A FRAMEWORK FOR EMBEDDED MALWARE CONSIDERATIONS: EMBEDDED SYSTEMS VULNERABILITIES TO INJECTION OF MALWARE IN A FIRMWARE UNDER ATTACK

HAIDAR ALSALEH, SUFIAN YOUSEF and OMAR ARABEYYAT
ANGLIA RUSKIN UNIVERSITY, UK

ABSTRACT

With the emergence and proliferation of new methods for implanting malware, the next frontiers in system vulnerabilities are the embedded viruses or system attacks through vulnerabilities below the application layer. This paper researches contemporary and potential attacks that implant malware functions underneath the operating system (OS) within the firmware. At the lower OSI layers, the Malware take control and command before starting the operating system. The common wisdom is that if the underlying firmware cannot be trusted, then the OS and the applications depended on the firmware also cannot be trusted.

This paper presents a comprehensive research on malwares in the BIOS and firmware and discusses and analyse them with specific interest towards the SMM. Furthermore this paper provides a novel and general solution approach to common firmware and for a secure processing framework.

This paper showed in a novel approach that if the firmware cannot be trusted then the OS and the software that are dependent on the underlying firmware might not be trusted.

Index Terms – Embedded Security, SMM, BIOS, Trusted Hardware, Root Kit.

1. Introduction

The most lethal form of embedded malware comes in the form of a rootkit attack. The first widely known instance of wide spread exploitation was in 2005. Within that incident the rootkit was devoid of known malicious intention when Sony Corporation of Japan implanted this modus operandi so as to hide its Intellectual Protection (IP) copy protection software (Michael 2010). The implementations involved the inclusion of eXtended Copy Protection (XCP) and MediaMax CD-3 software on Sony brand music CDs. This software silently and automatically installed on the Windows OS hosting the playing CDs, causing alteration of the operating system. To obfuscate itself, the Sony rootkit used and hid files, registry keys and processes that starts with the string name of sys. This obfuscation opened the door for malicious exploitation by others.

It is acknowledged that the less common systems such as Apple and Linux do face fewer attacks, as compared to Microsoft products. Also custom made and special applications as well as embedded processors are rarely attacked unless the attack is targeted (Chen 2010). This is not an indication of the sturdiness of the Apple or Linux design but is the result of the wide spread use of the MS System, making the later more readily available for an exploitation. Furthermore, unless the attack is targeted, attackers seek the most “Bang for the Buck” effect, which would require system types with wider platform proliferations such as usage across differing intellectual, social and economic system and applications.

There are other papers that researched the topic but were hardware specific. While other papers targeted the firmware but fell short of comprehensive analysis and lack effective recommendations for a secure platform.

Malware detection is based on Cohen theorem. Cohen (1986) built on the work of Goedel's, where Cohen implemented the "proof by contradiction" technique to show that no perfect virus checker can ever exist. However Cohen theorem is based on the anti-malware tools that are hosted on the operating system which are inert to malwares in the firmware, (this research uniquely present to show that it is highly effective method of an attack). Seshadri et al (2004) published their first work on Software-based attestation for embedded devices. Seshadri (2005) followed it by verifying integrity on legacy platforms. Their two papers presented a note worthy issues, however these papers and others failed to cover the SMM and the Hypervisor Virtual Machine VM malware concerns.

The VM vulnerabilities were discussed by King et al (2006) were the team showed that malware attack on the VM based Rootkit (VMBR) was possible. Building on previous work, Duflot et al (Duflot) worked on the privilege escalation issue by manipulating the SMM code. Duflot results were the bases for Embleton et al (2008) who to some extent discussed the SMM (System Management Mode) rootkit as an entry into the firmware.

This is a problematic topic to disquisition. Castelluccia et al (2009a) presented their work on the difficulty of software-based attestation for embedded devices asserting the difficulties associated with their research and presented their recommendation only to see in less than a year be refuted by Perrig (2010) of CyLab and Doorn of AMD. However it did not take long for Perrig paper itself to be re-refuted by Francillon (2010), and continue to confirm the difficulty of the subject by Castelluccia (2009b).

Early research by King (2006) showed that malware attack using methods implemented in the Virtual Machine based Rootkit (VMBR) was possible. However their research was based on simulation and their findings were narrow in its applications and limited to specific hardware. Loic Duflot (Duflot) followed King in showing that privileged mode escalation by manipulating the SMM code was possible and can result in malware code injection to switch the system operation mode to a higher user privilege level.

Emleton (2008) analysed the SMM threat as it relates to the invocation of SMM in a rootkit however their work was not fully published.

