Cloaker: Hardware Supported Rootkit Concealment

Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, Roy H. Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N Goodwin Ave, Urbana, IL 61801
{fdavid,emchan,jcarlyle,rhc}@uiuc.edu

Abstract

Rootkits are used by malicious attackers who desire to run software on a compromised machine without being detected. They have become stealthier over the years as a consequence of the ongoing struggle between attackers and system defenders. In order to explore the next step in rootkit evolution and to build strong defenses, we look at this issue from the point of view of an attacker. We construct Cloaker, a proof-of-concept rootkit for the ARM platform that is non-persistent and only relies on hardware state modifications for concealment and operation. A primary goal in the design of Cloaker is to not alter any part of the host operating system (OS) code or data, thereby achieving immunity to all existing rootkit detection techniques which perform integrity, behavior and signature checks of the host OS. Cloaker also demonstrates that a self-contained execution environment for malicious code can be provided without relying on the host OS for any services. Integrity checks of hardware state in each of the machine's devices are required in order to detect rootkits such as Cloaker. We present a framework for the Linux kernel that incorporates integrity checks of hardware state performed by device drivers in order to counter the threat posed by rootkits such as Cloaker.

1. Introduction

In order to surreptitiously control a compromised computer, an intruder typically installs software that tries to conceal malicious code. This software is commonly referred to as a rootkit. A rootkit hides itself and some malicious payload from the operating system, users and intrusion detection tools. The techniques utilized by rootkits to avoid detection have evolved over the years. Older rootkits modified system files and were easily detected by tools that checked for file integrity [29, 46] or rootkit signatures [38]. To avoid being detected by such tools, rootkit designers resorted to more complex techniques such as modifying boot sectors [33, 51] and manipulating the in-memory image of the kernel. These rootkits are susceptible to detection by tools that check kernel code and data for alteration [43, 13, 42, 21]. Rootkits that modify the system BIOS or device firmware [25, 26] can also be detected by integrity checking tools. More recently, virtualization technology has been studied as yet another means to conceal rootkits [31, 44]. These rootkits remain hidden by running the host OS in a virtual machine environment. To counter the threat from these Virtual Machine Based Rootkits (VMBRs), researchers have detailed approaches to detect if code is executing inside a virtual machine [20]. Is this the end of the line for rootkit evolution? We believe that other hardware features can still be exploited to conceal rootkits. For example, Shadow Walker [52] exploits the existence of separate instruction and data address translation buffers to hide itself. While Shadow Walker exhibits some weaknesses that allow it to be detected by existing approaches, we aim to show that it is possible to construct a rootkit that exploits changes to hardware state for more effective concealment. Studying the construction of such a rootkit fuels the proactive design and deployment of new countermeasures. Similar approaches have been used in the past by other researchers [31, 44, 15].

In this paper, we present Cloaker, an extremely stealthy proof-of-concept rootkit that exploits ARM processor features to avoid discovery. ARM processors power 90% of mobile handsets shipped today and five billion ARM processors have already been consumed by the mobile device market [6]. With four billion mobile phone subscriptions expected by 2009 [11] and an increasing deployment of ARM-based phones, it is essential that the threat posed by rootkits such as Cloaker be carefully evaluated. In contrast, the number of worldwide PC users is expected to be a little over one billion in 2009, reaching the two billion mark only by 2015 [56]. ARM-based mobile devices from several manufacturers have been broken into in the past and a number of such attacks are documented in online websites

and forums [3, 37]. There are also several mobile device viruses that use Bluetooth connectivity to propagate [15]. However, to the best of our knowledge, Cloaker represents the first exploration of an ARM processor specific rootkit.

In this paper, we demonstrate that most existing techniques used for rootkit detection are ineffective against Cloaker. Cloaker does not leave any detectable trace in the filesystem and is thus invisible to typical intrusion detection tools that scan filesystems. Similar to VMBRs, Cloaker does not modify OS code or data. Therefore, it cannot be detected by integrity checks of the host OS. Cloaker tweaks hardware registers and settings in a manner that allows it to execute without interfering with the existing OS. More specifically, Cloaker exploits an ARM processor setting that allows the interrupt vector to be located in an alternate region of memory. It also exploits hardware functionality that allows several entries in the address translation cache to be locked down and not be automatically removed using the typical approaches used by operating systems to flush it.

Most existing rootkits rely on the host OS for services such as networking, process management, memory management and other subsystems. Since Cloaker aims to leave the host OS untouched, it cannot use these services. This does not diminish the effectiveness of Cloaker. The design of Cloaker illustrates how malicious users can construct powerful rootkits even without hooking into the host OS. Cloaker provides a small and self-contained environment for malicious code; it provides services such as scheduling, networking and memory management. In essence, Cloaker is a malicious and hidden micro-OS environment that coexists with the existing OS on the device.

As a consequence of the decision to not modify any existing code on the system, Cloaker is a non-persistent rootkit and does not remain in the device after a restart. Given that most mobile phones are restarted very infrequently, we believe that Cloaker can still pose a serious threat to the device while it is powered on. A similar argument is also made by the authors of other rootkits [44, 19]. As we shall see later (in Section 8), non-persistence significantly reduces the detectability of rootkits.

Cloaker represents a single instance of a possible new class of rootkits which rely on architecture and device specific features to hide themselves. Rootkits such as Cloaker can be effectively discovered and disabled by code that has in-depth knowledge of hardware features and settings. Such code is typically present only in device drivers. Therefore, in order to detect rootkits like Cloaker, we advocate the design of a framework for OS device drivers that can check the integrity of their associated hardware. We believe that generic software-based rootkit detection approaches and third party antivirus tools running in user space are weaker than hardware state integrity checks performed by device drivers.

Our contributions in this work include

1. Exploration of the threat posed by rootkits that modify hardware settings for concealment.

2. Demonstration that a rootkit and malicious payloads which do not rely on the host OS for any services can be easily constructed within a small memory footprint.

3. Detection techniques for such advanced rootkits.

We have implemented Cloaker on the Texas Instruments OMAP1610 H2 mobile device development platform [54]. The OMAP1610 is a system-on-chip (SoC) design and is powered by an ARM926EJ-S processor core [5]. Our experiments with Cloaker use Linux as the host kernel. While many manufacturers are now shipping Linux-based mobile phones, there are other phone operating systems in widespread use as well. The vulnerabilities highlighted by Cloaker, however, may also be applicable to these other operating systems. In this paper, we focus on the ARM architecture. Similar attacks may also be possible on other architectures.

The remainder of this paper is organized as follows. We illustrate the role of a rootkit in an attacker's arsenal and describe defenses against intrusion in Section 2. In Section 3, we look at the evolution of rootkits in more detail, motivating the need to look ahead and study possible advances in rootkit technology. We present the design and implementation of Cloaker in Section 4. Section 5 describes several malicious payloads that we were able to build using Cloaker. The limitations of Cloaker are addressed in Section 6. We discuss some size and performance aspects of Cloaker in Section 7. Section 8 classifies existing rootkit detection approaches and evaluates their effectiveness against Cloaker. In Section 9, we present a framework for the Linux kernel that can be used by device drivers to check the integrity of hardware state as a countermeasure against Cloaker-like rootkits. Finally, we conclude in Section 10.

2. Intrusion, Rootkits and Defenses

Before we start discussing rootkits in detail, it is important to understand the role played by rootkits in a malicious attacker's arsenal. Figure 1 illustrates the use of a rootkit in a typical attack scenario. The attacker starts off with the goal of compromising a target machine. Once that goal is attained, the next step is to avoid detection. This is achieved by first removing all traces of intrusion. The attacker then installs a rootkit to hide malicious activity on the machine. Attackers typically use rootkits in conjunction with payloads such as keyloggers, spam relays and bots. The rootkit may also provide a covert means for the attacker to access the machine: usually referred to as a "backdoor".

