...
```

Replace ***<agent_uuid>*** with the agent's UUID.

If the value of **operational_state** is **Get Quote** and **attestation_count** is nonzero, the attestation of this agent is successful.

If the value of **operational_state** is **Invalid Quote** or **Failed** attestation fails, the command displays output similar to the following:

```
{"<agent_uuid>": {"operational_state": "Invalid Quote", ... "ima.validation.ima-
ng.not_in_allowlist", "attestation_count": 5, "last_received_quote": 1684150329,
"last_successful_attestation": 1684150327}}
```

3. If the attestation fails, display more details in the verifier log:

```
# journalctl --unit keylime_verifier
keylime.tpm - INFO - Checking IMA measurement list...
keylime.ima - WARNING - File not found in allowlist: /root/bad-script.sh
keylime.ima - ERROR - IMA ERRORS: template-hash 0 fnf 1 hash 0 good 781
keylime.cloudverifier - WARNING - agent D432FBB3-D2F1-4A97-9EF7-75BD81C00000
failed, stopping polling
```

Additional resources

- Enhancing security with the kernel integrity subsystem

6.11. CONFIGURING KEYLIME FOR MEASURED BOOT ATTESTATION

Configure Keylime to perform measured boot attestation by using the Integrity Measurement Architecture (IMA). This verifies the integrity of system components early in the boot process.

Prerequisites

- You have network access to the systems where the Keylime components are configured:

  Verifier

  For more information, see Deploying Keylime verifier from a package .

  Registrar

  For more information, see Deploying Keylime registrar from a package .

  Tenant

  For more information, see Deploying Keylime tenant from a package .

  Agent

  For more information, see Deploying Keylime agent from a package .

- Unified Extensible Firmware Interface (UEFI) is enabled on the agent system.

Procedure

1. On the monitored system where the Keylime agent is configured and running, install the **python3-keylime** package, which contains the **keylime-policy** tool:

   ```
   # dnf -y install python3-keylime
   ```

2. On the monitored system, generate a policy from the measured boot log of the current state of the system by using the **keylime-policy** tool:

   ```
   # keylime-policy create measured-boot --eventlog-file
   /sys/kernel/security/tpm0/binary_bios_measurements --output
   <./measured_boot_reference_state.json>
   ```

   - Replace ***<./measured_boot_reference_state.json>*** with the path where **keylime-policy** saves the generated policy.
   - If your UEFI system does not have Secure Boot enabled, pass the **--without-secureboot** argument.

IMPORTANT

The policy generated with **keylime-policy** is based on the current state of the system and is very strict. Any modifications of the system including kernel updates and system updates will change the boot process and the system will fail the attestation.

3. Copy the generated policy to the system where the **keylime_tenant** utility is configured, for example:

```
# scp root@<agent_ip>:<./measured_boot_reference_state.json>
<./measured_boot_reference_state.json>
```

4. On the system where the Keylime tenant is configured, provision the agent by using the **keylime_tenant** utility:

```
# keylime_tenant --command add --targethost <agent_ip> --uuid <agent_uuid> --mb_refstate
<./measured_boot_reference_state.json> --cert default
```

- Replace ***<agent_ip>*** with the agent's IP address.
- Replace ***<agent_uuid>*** with the agent's UUID.
- Replace ***<./measured_boot_reference_state.json>*** with the path to the measured boot policy.

If you configure measured boot in combination with runtime monitoring, provide all the options from both use cases when entering the **keylime_tenant --command add** command.

NOTE

You can stop Keylime from monitoring a node by using the **# keylime_tenant --command delete --targethost *<agent_ip>* --uuid *<agent_uuid>*** command.

You can modify the configuration of an already registered agent by using the **keylime_tenant --command update** command.

Verification

1. Reboot the monitored system and verify a successful attestation of the agent:

```
# keylime_tenant --command cvstatus --uuid <agent_uuid>
...
{"<agent_uuid>": {"operational_state": "Get Quote"..."attestation_count": 5
...
```

Replace ***<agent_uuid>*** with the agent's UUID.

If the value of **operational_state** is **Get Quote** and **attestation_count** is nonzero, the attestation of this agent is successful.

If the value of **operational_state** is **Invalid Quote** or **Failed** attestation fails, the command displays output similar to the following:

```
{"<agent_uuid>": {"operational_state": "Invalid Quote", ... "ima.validation.ima-
ng.not_in_allowlist", "attestation_count": 5, "last_received_quote": 1684150329,
"last_successful_attestation": 1684150327}}
```

2. If the attestation fails, display more details in the verifier log:

```
# journalctl -u keylime_verifier
{"d432fbb3-d2f1-4a97-9ef7-75bd81c00000": {"operational_state": "Tenant Quote Failed", ...
"last_event_id": "measured_boot.invalid_pcr_0", "attestation_count": 0,
"last_received_quote": 1684487093, "last_successful_attestation": 0}}
```

6.12. KEYLIME ENVIRONMENT VARIABLES

In RHEL, you can set Keylime environment variables to override the values from the configuration files. For example, when starting a container with the **podman run** command by using the **-e** option.

The environment variables have the following syntax:

```
KEYLIME_<SECTION>_<ENVIRONMENT_VARIABLE>=<value>
```

Where:

- ***<SECTION>*** is the section of the Keylime configuration file.
- ***<ENVIRONMENT_VARIABLE>*** is the environment variable.
- ***<value>*** is the value to which you want to set the environment variable.

For example, **-e KEYLIME_VERIFIER_MAX_RETRIES=6** sets the **max_retries** configuration option in the **[verifier]** section to **6**.

6.12.1. Verifier configuration

Table 6.1. [verifier] section

Configuration option	Environment variable	Default value
auto_migrate_db	**KEYLIME_VERIFIER_AUTO_MIGRATE_DB**	**True**
client_cert	**KEYLIME_VERIFIER_CLIENT_CERT**	**default**
client_key_password	**KEYLIME_VERIFIER_CLIENT_KEY_PASSWORD**	
client_key	**KEYLIME_VERIFIER_CLIENT_KEY**	**default**
database_pool_sz_ovfl	**KEYLIME_VERIFIER_DATABASE_POOL_SZ_OVFL**	**5,10**

Configuration option	Environment variable	Default value
database_url	KEYLIME_VERIFIER_DATABASE_URL	sqlite
durable_attestation_import	KEYLIME_VERIFIER_DURABLE_ATTESTATION_IMPORT	
enable_agent_mtls	KEYLIME_VERIFIER_ENABLE_AGENT_MTLS	True
exponential_backoff	KEYLIME_VERIFIER_EXPONENTIAL_BACKOFF	True
ignore_tomtou_errors	KEYLIME_VERIFIER_IGNORE_TOMTOU_ERRORS	False
ip	KEYLIME_VERIFIER_IP	127.0.0.1
max_retries	KEYLIME_VERIFIER_MAX_RETRIES	5
max_upload_size	KEYLIME_VERIFIER_MAX_UPLOAD_SIZE	104857600
measured_boot_evaluate	KEYLIME_VERIFIER_MEASURED_BOOT_EVALUATE	once
measured_boot_imports	KEYLIME_VERIFIER_MEASURED_BOOT_IMPORTS	[]
measured_boot_policy_name	KEYLIME_VERIFIER_MEASURED_BOOT_POLICY_NAME	accept-all
num_workers	KEYLIME_VERIFIER_NUM_WORKERS	0
persistent_store_encoding	KEYLIME_VERIFIER_PERSISTENT_STORE_ENCODING	
persistent_store_format	KEYLIME_VERIFIER_PERSISTENT_STORE_FORMAT	json

Configuration option	Environment variable	Default value
persistent_store_url	KEYLIME_VERIFIER_PERSISTENT_STORE_URL	
port	KEYLIME_VERIFIER_PORT	8881
quote_interval	KEYLIME_VERIFIER_QUOTE_INTERVAL	2
registrar_ip	KEYLIME_VERIFIER_REGISTRAR_IP	127.0.0.1
registrar_port	KEYLIME_VERIFIER_REGISTRAR_PORT	8891
request_timeout	KEYLIME_VERIFIER_REQUEST_TIMEOUT	60.0
require_allow_list_signatures	KEYLIME_VERIFIER_REQUIRE_ALLOW_LIST_SIGNATURES	True
retry_interval	KEYLIME_VERIFIER_RETRY_INTERVAL	2
server_cert	KEYLIME_VERIFIER_SERVER_CERT	default
server_key_password	KEYLIME_VERIFIER_SERVER_KEY_PASSWORD	default
server_key	KEYLIME_VERIFIER_SERVER_KEY	default
severity_labels	KEYLIME_VERIFIER_SEVERITY_LABELS	["info", "notice", "warning", "error", "critical", "alert", "emergency"]
severity_policy	KEYLIME_VERIFIER_SEVERITY_POLICY	[{"event_id": ".*", "severity_label" : "emergency"}]
signed_attributes	KEYLIME_VERIFIER_SIGNED_ATTRIBUTES	

Configuration option	Environment variable	Default value
time_stamp_authority_certs_path	**KEYLIME_VERIFIER_TIME_STAMP_AUTHORITY_CERTS_PATH**	
time_stamp_authority_url	**KEYLIME_VERIFIER_TIME_STAMP_AUTHORITY_URL**	
tls_dir	**KEYLIME_VERIFIER_TLS_DIR**	**generate**
transparency_log_sign_algo	**KEYLIME_VERIFIER_TRANSPARENCY_LOG_SIGN_ALGO**	**sha256**
transparency_log_url	**KEYLIME_VERIFIER_TRANSPARENCY_LOG_URL**	
trusted_client_ca	**KEYLIME_VERIFIER_TRUSTED_CLIENT_CA**	**default**
trusted_server_ca	**KEYLIME_VERIFIER_TRUSTED_SERVER_CA**	**default**
uuid	**KEYLIME_VERIFIER_UUID**	**default**
version	**KEYLIME_VERIFIER_VERSION**	**2.0**

Table 6.2. **[revocations]** section

Configuration option	Environment variable	Default value
enabled_revocation_notifications	**KEYLIME_VERIFIER_REVOCATIONS_ENABLED_REVOCATION_NOTIFICATIONS**	**[agent]**
webhook_url	**KEYLIME_VERIFIER_REVOCATIONS_WEBHOOK_URL**	

6.12.2. Registrar configuration

Table 6.3. **[registrar]** section

Configuration option	Environment variable	Default value

auto_migrate_db	**KEYLIME_REGISTRAR_AUTO_MIGRATE_DB**	**True**
database_pool_sz_ovfl	**KEYLIME_REGISTRAR_DATABASE_POOL_SZ_OVFL**	**5,10**
database_url	**KEYLIME_REGISTRAR_DATABASE_URL**	**sqlite**
durable_attestation_import	**KEYLIME_REGISTRAR_DURABLE_ATTESTATION_IMPORT**	
ip	**KEYLIME_REGISTRAR_IP**	**127.0.0.1**
persistent_store_encoding	**KEYLIME_REGISTRAR_PERSISTENT_STORE_ENCODING**	
persistent_store_format	**KEYLIME_REGISTRAR_PERSISTENT_STORE_FORMAT**	**json**
persistent_store_url	**KEYLIME_REGISTRAR_PERSISTENT_STORE_URL**	
port	**KEYLIME_REGISTRAR_PORT**	**8890**
prov_db_filename	**KEYLIME_REGISTRAR_PROV_DB_FILENAME**	**provider_reg_data.sqlite**
server_cert	**KEYLIME_REGISTRAR_SERVER_CERT**	**default**
server_key_password	**KEYLIME_REGISTRAR_SERVER_KEY_PASSWORD**	**default**
server_key	**KEYLIME_REGISTRAR_SERVER_KEY**	**default**
signed_attributes	**KEYLIME_REGISTRAR_SIGNED_ATTRIBUTES**	**ek_tpm,aik_tpm,ekcert**
time_stamp_authority_certs_path	**KEYLIME_REGISTRAR_TIME_STAMP_AUTHORITY_CERTS_PATH**	

time_stamp_authority_url	**KEYLIME_REGISTRAR_TIME_STAMP_AUTHORITY_URL**	
tls_dir	**KEYLIME_REGISTRAR_TLS_DIR**	**default**
tls_port	**KEYLIME_REGISTRAR_TLS_PORT**	**8891**
transparency_log_sign_algo	**KEYLIME_REGISTRAR_TRANSPARENCY_LOG_SIGN_ALGO**	**sha256**
transparency_log_url	**KEYLIME_REGISTRAR_TRANSPARENCY_LOG_URL**	
trusted_client_ca	**KEYLIME_REGISTRAR_TRUSTED_CLIENT_CA**	**default**
version	**KEYLIME_REGISTRAR_VERSION**	**2.0**

6.12.3. Tenant configuration

Table 6.4. **[tenant]** section

Configuration option	Environment variable	Default value
accept_tpm_encryption_algs	**KEYLIME_TENANT_ACCEPT_TPM_ENCRYPTION_ALGS**	**ecc, rsa**
accept_tpm_hash_algs	**KEYLIME_TENANT_ACCEPT_TPM_HASH_ALGS**	**sha512, sha384, sha256**
accept_tpm_signing_algs	**KEYLIME_TENANT_ACCEPT_TPM_SIGNING_ALGS**	**ecschnorr, rsassa**
client_cert	**KEYLIME_TENANT_CLIENT_CERT**	**default**
client_key_password	**KEYLIME_TENANT_CLIENT_KEY_PASSWORD**	
client_key	**KEYLIME_TENANT_CLIENT_KEY**	**default**

ek_check_script	**KEYLIME_TENANT_EK_CHECK_SCRIPT**	
enable_agent_mtls	**KEYLIME_TENANT_ENABLE_AGENT_MTLS**	**True**
exponential_backoff	**KEYLIME_TENANT_EXPONENTIAL_BACKOFF**	**True**
max_payload_size	**KEYLIME_TENANT_MAX_PAYLOAD_SIZE**	**1048576**
max_retries	**KEYLIME_TENANT_MAX_RETRIES**	**5**
mb_refstate	**KEYLIME_TENANT_MB_REFSTATE**	
registrar_ip	**KEYLIME_TENANT_REGISTRAR_IP**	**127.0.0.1**
registrar_port	**KEYLIME_TENANT_REGISTRAR_PORT**	**8891**
request_timeout	**KEYLIME_TENANT_REQUEST_TIMEOUT**	**60**
require_ek_cert	**KEYLIME_TENANT_REQUIRE_EK_CERT**	**True**
retry_interval	**KEYLIME_TENANT_RETRY_INTERVAL**	**2**
tls_dir	**KEYLIME_TENANT_TLS_DIR**	**default**
tpm_cert_store	**KEYLIME_TENANT_TPM_CERT_STORE**	**/var/lib/keylime/tpm_cert_store**
trusted_server_ca	**KEYLIME_TENANT_TRUSTED_SERVER_CA**	**default**
verifier_ip	**KEYLIME_TENANT_VERIFIER_IP**	**127.0.0.1**

verifier_port	**KEYLIME_TENANT_VERIFIER_PORT**	**8881**
version	**KEYLIME_TENANT_VERSION**	**2.0**

6.12.4. CA configuration

Table 6.5. **[ca]** section

Configuration option	Environment variable	Default value
cert_bits	**KEYLIME_CA_CERT_BITS**	**2048**
cert_ca_lifetime	**KEYLIME_CA_CERT_CA_LIFETIME**	**3650**
cert_ca_name	**KEYLIME_CA_CERT_CA_NAME**	**Keylime Certificate Authority**
cert_country	**KEYLIME_CA_CERT_COUNTRY**	**US**
cert_crl_dist	**KEYLIME_CA_CERT_CRL_DIST**	**http://localhost:38080/crl**
cert_lifetime	**KEYLIME_CA_CERT_LIFETIME**	**365**
cert_locality	**KEYLIME_CA_CERT_LOCALITY**	**Lexington**
cert_org_unit	**KEYLIME_CA_CERT_ORG_UNIT**	**53**
cert_organization	**KEYLIME_CA_CERT_ORGANIZATION**	**MITLL**
cert_state	**KEYLIME_CA_CERT_STATE**	**MA**
password	**KEYLIME_CA_PASSWORD**	**default**
version	**KEYLIME_CA_VERSION**	**2.0**

6.12.5. Agent configuration

Table 6.6. **[agent]** section

Configuration option	Environment variable	Default value
contact_ip	KEYLIME_AGENT_CONTACT_IP	127.0.0.1
contact_port	KEYLIME_AGENT_CONTACT_PORT	9002
dec_payload_file	KEYLIME_AGENT_DEC_PAYLOAD_FILE	decrypted_payload
ek_handle	KEYLIME_AGENT_EK_HANDLE	generate
enable_agent_mtls	KEYLIME_AGENT_ENABLE_AGENT_MTLS	true
enable_insecure_payload	KEYLIME_AGENT_ENABLE_INSECURE_PAYLOAD	false
enable_revocation_notifications	KEYLIME_AGENT_ENABLE_REVOCATION_NOTIFICATIONS	true
enc_keyname	KEYLIME_AGENT_ENC_KEYNAME	derived_tci_key
exponential_backoff	KEYLIME_AGENT_EXPONENTIAL_BACKOFF	true
extract_payload_zip	KEYLIME_AGENT_EXTRACT_PAYLOAD_ZIP	true
ip	KEYLIME_AGENT_IP	127.0.0.1
max_retries	KEYLIME_AGENT_MAX_RETRIES	4
measure_payload_pcr	KEYLIME_AGENT_MEASURE_PAYLOAD_PCR	-1
payload_script	KEYLIME_AGENT_PAYLOAD_SCRIPT	autorun.sh
port	KEYLIME_AGENT_PORT	9002

Configuration option	Environment variable	Default value
registrar_ip	KEYLIME_AGENT_REGISTRAR_IP	127.0.0.1
registrar_port	KEYLIME_AGENT_REGISTRAR_PORT	8890
retry_interval	KEYLIME_AGENT_RETRY_INTERVAL	2
revocation_actions	KEYLIME_AGENT_REVOCATION_ACTIONS	[]
revocation_cert	KEYLIME_AGENT_REVOCATION_CERT	default
revocation_notification_ip	KEYLIME_AGENT_REVOCATION_NOTIFICATION_IP	127.0.0.1
revocation_notification_port	KEYLIME_AGENT_REVOCATION_NOTIFICATION_PORT	8992
run_as	KEYLIME_AGENT_RUN_AS	keylime:tss
secure_size	KEYLIME_AGENT_SECURE_SIZE	1m
server_cert	KEYLIME_AGENT_SERVER_CERT	default
server_key_password	KEYLIME_AGENT_SERVER_KEY_PASSWORD	
server_key	KEYLIME_AGENT_SERVER_KEY	default
tls_dir	KEYLIME_AGENT_TLS_DIR	default
tpm_encryption_alg	KEYLIME_AGENT_TPM_ENCRYPTION_ALG	rsa
tpm_hash_alg	KEYLIME_AGENT_TPM_HASH_ALG	sha256

Configuration option	Environment variable	Default value
tpm_ownerpassword	**KEYLIME_AGENT_TPM_OWNERPASSWORD**	
tpm_signing_alg	**KEYLIME_AGENT_TPM_SIGNING_ALG**	**rsassa**
trusted_client_ca	**KEYLIME_AGENT_TRUSTED_CLIENT_CA**	**default**
uuid	**KEYLIME_AGENT_UUID**	**d432fbb3-d2f1-4a97-9ef7-75bd81c00000**
version	**KEYLIME_AGENT_VERSION**	**2.0**

6.12.6. Logging configuration

Table 6.7. **[logging]** section

Configuration option	Environment variable	Default value
version	**KEYLIME_LOGGING_VERSION**	**2.0**

Table 6.8. **[loggers]** section

Configuration option	Environment variable	Default value
keys	**KEYLIME_LOGGING_LOGGERS_KEYS**	**root,keylime**

Table 6.9. **[handlers]** section

Configuration option	Environment variable	Default value
keys	**KEYLIME_LOGGING_HANDLERS_KEYS**	**consoleHandler**

Table 6.10. **[formatters]** section

Configuration option	Environment variable	Default value
keys	**KEYLIME_LOGGING_FORMATTERS_KEYS**	**formatter**

Table 6.11. **[formatter_formatter]** section

Configuration option	Environment variable	Default value
datefmt	**KEYLIME_LOGGING_FORMATTER_FORMATTER_DATEFMT**	**%Y-%m-%d %H:%M:%S**
format	**KEYLIME_LOGGING_FORMATTER_FORMATTER_FORMAT**	**%(asctime)s.%(msecs)03d - %(name)s - %(levelname)s - %(message)s**

Table 6.12. **[logger_root]** section

Configuration option	Environment variable	Default value
handlers	**KEYLIME_LOGGING_LOGGER_ROOT_HANDLERS**	**consoleHandler**
level	**KEYLIME_LOGGING_LOGGER_ROOT_LEVEL**	**INFO**

Table 6.13. **[handler_consoleHandler]** section

Configuration option	Environment variable	Default value
args	**KEYLIME_LOGGING_HANDLER_CONSOLEHANDLER_ARGS**	**(sys.stdout,)**
class	**KEYLIME_LOGGING_HANDLER_CONSOLEHANDLER_CLASS**	**StreamHandler**
formatter	**KEYLIME_LOGGING_HANDLER_CONSOLEHANDLER_FORMATTER**	**formatter**
level	**KEYLIME_LOGGING_HANDLER_CONSOLEHANDLER_LEVEL**	**INFO**

Table 6.14. **[logger_keylime]** section

Configuration option	Environment variable	Default value

handlers	KEYLIME_LOGGING_LOGGER_KEYLIME_HANDLERS	
level	KEYLIME_LOGGING_LOGGER_KEYLIME_LEVEL	INFO
qualname	KEYLIME_LOGGING_LOGGER_KEYLIME_QUALNAME	keylime

CHAPTER 7. CHECKING INTEGRITY WITH AIDE

Use the Advanced Intrusion Detection Environment (AIDE) utility in RHEL to create a file database and check system file integrity. This capability helps detect intrusions by reporting unauthorized changes.

7.1. INSTALLING AIDE

To start file-integrity checking with AIDE, you must install the corresponding package and initiate the AIDE database.

Prerequisites

- The **AppStream** repository is enabled.

Procedure

1. Install the **aide** package:

```
# dnf install aide
```

2. Generate an initial database:

```
# aide --init
Start timestamp: 2024-07-08 10:39:23 -0400 (AIDE 0.16)
AIDE initialized database at /var/lib/aide/aide.db.new.gz

Number of entries: 55856

---------------------------------------------------
The attributes of the (uncompressed) database(s):
---------------------------------------------------

/var/lib/aide/aide.db.new.gz
…
  SHA512    : mZaWoGzL2m6ZcyyZ/AXTIowliEXWSZqx
              IFYImY4f7id4u+Bq8WeuSE2jasZur/A4
              FPBFaBkoCFHdoE/FW/V94Q==
```

3. Optional: In the default configuration, the **aide --init** command checks just a set of directories and files defined in the **/etc/aide.conf** file. To include additional directories or files in the AIDE database, and to change their watched parameters, edit **/etc/aide.conf** accordingly.

4. To start using the database, remove the **.new** substring from the initial database file name:

```
# mv /var/lib/aide/aide.db.new.gz /var/lib/aide/aide.db.gz
```

5. Optional: To change the location of the AIDE database, edit the **/etc/aide.conf** file and modify the **DBDIR** value. For additional security, store the database, configuration, and the **/usr/sbin/aide** binary file in a secure location such as a read-only media.

7.2. PERFORMING INTEGRITY CHECKS WITH AIDE

Use Advanced Intrusion Detection Environment (AIDE) to perform file-integrity checks by comparing the current file state against its internal database. This check produces a report detailing changes since the last run. You can use the **crond** service to schedule regular checks with AIDE.

Prerequisites

- AIDE is properly installed and its database is initialized. See Installing AIDE

Procedure

1. To initiate a manual check:

```
# aide --check
Start timestamp: 2024-07-08 10:43:46 -0400 (AIDE 0.16)
AIDE found differences between database and file system!!

Summary:
  Total number of entries: 55856
  Added entries:  0
  Removed entries:  0
  Changed entries:  1

---------------------------------------------------
Changed entries:
---------------------------------------------------

f   ...      ..S : /root/.viminfo

---------------------------------------------------
Detailed information about changes:
---------------------------------------------------

File: /root/.viminfo
  SELinux   : system_u:object_r:admin_home_t:s | unconfined_u:object_r:admin_home
            0                                  | _t:s0
…
```

2. At a minimum, configure the system to run AIDE weekly. Optimally, run AIDE daily. For example, to schedule a daily execution of AIDE at *04:05* a.m. by using the **cron** command, add the following line to the **/etc/crontab** file:

```
05 4 * * * root /usr/sbin/aide --check
```

See the **cron(8)** man page on your system for details on the **crond** scheduling daemon.

7.3. UPDATING AN AIDE DATABASE

Update the AIDE database after making legitimate system changes, such as installing a software update. This action prevents false positives during subsequent integrity checks.

Prerequisites

- AIDE is properly installed and its database is initialized. See Installing AIDE

Procedure

1. Update your baseline AIDE database:

   ```
   # aide --update
   ```

   The **aide --update** command creates the **/var/lib/aide/aide.db.new.gz** database file.

2. To start using the updated database for integrity checks, remove the **.new** substring from the file name.

7.4. FILE-INTEGRITY TOOLS: AIDE AND IMA

Compare file integrity tools, AIDE and Integrity Measurement Architecture (IMA), to select the one that best fits your scenario. This helps you decide on the appropriate method for checking and preserving file integrity.

Table 7.1. Comparison between AIDE and IMA

Question	Advanced Intrusion Detection Environment (AIDE)	Integrity Measurement Architecture (IMA)
What	AIDE is a utility that creates a database of files and directories on the system. This database serves for checking file integrity and detect intrusion detection.	IMA detects if a file is altered by checking file measurement (hash values) compared to previously stored extended attributes.
How	AIDE uses rules to compare the integrity state of the files and directories.	IMA uses file hash values to detect the intrusion.
Why	Detection - AIDE detects if a file is modified by verifying the rules.	Detection and Prevention - IMA detects and prevents an attack by replacing the extended attribute of a file.
Usage	AIDE detects a threat when the file or directory is modified.	IMA detects a threat when someone tries to alter the entire file.
Extension	AIDE checks the integrity of files and directories on the local system.	IMA ensures security on the local and remote systems.

7.5. CONFIGURING FILE INTEGRITY CHECKS WITH THE AIDE RHEL SYSTEM ROLE

You can configure Advanced Intrusion Detection Environment (AIDE) consistently across multiple systems by using the **aide** RHEL system role. The role automatically installs the **aide** package on all managed nodes.

Depending on your configuration, the **aide** role can perform the following actions:

- Initialize the AIDE database and store it on the control node
- Run AIDE integrity checks on the managed nodes

- Update the AIDE database and store it on the control node

Prerequisites

- You have prepared the control node and the managed nodes .
- You are logged in to the control node as a user who can run playbooks on the managed nodes.
- The account you use to connect to the managed nodes has **sudo** permissions for these nodes.

Procedure

1. Create a playbook file, for example, **~/playbook.yml**, with the following content:

```
---
- name: Configure system integrity
  hosts: managed-node-01.example.com
  tasks:
    - name: Configure file integrity checks with AIDE
      ansible.builtin.include_role:
        name: rhel-system-roles.aide.aide
      vars:
        aide_db_fetch_dir: files
        aide_init: true
        aide_check: false
        aide_update: false
        aide_cron_check: true
        aide_cron_interval: 0 12 * * *
```

The settings specified in the example playbook include the following:

aide_db_fetch_dir: files

Specifies the directory on the Ansible Control Node (ACN) for storing the AIDE database fetched from the remote nodes. With the default **files** value, the role stores the database in the same directory as the playbook. To store the database files somewhere else, specify a different path.

aide_check: false

Runs an integrity check on the remote nodes.

aide_update: false

Updates the AIDE database and stores it on the control node.

aide_cron_check: true

Configures a periodic **cron** job that activates AIDE integrity checks on the managed nodes.

aide_cron_interval: 0 12 * * *

Sets the interval for the **cron** job in the format ***<minute> <hour> <day_of_month> <month> <day of week>***. The value **0 12 * * *** sets it to run every day at noon.

For details about all variables used in the playbook, see the **/usr/share/ansible/roles/rhel-system-roles.aide/README.md** file on the control node.

2. Validate the playbook syntax:

```
$ ansible-playbook --syntax-check ~/playbook.yml
```

Note that this command only validates the syntax and does not protect against a wrong but valid configuration.

3. Run the playbook:

```
$ ansible-playbook ~/playbook.yml
```

7.6. ADDITIONAL RESOURCES

- Enhancing security with the kernel integrity subsystem

CHAPTER 8. MANAGING SUDO ACCESS

Manage **sudo** access to allow non-root users to securely use administrative commands without logging into the root account. Proper configuration helps control privileges and limits exposure.

8.1. USER AUTHORIZATIONS IN SUDOERS

In RHEL, the **/etc/sudoers** file and the **/etc/sudoers.d/** directory store user and group authorizations for the **sudo** command. You can use these configurations to specify which users can execute administrative commands on specific hosts.

You can apply rules to individual users and user groups. To simplify configurations for groups of hosts, commands, and users, you can use aliases. The **/etc/sudoers** file defines default aliases in the first section.

When a user enters a command with **sudo** for which the user does not have authorization, the system records a message that contains the string ***<username>* : user NOT in sudoers** to the journal log.

The default **/etc/sudoers** file provides information and examples of authorizations. You can activate a specific example rule by uncommenting the corresponding line. The section with user authorizations is marked with the following introduction:

```
## Next comes the main part: which users can run what software on
## which machines  (the sudoers file can be shared between multiple
## systems).
```

You can create new **sudoers** authorizations and modify existing authorizations by using the following format:

```
<username> <hostname.example.com>=(<run_as_user>:<run_as_group>) <path/to/command>
```

Where:

- ***<username>*** is the user that enters the command, for example, **user1**. If the value starts with **%**, it defines a group, for example, **%group1**.
- ***<hostname.example.com>*** is the name of the host on which the rule applies.
- The section **(*<run_as_user>*:*<run_as_group>*)** defines the user or group as which the command is executed. If you omit this section, ***<username>*** can use the command as root.
- ***<path/to/command>*** is the complete absolute path to the command. You can also limit the user to only performing a command with specific options and arguments by adding those options after the command path. If you do not specify any options, the user can use the command with all options.

You can apply the rule to all users, hosts, or commands by replacing any of these variables with **ALL**.

WARNING

With overly permissive rules, such as **ALL ALL=(ALL) ALL**, all users can run all commands as all users on all hosts. This presents serious security risks.

You can specify the arguments negatively by using the **!** operator. For example, **!root** specifies all users except root. Note that allowing specific users, groups, and commands is more secure than disallowing specific users, groups, and commands. This is because allow rules also block new unauthorized users or groups.

WARNING

Avoid using negative rules for commands because users can overcome such rules by renaming commands with the **alias** command.

The system reads the **/etc/sudoers** file from beginning to end. Therefore, if the file contains multiple entries for a user, the entries are applied in order. In case of conflicting values, the system uses the last match, even if it is not the most specific match.

To preserve the rules during system updates and for easier fixing of errors, enter new rules by creating new files in the **/etc/sudoers.d/** directory instead of entering rules directly to the **/etc/sudoers** file. The system reads the files in the **/etc/sudoers.d** directory when it reaches the following line in the **/etc/sudoers** file:

