LONGKIT – A Universal Framework for BIOS/UEFI Rootkits in System Management Mode

Julian Rauchberger[1], Robert Luh[2] and Sebastian Schrittwieser[2]
[1]*St. Poelten University of Applied Sciences, St. Poelten, Austria*
[2]*Josef Ressel Center TARGET, St. Poelten University of Applied Sciences, St. Poelten, Austria*
firstname.lastname@fhstp.ac.at

Keywords: Malware, Rootkit, BIOS, UEFI, System Management Mode.

Abstract: The theoretical threat of malware inside the BIOS or UEFI of a computer has been known for almost a decade. It has been demonstrated multiple times that exploiting the System Management Mode (SMM), an operating mode implemented in the x86 architecture and executed with high privileges, is an extremely powerful method for implanting persistent malware on computer systems. However, previous BIOS/UEFI malware concepts described in the literature often focused on proof-of-concept implementations and did not have the goal of demonstrating the full range of threats stemming from SMM malware. In this paper, we present LONGKIT, a novel framework for BIOS/UEFI malware in the SMM. LONGKIT is universal in nature, meaning it is fully written in position-independent assembly and thus also runs on other BIOS/UEFI implementations with minimal modifications. The framework fully supports the 64-bit Intel architecture and is memory-layout aware, enabling targeted interaction with the operating system's kernel. With LONGKIT we are able to demonstrate the full potential of malicious code in the SMM and provide researchers of novel SMM malware detection strategies with an easily adaptable rootkit to help evaluate their methods.

1 INTRODUCTION

Hiding malware such as rootkits or bootkits inside the BIOS/UEFI of a computer has long been deemed a theoretical threat rather than an actual attack surface. Implementation seemed too difficult and the benefits for malicious actors aiming for quick profits were considered negligible. However, with the recent rise of Advanced Persistent Threats (APTs) and state-sponsored attacks, sophisticated targeted attacks are now considered a realistic threat to businesses (Luh et al., 2016). For skilled attackers requiring high stealth and persistence rather than widespread infection, the BIOS/UEFI of a computer provides an ideal target as it allows their payload to act independently of the operating system while still maintaining full control over it. Moreover, in recent years, an increased focus on security in software development can be observed and common attack surfaces such as operating systems and web browsers have become more and more difficult to exploit. Today, a lot more investment is needed to compromise a system and "low-hanging fruits" are slowly disappearing, which will force attackers to find different targets (Forristal, 2011). Lower operational levels such as the firmware of a system are still underrepresented in security research and contain vulnerabilities that might prove more tempting a target in the long term.

The System Management Mode (SMM) is a legacy mode of operation available in x86 and x86-64 CPUs. Originally, SMM was intended to be used for maintenance tasks such as power and thermal management (Duflot et al., 2010). It is a highly privileged mode of operation which has free I/O access, can directly interact with memory and has no hardware memory protections enabled. The operating system itself is suspended during SMM and is therefore unable to enforce any security policies. To emphasize that SMM is even more privileged than hypervisors, it is often referred to as Ring -2 (Domas, 2015; Wojtczuk and Rutkowska, 2009).

Due to its high privileges, the SMM is one of the key areas of many low-level attacks described in the literature (Duflot et al., 2010; Kallenberg and Kovah, 2015; Domas, 2015; Duflot et al., 2006; Embleton and Sparks, 2008; Embleton et al., 2013; Schiffman and Kaplan, 2014). On modern operating systems, it provides a level of access even above the kernel. The main motivation for SMM attacks is therefore the escalation of privileges, often with the goal of installing

Rauchberger, J., Luh, R. and Schrittwieser, S.
LONGKIT – A Universal Framework for BIOS/UEFI Rootkits in System Management Mode.
DOI: 10.5220/0006165603460353
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 346-353
ISBN: 978-989-758-209-7

persistent malware on the system which is independent from the operating system.