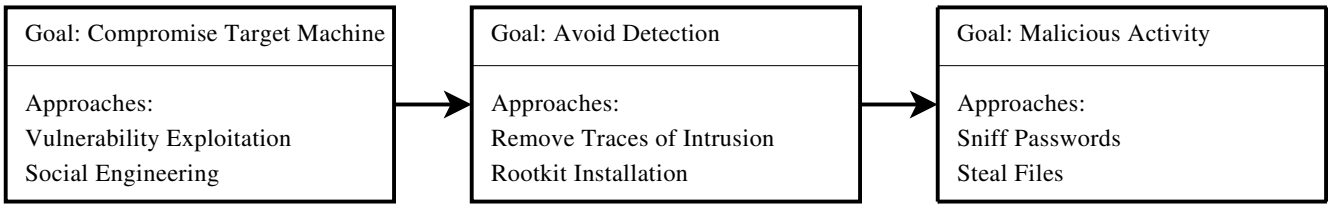

Figure 1. Role played by a rootkit during a typical attack

The first defensive wall against attackers is usually a combination of intrusion prevention mechanisms. Detection techniques are the next defensive wall and are necessary when prevention techniques fail. We address the specific problem of rootkits and their detection in this paper. Once an intrusion has been detected, there is also the associated problem of recovering the system to a clean state. This is addressed by recovery techniques.

Some examples of intrusion prevention techniques are use of a firewall, scanning code for viruses before execution, sandboxing techniques such as SELinux [35] and MAPbox [1], and control flow enforcement as used in Program Shepherding [32] and SecVisor [48]. Some examples of tools that aid intrusion analysis and recovery are BackTracker [30], the Repairable File Service [57] and Taser [22]. Recovery techniques usually rely on logs of system activity in order to restore legitimate files and mitigate the effects of the intrusion. There is a large body of work in intrusion prevention and recovery and a detailed discussion of these topics is outside the scope of this paper.

We assume the existence of vulnerabilities that allow for rootkit deployment. This paper focuses on the hiding and operational aspects of rootkits. While we discuss some possible malicious usage of rootkits in Section 5, addressing this aspect in additional detail is also outside the scope of this paper.

3. Rootkit Evolution

The sophistication and creativity in the construction of rootkits has seen significant increases over the course of time. This evolution is primarily driven by advances in rootkit and malware detection technology.

3.1. User Mode

Rootkits were originally designed as user space code that patched or replaced existing applications and provided a cover for malicious activity. For example, system programs (like `ps` and `ls` on Unix) are changed so that they do not reveal the presence of certain malicious processes or files. The hidden processes typically include some malicious payload. A significant number of such rootkits were collected and studied by Grizzard [24]. Because these rootkits usually create new files or modify existing files on the system, they are easily detected by filesystem integrity checking tools and signature checking tools. Integrity checking tools such as Tripwire [29] ensure the integrity of existing system files and folders. Signature checking tools search for traces of known malware in the system. Signature checking is used by chkrootkit [38] to scan for known rootkit modifications to system binaries. This approach is also widely adopted by most antivirus products in the market today.

3.2. Kernel Mode

To avoid detection by user space tools that scan the filesystem, rootkit authors resorted to modifying the OS kernel. Such kernel modifications are usually performed using loadable modules. A popular technique used by kernel mode rootkits is called "hooking". This involves modifying kernel instructions or function pointers to change control flow and run rootkit code. For example, some rootkits modify the results of system calls by subverting the control flow of the system call mechanism. This system call hijacking can be used to implement a filter constructed by the attacker. The filter hides the presence of malicious files, processes and sockets. Direct Kernel Object Manipulation (DKOM) is another approach used by modern kernel mode rootkits [19]. This avoids modifications of kernel code and relies only on modifications to kernel data in order to hide malicious processes.

Several kernel mode rootkits have been collected and analyzed by security researchers [24, 16]. These rootkits are harder to detect than user mode rootkits because they can thwart detection attempts from user space tools. Some of these rootkits are detectable by signature scanning or integrity checks of kernel code and critical kernel data such as system call tables and interrupt descriptor tables. In order to evade signature checking tools, rootkit authors have resorted to writing code that is polymorphic and changes its signature frequently. For example, this approach is adopted by newer versions of the Rustock rootkit [16]. Checks for

anomalous kernel behavior can also help reveal the presence of such rootkits. Researchers have investigated performing such checks from within a virtual machine monitor [21, 8].

3.3. Firmware

As kernel mode rootkit detectors improved, malware researchers began to explore more advanced ways for rootkits to evade detection. Researchers showed that rootkits could hide by modifying firmware in the system such as the ACPI BIOS [26] and the PCI BIOS [25]. Such rootkits can be detected by the Trusted Computing Group's Trusted Platform Module (TPM) [53] technology which checks the integrity of the OS image and all the firmware in the system. TPM chips are already available on a large number of PCs being sold today. Additionally, existing firmware-based rootkit prototypes are only useful for persistence. They still hook the kernel or use DKOM techniques once the kernel is booted. This makes them vulnerable to the same defenses used against kernel mode rootkits.

3.4. Virtual Machine Based Rootkits

Virtual Machine Based Rootkits (VMBRs) are yet another step in the evolution of rootkits. The intuition behind this approach is that virtual machine monitors (VMMs) are normally designed to be transparent to guest operating systems and this property can be used to hide a rootkit. VMBRs hide their presence by running the existing host OS inside a virtual machine. Virtualization protects the rootkit from detection tools running in the subverted guest machine. SubVirt [31] was the first prototype rootkit to demonstrate this concept. Hardware support for virtualization, such as Intel's VT [55] and AMD's SVM technology, can also be exploited to construct rootkits. The Blue Pill [44] rootkit leverages virtualization hardware support to better conceal itself and lower the performance overhead and memory footprint. Nevertheless, several detection approaches based on discrepancies between virtual hardware and physical hardware have been proposed to reveal the presence of VMMs and VMBRs [20].

3.5. Hardware Configuration

The detectability of rootkits is generally low when they do not modify any existing code or data in the host OS. This observation is exploited by VMBRs. We believe that the next step in rootkit evolution is to exploit the configuration and state of hardware in order to hide. Modern processors and peripheral devices incorporate many hardware configuration settings that govern their operation. These settings and other hardware features that are unused by the host OS may be manipulated in order to support rootkits.

The Shadow Walker [52] rootkit, for example, avoids detection by providing a different view of memory to integrity checking utilities. It does this by hooking the page fault handler of the OS and inserting inconsistent entries in the instruction and data Translation Lookaside Buffers (TLBs). Shadow Walker, however, exhibits a weakness similar to kernel mode rootkits and is prone to detection because it overwrites the OS page fault handler [52]. Cloaker also uses manipulation of hardware settings to hide, but goes one step further and completely avoids any changes to the host OS code or data.

4. Cloaker Design and Implementation

In this Section, we present the approach used by Cloaker to conceal itself. We also discuss some issues with its installation into the host OS. We show that it is possible to provide several services within Cloaker and describe a version of Cloaker that we have written for Linux as a demonstration.

4.1. Staying Hidden

Cloaker leverages two specific hardware features on ARM platforms to remain hidden from currently deployed rootkit detection techniques. The first feature is the ability to change the location of the interrupt vector by flipping a bit in a processor configuration register. On the ARM926EJ-S processor [5], this is bit 13 of coprocessor CP15 register C1. The two possible locations for the interrupt vector are at 0x00000000 and 0xFFFF0000 in virtual memory. Most operating systems for the ARM typically set this bit so that the interrupt vector is located at the high address. The page at 0x00000000 is marked as invalid. This helps to catch null pointer dereferences in OS code. This approach is adopted by the ARM versions of several open-source operating systems such as Linux, L4 [34] and Choices [17]. By changing the coprocessor bit, Cloaker is able to install a different interrupt vector at 0x00000000. This vector acts as a trampoline and allows Cloaker to assume control whenever an interrupt is triggered. This is illustrated in figure 2. A primary goal in the design of Cloaker was to remain hidden from the host OS without modifying any part of it. This interrupt interception approach provides Cloaker with periodic control over the processor and, therefore, obviates the need to hook OS code or data structures. An additional benefit of this approach is that the processor is always switched to a privileged mode when an interrupt is received. This grants Cloaker complete control over the processor.