This paper introduction is shown in section 1, the SMM Threats are presented in section 2, the SMM Mode Operations are explained in section 3, the Embedded Rootkit Analyses is shown in section 4. The full analysis of Firmware attacks is supplied in section 5 and the Firmware vulnerabilities in the BIOS32 are discussed in section 6. The simulation of the attack on the Firmware is given in section 7 and the paper is concluded in section 8.

2. SMM (System Management Mode) Threats

Most common computing platforms (including PDAs and IPhones, etc) share similar logical booting sequence. This booting sequence involves three distinctive stages with similar sequential order starting with System firmware execution; then pointing and starting the Operating system loader and finally loading the operating system (OS) from a memory location. Although this paper discusses the BIOS in the PC, it is equally viable to all other computing platforms including consumer goods.

The system firmware is the lowest common denominator and is the closest to the metal (hardware); this is sometimes referred to as bare-metal coding. In the PC world the system firmware would be hosted in the BIOS. In non-PC devices this would also be the BIOS, however the BIOS may not be a separate but Embedded Software hosted on a ROM or RAM hardware device, and is co-hosted with the system application software.

At stage one the system start with the Pre POST routine. This routine commences by initialising the hardware relevant to the system operations including the hardware initialisation of the hosting CPU then the RAM. After the RAM is initialised and the system firmware is copied, the Pre POST routine allocate and initialise tables allocated in the main system memory creating a 640k RAM segment, starting form space pointer 00 000 to A0 000. After the completion of Stage one, the System firmware goes into protected mode and passes control to Stage two.

To optimise memory each of the BIOS modules (except Bootblock; BootCode, and the Decompression routine; DECOMPCODE) are compressed using LZH compression methods and containing an 8-bit checksum. The decompression routine decompresses modules prior to execution and places the uncompressed mode into CPU RAM.

The SIMM code (in the firmware or BIOS) includes the instructions for connecting peripheral components and peripheral emulation to be available to the operating system upon physical interconnection.

The System Management Mode (SMM) is invoked by the execution of the SMM code (SMI handler) this is done during high-privilege mode. This mode is part of the firmware boot up cycle, and is invoked during the Power On Self Test (POST) stage in BIOS.

There are two main functional areas for firmware codes; the first is to initiate the boot up process, while the second is used for run time mode. The later is where the booting code copies the SMI handlers (interrupt code part of the SMM) for use during run time and in parallel with full OS operation.

3. SMM Mode Operations

There are two methods for entering the SMM normal-mode operations:

- SMI interrupt, Hardware assertion (SMI pin), non-maskable interrupts highest priority in the system.
- SMINT instruction; Software assertion entry through CPU instruction code (0F 38).

SMM is totally transparent to all application software on the host, including the protected-mode operating system.

The most common method for invoking the SMM mode is by the execution of the SMM code (SMI handler) in high-privilege mode during the firmware boot up cycle, as in the Power on Self Test (POST) stage in BIOS. Once the SMM is invoked it would be locked and placed into protected mode. To penetrate the SMM mode while it is protected, the malware must slither into the SMM while it is in open state or while the SMM mode is in the state of a running normal-mode operation which can be achieved either through SMI interrupt or through SMINT command instruction:

- SMI interrupt; Hardware physical assertion of the allocated pin in the hardware casing. This is intended to invoke the interrupt mode. This would lead to a non-maskable interrupts, which is the highest priority in the system. The non-maskable interrupt is a hardware event of high priority that must get the immediate attention of the CPU. In contrast maskable interrupt, would cause the CPU to continue computing instructions processing until it has time for a safe point for the CPU to take care of this interrupt.
- SMINT instruction; Software assertion that require code instructions invoked upon the CPU to trigger entry to the SMM through CPU instruction code (0F 38). Part of the SMM module within the firmware or the BIOS is the SIMM code which is a firmware code that includes the instructions for connecting peripheral components and serve as peripheral emulation needed to be available for the operating system while it is independent of the OS or the hosting system.

The SMM for the Intel IA-32 model design supports three operating modes and one quasi-operating mode. These four modes are listed below:

- Virtual-8086 mode — if the processor placed into protected mode, then it would support a quasi-operational mode known as virtual-8086. This mode is for executing 8086 software in a protected, multitasking environment.
- Real-address mode — In this mode the SMM operate in real time mode providing real time code execution services for software code capable of running on an Intel 8086 processor. While in this mode the SMM has the capability to switch to other modes. Protected mode — (32 or 64 bit operating system) Native operating mode with a set of architectural features and backward compatibility for the existing legacy software base.
- System management mode (SMM) — SMM first launched with the Intel386 SL processor however it became standard architectural feature for all subsequent IA-32 processors. The main function is to provide transparent mechanism for implementing power management and OEM compatibilities functionality.

