Oracle Linux

Working With UEFI Secure Boot

F40522-13
July 2023

Contents

Preface

Oracle Linux: Working With UEFI Secure Boot provides background and other related information about the UEFI Secure Boot feature and its implementation in Oracle Linux.

Note:

UEFI Secure Boot is also more commonly referred to as "Secure Boot" in this document.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at https://www.oracle.com/corporate/accessibility/.

For information about the accessibility of the Oracle Help Center, see the Oracle Accessibility Conformance Report at https://www.oracle.com/corporate/accessibility/templates/t2-11535.html.

Access to Oracle Support for Accessibility

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit https://www.oracle.com/corporate/accessibility/learning-support.html#support-tab.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

1

About UEFI Secure Boot

This chapter provides an overview of Oracle's Secure Boot feature, which you can use on systems that boot by using the Unified Extensible Firmware Interface (UEFI).

Note:

Some of the instructions and examples in this document may apply to multiple Oracle Linux releases. The examples use the `yum` command for compatibility with several supported versions of Oracle Linux. However, beginning with Oracle Linux 8, you can opt to use the `dnf` command, as appropriate.

For additional information about UEFI Secure Boot that is beyond the scope of this document, see the following documentation:

- Oracle Linux: UEFI Secure Boot Update Notices
- Sign Kernel Modules for Use With UEFI Secure Boot

Overview of Secure Boot

Operating system (OS) security requires that every layer below the OS layer also be secure. You cannot safely run secure software on a given CPU if it cannot be trusted to execute code correctly. Moreover, If the boot loader is tampered with or the firmware itself has been compromised, the kernel that is booted cannot be trusted.

UEFI Secure Boot is a platform feature within the UEFI specification that ensures that the system boots by using only the software that is trusted by the hardware manufacturer. Secure Boot provides a verification mechanism where the firmware validates a boot loader before executing it. This mechanism ensures that the code that is launched by a system's firmware is trusted. When the system starts, the firmware checks the signature for each piece of boot software, including the firmware drivers and the operating system itself. If the signatures are valid, the system boots, and the firmware relinquishes control to the OS.

Secure Boot uses cryptographic checksums and signatures to prevent malicious code from being loaded and executed early in the boot process, before the operating system has actually loaded. Every program that is loaded by the firmware includes a signature and a checksum that is validated by the firmware before the program is started. Secure Boot stops all attempts to execute an untrusted program to prevent any unexpected or unauthorized code from running in the UEFI-based environment.

Most UEFI compliant systems ship with Secure Boot enabled. These systems are also preloaded with Microsoft keys so that the firmware trusts binaries that are signed by Microsoft. By default, these systems do not run any unsigned code. However, you can change the firmware configuration to either enroll additional signing keys or to disable Secure Boot.

Note that Secure Boot is not intended to prevent users from controlling their own systems. You have the option of enrolling extra keys into the system to enable the signing of other programs. Further, some systems with Secure Boot enabled by default provide capability for removing platform-provided keys, which forces the firmware to trust user-signed binaries only.

How the Secure Boot Process Works

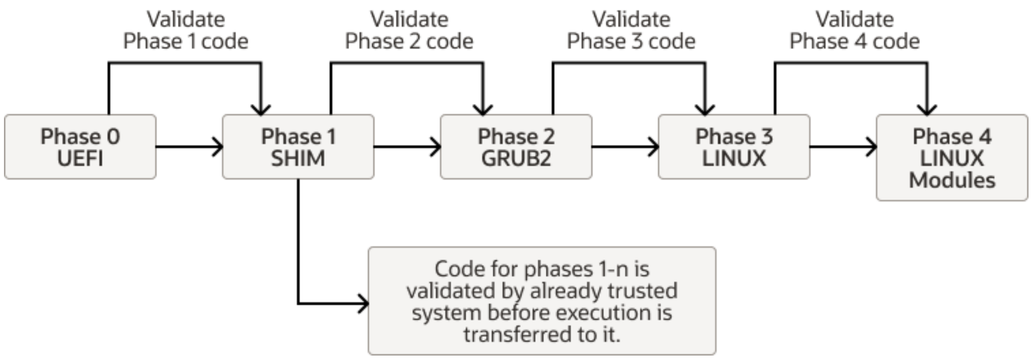
Each step in the Secure Boot process checks a cryptographic signature on the executable of the next step. The BIOS checks a signature on the boot loader and then the boot loader checks the signatures on all of the kernel objects that it loads.

Objects in the chain are typically signed by the software manufacturer by using private keys matching the public keys that are already in the BIOS. Altered modules or objects in the boot chain would have mismatched signatures, which prevents the device from booting the image. Otherwise, the platform boots successfully.

To meet the goals of Secure Boot, the Linux boot loader should provide authentication of the Linux kernel. In addition, the Linux distribution should provide further security enforcement in the kernels that it provides. The Shim first stage boot loader program provides a way to meet both of these goals. For more explanation, see Description of the Shim First Stage Boot Loader.

The following figure illustrates the Secure Boot process.

Figure 1-1 Secure Boot Process



Phase 0: The UEFI checks whether Secure Boot is enabled and loads the keys that it stores for this purpose from the UEFI Secure Boot key database.

Phase 1: The Shim software loads and UEFI validates the signature that was used to sign the Shim. If the signature is valid, the Shim can load. If the signature does not match a key in the UEFI Secure Boot key database, the Shim is unable to load. When the Shim is loaded, it is responsible for validation for all subsequent code. The Shim is responsible for maintaining its own MOK database, where alternate keys are stored for validation purposes.

Phase 2: The key that is used to sign the GRUB2 secondary bootloader is validated by the Shim. If validation passes, GRUB2 loads and Shim is able to validate the keys that are used to sign the kernel images available to GRUB2.

Phase 3: A valid kernel loads. The kernel has read access to the keys in the UEFI Secure Boot key database and the MOK database, as well as has its own set of trusted keys that are built into the kernel image itself.

Phase 4: The kernel validates the keys that are used to sign any additional modules that need to be loaded, as well as signed kernel images on `kexec` operations. Depending on the kernel implementation, the kernel would trust the keys in the UEFI Secure Boot database or the MOK database; or, it would only trust the keys that are built into the kernel image itself. Note that for `kexec` signed kernel images signature validation, the following keys can be used: UEFI, DB/MOK, DB keys, and also kernel builtin keys.

Secure Boot Limitations

Secure Boot is purposely designed to limit the applications that can run before the OS is booted because after the system is booted, the OS has no way of identifying any programs that were previously booted or even whether the system was booted securely. For example, if a boot kit is injected into the system prior to boot, Secure Boot could be rendered useless. Another example is the situation where an attacker finds a way to disable Secure Boot and then subsequently installs malware that could be interpreted by the OS as platform security and thus compromise it.

Further, Secure Boot might impact or limit the use of some features and user actions, including the activation of the Kernel Lockdown feature. This feature is designed to prevent both direct and indirect access to a running kernel image to protect against unauthorized modification of the kernel image and access to the security and cryptographic data that is located in the kernel's memory.

When Lockdown mode is activated, some of the features that you typically use to modify the kernel might be affected, including the following:

- Loading of kernel modules that are not signed by a trusted key.
- Use of `kexec` tools to start an unsigned kernel image.
- Hibernation and resume from hibernation modes.
- User space access to physical memory and I/O ports.
- Module parameters that enable you to set memory and I/O port addresses.
- On x86_64 systems only, writing to MSRs through `/dev/cpu/*/msr`. On Arm platforms, `/dev/cpu/*` is not supported.
- Use of custom ACPI methods and tables.
- Advanced Configuration and Power Interface (ACPI) and ACPI Platform Error Interface (APEI) error injection.

If you need to use any of these features, you can disable Secure Boot through the system's Extensible Firmware Interface (EFI) setup program. For further information, see Enabling and Disabling Secure Boot.

About Secure Boot Keys

All Secure Boot key types are similar in that each key is an example of the Public Key Infrastructure (PKI). As such, each key includes two long numbers that are used for encryption or, in the case of Secure Boot, for data authentication. Secure boot uses a private key and a public key.

