BootJacker: Compromising Computers using Forced Restarts

Ellick M. Chan, Jeffrey C. Carlyle, Francis M. David, Reza Farivar, Roy H. Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N Goodwin Ave
Urbana, IL 61801-2302
{emchan,jcarlyle,fdavid,farivar2,rhc}@illinois.edu

ABSTRACT

BootJacker is a proof-of-concept attack tool which demonstrates that authentication mechanisms employed by an operating system can be bypassed by obtaining physical access and simply forcing a restart. The key insight that enables this attack is that the contents of memory on some machines are fully preserved across a warm boot. Upon a reboot, BootJacker uses this residual memory state to revive the original host operating system environment and run malicious payloads. Using BootJacker, an attacker can break into a locked user session and gain access to open encrypted disks, web browser sessions or other secure network connections. BootJacker's non-persistent design makes it possible for an attacker to leave no traces on the victim machine.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security

General Terms

Security

Keywords

Security, attacks, memory remanence

1. INTRODUCTION

A plethora of security schemes have been deployed to protect information on computer systems that are vulnerable to physical theft or unauthorized access. Most systems minimally employ an authentication system that requires the user to enter a password before granting access to the system. Many systems also employ console or screen saver locks that require re-authentication if the user session has been idle for some period of time. Modern systems are capable of encrypting network connections and the contents of secondary storage for additional protection. To ensure secrecy, encryption keys used in such systems are typically not generated until after the user has successfully logged in. Once created, these keys are stored in volatile memory as part of the user's session state until the user logs out. It is commonly believed that if a computer is physically stolen, these encryption and authentication mechanisms will significantly hinder attackers from readily accessing stored secrets. In this paper, we demonstrate that this assumption is flawed and present a tool that allows attackers to bypass the system's authentication defenses and gain instant access to user sessions on a live system.

BootJacker is a proof-of-concept attack tool that utilizes an unconventional attack vector to break into the system: a forced restart. This attack exploits the observation that, on many computers, the contents of memory are preserved even after a restart. In fact, researchers have shown that the contents of memory are preserved to a great extent for several minutes after machines physically lose power [18, 19].

An attacker using BootJacker forces an immediate restart on the victim computer and then boots from an alternate device such as a CD or USB drive containing the malware. The sudden restart ensures that the normal shutdown procedure of the victim machine is circumvented, thus preventing security applications from clearing vital keys, and preserving the system's operational state in memory. BootJacker then patches the residual contents of memory with malware payloads and restores the state of the original system.

Although gaining superuser privileges on a victim computer by simply booting an alternate operating system from a peripheral device is trivial, BootJacker is more insidious because it allows an attacker to break into live user sessions. BootJacker thus provides access to open encrypted disks, VPN sessions, secure web browser sessions and other active applications.

In addition to providing unauthorized access to the victim computer, BootJacker's design also allows it to operate in a covert non-persistent mode, which protects the attack tool from discovery by host based intrusion detection systems. This is important if the intent of the attacker is to compromise a machine in place without raising suspicions. It is possible to use BootJacker in such a way that no changes are made to any non-volatile storage in the system. This ensures that minimal evidence of an intrusion remains after an attack.

BootJacker supports the execution of arbitrary malicious software payloads. The core of BootJacker operates like a small bootstrap environment that resuscitates the state of the core system hardware and software environment, while an extensibility framework allows the creation of custom malware payloads and device drivers. We highlight the threat posed by BootJacker by discussing the implications of two specific malicious payloads: one grants the attacker a command shell with superuser privileges and the other terminates security programs such as event logging services or intrusion detection systems. Similar to BootJacker, the payloads we

CCS'08, October 27–31, 2008, Alexandria, Virginia, USA.

have developed are designed to be stealthy, and are therefore also non-persistent.

Researchers have shown that encryption keys can be recovered from memory chips several minutes after power is disconnected [19]. Firewire ports on computers can also be used to directly access memory and obtain sensitive information [4, 29]. Such attacks are also possible even in the presence of technologies such as TPM [38] that provide secure key storage when the machine is powered down. These applications trust the operating system's ability to protect the in-memory plain-text keys when the system is active. While key retrieval attacks can locate secrets in memory, using the keys to gain access to secured information requires further work and may involve substantial effort in cases such as hijacking secure network connections. Unlike key retrieval attacks, BootJacker is able to provide the attacker with full access to the live victim operating environment within a matter of seconds. Additionally, BootJacker does not have to address the problem of locating keys in memory. This is especially helpful when dealing with arbitrary applications and security mechanisms whose key storage locations and formats are unknown.

The characteristics of many architectures and operating systems make them susceptible to the attack illustrated by BootJacker; however, it should be noted that the attack is very machine and operating system specific. This is primarily due to the assembly language and low level operating system resuscitation code required. In this paper, we will describe a version of BootJacker designed to target a uniprocessor Linux 2.6 kernel running on the popular x86 architecture.

BootJacker represents a new class of ephemeral malware that targets machines which leave residual memory state when restarted. Our motivation for creating BootJacker is to study vulnerabilities in systems by analyzing them from an attacker's perspective. BootJacker is similar in spirit to work by other security researchers [24, 10, 21].

The Computer Security Institute reported average losses of hundreds of thousands of dollars for cases of data and computer theft in a recent survey [16]. Currently, the losses from such incidents are limited by the use of technologies such as encrypted disks and secure network connections to private organizational networks. The monetary impact and frequency of these incidents may increase significantly if the vulnerabilities exposed by BootJacker are not sufficiently addressed.

Our contributions in this work show that:

1. Resuscitation of a complete system from preserved memory is possible after a forced restart.
2. Security programs continue to run correctly after resuscitation.
3. Mitigation of such attacks requires destruction of secrets upon reboot or the use of secure boot paths.

The rest of this paper is organized as follows. First, we present a detailed analysis of post-reboot memory properties in Section 2 and follow with a description of the attack process and vulnerabilities in Section 3. We present the technical details of recovering the host hardware and software environment in Section 4, and discuss the accompanying payloads in Section 5. Section 6 evaluates our design followed by a discussion of mitigation in Section 7. Finally, we explore related work and similar attacks in Section 8 before concluding in Section 9.