The SMM operation is a separate operating mode of the CPU, with distinct hardware and software dependency intended for use only by system firmware and not by application software or general-purpose system software.. The operating system is not aware of the SMM or the SMI mode. One exception to the SMI is that it can be invoked by the system. When the system invoke this mode, the CPU would function separately from Real, Virtual, or Protected modes, however it allows system designers to add components that operate transparently to the operating system and software applications (Advanced Micro Devices Inc, 1997).

During the SMI mode, all normal instruction; component initialisation; and operating system execution are suspended by the CPU however the CPU return system hardware and software control to the operating system upon completing the SMM mode, see Diagram 1, the Flow chart of SMM Routine.

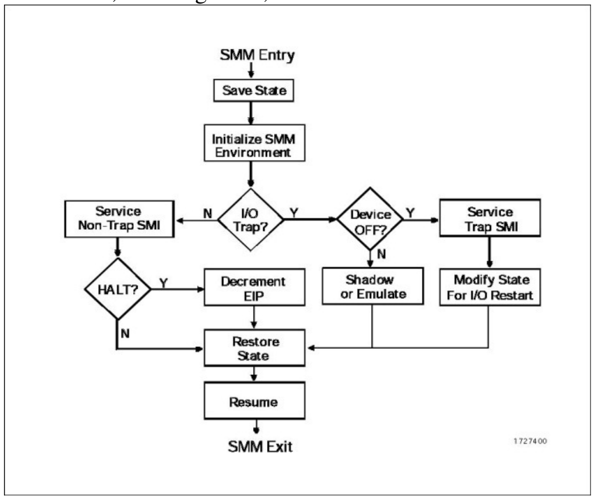


Diagram 1 Flow chart of SMM Routine
(Cyrix inc, 1998, P 6)

An example of an interrupt code implementations that halt processing and store registers is listed below, see Code 1 Auto Halt Restart Implementation Pseudo-Code:

```
Begin
; The Auto Halt Restart slot at register-offset 7F02h in SMRAM indicates
; To the SMM handler that the SMI interrupted the CPU during a HALT
; State Bit position 0 of 7F02h will be set to a one in this condition
{
if EFLAGS.21 is write able then ; should be done during ID process
        {
                if HLT instruction needs to be restarted then
        {
        if SMI during halt state then        ;bit 0 of offset 7F02h = 1
        set HLT restart slot to 00FFh        ;offset 7F00h in state save map
        }
        }
else
        SMM features are not supported
}
end
```

Code 1 Auto Halt Restart Implementation Pseudo-Code
(Advanced Micro Devices Inc, 1995, P 22)

Once the SMM code is initiated the system transfer control to the System Management Interrupt (SMI) in BIOS this is the POST stage and prior to booting the operating system. Access to the allocated memory is locked and secured from modification or override by the BIOS by setting the D_LCK bit to 1, memory can only be accessed if D_LCK is set to 0. This locked state is not accessible even by the Operating System. However it should be noted that if the malware code is executed before reaching the POST and prior to the setting of the D_LCK bit, then the SMI is accessible and vulnerable to altercation.

Upon entering the SMM the CPU clears the “IF” flag to disable INTR interrupts masks INTR, NMI, SMI, INIT, and A20M interrupts. However INTR can be enabled within SMM when the SMM handler set the IF flag to 1. A20M is disabled so that address bit 20 is never masked when in SMM mode (Advanced Micro Devices Inc, 2011). There are six steps to this process, see Diagram 2.

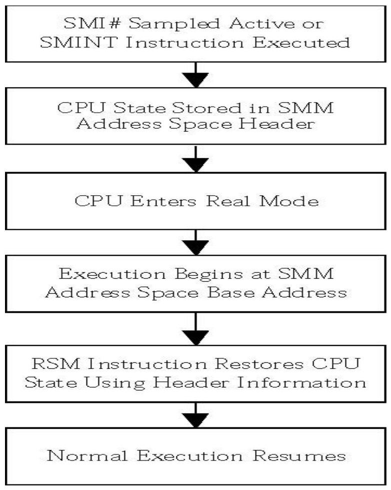

SMI# Sampled Active or SMINT Instruction Executed
CPU State Stored in SMM Address Space Header
CPU Enters Real Mode
Execution Begins at SMM Address Space Base Address
RSM Instruction Restores CPU State Using Header Information
Normal Execution Resumes

Diagram 2 SMI Execution Flow Diagrams
(Cyrix Inc, 1998, P 20)

4. Embedded Rootkit Analysis

Commercial off the shelf software tools (anti malware) target malware detections in the application layer (i.e. the operating system files, directories and data). Once the malware is detected, often by detection techniques such as the "Virus Signature" where engineers and researchers study the bit patterns and the footprint of the virus, identifying the code, procedures, steps and methods of a specific attack.

