Exposing Bootkits with BIOS Emulation

Lars Haukli

Blue Coat Systems
lars.haukli@bluecoat.com

July 28, 2014

Abstract

The security features added in modern 64-bit versions of Windows raise the bar for kernel mode rootkits. The introduction of Driver Signature Enforcement prevents malware from loading an unsigned kernel mode driver. PatchGuard was introduced to protect the integrity of the running kernel, in order to prevent rootkits from modifying critical structures or hooking system calls. Although time has shown that these security measures are not perfect, and may in fact be bypassed while actively running, an alternative approach is to subvert the system by running code before any of the security features kick in.

Secure Boot has been introduced to protect the integrity of the boot process. However, the model only works when booting from signed firmware (UEFI). Legacy BIOS systems are still vulnerable. The Master Boot Record, Volume Boot Record, and the bootstrap code all reside in unsigned sectors on disk, with no security features in place to protect them from modification.

Using a combination of low level anti-rootkit techniques, emulation, and heuristic detection logic, we have devised a way to detect anomalies in the boot sectors for the purpose of detecting the presence of bootkits.

1 Introduction

Rootkit authors are facing new challenges when trying to intrude into kernel mode on modern versions of Windows. In order to load a kernel mode driver, which is the most straightforward way of running code in the kernel, they have to overcome Driver Signature Enforcement. It should generally not be possible for them to obtain a valid digital signature on their modules, so they will need to get around this defense mechanism in some other way. Even if they manage to run code in kernel mode, they are still facing the challenge of bypassing Patch Guard [1,2]. This complicates what they are trying to achieve, namely to make changes in order to hide something.

The first kernel mode rootkit that managed to successfully compromise 64-bit Windows platforms was TDL4 [3]. It did so by manipulating the boot sectors. Hence, we classify it as a bootkit. During the boot process it would modify the memory contents of a dll that is loaded by the kernel, so that it could eventually load its unsigned driver. The chain of events leading up to this would not be possible without the very first step which involved modifying the boot sectors[1].

TDL4 turned out to still be detectable. The author has used several techniques to detect its presence in the past, including a very generic technique known as *cross view detection*. This detection technique is based on the observation that TDL4, just like most bootkits, would fake the contents of the boot sectors when they were read, by presenting the original contents instead of what was actually there. If you could manage to get around TDL4's modification of low level disk I/O, you could read the contents of the boot sectors in multiple ways. Discrepancies could then be detected by comparing the results of various read operations.

It is uncertain, but possible, that the authors of Rovnix [4, 5], also known as Cidox [6], chose a different approach because they realized that such detection mechanisms were widely used. In their bootkit, they did not opt for faking the contents of what they had changed in the boot sectors, but instead relied on polymorphic techniques for obfuscation. This is a well known approach to countering signature-based detections, but rarely seen applied to the 16-bit code within the boot sectors. When analyzing samples of this malware, we began to realize that it was time to explore new ways of generically detecting bootkits.

Targeting the boot sectors is in no way a new approach taken by malware. Actually, it dates back to the very first documented computer virus incidents of Elk Cloner, targeting Apple II, and Brain, targeting the IBM PC [7]. This was in the 80s.

Still, legacy BIOS is still in use today, so it would seem that old habits die hard. Anyone designing security solutions will know the challenges related to backwards compatibility. Secure Boot [8, 9], it would seem, is no exception. As long as there are no security mechanisms in place to protect the integrity of the very first parts of the code that is being run during the boot process, systems booting from a legacy BIOS will inherently be vulnerable. Recent reports indicate that bootkits are still widely used for purposes including conducting targeted attacks, and are in demand on the black market [10].

In this paper, we will explore how to analyze the behaviour of the code residing in the boot sectors, and show how we can detect bootkits by looking for anomalies. We analyze the boot code by emulating it, using a custom, emulated BIOS, that incorporates anti-rootkit techniques to bypass potential hooks which may prevent us from reading the true contents of the raw sectors on disk. We also show how to use this approach for disabling bootkits by breaking their load chain.

This paper is organized as follows. Section 2 explains the early parts of the

[1]In other words, it did not support UEFI systems

BIOS boot process leading up to the execution of bootmgr. In Section 3 we dive into the details of how to read raw sectors on disk by interfacing with the lowest level disk driver. We also cover how to bypass hooks at this level. Section 4 explains how to create the emulation environment in which we aim to analyze the code residing in the boot sectors, including emulating the BIOS. In Section 5 we discuss the anomalies in the behaviour of the boot code that we base our detection on. Section 6 covers how to regain control of compromised systems by retrieving the original contents of the boot sectors to break load chains. Finally, in Section 7, we discuss challenges with our approach when running on systems with non-standard boot loaders, before concluding the paper in Section 8.

2 BIOS Boot Sequence

Legacy BIOS systems will typically perform a Power-on self-test (POST) during their pre-boot sequence. The last action taken by the POST is to issue BIOS interrupt 19h, often referred to as the Bootstrap Loader Service. This will instruct the BIOS to attempt to load the sector at Logical Block Address (LBA) 0, or following the Cylinder-Head-Sector convention (CHS), head 0, cylinder 0, sector 1, into memory at address 0:7C00h[2], and transfer control there. The sector at LBA 0 is known as the Master Boot Record (MBR), and will be the starting point for our analysis of the boot process.

The MBR consists of both code and data [11]. The layout of this sector is defined in Listing 1. At this early point in the boot process, the CPU is running in 16-bit real mode. As the BIOS transfers control to the first byte of the MBR upon issuing interrupt 19h, this code assumes that its first instruction is being executed with the code segment set to 0, and the instruction pointer set to 7C00h. As this code is restricted to 440 bytes, it is both compact and very limited in functionality. The very first part of the code simply copies the main logic to a different memory location, so that it can use the memory range starting at 7C00h for later stages. It then transfers control to the main logic, now residing at this new memory location, which parses the partition table looking for the partition which is set to be active. If found, the sector holding the active partion, known as the NTFS Volume Boot Record, or simply VBR, is loaded into memory at address 0:7C00h, thus overwriting the previous content at this location. The partition table's layout is defined in Listing 2. It holds the information that the boot code needs to determine which parameters to pass to BIOS interrupt 13h, which is the routine responsible for loading sectors on disk into memory. The boot sequence continues at address 0:7C00h, this time holding the contents of the VBR, which is responsible for locating and loading the signed executable bootmgr into memory.

[2]This is a segmented memory address meaning that code segment is set to 0, and the instruction pointer is set to 7C00h

```
typedef struct _MBR
{
    BYTE      bootCode[440];
    DWORD     diskSignature;
    WORD      reserved;
    PARTITION partitionTable[4];
    WORD      sectorEndSignature;

} MBR;
```

Listing 1: Master Boot Record (MBR)

```
typedef struct _PARTITION
{
    BYTE  bootIndicator;
    BYTE  head;
    BYTE  sector;
    BYTE  cylinder;
    BYTE  type;
    BYTE  lastHead;
    BYTE  lastSector;
    BYTE  lastCylinder;
    DWORD relativeSector;
    DWORD numberSectors;

} PARTITION;
```

Listing 2: Partition

The Volume Boot Record, just like the Master Boot Record, contains both code and data [11]. Its layout is defined in Listing 3. This code also expects to be executed at 0:7C00h, but this time the first instruction is a simple jump to the main logic residing at the end of the structure. This structure includes the BIOS Parameter Block, holding information on the boot disk. Unless the field *hiddenSectors* dictates otherwise, the remainder of the boot code will reside in the 15 sectors[3] directly following the VBR. The hiddenSectors field simply indicates how many sectors on disk to skip before the boot code should expect to find the rest of the code. These sectors, commonly referred to as the Initial Program Loader (IPL), contain code only, and is defined in Listing 4. The IPL sectors will be loaded directly following the VBR in memory.

