Setup For Failure: Defeating Secure Boot

Corey Kallenberg *Sam Cornwell* *Xeno Kovah*
John Butterworth
`ckallenberg@mitre.org, scornwell@mitre.org`
`xkovah@mitre.org, jbutterworth@mitre.org`
The MITRE Corporation

Abstract

Secure Boot is a new UEFI feature that enforces a signature check on the boot loader before the firmware transfers control to the boot loader. This feature prevents the traditional ”bootkit” style of attack that infects the MBR in an effort to circumvent the operating system kernel while it is being loaded. However, UEFI implementations have added flexibility to how and when this policy is enforced. In order to be secure, this ”flexibility” is typically configured by the OEM. We will show how the standard UEFI interface provided to the operating system can be abused to re-configure the secure boot policy such that a malicious bootloader will be called by the firmware. Furthermore, we analyze the chipset protection mechanisms of the UEFI variable region of the SPI flash and note that they are necessarily weaker than the protection applied to the UEFI code region. We then show how SMI suppression can be used to allow ring 0 code to write directly to critical UEFI variables, bypassing any cryptographic authentication scheme normally employed by the UEFI Runtime Serivices. Both of these attacks can effectively disable Secure Boot, all while Secure Boot still reports itself as enabled.

1 Secure Boot Introduction

It is adventageous for malware to execute as early as possible on the system. This advantage is due to the fact that malicious code is generally capable of subverting legitimate code that is executed at a later time. For instance, BIOS level rootkits are capable of subverting most other code that will execute on the platform because the BIOS code executes immediately after platform reset. Another example is “bootkits”, which can subvert operating system kernels at load time. Bootkits have recently come back into vogue because of their ability to bypass the Windows driver signing enforcement [7]. Already we have seen bootkits for UEFI systems [5], but in practice these bootkits would be prevented by the UEFI security feature known as Secure Boot.

1.1 Secure Boot Design

Secure Boot is designed to prevent the execution of unauthorized code during the system boot up. This includes any code that is found that may need to be executed during the system start: for instance, PCI option ROMs and operating system boot loaders. Whenever one of these executables is discovered, the UEFI firmware checks if the executable is signed with an authorized key, or if a hash of the executable is stored in the authorized database. The authorized keys are stored in a UEFI variable known as the “KEK” and the authorized database is another UEFI variable known as the “DB”. There are other UEFI variables that Secure Boot may use such as a blacklist database of unauthorized images, but these are irrelevant for the attacks presented in this paper.

The open source UEFI reference implementation[2] allows us to look at the recommended implementation of Secure Boot. In the reference implementation, the function “DxeImageVerificationHandler” is called whenever an executable image is discovered that needs to be authorized. The decision of whether or not to proceed with image authorization is dependent on the origin of the executable in question. Four cases are considered: IMAGE_FROM_FV, IMAGE_FROM_OPTION_ROM, IMAGE_FROM_REMOVABLE_MEDIA, IMAGE_FROM_FIXED_MEDIA. IMAGE_FROM_FV represents an executable image that was found on the SPI flash. In the reference implementation, these executables are allowed to execute without any authorization. This policy makes sense when you consider that the executable contents of the SPI flash were verified during the platform firmware update process, assuming signed firmware update enforcement is enabled. If signed firmware updates are not enabled, or implemented incorrectly, then Secure Boot can be trivially bypassed [3]. This is because an attacker can insert malicious code onto the flash chip, which is always assumed to be trusted. In this paper, we assume signed firmware updates are enabled, and the policy is implemented correctly.

The remaining image origin cases (IMAGE_FROM_OPTION_ROM, IMAGE_FROM_REMOVABLE_MEDIA, IMAGE_FROM_FIXED_MEDIA) have their policies individually configured. In other words, the reference implementation allows a granular image authorization policy. One example policy may permit unsigned option ROMs to run, but deny the execution of unsigned executables found on the hard disk (such as the boot loader). The authors speculated that some OEMs would configure the Secure Boot policy to permit unsigned option ROMs to run, in order to allow aftermarket PCI expansion cards (such as video cards) to work seamlessly. This hypothesis led us to analyze the Secure Boot policies and implementations of real consumer-grade hardware.

1.2 SecureBoot Implementation

In practice, an OEM's implementation of the UEFI firmware can diverge quite significantly from the UEFI open source reference implementation. This is true of the Secure Boot sub-component of UEFI as well. While attempting to discover the Secure Boot policy for option ROMs on a Dell Latitude E6430 at revision A12, the following code was discovered:

```
lea      rax, gSetupVariableData
lea      rcx, VariableName ; "Setup"
mov      [rsp+38h+Data], rax ; Data
mov      rax, cs:gRuntimeServices
xor      r8d, r8d        ; Attributes
mov      [rsp+38h+argSetupVariableSize], 0C5Eh
call     [rax+EFI_RUNTIME_SERVICES.GetVariable
...
cmp      cs:gSecureBootPolicyFromSetup, cl
jnz      short policy_from_setup_variable
hardcode_policy:
mov      cs:gImageFromFVPolicy, cl
mov      cs:gImageFromXromPolicy, 4 ;DENY
mov      cs:gImageFromRemovablePolicy, 4
mov      cs:gImageFromFixedPolicy, 4
mov      cs:gSecureBootPolicyFromSetup, cl ;0
```