Arguably, it is possible to have a legitimate business case for rootkits. The first widely known instance of broad spread exploitation was in 2005. Within that incident the rootkit was devoid of known malicious intent at least in accordance with the corporation that implanted, Sony BMG Music. Similarly Dell announced that a malware code W32.Spybot program was detected on the embedded server management firmware of the Power Edge product line including Power Edge R310, Power Edge R410, Power Edge R510 and Power Edge T410 (Dell Inc, 2010).

Each rootkit differ in its disguise and functionality depending on intention; however all have common target by obfuscating one or several of the following system entities: Registry Keys, Files, Drivers, TCP/IP Ports, Processes, and Services.

A comparison of the information in PID (Process IDentifier) and TID (Thread IDentifier) of table PspCidTabl to table PsActiveProcessList would reveal hidden processes. Similarly noticeable heavy memory usage, I/O read, I/O writes maybe detected by the embedded systems tools. Another common Operating System rootkit malware involve process rights elevation such as in the NT Rootkit by hooking the SSDT to perform full ring0 (highest CPU priority) functionalities. Later modification involved process rights elevation surfaced with rootkits like FU and later improvement with FUto rootkit.

Rootkits are typically based on undiscovered vulnerabilities while obfuscations based on lack of system knowledge or lack of proper detection tools. Unless the malware is a known vulnerability attack, few people would know its details except to some including the malware writer. Once the vulnerability is known, with proper detection tools it would no longer be obfuscated and can be detected by anti-virus tools or engineering forensic methods, unless it is a polymorphic which can dynamically change its binary code or behaviours to avoid identification by a pre determined pattern of binary bytes or a pattern of functional characteristics.

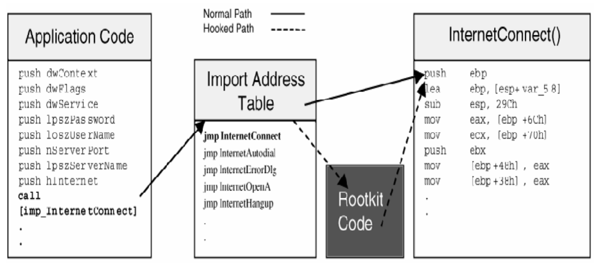
Generally there are three types of rootkits (the last is firmware relevant):

- Software embedded rootkit: reside within the Operating system
- Hypervisor rootkit: resides between the operating system and the hardware. Forcing the original operating system to run over obfuscated virtual machine, by creating a Hypervisor interface below the Hardware Abstraction Layer (HAL), "The HAL is a lightweight runtime environment that provides a device driver interface for programs to connect to the underlying hardware. The HAL application program interface (API) is integrated with the ANSI C standard library. (Altera Corporation, 2010).
- Firmware rootkit: resides inside the hardware beyond the reach of the operating System.

A rootkit placed in the Operating System hijacks system's behaviour by changing function start or function pointer to redirects execution flow by modifying the "syscall list" of the pointer in kernel structure; System Service Descriptor Table (SSDT) thereby controlling the behaviour of the function with hooks to the malicious code. Code illustration of creating a Hook-System-Call on a system under attack can be in the following format (Informative information for the uninformed, 2007):

```
PVOID HookSystemCall(
        PVOID SystemCallFunction,
        PVOID HookFunction)
{
        ULONG SystemCallIndex =
                *(ULONG
*)((PCHAR)SystemCallFunction+1);
        PVOID *NativeSystemCallTable =
                KeServiceDescriptorTable[0];
        PVOID OriginalSystemCall =

        NativeSystemCallTable[SystemCallIndex];

        NativeSystemCallTable[SystemCallIndex]      =
HookFunction;
        return OriginalSystemCall;
}
```

Similarly an application type rootkit redirect system calls by altering the Import Access Table (IAT), which is a CPU specific table localized in kernel used by the system for managing exceptions and interruptions, See Figure 2 Application code flow for normal execution path vs. hooked execution path of an IAT hook.



```
push dwContext
push dwFlags
push dwService
push lpszPassword
push loszUserName
push nServerPort
push lpszServerName
push hinternet
call
[imp_InternetConnect]
.
.
```

```
jmp InternetConnect
jmp InternetAutodial
jmp InternetErrorDlg
jmp InternetOpenA
jmp InternetHangup
.
.
```

```
push    ebp
lea     ebp, [esp+ var_5 8]
sub     esp, 29Ch
mov     eax, [ebp +6Ch]
mov     ecx, [ebp +70h]
push    ebx
mov     [ebp +48h] , eax
mov     [ebp +38h] , eax
.
.
```

Figure 2 Application code flow for normal execution path vs. hooked execution path of an IAT hook.
(Hoglund 2005, P 74)

Planting the rootkit in the SMM away from the operating system makes the rootkit superfluously stealthy. This is a numerous advantage however it would severely limit viral proliferation. This is so since the malware code implanted in the firmware must be expressly written for each system attacked.

