Thesis no: MECS-2014-11

Trusted memory acquisition using UEFI

Michel Nim Markanovic
Simeon Persson

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Engineering. The thesis is equivalent to 20 weeks of full-time studies.

Contact Information:
Author(s):
Michel Nim Markanovic
E-mail: mimc09@student.bth.se

Simeon Persson
E-mail: sipe09@student.bth.se

University advisor:
Prof. Stefan Axelsson
Dept. Computer Science & Engineering

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. For computer forensic investigations, the necessity of unmodified data content is of vital essence. The solution presented in this paper is based on a trusted chain of execution, that ensures that only authorized software can run. In the study, the proposed application operates in an UEFI environment where it has a direct access to physical memory, which can be extracted and stored on a secondary storage medium for further analysis.
Objectives. The aim is to perform this task while being sheltered from influence from a potentially contaminated operating system.
Methods. By identifying key components and establishing the foundation for a trusted environment where the memory imaging tool can, unhindered, operate and produce a reliable result.
Results. Three distinct states where trust can be determined has been identified and a method for entering and traversing them is presented.
Conclusions. Tools that does not follow the trusted model might be subjected to subversion, thus they might be considered inadequate when performing memory extraction for forensic purposes.

Keywords: UEFI, Secure Boot, trust, computer forensics

Contents

Abstract **i**

1 Introduction **1**
1.1 Background 1
1.2 The trust issue 2
1.3 Computer startup 3
1.4 UEFI 3
1.4.1 Signing and verifying in UEFI 4
1.5 Memory 5

2 Related Work **8**

3 Method **9**
3.1 Research questions 9
3.2 Hypothesis 9

4 Implementation **10**
4.1 Tepid reboot 10
4.2 Programmed reboot 11
4.3 System request magic 11
4.4 Reset switch 12
4.5 Memory dumping 12
4.6 The static chain of trust 13
4.6.1 OS 13
4.6.2 Tepid reboot 14
4.6.3 UEFI 14
4.6.4 Umemdmp 15
4.6.5 Storage device 16
4.7 Validation 17

5 Results **18**
5.1 Looking for known signatures 18
5.2 Complete memory imaging 18
5.3 Resisting obstruction 19

5.4 Regarding the hypothesis 19

6 Analysis **20**
6.1 Memory dumping 20

7 Conclusions and Future Work **22**
7.1 Conclusion 22
7.2 Future Work 22

Chapter 1

Introduction

To give the reader a brief introduction to the current state of the field of RAM memory forensics and UEFI, this chapter aims to discuss current issues and to prepare the reader for the proposed research questions presented in chapter 3.1.

1.1 Background

Computer forensics is a relatively young field of computer security and computer science. The aim is to scientifically assist computer crime investigations and aids data extraction or recovery from devices that may be malfunctioning. As a science it needs to have a sound methodology in order to identify, preserve and obtain data of importance from a system. Data of importance can be traces of cyber intrusion, signs of malware activity, confidential information such as company secrets and user password keys. This data is often obtained from a secondary storage device such as a hard drive, an USB-stick or a camera to name a few, or from the primary memory that is also known as RAM. The difference between primary and secondary storage devices are that primary memory is used for temporary storage while the computer is running and the data remains there until it is overwritten or the computer is powered down. Secondary memory is used for longterm storage for files and such, since the content is not lost when the computer is shut down. For instance, when a user is prompted for a password when logging on to a computer system the password will temporary reside in the RAM, where it might be retrieved by a forensic tool.

Traditional memory forensics typically consists of a RAM image extracted by using a framework such as Volatility[6]. The image is usually acquired by using OS specific API calls to dump the RAM. If there is a presence of anti-forensic malware residing within the kernel, it might subvert the kernel to hide its own presence in the memory image. Given this fact, we need a way to properly dump the entire RAM content without the possible interference of said malware. In the real mode, prior to OS boot, the BIOS has full access to the entire physical memory range. If we can somehow make a clean copy of the memory while in this state, we can be sure that at least no OS related malware has contaminated the image (there might however still exist a possibility that the BIOS itself might

be infected).

This project aims to address the creation of an uncontaminated memory image using UEFI, which is the successor to the traditional pre-boot environment known as BIOS. The task of addressing the memory acquisition consists of the development of a securely signed[1] custom UEFI firmware.

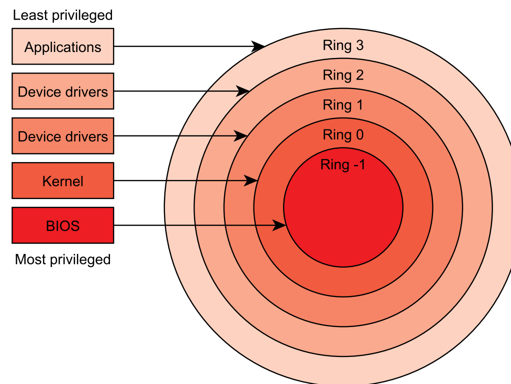

Figure 1.1: *Privilege-rings [8].*

The custom firmware will reside in "ring -1", which is located one step prior to the kernel, thus have the potential to reveal traces of malware that is operating in ring 0 and above.