The second ARM processor feature that is exploited by Cloaker is the ability to lock down entries in the address

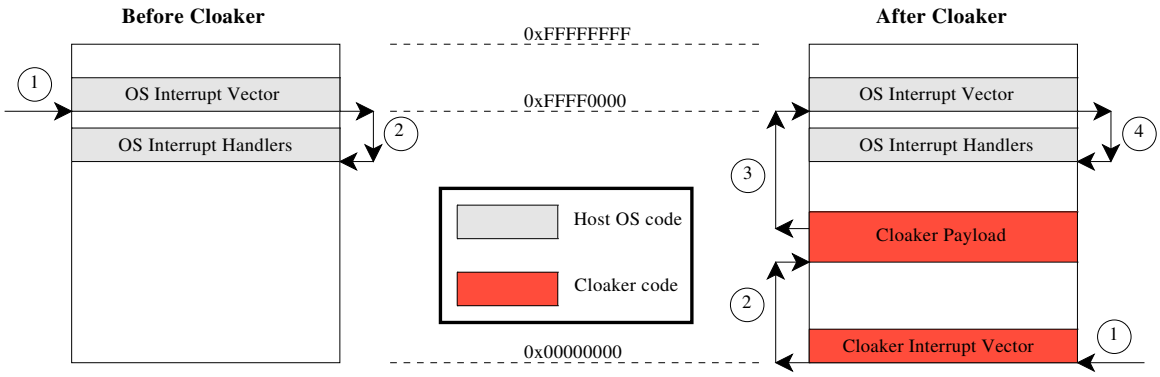

Figure 2. Cloaker operation through interrupt interception: The virtual memory map on the left shows the original system and the normal control flow for an interrupt. The one on the right shows the hijacked control flow after Cloaker is installed.

translation lookaside buffer (TLB). Locking an entry prevents it from being removed by the typical OS code that flushes the TLB upon address space changes. This essentially allows Cloaker to map in arbitrary memory regions into the virtual address space of the processor without modifying any portion of the OS-managed page tables. Cloaker places a TLB lockdown entry for a 4KB page at virtual address 0x00000000 in order to hide its trampoline interrupt vector. Cloaker places additional lockdown entries to hide its payloads. The ARM926EJ-S processor core has a unified data and instruction TLB. An 8 entry fully-associative portion of this TLB is used exclusively for holding locked down entries. Thus, Cloaker's usage of locked down TLB entries does not diminish the number of normal entries available to the host OS and cause related performance problems. The TLB entry lockdown feature is not always fully utilized by most operating systems and thus presents a covert means for hiding memory mappings. Since the ARM TLB cannot be read by the OS, detection tools cannot simply scan for the presence of malicious TLB entries.

Cloaker's usage of virtual memory at 0x00000000 breaks the semantics of null-pointer dereferences that occur inside the OS kernel. This is not an issue because such accesses only happen when there are errors inside the kernel. Current detection systems do not check for the validity of memory at addresses that the kernel expects to be invalid. Additionally, because Cloaker only executes when the processor is in a privileged mode, the mappings inserted by Cloaker are configured to allow access only from privileged processor modes. This ensures that the behavior of null-pointer dereferences in user space applications remains unchanged. This also ensures that user space detection tools which scan for malware signatures have no visibility into Cloaker.

The ARM processor automatically suspends further interrupts whenever an interrupt is received. Since Cloaker and its payloads execute only on interrupt handling paths, the host kernel or other device interrupts cannot preempt or interfere with Cloaker's activities. Also, as long as Cloaker executes payload code only on unpredictable (as far as the kernel is concerned) interrupts, such as interrupts from a keyboard or a network device, it does not raise any timing-related alarm flags in the kernel. For this reason, Cloaker does not interpose payload code on the host kernel's timer interrupt path. All timer interrupts are passed through to the host kernel using a short control path consisting of only a couple of instructions.

4.2. Installation

A rootkit installer is commonly referred to as a "dropper". Any vulnerability in the host OS that allows arbitrary code execution in a privileged mode of the ARM processor can be exploited by a dropper to install Cloaker. Kernel code is usually executed in some privileged processor mode. If the attacker gains superuser privileges on a system such as Linux, Cloaker can be installed by loading a dropper packaged as a kernel module. The module should be subsequently unloaded and deleted to remove traces of the intrusion. This approach may not be desired if kernel module insertion events are logged. Yet another approach to build a dropper would be to directly access kernel memory and inject rootkit code [47].

Cloaker needs some physical memory pages to store the trampoline interrupt vector and any malicious payloads. One possibility is to usurp unused memory on peripheral devices such as video chips, network chips or other peripherals. Memory-mapped NOR flash chips that are commonly

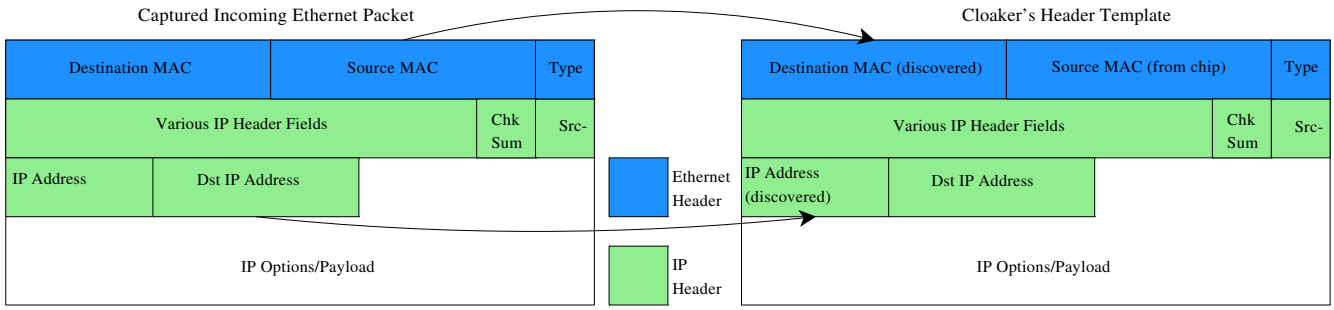

Figure 3. IP Networking in Cloaker: Creating a reusable network packet header

found on mobile devices support direct execution of code stored on them. This may also be used by rootkits. Some ARM SoC designs incorporate a small amount of on-chip internal RAM which can be used for fast program execution. If the host system does not make use of this memory, it can instead be used by Cloaker. A suitable alternative needs to be chosen depending on the target system. It should be noted that this is a one-time task and is only required for Cloaker's installation.

4.3. System Services

As noted previously, an important goal of Cloaker is to avoid detection by not performing any modifications to the host OS environment and not using any of its services. This means that Cloaker must support similar services that can be used by writers of malicious code. Since Cloaker does not aim to be a full-fledged OS, these services can be simple and can have a small memory footprint. Cloaker, therefore, provides a micro-OS environment for malware payloads. A related effort in providing small execution environments is TinyOS [27]. Cloaker services, however, need to address the additional challenge of staying hidden from the host OS. We discuss three representative services in this paper: memory management, interrupt management and network communication.

4.3.1. Memory Management

Cloaker includes a memory manager that uses the buddy memory allocation algorithm to provide dynamic memory allocation and deallocation services. This component is only designed to manage allocations within a contiguous range of memory and it does not support any memory protection semantics. The smallest possible allocation is configurable and can be as low as 8 bytes. The memory manager component is optional and is not required if payloads can manage their own memory usage using fixed compile-time data structures.