The above code initializes the Secure Boot policy that will later be used by DxeImageVerificationHandler. The critical observation is that the Secure Boot policy can originate from either the UEFI "Setup" variable, or it can be hardcoded. The existence and contents of the Setup variable determines which case will occur. The address of the "gSecureBootPolicyFromSetup" byte that is tested above, is *within* the buffer populated via the call to "GetVariable." Also within this buffer are the control variables "gImageFromFVPolicy", "gImageFromXromPolicy" and "gImageFromRemovablePolicy". Hence the contents of the Setup variable can entirely specify the Secure Boot policy of the target.

An observant reader will notice that the default hardcoded policy is to deny any unsigned executables that originate from fixed drives, removable drives or option ROMs. We did not find any cases of a relaxed option rom policy on the systems we investigated. Because of this, our attention was focused onto controlling the contents of the Setup variable.

2 Attacking The Setup Variable

UEFI variables can have several attributes that affect the security and writability of their contents. These attributes are encoded as a bitmask in the header information on the SPI flash where the UEFI variables are stored. Variables can be marked as "Authenticated", meaning that the contents of the variable can only be changed if knowledge of an authorized private key is known. The KEK and DB mentioned above are stored as authenticated variables, meaning that malicious code cannot arbitrarily change their contents. Alternatively, non-volatile variables that are marked with the "Runtime" attribute but lack the authenticated attribute, can be arbitrarily modified by the operating system. Windows 8 has introduced a new API that allows a privileged userland process to interact with UEFI variables. This API is available through the "GetFirmwareEnvironmentVariable" and "SetFirmwareEnvironmentVariable" functions. Hence, non-authenticated UEFI variables can be modified by a privileged Windows 8 userland process.

Importantly, the Setup variable expresses the runtime attribute, but lacks the authenticated attribute. Thus, a Windows 8 process running with administrator privileges could arbitrarily modify the contents of the Secure Boot policy. Then, the malicious administrator process could replace the legitimate Windows boot loader with a malicious one that subverts the Windows kernel at load time. This is a clear violation of Secure Boot, as this is exactly the kind of scenario that Secure Boot was designed to prevent.

2.1 Attribution

The Setup variable vulnerability we discovered was not present in the open source UEFI reference implementation. Furthermore, the vulnerability was discovered in multiple OEMs. From this, one can conclude the vulnerability was introduced by an Independent BIOS Vendor (IBV). IBVs often supply OEMs with frameworks on which the OEMs build the firmware for their individual systems. Analysis of the strings and GUIDs embedded in the firmware of the vulnerable systems leads us to conclude that the vulnerability was probably introduced by American Megatrends Inc.

2.2 Disclosure

This particular vulnerability was co-discovered by Intel and MITRE. Intel first presented this vulnerability at Black Hat USA 2013[3]. Although, they did not give details about the vulnerability as it was still unpatched at that time. Coincidentally, MITRE also discovered this vulnerability shortly after the Intel Black Hat USA presentation, while investigating option rom policies as discussed above. This vulnerability has been assigned tracking number CERT-758382.

3 SPI Flash Protections

UEFI non-volatile variables are placed on the SPI flash chip that also hosts the UEFI code. This presents an interesting scenario, as the UEFI code should not be arbitrarily modifiable, unlike UEFI variables which do have to remain writable so they can be updated by the operating system. Intel provides a number of SPI flash protection mechanisms that allow the necessary granular protection policy to be implemented.

3.1 BIOS_CNTL

The BIOS_CNTL register, found on Intel IO Control Hubs (ICHs) on older systems and Platform Control Hubs (PCHs) on newer systems, contributes several important bits towards the protection of the SPI flash. The BIOS Write Enable (BWE) bit is a writeable bit defined as follows. If BWE is set to 0, the SPI flash is readable but not writeable. If BWE is set to 1, the SPI flash is writeable. The BIOS Lock Enable bit (BLE), if set, generates a System Management Interrupt (SMI) if the BWE bit is written from a 0 to 1. The BLE bit can only be set once, afterwards it is only cleared during a platform reset. It is important to notice that the BIOS_CNTL register is not explicitly protecting the flash chip against writes. Instead, it allows the OEM to establish an SMM routine to run in the event that the BIOS is made writeable by setting the BWE bit. The expected mechanism of this OEM SMM routine is for it to reset the BWE bit to 0 in the event of an illegitimate attempt to write enable the BIOS.

The BIOS_CNTL register also provides a "SMM BIOS Write Protect" (SMM_BWP) bit on newers systems which only allows writes to the SPI flash chip while all processors are in SMM. However only 6 of the 8005 systems we surveyed[1] set this bit. This is potentially due to SMM_BWP being a relatively new feature. Due to the fact SMM_BWP is largely unused at this time, we do not consider it in the attacks presented in this paper.