1.2 The trust issue

The word secure itself describes a dichotomy, saying that something is either secure or not. If something is secure, it is then said to withstand all possible kinds of attacks presented to it, which, of course a computer system can not do. For instance, one of the major problems in computer security is that of a given systems inability to, with an infallible guarantee, prove itself to be able to perform a suitable task given to it. That is, the system may report to the user that it is performing a series of tasks, while it instead does something else.

Therefore, a new term is needed to suggest a sense of security in computer systems. That is, the concept of *trust*, which does not present a clear dichotomy like the one viewed in security. By being a more volatile term, an acceptable degrees of trust can be achieved in a computer program for instance, with rigorous analysis and testing made by a party that you *trust*. This strength is however also the obvious drawback to trusted systems, the user has to rely on confirmations made by others to achieve a feeling of security [13].

The topic of trust and the implementations chosen in this paper is further discussed in chapter 1.4.1 and 4.6.

[1] UEFI TPM - http://en.wikipedia.org/wiki/Trusted_Platform_Module

1.3 Computer startup

Once the power switch is toggled on, the computer device perform a power-on self-test[2]. This test is made in order to verify key components of the system. If errors are found they are reported back to the user at the end of the POST-sequence. For a reboot, the POST-sequence varies depending on reboot-method, that is, either a *cold* or *warm* reboot.

A cold reboot is issued when the devices power has been toggled off and then back on again, i.e by a reset button. The northbridge is then responsible for fetching the first instruction the CPU will execute after a reboot. During this type of reboot, the CPU will be directed to read the BIOS located in the flash memory.

A warm reboot on the other hand, is performed by the operating system, i.e when a user chooses to restart the system via software provided by the operating system. In this case, the northbridge will point the CPU to the RAM instead, from where already loaded BIOS code will be executed.

Since the introduction of DDR3 memory chips, the RAM is kept powered during system resets[3] – therefore retains all memory content[3]. This feature is used by the implementation presented in this paper, here called *tepid reboot*[4]. Tepid reboot is discussed and covered further in chapter 4.1.

1.4 UEFI

UEFI[5], earlier referred to as the successor to traditional BIOS, aims to give developers greater flexibility to develop and modify the pre-OS environment, while preserving traditional legacy BIOS features. One of the new features provided is the Secure Boot capability[5], further explained in the next section, which allows UEFI-applications and drivers to be cryptographically signed and verified prior to being run.

---

[2] Commonly abbreviated and referred to as POST.
[3] This does not include the power on/off button.
[4] Newly coined terminology to distinguish it from warm and cold boots.
[5] Unified Extensible Firmware Interface.

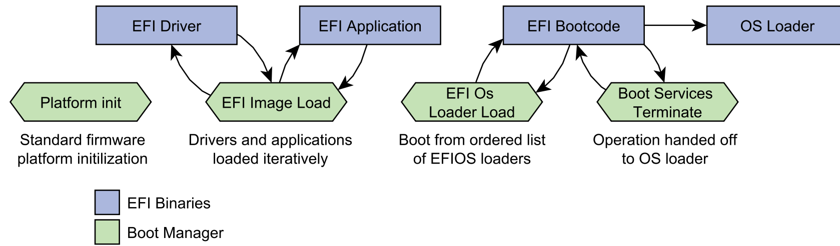

Figure 1.2: *Efi boot sequence [5].*

The UEFI boot sequence presented in figure 1.2 displays a simplified overview of the four different stages of boot. This project utilizes the UEFI image load stage, where the memory dumping application presented in this paper operates. It is also in this stage that Secure Boot grants or denies an UEFI application or driver running permissions. If execution is permitted, the memory dumping tool creates an image of the physical memory and saves it to a secondary device as stated in chapter 4.5.

Stages previous and following the UEFI Application stage are beyond the scope of this paper, however the interested reader can find a detailed explanation in *UEFI Networking and pre-os security* [5].

1.4.1 Signing and verifying in UEFI

In order to execute trusted software in a UEFI environment, the native Secure Boot capability is used, whose purpose is to make sure that unauthorized code can not be run during system boot. This is done by matching previously stored cryptographic certificates associated with the trusted software. Since Secure Boot only accepts certificates that follows the `.DER`[6] binary format specified by the X509 standard, it is vital for a signing party to generate certificates that can be converted to this format. These certificates[7] are ready to be stored in a set of different Secure Boot related databases built into the UEFI foundation on the system. In short, if a certain certificate used to sign a given software is missing from the database, that software can not be run. The signing procedure is as follows [20]:

1. Create a private key
2. Create a certificate using the private key
3. Sign the software with the certificate, i.e. by using sbsign[8]

[6]Distinguished Encoding Rules.
[7]This certificate could either be provided by a vendor or it could be generated by the user.
[8]This software securely signs an application, preferably by using trusted certificates already installed on the system.

