Kernel Data Attack is a Realistic Security Threat

Jidong Xiao[1], Hai Huang[2], and Haining Wang[3]

[1] College of William and Mary, Williamsburg VA 23185, USA,
[2] IBM T.J. Watson Research Center, Hawthorne NY 10532, USA,
[3] University of Delaware, Newark DE 19716, USA,

Abstract. Altering in-memory kernel data, attackers are able to manipulate the running behaviors of operating systems without injecting any malicious code. This type of attack is called kernel data attack. Intuitively, the security impact of such an attack seems minor, and thus, it has not yet drawn much attention from the security community. In this paper, we thoroughly investigate kernel data attack, showing that its damage could be as serious as kernel rootkits, and then propose countermeasures. More specifically, by tampering with kernel data, we first demonstrate that attackers can stealthily subvert various kernel security mechanisms. Then, we further develop a new keylogger called DLOGGER, which is more stealthy than existing keyloggers. Instead of injecting any malicious code, it only alters kernel data and leverages existing benign kernel code to build a covert channel, through which attackers can steal sensitive information. Therefore, existing defense mechanisms including those deployed at hypervisor level that search for hidden processes/hidden modules, or monitor kernel code integrity, will not be able to detect DLOGGER. To counter against kernel data attack, by classifying kernel data into different categories and handling them separately, we propose a defense mechanism and evaluate its efficacy with real experiments. Our experimental results show that our defense is effective in detecting kernel data attack with negligible performance overhead.

1 Introduction

When a system is compromised, attackers commonly leave malicious programs behind so as to allow the attackers to: (1) regain the privileged access to the compromised system without re-exploiting a vulnerability, and (2) collect additional sensitive information such as user credentials and financial records. To achieve these two goals, attackers have developed various kernel rootkits. Over the past years, kernel rootkits have posed serious security threats to computing systems. To defend against kernel level malware, a vast variety of approaches have been proposed. These approaches, either rely on additional hardware [27, 23], or leverage the virtualization technology [13, 32]for countering kernel level attacks. With these defense mechanisms, we can ensure the integrity of kernel code and read-only data, protect kernel hooks from being subverted to compromise kernel control flow, and prevent malicious code from running at the kernel level. Thus, most existing kernel level attacks can be effectively thwarted.

Therefore, attackers are aggressively seeking new vulnerabilities inside the kernel. Ideally, the new attacks should not inject any malicious code running at the kernel level. To this end, kernel data attack has already attracted some attention. By altering

kernel data only, without injecting any malicious code, attackers are able to manipulate kernel behaviors. Compared to existing kernel level malware, kernel data attack is more stealthy. This is because, most kernel code does not change during its whole lifetime, and thus, can be well monitored and protected with existing defenses. In contrast, most kernel data is supposed to be inherently changeable (except for read-only data), making it much harder to detect kernel data attacks.

Kernel data attack is first demonstrated by tampering with kernel data structures and showing four attack cases [4]. However, three of the four attack cases still require attackers run their malicious code at the kernel level, and the remaining case merely shows performance degradation. This raises several questions: what damage can a kernel data attack cause? can this type of attack really affect system security? can this type of attack achieve the same level of threat as existing kernel rootkits do? We attempt to answer these questions in this study.

In this paper, we first assume the role of attackers and explore the attack space of kernel data attack. Through novel kernel data manipulation, we demonstrate that kernel data attacks can introduce security threats as serious as existing kernel rootkits, including disabling various kernel-level security mechanisms and stealing sensitive information. And then we investigate, from the defenders' perspective, how to detect kernel data attack. The major contributions of this work are summarized as follows:

- We first systematically study the attack space of kernel data attack. After analyzing Linux kernel source code, we reveal that the attack space is enormous: in one of the latest Linux Kernel version (3.1.10), there are around 380,000 global function pointers and global variables in the Linux kernel, and the vast majority of these data are subject to change during runtime.
- By examining various Linux kernel internal defense mechanisms, we observe that the runtime behaviors of these mechanisms rely on some global kernel data. Altering these in-memory global kernel data, attackers can subvert these defense mechanisms. More specifically, we demonstrate that attackers can tamper with the Linux auditing framework, subvert the Linux AppArmor security module, and bypass NULL pointer dereference mitigation, on a victim machine. Thus, it is clear that kernel data attacks are realistic threats, even as serious as existing kernel rootkits, yet more stealthy than existing kernel rootkits, as they do not require the injection of any kernel-level malicious code.
- To further demonstrate the severity of kernel data attack, we design and implement a novel keylogger: DLOGGER. DLOGGER exploits an inherent property of the Linux proc file system, which is the bridge between the kernel space and the user space. In particular, by redirecting a proc file system pointer to a tty buffer, attackers can construct a covert channel, and then utilize this covert channel to monitor user input and steal sensitive information, such as passwords. DLOGGER is more stealthy than existing keyloggers, as it neither changes any kernel code nor runs a hidden process, which enables it to evade existing rootkit/keylogger detection tools.
- We propose a defense solution to detect kernel data attack. Our defense is built on the fact that there are different types of kernel data, which demonstrate different running behaviors and characteristics during runtime. By providing a kernel data

classification and treating different types of data separately, we evidence that the proposed defense is effective in detecting kernel data attack with negligible performance overhead.

2 Background

It is commonly known that operating systems have various vulnerabilities, and these vulnerabilities are often exploited by attackers to break into a system and gain root access. The focus of this paper, is not to discuss how to exploit these vulnerabilities; in contrast, we study the problem that, after a system is compromised by attackers, how to mask their presence and enable continued privileged access to the system, as well as collect additional sensitive information. To achieve these goals, attackers usually install rootkits. Modern rootkits usually run at the kernel level, and these rootkits are called kernel rootkits. Most of existing kernel rootkits attempt to modify kernel hooks and redirect these hooks to some malicious functions injected by the attackers. However, recent research work has demonstrated the effectiveness of protecting operating systems from a hypervisor level or using additional hardware. These defense frameworks would prevent attackers from installing any rootkits or running any malicious code at the kernel level.