The purpose of the VBR, together with the 15 sectors of IPL, is to parse the NTFS file system to locate bootmgr. Once this executable is found, it is loaded into memory, and execution is transferred to its first byte. This executable, just like ntldr on Windows XP, is responsible for loading the OS [12], meaning the kernel and boot drivers[4]. However, it is a signed executable on disk, so from this point in the boot process, the security model of Secure Boot should work. Our focus, for the purpose of exposing bootkits, will be the behaviour of the

[3]This is the normal size of the IPL

[4]On 64-bit versions of Windows, bootmgr will actually load an executable named winload.exe, which will load the core OS modules, but as this executable is digitally signed as well, it makes no difference to our analysis

code starting at the first instruction of the MBR, up to the point where the first instruction of bootmgr is executed, as all of these sectors may be modified by an intruder. Figure 1 illustrates how we expect this process to behave.

```
typedef struct _NTFS_VOLUME_BOOT_RECORD
{
    BYTE jumpInstruction[3];
    BYTE oemID[4];
    BYTE dummy[4];

    // BIOS Parameter Block
    struct NTFS_BPB
    {
        WORD      bytesPerSector;
        BYTE      sectorsPerCluster;
        WORD      reservedSectors;
        BYTE      fatCopies;
        WORD      rootDirEntries;
        WORD      smallSectors;
        BYTE      mediaDescriptor;
        WORD      sectorsPerFAT;
        WORD      sectorsPerTrack;
        WORD      numberOfHeads;
        DWORD     hiddenSectors;
        DWORD     largeSectors;
        DWORD     reserved;
        ULONGLONG totalSectors;
        ULONGLONG MFTLogicalClusterNumber;
        ULONGLONG MirrorLogicalClusterNumber;
        DWORD     clustersPerMFTRecord;
        DWORD     clustersPerindexRecord;
        ULONGLONG volumeSerial;
        DWORD     checksum;

    } bpb;

    BYTE bootStrapCode[426];
    WORD sectorEndSignature;

} NTFS_VOLUME_BOOT_RECORD;
```

Listing 3: NTFS Volume Boot Record (VBR)

```
typedef struct _IPL_SECTOR
{
    BYTE iplCode[512];

} IPL_SECTOR;
```

Listing 4: Initial Program Loader Sector

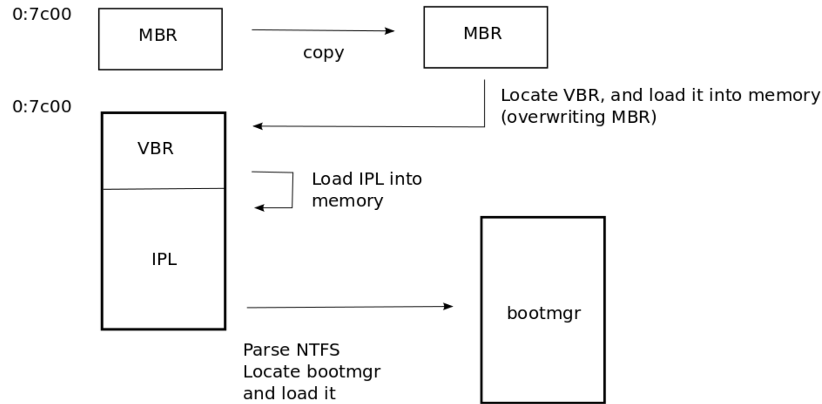

Figure 1: The early parts of the BIOS boot process

3 Interfacing with the disk

Rootkits being rootkits have a tendency of hiding contents on disk. For bootkits, it is quite common to fake any sector they have modified, in order to fool someone inspecting the boot sectors into believing that everything is fine. They might also incorporate techniques to complicate removal of the rootkit component, which for bootkits normally means complicating the process of writing to the boot sectors. Before attempting to emulate the boot process, we need to ensure that we are reading the true contents of the boot sectors.

Figure 2 shows the Windows Storage Stack [12]. It is an illustration of the various subsystems that a request to read contents on disk will go through. It does not matter whether the request is initiated from user mode, via NtReadFile in ntdll, anything layered above it, or from kernel mode via e.g. ZwReadFile. All requests will end up calling the function registered as NtReadFile within the System Service Descriptor Table (SSDT)[5]. From this point on, the requests will proceed as I/O Requests Packets (IRP) through several subsystems which are implemented by various kernel mode drivers. Eventually, it will end up in the disk subsystem, which sits on top of the Hardware Abstraction Layer (HAL).

[5]The SSDT is a table of function pointers to internal routines implemented within the kernel, and has been a popular place to install hooks over the years. Modifications of the SSDT is however protected by PatchGuard on 64-bit versions of Windows.

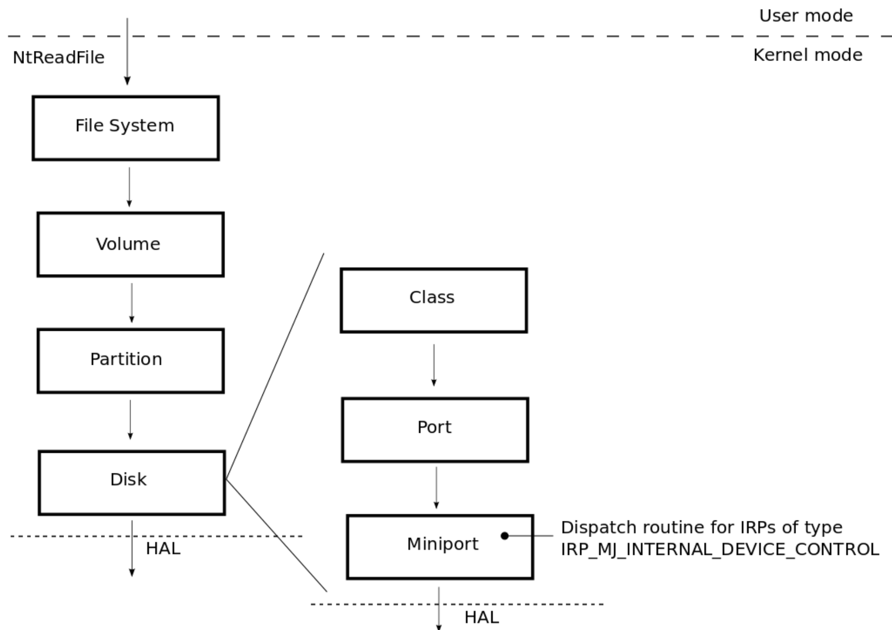

Figure 2: The Windows Storage Stack

The disk subsystem is conceptually divided into three parts[6], where the miniport driver is responsible for implementing hardware specific logic to read and write to raw sectors on disk. Both read and write requests will eventually manifest themselves as IRPs passed down to the miniport driver. Such IRPs will have their I/O control codes set to IRP_MJ_INTERNAL_DEVICE_CONTROL[7], and are expected to be processed by the dispatch routine registered to handle IRPs of this type. When the miniport driver is loaded, its initialization routine, DriverEntry, will initialize a driver object structure [14], and populate the corresponding elements of the *MajorFunction* array with entry points to the dispatch routines of its supported IRP types. This means that, if we can obtain the driver object of the miniport driver, we can retrieve a function pointer to a routine that implements reading and writing to raw sectors on disk from it.

