National Security Agency
Cybersecurity Technical Report

# UEFI Secure Boot Customization

March 2023 ver. 1.2

S/N: U/OO/168873-20
PP-23-0464

# Notices and history

## *Document change history*

| Date | Version | Description |
|---|---|---|
| 15 September 2020 | 1.0 | Publication release. |
| 16 September 2020 | 1.1 | Updated server UEFI hash interface image and text. |
| 14 March 2023 | 1.2 | Updated DB and DBX hash calculation information in section 4.3.3 to correctly handle EFI (PE/EFL) format. |

## *Disclaimer of warranties and endorsement*

The information and opinions contained in this document are provided "as is" and without any warranties or guarantees. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government, and shall not be used for advertising or product endorsement purposes.

## *Trademark recognition*

Dell, EMC, Dell EMC, iDRAC, Optiplex, and PowerEdge are registered trademarks of Dell, Inc.

HP, HPE, HP Enterprise, iLO, and ProLiant are registered trademarks of Hewlett-Packard Company.

Linux is a registered trademark of Linus Torvolds.

Microsoft, Hyper-V, Surface, and Windows are registered trademarks of Microsoft Corporation.

Red Hat, Red Hat Enterprise Linux (RHEL), CentOS, and Fedora are registered trademarks of Red Hat, Inc.

VMware and ESXI are registered trademarks of VMware, Inc.

Trusted Computing Group, TCG, Trusted Platform Module, TPM, and related specifications are property of the Trusted Computing Group.

Unified Extensible Firmware Interface, UEFI, UEFI Forum, and related specifications are property of the UEFI Forum.

# Publication information

## *Author(s)*

National Security Agency
Cybersecurity Directorate
Endpoint Security Division
Platform Security Section

## *Contact information*

Client Requirements / General Cybersecurity Inquiries:
Cybersecurity Requirements Center, 410-854-4200, Cybersecurity_Requests@nsa.gov

Media inquiries / Press Desk:
Media Relations, 443-634-0721, MediaRelations@nsa.gov

## *Purpose*

This document was developed in furtherance of NSA's cybersecurity missions. This includes its responsibilities to identify and disseminate threats to National Security Systems, Department of Defense information systems, and the Defense Industrial Base, and to develop and issue cybersecurity specifications and mitigations. This information may be shared broadly to reach all appropriate stakeholders.

## *Additional resources*

Please visit the NSA Cybersecurity GitHub at https://www.github.com/nsacyber/Hardware-and-Firmware-Security-Guidance for additional resources relating to UEFI Secure Boot and the customization process.

## Executive summary

Secure Boot is a boot integrity feature that is part of the Unified Extensible Firmware Interface (UEFI) industry standard. Most modern computer systems are delivered to customers with a standard Secure Boot policy installed. This document provides a comprehensive guide for customizing a Secure Boot policy to meet several use cases.

UEFI is a replacement for the legacy Basic Input Output System (BIOS) boot mechanism. UEFI provides an environment common to different computing architectures and platforms. UEFI also provides more configuration options, improved performance, enhanced interfaces, security measures to combat persistent firmware threats, and support for a wider variety of devices and form factors.

Malicious actors target firmware to persist on an endpoint. Firmware is stored and executes from memory that is separate from the operating system and storage media. Antivirus software, which runs after the operating system has loaded, is ineffective at detecting and remediating malware in the early-boot firmware environment that executes before the operating system. Secure Boot provides a validation mechanism that reduces the risk of successful firmware exploitation and mitigates many published early-boot vulnerabilities.

Secure Boot is frequently not enabled due to issues with incompatible hardware and software. Custom certificates, signatures, and hashes should be utilized for incompatible software and hardware. Secure Boot can be customized to meet the needs of different environments. Customization enables administrators to realize the benefits of boot malware defenses, insider threat mitigations, and data-at-rest protections. Administrators should opt to customize Secure Boot rather than disable it for compatibility reasons. Customization may – depending on implementation – require infrastructures to sign their own boot binaries and drivers.

Recommendations for system administrators and infrastructure owners:

- Machines running legacy BIOS or Compatibility Support Module (CSM) should be migrated to UEFI native mode.
- Secure Boot should be enabled on all endpoints and configured to audit firmware modules, expansion devices, and bootable OS images (sometimes referred to as Thorough Mode).
- Secure Boot should be customized, if necessary, to meet the needs of organizations and their supporting hardware and software.
- Firmware should be secured using a set of administrator passwords appropriate for a device's capabilities and use case.
- Firmware should be updated regularly and treated as importantly as operating system and application updates.
- A Trusted Platform Module (TPM) should be leveraged to check the integrity of firmware and the Secure Boot configuration.

# Contents

**Notices and history** ..... ii
Document change history ..... ii
Disclaimer of warranties and endorsement ..... ii
Trademark recognition ..... ii
**Publication information** ..... iii
Author(s) ..... iii
Contact information ..... iii
Purpose ..... iii
Additional resources ..... iii
**Executive summary** ..... iv
**Contents** ..... v
**1 Unified Extensible Firmware Interface (UEFI)** ..... 1
**2 UEFI Secure Boot** ..... 2
2.1 Platform-Specific Caveats ..... 4
**3 Use Cases For Secure Boot** ..... 5
3.1 Anti-Malware ..... 5
3.2 Insider Threat Mitigation ..... 6
3.3 Data-at-Rest ..... 7
**4 Customization** ..... 7
4.1 Dependencies ..... 7
4.2 Backup Factory Values ..... 8
4.2.1 Backup Secure Boot Values ..... 9
4.2.2 EFI Signature List (ESL) Format ..... 11
4.3 Initial Provisioning of Certificates and Hashes ..... 12
4.3.1 Create Keys and Certificates ..... 13
4.3.2 Sign Binaries ..... 14
4.3.3 Calculate and Capture Hashes ..... 15
4.3.4 Load Keys and Hashes ..... 17
4.4 Updates and Changes ..... 22
4.4.1 Update the PK ..... 22
4.4.2 Update a KEK ..... 22
4.4.3 Update the DB or DBX ..... 23
4.4.4 Update MOK or MOKX ..... 23
4.5 Validation ..... 23
**5. Advanced Customizations** ..... 24
5.1 Trusted Platform Module (TPM) ..... 24
5.2 Trusted Bootloader ..... 26
**6 References** ..... 27
6.1 Cited Resources ..... 27
6.2 Command References ..... 27
6.3 Uncited Related Resources ..... 27

**7 Appendix ..... 28**
7.1 UEFI Lockdown Configuration ..... 28
7.2 Acronyms ..... 30
7.3 Frequently Asked Questions (FAQ) ..... 32

# 1 Unified Extensible Firmware Interface (UEFI)

Unified Extensible Firmware Interface (UEFI) is an interface that exists between platform hardware and software. UEFI is defined and updated via specifications maintained by the UEFI Forum industry body. Support for UEFI is a requirement for some newer software and hardware. Legacy boot solutions, such as Basic Input/Output System (BIOS), are being phased out in 2020 (Shilov 2017).

UEFI defines a consistent Application Programming Interface (API) and a set of environment variables common to all UEFI platforms. Uniformity enables OS, driver, and application developers to build for UEFI regardless of platform, architecture, vendor, or assortment of system components. Manufacturers and developers can take advantage of UEFI's extensibility to create additional features, add new product support, and create protocols to support emerging solutions.

Legacy BIOS involves a wide variety of unique implementations, update solutions, and interpretations of platform services (e.g. Advanced Configuration and Power Interface (ACPI)). UEFI establishes a standard that separates portions of code into modules, defines mechanisms for module interaction, and empowers component vendors to reuse modules across product lines. Modules also enable vendors to swap out content via updates that can be delivered remotely over commercial infrastructure management and update solutions (Golden 2017).

UEFI boot occurs in standards-defined phases (UEFI Forum 2017). Figure 1 shows an overview of the phases. The SEC, PEI, DXE, and BDS phases are handled by platform firmware. The Bootloader and OS Kernel phases are handled by software.

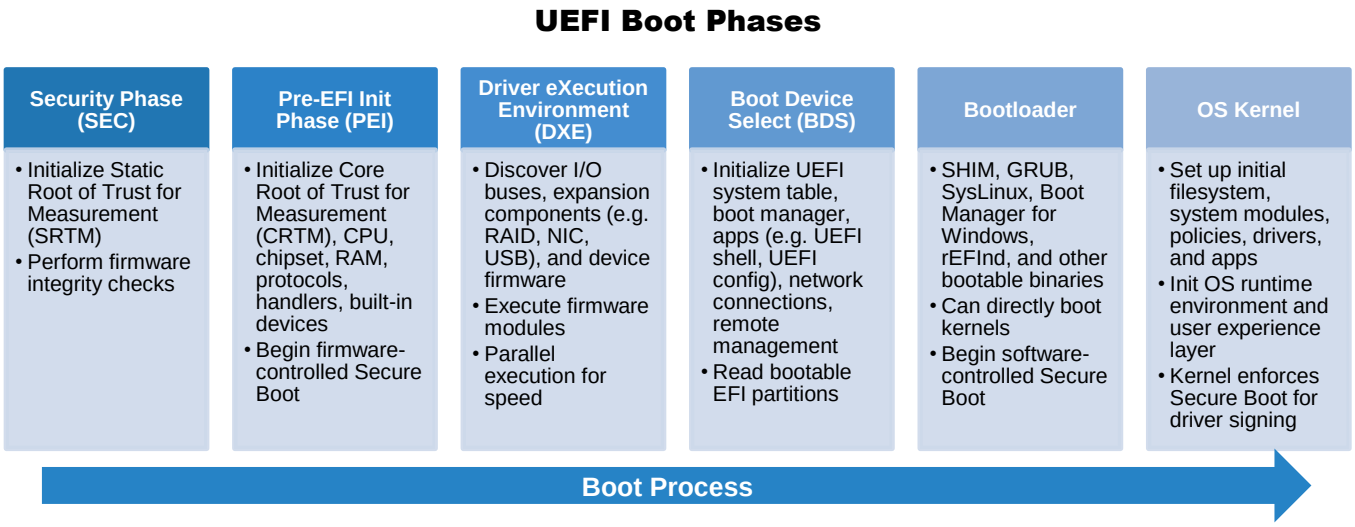



Figure 1 - An enumeration of UEFI firmware and software boot phases.

Legacy BIOS has been part of the computing ecosystem since 1975. UEFI entered the standards and commercial world in 2005 after having existed as an internal Intel Corporation project for many years prior (referred to as Extensible Firmware Interface – EFI). The UEFI Forum and vendor partners recognized the potential for disruption migrating from BIOS to UEFI would cause on the computing industry and established products. Therefore, UEFI implementations historically have offered the following operating modes to meet customer needs:

- **UEFI Native Mode** is UEFI without any accommodation for legacy devices. UEFI makes changes to the way devices and components execute their firmware and access system

resources as compared to older BIOS implementations. Native mode implements pure UEFI and requires devices, components, and software to be UEFI-ready. UEFI Native Mode is a requirement for utilizing UEFI Secure Boot.

- **Legacy BIOS Mode or Compatibility Support Module (CSM)** are accommodations for devices and components that are not designed for use with UEFI. BIOS behavior is emulated to allow devices incompatible with UEFI's architectural and access control paradigms to be used on modern systems. Leveraging legacy mode or CSM reintroduces security, access control, and memory vulnerabilities addressed by the UEFI standard and prohibits the use of UEFI Secure Boot.

## 2 UEFI Secure Boot

Secure Boot is a feature added to UEFI specification 2.3.1. Each binary (module, driver, kernel, etc.) used during boot must be validated before execution. Validation involves checking for the presence of a signature that can be validated by a certificate or by computing a SHA-256 hash that matches a trusted hash. Several value stores are used to identify content that is trusted or untrusted. Figure 2 shows the sequence of checks. The value stores are:

- **Platform Key (PK)** is the master hierarchy key certificate. Only one PK may exist on the system as a RSA-2048 public key certificate. In the most secure usage, PKs are unique per endpoint and maintained by the endpoint owner or infrastructure operators. The PK private key can sign UEFI environment variable changes or KEK, DB, and DBX changes that can be validated by the PK certificate. The PK cannot be used for signing executable binaries that are checked at boot time. Keep the PK private key secure and store it on a different device.
    - **Note:** A PK certificate must be in place for Secure Boot to begin enforcement. Some vendors ship devices with random PKs or a common/shared PK. Endpoint owners may also install their own PK as part of the customization process. Carefully consider the balance between administrative overhead and security. **A unique PK per endpoint provides greater security against UEFI compromise across an infrastructure, but may reduce the speed at which administrators can deploy changes compared to a common/shared PK.**
- **Key Exchange Keys (KEKs)** are normally used by vendors, such as the system vendor and the OS vendor, who have a need to update the DB or DBX. One or more KEKs are typically present on a system as RSA-2048 public key certificates. Different endpoints may have the same KEK(s) – they are not unique to an endpoint. KEKs may sign changes to the DB and DBX. KEKs can also be used to sign bootable content. However, replacing a KEK is difficult because involvement from the PK is required. Therefore, KEKs should only be used to make changes to the DB and DBX. Remember to keep the KEK private key secure.
- **Allow list Database (DB)** can contain SHA-256 hashes or RSA 2048 public key certificates. Binaries that have signatures that can be validated by a certificate will be allowed to execute at boot time. Likewise, binaries with computed SHA-256 hashes that match a trusted hash will also be allowed to boot even in the absence of a signature.
- **Deny list Database (DBX)** can contain SHA-256 hashes or RSA 2048 public key certificates. **The DBX has ultimate veto power at boot time.** Any binary hash that matches a DBX hash or has a signature verified by a DBX certificate will be prevented from executing at boot time. DBX is normally leveraged to target errantly signed binaries

such as malware or debug bootloaders. **DBX is normally checked first (except when MOKX is present; see below).**

- **Machine Owner Key (MOK)** is not part of the UEFI Secure Boot standard. MOK is used by Linux implementations. MOK functions identically to the DB and becomes initialized by the pre-bootloader Shim. Linux distributions utilize MOK keys to sign their own binaries rather than utilizing the process of having Microsoft or Original Equipment Manufacturers (OEM) sign every update or variation. Shim is signed by Microsoft and therefore works on most computers supporting Secure Boot Standard Mode. MOK can store SHA-256 hashes and RSA public key certificates. Some Linux kernels leverage MOK for driver signing checks instead of or in addition to DB, DBX, and KEK.
- **Machine Owner Key Deny list (MOKX)** is also not part of the UEFI Secure Boot standard. MOKX exists in Linux implementations and functions like the DBX. The bootloader Shim is responsible for initializing MOKX. Some Linux kernels leverage MOKX for driver signing checks instead of or in addition to DBX. **MOKX is normally checked first when present – even before the DBX.**

Figure 2 shows the order of operations during UEFI Secure Boot checks. MOKX and DBX are checked first since they have absolute veto power. If no match is made after checking the KEK(s), a binary is assumed to be untrusted. Reaching a denied (or unknown/no match) state only blocks the object that was checked – boot continues for other binaries.