3.2 Protected Range

Intel specifies a number of Protected Range registers that can also protect the flash chip against writes and/or reads. These 32bit registers specify Protected Range Base and Protected Range Limit fields that set the relevant regions of the flash chip for the Write Protection Enable and Read Protection Enable bits. When the Write Protection Enable bit is set, the region of the flash chip defined by the Base and Limit fields is protected against writes. Similarly, when the Read Protection Enable bit is set, that same region is protected against read attempts. The HSFS.FLOCKDN bit, when set, prevents changes to the Protected Range registers. Once set, HSFS.FLOCKDN can only be cleared by a global reset of the system.

3.3 Analysis

The Protected Range registers cannot be used to write-protect the flash region associated with UEFI variables, because this region has to remain writable during the operating system runtime. Therefore, OEMs are forced to rely on BIOS_CNTL protection of the UEFI variable region. A related result[6] demonstrates that sole reliance on BIOS_CNTL protection allows an SMM-present attacker write access to the flash. It follows that BIOS_CNTL protection is significantly weaker than Protected Range protection, as the attack surface against BIOS_CNTL includes any vulnerabilities that may allow an attacker to execute code in SMM. This paper further strengthens this point by demonstrating that *sole reliance on the BIOSWE flash protection mechanism is vulnerable to an attacker that can temporarily suppress SMIs.* With SMIs temporarily suppressed, a ring 0 attacker can set BIOSWE to 1, and the SMI that normally sets BIOSWE back to 0 will fail to execute. The ring 0 attacker can then make arbitrary writes to any flash regions that are not protected by Protected Range registers, which is necessarily the case for the UEFI variable region.

4 SMI Suppression

The crux of this attack is finding a way to temporarily suppress the execution of SMM. An inspection of the Intel datasheets[4] describes at least one way that SMIs can be disabled. The "SMI Control and Enable Register" (SMI_EN) contains the "GBL_SMI_EN" bit. Intel

[1]Survey conducted with Copernicus[1]

describes this bit as follows: When set to 0, no SMI will be generated by the PCH. When the SMI_LOCK bit is set, this bit cannot be changed. Of the 8005 systems we surveyed, 3216 (approximately 40%) did not set the SMI_LOCK bit. A greater number of systems could be made vulnerable to this by downgrading their BIOS revision, an operation that is typically allowed.

For systems that fail to set SMI_LOCK, the attack is relatively straightforward, assuming an attacker with ring 0 privileges. First the attacker temporarily disables SMIs by setting GBL_SMI_EN to 0. Next an attacker sets BIOSWE to 1, allowing write access to any flash regions not protected by Protected Range registers. Now an attacker is able bypass the normal authentication scheme that protects UEFI authenticated variables, and modify their contents by writing directly to the flash. From here, an attacker has a number of ways by which they can defeat Secure Boot. The method we chose was to insert into the DB variable a hash of our bootkit. We then replaced the legitimate Windows boot loader with our bootkit and reset the system. Upon reset, the UEFI firmware will transfer control to our bootkit as it is now indicated to be a legitimate image by way of the DB variable.

5 Conclusion

The authors believe Secure Boot is a valuable security mechanism that is needed to protect the boot process. However vendor implementations of this security feature need improvement before it can be considered robust. Vendors should not be exposing security critical configuration settings through non-authenticated UEFI variables. Furthermore, OEMs should ensure that they are using all of the flash protection mechanisms that Intel provides. In particular, the decision to not use the SMM_BWP bit is a poor one. Failure to set SMM_BWP results in security critical UEFI authenticated variables being exposed to an attacker who can temporarily SMIs. If OEMs were properly using SMM_BWP, the authenticated variables would only be vulnerable to an attacker who could execute arbitrary code in SMM. This is a much stronger security position.

References

[1] Copernicus: Question your assumptions about bios security. `http://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/copernicus-question-your-assumptions-about`. Accessed: 10/01/2013.

[2] Uefi Reference Implementation. `tianocore.sourceforge.net`.

[3] Y. Bulygin, A. Furtak, and O. Bazhaniuk. A tale of one software bypass of windows 8 secure boot. In *BlackHat*, Las Vegas, USA, 2013.

[4] Intel Corporation. Intel 7 Series Family Platform Controller Hub Datasheet. `http://www.intel.com/content/www/us/en/chipsets/7-series-chipset-pch-datasheet.html`.

[5] S. Kaczmarek. UEFI and Dreamboot. In *Hack In The Box*, Amsterdam, 2013.

[6] C. Kallenberg, J. Butterworth, X. Kovah, and C. Cornwell. Defeating Signed BIOS Enforcement. In *EkoParty*, Buenos Aires, 2013.

[7] P. Kleissner. Stoned Bootkit. In *BlackHat*, Las Vegas, USA, 2009.