Therefore, attackers need new attack strategies which do not require injecting any malicious code inside the kernel to compromise a victim system and gain a strong foothold on it. Currently, there are two possible approaches for attackers to reach this objective. First, return oriented programming (ROP) attack. ROP is an exploit technique that directs the program counter to run existing code while achieving malicious goals. Since its birth, it has drawn much attention, and has been extensively studied. On the offense side, a number of approaches have been proposed to make ROP more robust and resilient, such as [8, 35].On the defense side, a variety of defense mechanisms have been proposed, such as [26, 19].As an alternative, kernel data attack has not yet attracted enough attention. Under a kernel data attack, attackers have full access to kernel memory, but will not inject any new code or modify existing code that will be executed at the root privilege level. Since the kernel stores its data in memory, attackers can manipulate these data and then attempt to alter the running behaviors of the victim system. To some extent, kernel data attack is similar to ROP attack, as neither of them requires code injection. Therefore, ROP-based malware is sometimes called data-only malware [39]. However, ROP attack is very different from kernel data attack, and the major differences are two-fold. First, ROP attack generally starts with a buffer overflow vulnerability that enables attackers to overwrite the return address or jump address. In contrast, kernel data attack has nothing to do with buffer overflow vulnerability, but it requires that attackers have control of the kernel memory. Second, to perform ROP attack, attackers must have in-depth knowledge of stack structure and assembly code, and it takes non-trivial engineering efforts to construct the so-called gadgets, which are the foundation of ROP attack. In contrast, kernel data attack typically requires attackers to understand kernel code, which is usually C code, and once attackers know which data should be changed, mounting the attack is trivial.

2.1 Attack Space

Theoretically, attackers can exploit all the kernel data. However, there are different types of kernel data, which should be treated differently.

First of all, the kernel stores both local data and global data in its memory. While both of them may affect the running behaviors of an operating system, exploiting global data is more feasible because the memory locations of global data can be easily identified. Essentially, Linux exports all global symbols (including function names and variable names) to user space via a proc file system file, /proc/kallsyms. This file includes a symbol-to-virtual-memory-address mapping. Meanwhile, the Linux kernel also provides various kernel APIs for kernel modules to search and access these symbols. Therefore, identifying the memory location of every global symbol is a trivial task. Once we are aware of the memory location of our target symbol, which represents a piece of kernel data, we can change its value by writing to that virtual memory address and measure its impact to the system. By contrast, local data is usually stored in the kernel stack or kernel heap, identifying its address is a non-trivial task. In this work, we focus on global data but we plan to explore local data in our future work.

Next, kernel data can also be classified into function pointers and variables. Many existing kernel rootkits achieve their malicious goals by hooking function pointers, including system call handlers and virtual file system interface pointers. However, as we mentioned above, in order to evade the defense mechanisms deployed at the hypervisor level, attackers should not inject any malicious code that requires persistent running in the system. Therefore, we do not consider hooking any function pointers and our focus is on variables only.

Furthermore, kernel data can also be divided into read-only data and read-write data. Read-only data, literally, is the data that is not supposed to be changed during runtime, and one can only read from but not write to the data. A typical example is the system call table, which is a popular target for many kernel rootkits. Existing hypervisor-based defense systems have shown their effectiveness in the protection of kernel read-only data [43, 13]. However, protecting kernel read-write data is more challenging, as the vast majority of these data are subject to change at runtime.

To assess the space of kernel data attack, we perform a systematic study over Linux Kernel source code, and we quantify all the kernel global data, including function pointers and variables. Our finding is that, in the kernel we study (version 3.1.10), there are about 380,000 global variables and function pointers. It is obvious that if all the kernel global data could be potentially exploited, the attack space of kernel data attack is enormous. Even if we only consider the global read write variables, the space is still fairly large.

3 Kernel Data Attacks

In this section, we show various attack scenarios on kernel data. These attacks are by no means a comprehensive list of what is possible. We choose a few interesting examples to demonstrate some common techniques that an attacker can use to remain hidden while subverting various system security measures. These scenarios are illustrated on a Linux Qemu-KVM based virtual machine with configurations shown in Table 1. Although we

Table 1: System Configuration

Components	Specification
Host CPU	Intel Xeon 3.07GHz, Quad-Core
Host Memory	4GB
Host OS	OpenSuSE 12.3
Host Kernel	3.7.10-1.16-desktop x86_64
Qemu	1.3.1-3.8.1.x86_64
Guest Memory	1GB
Guest OS	OpenSuSE 11.3
Guest Kernel	2.6.34-12-desktop i686

perform our attacks on a virtual machine, they can be easily done on a physical machine without any changes. Section 3.1 shows how an attacker can bypass the Linux Auditing and AppArmor frameworks to avoid detection while setting up a backdoor / rootkit to further compromise a system. In Section 3.2, we show an attacker can leverage NULL pointer dereferencing to gain elevated privilege from a normal user account.

3.1 Bypass Linux Auditing and AppArmor

Tampering with Linux Auditing Framework The Linux Auditing framework records security events in a system. It consists of a kernel daemon that writes audit messages to disk, and several user-level utilities that are used to define security policies about what types of events should be recorded. The audit log can be examined to determine if certain security policies are violated, and if so, by whom and also from running what command. Security policies used by the Linux Auditing framework are defined in /etc/audit/audit.rules, and one of the commonly used policies is to define a list of sensitive files that should be monitored, such as follows.