Figure 2 - Order of operations during UEFI Secure Boot checks. Checks contained within dashed lines only take place when the Shim bootloader is used AND after its initialization in the UEFI bootloader phase (i.e. firmware OROMs are not checked against MOKX and MOK; kernels are checked against MOKX and MOK).

Vendor implementations of Secure Boot typically have the first three operating modes:

- **Standard Mode** enforces signature and hash checks on boot time executables. Standard mode is the default configuration for most modern computers, particularly those shipping with Microsoft Windows installed. A Microsoft KEK and pair of Microsoft DB certificates – one for validating Microsoft products and another for products evaluated by Microsoft – make up the minimal Standard Mode configuration. DBX hashes representing errantly signed or revoked boot time binaries are also typically included. System vendors may include their own KEK and/or DB certificate. Standard Mode supports many versions of Windows, Linux distributions, and a wide variety of hardware and software solutions.
    - **Note:** Switching to Standard Mode may set Secure Boot to factory default values and remove any custom values.
- **User/Custom Mode** also enforces signature and hash checks on boot time executables. However, unlike Standard Mode, Custom Mode allows the system owner to narrow or expand the selection of trusted hardware and software solutions by changing the contents of the Secure Boot PK, KEK, DB, and/or DBX data stores. Endpoint administrators may append new certificates and hashes to Secure Boot, or they may also remove, replace, or clear existing certificates and hashes. Custom Mode allows endpoints to be configured to

trust a narrow selection of hardware and software trusted by the owner, or expand upon the Standard Mode ecosystem.

- **Disabled Mode** does not utilize Secure Boot validation, so any well-structured EFI binary will execute at boot regardless of hashes or signatures. Disabled mode is the default in Legacy or Compatibility Support Module (CSM) modes.
- **Setup Mode** may be an option while a system does not have a PK installed. Setup mode typically allows for KEK, DB, and DBX values to be readily manipulated as the system owner "claims ownership" of the Secure Boot implementation. Establishing a PK will drop the system out of Setup Mode and into User/Custom Mode at the next boot.
- **Audit Mode** may be an option to gather debugging information about the results of Secure Boot checks. Administrators can see what parts of the boot process were validated, what the validation results were, and identify problems with boot components and policies to tailor implementation to their mission security needs.
- **Deployed Mode** may be an option which enforces the current Secure Boot configuration without the distinction of Standard vs User/Custom configuration. Values loaded into Secure Boot policy are enforced as is. The system does not distinguish between the factory default Standard values and User/Custom values.

Platform firmware performs boot signature checking up to the bootloader. Software components that participate in the boot process, such as the bootloader, kernel, initial file system, drivers, kernel modules, policies, and more, must continue the signature checking scheme in software. In Microsoft Windows, signature checking is performed by the Windows Boot Manager and Windows kernel. In Red Hat Enterprise Linux (RHEL), signature checking is performed by Shim, GRUB, and the Linux kernel. Red Hat utilizes a MOK stored in a Microsoft-signed build of Shim to validate GRUB, the kernel, drivers, and other binaries.

### *2.1 Platform-Specific Caveats*

The extent to which Secure Boot validates the boot process varies based on platform and boot configuration. In general, most enterprise UEFI implementations provide the following options:

- **Thorough or Full Boot** provides the maximum amount of protection by using Secure Boot throughout the boot process. Integrity, signature, and hash checks are performed. All authorized firmware binaries are executed. Alerts may be generated for hardware changes, chassis intrusions, and component states. The Thorough Boot option is typically the default behavior on servers, storage arrays, and blades. Thorough Boot prioritizes security over speed. Boot time takes the longest in Thorough Boot.
- **Fast Boot or Minimal Boot** minimizes boot time by skipping numerous checks, which may or may not include Secure Boot checks. Boot speed is prioritized over some security features and/or additional features and peripheral support at boot time. Malware like LoJax can slip by on some systems (Schlej 2018). Fast Boot is normally found enabled on consumer devices. When Fast Boot is a configurable toggle, disabling Fast Boot typically results in Thorough Boot.
    - **Note:** Fast/Minimal boot may behave differently depending on system vendor, and also vary across a single vendor's product line. A business-class desktop or server may perform all Secure Boot checks in Fast/Minimal while a consumer-oriented tablet or notebook from the same vendor skips checks.
- **Automatic Boot** attempts to detect when changes have occurred to the early stages of

UEFI boot. Automatic Boot invokes Fast/Minimal Boot when no changes are detected. Thorough/Full Boot is invoked once after each significant change is detected. Changing firmware, changing hardware, bootloader updates, or toggling options in UEFI configuration may be sufficient to trigger Thorough/Full Mode on the next boot.

**Always prefer the thorough or full boot option when unsure of the vendor implementation.** Fast, minimal, and automatic may miss changes that could compromise system integrity – again, depending on vendor implementation.

Some vendors also allow the use of Compatibility Support Module (CSM) Legacy Mode if Secure Boot fails. Such systems fall back to Legacy Mode when a Secure Boot check fails. **Disable CSM to prevent legacy fallback mode from bypassing Secure Boot protections.** Warnings and tooltips calling for CSM to stay enabled in UEFI configuration should be ignored unless a compatibility issue arises.

# 3 Use Cases For Secure Boot

## *3.1 Anti-Malware*

Secure Boot shares similarities with allow listing technologies. Rather than looking for malware via a long deny list of known-bad signatures, Secure Boot works from a short allow list of trusted certificates and hashes. Any binary that fails validation is prevented from running at boot-time.

Consider the case of a bootloader that ignores Secure Boot's software component and performs no signature checks. Such a bootloader could load any operating system, a compromised kernel, compromised modules, and other forms of malware. A bootloader debug policy with such characteristics accidentally leaked from Microsoft in 2016 (Mendelsohn 2016). The debug bootloader featured a signature trusted by the Microsoft Windows Production CA certificate stored in the DB of most machines.

Revoking the certificate by moving it to the DBX would invalidate a large number of otherwise trustworthy boot executables. System vendors chose to leverage the DBX by adding a SHA-256 hash of the debug bootloader. Because most machines have a Microsoft or system vendor KEK, a KEK-signed DBX append command via an update package was sufficient to deny list the debug bootloader.

UEFI implementations normally rely on a set of boot options to determine which devices and partitions get utilized. The options are checked sequentially until an option provides the opportunity to move beyond the BDS phase. **Failure of a boot option does not stop boot when other options are available.** A machine could fail Secure Boot validation on the debug Microsoft bootloader, but then succeed on the normal, non-debug bootloader or a PXE boot.

As another malware example, consider the case of a malicious UEFI module such as LoJax. LoJax is a malicious modification of the anti-theft solutions known as Computrace and LoJack. Secure Boot will not be able to validate LoJax against any DBX, DB, or KEK meaning that use of LoJax during boot should be prevented. However, many workstation systems ship configured in Fast Boot mode which skips checks on the PEI, DXE, and BDS phases of UEFI boot. Use Thorough Mode to force early-boot Secure Boot checks. Most servers ship with Thorough Mode enabled by default. **Always check UEFI configuration upon receipt of a new system**.

Figure 3 displays how the anti-malware properties of Secure Boot would affect LoJax. Assuming the system boots in Thorough Mode, LoJax would be denied execution at boot time while all other UEFI services operate normally. Modules and drivers in DXE can execute in parallel. Systems that pause and display a Secure Boot validation warning or error may need to be configured to

continue boot on errors/warnings or use a shorter message timeout.

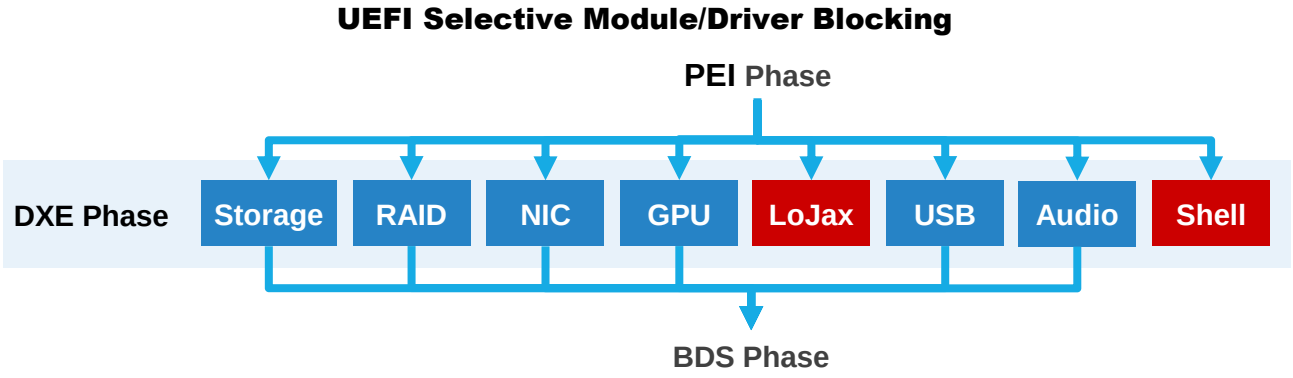



Figure 3 - Secure Boot in Thorough Boot mode denying execution to Lojax malware and a Shell app. Boot continues, although a warning or prompt about Secure Boot policy-violating content may be shown to the user.

### *3.2 Insider Threat Mitigation*

Organizations may block access to USB ports, restrict network use, and monitor user activity to combat insider threats. Secure Boot can help by closing a threat vector many organizations may not plan for – malicious physical access. Few restrictions and monitoring capabilities can cope with an insider that has physical access to a machine. The insider can boot to removable media or alter system hardware components.

Organizations can leverage Secure Boot to mitigate insider threat by removing the Microsoft UEFI Marketplace CA DB certificate and adding individual hardware components on a machine, such as the storage controller and network interfaces, to the DB allow list as SHA-256 hashes. Such an implementation allows Secure Boot, at boot time, to trust only the hardware that should be present on a machine rather than external devices. Insiders are unable to boot to external media or to unexpected network interfaces.

Additionally, removal of the Microsoft UEFI Marketplace CA DB certificate distrusts all versions of Linux. Shim, the Linux pre-bootloader, is signed by Microsoft. Organizations can sign or hash their own Shim to tailor boot to a specific build of Linux. Tailoring requires the organization to produce its own DB key and certificate. Insiders wouldn't be able to boot to Linux live images on removable or network media.

**Note:** Modification of the DB or DBX does not require modification of the KEK or PK. Partial customization is supported on most systems.

Finally, organizations can remove the Microsoft Windows Production CA DB certificate to distrust all versions of Windows and Microsoft bootloaders. Individual trusted bootloaders and kernel builds of Windows can be hashed and placed in the DB. Booting to older or unapproved versions of Windows would be impossible.

**Customizing Secure Boot to counter insider threat requires protection of the UEFI administrative credentials.** If the malicious actor can access the UEFI configuration, then the customizations can be reverted or disabled. Protect the UEFI administrative credentials and consider placing a unique credential on each endpoint.

### *3.3 Data-at-Rest*

Secure Boot can interact with Microsoft BitLocker and Linux Unified Key Subsystem (LUKS) Full Disk Encryption (FDE) solutions. Secure Boot configuration data is recorded to the TPM at boot time. BitLocker and LUKS (via extension) can use the TPM when wrapping keys for storage. Secure Boot data stores must be in the trusted state to unlock the storage volume decryption key. Tampering will change UEFI and/or Secure Boot values which would lead to failure to decrypt when unlocking the storage key.

Updates to Secure Boot or UEFI firmware require adjustment of BitLocker and LUKS TPM values. Many Windows UEFI update mechanisms automatically suspend BitLocker or prompt the user before applying the update. LUKS may have a similar mechanism depending on Linux distribution and selected options. BitLocker and LUKS protection can be enabled again on the next boot. Failure to disable BitLocker or LUKS prior to a firmware or Secure Boot update may require use of the system recovery key at the next boot or can cause permanent data loss if the recovery key cannot be found.