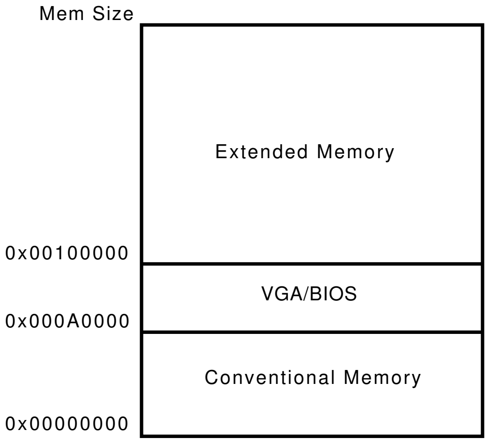

Figure 1: Physical memory on a PC

2. MEMORY REMANENCE ON RESTART

It is a well known fact that, on many machines, data in memory is preserved across a reboot or a brief power outage [18, 19, 9, 33, 1, 17, 27]. As far back as in 1978, researchers have shown that if DRAM is cooled with liquid nitrogen, its data retention period could last up to a week [27]! This property is central to the operation of BootJacker. In this section, we explore the memory remanence properties of PCs.

Our experiments indicate that machines which use ECC memory do not retain software-accessible memory contents after a restart. All reads from ECC memory after a reset always return 0 because ECC memory has parity bits that must be initialized by the BIOS at boot time [19]. Thus, computers that use ECC memory are not vulnerable to reboot-based attacks. We observed that disabling ECC functionality on ECC memory modules using BIOS settings makes these modules behave like non-ECC memory modules. The reader is referred to [19] for a more detailed analysis of memory remanence effects in DRAM chips. In this section, we will describe our study of post-bootup memory remanence characteristics on several x86-based systems.

Even if a system's complete memory image is preserved at the instance that a computer is restarted, portions of the image are clobbered when the BIOS executes. When a PC is booted, the processor begins the initialization sequence by executing startup code from the BIOS. The BIOS is responsible for initializing the core hardware, performing any required selection of boot devices, and loading the boot loader or operating system from the selected boot device.

On a PC, there are two boot modes: cold and warm. A cold boot is performed when the machine is initially powered on. Cold boots perform significant hardware initialization and memory testing, which usually overwrites most of physical memory. Warm boots can be triggered by the reset button (if available) or by a reboot instruction from the running operating system. On a warm boot, there is no interruption in power and memory is usually left intact. In particular, a reboot initiated by the Linux kernel sets a CMOS flag that instructs the BIOS to bypass memory tests.

When booting, the processor starts running in x86 real mode and can initially address up to one megabyte of memory. Only the first 640 kilobytes of addressable memory is RAM. The remainder of the one megabyte addressable region is reserved for memory mapped I/O devices such as the display and other peripherals. This

Table 1: Regions of extended memory overwritten or not recoverable after BIOS execution

Computer	Type	Memory size	Overwritten or unrecoverable extended memory
HP/Compaq 8510w	Laptop	2 GB	1 MB @ 0x200000, 128 bytes @ 0x300000
HP/Compaq Presario 2800T	Laptop	392 MB	170 KB @ 0x100000
Lenovo Thinkpad T61p	Laptop	2 GB	3 MB @ 0x100000, 84 bytes @ various locations
IBM Thinkpad T41p	Laptop	2 GB	3 MB @ 0x100000, 84 bytes @ various locations
Dell Inspiron 600M	Laptop	512 MB	512 MB - completely reset
Dell Inspiron 8600	Laptop	1 GB	1 GB - completely reset
IBM IntelliStation M Pro	Desktop	512 MB	3 MB @ 0x100000, 856 bytes @ various locations (ECC off)
Bochs [7]	Emulator	128 MB	none
QEMU [3]	Emulator	128 MB	none

limitation is a consequence of the history and evolution of the PC architecture. Figure 1 shows the physical memory layout of a typical PC. The first 640 kilobytes of memory (up to 0x000A0000) is generally referred to as "Conventional Memory", while memory above the 1 MB mark is known as "Extended Memory". When the BIOS executes, it runs in conventional memory while it performs initialization and loads the boot sector from a disk.

On some machines, the system and peripheral BIOSes also access and overwrite select portions of extended memory. In order to perform a more detailed analysis of this issue, we have developed a RAM tester tool that analyzes the memory preservation characteristics of a computer. The tester is a stand alone program that is loaded off a CD and uses only conventional memory to operate. The tester fills extended memory with a bit pattern, reboots the computer and checks for regions that have been overwritten or are unrecoverable. Table 1 presents the results of running our tool on several machines without ECC. These results show that various machines which differ in age and origin only overwrite a few megabytes of memory and are thus vulnerable to memory preservation attacks. The characteristic clobbering behavior of various machines is possibly a consequence of different BIOS implementations or hardware configuration discrepancies. We have not attempted to reverse engineer the BIOSes on these machines to determine the exact reasons for these differences; however, we have noticed that on test machines, the absence or presence of various PCI expansion cards helps to determine the clobber signature of memory. Therefore, we believe that part of this effect can be attributed to the actions of expansion card firmware.

Another important characteristic to consider is the behavior of the processor cache during a soft restart. This concern is significant in the context of a write-back cache, since uncommitted data may be lost across a reboot. Our experiments on hardware indicate that the cache is disabled by the processor and transparently recovered after a reset. We believe that when caching is re-enabled, the contents of the cache are preserved and remain intact for the final restoration to the original executing environment. When BootJacker re-enables virtual memory and caching, special care is taken to avoid clearing or invalidating cache lines.

3. ATTACK AND VULNERABILITIES

To illustrate how BootJacker can be used, we describe an attack involving a perpetrator, a victim machine, and a bootable storage device such as a CD or a USB drive with BootJacker installed.

The attack progresses as follows:

1. Physical access to the victim machine is obtained.
2. The victim is forced to immediately restart.
3. The bootable device is connected to the victim.
4. BootJacker is booted instead of the host operating system.
5. BootJacker revives the host software environment and allows the attacker to break into the system and run arbitrary payloads.
6. Optionally, the machine can be returned to the unsuspecting owner to avoid raising theft alarms.

To avoid detection, the attacker must execute these actions in a specific order. If the attacker inserts the device with BootJacker before forcing a reboot, the victim's OS or intrusion detection software may log this suspicious event. Figure 2 shows how an attacker using BootJacker diverts the normal bootup control flow.

