AFRL-RI-RS-TR-2020-150
VOLUME 2 of 2

SECURE BOOT DEFICIENCIES

NTELIGEN

AUGUST 2020

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2020-150 VOL 2 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
MATTHEW P. SHAVER
Work Unit Manager

/ S /
SCOTT D. PATRICK
Deputy Chief, Information Intelligence
Systems and Analysis Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* AUGUST 2020	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* NOV 2019 – JUN 2020

4. TITLE AND SUBTITLE SECURE BOOT DEFICIENCIES VOLUME 2 of 2	**5a. CONTRACT NUMBER** FA8750-20-C-0539
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 62788F
6. AUTHOR(S) David Schloss, Mike Hike, Roman Grewal, Erik Carlson	**5d. PROJECT NUMBER** E2DC
	5e. TASK NUMBER SB
	5f. WORK UNIT NUMBER DS
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Nteligen, 6716 Alexander Bell Dr Ste.120, Columbia, MD, 21146	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RIEBA, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2020-150 VOL 2

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# AFRL-2020-0021
Date Cleared: 04 AUG 2020

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This technical report captures the research effort into two deficiencies encountered while implementing Secure Boot technology. The first deficiency discovered is that OpROMs will fail to load on a system that has custom Secure Boot keys in its key variables and has removed all manufacturer keys from the key variables. The second deficiency is that Secure Boot fails to validate the digital signatures of certificates within the Secure Boot key variables in standard Public Key Infrastructure (PKI) certificate hierarchy operation. The technical details each deficiency clearly and expands the research methodology applied to each. With the insights gained from our research, we make informed findings and recommendations on how to mitigate each deficiency. Lastly, within this technical report, we present best practices on how to implement Secure Boot technology.

15. SUBJECT TERMS

Secure Boot, OpROM Deficiency Research, Certificate Hierarchy Deficiency, Secure System Secure Boot Implementation

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** MATTHEW P. SHAVER
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	42	**19b. TELEPHONE NUMBER** *(Include area code)* N/A

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 SUMMARY

In this technical report, we capture the research effort into two deficiencies encountered while implementing Secure Boot for a secure system. The first deficiency discovered is that OpROMs fail to load on a system that has custom Secure Boot keys in its key variables and has removed all manufacturer keys from the key variables. The second deficiency is that, by design, Secure Boot does not validate the digital signatures of certificates within the Secure Boot key variables in standard Public Key Infrastructure (PKI) certificate hierarchy operation. The technical details of each deficiency are detailed further within this report. Additionally, this report expands upon the research methodology applied to each deficiency. With the insights gained from our research, we make informed findings and recommendations on how to mitigate each deficiency. Lastly, within this technical report, we present best practices on how to implement Secure Boot within a secure system.

2.0 INTRODUCTION

2.1 Introduction

Universal Extensible Firmware Interface (UEFI) is the standard interface platform that replaces legacy basic input/output system (BIOS) in modern computer hardware. UEFI provides hosts computers additional capabilities over legacy BIOS, one of which is Secure Boot. Secure Boot, when enabled, provides a firmware-based boot integrity verification process that ensures the system is operating in a known state and running known software after the boot cycle. This process is defined and governed by the UEFI 2.8 Errata A specification published by the UEFI Forum. The UEFI Forum is comprised of the world's foremost researchers in academia, eminent computer scientists, and technology industry leaders from more than 250 member companies, working to develop and maintain the full suite of the UEFI and ACPI specifications. In this technical report, we aim to detail our research into the two deficiencies highlighted within Secure Boot implementation within secure systems. The paper begins by exploring the deficiencies identified. Next, details the research methodologies. Lastly is a discussion of conclusions and recommendations regarding both deficiencies.

2.1.1 Extensible Firmware Interface Files

UEFI uses a specific data type, known as Extensible Firmware Interface (EFI) files, to facilitate the execution of applications and drivers. An application is software that boots an operating system, whereas a driver provides the UEFI access to a device controller. EFI files use the Portable Executable Common Object File Format (PE/COFF) for their structure. The digital signature of the EFI file is embedded directly within the file itself. The PE header of the EFI file contains an array of data directory pointers, the 5th of which is the pointer to a list of certificates, any of which may contain a digital signature. The digital signatures within an EFI are in Microsoft Authenticode digital signature format. When an EFI file is signed, it ignores specific fields within the PE format, as indicated in the figure below. Since the Authenticode hash process ignores the certificate list storing the embedded signature, this does not affect the hash value of the EFI file it is hashing. In theory, multiple sources of authority can sign an EFI. In certain cases, a secure system vendor might want to be the only Secure Boot signing authority for a system they develop.

