College of
William & Mary
Department of Computer Science

WM-CS-2008-05

Detecting Kernel Level Keyloggers Through Dynamic Taint Analysis

Duy Le, Chuan Yue, Tyler Smart, Haining Wang

May 7, 2008

Detecting Kernel Level Keyloggers Through Dynamic Taint Analysis

Duy Le, Chuan Yue, Tyler Smart, Haining Wang

Abstract

Keyloggers as invisible keystroke recorders have posed a serious threat to user privacy and security. It is difficult to detect keyloggers, especially kernel keyloggers that operate at the operating system's kernel level, because of their inconspicuous activities and flexible interception methods. In this paper, we propose a framework using a dynamic taint analysis technique to detect kernel level keyloggers. Our design is originated from the observation that kernel keyloggers usually manipulate the data flow of a keyboard driver in order to record typed keystrokes. By tainting and monitoring the keystroke data, this framework detects and analyzes any illegitimate uses of the tainted keystroke data. Based on Argos, a system that can perform host-based intrusion detection and support dynamic taint analysis, we build a prototype of the proposed framework and evaluate its effectiveness. Our experimental results show that the proposed framework can accurately detect kernel level keylogging activities and identify their root causes.

1 Introduction

Keyloggers are software or hardware tools that capture a computer user's keystrokes and then send this information back to attackers. While they have been around for decades, keyloggers are now posing a much greater threat than before because of their rapid growth and wide spreading over the Internet. According to a report published in March 2007 by Kaspersky Lab [9], the number of keyloggers rose by more than 500% between January 2003 and July 2006. Meanwhile, keyloggers are becoming more diverse, sophisticated, evasive, and increasingly difficult to detect, particularly those kernel keyloggers that reside at the OS kernel level and can directly access various system sources to intercept user inputs.

A kernel level keylogger can act as keyboard driver or replace some functions of an original driver to obtain any information from a keyboard. Comparing to application level keyloggers, kernel level keyloggers are more difficult to write, but are more powerful and harder to detect. The detection challenge is mainly due to their flexibility and effectiveness in embedding and hiding themselves at the low system level [30, 10]. Currently, very few solutions exist to detect kernel level keyloggers, and the existing application level keylogger detection techniques are not applicable to the detection of kernel level keyloggers.

In this paper, we present a framework to detect kernel level keyloggers by using a dynamic taint analysis technique. The design of the framework is based on the observation that kernel keyloggers usually manipulate the data flow of a keyboard driver in order to record typed keystrokes. Its key feature is to first taint (i.e., mark) the keystroke data when it comes into a keyboard driver, and then track the tainted data as it passes through keyboard driver components. If any modification occurred to the tainted data before it is sent to user level applications, we can accurately detect the modification and identify its root cause.

Our framework uses a host-based Intrusion Detection System (IDS) to taint, monitor, and examine the keyboard data at the keyboard device driver level. The tainted data includes the keyboard data content and its memory address. When a kernel level keylogger invokes its logging module to write the tainted keyboard data, this malicious use will be captured and analyzed by our detection module implemented in the IDS. The synergy of the interception inside the kernel level system calls and the analysis of dynamically tainted data makes our framework a unique and effective approach to detecting kernel keyloggers.

- System call triggering: We write a new system call that works as a trigger to pass the control of the keystroke data flow from the OS kernel to the IDS. The keyboard data is tainted and monitored by the IDS before it is accessed by the keyloggers. The trigger works well, due mainly to the two facts: the keyboard driver structure is generic and the primitive OS system calls are modifiable.

- Dynamic data tainting: We take advantage of the functionality of the IDS to dynamically taint and monitor keystroke data flow. The tainting is applicable in that many IDSes are now capable of monitoring the kernel level data flow and modifying it as needed.

We build a prototype of the proposed framework based on Argos, a system that performs host-based IDS functionalities and supports dynamic taint analysis [26]. We validate the effectiveness of the prototype by distinguishing Vlogger, a typical Linux kernel keylogger [8], from a number of representative benign applications. Our experimental results demonstrate that the proposed framework can accurately detect the existence of kernel level keylogging activities and identify their root causes, while incurring moderate overhead.

The remainder of this paper is structured as follows. In Section 2, we review the background of keylogging techniques. In Section 3, we analyze the Linux keylogging model. In Section 4, we detail the proposed detection framework. In Section 5, we evaluate the effectiveness of our prototype implementation. In Section 6, we survey related work. In Section 7, we discuss the limitations of our framework and suggest future works. Finally, in Section 8, we conclude the paper.

2 Keyloggers

Keyloggers are generally classified into two major categories: hardware keyloggers and software keyloggers [30, 13, 35, 32]. Hardware keyloggers are small devices that can be easily plugged in between a computer keyboard and computer ports such as PS/2 or USB, while software keyloggers are programs running on a computer.

