Detecting kernel rootkits

Ashwin Ramaswamy
Department of Computer Science
Dartmouth College

Masters Thesis Proposal
Dartmouth Computer Science Technical Report TR2008-627

Proposal Presentation Date: 2 September, 2008

Abstract

Kernel rootkits are a special category of malware that are deployed directly in the kernel and hence have unmitigated reign over the functionalities of the kernel itself. We seek to detect such rootkits that are deployed in the real world by first observing how the majority of kernel rootkits operate. To this end, comparable to how rootkits function in the real world, we write our own kernel rootkit that manipulates the network driver, thus giving us control over all packets sent into the network.

We then implement a mechanism to thwart the attacks of such rootkits by noticing that a large number of the rootkits deployed today rely heavily on the redirection of function pointers within the kernel. By overwriting the desired function pointer to its own function, a rootkit can perform a proverbial *man-in-the-middle* attack.

Our goal is not just the detection of kernel rootkits, but also to levy as little an impact on system performance as possible. Hence our technique is to leverage existing kernel functionalities (in the case of Linux) such as kprobes to identify potential attack scenarios from *within* the sytem rather than from outside it (such as a `VMM`). We hope to introduce real-world security in devices where performance and resource constraints are tantamount to security considerations.

1 Introduction

A looming threat of malware is not an unfamiliar observation in the modern world. Malware threats are steadily venturing into diverse and intricate territories such as obfuscated

packing, local/global environment awareness, and command-driven scalable architectures. With sustained exposure to the Internet, a majority of systems are now, more than ever, vulnerable to deceptive software; either downloaded manually or injected surreptitiously into the system. In this thesis, we focus on a particular subdomain of malware called rootkits.

The term *rootkit* originates from the composition of the individual terms *root*, referring to the highest privilege of access that can be obtained in a traditional Unix-based operating system, and *kit*, referring to a set of programs that are designed to exploit a target system, gain root access and then maintain it without tripping any alarms. Rootkits are no longer an emerging threat. Their establishment in current systems is ubiquitous, and their presence is undeniable in any modern system infected with malware. Rootkits differ in a few ways from other malware such as viruses or worms. For one, they do not propagate viciously; in fact they almost never clone themselves. Rootkits assume that the system has already been rooted[1], and therefore their objectives are fundamentally different from viruses or worms, whose goals are to first break into a target system by exploiting its weaknesses and vulnerabilities. By contrast, a rootkit aims to *maintain* privileged access without alerting the user or system administrator. This challenge necessitates a different set of priorities for the rootkit program. Kernel or application vulnerabilities no longer matter for the rootkit author, who instead searches for sly, beguiling tricks to play on the operating system that would allow him to remain inconspicuous within the remote machine after rooting it.

Broadly, rootkits can be divided into two categories:

1. User-level
2. Kernel-level

User-level rootkits are programs that overwrite the filesystem binaries and libraries with customized versions that accomplish the desired "hiding" goals. Programs such as `ls`, `ps`, `netstat` are some common binaries that are likely to be modified to hide the rootkit and its accomplices from an unsuspecting user. In addition, a user-space rootkit might also modify configuration files and scripts in order to guarantee persistence and remain incognito. The prevailing weakness of such malware is that there exist standard techniques to easily detect, prevent and take corrective action against these rootkits. Software such as Tripwire [5] and AIDE [1] have proven themselves to be effective in most cases. Moreover, user-space rootkits, with limited manoeuvreability, are unable to surpass these detection techniques without stepping into the kernel itself, which leads us to the second category.

[1] *rooted* is a hackish term for gaining unauthorized root privileges on a remote system.

Kernel-space rootkits are more intricate and progressive than their user-space counterparts. With techniques ranging from naïve redirection of system-calls and altering interrupt handlers to modifying scheduler lists and `VFS` redirection, kernel rootkits are gaining tremendous momentum as the rootkit authors aspire to sustain their attacks and elude detection even from paranoid system adminstrators. There exist numerous techniques for a rootkit to "enter" the kernel. In Unix-based systems, rootkits typically employ the loadable kernel module (LKM) interface to insert their code into the kernel. Another avenue of attack is through the `/dev/mem` and `/dev/kmem` devices that allow direct memory access to system memory. Windows rootkits usually utilize the Direct Kernel Object Manipulation (DKOM) technique to modify kernel objects that control a variety of kernel behavior such as process management, network management etc. In this thesis, we focus only on Linux rootkits, but the underlying techniques remain universal and hence can be utilized for similar detection mechanisms in other operating system kernels such as Solaris, Mach, Windows etc.

There has also emerged a relatively new breed of rootkits called hypervisor rootkits [27], which exploit the virtualization layer to accomodate themselves *below* the operating system itself, and thereby gain total control over the entire system. Such rootkits are expected to faithfully duplicate the underlying hardware under all circumstances. However, faithful replication of hardware is believed to be hard and error-prone [12] due to various anomalies and discrepancies that can be caught at the CPU, MMU, TLB, timer chips etc. This is troublesome for the rootkit since now it has to not only duplicate the hardware, but also all the hardware's faults and anomalies just to avoid detection from the adversary.

There exist other categorizations for rootkits, focusing more on their attack strategies rather than their domain of deployment. Rutkowska [26] presents a malware taxonomy describing four classes of malware ranging from Type `0` to Type `III` of increasing threat-levels. Type 0 Malware includes just applications that are *harmful* to the system, but which do not subvert the operating system in any way. An example of such malware would be a program that deletes all files on disk. Type I Malware modifies resources in the system that were designed to be constant, such as the system-call table, kernel code sections etc. Type II Malware modifies the dynamic resources in the system, such as kernel data section, function pointers, data structures etc. Finally, Type III Malware comprise of hypervisor rootkits.

Baliga et al. [8] focus on the attack vectors of kernel rootkits, categorizing them into:

- Control Hijacking
- Control Interception

- Control Tapping
- Data Value Manipulation
- Inconsistent Data Structures

In this thesis, we primarily focus on Type `II` rootkits employing Control Interception and Control Tapping techniques.

2 Motivating Scenario

Broadly speaking, anomaly-based intrusion detection is an expensive process. While implementing an IDS, one has to be prudent to not adversely affect the performance of the operating system itself. Anomaly detection, by its very nature, searches for the presence of abberant behavior either within application processes or within the kernel. Performance consequences in the former might still be grudgingly accepted since only the monitored process(es) are being affected, but similar performance impacts on the kernel can escalate, effectively crippling all applications (including network-facing services) running on the system. Hence, one has to be extremely cautious when deploying an IDS in a real world scenario, to not only take into consideration the guarantee of security that the IDS provides, but also the real-time impact the IDS software might have on the normal functioning of the monitored system.

Having established that ensuring kernel integrity from rootkits is a vital challenge today, we observe that a large number of anomaly-based rootkit detectors are backed by virtual machine (VM) technology. While VMMs are generally thought to be a safe haven for monitoring software (since malicious software in the guest is assumed to be incapable of "escaping" into the host), hardware-assisted virtual machine technology for fully virtualized guests impacts system performance either to a lesser or greater extent, depending on one's point of view, and the actual tests conducted on the VMs. But the average performance hit was about 6% for Vmware and 15% for Xen. Vmware's report [28] has detailed analysis on the latency delays for both the Vmware ESX Server and Xen 3.0.3 hypervisor. In the embedded domain, it would be unpropitious to deploy these systems because of limited resources, performance constraints and real-time requirements, specifically within the US power grid infrastructure. Security in some of these systems seems to be lacking, as evidenced in the recent work done by our colleagues in attacking networked set-top boxes widely deployed across campus [6].