4. Enroll the certificate in appropriate UEFI databases

These databases and the verification process is further described in detail in chapter 4.6.3.

The trust is established via the help of the native Secure Boot capability [5], which allows the UEFI-application to be cryptographically signed and verified, thus ensuring that its integrity is intact when run and that only other trusted devices can utilize it. Since the extraction occurs in the UEFI image load stage, which in turn operates in real mode, there can be a high degree of assurance that kernel malware has not yet been instantiated.

The signing is of utmost importance, since it is the linchpin in the *the static chain of trust*[9]. Without this feature, the integrity of the chain can not be assured and the consequences can result in, amongst other things, the masquerading of sinister software as demonstrated by Kallenberg et al. [15].

1.5 Memory

As mentioned in the introduction, the purpose of RAM memory, also known as secondary memory, is to preserve data that does not need permanent storage. In the case of an system intrusion, an intruder might clear log entries in order to hide her presence in the system. It is vital to a forensics team to preserve RAM content since it might contain traces of malicious activities.

When we discuss RAM memory we make an distinction between *physical* and *virtual* memory. The physical memory space normally consists of available memory alongside with reserved memory segments for peripheral devices such as external video cards, sound cards etc. The physical address space usually consists of a direct map over this physical memory and is available under `/dev/mem` device in various Unix-like systems and in `\\.\PhysicalMemory` under Windows. Alas, direct access to this device is not granted in protected mode[10], unless the region has already been mapped into virtual address space. Virtual space provides the user land the benefits of running multiple applications simultaneously, even though there might not be enough available RAM on the system to do so. This works by saving down parts of RAM content that has not been recently used to a secondary storage media. The mapping of physical address space into virtual is done by the Memory Management Unit (MMU) along with *page tables*. This means that each time a program needs access to physical memory, it performs a look-up on the given virtual address. The virtual address is treated as a bit field where each field is used to index into the page tables. With the help from control registers CR0 up to CR3, the MMU can correctly translate and aid the kernel

[9] Described in full in the dedicated chapter 4.6

[10] Operating system mode where memory access is restricted.

resolve possible errors such as fetching a page that is associated with a virtual address, and that is currently not present in memory (page fault).

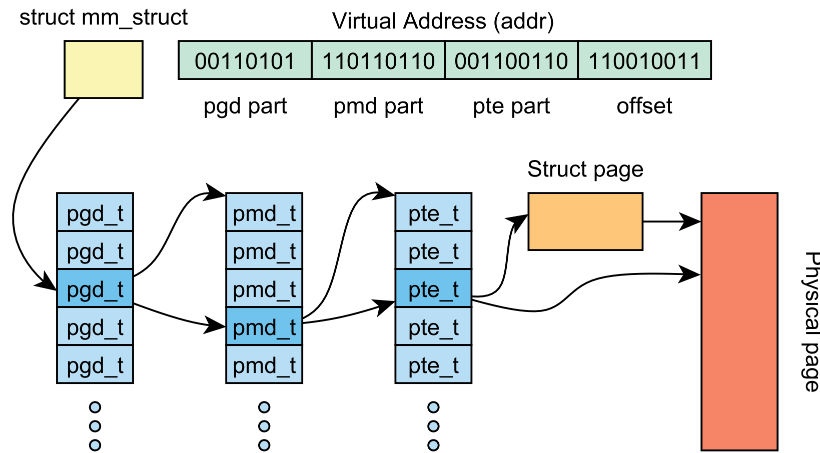

Figure 1.3: *Virtual to physical address lookup. Both hardware and software relationships are depicted [17]. Page directories (pgd_t), tables (pmd_t) and entries (pte_t) combined points to a physical address.*

UEFI environment operates in real mode, meaning, amongst other things, that there is no virtual mapping and applications have direct access to physical memory. At the end of the UEFI boot sequence, the function GetMemoryMap() is called in order to acquire a map of available RAM. This procedure was, in previous BIOS boot, known as E820 since an BIOS interrupt 15 was issued with the value of 0xE820 in order to retrieve the memory geometry. This call is followed by the EXIT_BootServices() method, which exits the boot sequence and passes on the memory map to the kernel. As stated previously, the physical memory needs to be mapped into the virtual address space. The Linux kernel function kmap() creates a new page mapping and returns a virtual address to it. Although a physical memory dump can be analyzed as-is, it greatly benefits forensic personnel when properly mapped with pages since related data is aligned to provide a context.

Peripheral devices that are attached to the PCI bus, such as graphic cards, are able to access the systems primary memory via the hardware component *northbridge*. These devices are commonly equipped with their own dedicated memory area, which can be populated by a copy of the systems physical memory requested by the northbridge. This direct operation is what is known as Direct Memory Access[11]. This process is used by Carrier et al. in their hardware based solution, Tribble, for acquiring memory [10].

[11] Commonly known as DMA.