Interfacing with the disk at this level means that we do not need to worry about hooks that might be present in the layers above. It also means that we don't have to implement hardware specific logic ourselves, which may become somewhat cumbersome depending on the types of disks we want to support.

An example of an alternative approach for reading raw sectors on disk is given in Appendix A. This code reads the MBR using Programmable I/O (PIO) [15, 16]. The advantage of using a low level approach such as this, is

[6]This does not necessarily mean that it is implemented in three distinct drivers
[7]IRP_MJ_INTERNAL_DEVICE_CONTROL is a constant defined to be 15

that it is highly unlikely that a rootkit will ever be able to fake what you read. The major drawback is that the approach will not work for all disks.

3.1 Locating the miniport driver

The first challenge is to obtain the device object [17] of the miniport driver related to the boot disk. The device object includes information on the driver's location within the driver stack, and also holds a pointer to the corresponding driver object. We will be needing this device object when we later attempt to call one of its dispatch routines.

The system we are running on might have several disks, so we need to determine which disk the system is booting from. Every disk has a unique disk number. If we can obtain the disk number of the boot disk, we can uniquely identify the disk we attempting to read the MBR from.

One approach for obtaining this is to make use of the GetSystemDirectory routine, and then retrieve the drive letter from that, assuming that the system boots from the same volume as it is installed onto. If the drive letter is *C*, we may open a handle to the volume named \\.*C:*. Using this handle, we can call the DeviceIoControl routine to retrieve the volume disk extents structure [18] related to it. This structure holds a disk extent structure for every disk that makes up the volume, which includes the disk number. Assuming that the system will boot from the first disk within the volume, we retrieve the disk number from the first element in the array.

Using the disk number, we may now retrieve the boot disk's device object. A way to do this is to start by opening a handle to either *Device**HarddiskX**Partition0* or *Device**HarddiskX**DR0*, where X is the disk number. Do this in kernel mode, and then use the ObReferenceObjectByHandle routine to retrieve the corresponding file object [19]. This structure contains the device object corresponding to the highest level disk driver, i.e. the driver on top of the disk driver stack within the disk subsystem as illustrated in Figure 2. We aim to interface with the miniport driver, which is the lowest level driver. Whenever an IRP is to be passed from one driver on the stack to the one just below it, the I/O manager needs to know where to pass it to. This information is kept within the device objects; specifically, it is kept within the *DeviceObjectExtension* member. This member is defined in Listing 5.

```
typedef struct _DEVOBJ_EXTENSION
{
    SHORT          Type;
    SHORT          Size;
    PDEVICE_OBJECT DeviceObject;
    ULONG          PowerFlags;
    PVOID          Dope;
    ULONG          ExtensionFlags;
    PVOID          DeviceNode;
    PDEVICE_OBJECT AttachedTo;
    LONG           StartIoCount;
    LONG           StartIoKey;
    ULONG          StartIoFlags;
    PVPB           Vpb;

} DEVOBJ_EXTENSION;
```

Listing 5: Device Object Extention

The field *AttachedTo* holds the device object of the next disk driver in the stack, i.e. the disk driver that sits just below it. For the disk driver at the very bottom of the stack, this field is *null*. Hence, in order to retrieve the device object of the lowest level disk driver, we simply walk the chain until we reach the end of what is essentially a null-terminated linked list.

At this point, it might be tempting to retrieve the function pointer of the dispatch routine, as this is trivially obtained from the device object by first retrieving its driver object, and then inspecting the MajorFunction array. The problem with this approach, is that this has become a popular place for rootkits to install hooks. Thus, we might still end up being tricked by an active rootkit component into reading faked content. When designing anti-malware software that is expected to run on potentially compromised systems, it is generally a good idea to be paranoid, so we need to obtain the function pointer of the dispatch routine from a trustworthy source. Furthermore, even after obtaining the correct location of the miniport driver's dispatch routine, we also need to ensure that it has not been inline hooked[8].

3.2 Overcoming function pointer hooks

Whereas unprotected memory areas are susceptible to manipulation, signed executables on disk are generally not. As the dispatch routine we are seeking obviously resides within an executable on disk, we will be using its signed contents as our trustworthy source of information. If we can find the miniport driver's dispatch routine within its executable image on disk, we can then work out where it should reside in memory. This would provide us with the original function pointer which may have been replaced within the miniport driver's driver object in memory, and thus give us a chance to bypass function pointer hooks within the MajorFunction array. Additionally, this will also expose the

[8]An *inline hook* is an alternative way to hook a routine by replacing some of its contents with an instruction that transfers control to the hook destination

expected contents of the dispatch routine, which we can use to bypass inline hooks. Figure 3 shows a simplified example of a DriverEntry routine.

```
NTSTATUS DriverEntry(__in DRIVER_OBJECT *pDriverObject, __in UNICODE_STRING *pRegistryPath)
{
    // ...

    // Set dispatch routines
    pDriverObject->MajorFunction[IRP_MJ_CREATE]                  = Dispatch_Dummy;
    pDriverObject->MajorFunction[IRP_MJ_CLOSE]                   = Dispatch_Dummy;
    pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]          = Dispatch_DeviceControl;
    pDriverObject->MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL] = Dispatch_InternalDeviceControl;
    pDriverObject->MajorFunction[IRP_MJ_PNP]                     = Dispatch_PnP;
    pDriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL]          = Dispatch_SystemControl;
    pDriverObject->MajorFunction[IRP_MJ_POWER]                   = Dispatch_Power;

    // ...

    return STATUS_SUCCESS;
}
```

Figure 3: A simplified example of a DriverEntry routine

The DriverEntry routine is the entry point of the executable, and will be called whenever the driver is loaded. During initialization, it will set up its dispatch routines within its driver object's MajorFunction array, so that the I/O Manager can locate them when it needs to. While the DriverEntry routine is the entry point of the executable, and thus also exported, the dispatch routines are internal, non-exported routines within the image. Because of this, we need to come up with a way to locate the dispatch routines, and specifically the dispatch routine responsible for handling internal device control requests, within the driver's executable.

One approach for doing this is to rely on *recursive disassembly*. If we manage to find the instructions that initialize the MajorFunction array, then we may retrieve the location of the dispatch routines from there. As these instructions may reside within a subroutine called from the DriverEntry routine, we have to apply the disassembly approach recursively as we locate subroutines.

Figure 4 shows an example of the code that we are looking for. It is obtained by disassembling a driver that sets up dispatch routines. If you study its logic, you will see that it resembles that of Figure 3.

In this case, we see that the address of the routine named *Dispatch_InternalDeviceControl* is loaded into *rax*, and then moved to offset *E8h* within the structure pointed to by *rsi*. Although not shown in the disassembly, *rsi* is actually pointing to the start of the driver object structure. If you study the layout of the 64-bit version of the driver object structure, you will find that the dispatch routine for internal device control requests is stored at this exact offset. Hence, if we retrieve the address of this routine from the disassembled code, we can compute the actual location we are searching for.