4.3.2. Interrupt Management

Cloaker includes an interrupt management library which encapsulates the interaction with interrupt hardware. It supports functionality such as lookups of the current interrupt number, enabling and disabling specific interrupts and triggering interrupts from software. It also incorporates an interrupt dispatching framework that handles registration of interrupt handlers and calls them when interrupts are received. This library can be used by malware such as keyloggers to intercept keyboard interrupts.

4.3.3. Networking

Cloaker uses direct interaction with the network chip in order to provide a network communication service. In order to do this, Cloaker uses a private software driver for the chip that provides a packet receive and packet send API. Cloaker supports parsing of Ethernet and IP headers, but does not include a complete network stack. Nevertheless, Cloaker is able to establish IP network communication with arbitrary external entities using a discovery process that determines two important parameters: the host's IP address and the gateway's MAC address. The discovery process examines random packets from the network until a valid incoming packet from an external machine is received. The last hop for an incoming packet from an external machine that is not on the same subnet is typically between the network gateway and the host system. Therefore, the host's IP address is obtained from the destination IP address in the packet and the gateway's MAC address is the same as the source MAC address in the packet. This whole network initialization in Cloaker is OS-independent. It may be possible to obtain these directly from the host kernel memory, but this is kernel dependent as it would require knowledge about kernel data structures.

Once the host's IP and gateway's MAC addresses are known, a pre-fabricated Ethernet packet header template is filled in with these values. This is illustrated in figure 3. Note that the source MAC address can be obtained from the

network chip. The destination IP field can be set to any IP target. This pre-fabricated header can be now be used to transmit an arbitrary number of packets because commonly used IP headers do not incorporate any replay prevention mechanism. Cloaker does, however, recompute the packet checksum in the IP header for every packet that it sends. This is because routers usually drop packets with invalid checksums.

Cloaker's networking support completely bypasses the host kernel driver and network stack. We currently use a significantly stripped down version of the Linux Ethernet driver for the SMC91X chip on the OMAP1610 H2 board. Cloaker's modified driver reads incoming packets from the chip but does not clear the packets from the chip's buffers. This allows intercepted packets to be also correctly received by the host kernel's driver.

4.4. Reducing the Amount of Code

Cloaker's implementation attempts to achieve a small memory footprint. A key observation that significantly helps reduce the amount of rootkit code required to drive devices and provide services is that part of this job may have already been performed by the host kernel. For example, device drivers in Cloaker do not need to include initialization code for devices that are already initialized by the kernel. It is possible to achieve further reductions in code size by adopting a technique that reuses existing kernel code at runtime without tainting kernel data. It is important to ensure that kernel data is not modified because this opens up the rootkit to detection. We start by identifying kernel code that operates only on arguments passed in through registers rather than by looking up and using global variables. Such code can be directly called from within the rootkit with arguments that point to rootkit data instead of kernel data. Thus, we are able to run kernel code and not affect kernel data. This, however, makes the rootkit closely tied to the kernel.

The need to rewrite OS-like code in Cloaker to support malware is a limitation and is the price paid for avoiding detection by OS integrity checkers. When developing Cloaker, we discovered a means to simplify the writing of rootkit code and reduce the burden on programmers. While the discussion in the previous paragraph focused on execution time reuse of kernel code, we also managed to reuse kernel code during compile time by building Cloaker in the build environment for the kernel. During the development of Cloaker for Linux, we were able to reuse a large number of Linux utility macros and support functions by building the rootkit as part of the kernel build process. It should be noted that we are not building Cloaker into the kernel; we are only reusing kernel headers and libraries to build Cloaker as an independent executable. Also, while execution-time code reuse helps reduce the memory footprint, compile-time code reuse primarily helps reduce programmer effort.

4.5. Cloaker for Linux

We wrote Cloaker for kernel version 2.6.23.1, which is the latest stable version of the Linux kernel as of this writing. The majority of Cloaker's code is written in the C language. A small section of the code that deals with the initial interrupt handling is written in assembly. Our current insertion strategy for Cloaker is to use superuser privileges to load a kernel module which is unloaded immediately afterward. More untraceable insertion techniques could be used, but this topic is not the focus of this work.

The Cloaker installer in the module first disables interrupts to avoid any interference. In order to insert the trampoline interrupt vector and payload, some physical memory is required that is not used by Linux. We currently request free pages from the kernel and translate them into physical pages for use by the rootkit. When the module is unloaded, these pages are not deallocated. To the kernel, this process essentially looks like a memory leak. It should be noted that, unlike rootkits that use DKOM, this does not alter the functional aspects of the kernel and can therefore, remain unnoticed. Stealthier alternatives to obtain memory were presented earlier in this Section. Cloaker then inserts entries in the TLB lockdown region. It installs the trampoline interrupt vector and its payload before re-enabling interrupts.

5. Malicious Payloads

In this Section, we show (by construction) that Cloaker services can be used to carry out several typical malicious activities. Malicious payloads generally perform one or more of the following activities: intercepting I/O operations, stealing information from the host, covertly receiving commands or transmitting data. When using Cloaker, malicious payloads need to conform to the same stringent policy of not modifying OS code or data in order to avoid detection. Additionally, since all payloads are executed in processor interrupt context with further interrupts placed on hold, they should complete execution quickly at every invocation. This avoids delaying timing sensitive interrupts such as the host kernel timer.

In the previous Section, we described how Cloaker establishes covert communication with external entities using direct access to network devices. Malicious payloads can use similar direct access to other devices in order to intercept communications or steal information from the host. For example, direct device access can be used to intercept keystrokes or obtain sensitive information from flash

memory or disks. Alternatively, payloads can snoop on the buffers used by the OS to interact with these devices.

We used a combination of Cloaker services and direct device access to construct three malicious payloads: a keystroke sniffer (Keylogger), a network distributed denial-of-service attack tool (DDosIP) and a data stealer (Informant). These payloads are written in the C programming language.

5.1. Keylogger

We built a keylogger that captures all user input received through a serial port on the OMAP1610 H2 board. The serial port device is a National Semiconductor 16550 UART, which is also found in most modern PCs. It has dual 64 byte transmit and receive FIFO buffers. The keylogger registers with Cloaker to receive the interrupts from the serial port. Whenever it sees any data in the receive buffer, it reads and logs the data in local memory. However, the act of reading data from the serial port automatically empties the read-only receive buffer and prevents the input from being sent to the OS. In order to solve this problem, we exploit a debug setting in the UART chip that loops back the transmit lines to the receive lines. We enable loopback, write out the logged keystrokes and then disable loopback. This ensures that the input is received by the OS as expected.

Many devices such as GPS receivers, RFID readers and external input devices also interact with the kernel through serial interfaces. Our keylogger can be used to intercept traffic between these devices and the OS to capture sensitive information.

Another possible implementation of a keylogger would be to directly sniff OS input buffers. This is, however, OS-specific and it is possible that the keylogger fails to capture some keystrokes. This scenario could happen if the input were consumed by the host OS before the keylogger could execute.

5.2. DDosIP

A Distributed Denial-of-Service (DDoS) attack on a target IP address is performed using a large number of compromised machines, each of which can be programmed to send out a low bandwidth attack in order to avoid raising alarms at the sources. For mobile devices using wireless connectivity, the low bandwidth usage also helps ensure that battery life and network performance are not significantly disrupted.

The DDosIP payload uses Cloaker's interrupt management services to enable an unused hardware timer on the OMAP1610 and registers to receive the interrupts from this timer. This allows it to periodically send a packet when it is invoked by the interrupt handler. DDosIP makes use of the networking support in Cloaker and can fill a packet with arbitrary content before sending it over the network. The bandwidth of the attack is configurable by modifying the interval between timer interrupts.

5.3. Informant

The Informant payload uses Cloaker network services to communicate with an external machine. In this case, the goal is to send out sensitive information from the host. Informant can be statically configured with the IP address of the machine to which the packets are sent. A custom tool at the receiver is used to parse incoming packets and present the contents of messages to the attacker. We currently use Informant to send out the logs from the keylogger payload whenever its buffer is full. One can envision other attacks which use Informant; for example, stealing data from flash memory or host kernel memory.