```
#includedir /etc/sudoers.d
```

Note that the number sign (**#**) at the beginning of this line is part of the syntax and does not mean the line is a comment. The names of files in that directory must not contain a period and must not end with a tilde (~).

For more information, see the **sudo(8)** and **sudoers(5)** man pages on your system.

8.2. GRANTING SUDO ACCESS TO A USER

In RHEL, you can allow non-root users to execute administrative commands by granting them **sudo** access. This ensures that users can perform privileged tasks securely without requiring the root user's password.

When users need to perform an administrative command, they can precede that command with **sudo**. If the user has authorization for the command, the command is executed as if they were root.

Be aware of the following limitations:

- Only users listed in the **/etc/sudoers** configuration file can use the **sudo** command.

- The command is executed in the shell of the user, not in the root shell. However, there are some exceptions such as when full **sudo** privileges are granted to any user. In such cases, users can switch to and run the commands in root shell. For example:

- **sudo -i**

- **sudo su -**

Prerequisites

- You have root access to the system.

Procedure

1. As root, open the **/etc/sudoers** file.

   ```
   # visudo
   ```

   The **/etc/sudoers** file defines the policies applied by the **sudo** command.

2. In the **/etc/sudoers** file, find the lines that grant **sudo** access to users in the administrative **wheel** group.

   ```
   ## Allows people in group wheel to run all commands
   %wheel        ALL=(ALL)       ALL
   ```

3. Make sure the line that starts with **%wheel** is not commented out with the number sign (**#**).

4. Save any changes, and exit the editor.

5. Add users you want to grant **sudo** access to into the administrative **wheel** group.

   ```
   # usermod --append -G wheel <username>
   ```

   Replace ***<username>*** with the name of the user.

Verification

- Verify that the user is in the administrative **wheel** group:

  ```
  # id <username>
  uid=5000(<username>) gid=5000(<username>) groups=5000(<username>),10(wheel)
  ```

8.3. ENABLING UNPRIVILEGED USERS TO RUN CERTAIN COMMANDS

As an administrator, you can allow unprivileged users to enter certain commands on specific workstations by configuring a policy in the **/etc/sudoers.d/** directory. This helps improve security compared to granting full **sudo** access.

Specifying a policy allowing only certain commands is more secure than granting full **sudo** access to a user or giving someone the root password for the following reasons:

- More granular control over privileged actions. You can allow a user to perform certain actions on specific hosts instead of giving them full administrative access.
- Better logging. When a user performs an action with **sudo**, the action is logged with their user name, not just root.
- Transparent control. You can set email notifications whenever a user attempts to use **sudo** privileges.

For additional details and examples, see the **sudoers(5)** man pages on your system.

Prerequisites

- You have root access to the system.

Procedure

1. As root, create a new **sudoers.d**/ directory under /**etc**/:

   ```
   # mkdir -p /etc/sudoers.d/
   ```

2. Create a new file in the /**etc**/**sudoers.d**/ directory:

   ```
   # visudo -f /etc/sudoers.d/<filename>
   ```

   The file opens automatically.

3. Add the following line to the /**etc**/**sudoers.d**/***<filename>*** file:

   ```
   <username> <hostname.example.com> = (<run_as_user>:<run_as_group>)
   <path/to/command>
   ```

   - Replace ***<username>*** with the name of the user.
   - Replace ***<hostname.example.com>*** with the URL of the host.
   - Replace **(*<run_as_user>*:*<run_as_group>*)** with the user or group as which the command can be executed. If you omit this section, ***<username>*** can execute the command as root.
   - Replace ***<path/to/command>*** with the complete absolute path to the command. You can also limit the user to only performing a command with specific options and arguments by adding those options after the command path. If you do not specify any options, the user can use the command with all options.
   - To allow two and more commands on the same host on one line, you can list them separated by a comma followed by a space.
     For example, to allow **user1** to execute the **dnf** and **reboot** commands on **host1.example.com**, enter **user1 host1.example.com = /bin/dnf, /sbin/reboot**.

4. Optional: To receive email notifications every time the user attempts to use **sudo** privileges, add the following lines to the file:

   ```
   Defaults    mail_always
   Defaults    mailto="<email@example.com>"
   ```

5. Save the changes, and exit the editor.

Verification

1. To verify if a user can run a command with **sudo** privileges, switch the account:

   ```
   # su <username> -
   ```

2. As the user, enter the command with the **sudo** command:

   ```
   $ sudo <command>
   [sudo] password for <username>:
   ```

   Enter the user's **sudo** password.

3. If the privileges are configured correctly, the system displays the list of commands and options. For example, with the **dnf** command, it shows the following output:

   ```
   ...
   usage: dnf [options] COMMAND
   ...
   ```

   If the system returns the error message ***<username>* is not in the sudoers file. This incident will be reported**, the file for ***<username>*** in **/etc/sudoers.d/** does not exist.

   If the system returns the error message ***<username>* is not allowed to run sudo on *<host.example.com>***, the configuration was not completed correctly. Ensure that you are logged in as root and that the configuration was performed correctly.

   If the system returns the error message **Sorry, user *<username>* is not allowed to execute '*<path/to/command>*' as root on *<host.example.com>*.**, the command is not correctly defined in the rule for the user.

8.4. APPLYING CUSTOM SUDOERS CONFIGURATION BY USING RHEL SYSTEM ROLES

You can use the **sudo** RHEL system role to apply custom **sudoers** configuration on your managed nodes. That way, you can define which users can run which commands on which hosts, with better configuration efficiency and more granular control.

Prerequisites

- You have prepared the control node and the managed nodes .
- You are logged in to the control node as a user who can run playbooks on the managed nodes.
- The account you use to connect to the managed nodes has **sudo** permissions for these nodes.

Procedure

1. Create a playbook file, for example, **~/playbook.yml**, with the following content:

   ```
   ---
   ```

```
- name: "Configure sudo"
  hosts: managed-node-01.example.com
  tasks:
    - name: "Apply custom /etc/sudoers configuration"
      ansible.builtin.include_role:
        name: redhat.rhel_system_roles.sudo
      vars:
        sudo_sudoers_files:
          - path: "/etc/sudoers"
            user_specifications:
              - users:
                  - <user_name>
                hosts:
                  - <host_name>
                commands:
                  - <path_to_command_binary>
```

The settings specified in the playbook include the following:

users

The list of users that the rule applies to.

hosts

The list of hosts that the rule applies to. You can use **ALL** for all hosts.

commands

The list of commands that the rule applies to. You can use **ALL** for all commands.

For details about all variables used in the playbook, see the **/usr/share/ansible/roles/rhel-system-roles.sudo/README.md** file on the control node.

2. Validate the playbook syntax:

```
$ ansible-playbook --syntax-check ~/playbook.yml
```

Note that this command only validates the syntax and does not protect against a wrong but valid configuration.

3. Run the playbook:

```
$ ansible-playbook ~/playbook.yml
```

Verification

1. On the managed node, verify that the playbook applied the new rules.

```
# cat /etc/sudoers | tail -n1
<user_name> <host_name>= <path_to_command_binary>
```

CHAPTER 9. CONFIGURING AUTOMATED UNLOCKING OF ENCRYPTED VOLUMES BY USING POLICY-BASED DECRYPTION

Policy-Based Decryption (PBD) enables unlocking encrypted root and secondary storage volumes on physical and virtual machines. PBD uses a variety of unlocking methods, such as user passwords, Trusted Platform Module (TPM) devices, PKCS #11 devices, or a network server.

PBD allows combining different unlocking methods into a policy, which makes it possible to unlock the same volume in different ways. The current implementation of the PBD in Red Hat Enterprise Linux consists of the Clevis framework and plugins called *pins*. Each pin provides a separate unlocking capability. Currently, the following pins are available:

tang

Allows unlocking volumes by using a network server.

tpm2

Allows unlocking volumes by using a TPM2 policy.

pkcs11

Allows unlocking volumes by using a PKCS #11 URI.

sss

Allows deploying high-availability systems by using the Shamir’s Secret Sharing (SSS) cryptographic scheme.

9.1. NETWORK-BOUND DISK ENCRYPTION

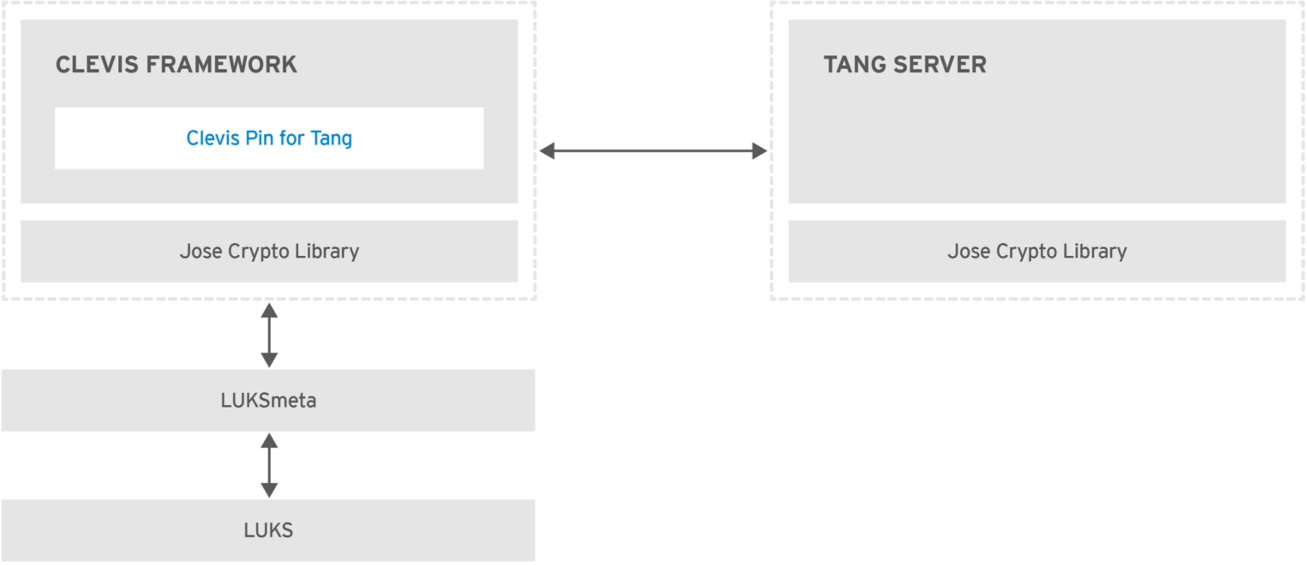
The Network-bound Disc Encryption (NBDE) is a subcategory of Policy-Based Decryption (PBD) that allows binding encrypted volumes to a special network server. The current implementation of the NBDE includes a Clevis pin for the Tang server and the Tang server itself.

Clevis and Tang are generic client and server components that provide network-bound encryption. In RHEL, they are used in conjunction with LUKS to encrypt and decrypt root and non-root storage volumes, thereby accomplishing Network-bound Disk Encryption.

9.1.1. NBDE components

In RHEL, NBDE is implemented through the following components and technologies:

Figure 9.1. NBDE scheme when using a LUKS1-encrypted volume. LUKSmeta is not used for LUKS2 volumes.



RHEL_453350_0717

Tang

Tang is a server for binding data to network presence. It makes a system containing your data available when the system is bound to a certain secure network. Tang is stateless and does not require TLS or authentication. Unlike escrow-based solutions, where the server stores all encryption keys and has knowledge of every key ever used, Tang never interacts with any client keys. Therefore, it never gains any identifying information from the client.

Clevis

Clevis is a pluggable framework for automated decryption. In NBDE, Clevis provides automated unlocking of LUKS volumes. The **clevis** package provides the client side of the feature.

Clevis pin

A Clevis pin is a plugin into the Clevis framework. One of such pins is a plugin that implements interactions with the NBDE server - Tang.

José

Both client- and server-side components use the José library to perform encryption and decryption operations.

LUKSMeta

The LUKS version 2 (LUKS2) is the default disk-encryption format in RHEL. Therefore, the provisioning state for NBDE is stored as a token in a LUKS2 header. The leveraging of provisioning state for NBDE by the **luksmeta** package is used only for volumes encrypted with LUKS1.

The Clevis pin for Tang supports both LUKS1 and LUKS2 without requiring specification. Clevis can encrypt plain text files but you have to use the **cryptsetup** tool for encrypting block devices.

9.1.2. NBDE principles

When you begin provisioning NBDE, the Clevis pin for Tang server gets a list of the Tang server’s advertised asymmetric keys. Alternatively, since the keys are asymmetric, a list of Tang’s public keys can be distributed out of band so that clients can operate without access to the Tang server. This mode is called *offline provisioning*.

The Clevis pin for Tang uses one of the public keys to generate a unique, cryptographically strong encryption key. Once the data is encrypted by using this key, the key is discarded. The Clevis client should store the state produced by this provisioning operation in a convenient location. This process of encrypting data is the *provisioning step*.

When the client is ready to access its data, it loads the metadata produced in the provisioning step, and it responds to recover the encryption key. This process is the *recovery step*.

In NBDE, Clevis binds a LUKS volume by using a pin so that it can be automatically unlocked. After successful completion of the binding process, the disk can be unlocked using the provided Dracut unlocker.

For more information, see the **tang(8)**, **clevis(1)**, **jose(1)**, and **clevis-luks-unlockers(7)** man pages on your system.

NOTE

If the **kdump** kernel crash dumping mechanism is set to save the content of the system memory to a LUKS-encrypted device, you are prompted for entering a password during the second kernel boot.

Additional resources

- NBDE (Network-Bound Disk Encryption) Technology (Red Hat Knowledgebase)
- How to set up Network-Bound Disk Encryption with multiple LUKS devices (Clevis + Tang unlocking) (Red Hat Knowledgebase)

9.2. DEPLOYING A TANG SERVER

You can use a Tang server to unlock LUKS-encrypted volumes on Clevis-enabled clients automatically. In the minimalist scenario, you deploy a Tang server by installing its package and entering a single command.

Prerequisites

- Commands that start with the **#** command prompt require administrative privileges provided by **sudo** or root user access. For information on how to configure **sudo** access, see Enabling unprivileged users to run certain commands.

Procedure

1. Install the **tang** package and its dependencies:

```
# dnf install tang
```

2. Start and enable the **tangd** service:

```
# systemctl enable tangd.socket --now
```

Because **tangd** uses the **systemd** socket activation mechanism, the server starts as soon as the first connection comes in. A new set of cryptographic keys is automatically generated at the first start. To perform cryptographic operations such as manual key generation, use the **jose** utility as described in the **jose(1)** man page on your system.

Verification

- On your NBDE client, verify that your Tang server works correctly by using the following command. The command must return the identical message you pass for encryption and decryption:

  ```
  # echo test | clevis encrypt tang '{"url":"<tang.server.example.com:80>"}' -y | clevis decrypt
  test
  ```

Next steps

- For deployments on a different port than 80, see Deploying a Tang server with SELinux in enforcing mode.

9.3. DEPLOYING A TANG SERVER WITH SELINUX IN ENFORCING MODE

Because a different service on your server might occupy port 80, which the Tang server uses by default, you can deploy Tang on a custom port, ideally as a confined service in SELinux enforcing mode.

Prerequisites

- The **policycoreutils-python-utils** package and its dependencies are installed.
- The **firewalld** service is running.

Procedure

1. To install the **tang** package and its dependencies, enter the following command as **root**:

   ```
   # dnf install tang
   ```

2. Pick an unoccupied port, for example, *7500/tcp*, and allow the **tangd** service to bind to that port:

   ```
   # semanage port -a -t tangd_port_t -p tcp 7500
   ```

   Note that a port can be used only by one service at a time.

3. Open the port in the firewall:

   ```
   # firewall-cmd --add-port=7500/tcp
   # firewall-cmd --runtime-to-permanent
   ```

4. Enable the **tangd** service:

   ```
   # systemctl enable tangd.socket
   ```

5. Create an override file:

   ```
   # systemctl edit tangd.socket
   ```

6. In the following editor screen, which opens an empty **override.conf** file located in the **/etc/systemd/system/tangd.socket.d/** directory, change the default port for the Tang server from 80 to the previously picked number. Insert the following code snippet between the lines starting with **# Anything between here** and **# Lines below this**, otherwise the system discards your changes:

   ```
   [Socket]
   ListenStream=
   ListenStream=7500
   ```

7. Save the changes and exit the editor. In the default **vi** editor, you can do that by pressing **Esc** to switch into command mode, entering **:wq**, and pressing **Enter**.

8. Reload the changed configuration:

   ```
   # systemctl daemon-reload
   ```

9. Check that your configuration is working:

   ```
   # systemctl show tangd.socket -p Listen
   Listen=[::]:7500 (Stream)
   ```

10. Start the **tangd** service:

    ```
    # systemctl restart tangd.socket
    ```

    Because **tangd** uses the **systemd** socket activation mechanism, the server starts as soon as the first connection comes in. A new set of cryptographic keys is automatically generated at the first start. To perform cryptographic operations such as manual key generation, use the **jose** utility as described in the **jose(1)** man page on your system.

Verification

- On your NBDE client, verify that your Tang server works correctly by using the following command. The command must return the identical message you pass for encryption and decryption:

  ```
  # echo test | clevis encrypt tang '{"url":"<tang.server.example.com:7500>"}' -y | clevis decrypt
  test
  ```

9.4. ROTATING TANG SERVER KEYS AND UPDATING BINDINGS ON CLIENTS

For security reasons, rotate your Tang server keys and update existing bindings on clients periodically. The precise interval at which you should rotate them depends on your application, key sizes, and institutional policy.

Alternatively, you can rotate Tang keys by using the **nbde_server** RHEL system role. See Using the nbde_server system role for setting up multiple Tang servers for more information.

> WARNING
>
> Removing the old keys while clients are still using them can result in data loss. If you accidentally remove such keys, use the **clevis luks regen** command on the clients, and provide your LUKS password manually.

Prerequisites

- A Tang server is running.
- The **clevis** and **clevis-luks** packages are installed on your clients.

Procedure

1. Rename all keys in the **/var/db/tang** key database directory to have a leading **.** to hide them from advertisement. Note that the file names in the following example differs from unique file names in the key database directory of your Tang server:

```
# cd /var/db/tang
# ls -l
-rw-r--r--. 1 root root 349 Feb  7 14:55 UV6dqXSwe1bRKG3KbJmdiR020hY.jwk
-rw-r--r--. 1 root root 354 Feb  7 14:55 y9hxLTQSiSB5jSEGWnjhY8fDTJU.jwk
# mv UV6dqXSwe1bRKG3KbJmdiR020hY.jwk .UV6dqXSwe1bRKG3KbJmdiR020hY.jwk
# mv y9hxLTQSiSB5jSEGWnjhY8fDTJU.jwk .y9hxLTQSiSB5jSEGWnjhY8fDTJU.jwk
```

2. Check that you renamed and therefore hid all keys from the Tang server advertisement:

```
# ls -l
total 0
```

3. Generate new keys using the **/usr/libexec/tangd-keygen** command in **/var/db/tang** on the Tang server:

```
# /usr/libexec/tangd-keygen /var/db/tang
# ls /var/db/tang
3ZWS6-cDrCG61UPJS2BMmPU4I54.jwk zyLuX6hijUy_PSeUEFDi7hi38.jwk
```

4. Check that your Tang server advertises the signing key from the new key pair, for example:

```
# tang-show-keys 7500
3ZWS6-cDrCG61UPJS2BMmPU4I54
```

5. On your NBDE clients, use the **clevis luks report** command to check if the keys advertised by the Tang server remains the same. You can identify slots with the relevant binding using the **clevis luks list** command, for example:

```
# clevis luks list -d /dev/sda2
1: tang '{"url":"http://tang.srv"}'
# clevis luks report -d /dev/sda2 -s 1
```

```
...
Report detected that some keys were rotated.
Do you want to regenerate luks metadata with "clevis luks regen -d /dev/sda2 -s 1"? [ynYN]
```

6. To regenerate LUKS metadata for the new keys, either press **y** to the prompt of the previous command or use the **clevis luks regen** command:

```
# clevis luks regen -d /dev/sda2 -s 1
```

7. When you are sure that all old clients use the new keys, you can remove the old keys from the Tang server, for example:

```
# cd /var/db/tang
# rm .*.jwk
```

9.5. CONFIGURING AUTOMATED UNLOCKING BY USING A TANG KEY IN THE WEB CONSOLE

You can configure automated unlocking of a LUKS-encrypted storage device by using a key provided by a Tang server in the RHEL web console GUI.

Prerequisites

- You have installed the RHEL 10 web console.
  For instructions, see Installing and enabling the web console .

- The **cockpit-storaged** and **clevis-luks** packages are installed on your system.

- The **cockpit.socket** service is running at port 9090.

- A Tang server is available. See Deploying a Tang server with SELinux in enforcing mode for details.

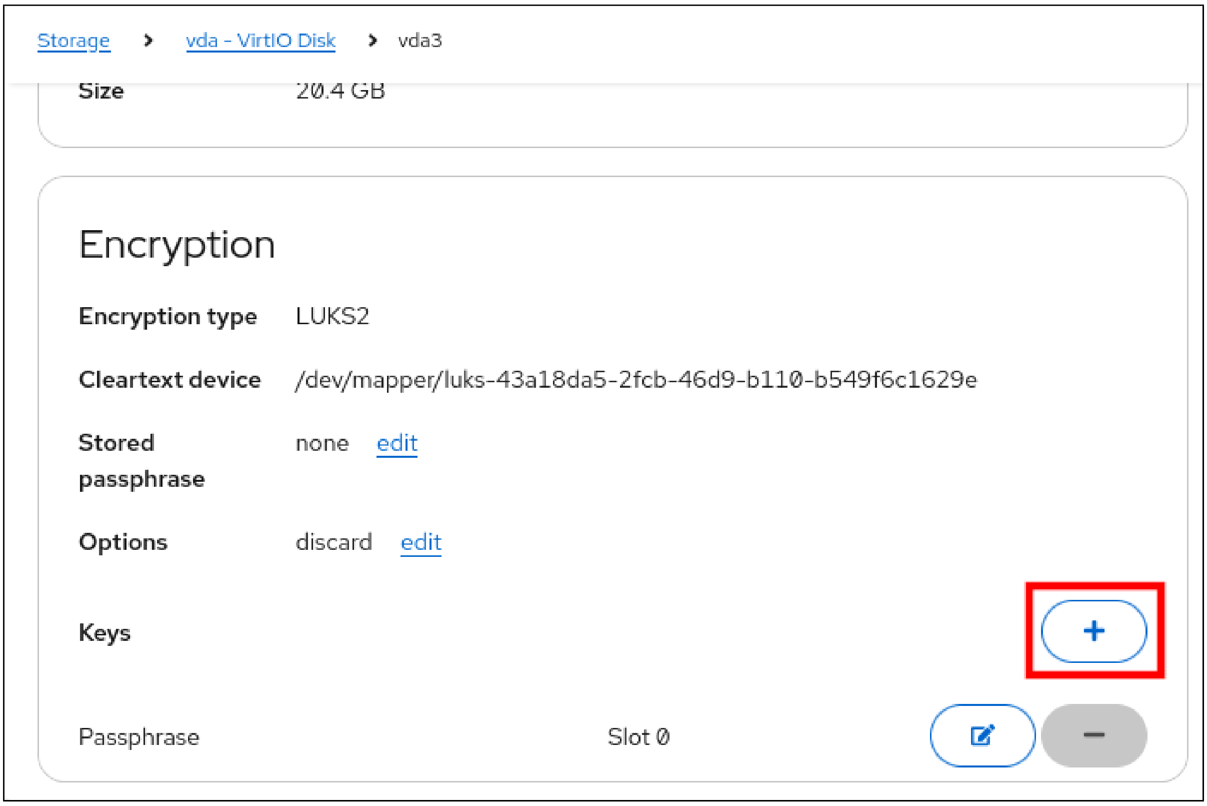
- You have **root** privileges or permissions to enter administrative commands with **sudo**.

Procedure

1. Log in to the RHEL 10 web console.

2. Switch to administrative access, provide your credentials, and click **Storage**. In the **Storage** table, click the disk that contains an encrypted volume you plan to add to unlock automatically.

3. In the following page with details of the selected disk, click **+** in the **Keys** section to add a Tang key:



Storage > vda - VirtIO Disk > vda3

Size 20.4 GB

Encryption

Encryption type LUKS2

Cleartext device /dev/mapper/luks-43a18da5-2fcb-46d9-b110-b549f6c1629e

Stored passphrase none edit

Options discard edit

Keys

Passphrase Slot 0

4. Select **Tang keyserver** as **Key source**, provide the address of your Tang server, and a password that unlocks the LUKS-encrypted device. Click **Add** to confirm.
   The following dialog window provides a command to verify that the key hash matches.

5. In a terminal on the Tang server, use the **tang-show-keys** command to display the key hash for comparison. In this example, the Tang server is running on the port *7500*:

```
# tang-show-keys 7500
x100_1k6GPiDOaMlL3WbpCjHOy9ul1bSfdhI3M08wO0
```

6. Click **Trust key** when the key hashes in the web console and in the output of previously listed commands are the same.

Next steps

- After you select an encrypted root file system and a Tang server, you can skip adding the **rd.neednet=1** parameter to the kernel command line, installing the **clevis-dracut** package, and regenerating an initial RAM disk (**initrd**). For non-root file systems, the web console now enables the **remote-cryptsetup.target** and **clevis-luks-akspass.path** systemd units, installs the **clevis-systemd** package, and adds the **_netdev** parameter to the **fstab** and **crypttab** configuration files.

Verification

1. In the **Storage** table in the RHEL web console, check that the newly added Tang key is now listed in the **Keys** section with the **Keyserver** type.

2. Verify that the bindings are available for the early boot, for example:

```
# lsinitrd | grep clevis-luks
lrwxrwxrwx   1 root     root          48 Jan  4 02:56
```

```
etc/systemd/system/cryptsetup.target.wants/clevis-luks-askpass.path ->
/usr/lib/systemd/system/clevis-luks-askpass.path
…
```

Additional resources

- Getting started with the RHEL web console

9.6. BASIC NBDE AND TPM2 ENCRYPTION-CLIENT OPERATIONS

You can learn basic functionality provided by Clevis on examples containing plain text files. You can also use them for troubleshooting your NBDE or Clevis+TPM deployments.

The Clevis framework can encrypt plain text files and decrypt both ciphertext in the JSON Web Encryption (JWE) format and LUKS-encrypted block devices. Clevis clients can use either Tang network servers, Trusted Platform Module 2.0 (TPM 2.0) chips, or PKCS #11 devices for cryptographic operations.

Encryption client bound to a Tang server

- To check that a Clevis encryption client binds to a Tang server, use the **clevis encrypt tang** subcommand:

```
$ clevis encrypt tang '{"url":"http://tang.srv:port"}' < input-plain.txt > secret.jwe
The advertisement contains the following signing keys:

_OsIk0T-E2l6qjfdDiwVmidoZjA

Do you wish to trust these keys? [ynYN] y
```

Change the ***http://tang.srv:port*** URL in the previous example to match the URL of the server where **tang** is installed. The ***secret.jwe*** output file contains your encrypted ciphertext in the JWE format. This ciphertext is read from the ***input-plain.txt*** input file.

Alternatively, if your configuration requires a noninteractive communication with a Tang server without SSH access, you can download an advertisement and save it to a file:

```
$ curl -sfg http://tang.srv:port/adv -o adv.jws
```

Use the advertisement in the ***adv.jws*** file for any following tasks, such as encryption of files or messages:

```
$ echo 'hello' | clevis encrypt tang '{"url":"http://tang.srv:port","adv":"adv.jws"}'
```

- To decrypt data, use the **clevis decrypt** command and provide the ciphertext (JWE):

```
$ clevis decrypt < secret.jwe > output-plain.txt
```

- The **clevis**, **clevis decrypt**, and **clevis encrypt tang** commands without any arguments show the built-in CLI help, for example:

```
$ clevis encrypt tang
Usage: clevis encrypt tang CONFIG < PLAINTEXT > JWE
…
```

- For more information and examples, see the **clevis-encrypt-tang(1)**, **clevis-luks-unlockers(7)**, and **clevis(1)** man pages on your system.

Encryption client that uses TPM 2.0

- To encrypt by using a TPM 2.0 chip, use the **clevis encrypt tpm2** subcommand with the only argument in form of the JSON configuration object:

```
$ clevis encrypt tpm2 '{}' < input-plain.txt > secret.jwe
```

To choose a different hierarchy, hash, and key algorithms, specify configuration properties, for example:

```
$ clevis encrypt tpm2 '{"hash":"sha256","key":"rsa"}' < input-plain.txt > secret.jwe
```

- To decrypt the data, provide the ciphertext in the JSON Web Encryption (JWE) format:

```
$ clevis decrypt < secret.jwe > output-plain.txt
```

- The pin also supports sealing data to a Platform Configuration Registers (PCR) state. That way, the data can only be unsealed if the PCR hashes values match the policy used when sealing.
  For example, to seal the data to the PCR with index 0 and 7 for the SHA-256 bank:

```
$ clevis encrypt tpm2 '{"pcr_bank":"sha256","pcr_ids":"0,7"}' < input-plain.txt > secret.jwe
```

> **WARNING**
>
> Hashes in PCRs can be rewritten, causing you to no longer be able to unlock your encrypted volume. For this reason, add a strong passphrase that allows you to unlock the encrypted volume manually even when a value in a PCR changes.
>
> If the system cannot automatically unlock your encrypted volume after an upgrade of the **shim-x64** package, follow the steps in the Clevis TPM2 no longer decrypts LUKS devices after a restart Red Hat Knowledgebase solution.

- For more information, see the **clevis-encrypt-tpm2(1)** and **clevis-luks-bind(1)** man pages on your system.

9.7. CONFIGURING NBDE CLIENTS FOR AUTOMATED UNLOCKING OF LUKS-ENCRYPTED VOLUMES

With the Clevis framework, you can configure clients for automated unlocking of LUKS-encrypted volumes when a selected Tang server is available. This creates an NBDE (Network-Bound Disk Encryption) deployment.

Prerequisites

- A Tang server is running and available.

Procedure

1. To automatically unlock an existing LUKS-encrypted volume, install the **clevis-luks** subpackage:

```
# dnf install clevis-luks
```

2. Identify the LUKS-encrypted volume for PBD. In the following example, the block device is referred as */dev/sda2*:

```
# lsblk
NAME                                         MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINT
sda                                            8:0    0    12G  0 disk
├─sda1                                         8:1    0     1G  0 part  /boot
└─sda2                                         8:2    0    11G  0 part
  └─luks-40e20552-2ade-4954-9d56-565aa7994fb6 253:0    0    11G  0 crypt
    ├─rhel-root                              253:0    0   9.8G  0 lvm   /
    └─rhel-swap                              253:1    0   1.2G  0 lvm   [SWAP]
```

3. Bind the volume to a Tang server using the **clevis luks bind** command:

```
# clevis luks bind -d /dev/sda2 tang '{"url":"http://tang.srv"}'
The advertisement contains the following signing keys:

_OsIk0T-E2l6qjfdDiwVmidoZjA

Do you wish to trust these keys? [ynYN] y
You are about to initialize a LUKS device for metadata storage.
Attempting to initialize it may result in data loss if data was
already written into the LUKS header gap in a different format.
A backup is advised before initialization is performed.