Supervisory Control and Data Acquisition (SCADA) systems are known for their antiquated security mechanisms in the millions of devices deployed today. Considering the elongated lifetime of these systems, security not only needs to be interwoven into the system, but also needs to be flexible for upgrades in the future. According to the security agency Riptech (now a part of Symantec), there is a lack of real-time monitoring on SCADA systems [24] due to the incapability of these systems to handle large amounts of data without performance impacts. Our goal is to introduce real-world security in these devices where performance and resource constraints are tantamount to security considerations.

3 Background

3.1 Approaches to Intrusion Detection

There exist two main approaches to Intrusion Detection (ID).

1. Signature-based: This detection mechanism relies on a known set of signatures or patterns that are derived from analyzing specific known malware (rootkits, in our case). This kind of IDS is fast since the system only needs to check for a finite number of signatures, whose presence usually indicates a compromised system. As such, newer or even hybrids of older attacks might bypass the security system altogether. Also in the specific case of rootkit detectors, signature-based methods [19] [4] [2] [25] [21] must be run manually (or periodically). Such detectors do not provide continuous *inline* monitoring of the host system.

2. Anomaly-based: These systems are more generic than their signature-based counterparts, observing either behavior patterns of processes over time [11] [13] or call stack monitoring [10] or other patterns of established behavior. Any deviation from the normal order would indicate aberrant behavior and hence a high likelihood of intrusion. Such systems usually suffer from slow learning times, detection latencies and false positives. In the subdomain of rootkit detection, anomaly detectors employ VMMs to enforce containment and isolation properties on the guest machine.

3.2 Loadable kernel modules (LKM)

The Linux kernel in general, exhibits a mature modular design, some examples being its kernel module interface and device driver architecture. Linux kernel modules (LKMs) can be thought of as *plug-and-play* software that can be inserted dynamically into the kernel, providing customized services that can be installed, configured and removed as necessary.

The `kmod` driver provides load-on-demand functionalities to the kernel modules residing on the system, so that an LKM would be injected into the kernel only when required and otherwise reside passively on disk. Interfaces in the kernel are exported to the LKMs for the module code to *hook* into the existing kernel. However, even as inserting LKMs requires root privileges on the box, as we shall see, this opens up a major avenue of attack for rootkits that use the same interface to subvert the machine.

3.3 kmem and LKM rootkits

Along with the LKM interface, Linux also provides a driver at `/dev/kmem` to directly access kernel memory. This can be used for applying bug-fixes or dynamically patching the kernel without incurring the overhead of a system reboot. The kmem and LKM interfaces are the two most widely abused boundaries in the Linux kernel, with rootkits routinely exploiting them for entry into the kernel. In fact, there have been cases of administrators disabling loadable modules in the kernel to completely avoid rootkit attacks. But this still admits `/dev/kmem` rootkits. There do exist slight differences in rootkits deployed through these two interfaces. `kmem` rootkits are admittedly weaker than LKM rootkits, mostly due to the fact that the rootkit now has to *guess* more often, since kernel-exported interfaces are not available. But `kmem` rootkits can rely on the driver always being available while in the case of LKM rootkits there is the possibility that the administrator might have crippled the interface. There have also been efforts to disable the `kmem` interface altogether[2], but on the 2.6.19.7 kernel we tested on, which was running Ubuntu 7.04, `/dev/kmem` could still be `mmapped`[3]. Figure 1 showcases the system architecture with respect to LKMs and the kmem interface.

3.4 Kprobes

Kernel probing through kprobes [18] is a mechanism developed by IBM researchers and introduced in Linux kernel version 2.6.9 that allows the execution of arbitrary handler code at any point during the kernel's execution. Since probe handlers are written within LKMs, one typically needs root privileges in order to use this facility. Kprobes is extremely useful for debugging specific parts of the kernel without the need for traditional recompile/reboot/test cycles, thus saving development costs and time. Furthermore, kprobes provides better tracing support for the kernel, facilitating deeper kernel transparency and better performance evaluations.

[2]http://kerneltrap.org/mailarchive/linux-kernel/2008/2/11/809424

[3]This is still the case in the latest versions of the kernel.

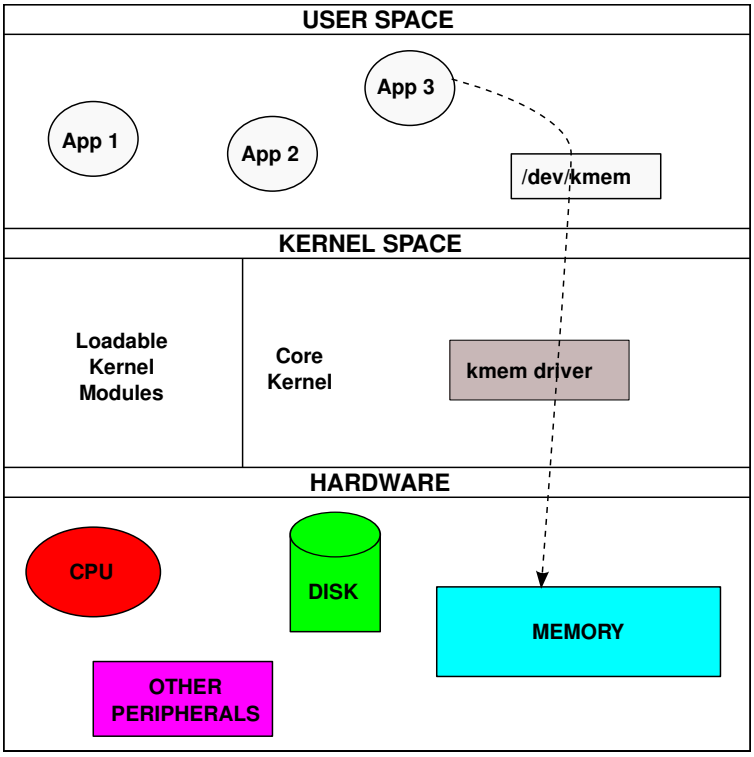

Figure 1: System architecture with respect to LKM and kmem interfaces. `App3` is a userspace application that uses `/dev/kmem` to access raw system memory.

Purely from a functional viewpoint, kprobes is similar to Solaris' DTrace [9], the latter being a better cohesion of tracing/trapping hooks with the core kernel than kprobes. The tracing code in DTrace is written in the `D` language that faintly mimics `C`, while kprobes handlers are written in `C` within LKMs. DTrace does not expect its users to be overly familiar or knowledgeable with the operating system internals, but kprobes demands it. On the other hand, the `C` language flexibility of kprobe handlers allows probing code to be more rigorous and expressive. Recently, a project called SystemTap [22] was undertaken whose current members include Red Hat, IBM, Intel and Hitachi, which aims to provide a usable aggregation interface on top of kprobes and make it more suitable for troubleshooting and diagnostic purposes.