Malware running in System Management Mode provides attackers with several more advantages which traditional kernel or userland based malware does not have due to access restrictions. It has extremely high stealth and persistence capabilities: under normal circumstances, System Management RAM (SMRAM) cannot be read or written to from outside the SMM, not even with Ring 0 privileges (Kallenberg and Kovah, 2015). Thus, SMM malware, which does not alter the operating system, is very hard to detect, requiring manual dumping and reverse engineering of the System Management Interrupt (SMI) handler.

Contributions. The main contributions of this paper are:

- We introduce a flexible framework for BIOS/UEFI rootkits in the SMM.
- We show how SMM rootkits can access the entire 64-bit address space of the virtual memory by entering the Long Mode.
- We explain LONGKIT's ability of interfering with the operating system's kernel by locating and parsing the page table used by the OS.
- We present a prototype of the LONGKIT framework as well as the implementation and evaluation of two typical rootkit functionalities (login bypass and system call hooking).

2 BACKGROUND

SMM can only be entered when the CPU receives a System Management Interrupt (SMI) which is a hardware interrupt. However, triggering an SMI in software is also possible, for instance by writing to the Advanced Power Management Control Register (APMC) (Duflot et al., 2010). Upon receiving an SMI, the CPU will enter SMM and execute the System Management Interrupt Handler which is located in System Management RAM (SMRAM) (Duflot et al., 2010). SMRAM is a special region in memory which, if correctly configured, can only be read and written to when the CPU is in System Management Mode (Duflot et al., 2006). A part of SMRAM is reserved for the state save area where the contents of most CPU registers are stored when entering SMM. When the SMI handler has finished, it executes the `RSM` assembly instruction which restores the registers with the values in the state save area and returns control to the operating system.

A recurring question regarding SMM malware revolves around the level of control the SMM has over the system and what malicious actions it can execute (Kallenberg et al., 2014; Kallenberg and Kovah, 2015). Since SMM has direct access to physical RAM, malware running inside its boundaries can essentially do everything lower privileged malware can. The only difference is that said functionality might be more difficult to implement because there is no automatic translation between virtual and physical addresses (Duflot et al., 2006) and it is not possible to directly call APIs of the operating system. Attackers might have to reimplement certain functionality which is otherwise easily accessible.

By demonstrating the very real threat posed by novel solutions such as LONGKIT, we hope to shift attention to the SMM attack surface and inspire future research in this area.

2.1 Related Work

In the past, the SMM was shown to be exploitable by attackers to bypass certain security features (Duflot et al., 2010). The authors identified four fundamental flaws in the design of the SMM which they attribute to the fact that security has mostly been an afterthought in the specification. Moreover, they also identified two common design flaws in SMI handlers. These are not flaws in the specification but rather commonly made programming mistakes which subvert security. Similar observations have been made by (Kallenberg and Kovah, 2015).

Boot script vulnerabilities are a class of attacks that apply only to UEFI systems as they rely on interpretation of the S3 boot script data structure which is a feature of UEFI firmware. S3 resume allows for a faster startup during a suspend/resume cycle by skipping certain parts of a normal boot sequence and executing the S3 boot script to restore configuration. If this script is stored insecurely, attackers with Ring 0 privileges can modify it and execute arbitrary code during early boot (Wojtczuk and Kallenberg, 2014).

Additionally to the classes of attack vectors specified above, other, less generic exploits have been published in the past. *Speed Racer* (Kallenberg and Wojtczuk, 2015) is a race condition found on multi-core systems with chipsets lacking or making no use of the SMM_BWP (SMM BIOS Write Protect Disable) register. When this register is set, the BIOS region is only writable if all processors are in SMM. If this feature is missing or unused, a race condition exists which allows reflashing of firmware by continuously attempting to unlock the BIOS region on one core and writing to it on another core. In some cases, the firmware can-