It is important to remember that packets sent through the network by Informant or the DDosIP payload can raise suspicions and can be detected by external network-based intrusion detection systems that snoop on traffic to and from the device. It may be possible to use covert channel techniques to bypass such detectors.

6. Limitations

Since Cloaker exploits hardware features to hide, it is necessarily closely tied to the hardware platform. Features and settings can change between processor versions resulting in the need for possible code or design changes. New device drivers would have to be written for Cloaker if the chipsets on the platform are changed. Unlike generic rootkits that are mostly tied only to the OS, Cloaker is inherently less portable across hardware.

To avoid detection, Cloaker does not use the services provided by the host OS and does not attempt to modify the behavior of the host OS to hide information such as processes and files. Instead, Cloaker provides its own set of services. Malware writers are forced to use a non-familiar and highly constrained operating environment to write payload code. Therefore, unlike VMBRs which allow malware writers to use arbitrary code and still avoid detection, rootkits like Cloaker are more difficult to use.

Eschewing the use of OS services does not imply that Cloaker is OS-independent. While developing Cloaker and experimenting with various configuration options in the Linux kernel, we discovered that enabling kernel support for handling memory access alignment faults disables Cloaker. On the ARM architecture, accesses to memory words must be aligned to 32 bit boundaries. Alignment faults allow the kernel to emulate non-aligned accesses correctly. Further investigation revealed that, once this func-

tionality was enabled, the configuration register with the bit that Cloaker relies on for interrupt redirection was being reset every time a user mode program was interrupted. This is because another bit in the same register determines whether alignment faults are generated and Linux keeps overwriting the register to manipulate it. This illustrates that rootkits such as Cloaker are closely tied to the OS and its configuration.

7. Evaluation

In this Section, we evaluate Cloaker and our sample malicious payloads in terms of size and performance.

7.1. Size and Code Complexity

Cloaker is an extremely small rootkit. Our payloads also have a small memory footprint. Table 1 shows the number of lines of code in each component and the corresponding compiled size. The size of Cloaker includes the trampoline vector, private stack, an interrupt management library, networking support, and a set of minimal device drivers. The numbers in the table show that we are able to write extremely small and compact code that can support several malicious activities using Cloaker. In particular, the small sizes of the payloads are due to the fact that they are able to efficiently utilize Cloaker services through simple function calls. The current code is also not optimized for size. It may be possible to achieve further reductions in size through aggressive code optimization.

We were able to write all our payload examples without the use of the optional memory management library. This library may be required for more complex payloads that require dynamic memory allocation and deallocation facilities. We list this library's size in the table because it illustrates that a reasonably small dynamic allocator can also be incorporated if desired.

The total size in bytes for all the components (installer, rootkit and payload) without the optional memory management component is 5036 bytes. When packaged as a Linux kernel module, additional code required by the module loader mechanism boosts the size to 28,470 bytes. Once installed, Cloaker, its payloads, its stack and data all fit comfortably within the first two 4 KB pages of virtual memory.

Table 1. Sizes of Cloaker and payloads

Component	Lines of Code (C semicolons)	Compiled Size (bytes)
Cloaker	147	3544
MemoryManagement	101	2008
Keylogger	34	384
DDosIP	12	256
Informant	11	220
Installer/Dropper	63	632

7.2. Performance

Since Cloaker intercepts all processor interrupts, and occasionally executes payload code, system performance is affected. In order to measure changes in performance, we use the lmbench [36] benchmarking suite (version 2) for Linux. The benchmarks were performed using version 2.6.23.1 of the Linux kernel, and with the ARM core clocked at 120 MHz. All performance numbers in this Section are reported as an average of 5 independent measurements with error estimates provided by the sample standard deviation.

Table 2, Table 3 and Table 4 show Cloaker's effects on the benchmarks gathered by lmbench. For more details about each specific benchmark, the reader is referred to the lmbench documentation. In each table, the first row of results lists the measurements for Linux without the rootkit. The second row corresponds to Linux with Cloaker installed. In this configuration, Cloaker is intercepting all interrupts to the system, but it is not running any payloads. The third row corresponds to the case when Cloaker is being used to send attack packets over the Ethernet using the DDosIP payload. The attack bandwidth for all experiments is a 2 KB/s data stream.

Table 2 illustrates the effects of Cloaker on user-space processes interacting with the OS through several standard system calls. The benchmark measurements show that Cloaker's installation has very little effect on the performance of the host OS, even while running the low-bandwidth DDosIP payload. In all these experiments, there is no user input and therefore, the keylogger payload is not involved.

Table 3 shows the effect on context switching time between different numbers of processes with different working set sizes. Some additional latency and bandwidth measurements reported by lmbench are shown in Table 4. Latencies are shown for pipe and Unix socket communication and for servicing a memory protection and page fault. Bandwidth measurements are reported for pipe and Unix socket communication and file access. Again, the performance numbers in all three rows are comparable. This suggests that existing rootkit detection suites that rely on OS performance changes may not be able to easily discern the presence of Cloaker.

We use the *ttcp* network benchmark tool to measure the impact of Cloaker and the DDosIP payload on network bandwidth. These experiments were performed using the OMAP1610 H2's 10 Mbps Ethernet controller. Bandwidth

Table 2. lmbench: Process operations - times in microseconds (smaller is better)

	null-call	null-I/O	stat	open-close	sig-inst	sig-handle	fork-proc	exec-proc
Linux	1.21 ± 0	7.5 ± 0.7	31.4 ± 1.3	65.3 ± 2.4	15.3 ± 2.0	26.4 ± 0.2	7721 ± 178	8958 ± 33
Linux+Cloaker	1.34 ± 0	8.1 ± 0.1	31.4 ± 0.9	69.9 ± 3.1	13.9 ± 0.6	27.9 ± 0.6	7726 ± 100	9028 ± 33
Linux+DDosIP	1.34 ± 0	8.1 ± 0.2	32.3 ± 1.4	68.1 ± 2.4	15.2 ± 2.2	27.8 ± 0.3	7847 ± 206	9026 ± 66

Table 3. lmbench: Context switching times in microseconds (smaller is better)

	2p-0K	2p-16K	2p-64K	8p-16K	8p-64K	16p-16K	16p-64K
Linux	332.2 ± 0.9	659.9 ± 4.6	1295.1 ± 10.4	783.7 ± 5.3	1429.2 ± 10.0	781.2 ± 4.4	1417.5 ± 6.10
Linux+Cloaker	339.2 ± 3.4	669.4 ± 5.3	1305.8 ± 6.90	796.6 ± 7.5	1441.1 ± 17.7	791.5 ± 6.8	1435.3 ± 7.70
Linux+DDosIP	338.3 ± 3.2	665.6 ± 6.5	1317.7 ± 28.3	786.9 ± 3.1	1424.0 ± 15.1	790.3 ± 2.9	1428.3 ± 10.7

Table 4. lmbench: Latencies (microseconds, smaller is better) and local I/O bandwidth (MB/s, larger is better)

	Latencies				Bandwidth		
	Pipe	AFSocket	ProtFault	PgFault	Pipe	AFSocket	FileOpenClose
Linux	729.2 ± 2.00	1341 ± 125	2.6 ± 0.7	167.4 ± 3.7	9.2 ± 0.3	9.6 ± 0.4	15.6 ± 0
Linux+Cloaker	740.5 ± 2.00	1314 ± 108	1.1 ± 0.3	172.4 ± 6.1	9.4 ± 0.2	9.8 ± 0.1	15.5 ± 0
Linux+DDosIP	720.9 ± 44.1	1364 ± 132	1.1 ± 0.5	172.6 ± 0.9	9.5 ± 0.2	9.9 ± 0.1	15.5 ± 0

Table 5. Network bandwidth measurement with ttcp (KB/s, larger is better)

	MTU=1500	MTU=500
Linux	774.67 ± 3.37	566.99 ± 2.23
Linux+Cloaker	771.42 ± 5.43	559.62 ± 2.60
Linux+DDosIP	742.47 ± 6.93	515.20 ± 5.45

is measured using a 16 MB data download to the device through a TCP connection. Table 5 shows the results when the maximum transmission unit (MTU) is set to 1500 bytes (Ethernet default) and when the MTU is set to 500 bytes (dialup default). Installing Cloaker causes a very small drop in bandwidth which is more pronounced for the smaller MTU. When running the DDosIP payload, the bandwidth drops more significantly. The bandwidth drop is due to some contention for buffers on the network chip and some performance issues with our network chip driver (when the host driver is also using the chip).