We now take a brief look at how kprobes is implemented. This will aid in a better understanding of our rootkit detection system. The first step in registering a probe handler

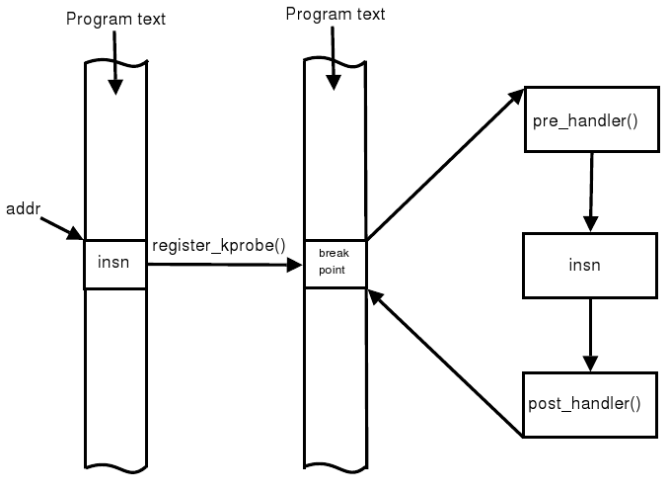

Figure 2: Kprobes control flow (duplicated from [18])

is to identify the target kernel code to probe. This can either be the name of a function or the address of some instruction in the kernel's text. Given this, kprobes provides two kinds of handlers: a pre-handler that gets called *before* the target instruction is executed, and a post-handler that is called *after* the instruction is executed. This delivers a higher granularity of control over the target instruction. Internally, probes are implemented by overwriting the destination with a breakpoint instruction[4]. When the breakpoint is hit (also known as *hitting* the probe), the interrupt handler takes over, validates that the origin of the breakpoint was indeed a kprobe, and transfers control to the pre-handler (if any). When this handler returns, a copy of the actual target instruction is run, and if a post-handler was registered, it is then called (see Figure 2). Normal execution flow then proceeds as before.

4 Requirements

The detection system we implement must be able to satisfy the following requirements:

1. To dynamically discover kernel hooks to protect, i.e. not rely on static tools or verification processes.

[4]On `x86` systems, the opcode of a breakpoint instruction is 0xCC.

2. To perform "in-place" or synchronous ID, i.e. to line up our detector with the kernel control-flow so we do not open up a time-slot during which one can deploy an *ephemeral* rootkit.

3. To *not* use virtual machines, tainting or other high-cost observation mechanisms.

4. To be performance-aware and practical in deployment. We accomplish this using already existing kernel frameworks such as kprobes.

5. To support detection in scenarios where the rootkit is already present on the system when the IDS is deployed, or where the rootkit gets installed at some later point in time.

5 System Design

The rootkits we study modify normal control-flow within the kernel through a technique commonly known as function-pointer hijacking (in fact, almost all complicated rootkits fall under this category). The technique is quite simple. A rootkit that wishes to "hide" itself first identifies the functions (and the corresponding function-pointers pointing to them) that it needs to subvert. For example system-calls such as open, close, getdents, read, write etc are possible candidates for hijacking, since gaining control over these kernel functions gives the hacker the right to allow/deny/modify access to the entire file-system. The function pointers in this case would lie within the kernel system-call table.

After identifying the target function (in this case, one of the above system calls), the rootkit then overwrites the system-call table entry to point to its own function. When this rootkit function gets called, it performs the necessary operations and finally calls the original function, possibly modifying its return value as well. Note that hijacked system-calls are the simplest case to detect, because the addresses of these syscall functions are fixed during kernel compilation itself and hence it is easy to check if the system-call table always points to these *fixed* addresses or not. But if the functions are dynamically loaded into the kernel through LKMs, then there are no such fixed addresses. Every function in any loaded LKM is likely to be pointed at, and there exist dense scenarios such as hijacking at the `VFS` layer, network device layer, binary format handling routines etc. It is important to realize that detecting a hijacked pointer (other than the system-call table entries) is difficult since the function-pointer was *meant* to be modified by a legitimate driver or module in the first place.

We divide the design of our system into three stages. We first show how modern rootkits can be more effectively written by allowing the rootkit to *search* for the desired hook, given just a minimal "semantic" specification of the memory region that is in close proximity to the hook. This frees the rootkit from depending on LKM functionalities being enabled in the kernel and allows for more diverse and hard-to-detect `kmem` rootkits. We then show how to discover the function-pointers that are potentially vulnerable to rootkits by searching for such pointer hooks in kernel memory, inserting probes at the functions pointed to by them, and validating that they're indeed called by an indirect assembly `CALL` instruction. Finally, we implement a rigorous detection mechanism on top of kprobes, using a subset of the previously learnt hook locations.

5.1 A network driver rootkit

We first see how rootkits do not even need LKM functionality in order to subvert the system. We notice that there exist exported kernel interfaces through which a module developer can gain access to several internal data structures and overwrite certain parts of the kernel that are not meant to be modified. However, the administrator of a system can choose to disable the loadable module interface, in which case many of these attacks are not possible. But the */dev/kmem* interface that allows one to read or write into the kernel memory is still a dangerous foothold for integrating malicious code into the kernel. The lack of kernel-exported interfaces that a typical LKM has access to, means that a rootkit deployed through the `kmem` driver can no longer be agnostic about where in memory the structure actually resides, since now, it has to *search* for the structure in memory.

There exist only a handful of known rootkits which employ the `kmem` interface and all of them have hardcoded addresses about the target data structure; or attempt to read the local "System.map" file to determine the address (which can be easily thwarted by removing the file from disk). What we first propose is to search for relevant structures in kernel memory using only a small amount of semantic information. Our aim is to make this process generic enough to work on multiple kernel versions and across varying architectures. Examples of such semantic information are the MAC address of the target network driver, the major and minor numbers of the character or block driver, address range patterns in the binary-format structure etc. However, naïvely searching for such patterns leads to a large number of "positive" matches, because the query is small enough to warrant such hits.

The first observation that we leverage in order to write a `kmem` rootkit relates to the general layout of kernel data structures in memory. Linked lists, in particular doubly-linked

lists, are the backbone of a majority of kernel data structures. The Linux kernel provides a framework for efficiently embedding these doubly-linked lists into any given structure. This is how most structures are "chained" together. Examples of such lists are lists of character devices and block devices, lists of network devices, lists of processes and LKMs, a scheduler list of processes, a list of binary-format handlers, lists of file-systems, inodes, dnodes, timer lists and so on. Now in order to support extensibility, these structures have function-pointers embedded within them that the necessary modules should modify to point to their own functions. We note that these are just a subset of function pointer "hooks" that are vulnerable to rootkits.

We then design our own rootkit that hijacks the network driver's `send` function, thus giving us control over all packets sent into the network. This is particularly dangerous since the packets we modify cannot be revealed by a local sniffer on the host such as tcpdump, ethereal etc. Our "semantic" specification in this case consists of just the MAC address of the network card. We then search for the MAC string over the entire kernel memory, which might potentially result in a large number of matches (when we performed this experiment, we got about 93 matches). But since we know that all network devices are chained together, we narrow down the search results by looking for doubly-linked lists within the vicinity of the previous hits (the range of the vicinity is approximated by the size of the kernel's `net_device` structure).

This next set of matches should ideally be just one (the target match), but in case there is more than one, we append a stronger specification such as the possible range of values at addresses close to the match (for example the MTU value, interface name etc). Finally, the `send` function-pointer is identified by a fixed offset from the linked-list as determined from the `net_device` structure. We first verify that it points to a valid function prelude and then modify this pointer to point to our rootkit function that changes the packets as necessary and then calls the original function to deliver them into the network.

What this implementation aims to show is that rootkits can completely bypass standard kernel interfaces and instead rely only on the kernel memory to *discover* their desired hooks, thus making them possibly even more difficult to detect.

5.2 Identifying hooks in kernel memory to aid rootkit detection

As we have established, overwriting pointer hooks within data structures is an increasingly apparent technique that rootkits employ to modify kernel behavior and remain hidden.