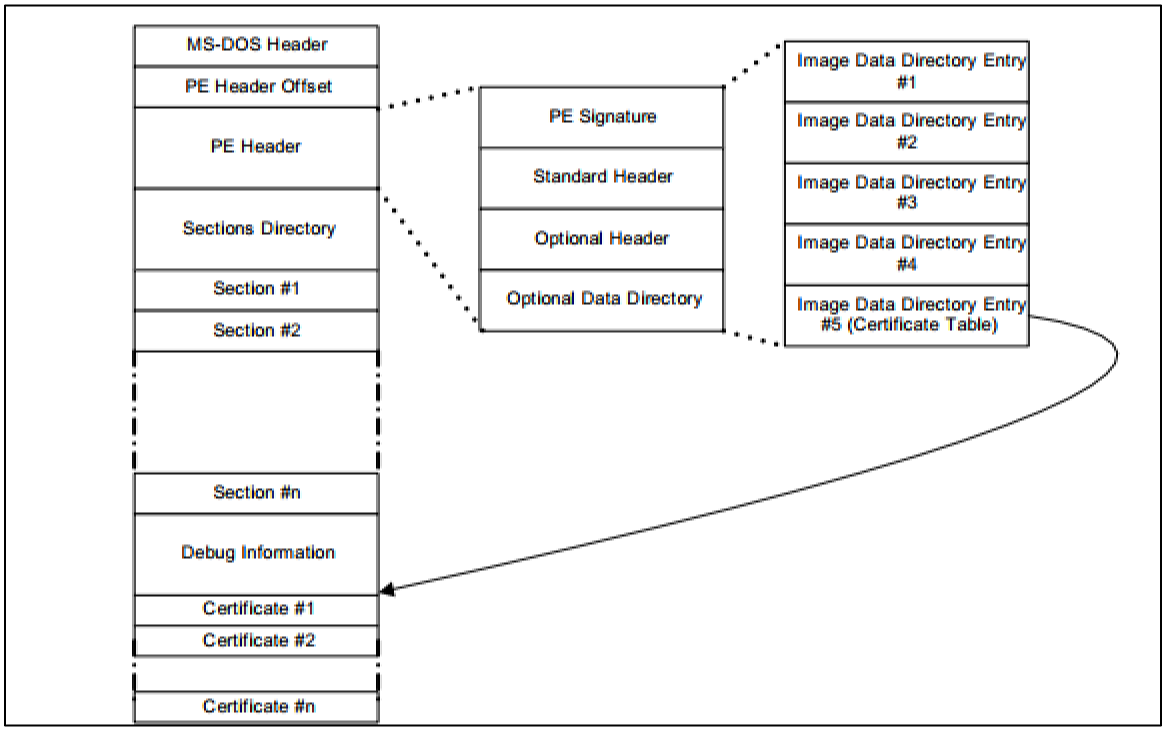


Figure 1. EFI File Embedded Certificate Layout

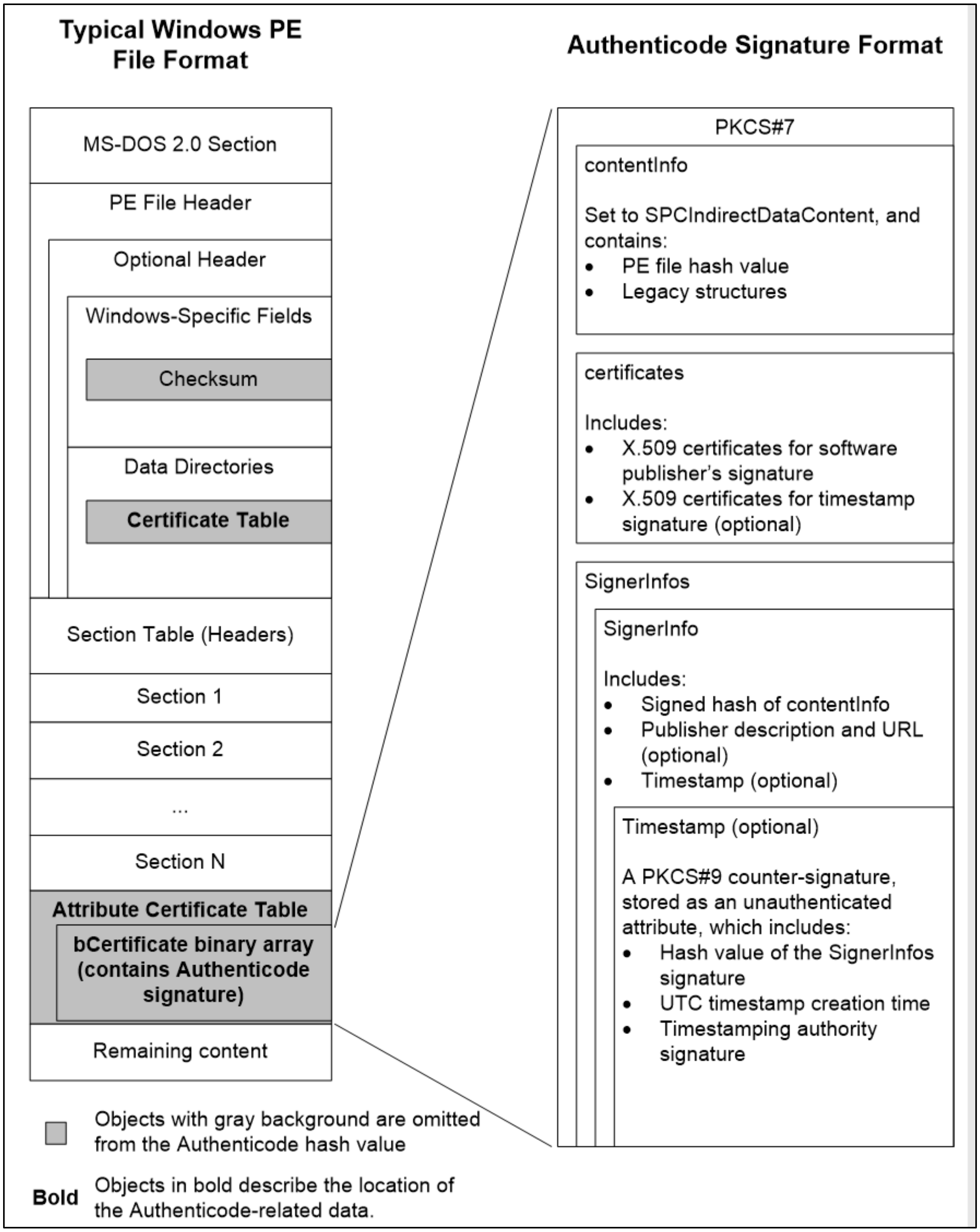

Typical Windows PE File Format
Authenticode Signature Format
MS-DOS 2.0 Section
PE File Header
Optional Header
Windows-Specific Fields
Checksum
Data Directories
Certificate Table
Section Table (Headers)
Section 1
Section 2
...
Section N
Attribute Certificate Table
bCertificate binary array (contains Authenticode signature)
Remaining content
PKCS#7
contentInfo
Set to SPCIndirectDataContent, and contains:
• PE file hash value
• Legacy structures
certificates
Includes:
• X.509 certificates for software publisher's signature
• X.509 certificates for timestamp signature (optional)
SignerInfos
SignerInfo
Includes:
• Signed hash of contentInfo
• Publisher description and URL (optional)
• Timestamp (optional)
Timestamp (optional)
A PKCS#9 counter-signature, stored as an unauthenticated attribute, which includes:
• Hash value of the SignerInfos signature
• UTC timestamp creation time
• Timestamping authority signature
▨ Objects with gray background are omitted from the Authenticode hash value
Bold Objects in bold describe the location of the Authenticode-related data.

Figure 2. PE File Format and Authenticode Signature Format

2.1.2 3rd-party Option Read-Only Memory

An important concept related to this research effort is 3rd-party Option Read-Only Memory (OpROM). OpROM firmware can reside either within the UEFI BIOS or on a 3rd-party device (i.e., an expansion card). When initialized, the system loads the OpROM into memory and registers the 3rd-party device with the UEFI BIOS. EFIs for the 3rd-party devices reside within the structure of their associated OpROMs, and these are the executable elements that perform the registration. OpROMs referenced within this paper are strictly UEFI OpROMs and not legacy BIOS OpROMs. Legacy BIOS OpROMs are firmware that works with the Personal Computer Advanced Technology (PC-AT) system architecture. PC-AT is the system architecture of the majority of computers before the release of the UEFI specification. Legacy BIOS OpROMs did not work with Secure Boot and were not used when the issues calling for this research were encountered. Also note, these are 3rd-party OpROMs, which means that the OpROM resides on an expansion card or device that was not manufactured by the platform OEM, and thus the firmware of the OpROM was developed by a 3rd-party. For purposes of this paper, consider that the first party is the OEM, 2nd-party is the equipment owner, and 3rd-party is an optional add-in equipment manufacturer.

2.1.3 Secure Boot Integrity Validation Process

Secure Boot provides system boot integrity validation by performing validation of the embedded digital signature within an EFI file. Additionally, Secure Boot can validate an EFI before execution by comparing it with a known hash value of the EFI file. If Secure Boot is enabled and either of these mechanisms fails to validate the EFI file, then it will not be allowed to execute. UEFI firmware has four authenticated variables that can store either PKI certificates or hash values used to validate the EFIs. In Secure Boot, an authenticated variable is a UEFI variable that requires additional authentication data to be validated prior to being updated. Only with the appropriate authentication data can Secure Boot variables be updated. The following provides further details on the Secure Boot authenticated variables:

- Enrolled in the PK variable is a Distinguished Encoding Rules (DER) formatted X.509 certificate (PK_{pub}), which has a corresponding RSA-2048 or greater private key (PK_{priv}) stored by the platform vendor. The PK_{priv} digitally signs UEFI payloads that update the PK variable or the KEK variable.
- The KEK is not a single X.509 certificate but a signature database variable that can store one or more DER formatted X.509 certificates. Vendors of widely distributed bootable software or firmware have hardware manufacturers deploy their certificates within the KEK. Microsoft is one such example of an operating system vendor that works with manufacturers to deploy their public KEK certificate widely so most systems can boot Windows with ease if Secure Boot is enabled. The PK_{priv} is used to digitally sign UEFI payloads that update the PK variable or the KEK variable.
- Signatures Database (db): The db stores the DER formatted X.509 certificates used to validate the signature of bootable firmware and software. Additionally, the db can store hash values of EFI files, which also can perform EFI execution validation. Like the KEK, the db is a signature database variable. The KEK_{priv} or PK_{priv} is used to digitally sign UEFI payloads that update the db variable.

- Forbidden Signatures Database (dbx): The dbx is a signature database variable that forbids the execution of EFIs. The dbx variable can store DER formatted X.509 certificates or EFI file hashes. If an EFI attempts to execute using a certificate or hash contained in the dbx, then execution of that EFI is halted. The KEK_{priv} or PK_{priv} is used to sign UEFI payloads that update the db variable digitally.

There are two mechanisms in which the Secure Boot system can validate EFIs.

1. **EFI signature validation:** KEK_{priv} or db_{priv} creates an embedded digital signature within an EFI file. Upon boot, validation of an EFI file embedded digital signature will be performed using the KEK_{pub} or db_{pub} X.509 certificates. If either validates the digital signature, the EFI is allowed to continue execution.
2. **EFI file hash validation:** A hash of the EFI is generated and checked against the hash values that are in either the KEK signature database or the db signature database. If the hash value is in either signature database variable, the EFI file is allowed to continue execution.

EFIs are also explicitly compared to the dbx as well. If an EFI is signed by a certificate or its hash value is in the dbx, the EFI is not allowed to execute. During the boot process, EFI files are executed from their respective devices with the OS EFIs launched from the hard drive boot sector and OpROMs from the device firmware. Any EFI that has a valid signature or hash is allowed to boot. Otherwise, the system boot continues without that device or OS enabled. Prohibiting invalid EFIs from booting ensures that only approved and trusted software is executed during bootstrap.

By using Secure Boot, an administrator can guarantee that a device, if allowed to boot, has booted into a known state using known and trusted EFI images validated by their signature or hash value, which are approved by a recognized authority. Secure Boot based system integrity protects the system from attacks where the bootable software or firmware in the boot chain has been covertly or unintentionally modified to perform unanticipated actions or replaced with malware or unknown bootable artifacts.

2.1.4 Problem Discussion

2.1.4.1 3rd-party OpROMs Deficiency

Secure Boot does not preclude systems from being booted by industry-standard software loads, such as Windows or Red Hat Enterprise Linux, as long as the manufacturer loaded certificates are present in the Secure Boot variables. Keeping manufacturer certificates and hashes in the signature database variables would allow an attacker to boot the system into an alternate or vulnerable state, allowing manipulation of the machine and resident data in unanticipated ways.