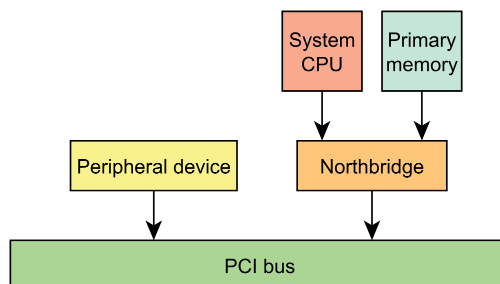

Figure 1.4: *DMA can be performed by peripheral devices independently of the CPU.*

Using DMA is not without a major risk, namely the consequences of accidentally reading a reserved device memory region may include unaccounted for restarts, memory corruption and system crashes. To remedy these situations reserved memory needs to be avoided when performing a system memory extraction. However, subjected areas differ between systems and can not be generalized when comparing two different dumps, acquired by different memory forensic applications, on the same target system.

When performing a memory image analysis one has to keep in mind that the memory map is very specific to the target system. For instance, the distribution, kernel version, installed hardware, architecture etc. all contribute to how the memory is finally mapped on a running machine. In order to be able to fully understand a memory dump, the whole environment of where the memory was obtained needs to be replicated.

Chapter 2

Related Work

As mentioned in section 1.1, the go-to framework for memory image processing is the platform-independent project Volatility. It provides an interactive Python shell and a collection of open source forensic tools. Generally, memory access is acquired by requesting a memory range through an OS API call, but we have also seen over the years that DMA[1] in a pre-boot environment such as BIOS can map and extract memory. The latter type of memory mapping can be useful in situations where anti-forensic techniques, or malware are suspected to interfere with the memory image creation process [9]. This is achieved by handling the mapping in the pre-boot environment, before the actual operating system is loaded, thus rendering the potential malware impact moot, since it operates in a lower ring of privilege.

The hardware solution Tribble is a device that plugs into the PCI slot of a target device [10]. Tribble utilizes northbridge DMA to acquire a RAM image into an external storage medium. During the process the CPU execution is halted in order to prevent further memory manipulation made by the operating system. However, this device has to be pre-installed on a target system in order to perform the intended memory imaging.

Actaeon is a Volatility plugin used for extracting memory exclusively from virtual machines [16]. The solution scans for the hypervisor and can identify whether if the virtual machines are run in a parallel or a nested mode. Since Actaeon does not map non-virtual devices, it can not be used to map the entire RAM area for a given hardware system.

[1] Direct Memory Access.

Chapter 3

Method

This chapter discusses the approach chosen to implement the acquiring utility along with the research questions followed by some arguments describing the advantages and drawbacks with the given solution.

3.1 Research questions

1. Is it possible to use UEFI software to extract known signatures from the random access memory, that have been generated in a operating system environment.

2. Can UEFI software be used in order to extract a complete image of the RAM?

3. In order to extract a complete memory image, is it possible to withstand obstruction from a potential operating system rootkit?

3.2 Hypothesis

A given memory subset $m_0 \subset M$, where M is the fully accessible primary memory area extracted by `Umemdmp`[1], is compared with another memory subset $m_1 \subset M$ given by the UEFI memory inspection tool `dmem`[2].
H_0: M_0 is significantly different from M_1.
H_1: M_0 is equivalent to M_1.

If the null-hypothesis can not be rejected, the experiment has failed and the custom firmware can not be used for forensic purposes.

The implementation method is chosen to answer the research questions and validate the hypothesis. Details for the process are given in chapter 4.

[1] `Umemdmp` is the authors custom made memory dumping tool that honors the chain of trust.
[2] Tianocores UEFI memory inspection utility. It is unable to write memory content to file and also requires the EDK2 suite to be installed on the system [19].

Chapter 4

Implementation

The following chapter covers the implementations used to verify the hypotheses given in this paper. First, an introduction to the *static chain of trust* is presented in section 4.6. Further, the topic of *tepid reboots* is presented in 4.1 needed to initiate the *static chain of trust*. Section 4.2 to 4.4 discusses the different methods of rebooting a system, that is, the different ways to establish the first link in the *static chain of trust*. Following is the UEFI implementation of a memory imager discussed in section 4.5, which is presented as the concluding link in the *static chain of trust*.

4.1 Tepid reboot

This section discusses methods of system resets occurring on an already running system and the levels of trust that can be achieved for each. It is important to distinguish the chosen restart method from a *warm reboot*, which normally implies that the BIOS image is loaded from RAM instead of the image residing in on-chip flash memory [11]. This ordinary *warm reboot* not only breaks the chain of trust, it also enables malware injected from an operating system level to be loaded on boot. Therefore, we have chosen to call the reboot method used in this paper *tepid reboots*.

One crucial goal of this paper is to make sure that memory is preserved during an ongoing restart, since a lengthy absence of electrical current will clear the memory content – due to the volatile nature of RAM. To make sure that the state of the memory image is unaffected by a system restart, a series of tests has been made for each tepid reboot method:

1. A known value is written to RAM, which is then found and compared in the memory image extracted by our program.
2. Another memory image has been created from a runtime environment, which is later compared to a memory image created by our program.
3. A series of memory regions is cryptographically hashed by an external program, which is compared by hashes generated from our program.