It turns out that this logic is quite common amongst most drivers at this level. As they are all initializing the same structure, the offsets will be the same. Obviously, details such as which registers are used, and the layout of the instructions will vary. An interesting observation, however, is that for this driver

```
lea     rax, DriverUnload
mov     [rsi+68h], rax
lea     rax, Dispatch_InternalDeviceControl
xor     ecx, ecx
mov     [rsi+0E8h], rax ; Set IRP_MJ_INTERNAL_DEVICE_CONTROL
lea     rax, Dispatch_Dummy
mov     r8d, 'PedI'
mov     [rsi+70h], rax  ; Set IRP_MJ_CREATE
mov     [rsi+80h], rax  ; Set IRP_MJ_CLOSE
lea     rax, Dispatch_DeviceControl
mov     [rsi+0E0h], rax ; Set IRP_MJ_DEVICE_CONTROL
lea     rax, Dispatch_Power
mov     [rsi+120h], rax ; Set IRP_MJ_POWER
lea     rax, Dispatch_PnP
mov     [rsi+148h], rax ; Set IRP_MJ_PNP
lea     rax, Dispatch_SystemControl
mov     [rsi+128h], rax ; Set IRP_MJ_SYSTEM_CONTROL
```

Figure 4: Disassembled driver code setting up dispatch routines

type, they all seem to register dispatch routines for at least four types of requests: *Power*, *PnP*, *Device Control*, and *Internal Device Control*. Additionally, they all register a *DriverUnload* routine to be called whenever the driver is unloaded. This observation is critical to the success of our approach described shortly.

As we cannot make assumptions of the whereabouts of the instructions we are looking for, we need to analyze entire routines. This is also true for any subroutine we come across. Determining the end of any disassembled routine is a non-trivial task, but this is outside the scope of this paper[9], so let us simply assume that we can at least get close to analyzing entire routines.

Our approach is to search for instructions of interest that modify memory. These instructions should follow the format: *mov [reg + offset], dispatch_routine*, where *reg* is any register, and *dispatch_routine* is anything that could hold the address of a routine. In scenarios such as the one illustrated in Figure 4, where the address is kept in a register, we will need to know its value at that point. This means that we will have to store the state of all interesting registers as we progress with the disassembly of each routine, so that we can keep track of instructions such as *lea rax, dispatch_routine*.

The next step is to find the address of interest, namely that of the dispatch routine for internal device control requests. Simply looking for the pattern *mov [reg + E8h], dispatch_routine* will not cut it, as it is too prone to causing false positives: the offset *E8h* could very well be used for all sorts of other structures. This is where our critical observation comes into play. Instead of looking for this offset exclusively, we make the assumption that all five offsets[10] will occur

[9] We are disassembling a single routine multiple times in order to achieve this. Routines may have several return instructions, and code that comes after a return instruction may still belong to the routine

[10] Note that *DriverUnload* also has an offset in the driver object structure

within the same routine. In this way, we analyze entire routines, and collect the addresses of all five routines in the process. When we have reached the end of a routine, we check if we have successfully obtained the address for each of the five, and if this is the case, we treat the address corresponding to the offset of internal device control as the routine we are looking for.

As it turns out, not all drivers actually initialize their dispatch routines within the MajorFunction array themselves. On 64-bit versions of Windows, it is actually quite common to call an external driver during initialization, and let this driver do the job instead. For us, this means that we will not always find the dispatch routine we are looking for by simply looking at the miniport driver we are analyzing. Sometimes, we need to extend our analysis to include a second driver. The usual suspects are *storport.sys*, *ataport.sys*, and *scsiport.sys*.

These three drivers each export an initialization routine named *StorPortInitialize*, *AtaPortInitialize*, or *ScsiPortInitialize* respectively. Because of the way these routines are named, it becomes a trivial task to recognize them. During disassembly, we look for calls to imported routines. For each one we find, we use simple string comparison to check if they follow the format **PortInitialize*[11]. If we find such a call, we determine which driver they are imported from, and then perform the exact same analysis on this driver, starting from the exported routine.

At this point, regardless of which driver implements the dispatch routine, we should have found its location within the executable image on disk, meaning we have obtained its Relative Virtual Address (RVA). From this, we can trivially obtain its Virtual Address (VA) by adding the RVA to the base address of the module it should reside in, i.e. the address where the driver is loaded in memory. This will yield the location of the dispatch routine we would expect to find within the miniport driver's driver object, and if this address differs from what we find in the MajorFunction array in memory, we already know that something is amiss.

Even if the function pointer itself has not been modified, inline hooks could still exist within the routine. In the next section, we will discuss how to detect and bypass such code modifications.

3.3 Overcoming inline hooks

Inline hooking is a popular alternative to function pointer hooking. With this approach, rootkits hot patch a routine in memory with a branch instruction that will transfer control to the hook destination. Normally, the first few bytes of a routine are replaced. Even though this technique serves the exact same purpose as any other hook mechanism, detecting and bypassing such hooks becomes slighly more involved.

The first step to overcoming inline hooks is to detect them. Our approach for doing this is to analyze the routine within the image on disk, and compare

[11] We can ignore caps in this string comparison

it to the corresponding memory contents[12]. If we find a discrepancy, we will need to construct a mechanism that will enable us to call the original routine in a way that avoids control being transferred to the hook destination.

Branch instructions in the original routine are not only problematic to us, but for the rootkits as well. Placing inline hooks after a branch instruction may lead to the hook destination not being reached, which is why you will see that such hooks are normally placed at the very start of a routine. In the discussion that follows, let us assume that this is the case.

Our approach to bypassing inline hooks is to construct a trampoline [20] that will consist of the original instructions retrieved from the image on disk, followed by a branch instruction that will transfer control to the next instruction in the routine, directly following the modified contents. This will effectively bypass the inline hook.

As we detect the presence of an inline hook, we take note of how many bytes have been patched. When constructing a trampoline to bypass the inline hook, we need to steal instructions from the original routine. In this process, it is vital that we ensure that these instructions are valid. The patch performed by the rootkit will often span several instructions, and may even cut an instruction in half. Hence, we need to resort to disassembly.

We treat the number of modified bytes as the minimum of what we need to steal from the original routine. Then, we disassemble the routine on disk to figure out how many instructions we need to incorporate into our trampoline. It does not matter if we end up stealing a few more bytes than what was patched, as long as we make sure we are stealing bytes that make up valid instructions.

Directly following our stolen instructions, our trampoline needs to have a branch instruction that will transfer control to the next whole instruction following the patched bytes. On 64-bit platforms, a simple *jmp* instruction will often not work as it will branch to a location relative to where it comes from, and this difference needs to be expressed as a 32-bit number. There are several ways to do this [21], but the author prefers to use a *push low_dword*, followed by a *mov [rsp + 4], high_dword* and a *ret* instruction, that enables specifying the entire 64-bit address of the target location, without modifying any registers.

3.4 Interfacing with the dispatch routine

All dispatch routines take a device object and an IRP as parameters. We have already obtained the first parameter, and will explain how to set up the second one in this section.

The miniport driver expects to process an IRP that is passed down from higher level drivers. It also assumes that the IRP should be passed up the chain again once the reading or writing operation has completed. We would like to act as though we are sitting just on top of this driver in the chain, without passing it back up. The I/O Manager will normally create the IRPs for regular I/O

[12] When comparing contents on disk with contents in memory, care must be taken regarding relocations

operations [12], and is responsible for passing them back up the chain whenever drivers call the IoCompleteRequest routine.