In secure systems, it is recommended to delete manufacturer deployed certificates and hashes from the Secure Boot variables. After removal, certificates, and hashes that are generated by the secure system vendor are added to the relevant Secure Boot variables. By doing this, the secure system vendor can ensure that a system with Secure Boot enabled only boots software or firmware approved and signed by that vendor. The replacement of the certificates and hashes prevents a malicious user from purposely altering the system or a misguided administrator from mistakenly making unauthorized or unwise changes.

Following this guidance is an excellent approach to increasing the security posture of secure devices. However, it does not account for 3rd-party OpROMs containing EFI drivers for 3rd-party system devices. In this scenario, the removal of factory certificates and hashes (to prevent the booting of images not signed by the secure system vendor) prevents the EFI drivers from these OpROMs from being executed. In many cases, this renders the associated system devices unusable by the secure system. This problem was discovered on a secure system development effort, by removing the factory certificates and hashes from the Secure Boot KEK and db variables of a test system and subsequently attempting system boot. Removal of the manufacturer certificates and hashes resulted in the system failing to load several Hardware (HW) device OpROMs. These included OpROMs for Small Computer System Interface (SCSI) based hard drives attached to a Redundant Array of Independent Disks (RAID) controller and non-motherboard Network Interface Cards. The result was that these devices were unavailable for use after system boot.

Further exasperating this issue is the lack of transparent methodologies by which to resign or generate hashes of the EFI drivers contained within these 3rd-party OpROMs. Device manufacturers package the EFI drivers into their associated OpROMs and offer no clear or consistent methodology for extraction and resigning when clearing Secure Boot variables of manufacturer certificates and EFI hashes. Simply put, the 3rd-party manufacturers rely on the presence of established software vendor's Secure Boot certificates to sign their OpROMs

2.1.4.2 Secure Boot Certificate Hierarchy Deficiency

An issue unrelated to the 3rd-party OpROMs was encountered during Secure Boot implementation on a secure system when replacing the values of the PK, KEK, and db. Based on the UEFI specification of Secure Boot, it was interpreted that the PK, KEK, and db form a PKI validation hierarchy, each validating its respective child before the certificates in those Secure Boot variables can be used. Upon replacement of the PK, while keeping the hardware manufacturers' certificates in the KEK and db signature database variables, it was found that the manufacturer certificates could still be used to validate signed EFIs. According to our interpretation of Secure Boot operation within the UEFI specification, this should have been prevented as the PK certificate did not sign the KEK and db certificates. When this issue was first encountered, it was incongruent with our interpretation of the Secure Boot specification and thus raised questions about the compliance of UEFI implementations within hardware vendors. Later in this document, we will discuss how the initial deficiency discovery ended up being based on our misinterpretation of the UEFI specification.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Secure Boot Research

The Secure Boot research project was undertaken to find more information about the breadth of issues that exist while attempting to implement Secure Boot within a secure system. Through our research methodology, we plan to understand the deficiencies as discussed and to develop a path forward to improve UEFI specification and industry standards when implementing their UEFI compliant solutions for secure systems.

Since the two deficiencies discovered were unrelated, we decided to separate each problem into its own research effort. By decoupling the problems, it allowed us to work them in parallel and to limit any blockers during research and testing. In sections 3.1.1 and 3.1.2, we expand on our research approach for each of the deficiencies in greater detail.

It is of note that while we did separate the Secure Boot deficiencies into separate research efforts, there were three related areas of interest examined to inform the outcome of our research. These areas were deficiencies of Secure Boot operation and specification and Hardware Vendors Secure Boot implementation. With the information gathered from these sources, we hope to come to a better understanding of exactly how Secure Boot operates so we can make informed recommendations on how to best implement Secure Boot within a secure system.

3.1.1 3rd-Party OpROM Deficiency Research Methodology

When considering the problem of 3rd-party OpROMs not booting under a custom Secure Boot policy, it was essential to clearly state the problem and understand as much as possible regarding how OpROMs are loaded during boot. From this stance, we were able to establish a defined test procedure replicating the problem. Using controlled server hardware that is class equivalent to secure systems, we created test procedures that emulate the removal of manufacturer loaded Secure Boot certificates replacing them with in-house generated Secure Boot certificates. Once we were able to replicate the OpROM boot issue, we sought to find a solution that allows OpROMs to boot without having the manufacturer certificates loaded in the KEK or db signature database variables. We took two different approaches to the development of a solution in the hopes of generating a proof-of-concept before the end of the research effort. In APPENDIX A, we have listed all of our testing procedures for flashing firmware onto PCI devices. Also, we have a test procedure for booting a device using a custom Secure Boot policy to test if firmware signed by us could be successfully booted. Lastly, with the body of knowledge gained, we generated a set of best practices related to Secure Boot implementation within a secure system.

3.1.2 Secure Boot Certificate Hierarchy Deficiency Research Methodology

When considering the Secure Boot certificate hierarchy deficiency, we saw the end goal of this research area slightly differently than that of the OpROM issue. Where the OpROM issue ostensibly resulted in a proof-of-concept solution, the research around this deficiency could only result in guidance to system hardware manufacturers. Our methodology on researching the certificate hierarchy issue started similar to the OpROM issue but diverged when it came to replicating the issue and best-practices creation. The approach taken was to fully understand how Secure Boot certificates validate one another by reading UEFI specification documentation and through interactions with the system hardware manufacturers. Once understood, we consulted with secure system vendors to see if they had encountered this problem and determine what their solution if any, might be. Lastly, with the body of knowledge gained, we planned to compile our findings and provide them back to the system hardware manufacturers and Air Force Research Laboratory (AFRL) as part of this final technical report.

3.1.3 Secure Boot Operation and Specification Research Methodology

In our research, it was critical to understand the raw fundamentals of how Secure Boot operates. Understanding how Secure Boot works is the foundation of our findings and recommendations for Secure Boot implementation within secure systems. A body of knowledge was built by researching the Secure Boot specification, Microsoft Secure Boot implementation guidance, and UEFI technical forums. These materials helped us gain a complete understanding of Secure Boot EFI validation operations, Secure Boot key structure, EFI file structure, OpROM structure, and Secure Boot variable authentication.

3.1.4 Commercial Vendor Research Methodology

Supplementing the knowledge gathered from the UEFI specification, we worked with commercial vendors involved in the specification and implementation of Secure Boot. Both deficiencies needed information from the vendors to establish informed findings and solutions. We worked with Dell, Hewlett Packard Enterprise (HPE), and Microsoft. Microsoft was involved because they were heavily involved in authoring and contributing to the UEFI specification. Microsoft also provides a widely distributed certificate for hardware vendors to include as a default KEK and db certificate within the Secure Boot system. This allows Microsoft signed EFI images to boot with little effort in when enabling Secure Boot on a system. Also, Microsoft maintains a service in which 3rd-party developers can submit their EFI files to them to be digitally signed. Since we were concerned with OpROM signing in this research effort, an attempt to reach out to Microsoft was made.

Dell was involved since they are a major server hardware producer for secure systems that utilize Secure Boot functionality. Both of the Secure Boot deficiencies were first encountered on a Dell PowerEdge R640 server during secure system development. The research effort had access to a Dell PowerEdge R740 server, which offers the Secure Boot capability through its UEFI architecture. Using this server, we can replicate the exact deficiencies that were encountered while developing the secure system (See APPENDIX A for Secure Boot test procedures). Also, we worked directly with Dell's technical support to get direct answers to our deficiency questions.

HPE was contacted since they are also a major server hardware producer for secure systems. HPE also constructs its servers to the UEFI specification and offers the Secure Boot capability. One pivotal intent during the research was to compare the Secure Boot capability of comparable secure system hardware to frame our best-practices and recommendations more abstractly. There should not be a limit on what platform vendor is chosen as long as they remain UEFI compliant. On this project, we also investigated and ran test procedures (See APPENDIX A for Secure Boot test procedures) against an HPE DL380 server. The HPE DL380 offered similar UEFI capabilities as the Dell PowerEdge R740, so it was a good measure of how different UEFI architectures were implemented. Additionally, HPE technical support made themselves available to us to ask detailed questions about the Secure Boot deficiencies, as well.

Table 1 lists the commercial vendors that were contacted to further our knowledge of Secure Boot and how it operates in their systems.

Table 1 Commerical Vendor Research Summary

Vendors	Method of Research
Hewlett Packard Enterprise (HPE)	Multiple discussions via conference call with HPE engineers and emails were exchanged as well. Additional research was performed utilizing and HPE DL380 server.
Dell	Multiple discussions via conference call with Dell technical support and emails were exchanged with their engineers. Additional research was performed utilizing and Dell PowerEdge R740 server.
Microsoft	Attempts were made to contact Microsoft technical support via phone and email, but we received no response. Microsoft provided research material primarily through its publicly accessible technical support web pages, as listed in the Reference section (7.0).

