Attacking UEFI Boot Script

Rafal Wojtczuk
Bromium, Inc.
Email: rafal@bromium.com

Corey Kallenberg
Email: coreykal@gmail.com

January 1, 2015

***Abstract*—UEFI Boot Script is a data structure interpreted by UEFI firmware during S3 resume. We show that on many systems, an attacker with ring0 privileges can alter this data structure. As a result, by forcing S3 suspend/resume cycle, an attacker can run arbitrary code on a platform that is not yet fully locked. The consequences include ability to overwrite the flash storage and take control over SMM.**

I. Introduction

The attacks described in this paper are specific to certain functionality (named "Boot Script") of UEFI. Nowadays the vast majority of PCs run UEFI firmware. The full UEFI specification can be found at [1].

One of responsibilities of the firmware is to lock the platform so that after the operating system has been started, even code with ring0 privileges cannot modify flash storage or SMM at will. The reason is both stability and security - if the operating system is infected with malicious ring0 code, then in absence of flash and SMM locking, malware can install backdoors or rootkits in SMM [8] or flash [9], thus gaining unmatched persistence or invisibility. Additionally, control over SMM allows bypass of Intel TXT (in the common scenario when STM is not used) [7].

II. UEFI Boot Script concept

Shortly, UEFI Boot Script is a data structure (stored in memory) interpreted by UEFI firmware during S3 resume. The details can be found in [2]; the actual data structures are defined in [3]. The most relevant quote from [2] is:

... the Framework provides a boot script that lets the S3 resume boot path avoid the DXE phase altogether, which helps to maximize optimum performance. During a normal boot, DXE drivers record the platforms configuration in the boot script, which is saved in NVS. During the S3 resume boot path, a boot script engine executes the script, thereby restoring the configuration.

III. The possible vulnerability

UEFI Boot Script consists of a sequence of "opcodes" and their arguments. For our purposes, the most interesting one is EFI_BOOT_SCRIPT_DISPATCH_OPCODE. It takes a single argument, a function address X. Firmware interprets this opcode by executing code at X. It means that if we can achieve any of the below:

1) Alter the content of the Boot Script (insert a custom dispatch opcode)
2) Alter the target of any of existing EFI_BOOT_SCRIPT_DISPATCH_OPCODE
3) Alter the data structures used by firmware to locate the Boot Script

then we can force S3 suspend/resume cycle and run arbitrary code in the context of the Boot Script interpreter.

The gain for the attacker is that at the moment of Boot Script execution, usually many of the platform registers are not locked yet. Examples are *BIOS_CNTL* (that controls ability to overwrite SPI flash), or *TSEG* (that controls SMM protection). Even if ring0 privileges are required to alter the Boot Script execution, it still means privilege escalation, as explained in the introduction.

Note that modern platforms have a lot of lockable registers with various interesting semantics; for this paper, we focus on ones related to flash write protection and SMM.

IV. Securing the Boot Script

There is a way to store the Boot Script and related data structures securely, namely, store them in SMM memory, because operating system cannot alter SMM. EFI Development Kit sources [4] includes "Lockbox" concept, whose purpose is precisely to keep certain data structures in a secure location.

However, it seems that the actual UEFI implementations by popular vendors are not secure. Our experimentation showed that on all tested systems, it is possible to conduct one of the three attacks enumerated in the previous section, resulting in Boot Script compromise.

V. Case studies

We observed that the Boot Script-related code differs significantly across vendors (even the format of the Boot Script varies). Therefore, reverse engineering of firmware modules is necessary to build a successful exploit. A good way to start is to search the EFI modules for the EFI_PEI_S3_RESUME_PPI_GUID, defined in EDK as **0x4426CCB2, 0xE684, 0x4a8a,** {**0xAE, 0x40, 0x20, 0xD4, 0xB0, 0x25, 0xB7, 0x10**}; the Boot Script interpreter should be in the module implementing this PPI.

A. Dell E6540

On this system, the location of the Boot Script is determined in the following way:

1) The EFI variable **AcpiGlobalVariable** is read into X

2) X is treated as a pointer; another pointer B is retrieved from the memory at X+0x18. B holds the address of the Boot Script.

On the test system, the actual value of X was 0xc9fe7e18. According to EFI memory map, this address was in "ACPI NVS" region. Such regions contain ACPI private data, and the operating system normally does not use it in order to not destabilize the system. However, this is just normal RAM, and there is nothing that prevents OS from altering this memory[1]. The actual attack (naturally, it requires access to physical memory, so normally kernel privileges) consists of the following steps:

- Place a shellcode S at physical address Y
- Build a custom boot script, by concatenating EFI_BOOT_SCRIPT_DISPATCH_OPCODE(Y) with the original boot script; place it at physical address Z
- Overwrite the pointer to Boot Script (in the above example, at 0xc9fe7e18+0x18) with Z
- Force S3 suspend/resume cycle[2].