Listing 1.1: Rules Setting in /etc/audit/audit.rules

```
-w /etc/shadow -p rwxa
-w /etc/passwd -p rwxa
```

These two rules instruct the auditing system to keep track of all file accesses to /etc/shadow and /etc/passwd, which contain critical user account information such as user ID, group ID, and encrypted password. To avoid being detected, an attacker with root access often turns off any auditing or monitoring tools before making further changes to such sensitive files. Furthermore, the attacker could also install a modified version of the auditing or monitoring tool to hide any trojan processes or files. However, such changes are still easily detectable by external monitoring tools, e.g., in the hypervisor. An alternative and less conspicuous method to bypass the auditing system's detection is by identifying and modifying kernel data that has an impact to the auditing system's code executing path, and if modified in a certain way, a critical block of the security code could be partially or even completely circumvented. In the case of the Linux Auditing framework, we found that the kernel function audit_filter_syscall (invoked by audit_syscall_entry) is responsible for writing audit records to a log. The following code snippet (from kernel/auditsc.c) shows this procedure.

```
void audit_syscall_entry(int arch, int major,
                unsigned long a1, unsigned long a2,
                unsigned long a3, unsigned long a4)
{

...
      context->dummy = !audit_n_rules;
      if (!context->dummy && state == AUDIT_BUILD_CONTEXT) {
            context->prio = 0;
                state = audit_filter_syscall(tsk,
            context,
            &audit_filter_list[AUDIT_FILTER_ENTRY]);
    }
...

}
```

We noticed that audit_n_rules is a globally defined variable. By identifying its location via a symbol lookup and setting it to zero would prevent the Linux Auditing system from keeping track of file system accesses. To evaluate the effectiveness of this attack, we first enable the Linux Auditing system and let it load the predefined rules. We then open /etc/passwd and /etc/shadow, and as expected, the corresponding auditing messages are written to the system log.These messages include detailed information, such as the name of the file that was accessed, access time, message id, the accessing system call used and its arguments.

Next we change the value of audit_n_rules. By searching in /proc/kallsyms, we found its address is 0xc0a61ee4. After writing a zero to this address, we can easily set the audit_n_rules's value. Consequently, it no longer writes auditing messages to the system log when /etc/passwd and /etc/shadow are accessed.

Subverting the Linux AppArmor Framework AppArmor stands for Application Armor, and it is implemented as a Linux kernel module. Providing mandatory access control, it allows system administrators to associate each program with a security profile that restricts its capabilities, e.g., access to certain resources such as files and sockets.

AppArmor supports three profile modes: enforce, complain, and kill. "Enforce" means the predefined policies will be enforced. "Complain" means AppArmor will only report violations but will not take any actions. And "kill" means a program that violates a predefined policy will be killed. In addition, AppArmor also supports auditing, and it implements five types of auditing services including: normal, quiet_denied, quiet, noquiet, and all.

To bypass Linux AppArmor, we manipulate two variables, which are g_profile_mode and g_apparmor_audit. Both of them are defined in apparmor/lsm.c as enum type variables. The variable of g_profile_mode controls the profile mode and g_apparmor_audit controls the auditing type. The values of g_profile_mode and g_apparmor_audit can be altered so that AppArmor is running at the complain profile mode and the quiet audit type to prevent policy violations from being reported.

To evaluate the effectiveness of this attack, we write a test program called TestApp. This program attempts to read, write, and access certain files. We then define a corresponding AppArmor policy for this program. The policy states that TestApp is not allowed to read or write to File A, and is allowed to read File B but not allowed to write to it. When TestApp runs, AppArmor correctly identifies access violations according to

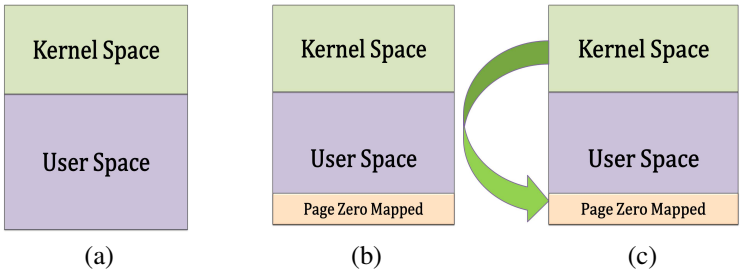

Fig. 1: Exploiting Kernel NULL Pointer Dereference

(a) Initial state; (b) Map page zero and put malicious shellcode in page zero; (c) Trigger kernel NULL pointer dereference

the defined policy. Access violations are prevented and also logged. However, after we set g_profile_mode to 1 and set g_apparmor_audit to 2 (corresponding to the "complain" profile mode and the "quiet" audit type) by directly altering their values in memory, AppAmor can no longer prevent access violation or report anything even if the policy is violated.

3.2 Bypass NULL Pointer Dereference Mitigation

A NULL pointer dereference happens when a program attempts to read from or write to an invalid (and more specifically, NULL) memory location. It is commonly caused by a software bug in the program. In user programs, it causes segmentation faults; and in kernel code, it could cause system crashes. It has been demonstrated in recent years how the behavior of kernel NULL pointer dereferencing can be exploited to facilitate privilege escalation [37, 33, 25]. As a matter of fact, 2009 has been proclaimed by some security researchers as "the year of the kernel NULL pointer dereference flaw" [10].

By default, a NULL pointer does not correspond to any valid memory address. To exploit the NULL pointer dereference vulnerability, an attacker maps the NULL page (i.e., page zero) with a mmap() system call, puts malicious shellcode into it, and then forces a NULL pointer dereference. If done correctly, this allows an attacker to gain root access with full control of the operating system [12, 34]. Figure 1 depicts this procedure.

