Rootkit Detection via Kernel Code Tunneling

Mihai Chiriac
mihai.chiriac@gmail.com

Abstract

Current rootkit detection engines either use methods like "cross view", or analyze specific data and code areas. However, rootkits are getting more and more complex. No more are inline patches limited to the first bytes of a function: we can now find them anywhere in the execution flow. Instead of a simple jump/call to the malicious code, complex control transfer trampolines are now commonplace - and they can only be detected with a dynamic analyzer.

We present a novel rootkit detection technique called "kernel code tunneling". The technique uses a custom-made dynamic instrumentation framework to analyze execution flow. While similar dynamic instrumentation engines do exist, our engine offers significant advantages. First, it was designed for kernel mode operation. Second, it was designed to correctly handle potentially offensive code.

In this paper, we will present the design of a kernel-based dynamic instrumentation engine, we will analyze various tunneling sessions, with and without active rootkits and we will look at specific cases when instrumentation has provided us with enough data to effectively clean the infected system.

Introduction

Fifteen years ago, the average malware writer was an adolescent male, wanting to be noticed by his peers. Things have dramatically changed in the past years. No more are cyber criminals motivated by goals such as fame, fun or revenge: they now seek financial profit and want to stay away from the public eye. In order to achieve financial profit, they need their "creations" to stay undetected for as long as possible.

As the name suggests, a targeted attack is a cyber attack directed at specific individuals or organizations. Specifically crafted malware, only distributed to a handful of computers, may be extremely difficult to show up on the radar of security companies, especially with the current flood of malicious software (exempli gratia, over twenty million malware strains appeared in 2010 alone, a daily average of 55,000 new threats [1]) - almost like spotting a needle in a haystack. While offline (also known as server-side) polymorphism is notoriously more difficult to detect than classic polymorphism, and anti-anti-virus tricks definitely complicate detection, the ultimate tool to achieve stealth (and thus, stay undetected) is the rootkit.

Rootkits are sophisticated tools that allow pieces of malware to stay hidden, once they are installed on a system. In 2008, more than half of the biggest spam botnets used kernel rootkits, with numbers continuing to grow in 2009 and 2010 [2]. Also, more advanced techniques began to be utilized, further complicating detection and removal.

Classic detection methods

Rootkits achieve "stealth" by subverting operating system functionality, typically by modifying key data structures and/or operating system code. Thus, careful investigation of these areas is an obvious first approach. In kernel mode, rootkits may modify data structures such as the Interrupt Descriptor Table (IDT) or the System Service Dispatch Table (SSDT), among others. Processor model specific register

SYSENTER_EIP_MSR contains the address of the code running at privilege level 0 responsible for servicing system calls. Normally, all vectors in the IDT/SSDT and the SYSENTER_EIP_MSR point to addresses from within NTOSKRNL, the Windows Kernel, therefore a trivially tampered data structure is detectable with a simple bounds check.

Instead of modifying these data structures, rootkits may alter the kernel code itself, typically by detouring normal execution through the malware's own code. Since the detour itself is usually performed by inserting a simple branch instruction to the malicious handler (a technique known as inline patching), common detection routines also perform basic tests, such as checking whether the destination of a branch instruction resides outside the bounds of the currently analyzed module.

However, advancements in rootkit technology greatly complicate detection: both the **Rustock.C** (Fig. 1) and **TDL3** (Fig. 2) rootkits give control to their respective malicious routines via code trampolines that reside in "legal" areas. Finding the real destination of these trampolines may be achievable with a static analyzer, but considering that the trampolines may be changed from version to version - not to mention polymorphic trampolines - a dynamic analyzer will give the best results in the general case. It is also worth mentioning that inline patches are no longer limited to the first bytes of a function, as they can be found anywhere in the execution flow.

```
push cs
nop
sub esp, 4
mov dword ptr [esp], _address
retf
```

Fig. 1, Rootkit.Rustock.C trampoline

```
mov eax, dword ptr [FFDF0308]
jmp dword ptr [eax+FC]
```

Fig. 2 Rootkit.TDL3 trampoline

Cross-view is a widely used rootkit detection technique that involves comparing a "high level" view of an operating system resource with a "low level" view. Since rootkits are used to hide operating system resources such as files, a rootkit detection tool may compare the result of a high-level function like **FindNextFileA** with the result of a lower-level function like **NtQueryDirectoryFile**, or even the results of a raw file system and disk parser. For processes, a rootkit detection tool may use the high-level function **CreateToolhelp32Snapshot**, the lower-level function **NtQuerySystemInformation** or the results of a low-level parser of EPROCESS kernel structures, a circular, doubly-linked list that can be referenced via the **PsActiveProcessHead** kernel variable or via the exported function **PsGetCurrentProcess**. A low-level view of the process list may also be achieved by inspecting the **PspCidTable** data structure, or even using other techniques such as process identifier brute-forcing.