So determining such hooks is an arms-race between rootkit developers and the defenders attempting to detect and prevent such attacks. However, only little research [30] [29] has been done to automatically uncover these hooks within the kernel for defense purposes. What we propose is a technique to identify such function-pointer hooks within the kernel by systematically "walking" through the kernel memory. Since pointers are always word-aligned (and the word size on a 32-bit machine is 4 bytes), we attempt to search the kernel memory through `/dev/kmem` in increments of 4 bytes.

The Linux kernel is statically mapped during bootup at the virtual address `0xc0000000 + 0x100000`. The virtual-to-physical mapping is just a linear transformation from this base address, i.e. virtual address `0xc0000000` corresponds to physical address `0x0`. If the amount of physical memory installed on the system is above 896M, then by default, the first 896M is statically mapped, and the rest is considered as *highmem* region, which is dynamically allocated as necessary. In our case, the amount of physical memory on the VMware machine was 710M. To search for function-pointer hooks, we only need to walk the kernel's data section and not its text, so we skip the initial text region (demarcated by the kernel symbols `_stext` and `_etext`), align the start pointer on the next word boundary and traverse the memory contiguously. Our search pertains only to the physical memory mapped directly into the kernel and we do not attempt to traverse the kernel high memory. The algorithm is shown below:

1: `kdata_start` $\leftarrow$ `_etext` {*adjusted to a word boundary*}
2: `kdata_end` $\leftarrow$ `kdata_start` + `Size-of-physical-memory`
3: **for** w = `kdata_start` to `kdata_end` **do**
4: $v \leftarrow *w$. {Dereference w}
5: **if** v is a valid address in the kernel text **then**
6: **if** $*v$ is a valid function prelude **then**
7: Insert a kprobe at v.
8: **end if**
9: **end if**
10: $w \leftarrow w + 4$.
11: **end for**

A valid x86 function prelude consists of the following assembly statements:

```
push %ebp
mov %ebp, %esp
```

Since we built the kernel with frame pointers enabled, every function will have this signature at the start. Hence a pointer is a function-pointer iff it points to a valid prelude.

After inserting all the probes as identified by the memory walk, we exercise the system by running the test suite from the Linux Test Project (LTP) [3]. LTP is a project started by SGI and currently being maintained by IBM, that consists of a wide range of test cases meant to validate every aspect of the Linux kernel. A complete run of the test suite takes anywhere from 2-4 hours depending on system load. This duration might be further amplified by the fact that we could potentially have inserted a few thousand probes across the kernel. Basically, the idea here is to "hit" as many of the probes as we can, so it can be determined whether those addresses *did* contain function-pointers. Hence choosing an appropriate suite (in our case, LTP) is paramount, since we want to exercise the kernel as much as possible in order to maximize the chances of hitting the probes. Note that it might be possible to never hit some of the probes, since the corresponding functions might never get called (such as those in unused drivers) or only get called during system bootup.

When a probe is hit, in its probe handler, we need to ascertain if the client function was actually called through a pointer. To do this, we need to understand how pointer calls are achieved in assembly. Any function call is conducted through an assembly `CALL` instruction. The operands of this instruction indicate which function to call. In the case of a normal function call, the operand is a fixed value that points to the address of the target function. But when calling a function through a pointer in x86, there are two ways in which the call can be accomplished.

1. Through memory: The operand of `CALL` is a memory address. When the `CALL` instruction is executed, the contents of that address is first fetched, then treated as the address of the target function, and finally the call is performed.

2. Through a register: The operand of `CALL` is the index of a suitable register. The address of the function is loaded into this register just before the `CALL` instruction. The `CALL` instruction then takes this address from the register and calls the function it points to.

Calling a function through a pointer is also called an *indirect* function call, as opposed to a *direct* function call. Our job then, in the probe handler, is to verify if the current function was the result of an indirect call. To do this, we retrieve the return address from the stack, and disassemble it in reverse. Since we know that the last instruction is a `CALL`, we speculatively disassemble the previous bytes until we can form a valid `CALL` instruction at that position. We then analyze the instruction's arguments, and conclude if the call was a direct or indirect one. In the former case, we do nothing. In case of the latter, we first record the target function address. Next, in order to retrieve the address of the function-pointer hook, we land in two cases.

1. Memory-indirect `CALL`: In this case, the address of the hook is already in the operand. Since we've disassembled the call instruction, we can get this address quite easily. Examples of such calls are:
   call *(%eax)
   call *0x4(%eax)

2. Register-indirect `CALL`: Here the operand only has an index of some register (eg: call *%eax). To obtain the hook address, we would need to disassemble in reverse and look at the last point in code where the register was written into. This value would give us the required address. Traditional forward disassembly of code disassembles instructions starting from the base address, and moving downwards to lower memory locations, while reverse disassembly typically needs to disassemble instructions at memory locations that are higher than the base. But such reverse disassembly is never an easy task, mostly due to the fact that there is no knowledge of where the previous instruction begins or what its length is. This makes reverse disassembly a clumsy and inaccurate process. But we still somehow need to know what the previous instructions are.

   We get at them through forward disassembly, starting the disassembly from the point where the function begins, and constructing a control flow graph(CFG) of the current function. We can do this because we know that every function has to start with a prelude; so we first search back (from the return EIP) till the point where we find a valid prelude, then disassemble forward, while creating a CFG of the complete function (we explain in section 5.3 how we construct the CFG).

   After constructing the CFG, we locate the node in which the `CALL` instruction lies, mark the desired register that makes up the call, and search *backwards* from this node for the instruction that writes into the required register. In case of multiple registers (such as base and indexed registers) present in the `CALL`, we repeat this procedure for each individual register. To search backwards in the CFG, we simply reverse the edges of the graph and follow the paths. Within each node, we traverse its instructions in reverse. After locating this instruction(s), we can determine the address of the hook by analyzing the contents of processor registers when this instruction(s) was executed by the kernel. The above process assumes that functions are linearly placed in memory, which is the case with the Linux kernel. This technique is shown in Figure 3.

To illustrate, consider the following fictitious example. The instructions listed below set up a register-indirect call:

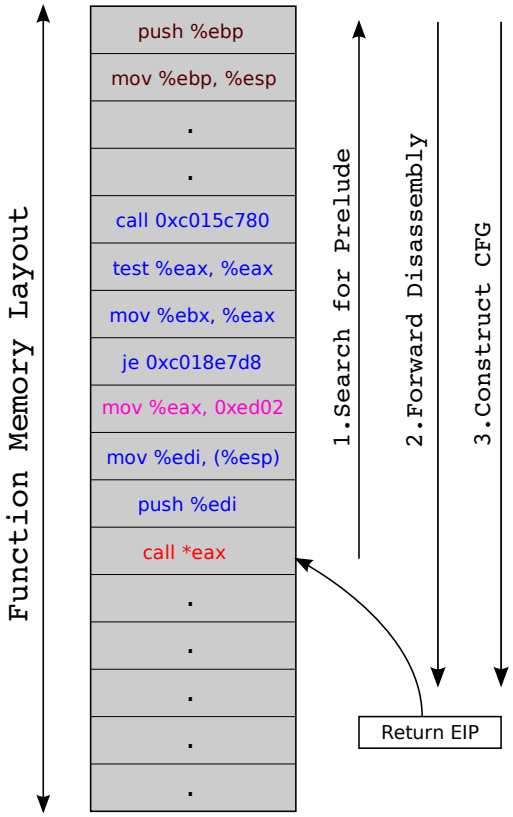

Figure 3: Finding the operand value in an indirect CALL

```
1. mov 0xc0454f70, ebx
2. add ecx, edx
3. mov [ebx+4], eax
4. cmp edx, esi
5. jne 0x300
6. call *eax
7. jmp 0x100
8. nop
```

Simply analyzing the register contents at instruction 6 doesn't get us anywhere since eax contains the address of the function, not the hook address. The last instruction that wrote from memory into eax is at position 3. After determining this, if we analyze the register contents at that instruction, then adding 4 to the contents of ebx gives us the hook address.