To mitigate this exploit, Linux introduces a variable called mmap_min_addr, which specifies the minimum virtual address that a process is allowed to map. It is set to be 4096 on x86 machines as default. By setting mmap_min_addr to 0, we can bypass this mitigation mechanism. Linux kernel actually exports mmap_min_addr to user space via the proc file system, so that system administrators can tune this variable. Consequently, any manipulation to this variable is noticeable by system administrators. To address this problem, we observe that for many proc file system entries, Linux kernel associates them with one variable and one pointer. While the variable is used by the core kernel, the pointer is used by the proc file system. In a healthy system, this pointer points to the memory location that stores this variable. Figure 2 shows this relationship. To avoid detection, we can redirect it to another memory location, which we call a *safe memory location*. A safe memory location refers to a memory address that is rarely or never used by the kernel. For example, we observe that there is a 4K gap between the end

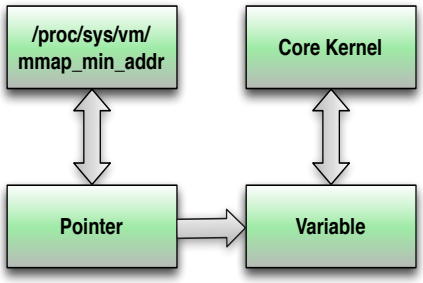

Fig. 2: Original Relationship

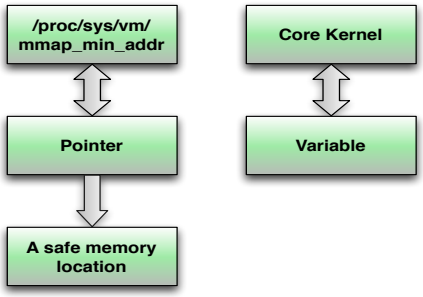

Fig. 3: After Manipulation

```
test@linux:~/test> id
uid=1001(test) gid=100(users) groups=33(video),100(users)
test@linux:~/test> whoami
test
test@linux:~/test> ./myexploit
[*] Resolving kernel addresses...
 [+] Resolved econet_ioctl to 0xf802cda8
 [+] Resolved econet_ops to 0xf802cfe4
 [+] Resolved commit_creds to 0xc104761b
 [+] Resolved prepare_kernel_cred to 0xc1047517
[*] Calculating target...
[*] Triggering payload...
[*] Got root!
linux:~/test> id
uid=0(root) gid=0(root)
linux:~/test> whoami
root
linux:~/test>
```

Fig. 4: Privilege Escalation Attack

of the kernel read-only data section and the start of the kernel read-write data section, which can be used for this purpose as the kernel normally does not access any of these addresses. We also set the safe memory location's value as the default mmap_min_addr value, which is 4096. By doing so, we dissolve the connection between the pointer and the variable. This new relationship is illustrated in Figure 3. Any subsequent read or write access to /proc/sys/vm/mmap_min_addr would only access the safe memory location. This leaves the attacker free from changing the value of the corresponding kernel variable to anything he wishes without being detected by security tools that monitor abnormalities in the proc file system.

To evaluate the effectiveness of this attack, we write a C program that invokes the mmap() system call to map page zero. When mmap_min_addr is by default set to 4096, our program simply fails, and the mmap() system call returns EACCES (indicating permission denied). Next, we set the variable mmap_min_addr to zero, and run the program again, this time the mmap() system call succeeds. Once this has been done, by exploiting the notorious sock_sendpage() NULL pointer dereference vulnerability (available since 2001 and was only discovered in 2009 [20]), we verified that a local unprivileged user can execute arbitrary code in kernel context and gain root privilege. This privilege escalation attack is shown in Figure 4.

Table 2: Summary of Rootkits with/without Keylogger Feature

Rootkit Name	Attack Vector	Keylogger	Code Injection
Complete Rootkits:			
Adore-ng-2.6	proc fs file operations table	No	Yes
SucKIT-2	interrupt descriptor table	No	Yes
DR	debug register	No	Yes
enyelkm v1.1	system call table, interrupt descriptor table	No	Yes
Knark 2.4.3	system call table, proc fs file operation table	No	Yes
KBeast-v1	system call table	Yes	Yes
Sebek 3.1.2b	system call table	Yes	Yes
Mood-nt 2.3	system call table	Yes	Yes
Demonstrates Key Logging Only:			
Linspy v2beta2	system call table	Yes	Yes
kkeylogger	system call table	Yes	Yes
vlogger	tty $\rightarrow ldisc.receive\_buf$	Yes	Yes

4 Keylogger Design and Implementation

While the attacks we presented in Section 3 can passively bypass some of the existing security frameworks, we now demonstrate an active kernel data attack in the form of a keylogger. A keylogger is a type of surveillance software[4] that records the keystrokes typed by a user. Over the years, keyloggers have been demonstrated to be a tremendous threat in the real world. For example, in 2008, a keylogger harvested over 500,000 online banking and other account information [30]. And then in 2013, 2 million Facebook, Gmail, and Twitter passwords were compromised by a keylogger [1].

Keyloggers are commonly implemented as a part of kernel rootkits. Before we present the design of our keylogger, we first studied 10 existing rootkits, as shown in Table 2. Most of these rootkits were also studied by many recent research efforts [29, 15, 14, 22, 3, 27].

From Table 2, we can see that among the 10 rootkits we have studied, six of them have a keylogging feature, including KBeast, Sebek, Mood-nt, Linspy, kkeylogger, and vlogger. Except for vlogger, the other five rootkits use similar techniques to record keyboard inputs, i.e., by intercepting read or write system calls. By contrast, vlogger [31] attempts to hijack the tty buffer processing function, instead of intercepting read/write system calls.