Cross-view's inherent need of going "deeper" can be seen as some form of Achilles' heel, as most rootkits also find much deeper hiding places. For example, **TDL3** infects the lowest-level disk miniport driver, greatly complicating the task of obtaining a "real" view of the disk; moreover, the same **TDL3** does not need a process of its own, as it injects its ring-3 code into svchost.exe, and keeps all its files inside a custom file system.

A rootkit detection tool may also check the integrity of loaded code (for example, by checking whether the loaded code perfectly matches the code on disk, after applying relocation information), but as we will see later in the paper it is perfectly possible to alter the normal code flow without altering a single byte of the original code.

Dynamic binary instrumentation

Dynamic binary instrumentation is widely used to measure a product's performance, to diagnose errors or simply to analyze a program's behavior. From a security point of view, it has been used in the context of ABI enforcement, a technique known as "program shepherding" [3].

From a technical point of view, our dynamic binary instrumentation framework consists of:

1) A code generation engine, generally responsible of adding instrumentation code to the original code; however, even if most instructions are copied on a 1:1 basis, there are some notable exceptions:

a) Possibly offensive instructions are either replaced, or preceded by special instrumentation code

b) Branch instructions are replaced with a set of specific routines, responsible of returning control back to the instrumentation engine. Fig. 3 shows a basic block starting at linear address 4017F7 and ending with the branch instruction at 4017FF (thus, the block itself ends at 401805). Should the branch condition be met, the processor will jump to the "branch taken" address, which is 401FE5; else, it will "fall through" to address 401805. Fig. 4 shows the translated basic block; first, we see that except for the branch instruction, all other instructions were copied on a 1:1 basis. It is also worth mentioning that in this case it was possible to compute both the "Branch Taken" and "Fall Through" addresses at translation time, therefore we simply need to return control back to the instrumentation engine, specifying the updated Instruction Pointer.

```
.4017F7 43                inc ebx
.4017F8 83 7D CC 00       cmp byte ptr [ebp-34], 00
.4017FC 89 5D B8          mov dword ptr [ebp-48], ebx
.4017FF 0F 8C E0 07 00 00 jl 401FE5
.401805
```

Fig. 3, A normal basic block

```
.3370000 43                inc ebx
.3370001 83 7D CC 00       cmp byte ptr [ebp-34], 00
.3370005 89 5D B8          mov dword ptr [ebp-48], ebx
.3370008 0F 8C ?? ?? ?? ?? jl __branch_taken
__fall_through:
                           JUMP_TO_VM (401805)
__branch_taken:
                           JUMP_TO_VM (401FE5)
```

Fig. 4, A translated basic block

It is extremely important not to affect execution in any way. The "JUMP_TO_VM" macro should not pollute the stack in any way, should not modify any of the CPU registers or a memory area outside of its own data structures. Therefore, the engine keeps a per-thread data structure called a "shadow stack", used to spill registers that can be modified by the code generation engine. An example basic block can be seen in Fig. 5.

```
.3370000 43                inc ebx
.3370001 83 7D CC 00       cmp byte ptr [ebp-34], 00
.3370005 89 5D B8          mov dword ptr [ebp-48], ebx
.3370008 0F 8C ?? ?? ?? ?? jl __branch_taken
__fall_through:
```

```
            xchg esp, dword ptr [__shadow_stack]
            pushf
            pushad
            JUMP_TO_VM (401805)
            popad
            popfd
            xchg esp, dword ptr [__shadow_stack]
            jmp dword ptr [__shadow_eip]
__branch_taken:
            […]
```

Fig. 5, A translated basic block, no stack pollution, no register alteration

2) A basic block manager, responsible of keeping a list of already translated basic blocks (also known as a "basic block cache") for quick retrieval. Keeping a basic block cache greatly increases execution speed, as translation time is quickly amortized when we encounter loops.