## 4 Customization

Modifying Secure Boot may render a system **unbootable**. The system is not “bricked” or permanently damaged. If a system enters the unbootable state try – in order – rebooting, temporarily disabling Secure Boot, reverting to the default Secure Boot configuration, or performing a firmware reset.

### *4.1 Dependencies*

Dell PowerEdge R640 with iDRAC9, Dell OptiPlex 9020, and Dell Precision 7710 were used while testing commands in the customization section. Instructions relevant to Windows were tested on Windows 10 version 1809. Instructions relevant to Linux were tested on Red Hat Enterprise Linux (RHEL) 7.6.

The following dependencies are required for **all devices** intended to receive Secure Boot customization:

- A device with support for UEFI boot and Secure Boot customization. Not all devices allow Secure Boot customization (e.g. Microsoft Surface devices).
- An operating system that supports UEFI boot. The OS does not need to support Secure Boot. Most products that advertise Secure Boot support include Microsoft signatures for boot binaries. Secure Boot customization does not require Microsoft signatures. Operating systems and hypervisors that are compatible with UEFI boot include:
    - Microsoft Windows 10, 8.1, 8, or 7
    - Red Hat Enterprise Linux (RHEL) 8, 7, or 6
    - Hypervisors that supports UEFI boot for their kernels such as VMware ESXI 7.0, 6.7, or 6.5 or Microsoft Hyper-V 6.0 or 5.0
        - UEFI emulation for VMs is not required. If supported, then VMs may support Secure Boot customization if and only if the hypervisor provides the customization option.

The following dependencies are required on a **development or testing machine**:

- (Linux and/or Windows) Openssl 0.9.8

- (Windows only) PowerShell 3.0 or newer
- (Linux only) SBSignTools 0.9 or newer from distribution repository or https://git.kernel.org/scm/linux/kernel/git/jejb/sbsigntools.git
- (Linux only) PESign 0.9 or newer from distribution repository or https://github.com/rhboot/pesign
- (Linux only) EfiTools 1.8 or newer from distribution repository or https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools.git
- (Linux only) Shim bootloader 1.0.4 or newer from https://github.com/rhboot/shim

Keys, certificates, hashes, and other data can be generated on one machine to be shared on other devices. User endpoints should not generate Secure Boot content. User endpoints should also not store any private keys relating to Secure Boot values.

The Shim bootloader included with Linux distributions normally features an OS vendor MOK provided at compile time. Deletions and additions to the MOK database may be ignored by Shim instances included with distributions depending on compilation options. Compile a custom Shim from source to disable the inclusion of an OS vendor certificate in the MOK. Both Shim and GRUB are capable of reading UEFI Secure Boot values so an OS vendor MOK may not be necessary during full customization. A vendor MOK from Red Hat, for example, will validate many RHEL, CentOS, and Fedora images and allow them to boot with more boot flexibility than desired in some use cases. The following sections assume MOK is not utilized.

### *4.2 Backup Factory Values*

Figure 4 displays the distribution of certificates and hashes in a Dell system at the time of publication. The Dell systems used to produce this report feature a PK certificate, Microsoft KEK certificate, two Microsoft DB certificates, and several DBX SHA-256 hashes. Newer systems add a second KEK and some hashes to the DB. Individual models vary. Key distribution from other vendors will be similar. DB and DBX may change over time via updates. Additional SHA-256 hashes in the DB and DBX are likely and have been omitted to save space. Backing up factory values requires saving values in each of the Secure Boot value stores (PK, KEK, DB, and DBX).

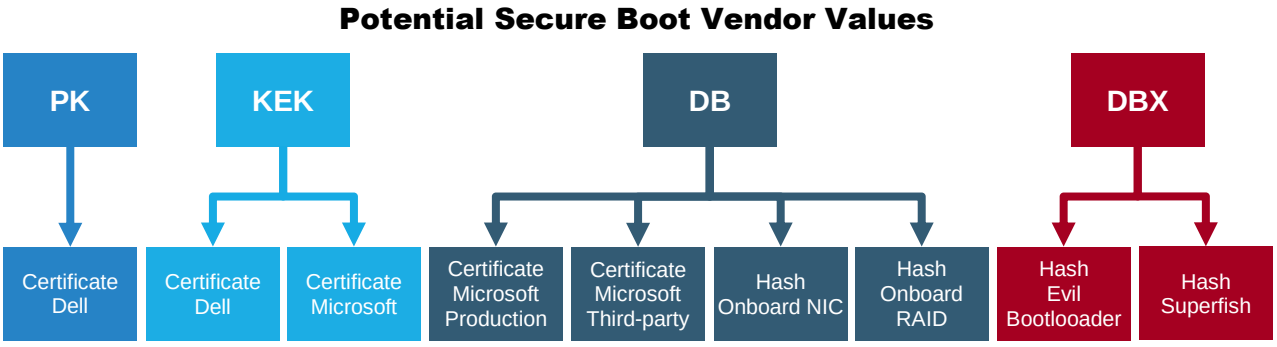



Figure 4 – Abbreviated distribution of certificates and hashes in one of the author's Dell systems.

### 4.2.1 Backup Secure Boot Values

#### *Linux Terminal*

Linux provides multiple solutions for reading UEFI Secure Boot values. Two tools are commonly available: efivar and efi-readvar (part of the efi-tools package). Both applications can output Secure Boot values, but only efi-readvar can export data to EFI Signature List (ESL) files. Each ESL can contain multiple entries. For example, the db.old.esl may contain multiple certificates and multiple SHA-256 hashes in the same ESL file. Use the following commands to backup factory values:

```
efi-readvar –v PK -o PK.old.esl
efi-readvar –v KEK –o KEK.old.esl
efi-readvar –v db –o db.old.esl
efi-readvar –v dbx –o dbx.old.esl
```

Break individual certificates and hashes out into discrete files. The following commands will result in DER-format certificates and SHA-256 hashes. Certificate file extensions of DER are equivalent to CER and may not be recognized by OS utilities (renaming extensions may be helpful). Hash file extensions of HASH are binary blobs equivalent to HSH used by many UEFI implementations. The HASH and HSH extensions are likely not recognized by OS utilities.

```
sig-list-to-certs PK.old.esl PK
sig-list-to-certs KEK.old.esl KEK
sig-list-to-certs db.old.esl db
sig-list-to-certs dbx.old.esl dbx
```

Unfortunately, hash files do not contain meta information used to derive meaning. Hashes are presented as binary data with no file name, purpose, or timestamp associated with them. Consult the system vendor to determine the purpose of a hash or search for the value via the Internet.

#### *Windows PowerShell*

Backup the existing Secure Boot values to EFI Signature Lists (ESL) via PowerShell. Each list can be later restored by Set-SecureBootUEFI if needed.

```
Get-SecureBootUEFI –Name PK –OutputFilePath PK.old.esl
Get-SecureBootUEFI –Name KEK –OutputFilePath KEK.old.esl
```

```
Get-SecureBootUEFI –Name db –OutputFilePath db.old.esl
Get-SecureBootUEFI –Name dbx –OutputFilePath dbx.old.esl
```

There is no built-in way to process ESL files to separate individual certificates and hashes. External utilities, such as sig-list-to-certs from efitools, can be used to separate the certificates and hashes should more than one exist in each file. Certificates in the ESL files are DER encoded. See section 4.2.2 for information about ESL file anatomy to enable a manual separation of certificates and hashes.

### UEFI Configuration

Some UEFI configuration tools feature a Secure Boot key management menu. Image 1 displays an example implementation. The option to select PK, KEK, DB, or DBX is usually available next to a "save to file" or "export" option. Save each value store to an external USB drive or to a memorable place within the system's storage drive if offered. Some utilities can only save backup files to the EFI directory on storage drives. Backups may have the .bin extension or no extension at all. However, the format will be an EFI Signature List (ESL) detailed in the section 4.2.2.

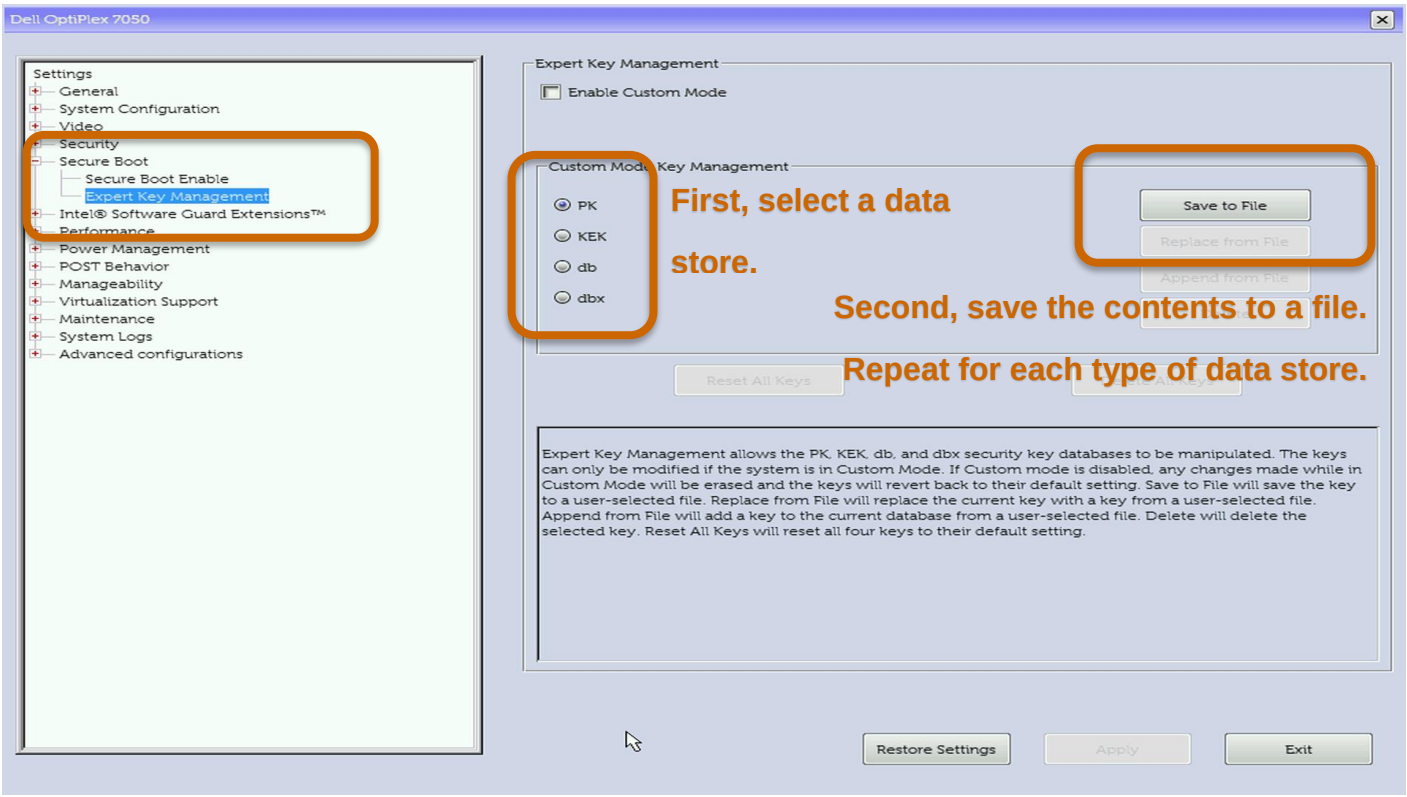



Image 1 - Dell OptiPlex 7050 workstation UEFI configuration screenshot showing default Secure Boot policy export.

### Keytool

Keytool is an EFI utility application that can be booted like a bootloader or kernel. Use the "boot to file" or "one shot boot menu" or "add boot option" capabilities of most UEFI implementations to add keytool.efi as a bootable target. Bcdedit can be used to add keytool.efi from within Windows, and efibootmgr can be used from the Linux terminal (Keytool must be in the EFI boot directory). Once Keytool has loaded, use the "save keys" option to automatically write ESL files for each

data store. The files will be PK.esl, KEK.esl, db.esl, dbx.esl, and MokList.esl. The files will be stored together in the same path as Keytool.

```
efibootmgr –c –L "KeyTool" –l "\EFI\redhat\keytool.efi"

bcdedit /copy {bootmgr} /d "KeyTool"
bcdedit /set {<GUID from previous command>} path \EFI\utils\keytool.efi
```

#### *Dell RACADM*

Vendor-specific, remote scripting solutions can be leveraged to interact with Secure Boot. A wide variety of platforms and solutions exist. Dell iDRAC 9 and RACADM have been chosen as an example. Equivalents likely exist for servers from other vendors.