We can also see from Table 2 that, no matter which approach they use, existing keyloggers rely heavily on hooking kernel function pointers to interpose its own functions. As we described before, recent advances on the defense side have already demonstrated their effectiveness in defeating this type of attack, therefore, a new attack method is needed. In this work, we propose a new keylogger, called DLOGGER[5], which only relies on manipulating kernel data. The key idea behind DLOGGER is that, when the keyboard receives any input information, that piece of information must be transferred into the kernel (via the keyboard driver) and stored in a memory buffer. A keylogger should grab that information and pass it to the user space. Since we are not allowed to run our own code, we have to enable the kernel do the information passing, i.e., pass the data from the kernel into the user space. Fortunately, the Linux kernel does provide such an avenue, the proc file system (and also the sysfs file system), which bridges the

[4] To be accurate, keyloggers can be classified into software and hardware types, but in this work, our focus is on software keyloggers, in particular, kernel level keyloggers.

[5] DLOGGER, denotes Data only attack based keyLOGGER.

kernel space and the user space. Thus, if we can direct the kernel to pass the information from its memory buffer into a proc file system buffer, or if we can redirect a proc file system pointer to that memory buffer, then we can expect that, by reading from a file under the /proc directory, an ordinary user can collect that information.

The detailed explanation of our design is as follows. To receive user input, the Linux kernel emulates several terminal devices, called ttys, and the first emulated terminal device is referred to as tty1. For each emulated terminal device, the kernel would generate a file under the /dev directory, as the Linux system views every device as a file. So, /dev/tty1 represents the emulated terminal device tty1. The kernel defines a data structure called *struct tty_struct* (include/linux/tty.h), which refers to one tty terminal device. And *struct tty_struct* has a field called *char * read_buf*, which is exactly the memory buffer to accommodate the user input from that emulated terminal device. By opening the device file /dev/tty1, we can get its file descriptor, which has a pointer pointing to the *struct tty_struct*. Once we access the *struct tty_struct*, we can locate the address of its *read_buf*. Then we need to pick up a proc file system pointer, and let it point to this memory buffer. The selected proc file system pointer should represent a proc file that is rarely accessed by system administrators. Given the fact that there are a large number of files under a proc file system, a vast majority of files under /proc would rarely, if not never, be accessed. In our experiments, we choose /proc/sys/kernel/modprobe. In a healthy system, cat /proc/sys/kernel/modprobe would display the path of the modprobe binary[6], which by default, is /sbin/modprobe. The kernel defines a char pointer called modprobe_path, which just points to the string "/sbin/modprobe". Consequently, if we set this char pointer to the tty_struct's read_buf, we can expect that any read to /proc/sys/kernel/modprobe would display the content of the tty read buffer, which should be the user input from keyboard.

Figures 5, 6, 7 illustrate how DLOGGER differs from existing keyloggers. Figure 5 shows the normal data flow, i.e., when there is no keylogger. Figure 6 shows the data flow of a traditional keylogger, and Figure 7 shows the data flow of DLOGGER. It can be seen from these figures, while existing keyloggers actually change the data flow, DLOGGER does not, instead, it creates a new branch to collect the information.

We then validate the efficacy of DLOGGER. After login as the root user from a tty terminal, we input our password, and type several commands. We then try to login remotely as an ordinary user, by reading the /proc/sys/kernel/modprobe file, we can see the information typed in the tty terminal.

5 Defense

In this section, we first present a defense mechanism to detect kernel data attacks by classifying kernel data into different types. Then, we evaluate the effectiveness of the proposed defense and measure its overhead in terms of CPU and memory usage.

5.1 Defense Mechanism

We observe that kernel data can be classified into the following four types:

- Type 1: Read-only data.

[6] The modprobe binary is a program to add or remove loadable modules to/from the Linux kernel.

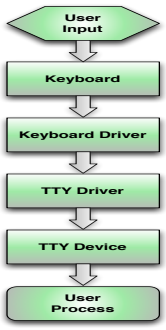

Fig. 5: Normal Data Flow

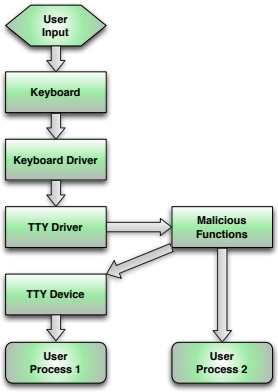

Fig. 6: Data Flow in a Traditional Keylogger

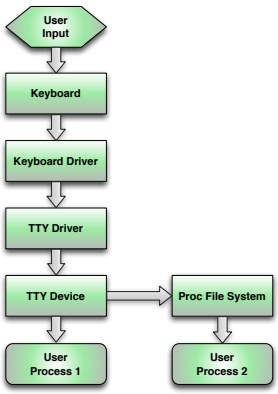

Fig. 7: Data Flow in DLOGGER

- Type 2: Modifiable data that normally remains constant across different systems.
- Type 3: Modifiable data that normally does not change, but can differ from system to system.
- Type 4: Modifiable data that changes frequently.

The different types of data exhibit different runtime behavioral characteristics. An effective defense mechanism, thus, should be tailored to the characteristics of the different types. Our defense mechanism consists of two phases: splitting phase and monitoring phase.