We also measured the additional delay introduced by the keylogger which intercepts keyboard input to the OS. Our measurements show that the overhead of this operation is 46 nanoseconds. Human users cannot perceive this delay in keyboard input processing.

The measurements in this Section show that Cloaker's techniques minimally affect critical operations in the host kernel.

8. Existing Rootkit Detection Techniques

Cloaker was designed to demonstrate that rootkits using hardware-supported concealment can remain undetected by currently available approaches. This is because few existing detection techniques attempt to check the integrity of hardware state. In this Section, we examine this aspect of Cloaker in additional detail. Existing tools use integrity checks, signature checks, VMM presence checks and scans of physical memory using hardware in order to detect rootkits.

8.1. Classification

8.1.1. Integrity Checks

Integrity checker tools for rootkit detection can be classified into three types: file integrity checkers, platform integrity checkers and kernel integrity checkers. File integrity checking tools periodically compute hashes of system files that are not expected to change and compare these hashes with their original values. Some examples of such tools are Tripwire [29] and Samhain [46].

AEGIS [4] and Trusted Platform Module (TPM) [53] technology can be used to detect rootkits that modify the OS or firmware. The TPM contains special Platform Configuration Registers (PCRs) that reflect the configuration of the machine. These PCRs contain a checksum of all code that is used to boot the machine, including the BIOS, firmware, bootloader and kernel. If the BIOS or peripheral firmware is modified, the TPM's PCR value would not match, thereby detecting the alteration.

In order to detect rootkits that perform on-the-fly modifications to code or data in the running kernel, integrity checks can also be applied to the kernel. Kernel code and immutable data can be checksummed in order to detect alteration. This operation could also be performed using hardware to prevent interference from malware. For example, CoPilot [40] is a PCI card which directly reads kernel memory using DMA and uses hashes to ensure that the integrity of the kernel is not compromised. Some newer ARM chips include a similar hardware component called the Run-Time Integrity Checker (RTIC) [7]. Performing such checks using specialized hardware has the additional benefit of freeing up the processor to do other work and thus reduces overheads. Another technique is to use performance counters on the processor, if available, to check if unexpected extra code is being executed.

Checking the integrity of dynamically changing kernel data is a challenging problem. The difficulty in performing such checks is exploited by rootkits that use a technique called Direct Kernel Object Manipulation (DKOM). Since rootkits that use DKOM do not interpose any code on the execution path of the kernel, detection techniques that just check for changes to kernel code cease to work. In order to deal with this problem, researchers have developed several techniques that check for suspicious behavior of the kernel.

Virtual machine introspection [21] provides significant visibility into the OS being protected and can be used to detect kernel mode rootkits. For example, the kernel process table can be examined by a VMM and compared against the process list reported by a user space program. Memory writes to critical kernel structures may be controlled. In general, the kernel can be scanned for a number of telltale signs of intrusion.

Strider Ghostbuster [10] uses a technique called cross-view diff to detect rootkits. Similar to some of the detection techniques possible with virtual machine introspection, this involves comparing information from two vantage points. For example, inconsistencies between the raw registry files on disk and information reported by querying the registry through kernel APIs signal the presence of a rootkit. While Strider Ghostbuster is not publicly available, Microsoft has released RootkitRevealer [12] which uses a similar approach. RootkitRevealer, however, does not detect non-persistent rootkits because it focuses only on hidden files and registry entries on disk. Klister [43] reads internal kernel data structures in order to reveal any hidden processes in the system.

VICE [13] and Patchfinder [42] check for rootkits that hook into control flow paths in the kernel. Other related techniques include proving integrity to a remote party (remote attestation) as used in SWATT [50] and Pioneer [49] and detecting malware using a semantics-based approach [41].

8.1.2. Signature Checks

While integrity checking tools attempt to discover any alterations to the host system, signature checking tools scan for the presence of known malware. The benefit of signature checks is apparent in dynamic environments where integrity checks are difficult or infeasible to perform. This approach is used by chkrootkit [38] and many commercial antivirus tools to identify rootkits that modify files on the system in known ways. Signature checking tools are heavily dependent upon their database of signatures which needs to be constantly updated in order to protect against newer threats. The inability to detect new malware for which signatures are unavailable is the biggest drawback of this approach.

In addition to scanning the disk, physical memory can also be scanned for malware signatures. Again, hardware can be used to avoid reliance on the correctness of the host kernel. Tribble [14, 23] is a PCI card that can be used to save a copy of physical memory for analysis. Firewire ports have also been used to access system memory [9]. An advantage of using hardware-based scanners is the ability to check the entirety of physical memory (as opposed to just the kernel virtual address space) for malware signatures.

Unfortunately, it has already been demonstrated that such hardware-based memory scanning tools can be deceived [45]. It is interesting to note that the demonstrated attacks against hardware-based memory scanners also make use of hardware configuration manipulation (like Cloaker): in this case, on AMD64 hardware. Yet another attack against signature scanning tools is adopted by malware code that is designed to mutate itself (polymorphic malware).

8.1.3. VMM Detectors

VMBRs are susceptible to a variety of detection techniques that check for hardware and timing discrepancies [20, 18]. Hardware discrepancies arise due to the need for creating a virtual hardware environment for the host OS. Timing discrepancies arise because some operations take different amounts of time on native hardware when compared to virtual hardware. Garfinkel argues that it may not be worth it to even write VMM-based malware because it essentially reduces the difficult problem of detecting rootkits to the easier problem of detecting unexpected VMMs [20].

Table 6. Effectiveness of existing rootkit detection techniques

	Rootkits						
	User Mode		Kernel Mode		Firmware	VMBRs	Cloaker
Technique	Persistent	Non-Persistent	Persistent	Non-Persistent			
File/Disk Integrity Checks	✓	×	✓	×	×	×	×
Platform Integrity Checks (TPM)	×	×	✓	×	✓	×	×
Runtime Kernel Integrity Checks	-	-	✓	✓	✓	×	×
File/Disk Signature Checks	✓	×	✓	×	×	×	×
Physical Memory Signature Checks	✓	✓	✓	✓	✓	✓	✓
VMM Detectors	-	-	-	-	-	✓	×

8.2. Effectiveness

Table 6 lists existing approaches used for rootkit detection and compares their detection abilities. We classify user and kernel mode rootkits into persistent and non-persistent versions. Persistent rootkits modify files or sectors on disk and can therefore be detected by integrity checks or signature checks of files or disk sectors. An alternative means to achieve persistence is to modify firmware. Existing firmware rootkits eventually modify kernel structures and can therefore be detected by kernel integrity and signature checks. It may be possible to operate such rootkits in a self-contained manner similar to Cloaker to avoid detection, but this is not used in existing designs. In any case, all firmware rootkits are detectable using TPM technology. Non-persistent rootkits can only be detected by scanning memory at runtime or by watching for suspicious behavior. Cloaker adopts non-persistence for this reason.

Since Cloaker deliberately avoids modifications to the running kernel, both checksum-based and behavior-based kernel integrity checks are rendered ineffective. The version of Cloaker presented in this paper is vulnerable to signature scans of physical memory. Polymorphic techniques could be adopted to weaken this defense against Cloaker, but we have not yet explored this direction.