To back up the existing Secure Boot values via RACADM, first establish a secure remote connection. Use the following command to take inventory of all configured Secure Boot values.

```
racadm bioscert view –all
```

Each certificate will have a corresponding thumbprint value. Each hash will have a corresponding hash value. Cycle the -t flag value (0 for PK, 1 for KEK, 2 for DB, and 3 for DBX) to access each Secure Boot data store. Cycle the –k value (0 for certificate thumbprint, 1 for hex hash) to switch selection mode. Enter a specific thumbprint or hash after the –v to select the individual record. RACADM does not produce ESLs – only individual records. DER and CER extensions are interchangeable. HSH files are binary blobs.

```
racadm bioscert export –t 0 –k 0 –v <thumbprint> -f PK.der
racadm bioscert export –t 1 –k 0 –v <thumbprint> -f KEK_1.der
racadm bioscert export –t 2 –k 0 –v <thumbprint> -f DSK_1.der
racadm bioscert export –t 2 –k 1 –v <hex_hash> -f DB_1.hsh
```

### 4.2.2 EFI Signature List (ESL) Format

ESL files contain binary data corresponding to the following format:

```
EFI_SIGNATURE_LIST {
 EFI_GUID SignatureType {
  UINT32 Data1
  UINT16 Data2
  UINT16 Data3
  UINT8 Data4[8] }
 UINT32 SignatureListSize
 UINT32 SignatureHeaderSize //usually 00000000
 UINT32 SignatureSize
 UINT8 SignatureHeader[SignatureHeaderSize]//usually omitted
 EFI_SIGNATURE_DATA Signature[SignatureSize] {
  UUID OriginatorUUID
  UINT8 Payload[SignatureSize - sizeof(UUID)] } }
```

Each ESL file contains one or more signature list structures. An individual signature list structure

can only contain objects of the certificate type or the hash type. Both certificates and hashes cannot coexist in the same list structure. However, they may both occupy the same ESL file if both a certificate signature list structure and a hash signature list structure are defined in sequence.

Figure 5 provides an example ESL file in hexadecimal representation (ESL files are binary files; not text). A single hash is present in the example file. The hash was taken from the HelloWorld.efi binary in efi-tools.

**Sample EFI Signature List (ESL) File**

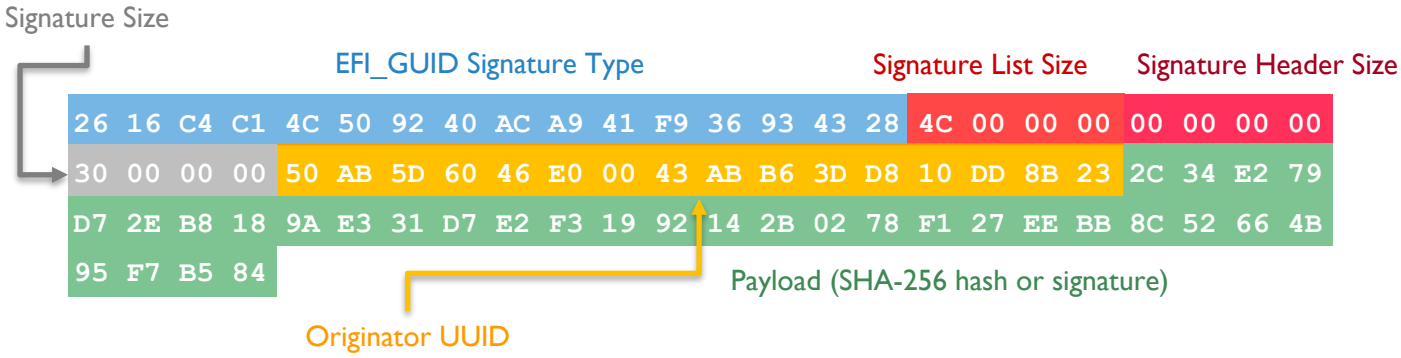



Figure 5 - An ESL file containing a single SHA-256 entry is displayed in hexadecimal format.

Table 1 lists EFI_GUID values for common ESL signature list data objects. Binary files output by efi-readvar and Get-SecureBootUEFI typically present values in Little Endian format. Source code and documentation usually display values in the Big Endian format. The UINT32 and UINT16 values will have a different byte order depending on where and how data is viewed.

| EFI_GUID Name | Value |
|---|---|
| EFI_CERT_X509_GUID | 0xA5C059A1, 0x94E4, 0x4AA7, 0x87, 0xB5, 0xAB, 0x15, 0x5C, 0x2B, 0xF0, 0x72 |
| EFI_CERT_SHA256_GUID | 0xC1C41626, 0x504c, 0x4092, 0xAC, 0xA9, 0x41, 0xF9, 0x36, 0x93, 0x43, 0x28 |

Table 1 – Common EFI_GUID values for signature list objects

Note that GUIDs and UUIDs are similar. However, EFI GUID structures observe an 8-4-4-16 format in source code. UUID structures, in contrast, observe an 8-4-4-4-12 format.

### *4.3 Initial Provisioning of Certificates and Hashes*

Initial provisioning of a system requires the creation of three new signing keys. The first will be a new PK, the second a new KEK, and the third will be placed in the DB. No DBX entry will be used. This section also requires the creation of a new hash to be placed in the DB. Assume that the DB signing key will be used to sign bootloaders and kernels while the hash represents a RAID controller. In a later section, the KEK will be used to authorize a DB change.

### 4.3.1 Create Keys and Certificates

#### *OpenSSL (Linux and Windows)*

The CRT file extension is used to denote PEM certificates, and the CER file extension is used to denote DER-encoded certificates. The PEM and DER extensions are not used because many UEFI configuration interfaces and OS implementations do not recognize PEM and DER as valid certificate file extensions.

The following instructions create three keys with self-signed certificates in PEM format. Keys intended for the DB or DBX are labeled as Database Signing Key (DSK):

```
openssl req –new -x509 –newkey rsa:2048 –subj "/CN=Custom PK/" –keyout
 PK.key –out PK.crt –days 3650 –nodes –sha256
openssl req –new -x509 –newkey rsa:2048 –subj "/CN=Custom KEK/" –keyout
 KEK.key –out KEK.crt –days 3650 –nodes –sha256
openssl req –new -x509 –newkey rsa:2048 –subj "/CN=Custom DB Signing Key
 1/" –keyout dsk1.key –out dsk1.crt –days 3650 –nodes –sha256
```

The following instructions create Certificate Signing Requests (CSR) for the KEK and DSK. UEFI lacks the ability to process certificate chains or check revocation lists so the utility of using CSRs is limited. A CSR can also be generated for the PK, but is omitted in this example. **Generating CSRs is optional.**

```
openssl req –out KEK.csr –key KEK.key –new
openssl req –out dsk1.csr –key dsk1.key –new
```

The CSRs are signed by a Certificate Authority (CA). The CA signing commands are normally executed by the CA owner and are provided in case the local organization has its own CA. The length of a certificate's validity may vary according to policies. Remember to flag, via CA configuration, the signed KEK and DSK certificates as able to perform signing actions.

```
openssl x509 –CA ca.crt –Cakey ca.key –Caserial ca.seq –in KEK.csr –req –
 days 3650 –out KEK.crt
openssl x509 –CA ca.crt –Cakey ca.key –Caserial ca.seq –in dsk1.csr –req –
 days 3650 –out dsk1.crt
```

**The following instructions convert PEM certificates into DER format.** Most UEFI implementations require DER format certificates when loading through the UEFI configuration interface (may also be referred to as F2 BIOS configuration).

```
openssl x509 –outform der –in PK.crt –out PK.cer
openssl x509 –outform der –in KEK.crt –out KEK.cer
openssl x509 –outform der –in dsk1.crt –out dsk1.cer
```

#### *Windows PowerShell*

Windows machines have alternative options to OpenSSL. Built-in utilities, provided by Microsoft, can be leveraged instead of open source solutions. However, most UEFI implementations prefer cross-platform implementations that may not accept keys, certificates, and signatures created by Microsoft utilities. Also, not all OpenSSL features are duplicated by Microsoft utilities.

To create new keys and certificates:

```
makecert –n "CN=Custom PK" –a sha256 –r –sv PK.pvk PK.cer
makecert –n "CN=Custom KEK" –a sha256 –r –sv KEK.pvk KEK.cer
makecert –n "CN=Custom DSK1" –a sha256 –r –sv DSK1.pvk DSK1.cer
```

To convert the keys from PVK to PFX format for use with Microsoft's signing tool:

```
pvk2pfx –pvk DSK1.pvk –spc PK.cer –pfx PK.pfx –f
pvk2pfx –pvk DSK1.pvk –spc KEK.cer –pfx KEK.pfx –f
pvk2pfx –pvk DSK1.pvk –spc DSK1.cer –pfx DSK1.pfx –f
```

### 4.3.2 Sign Binaries

#### *Linux Terminal*

A tool named pesign can provide information about signatures contained in a binary. Use the following command to list signatures. The file shimx64.efi is used as an example:

```
pesign -S -i=shimx64.efi
```

Pesign can also be used to remove signatures. **Most UEFI implementations only read one/the first signature in a binary. Remove or overwrite existing signatures before signing.** Use the following command to remove all signatures or add the -u option to specify a signature:

```
pesign -r -i=shimx64.efi -o=shimx64.efi
```

A tool named sbsign or sbsigntool can be downloaded for use on Linux. SBSign can sign a variety of EFI files – most importantly bootloaders and kernels – for use with customized Secure Boot. SBSign can be used to sign content for Linux, Windows, hypervisors, and more as long as binaries follow EFI specifications.

The following example command signs the shimx64.efi bootloader. The signed file will be output as shimx64.efi.signed which may need to be renamed because some UEFI implementations ignore bootable files that do not end in .efi. Sign-in-place does not function at the time of publication. Note that Shim is originally signed with a Microsoft UEFI Marketplace key – a signature that should be removed with pesign prior to signing with sbsign *if and only if* the Microsoft UEFI certificates have been removed from Secure Boot. Make a backup copy of binaries that have been signed by an external source in case reverting to a factory configuration is necessary.

```
sbsign --key dsk1.key --cert dsk1.crt shimx64.efi
```

Remember to sign the pre-bootloader (Shim), bootloader (GRUB), and kernel at a minimum. Files are named differently based on distribution and version.

#### *Windows PowerShell*

Some versions of signtool do not automatically overwrite signatures. To remove an existing signature from an EFI binary (such as Shim):

```
signtool remove /s shimx64.efi
```

To sign an EFI binary (such as Shim) using the PFX key:

```
signtool sign /f DSK1.pfx /fd sha256 shimx64.efi
```

Remember that the Windows bootloader and kernel are already signed by Microsoft. A copy of Shim supplied from a leading Linux distribution, such as Red Hat Enterprise Linux, also carries a

Microsoft signature. Do not delete or remember to append Microsoft's DB keys back into the Secure Boot DB to enable use of the Microsoft signing chain. Also append the Microsoft KEK to the Secure Boot KEK list to enable automatic additions to Secure Boot's DB via Windows Update. Remember to include DBX entries intended to revoke select Microsoft signatures too.

### 4.3.3 Calculate and Capture Hashes

Hashes used by Secure Boot must be in the SHA-256 format. There are multiple ways to represent hashes: binary BIN or HSH files, hexadecimal TXT files, and binary ESL files. UEFI configuration utilities typically use binary files with the HSH extension. Keytool and command line utilities use ESL.

HelloWorld.efi is used for the following examples. DB allow list hashes should normally be reserved for content that cannot be signed or cannot be altered from the vendor-provided state (e.g. storage array controller firmware or a hypervisor binary that already has a vendor signature). DBX deny list hashes should normally be reserved to remove trust from signed binaries without revoking the corresponding certificate/key (e.g. previously signed bootloader that is vulnerable to recent exploits). Applying a signature and creating a DB hash for the same binary is redundant and unnecessary.

Some systems are capable of generating hashes of their storage controllers as well as network interfaces and other components. Some vendors provide Secure Boot hashes of expansion devices via their websites or upon request. End users are usually not permitted to sign their own firmware images for expansion devices thus necessitating hash capture and loading to the DB.

#### *Linux Terminal*

To create a text hexadecimal, a binary hash, and an ESL file:

```
hash-to-efi-sig-list helloworld.efi helloworld.esl | cut –d “ “ –f 3 >
  shimx64.txt
tail –c 32 helloworld.esl > helloworld.hsh
```

The above commands create individual hash files. TXT indicates a hexadecimal hash file while HSH represents a binary hash file. The above commands also produce an ESL file with a single hash. Multiple hashes can be compiled into a single ESL file, although this example only incorporates one hash. Processing multiple EFI files at once will necessitate changes to the cut and tail commands. ESL files can be signed to become AUTH files. See section 4.3.1.

#### *Windows PowerShell*

To create a text hexadecimal hash:

```
$hashString = Get-AppLockerFileInformation helloworld.efi | select –
  ExpandProperty hash | select –ExpandProperty HashDataString
$hashString.Trim(“0x”) > helloworld.txt
```

To create a binary hash:

```
$hashString = get-filehash –algorithm SHA256 helloworld.efi | select –
  ExpandProperty hash
$hashBytes = [byte[]]::new($hashString.length / 2)
For($i=0; $i –lt $hashString.length; $i+=2) { $hashBytes[$i/2] =
  [convert]::ToByte($hashString.Substring($i, 2), 16) }
```

```
$hashBytes | set-content helloworld.hsh –encoding byte
```

The above commands create individual hash files – TXT format for hexadecimal characters and HSH for binary. Some loading methods may require the use of individual hashes. Hashes can also be consolidated into a single ESL file which can be signed to become an AUTH file. Creating an ESL file via PowerShell is a manual process due to the lack of an available Windows utility. An additional 44 byte header must be added to the HSH to create an ESL. See section 4.2.2 for details.