not re-lock memory fast enough and the write will go through. Another attack, *The Memory Sinkhole* (Domas, 2015), abuses a legacy feature of modern processors that allows remapping of the Advanced Programmable Interrupt Controller (APIC) registers to a chosen address in memory. If this remapping is made to overlap SMRAM, reads and writes are redirected to the APIC, allowing an attacker to "sinkhole" a small range of memory in SMRAM. By attacking UEFI template code provided by Intel and used in most modern UEFI implementations, the SMI Handler can be forced to use a crafted Global Descriptor Table (GDT) and jump outside of SMRAM to allow hijacking of execution flow. A memory caching issue that can be exploited to take control of SMM has been independently discovered by (Duflot et al., 2009) as well as (Wojtczuk and Rutkowska, 2009). It is effectively a cache poisoning attack that is being conducted by marking the SMRAM region as cacheable, writing the code to be executed to the physical address which will then be cached and finally triggering an SMI which will now execute the SMI handler from cache rather than the real one. Furthermore, firmware older than 2006 seemed to commonly have the problem of not making use of security features such as write protection of certain memory regions or registers, making exploitation trivial (Wojtczuk and Rutkowska, 2009).

Practical Examples. To demonstrate the capabilities of SMM malware, proof-of-concept implementations have been created by several researchers. *LightEater* (Kallenberg and Kovah, 2015) persistently infects SPI flash memory to show how SMM-based malware can subvert the security of live operating systems (e.g. Tails) booted from external media that normally leave no trace on the hard disk drive of the system. After initial infection during the use of Microsoft Windows, LightEater is able to fingerprint the currently running OS and only execute its malicious payload upon detection of Tails. It has been specifically developed to scan the RAM for GPG keys, passphrases and decrypted emails of the Claws email agent.

(Domas, 2015) demonstrated the use of an SMM rootkit based on research by Dmytro Oleksiuk Userland processes signal the rootkit that they wish to be elevated by writing a magic number to a specific register. During the next System Management Interrupt, the rootkit detects the magic number and grants the process root privileges. This is possible only because code running in SMM is free to modify RAM as it sees fit, including data structures held by the kernel.

The Watcher is a proof-of-concept rootkit whose only capability is to scan memory for a certain signature. If this signature is found, data following thereafter will be executed as code by the rootkit (Kallenberg et al., 2014). The idea behind *The Watcher* is simple yet versatile and powerful. Since it directly scans physical memory for a signature, all an attacker has to do is place the signature and the code they want to execute anywhere in RAM. This could be done by embedding it in a document or simply by sending a network packet containing the payload to an arbitrary port. As long as the data will be stored in memory for some time, *The Watcher* will find and execute it. A demo of this idea has been implemented and is available online[1].

Classified documents leaked by Edward Snowden show that the NSA has had the capability to infect BIOSs since at least 2008 (Appelbaum et al., 2013). These "implants" provide advanced stealth and persistence capabilities and have been specifically created for a wide range of platforms. The NSA shows a clear preference to attack BIOSs rather than the operating system itself, indicating that such malware works very well and has low detection rates.

3 SMM ROOTKIT CONCEPT

In this section we introduce the LONGKIT framework and discuss how our framework approach differs from existing solutions. Additional information about system architecture and the low-level CPU interaction can be found in Intel's Software Developer Manual (Intel, 2016).

LONGKIT is a fully functional exemplary implementation of a BIOS-based rootkit making use of System Management Mode (SMM) for advanced stealth and persistence. It does not alter the contents of the hard drive in any way and is stored solely in BIOS flash memory. LONGKIT takes control of the operating system by installing a malicious System Management Interrupt (SMI) handler at boot time. By overwriting the handler for interrupt 0x01, it can intercept debug exceptions that are generated when a hardware breakpoint is encountered. To hide the modifications made to the interrupt handler in RAM, a hardware breakpoint is configured to watch reads and writes of the modified memory area. If the operating system tries to access that area, LONGKIT will be alerted and can take countermeasures to avoid detection. In order to hide modifications made to the debugging registers, a special feature of the x86 architecture is being used. Setting the General Detect Enable (GD) flag in

[1] https://github.com/scumjr/the-sea-watcher (last accessed 8.8.2016)

the DR7 register activates debug register protection, which causes any `MOV` instruction that accesses a debug register to also generate a debug exception. This allows LONGKIT to hide its presence when the system tries to read or write a debug register.

In order to allow for greater compatibility with other BIOSs which might configure SMRAM differently or require to append the malicious SMI handler after the original, all code has been written position-independent, ensuring it can run from anywhere in memory.