Do you wish to initialize /dev/sda2? [yn] y
Enter existing LUKS password:
```

This command performs four steps:

a. Creates a new key with the same entropy as the LUKS master key.

b. Encrypts the new key with Clevis.

c. Stores the Clevis JWE object in the LUKS2 header token or uses LUKSMeta if the non-default LUKS1 header is used.

d. Enables the new key for use with LUKS.

> **NOTE**
>
> The binding procedure assumes that the header contains at least one free LUKS password slot. The **clevis luks bind** command takes one of the slots.

Now, you can unlock the volume with your existing password as well as with the Clevis policy.

4. To enable the early boot system to process the disk binding, use the **dracut** tool on an already installed system. In RHEL, Clevis produces a generic **initrd** (initial RAM disk) without host-specific configuration options and does not automatically add parameters such as **rd.neednet=1** to the kernel command line. If your configuration relies on a Tang pin that requires network during early boot, use the **--hostonly-cmdline** argument and **dracut** adds **rd.neednet=1** when it detects a Tang binding:

   a. Install the **clevis-dracut** package:

   ```
   # dnf install clevis-dracut
   ```

   b. Regenerate the initial RAM disk:

   ```
   # dracut -fv --regenerate-all --hostonly-cmdline
   ```

   c. Alternatively, create a .conf file in the **/etc/dracut.conf.d/** directory, and add the **hostonly_cmdline=yes** option to the file. Then, you can use **dracut** without **--hostonly-cmdline**, for example:

   ```
   # echo "hostonly_cmdline=yes" > /etc/dracut.conf.d/clevis.conf
   # dracut -fv --regenerate-all
   ```

   d. If your scenario requires that networking for a Tang pin is available during early boot, use the **grubby** tool on the system where Clevis is installed:

   ```
   # grubby --update-kernel=ALL --args="rd.neednet=1"
   ```

Verification

1. Verify that the Clevis JWE object is successfully placed in a LUKS header:

   ```
   # clevis luks list -d /dev/sda2
   1: tang '{"url":"http://tang.srv:port"}'
   ```

2. Check that the bindings are available for the early boot, for example:

   ```
   # lsinitrd | grep clevis-luks
   lrwxrwxrwx   1 root     root           48 Jan  4 02:56
   etc/systemd/system/cryptsetup.target.wants/clevis-luks-askpass.path ->
   /usr/lib/systemd/system/clevis-luks-askpass.path
   …
   ```

Additional resources

- Network boot options
- Looking forward to Linux network configuration in the initial RAM disk (initrd) (Red Hat Enable Sysadmin)

9.8. CONFIGURING NBDE CLIENTS WITH STATIC IP CONFIGURATION

To use NBDE for clients with static IP configuration (without DHCP), you must pass your network configuration to the Dracut tool manually.

Prerequisites

- A Tang server is running and available.
- The NBDE client is configured for automated unlocking of encrypted volumes by the Tang server.
  For details, see Configuring NBDE clients for automated unlocking of LUKS-encrypted volumes .

Procedure

1. You can provide your static network configuration as a value for the **kernel-cmdline** option in a **dracut** command, for example:

```
# dracut -fv --regenerate-all --kernel-cmdline
"ip=192.0.2.10::192.0.2.1:255.255.255.0::ens3:none nameserver=192.0.2.100"
```

2. Alternatively, create a .conf file in the **/etc/dracut.conf.d/** directory with the static network information and then regenerate the initial RAM disk image:

```
# cat /etc/dracut.conf.d/static_ip.conf
kernel_cmdline="ip=192.0.2.10::192.0.2.1:255.255.255.0::ens3:none
nameserver=192.0.2.100"
# dracut -fv --regenerate-all
```

9.9. CONFIGURING MANUAL ENROLLMENT OF LUKS-ENCRYPTED VOLUMES BY USING A TPM 2.0 POLICY

Configure unlocking of LUKS-encrypted volumes by using a Trusted Platform Module 2.0 (TPM 2.0) policy.

The **clevis-encrypt-tpm2(1)** man page on your system provides a complete reference of available parameters.

Prerequisites

- An accessible TPM 2.0-compatible device.
- A system with the 64-bit Intel or 64-bit AMD architecture.

Procedure

1. Install the **clevis-luks** subpackage:

```
# dnf install clevis-luks
```

2. Identify the LUKS-encrypted volume for PBD. In the following example, the block device is referred as */dev/sda2*:

```
# lsblk
NAME                                          MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINT
sda                                             8:0    0    12G  0 disk
├─sda1                                          8:1    0     1G  0 part  /boot
└─sda2                                          8:2    0    11G  0 part
  └─luks-40e20552-2ade-4954-9d56-565aa7994fb6 253:0    0    11G  0 crypt
    ├─rhel-root                               253:0    0   9.8G  0 lvm   /
    └─rhel-swap                               253:1    0   1.2G  0 lvm   [SWAP]
```

3. Bind the volume to a TPM 2.0 device with the **clevis luks bind** command, for example:

```
# clevis luks bind -d /dev/sda2 tpm2 '{"hash":"sha256","key":"rsa"}'
...
Do you wish to initialize /dev/sda2? [yn] y
Enter existing LUKS password:
```

This command performs four steps:

a. Creates a new key with the same entropy as the LUKS master key.

b. Encrypts the new key with Clevis.

c. Stores the Clevis JWE object in the LUKS2 header token or uses LUKSMeta if the non-default LUKS1 header is used.

d. Enables the new key for use with LUKS.

NOTE

The binding procedure assumes that there is at least one free LUKS password slot. The **clevis luks bind** command takes one of the slots.

Alternatively, if you want to seal data to specific Platform Configuration Registers (PCR) states, add the **pcr_bank** and **pcr_ids** values to the **clevis luks bind** command, for example:

```
# clevis luks bind -d /dev/sda2 tpm2
'{"hash":"sha256","key":"rsa","pcr_bank":"sha256","pcr_ids":"0,1"}'
```

IMPORTANT

Because the data can only be unsealed if PCR hashes values match the policy used when sealing and the hashes can be rewritten, add a strong passphrase that enables you to unlock the encrypted volume manually when a value in a PCR changes.

If the system cannot automatically unlock your encrypted volume after an upgrade of the **shim-x64** package, follow the steps in the Clevis TPM2 no longer decrypts LUKS devices after a restart Red Hat Knowledgebase solution.

4. To enable the early boot system to process the disk binding, use the **dracut** command on an already installed system:

```
# dnf install clevis-dracut
# dracut -fv --regenerate-all
```

Verification

- Verify that the Clevis JWE object is successfully placed in a LUKS header:

```
# clevis luks list -d /dev/sda2
1: tpm2 '{"hash":"sha256","key":"rsa"}'
```

9.10. CONFIGURING UNLOCKING OF LUKS-ENCRYPTED VOLUMES BY USING A PKCS #11 PIN

You can configure unlocking of LUKS-encrypted volumes by using a device compatible with PKCS #11, which can be either a smart card or a hardware security module (HSM).

Automated unlocking of encrypted volumes with a Clevis PKCS #11 pin requires also changes in the **/etc/crypttab** file, which configure the **systemd** manager to use an **AF_UNIX** socket to wait for the key for unlocking the volumes instead of prompting the user through the console.

The Clevis PKCS #11 unit file configures the socket in the **/run/systemd/clevis-pkcs11.sock** file for sending and receiving the information about disk unlocking. For disks unlocked through a Clevis PKCS #11 pin, you must configure the socket file as a key file. See the **clevis-encrypt-pkcs11(1)** man page on your system for more information.

Prerequisites

- The PKCS #11 device is already configured and accessible.
- The **clevis-pin-pkcs11** package is installed.
- The volume has at least one free LUKS password slot for the **clevis luks bind** command.

Procedure

1. Identify the LUKS-encrypted volume for PBD. In the following example, the block device is referred as */dev/sda2*:

```
# lsblk
NAME                                           MAJ:MIN RM  SIZE RO TYPE  MOUNTPOINT
sda                                              8:0    0   12G  0 disk
├─sda1                                           8:1    0    1G  0 part  /boot
└─sda2                                           8:2    0   11G  0 part
  └─luks-40e20552-2ade-4954-9d56-565aa7994fb6 253:0    0   11G  0 crypt
    ├─rhel-root                                253:0    0  9.8G  0 lvm   /
    └─rhel-swap                                253:1    0  1.2G  0 lvm   [SWAP]
```

2. Identify the URI of the PKCS #11 device you want to use for unlocking volumes, for example:

```
$ pkcs11-tool -L | grep uri
```

```
uri            :
pkcs11:model=PKCS%2315%20emulated;manufacturer=piv_II;serial=42facd1f749ece7f;token=
clevis
uri            :
pkcs11:model=PKCS%2315%20emulated;manufacturer=OpenPGP%20project;serial=000f060
80f4f;token=OpenPGP%20card%20%28User%20PIN%29
```

3. Bind the volume to a PKCS #11 device with the **clevis luks bind** command, for example:

```
# clevis luks bind -d /dev/sda2 pkcs11
'{"uri":"pkcs11:model=PKCS%2315%20emulated;manufacturer=OpenPGP%20project;serial=0
00f06080f4f;token=OpenPGP%20card%20%28User%20PIN%29;id=%03;object=Authenticatio
n%20key;type=public"}'
…
Do you wish to initialize /dev/sda2? [yn] y
Enter existing LUKS password:
```

This command performs the following steps:

a. Creates a new key with the same entropy as the LUKS master key.

b. Encrypts the new key with Clevis.

c. Stores the Clevis JWE object in the LUKS2 header token or uses LUKSMeta if the non-default LUKS1 header is used.

d. Enables the new key for use with LUKS.

4. Optionally: If your scenario requires specifying the module to use, add the module-path URI parameter:

```
# clevis luks bind -d /dev/sda2 pkcs11 '{"uri":"pkcs11:module-
path=/usr/lib64/libykcs11.so.2";model=PKCS%2315%20emulated;manufacturer=OpenPGP%2
0project;serial=000f06080f4f;token=OpenPGP%20card%20%28User%20PIN%29;id=%03;obj
ect=Authentication%20key;type=public}'
```

5. Enable the **clevis-luks-pkcs11-askpass.socket** unit:

```
# systemctl enable --now clevis-luks-pkcs11-askpass.socket
```

6. Open the **/etc/crypttab** file in a text editor and identify the line containing the LUKS-encrypted volume you want to unlock by the PKCS #11 pin, for example:

```
luks-6e38d5e1-7f83-43cc-819a-7416bcbf9f84 UUID=6e38d5e1-7f83-43cc-819a-
7416bcbf9f84 - -
```

7. Replace the dashes with the **/run/systemd/clevis-pkcs11.sock** file path and the **keyfile-timeout** option:

```
luks-6e38d5e1-7f83-43cc-819a-7416bcbf9f84 UUID=6e38d5e1-7f83-43cc-819a-
7416bcbf9f84 /run/systemd/clevis-pkcs11.sock keyfile-timeout=30s
```

The **keyfile-timeout** option provides a fall-through mechanism for when an unlocking error occurs and the system requires entering the passphrase manually through the console.

8. Save the changes and exit the editor.

9. To enable the early boot system to process the disk binding, which is required for unlocking root file systems, use the **dracut** tool on an already installed system:

   ```
   # dracut -fv --regenerate-all
   ```

10. Restart the system.
    During the following boot process, the system prompts for the PKCS #11 device PIN and decrypts the corresponding configured encrypted disk only in case you enter the correct PIN.

Verification

1. Instead of manually testing the boot process, you can encrypt and decrypt a text message with the following command:

   ```
   # echo "top secret" | clevis encrypt pkcs11 '{"uri":"pkcs11:module-
   path=/usr/lib64/libykcs11.so.2?pin-value=<PIN>"}' | clevis decrypt
   ```

   Replace ***<PIN>*** with a PIN value. You must enter this PIN value to decrypt the message.

2. Verify that the Clevis JWE object is successfully placed in a LUKS header, for example:

   ```
   # clevis luks list -d /dev/sda2
   1: pkcs11 '{"uri": "pkcs11:model=PKCS%2315%20emulated;manufacturer=piv_II;
   serial=0a35ba26b062b9c5;token=clevis;id=%02;object=Encryption%20Key?
   module-path=/usr/lib64/libykcs11.so.2"}'
   ```

9.11. REMOVING A CLEVIS PIN FROM A LUKS-ENCRYPTED VOLUME

You can remove the metadata created by the **clevis luks bind** command and also wipe a key slot that contains a passphrase added by Clevis.

In RHEL, you can use a single command to remove both the metadata created by the binding step and wipe the key slot on a selected device. See the **clevis-luks-unbind(1)** man page for more information.

Prerequisites

- A LUKS-encrypted volume with a Clevis binding.

Procedure

- Remove a Clevis pin from a LUKS-encrypted volume, for example:

  ```
  # clevis luks unbind -d /dev/sda2 -s 1
  ```

Verification

- The output of the **clevis luks list** command is empty, for example:

  ```
  # clevis luks list -d /dev/sda2
  ```

Alternatively, you can check that an output of the **cryptsetup luksDump** command does not contain the **clevis** string.

9.12. CONFIGURING AUTOMATED ENROLLMENT OF LUKS-ENCRYPTED VOLUMES BY USING KICKSTART

You can configure an automated installation process that uses Clevis for the enrollment of LUKS-encrypted volumes.

You can use an analogous procedure when using a TPM 2.0 policy instead of a Tang server.

Procedure

1. Instruct Kickstart to partition the disk so that LUKS encryption is enabled with a temporary password for all mount points except **/boot**. The password is temporary for this step of the enrollment process.

   ```
   part /boot --fstype="xfs" --ondisk=vda --size=256
   part / --fstype="xfs" --ondisk=vda --grow --encrypted --passphrase=temppass
   ```

   Note that OSPP-compliant systems require a more complex configuration, for example:

   ```
   part /boot --fstype="xfs" --ondisk=vda --size=256
   part / --fstype="xfs" --ondisk=vda --size=2048 --encrypted --passphrase=temppass
   part /var --fstype="xfs" --ondisk=vda --size=1024 --encrypted --passphrase=temppass
   part /tmp --fstype="xfs" --ondisk=vda --size=1024 --encrypted --passphrase=temppass
   part /home --fstype="xfs" --ondisk=vda --size=2048 --grow --encrypted --
   passphrase=temppass
   part /var/log --fstype="xfs" --ondisk=vda --size=1024 --encrypted --passphrase=temppass
   part /var/log/audit --fstype="xfs" --ondisk=vda --size=1024 --encrypted --
   passphrase=temppass
   ```

2. Install the related Clevis packages by listing them in the **%packages** section:

   ```
   %packages
   clevis-dracut
   clevis-luks
   clevis-systemd
   %end
   ```

3. Optional: To ensure that you can unlock the encrypted volume manually when required, add a strong passphrase to the volume before you remove the temporary passphrase.

4. Call **clevis luks bind** to perform binding in the **%post** section. Afterward, remove the temporary password:

   ```
   %post
   clevis luks bind -y -k - -d /dev/vda2 \
   tang '{"url":"http://tang.srv"}' <<< "temppass"
   cryptsetup luksRemoveKey /dev/vda2 <<< "temppass"
   dracut -fv --regenerate-all
   %end
   ```

If your configuration relies on a Tang pin that requires network during early boot or you use NBDE clients with static IP configurations, you must modify the **dracut** command as described in Configuring NBDE clients with static IP configuration .

WARNING

The **cryptsetup luksRemoveKey** command prevents any further administration of a LUKS2 device. You can recover a removed master key by using the **dmsetup** command only for LUKS1 devices. See the **dmsetup(8)** man page on your system for more information.

Additional resources

- Automatically installing RHEL

9.13. CONFIGURING AUTOMATED UNLOCKING OF A LUKS-ENCRYPTED REMOVABLE STORAGE DEVICE

You can set up an automated unlocking process of a LUKS-encrypted USB storage device.

You can use an analogous procedure when using a TPM 2.0 policy instead of a Tang server.

Procedure

1. Install the **clevis-udisks2** package:

```
# dnf install clevis-udisks2
```

2. Restart the system, and then perform the binding step using the **clevis luks bind** command as described in Configuring NBDE clients for automated unlocking of LUKS-encrypted volumes , for example:

```
# clevis luks bind -d /dev/sdb1 tang '{"url":"http://tang.srv"}'
```

Verification

1. The LUKS-encrypted removable device can be now unlocked automatically in your GNOME desktop session.

2. The device bound to a Clevis policy can be also unlocked by the **clevis luks unlock** command, for example:

```
# clevis luks unlock -d /dev/sdb1
```

9.14. DEPLOYING HIGH-AVAILABILITY NBDE SYSTEMS

Tang provides two methods for building high-availability deployments of Network-bound Disc Encryption (NBDE): client redundancy and key sharing.

Client redundancy (recommended)

Clients should be configured with the ability to bind to multiple Tang servers. In this setup, each Tang server has its own keys and clients can decrypt by contacting a subset of these servers. Clevis already supports this workflow through its **sss** plugin. Use this method for a secure high-availability deployment.

Key sharing

For redundancy purposes, more than one instance of Tang can be deployed. To set up a second or any subsequent instance, install the **tang** packages and copy the key directory to the new host by using **rsync** over **SSH**. This method of sharing keys increases the risk of key compromise and requires additional automation infrastructure.

High-available NBDE that uses Shamir's Secret Sharing

Shamir's Secret Sharing (SSS) is a cryptographic scheme that divides a secret into several unique parts. To reconstruct the secret, a number of parts is required. The number is called the threshold and SSS is also referred to as a thresholding scheme.
Clevis provides an implementation of SSS. It creates a key and divides it into several pieces. Each piece is encrypted using another pin including even SSS recursively. Additionally, you define the threshold **t**. If an NBDE deployment decrypts at least **t** pieces, then it recovers the encryption key and the decryption process succeeds. When Clevis detects a smaller number of parts than specified in the threshold, it prints an error message.

Example 9.1. Redundancy with two Tang servers

The following command decrypts a LUKS-encrypted device when at least one of two Tang servers is available:

```
# clevis luks bind -d /dev/sda1 sss '{"t":1,"pins":{"tang":[{"url":"http://tang1.srv"},
{"url":"http://tang2.srv"}]}}'
```

The previous command used the following configuration scheme:

```
{
   "t":1,
   "pins":{
      "tang":[
         {
            "url":"http://tang1.srv"
         },
         {
            "url":"http://tang2.srv"
         }
      ]
   }
}
```

In this configuration, the SSS threshold **t** is set to **1** and the **clevis luks bind** command successfully reconstructs the secret if at least one from two listed **tang** servers is available.

Example 9.2. Shared secret on a Tang server and a TPM device

The following command successfully decrypts a LUKS-encrypted device when both the **tang** server and the **tpm2** device are available:

```
# clevis luks bind -d /dev/sda1 sss '{"t":2,"pins":{"tang":[{"url":"http://tang1.srv"}], "tpm2":
{"pcr_ids":"0,7"}}}'
```

The configuration scheme with the SSS threshold 't' set to '2' is now:

```
{
   "t":2,
   "pins":{
      "tang":[
         {
            "url":"http://tang1.srv"
         }
      ],
      "tpm2":{
         "pcr_ids":"0,7"
      }
   }
}
```

For details and more examples, see the **High Availability** section in the **tang(8)** man page, the **Shamir's Secret Sharing** section in the **clevis(1)** man page, and the **clevis-encrypt-sss(1)** man page on your system.

9.15. DEPLOYMENT OF VIRTUAL MACHINES IN A NBDE NETWORK

When you configure Network-bound Disc Encryption (NBDE) for virtual machines, use only secure practices in production environments.

The **clevis luks bind** command does not change the LUKS master key. This implies that if you create a LUKS-encrypted image for use in a virtual machine or cloud environment, all the instances that run this image share a master key. This is extremely insecure and should be avoided at all times.

This is not a limitation of Clevis but a design principle of LUKS. If your scenario requires having encrypted root volumes in a cloud, perform the installation process for each instance of Red Hat Enterprise Linux in the cloud as well. The images cannot be shared without also sharing a LUKS master key.

To deploy automated unlocking in a virtualized environment, use tools such as Lorax or **virt-install** together with a Kickstart file or another automated provisioning tool to ensure that each encrypted VM has a unique master key. See Configuring automated enrollment of LUKS-encrypted volumes by using Kickstart) for detailed instructions.

9.16. BUILDING AUTOMATICALLY-ENROLLABLE VM IMAGES FOR CLOUD ENVIRONMENTS BY USING NBDE

Deploying automatically-enrollable encrypted images in a cloud environment can provide a unique set of challenges. As in other virtualization environments, you should reduce the number of instances started from a single image to avoid sharing the LUKS master key.

The best practice is to create customized images that are not shared in any public repository and that provide a base for the deployment of a limited amount of instances. The exact number of instances to create should be defined by deployment's security policies and based on the risk tolerance associated with the LUKS master key attack vector.

To build LUKS-enabled automated deployments, systems such as Lorax or virt-install together with a Kickstart file should be used to ensure master key uniqueness during the image building process.

Cloud environments enable two Tang server deployment options:

- The Tang server can be deployed within the cloud environment itself. Deploying Tang natively in the cloud allows for easy deployment. However, given that it shares infrastructure with the data persistence layer of ciphertext of other systems, it might be possible for both the Tang server's private key and the Clevis metadata to be stored on the same physical disk. Access to this physical disk permits a full compromise of the ciphertext data.

- The Tang server can be deployed outside of the cloud on independent infrastructure with a VPN link between the two infrastructures.

IMPORTANT

Always maintain a physical separation between the location where the data is stored and the system where Tang is running. This separation between the cloud and the Tang server ensures that the Tang server's private key cannot be accidentally combined with the Clevis metadata. It also provides local control of the Tang server if the cloud infrastructure is at risk.

9.17. DEPLOYING TANG AS A CONTAINER

The **tang** container image provides Tang-server decryption capabilities for Clevis clients that run either in Red Hat OpenShift Container Platform (OCP) clusters or in separate virtual machines.

Prerequisites

- The **podman** package and its dependencies are installed on the system.

- You have logged in on the **registry.redhat.io** container catalog using the **podman login registry.redhat.io** command. See Red Hat Container Registry Authentication for more information.

- The Clevis client is installed on systems containing LUKS-encrypted volumes that you want to automatically unlock by using a Tang server.

Procedure

1. Pull the **tang** container image from the **registry.redhat.io** registry:

```
# podman pull registry.redhat.io/rhel10/tang
```

2. Run the container, specify its port, and specify the path to the Tang keys. The following example runs the **tang** container, specifies the port *7500*, and indicates a path to the Tang keys of the **/var/db/tang** directory:

```
# podman run -d -p 7500:7500 -v tang-keys:/var/db/tang --name tang
registry.redhat.io/rhel10/tang
```

Note that Tang uses port 80 by default but this might collide with other services such as the Apache HTTP server.

3. Optional: For increased security, rotate the Tang keys periodically. You can use the **tangd-rotate-keys** script, for example:

```
# podman run --rm -v tang-keys:/var/db/tang registry.redhat.io/rhel10/tang tangd-rotate-keys
-v -d /var/db/tang
Rotated key 'rZAMKAseaXBe0rcKXL1hCCIq-DY.jwk' -> .'rZAMKAseaXBe0rcKXL1hCCIq-
DY.jwk'
Rotated key 'x1AIpc6WmnCU-CabD8_4q18vDuw.jwk' -> .'x1AIpc6WmnCU-
CabD8_4q18vDuw.jwk'
Created new key GrMMX_WfdqomIU_4RyjpcdlXb0E.jwk
Created new key _dTTfn17sZZqVAp80u3ygFDHtjk.jwk
Keys rotated successfully.
```

Verification

- On a system that contains LUKS-encrypted volumes for automated unlocking by the presence of the Tang server, check that the Clevis client can encrypt and decrypt a plain text message using Tang:

```
# echo test | clevis encrypt tang '{"url":"http://localhost:7500"}' | clevis decrypt
The advertisement contains the following signing keys:

x1AIpc6WmnCU-CabD8_4q18vDuw

Do you wish to trust these keys? [ynYN] y
test
```

The previous example command shows the **test** string at the end of its output when a Tang server is available on the *localhost* URL and communicates through port *7500*.

9.18. CONFIGURING NBDE BY USING RHEL SYSTEM ROLES

You can use the **nbde_client** and **nbde_server** RHEL system roles for automated deployments of Policy-Based Decryption (PBD) solutions by using Clevis and Tang.

The **rhel-system-roles** package contains these system roles, the related examples, and the reference documentation.

9.18.1. Using the nbde_server RHEL system role for setting up multiple Tang servers

By using the **nbde_server** system role, you can deploy and manage a Tang server as part of an automated disk encryption solution.

This role supports the following features:

- Rotating Tang keys
- Deploying and backing up Tang keys

Prerequisites

- You have prepared the control node and the managed nodes .
- You are logged in to the control node as a user who can run playbooks on the managed nodes.
- The account you use to connect to the managed nodes has **sudo** permissions for these nodes.

Procedure

1. Create a playbook file, for example, **~/playbook.yml**, with the following content:

   ```
   ---
   - name: Deploy a Tang server
     hosts: tang.server.example.com
     tasks:
     - name: Install and configure periodic key rotation
       ansible.builtin.include_role:
           name: redhat.rhel_system_roles.nbde_server
       vars:
         nbde_server_rotate_keys: yes
         nbde_server_manage_firewall: true
         nbde_server_manage_selinux: true
   ```

   This example playbook ensures deploying of your Tang server and a key rotation.

   The settings specified in the example playbook include the following:

   nbde_server_manage_firewall: true

   Use the **firewall** system role to manage ports used by the **nbde_server** role.

   nbde_server_manage_selinux: true

   Use the **selinux** system role to manage ports used by the **nbde_server** role.
   For details about all variables used in the playbook, see the **/usr/share/ansible/roles/rhel-system-roles.nbde_server/README.md** file on the control node.

2. Validate the playbook syntax:

   ```
   $ ansible-playbook --syntax-check ~/playbook.yml
   ```

   Note that this command only validates the syntax and does not protect against a wrong but valid configuration.

3. Run the playbook:

   ```
   $ ansible-playbook ~/playbook.yml
   ```

Verification

- On your Network-Bound Disk Encryption (NBDE) client, verify that your Tang server works correctly by using the following command. The command must return the identical message you pass for encryption and decryption:

```
# ansible managed-node-01.example.com -m command -a 'echo test | clevis encrypt tang
'{"url":"<tang.server.example.com>"}' -y | clevis decrypt'
test
```

9.18.2. Setting up Clevis clients with DHCP by using the **nbde_client** RHEL system role

The **nbde_client** system role enables you to deploy multiple Clevis clients in an automated way.

This role supports binding a LUKS-encrypted volume to one or more Network-Bound (NBDE) servers - Tang servers. You can either preserve the existing volume encryption with a passphrase or remove the passphrase. After removing the passphrase, you can then unlock the volume only using NBDE. This is useful when a volume is initially encrypted using a temporary key or password that you should remove after you provision the system.

If you provide both a passphrase and a key file, the role uses what you have provided first. If it does not find any of these valid, it attempts to retrieve a passphrase from an existing binding.

Policy-Based Decryption (PBD) defines a binding as a mapping of a device to a slot. This means that you can have multiple bindings for the same device. The default slot is slot 1.

NOTE

The **nbde_client** system role supports only Tang bindings. Therefore, you cannot use it for TPM2 bindings.

Prerequisites

- You have prepared the control node and the managed nodes .
- You are logged in to the control node as a user who can run playbooks on the managed nodes.
- The account you use to connect to the managed nodes has **sudo** permissions for these nodes.
- A volume that is already encrypted by using LUKS.

Procedure

1. Create a playbook file, for example, **~/playbook.yml**, with the following content:

```
---
- name: Configure clients for unlocking of encrypted volumes by Tang servers
  hosts: managed-node-01.example.com
  tasks:
    - name: Create NBDE client bindings
      ansible.builtin.include_role:
        name: redhat.rhel_system_roles.nbde_client
      vars:
        nbde_client_bindings:
          - device: /dev/rhel/root
            encryption_key_src: /etc/luks/keyfile
            nbde_client_early_boot: true
            state: present
            servers:
              - http://server1.example.com
            - http://server2.example.com
        - device: /dev/rhel/swap
          encryption_key_src: /etc/luks/keyfile
          servers:
            - http://server1.example.com
            - http://server2.example.com
```

This example playbook configures Clevis clients for automated unlocking of two LUKS-encrypted volumes when at least one of two Tang servers is available.

The settings specified in the example playbook include the following:

state: present

The values of **state** indicate the configuration after you run the playbook. Use the **present** value for either creating a new binding or updating an existing one. Contrary to a **clevis luks bind** command, you can use **state: present** also for overwriting an existing binding in its device slot. The **absent** value removes a specified binding.

nbde_client_early_boot: true

The **nbde_client** role ensures that networking for a Tang pin is available during early boot by default. If your scenario requires to disable this feature, add the **nbde_client_early_boot: false** variable to your playbook.
For details about all variables used in the playbook, see the **/usr/share/ansible/roles/rhel-system-roles.nbde_client/README.md** file on the control node.

2. Validate the playbook syntax:

```
$ ansible-playbook --syntax-check ~/playbook.yml
```

Note that this command only validates the syntax and does not protect against a wrong but valid configuration.

3. Run the playbook:

```
$ ansible-playbook ~/playbook.yml
```

Verification

1. On your NBDE client, check that the encrypted volume that should be automatically unlocked by your Tang servers contain the corresponding information in its LUKS pins:

```
# ansible managed-node-01.example.com -m command -a 'clevis luks list -d /dev/rhel/root'
1: tang '{"url":"<http://server1.example.com/>"}'
2: tang '{"url":"<http://server2.example.com/>"}'
```

2. If you do not use the **nbde_client_early_boot: false** variable, verify that the bindings are available for the early boot, for example:

```
# ansible managed-node-01.example.com -m command -a 'lsinitrd | grep clevis-luks'
lrwxrwxrwx   1 root     root           48 Jan  4 02:56
etc/systemd/system/cryptsetup.target.wants/clevis-luks-askpass.path ->
/usr/lib/systemd/system/clevis-luks-askpass.path
…
```

9.18.3. Setting up static-IP Clevis clients by using the nbde_client RHEL system role

The **nbde_client** RHEL system role supports only scenarios with Dynamic Host Configuration Protocol (DHCP). On an Network-Bound Disk Encryption (NBDE) client with static IP configuration, you must pass your network configuration as a kernel boot parameter.

Typically, administrators want to reuse a playbook and not maintain individual playbooks for each host to which Ansible assigns static IP addresses during early boot. In this case, you can use variables in the playbook and provide the settings in an external file. As a result, you need only one playbook and one file with the settings.

Prerequisites

- You have prepared the control node and the managed nodes .
- You are logged in to the control node as a user who can run playbooks on the managed nodes.
- The account you use to connect to the managed nodes has **sudo** permissions for these nodes.
- A volume that is already encrypted by using LUKS.

Procedure

1. Create a file with the network settings of your hosts, for example, **static-ip-settings-clients.yml**, and add the values you want to dynamically assign to the hosts:

```
clients:
  managed-node-01.example.com:
    ip_v4: 192.0.2.1
    gateway_v4: 192.0.2.254
    netmask_v4: 255.255.255.0
    interface: enp1s0
  managed-node-02.example.com:
    ip_v4: 192.0.2.2
    gateway_v4: 192.0.2.254
    netmask_v4: 255.255.255.0
    interface: enp1s0
```

2. Create a playbook file, for example, **~/playbook.yml**, with the following content:

```
- name: Configure clients for unlocking of encrypted volumes by Tang servers
  hosts: managed-node-01.example.com,managed-node-02.example.com
  vars_files:
    - ~/static-ip-settings-clients.yml
  tasks:
    - name: Create NBDE client bindings
      ansible.builtin.include_role:
        name: redhat.rhel_system_roles.network
      vars:
        nbde_client_bindings:
          - device: /dev/rhel/root
            encryption_key_src: /etc/luks/keyfile
            servers:
              - http://server1.example.com
              - http://server2.example.com
          - device: /dev/rhel/swap
            encryption_key_src: /etc/luks/keyfile
            servers:
              - http://server1.example.com
              - http://server2.example.com

    - name: Configure a Clevis client with static IP address during early boot
      ansible.builtin.include_role:
        name: redhat.rhel_system_roles.bootloader
      vars:
        bootloader_settings:
          - kernel: ALL
            options:
              - name: ip
                value: "{{ clients[inventory_hostname]['ip_v4'] }}::{{ clients[inventory_hostname]['gateway_v4'] }}:{{ clients[inventory_hostname]['netmask_v4'] }}::{{ clients[inventory_hostname]['interface'] }}:none"
```

This playbook reads certain values dynamically for each host listed in the **~/static-ip-settings-clients.yml** file.

For details about all variables used in the playbook, see the **/usr/share/ansible/roles/rhel-system-roles.network/README.md** file on the control node.

3. Validate the playbook syntax:

```
$ ansible-playbook --syntax-check ~/playbook.yml
```

Note that this command only validates the syntax and does not protect against a wrong but valid configuration.

4. Run the playbook:

```
$ ansible-playbook ~/playbook.yml
```

Additional resources

- Looking forward to Linux network configuration in the initial ramdisk (initrd)

CHAPTER 10. BLOCKING AND ALLOWING APPLICATIONS BY USING FAPOLICYD

Use the **fapolicyd** framework to implement an application allowlist solution that helps increase system security. Setting and enforcing a policy that either allows or denies application execution based on a rule set efficiently prevents the execution of unknown and potentially malicious software.

10.1. THE FAPOLICYD FRAMEWORK STRUCTURE

The **fapolicyd** software framework controls the execution of applications based on a user-defined policy. This is one of the most efficient ways to prevent running untrusted and possibly malicious applications on the system.

10.1.1. Components and trust

The **fapolicyd** framework provides the following components:

- **fapolicyd** service
- **fapolicyd** command-line utilities
- **fapolicyd** RPM plugin
- **fapolicyd** rule language
- **fagenrules** script

The administrator can define **allow** and **deny** execution rules for any application with the option to audit based on a path, hash, MIME type, or trust.

The **fapolicyd** framework introduces the concept of trust. An application is trusted when the system package manager correctly installs it and therefore registered in the system RPM database. The **fapolicyd** daemon uses the RPM database as a list of trusted binaries and scripts.

The **fapolicyd** RPM plugin registers any system update that is handled by either the DNF or RPM package manager. The plugin notifies the **fapolicyd** daemon about changes in this database. Other ways of adding applications require the creation of custom rules and restarting the **fapolicyd** service.

For more information, see the **fapolicyd**-related man pages listed by using the **man -k fapolicyd** command on your system.

10.1.2. Configuration files and directories

The **fapolicyd** service configuration is located in the **/etc/fapolicyd/** directory with the following structure:

- The **/etc/fapolicyd/fapolicyd.trust** file contains a list of trusted files. You can also use multiple trust files in the **/etc/fapolicyd/trust.d/** directory.
- The **/etc/fapolicyd/rules.d/** directory contains files with **allow** and **deny** execution rules. The **fagenrules** script merges these component rules files to the **/etc/fapolicyd/compiled.rules** file.

- The **fapolicyd.conf** file contains the daemon's configuration options. This file is useful primarily for performance-tuning purposes.

10.1.3. Rules

Rules in **/etc/fapolicyd/rules.d/** are organized in several files, each representing a different policy goal. The numbers at the beginning of the corresponding file names determine the order in **/etc/fapolicyd/compiled.rules**:

10

Language rules.

20

Dracut-related Rules.

21

Rules for updaters.

30

Patterns.

40

ELF rules.

41

Shared objects rules.

42

Trusted ELF rules.

70

Trusted language rules.

72

Shell rules.

90

Deny execute rules.

95

Allow open rules.

For more information and examples, see the documentation installed with the **fapolicyd** package in the **/usr/share/doc/fapolicyd/** directory, the **/usr/share/fapolicyd/sample-rules/README-rules** file, and the **fapolicyd.rules(5)** and **fagenrules(8)** man pages on your system.

10.1.4. Integrity checking

You can use one of the following ways for **fapolicyd** integrity checking:

- File-size checking
- Comparing SHA-256 hashes
- Integrity Measurement Architecture (IMA) subsystem

By default, **fapolicyd** does no integrity checking. Integrity checking based on the file size is fast, but an attacker can replace the content of the file and preserve its byte size. Computing and checking SHA-

256 checksums is more secure, but it affects the performance of the system. The **integrity = ima** option in **fapolicyd.conf** requires support for files' extended attributes (also known as ***xattr***) on all file systems containing executable files.

Additional resources

- Enhancing security with the kernel integrity subsystem

10.2. DEPLOYING FAPOLICYD

When deploying the **fapolicyd** application allowlisting framework, you can either try your configuration in permissive mode first or directly enable the service in the default configuration.

Procedure

1. Install the **fapolicyd** package:

   ```
   # dnf install fapolicyd
   ```

2. Set the Audit subsystem for recording **fapolicyd** events:

   ```
   # auditctl -w /etc/fapolicyd/ -p wa -k fapolicyd_changes
   # service try-restart auditd
   ```

3. Optional: To try your configuration first, change mode to permissive.

   a. Open the **/etc/fapolicyd/fapolicyd.conf** file in a text editor of your choice, for example:

   ```
   # vi /etc/fapolicyd/fapolicyd.conf
   ```

   b. Change the value of the **permissive** option from **0** to **1**, save the file, and exit the editor:

   ```
   permissive = 1
   ```

   Alternatively, you can debug your configuration by using the **fapolicyd --debug-deny --permissive** command before you start the service. See the Troubleshooting problems related to fapolicyd section for more information.

4. Enable and start the **fapolicyd** service:

   ```
   # systemctl enable --now fapolicyd
   ```

5. If you enabled permissive mode through **/etc/fapolicyd/fapolicyd.conf**:

   a. Use your applications.

   b. Check Audit logs for **fanotify** denials, for example:

   ```
   # ausearch -ts recent -m fanotify
   ```

   c. When debugged, disable permissive mode by changing the corresponding value back to **permissive = 0**, and restart the service:

```
# systemctl restart fapolicyd
```

Verification

1. Verify that the **fapolicyd** service is running correctly:

```
# systemctl status fapolicyd
● fapolicyd.service - File Access Policy Daemon
    Loaded: loaded (/usr/lib/systemd/system/fapolicyd.service; enabled; preset: disabled)
    Active: active (running) since Tue 2024-10-08 05:53:50 EDT; 11s ago
…
Oct 08 05:53:51 machine1.example.com fapolicyd[4974]: Loading trust data from rpmdb
backend
Oct 08 05:53:51 machine1.example.com fapolicyd[4974]: Loading trust data from file
backend
Oct 08 05:53:51 machine1.example.com fapolicyd[4974]: Starting to listen for events
```

2. Log in as a user without root privileges, and check that **fapolicyd** is working, for example:

```
$ cp /bin/ls /tmp
$ /tmp/ls
bash: /tmp/ls: Operation not permitted
```

Additional resources

- No FANOTIFY event seen in the audit log despite fapolicyd blocking operations (Red Hat Knowledgebase)

10.3. MARKING FILES AS TRUSTED USING AN ADDITIONAL SOURCE OF TRUST

The **fapolicyd** framework trusts files contained in the RPM database. You can mark additional files as trusted by modifying sources of trust.

You can modify the **/etc/fapolicyd/fapolicyd.trust** plain text file or files in the **/etc/fapolicyd/trust.d** directory, either directly using a text editor or through **fapolicyd-cli** commands. See the **fapolicyd.trust(13)** and **fapolicyd-cli(8)** man pages on your system for more details.

NOTE

For performance reasons, mark files as trusted using **fapolicyd.trust** or **trust.d/** rather than write custom **fapolicyd** rules.

Prerequisites

- The **fapolicyd** framework is deployed on your system.

Procedure

1. Copy your custom binary to the required directory, for example:

```
$ cp /bin/ls /tmp
$ /tmp/ls
bash: /tmp/ls: Operation not permitted
```

2. Mark your custom binary as trusted, and store the corresponding entry to the **myapp** file in **/etc/fapolicyd/trust.d/**:

   ```
   # fapolicyd-cli --file add /tmp/ls --trust-file myapp
   ```

   - If you skip the **--trust-file** option, then the previous command adds the corresponding line to **/etc/fapolicyd/fapolicyd.trust**.
   - To mark all existing files in a directory as trusted, provide the directory path as an argument of the **--file** option, for example:

     ```
     # fapolicyd-cli --file add /tmp/my_bin_dir/ --trust-file myapp
     ```

3. Update the **fapolicyd** database:

   ```
   # fapolicyd-cli --update
   ```

NOTE

Changing the content of a trusted file or directory changes its checksum, and therefore,**fapolicyd** no longer considers it trusted.

To restore trust in the new content, refresh the file trust database by using the **fapolicyd-cli --file update** command. If you do not provide any argument, the entire database refreshes. Alternatively, you can specify a path to a specific file or directory. Then, update the database by using **fapolicyd-cli --update**.

Verification

1. Check that you can execute your custom binary, for example:

   ```
   $ /tmp/ls
   ls
   ```

10.4. ADDING CUSTOM ALLOW AND DENY RULES FOR FAPOLICYD

The default set of **fapolicyd** rules does not affect system functions. For custom scenarios, such as storing binaries and scripts in a non-standard directory or adding applications without the DNF or RPM installers, you must either mark additional files as trusted or add new custom rules.

For basic scenarios, see Marking files as trusted using an additional source of trust . In more advanced scenarios such as allowing to execute a custom binary only for specific user and group identifiers, add new custom rules to the **/etc/fapolicyd/rules.d/** directory.

The following steps demonstrate adding a new rule to allow a custom binary.

For more information and examples, see the documentation installed with the **fapolicyd** package in the **/usr/share/doc/fapolicyd/** directory, the **/usr/share/fapolicyd/sample-rules/README-rules** file, and the **fapolicyd.rules(5)** and **fagenrules(8)** man pages on your system.

Prerequisites

- The **fapolicyd** framework is deployed on your system.

Procedure

1. Copy your custom binary to the required directory, for example:

```
$ cp /bin/ls /tmp
$ /tmp/ls
bash: /tmp/ls: Operation not permitted
```

2. Stop the **fapolicyd** service:

```
# systemctl stop fapolicyd
```

3. Use debug mode to identify a corresponding rule. Because the output of the **fapolicyd --debug** command is verbose and you can stop it only by pressing **Ctrl**+**C** or killing the corresponding process, redirect the error output to a file. In this case, you can limit the output only to access denials by using the **--debug-deny** option instead of **--debug**:

```
# fapolicyd --debug-deny 2> fapolicy.output &
[1] 51341
```

Alternatively, you can run **fapolicyd** debug mode in another terminal.

4. Repeat the command that **fapolicyd** denied:

```
$ /tmp/ls
bash: /tmp/ls: Operation not permitted
```

5. Stop debug mode by resuming it in the foreground and pressing **Ctrl**+**C**:

```
# fg
fapolicyd --debug 2> fapolicy.output
^C
…
```

Alternatively, kill the process of **fapolicyd** debug mode:

```
# kill 51341
```

6. Find a rule that denies the execution of your application:

```
# cat fapolicy.output | grep 'deny_audit'
…
rule=13 dec=deny_audit perm=execute auid=0 pid=6855 exe=/usr/bin/bash : path=/tmp/ls
ftype=application/x-executable trust=0
```

7. Locate the file that contains a rule that denies the execution of your custom binary. In this case, the **deny_audit perm=execute** rule belongs to the **90-deny-execute.rules** file:

```
# ls /etc/fapolicyd/rules.d/
10-languages.rules  40-bad-elf.rules     72-shell.rules
20-dracut.rules     41-shared-obj.rules  90-deny-execute.rules
21-updaters.rules   42-trusted-elf.rules 95-allow-open.rules
30-patterns.rules   70-trusted-lang.rules


# cat /etc/fapolicyd/rules.d/90-deny-execute.rules
# Deny execution for anything untrusted

deny_audit perm=execute all : all
```

8. Add a new **allow** rule to a file that lexically *precedes* the rule file that contains the rule that denied the execution of your custom binary in the **/etc/fapolicyd/rules.d/** directory.

   a. Create the rule file and open it in a text editor of your choice, for example:

   ```
   # touch /etc/fapolicyd/rules.d/80-myapps.rules
   # vi /etc/fapolicyd/rules.d/80-myapps.rules
   ```

   b. Insert the following rule to the **80-myapps.rules** file:

   ```
   allow perm=execute exe=/usr/bin/bash trust=1 : path=/tmp/ls ftype=application/x-executable trust=0
   ```

   Alternatively, you can allow executions of all binaries in the **/tmp** directory by adding the following rule to the rule file in **/etc/fapolicyd/rules.d/**:

   ```
   allow perm=execute exe=/usr/bin/bash trust=1 : dir=/tmp/ trust=0
   ```

   IMPORTANT

   To make a rule effective recursively on all directories under the specified directory, add a trailing slash to the value of the **dir=** parameter in the rule (/**tmp**/ in the previous example).

9. Prevent changes in the content of your custom binary

   a. Define the required rule using an SHA-256 checksum:

   ```
   $ sha256sum /tmp/ls
   780b75c90b2d41ea41679fcb358c892b1251b68d1927c80fbc0d9d148b25e836  ls
   ```

   b. Change the rule to the following definition:

   ```
   allow perm=execute exe=/usr/bin/bash trust=1 : sha256hash=780b75c90b2d41ea41679fcb358c892b1251b68d1927c80fbc0d9d148b25e836
   ```

10. Check that the list of compiled rules differs from the rule set in **/etc/fapolicyd/rules.d/**:

```
# fagenrules --check
/usr/sbin/fagenrules: Rules have changed and should be updated
```

11. Update the list, which is stored in the **/etc/fapolicyd/compiled.rules** file:

    ```
    # fagenrules --load
    ```

12. Check that your custom rule is in the list of **fapolicyd** rules before the rule that denied the execution:

    ```
    # fapolicyd-cli --list
    ...
    13. allow perm=execute exe=/usr/bin/bash trust=1 : path=/tmp/ls ftype=application/x-executable trust=0
    14. deny_audit perm=execute all : all
    …
    ```

13. Start the **fapolicyd** service:

    ```
    # systemctl start fapolicyd
    ```

Verification

1. Check that your custom binary can be now executed, for example:

    ```
    $ /tmp/ls
    ls
    ```

10.5. ENABLING FAPOLICYD INTEGRITY CHECKS

By default, **fapolicyd** does not perform integrity checking. You can configure **fapolicyd** to perform integrity checks by comparing either file sizes or SHA-256 hashes. You can also set integrity checks by using the Integrity Measurement Architecture (IMA) subsystem.

Prerequisites

- The **fapolicyd** framework is deployed on your system.

Procedure

1. Open the **/etc/fapolicyd/fapolicyd.conf** file in a text editor of your choice, for example:

    ```
    # vi /etc/fapolicyd/fapolicyd.conf
    ```

2. Change the value of the **integrity** option from **none** to **sha256**, save the file, and exit the editor:

    ```
    integrity = sha256
    ```

3. Restart the **fapolicyd** service:

    ```
    # systemctl restart fapolicyd
    ```

Verification

1. Back up the file used for the verification:

   ```
   # cp /bin/more /bin/more.bak
   ```

2. Change the content of the **/bin/more** binary:

   ```
   # cat /bin/less > /bin/more
   ```

3. Attempt to use the changed binary as a regular user and verify **fapolicyd** denies the operation:

   ```
   # su example.user
   $ /bin/more /etc/redhat-release
   bash: /bin/more: Operation not permitted
   ```

4. Revert the changes:

   ```
   # mv -f /bin/more.bak /bin/more
   ```

10.6. TROUBLESHOOTING PROBLEMS RELATED TO FAPOLICYD

The **fapolicyd** application framework provides tools for troubleshooting the most common problems and you can also add applications installed with the **rpm** command to the trust database.

Installing applications by using RPM

If you install an application by using the **rpm** command, you have to perform a manual refresh of the **fapolicyd** RPM database:

1. Install your *<application>*:

   ```
   # rpm -i <application>.rpm
   ```

2. Refresh the database:

   ```
   # fapolicyd-cli --update
   ```

   If you skip this step, the system can freeze and must be restarted.

Service status

If **fapolicyd** does not work correctly, check the service status:

```
# systemctl status fapolicyd
```

fapolicyd-cli checks and listings

The **--check-config**, **--check-watch_fs**, and **--check-trustdb** options help you find syntax errors, not-yet-watched file systems, and file mismatches, for example:

```
# fapolicyd-cli --check-config
Daemon config is OK
```

```
# fapolicyd-cli --check-trustdb
/etc/selinux/targeted/contexts/files/file_contexts miscompares: size sha256
/etc/selinux/targeted/policy/policy.31 miscompares: size sha256
```

- Use the **--list** option to check the current list of rules and their order:

```
# fapolicyd-cli --list
…
9. allow perm=execute all : trust=1
10. allow perm=open all : ftype=%languages trust=1
11. deny_audit perm=any all : ftype=%languages
12. allow perm=any all : ftype=text/x-shellscript
13. deny_audit perm=execute all : all
…
```

Debug mode

Debug mode provides detailed information about matched rules, database status, and more. To switch **fapolicyd** to debug mode:

1. Stop the **fapolicyd** service:

```
# systemctl stop fapolicyd
```

2. Use debug mode to identify a corresponding rule:

```
# fapolicyd --debug
```

Because the output of the **fapolicyd --debug** command is verbose, you can redirect the error output to a file:

```
# fapolicyd --debug 2> fapolicy.output
```

Alternatively, to limit the output only to entries when **fapolicyd** denies access, use the **--debug-deny** option:

```
# fapolicyd --debug-deny
```

Removing the **fapolicyd** database

To solve problems related to the **fapolicyd** database, try to remove the database file:

```
# systemctl stop fapolicyd
# fapolicyd-cli --delete-db
```

WARNING

Do not remove the **/var/lib/fapolicyd/** directory. The **fapolicyd** framework automatically restores only the database file in this directory.

Dumping the **fapolicyd** database

The **fapolicyd** contains entries from all enabled trust sources. You can check the entries after dumping the database:

```
# fapolicyd-cli --dump-db
```

Application pipe

- In rare cases, removing the **fapolicyd** pipe file can solve a lockup:

  ```
  # rm -f /var/run/fapolicyd/fapolicyd.fifo
  ```

10.7. PREVENTING USERS FROM EXECUTING UNTRUSTWORTHY CODE BY USING THE FAPOLICYD RHEL SYSTEM ROLE

You can automate the installation and configuration of the **fapolicyd** service by using the **fapolicyd** RHEL system role.

With this role, you can remotely configure the service to allow users to execute only trusted applications, for example, the ones which are listed in the RPM database and in an allow list. Additionally, the service can perform integrity checks before it executes an allowed application.

Prerequisites

- You have prepared the control node and the managed nodes .
- You are logged in to the control node as a user who can run playbooks on the managed nodes.
- The account you use to connect to the managed nodes has **sudo** permissions for these nodes.

Procedure

1. Create a playbook file, for example, **~/playbook.yml**, with the following content:

   ```
   ---
   - name: Configuring fapolicyd
     hosts: managed-node-01.example.com
     tasks:
       - name: Allow only executables installed from RPM database and specific files
         ansible.builtin.include_role:
           name: redhat.rhel_system_roles.fapolicyd
         vars:
           fapolicyd_setup_permissive: false
           fapolicyd_setup_integrity: sha256
           fapolicyd_setup_trust: rpmdb,file
           fapolicyd_add_trusted_file:
             - <path_to_allowed_command>
             - <path_to_allowed_service>
   ```

   The settings specified in the example playbook include the following:

   fapolicyd_setup_permissive: *<true|false>*

Enables or disables sending policy decisions to the kernel for enforcement. Set this variable for debugging and testing purposes to **false**.

fapolicyd_setup_integrity: *<type_type>*

Defines the integrity checking method. You can set one of the following values:

- **none** (default): Disables integrity checking.
- **size**: The service compares only the file sizes of allowed applications.
- **ima**: The service checks the SHA-256 hash that the kernel's Integrity Measurement Architecture (IMA) stored in a file's extended attribute. Additionally, the service performs a size check. Note that the role does not configure the IMA kernel subsystem. To use this option, you must manually configure the IMA subsystem.
- **sha256**: The service compares the SHA-256 hash of allowed applications.

fapolicyd_setup_trust: *<trust_backends>*

Defines the list of trust backends. If you include the **file** backend, specify the allowed executable files in the **fapolicyd_add_trusted_file** list.

For details about all variables used in the playbook, see the **/usr/share/ansible/roles/rhel-system-roles.fapolicyd.README.md** file on the control node.

2. Validate the playbook syntax:

```
$ ansible-playbook ~/playbook.yml --syntax-check
```

Note that this command only validates the syntax and does not protect against a wrong but valid configuration.

3. Run the playbook:

```
$ ansible-playbook ~/playbook.yml
```

Verification

- Execute a binary application that is not on the allow list as a user:

```
$ ansible managed-node-01.example.com -m command -a 'su -c
"/bin/not_authorized_application " <user_name>'
bash: line 1: /bin/not_authorized_application: Operation not permitted non-zero return code
```

CHAPTER 11. PROTECTING SYSTEMS AGAINST INTRUSIVE USB DEVICES

Use USBGuard to protect your Red Hat Enterprise Linux system against intrusive USB devices. This framework helps prevent attacks, such as those involving malware or trojans loaded onto USB drives.

11.1. USBGUARD INTRUSION PREVENTION

With the USBGuard software framework, you can protect your systems against intrusive USB devices by using basic lists of permitted and forbidden devices based on the USB device authorization feature in the kernel.

The USBGuard framework provides the following components:

- The system service component with an inter-process communication (IPC) interface for dynamic interaction and policy enforcement
- The command-line interface to interact with a running **usbguard** system service
- The rule language for writing USB device authorization policies
- The C++ API for interacting with the system service component implemented in a shared library

The **usbguard** system service configuration file (**/etc/usbguard/usbguard-daemon.conf**) includes the options to authorize the users and groups to use the IPC interface.

IMPORTANT

The system service provides the USBGuard public IPC interface. In Red Hat Enterprise Linux, the access to this interface is limited to only the root user by default.

Consider setting either the **IPCAccessControlFiles** option (recommended) or the **IPCAllowedUsers** and **IPCAllowedGroups** options to limit access to the IPC interface.

Ensure that you do not leave the Access Control List (ACL) unconfigured because this exposes the IPC interface to all local users and allows them to manipulate the authorization state of USB devices and modify the USBGuard policy.

11.2. INSTALLING USBGUARD

Install the USBGuard framework to begin defining policies that govern which USB devices are allowed to connect to your Red Hat Enterprise Linux system. This is the first step in defending against physical security breaches.

Procedure

1. Install the **usbguard** package:

```
# dnf install usbguard
```

2. Create an initial rule set:

```
# usbguard generate-policy > /etc/usbguard/rules.conf
```

3. Start the **usbguard** daemon and ensure that it starts automatically on boot:

```
# systemctl enable --now usbguard
```

Verification

1. Verify that the **usbguard** service is running:

```
# systemctl status usbguard
● usbguard.service - USBGuard daemon
   Loaded: loaded (/usr/lib/systemd/system/usbguard.service; enabled; vendor preset:
disabled)
   Active: active (running) since Thu 2019-11-07 09:44:07 CET; 3min 16s ago
     Docs: man:usbguard-daemon(8)
 Main PID: 6122 (usbguard-daemon)
    Tasks: 3 (limit: 11493)
   Memory: 1.2M
   CGroup: /system.slice/usbguard.service
           └─6122 /usr/sbin/usbguard-daemon -f -s -c /etc/usbguard/usbguard-daemon.conf

Nov 07 09:44:06 localhost.localdomain systemd[1]: Starting USBGuard daemon...
Nov 07 09:44:07 localhost.localdomain systemd[1]: Started USBGuard daemon.
```

2. List USB devices recognized by USBGuard:

```
# usbguard list-devices
4: allow id 1d6b:0002 serial "0000:02:00.0" name "xHCI Host Controller" hash...
```

11.3. BLOCKING AND AUTHORIZING A USB DEVICE BY USING THE CLI

You can set USBGuard to allow, block, or reject a specific USB device by using the **usbguard** command in your terminal. This setting persists while USBGuard is running.

USBGuard uses the terms ***block*** and ***reject*** with the following meanings:

block

Do not interact with this device for now.

reject

Ignore this device as if it does not exist.

For more information, see the built-in help listed by using the **usbguard --help** command and the **usbguard(1)** man page on your system.

Prerequisites

- The **usbguard** service is installed and running.

Procedure

1. Determine the ID of the USB device by listing the devices recognized by USBGuard:

```
# usbguard list-devices
```

```
1: allow id 1d6b:0002 serial "0000:00:06.7" name "EHCI Host Controller" hash
"JDOb0BiktYs2ct3mSQKopnOOV2h9MGYADwhT+oUtF2s=" parent-hash
"4PHGcaDKWtPjKDwYpIRG722cB9SlGz9l9Iea93+Gt9c=" via-port "usb1" with-interface
09:00:00
...
6: block id 1b1c:1ab1 serial "000024937962" name "Voyager" hash
"CrXgiaWIf2bZAU+5WkzOE7y0rdSO82XMzubn7HDb95Q=" parent-hash
"JDOb0BiktYs2ct3mSQKopnOOV2h9MGYADwhT+oUtF2s=" via-port "1-3" with-interface
08:06:50
```

2. Authorize a device to interact with the system:

   ```
   # usbguard allow-device <ID>
   ```

3. Deauthorize and remove a device:

   ```
   # usbguard reject-device <ID>
   ```

4. Deauthorize and retain a device:

   ```
   # usbguard block-device <ID>
   ```

11.4. PERMANENTLY BLOCKING AND AUTHORIZING A USB DEVICE

You can permanently block and authorize a USB device by using the **-p** option. This adds a device-specific rule to the current policy and persists across restarts and reboots.

USBGuard uses the terms ***block*** and ***reject*** with the following meanings:

block

Do not interact with this device for now.

reject

Ignore this device as if it does not exist.

Prerequisites

- The **usbguard** service is installed and running.

Procedure

1. Configure SELinux to allow the **usbguard** daemon to write rules.

   a. Display the **semanage** Booleans relevant to **usbguard**.

   ```
   # semanage boolean -l | grep usbguard
   usbguard_daemon_write_conf     (off  ,  off)  Allow usbguard to daemon write conf
   usbguard_daemon_write_rules    (on   ,   on)  Allow usbguard to daemon write rules
   ```

   b. If the **usbguard_daemon_write_rules** Boolean is turned off, turn it on.

   ```
   # semanage boolean -m --on usbguard_daemon_write_rules
   ```

2. Determine the ID of the USB device by listing the devices recognized by USBGuard:

   ```
   # usbguard list-devices
   1: allow id 1d6b:0002 serial "0000:00:06.7" name "EHCI Host Controller" hash
   "JDOb0BiktYs2ct3mSQKopnOOV2h9MGYADwhT+oUtF2s=" parent-hash
   "4PHGcaDKWtPjKDwYpIRG722cB9SlGz9l9Iea93+Gt9c=" via-port "usb1" with-interface
   09:00:00
   ...
   6: block id 1b1c:1ab1 serial "000024937962" name "Voyager" hash
   "CrXgiaWIf2bZAU+5WkzOE7y0rdSO82XMzubn7HDb95Q=" parent-hash
   "JDOb0BiktYs2ct3mSQKopnOOV2h9MGYADwhT+oUtF2s=" via-port "1-3" with-interface
   08:06:50
   ```

3. Permanently authorize a device to interact with the system:

   ```
   # usbguard allow-device <ID> -p
   ```

4. Permanently deauthorize and remove a device:

   ```
   # usbguard reject-device <ID> -p
   ```

5. Permanently deauthorize and retain a device:

   ```
   # usbguard block-device <ID> -p
   ```

Verification

1. Check that the USBGuard rules include the changes you made.

   ```
   # usbguard list-rules
   ```

11.5. CREATING A CUSTOM POLICY FOR USB DEVICES

Create a custom USBGuard policy to meet your specific security requirements. By defining multiple rules, you gain fine-grained control over which USB devices are authorized to interact with your system.

Prerequisites

- The **usbguard** service is installed and running.
- The **/etc/usbguard/rules.conf** file contains an initial rule set generated by the **usbguard generate-policy** command.

Procedure

1. Create a policy which authorizes the currently connected USB devices, and store the generated rules to the **rules.conf** file:

   ```
   # usbguard generate-policy --no-hashes > ./rules.conf
   ```

   The **--no-hashes** option does not generate hash attributes for devices. Avoid hash attributes in your configuration settings because they might not be persistent.

2. In the **rules.conf** file, add, remove, or edit the rules as required by using a text editor. For example, the following rule allows only devices with a single mass storage interface to interact with the system:

   ```
   allow with-interface equals { 08:*:* }
   ```

   See the **usbguard-rules.conf(5)** man page for a detailed rule-language description and more examples.

3. Install the updated policy:

   ```
   # install -m 0600 -o root -g root rules.conf /etc/usbguard/rules.conf
   ```

4. Restart the **usbguard** daemon to apply your changes:

   ```
   # systemctl restart usbguard
   ```

Verification

1. Check that your custom rules are in the active policy, for example:

   ```
   # usbguard list-rules
   …
   4: allow with-interface 08:*:*
   …
   ```

11.6. CREATING A STRUCTURED CUSTOM POLICY FOR USB DEVICES

Create a structured custom USBGuard policy in RHEL to manage complex security scenarios. By organizing rules into separate configuration files, you ensure easier maintenance and comprehensive security coverage.

You can organize your custom USBGuard policy in several **.conf** files within the **/etc/usbguard/rules.d/** directory. The **usbguard-daemon** then combines the main **rules.conf** file with the **.conf** files within the directory in alphabetical order.

Prerequisites

- The **usbguard** service is installed and running.

Procedure

1. Create a policy which authorizes the currently connected USB devices, and store the generated rules to a new **.conf** file, for example, ***<policy.conf>***.

   ```
   # usbguard generate-policy --no-hashes > ./<policy.conf>
   ```

   The **--no-hashes** option does not generate hash attributes for devices. Avoid hash attributes in your configuration settings because they might not be persistent.

2. Open the ***<policy.conf>*** file with a text editor of your choice, and select the lines with the rules that you want to record, for example:

```
…
allow id 04f2:0833 serial "" name "USB Keyboard" via-port "7-2" with-interface { 03:01:01
03:00:00 } with-connect-type "unknown"
…
```

3. Copy the selected lines into a separate **.conf** file.

   NOTE

   The two digits at the beginning of the file name specify the order in which the daemon reads the configuration files.

   For example, to copy the rules for your keyboards into a new **.conf** file:

   ```
   # grep "USB Keyboard" ./<policy.conf> > ./<10keyboards.conf>
   ```

4. Install the new policy to the **/etc/usbguard/rules.d/** directory.

   ```
   # install -m 0600 -o root -g root <10keyboards.conf>
   /etc/usbguard/rules.d/<10keyboards.conf>
   ```

5. Move the rest of the lines to the main **rules.conf** file.

   ```
   # grep -v "USB Keyboard" ./policy.conf > ./rules.conf
   ```

6. Install the remaining rules.

   ```
   # install -m 0600 -o root -g root rules.conf /etc/usbguard/rules.conf
   ```

7. Restart the **usbguard** daemon to apply your changes.

   ```
   # systemctl restart usbguard
   ```

Verification

1. Display all active USBGuard rules.

   ```
   # usbguard list-rules
   …
   15: allow id 04f2:0833 serial "" name "USB Keyboard" hash
   "kxM/iddRe/WSCocgiuQlVs6Dn0VEza7KiHoDeTz0fyg=" parent-hash
   "2i6ZBJfTl5BakXF7Gba84/Cp1gslnNc1DM6vWQpie3s=" via-port "7-2" with-interface {
   03:01:01 03:00:00 } with-connect-type "unknown"
   …
   ```

2. Display the contents of the **rules.conf** file and all the **.conf** files in the **/etc/usbguard/rules.d/** directory.

   ```
   # cat /etc/usbguard/rules.conf /etc/usbguard/rules.d/*.conf
   ```

3. Verify that the active rules contain all the rules from the files and are in the correct order.

11.7. AUTHORIZING USERS AND GROUPS TO USE THE USBGUARD IPC INTERFACE

Authorize specific users or groups to use the USBGuard public IPC interface in RHEL to allow non-root management of USB devices. Grant access by using the **usbguard add-user** subcommand or editing the configuration file to prevent unauthorized manipulation of USB device authorization policies.

Prerequisites

- The **usbguard** service is installed and running.
- The **/etc/usbguard/rules.conf** file contains an initial rule set generated by the **usbguard generate-policy** command.

Procedure

1. Edit the **/etc/usbguard/usbguard-daemon.conf** file with the rules you want to add. For example, to allow all users in the **wheel** group to use the IPC interface, add this line:

   ```
   IPCAllowGroups=wheel
   ```

2. You can add users or groups also with the **usbguard** command. For example, the following command enables a user to have full access to the **Devices** and **Exceptions** sections and to list and modify the current policy:

   ```
   # usbguard add-user <user_name> --devices ALL --policy modify,list --exceptions ALL
   ```

   Replace ***<user_name>*** with the user name that should receive these permissions.

   You can remove the granted permissions for a user by using the **usbguard remove-user *<user_name>*** command.

3. Restart the **usbguard** daemon to apply your changes:

   ```
   # systemctl restart usbguard
   ```

11.8. LOGGING USBGUARD AUTHORIZATION EVENTS TO THE LINUX AUDIT LOG

By default, the **usbguard** daemon logs events to the **/var/log/usbguard/usbguard-audit.log** file. You can integrate logging of USBguard authorization events to the standard Linux Audit log.

Prerequisites

- The **usbguard** service is installed and running.
- The **auditd** service is running.

Procedure

1. In the **/etc/usbguard/usbguard-daemon.conf** file, change the **AuditBackend** option from **FileAudit** to **LinuxAudit**:

```
AuditBackend=LinuxAudit
```

2. Restart the **usbguard** daemon to apply the configuration change:

   ```
   # systemctl restart usbguard
   ```

Verification

1. Query the **audit** daemon log for a USB authorization event, for example:

   ```
   # ausearch -ts recent -m USER_DEVICE
   ```