5. Analysis of a Firmware Attacks

Firmware vulnerabilities unlock and expose the firmware (or the BIOS) to an attack. This should not be confused with the extensively researched vulnerabilities associated with the Master Boot Record (MBR) Viruses.

MBR is considered a legacy type malware but certainly is not an inert in its effect. It is intended as a malicious attack works by erasing the first memory sector of the booting device thereby rendering the system not bootable. The MBR resides in the first externally accessible (outside the hard drive) sector. This sector is considered as track zero (0), number of tracks depends on drive size, and each track has sixty-three sectors. Track 0 functions as the disk-mapping directory containing the hard disk partition tables as well as the initial loader to the disk operation. Sixty-two sectors after the location of the MBR, the DOS Boot Record (DBR) is located which contains the initial loader of an operating system and the logical drive mapping information. In contrast to the hard disk sectors layout, in floppy drive, the Floppy Boot Record (FBR) is located on the first track of a diskette and it is used for the same purposes as the DBR (F-Secure, 2009).

Malwares in the BIOS is the most lethal firmware hack. It can be in the form of firmware corruption leading to a Denial of Service (DoS), or could be persistent, subvert, and undetectable even with powerful anti malware tools leading to a persistent malware as part of legitimate software implanted and embedded in the hardware. If the intention were propagation then a boot virus that is copied or downloaded externally such as the Internet would infect the hard drive. The hard drive would in turn be triggered to infect other memory devices (USB insertion would trigger a virus residing in the drive to replicate itself on the USB thumb drive).

Firmware vulnerabilities are classified into three groups as they relate to the functionality of an attack:

1. Vulnerabilities existing in the system firmware
2. Physical attack on firmware.
3. Malicious code integrated within the flash chip

6. Known Vulnerabilities in the Firmware BIOS32

To facilitate the marketing of their devices, hardware manufacturers provide BIOS systems information for vendors to access an OS based on the 32-bit architecture. A vendor utilizes this info to build new services and features, such as the direct-to-kernel binary execution.

To cause a direct-to-kernel binary execution a designer of a specific vendor would invoke this service by placing a BIOS32 header somewhere in the E000:0000 to F000:FFFF memory region, 16-byte aligned. The headers structure is in the following programming outline (Sacco, 2010):

Offset	Bytes	Description
0	4	Signature
4	4	Entry point for the BIOS32 Service (here hacker place pointer to malicious code)
	8	Revision level, (hacker would put rev 0)
9	1	Length of the BIOS32 Headers in paragraphs (hacker would insert length)
10	1	8-bit Checksum.
11	5	Reserved for future use

In systems operation, the OS consider the BIOS as a trusted platform. Therefore, since the OS always trust the BIOS this can be taken advantage of by hackers by simply inserting a header to the malware code instead of the expected software code in the BIOS header pointing to the OS.
In the structure above such insertion utilizes part of the 16 bytes placing the malicious code in offset 4 using 4 bytes. Upon system restart this routine would be executed, however instead of a call be placed to the BIOS, the pointer at offset 4 would be called. This pointer points to any offending code. Upon the execution of this offending code, the pointer return to a similar offset (without malware code) so as to normally initiate the BIOS starts up process:

Offset	Bytes	Description
0	4	Signature
4	4	Entry point for the BIOS32 Service
8	1	Revision level
9	1	Length of the BIOS32 Headers in paragraphs
10	1	8-bit Checksum.
11	5	Reserved for future use

Such an attack is similar in function to a network attack on communication protocol such as TCP/IP, called man in the middle attack.

Another method is for the hacker to locate system services in memory. This is accomplished by initiating a pattern search of known system services in memory. Once a specific service is located and confirmed with a checksum, the hacker would inject a pointer to the offending code. One such commonly available service found in typical modern BIOS is a USB function such as the Plug and Play ($PnP); others are the Post Memory Manager ($PMM) and the BIOS32 (discussed above).