3.1 SMRAM Infection

SeaBIOS[2] is an open source implementation of a x86 BIOS and the default firmware for QEMU and KVM but can also run natively on hardware with the use of coreboot[3]. It behaves like most other commercial BIOSs and uses certain platform security features to lock SMRAM during boot, making it impossible to read or write to it when not executing in SMM. To install LONGKIT, a modified version of SeaBIOS is being used in our example implementation (see section 4 for additional information). SeaBIOS copies the malicious SMI handler to SMRAM prior to locking it. The original SeaBIOS SMI handler is being copied after LONGKIT and will be invoked by it as necessary in order to preserve original functionality. Besides installing LONGKIT at boot time by means of modifying the BIOS/UEFI of the motherboard, it would also be possible to install it at runtime. This requires the use of a firmware-specific exploit (see Section 2.1).

3.2 Rootkit: Bootstrapping

When SMM is entered and LONGKIT starts to execute, it first needs to locate itself in memory in order to allow for relative addressing of its data structures. To do this, a trick commonly used in position-independent shellcode is being employed. After configuring a temporary stack, a relative `CALL` will push the address of the next instruction on the stack which can then be retrieved with a `POP`.

The instruction pointer in SMM as pushed by `CALL` is not the physical address but rather a virtual one, relative to the code segment (CS) register. To ensure addressing behaves consistently and does not become more complicated than necessary, all of these segments are configured the same as CS.

After bootstrapping has been completed, LONGKIT checks if the SMI was actually generated to invoke it or if the original SMI handler should be executed. Since the hijacked debug exception handler will always invoke LONGKIT by writing a magic number to the APMC I/O port, it checks port 0xB2 for this predefined constant. If it is found, LONGKIT will enter Long Mode and further execute its functionality. If not, it will transfer code execution to the start of the original SeaBIOS SMI handler.

3.3 Rootkit: Entering Long Mode

After the initial execution environment has been configured, LONGKIT switches the CPU to Long Mode in order to have full access to 64 bit features. Entering Long Mode allows addressing of more than 4GB of RAM, accessing the full 64 bit of certain registers and making use of instruction pointer relative addressing, which makes writing position-independent code easier. Although undocumented by Intel, it is possible to directly switch from real mode to Long Mode, completely skipping protected mode in the process. We make use of this technique to avoid unnecessary complexity.

In Long Mode, paging is mandatory and cannot be deactivated. Therefore, the first step is to set up a valid page table structure inside SMRAM. To make paging as simple as possible, all memory is identity-mapped, which means virtual and physical addresses will always be exactly the same. For our demonstration, we only map the first gigabyte as this is enough to demonstrate the concept. We set up a PML4 table in memory with a single entry that points to a PDP table, which again has only a single entry that points to a PD table. This last table references 512 2MB pages that identity-map the first lower GB of RAM. For further details on the layout of these tables, see Section 3.5.

After page tables have been set up and loaded in CR3, LONGKIT sets some flags to prepare for the transition to Long Mode. First, the Physical Address Extension (PAE) bit has to be set in Control Register 4 (CR4). Next, we set the Long Mode Enable (LME) bit which is bit 8 in the Extended Feature Enable Register (EFER). Finally, bit 0 (Protected Mode Enable) and 31 (Paging) are set in Control Register 0 (CR0). After this, execution is now in 32 bit compatibility mode. In order to execute real 64 bit code, a global descriptor table has to be configured and the CS updated to the correct segment.

In order to set up a minimal GDT for LONGKIT, three segment descriptors are required. Each descriptor is 8 byte in size and should be aligned on an 8 byte boundary. The first one is a null descriptor, which by definition should always be present and has every bit

[2]https://www.seabios.org/SeaBIOS (last accessed 8.8.2016)

[3]https://www.coreboot.org/ (last accessed 8.8.2016)

set to 0. The second one is a 64 bit code segment descriptor that will be used for the CS register, and the third is a data segment descriptor, which we will use for all other segment selector registers.