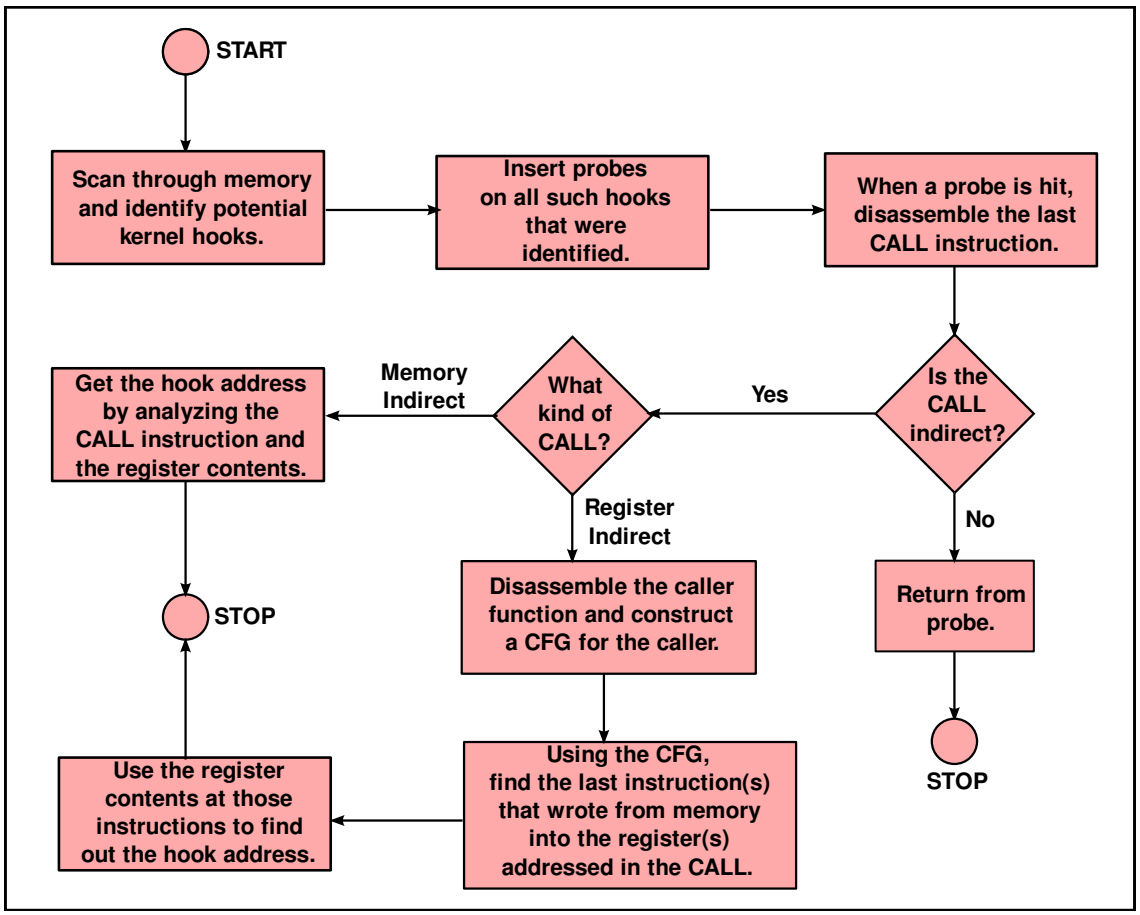

Figure 4: Flow diagram of the setup phase.

The general flow of our setup phase is shown in figure 4.

In our particular case, computing the address of the hook is necessary only if the target function is within a kernel module, since all functions within the core kernel have fixed addresses across reboots. In general, knowing the hook address makes our process more generic and hence this step can assist other detection systems as well, particularly those that require the hook address in addition to the target. Also, a manual inspection of these pointer references within the kernel source code might help in identifying other potential locations for rootkits to hook.

Our current implementation uses a user-level process to scan the kernel memory through

`/dev/kmem`, then interacts with a kernel module through a customized character device driver. The LKM inserts the requested kprobes and returns the results back to the user process, which analyzes the result and finally creates a table of hook addresses and the corresponding functions they point to. We call this system as our setup system (or setup phase), since the identified hooks serve as the input for our intrusion detection system.

Another implementation approach would be to use a VMM for just this phase. Since our setup phase is decoupled from the actual intrusion detection system, it would be possible to use a VMM to determine the kernel hooks. Such an approach could either walk through kernel memory (as we do), or could disassemble each instruction as it is executed on the host, and record the target in case of an indirect CALL instruction executed from kernel-space. We choose to go with a non-VMM implementation, mainly due to the fact that retrieving kernel/system state from *outside* the system (such as from a VMM) is harder than retrieving it from within. Also, this approach helps us share code between our setup phase and intrusion detection system (which is also built atop kprobes), which would not have been possible had we gone with a VMM implementation.

5.3 An anomaly-based rootkit detection system

To implement an anomaly-based system, we first have to characterize the behavior of a majority of existing rootkits that use pointer hooks. Recall that pointer hooks are essentially function pointers that are present in the kernel's data sections. The addresses of these hooks can be either static or dynamic, depending on where the hooks are located. We henceforth refer to the function that is called using a pointer hook as a *hook function*. A rootkit hook function is a function that lies within a rootkit, while a kernel hook function is one that lies within the kernel or in non-rootkit LKMs. Notice that the primary goal of a rootkit is to *hide* itself. Every rootkit must adhere to this most basic criterion, and in order to do so, a rootkit must still maintain the exact same kernel interface that previously existed, since altering the interface would expose the presence of a rootkit. In his book on rootkits [14], Joseph Kong describes hooking as "a programming technique that employs handler functions (called hooks) to modify control flow. Typically, a hook will call the original function at some point in order to preserve the orginal behavior".

Preserving the original control flow (Figure 5) is paramount to the rootkit's objective of hiding itself, since otherwise it exposes a completely different functionality of the system that can be detected by a HIDS or even a skeptical sysadmin. Not calling the kernel hook function would mean that the underlying functionality would be broken, which the rootkit

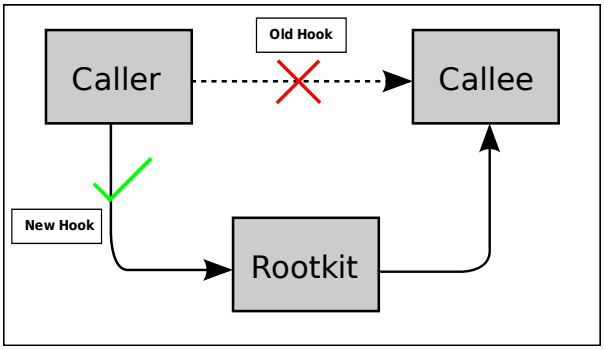

Figure 5: Hooking kernel execution through a rootkit

author does do not wish as the rootkit then becomes easily visible and detectable.