### *UEFI Configuration*

Some UEFI configuration interfaces allow the capture of system hardware hashes. Most servers – and systems that are placed in thorough boot (non-fast boot) mode – audit the hashes or signatures of system hardware resources in addition to software such as Shim, GRUB, and the Windows bootloader. Hardware resources typically audited at boot time include network interfaces, storage controllers, video cards, and storage devices. Hashes representing system hardware may be preloaded into the DB by the system vendor, provided via UEFI configuration, listed in a system manifest, listed online, or provided upon customer request.

Some vendors consider boot hashes proprietary information. Be sure to indicate to the vendor that SHA-256 hashes of component firmware for use with UEFI Secure Boot customization are desired. Hashes of UEFI firmware (e.g. SEC and PEI phases) are not necessary. Image 2 displays the UEFI Configuration hash capture mechanism of a Dell PowerEdge R740. Each hardware component can have a SHA-256 hash written to the boot partition or an external storage device for importation into a customized Secure Boot policy (configuration usually cannot traverse file systems beyond the boot partition). Then, the hashes should be loaded into the DB or DBX.

**Note:** Only Dell servers from the 14th generation (and some models from the 13th generation) provide UEFI configuration GUI and RACADM CLI mechanisms for capturing hashes at the time of this report's publication. Image 2 displays a Dell 14th generation server configuration interface featuring hash capture. Similar options are not found in Dell workstation products nor products from other vendors as of publication time.

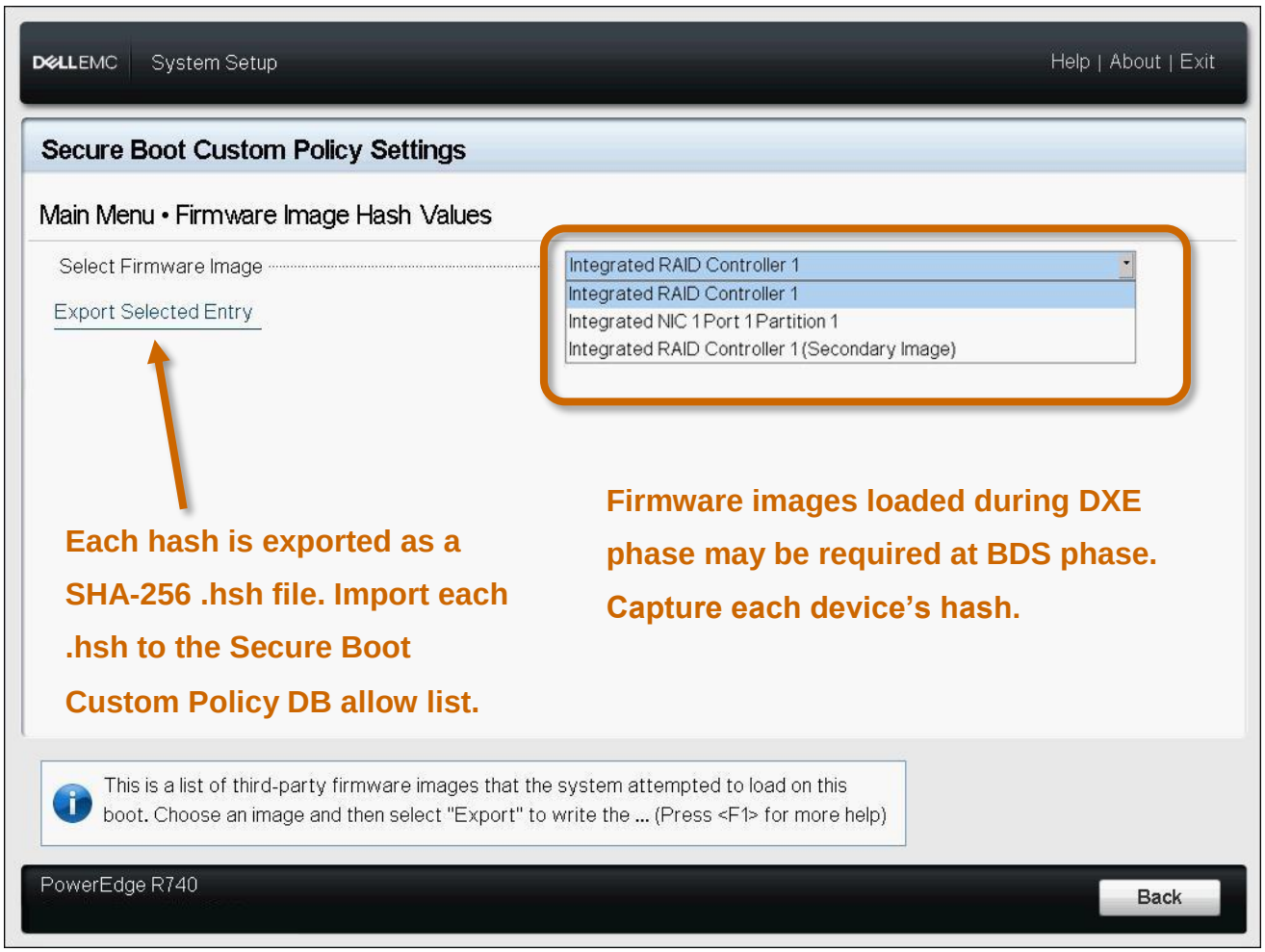



Image 2 - A Dell PowerEdge R740 server firmware component hash export utility contained within the F2 UEFI Configuration interface. The custom policy option needed to be enabled to expose hash export functionality. The hash capture feature was not available on the Optiplex 7050 – shown in Image 1 – at publication time.

### 4.3.4 Load Keys and Hashes

#### *Linux Terminal*

Certificates and hashes must be converted to ESL files before they may be loaded into Secure Boot. The following commands perform conversion. HelloWorld.efi is used as an example EFI binary to hash, and multiple EFI binaries can be listed. However, `hash-to-efi-sig-list` does not allow hashing of drivers, modules, or non-EFI binaries or input of external/arbitrary hashes (e.g. OpenSSL generated hash).

```
cert-to-efi-sig-list -g "$(uuidgen)" PK.crt PK.esl
cert-to-efi-sig-list -g "$(uuidgen)" KEK.crt KEK.esl
```

```
cert-to-efi-sig-list –g "$(uuidgen)" dsk1.crt dsk1.esl
hash-to-efi-sig-list helloworld.efi hashes.esl
```

Some tools require the use of signed ESL files – AUTH files – even when Secure Boot is not enforcing or does not have a PK loaded. **Only AUTH files can be used to carry out updates to Secure Boot's value stores while Secure Boot is enforcing checks.** Changes to the PK and KEK(s) can only be authorized by a PK. Changes to the DB and DBX can be authorized by a KEK. Signing the PK with itself is redundant to some implementations, but Keytool will not recognize ESL extension files as input. The last command simply renames the PK ESL file to an AUTH file.

```
sign-efi-sig-list –k PK.key –c PK.crt PK PK.esl PK.auth
sign-efi-sig-list –k PK.key –c PK.crt KEK KEK.esl KEK.auth
sign-efi-sig-list –k KEK.key –c KEK.crt db dsk1.esl dsk1.auth
sign-efi-sig-list –k KEK.key –c KEK.crt db hashes.esl hashes.auth
cp PK.esl PKnoauth.auth
```

Loading data into Secure Boot must be done with the DB or DBX first, then the KEK, and finally the PK. Once the PK is loaded, Secure Boot will restrict all four value stores to signed updates only and may automatically go into enforcing mode. Add the `-a` flag when loading DSK or KEK to append values to the existing entries rather than erasing existing values.

```
efi-updatevar –e –f dsk1.esl db
efi-updatevar –a –e –f hashes.esl db
efi-updatevar –e –f KEK.esl KEK
efi-updatevar –e –f PK.esl PK
```

If the above commands fail, use the AUTH files instead of ESL files. Also try the PKnoauth.auth file. Use of the append feature may also experience key store size limitations. Some systems do not support multiple KEK values, and some have tight limits on the size of the DB and DBX.

The above commands are not guaranteed to work due to the number and variety of vendor implementations. Permission errors are common due to UEFI implementation issues. Try another method of loading values if permission errors are unavoidable. Notify the OEM of UEFI Secure Boot flaws if the other methods fail too.

### *Windows PowerShell*

While Secure Boot is in setup mode, PowerShell commands may be able to update Secure Boot values. The following commands create ESL data objects.

```
$dbobject = ( Format-SecureBootUEFI –Name db –SignatureOwner 00000000-0000-
  0000-0000-000000000000 –Time 2018-01-01-T01:01:01Z –CertificateFilePath
  dsk1.crt –FormatWithCert –SignableFilePath db.esl )
$KEKobject = ( Format-SecureBootUEFI –Name KEK –SignatureOwner 00000000-
  0000-0000-0000-000000000000 –Time 2018-01-01-T01:01:01Z –
  CertificateFilePath KEK.crt –FormatWithCert –SignableFilePath KEK.esl )
$PKobject = ( Format-SecureBootUEFI –Name PK –SignatureOwner 00000000-0000-
  0000-0000-000000000000 –Time 2018-01-01-T01:01:01Z –CertificateFilePath
  PK.crt –FormatWithCert –SignableFilePath PK.esl )
```

```
$dbhashobj = ( Format-SecureBootUEFI –Name db –SignatureOwner 00000000-
 0000-0000-0000-000000000000 –Time 2018-01-01-T01:01:01Z –ContentFilePath
 helloworld.hsh –Algorithm sha256 –SignableFilePath dbhash.esl )
```

PowerShell can also be used to convert ESL files into AUTH files. **Only AUTH files can be used to update Secure Boot values while enforcing signature checks.** The PK can sign itself and KEK(s). A KEK can sign DB data. Similar to Linux, a copy of the unsigned PK file is generated in case Keytool needs to be executed. Keytool only accepts files with the AUTH extension when setting the PK. First, convert OpenSSL keys to the PFX format if necessary:

```
openssl pkcs12 –export –in PK.crt –inkey PK.key –out PK.pfx –name "PK"
openssl pkcs12 –export –in KEK.crt –inkey KEK.key –out KEK.pfx –name "KEK"
openssl pkcs12 –export –in dsk1.crt –inkey dsk1.key –out dsk1.pfx –name
 "dsk1"
```

Next, sign ESL files to create AUTH files:

```
signtool sign /fd sha256 /p7 .\ /p7co 1.2.840.113549.1.7.1 /p7ce db.auth /a
 /f .\KEK.pfx /p password db.esl
signtool sign /fd sha256 /p7 .\ /p7co 1.2.840.113549.1.7.1 /p7ce
 dbhash.auth /a /f .\KEK.pfx /p password dbhash.esl
signtool sign /fd sha256 /p7 .\ /p7co 1.2.840.113549.1.7.1 /p7ce KEK.auth
 /a /f .\PK.pfx /p password KEK.esl
signtool sign /fd sha256 /p7 .\ /p7co 1.2.840.113549.1.7.1 /p7ce PK.auth /a
 /f .\PK.pfx /p password PK.esl
cp PK.esl PKnoauth.auth
```

Loading data into Secure Boot must be done with the DB or DBX first, then the KEK, and finally the PK. Once the PK is loaded, Secure Boot will restrict all four value stores to signed updates-only and may automatically go into enforcing mode. Add the `–AppendWrite` flag when loading the DSK or KEK to append values to the existing entries rather than overwriting existing values.

```
$dbobject | Set-SecurebootUEFI
$dbhash | Set-SecurebootUEFI -AppendWRite
$KEKobject | Set-SecurebootUEFI
$PKobject | Set-SecurebootUEFI
```

Alternatively, use the following commands to utilize AUTH files for signed updates (`–AppendWrite` may also be added to the following commands):

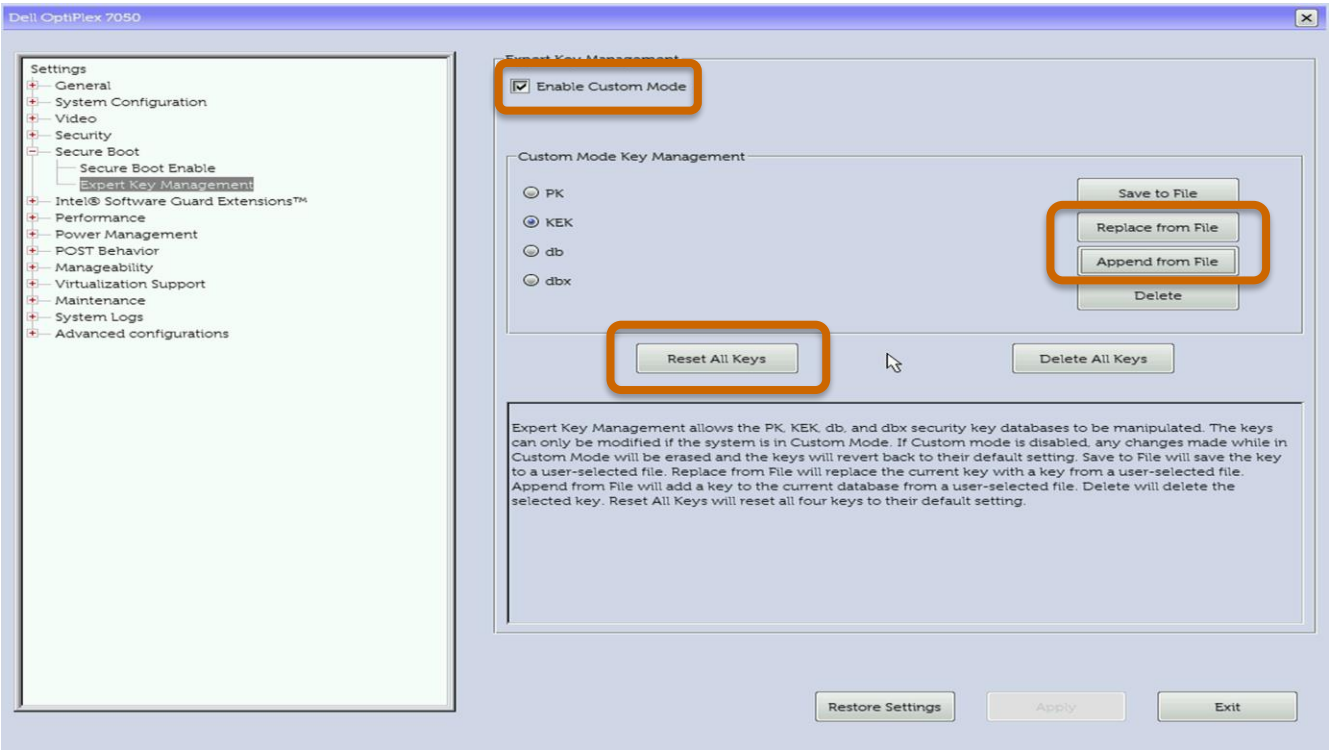
```
$dbobject | Set-SecurebootUEFI –SignedFilePath db.auth
$dbhash | Set-SecurebootUEFI –SignedFilePath dbhash.auth -AppendWrite
$KEKobject | Set-SecurebootUEFI –SignedFilePath KEK.auth
$PKobject | Set-SecurebootUEFI –SignedFilePath PK.auth
```

### *UEFI Configuration*

UEFI Configuration implementations typically have some sort of toggle or mode setting that allows Secure Boot customization. Some machines may have a state called Setup Mode that allows the replacement or appending of new values. Setup Mode transitions to User Mode once customization values are successfully loaded. Some implementations only offer User or Custom

Mode – setup is implied if User/Custom is set while Secure Boot is disabled.