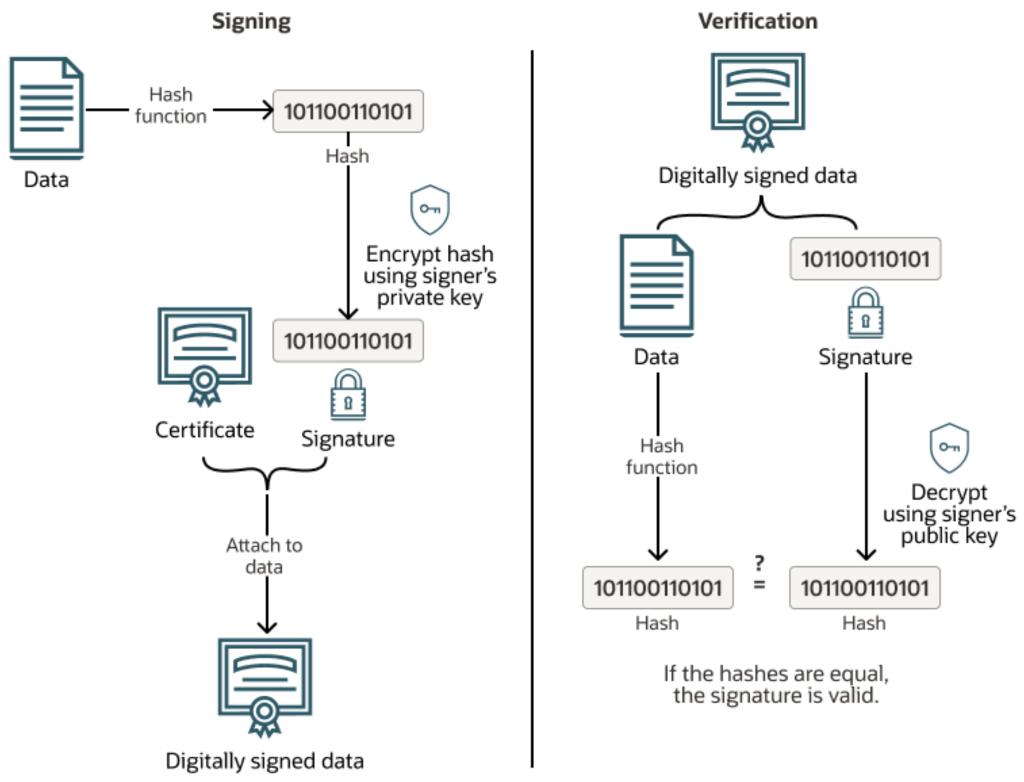
The *private key* is used to sign a file, which is an EFI program, and that signature is then appended to the program.

The *public key* must be made publicly available. For Secure Boot, this key is embedded in the firmware itself or is stored in NVRAM. You can use the public key in conjunction with the signature to verify that the file has not been modified and also to verify that the file was signed with a key that matches the public key currently in use.

For more information about PKI, see Oracle Linux: Managing Certificates and Public Key Infrastructure.

The following figure more clearly illustrates the key signing and verification process that Secure Boot uses.

Figure 1-2   Secure Boot Key Signing and Verification Process



The data is signed with a private key. This process generates a hash of the data and uses the signer's private key to encrypt the hash. When the data needs to be validated, the signer's public key is used to decrypt the signature to obtain the hash of the data that was signed. A hash of the data that is being validated is generated and can be compared to the hash that was decrypted from the signature. A match between the hashes indicates that the data is unchanged since it was signed.

Although UEFI specification supports several key types, the X.509 key is commonly used. X.509 is a standard format for public key certificates and digital documents that securely associate cryptographic key pairs with identities, individuals, and organizations. This key is the easiest to generate when using OpenSSL. Using the X.509 key meets an additional requirement that the platform key (PK) must be an X.509 key. The X.509 key is stored in DER (Distinguished Encoding Rules) format. The DER key can be base64 encoded and stored as text in a PEM file.

The X.509 certificate includes a public key, a digital signature, as well as information about the identity that is associated with the certificate and its issuing certificate authority (CA). The public key and a private key form a key pair. The private key is kept secure, and the public key is included in the certificate. With the key pair, the owner of the private key can digitally sign documents that can be verified by anyone with the corresponding public key. The key pair also enables third parties to send messages that are encrypted with the public key that only be decrypted by the owner of the private key.

Note:

Mentions of the terms "key" and "public key" in the UEFI specification refer to the public part, the X.509 certificate; whereas, OpenSSL uses the term key to mean the private key that is used for signing. As such, all Secure Boot keys should be X.509 keys and *not* OpenSSL keys.

More recent kernel images use the PKCS#7 key type. PKCS (Public Key Cryptography Standards) is a set of standards for the generation and verification of digital signatures and certificates. Note that PKCS#7 is a method of storing signed or encrypted data, including X.509 certificates. This point can be confusing because the key that is stored within a PKCS#7 structured DER or PEM formatted file is still an X.509 key. Nonetheless, the type and therefore the expected format when processing the file is different. It is important to be aware of this distinction when signing kernel modules, as it is important that you use the correct tool when performing the signing operation.

Four types of Secure Boot keys are built into the firmware. Moreover, a fifth key can be used by the Secure Boot Shim, while a sixth key can be built into Oracle Linux kernel image:

Platform Key (PK)
Is the top-level key type that is used in Secure Boot. The PK offers complete control of the secure boot key hierarchy. The holder of the PK can install a new PK and update the Key Encryption Key (KEK). UEFI Secure Boot supports a single PK that is usually provided by the motherboard manufacturer. As a result, only the motherboard manufacturer has complete control over the system. You can control the Secure Boot process by replacing the PK with your own version.

Key Exchange Key (KEK)
Is a key type that is used to sign keys so that the firmware accepts the keys as valid when entering them into the database. Without the KEK, the firmware cannot detect whether a new key is valid or introduced by Malware. If KEK were absent, Secure Boot would require that the databases remain static. Also, because a critical point of Secure Boot is DBX, a static database would become unworkable. Hardware often ships with two KEKs: one from Microsoft and one from the motherboard manufacturer, thus enabling either party to issue updates.

UEFI Secure Boot Database Key (DB)
Is a key type that is most often associated with Secure Boot, mainly because this key is used to sign or verify the binaries that you run, such as boot loaders, boot managers, shells, drivers, and so. Most hardware is shipped with two Microsoft keys installed. Microsoft uses one of these keys and the other key is used to sign third-party software, such as Shim. Some hardware also is shipped with keys that are created by the computer manufacturer or other parties. It is important to note that the database can hold multiple keys for different purposes. Note also that the database can contain both public keys that are matched to private keys

(which can be used to sign multiple binaries), as well hashes (which describe individual binaries).

Forbidden Signature Database Key (DBX)
Is a key type that contains keys and hashes that correspond to known Malware or other undesirable software. If a binary matches a key or hash that is in both the UEFI Secure Boot key database and the DBX, the DBX takes precedence. This facility prevents the use of a single binary even if the binary is signed by a key that you do not want to revoke because it has been used to sign several, legitimate binaries.