As we are imitating the higher level driver, we need to allocate and set up an IRP so that the miniport driver knows what to do. This includes an I/O Stack Location which will hold the actual parameters with instructions on what to read or write. It also includes an IoCompletion routine, that will be called by the I/O Manager when processing has completed. We use the latter as a way of passing control back to us after the miniport driver has done its work. The completion routine will also destroy the IRP.

Within its I/O Stack Location, the miniport driver expects to find a SCSI Request Block (SRB) [13]. This structure holds various data on the disk, such as Path ID, Target ID, Lun[13], and a few other less interesting details on the request. The important part is stored in the Command Descriptor Block (CDB) [22], at the very end of the SRB structure.

The size and layout of the CDB varies according to what type of command we are issuing. The two most common commands for reading raw sectors are *READ (10)* and *READ (16)*. The former is typically used for disks where 32 bits are sufficient to describe each possible value of LBA. For disks that have more than 2^{32} sectors, we must use the latter. The former command is described in Listing 6.

```
typedef struct _SCSI_CDB_READ_10
{
    UCHAR  operationCode; // set to 0x28 for read, or 0x2A for write
    UCHAR  options;
    ULONG  logicalBlockAddress;
    UCHAR  reserved;
    USHORT transferLength;
    UCHAR  control;

} SCSI_CDB_READ_10;
```

Listing 6: Expected layout of CDB for SCSI command READ (10)

In order to query the capacity of the disk, we may use the *READ CAPACITY (10)* command as described in Listing 7. Larger disks will set the *Returned Logical Block Address* field in the *parameter data* to -1 to indicate that the number of blocks exceed the maximum value possible to specify with 32 bits.

```
typedef struct _SCSI_CDB_READ_CAPACITY_10
{
    UCHAR  operationCode;       // set to 0x25
    UCHAR  reserved;
    ULONG  logicalBlockAddress; // set to zero if the PMI bit is cleared
    USHORT reserved2;
    UCHAR  reserved3;           // bit 0 is PMI (Partial Medium Indicator)
    UCHAR  control;

} SCSI_CDB_READ_CAPACITY_10;
```

[13] These three fields are only applicable for SCSI devices, and may be set to *null* for ATA/IDE devices

```
typedef struct _SCSI_READ_CAPACITY_10_PARAMETER_DATA
{
    ULONG returnedLogicalBlockAddress; // number of logical blocks (sectors)
    ULONG blockLengthInBytes;          // bytes per sector

} SCSI_READ_CAPACITY_10_PARAMETER_DATA;
```

Listing 7: READ CAPACITY (10) and its parameter data

4 Emulating the boot sequence

Our aim is to analyze the very start of the BIOS boot sequence in order to look for anomalies in its behaviour. Within the MBR, VBR, and IPL, we will find 16-bit code intended to be run in real mode. Its purpose is to locate and load bootmgr, which is the executable responsible for loading the operating system. Once this executable is found, control is transferred to it, and the boot process continues to load the kernel and the boot drivers. As bootmgr is a signed executable on disk, it should not be possible to modify it. That is, if it is modified, we should be able to detect this by other means. Hence, we will analyze the boot sequence starting from the first instruction of the code in MBR, until it reaches the first instruction of bootmgr.

Even though the code we are analyzing is compact, its functionality is very limited. The logic boils down to finding and loading sectors on disk into memory. In order to do this, it relies on BIOS interrupts. The code will determine which sector to load, and then issue an interrupt 13h. This BIOS interrupt accepts various ways of specifying which sector to load, both in LBA and CHS form, and a destination address in memory to specify where to load it.

When attempting to emulate this code, we will not only need to emulate the 16-bit code itself. We will also need to emulate the BIOS interrupts. This does not mean that we need to emulate the entire functionality of the BIOS however: it will suffice to emulate those interrupts that are issued by the code we are analyzing. Standard boot loaders mostly rely on interrupt 13h. We are however running on a potentially compromised system, so we need to incorporate our anti-rootkit techniques for reading raw sectors on disk, as explained in Section 3, into our interrupt 13h handler.

We will start by setting up our emulation environment. Writing an emulator capable of emulating 16-bit code is outside the scope of this paper, so we will assume that we have one. This emulator will have its own separate memory space, which is isolated in the sense that it is only accessible to the code that we are emulating, and of course, the emulator itself. We will start by loading our BIOS into this memory.

4.1 BIOS emulation

For the purpose of emulating the functionality of the BIOS, we have written a custom, minimalistic BIOS in 16-bit assembly. We will load this BIOS into

emulator memory at F000:FC00[14]. Its entry point will be F000:FFF0, and this is where emulation will start. Since there is no hardware involved, we do not need to bother with the POST. Instead, we mimic a *warm start*, which is what normally happens when a system is rebooted. That is, we set the code segment to F000h, and the instruction pointer to FFF0h, and then start emulating. At this location, our BIOS has a *jmp* instruction that will invoke its initialization code[15].

Our BIOS aims to only fulfill some minimal requirements, as we are only interested in emulating the boot sequence up to a certain point. It starts out by setting up the stack. Then, it proceeds to initialize the Interrupt Vector Table (IVT). This structure serves the same purpose as the Interrupt Descriptor Table (IDT) in protected mode, which is to store up to 256 interrupt vectors that hold the code segment and instruction pointer of the various interrupt handlers. This IVT resides from address 0:0 to 0:3FF, as each element describing the location of the routine responsible for handling the interrupt takes 4 bytes. Whenever the code we are emulating issues an interrupt, the emulator will emulate this instruction by looking up the corresponding interrupt handler in the IVT, and then transfer control there. This means that, whenever we encounter an interrupt 13h in the code we are emulating, the emulator will set the code segment and instruction pointer to what is stored in the corresponding element within the IVT, effectively calling the interrupt handler responsible for loading sectors on disk into memory.

We register interrupt handlers for interrupts *10h*, *13h*, and *16h*, which are the routines for handling *video*, *disk I/O*, and *keyboard* services respectively. The rest of the IVT is populated with dummy routines that will simply issue an *iret* instruction causing emulation to continue, without any action taken, should they be called. Whenever an interrupt handler is called, the *ah* register is expected to hold a *function code*. This will tell the BIOS which service to perform. As an example, for interrupt 13h, setting the value of *ax* to 2 will instruct the BIOS to read from a sector (i.e. load it into memory), whereas setting the value to 3 will instruct it to write to a sector. We do not need to implement the functionality of every possible function code. Instead, for function codes that aren't normally used by the boot loader, we will simply pretend as if nothing happend and let emulation continue.

For the interrupt *10h* routine concerning video features, there are two function codes that we should include. Function code 3, *Get cursor position and shape*, and function code 15, *Get current video mode*. Boot loaders may call either of these two and expect to get some valid data back. We do not intend to support video output, we just aim to satisfy the majority of boot loaders to enable emulating them up until the point where they load bootmgr.

The reason why we choose to implement interrupt *16h* will be discussed in more detail in Section 7. For now, it will suffice to say that we would like to know if the boot loader expects keyboard input, as this will complicate the

[14]The BIOS normally resides at the end of the F000h segment

[15]This means that we are actually emulating the BIOS initialization code

emulation process.

The most import interrupt by far is interrupt 13h. We will have to support regular read and write function codes (2h and 3h), the extended read and write function codes (42h and 43h) intended for accessing larger disks, as well as the regular and extended version of retrieving drive parameters (8h and 48h). For function code 41h, *Check Extentions Present*, it is a good idea to indicate that they are, so that boot loaders may use the extended features if they require it.