A hook function calling another hook function is exactly the kind of behavior we wish to detect. Since the hook functions in both cases are indirect `CALL` instructions, and the arguments passed to both these functions are identical (or at least close to being identical), we should be able to detect such a scenario and report the existence of a rootkit. We leverage the kprobes feature of the Linux kernel to insert probes on the functions identified from the setup phase, and when the probes are hit, we scan for the possibility of the above attack vector. We also make the assumption that the kernel text is unmodified by a rootkit. This is not a limiting factor of our system since kernel text modifications are detectable by many existing systems [25] [16] [17], and so is a dangerous avenue for any rootkit trying to avoid detection. Moreover, having the entire region of kernel data at its disposal, the rootkit has the capability to perform subtler attacks than simply overwriting kernel text, which can be detected almost instantaneously by an external detector. Also, maintaining the integrity of kernel text is an orthogonal problem which we do not tackle here.

Our method works as follows. During the setup phase, if the hook function is determined to be in an LKM, since across a reboot the address of this function changes, we first put a kprobe on the caller's `CALL` instruction. When this probe is hit, we disassemble the `CALL` and obtain the address of the target (callee) function. We then proceed as follows.

Given the hook function (either directly through the setup phase or by determining it through the above method), we first insert a kprobe on the given address. Our detection system then passively waits until the function is called during normal kernel execution. Note

that if the function is never called, then there is no overhead involved. When the function gets called, the kprobes framework transfers control to our registered handler. In the probe handler, we do the following:

- First determine whether the function was called through a pointer hook. To do this, we repeat the same procedure as we did in the setup phase, i.e. look at the return address on the stack, disassemble the last `CALL` instruction and verify that it is an indirect call. In case of a direct call, we return. Note that we still need to repeat this step, since the same function can be called both directly and indirectly from different parts of the kernel.

- We now turn to the detection phase. Let h be a hook, and F be the corresponding function that we've put a probe on. If a rootkit gets installed in the presence of our system, then the rootkit would modify h to point to its own hook-function F'. Since the original functionality must be unhindered, F' still calls F. So the call chain now looks like $F' \rightarrow A \rightarrow B \rightarrow ... \rightarrow F$. Here $A, B...$ might be intermediate functions in the rootkit itself. Hence, all we need to do is look up the call-chain from F, and for each return EIP, disassemble the previous `CALL`, check if it is an indirect one, and if so, then verify the arguments passed to that function (in this case only F', since A, B,... are all direct function calls) with the arguments that this function (F) got. If similar, then we report a rootkit above us.

  Since it is impossible to determine the number of arguments by just looking at the stack, we use a heuristic that approximates this number to four, and claim that both the function arguments are similar if at least half of them are identical.

- A more rigorous case is when a rootkit is already present when our system gets installed. In this case, the function that we put a probe on is the rootkit function itself. But we don't know that yet. The call-chain still looks like $F' \rightarrow A \rightarrow B \rightarrow ... \rightarrow F$, but the probe is now on F'.

- To handle this scenario, we first disassemble the current function (F') in memory. We show later how to do this. Then we put probes on all functions that F' calls (both directly and indirectly). At the same time, we also put probes on all the return instructions in F' (since we don't exactly know which one might get called). Now when those underlying function probes are hit, we repeat the same process, i.e. disassemble the function and put probes on all function calls and return instructions in the disassembly, and so on. Note that if F' (or any of the functions $A, B, ...$) call a kernel function through a direct `CALL`, then we need not enter that kernel function, since by

assumption, the kernel text is unmodified. Hence we only put probes on indirect `CALL`s and on direct `CALL`s whose target is *not* within the core kernel.

- We then pass context information across all these probes, so that when they're "hit", they look up their context to determine what to do. The context supplied to direct function `CALL`s indicates the probe handler should continue putting probes on functions underneath it, while the context for an indirect `CALL` indicates that the handler should *check* its arguments against those of F', and also continue inserting probes if the function is not within the core kernel.

  So, in the above call-chain, the probe-handler for F' puts a probe on A, A's handler puts a probe on B, B's handler puts a probe on F, and F's handler checks its arguments against F' (since F is an indirect `CALL` from B), and in this case since they're similar, the rootkit gets detected.

- Finally, when any of the functions return, one of the `RET` probes that we installed gets called. We don't know which one, but it doesn't matter. When our `RET` probe handler is called, we uninstall all the probes for that particular function.

Now, to identify all `CALL`s within a function, we first need to find out where the current function ends in memory. It is not quite as simple as searching for the first `RET` instruction, since the same function might have multiple points of return. Hence stopping at the first `RET` would mean that we only partially disassemble the function and hence might miss out on the call to the original function. To solve this, we construct a control-flow graph `CFG` beginning from the function prelude and disassembling each instruction that we encounter.

Briefly, a `CFG` consists of basic-blocks that are delimited only by jump or return instructions. Each node in the `CFG` represents a contiguous sequence of instructions that has a *single* jump or return instruction at the end. An edge exists between two nodes n_1 and n_2 if the target of n_1's (conditional or unconditional) jump is *some* instruction in n_2, or if n_2 contains the successive instruction that follows n_1's unconditional jump. The second case is necessary because in case of an unconditional jump, the program can either take the jump, or ignore it and continue with the next successive instruction. The `CFG` is not a `DAG` (directed acyclic graph) since there might exist cycles between the nodes. A start node is one that has the function prelude instructions, while a leaf node in the graph is one that has no outgoing edges. Typically, this is also a node that has `RET` as the last instruction.

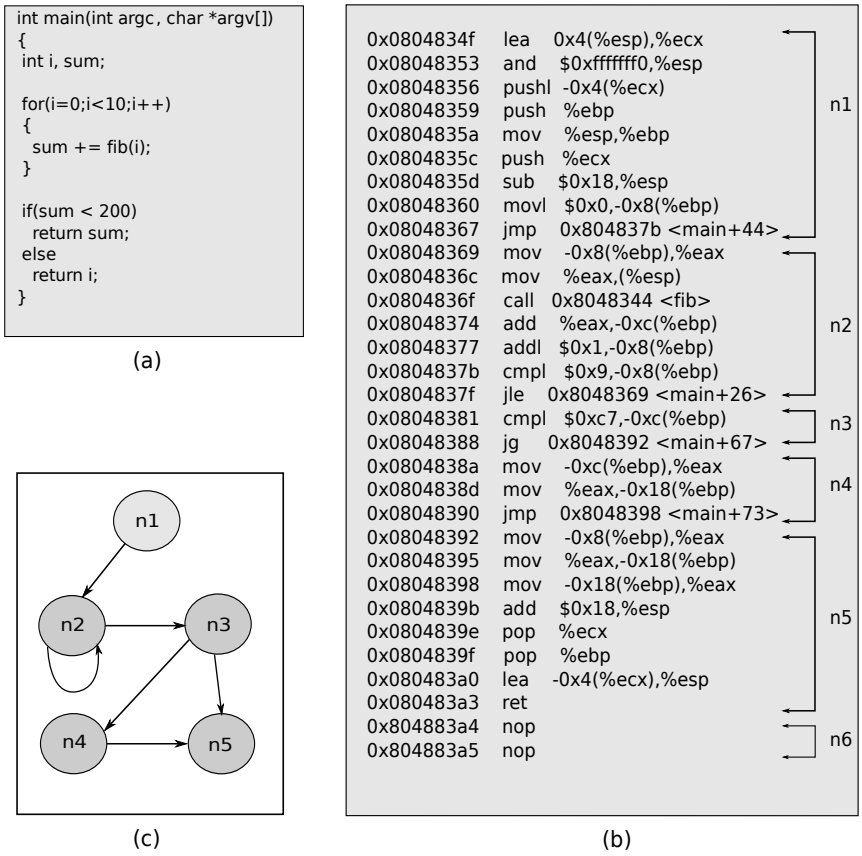

Figure 6: Construction of a control-flow graph

After constructing this `CFG`, we simply run a connectivity algorithm such as breadth-first-search (`BFS`) to traverse all nodes in the graph. For every node encountered, we determine if any `CALL` instructions make up that node. If so, then we insert a probe on the target. Similarly, if there is a `RET` instruction within the node, we insert a probe on the instruction itself. Thus we can determine all `CALL` and `RET` instructions within a given function.