Other attacks include the "Boot/Stoned.Monkey" which re-routs the BIOS-level disk calls through its own code. As such it is not a direct attack on the BIOS but rather it is a mechanism to place the BIOS in an un-trusted mode; "Boot virus" which traps the BIOS functions disk interrupt vector Int 13h, then write to the MBR (discussed earlier); Physical attack on Firmware by data alteration such as the routine in the "W32.Mypics.Worm" which overwrite the high byte of the two-byte CMOS checksum value in the system BIOS, resulting in the computer displaying a system BIOS error and failure to boot. This propagates by automatically sending itself through the contacts list of the Outlook address book; "Chernobyl Virus (CIH)" also known as Spacefiller, this malware scheme accomplished by two steps payload approach. In the first step the Virus deletes the contents of the partition table and fills the first 1024 Kbytes of the host's boot hard drive with zeroes, beginning at memory location sector 0. The second step is to attack the i430TX motherboard BIOS. CIH only affected Windows computer operating systems; Windows 95 kernel and subsequently released versions including Windows 95, Windows 98 and Windows ME, which were widespread operating system at that time; "Bare-metal BIOS bug" privilege escalation rootkit vulnerability found in Intel manufactured Desktop system Boards, in which under specified circumstances allow privilege escalation by software running administrative (ring 0) to modify software code running in System Management Mode (SMM).

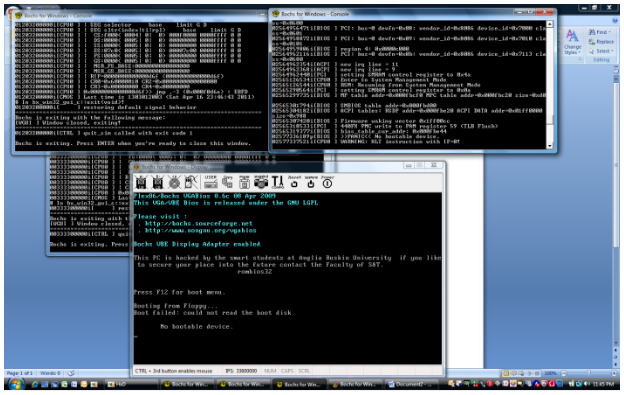
7. Simulation of an Attack on the Firmware

Every traditional computing system such as the PC, IPhone, IPad, and other hand held devices encompass an embedded firmware. The closest to the metal in every case is the BIOS. This simulation is a proof of concept on any of these devices; however the most accessible and technically friendly device is the PC. As such, the simulation included a set-up of a desktop PC with an AMI BIOS. Even though the operating system is irrelevant, this set-up included a Windows XP SP3 platform with BIOS editing and flashing tools.

Through this research it was possible to access the BIOS and alter the BIOS code so as to simulate an actual attack in progress. The first attempt was altering the POST routine, resulting in placing the booting process in suspense prompting the user for an input. The input requested was a muted request, but meant to show a hack at the metal level. The next step was to flash the BIOS with a clean code then the Platform functioned as normal. In the second attempt the system indicated that the booting process was corrupt (in suspended state) and the BIOS would not proceed beyond the initial booting process, it appeared that it was stuck in the POST routine.

The felicity achieve by the first altered scenario, was at the expense of a presumed falsity that the BIOS would always be reloadable. This experimentation rendered the first system unoperateable, which was a calculated risk. The proof of concept implementation in hardware is problematic since each modification to the firmware requires a new hardware flashing risking permanent corruption to the loading module, which corrupts the boot up process.

In the second simulation/testing set-up, it was relied on running simulation on a PC platform using a "PC emulator" called Bochs. Figure 3 Screen capture of the working environment using Bochs simulator. The set-up shows that the BIOS after a hack was not able to hand over booting instructions to a working bootable device.


Bochs for Windows - Console
Bochs for Windows - Console
Bochs for Windows - Display
USER
Copy
Paste
Snapshot
CONFIG
Reset
SUSPEND
Power
Change Styles
CTRL + 3rd button enables mouse
IPS: 28800000
NUM
CAPS
SCRL

```
01203200000i[CPU0 ] | SEG selector     base    limit G D
01203200000i[CPU0 ] | SEG sltr(index|ti|rpl)     base    limit G D
01203200000i[CPU0 ] |  CS:f000( 0004| 0|  0) 000f0000 0000ffff 0 0
01203200000i[CPU0 ] |  DS:0000( 0005| 0|  0) 00000000 0000ffff 0 0
01203200000i[CPU0 ] |  SS:0000( 0005| 0|  0) 00000000 0000ffff 0 0
01203200000i[CPU0 ] |  ES:07c0( 0005| 0|  0) 00007c00 0000ffff 0 0
01203200000i[CPU0 ] |  FS:0000( 0005| 0|  0) 00000000 0000ffff 0 0
01203200000i[CPU0 ] |  GS:0000( 0005| 0|  0) 00000000 0000ffff 0 0
01203200000i[CPU0 ] |  MSR_FS_BASE:0000000000000000
01203200000i[CPU0 ] |  MSR_GS_BASE:0000000000000000
01203200000i[CPU0 ] | RIP=000000000000d6f (000000000000d6f)
01203200000i[CPU0 ] | CR0=0x60000010 CR2=0x0000000000000000
01203200000i[CPU0 ] | CR3=0x00000000 CR4=0x00000000
01203200000i[CPU0 ] 0x000000000000d6f>> jmp .-3 (0x000f0d6e) : EBFD
01203200000i[CMOS ] Last time is 1303012003 (Sat Apr 16 23:46:43 2011)
# In bx_win32_gui_c::exit(void)!
01203200000i[     ] restoring default signal behavior
========================================================================
Bochs is exiting with the following message:
[WGUI ] Window closed, exiting!
========================================================================
01203200000i[CTRL ] quit_sim called with exit code 1

Bochs is exiting. Press ENTER when you're ready to close this window.
```