4.0 RESULTS AND DISCUSSION

4.1 Research Results Summary

In this section, we present the results of our research into the two primary Secure Boot deficiencies introduced in Section 2.0. The information gathered from UEFI specification, commercial vendors, and secure system vendors, as part of our research methodology is presented.

4.1.1 Secure Boot Operational Behavior

This section provides detail on the operation and behavior of Secure Boot. As not to repeat the information about Secure Boot operation presented in the Introduction (Section 2.0), this section is meant to provide additional detail into areas that help build the foundation for our results and recommendations going forward.

4.1.1.1 Secure Boot Modes

Secure Boot has different operating modes in which Secure Boot operations behave differently. The two primary modes we are concerned with are Setup Mode and User Mode. Setup mode effectively means that Secure Boot is not enforcing any validation behavior and has turned off authentication of Secure Boot key variables. User Mode is when Secure Boot is in EFI image validation enforcing mode and has enabled authentication of any operation that attempts to overwrite a Secure Boot key variable.

4.1.1.2 Authenticated Variables and Signature Databases

In the introduction, we introduced the Secure Boot variables that are at the core of both deficiencies, the PK, the KEK, the db, and dbx. Through reading the UEFI specification, we learned more about the structure of the Secure Boot variables and how they operate. All of the Secure Boot key variables are authenticated variables. This means that to overwrite Secure Boot certificates if Secure Boot is in User Mode, the payload containing the new certificates must be signed by a higher tier Secure Boot private key. The following is the Secure Boot digital signature hierarchy for overwriting Secure Boot variables:

- PK: Overwriting the PK variable while in User Mode requires that the old PK private key signs the payload containing the new PK certificate.
- KEK: Overwriting the KEK signature database variable while in User Mode requires that the current PK private key signs the payload containing the new KEK signature database.
- db: Overwriting the db signature database variable while in User Mode requires that the payload containing the new db signature database is signed by the current KEK private key or PK private key.
- dbx: Overwriting the dbx signature database variable while in User Mode requires that the payload containing the new dbx signature database is signed by the current KEK private key or PK private key.

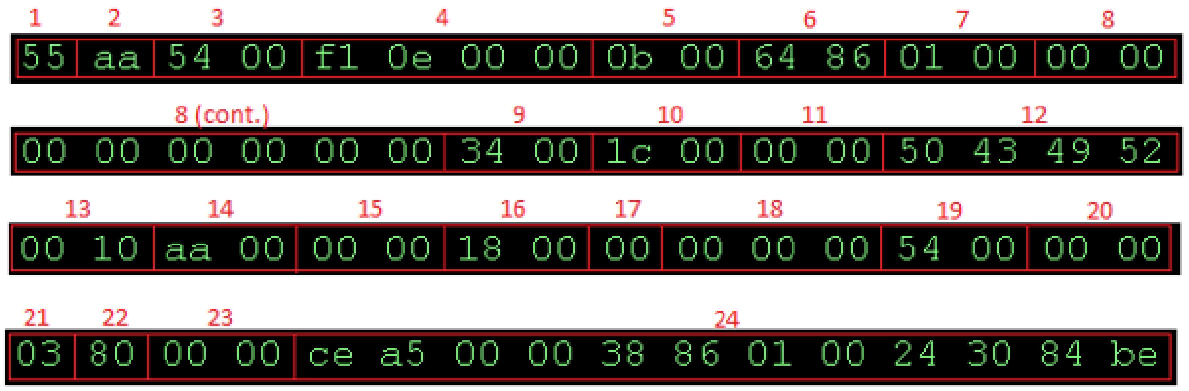
The KEK, db, and dbx are all signature database variables. Signature databases are a versatile Secure Boot construct that can contain many different forms of signature types. The signatures described here are not digital signatures, but data that can be of any of the forms within the UEFI signature type specification. Signatures are aggregated into lists, and the signature database variable can contain many signature lists, each with its own type. The aggregate of signature lists allows the KEK, db, and dbx to contain both DER-formatted X.509 certificates and hash values. In the UEFI Specification 2.8, section 32.4.1 goes into greater detail about how signature databases are formatted and lists all the types of signatures that are allowed within the signature data.

4.1.1.3 OpROMs

'Table 2 Recommended PCI Device Driver Layout' contains the recommended OpROM header from the UEFI specification v2.8, Table 135. 'Figure 4.1 OpROM Header of an actual OpROM file' contains a hex dump of the header from an actual OpROM file from Dell. Each section of the OpROM header is labeled to its corresponding section in the OpROM header table. While inspecting OpROM data in various drivers downloaded from Dell and HPE, different versions of the Peripheral Component Interconnect (PCI) Expansion ROM (PCIR) Data Structure were encountered. In Table 3, items 16 and 17 have the values 0x0018 and 0x00, respectively. These values correspond to version 2.2 of the PCIR Data Structure. A different file was encountered during our research that matched version 3.0 of the PCIR Data Structure. The values in items 16 and 17 in the version 3.0 PCIR data structure were 0x001c and 0x03, respectively. The significance of different versions impacts how a digital signing solution would be implemented. Specific implementations would require that a developer thoroughly inspect all OpROM headers to ensure as EFI data is being extracted, the proper location within the OpROM is being inspected based on the PCI version number.

Table 2 Recommended PCI Device Driver Layout

Offset	Byte Length	Value	Description
0x00	1	0x55	ROM Signature, byte 1
0x01	1	0xAA	ROM Signature, byte 2
0x02	2	XXXX	Initialization Size – size of this image in units of 512 bytes. The size includes this header
0x04	4	0x0EF1	Signature from EFI image header
0x08	2	XX 0x0B 0x0C	Subsystem Value from the PCI Driver's PE/COFF Image Header Subsystem Value for an EFI Boot Service Driver Subsystem Value for an EFI Runtime Driver
0x0a	2	XX 0x014C 0x0200 0x0EBC 0x8664 0x01c2 0xAA64	Machine type from the PCI Driver's PE/COFF Image Header IA-32 Machine Type Itanium processor type EFI Byte Code (EBC) Machine Type X64 Machine Type ARM Machine Type ARM 64-bit Machine Type
0x0C	2	XXXX 0x0000 0x0001	Compression Type Uncompressed Compressed following the UEFI Compression Algorithm.
0x0E	8	0x00	Reserved
0x16	2	0x0034	Offset to EFI Image
0x18	2	0x001C	Offset to PCIR Data Structure
0x1A	2	0x0000	Padding to align PCIR Data Structure on a 4 byte boundary
0x1C	4	'PCIR'	PCIR Data Structure Signature
0x20	2	XXXX	Vendor ID from the PCI Controller's Configuration Header
0x22	2	XXXX	Device ID from the PCI Controller's Configuration Header
0x24	2	0x0000	Reserved
0x26	2	0x0018	The length if the PCIR Data Structure in bytes
0x28	1	0x00	PCIR Data Structure Revision. Value for PCI 2.2 Option ROM
0x29	3	XXXX	Class Code from the PCI Controller's Configuration Header
0x2C	2	XXXX	Code Image Length in units of 512 bytes. Same as Initialization Size
0x2E	2	XXXX	Revision Level of the Code/Data. This field is ignored
0x30	1	0x03	Code Type
0x31	1	XX	Indicator. Bit 7 clear means another image follows. Bit 7 set means that this image is the last image in the PCI Option ROM. Bits 0–6 are reserved.
		0x00 0x80	Additional images follow this image in the PCI Option ROM This image is the last image in the PCI Option ROM
0x32	2	0x0000	Reserved
0x34	X	XXXX	The beginning of the PCI Device Driver's PE/COFF Image



```
55 aa 54 00 f1 0e 00 00 0b 00 64 86 01 00 00 00
00 00 00 00 00 00 34 00 1c 00 00 00 50 43 49 52
00 10 aa 00 00 00 18 00 00 00 00 00 54 00 00 00
03 80 00 00 ce a5 00 00 38 86 01 00 24 30 84 be
```

Figure 3. OpROM Header of an actual OpROM file

4.1.2 Microsoft Secure Boot Key Deployment

Per Microsoft Secure Boot online documentation, Microsoft works with hardware vendors to ensure that their certificates are deployed as default values in the KEK and db signature database variables on vendor hardware. Deploying in this way assists with the boot of Microsoft and OEM hardware booting if Secure Boot is enabled. Microsoft provides a service where they digitally sign OEM firmware with their db_{priv} key so the firmware can boot with the Microsoft db_{pub} certificate.. Controlling the PKI chain would be accomplished through the removal of default manufacturer-provided certificates, replacing them with ones provided by the secure system vendor.