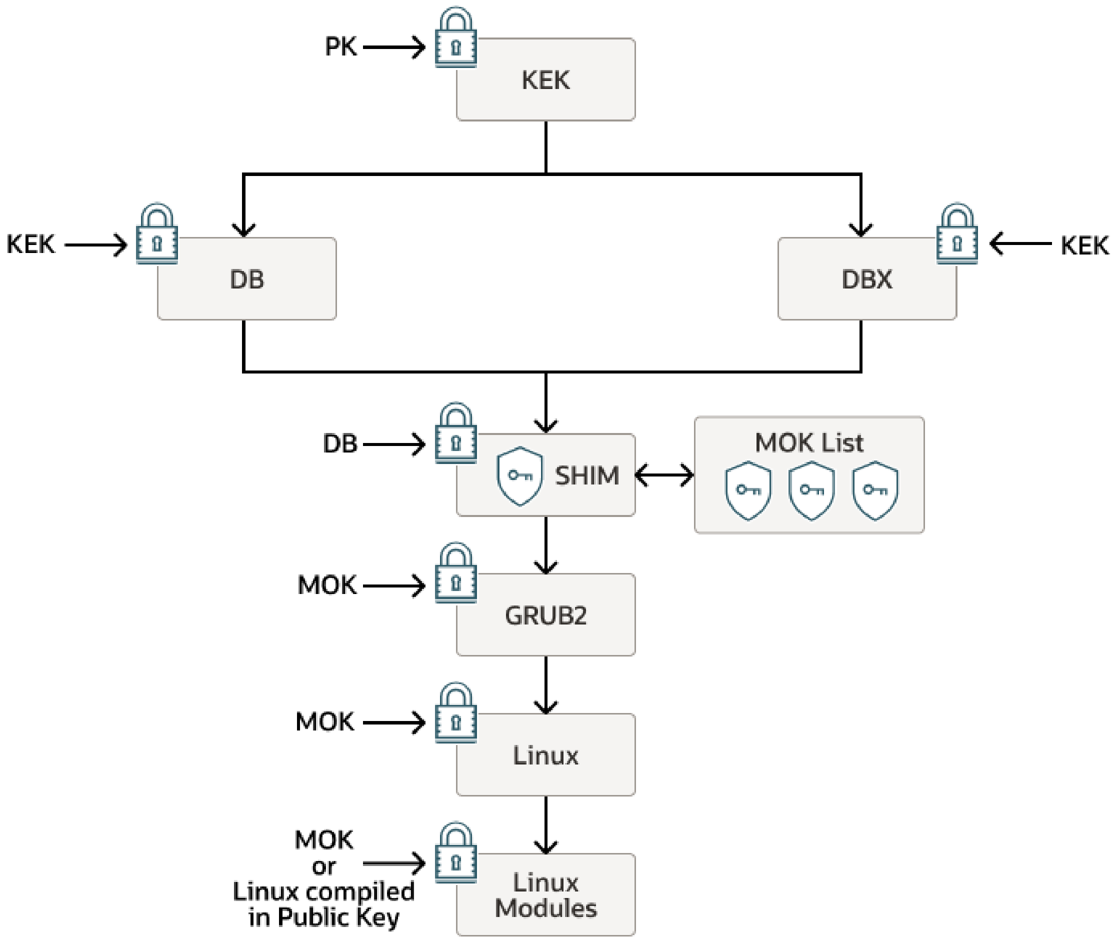
Machine Owner Key (MOK)
Is a key type that is equivalent to a DB key. You use the MOK key type to sign boot loaders and other EFI executables; or, you can use this key type to store hashes that correspond to individual programs. MOKs are not a standard part of Secure Boot; however, they are used by the Shim and PreLoader programs to store keys and hashes. The major capability that is provided by the MOK facility is the ability to add public keys to the MOK list without requiring the key chain back to another key that is already in the KEK database. The MOK facility is an ideal way to test newly generated key pairs and the kernel modules that are signed with them. When a key is on the MOK list, it will be automatically propagated to the system key ring on every subsequent boot when UEFI Secure Boot is enabled. Note that to enroll a MOK key, you must manually do so on each target system by using the UEFI system console.

Public Key in the kernel
Are keys that can be built into the OS so that they can be used to check the signatures as kernel modules are loaded. Normally, when the `CONFIG_MODULE_SIG_KEY` parameter is unchanged from the default, the kernel build automatically generates a new keypair by using OpenSSL, if one does not exist. Then, when `vmlinux` is built, the public key is built into the kernel. Note that the Secure Boot implementation in UEK R6, prior to UEK R6U3, requires that all modules are signed using a key that is built into the kernel. Earlier UEK R6 kernels do not apply the same level of trust to kernel modules that are signed using a key that is only present in the MOK database.

Description of the Secure Boot Key Implementation

The following figure graphically depicts how the Secure Boot key implementation works in Oracle Linux.

Secure Boot Key Implementation



At the UEFI firmware level, the Platform Key (PK) is used to validate the Key Exchange Key (KEK) which is, in turn used to validate all Database Keys (DB) and all DBX Keys (DBX). The DB keys are used with the DBX keys to validate the the key used to sign the Shim binary. After the Shim is validated, keys stored within the MOK list and loaded by the Shim can be trusted to perform validation for subsequent operations. The GRUB2 secondary bootloader is validated by using a key within the MOK list, as well as the MokListX (forbidden MOK keys); where for the latter, the target is similar to DBX but on the MOK level. Equally, when GRUB2 loads the kernel, validation of the kernel image binary is performed against the keys in the MOK list before the kernel can be loaded. Finally, after the kernel is loaded, the Linux kernel modules or Linux kernel images that are used for `kexec` operations can be validated, either against the MOK list or against any public keys that are compiled directly into the Linux kernel.

One exception to this implementation is noted for UEK R6 releases prior to UEK R6U3, where at the final stage, when the kernel modules are loaded, the Secure Boot implementation does not trust kernel modules that are signed just with the keys that are stored in the MOK list or in the UEFI Secure Boot key database. In this case, all kernel modules must be signed with a key that is compiled into the kernel before the kernel can load the module. More information on this process is provided in Inserting the Module Certificate in the Kernel and Signing the Kernel Image. From UEK R6U3 onward, kernel modules signed with keys stored in the MOK list are trusted.

Description of the Shim First Stage Boot Loader

Shim is a basic software package that is designed to work as a first-stage boot loader on UEFI-based systems.

Systems that are capable of using UEFI Secure boot usually ship with the following two keys:

- Microsoft Windows Production PCA 2011
- Microsoft Corporation UEFI CA 2011

When Secure Boot is enabled on the system, only those programs that are signed with either of the previous two keys will boot. For the Shim first stage boot loader, Oracle uses a process that is agreed upon with Microsoft to sign Oracle's version of Shim with the Microsoft Corporation UEFI CA 2011 CA key. Embedded certificates within the Oracle shim validate the signed 2nd stage boot loader and the kernel. See Description of the Secure Boot Key Implementation.

About the MOKManager Utility

MokManager is a utility that you can use to manage the MOK list. A MOK is a key that is used to verify 2nd stage boot loaders, kernels, and kernel modules from Shim. In Oracle Linux, the MokManager utility is installed in the EFI System Partition (ESP), within the `/boot/efi/EFI/redhat` directory. Originally called `MokManager.efi`, this file has since been renamed to `mmx64.efi` for x86_64 platforms or `mmaa64.efi` for Arm platforms.

Machine Owner Key (MOK) keys can be placed in the ESP and then installed from the MokManager during boot.

A MOK Forbidden Signature Database (MOKx) that is similar to the DBX also exists. The MOKx enables a user to forbid a particular kernel or kernel module from being loaded, as identified by the public key or the signature. The MOKx also forbids any second stage boot loader or binary that Shim boots, similar to `grub2/fwupd/mmx64/mmaa64/etc`.

You can also use the related `mokutil` utility to install MOK keys from Oracle Linux. For more information about this utility, see Tools and Applications for Administering Secure Boot.

How Secure Boot Is Enforced Within Oracle Linux

The enforcement of Secure Boot within Oracle Linux includes certain restrictions, most of which are implemented to prevent Oracle Linux from being used as a boot loader through the `kexec` tool, which would break the Secure Boot chain of trust. Up until recently, these restrictions were not enforced in the mainline kernel. The restrictions, which were originally referred to as *securelevel* and more recently referred to as *lockdown*, prevent access to Ring-0 when Secure Boot is enabled, even by the `root` user.

The following are details of the how Secure Boot is enforced in Unbreakable Enterprise Kernel (UEK) and Oracle Linux releases:

- **Unbreakable Enterprise Kernel (UEK)**

The following information applies to Secure Boot enforcement in UEK releases:

- **UEK R4 (Securelevel)**

  UEK R4 contains out-of-tree patches that are based on the addition of a course-grained, run-time configuration option for restricting the ability of user space to modify the running kernel with a BSD-style securelevel. It provides a run-time configuration variable at `/sys/kernel/security/securelevel`, which can be written to by `root`.

- **UEK R5 (Lockdown)**

  The Kernel Lockdown feature is designed to prevent both direct and indirect access to a running kernel image. The feature attempts to protect against unauthorised modification of the kernel image. It also prevents access to security and cryptographic data that is located in kernel memory, while still permitting driver modules to be loaded. When Secure Boot is enabled in UEK R5, lockdown is automatically enabled.

- **UEK R6 and R7 (Lockdown)**

  UEK R6 uses the Lockdown feature that is contained within the mainline Linux 5.4 kernel . The 5.4 Lockdown contains three different modes: none, integrity, and confidentiality. If set to integrity, kernel features that enable user space to modify the running kernel are disabled. If set to confidentiality, kernel features that enable user space to extract confidential information from the kernel are also disabled. When Secure Boot is enabled in UEK R6, lockdown integrity mode is automatically enabled.

  > **Note:**
  >
  > On aarch64 platforms, Secure Boot has been implemented and kernel images are signed starting with UEK R7 U1.

- **Red Hat Compatible Kernel (RHCK)**

  The following information applies to the Secure Boot implementation on RHCK for Oracle Linux:

  - **Oracle Linux 7 (Lockdown)**

    RHCK for Oracle Linux 7 uses the same course-grained, run-time configuration option for restringing user space's ability to modify the running kernel with a BSD-style securelevel, as in UEK R4.

  - **Oracle Linux 8 (Lockdown)**

    RHCK for Oracle Linux 8 uses an older implementation of Lockdown than is enabled in UEK R5, but is more recent than the BSD-style securelevel implementation in UEK R4.