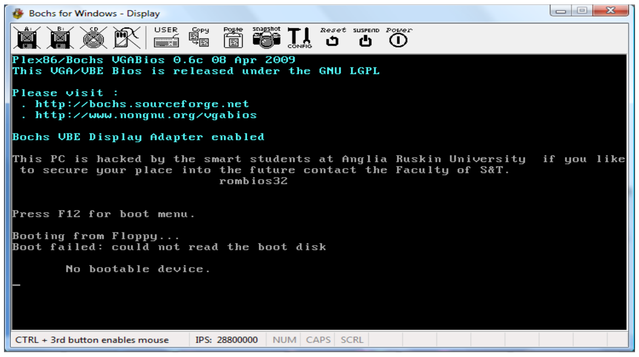
```
ss=0x0600
02564956471i[BIOS ] PCI: bus=0 devfn=0x08: vendor_id=0x8086 device_id=0x7000 cla
ss=0x0601
02564958072i[BIOS ] PCI: bus=0 devfn=0x09: vendor_id=0x8086 device_id=0x7010 cla
ss=0x0101
02564959806i[BIOS ] region 4: 0x0000c000
02564962116i[BIOS ] PCI: bus=0 devfn=0x0b: vendor_id=0x8086 device_id=0x7113 cla
ss=0x0680
02564962354i[ACPI ] new irq line = 11
02564962368i[ACPI ] new irq line = 9
02564962440i[PCI  ] setting SMRAM control register to 0x4a
02565126534i[CPU0 ] Enter to System Management Mode
02565126544i[CPU0 ] RSM: Resuming from System Management Mode
02565290564i[PCI  ] setting SMRAM control register to 0x0a
02565299735i[BIOS ] MP table addr=0x000fbcf0 MPC table addr=0x000fbc20 size=0xd0

02565301794i[BIOS ] SMBIOS table addr=0x000fbd00
02565304182i[BIOS ] ACPI tables: RSDP addr=0x000fbe20 ACPI DATA addr=0x01ff0000
size=0x988
02565307420i[BIOS ] Firmware waking vector 0x1ff00cc
02565318533i[PCI  ] 440FX PMC write to PAM register 59 (TLB Flush)
02565319377i[BIOS ] bios_table_cur_addr: 0x000fbe44
02577336189p[BIOS ] >>PANIC<< No bootable device.
02577337521i[CPU0 ] WARNING: HLT instruction with IF=0!
```

```
Plex86/Bochs VGABios 0.6c 08 Apr 2009
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

This PC is hacked by the smart students at Anglia Ruskin University  if you like
 to secure your place into the future contact the Faculty of S&T.
                            rombios32

Press F12 for boot menu.

Booting from Floppy...
Boot failed: could not read the boot disk

       No bootable device.
```

Figure 3 Screen capture of the working environment

Figure 4 Display of the working screen of BIOS attack in progress. delaying the screen capture of hacked BIOS. Not only is the booting sequence cannot proceed "no bootable device" on the twelfth text line, but also the hacking included an inserted text message as a proof of concept. This hacked PC would display a message on the screen upon the booting stage stating on the seventh text line the following hacked text:

"This PC is hacked by the smart students at Anglia Ruskin University if you like to secure your place into the future contact the Faculty of S&T."



```
Plex86/Bochs VGABios 0.6c 08 Apr 2009
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

This PC is hacked by the smart students at Anglia Ruskin University  if you like
 to secure your place into the future contact the Faculty of S&T.
                            rombios32

Press F12 for boot menu.

Booting from Floppy...
Boot failed: could not read the boot disk

       No bootable device.

_
```

Figure 4 Display of the working screen of BIOS attack in progress.

8. Conclusion

This paper presented the technical knowledge and researched malware vulnerabilities below the OS.