Splitting Phase In this phase, we split different types of data into different lists. To accomplish this, we first set up multiple identical virtual machines. In our experiment we used two VMs, which we observed were as good as if we had more VMs, VM1 and VM2. Algorithm 1 describes the subsequent splitting procedure. Essentially, we get all the variable symbols from /proc/kallsyms and put them into a list called ListAll. We then use List1, List2, List3, and List4 to represent the symbol lists for Type 1, Type 2, Type 3, and Type 4 data, respectively. The algorithm consists of three steps:

– The first step is to get List 1. It is rather straightforward, as these symbols are explicitly marked in the /proc/kallsyms with a "r" or "R". Therefore, we extract all these symbols and put them into List1, and leave the rest into ListRW (denoting a list of read-write symbols).
– The next step is to extract List2 from ListRW. Since Type 2 data remain the same across VM1 and VM2, for the symbols in ListRW, we get their values from VM1 and VM2, identify those equivalent pairs, and put these symbols into List2, and leave the rest into ListDiff.
– The final step is to extract Type 3 and Type 4 data from ListDiff. Since data in Type 3 rarely change, for each symbol in ListDiff, we measure its value from VM1 at multiple points during a span of multiple days. If no changes were identified, we put it into List3; otherwise, we put it into List4. To avoid false positives in the later

Algorithm 1 Get Type 1,2,3,4 symbols

Require: /proc/kallsyms, VM1, VM2
Ensure: VM1 and VM2 run the same OS, have the same configuration
  $ListAll \leftarrow Get$ all variable symbols from /proc/kallsyms
  $List1 \leftarrow Get$ all read only symbols from /proc/kallsyms
  $ListRW \leftarrow ListAll - List1$
  for each symbol in $ListRW$ **do**
    $var1 \leftarrow get$ its value from VM1
    $var2 \leftarrow get$ its value from VM2
    if $var1 = var2$ **then**
      $List2 \leftarrow insert$ symbol
    else
      $ListDiff \leftarrow insert$ symbol
    end if
  end for
  for each symbol in $ListDiff$ **do**
    $var0min \leftarrow get$ its value from VM1
    $ListPair0 \leftarrow insert$ (symbol,var)
  end for
  wait for a predefined time interval $T1$
  for each symbol in $ListDiff$ **do**
    $vartmin \leftarrow get$ its value from VM1
    $var0min \leftarrow get$ its value from $ListPair0$
    if $var0min = vartmin$ **then**
      $List3 \leftarrow insert$ symbol
    else
      $List4 \leftarrow insert$ symbol
    end if
  end for

monitoring phase, this step should be run iteratively, so that we can ensure List 3 only contains data that rarely change or never change.

Monitoring Phase After we have built List1, List2, List3, and List4, we can start the monitoring phase. Typically, an attack target falls into either Type 1, Type 2, or Type 3 categories. As for Type 4 data, they can be divided into six subtypes, which are shown in Table 3.

Type 4 data are very system-specific. An example of such data is jiffies, which is a global variable Linux kernel used to keep track of the number of ticks since the system last booted. It is highly unlikely that the jiffies values would be the same between two systems. We did not discover any Type 4 data that could have been attacked similarly to those we illustrated before, and thus, we focus on the other data types for building our defense mechanism. Type 1 data are read-only, and prior works [13, 43] have already demonstrated thoroughly how to defeat such attacks.

To detect attacks against Type 2 data, one can use an approach similar to PeerPressure [41], where collective information across peer machines dictates what is normal and what is abnormal for data values. For Type 3 data, as they normally do not change once initialized, one can record their initial values and periodically compare their current values against initial ones, similar to the Tripwire [17] approach. Algorithm 2 depicts this monitor procedure.

5.2 Defense Evaluation

To evaluate the effectiveness and performance overhead of our defense mechanism, we conducted several experiments on a hypervisor running two VMs. Based on Algorithm 2, we developed a tool that runs on the hypervisor level and monitors both Type 2 and Type 3 data. Any kernel data attacks (those described in Sections 3 and 4) can

Algorithm 2 Monitor Type 2,3 symbols

Require: List2, List3, VM1, VM2
Ensure: VM1 and VM2 run the same OS, have the same configuration
  for each symbol in List3 **do**
    $varinit \leftarrow get$ its value from VM1
    $ListPair0 \leftarrow insert$ (symbol,$varinit$)
  end for
  loop
    for each symbol in List2 **do**
      $var1 \leftarrow get$ its value from VM1
      $var2 \leftarrow get$ its value from VM2
      if $var1 \neq var2$ **then**
        Raise Alarm: Symbol Value Changed
      end if
    end for
    wait for a predefined time interval $T2$
    for each symbol in List3 **do**
      $varrun \leftarrow get$ its value from VM1
      $varinit \leftarrow get$ its value from $ListPair0$
      if $varrun \neq varinit$ **then**
        Raise Alarm: Symbol Value Changed
      end if
    end for
  end loop

Table 3: Global Variables Belong to Type 4

Category	Example Variable	Meaning
Timing Related	jiffies	The number of clock ticks have occurred since the system booted
Random Numbers	skb_tx_hashrnd	A random hash value for socket buffer
Runtime Workload Related	nr_files	Number of opened files
Index Related	log_start	Index into log_buf
Cookies	fsnotify_sync_cookie	Cookies used by fsnotify to synchronize monitored events
Spinlocks and Semaphores	pidmap_lock	Spinlock for pidhash table

be easily detected. Naturally, the detection response time and performance overhead are mainly dependent on the time interval described in Algorithm 2. A shorter interval leads to a shorter response time but a higher performance overhead. However, since the tool is rather lightweight, we do not expect it to cause any noticeable performance overhead. For example, when we set the time interval to 500 milliseconds, and we mounted the attacks presented in Sections 3 and 4, all the attacks can be detected in about 1 second. We measured the monitoring tool's performance overhead by using the cuadro benchmark [11], which shows the CPU overhead is less than 2% when the time interval is set to 500 milliseconds. We also run a Linux kernel decompression task, in which a standard Linux kernel source package linux-2.6.34.tar.bz2 is decompressed with the *tar* program. The result also shows that the monitor tools incurs less than 2% of runtime overhead when the time interval is set to 500 milliseconds. Table 4 lists these experimental results. We also observed that the memory usage of our defense is no more than 0.3%. Therefore, the performance overhead induced by our defense is negligible. Additionally, since we use an iterative approach in the splitting phase, we ensure only those data that never change are classified as Type 3, and thus, there is no false positive during the monitoring phase.