Enabling and Disabling Secure Boot

You can enable and disable Secure Boot by accessing the system's EFI setup program. For instructions on enabling and disabling Secure Boot through the EFI setup program for a specific hardware, refer to the manufacturer's instructions.

Note that if you are having trouble working with the system UEFI, you can disable any further validation for Secure Boot at the level of the Shim. Although you can disable further validation for Secure Boot from user space, you still need physical access to the system at boot so that you can access the MokManager utility when the Shim loads. For more information, see Disabling Secure Boot by Using the mokutil Utility.

2

Tools and Applications for Administering Secure Boot

This chapter provides a basic summary of the tools and applications that you can use to administer Secure Boot in Oracle Linux.

About the pesign Tool

The `pesign` tool is a command-line tool for manipulating signatures and cryptographic digests of UEFI applications. You can use the `pesign` tool to sign kernels for both GRUB2 and Shim.

You can also use the `pesign` tool for printing binary signature information. Note that to be recognized, the `pesign` package for v0.112-22 and later, which is available in Oracle Linux 8, also provides the `pesigcheck` tool that you can use to verify a signature versus an exact public certificate.

The `pesign` tool accepts the X.509 certificate/key pair and signs a PE-COFF binary with it. The Oracle Linux kernel has an EFI boot stub that wraps the `bzImage` file as a PE-COFF binary according to standard UEFI implementation. This implementation provides you with the option to boot the kernel directly from UEFI without requiring a boot loader. In addition, the `pesign` can perform the required signing. See About Secure Boot Keys.

Typically, you are likely to use the `pesign` tool to sign a kernel image with a custom key that you might create to sign custom kernel modules, as described in Signing Kernel Images and Kernel Modules for Use With Secure Boot. Additionally, you may choose to use the `pesign` tool in situations where you might want to enroll the hash for a particular kernel within the MOK database so that the kernel can be loaded at boot, even if the Shim does not contain its certificate. You can use the `pesign` tool to extract a kernel hash from the signed kernel binary so that you can enroll it by using the `mokutil` tool. This process is described in Use mokutil to Update Signature Keys for UEFI Secure Boot.

About the efibootmgr Application

Oracle Linux provides the `efibootmgr` user space application that you can use to modify the Intel Extensible Firmware Interface (EFI) Boot Manager. You use the application to perform several tasks, including the following:

- Create and destroy boot entries.
- Change the boot order.
- Change the next running boot option.

This tool is a general usage application and does not specifically relate to Secure Boot, but it does allow you to manage UEFI boot options directly from user space, which can make it easier to debug and resolve some UEFI boot issues directly from the command line. For more information and examples, see the `efibootmgr(8)` manual page.

About the mokutil Utility

An important part of the Shim design is to provide users with the ability to control their own systems. The distribution vendor key is built into the Shim binary itself, but an extra database with keys that can be managed by the user, the so-called Machine Owner Key (MOK) list or database, is also provided. You can use the `mokutil` utility to add and remove keys in the MOK list, which remain completely separate from the distribution vendor key.

The MOK database is useful for adding certificates for custom-built kernels or kernel modules, where the keys that are used to sign these components, or, at least, the CA certificate for those keys, is not present within the UEFI Secure Boot key database.

Although keys can be manually added to the UEFI Secure Boot key database, this process can be difficult. Instructions are usually specific to the firmware itself and users must refer to hardware documentation to make these changes. Furthermore, keys need to be directly accessible to UEFI, which is often not a straightforward operation and keys may need to be copied to appropriate media so that UEFI is able to access them. The `mokutil` utility enables you to make keys available to the MOK database directly from user space on the command line. Because keys are often created or extracted in this space, `mokutil` is the most appropriate tool to use for managing keys that are used for Secure Boot.

Note that although `mokutil` is run from user space, it does not actually update the MOK database directly. Instead, it makes keys available to the MOK Management service and triggers the Shim to display the MOK Management menu at boot. This process ensures that keys or hashes are only enrolled within the MOK database by somebody who has physical access to the system and prevents a malicious application or user from modifying the MOK database directly from user space.

Typical use-case scenarios where you might use the `mokutil` utility include adding keys for custom modules that are not included and signed with the distribution and which you have had to sign yourself, as described in Signing Kernel Images and Kernel Modules for Use With Secure Boot; and, adding keys or hashes for custom kernels for which the signing key is either revoked or you have built from source yourself, as described in Use mokutil to Update Signature Keys for UEFI Secure Boot. Finally, you can also use the `mokutil` utility to effectively disable Secure Boot operations from the Shim upward, as described in Disabling Secure Boot by Using the mokutil Utility.

For additional information, see the `mokutil(1)` manual page.

Disabling Secure Boot by Using the mokutil Utility

In the event that you find it difficult to control the Secure Boot state through the EFI setup program, you can optionally use the `mokutil` utility to disable Secure Boot at the level of the Shim, which means that although UEFI Secure Boot is enabled, no further validation takes place after the Shim is loaded.

The following steps apply to operating systems that are loaded through Shim and GRUB:

1. Run the `mokutil` command to disable Secure Boot at the Shim level:

   ```
   sudo mokutil --disable-validation
   ```

2. Choose a password that is between 8 and 16 characters and then enter the same password to confirm.
3. Reboot the system.
4. Press a key to perform MOK management, when prompted.
5. Select the **Change Secure Boot state** option.
6. Type each character for the password that you chose, as requested, to confirm the change.

   You have to press Return (or Enter) after each character.
7. Select Yes, then select Reboot to reboot the system.

In the event that you need to re-enable the Secure Boot state at the Shim level, you can do so by running the following command and then repeating Steps 2-7.

```
sudo mokutil --enable-validation
```

Validating SBAT Status by Using the mokutil Utility

Starting from version 15.3 of the `shim` package on Oracle Linux 8 and Oracle Linux 9, Oracle has been using UEFI Secure Boot Advanced Targeting (SBAT) to revoke older versions of core boot components such as `grub2` and `shim` by setting generation numbers in the `.sbat` section of the UEFI binary. The generation number set in a UEFI binary defines its revocation level.

To confirm whether UEFI Secure Boot is active, use the `--sb-state` option with the `mokutil` command:

```
mokutil --sb-state
```

Starting from version 0.6.0 of the `mokutil` package, the `mokutil` command can be used to review and update UEFI SBAT revocation status. To review the current UEFI SBAT level on which the current system is running, use the `--list-sbat-revocations` option:

```
mokutil --list-sbat-revocations
```

You can change the SBAT policy that applies from the next reboot. Setting the SBAT policy to `latest` applies the latest SBAT revocations and prevents the system from booting older `grub2` and `shim` packages operating at an earlier SBAT level, and `previous` falls back to the previous SBAT revocation level:

```
mokutil --set-sbat-policy latest
```

```
mokutil --set-sbat-policy previous
```

For systems with UEFI Secure Boot enabled, the default SBAT policy is `previous` . Both the `latest` and `previous` SBAT policies only set a revocation level that's the same or later than it was when the latest `shim` package was installed.

For troubleshooting purposes it's also possible to reset the SBAT policy to the default revocation level. That can be achieved by disabling UEFI Secure Boot and then setting the `delete` SBAT policy:

```
mokutil --set-sbat-policy delete
```

Note:

You can review the `.sbat` metadata used by `grub2` and `shim` by using the `objdump` command. For example, on an x86_64 system you can run the following commands:

```
objdump -s -j .sbat grubx64.efi
```

```
objdump -s -j .sbat shimx64.efi
```

To review the current SBAT policy levels for the provided shim:

```
objdump -s -j .sbatlevel shimx64.efi
```

About the dbxtool Command

The `dbxtool` command combines a command-line tool with the `systemd` service that is used to apply UEFI Secure Boot DBX updates. The tool enables you to operate on the UEFI Forbidden Signature Database, also known as the DBX revocation list. For example, you can use the command to list the current DBX contents, as well as update the current contents to a newer version.

UEFI DBX files are made available at https://uefi.org/revocationlistfile. The DBX prevents software that is signed using a compromised key from loading. This helps to protect the integrity of the Secure Boot framework and makes it possible to avoid having to manually manage a static database of keys.

The most current UEFI DBX files are packaged with the `dbxtool` package and updates are present within each subsequent release of this package on the Oracle Linux yum server. The DBX files are located in `/usr/share/dbxtool/` and `/usr/share/dbxtool-oracle/`. As long as the `dbxtool systemd` service is running, DBX updates should be handled automatically.

You may use this tool, manually, if you are notified of a CVE that requires a DBX update.