The specific vulnerabilities exploited by this attack are:

Physical Access: An attacker requires physical access to the machine in order to carry out this attack. This may be accomplished by either theft or by temporarily accessing an unattended victim computer.

Secrets in Memory: BootJacker is designed to quickly bypass authentication mechanisms and gain access to secrets in volatile memory. This requires the victim machine to be active at the time of the attack. If the attacker needs to physically relocate a desktop computer, techniques such as HotPlug [39] can be used to keep a running system live while it is in transit.

Forced Immediate Restart: The attacker should be able to force the victim to immediately restart in order to ensure that the OS state is preserved when starting BootJacker. This may be accomplished via activation of the reset button or a momentary loss of power. Alternatively, operating system level emergency restart key sequences may be used. The Linux kernel, for example, supports a set of debug hotkeys called the Magic SysRq (System Request) keys, one of which (Alt-SysRq-B) forces the computer to reboot immediately. This immediate reboot does not invoke any code that is normally executed during a clean shutdown of the machine. Thus, the pristine software environment is preserved in memory. The SysRq debug keys are enabled by default in many Linux distributions, and can be triggered with ease by an attacker with physical access.

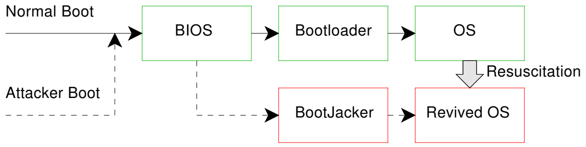

Figure 2: Boot control flow for normal and altered boot paths

Alternate Boot Path: A successful attack depends on the ability of the victim computer to start from an external boot device. While this functionality can be disabled or password protected in the BIOS, many machines still have this ability enabled by default to allow the use of recovery CDs by technical support personnel. Even if the BIOS policy restricts booting from an alternate device, it is possible that a boot loader installed on the host system can be used to circumvent the system policy to load BootJacker from an external device if the loader isn't properly configured for security. All Linux distributions use a boot loader and we have verified that one of the most popular boot loaders, GRUB can be instructed (at runtime) to boot from a different device irrespective of the BIOS boot policy.

Memory Preservation: Ideally, all of physical memory would be preserved after a reboot, but in reality, as described in Section 2, some portions of memory are overwritten by the victim's BIOS and this limitation needs to be addressed by BootJacker. Unrecoverable memory can make it difficult for BootJacker to revive the host software environment, especially if any critical kernel regions are corrupted. By running our RAM tester and analyzing the source code of Linux, we have determined that the contents of conventional memory have no operational effect on the system, even if the values are completely clobbered. This is because memory is not contiguous, and contains several holes around the 1 MB and 16 MB regions that are reserved for various devices such as expansion ROMs, legacy video cards, DMA regions and BIOSes. Since Linux primarily uses extended memory, any corrupted kernel regions in this memory presents a significant problem for BootJacker. In some cases where portions of kernel code are overwritten, BootJacker can reload the kernel code section from the disk; however, when kernel data is corrupted, complete system recovery is nontrivial and we do not attempt to address this problem. Thus, our experiments assume that vital kernel data remains intact.

4. SYSTEM RESUSCITATION

BootJacker is designed to revive a system's software environment and hardware devices through resuscitation. After a machine is forcefully rebooted, the attacker reconfigures the victim's BIOS, if necessary, to load BootJacker. BootJacker is loaded into conventional memory and only uses memory below the 640 kilobyte limit in order to avoid inflicting further damage to the preserved memory above 1 megabyte. BootJacker currently uses a two stage loading process. The first stage uses a slightly modified version of the GRUB boot loader [15] to load any required support files off the disk and the rest of BootJacker into memory. This modified boot loader is part of BootJacker and resides on the attacker's boot media; it should not be confused with the boot loader on the victim machine. The second stage boot process executes the core system resuscitation code and attack payloads.

4.1 Software Resuscitation

In this section, we describe the process of resuscitating a Linux environment on a victim machine. BootJacker requires in-depth knowledge of kernel data structures and their layout (symbol table) in the target Linux kernel in order to successfully revive a system. For example, in order to restore the correct virtual memory mappings for Linux, BootJacker needs to discover the address of the page tables of the interrupted task. Such information is incorporated into BootJacker by building it with Linux kernel header files and using symbol information from the linked kernel. A compiled version of BootJacker is therefore only effective against a specific kernel version and configuration. This is not a significant limitation because most major Linux distributions do not update kernels frequently and few people build and run custom kernels. BootJacker can generally accommodate changes in the kernel when it is recompiled, however, major Linux structural changes may require some additional engineering effort to support.

The objective of software resuscitation is to fabricate a processor context (register state) that can be used to resume execution of the host system environment. We present a scheme that fabricates a valid resume context for the Alt-SysRq-B reboot method (as discussed in Section 3). The key idea is to load the processor registers with values that cause the SysRq handler in Linux to mimic a return from the reboot function, as if it had no operational effect. This allows execution to continue in spite of an actual reboot. The advantage of using this approach to revive the kernel, as opposed to reviving the kernel from some other entry point, is that the logical control flow in the host system is preserved and the semantics of constructs such as locks and semaphores are respected.

In order to construct a valid resume context, it is necessary to understand the operation of the Linux Magic SysRq debug system. The SysRq system in Linux is hooked into the keyboard driver and called when the driver detects the Alt-SysRq hotkey sequence. When the SysRq handler is executed, it acquires a spin lock and disables interrupts. In this context, the system is guaranteed to have a very predictable register state and call chain. We rely on these salient characteristics to construct a resume context safely.

Table 2 lists the various x86 registers and describes the data sources used to reconstruct the values in each register. BootJacker first locates the stack that was in use at the time of restart. Depending on the kernel configuration, this can be one of several known stacks. Figure 3 shows a glimpse of the stack contents just prior to restart. A number of known frames exist on this stack prior to executing the reboot code. These stack frames are scanned by BootJacker to recover register values to build a plausible resume context. Restore values for ESP and EBP are obtained by locating the frame for the *sysrq_handle_reboot()* function on the stack. Then, EIP is set to the return address of the *emergency_restart()* frame. Now, the execution context is prepared to synthesize a return from the reboot function as if the restart had been magically rolled back.