4.1.3 Hardware Vendor Results

Table 3 Hardware Vendor Findings

Vendors	Findings
Hewlett Packard Enterprise (HPE)	HPE has proprietary boot software that functions similarly to Secure Boot and was not investigated due to its proprietary nature. Their proprietary boot solution was specific to their hardware and did not involve Secure Boot functionality. HPE has no current method of resigning OpROMs that come from 3rd-party device manufacturers. However, HPE does have the ability to generate hash values of pre-loaded EFIs found on the server. Additionally, HPE provides access to a UEFI Command Line Interface (CLI) that can manipulate Secure Boot authenticated variables.
Dell	Dell has no current method of resigning OpROMs that come from 3rd-party device manufacturers. Dell has a way to generate hash values of pre-loaded EFIs found on the on GEN 14 or later servers. Dell's solution is to have the server boot into a Secure boot discovery mode. Booting, in this manner, allows the server to discover preloaded EFIs, calculate their hash values, and add them to the db signature database variable.
Microsoft	Microsoft was unresponsive to questions and was unable to provide support to our Research into Secure Boot vulnerabilities.

4.1.4 Secure Boot Deficiency Results

4.1.4.1 3rd-Party OpROMs

The first deficiency discussed was the inability to re-sign 3rd party OpROMs with a custom key when Secure Boot certificates and hashes deployed by the HW manufacturer are removed. The research was done into how OpROMs could be loaded when Secure Boot is enabled, and a custom Secure Boot key chain is used. The procedures for testing candidate solutions for resigning OpROMs reside in Appendix A. The research was performed using a Dell PowerEdge R740 server and HPE DL380 server with OpROMs that were downloaded off of their respective websites. The OpROMs we experimented with were for a RAID controller on both manufacturers' system. The Dell RAID controller is the PERC H730P adapter. The HPE RAID controller is the HPE Smart Array P408 adapter. The UEFI specification contains a layout for the recommended data structure of OpROMs. However, since the UEFI specification does not specify or maintain the OpROM data structure format, there could be cases where a UEFI-compatible OpROM may not have the structure designated in the UEFI specification. The OpROM data structure is maintained in the PCI Firmware Specification, the latest version of which is 3.2. The PCI Device Driver Layout structure allows the individual Portable Executable/Common Object File Format (PE/COFF images (EFI files)) to be found and extracted from an OpROM. Through extensive experimentation using our test procedures in Appendix A, we attempted to create a proof-of-concept where we extracted EFI data, resigned it, and then repackaged it back into the OpROM structure. Ostensibly this should work since the signature data width is constantly given a designated digital signature algorithm. Unfortunately, we were unable to produce a bootable solution where a repackaged OpROM was able to boot under a customer Secure Boot certificate chain. We had difficulty in determining the exact location within the OpROMs where we would need to overwrite the EFI embedded digital signature bytes. We believe it is possible to develop a tool that can automatically parse OpROMs for EFI data and applying a digital signature in the correct location within the EFI data, but we were unable to develop this tool under this current research effort.

An alternative solution would be adding the hash values of the EFIs within approved OpROMs to the db. Hashes are an acceptable means of validating a bootable piece of software or firmware within Secure Boot. Hash values of bootable EFIs can be stored in the db variable. Dell and HPE both have methods to calculate hash values of pre-loaded EFIs and add them to the db variable. This means that if an OpROM has EFIs within, they can be hashed by UEFI firmware functionality, and those hashes enrolled into the db variable of Secure Boot. HPE granted our research team access to their firmware engineering staff to gain a deeper understanding of how they handle 3rd-party device firmware signing. The HPE firmware engineering team offered to provide access to a tool their team uses to dissect an OpROM and sign any interior PE/COFF image files (EFI files). Access to this tool was not available until after we had concluded this research effort.

4.1.5 Attempts to sign OpROMs

Throughout this research effort, many attempts to manually sign the OpROMs were explored. Had they been successful, this would have provided the first step towards a solution. Specifically, we sought to unpackage UEFI capsules (the packages containing OpROMs and are used to install OpROMs), sign the individual OpROMs, and repackage the UEFI capsule. Doing this would have allowed removal of all manufacturer certificates from Secure Boot key variables without rehashing the system. Ultimately, we were able to unpackage the OpROMs from the UEFI capsule, but any attempts to sign or reinsert the OpROMs failed. An alternative solution would be to put the hashes of the extracted OpROMs into the db and load the original OpROM from the vendor. Most UEFI platforms come with a boot option to access the UEFI shell. The UEFI shell is a direct command-line interface to the capabilities UEFI provides. Unfortunately, we were unable to perform the OpROM hash enrollment through the UEFI BIOS or in the UEFI shell. The HPE DL380 server that we used during testing allowed EFI hashes to be added to the db, but we were unable to extract an OpROM and present it to the UEFI BIOS such that it would add the hash to the db. Also, we could not manually insert a hash into the db or add one via the UEFI shell since this is not an allowed feature of hash enrollment within Secure Boot. The Dell server that used in testing did not have a UEFI shell option, and there was no mechanism to insert a hash into the db manually as well.

Some 3rd-party tooling exists aimed at working with EFI images, and a specific tool does exist that is intended to overwrite the digital signature of an EFI file. This tool is called sbsigntool and is available through the Ubuntu apt package manager. We attempted to sign OpROMs using sbsigntool. The sbsigntool can sign an EFI image with a either KEK_{priv} or db_{priv} keys from the Secure Boot keychain, have Secure Boot recognize the signature, and load the EFI image. This tool can detect signatures it applies, signs over top of them, and verify that a public certificate can be used to validate the signature on the image. While this tool functions for EFI files, it was not effective in overwriting the digital signatures within OpROMs.

Attempts at creating an OpROM resigning proof-of-concept were also made using tools from the tianocore/edk2 project on GitHub. Tianocore is an Intel reference implementation of the UEFI specification. Tianocore is frequently used by firmware developers to test if what they are creating is UEFI compliant. Initially, we tried the Rsa2048Sha256Sign.py script from EDK 2 to resign an OpROM. The tianocore wiki claims that this script is used to sign EFI images; however, we found this script did not apply the signature onto the file in the correct location, and its output did not match the output from sbsigntool. Additionally, this tool is unable to detect a signature that is placed onto a file, despite having a decode feature. This tool was able to sign any file that was provided, though attempts to re-sign a file using this tool would prepend the signature to the previously signed file, resulting in a file with two signatures.

Another tool that was experimented with was EfiRom, also a part of the tianocore/edk2 project on GitHub. EfiRom is used to create a UEFI capsule from EFI images and OpROMs. Basic validation of input files is performed, and OpROMs extracted from a UEFI capsule passed this validation, indicating that the extraction process is correct. When this tool adds an EFI image to the UEFI capsule, it prepends an OpROM header to the image, with each input file appended together into a final UEFI capsule. However, when examining an OpROM from a vendor, there is extra unknown data around the OpROM data within the vendor firmware file. Without a way to extract and modify the unknown data to mesh correctly with the signed OpROM data, we are unable to use this tool to create signed UEFI capsules.

4.1.5.1 Secure Boot Certificate Hierarchy

The other Secure Boot deficiency was that the PKI hierarchy within the Secure Boot certificates did not appear to be validated upon import or use of the KEK or db. Verification of the removal of the PKI chain has been performed on Dell PowerEdge R740 and HP DL380 servers. Initially, the issue was found on a Dell PowerEdge R640 platform. The test procedures to replicate this deficiency can be seen in section Appendix A. After discussing this issue further with the HPE firmware engineering team, they have clarified an aspect of the Secure Boot specification that we had misinterpreted. While Secure Boot does validate certificates that are imported into the authenticated variables, it does it in a slightly different way than the widely fielded PKI infrastructure most professionals use. Validation is not performed upon the use of the certificates in the Secure Boot variables, but upon the enrollment of the Secure Boot variable certificate and hash payloads. For Secure Boot to enroll new certificates and hash values in a variable, the payload providing the certificates and hash values must be signed by the proper tier private key. The following is a summary of which private key associated with each Secure Boot certificate must be used to sign the corresponding enrollment payload:

- **PK Certificate Enrollment:** The prior PK private key must sign the UEFI certificate payload if the PK variable is not currently empty. Otherwise, the payload can be unsigned.

- **KEK Certificate or Hash Enrollment:** The PK private key must sign the UEFI certificate payload to the KEK signature database variable.
- **db Certificate or Hash Enrollment:** The PK private key or KEK private key must sign the UEFI certificate/hash payload to the db signature database variable.
- **dbx Certificate or Hash Enrollment:** The PK private key or KEK private key must sign the UEFI certificate/hash payload to the dbx signature database variable.