3 Signing Kernel Images and Kernel Modules for Use With Secure Boot

This chapter provides instructions on signing kernel modules for Secure Boot.

> **NOT_SUPPORTED:**
>
> Oracle does not support any modules that are built from source directly outside of Oracle's supported release mechanisms. Support may be provided from your hardware vendor.

A system in Secure Boot mode only loads boot loaders and kernels that have been signed by Oracle. In some cases, you may need to build and install a third-party module to enable particular hardware on your system. If you still require UEFI Secure Boot, the module must be signed with a key that can be validated against a certificate within the UEFI Secure Boot key database or within the Shim MOK Manager key database so that it is recognized at boot. Note that if you are running UEK R6, UEK R6U1 or UEK R6U2 the key that is used to sign the module must be compiled into the kernel and then the kernel must be signed again. Note also that the kernel signing key must be added to the either the UEFI or MOK database. As of UEK R6U3, the kernel allows external modules to load under Secure Boot if the signing key is already enrolled in the MOK database.

> **Important:**
>
> Using the MOK utility with your system may depend on server firmware implementation and configuration. Check that your server supports this before attempting to manually update signature keys used for UEFI Secure Boot. If you are unsure, do not follow the instructions provided here.
>
> Adding certificates to the MOK database by using the MOK utility requires that you have physical access to the system so that you can complete the enrollment request after the Shim is loaded by UEFI. If you do not have physical access to the system, do not follow the instructions that are provided here.

Due to differences in kernel releases, instructions on how to sign modules may differ, depending on the kernel version that you are using. One important difference that you should note is that the key signature type changed from X.509 to PKCS#7 within the module signature. This change means that although the process to sign the module by using the `sign-file` utility that is provided with each kernel source is still used, you may be required to use a utility that is more appropriate to the specific kernel for which you are signing a module.

More importantly, prior to UEK R6U3, UEK R6 kernels did not offer the same level of trust for the keys that are available for UEFI or the Shim MokManager key database. If you are running UEK R6, UEK R6U1 or UEK R6U2, you must additionally insert the key into the kernel boot image so that the kernel has access to this key from the kernel builtin trusted keys keyring. For this reason, additional steps are required when signing a module for any UEK R6 release kernel prior to UEK R6U3. The `insert-sys-cert` utility, which is provided with the UEK R6 kernel source, enables you to insert a key into a compressed kernel image. When the kernel image has been modified, you must sign it by using the `pesign` tool and then add the signing key to either the UEFI or MOK database.

Requirements for Signing Kernel Images and Kernel Modules

Before you can sign a module, you must install several required packages, including the kernel source for the kernel where the module is loaded. You also require a signing certificate for a key pair that you have created for this purpose. For UEK R6, some additional steps are required so that you can use tools to load your module signing key into the builtin kernel keyring.

Installing Required Packages

A standard minimal installation of the latest Oracle Linux 7, Oracle Linux 8 or Oracle Linux 9 update release is required for this procedure. Note that you must enable any required repositories prior to installing the required packages.

Oracle Linux 7

Obtain the kernel source for your environment. If you are using the Unbreakable Enterprise Kernel. the source is available in the `kernel-uek-devel` package. If you are using RHCK, the source is available in the `kernel-devel` package. If you are using UEFI Secure Boot functionality, Oracle recommends installing and using UEK.

```
sudo yum install kernel-uek-devel
```

Update the system to ensure that you have the most recent kernel and related packages.

```
sudo yum update
```

If the kernel is updated as part of this operation, you should reboot before you continue with any kernel-signing operations to avoid confusion around the kernel version that you are working with and the currently running kernel.

Enable the `optional_latest` repository in your yum configuration.

```
sudo yum-config-manager --enable ol7_optional_latest
```

Install the utilities that are required to perform module signing operations.

```
sudo yum install openssl keyutils mokutil pesign
```

If you need to build module source, you may additionally wish to install the `Development Tools` group to ensure that build tools are available to you, for example:

```
sudo dnf group install "Development Tools"
```

Oracle Linux 8 and Oracle Linux 9

Obtain the kernel source for your environment. If you are using the Unbreakable Enterprise Kernel, the source is available in the `kernel-uek-devel` package. If you are using RHCK, the source is available in the `kernel-devel` package. If you are using UEFI Secure Boot functionality, Oracle recommends installing and using UEK.

```
sudo dnf install kernel-uek-devel
```

Update the system to ensure that you have the most recent kernel and any related packages.

```
sudo dnf update
```

If the kernel is updated as part of this operation, you should reboot before you continue with any kernel-signing operations to avoid confusion around the kernel version that you are working with and the current running kernel.

Install the utilities required to perform module signing operations.

```
sudo dnf install openssl keyutils mokutil pesign
```

If you need to build module source, you may additionally wish to install the `Development Tools` group to ensure that build tools are available to you, for example:

```
sudo dnf group install "Development Tools"
```

Generating a Signing Certificate

If you do not already have a signing certificate that you intend to use to sign your modules or kernel images, you can generate one by using the OpenSSL utilities that are available in Oracle Linux. For more information about OpenSSL and the public key infrastructure, see Oracle Linux: Managing Certificates and Public Key Infrastructure.

1. Create a configuration file that OpenSSL can use to obtain default values when generating your certificates. You can create this file at any location, but it is useful to keep it with the rest of your OpenSSL configuration in `/etc/ssl/x509.conf`. The file should look similar to the following:

   ```
   [ req ]
   default_bits = 4096
   distinguished_name = req_distinguished_name
   prompt = no
   string_mask = utf8only
   x509_extensions = extensions

   [ req_distinguished_name ]
   O = Module Signing Example
   CN = Module Signing Example Key
   emailAddress = first.last@example.com

   [ extensions ]
   basicConstraints=critical,CA:TRUE
   keyUsage=digitalSignature
   extendedKeyUsage = codeSigning
   subjectKeyIdentifier=hash
   authorityKeyIdentifier=keyid:always
   ```

You should edit the `O`, `CN` and `emailAddress` fields to something more appropriate. Note that in the `extensions` section of the configuration, the `keyUsage` field is set as `digitalSignature`. Additionally, the `extendedKeyUsage` option is set to `codeSigning` for compatibility with some key verification tools.

2. Generate a new key pair using this configuration file:

   ```
   sudo openssl req -x509 -new -nodes -utf8 -sha512 -days 3650 -batch -
   config /etc/ssl/x509.conf -outform DER -out /etc/ssl/certs/pubkey.der -
   keyout /etc/ssl/certs/priv.key
   ```

   This signing certificate is valid for 10 years (3,650 days). Ensure that the keys are adequately protected.

3. Export the certificate in PEM format:

   ```
   sudo openssl x509 -inform DER -in /etc/ssl/certs/pubkey.der -out /etc/ssl/
   certs/pubkey.pem
   ```

Signing the Kernel for Secure Boot

UEFI Secure Boot requires that kernels are signed with a trusted certificate to prevent attackers from installing and running unauthorized operating systems.

Oracle signs kernel releases provided through supported software channels to verify their origin and integrity. For a custom kernel to perform the `boot` or `kexec` operations you must sign the kernel image by using your own signing certificate and confirming that you trust kernel images that are signed with that certificate.

Configure an NSS Database

The `pesign` tool that you use to sign the kernel expects the kernel signing key to be stored within an NSS database, which is designed for storing complete sets of keys.

Create an NSS database to use with `pesign`:

```
sudo certutil -d . -N

Enter a password which will be used to encrypt your keys.
The password should be at least 8 characters long,
and should contain at least one non-alphabetic character.

Enter new password:
Re-enter password:
```

When you use the `certutil` command, you are prompted for a password for the NSS database. Choose a password for the database. This password is required when signing the kernel.

NSS utilities are only capable of working with PKCS#12 formatted key files. For this reason, you must export a PKCS#12 version of the kernel signing key so that you are able to sign the kernel image, for example:

```
sudo openssl pkcs12 -export -inkey /etc/ssl/certs/priv.key -in /etc/ssl/certs/
pubkey.pem -name cert -out /etc/ssl/certs/cert.p12

Enter Export Password:
Verifying - Enter Export Password:
```

Note:

The previous step requires that you enter a password for the PKCS#12 archive. It is useful if this password matches the password that you use for the NSS database, where this data is ultimately stored.