3) A self-modifying code manager is mandatory, considering that we may be dealing with offensive code. Not handling self-modifying code can have disastrous consequences: if a program modifies a basic block that has already been executed (therefore, it is present on our basic block cache) we might execute the old, stored code, instead of executing the modified one. In user mode, our engine handles self modifying code by making sure that every basic block resides in write-protected memory - if not, we change the memory attributes ourselves - and analyzing possible "Access Violation" exceptions. If a write would indeed modify one of our translated basic blocks, we'd just delete it from our basic block cache and temporarily remove the write protection to allow the write operation. An interesting particular case arises when an instruction modifies its own basic block – in that case, we have to re-translate the basic block and resume execution accordingly.

4) An asynchronous task handler is an extremely important component of our framework. A well-known anti-debugging trick is to transfer control by generating an exception. Our engine handles this situation by hooking the **KiUserExceptionDispatcher** function and by inspecting its parameters. The first parameter is an **EXCEPTION_RECORD** structure, which contains valuable information such as the exception code and the exception address. If an exception occurs inside our translated code, we need to update the **ExceptionAddress** member with the correct value (thus, we need to keep a mapping between the real code and the translated code). The second parameter is a **CONTEXT** structure, which contains the CPU registers from when the exception occurred. Just like with the exception address, we need to update the **Eip** member with the correct value.

We also need to correctly handle APCs (Asynchronous Procedure Calls) and user mode callbacks sent by win32k.sys; our engine handles these events by hooking **KiUserApcDispatcher** and **KeUserModeCallback**, respectively.

While testing the engine, we have observed a series of speed optimizations:

1) In most cases, we were able to directly link translated basic blocks; this happens when we're able to compute a block's successors at translation time (Fig. 6). Initially, the "cache_fall_through" and "cache_branch_taken" variables point to a subroutine that finds the successor(s). All subsequent executions of the basic block will use the cached addresses.
2) For basic blocks with successors that cannot be computed statically (Fig. 7A), we first attempt to find a match within

the last 4 successors (Fig. 7B). We also notice that since the lea/jecxz pair does not alter CPU flags, there's no need for the expensive pushfd/popfd pair.

```
.3370000 43                inc ebx
.3370001 83 7D CC 00       cmp byte ptr [ebp-34], 00
.3370005 89 5D B8          mov dword ptr [ebp-48], ebx
.3370008 0F 8C ?? ?? ?? ??  jl __branch_taken
__fall_through:
         jmp dword ptr [_BB.cache_fall_through]
__branch_taken:
         jmp dword ptr [_BB.cache_branch_taken]
```

Fig. 6, A translated basic block, successors are directly linked.

```
.405B17 FF 24 95 20 5B 40 00  jmp dword ptr [405B20+edx*4]
```

Fig. 7A, An indirect control transfer instruction.

```
SPILL_EAX
SPILL_ECX
mov eax, dword ptr [405B20+edx*4]
lea ecx, dword ptr [eax - _real_address_1]
jecxz _1
[…]
// no match, so JUMP_TO_VM (eax)
_1:
RESTORE_EAX
RESTORE_ECX
jmp _translated_address_1
[…]
```

Fig. 7B, Handling indirect control transfer

Normally, we don't need to add instrumentation code to every basic block. For the purpose of rootkit detection we only need to keep the list of executed basic blocks, the list of possibly offensive instructions discovered during translation, and other flags, such as the presence of "garbage", do-nothing instructions. Most of these operations can be performed only once, at translation time. Thus, our instrumentation engine is able to reach speeds that are comparable to native execution (average slowdown of only 25%).

Dynamic binary instrumentation in kernel mode

Porting our dynamic binary instrumentation engine to kernel mode proved to be a tedious task. First of all, we needed to be able to analyze code running not only at PASSIVE_LEVEL, but also at APC or DPC levels. We wanted to control each and every aspect of the code generation engine, therefore:

a) We've developed a custom memory manager. Our engine's memory requirements and access patterns were very

simple: we needed to store the basic block structures, the basic block cache and instrumentation information. Since most of these structures needed to be accessible by code running at any IRQL, our memory manager allocates during initialization a chunk of memory from the non-paged pool and partitions it as needed.