6 Discussion

In this section, we discuss the limitations and extensions of kernel data attack. While we have mainly demonstrated malicious exploits based on global variables, attackers can

Table 4: CPU Overhead (%)

Benchmark \ Interval	1000ms	500ms	100ms	0ms
Cuadro	0.35	1.69	2.88	3.28
Kernel Decompression	0.70	1.87	3.10	4.06

also potentially misuse local variables. Local variables are stored in kernel stacks or heaps. A sophisticated attacker can explore the kernel memory to identify the locations of any exploitable local variables. In fact, manipulating local variables could make the attack even more undetectable as knowing what is a good value of every local variable is almost impossible.

Linux kernel extensively uses linked list data structures. Many of these linked lists change frequently, e.g., the linked list representing the current running processes. An element is added to the list when a process is created, and is removed when the process exits. In a running system, as there are a lot of process creation and destroy events, this linked list changes almost constantly, which makes anomaly detection on the linked list a daunting task. A common attack many existing rootkits use is to remove certain elements from the process linked list (used by the *ps* command) to hide certain malicious processes. This works well due to the fact that the CPU scheduler uses another process linked list when scheduling processes. As stated earlier, it is becoming increasingly difficult to inject malicious code and launch malicious processes as it is easily detectable by many security tools. However, a similar attack can still be mounted, but in a reverse manner, i.e., by removing an element from the CPU scheduler linked list but keeping it in the *ps* linked list, an attacker can prevent a benign process (e.g., a process launched by a security tool) from being scheduled. Even if system administrators periodically check if this process is still running by using the *ps* command, they will be deceived to believe that the process is running normally.

A limitation of the kernel data attack is that it no longer works when the target system reboots as all the modified data are in memory. One could persist all the kernel data changes by modifying system initialization scripts, but this will render the attack more prone to be detected. However, as non-volatile memory technology is getting cheaper and denser, many researchers [21, 2, 7, 38] believe that it will soon appear on the processor memory bus complementing the traditional memory. As non-volatile memory becomes more prevalent, it will make kernel data attack easier to be mounted and last longer. In addition, some of the global data can be accessed by multiple processes/threads, modifying these data might cause side effects. Therefore, how to ensure the safe execution of the OS kernel is something that attackers have to handle.

7 Related Work

Kernel Data Attack and Defense: Although kernel data consists of both function pointers and variables, most attacks against function pointers still require injecting new code. Therefore, we do not categorize these attacks as kernel data attacks. Also, defeating such an attack is straightforward, either by protecting function pointers [42] or monitoring system control flow integrity [29]. In contrast, kernel data attacks that only manipulate variables or variable pointers, instead of function pointers, are more stealthy and harder to defeat. This type of attack is defined by Chen et al. [9] as non-control-

data attack. But they demonstrated the viability of such an attack at the application level rather than at the kernel level. The possibility of mounting this type of attack at the kernel level is first presented in [4]. However, among the four different attack cases shown in the work, three of them still require attackers run their own code at the kernel level; the remaining one merely degrades system performance by manipulating memory page related data.

To defend against non-control data attacks, Baliga et al. [3] proposed Gibraltar, which infers kernel invariants during the training stage and protects the integrity of these invariants at runtime. Petroni et al. [28] and Hofmann et al. [15] both proposed a specification based solution, which requires users manually specify integrity check policies. Although these solutions are effective in defeating several rootkits that manipulate non-control-data, they rely heavily on a prior knowledge of the attacks, which limits themselves to deal with existing rootkits only. As the space for kernel data attack is enormous, attackers have sufficient target data to exploit and bypass these defense tools. In addition, Bianchi et al. [6] designed Blacksheep, which aims to detect kernel data attack, but it ends up failing to do so. The main reason is that their approach is mainly through analyzing memory dump, which includes all kinds of kernel data. They computed a difference for all the data between multiple dump images, and a significant difference between them would raise an alarm. They finally conceded that their approach is not effective against kernel data attack, and they attributed it to that kernel data continues to change while taking memory dumps.

Keylogger: Keyloggers, including both software and hardware keyloggers, have been studied extensively over the past years. Compared with software keyloggers, hardware based keyloggers [46, 5, 40, 18], either rely on an external device to collect the acoustic or electromagnetic emanations of the keyboard, or utilize a GPU to monitor the keyboard buffer. A common limitation of these hardware based keyloggers is that attackers must have physical access to the victim system, which to some extent, restricts the impact of this attack. In contrast, software based keyloggers do not have this limit. To defend against software based keyloggers, as well as other malware that collects user privacy information, a number of taint analysis based solutions have been proposed [45, 44, 24, 16]. The key observation of these approaches is that keyloggers or malware usually incur suspicious access to sensitive information. By tracking the information flow, these tools are able to accurately detect privacy leakage. However, these approaches usually induce significant performance degradation, and thus might not be suitable for deploying in production systems. For instance, Panorama causes a system slowdown by a factor of 20. Moreover, Slowinska et al. [36] evaluated the practicality of taint analysis, and found that most of the existing solutions have serious drawbacks; finally, they concluded that taint analysis "may have some value in detecting memory corruption attacks, but it is fundamentally not suitable for automated detecting of privacy-breaching malware such as keyloggers".