Experimentation shows that S executed successfully, and during its execution the state of the platform was:

1) no flash write protection
2) Access to SMM memory (controlled by SMRR registers) already locked
3) TSEG register not locked

The first point requires no comment. The second one means that PEI phase has already configured SMRR registers, and CPU cannot access SMM memory outside of SMM mode. However, TSEG register (that normally points to SMM region) was not locked. This register defines what memory region should be protected against DMA. As a result, the shellcode S was able to set TSEG to some dummy value (0xff000001). At this stage, shellcode S has the ability to access SMM memory via DMA.

However, as the shellcode executes in a very primitive environment, actually setting up a DMA transaction in this shellcode would require manually programming a piece of PCI hardware, which requires a lot of work. The better solution is to exit shellcode and let S3 resume complete[3]. Then, scheduling an arbitrary DMA from the operating system level is much simpler, as we can use the existing drivers from the operating system [5]. We have verified it was indeed the case - after S3 resume completed, we could access SMM memory via DMA transaction from SMM memory to a file on a hard drive. Particularly, it was possible to replace or hook the original SMI handler with a custom code, thus gaining ability to execute arbitrary code in SMM.

[1]In the past, there were known cases when e.g. SMM stored critical data in ACPI NVS, allowing for privilege escalation from OS to SMM; see e.g. the "bonus track" in [6].

[2]On Linux, it can be done via *rtcwake* utility; on Windows, one can use Task Scheduler to force a wake up from S3 sleep.

[3]Apparently, locking TSEG to a dummy value did not affect stability of S3 resume.

B. Systems behaving in a similar way

- Dell E6430
- HP EliteBook 850G1
- dq57tml board, AMI firmware

The attack very similar to the one described above worked, with unprotected flash and unlocked TSEG being available to the shellcode.

C. dq57tml board, EDK2 development firmware

In this case, the Boot Script and the information about its location were stored securely in SMM Lockbox. However, the Boot Script contained DISPATCH opcodes with target in ACPI NVS[4]. After overwriting the DISPATCH target with shellcode, it ran fine, with TSEG unlocked and flash unprotected.

VI. Miscellaneous

We have notified CERT (CERT VU #976132 was assigned) and the vendors about the issue in August 2014. The BIOS updates are expected to be available around the end of 2014.

The issue was independently dicovered by Intel Advanced Threat Research Team[10].

References

[1] *UEFI specification*, http://www.uefi.org/sites/default/files/resources/2_4_Errata_B.pdf
[2] Intel, *Intel Platform Innovation Framework for EFI S3 Resume Boot Path Specification*, http://www.intel.com/content/dam/doc/reference-guide/efi-s3-resume-boot-path-specification.pdf
[3] Intel, *EFI Boot Script Specification*, http://www.intel.com/content/dam/doc/guide/efi-boot-script-specification-v091.pdf
[4] EFI Dev Kit (EDK), http://tianocore.sourceforge.net/wiki/EDK2
[5] Rafal Wojtczuk, *Subverting the Xen hypervisor*, https://www.blackhat.com/presentations/bh-usa-08/Wojtczuk/BH_US_08_Wojtczuk_Subverting_the_Xen_Hypervisor.pdf
[6] Rafal Wojtczuk, Alexander Tereshkin, *Attacking Intel BIOS*, https://www.blackhat.com/presentations/bh-usa-09/WOJTCZUK/BHUSA09-Wojtczuk-AtkIntelBios-SLIDES.pdf
[7] Rafal Wojtczuk, Joanna Rutkowska, *Attacking Intel Trusted Execution Technology*, https://www.blackhat.com/presentations/bh-dc-09/Wojtczuk_Rutkowska/BlackHat-DC-09-Rutkowska-Attacking-Intel-TXT-slides.pdf
[8] Shawn Embleton, Sherri Sparks, *A New Breed of Rootkit: The System Management Mode (SMM) Rootkit*, https://www.blackhat.com/presentations/bh-usa-08/Embleton_Sparks/BH_US_08_Embleton_Sparks_SMM_Rootkits_WhitePaper.pdf
[9] Anibal L. Sacco, Alfredo A. Ortega, *Persistent BIOS Infection*, http://www.coresecurity.com/files/attachments/ Persistent_BIOS_Infection_CanSecWest09.pdf
[10] *Intel Advanced Threat Research Team*, http://www.intelsecurity.com/advanced-threat-research/

[4]The target of this DISPATCH was in the body of PchS3Support DXE module.