Since the GDT is no longer used for segmentation in Long Mode, all base and limit fields can be set to 0.

Now that the GDT has been defined, it can be loaded with the LGDT assembly instruction. LGDT has to be passed the address of a special structure which contains information about the actual GDT. For our rootkit it is important to note that the GDT address in the pointer structure has to be the real, physical address and not the address relative to the current CS.

After the GDT has been correctly set up, the segment registers have to be updated. As the CS register cannot be directly written to, it has to be changed by performing a far jump into the code segment. Again, the address jumped to has to be the actual, physical address and cannot be relative to the CS like the jumps in real mode were. The correct segment selector for the code segment is 8, as it is the second descriptor. When the jump has been performed, the CPU is in Long Mode and all additional features are available. The last thing to do now is to update the rest of the segment registers with the offset of the data segment defined in the GDT and to re-setup the stack since the stack pointer is no longer relative to the SS register.

3.4 Invoking SMM from Debug Exceptions

The next step in taking control of the operating system is to find a way to reroute debug exceptions into system management mode so LONGKIT can handle them. We decided to overwrite the operating system interrupt handler for interrupt 0x01 with a new handler that generates an SMI. Since this requires modification of RAM the OS has access to, we also needed to find a way to hide the changes. By setting a hardware breakpoint on read and write access of the modified memory location, LONGKIT can undo all changes before the OS can process them. This also raises an additional constraint for our exception handler: since a hardware breakpoint can only monitor up to 8 bytes of memory at a time, the handler cannot be bigger than that or we would have to use multiple breakpoints. Listing 1 shows the handler we decided to use.

Listing 1: Debug Exception Handler

```
push rax
mov al, MAGIC_NUMBER
out SMI_PORT, al
pop rax
iretq
```

First, RAX is pushed on the stack as we need to modify and later restore its contents. When an interrupt handler is invoked in Long Mode, only the SS, RSP, RFLAGS, CS and RIP registers are saved. If the handler modifies additional registers, it has to take care of restoring their contents afterwards or the interrupted code might behave unexpectedly.

By writing the MAGIC_NUMBER constant to the SMI_PORT, LONGKIT is subsequently invoked. After that, RAX is restored and IRET is used to return from the interrupt handler. Restoring of RAX could also be done from inside SMM, in case any need to further shorten the handler should arise.

To overwrite the operating system handler, it needs to be located first. The address of the interrupt handler can be found in the interrupt descriptor table (IDT). The location of the IDT itself is loaded in the interrupt descriptor table register (IDTR). As the content of the IDTR is stored in the state save area upon entering SMM, the most consistent way of getting it is by loading it from there. The offset is 0xFE88 from SMBASE. The IDTR however does not contain the physical address of the IDT but rather the virtual address used by the operating system. In order to get the physical location it is therefore necessary to parse the page table used by the operating system and manually translate the address.

3.5 Translating a Virtual to a Physical Address

Depending on whether 2MB or 4KB pages are used, there are either 3 or 4 levels of page tables to parse. 1GB page tables are not supported by LONGKIT as they are not used by the operating system we tested.

The first step in translating a virtual to a physical address is to find the Page Map Level 4 (PML4) table. This can be achieved by loading the content of the CR3 register from the state save area, the offset from SMBASE is 0xFF50.

The virtual address will then be split into multiple parts which are used as indices into the page table directories.

3.6 Hijacking the Debug Exception Handler

The next step for LONGKIT is to take control of the operating system's debug exception handler.

After the physical address of the interrupt descriptor table has been located, its contents can be parsed to find the debug exception handler in memory. The debug exception handler has interrupt number 0x01

and therefore starts at offset 16. We are only interested in the location of the handler, which is stored in multiple parts in the "Offset" fields.

After reading the virtual address of the handler, it has to be translated to a physical address. For this, the same code that was used to translate the IDT location can be used. Now that the actual location of the code in memory is known, it can simply be overwritten with our malicious handler that invokes the SMM rootkit.

In order to be able to restore it later on, LONGKIT first copies the legitimate exception handler to SMRAM before overwriting it.