Add the PKCS#12 version of the kernel signing key to the new database. You are prompted first for the password of the NSS database that you have just created and then you are prompted for the password used when you exported the PKCS#12 key file.

```
sudo pk12util -d . -i cert.p12

Enter Password or Pin for "NSS Certificate DB":
Enter password for PKCS12 file:
pk12util: PKCS12 IMPORT SUCCESSFUL
```

Sign the Kernel Image

Use the `pesign` utility to remove the existing PE signature and resign the kernel with the new one by using the kernel signing key that is stored in the NSS database. Note that you are prompted for the password of the NSS certificate database that you created in Configure an NSS Database.

```
sudo pesign -u 0 -i /boot/vmlinuz-$(uname -r) --remove-signature -o vmlinuz.unsigned

sudo pesign -n . -c cert -i vmlinuz.unsigned -o vmlinuz.signed -s

Enter Password or Pin for "NSS Certificate DB":
```

Copy the signed kernel back to `/boot`. Note that the copy command uses the `-b` option to create a backup of the original kernel image:

```
sudo cp -bf vmlinuz.signed /boot/vmlinuz-$(uname -r)
```

Updating the MOK Database

Note:

The following instructions apply to all kernel types. The MOK database enables you to insert keys into the trusted keys within the UEFI Shim, directly from user space. This method is generally easier than manually adding them to the UEFI Secure Boot Key Database. Either approach should work; however, by adding the keys to the MOK database, the instructions are not tied to particular hardware and there is no requirement to copy keys to storage that is accessible to UEFI.

For UEK R6 kernels prior to UEK R6U3 you must enroll the key that was used to sign the kernel image into the MOK database, while for all other kernels, the key that is used to sign the module itself is enrolled into the database. These keys might be the same, if you used the same key to sign the module and the kernel.

Because the key that you created is not included in the UEFI Secure Boot Key Database, you must enroll it into the MOK database in the Shim by using the `mokutil` command:

```
sudo mokutil --import /etc/ssl/certs/pubkey.der
```

The previous command prompts you for a single-use password that you use when the MOK Management service enrolls the key after you reboot the system.

Reboot the system.

The UEFI Shim should automatically start the `Shim UEFI key manager` at boot. Be sure to hit a key within 10 seconds to interrupt the boot process to enroll your MOK key.

1. Press any key to perform MOK Management.
2. Select `Enroll MOK` from the menu.
3. Select `View key 0` from the menu to display the key details.
4. Verify that the values presented match the key that you used to sign the module and that you inserted into the kernel image, then press any key to return to the `Enroll MOK` menu.
5. Select `Continue` from the menu.

   The `Enroll the key(s)?` screen is displayed.
6. Select `Yes` to enroll the key.
7. When prompted for a password, enter the password that you used when you imported the key by using the `mokutil` command.

   The key is enrolled within the UEFI Secure Boot key database.
8. When the `Perform MOK management` screen is displayed, select `Reboot` from the menu.

Signing the Kernel Module for Secure Boot

UEFI Secure Boot requires that kernel modules are signed with a trusted certificate to prevent attackers from installing and running unauthorized operating systems and malicious drivers.

Oracle signs kernel modules provided through supported software channels to verify their origin and integrity, but to install additional drivers you must create your own signing certificate, sign the relevant kernel module and confirm that you trust kernel modules that have been signed with that certificate.

Sign the Kernel Module

The `sign-file` utility ensures that the module is signed correctly for the kernel. This utility is provided within the kernel source. The instructions provided here assume that you are signing a module for the currently running kernel. If you intend to sign a module for a different kernel, you must provide the path to the `sign-file` utility within the correct kernel version source. If you do not use the correct utility, the signature type for your module may not align correctly with the expected signature type.

To sign the module, run the `sign-file` utility for your currently running kernel and provide it with the path to your private key and the public key that you created for the purpose of signing your modules:

```
sudo /usr/src/kernels/$(uname -r)/scripts/sign-file sha512 /etc/ssl/certs/
priv.key /etc/ssl/certs/pubkey.der /lib/modules/$(uname -r)/path/to/module.ko
```

Note that the module should already be installed into `/lib/modules/` and you need to provide the correct path to the module.

When the module is signed, you can check the signature by running the `modinfo` command, for example:

```
modinfo hello

filename:       /lib/modules/5.4.17-2036.103.3.1.el8uek.x86_64/extra/hello.ko
description:    Hello World Linux Kernel Module
author:         A.Developer
license:        GPL
srcversion:     D51FB4CF0B86314953EE797
depends:
retpoline:      Y
name:           hello
vermagic:       5.4.17-2036.103.3.1.el8uek.x86_64 SMP mod_unload modversions
sig_id:         PKCS#7
signer:         Module Signing Example Key
sig_key:        AB:2C:E3:AB:87:D9:9C:6A:31:B8:80:20:D4:92:25:F3:9A:26:DC
sig_hashalgo:   sha512
signature:   9F:B0:25:CB:14:C1:C7:10:7F:60:1E:E6:66:82:64:58:91:1F:01:A5:
        D9:03:1B:9C:2D:42:00:45:78:2B:FA:70:F8:C7:3B:1A:A2:42:00:09:
        33:E0:81:1D:C6:E6:46:A5:FE:8B:9F:8C:3D:4E:A1:3A:05:52:ED:F6:
        25:F9:88:98:D3:70:78:1D:7E:63:F3:73:C8:C8:14:C2:3A:52:B4:8F:
        4C:8D:80:D9:0D:24:F8:C9:B1:28:82:B6:A9:27:56:C6:86:80:25:A5:
        75:C8:78:A9:30:BD:01:4C:DD:43:7F:FD:41:98:2C:59:21:7D:39:17:
        EC:2C:C1:65:1D:95:F0:09:C7:F6:45:10:83:15:78:A2:EE:D4:73:79:
        B2:F0:57:C1:96:B3:4C:43:B8:D1:87:94:50:61:D6:EC:50:2B:6A:6C:
        5C:C1:3E:8C:CB:6F:19:DC:EF:6C:12:07:03:99:B7:B3:22:0B:F6:AC:
        CB:40:C6:34:15:EA:1F:88:D4:4E:1C:87:2D:5A:92:F7:12:A6:E7:91:
        B3:80:AA:80:8F:49:B7:F0:F0:97:05:09:7A:65:30:4A:AE:10:BE:9F:
        6A:E4:B2:24:BE:1A:21:D0:F6:15:05:DA:2C:64:EA:B2:8E:AC:6F:18:
        40:65:21:F6:AA:17:31:AE:3F:3A:43:DB:A8:BC:71:79:EF:11:18:DE:
        86:EE:74:2A:E0:44:FC:B3:FF:CB:CB:F0:CA:BD:7B:A1:57:84:D8:A6:
        91:E5:B8:EF:1B:8A:63:16:43:03:AF:C4:C7:BF:52:9A:A9:23:75:C6:
        42:54:69:4E:3D:51:56:5A:9D:9B:C7:11:5E:9A:30:87:5F:F3:5E:C3:
        AE:2C:1D:6F:C9:4D:15:E5:CF:EC:46:0E:EF:D9:BB:2F:DF:DF:54:EA:
        F3:B6:9C:A3:6F:80:19:B9:DF:FA:2A:30:4E:2E:70:74:11:F9:5C:F6:
        EE:1A:DF:86:C4:2B:36:7E:B4:A4:D4:7E:30:19:1A:D1:92:D3:A7:FB:
        53:BF:67:C3:65:9E:4B:92:F0:6C:D4:6C:05:9B:0F:BF:D1:5B:CB:86:
        AE:68:00:AE:43:53:8B:7D:7E:18:20:CD:65:68:6C:4A:0D:93:A4:54:
        09:39:9C:D3:BD:CD:17:B6:8A:D3:62:0C:CA:A8:FD:1A:52:CE:29:A0:
        93:BF:AD:D2:58:3F:EA:4E:4B:50:31:6F:F6:B2:1E:87:C4:0A:9D:E4:
        43:E9:C7:CA:E9:CB:EF:A6:61:5B:DA:01:33:37:66:DB:16:8D:7C:D7:
        30:39:57:D4:0C:1A:54:AE:91:7B:FE:35:10:CC:34:03:99:EA:5A:57:
        E0:95:61:02:42:95:A2:F5:2E:72:30:95
```

Updating the MOK Database with the Kernel Module Certificate

Note:

The following instructions apply to all kernel module types. The MOK database enables you to insert keys into the trusted keys within the UEFI Shim, directly from user space. This method is generally easier than manually adding them to the UEFI Secure Boot Key Database. Either approach should work; however, by adding the keys to the MOK database, the instructions are not tied to particular hardware and there is no requirement to copy keys to storage that is accessible to UEFI.

You do not need to perform this step for kernel modules that are used with any UEK R6 kernels prior to UEK R6U3. For more information, see Setting Kernel Module Certificate Trust for UEK R6.