This paper showed in a novel approach that if the firmware cannot be trusted then the OS and the software that are dependent on the underlying firmware might not be trusted.

This paper is also unique in that it provided comprehensive and novel analysis that can be applied to any malware. This would help future researchers and practitioners in exposing future polymorphic and evolutions of these vulnerabilities.

References

[1] Advanced Micro Devices, Inc, 1995. AMD BIOS Development Guide, USA, P 22.

[2] Advanced Micro Devices, Inc, 1997. *Elan SC400 and Elan SC410 Microcontrollers User's Manual*, Advanced Micro Devices, USA, P 3-5.

[3] Advanced Micro Devices, Inc, 2011. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 14h Models 00h-0Fh Processors, 43170 Rev 3.09 - May 02, 2011.* Advanced Micro Devices, Inc, USA.

[4] Altera Corporation, 2010. *Nios II Software Developer's Handbook*, Altera Corporation, USA, P 5-1.

[5] Castelluccia C, Francillon A, Perito D, Soriente C, 2009a. On the difficulty of software-based attestation of embedded devices. *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, November 2009.

[6] Castelluccia C, 2009b. *Despite what some people say, Code Attestation of Embedded Devices is still difficult*, Senior Research Scientist the French National Institute for Research in Computer Science and Control, France. [online] Available at: <http://planete.inrialpes.fr/~ccastel/>

[7] Chen, T.M, 2010. Stuxnet, the real start of cyber warfare?, *IEEE Communications Society*, November/December 2010, Vol. 24, No. 6.
Cyrix Inc, 1998. *Application Note 107 - MII SMM DESIGN GUIDE*. [online] Available at: <http://datasheets.chipdb.org/Cyrix/M2/an107.pdf

[8] Dell Inc, 2010. *PowerEdge R410 replacement motherboard contains malware?!*. [online] Available at: <http://en.community.dell.com/support-forums/servers/f/956/t/19339458.aspx>

[9] Duflot L, Etiemble D, Grumelard O, year unknown. Using CPU System Management Mode to Circumvent Operating System Security Functions, DCSSI 51 bd. De la Tour Maubourg 75700 Paris Cedex 07 France

[10] Embleton S, Sparks S, Zou C, 2008. SMM Rootkits: A New Breed of OS Independent Malware, *SecureComm '08 Proceedings of the 4th international conference on Security and privacy in communication networks,* Association of Computing Machinery, USA.

[11] Francillon A, Castelluccia C, Perito D, Soriente C, 2010. Comments on Refutation of On the difficulty of Software-Based Attestation of Embedded Devices. [online] Available at: <http://planete.inrialpes.fr/~ccastel/PAPERS/2010_CCS_attestation_comments_on_rebutal.pdf>

[12] F-Secure, 2009. *Boot virus*. [online] Available at: <http://www.f-secure.com/v-descs/boovirus.shtml>

[13] Hoglund G, Butler J, 2005. *Rootkits: Subverting the Windows Kernel*, Addison Wesley Professional, 2ed, USA, P 74.
Informative information for the uninformed, 2007. *SSDT*. [online] Available at: <http://uninformed.org/index.cgi?v=8&a=2&p=10>

[14] Intel Corporation, 2008b. *Intel ID Intel® Desktop and Intel® Mobile Boards Privilege Escalation INTEL-SA-00017*. [online] Available at: < http://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00017&languageid=en-fr>

[15] King S.T., Chen P.M., Wang Y, Verbowski C, Wang H.J., Lorch J.R., 2006. Subvirt: Implementing malware with virtual machines, *IEEE Symposium on Security and Privacy*
Malwarecookbook, 2011. *malwarecookbook - Revision 26: /trunk/17/6*. [online] Available at: <http://malwarecookbook.googlecode.com/svn/!svn/bc/26/trunk/17/6/>

[16] Michael C, 2010. *How to detect system management mode (SMM) rootkits*, Searchsecurity. [online] Available at: <http://searchsecurity.techtarget.com/tip/0,289483,sid14_gci1334155,00.html>

[17] Perrig A. and van Doorn L. 2010. Refutation of on the difficulty of software-based attestation of embedded devices. Based on the Paper Castelluccia C, Francillon A,

[18] Perito D, Soriente C, 2009. On the difficulty of software-based attestation of embedded devices. Proceedings of ACM *Conference on Computer and Communications Security (CCS)*, November 2009.

[19] Ravi S, Raghunathan A, Kocher P, Hattangady S 2004. Security in Embedded Systems: Design Challenges, *Association for Computing Machinery Transactions on Embedded Computing Systems*, Vol. 3, No. 3, August 2004, Pages 461-491

[20] Sacco A, 2010. *Persistent BIOS Infection*, Core Security Inc, USA.