These UEFI payloads are the validated elements, not the certificates themselves. Further analysis of the portion of the UEFI specification that details the SetVariable() function provided a clearer understanding that the Secure Boot key hierarchy is about authenticating payloads used to update the Secure Boot variables rather than providing a digital signature chain within the variable data directly. Specifically, the status codes in section 8.2.1 of the UEFI specification have a code called EFI_SECURITY_VIOLATION. This code is raised if the payload provided in the SetVariable() function does not pass the validation check carried out by the firmware. In conclusion, our interpretation of the Secure Boot certificate validation was incorrect given our PKI validation assumptions; thus, this is not an issue within Secure Boot. We feel this could result in a "gotcha" moment for secure system developers looking to understand the operation of the Secure Boot key hierarchy. PKI is presented as being part of the root of trust within Secure Boot. While it does provide protection to the variable write access, it does not protect in the way web-based PKI certificate validation operations occur. The following figure shows the format of UEFI authenticated variable input data, which can be used to update the PK, KEK, db, or dbx variables.

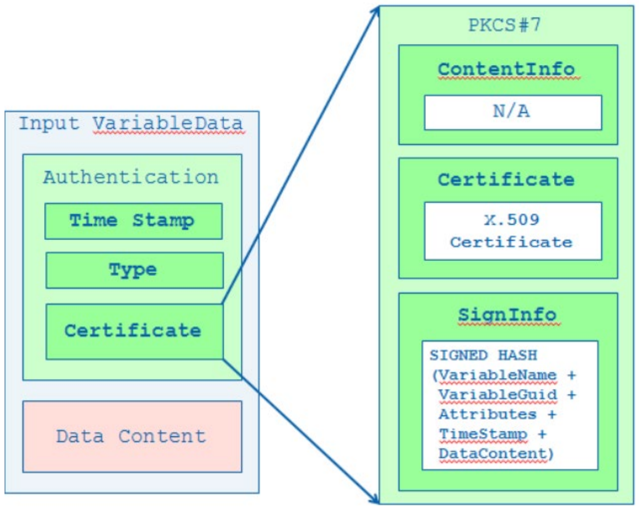


Figure 4. UEFI Authenticated Variable Data Format

The signature used to validate the incoming data is generated by a private key from a higher tier Secure Boot register. The only time the data is validated is when this data format is used to update a Secure Boot register, not upon use. See the following flow diagram to see which Secure Boot certificate is used to validate Secure Boot variable input data.

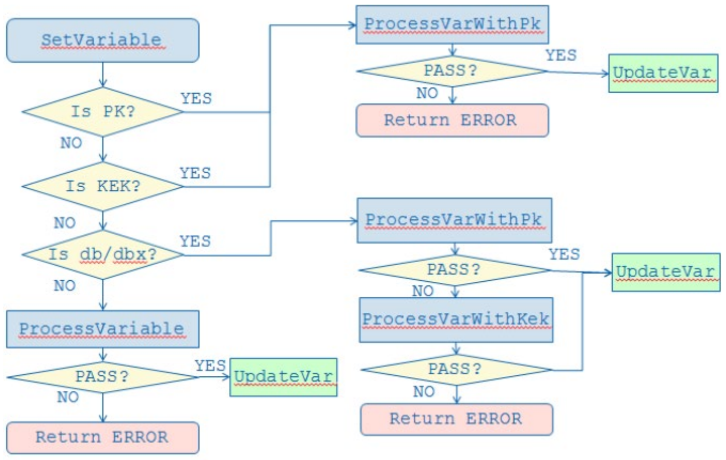


Figure 5. Secure Boot Variable Authentication Flow

Conversely, the PKI validation used in SSL on the internet uses a hierarchy of certificates to ensure a certificate on the net is validated by a trusted entity on the internet. See the following figure that shows how this validation is achieved.

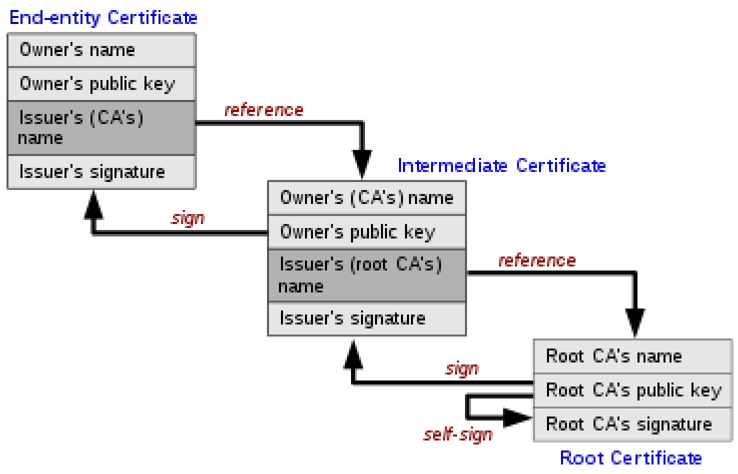


Figure 6. SSL PKI Chain of Trust Certificate Validation

Each time a certificate in SSL is used it will verify that the issuer signature is valid against the issuer's public X.509 certificate.

The language within the UEFI specification could be improved to make this clearer to the uninitiated reader.

4.2 Secure Boot Analysis

This section discusses the impacts of a Secure Boot enabled system from development to deployment.

4.2.1 Impacts during development

Identifying and developing a method of resigning OpROMs within a secure system platform may be quite time and budget consuming. Since we did not come to a proof-of-concept for an abstract way of doing this for all OpROMs, it would require custom tooling to be created by the secure system developers to be able to accomplish this. The same impact applies if the secure system developers attempt an OpROM hashing solution. That being said, once the method of either of these is found, vendor tooling could notionally be generated to reduce the impact of future OpROM support. Though it should be of note that such a solution would likely be a risk against future platform or vendor updates that are proprietary.

4.2.2 Operational impacts

The most significant impact of Secure Boot setup, deployment, and maintenance would be found in the maintenance phase due to required downtime to replace Secure Boot certificates and hashes in the Secure Boot variables. If a secure system relies on verifying OpROMs through embedded digital signatures, certificates would need to be replaced due to certificate expiration. This could be mitigated by giving certificates a long expiration period. If a secure system relies on the hash enrollment solution for verifying OpROMs, each time a piece of firmware is upgraded, the entire hash signature database would need to be overwritten in the Secure Boot db variable. Downtime can be mitigated by having an adequate system failover structure wherein Secure Boot certificates expiration dates are staggered. Firmware updates were not added as an operational impact because these updates are already part of assumed downtime structure in existing secure system maintenance.

4.2.3 Organizational impacts

Since the organization deploying the secure system does not need to maintain the keys, certificates, or hashes for Secure Boot, there is no perceived impact to an organization deploying a secure system. There is an impact on the secure system vendor as they must maintain a PKI solution that maintains Secure Boot keys for the various models of secure systems they manufacture as well as meeting the security requirements for each customer.

4.3 Analysis of the technical aspects of a Secure Boot Implementation

4.3.1 Summary of improvements

System boot integrity is more secure through proper setup, deployment, and maintenance of the Secure Boot system on a UEFI enabled platform. The guidance for the use of this capability requires the removal of all pre-loaded Secure Boot keys and that they are replaced with secure system vendor-generated PKI keys, certificates, and hashes. This eliminates the risk of tampering during shipment & secure system procurement and prevents commercial organizations (i.e., HW manufacturers) having their keys exposed, thus giving adversaries the ability to boot a malicious bootable software element on a secure system. It also prevents unwise or malicious boot actions performed by a system administrator by preventing them from booting into an alternate state using non-vendor supplied media.

Secure Boot allows EFI hashes to be used to validate bootable elements. In specific servers, it has been shown that EFI hashes can be generated and enrolled in the db if they are pre-loaded onto the system. Manufacturer loaded Secure Boot certificates, and hashes should first be removed from the PK, KEK, and db, and then using mechanisms on the hardware, regenerate the hash values of the loaded OpROMs.

4.3.2 Disadvantages and limitations

Development of new secure systems could be impacted if development teams are not experienced in UEFI system boot concepts, or lack the appropriate tools to perform PKI signature generation of UEFI boot elements. This research effort forced our team to dig into the depths of Secure Boot specification to understand even some of the more basic concepts related to Secure Boot key management. Without that knowledge, the replacement of manufacturer keys with certificates maintained by the secure system developer could result in misinterpretation and misunderstanding of how the key and certificate material is validated. Additionally, the lack of accessible and reliable Secure Boot signing tools for OpROMs limits the effectiveness of development teams if they need to use 3rd-party PCI devices critical to the operation of their secure system. At this time, there appears no reliable and readily available way to sign OpROMs that come with these PCI devices. Thus, it becomes quite problematic to boot them upon system startup when Secure Boot is enabled.

4.3.3 Alternatives and trade-offs considered

Alternatives to Secure Boot were not considered.