Even though the parameters to these interrupts are somewhat obscure, they are well documented [23]. For the function codes retrieving disk parameters, we obtain the required information on the boot disk before we start emulation, so that we can indicate the size of the disk, and so on, in the format that the boot loader expects. When it comes to the function codes for reading from disk, we need to incorporate our anti-rootkit techniques from Section 3. As this operation is rather involved, we break out of the emulation loop at this point. The BIOS will indicate which sector to read to the emulator, and emulation will commence again once the reading operation has completed, that is, when the contents from disk has been written to emulator memory. As boot loaders should never need to write contents to disk, it is generally a good idea to ignore write requests. Figure 5 illustrates what happens when we encounter an interrupt 13h in the emulated boot loader.

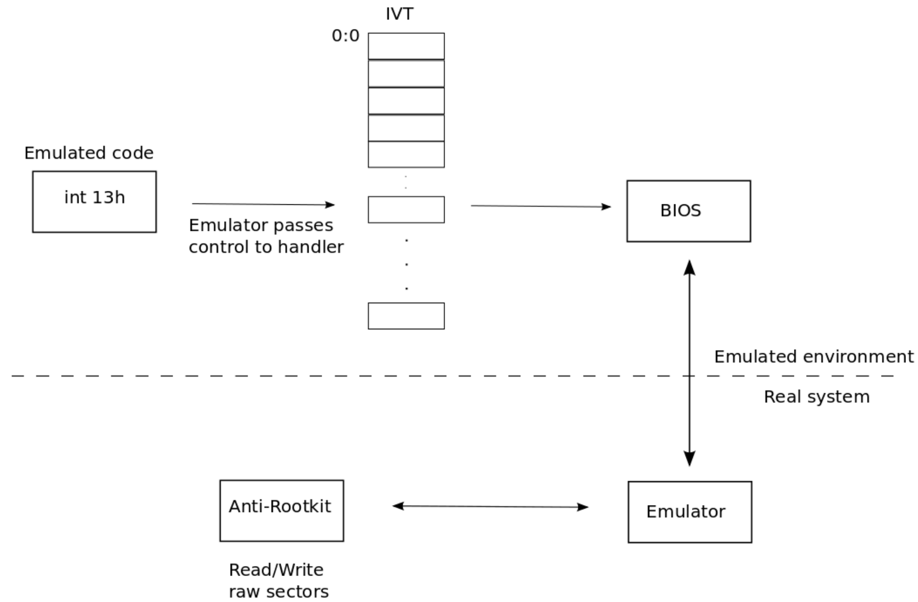

Figure 5: Emulating interrupt 13h

Some boot loaders expect to find various data on the current system stored in the BIOS Data Area [24], which is located at 40:0h. We do not need to populate the entire thing, but a few fields such as *memory size* at 40:13h should at least be present. This is not a structure we are planning to use for anything, but rather we just need to populate it with enough data to make most boot loaders work.

The last action taken by our initialization code is to fake interrupt 19h. Even though we could have issued an interrupt 19h to our own code within the emulator, it's simpler, and more straightforward, to do this outside of the emulated code. Actually, we read the contents of the MBR, and write it to emulator memory at 0:7C00h, before starting the emulation process. This way, the only action we need to take from within our BIOS is to set the expected values of two registers, and issue a *far jmp* to 0:7C00h, thus starting emulation of the first instruction in *MBR*. The expected values of the two registers are 201h in *ax*, and 80h in *dl*. Failing to do this actually causes some boot loaders to fail. From here on, emulation continues with the code residing in the MBR.

5 Heuristic Detection Techniques

Bootkits will generally aim to load an unsigned kernel mode driver at some point during the boot process. This possibility opens up as they manipulate the boot sectors so that they manage to run code before various security features kick in. The details of exactly how this is achieved varies between different bootkit implementations. What is common among most bootkit implementations, however, is that they normally do not rely on a single modification of the boot process to achieve their goal. Instead, they use a series of modifications that eventually enable them to load a driver. We will refer to the series of modifications and resulting control transfers as the bootkit's *load chain*.

Simply put, it is rather inconvenient to load a kernel mode driver before the kernel itself has loaded. The attack surface from an intruder's point of view is a total of roughly 17 unsigned sectors on disk that make up the MBR, VBR, and IPL. The time window for the initial modification, that is, the first step in the load chain, ends when bootmgr has been loaded into memory and control has been transferred to it. At this point, the bootkit must have already manipulated the boot process in some way, so that it can regain control at a later stage. Most bootkits rely on regaining control after the kernel and a few required boot drivers have been loaded, at which time they may find a chance to load their unsigned driver.

We are focusing our analysis exclusively on the emulation of the MBR, VBR, and IPL, under the assumption that something must be modifed during their execution. Furthermore, we are expecting to observe the same modifications taking place during emulation of this code.

At this early stage, there are severe limitations to what an intruder may achieve. The code introduced by the bootkit will be interfacing with the BIOS. This means that, when we emulate it, it will interface with our custom BIOS;

normally to read sectors on disk, and modify contents in memory, in an attempt at eventually passing control to the next stage of its load chain.

5.1 Interrupt 13h hooks

Bootkits often seek to perform in-memory patching of executables that have not yet been loaded. Whereas modifying contents on disk have a high chance of triggering alarms, modifying contents in memory is normally a safer bet. They might attempt to modify bootmgr, the kernel, or perhaps even some of the boot drivers, in order to ensure that control is returned to some other code they control at a certain stage in the boot process. A common technique used to accomplish this is to hook the interrupt 13h handler. There are no security mechanisms in place to ensure that the IVT has not been corrupted at this stage. Hence, several bootkit implementations will replace the interrupt vector corresponding to interrupt 13h with a location within their own code, while storing its original contents, so that they can call the routine that performs the actual operation. In this way, they are capable of intercepting every disk I/O operation that will be performed later on in the boot process. Typically, their hook routine will look for a specific byte pattern. If the pattern is found, they will know where to patch. Otherwise, they will simply call the original BIOS interrupt handler as if nothing happened. This enables them to modify contents on the fly, as it is being read from disk and loaded into memory, effectively patching memory contents.

Being wary of this trick, we take note of the original value of our interrupt 13h vector once it has been initialized. When emulation is complete, we verify that it is still intact. If the contents of the IVT within emulator memory has been modified, it usually indicates trouble, and may be treated as an anomaly. There are however examples of software solutions that hook interrupt 13h with good intentions, such as full disk encryption solutions. Challenges related to this practice are discussed in Section 7.

5.2 Patching bootmgr in memory

When emulation reaches the point where bootmgr is just about to run, the entire contents of it has already been loaded into memory. Bootmgr is a rather special executable, and is not loaded the same way Windows normally loads executables. Instead, it is simply loaded into memory just as it appears on disk, meaning its memory image should be identical to its disk image. As its image on disk is digitally signed, we can trust that it has not been tampered with. Should the memory image not match the image on disk, we will treat this as an anomaly. This usually means that code in the boot loader has hooked interrupt 13h, and patched bootmgr while it was loaded into memory. There should be no legitimate reasons for patching bootmgr, so this anomaly may be used to detect bootkits in a safe manner without risking false positives.