Image 3 shows the customization screen on a Dell OptiPlex 7050. Checking "Enable Custom Mode" is required to replace or append values. Checking the box does not clear any data from Secure Boot – only the "Replace from File" and "Delete" options clear data. Use the "Replace from File" option to overwrite the existing PK, KEK, DB, or DBX values. Use the "Append from File" option to add additional certificates and/or hashes to the factory-default Microsoft and Dell values. Certificates in the DER format and SHA-256 hashes in the HSH format are accepted.




Images 3 – Screenshot from a Dell OptiPlex 7050. The Secure Boot Custom Policy configuration options are shown along with the selections to append, replace, or remove data.

DER and HSH files should be placed on a thumb drive or within the /boot/efi directory for easy access. Image 4 shows the file browser available through UEFI Configuration. The file browser does not support all file systems (e.g. NTFS and EXT4 usually are not supported).

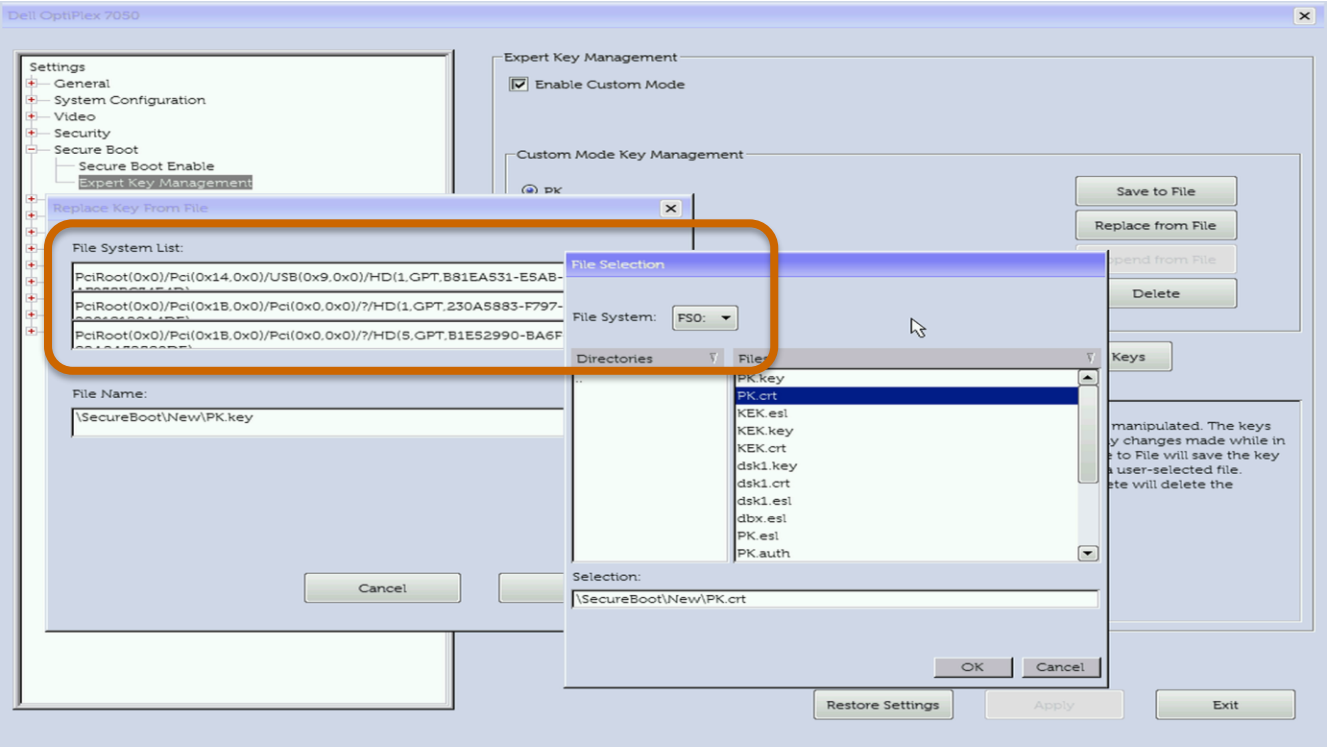



Image 4 – Screenshot from a Dell Optiplex 7050 showing the file browser available within F2 UEFI Configuration.

### Keytool

Keytool has the ability to edit Secure Boot data stores. First select Edit; then select DB, DBX, KEK, or PK (PK should be last). The edit screen will show a UUID for each value present. Some UUIDs may be identical or zeros depending on how each was loaded. Use the Add New Key option to append a new certificate or hash (ESL or AUTH format required). Use the Replace option to swap existing UUID entries with new values.

Keytool may or may not have the ability to replace or delete the existing keys and start fresh depending on UEFI implementation. Keytool is usually a reliable way to replace the PK even when UEFI configuration or command line calls fail. Keytool is easiest to use when the custom Secure Boot ESL and AUTH files are located in the same directory as the Keytool.efi file. Launching Keytool may require setting it as a boot entry via UEFI configuration, efibootmgr in Linux, bcdedit in Windows, or by launching it via UEFI Shell. See section 4.2.1 for more details.

### Dell RACADM

First establish a secure remote connection to the target system. By default, RACADM appends values to the Secure Boot data stores – overwriting is not performed. To delete all existing values, use:

```
racadm bioscert delete -all
```

To selectively delete existing values, use the –t flag to specify data store (0 for PK, 1 for KEK, 2 for DB, and 3 for DBX), optionally add the –k value for form factor (0 for certificate, 1 for hash), and optionally add the –v flag (certificate thumbprint or hex hash) to remove a specific entry.

```
racadm bioscert delete –t 0 –k 0 –v <PK_thumbprint>
```

To import new certificates and hashes, use the –t flag to specify data store and the –k flag for form factor. Use the –f flag for filename.

```
racadm bioscert import –t 0 –k 0 –f PK.der
racadm bioscert import –t 1 –k 0 –f KEK.der
racadm bioscert import –t 2 –k 0 –f dsk1.der
racadm bioscert import –t 2 –k 1 –f hash.hsh
```

### *4.4 Updates and Changes*

Updates and changes require repeating many of the steps found in the "4.3 Initial Provisioning of Certificates and Hashes" section. Updates to the DB or DBX must be signed by a KEK. Updates to a KEK must be signed by the PK. Unsigned updates or "noauth" updates are not permitted while UEFI Secure Boot is in enforcing, user, or custom mode (vendors may use different terminology).

#### 4.4.1 Update the PK

First, identify the mechanism for loading the new PK. Remote console, UEFI configuration, and Keytool typically permit PK replacement once Secure Boot has been temporarily disabled or placed into Custom/Setup mode. Run-time scripting solutions and Keytool require the new PK to be signed by the old PK when replacing the PK value while Secure Boot is active/enforcing, and physical presence is usually required to confirm the change on next boot.

Continue by ensuring the new PK is in the proper format and state for the selected loading method. Create a new RSA 2048 key pair and certificate unless a certificate has already been provided for use. Have a CA sign the certificate, if required, before use. For UEFI configuration and scripting solutions, ensure that the certificate is in DER/CER format and convert if necessary. For Keytool and console commands, create an ESL file, unsigned "noauth" file based on the ESL, self-signed AUTH file, or AUTH file signed by the currently loaded PK which will be replaced.

Finally, validate that Secure Boot is enabled and query the UEFI variable representing the new PK. Verify that the new PK is utilized.

#### 4.4.2 Update a KEK

First, identify the mechanism for loading the new KEK. Remote console, scripting, UEFI configuration, and Keytool are all possible solutions. Remote console, UEFI configuration, and Keytool usually allow unsigned KEK changes while Secure Boot is disabled. Remote console and UEFI configuration usually allow unsigned KEK changes while Secure Boot is in Custom/User mode. Run-time scripting solutions and Keytool require each KEK update ESL to be signed by the PK while Secure Boot is active/enforcing. The existing KEKs may optionally be preserved when loading the new KEK.

Continue by ensuring the new KEK is in the proper format and state. Create a new RSA 2048 key pair and certificate unless a certificate has already been provided for use. Have a CA sign the certificate, if required, before use. For UEFI configuration and scripting solutions, ensure that the certificate is in DER/CER format and convert if necessary. For Keytool and console commands, create an ESL file and, if available, a PK-signed AUTH file.

Finally, validate that Secure Boot is enabled and query the UEFI variable representing the new KEK. Consider testing the new KEK by signing DB and/or DBX changes following the instructions

in the next section.

### 4.4.3 Update the DB or DBX

First, identify the mechanism for loading the new DB or DBX value. Remote console, scripting, UEFI configuration, and Keytool are all possible solutions. DB updates can take the form of certificates, SHA-256 hashes, or ESL/AUTH files. DB updates that are signed by a KEK are permissible at all times. Unsigned updates can be accomplished via UEFI Configuration and remote console tools.

Continue by ensuring that the new DB entry is in the correct format. For a certificate, create a new RSA 2048 key pair and certificate unless a new certificate has already been provided for use. Convert to DER/CER format if in PEM/CRT format. Place the certificate in an ESL file and sign it with a KEK for the endpoint receiving the update.

For a hash, validate that the SHA-256 format is correct. Convert the hash file into an ESL file. Have the private key of a KEK sign the ESL file to convert the ESL into an AUTH file.

### 4.4.4 Update MOK or MOKX

Changes to MOK and MOKX require the use of mok-manager (mmx64.efi), mok-util (mok-util.efi), or Keytool (keytool.efi). Keys and hashes used are identical to those stored in the DB and DBX. However, MOK tools require data to be provided in only ESL or AUTH format. Section 4.3.4 provides instructions for interacting with Keytool.

## *4.5 Validation*

### *UEFI Messages*

UEFI error messages are normally printed to the primary display adapter and logged in the UEFI and OS event logs. Remote management tools, such as Dell iDRAC and HP iLO, also register UEFI events in a Baseband Management Controller (BMC) log. Some systems provide only error messages while other systems may also provide success messages. An absence of error messages, Secure Boot enabled in custom mode, and successful boot may indicate a valid launch. However, administrators should double-check that the signatures on bootable binaries match trusted certificates. Unintentionally leaving MOK or the Windows Production CA certificates in place is a common implementation oversight that looks like a success. Untrusted code may also be skipped, without an error message, hiding a potential problem.

### *Linux*

Use dmesg to determine if Secure Boot is enabled, enforcing, and what values are in use. The first command below will show only Secure Boot status. A status of "could not be determined" means that Secure Boot is not operating. The second command will return summary information about value stores, certificates, and hashes detected during boot (value stores can be read without Secure Boot being in an enforcing mode). Both commands may be run with user permissions.

```
dmesg | grep –i "secure boot"
dmesg | grep –i uefi
```

More specific information can be gathered via using efi-readvar. In particular, watch for the presence of unintended certificates in the DB or MOK. Use the -v and -s options to select a specific variable type and entry:

```
efi-readvar
efi-readvar –v db –s 0
```

Finally, mokutil can be queried to check Secure Boot enforcement status by using the following command:

```
mokutil --sb-status
```

#### *PowerShell*

PowerShell has a straightforward way to verify that Secure Boot is enabled, loaded with keys, and enforcing. The following command will return True when Secure Boot is not disabled, has a PK, and bootable binaries passed signature checks.

```
Confirm-SecureBootUEFI
```

#### *Dell RACADM*

Use the following command:

```
racadm get BIOS.SysSecurity.SecureBoot
```

A result of "enabled" or 1 indicates that Secure Boot is successfully provisioned and enforcing on the queried endpoint.