In addition to the support for locking down TLB entries, many ARM processors include support for locking down lines in the processor cache. This feature can also be used to hide malware from hardware-based signature scanners on peripheral buses. While the entire rootkit may not fit inside the cache, arbitrary portions of the rootkit can be stored in the cache and the corresponding locations in physical memory can be overwritten with random data to thwart scanners that only scan physical memory. Cloaker currently does not support this feature.

VMM detectors address the specific problem of checking for unexpected virtual machine monitors and are therefore not applicable to any of the other types of rootkits.

The analysis in this Section shows that Cloaker is extremely effective at thwarting most existing rootkit detection approaches. We would also like to note that the payload examples that we implement and discuss in Section 5, follow the same hiding principles as Cloaker and are equally stealthy.

9. Checking Integrity of Hardware State

In order to check for the presence of rootkits like Cloaker that modify hardware state, this state can be compared with host kernel expectations. Device drivers generally contain information about the expected state of the hardware that they control. Thus, we believe that an effective countermeasure against hardware supported rootkits is to allow each individual device driver to check the integrity of the state in its corresponding hardware device. We assume that devices drivers are non-malicious and signed by a trusted party.

We design and implement a framework for the Linux kernel that allows device drivers to register integrity check routines which are used at run-time to check for malware. This framework is not architecture specific and can be used on any platform supported by Linux. The design for the framework is illustrated in figure 4. The registered integrity check functions are grouped together as a table of function pointers in a separate section of the kernel at compile time. The kernel has an integrity check management component which can invoke these functions at run-time to examine hardware state integrity. All detected inconsistencies are reported. The presence of even a single inconsistency may indicate the presence of Cloaker-like malware.

The framework itself needs to be protected from malicious code. This is possible using existing approaches to ensure code integrity. For example, hashes can be used to detect unexpected changes to code and static data. This can be augmented with hardware-supported code protection technologies such as AEGIS [4], ARM's TrustZone [2] or DRTM [28].

We used the framework to add integrity check routines for several devices on the OMAP1610 H2 platform. An in-

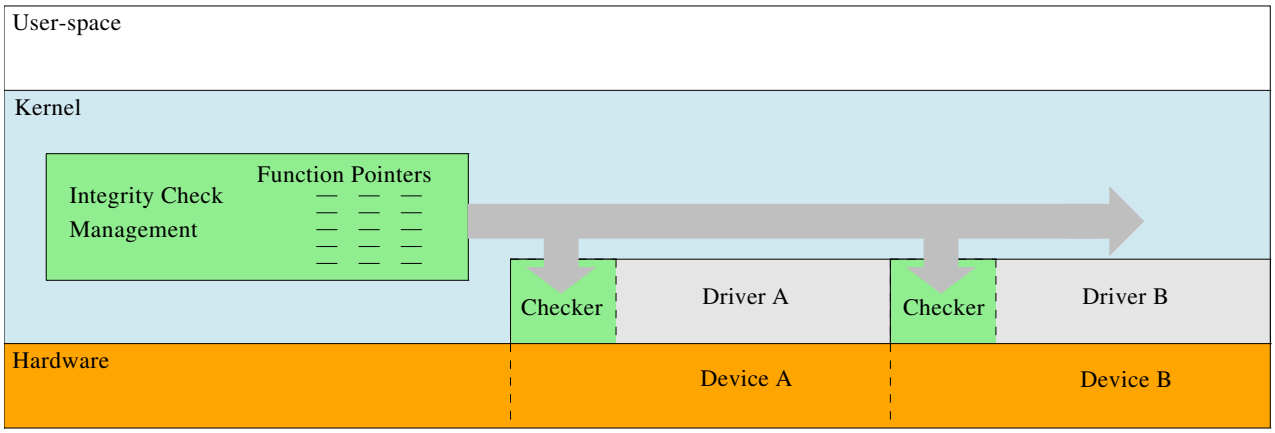

Figure 4. Hardware State Integrity Check Framework

tegrity check routine implemented using our framework examines the validity of the contents in the coprocessor CP15 register C1. This would report an inconsistency if Cloaker were installed because the bit governing the location of the interrupt vector would be flipped. In addition to this check, integrity checks were also written for the serial port driver and the OMAP1610 interrupt controller driver. The serial port driver controls multiple physical hardware devices and the associated integrity check routine checks for valid state in all of them. More specifically, the function checks that the serial port control registers are consistent with the values in the kernel. Unfortunately, since our keylogger payload controls serial ports in a way that always restores the control register state, this integrity checker is ineffective against it. For the interrupt controller, the integrity checker detects if any unexpected interrupt is enabled. This detects the DDosIP payload because it relies on an unused timer interrupt to execute. These and other similar checks make it difficult for rootkits like Cloaker to remain undetected in our framework-enhanced version of Linux.

This detection framework may also be extended and used to disable the functional or stealth aspects of rootkits. This is possible if, in addition to raising an alert, the integrity check routines also attempt to fix the problem by correcting unexpected state alterations. Care would have to be taken when disabling rootkits in this manner to prevent incorrect fixes from crashing the system. We are in the process of further exploring this direction.

Requiring device drivers to provide check routines seems like a difficult task to complete for existing operating systems. It is possible that once such a framework and API is standardized and made available, manufacturers of devices can ship drivers with such integrity check routines for their devices. This should improve security in the long term. Integrity check routines can also be used as a tool for improving reliability. By running periodic integrity checks, it may be possible to detect faults sooner and apply fixes, avoiding system crashes. These reliability and security benefits may motivate adoption of such frameworks.

Other solutions to detecting Cloaker-like rootkits may be possible. While hardware memory scanners that sit on peripheral buses may be tricked, it may be possible to construct better malware detectors by designing hardware that is located close to the processor and monitors its operations. This could be achieved by extending existing designs for processor monitoring hardware such as the Reliability and Security Engine (RSE) [39] to check for unexpected control flow changes in the processor. We leave this as an open research problem.

10. Concluding Remarks

In this paper, we have illustrated the threat posed by changes to hardware settings made by malicious code. We have demonstrated that it is possible to construct a usable rootkit that makes no changes to the host OS code or data and thus, evades detection by existing approaches.

Cloaker is not without limitations though. Unlike VMBRs, which provide a rich and easy-to-use environment for rootkit writers, Cloaker provides a more constrained environment. Rootkits like Cloaker are also closely tied to the hardware and host OS. Whilst these factors may dissuade widespread adoption, a determined attacker may still construct and deploy such a rootkit.

We have shown and analyzed in detail one possible hardware configuration based exploit that malware writers can use for rootkit concealment. Other hardware features can also be used to hide malware. As described elsewhere in this paper, cache line lockdowns and the ability to execute code out of NOR flash can be used to build stronger variants

of Cloaker. Mobile phones have some unique characteristics that differentiate them from desktop or server systems. Because of the need to conserve battery life, they are usually designed to switch to a low power mode whenever possible. On the ARM processor, an instruction can be executed that places it in a state called "deep sleep mode", from which it can only be awakened by an interrupt. This feature may be exploited to run malicious code when the host OS expects the processor to be sleeping. By configuring unused timers or some other device to wake up the processor earlier than expected, malware can execute on the processor without being detected by timing based countermeasures in the OS. While this may result in decreased battery life, we believe that it is difficult to correlate small variations in battery life to the presence of malware.

It is important to realize that more sophisticated rootkits may be constructed by incorporating advanced knowledge about the host OS. The ability to interpret host kernel structures may be used to defeat detection mechanisms or support additional malware. For example, Cloaker only supports Ethernet-like networks with no encryption. If Cloaker were augmented with information about host kernel data structures, it would be possible to intercept encrypted network protocols by stealing keys from the host kernel.