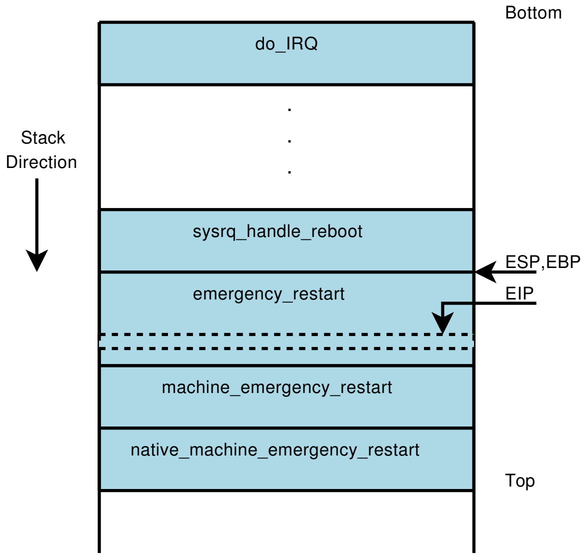

Figure 3: Stack layout at the time of reboot

Table 2: Recovering register state

Register	Description	Source
EAX,EBX,ECX,EDX	General Purpose	Restored from stack
ESI,EDI	General Purpose	Restored from stack
ESP,EBP	Stack, Frame Pointer	SysRq handler stack
EIP	Instruction Pointer	SysRq handler stack
EFL	Flags	Restored from stack
CPL	Privilege Level	Kernel privilege level
CS,DS,SS	Code, Data, Stack Segments	Static value in kernel
ES,FS,GS	Segment Registers	Current task data structure
LDT,GDT	Descriptor Tables	Static value in kernel
TR	Task Pointer	Static value in kernel
CR0	Processor Settings	Static known value
CR2	Page Fault Address	Recovery not required
CR3	Page Table Pointer	Current task data structure
CR4	Advanced Control Flags	Static known value

The *emergency_restart()* function does not have a return value and BootJacker does not have to worry about restoring any of the general purpose registers or processor flags. Once the resume context is restored onto the processor, the return code in the function *sysrq_handle_reboot()* restores the values of these registers from the stack where they were saved in the function call preamble.

The values of the CS, DS and SS segment registers are constant within the kernel and these constants are statically compiled into BootJacker, while the values of the ES, FS, and GS registers are derived from the currently running task. Linux maintains a pointer to the data structure representing the currently running task in a global variable. BootJacker examines this variable and restores the ES, FS and GS registers with the values found in this data structure.

LDT and GDT always maintain a constant value independent of the executing thread. The task register, TR, holds relevant information about the current task. Linux uses software context switching, so there is only one TR per CPU, and its constant value is loaded by the kernel. All of these constant values are determined at build time and statically compiled into BootJacker. CR2 holds the fault address, which does not need to be recovered, and CR3 is the page table pointer, which can be inferred from the current task's virtual memory data structures. This information is required to activate the task's virtual memory mappings in the MMU. The virtual address space is constructed by consulting the page tables of CR3 and the values of the segment registers selected in the GDT and LDT tables.

Once BootJacker has constructed a valid resume context, it must load this context into the processor and enable the new page tables. This poses an interesting problem because the code that reprograms the MMU to use the new set of page tables needs ensure that the next prefetched instruction is available in the old and new address spaces, otherwise the processor would generate an instruction fetch fault upon access. Unfortunately, BootJacker is loaded into conventional memory, which is not available in the standard kernel virtual memory map. BootJacker circumvents this problem by creating a common trampoline memory area that is available in both address spaces. It copies the code that completes the context restore process into an unused portion of the Linux kernel's log message buffer. When BootJacker returns to the host environment, it executes this code to atomically switch the virtual address space and return to the original host environment.

There are some limitations on the types of functions that can be called by BootJacker before it returns control to the kernel. Interrupts are disabled while BootJacker is running and it is essentially running in an atomic interrupt context; therefore, as with a real kernel interrupt handler, no functions that might sleep or otherwise invoke *schedule()* can be called. To work around this limitation, a standard Linux kernel API may be used to schedule an asynchronous callback to be run once interrupts are re-enabled; however, since BootJacker will no longer be mapped into memory, the actual machine code for the callback must be copied into an area accessible from the kernel address space in the same way that the trampoline code does. This technique is used by the RootShell payload discussed in Section 5.

4.2 Hardware Resuscitation

Upon reboot, system hardware may be reinitialized to a state that is desynchronized with the expectations of the host Linux kernel. BootJacker needs to ensure that hardware devices such as keyboards, disk controllers and Ethernet devices continue to work after the software environment is revived. Surprisingly, BootJacker can rely on Linux to revive certain devices automatically. In order to handle some classes of buggy or unresponsive hardware, Linux includes code to retry operations or reinitialize devices upon an error. BootJacker takes advantage of such support if it is available. For cases where Linux is unable to revive hardware on its own, BootJacker incorporates custom code to assist resuscitation.

BootJacker's hardware resuscitation process is similar to the actions performed by the system when resuming from sleep or suspend mode. Normally, the operating system saves resume structure state to a disk or memory area upon a suspend. These resume structures describe the exact state of the machine at the time of suspend. Hardware devices and the CPU are then shut down and their transient state is subsequently lost. The process of rebooting incurs a similar loss of state. When the machine is resumed, the hardware and processor state are restored from a resume image. BootJacker does not have the luxury to save any state before the reboot and instead must fabricate an analogous restore state by scanning for clues in the residual memory image.

In this section, we describe the process of resuscitating the core hardware on a uniprocessor PC such as the keyboard, mouse, VGA graphics chip, interrupt controller, system timer and Ethernet chip. Most modern PCs have core hardware that is similar and backwards compatible with older revisions [30]. We describe restoration support for basic peripherals; this configuration is generally sufficient for system recovery. Each hardware device has its own peculiarities, and the recovery process is tailored to device semantics. We detail specific restoration challenges on a case by case basis.

Interrupt Controller: x86 PCs use an i8259-compatible interrupt controller. This controller is responsible for arbitrating interrupts from various peripherals such as the keyboard, disk and network interface to the CPU. The i8259 controller supports interrupt masking and prioritization. On a PC, the highest priority interrupt is the system timer, followed by the keyboard. Other peripherals are usually assigned a dynamic interrupt number and priority by the PCI or plug and play (PNP) controller.