5.3 Behavioural anomalies when replacing MBR

In cases where the MBR has been replaced, which is the most common approach for bootkits by far, we may observe an interesting artifact in the behaviour of the boot code. Even though bootkits aim to manipulate the boot process, they normally do not want to contribute to it. That is, they generally do not include any code that actually takes part in loading the required components. Instead, they usually load the original MBR after they have made the changes they need to at this stage, and then pass control back to it. From here on, they let the boot process continue as if nothing has happened, and wait for control to be passed back to code they control at a later point. An intruder does not want to change more than is strictly necessary for the successful compromise of the system, as this can result in the system not booting at all. This approach also gives the bootkit a higher chance of working across multiple systems.

As the code residing in the original MBR expects to be run at 0:7C00h, the boot code planted by bootkits will normally load the original MBR at this exact address. Consequently, we will observe an anomaly in the number of times instructions at address 0:7C00h are emulated. This is illustrated in Figure 6.

The normal case is to emulate instructions at address 0:7C00h twice. The first time will be the first instruction of the MBR, and the second time will be the first instruction of the VBR. When bootkits have replaced the MBR however, the first instruction will be the first instruction of the bootkit's MBR. As this MBR seeks to load the original MBR to the location where it is currenly residing, it too, just like regular MBRs, will normally copy itself to a different memory location before loading the original (overwriting itself). This will lead to instructions at 0:7C00h being executed three times, which should be considered an anomaly. If we notice this artifact when we emulate the boot process, we know that the MBR has been replaced.

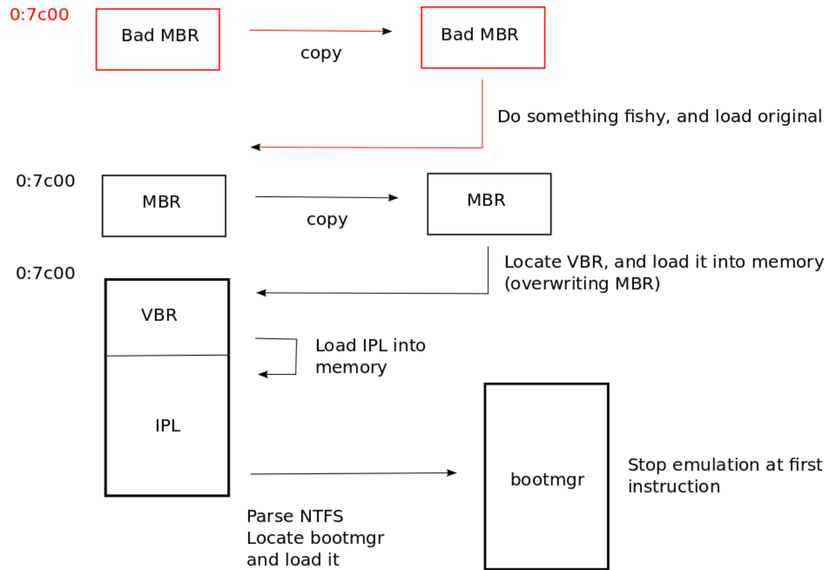

Figure 6: Normal boot code behaviour for bootkits targeting the MBR

6 Disabling bootkits

The most straightforward approach to disabling rootkits in general is to break their load chain. For bootkits, this can be achieved by retrieving and restoring the original contents of the modified parts of the boot sectors. The techniques detailed in this section are based on the observation that bootkits normally restore the original contents in memory during boot. Thus, at some point during the emulation process, we should be able to obtain the original contents from emulator memory. The key is to determine what has been modified.

From a recovery point of view, bootkits may be divided into two categories: those that modify the MBR, and those that either modify the VBR and/or IPL. We may distinguish between the two using the technique described in Section 5.3, i.e. by counting the number of times instructions are executed at 0:7C00h.

For bootkits targeting the MBR, we will normally find the original MBR contents in emulator memory the second time 0:7C00h is being emulated. Hence, we simply stop emulation at this point, and recover the contents.

In cases where either the VBR or IPL has been targeted, we let emulation continue all the way to the first instruction of bootmgr. At this point, we retrieve the contents of both the VBR and the IPL from emulator memory. Instead of determining exactly which part has been modified, we intend to replace all related sectors on disk with what we have recovered in this case.

By writing the recovered contents back to where it should reside on disk, we

are effectively breaking the bootkit's load chain. As this causes the bootkit to no longer load during boot, we finish off by rebooting the system. When it comes back up, the rootkit component should be gone, and further inspection of the system may be performed to reveal any components that may have previously been hidden.

Keeping in mind that bootkits often have a habit of blocking write operations to the boot sectors, we use anti-rootkit techniques as discussed in Section 3 when writing the recovered contents back to where it belongs on disk. If we choose not to utilize such techniques, we may be fooled into believing that the operation succeeded even though it did not.

It is worth mentioning that these operations should be handled with care, as they can easily be detrimental to the system. One approach to minimize the chance of causing damage is to make use of emulation to verify that the system will still boot after the intended changes have been made. Although this will never ensure that the operation will not cause harm, it will at least flag those cases where it definitely does.

7 Challenges

Even though heuristic detection techniques are efficient at detecting threats that have never before been seen, they inherently also have the potential of causing false positives. When flagging a certain behaviour as malicious, one will risk flagging benign software that have similar behaviour as well.

The severity of false positives in detection techniques depends on the scenario in which they are used, as well as which other techniques they are combined with. As an example, we can choose to classify a modification of the MBR as malicious, and at the same time use a whitelisting approach to avoid triggering a detection on systems that have well known boot loaders installed, such as GRUB. Combining techniques to minimize the chance of false positives is however outside the scope of this paper. Instead, we will discuss a few scenarios that complicates using the emulation approach for detection.

Inherently, when emulating the boot process, we are assuming that there is no need for user input at any stage. Our assumption should hold true for any bootkit that has modified the process, as they have everything to gain from not notifying the user of their presence. The assumption does however not hold true for boot loaders that enable loading multiple operating systems, nor for full disk encryption solutions that require users to enter a password during boot.

Some boot loaders will present the user with a choice of which operating system to boot. Even though it is perfectly possible to send keyboard input commands to our BIOS, it is hard to know which choice to make.

For systems running a full disk encryption solution that asks for a password at startup, this problem becomes even harder. Unless we can somehow retrieve the password to continue the boot process, we have no chance of making our emulation approach work.

There is also another challenge with full disk encryption solutions: they

will normally hook interrupt 13h so that they can decrypt sectors as they are loaded into memory. Hence, during emulation, we might observe boot code that manipulates the IVT set up by our BIOS. If hooking interrupt 13h is classified as malicious, this can lead to false positives.

We have not invested any time to research how to get around these problems in certain cases, nor in general. Preseumably, it should not be possible at all in the general case. What we can do, however, is to detect whenever the boot code asks for user input. In this way, we can at least know that our emulation approach will not work for the system we are currently running on.

In order to query for user input, the boot code will issue interrupt 16h to poll for keystrokes. This is trival to detect during emulation as the code will call our BIOS. We assume that the boot code will never poll for keyboard input unless it is expecting the user to enter something, so if this interrupt is issued, we abort emulation, and report that we cannot determine whether or not the system is compromised by using our emulation approach.

8 Conclusion

In this paper, we have shown how bootkits may be detected by looking for anomalies in the behaviour of the code residing in the boot sectors. Our approach is to emulate the boot code using a custom BIOS that incorporates techniques to interface with the lowest level disk driver in ways that bypass potential hooks. This heuristic detection technique is highly generic in nature, but face challenges for non-standard boot loaders. We have also shown how to use the same approach for breaking bootkits' load chains, so that we can disable active rootkit components to regain control of compromised systems.