If these tests can be concluded as satisfactory for a reboot method, the given method is said to be a trusted entrypoint to the static chain of trust.

The results of these tests is discussed in the chapter *Validation*. Now, the chosen methods of tepid rebooting is presented and discussed in further detail.

4.2 Programmed reboot

A small snippet of C code, available in listing 4.1, is presented to showcase the simplicity of the implementation chosen in this paper. The objective performed by this program is to perform an immediate tepid reboot while keeping the memory footprint as small as possible. By not syncing the computers hard drives or performing shutdown procedures, the actual restart occurs as fast as possible – giving other software a very small time frame to be able to further touch and manipulate RAM content, especially since memory read or write access has not been locked by the software. Even though this limitation is sufficient enough to argue that the program is unusable in a forensic investigation, another major flaw is present. Namely, one can easily intercept the system call used in this program (reboot) and potentially alter the execution path in order to corrupt the current memory image or execute malicious code.

Therefore it is concluded that this method is deemed untrustworthy and should be avoided when performing the memory extraction methods presented in this paper.

Listing 4.1: reboot.c

```
1  #include <unistd.h>
2  #include <linux/reboot.h>
3
4
5  int
6  main(int argc, char** argv){
7    reboot(
8      LINUX_REBOOT_CMD_RESTART,
9      LINUX_REBOOT_CMD_CAD_ON
10   );
11   return 0;
12 }
```

4.3 System request magic

The system request key, or SysRq[1] as printed on the keyboard, used together with specific key combinations could also be used to perform a tepid reboot. To

[1] The SysRq key is shared with print screen button.

be able to utilize this feature the system kernel has to be compiled with the `CONFIG_MAGIC_SYSRQ` option, which often is the case in modern UNIX distributions [12]. Since this functionality is implemented as a part of the keyboard driver, it is in the most cases able to communicate directly with even a frozen system (unless the kernel itself is dead). To restart the system one simply uses the magic SysRq combination `<Alt + SysRq + b>`, which results in an immediate reboot.

Since this functionality is embedded in both the UNIX kernel and the keyboard driver, one has to subvert either one to be able to break the chain of trust. Having a chance of malicious subversion, we argue that the preferred reboot method is presented in section 4.4.

4.4 Reset switch

Finally, the use of the physical reset button to establish the trusted chain of events is presented. By simply pushing the reset button, a hardware instruction to reset the system is immediately sent to the motherboard. The restart can be viewed as a *cold reboot*, mainly because the system is performing a power on self test (POST) – even though the system is initiated from a previously powered state. Since the motherboard is powered during the whole procedure the same can be said about memory, therefore leaving the content intact during the operation [11]. By using this method one can be sure that the reset signal can not be intercepted by malicious software in an operating system. Hence, this method should be used when performing a forensic memory acquisition from a system.

4.5 Memory dumping

Since UEFI provides unobstructed access to the system memory, reading from a memory segment is trivial, especially since EDK2 provides an UEFI compatible port of the standard C libraries [14]. Therefore, one can abuse the power given in order to read and copy the memory simply by treating it as a consecutive array of bytes. This also results in the ability to choose where in memory the dumping shall begin and for how long it should read. Listing 4.2 shows a small snippet of code describing this ability.

Listing 4.2: umemdmp.c

```
1  /*Code ommitted*/
2
3  fwrite(
4      (VOID*)(UINTN)(StartAddress),
5      MemoryLength
6      1,
7      FilePointer
```

```
8     );
9
10 /*Code ommitted*/
```

The first argument to the C function `fwrite` is a pointer to a buffer address from where there desired data should be read from. In the example code above, a user specified variable `StartAddress` is used to tune where in memory the read shall start. This is doable simply because no boundary checking or memory restrictions occur in the UEFI environment. The second argument specifies the amount of bytes to read from the buffer, which is then written to the file descriptor pointed to by the fourth argument.

To invoke the software, it should either be run from a UEFI shell, often included by motherboard providers, or directly booted from a USB-device. More detailed guidelines and considerations are discussed in chapter 6

4.6 The static chain of trust

Here, the importance of the chain of trust model is explained. The purpose of this model is to maintain the integrity of system acquisition from start to finish. This acts as an extra layer of confidence for the end user, since each link can be verified and is dependent on the preceding stage to proceed execution. Since each link is relatively small and considered isolated, it reacts to external attempts of modification or interruption of the system. Each type of threat countermeasure is further explained in the description of each individual link below.

Figure 4.1: *The chain of trust illustrates the different stages of trust traversed during a forensic image caption.*

4.6.1 OS

A running systems operating system is considered to be the initial link in the chain of trust, since it maintains all RAM memory ready to be acquired by forensic personnel. The threats at this stage includes rootkits capable of contaminating the system RAM. These might fool traditional memory forensic toolkits by presenting falsified, modified or fabricated memory content when memory dumping is performed, since these toolkits requests memory via operating system API function calls. If a system is contaminated such function calls can be hooked in order to return such falsified data, or simply nothing at all [9].