Because the key that you created is not included in the UEFI Secure Boot Key Database, you must enroll it into the MOK database in the Shim by using the `mokutil` command:

```
sudo mokutil --import /etc/ssl/certs/pubkey.der
```

The previous command prompts you for a single-use password that you use when the MOK Management service enrolls the key after you reboot the system.

Reboot the system.

The UEFI Shim should automatically start the `Shim UEFI key manager` at boot. Be sure to hit a key within 10 seconds to interrupt the boot process to enroll your MOK key.

1. Press any key to perform MOK Management.
2. Select `Enroll MOK` from the menu.
3. Select `View key 0` from the menu to display the key details.
4. Verify that the values presented match the key that you used to sign the module and that you inserted into the kernel image, then press any key to return to the `Enroll MOK` menu.
5. Select `Continue` from the menu.

   The `Enroll the key(s)?` screen is displayed.
6. Select `Yes` to enroll the key.
7. When prompted for a password, enter the password that you used when you imported the key by using the `mokutil` command.

   The key is enrolled within the UEFI Secure Boot key database.
8. When the `Perform MOK management` screen is displayed, select `Reboot` from the menu.

Setting Kernel Module Certificate Trust for UEK R6

Note:

The following instructions apply to UEK R6 releases prior to UEK R6U3. The secure boot implementation in UEK R6U3 is updated to additionally trust modules signed using platform certificates that are available in the UEFI and MOK databases.

Inserting the Module Certificate in the Kernel and Signing the Kernel Image

After a module has been signed, the key that was used to sign it must be inserted into the compiled kernel image, because UEK R6 only trusts modules that are signed with keys that are listed in the kernel builtin, trusted keyring. Because the kernel image is modified, it must be signed again and the certificate that is used to sign the kernel must then be added to the UEFI or MOK database. If your modules are already signed by a vendor, these certificates may be different. The following steps apply to the kernel signing certificate and are only required if you are using a UEK R6 kernel prior to UEK R6U3.

It is useful to perform the following steps in the same directory that your certificates are stored. Run these commands as the `root` user.

```
cd /etc/ssl/cert
```

Insert the Module Certificate in the Kernel Image

Use the `insert-sys-cert` utility provided in the kernel source to insert the raw DER certificate that was used to sign the module into the compressed kernel boot image:

```
sudo /usr/src/kernels/$(uname -r)/scripts/insert-sys-cert -s /boot/System.map-$(uname -
r) -z /boot/vmlinuz-$(uname -r) -c pubkey.der

INFO:    Executing: gunzip <vmlinux-8ig4vO >vmlinux-QyW44r
WARNING: Could not find the symbol table.
INFO:    sym:    system_extra_cert
INFO:    addr:   0xffffffff82891616
INFO:    size:   8194
INFO:    offset: 0x1c91616
INFO:    sym:    system_extra_cert_used
INFO:    addr:   0xffffffff82893618
INFO:    size:   0
INFO:    offset: 0x1c93618
INFO:    sym:    system_certificate_list_size
INFO:    addr:   0xffffffff82893620
INFO:    size:   0
INFO:    offset: 0x1c93620
INFO:    Inserted the contents of pubkey.der into ffffffff82891616.
INFO:    Used 1481 bytes out of 8194 bytes reserved.
INFO:    Executing: gzip -n -f -9 <vmlinux-QyW44r >vmlinux-M5uNR6
```

Important:

Only a single custom certificate can be added to the kernel because the compressed size of the kernel's boot image cannot increase. Do *not* attempt to add multiple certificates to the kernel boot image.

Sign the Kernel Image

Use the `pesign` utility to remove the existing PE signature and resign the kernel with the new one by using the kernel signing key that is stored in the NSS database. Note that you are prompted for the password of the NSS certificate database that you created in Configure an NSS Database.

```
sudo pesign -u 0 -i /boot/vmlinuz-$(uname -r) --remove-signature -o
vmlinuz.unsigned

sudo pesign -n . -c cert -i vmlinuz.unsigned -o vmlinuz.signed -s

Enter Password or Pin for "NSS Certificate DB":
```

Copy the signed kernel back to `/boot`. Note that the copy command uses the `-b` option to create a backup of the original kernel image:

```
sudo cp -bf vmlinuz.signed /boot/vmlinuz-$(uname -r)
```

Updating the MOK Database

Note:

The following instructions apply to all kernel types. The MOK database enables you to insert keys into the trusted keys within the UEFI Shim, directly from user space. This method is generally easier than manually adding them to the UEFI Secure Boot Key Database. Either approach should work; however, by adding the keys to the MOK database, the instructions are not tied to particular hardware and there is no requirement to copy keys to storage that is accessible to UEFI.

For UEK R6 kernels prior to UEK R6U3 you must enroll the key that was used to sign the kernel image into the MOK database, while for all other kernels, the key that is used to sign the module itself is enrolled into the database. These keys might be the same, if you used the same key to sign the module and the kernel.

Because the key that you created is not included in the UEFI Secure Boot Key Database, you must enroll it into the MOK database in the Shim by using the `mokutil` command:

```
sudo mokutil --import /etc/ssl/certs/pubkey.der
```

The previous command prompts you for a single-use password that you use when the MOK Management service enrolls the key after you reboot the system.

Reboot the system.

The UEFI Shim should automatically start the `Shim UEFI key manager` at boot. Be sure to hit a key within 10 seconds to interrupt the boot process to enroll your MOK key.

1. Press any key to perform MOK Management.
2. Select `Enroll MOK` from the menu.
3. Select `View key 0` from the menu to display the key details.
4. Verify that the values presented match the key that you used to sign the module and that you inserted into the kernel image, then press any key to return to the `Enroll MOK` menu.
5. Select `Continue` from the menu.

   The `Enroll the key(s)?` screen is displayed.
6. Select `Yes` to enroll the key.
7. When prompted for a password, enter the password that you used when you imported the key by using the `mokutil` command.

   The key is enrolled within the UEFI Secure Boot key database.
8. When the `Perform MOK management` screen is displayed, select `Reboot` from the menu.

Validating That a Key Is Trusted

After the system is booted, you can validate whether a key is included in the appropriate kernel keyring. Validation depends on the kernel version that you are running. Also, the keyring name that you need to check varies, as the implementation has changed across kernel versions.

If the key that was generated for the purpose of signing custom modules is listed within the correct keyring, you are able to load modules that are signed with this key while in Secure Boot mode.

UEK R7 Releases

UEK R7 introduces the `.machine` keyring. Keys within the `.machine` keyring are trusted for use within the Oracle Linux kernel. All of the MOK keys are loaded into this keyring at boot.

Just like the `.builtin_trusted_keys` keyring, the new `.machine` keyring is linked to the `.secondary_trusted_keys` keyring. With this link in place, the `.machine` keyring is referenced when the kernel uses the `.secondary_trusted_keys` keyring to validate a signed kernel module, as illustrated by the following example:

```
sudo keyctl show %:.secondary_trusted_keys

Keyring
 772746105 ---lswrv      0     0  keyring: .secondary_trusted_keys
 252396885 ---lswrv      0     0   \_ keyring: .builtin_trusted_keys
 660166481 ---lswrv      0     0   |   \_ asymmetric: Oracle CA Server:
702a35b0d12005e5010c0614f7b8abf7c5bd5f73
  86702374 ---lswrv      0     0   |   \_ asymmetric: Oracle IMA signing CA:
a2f28976a05984028f7d1a4904ae14e8e468e551
 247354640 ---lswrv      0     0   |   \_ asymmetric: Oracle America, Inc.: Ksplice
Kernel Module Signing Key: 09010ebef5545fa7c54b626ef518e077b5b1ee4c
 264616160 ---lswrv      0     0   |   \_ asymmetric: Oracle Linux Kernel Module
```

```
Signing Key: 2bb352412969a3653f0eb6021763408ebb9bb5ab
 772320403 ---lswrv      0     0   \_ keyring: .machine
 450491670 ---lswrv      0     0       \_ asymmetric: Oracle America, Inc.:
7c552922286d66bcb33c53d7ee0f1cd716ecdc63
 100307441 ---lswrv      0     0       \_ asymmetric: Oracle America, Inc.:
39bff3f0f578f26e527321cafda2a9cdbd71143c
 688922247 ---lswrv      0     0       \_ asymmetric: Oracle America, Inc.:
4ff35c3e09ce586fa776d56468d86b022af272f1
```

RHCK on Oracle Linux 8, RHCK on Oracle Linux 9 and UEK R6U3 or Later Updates

For RHCK on Oracle Linux 8, RHCK on Oracle Linux 9 and UEK R6U3 kernels or later, keys within both the `builtin_trusted_keys` keyring and the `platform` keyring are trusted for both module signing and for the `kexec` tools, which means you can follow the standard procedure to sign a module and add it to the MOK database for the key to appear in the `platform` keyring and it is automatically trusted.