3.7 Debug Registers

Now that all hardware breakpoints will cause an SMI to be generated and LONGKIT to be run, the debug registers have to be configured. In total, there are four usable debug registers on x86-64 CPUs: DR0 to DR3 which contain the linear addresses on which execution should break, DR6 which contains information about what condition triggered the debug exception, and DR7, which configures the exact behavior of the breakpoints.

DR0 to DR3 are not stored in the SMRAM state save area upon entering SMM and can therefore be modified directly. DR6 and DR7 are stored there and need to be modified in the state save area as their contents will be restored upon exiting.

DR7 is called the debug control register as it can be used to specify on which conditions the addresses in DR0 to DR3 should generate a debug exception.

Bit 13 of DR7 is called the general detect enable (GD) flag and if set, enables debug register protection which will generate a debug exception prior to any `MOV` instructions that reads or writes a debug register. The flag is used by this rootkit for stealth purposes in order to detect when the operating system is trying to make use of the debug registers.

DR6, the debug status register, reports the reason the last debug exception has been triggered. The first four bits indicate which of the four breakpoints have been triggered. They may or may not be set if the corresponding breakpoint has not been enabled, so debug handlers should only check the bits of the enabled breakpoints.

Additionally, we have to take into account that debug exceptions are generated for execution breakpoints before the monitored instruction is being executed. This means that upon returning from SMM, the instruction would immediately generate a new debug exception, causing an infinite loop. To fix this, the resume flag (RF) has to be set in the RFLAGS register at the time the instruction is executed. The RFLAGS register is stored on the interrupt handler stack and will be restored from there when `IRET` is executed.

3.8 Debug Register Setup

LONGKIT will use DR0 to create an 8 byte wide read-/write breakpoint over the modified debug exception handler. If this breakpoint is ever activated, the OS tried to read or write to the modified area in RAM. In this case, LONGKIT will attempt to hide its presence from the system. The other three debug breakpoint registers DR1 to DR3 can be used to implement traditional rootkit functionality such as backdoors or monitoring and modification of OS functionality.

3.9 Debug Register Stealth Features

The contents of the DR6 register at the time of a Debug Exception can also be used to detect debugging and rootkit detection attempts performed by the operating system. Bit 14 (BS) is set if the exception has been caused by single stepping, strongly indicating the use of a debugger. The 13th bit (BD) signifies that the cause of the exception was a `MOV` instruction which accessed a debug register.

If either of those bits has been set or if read/write access to the patched debug exception handler has been detected, LONGKIT removes itself from the operating system by restoring the old debug exception handler and clearing the debug registers.

The patched handler pushed `RAX` on the stack before invoking SMM. Therefore, these changes also have to be undone before returning execution.

In order to achieve a high level of stealthiness, an additional problem has to be solved. Although we created a breakpoint that monitors access of the modified debug handler, exceptions will only be generated after the corresponding instruction has been executed. That means even if we immediately restore the old handler, the first read will go through, leaking up to 8 bytes. However, since the OS has had no chance to process the contents of this read yet, the contents of the register used to read the handler can still be changed from within SMM by patching it in the state save area. Our example implementation searches all general purpose registers for the 8 bytes used for our patched debug exception handler and replaces them with the original bytes if found. This implementation has been tested and found to work well when `memcpy` is being used to read the contents of the handler.

4 SECURITY EVALUATION

To allow for easier testing, a prototype of LONGKIT was developed using the QEMU emulator with KVM on Linux. The virtual machine was booted with a modified version of SeaBIOS which copies the malicious SMI handler into SMRAM before it is being locked. For evaluation purposes, the described methods have been tested using QEMU 2.6.0-rc3 on an Ubuntu host with Linux kernel 4.4 and KVM enabled. To demonstrate how our code facilitates the implementation of traditional rootkit functionality in SMM, we added two commonly found features: monitoring of a system call and a backdoor to the root user account. In our example implementation, DR1 is used to create an 8 byte wide read/write breakpoint on the entry for `execve` in the Linux syscall table. The breakpoint is used to intercept all invocations of the `execve` syscall and demonstrates how LONGKIT can be used to directly monitor what processes are executed on the OS.