5.0 CONCLUSION

The research into Secure Boot demonstrates issues prevalent on multiple vendor platforms. The PKI chain, when removed and replaced, does not allow for an operating system to boot due to the inability to load OpROMs, and thus hard drives on RAID controllers would not likely be present. The removal can also affect other 3rd-party PCI devices such as network interface cards as well. Any device with an unsigned or invalid signature OpROM fails to load under the circumstance mentioned. These issues can be reproduced on multiple commodity hardware platforms. At this time, our research was unable to produce a proof-of-concept of a way to resign OpROMs to they can boot with a custom Secure Boot policy. We believe it is possible that a tool to resign OpROMs can be created, but requires additional investigation into the PCI firmware specification. It is possible to hash all OpROMs that are loaded into the system and enroll these hashes into the db Secure Boot variable. This hash enrollment offers a solution to the OpROM issue but comes with limitations as detailed in section 6.1.2.

The second issue encountered ended up being our misinterpretation of how the PKI chain within Secure Boot validates certificate and hash enrollment into authenticated variables. That being said, the UEFI specification could have made it more evident in the PK, KEK, db, and dbx enrollment sections how the variable authentication functions. This was only clearly stated in the SetVariable() function definition of the specification. The Secure Boot section of the specification relating to variable enrollments is quite general when it states how values are authenticated. The chain of keys in Secure Boot is meant to validate UEFI payloads for child certificate and hash value enrollment. This does not mean the data within the payload needs to be signed by the parent key, instead the UEFI payload containing update values for the child Secure Boot variable. The signing of the UEFI payload protects the Secure Boot variables from erroneous or malicious modification. Also, the payload validation allows more certificates and hash values to be added to the variables than just the ones being created by the secure system vendor, as long as the secure system vendor approves them with their private key. The validation structure maintains the same level of security since it still requires the secure system vendor PKI solution to sign the certificates and hashes being added to a Secure Boot authenticated variable (PK, KEK, db, or dbx).

Outreach to stakeholders is ongoing, with updates occurring daily. Nteligen has had modest success with getting in contact with secure system vendors thus far, and continue to reach out until a full understanding of each vendor's usage of Secure Boot is determined.

6.0 RECOMMENDATIONS

6.1.1 General Recommendations

If possible, we recommended that all secure systems should use secure system vendor-generated PKI certificate and hash material in Secure Boot authenticated variables for a given secure system deployment. At this time, if 3rd-party OpROMs hinder the operation of a secure system in this setup, it is recommended to use the EFI hash value enrollment solution to shed reliance on manufacturers deployed Secure Boot certificates.

7.0 REFERENCES

- https://uefi.org/about
- Unified Extensible Firmware Interface (UEFI) Specification. Version 2.8. March 2019
- https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/windows-secure-boot-key-creation-and-management-guidance
- "Microsoft Windows Authenticode Portable Executable Signature Format, Version 1.0" heading at "Links to UEFI-Related Documents" (http://uefi.org/uefi).
- Yao, Jiewen, Zimmer, Vincent J. "A Tour Beyond BIOS Implementing UEFI Authenticated Variables in SMM with EDKII". Sept. 2014. Intel.

APPENDIX A – SECURE BOOT TEST PROCEDURES

The following appendix provides test procedures used to replicate the Secure Boot deficiencies we identified. In combination with that, we have provided test procedures that explain how to reset the Dell and HPE server hardware back to a known good state as the procedures can involve overwriting firmware. If we don't have these procedures, the system could be in an invalid state ahead of executing a new test.

Additionally, we have provided test procedures detailing how to test a Secure Boot deficiency solution candidate against a specified server.

A.1 Test Procedures to Reproduce Secure Boot Deficiencies

The test procedures below reproduce the Secure Boot deficiencies on the Dell PowerEdge R740 and HPE DL380 servers. To research the deficiencies, a RAID controller was used as an exemplar 3rd-party device within a server that wouldn't boot if Secure Boot is enabled.

A.1.1 Secure Boot Deficiency Reproduction Requirements

The verification points (VP) below identify when the proper state of the test procedure has been reached to state if the Secure Boot deficiency has been reproduced accurately.

Verification Points

Each VP maps to a requirement. The requirement is a test plan concept we incorporated into the structure of the procedures used to reproduce the Secure Boot deficiencies. In a traditional test plan, each requirement is a contract requirement captured in a Requirements Traceability Verification Matrix (RTVM). For this research effort, we created two requirements that represent the end state of when a Secure Boot deficiency is reproduced. Each ID number should be associated with the RTVM ID for each requirement, but since we do not have an RTVM for this effort, we left these blank. The VP number is referenced within each Secure Boot deficiencies reproduction procedure, indicating at which step the procedure should have reproduced the deficiency.

Table A-1 Requirement to Verification Objective

VP	ID	Requirement	Verification
1.	N/A	Verify the trusted operating system or bootable software boots correctly.	Should not be able to boot into an unsigned bootloader.
2.	N/A	Verify that bootable devices are unavailable.	Should not be able to boot into unsigned devices.

A.1.2. Secure Boot Certificate Hierarchy Deficiency Reproduction Procedure

The following table contains procedures on how to reproduce the Secure Boot deficiency where the Secure Boot certificate structure does not validate according to specification.

Table A-2 Secure Boot Certificate Hierarchy Deficiency Reproduction Procedure

#	Steps/ Test	Description	Comment
1.	Step	**Test Machine (Secureboot):** Install CentOS	
2.	Step	**Test Machine (Secureboot):** Verify all default certs are available on PK, KEK, and db.	In the case of non-default configuration, revert all Secure Boot keys to the default configuration.

3.	Step	**Local Machine:** Generate the PK, KEK, and db private keys and self-signed certificates	$ openssl req -new -x509 -newkey rsa:2048 -subj "/CN=SecBoot PK/" -keyout PK.key -out PK.crt -days 3650 -nodes -sha256 -outform PEM
			
			$ openssl x509 -in PK.crt -out PK.cer -outform DER
			
			$ openssl req -new -x509 -newkey rsa:2048 -subj "/CN=SecBoot KEK/" -keyout KEK.key -out KEK.crt -days 3650 -nodes -sha256 -outform PEM
			
			$ openssl x509 -in KEK.crt -out KEK.cer -outform DER
			
			$ openssl req -new -x509 -newkey rsa:2048 -subj "/CN=SecBoot DB/" -keyout DB.key -out DB.crt -days 3650 -nodes -sha256 -outform PEM
			
			$ openssl x509 -in DB.crt -out DB.cer -outform DER

			The dbx is not created in this process since a list of invalid EFI binaries has not been considered yet. Once this research has been performed, the generation of the forbidden signature database is the equivalent of creating the DB above.
4.	Step	**Local Machine:** Create writable media that can be read by the secure system firmware.	Usually, this is FAT32. Media types must be allowed by the secure system deployment environment.
5.	Step	**Test Machine (Secureboot):** Verify Secure Boot is disabled	1. Turn on UEFI enabled PC 2. Enter BIOS 3. Ensure Secure Boot is disabled (if not, disable, reboot, and start from step 1)
6.	Test	**Test Machine (Secureboot):** Load PK into UEFI	1. Delete the pre-loaded PK certificate 2. Import the generated PK certificate from the media 3. Reboot 4. Enter UEFI BIOS 5. Enable Secure Boot 6. Reboot
7.	VP 1	**Test Machine (Secureboot):** Verify the trusted operating system or bootable software boots correctly	Expected: Should not be able to boot into unsigned bootloader Actual: Able to boot into the operating system UEFI BIOS allows you to boot into bootloader with Microsoft PK missing.

A.1.3 Third-party OpROM Deficiency Reproduction Procedure

The following table contains procedures on how to reproduce the Secure Boot deficiency when OpROMs are not signed by the enabled KEK or db certificates and fail to load during boot.

Table A-3 Third-party OpROM Deficiency Reproduction Procedure

#	Steps/ Test	Description	Comment
1.	Step	**Test Machine (Secureboot):** Install CentOS	
2.	Step	**Test Machine (Secure boot):** Verify all default certs are available on PK, KEK, and db.	In the case of non-default configuration, revert all signatures to the default configuration.