We hope that our work motivates future computer system designers to carefully evaluate security gaps at the boundary between computer architecture and system software and consider deploying a defensive framework similar to the one presented in this paper, or other appropriate countermeasures.

Cloaker is not a completely weaponized rootkit and does not pose an immediate security threat. Nevertheless, we do not intend to make the source code for Cloaker available to the general public. Additional details regarding this project and the source code for the detection framework can be obtained from our website at `http://srgsec.cs.uiuc.edu/`.

Acknowledgments

We would like to thank the anonymous reviewers for their invaluable constructive feedback. Hao Zhu and Raheem Syed helped implement the code for Cloaker's memory management component. This research was supported by grants from DoCoMo Labs USA and Motorola. Our experiments were performed on hardware generously donated by Texas Instruments.

References

[1] A. Acharya and M. Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In *Proceedings of the 9th USENIX Security Symposium*, pages 1–17, Berkeley, CA, USA, 2000. USENIX Association.

[2] T. Alves and D. Felton. TrustZone: Integrated Hardware and Software Security. ARM White Paper, Jul 2004.

[3] AppSnapp iPhone/iPod Touch Unlocker. `http://www.jailbreakme.com/`.

[4] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 65–71, Washington, DC, USA, 1997. IEEE Computer Society.

[5] ARM. ARM926EJ-S Technical Reference Manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0198d/DDI0198_926_TRM.pdf`, Jan 2004.

[6] ARM. Five Billionth ARM Processor for Mobile Devices. `http://www.arm.com/news/16535.html`, Feb 2007.

[7] A. Ashkenazi. Security Features in the i.MX31 and i.MX31L Multimedia Applications Processors. Freescale Semiconductor White Paper, Document Number: IMX31SECURITYWP, 2005.

[8] A. Baliga, X. Chen, and L. Iftode. Paladin: Automated Detection and Containment of Rootkit Attacks. Technical Report DCS-TR-593, Rutgers, Jan 2006.

[9] M. Becher, M. Dornseif, and C. N. Klein. FireWire: all your memory are belong to us. In *CanSecWest*, Vancouver, Canada, May 2005.

[10] D. Beck, B. Vo, and C. Verbowski. Detecting Stealth Software with Strider GhostBuster. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 368–377, Washington, DC, USA, 2005. IEEE Computer Society.

[11] A. Browne. Multiple SIMs: Quantifying the phenomenon taking mobile penetration beyond 100%. Informa Telecoms and Media Report, May 2007.

[12] Bryce Cogswell and Mark Russinovich. RootkitRevealer. `http://www.microsoft.com/technet/sysinternals/Utilities/RootkitRevealer.mspx`, 2006.

[13] J. Butler and G. Hoglund. VICE - Catch the hookers! In *Black Hat USA*, Las Vegas, July 2004.

[14] B. D. Carrier and J. Grand. A Hardware-Based Memory Acquisition Procedure for Digital Investigations. *Digital Investigation*, 1(1):50–60, 2004.

[15] J. Cheng, S. H. Wong, H. Yang, and S. Lu. SmartSiren: Virus Detection and Alert for Smartphones. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, pages 258–271, New York, NY, USA, 2007. ACM.

[16] K. Chiang and L. Lloyd. A Case Study of the Rustock Rootkit and Spam Bot. In *Proceedings of the First Workshop on Hot Topics in Understanding Botnets*, April 2007.

[17] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. Porting Choices to ARM Architecture Based Platforms. Technical Report UIUCDCS-R-2007-2830, University of Illinois at Urbana-Champaign, March 2007.

[18] P. Ferrie. Attacks on Virtual Machine Emulators. Symantec Security Response White Paper, 2006.

[19] Fuzen Op. The FU rootkit. `http://www.rootkit.com/project.php?id=12`.

[20] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is Not Transparency: VMM Detection Myths and

Realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, Berkeley, CA, USA, May 2007. USENIX Association.

[21] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2003.

[22] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The Taser Intrusion Recovery System. In *Proceedings of the twentieth ACM Symposium on Operating Systems Principles*, pages 163–176, New York, NY, USA, 2005. ACM.

[23] J. Grand and B. Carrier. United States Patent 7181560: Method and apparatus for preserving computer memory using expansion card. Feb 2007.

[24] J. Grizzard. *Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems*. PhD thesis, Georgia Institute of Technology, 2006.

[25] J. Heasman. Implementing and Detecting a PCI Rootkit. Technical report, Next Generation Security Software Ltd, November 2006.

[26] J. Heasman. Implementing and Detecting an ACPI BIOS Rootkit. In *Black Hat Europe*, Amsterdam, March 2006.

[27] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Nov 2000.

[28] B. Kauer. OSLO: Improving the Security of Trusted Computing. In *Proceedings of the 16th USENIX Security Symposium*, pages 229–237, August 2007.

[29] G. H. Kim and E. H. Spafford. Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. Technical Report CSD-TR-93-071, Purdue, 1993.

[30] S. T. King and P. M. Chen. Backtracking Intrusions. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 223–236, New York, NY, USA, 2003. ACM.

[31] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society.

[32] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, August 2002.

[33] N. Kumar and V. Kumar. Vboot Kit: Compromising Windows Vista Security. In *Black Hat Europe*, Amsterdam, March 2007.

[34] J. Liedtke. On μ-Kernel Construction. In *Proceedings of the fifteenth Symposium on Operating Systems Principles*, pages 237–250, New York, NY, USA, 1995. ACM.

[35] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 29–42, 2001.

[36] L. W. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 279–294, 1996.

[37] MotorolaFans Forum. `http://www.motorolafans.com/`.

[38] N. Murilo and K. Steding-Jessen. chkrootkit. `http://www.chkrootkit.org`.

[39] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu. An Architectural Framework for Providing Reliability and Security Support. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 585–594, 2004.

[40] J. Nick L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, Berkeley, CA, USA, 2004. USENIX Association.

[41] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. A Semantics-Based Approach to Malware Detection. In *Proceedings of the 34th Symposium on Principles of Programming Languages*, pages 377–388, New York, NY, USA, 2007. ACM.

[42] J. Rutkowska. Patchfinder2. `http://www.rootkit.com/project.php?id=15`.

[43] J. Rutkowska. Detecting Windows Server Compromises. In *HiverCon Security Conference*, Dublin, Ireland, November 2003.

[44] J. Rutkowska. Subverting Vista Kernel For Fun And Profit. In *SyScan 2006*, Singapore, July 2006.

[45] J. Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition. In *Black Hat DC*, Arlington, USA, Feb 2007.

[46] SAMHAIN file integrity / intrusion detection system. `http://la-samhna.de/samhain/index.html`.

[47] Sd and Devik. Linux On-the-fly Kernel Patching without LKM. *Phrack*, 11(58), 2002.

[48] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the twenty-first Symposium on Operating Systems Principles*, pages 335–350, New York, NY, USA, 2007. ACM.

[49] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *Proceedings of the twentieth ACM Symposium on Operating Systems Principles*, pages 1–16, New York, NY, USA, 2005. ACM.

[50] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 272–282, Washington, DC, USA, 2004. IEEE Computer Society.

[51] D. Soeder and R. Permeh. eEye BootRoot. In *Black Hat USA*, Las Vegas, July 2005.

[52] S. Sparks and J. Butler. Raising The Bar for Windows Rootkit Detection. *Phrack*, 11(63), August 2005.

[53] TCG. Trusted Platform Module version 1.2. `http://www.trustedcomputinggroup.org/specs/TPM/`.

[54] Texas Instruments OMAP Platform. `http://focus.ti.com/omap/docs/omaphomepage.tsp`.

[55] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *Computer*, 38(5):48–56, 2005.

[56] S. Yates. Worldwide PC Adoption Forecast, 2007 To 2015. Forrester Research Report, June 2007.

[57] N. Zhu and T. Chiueh. Design, Implementation, and Evaluation of Repairable File Service. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 217–226, 2003.