References

[1] Ken 'Skywing' Johnson, Matt 'Skape' Miller: Bypassing PatchGuard on Windows x64, `http://www.uninformed.org/?v=3&a=3`, 2005

[2] Ken 'Skywing' Johnson: PatchGuard Reloaded: A Brief Analysis of PatchGuard Version 3, `http://uninformed.org/index.cgi?v=8&a=5`, 2007

[3] Eugene Rodionov, Aleksandr Matrosov: The Evolution of TDL: Conquering x64, `http://www.eset.com/us/resources/white-papers/The_Evolution_of_TDL.pdf`

[4] David Harley, Aleksandr Matrosov, Eugene Rodionov: Hasta La Vista, Bootkit: Exploiting the VBR, `http://www.welivesecurity.com/2011/08/23/hasta-la-vista-bootkit-exploiting-the-vbr`, 2011

[5] Aleksandr Matrosov: Rovnix bootkit framework updated, `http://www.welivesecurity.com/2012/07/13/rovnix-bootkit-framework-updated`, 2012

[6] Vyacheslav Zakorzhevsky: Cybercriminals switch from MBR to NTFS, https://www.securelist.com/en/blog/517/Cybercriminals_switch_from_MBR_to_NTFS

[7] Peter Szor: The Art of Computer Virus Research and Defense. Addison-Wesley, 2005

[8] Wikipedia: Unified Extensible Firmware Interface, http://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface#Secure_boot

[9] Microsoft TechNet: Securing the Windows 8 Boot Process, http://technet.microsoft.com/en-us/windows/dn168167.aspx

[10] Vyacheslav Rusakov, Sergey Golovanov: Attacks before system startup, http://securelist.com/blog/research/63725/attacks-before-system-startup/

[11] Jason Medeiros: NTFS Forensics: A Programmer's View of Raw Filesystem Data Extraction, http://www.grayscale-research.org/new/pdfs/NTFS%20forensics.pdf, Greyscale Research, 2008

[12] Mark E. Russinovich, David A. Solomon, Alex Ionescu: Windows Internals, Fifth Edition, Microsoft Press, 2009

[13] MSDN: SCSI_REQUEST_BLOCK structure, http://msdn.microsoft.com/en-us/library/windows/hardware/ff565393(v=vs.85).aspx

[14] MSDN: DRIVER_OBJECT structure, http://msdn.microsoft.com/en-us/library/windows/hardware/ff544174(v=vs.85).aspx

[15] OSDev Wiki: ATA PIO Mode, http://wiki.osdev.org/ATA_PIO_Mode

[16] OSDev Wiki: ATAPI, http://wiki.osdev.org/ATAPI

[17] MSDN: DEVICE_OBJECT structure, http://msdn.microsoft.com/en-us/library/windows/hardware/ff543147(v=vs.85).aspx

[18] MSDN: VOLUME_DISK_EXTENTS structure, http://msdn.microsoft.com/en-us/library/windows/desktop/aa365727(v=vs.85).aspx

[19] MSDN: FILE_OBJECT structure, http://msdn.microsoft.com/en-us/library/windows/hardware/ff545834(v=vs.85).aspx

[20] Bill Blunden: The Rootkit Arsenal, Wordware, 2009

[21] Insanely Low-Level, An Arkon Blog: Trampolines In x64, http://www.ragestorm.net/blogs/?p=107

[22] Seagate: SCSI Commands Reference Manual, http://www.seagate.com/staticfiles/support/disc/manuals/scsi/100293068a.pdf

[23] Wikipedia: INT 13H, http://en.wikipedia.org/wiki/INT_13H

[24] BIOS Central: BIOS Data Area, http://www.bioscentral.com/misc/bda.htm

A Reading the MBR using PIO

```
#define ATAPI_DRIVE_MASTER              0xa0
#define ATAPI_DRIVE_SLAVE               0xb0
#define ATAPI_BUS_PRIMARY               0x1f0
#define ATAPI_BUS_SECONDARY             0x170
#define ATAPI_DATA_PORT(bus)            (bus)
#define ATAPI_FEATURES(bus)             (bus + 1)
#define ATAPI_SECTOR_COUNT_PORT(bus)    (bus + 2)
#define ATAPI_ADDRESS1_PORT(bus)        (bus + 3)
#define ATAPI_ADDRESS2_PORT(bus)        (bus + 4)
#define ATAPI_ADDRESS3_PORT(bus)        (bus + 5)
#define ATAPI_DRIVE_SELECT_PORT(bus)    (bus + 6)
#define ATAPI_COMMAND_PORT(bus)         (bus + 7)
#define ATAPI_DCR(bus)                  (bus + 0x206)
#define ATAPI_STATUS_DRQ_FLAG           (1 << 3)
#define ATAPI_STATUS_BSY_FLAG           (1 << 7)
#define ATAPI_COMMAND_READ_WITH_RETRY   0x20
#define ATAPI_COMMAND_WRITE_WITH_RETRY  0x30
#define MAX_STATUS_POLL_SPIN_COUNT      0x8000
#define SECTOR_SIZE                     0x200


NTSTATUS ReadMBRUsingPIO(VOID *pMBR, INT drive, INT bus)
{
    NTSTATUS  ntStatus    = STATUS_SUCCESS;
    INT       status      = 0;
    UINT      i           = 0;
    UINT      spinCounter = 0;
    WORD     *pOutput;

    if (NULL == pMBR)
    {
        return STATUS_INVALID_PARAMETER;
    }

    pOutput = (WORD *)pMBR;

    //
    // Ensure exclusive access to CPU before proceeding (not shown)
    //
    GainExclusiveAccessToCPU();

    //
    // Set options for reading from MBR
    //
    _outp(ATAPI_DRIVE_SELECT_PORT(bus), drive);
    _outp(ATAPI_SECTOR_COUNT_PORT(bus), 1);
    _outp(ATAPI_ADDRESS1_PORT(bus), 1);          // Read from sector: 1
    _outp(ATAPI_ADDRESS2_PORT(bus), 0);          // Read from cylinder low part: 0
```

```
    _outp(ATAPI_ADDRESS3_PORT(bus), 0);          // Read from cylinder high part: 0
    _outp(ATAPI_COMMAND_PORT(bus), ATAPI_COMMAND_READ_WITH_RETRY);

    //
    // Wait until the sector buffer is ready
    // Currently we poll until DRQ is set, and BSY is reset
    //
    // Since error conditions will set all (or most) bits of status to high,
    // there is a chance that we might end up polling indefinitely since
    // we are waiting for BSY to be reset (i.e. set low).
    // This is why we are using a max counter in an attempt to avoid nasty hangs.
    //
    do
    {
        status = _inp(ATAPI_COMMAND_PORT(bus));
        spinCounter++;

        if (spinCounter > MAX_STATUS_POLL_SPIN_COUNT)
        {
            ntStatus = STATUS_UNSUCCESSFUL;
            goto _done;
        }

    } while ( !(status & ATAPI_STATUS_DRQ_FLAG) && (status & ATAPI_STATUS_BSY_FLAG) );

    //
    // Read contents of MBR
    //
    for (i = 0; i < SECTOR_SIZE / sizeof(WORD); i++)
    {
        pOutput[i] = _inpw(ATAPI_DATA_PORT(bus));
    }

_done:
    //
    // Critical part done - Release exclusive access to CPU (not shown)
    //
    ReleaseExclusiveAccessToCPU();

    return ntStatus;
}
```

Listing 8: Reading the MBR using PIO