Because it is possible that a key can be loaded into the `builtin_trusted_keys` keyring, you should check both keyrings for the module signing key, for example:

```
sudo keyctl show %:.builtin_trusted_keys

Keyring
 441234704 ---lswrv      0     0  keyring: .builtin_trusted_keys
 798307349 ---lswrv      0     0   \_ asymmetric: Oracle CA Server:
32a7ceb6c56614c69b4729b455254bfaf09569a4
 277992501 ---lswrv      0     0   \_ asymmetric: Oracle Linux RHCK Module
Signing Key: dd995b155c19b3a7c3ef7707b969e25f9639666e
1000618915 ---lswrv      0     0   \_ asymmetric: Red Hat Enterprise Linux
kpatch signing key: 4d38fd864ebe18c5f0b72e3852e2014c3a676fc8
 199403819 ---lswrv      0     0   \_ asymmetric: Red Hat Enterprise Linux
Driver Update Program (key 3): bf57f3e87362bc7229d9f465321773dfd1f77a80


sudo keyctl show %:.platform

Keyring
 705628740 ---lswrv      0     0  keyring: .platform
  89698906 ---lswrv      0     0   \_ asymmetric: Microsoft Corporation UEFI CA
2011: 13adbf4309bd82709c8cd54f316ed522988a1bd4
 497244381 ---lswrv      0     0   \_ asymmetric: Oracle America, Inc.:
d6ee3a06a222bf4244b8986a531046e59c14eeef
 710039804 ---lswrv      0     0   \_ asymmetric: Oracle America, Inc.:
c65d1d746ae4cb127762e1dbd7ade48215703c5c
 730271863 ---lswrv      0     0   \_ asymmetric: Oracle America Inc.:
2e7c1720d1c5df5254cc93d6decaa75e49620cf8
 535985802 ---lswrv      0     0   \_ asymmetric: Oracle America, Inc.:
795c5945e7cb2b6773b7797571413e3695062514
 607819007 ---lswrv      0     0   \_ asymmetric: Oracle America, Inc.:
f9aec43f7480c408d681db3d6f19f54d6e396ff4
  99739320 ---lswrv      0     0   \_ asymmetric: Oracle America, Inc.:
430c85cb8b531c3d7b8c44adfafc2e5d49bb89d4
 231916335 ---lswrv      0     0   \_ asymmetric: Microsoft Windows Production
PCA 2011: a92902398e16c49778cd90f99e4f9ae17c55af53
 866576656 ---lswrv      0     0   \_ asymmetric: Oracle Linux Test Certificate:
d30dffa37bec20ecfb1d3caee53cd746282e8cad
 230958440 ---lswrv      0     0   \_ asymmetric: Module Signing Example Key:
a43b4e638874b0656db2bc26216f56c0ac39f72b
```

UEK R6 releases prior to UEK R6U3

For UEK R6 releases prior to UEK R6U3, only those keys that are listed in the kernel `builtin_trusted_keys` keyring are trusted for module signing. For this reason, module signing keys are added to the kernel image as part of the process for signing modules.

To check this keyring to determine whether the module signing key that you created is listed:

```
sudo keyctl show %:.builtin_trusted_keys

Keyring
 892034081 ---lswrv      0     0  keyring: .builtin_trusted_keys
 367808024 ---lswrv      0     0   \_ asymmetric: Oracle CA Server:
fbcd3d4d950c6b2b0e01f0a146c5a4e3855ae704
 230958440 ---lswrv      0     0   \_ asymmetric: Module Signing Example Key:
a43b4e638874b0656db2bc26216f56c0ac39f72b
 408597579 ---lswrv      0     0   \_ asymmetric: Oracle America, Inc.: Ksplice Kernel
Module Signing Key: 09010ebef5545fa7c54b626ef518e077b5b1ee4c
 839574974 ---lswrv      0     0   \_ asymmetric: Oracle Linux Kernel Module Signing
Key: 2bb352412969a3653f0eb6021763408ebb9bb5ab
```

UEK R5

For UEK R5, the builtin kernel keys, the UEFI Secure Boot database keys, and the MOK database keys all appear in the kernel keyring, which is named `secondary_trusted_keys`. Keys in this keyring are trusted for module signing and also for the `kexec` tools.

To check this keyring to determine whether the module signing key that you created is listed:

```
sudo keyctl show %:.secondary_trusted_keys

Keyring
 847333689 --alswrv      0     0  keyring: .secondary_trusted_keys
 304199016 --alswrv      0     0   \_ keyring: .builtin_trusted_keys
1008872915 --alswrv      0     0   |   \_ asymmetric: Oracle CA Server:
911ccad43f8dcc5e169cf7d89af9d28edd7e68aa
 252474061 --alswrv      0     0   |   \_ asymmetric: Oracle America, Inc.: Ksplice
Kernel Module Signing Key: 09010ebef5545fa7c54b626ef518e077b5b1ee4c
 164249372 --alswrv      0     0   |   \_ asymmetric: Oracle Linux Kernel Module
Signing Key: 2bb352412969a3653f0eb6021763408ebb9bb5ab
 567204509 --alswrv      0     0   \_ asymmetric: Oracle America Inc.:
7b13dad59daa6adeacfff53c9a3b7e4f61fe068a
 202164133 --alswrv      0     0   \_ asymmetric: Oracle America Inc.:
9c68fc11b0f85979388df4c8cc2c9c098928eda9
 600928065 --alswrv      0     0   \_ asymmetric: Oracle America Inc.:
fe1ea0d2db0446565aec7d637e2509c9900398c1
1023599957 --alswrv      0     0   \_ asymmetric: Oracle America Inc.:
bc5971954c749d3d309852ef0f3c236fa498e8f6
 837499579 --alswrv      0     0   \_ asymmetric: Microsoft Corporation UEFI CA 2011:
13adbf4309bd82709c8cd54f316ed522988a1bd4
 230958440 ---lswrv      0     0   \_ asymmetric: Module Signing Example Key:
a43b4e638874b0656db2bc26216f56c0ac39f72b
 260150072 --alswrv      0     0   \_ asymmetric: Microsoft Windows Production PCA
2011: a92902398e16c49778cd90f99e4f9ae17c55af53
```

UEK R4 and RHCK on Oracle Linux 7

For UEK R4 and RHCK (3.10.0) on Oracle Linux 7 the builtin kernel keys, the UEFI Secure Boot database keys, and the MOK database keys all appear in the kernel keyring, which is named `system_keyring`. Keys in this keyring are trusted for module signing and also for the `kexec` tools.

To check this keyring to determine whether the module signing key that you created is listed:

```
sudo keyctl show %:.system_keyring

Keyring
 490420691 --alswrv      0     0  keyring: .system_keyring
 985562533 --alswrv      0     0   \_ asymmetric: Oracle America, Inc.: Ksplice
Kernel Module Signing Key: 09010ebef5545fa7c54b626ef518e077b5b1ee4c
 117527109 --alswrv      0     0   \_ asymmetric: Oracle America Inc.:
7b13dad59daa6adeacfff53c9a3b7e4f61fe068a
 752001237 --alswrv      0     0   \_ asymmetric: Oracle America Inc.:
9c68fc11b0f85979388df4c8cc2c9c098928eda9
 405618281 --alswrv      0     0   \_ asymmetric: Oracle America Inc.:
fe1ea0d2db0446565aec7d637e2509c9900398c1
1018396318 --alswrv      0     0   \_ asymmetric: Oracle America Inc.:
bc5971954c749d3d309852ef0f3c236fa498e8f6
 264415311 --alswrv      0     0   \_ asymmetric: Microsoft Corporation UEFI CA
2011: 13adbf4309bd82709c8cd54f316ed522988a1bd4
 811847839 --alswrv      0     0   \_ asymmetric: Microsoft Windows Production
PCA 2011: a92902398e16c49778cd90f99e4f9ae17c55af53
  38542369 --alswrv      0     0   \_ asymmetric: Oracle CA Server:
7efd70c89ee321c6be7098cd17fe7f4eded76a9e
 230958440 ---lswrv      0     0   \_ asymmetric: Module Signing Example Key:
a43b4e638874b0656db2bc26216f56c0ac39f72b
 285314597 --alswrv      0     0   \_ asymmetric: Oracle Linux Kernel Module
Signing Key: 2bb352412969a3653f0eb6021763408ebb9bb5ab
```