Obviously, a potentially contaminated operating system can not be trusted when performing important memory extraction. This is why this node is considered only to be the offshoot into an sequence where trust can be enforced.

4.6.2 Tepid reboot

As previously discussed in section 4.1, *tepid rebooting* is the act of rebooting a running system without altering RAM memory. Here, the different levels of trust, that is achieved by the reboot methods, are explained in more depth in each following respective section.

Programmed reboot
This method is susceptible for signal hooking. An application with malicious intent can listen for and intercept the system call `reboot`, giving it the ability to remove traces of itself from the system memory. This makes the programmed reboot a less preferable method to proceed the traversal in the chain of trust.

System request magic
Although the system request method is preferable to programmed reboot, since it operates against an integrated kernel module and can not be trapped. In order to attack this functionality an attacker must replace either the kernel module or the keyboard driver with a custom one. This is considered a more sophisticated attack vector, however it is still doable given the sufficient resources like a root privilege.

Reset switch To perform this action, the user has to have access to a physical restart button on the computer. This might be problematic on some laptops and other devices that lacks the button in question. The power button does not perform a tepid reboot, since it cuts the power supply briefly, leaving the RAM content deteriorating over time – thus breaking one of the golden rules of computer forensics[2]. When the reset switch is pressed it instantly reboots the device without waiting for the system to make a clean shutdown, leaving the RAM content unaffected in its current state.

The preferred, and most trustworthy, method is using the reset switch when performing a tepid reboot.

4.6.3 UEFI

UEFI, as described by section 1.4, provides a set of databases in which the cryptographically generated certificates resides along with the Secure Boot feature.

[2] A forensically sound copy should not alter the original content, as described by Carrier et al. [10]

It forces an application to be considered trustworthy in order to run. For an application to be trusted, the certificate needs to be signed by a trusted part and stored in at least one of the allowing databases. These databases are:

Platform Key (PK)
This database contains the basis of the trust environment. The platform keys residing here is often held by the platform manufacturer, which are able to acknowledge and maintain trusted parties that wish to install their certificates in the other databases.

Key Exchange Key (KEK)
Keeps track of the trusted keys that can be used to change the content of the Allowed database.

Allowed
Here, the trusted certificates are stored that verifies the applications that wants to execute in the Secure Boot environment. If the applications certificate is present here and simultaneously not present in the forbidden database, the Secure Boot considers it authorized to run.

Forbidden
As the name suggests, entries stored here will deny all applications signed with these certificates, leaving them inoperable on a system using Secure Boot.

Because of the setup of the infrastructure, the trust established in this link ultimately relies on the vendor to be trustworthy with the important keys. In order for an attacker to infiltrate this link, one must either crack the cryptographic certificates, which is considered to be very time consuming, or get the hold of a leaked platform key to sign malicious applications.

4.6.4 Umemdmp

`Umemdmp`, being signed with sbsign allows the user to choose a certificate from a vendor the user finds trustworthy. If no vendors are deemed trustworthy enough the user may generate a set of keys and certificates. To further enforce the sense of trust, the source code can be read and inspected by anyone – since the code is open. UEFI software are allowed to import variables from an operating system, however to reduce the amount of attack vectors against the program [15], no such operating system defined variables are used or read. Hash sums are also generated beforehand for both the x86 and x64 versions of `Umemdmp`, as an extra dimension of program validity.

A bootkit poses as a threat, since it would bypass Secure Boot, as proven by Kallenberg et al., and act as an platform which could execute a modified version

of `Umemdmp`.

The user interaction with the `Umemdmp` application is straight forward, the program accepts the desired amount of bytes to extract as the first parameter and the start address as the second as shown in listing 4.3. This will produce a file named `dump`, in the same location as `Umemdmp`, containing the extracted memory data.

Listing 4.3: Umemdmp example showing how to extract 100 MiB starting from address 0x0.

```
Fs0:> Umemdmp.efi 104857600 0x0
Fs0:>
```

4.6.5 Storage device

In the concluding link of the chain of trust, the secondary storage device should be formatted and clean in order to prevent potential auto-loading operating system applications to alter the content of the memory dump.

4.7 Validation

In order to obtain and validate memory the process is as follows:

Algorithm 1 Pseudo code for Umemdmp

```
1:  NrOfBytes ← 1024
2:  Address ← 0x0
3:  Buffer ← 0
4:  OuputFile ← 0
5:  while Address ≤ MaxAddress do
6:      if Address = ReservedAddress then
7:          Address ← Address + NrOfBytes
8:          Continue loop
9:      else
10:         Read and save NrOfBytes bytes into Buffer starting at Address
11:         Read and save NrOfBytes bytes to OutputFile starting at Address
12:         if Buffer ≠ OutputFile then
13:             Report error
14:             Break
15:         end if
16:         Address ← Address + NrOfBytes
17:     end if
18: end while
```

The program reads 1 `KiB` blocks of memory at a time, which it stores in a temporary buffer and writes the same amount to file. A comparison between the buffer and file content is then done, if there is no difference the program continues to fetch the next block until the highest memory address has been reached. This test is done in order to assure that no data is being tampered with by our program.