b) We've removed any kind of concurrent access control from the instrumentation engine. In user mode, our engine may concurrently analyze any number of threads. The situation in kernel mode is much more complex, as it would be theoretically possible for a thread to have to wait before being granted access to a shared resource (the basic block cache, for example). *If we want to instrument more than one thread, we can create a new instance of the engine.*

c) Detecting (and modifying) exceptions can theoretically be implemented by hooking various vectors from the **IDT**. In its current implementation, our engine does not monitor exceptions. It is possible for a rootkit to use hardware "read" breakpoints on its own code, and therefore detect that its code is being read. We are considering several tactics to circumvent this behavior.

d) Self modifying code is extremely difficult to detect in kernel mode; theoretically, we can use the same approach used in user mode, but practically it is almost impossible to detect all variations (a rootkit may directly modify page attributes, etc). Our current kernel mode implementation does not use the "direct basic block linking" optimization, so every transfer goes through the engine, which is responsible of verifying, via a set of checksums, that no modifications have occurred. However, if a basic block modifies itself (assuming the block is large enough and the modification occurs beyond the pre-fetch queue) our engine will execute the old, unmodified, code. Several strategies are being considered to prevent this behavior.

Analysis – reading the MBR

The device drivers involved in managing a particular storage device are collectively known as a storage stack [4]. If an application attempts an operation on a storage device, the request will first be received by the I/O Manager; the I/O Manager will, in turn, send the request to the File System; the File System will translate file addresses to volume addresses, and will forward the request to the Volume Manager.

Windows supports basic volumes (on a single partition) and dynamic volumes (can span across multiple partitions). Thus, the Volume Manager will forward the initial I/O request to the Partition Manager.

The lower-level drivers are the "Class drivers" - used to manage a particular device type, such as disks, or tapes, "Port drivers" that manage a specific transport (Storport for SCSI and RAID, Atapi for IDE-based devices), and "Miniport drivers", which are vendor supplied and manage hardware-specific details [4].

Appendix 1 shows an basic instrumentation session – we use a handle to \\.\PHYSICALDRIVE0, and we try to read, using **ZwReadFile**, its first sector. We see how the instrumentation engine analyzes - basic block by basic block - the entire code path.

We start in our own driver (KLup.sys) and we see how the engine arrives at **ZwReadFile**, down through the I/O subsystem, until we reach PartMgr.sys, which in turn calls ClassPNP.sys and Disk.sys. Our test system has an IDE hard disk, so the execution flows down to Ataport.sys, Atapi.sys and IntelIDE.sys (our miniport driver), and back up, to ClassPNP, PartMgr and finally, to KLup.sys, where instrumentation ends.

Once the instrumentation has ended, we can start an analysis session, using all the data that we've gathered: effective addresses of every block of code, their contents and various translation-related statistics (particularly "garbage" code that does not seem to be generated by a compiler). We see that each and every address is valid (i.e. it belongs to one of the code sections of a legally loaded module) and the code matches 1:1 the code on disk, so we assume that this particular code flow hasn't been altered.

Appendix 2 shows the same basic instrumentation task: we use a previously opened handle to \\.\PHYSICALDRIVE0 and we try to read its first sector, again using **ZwReadFile**. The difference is that we run the engine on a machine infected with the **TDL3** rootkit.

We again start in our own driver, KLup.sys, and we reach **ZwReadFile** and PartMgr.sys, which in turn calls ClassPNP.sys and Disk.sys. This particular system (a VMWare machine) has a simulated SCSI hard disk, therefore we should execute code from Storport.sys. However, we see that from ClassPNP.sys execution jumps to the *resource section* of "lsi_scsi.sys", where there's a code trampoline (Fig. 2) to an unclaimed memory zone – surely, it's where the rootkit resides. Our instrumentation engine continues and eventually reaches Storport.sys, then the real code of lsi_scsi.sys, then back to KLup, normally passing through ClassPNP and disk.sys.

In this analysis session we've noticed the following abnormalities:

- *A code trampoline outside of the code sections of lsi_scsi.sys*
- *A mismatch between the memory and disk images of lsi_scsi.sys (interestingly, NOT the code section!)*
- *Execution of "orphaned" code – code that does not belong to a legally loaded module*

Fig. 8 shows a greatly simplified hook routine; we notice that it first checks the input parameters and alters them as needed. If necessary, it gives control to the original handler and alters the original results.

```
Handler ()
{
      Check / Alter Input ()
      Call Original Handler ();
      Check / Alter Results ();
}
```

Fig. 8, Simplified rootkit handler

System disinfection

Obviously, the best method to clean a rootkit-infected machine is by booting the system from a clean disk. If a rootkit is already present in memory it may interfere with the disinfection itself. However, our engine offers interesting "live" system disinfection capabilities.