Upon reboot, the system BIOS completely resets the i8259 controller. All interrupts are masked and disabled. During recovery, we re-enable interrupts of active devices upon resuscitation. The i8259 also supports interrupt renumbering, which allows the controller to shift interrupt numbers by a base offset. Linux uses this facility to remap i8259 interrupts number 0-7 to Linux interrupts 32-39. Linux combines interrupts and processor exceptions (such as page faults) into a common linear exception table for fast lookup. BootJacker must restore this offset, otherwise Linux misinterprets the interrupt number read from the controller as a different processor interrupt, and causes a kernel panic by running the incorrect handler.

Some modern computers, especially those with multiple processors, use a newer interrupt controller called the Advanced Programmable Interrupt Controller (APIC). This controller is backward compatible with the i8259, and its semantics are similar. The code for resuscitating an APIC on a uniprocessor machine is similar to that of an i8259. At this time, we do not support multiprocessor machines.

System timer: PCs have three i8253 programmable interrupt timers (PIT). Each timer can be configured for periodic or one-shot interrupts in accordance to a countdown value. The Linux kernel only uses the first timer as the system timer. The default setting for Linux is to configure a 100 Hz timer to periodically activate the scheduler. The other two timers are used by various drivers and applications.

Upon reboot, these timers are left intact. We do not need to reinitialize any of them. Although these timers do not need to be restored, their semantics are inconsistent when the system is resuscitated. For example, the system timer interrupt is not received by the kernel for the duration of the reboot, and hence the Linux *jiffies* software timer is suspended. Therefore, system time lags behind wall clock time. In order to preserve the semantics of software that relies on the correct time, we restore the system's time by reading the wall clock time from the hardware clock, which is always running (powered by a battery) even when the computer is turned off.

Starting with the Pentium, newer x86 processors include a monotonically increasing 64-bit CPU cycle time-stamp counter which is reset to zero whenever the processor is reset. The value of this counter can be read using the *rdtsc* instruction. Since this counter will be reset when the system is hijacked, code watching this counter for unexpected changes may be able to detect a discrepancy. In order to thwart detection, we estimate the value of the counter based on the system uptime and write this calculated value back to the counter using the *wrmsr* instruction.

PS/2 Keyboard/Mouse: PS/2 keyboards and mice are driven by an i8042 controller. This controller is responsible for receiving keystrokes, mouse movements, and mouse clicks as well as setting keyboard LEDs. The controller itself is reinitialized by the BIOS at boot time. When GRUB runs, it initializes the controller and since Linux uses the same keyboard settings, no special re-initialization is required.

By default, the mouse is disabled by the BIOS at boot time. BootJacker sends the i8042 controller a command to re-enable the mouse port. Some PS/2 mice have settings for the scroll wheel and extra buttons. These settings are reinitialized by BootJacker to conform to the protocol of a standard scroll wheel mouse. Custom mouse protocols and advanced settings are vendor-dependent, and are thus not restored.

Display: Although there are a myriad of video chips and drivers in the market, most chipsets support the standard VGA and VESA video modes. In Linux, the X server provides the graphics console and directly controls the hardware. When BootJacker reinitializes the device, it sets the video chip back to a safe state in text mode. Once the system is revived, the attacker can manually switch back to the X graphics console. The X server will perform the necessary re-initialization of the video device when switching back to the graphics mode.

Disk: The IDE controller connects peripherals such as hard drives and optical disks to the system. Upon reboot, the BIOS reinitializes the disks and disables their corresponding interrupts on the i8259.

BootJacker eschews re-initialization of the disk controller, and instead relies on Linux's error recovery routines. IDE drives are accessed using a request/response command interface. If any command fails to respond due to disk or bus communication errors, Linux will retry the command or reinitialize the disk controller. This facility is used to deal with buggy IDE controllers and disks, which may hang under certain errant conditions.

When a disk is first accessed after BootJacker is run, there is an initial three second lag while the initial IDE command times out and Linux resets the controller. Linux reports this disk error as a lost IDE interrupt, and transparently retries the command. Since the access is successful upon retry, no data read errors are reported, and the disk continues to operate smoothly without error.

Coprocessor units: The Intel architecture has support for various coprocessors that are directly attached to the main CPU. The most popular unit is the i387-compatible math coprocessor or floating point unit (FPU). Since this coprocessor has become a fundamental part of the architecture, the kernel and user space C libraries expect it to be present and functional. Whenever the kernel performs a context switch, it saves the current process's FPU state and restores the FPU state of the process to be switched in. The user space GNU C library also initializes the floating point unit during C program startup to ensure a predictable FPU state for C programs.

Upon reboot, the FPU is disabled by the system firmware. BootJacker must reset the FPU and re-enable it in order to clear any pending exceptions. Otherwise, the kernel's context switch code will fail when it suddenly discovers that the floating point unit is off, and the FPU state cannot be saved or restored. The user space FPU settings are restored by the kernel's context switch code, so that applications do not mysteriously fail.

Network: Upon reboot, the BIOS probes network devices and runs each device's respective firmware. This supports functionality such as network booting. We have found that many network chips are non-functional after resuscitation, so we must perform some re-initialization. Luckily, the Linux network driver model allows ethernet devices to specify a transmit timeout function. This functionality allows a network interface to run a recovery routine if a transmission is held up in the send buffer, and does not get sent within a requisite timeout period. As part of the resuscitation process, BootJacker uses the Linux network device API to iterate through all network interfaces and calls each transmit timeout function. In many cases, this primes the network chip, and restores functionality. Since this API is part of the kernel's generic network device interface API, it does not depend on the card vendor, so many chipsets are supported without additional effort. We have confirmed that this approach works with the Intel PRO/100 and Intel PRO/1000 Ethernet adapters.

Typically, an Ethernet chip contains send and receive buffers. These buffers act as holding areas to queue outgoing packets when the Ethernet line is busy, and receive buffers to temporarily store unprocessed packets. Although BootJacker is able to recover the functionality of the Ethernet interface, there is the issue of packet loss while the computer is rebooting. These transient faults are always handled at higher network layers such as TCP and network connections do not usually experience any adverse affects. Since the reboot process requires less than half a minute, applications do not time out and subsequently close their connections.