The location of the syscall table is hardcoded for Ubuntu Server 14.04 LTS and would need to be updated for other distributions. Alternatively, the syscall table could also be dynamically located in memory.

DR2 is configured as an execution breakpoint on the `commit_creds` kernel function, again hardcoded for Ubuntu Server 14.04 LTS. This function is invoked whenever the Linux kernel sets new credentials, for instance when executing a process as a different user. LONGKIT intercepts these calls and will, upon detection of a certain UID, replace the committed user credentials with root credentials. This effectively allows the backdoor user to be treated by the system as if it were root. The breakpoint shows that LONGKIT is not only capable of monitoring the operating system but can also directly influence it, e.g. by changing function call arguments like in this example.

To evaluate the stealthiness of our approach, we wrote a kernel module which tries to access the debug registers and found that this will be reliably detected by LONGKIT in all tested scenarios. We also found that LONGKIT immediately detects the use of GDB (GNU Project Debugger) on the compromised system and hide its presence by erasing its hooks from memory. If debuggers are being used to create hardware breakpoints, LONGKIT will transparently detect their use and hide itself at the time the breakpoints are created, allowing the debugger to execute normally. We simulated a tool which tries to specifically detect LONGKIT by reading the interrupt 0x01 handler and comparing it to a known good value. When using `memcpy` (which processes 8 bytes at a time) to read the full handler, LONGKIT successfully detects the tool and patches the register used, making it completely invisible even for the OS kernel.

4.1 Countermeasures

Due to the high privileges of System Management Mode, devising a general purpose detection method for arbitrary SMM malware is a complex problem. In order to defend against SMM malware with currently existing technologies, the best way is to avoid infection in the first place or reliably detect tampering with the BIOS/UEFI firmware. Computer systems can be either infected at runtime by the use of one or more SMM exploits or through manual flashing of the firmware which requires access to the hardware. To avoid runtime infection, it is advised to keep the BIOS/UEFI firmware fully updated at all times. Technologies such as Intel's Chipsec[4] can be used to evaluate the security of firmware. Additionally, research suggests that firmware older than 2006 often makes no use of even basic security features and can be easily exploited (Wojtczuk and Rutkowska, 2009). If such firmware is being used, it should be very carefully examined to make sure none of the known issues affect it. Even if all these measures are taken, advanced attackers might still be able to compromise SMM by using previously unknown attack vectors. Existing attacks often need kernel privileges to be carried out successfully, which makes it likely that newly discovered vulnerabilities have the same limitation. Hardening the operating system so malicious actors cannot easily run their code in Ring 0 thus helps to protect against SMM malware.

If the adversary has access to the hardware and can install arbitrary firmware, defense should focus on detecting such changes by checking the integrity of BIOS/UEFI flash memory. One way would be the use of a Trusted Platform Module (TPM) to detect changes made to the firmware. The problem with this approach, however, is that it uses the BIOS/UEFI to check its own integrity, which is impossible to do securely in the presence of BIOS/UEFI-based malware. A proof-of-concept implementation exists that is able to infect the firmware and bypass TPM-based integrity checks (Butterworth et al., 2013). A better option is the use of Intel "Boot Guard" technology which moves the root of trust from the firmware to the CPU. If Boot Guard is supported, the Intel Authenticated Code Module (ACM) can be launched to measure the first code in BIOS/UEFI and put the result into the TPM. The ACM code itself is signed by Intel and the signature is checked directly on the

[4]https://github.com/chipsec/chipsec (last accessed 8.8.2016)

CPU, making this a viable option to detect unauthorized modifications (Intel, 2013). Another way is the use of the Intel Trusted Execution Technology (TXT) which can be used with Intel's open source bootloader "tboot". It should be noted that in order for TXT to be secure, the use of Dual Monitor Mod (DMM) is required because otherwise attackers might be able to avoid detection by hiding in SMRAM (Kallenberg and Kovah, 2015).