8 Conclusion

Without injecting any kernel-level malicious code, attackers can launch a kernel data attack in a much more stealthy manner by merely altering kernel data. However, whether kernel data attack could cause serious security damage to a victim system is unanswered question. In this paper, we have demonstrated the severity of kernel data attack. In par-

ticular, we have shown that by altering in-memory global kernel data, attackers can bypass the Linux Auditing framework, the Linux Apparmor framework, and the NULL pointer dereference mitigation, which significantly facilitates malicious privilege escalation. To further demonstrate the security threat posed by kernel data attack, we have designed and implemented a new keylogger. Our keylogger is more stealthy than existing keyloggers and is able to evade the existing rootkit/keylogger detection tools, as it neither changes any kernel code nor requires running a hidden process. Therefore, we conclude that kernel data attacks are indeed realistic security threats. To counter against kernel data attacks, we have proposed a defense mechanism that classifies kernel data into four different types and handles these different types of kernel data separately. Our experimental results show that our proposed defense is very effective against kernel data attacks.

References

1. 2 million facebook, gmail and twitter passwords stolen in massive hack. 2013. http://money.cnn.com/2013/12/04/technology/security/passwords-stolen/.
2. K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems (HotOS)*, pages 2–7. USENIX Association, 2011.
3. A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Annual Computer Security Applications Conference (ACSAC)*, pages 77–86. IEEE, 2008.
4. A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *IEEE Symposium on Security and Privacy (SP)*, pages 246–251. IEEE, 2007.
5. Y. Berger, A. Wool, and A. Yeredor. Dictionary attacks using keyboard acoustic emanations. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 245–254. ACM, 2006.
6. A. Bianchi, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Blacksheep: Detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, pages 341–352. ACM, 2012.
7. A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 385–395. IEEE Computer Society, 2010.
8. S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 559–572. ACM, 2010.
9. S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium*, pages 12–12, 2005.
10. M. Cox. Red hat's top 11 most serious flaw types for 2009. https://lwn.net/Articles/374752/, 2010.
11. Cuadro cpu benchmark. http://sourceforge.net/projects/cuadrocpubenchm.
12. N. Elhage. Much ado about null: Exploiting a kernel null dereference. https://blogs.oracle.com/ksplice/entry/much_ado_about_null_exploiting1.
13. T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Symposium on Network and Distributed Systems Security (NDSS)*, pages 191–206, 2003.

14. Z. Gu, W. N. Sumner, Z. Deng, X. Zhang, and D. Xu. Drip: A framework for purifying trojaned kernel drivers. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2013.
15. O. Hofmann, A. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with osck. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 279–290. ACM, 2011.
16. M. G. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Symposium on Network and Distributed Systems Security (NDSS)*, 2011.
17. G. H. Kim and E. H. Spafford. The design and implementation of tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS)*, pages 18–29. ACM, 1994.
18. E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis, and S. Ioannidis. You can type, but you can't hide: A stealthy gpu-based keylogger. *Proceedings of the 6th European Workshop on System Security (EuroSec)*, 2013.
19. J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, pages 195–208. ACM, 2010.
20. Linux kernel 'sock_sendpage()' null pointer dereference vulnerability. `http://www.securityfocus.com/bid/36038`.
21. R. Liu, D. Shen, C. Yang, S. Yu, and C. M. Wang. Nvm duet: unified working memory and persistent store architecture. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 455–470. ACM, 2014.
22. Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi. Cpu transparent protection of os kernel and hypervisor integrity with programmable dram. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 392–403. ACM/IEEE, 2013.
23. H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, pages 28–37. ACM, 2012.
24. J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 13th Annual Symposium on Network and Distributed System Security Symposium (NDSS)*, 2005.
25. T. Ormandy. Another kernel null pointer vulnerability. `http://lwn.net/Articles/347006/`.
26. V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy (SP)*, pages 601–615. IEEE, 2012.
27. N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th Conference on USENIX Security Symposium*, pages 179–194, 2004.
28. N. L. Petroni Jr, T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th conference on USENIX Security Symposium*, pages 15–22, 2006.
29. N. L. Petroni Jr and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 103–115. ACM, 2007.
30. D. Raywood. Sinowal trojan steals data from around 500,000 cards and accounts. *SC Magazine*, 2008.

31. rd. Writing linux kernel keylogger. https://www.thc.org/papers/writing-linux-kernel-keylogger.txt.
32. R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Recent Advances in Intrusion Detection (RAID)*, pages 1–20. Springer, 2008.
33. D. Rosenberg. Interesting kernel exploit posted. https://lwn.net/Articles/419141/.
34. D. Rosenberg. Linux kernel <= 2.6.37 - local privilege escalation. http://www.exploit-db.com/exploits/15704/.
35. E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th Conference on USENIX Security Symposium*, 2011.
36. A. Slowinska and H. Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European Conference on Computer systems (EuroSys)*, pages 61–74. ACM, 2009.
37. B. Spengler. On exploiting null ptr derefs, disabling selinux, and silently fixed linux vulns. http://seclists.org/dailydave/2007/q1/224.
38. S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, pages 61–75, 2011.
39. S. Vogl, J. Pfoh, T. Kittel, and C. Eckert. Persistent data-only malware: Function hooks without code. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.
40. M. Vuagnoux and S. Pasini. Compromising electromagnetic emanations of wired and wireless keyboards. In *Proceedings of the 18th Conference on USENIX Security Symposium*, pages 1–16, 2009.
41. H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *Proceedings of the 6th USENIX conference on Operating Systems Design and Implementation (OSDI)*, volume 4, pages 245–257, 2004.
42. Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pages 545–554. ACM, 2009.
43. J. Xiao, Z. Xu, H. Huang, and H. Wang. Security implications of memory deduplication in a virtualized environment. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.
44. H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *Proceedings of the 15th Annual Symposium on Network and Distributed Systems Security (NDSS)*, 2008.
45. H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 116–127. ACM, 2007.
46. L. Zhuang, F. Zhou, and J. D. Tygar. Keyboard acoustic emanations revisited. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pages 373–382. ACM, 2005.