3.	Step	**Local Machine:** Generate the PK, KEK, and db private keys and self-signed certificates	$ openssl req -new -x509 -newkey rsa:2048 -subj "/CN=SecBoot PK/" -keyout PK.key -out PK.crt -days 3650 -nodes -sha256 -outform PEM
			
			$ openssl x509 -in PK.crt -out PK.cer -outform DER
			
			$ openssl req -new -x509 -newkey rsa:2048 -subj "/CN= SecBoot KEK/" -keyout KEK.key -out KEK.crt -days 3650 -nodes -sha256 -outform PEM
			
			$ openssl x509 -in KEK.crt -out KEK.cer -outform DER
			
			$ openssl req -new -x509 -newkey rsa:2048 -subj "/CN= SecBoot DB/" -keyout DB.key -out DB.crt -days 3650 -nodes -sha256 -outform PEM
			
			$ openssl x509 -in DB.crt -out DB.cer -outform DER

			The dbx is not created in this process since a list of invalid EFI binaries has not been considered yet. Once this research has been performed, the generation of the forbidden signature database is the equivalent of creating the db above.
4.	Step	**Local Machine:** Create writable media that can be read by the secure system firmware.	Usually, this is FAT32. Media types must be allowed by the secure system deployment environment.
5.	Step	**Test Machine (Secureboot):** Verify Secure Boot is disabled	1. Turn on UEFI enabled PC 2. Enter BIOS 3. Ensure Secure Boot is disabled (if not, disable, reboot, and start from step 1)
6.	Test	**Test Machine (Secureboot):** Load PK into UEFI	1. Delete the pre-loaded PK certificate 2. Import the generated PK certificate from the media
7.	Step	**Test Machine (Secureboot):** Load KEK certificate into UEFI	1. Delete the pre-loaded KEK certificate 2. Import the generated KEK certificate from the media.
8.	Step	**Test Machine (Secureboot):** Load DB certificate into UEFI	1. Delete the pre-loaded db certificate 2. Import the generated db certificate from the media
9.	Step	**Test Machine (Secureboot):** Enable Secure Boot	1. Reboot 2. Enter UEFI BIOS 3. Enable Secure Boot 4. Reboot
10.	VP 2	**Test Machine (Secureboot):** Verify bootable devices are unavailable	Expected: Should not be able to boot into unsigned devices Actual: Should not be able to boot into unsigned devices Currently, there is no way to sign Options ROMs.

A.1.4 Overwrite RAID Controller Firmware Procedures

This test procedure details how to overwrite the RAID Controller firmware. This procedure was created because a fundamental process on how to overwrite firmware on our candidate 3rd-party test device (RAID controller) had to be set in stone. We did not want to enter a system state where we did not know the version of the firmware of a device before executing a test of a proof-of-concept solution. There are two VPs for this procedure since we created procedures that are specific to each of the hardware platforms we were testing on; both the Dell PowerEdge R740 and the HPE DL380.

Verification Procedures

Table A-4 Requirement to Verification Objective for RAID Controller

VP	ID	Requirement	Verification
1.		Verify version overwrite of RAID Controller firmware on Dell PowerEdge R740	Verify version change
2.		Verify version upgrade of RAID Controller firmware on HPE DL380	Verify version change

Overwrite RAID Controller Firmware on Dell Procedure

Table A-5 Overwrite RAID Controller Firmware on Dell PowerEdge R740

#	Steps /Test	Description	Comment
1.	Step	**Local Machine:** Download firmware BIN file	1. Visit DELL website 2. Locate the latest version of firmware for RAID Controller 3. Download file
2.	Step	**Local Machine:** Create USB Drive	1. Insert a USB drive into the computer 2. Right-click USB 3. Select format drive 4. Format to FAT32 Media types must be allowed by the secure system deployment environment. The USB is used for the storage of firmware files.
3.	Step	**Local Machine:** Transfer BIN file into the USB root directory	
4.	Step	**Test Machine (Secureboot):** Verify Secure Boot is disabled	1. Turn on UEFI enabled PC 2. On system boot, enter Ctrl-Alt-Del to bring up the menu 3. Enter <F2> for System Setup 4. Enter "System BIOS" 5. Enter "System Security" 6. Set Secure Boot to "Disabled" 7. Click Finish 8. Enter Yes in Exit dialog 9. On system boot, enter Ctrl-Alt-Del to bring up the menu 10. Enter <F2> for System Setup 11. Enter "System BIOS" 12. Enter "System Security" 13. Verify Secure Boot is "Disabled"
5.	Step	**Test Machine (Secureboot):** Enter OS Shell	1. Wait for system boot 2. Boot off the hard drive

			3. At OS shell enter user credentials and password
6.	Test	**Test Machine (Secureboot):** Mount USB drive	1. Insert a USB drive into the server 2. Use “fdisk –l“ to list devices 3. Find /dev device mapping for USB drive from the output 4. Create a mount directory 5. Mount <deviceName> <mountDirectory>
7.	Step	**Test Machine (Secureboot):** Execute BIN file	1. Change directory to <mountDirectory> 2. Copy BIN file to the user home 3. Shell execute BIN File Note: Resolve package dependency errors
8.	Step	**Test Machine (Secureboot):** Reboot into BIOS	1. Enter reboot in shell 2. On system boot, enter Ctrl-Alt-Del to bring up the menu 3. Enter <F2> for System Setup 4. Enter “System BIOS”
9.	VP 1	**Test Machine (Secureboot):** Verify version upgrade of RAID Controller firmware	1. Enter ‘Device Settings’ 2. Enter Raid Controller Configuration 3. Enter ‘Controller Management’ Expected: Firmware version number changed. Actual: Firmware version number changed.
10.	Step	**Test Machine (Secureboot):** Revert machine to known good version	Repeat 3 – 10 with known good firmware version to revert to clean slate. **Known good firmware:** RAID Controller: Name: PERC H730P Adapter Serial Number: 78800KH Package Version: 25.5.0.0018 Firmware Version: 4.270.00-8112 NVDATA Version: 3.1511.00-0014 Expected: Firmware version number changed to a known good state. Actual: Firmware version number changed to a known good state.

Overwrite RAID Controller Firmware on HPE Procedure

Table A-6 Overwrite RAID Controller Firmware on HPE DL380

#	Steps /Test	Description	Comment
1.	Step	**Local Machine:** Download firmware RPM file	1. Visit HP website 2. Locate the latest version of firmware for RAID Controller 4. Download file
2.	Step	**Local Machine:** Create USB Drive	1. Insert the USB drive into the computer 2. Right-click USB 3. Select format drive 4. Format to FAT32 Media types must be allowed by the secure system deployment environment. The USB is used for the storage of firmware files.
3.	Step	**Local Machine:** Transfer RPM file into the USB root directory	
4.	Step	**Test Machine (Secureboot):** Verify Secure Boot is disabled	1. Turn on UEFI enabled PC 2. Enter <F9> for System Utilities 3. Enter <F1> for Continue 4. Enter “System Configuration” 5. Enter “BIOS/Platform Configuration” 6. Enter “Server Security” 7. Enter “Secure Boot Settings” 8. Set “Attempt Secure Boot” to “Disabled” 9. Enter “Commit Changes and Exit” 10. Click Exit 11. Enter <F12> Save Changes and Exit 12. Enter “Ok” in the popup dialog 13. Enter “Reboot” in the popup dialog 14. Wait for the machine to reboot 15. Enter <F9> for System Utilities 16. Enter <F1> for Continue 17. Enter “System Configuration” 18. Enter “BIOS/Platform Configuration” 19. Enter “Server Security” 20. Enter “Secure Boot Settings” 21. Verify “Attempt Secure Boot” is “Disabled”

5.	Step	**Test Machine (Secureboot):** Enter OS Shell	1. Wait for system boot 2. Boot off the hard drive 4. At OS shell enter user credentials and password
6.	Test	**Test Machine (Secureboot):** Mount USB drive	1. Insert a USB drive into the server 2. Use “fdisk –l“ to list devices 3. Find /dev device mapping for USB drive from the output 4. Create a mount directory 5. Mount <deviceName> <mountDirectory>
7.	Step	**Test Machine (Secureboot):** Execute BIN file	1. Change directory to <mountDirectory> 2. Copy BIN file to the user home 3. Shell execute BIN File Note: Resolve package dependency errors
8.	Step	**Test Machine (Secureboot):** Reboot into BIOS	1. Enter reboot in shell 2. Enter <F9> for System Utilities Note: To see whether or not the OpROM was loaded, watch the checkbox on the lower right part of the screen during the boot process. If a check appears in the Smart Array box, the OpRom was loaded
9.	Step	**Test Machine (Secureboot):** Navigate to RAID controller management	1. Enter System Information 2. Enter Firmware Information
10.	VP 2	**Test Machine (Secureboot):** Verify version upgrade of RAID Controller firmware	Expected: HP SmartArray firmware version number changed. Actual: HP SmartArray firmware version number changed.
11.	Step	**Test Machine (Secureboot):** Revert machine to clean slate	Repeat 3 – 10 with desired --firmware version-- to revert to clean slate. Expected: Firmware version number changed to the previous state. Actual: Firmware version number changed to the previous state.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

AFRL - Air Force Research Laboratory
AFT - Assured File Transfer
AGIS - Advanced Guard of Information Systems
BIOS - Basic Input/Output System
CONOPS - Concept of Operations
CVE - Common Vulnerabilities and Exposures
db - Signature Database
dbx - Forbidden Signature Database
EFI - Extensible Firmware Interface
GD - General Dynamics
HPE - Hewlett Packard Enterprise
HW - Hardware
KEK - Key Exchange Key
OpROM - Option Read Only Memory
OS - Operating systems
PE/COFF - Portable Executable Common Object File Format
PKI - Public Key Infrastructure
RAID - Redundant Array of Independent Disks
SCSI - Small Computer System Interface
UEFI - Universal Extensible Firmware Interface