5 CONCLUSION

In this paper, we introduced LONGKIT, a novel framework for BIOS/UEFI malware in the SMM which is written entirely position-independent, fully supports 64-bit computers and is memory-layout aware for targeted interaction with the operating systems kernel. With the LONGKIT prototype we were able to show the full potential of concealed malware in the SMM, where most previous approaches were limited to simple proof-of-concept scenarios. By implementing two typical rootkit functionalities (authentication bypass and system call hooking), we demonstrated the effectiveness of the LONGKIT framework in real-world applications. Furthermore, we discussed the stealthiness of LONGKIT and possible ways of identifying future SMM-based malware.

Based on our comprehensive research and the implementation details provided, further investigation into practical countermeasures of BIOS/UEFI malware becomes possible. In order to encourage future research into malware and APT defense, all code will be made available for researchers upon request. Besides malware, LONGKIT provides many features which are likely also useful for other SMM research. The whole code being position independent makes it very versatile and easily reusable in many different scenarios. For example, LONGKIT could serve as a foundation for the development of an open source alternatives to the currently often closed source SMI handlers provided by many manufacturers. At the time of writing, the SMI handler used by SeaBIOS is very minimal and LONGKIT could be used to develop more sophisticated functionality.

ACKNOWLEDGEMENTS

The financial support by the Austrian Federal Ministry of Science, Research and Economy and the National Foundation for Research, Technology and Development is gratefully acknowledged.

REFERENCES

Appelbaum, J., Horchert, J., and Stöcker, C. (2013). Shopping for spy gear: Catalog advertises nsa toolbox. (last access: 9.8.2016).

Butterworth, J., Kallenberg, C., Kovah, X., and Herzog, A. (2013). Bios chronomancy: Fixing the static core root of trust for measurement. *ACM Conference on Computer and Communications Security, Berlin, Germany*.

Domas, C. (2015). The memory sinkhole - unleashing an x86 design flaw allowing universal privilege escalation. *BlackHat, Las Vegas, USA*.

Duflot, L., Etiemble, D., and Grumelard, O. (2006). Using cpu system management mode to circumvent operating system security functions. *CanSecWest, Vancouver, Canada*.

Duflot, L., Levillain, O., Morin, B., and Grumelard, O. (2009). Getting into the smram: Smm reloaded. *CanSecWest, Vancouver, Canada*.

Duflot, L., Levillain, O., Morin, B., and Grumelard, O. (2010). System management mode design and security issues. *IT-DEFENSE, Brühl, Germany*.

Embleton, S. and Sparks, S. (2008). Smm rootkits. *SecureComm, Istanbul, Turkey*.

Embleton, S., Sparks, S., and Zou, C. C. (2013). Smm rootkit: a new breed of os independent malware. *Security and Communication Networks*.

Forristal, J. (2011). Hardware involved software attacks. *CanSecWest, Vancouver, Canada*.

Intel (2013). Hardware-based security for intelligent retail devices. (last access: 9.8.2016).

Intel (2016). Intel 64 and ia-32 architectures software developers manual.

Kallenberg, C. and Kovah, X. (2015). How many million bioses would you like to infect. *CanSecWest, Vancouver, Canada*.

Kallenberg, C., Kovah, X., Butterworth, J., and Cornwell, S. (2014). Extreme privilege escalation on windows 8/uefi systems. *BlackHat, Las Vegas, USA*.

Kallenberg, C. and Wojtczuk, R. (2015). Speed racer: Exploiting an intel flash protection race condition. *Bromium Labs*.

Luh, R., Marschalek, S., Kaiser, M., Janicke, H., and Schrittwieser, S. (2016). Semantics-aware detection of targeted attacks: a survey. *Journal of Computer Virology and Hacking Techniques*.

Schiffman, J. and Kaplan, D. (2014). The smm rootkit revisited: Fun with usb. *Availability, Reliability and Security (ARES), Fribourg, Switzerland*.

Wojtczuk, R. and Kallenberg, C. (2014). Attacking uefi boot script. *31st Chaos Communication Congress, Hamburg, Germany*.

Wojtczuk, R. and Rutkowska, J. (2009). Attacking smm memory via intel cpu cache poisoning. *Invisible Things Lab*.