Figure 6 demonstrates how a `CFG` is constructed from memory. Figure 6(a) shows a simple C program and 6(b) shows its corresponding disassembly with the `CFG` nodes marked by the side. There are some trailing `nop` instructions towards the end (after the `RET`) that are not part of the main function. Let us now start our initial memory disassembly from address 0x0804834f. The CFG for the corresponding disassembly is shown in 6(c). The edge from

n_1 to n_2 is due to the `jmp` at address 0x08048367. Node n_2 has an edge onto itself due to the `jmp` instruction at address 0x0804837f. But since this is an unconditional `jmp`, and the next instruction is a part of node n_3, there is also an edge from n_2 to n_3. The rest of the graph is constructed similarly.

Running a BFS from node n_1 allows us to determine all `CALL` instructions (in this case, only the one at 0x0804836f) and `RET` instructions (again, only one at 0x080483a3). Hence, we can reliably identify all function calls and return instructions that belong to our characteristic function and insert probes on them to aid our detection process.

6 Related Work

6.1 SBCFI

Control-flow integrity (`CFI`) is a security technique to ensure that the execution paths taken by a program at runtime conform to a `CFG` that is compiled statically by analyzing the program source code. Petroni and Hicks present an imitation of `CFI` called state-based `CFI` or `SBCFI` [20] that validates the data structures of the kernel and the kernel state itself as a *whole* instead of tracking the individual execution branches as they occur. This has the advantage that the introspecting process can be external to the system (in a `VMM`), but opens up a *window* during which an ephemeral (or non-persistent) rootkit might be deployed.

Their learning process is done statically by analyzing the kernel code for global variables and tracking their usage across the kernel, and validating them at runtime. The static nature of their system and learning inflexibility (due to the rapidly changing nature of the Linux kernel) are some shortcomings of this approach. The performance of `SBCFI` is also shown to incur close to 40% overhead on a typical machine running on Xen, which makes it quite unrealistic for embedded systems.

6.2 Static Binary Analysis

Kruegel, Robertson and Vigna [15] propose a method to prevent rootkits from entering the kernel by statically analyzing all the LKMs *before* they are loaded into the kernel. Using a symbolic execution technique that "simulates the program execution using variable names rather than actual values for input data", the LKM's behavior is determined to be either conforming or surreptitious. However, a static white-list of valid memory regions and kernel symbols are necessary in order to distinguish a malicious LKM from an unsuspecting one.

Building such comprehensive lists is of course a huge problem for such large and constantly evolving systems. Another drawback of this approach is that it only monitors LKM rootkits leaving the door open for *kmem* rootkits, that as we've shown can be just as devastating to the kernel.

6.3 Paladin

Paladin [7] divides the system into two *protected zones*, which are safeguarded from illegal access from rootkits, or other kinds of malware. The *Memory Protected Zone* consists of the kernel hook tables and other static regions of the kernel that need to be protected from rootkits, while the *File Protected Zone* consists of system binaries and libraries that need to be prevented against modifications. Given the specifications of these zones, Paladin uses a `VMM` to monitor write accesses across the system for validity and any time an invalid access into any of the above zones is detected, a process-tracker component identifies the entire process subtree for the process that elicited the write and kills all processes within this tree. A Paladin driver resides on the guest machine to facilitate this process and interacts with the monitor to aid detection. This serves as an excellent prevention mechanism too, except that the specifications of *MPZ* and *FPZ* are again static and a comprehensive survey of them is infeasible. Also, just monitoring for writes to specified locations and preventing them does not prevent rootkits from hijacking function hooks within data structures since these locations are *meant* to be overwritten.

6.4 HookMap

Wang et. al [29] propose a technique to uncover hooks in the kernel by tracking the kernel-level control flow for an interested application using a QEMU-based virtual machine monitor. Their basic idea is quite similar to ours - a rootkit wishing to hide itself - in this case, from certain user-level applications such as ls, ps, netstat or explicit monitoring agencies such as Tripwire [5] or AIDE [1]. To do this, a rootkit would have to hook locations that lie in the execution paths of these programs. Hence tracking the kernel control flow for these applications would reveal those hooks (which, as we know, are just indirect function calls). Cataloguing the hooks would serve an IDS that leverages these findings to protect such pointers. HookMap has the distinct advantage of narrowing its playing field to present a concise description of just the necessary hooks that need to be protected within the kernel. Moreover, hooks that do not lie in the execution paths of any of these programs would still go undetected. Our aim is broader and we cover most of the kernel hooks in the data region that have a reasonable chance of getting manipulated by any rootkit.

6.5 NICKLE

The NICKLE system by Riley et. al [23] implements a shadow memory controlled by a `VMM` for the entire region of kernel memory in the guest machine. Upon guest bootup, all known authenticated kernel instructions are copied into the shadow memory, and at runtime each kernel instruction fetch is verified by comparing the shadow memory maintained by the `VMM` with the actual physical memory at that location. Any differences here would indicate the presence of a rootkit, and thus prevent it from executing on the guest system. LKMs also need to get authenticated before their insertion since NICKLE cannot distinguish between a valid and a malicious kernel module. A disadvantage of this kind of authentication scheme is that it needs to be manually performed every time a module is inserted into the kernel, and in-depth analysis is necessary to ensure that the LKM does not invalidate the kernel.

7 Evaluation

We plan to evaluate our system by testing it against a sampling of real-world rootkits. For showcasing diversity, each rootkit would ideally use a different hooking mechanism to subvert the kernel. We would use these rootkits to test both cases: when the IDS is installed before the rootkit is deployed, and when the IDS is installer after rootkit deployment.

We would also like to come up with some possible attack scenarios against our system and show how our system currently mitigates them or how improvements can be made to the system to detect such cases. For example, we might consider the scenario when the rootkit copies the original kernel hook function and executes it, instead of jumping to the function. But such a scenario is mitigated by just enabling the processor's NX bit, which prohibits instruction execution from any memory page that has been written into. Hence any copy-and-execute technique would fail on such machines.

We would also conduct a performance analysis of the host system to estimate the overhead imposed by kprobes and our IDS. Since the overhead of kprobes depends on the actual number of probes inserted on the system, we could also calculate the system overhead as a measure of the probe count.

8 Improvements

We hope to improve some of the heuristics proposed here. For example, instead of using a heuristic to determine the number of arguments that a function takes, we can use the

setup process to conclusively determine this number and feed it into our detection system along with the initial list of function addresses. Similarly, in the setup phase, we could also double-check the address of the prelude that we obtain by confirming this address with the one reported by gdb (when run on the debug version of the Linux kernel). This confirmation might be necessary since we can envision a scenario where the hex-code constituting the prelude might be the trailing portion of a valid x86 instruction, in which case our prelude-estimation algorithm would return an incorrect value. Finally, we are also looking at caching opportunities for all probes inserted into the kernel, mainly for performance reasons.