Fig. 9A shows a basic block belonging to TDL3's hooking routine. We could simply patch the conditional branch instruction, and transform it into an unconditional branch (Fig. 9B) – however, even if the solution is very effective, it's not very elegant. Furthermore, the piece of malware could check its own integrity, via checksums or other means.

An elegant solution is to keep "pairs" or basic blocks: *original* and *disarmed*. We currently have a hardcoded list or "pairs", but we are planning to create signature files. Our code generation engine follows a simple rule: if a basic block is matched exactly to an *original* block, then translate it as if it was the *disarmed* one. This way, *we effectively execute the clean code path without altering a single bit of existing code*, obtaining the desired results (e.g. read/write the real disk sectors, etc). This technique is extremely useful in detection and identification – as we execute the clean code path we have access to the full range of existing detection technologies.

```
.822E3B46 8B 44 24 20        mov     eax, [esp+20h]
.822E3B4A 8B 78 60           mov     edi, [eax+60h]
.822E3B4D 80 3F 0F           cmp     byte ptr [edi], 0Fh
.822E3B50 0F 85 95 01 00 00  jnz     __Original_Handler
```

Fig. 9A, Basic block belonging to TDL3's hook (original)

```
.822E3B46 8B 44 24 20        mov     eax, [esp+20h]
.822E3B4A 8B 78 60           mov     edi, [eax+60h]
.822E3B4D 80 3F 0F           cmp     byte ptr [edi], 0Fh
.822E3B50 90                 nop
.822E3B51 E9 95 01 00 00     jmp     __Original_Handler
```

Fig. 9B, Basic block belonging to TDL3's hook (disarmed)

Conclusions and future work

Kernel code instrumentation is a very powerful rootkit detection technology. We've shown how instrumenting a simple **ZwReadFile** call can help us analyze the entire storage stack, from the file system to the volume and partition managers, to the class, port and miniport drivers, and back up the chain. A call to **ZwCreateFile** helps us analyze all the file system filter drivers and the file system driver itself and it's an invaluable tool in detecting rootkits that hide files or folders. We suggest that analyzing less than a dozen of API calls, in particular those related to the file system, the registry, process and thread management, one could detect the majority of kernel rootkits. We've shown that the extreme level of control offered by dynamic binary instrumentation can help augmenting detection, but also the more elegant implementation of disinfection routines.

A particular class of kernel rootkits cannot be detected by our engine and has to be identified using other technologies. **DKOM** (Direct Kernel Object Manipulation) does not modify the code, or the code flow in any way, since it only alters kernel data structures, such as the EPROCESS list.

Another type of rootkit, called "Shadow Walker" [5], works by desynchronizing the CPU Data Translation Look-aside Buffers (DTLBs) from the Instruction Look-aside Buffers (ITLBs). Since the ITLBs contain the virtual to physical translations for code and the DTLBs contain the virtual to physical translation for data, desynchronizing the two means that it's possible for virtual address X to contain malicious code when executed and clean code when read. This is possible because virtual address X can point to a specific physical page when it's executed (according to the ITLB) and to a completely different physical page when it's read (according to the DTLB). Interestingly, in this case, dynamic binary instrumentation will execute the *clean* code, as read by the code generator, which may be enough to detect the rootkit.

Our current implementation only performs instrumentation on 32-bit code. Recently (summer of 2010) TDL3 (now TDL4) was updated to include a 64-bit component, and we expect others to follow suit, so we are currently developing a 64-bit version.

We are also working on analyzing network calls via Windows Sockets Kernel (WSK). This technique will allow our engine to detect most traffic-filtering kernel malware.

Appendix 1

-attached as appendix_1.txt

Appendix 2

-attached as appendix_2.txt

References

[1] BitDefender report, 2010, www.bitdefender.com

[2] "The top 10 spam botnets: New and improved", www.techrepublic.com, accessed February 2010

[3] "Secure Execution via Program Shepherding", *Vladimir Kiriansky, Derek Bruening, Saman Amarasinghe,* USENIX Security Symposium, August 2002

[4] "Microsoft Windows Internals, 4th Edition", *Mark E. Russinovich and David A. Solomon,* 2004

[5] "Shadow Walker – Raising the bar for rootkit detection", *Sherri Sparks, Jamie Butler,* BlackHat Briefings, 2005