To increase the level of assurance that the memory content is left unmodified, the third party UEFI tool `dmem`, provided by Tianocore, can be used to inspect the RAM. A possible change would be the use of a hash function for each block iteration, however this addition is not beneficial since `dmem` cannot hash memory segments and therefore no comparison can be made.

A known character sequence is written to RAM from the operating system environment, subsequently a tepid reboot is performed. If the known sequence is found within the memory image created by `Umemdmp`, hypothesis 3.2 is considered fulfilled. Also, the research questions from section 3.1 can be concluded with the method above.

Chapter 5

Results

This chapter covers the finds in regards to acquisition in a trusted way. The following topics provides the outcome for each respective research question and hypothesis. A broader analysis is given in the next chapter.

All results were obtained from a test machine using a Gigabyte GA-C1037UN-EU UEFI compatible motherboard. This model is however not equipped with the Secure Boot option. All tests related to secure signing are conducted in Tianocores `OVFM` environment with the `QEMU` processor emulator and emulated Secure Boot. The code, that targets the 64 bit architecture, is compiled and signed with a generated 2048 bits RSA key. This key is used in the creation of the X509 certificate needed to sign the application. This certificate is also enrolled in the KEK database. With Secure Boot enabled the signed version of `Umemdmp` is able to run while an unsigned version of `Umemdmp` is denied execution.

Once `Umemdmp` was signed and run it was evident that this process of extracting and storing memory content was very time consuming.

5.1 Looking for known signatures

By inserting a known signature from the operating system, consisting of 200 `MiB` worth of the character `A`. The reason to this large payload is not only to make the search less tedious, but more importantly, to be able to verify that the memory content does not deteriorate when tepid rebooting. `Umemdmp` was able to extract the entire signature being located in the midst of physical RAM. After the signature was discovered in the `Umemdmp` generated image it was verified with `dmem` to be a mirrored copy. The whole process was performed abiding by the chain of trust principle, thus completes the first research question to a satisfactory degree.

5.2 Complete memory imaging

The process of performing a full memory image recovery was found to be inconclusive, due to the amount of unknowns involved. For instance, some reserved

memory segments has to be identified and excluded from the imaging process. Because these areas differ ever so slightly between systems, a full memory dump can not be made by a general application. Furthermore, the resulting memory image can not be analyzed on another system, unless it is a replica of the system from where the memory was extracted, as discussed in section 1.5.

5.3 Resisting obstruction

Owing to the chain of trust model, the operating system is forced to surrender its potentially unreliable state of execution to a trusted state, starting with the tepid reboot. Using a non interceptable tepid reboot method[1], as discussed in section 4.1, no processes residing in the operating system can affect the transition from a unknown to a trusted state – given that the UEFI variables are correctly handled [15]. When in the UEFI state, according to the Secure Boot philosophy, no unsigned software can be run.

5.4 Regarding the hypothesis

No difference has been identified between M_0 and M_1, thus the H_0 hypothesis is rejected leaving H_1 favored. Therefore, M_0 and M_1 is concluded to be identical and `Umemdmp` can be used in forensic investigations.

[1] Concluded to be the hardware reset button

Chapter 6

Analysis

The trust concept discussed throughout this paper suggests methods that does observe several known and interlinked states. Each state is dependent on a previous one and therefore give rise to a transitive relation between the initial state and the end state. This behavior has been closely examined in 4.6, *The static chain of trust*. The initial state, here presented as the transition between an operating system and the pre-boot environment[1], is required to be very reliable in order to bootstrap and keep the transitive trust relationship intact. If the initial state can not meet a satisfactory reliance, i.e the user can not perform a recommended variant of *tepid reboot*, the previously mentioned relationship can not be established. However, even with a successful initiation the UEFI platform can be vulnerable to exploitation, as demonstrated by Kallenberg et al. [15], due to vendor misconfiguration. This type of mishandling naturally breaks the chain of trust. It can be concluded that, although sound in theory, the trust model presented is difficult to uphold because all involved parties have to actively strive to achieve a sense of verifiability. Trust is ultimately not definitive in a measurable way but instead is built on the users faith in external service providers.

6.1 Memory dumping

In regards to the results presented in the previous chapter, `Umemdmp` is capable of extracting fragments of a memory image not related to some reserved areas. While conducting the acquisition experiments the system became unresponsive when certain memory areas were accessed. Herein lies the dangers in making a general tool for this purpose since system hangups of this severity might cause the inability to obtain the desired data. Clever software could target these memory segments in order to conceal data that could be important in a forensic investigation.

A general problem in analyzing obtained memory images is that they are dependent on the origin system setup, such as architecture, kernel version and so on. Even state of the art forensic toolkits such as Volatility all suffer from this

[1] The UEFI environment bundled with an enabled Secure Boot.

debilitating dilemma. The consequences of this is that in order fully be able to perform a memory analysis, the investigator need to use the source system or an exact copy of it. Since the target system might be untrustworthy or subverted, interference with the ongoing investigation is a potential threat. The analysis can be performed on a mirrored system[2] if there is a suspicion of the target system being contaminated.