## 5. Advanced Customizations

Secure Boot is designed to complement many existing security solutions. Technologies such as security chips, boot image protection, memory protections, side channel mitigations, virtualization, malware scanners, and similar can operate alongside Secure Boot. This section focuses on a pair of boot security solutions that may seem redundant with Secure Boot. However, proper implementation can provide a defense-in-depth security solution.

### *5.1 Trusted Platform Module (TPM)*

Trusted Platform Module (TPM) may be leveraged to validate the integrity of UEFI Secure Boot. TPM Platform Configuration Register (PCR) 7 captures integrity measurement events that summarize the PK, KEK, DB, and DBX. Use the values contained within the PK, KEK, DB, and DBX to calculate what PCR 7 should be, and compare the calculated value to the value reported at run time.

Note that Shim extends MOK, MOKX, GRUB, and kernel measurements into PCR 7. Be sure to include these extensions when calculating PCR 7. Remember that MOK is similar to the DB while MOKX is similar to the DBX.

A TPM Quote Digest is a summary of PCR values. A PCR is a digest/summary of individual measurement events. A measurement event contains the Event Digest which, in the case of PCR 7, is the summary/hash of an individual UEFI variable.

Figure 6 -shows the relationship between TPM Quote, PCR, and Event/Measurement. TPM Quotes, PCRs, and measurement events are made up of a series of one-way SHA hashes. Knowing the data used to create a measurement event allows administrators/developers to wrap the data in the appropriate structures and calculate the measurement event. Knowing all the measurement events for a specific PCR allows an administrator/developer to calculate the PCR. Knowing all the PCR values allows an administrator/developer to calculate a Quote. The reverse direction is not possible due to the one-way nature of SHA hashes and TPM extensions.

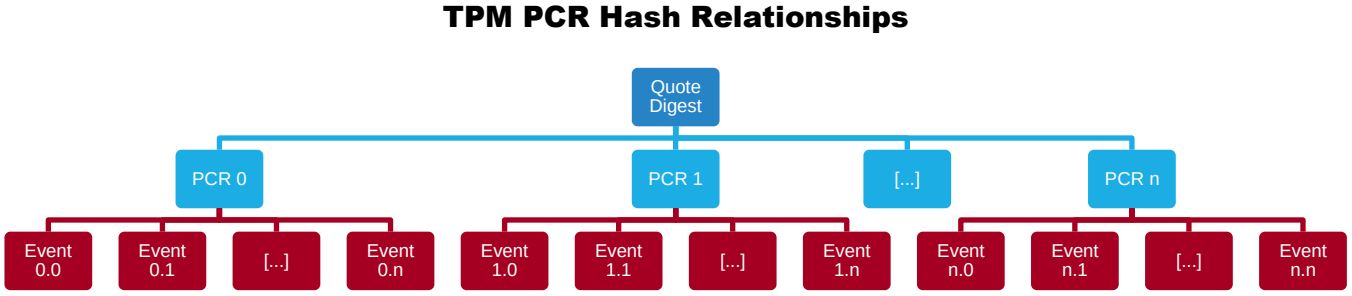



Figure 6 - The relationship between Quote, PCR, and Event/Measurement.

To calculate PCR 7 when Secure Boot values are known, consult the TCG EFI Platform Specification (TCG 2014; Section 7.1). Each TPM event log record contains the following information found in section 7.1:

```
typedef struct{

  TCG_PCRINDEX    PCRindex;
  TCG_EVENTTYPE   EventType;
  TCG_DIGEST      Digest;     //Event measurement
                              //Hash of EFI_VARIABLE_DATA
  UINT32          EventSize;
  UINT8           Event[1];   //EFI_VARIABLE_DATA

} TCG_PCR_EVENT;
```

The measurement information used to extend PCRs is captured in the TCG_PCR_EVENT TCG_DIGEST object as defined in the UEFI Specification (TCG 2014; Section 7.8). The Digest will be a SHA-1 hash in the case of TPM 1.x. In the case of TPM 2.x TCG_PCR_EVENT records for SHA-1, SHA-256, SHA-384, SHA-512, and/or other hash algorithms will be recorded since TPM 2.x supports multiple collections of PCRs at different hash strengths (TPM 2.x is “Crypto Agile” with a wide variety of implementations possible).

**The Digest values are not hashes of raw data**, defined as individual certificates and hashes, present in the DB, DBX, KEK, and PK. Digest values are hashes of the raw data wrapped in EFI metadata. In other words: Secure Boot data records, such as a DB hash or a KEK certificate, are placed in an EFI_SIGNATURE_DATA structure that is a component of the EFI_VARIABLE_DATA structure. EFI_VARIABLE_DATA is the structure that is hashed to form a TCG_DIGEST measurement which is extended to a PCR.

Each Digest value is the hash of an EFI_VARIABLE_DATA structure. EFI_VARIABLE_DATA is defined by UEFI Forum’s UEFI Specification (UEFI Forum 2017; Section 31.4). For each Secure Boot entry in the PK, KEK, DB, and DBX, hash the following structure to determine the measurement data used to extend a PCR:

```
typdef struct{

  EFI_GUID        VariableName;       //see table below
  UINT8           UnicodeNameLength;  //db, PK = 2; dbx, KEK = 3
  UINT8           VariableDataLength; //SignatureOwner +
SignatureData[]
```

```
  CHAR16          Unicodename[];      //db, dbx, KEK, PK in
unicode chars
  UINT8           VariableData[];     //EFI_SIGNATURE_DATA

} EFI_VARIABLE_DATA
```

EFI_GUID values are also defined by the UEFI Forum standards body (UEFI Forum 2017; Section 7.3). EFI_GUID values used to describe TPM events are similar to the ones found in ESL files. Table 2 shows the GUIDs that are likely to be observed.

| EFI_GUID | Value |
|---|---|
| DB and DBX records identified as EFI_IMAGE_SECURITY_DATABASE_GUID | 0x719B2CB, 0x93CA, 0x11D2, 0Xaa, 0x0D, 0x00, 0xE0, 0x98, 0x03, 0x2B, 0x8C |
| PK and KEK records identified as EFI_GLOBAL_VARIABLE | 0x8BE4DF61, 0x93CA, 0x11D2, 0xAA, 0x0D, 0x00, 0xE0, 0x98, 0x03, 0x2B, 0x8C |

Table 2 – EFI GUIDs observed with TPM events.

The UINT8 VariableData array contains the structure EFI_SIGNATURE_DATA. The entire certificate or hash binary blob contributing to a given PCR event/measurement is stored in the SignatureData array.

```
typdef struct{

  EFI_GUID        SignatureOwner;
  UINT8           SignatureData[];  //certificate or hash raw data

} EFI_SIGNATURE_DATA
```

### *5.2 Trusted Bootloader*

Trusted bootloaders use both UEFI Secure Boot and TPM. Secure Boot performs an active boot-time signature enforcement role while TPM records the state of the machine during UEFI initialization – that is to say TPM provides a check on Secure Boot's state. Examples of trusted bootloaders include Trusted Shim (TPM-extended Shim), Trusted GRUB, Trusted Boot (TBoot), TPM-rEFInd, newer Windows bootloaders, and similar boot-time security solutions. Some trusted bootloaders can be provided a "check file" or "configuration file" that includes TPM PCR hashes. The bootloader and supporting check/configuration file may also be signed by a key recognized by Secure Boot.

The TPM PCR values queried at boot time may differ from those reported from within the operating system. Bootloaders typically do not extend PCRs 0-3. Shim is known to extend PCR 7.

Always validate the signatures present on a bootloader. Bootloaders typically have a signature from the OS vendor or Microsoft which are typically intended for use with Secure Boot in the default, system vendor-provided state. When customizing Secure Boot, always ensure that specific bootloaders work as intended. Developing the Secure Boot customization guidance in this document revealed a common mistake of accidentally leaving a DB or MOK certificate behind resulting in trusting more hardware and software objects than intended at boot time.

Some bootloaders are incorporated into Full Disk Encryption (FDE) solutions and wrap a decryption key with a specific set of TPM PCR values. Ensure that PCR 7 is one of the PCRs in

the selection mask. A PCR-wrapped secret will only be revealed when PCR 7 is in the correct state thus providing confidence in the integrity of Secure Boot values corresponding to a specific PCR 7 value.

# 6 References

## *6.1 Cited Resources*

Golden, Barry. "Windows UEFI firmware update platform." Windows Documents, Microsoft Corporation, 20 Apr. 2017, https://docs.microsoft.com/en-us/windows-hardware/drivers/bringup/window-uefi-firmware-update-platform

Mendelsohn, Tom. "Secure Boot snafu: Microsoft leaks backdoor key, firmware flung wide open." Ars Technica, Conde Nast, 11 Aug. 2016, https://arstechnica.com/information-technology/2016/08/microsoft-secure-boot-firmware-snafu-leaks-golden-key

Schlej, Nikolaj. Twitter. 27 Sep. 2018. https://twitter.com/NikolajSchlej/status/1045359752077660161

Shilov, Anton. "Intel to Remove Legacy BIOS Support from UEFI by 2020." AnandTech, Future PLC, 22 Nov. 2017, https://www.anandtech.com/show/12068/intel-to-remove-bios-support-from-uefi-by-2020

Trusted Computing Group (TCG). "TCG EFI Platform Specification For TPM Family 1.1 or 1.2." TCG Published Specifications. 27 Jan. 2014, https://trustedcomputinggroup.org/wp-content/uploads/TCG_EFI_Platform_1_22_Final_-v15.pdf

UEFI Forum. "Unified Extensible Firmware Interface Specification." UEFI Forum Published Specifications. May 2017, https://uefi.org/sites/default/files/resources/UEFI_Spec_2_7.pdf

## *6.2 Command References*

Bottomley, James. "UEFI Secure Boot." James Bottomley’s random Pages. 8 Jul. 2012. https://blog.hansenpartnership.com/uefi-secure-boot

Murphy, Finnbarr. "List EFI Configuration Table Entries." Musings of an OS plumber. 24 Oct. 2015. https://blog.fpmurphy.com/2015/10/list-efi-configuration-table-entries.html

Sakaki. "Sakaki’s EFI Install Guide/Configuring Secure Boot." Gentoo Wiki, Gentoo Linux. 29 Aug. 2017. https://wiki.gentoo.org/wiki/Sakaki’s_EFI_Install_Guide/Configuring_Secure_Boot

## *6.3 Uncited Related Resources*

Hucktech. Firmware Security. 28 Jan. 2019. https://firmwaresecurity.com

NSA analysts, researchers, and contractors who contributed to pilots of customized Secure Boot. See https://www.github.com/nsacyber/Hardware-and-Firmware-Security-Guidance/tree/master/secureboot for more resources, scripts, and solutions.

Partners, vendors, and support personnel who provided information and produce improvements.

# 7 Appendix

## *7.1 UEFI Lockdown Configuration*

| Option | Recommended Setting | Comment |
|---|---|---|
| **Admin password** | Set | UEFI administrative control options access |
| **Boot mode** | UEFI | Use UEFI boot mode instead of Legacy, CSM, or BIOS |
| **Boot sequence** | * | OS drive first. Disable devices not used for boot |
| **C states / S3 sleep** | Enable | CPU energy-saving features |
| **Chassis intrusion** | | Log case-opening events |
| **Computrace** | | Anti-theft solution on some machines |
| **CPU XD support** | Enable | Execute-disable bit feature |
| **eSATA port** | Disable | Enable if external SATA ports are used |
| **ExpressCard** | Disable | Enable if required by expansion device |
| **Extended Page Tables / EPT** | Enable | Intel-only. Equivalent to RVI |
| **External USB ports** | * | Disable unused ports |
| **Fan control** | Auto | Customizable cooling fan thresholds/levels |
| **Fastboot** | Auto | Shortens some device self-check routines |
| **Free-fall protection** | | Relevant to spinning platter hard drives |
| **HyperThread / SMT** | Enable | CPU scheduling optimizer |
| **Integrated NIC** | Enable | Enable PXE if required by organization; Disable if not used |
| **Internal modem** | Disable | Enable if required for legacy network |
| **Keyboard backlight** | | May have levels of brightness |
| **Legacy OROMs** | Disable | Disable unless required by expansion devices (video card, storage controller, etc.) |
| **Microphone** | | Defer to organizational policies |
| **Module bay** | Enable | Laptops with hot-swap bays; Controls disc media device |
| **Multi-core support** | All | Controls energy use, heat, and performance of CPU |
| **Non-admin password changes** | Disable | Do not allow non-admins to alter system config |
| **Non-admin user setup lockout** | Enable | Only allow admins into UEFI config |
| **Optimus / Dynamic graphics** | Enable/Auto | Energy-saving graphics switching |
| **OROM keyboard access** | Disable | Only enable for administrators |
| **Overclocking** | | Increase CPU performance above factory limits |
| **Parallel Port** | Disable | Enable if required for legacy device |
| **Password bypass** | | Defer to organizational policies |
| **Password configuration** | | Defer to organizational policies |
| **Rapid start** | | Accelerated boot from slow storage drives |