9 Timeline

- October:
  - Complete the setup phase and build a table of kernel hooks and associated caller/callee functions.
- November:
  - Perform improvements to the setup phase as indicated in section 8.
  - Embed a suitable disassembler into the IDS LKM.
- December/January:
  - Build the intrusion detection system as an LKM.
  - Improve on performance by caching the inserted probes.
- February:
  - Evaluate the prototype on real-world rootkits and also conduct a performance analysis.
- March 1, 2009: Begin writing thesis.
- March 31, 2009: Send thesis draft to advisor for feedback.
- April 15, 2009: Send thesis to committee.
- May 1, 2009: Thesis defense.
- May 15, 2009: Complete revisions suggested by committee.

- May 15, 2009: Turn in official copy of completed thesis to Office of Graduate Studies, no later than 3PM.

10 Future work

Future work would involve developing hardware-based security for our detection system. We can achieve rigid security by building hardware primitives that complement our system by providing better memory guarantees for both the kernel code and user level applications. We have done some preliminary work in this direction in our proposal for a Better Mousetrap[ref].

11 Conclusion

In this paper, we have shown the emerging capabilities of contemporary rootkits, and also developed an analysis engine for the kernel data where we automatically recover function pointer hooks used within the kernel. We then leverage our findings by building a low-overhead, rigorous detection system for rootkits intercepting kernel control flow. We also hope to introduce a self-healing component to our system that recovers from a rootkit while simultaneously ensuring normal system functioning.

References

[1] AIDE. *http://sourceforge.net/projects/aide*.

[2] chkrootkit. *http://www.chkrootkit.org*.

[3] Linux Test Project. *http://ltp.sourceforge.net*.

[4] Rootkit Hunter. *http://rkhunter.sourceforge.net*.

[5] Tripwire. *http://www.tripwire.com*.

[6] Baek, K., Bratus, S., Sinclair, S., and Smith, S. W. Attacking and Defending Networked Embedded Devices. In *WESS '07*.

[7] Baliga, A., Chen, X., and Iftode, L. Paladin: Automated Detection and Containment of Rootkit Attacks. In *Technical Report DCS-TR-593, Rutgers University, Department of Computer Science* (2006).

[8] Baliga, A., Kamat, P., and Iftode, L. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. *Security and Privacy, 2007. SP '07. IEEE Symposium on* (May 2007), 246–251.

[9] Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. Dynamic Instrumentation of Production Systems. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference* (2004), USENIX Association, p. 2.

[10] Feng, H. H., Kolesnikov, O. M., Fogla, P., Lee, W., and Gong, W. Anomaly Detection Using Call Stack Information. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2003), IEEE Computer Society, p. 62.

[11] Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. A Sense of Self for Unix Processes. *sp 00* (1996), 0120.

[12] Garfinkel, T., Adams, K., Warfield, A., and Franklin, J. Compatibility is not Transparency: VMM Detection Myths and Realities. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–6.

[13] Hofmeyr, S. A., Forrest, S., and Somayaji, A. Intrusion detection using Sequences of System Calls. *J. Comput. Secur. 6*, 3 (1998), 151–180.

[14] Kong, J. *Designing BSD Rootkits*. No Starch Press, 2007.

[15] Kruegel, C., Robertson, W., and Vigna, G. Detecting Kernel-Level Rootkits Through Binary Analysis. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 91–100.

[16] Lawless, T. St.Michael and St.Jude. *http://sourceforge.net/projects/stjude*.

[17] Loscocco, P. A., Wilson, P. W., Pendergrass, J. A., and McDonell, C. D. Linux kernel Integrity Measurement using Contextual Inspection. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing* (New York, NY, USA, 2007), ACM, pp. 21–29.

[18] Mavinakayanahalli, A., Panchamukhi, P., Keniston, J., Keshavamurthy, A., and Hiramatsu, M. Probing the Guts of Kprobes. In *OLS '06*.

[19] Microsoft. Rootkit Revealer.

[20] Nick L. Petroni, J., and Hicks, M. Automated Detection of Persistent Kernel Control-flow Attacks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), ACM, pp. 103–115.

[21] POUIK, AND YPERITE. Zeppoo. *http://sourceforge.net/projects/zeppoo*.

[22] PRASAD, V., COHEN, W., EIGLER, F. C., HUNT, M., KENISTON, J., AND CHEN, B. Locating System Problems Using Dynamic Instrumentation. In *OLS '05*.

[23] RILEY, R., JIANG, X., AND XU, D. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *RAID* (2008).

[24] RIPTECH. Understanding SCADA system security vulnerabilities. *http://www.iwar.org.uk/cip/resources/utilities/SCADAWhitepaperfinal1.pdf*.

[25] RUTKOWSKA, J. System Virginity Verifier.

[26] RUTKOWSKA, J. Introducing Stealth Malware Taxonomy. COSEINC Advanced Malware Labs.

[27] RUTKOWSKA, J. Subverting Vista Kernel For Fun And Profit. In *SyScan* (2006).

[28] VMWARE. A Performance Comparison of Hypervisors. *http://www.vmware.com/pdf/hypervisor_performance.pdf*.

[29] WANG, Z., JIANG, X., CUI, W., AND WANG, X. Countering Persistent Kernel Rootkits Through Systematic Hook Discovery. In *RAID* (2008).

[30] YIN, H., LIANG, Z., AND SONG, D. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *NDSS* (2008).

Acknowledgements

I would like to sincerely thank Sergey Bratus and Sean W. Smith for their invaluable guidance and support.

APPENDIX

Michael E Locasto's comments at the presentation

General Comments:

1. Number the slides for easy reference.

2. Too much text on many slides.

(a) "Introduction"

(b) "The Magic Trick"

(c) "Features of Intrusion Detection Systems"

(d) "Details of Hookmap and SBCFI"

(e) slide 10, 11, 13, 17, 23, 27

3. Slide 18 should come earlier and not interrupt your exposition of the "Pattern Searching" technique

4. Last bullet point on slide 36 is redundant with itself

5. Signature-based systems do have false positves.

6. Slide 26 is an example of a good concise slide.

7. It would be interesting to think about how to deal with or compensate for "poisoned" training data (what we might *think* are "pristine" kernels)

8. It would be interesting to generalize the data pattern searching as we discussed.

Questions:

1. Why is your approach to using a kernel-level detector not going to be defeated by having a compromised kernel in the first place? How can something reliably measure itself?

2. Is your approach fundamentally new? Why? (single best justification in one sentence). How does it compare to static cryptographic hashes of kernel text sections (i.e., Petroni et al./CoPilot/Komoko)?

3. Can your approach be used to detect other malware like viruses and operational worms? What are its limits?

4. What is the value of an alternative approach that frustrates the installation of these type of rootkits by having large amounts of misdirection or artificial diversity to make the pattern searching take a long time or install itself into a useless "jail" list?

5. What other applications are there for data structure-based search?

6. Are there counterexamples where your approach will give a false positive?

7. Is your approach standalone? That is, does it require the presence or existence of some other deterrence measure that raises the bar to a point where you're detecting new behavior, or a deterrence measure that elminates a blind spot in your approach? Is your system meaningful in the absence of this type of deterrence?

8. What is your measured overhead?

9. Now that you've given the presentation, do you have a better answer as to WHY you're focusing on Type II rootkits that modify control structures? A better answer to Why is that no-one has come up with a good detection mechanism for this type because it is fundamentally difficult to have a good a priori idea of dynamic behavior (not just that they are popular).

10. What future properties of OSs might invalidate your control flow pattern used to characterize anomalies? Object-based OSs? Microkernels?