5. PAYLOADS

BootJacker is designed to host a whole new class of malware which relies on patching residual memory images. As such, BootJacker provides a mini-environment for malware writers to customize payload code. BootJacker provides a customizable hook point right before the system resumes. At this point, kernel data structures may be accessed, and many kernel functions can be called directly. Malware payloads using the BootJacker framework can look up data structures and function pointers exported by the special *ksymtab* symbol table in the kernel. They can search for running processes, files or socket resources and modify them, if required.

In order to illustrate the range of attacks possible with BootJacker, we have developed several malicious payload applications. In this section, we will delve into the specifics of two of these payloads: RootShell, which provides the attacker with a superuser command shell and Terminator, which kills screen savers, system loggers, and other security utilities. Both payloads were relatively straightforward to write and they employ standard kernel features such as spawning processes and sending signals. Though we do not elaborate in this paper, BootJacker can also be used to install other generic malware such as rootkits and spyware.

5.1 RootShell

RootShell is a superuser shell spawned by BootJacker. Attackers can use this shell to interactively explore the system or run customized payloads, including loading other kernel modules or rootkits. At the point that RootShell runs, Terminator has already cleared the system of security software.

When the RootShell payload is invoked by BootJacker, it starts a new thread in the host kernel which invokes the *kernel_execve()* function to load and run the shell program on the host computer with superuser privileges.

The shell is spawned on one of the unused text mode virtual consoles (accessed by special keyboard sequences) of the Linux system. Like BootJacker, this payload is non-persistent and does not leave any trace in the system; however, the effective stealthiness of this payload also depends on the operations that the attacker performs using the shell. This is because RootShell does not provide any support for roll-backs of persistent operations performed by the attacker. It is up to the attacker to ensure that malicious actions do not leave any persistent traces such as shell command histories on the victim machine.

5.2 Terminator

The Terminator payload assists a wide variety of attacks by defeating security and logging software on the system. It kills the system logger daemon, antivirus software, intrusion detection tools and other security software. Terminator accomplishes this task by scanning the process list in the kernel and sending appropriate termination signals to the victim processes.

Once Terminator completes, the attacker can run other attack tools unencumbered by security tools and without worry that actions may be logged. Since the screen saver and other authentication tools are killed, locked desktop sessions become fully accessible to the attacker. Session-based applications such as web browser SSL connections can then be accessed by the attacker directly from the user interface.

6. EVALUATION

To show that BootJacker is able to successfully hijack computers, we evaluate its correctness and ability to circumvent various security measures. Correctness is assessed by comparing the output of several applications in a non-hijacked environment against corresponding output in a hijacked environment. Effectiveness against security programs is demonstrated by circumventing several encryption utilities. In each case, BootJacker is able to compromise the system without breaking the semantics of running applications.

Please note that the actual encryption algorithms are not understood by BootJacker, and that the security mechanism defeated here is the screensaver or console lock. These tests merely demonstrate that the security applications are not sensitive to a mediated reboot.

6.1 Effectiveness

BootJacker has been tested on all of the machines in Table 1 that do not reset memory completely; however, we use an IBM IntelliStation M Pro computer with a 2 GHz Intel Pentium 4 processor, 512 MB RAM, IDE disk, and an Intel PRO/100 network card for all of the experiments in this section. The generic hardware resuscitation support described in Section 4 is able to completely revive the software environment in this computer after a restart.

To confirm that BootJacker is able to correctly resuscitate the system without corrupting any memory or the processor cache, we run a standard memory tester called *memtester* [6], and reboot the machine during several test runs. The tests we selected create patterns in two separate memory areas by either writing random values or performing some simple computations such as multiplication or division. It then compares the two areas to ensure that no memory errors have occurred. We ran over 10 trials of these tests on our evaluation system while restarting the system in the middle of the test. In all of these cases, memtester does not report any errors.

In order to verify the correctness of the system after resuscitation, we force a restart in the middle of several different tasks and examine if the tasks complete correctly after the resuscitation.

The tasks performed are:

1. *gcc*: Compilation of the C source file containing the H.264/MPEG-4 AVC video compression codec in the MPlayer [37] media program.
2. *gzip*: File compression using the deflate compression algorithm.
3. *wget*: File download.
4. *convert*: JPEG image encoding.
5. *aespipe*: AES file encryption.

For the last four tasks, we use a 100 MB TIFF image file as the input. For each task, we compare MD5 hashes of the output file generated without a BootJacker interruption against results obtained from a test with a restart in the middle of task execution. We have run each experiment at least 10 times and have not encountered any errors.

The primary goal of BootJacker is to provide the attacker with access to protected data on a live system transparently. In order to demonstrate that this goal has been attained, we selectively test several applications and kernel-level subsystems that provide secure access to data and network resources. All of these programs store temporary keys in volatile memory.

The applications we test are:

1. *SSH* secure shell connection between the victim and another host.
2. *SSL* web browser session to a secure web server.
3. *PPTP* [20] VPN connection to a secure university network.
4. *dm-crypt* [12] encrypted file system.
5. *Loop-AES* [5] encrypted file system.

In the first two cases, the secret keys are located in user space. In the remaining cases, the keys are stored in kernel memory. We perform each attack on a machine that is locked by a user (using a console locking program) with these programs running in the background. In each case, the Terminator payload was used to terminate the lock program after resuscitation.

Our SSH tests use OpenSSH version 4.6 as the client and server. OpenSSH supports various modes of the AES, DES, Blowfish and ARC4 ciphers. In each case, BootJacker was able to allow an attacker to access the secure shell irrespective of the underlying encryption scheme or key size that was used. The SSL test consists of a web browser session connected to a popular web e-mail service using TLS 1.0 and AES encryption. After restarting, we were still able to browse mail folders, read messages, and send e-mail using the hijacked session. The VPN client we used was based on the Point-to-Point Tunneling Protocol (PPTP) [20] with 128 bit keys. VPN connections and application sessions remained operable after a successful attack, and new connections could be established over the open tunnel. This implies that enterprise level perimeter defenses can be successfully penetrated using BootJacker. We also tested BootJacker against two popular Linux file system encryption systems: dm-crypt and Loop-AES. Both systems were configured to use 256 AES CBC encryption. We were able to access the contents of the file system successfully after an attack.