The bottle neck in relation to memory extraction time is thought to be caused by the loaded UEFI drivers, defaulting to the most primitive and backward compatible storage device driver unless explicitly specified in code. This topic is briefly discussed in the future work section.

[2]The mirrored system is equivalent to the target system and equipped with the same hardware.

Chapter 7

Conclusions and Future Work

This chapter presents a summary of the proposed methods for utilizing a trusted ground for memory acquisition using the UEFI environment. In the section for future work, a brief discussion of how to further improve `Umemdmp` is found.

7.1 Conclusion

In this paper, we presented *the static chain of trust* as a good model for transferring trust and assuring that the desired course of execution is being taken. This ensures the user that the correct software is run. Trust is however not without its complications, in the end, it is up to the user to decide whether a third part is deemed reliable or not. As addressed in previous chapters, mistakes made by trusted parties can still occur and if they are severe enough it affects the entire chain of trust, leaving the system more difficult to validate. This is especially serious if there is a strong suspicion that the system is already contaminated. In order to acquire a forensically valid memory image, there must not be any fabrication or modification involved in the process. The proposed solution `Umemdmp` accomplishes this while honoring the chain of trust.

7.2 Future Work

The state of `Umemdmp` can be improved by the use of *monces*[1], as introduced by Müller et al. [18], in order to further strengthen the credibility of the application. A hash sum could also be generated and displayed during program startup to provide a sense of integrity, this however, will not improve the chain of trust since bootkits can easily fool the user by mimicking this procedure and present the expected checksum. `Umemdmp` should make sure that the target secondary storage media is formatted before saving a memory image to it, in order to prevent potentially hostile auto loading software to interfere with an analysis. In the current version of `Umemdmp` the binary is considered a UEFI-application and runs accordingly. To have it start in an earlier stage of the UEFI boot sequence, as

[1] Authentication messages intended to be used once.

described in chapter 1.4, a conversion into an UEFI-driver should be considered. This is appropriate in order to minimize the risk that another running driver or application interferes with `Umemdmp`.

As stated in the results and analysis sections the performance of the solution with respect to time consumption needs to be addressed by i.e. investigating if old USB drivers are the cause. The UEFI GetMemoryMap() function can be utilized to identify reserved peripheral memory addresses in order to avoid these regions, because of the unexpected behavior occurring when reading them.

References

[1] Unified Extensible Firmware Interface Specification, Version 2.4 Errata A, December, 2013
[2] Analysis of the building blocks and attack vectors associated with the Unified Extensible Firmware Interface (UEFI), Jean-François Agneessens, Manuel Humberto Santander Pelaez , January 27th, 2014
[3] On the Practicability of Cold Boot Attacks, Michael Gruhn and Tilo Müller, 2013
[4] UEFI Secure boot in modern computer security solutions, Richard Wilkins and Brian Richardson, September 2013
[5] UEFI Networking and pre-os security, Intel technology journal volume 15, issue 1, 2011
[6] The Volatility framework. In code.google.com. Retrieved February 27, 2014, from https://code.google.com/p/volatility/
[7] Memtest86. In memtest.org. Retrieved May 2, 2014, from http://www.memtest.org/
[8] Ring (computer security). In Wikipedia. Retrieved February 27, 2014, from http://en.wikipedia.org/wiki/Ring_%28computer_security%29
[9] Anti-forensic resilient memory acquisition, Johannes Stüttgen , Michael Cohen, Digital Investigation 10 (2013) S105–S115
[10] A Hardware-Based Memory Acquisition Procedure for Digital Investigations, Brian D. Carrier and Joe Grand, 2004
[11] Reboot (computing). In Wikipedia. Retrieved May 3, 2014, from http://en.wikipedia.org/wiki/Reboot_%28computing%29#Cold_vs._warm_reboot
[12] http://linuxgazette.net/issue81/vikas.html
[13] Security in Computing, Fourth Edition, harles P. Pfleeger and Shari Lawrence Pfleeger, 2006
[14] http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=EDK2
[15] Setup For Failure: Defeating Secure Boot, Corey Kallenberg, Sam Cornwell, Xeno Kovah and John Butterworth, 2014
[16] Hypervisor Memory Forensics, Mariano Graziano, Andrea Lanzi, and Davide Balzarotti, 2013
[17] Linux device drivers 2nd edition, Alessandro Rubini and Jonathan Corbet, 2001
[18] Stark Tamperproof Authentication to Resist Keylogging, Tilo Müller, Hans

Spath, Richard Mäckl, and Felix C. Freiling, 2013
[19] Tianocore EDK2. In sourceforge.net. Retrieved May 13, 2014, from
http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=EDK2
[20] Secure Boot. In wiki.ubuntu.com. Retrieved May 16, 2014, from
https://wiki.ubuntu.com/SecurityTeam/SecureBoot