| Option | Recommended Setting | Comment |
|---|---|---|
| **Rapid virtualization indexing / RVI** | Enable | AMD-only. Equivalent to EPT |
| **SATA Operation** | AHCI | Enable RAID or IRST (Intel Rapid Storage Technology) if appropriate |
| **SATA password** | Not set | Stops boot drive access. Interrupts updates |
| **SATA ports** | Connected only | Disable SATA ports not in use |
| **Secure Boot custom mode** | Disable | Enable custom if using custom key chain |
| **Serial Port** | Disable | Enable if required for legacy device |
| **SMART Reporting** | Enable | Storage drive error reporting mechanism |
| **SmartCard** | | Storage drive error reporting function |
| **SpeedStep / CPU power states** | Enable | CPU energy-saving features |
| **Storage OROM access** | Disable | Only enable for administrators |
| **Strong passwords** | Enable | Applies password complexity requirements to UEFI configuration accounts |
| **System password** | Not set | Stops system boot process. Interrupts updates |
| **Tagged TLB** | Enable | |
| **TPM ACPI support** | Enable | Controls loading of measurements during boot |
| **TPM PPI deprovision override** | Enable | Allows OS to clear and re-enable TPM |
| **TPM PPI provision override** | Enable | Allows OS to activate TPM |
| **TPM security** | Enable and Activate | Send power and I/O to the TPM |
| **Trusted execution / TXT** | | Windows: used when Trusted eXecution Engine (TXE) is installed. Linux and hypervisors: install TBoot and follow directions. Provision with TXT disabled. Enabling TXT locks NVRAM |
| **TurboBoost / TurboCore** | Enable | CPU performance boost feature |
| **UEFI Network Stack** | Enable | Enable if PXE or image servers are used by organization; Disable if not used |
| **UEFI Secure Boot** | Enable | Use in conjunction with supporting OS and/or hypervisor |
| **Unobtrusive mode** | | Disables or dims system indicator lights |
| **USB Boot Support** | Disable | Allows USB devices to boot; May be needed by some developers |
| **USB power share** | Disable | Charges devices through USB power |
| **USB wake support** | | Allow USB devices to wake computer on action |
| **User password** | Set | UEFI user boot configuration options access |
| **Video adapter** | Auto | Switches between integrated and discrete graphics if present |
| **Virtualization / VT-x / VPro** | Enable | Virtualization extensions for hypervisors |
| **VT-d / Virt directed I/O** | Enable | Hypervisor performance optimization |
| **Wake on AC** | | Influences boot behavior after power loss |

| Option | Recommended Setting | Comment |
|---|---|---|
| **Wake on LAN** | | Allows monitoring of network traffic for wake commands |
| **Webcam** | | |
| **Wireless switch changes** | | Defer to organizational wireless access policy |
| **WLAN** | | Wireless network toggle |
| **WWAN** | | Cellular network toggle |
| **XMP memory profiles** | | High-performance RAM profiles |

## *7.2 Acronyms*

| Acronym | Meaning |
|---|---|
| **ACPI** | Advanced Configuration and Power Interface |
| **AD** | Microsoft corporation product Active Directory |
| **AHCI** | Advanced Host Controller Interface |
| **AMD** | Microprocessor company named Advanced Micro Devices |
| **ARM** | Microprocessor company formerly known as Advanced RISC Machine |
| **BDS** | Boot Device Select UEFI boot phase |
| **BIOS** | Basic Input/Output System |
| **BMC** | Baseband Management Controller |
| **CA** | Certificate Authority |
| **CPU** | Central Processing Unit |
| **CRTM** | Core Root of Trust for Measurement starts system integrity hashing chain |
| **CSM** | Compatibility Support Module providing some BIOS functions omitted from UEFI |
| **DB** | Secure Boot Allow list Database |
| **DBK** | Database Key used with Secure Boot databases |
| **DBX** | Secure Boot Deny list Database |
| **DoD** | US government Department of Defense |
| **DOS** | Disk Operating System |
| **DXE** | Driver Execution Environment UEFI boot phase |
| **EFI** | Extensible Firmware Interface – the foundation which UEFI is built upon. Originally created by Intel corporation as a proprietary solution. Binaries designed to run in the UEFI environment may also be called EFI binaries as opposed to UEFI binaries |
| **EPT** | Extended Page Tables Intel corporation equivalent to RVI |
| **eSATA** | External Serial Advanced Technology Attachment |
| **FIPS** | Federal Information Processing Standard |
| **GNOME** | Linux desktop user environment |
| **GPT** | GUID Partitioning Table |
| **GRUB** | Linux boot loader |
| **GUI** | Graphical User Interface |
| **HDD** | Hard Disk Drive |
| **I/O** | Input/Output |

| Acronym | Meaning |
|---|---|
| **IMA** | Integrity Measurement Architecture provides runtime TPM hashing |
| **IRST** | Intel corporation Rapid Storage Technology for attached storage disks |
| **IT** | Information Technology (department or device) |
| **KEK** | Secure Boot Key Exchange Key |
| **LAN** | Local Area Network connection |
| **LCP** | Launch Control Policy used by TBoot |
| **LDAP** | Lightweight Directory Access Protocol is Linux equivalent to Microsoft AD |
| **LUKS** | Linux Unified Key Setup used for drive encryption |
| **MBR** | Master Boot Record partition scheme |
| **MBR2GPT** | Utility to convert from MBR disks to GPT disks |
| **MOK** | Machine Owner Key used for Linux extension of Secure Boot |
| **NIC** | Network Interface Controller |
| **NVRAM** | Non-Volatile Random-Access Memory storage space on TPMs |
| **OROMs** | Option Read-Only Memory firmware configuration branching mechanism |
| **OS** | Operating System such as Microsoft Windows or Red Hat Linux |
| **PC** | Personal Computer |
| **PCR** | Platform Configuration Register used by TPM to store hashes of integrity hashes |
| **PEI** | Pre-EFI Initialization phase for UEFI boot |
| **PK** | Secure Boot Platform Key |
| **PPI** | Physical Presence Interface |
| **RAID** | Redundant Array of Independent Disks |
| **RAM** | Random-Access Memory |
| **rEFInd** | UEFI Boot Loader |
| **RHEL** | Red Hat Enterprise Linux operating system |
| **RISC** | Reduced Instruction Set Computer |
| **ROM** | Read-Only Memory |
| **RSA** | Ron Rivest, Adi Shamir, and Leonard Adleman cryptosystem algorithms |
| **RVI** | Rapid Virtualization Indexing AMD corporation equivalent to EPT |
| **S3** | Sleep state 3 shuts down power to most PC components except RAM |
| **SATA** | Serial Advanced Technology Attachment |
| **SEC** | Security phase of UEFI boot |
| **SHA** | Secure Hashing Algorithm |
| **SMT** | Symmetric Multithreading for multiple CPU cores, threads, paths |
| **TBoot** | Trusted Boot open source Intel mechanism |
| **TLB** | Translation Look-aside Buffer memory management accelerator |
| **TPM** | Trusted Platform Module security chip |
| **TXE** | Trusted Execution Environment restricted kernel memory space |
| **TXT** | Intel corporation Trusted Execution Technology |
| **UEFI** | Unified Extensible Firmware Interface that is a derivative from the proprietary EFI solution created by Intel corporation. Governed by an industry consortium called the UEFI Forum |

| Acronym | Meaning |
|---|---|
| **USB** | Universal Serial Bus connects peripheral devices |
| **VMK** | Volume Management Key for Microsoft Bitlocker |
| **VPRO** | Intel corporation branding for devices supporting multiple virtualization enhancements and TBoot |
| **VSM** | Virtual Secure Mode suite of device-hardening features in Microsoft Windows |
| **VT-d** | Virtualization Technology for Directed I/O |
| **WLAN** | Wireless Local Area Network |
| **WLAN** | Wireless Local Area Network |
| **WWAN** | Wireless Wide Area Network normally indicates presence of cellular adapter |
| **XD** | Execute Disable bit allows CPU to disable execution in memory spaces |
| **XMP** | Extreme Memory Profile used for controlling RAM timing |

## *7.3 Frequently Asked Questions (FAQ)*

**Does Secure Boot customization require replacing the PK and KEK?**

No. Secure Boot customization can be partial in implementation. Customizers may add/append additional records to the DB, DBX, or KEK without clearing or replacing existing values. Likewise, customizers may remove individual records from the DB, DBX, or KEK rather than completely clearing each value store.

**What is the difference between the Microsoft Windows Production CA and UEFI Third Party Marketplace CA DB certificates?**

The Microsoft Windows Production CA signs all things specific to the Windows operating system environment. The Windows boot manager, kernel, and drivers are commonly validated by the Production CA cert. The UEFI Third Party Marketplace CA signs content not related to Windows such as storage controller firmware, graphics card firmware, UEFI driver modules, and Linux bootloaders.

**How do I make a driver compatible with Secure Boot?**

Many Linux anti-malware solutions include drivers that do not have Secure Boot signatures. To solve the problem, do not disable Secure Boot. Instead, create an RSA 2048 public key certificate. Use the corresponding private key to sign the driver using sbsigntool, pesign, or similar. Switch to Secure Boot custom/user mode in the UEFI configuration, and then append the custom certificate into the machine's DB using UEFI configuration, Keytool, or similar. Do not make changes to the PK, KEK, or DBX. The driver should be validated by the custom certificate following the next boot. Remember to sign updates to the driver before distributing to endpoints.

**How do I revoke a threat like BlackLotus or BootHole or similar signed EFI executable?**

Revoking signed EFI executables requires updating the DBX. If system and OS vendors are unable to provide DBX updates, then the customer may need to produce their own. Follow these steps:

1. Identify specific EFI binaries that need to be revoked.
2. Calculate hashes for the EFI binaries. Note that tools aware of Portable Executable (PE/EFL) format must be used. A sha256sum or OpenSSL digest hash of an entire binary will result in the wrong hash. Only executable portions of the binary are hashed for inclusion into the DB/DBX. See section 4.3.3.

3. If your BIOS, UEFI configuration, or remote management tool accepts hashes, then submit them.
4. If hash files were not sufficient, then create ESL files. Try to load the ESL files.
5. If the ESL files are rejected, it's likely due to a lack of signature. Follow instructions in section 4.3.1 to create a key and certificate used when signing ESL files. This new certificate may need to be loaded to the DB or as the PK.
6. Use the new certificate and key to sign ESL files into AUTH files.
7. Install the new hashes and possibly the new certificate following instructions in section 4.3.4.

**How do I revoke signatures?**

First, determine which certificate is responsible for validating a revoked signature. UEFI Secure Boot has limited space available – the amount varies based on make and model of device. If a large number of signatures are to be revoked, consider migrating to a new certificate and placing the old one in the DBX. If a manageable number of signatures are to be revoked, create a list of SHA-256 hashes corresponding with each binary to be revoked. Compile the hashes into an ESL file. Use Keytool to load the ESL file into the DBX at boot time.

**Does UEFI Secure Boot understand Certificate Revocation Lists (CRL)?**

No. Most UEFI implementations lack the memory space and processing power needed to navigate the internet and parse CRL information. Revocations and certificate chains are ignored by Secure Boot. Software and system vendors usually provide DBX patches to handle revocation actions.

**My endpoint won't accept a new KEK, DB, or DBX entry. What should I do?**

First, check the firmware version of the endpoint to determine if an update is available. Individual firmware releases can contain bugs to the Secure Boot customization implementation. Next, check to see if there are known limitations to a specific make and model of endpoint. You may need to reach out to the system vendor if a firmware update does not resolve the problem and firmware storage capacity is not an issue.

**Where is MOK and MOKX?**

Machine Owner Key (MOK) and MOK Exclusion (MOKX) are extensions of UEFI Secure Boot. The bootloader Shim is responsible for setting up MOK and MOKX. Shim is usually found on Linux systems and not found on Windows systems. MOK and MOKX do not exert any enforcement action until the Bootloader Phase of UEFI Boot (i.e. boot devices, OROMs, and firmware modules are not checked against MOK and MOKX).

Shim features a signature from Microsoft and embedded MOK certificate from a Linux distribution or power user. Shim and MOK allow the open source software community to realize the advantages of Secure Boot without needing to seek Microsoft review/approval for every bootloader, kernel, and module. Microsoft signs Shim, Shim sets up MOK, MOK validates the second bootloader (commonly GRUB), MOK validates the Linux kernel, and MOK validates kernel modules. Most computing products available today do not ship with a Linux distribution KEK or DB certificate – Shim creates a software solution to a firmware limitation driven by market share.

MOK functions like the DB, and MOKX functions like the DBX. MOK and MOKX extend the function of DB and DBX, effectively. Remember that DB and DBX are available prior to the bootloader phase of UEFI boot. However, MOK and MOKX are initialized during the bootloader

phase if and only if Shim is used. MOK and MOKX can only be used part-way through the bootloader phase and in following phases.

**What devices ship with UEFI Secure Boot as an option?**

Most business and consumer devices intended to run Microsoft Windows support Secure Boot. Servers, blade arrays, laptops, desktops, tablets/2-in-1s, all-in-one PCs, small form factor PCs, mobile phones, Internet of Things (IOT) devices, and similar products are likely to have Secure Boot support. Devices supporting other operating systems may also have unutilized Secure Boot support.

**Where can I get more information, scripts, guidance, strategies, and other resources?**

Visit the NSA Cybersecurity GitHub at https://www.github.com/nsacyber/Hardware-and-Firmware-Security-Guidance for additional resources. A section specific for Secure Boot is located at https://www.github.com/nsacyber/Hardware-and-Firmware-Security-Guidance/tree/master/secureboot. Scripts, use cases, and resources for navigating customization on a variety of vendor implementations will be posted over time.