These experiments demonstrate the significance of our attack. Other published key retrieval attacks [19, 29, 36, 23, 28, 22] must be tailored to specific key types, sizes, applications or kernel components. Such attacks require extra support to use the retrieved keys to gain access to secure sessions. BootJacker allows the attacker to access the live system and immediately explore or manipulate protected data.

6.2 Size

BootJacker is written in a mixture of C and x86 assembly code. Table 3 shows that BootJacker is a very small program. The lines of code reported in the table are from the output of the *sloccount* program. We break down the size of each component into core C and assembly code, hardware-specific resuscitation code and payload code. To minimize its size, BootJacker reuses existing kernel code to reinitialize the disk and network by calling into function pointers; the size of this kernel code is device-specific and therefore excluded from the table, since these components are not part of BootJacker. The small compiled size of BootJacker and its payloads implies that a minimal amount of host memory is overwritten and lost when BootJacker is loaded onto the system. The current version of BootJacker depends on the GRUB boot loader for disk support. Because this increases the memory footprint, we are working on eliminating the dependency on GRUB by directly including disk support code within BootJacker. Also, we have not attempted to aggressively optimize the code in BootJacker for size.

Table 3: Sizes of BootJacker and payloads

Component	Lines of Code	Compiled Size (bytes)
BootJacker (C)	998	11,633
BootJacker (Assembly)	135	431
BootJacker (Hardware)	305	1,502
Terminator	34	285
RootShell	52	332
Grub	14 (modified)	101,266

6.3 Time

On all evaluation machines, we were able to execute this attack in less than one minute, and on many of them were able to open a superuser shell in less than 30 seconds. Most of this time is consumed by the BIOS boot sequence. Once loaded, BootJacker takes a negligible amount of time to revive the software environment. Although most of the hardware in the system is revived quickly, it takes a few seconds for the Linux recovery routines to reset the hard disk. Ethernet connections resume nearly instantaneously, as BootJacker proactively calls the transmit timeout routine for the ethernet driver to initiate chip recovery. Since many Internet protocols gracefully handle dropped or lost packets, sessions tend to survive the short reboot process.

7. MITIGATION

The threats posed by BootJacker can be countered by reconfiguring systems so that they do not satisfy the required preconditions of the attack. The most effective defense against such an attack is to avoid any memory remanence issues by ensuring that secrets are not present in volatile memory whenever a computer is vulnerable to unauthorized access. This approach would require extensive operating system and application support. For example, if a computer is locked using a screen saver, all secure connections should be dropped and secrets should be erased from memory. The connections should be reestablished only after the user has re-authenticated. All applications that depend on secure devices or connections would have to be suspended so that they don't fail when trying to access the corresponding device (such as an encrypted disk). An OS-level solution would be to stop all computation and encrypt memory until the user has authenticated. Unfortunately, these approaches may not always be desirable or convenient.

There are several other mitigation techniques that can significantly impede and deter an attacker. One option is to require password authentication in the BIOS before booting the system. The pre-boot authentication supported by some machines with a TPM chip also provides similar protection. An alternative to requiring boot time authentication is to ensure that the boot path is completely protected from possible redirection by requiring authentication at the BIOS and boot loader level for any changes to the boot configuration. This approach is only effective if the configured boot order first attempts to boot the operating system from a trusted disk. The Aegis [2] system ensures the security of the boot path and therefore prevents unauthenticated booting. An alternate option is to zero out memory at boot time before loading the OS. This ensures that all secrets in volatile memory are erased and consequently prevents BootJacker from reviving the system. To impede Terminator, the operating system can also attempt to respawn the screensaver or security applications if they are improperly terminated. Finally, the use of ECC memory starves BootJacker of the requisite remenance property.

These solutions for protecting secrets in memory are not completely effective because many memory imaging techniques can be used to save and restore memory. This includes attacks via FireWire [11] as well as physical removal of the memory modules [19]. If the memory can be dumped, BIOS and boot loader authentication can be bypassed by simply reviving the system on a different computer without BIOS protection.

BootJacker does not faithfully restore all hardware state; therefore it is possible that a task running on the resuscitated system could detect BootJacker by watching for unexpected changes in the hardware state. If it is known that such a process is running, detection can be avoided by using the Terminator payload to stop the detection process before resuscitating Linux.

8. RELATED WORK

Until recently, only a few simple software-based attacks were feasible with physical access. An attacker could directly access data on storage devices or install malicious software such as keyloggers or trojans. These classic attacks have spurred the adoption of storage encryption technologies and malware detection tools. Unfortunately, such defenses do not provide complete protection in many cases. Several researchers have explored directly retrieving secret keys from memory. One such attack exploits memory remanence [19] and the other attack exploits the FireWire protocol to access arbitrary physical memory [29, 11]. Unlike these simplistic memory and key acquisition tools, BootJacker is more sophisticated and allows attackers to bypass authentication in the victim machine's OS to access the full software environment directly. A similar attack employs a FireWire port on the victim to bypass password authentication on Windows [4]; however, unlike this attack, BootJacker does not require the presence of a FireWire port.

Recovering information from a system after a restart has been investigated by researchers working on system reliability techniques. A locked up operating system can be been recovered after an ARM processor reset is triggered by a watchdog timer [9]. BootJacker can be enhanced with similar recovery techniques in order to recover the host system from arbitrary restart locations. This approach would be required if the Alt-SysRq-B reboot method is unavailable and a restart can only be forced through a power interruption. BootJacker builds upon this early work on ARM-based mobile phones, but expands on the technique in much greater detail due to the higher complexity of the x86 architecture.

Vbootkit [26] and eEye BootRoot [34] also use an alternate boot path to install malware in the system; however, unlike BootJacker, these systems only install code that is executed upon the next boot cycle and do not attempt to recover information from memory and revive the live system.

BootJacker is similar in spirit to other non-persistent malware [31, 13], and it is also designed to avoid leaving any evidence of a break-in. The payloads we discussed are capable of ensuring that security programs such as intrusion detection and logging systems are stopped or terminated when the attack is in progress.

There are many published techniques that address operating system-level attacks [32, 14, 8, 25]. These projects primarily address code-injection and data-injection type attacks and they can be extremely effective for thwarting various attack classes. The attack illustrated by BootJacker is not prevented by such defensive techniques because it uses an unconventional entry point into the system.

9. CONCLUDING REMARKS

BootJacker demonstrates that memory remanence attacks can be used to bypass most modern security mechanisms and that such attacks represent a significant threat which needs to be addressed by the security community. Although we have discussed some possible mitigation techniques in this paper, we believe that this issue requires further investigation and study.

Is this attack possible with other operating systems? While access to the Linux kernel source code catalyzed the development of BootJacker, we believe that similar attack tools could also be developed for various closed source operating systems, albeit with significantly more effort.

BootJacker is not a generic attack tool that can be directly applied to arbitrary Linux kernel versions and hardware; however, since most corporations and organizations tend to roll out a standardized software configuration for many of their machines, diversity is less of an issue. Nevertheless, in most cases, a simple recompilation of BootJacker is sufficient to retarget a different kernel version or configuration. Resuscitating arbitrary hardware devices is a more significant problem. Our design does ameliorate this problem by taking advantage of the host Linux kernel's support to recover some classes of devices. However, this is still a challenging problem and may limit adoption of this attack approach by all but the most determined attackers.

Attack tools such as BootJacker exploit one of many security gaps at the boundary between systems software and architecture. BootJacker and other such attacks [31, 35, 19] are possible because the assumptions made by security services in system software do not necessarily match the functionality or behavior of the underlying architecture. We hope that our work encourages researchers and system developers to study this aspect in greater detail.

Although BootJacker has been presented as an attack tool, the techniques developed have tremendous potential for legitimate forensic analysts who need to swiftly recover information. Coupled with memory dump and virtual machine technologies, this tool can be greatly enhanced to increase recovery rates even with diverse hardware and software configurations. Cooperation from proprietary operating system vendors can help foster the development of BootJacker into an indispensible forensics tool.

To help users determine if their machines are vulnerable to the flaws exposed by BootJacker, we plan to release a limited testing and diagnostics tool on our website at http://srgsec.cs.uiuc.edu.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. The following people (in alphabetical order) contributed to beneficial discussions during the development and writing process: Rob Adams, Ray Essick, Michael LeMay, Jason Lowe and Dale Rahn. This research was supported by grants from DoCoMo Labs USA, Motorola and a Siebel Fellowship. The views expressed are those of the authors only.

10. REFERENCES

[1] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, First edition, January 2001.

[2] 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.

[3] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[4] A. Boileau. Hit By A Bus: Physical Access Attacks with Firewire. In *RUXCON*, Sydney, Australia, Sep 2006.

[5] D. Bryson. The Linux CryptoAPI A User's Perspective, May 2001.

[6] C. Cazabon. MemTester. `http://pyropus.ca/software/memtester/`.

[7] O. S. Community. Bochs IA-32 Emulator Project. `http://www.gnu.org/software/grub/`.

[8] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *SOSP '07: Proceedings of the Twenty First ACM Symposium on Operating Systems Principles*, October 2007.

[9] F. M. David, J. C. Carlyle, and R. H. Campbell. Exploring Recovery from Operating System Lockups. In *USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[10] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. Cloaker: Hardware Supported Rootkit Concealment. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[11] M. Dornseif. 0wned by an iPod. In *PacSec*, 2004.

[12] C. Fruhwirth. New Methods in Hard Disk Encryption. Technical report, Vienna University of Technology, June 2005.

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

[14] 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.

[15] GNU. GRand Unified Bootloader. `http://www.gnu.org/software/grub/`.

[16] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson. CSI/FBI Computer Crime and Security Survey. Computer Security Institute, 2005.

[17] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Proceedings of the 6th USENIX Security Symposium*, pages 77–90, July 1996.

[18] P. Gutmann. Data Remanence in Semiconductor Devices. In *Proceedings of the 10th USENIX Security Symposium*, pages 39–54, Berkeley, CA, USA, 2001. USENIX Association.

[19] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, and J. A. Calandrino. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[20] K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, and G. Zorn. Point-to-Point Tunneling Protocol (PPTP). RFC 2637 (Informational), July 1999.

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

[22] S. Jarecki, N. Saxena, and J. H. Yi. An attack on the proactive RSA signature scheme in the URSA ad hoc network access control protocol. In *SASN '04: Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*, pages 1–9, New York, NY, USA, 2004. ACM.

[23] J. Kelsey, B. Schneier, and D. Wagner. Related-key cryptanalysis of 3-way, biham-des, cast, des-x, newdes, rc2, and tea. In *ICICS '97: Proceedings of the First International Conference on Information and Communication Security*, pages 233–246, London, UK, 1997. Springer-Verlag.

[24] 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.

[25] 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.

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

[27] W. Link and H. May. Eigenshaften von MOS-Ein-Transistorspeicherzellen bei tieften Temperaturen. In *Archiv fur Elektronik und Ubertragungstechnik*, pages 33–229–235, June 1979.

[28] J. Loughran and T. Dowling. A java implemented key collision attack on the data encryption standard (des). In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 155–157, New York, NY, USA, 2003. Computer Science Press, Inc.

[29] J. Mäkinen. Automated OS X Macintosh password retrieval via firewire. `http://blog.juhonkoti.net/2008/02/29/automated-os-x-macintosh-password-retrieval-via-firewire`, 2008.

[30] M. A. Mazidi and J. G. Mazidi. *80x86 IBM PC and Compatible Computers: Assembly Language, Design, and Interfacing; Volume I and II*. Prentice Hall PTR, Upper Saddle River, NJ, USA, fourth edition, 2002.

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

[32] 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.

[33] S. Skorobogatov. Low-Temperature Data Remanence in Static RAM. Technical report, University of Cambridge Computer Laboratory technical report No. 536, June 2002.

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

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

[36] A. Stubblefield, J. Ioannidis, and A. D. Rubin. A key recovery attack on the 802.11b wired equivalent privacy protocol (wep). *ACM Transactions on Information Systems Security*, 7(2):319–332, 2004.

[37] M. team. MPlayer. `http://www.mplayerhq.hu/`.

[38] Trusted Computing Group. Trusted Platform Module version 1.2. `http://www.trustedcomputinggroup.org/specs/TPM/`.

[39] WiebeTech. HotPlug: Transport a live computer without shutting it down. `http://www.wiebetech.com/products/HotPlug.php`, 2008.