Camouflaged as part of a keyboard cable or as built-in components of keyboards, hardware keyloggers can

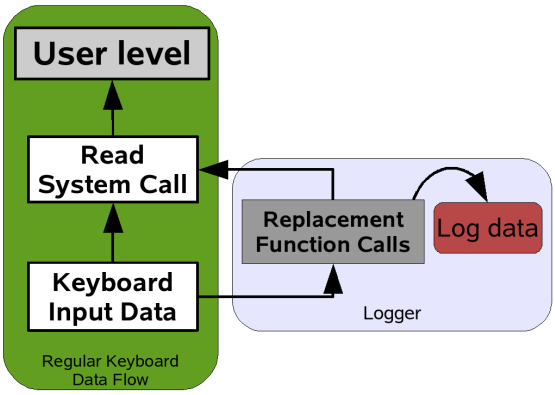

Figure 1. Keystroke data flow alteration made by kernel level keyloggers

be unnoticed by human users for a long time, and their logging activities can be hardly detected by host-based IDSes. Hardware keyloggers can also be more harmful than software keyloggers because they are capable of logging keystrokes from the moment a computer is turned on and thus can even acquire BIOS passwords. However, they are not as pervasive as software keyloggers because the installation of a hardware keylogger requires physical accesses to the victim's computer.

By contrast, software keyloggers can be easily developed and distributed by attackers. Without users' awareness, software keyloggers can be installed on computers either by piggybacking with normal file downloads, or by launching "drive-by download" attacks [24]. There are two kinds of software keyloggers: kernel level keyloggers *(kernel keyloggers)* and application level keyloggers, depending on at which system level a keylogger is running. Application level keyloggers generally use high level system functionalities to monitor keystrokes. For example, in Windows systems, the *SetWindowsHookEx* function can install an application-defined hook procedure into a hook chain to monitor certain keyboard events, and the *GetAsyncKeyState* function can determine whether a key is up or down when the function is called and whether the key is pressed after a previous call to this function. Application level keyloggers are easy to write and are relatively easy to detect. Both signature-based and hook-based techniques can effectively detect these application level keyloggers [30].

Kernel lever keyloggers run at the OS kernel level and record keystroke data directly from keyboards. In general, a kernel level keylogger works by replacing a specific keyboard driver function call with its own function call that includes a logging module, as shown in Figure 1. This logging module is programmed to record all keystrokes passed through the modified function call. Such a logging mechanism makes a kernel level keylogger almost invisible to existing anti-keylogger mechanisms, which mainly run at the application level. Detecting and preventing kernel level keylogger is still a challenging task [32, 35]. However, although these kernel keyloggers can effectively hide their logging activities from application level detectors, they usually need to maliciously access keystroke data and record it. This observation motivates us to explore the potential of using dynamic taint analysis techniques to detect kernel level keyloggers.

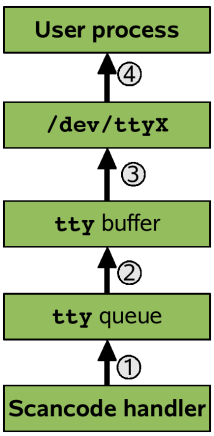

Figure 2. The general model of Linux keyboard driver

3 Linux Keylogging Model

Keyboard drivers under different OS kernels have similar structures and operations [11, 13]. They normally consist of multiple components performing different operations upon keystroke data. In this section, we analyze the Linux kernel keylogging model in detail, due mainly to the open source nature of the Linux system.

In the Linux system, a kernel keylogger is usually complied as a loadable kernel module (LKM), which can be flexibly loaded into the kernel. LKMs generally can access hardware devices, and control or record exchanged data as needed. In particular, keylogging modules can be placed into different keyboard driver components to obtain keyboard data, pass it to user level, and then store it into a file, or send it out over the network.

Linux kernel keyloggers work via one of the two methods: registering interrupt handlers or replacing driver functions. The first method is based on the Intel architecture, in which the interrupt request (IRQ) of a keyboard controller is a given number (i.e. IRQ 1), and both keyboard data and register status are exchanged through the fixed ports `0x60` and `0x64`. A keylogging module registers its own IRQ handler to read the keyboard data and its status. However, this method is not portable because of its platform dependency. The second method simply replaces certain keyboard driver functions with the maliciously modified ones, and thus is more generic. In the following, we describe the different functionalities of Linux keyboard driver components and discuss the corresponding replacements by keyloggers.

Figure 2 illustrates the general model of Linux keyboard driver. When a keyboard button is pressed, a sequence of up to six scancodes are generated by the keyboard chip. These unreadable scancodes are transfered to the keyboard driver, and then parsed and converted into a series of press or release key actions based on the translation table of the scancode handler component (1).

After the key actions are converted into key symbols with reference to an appropriate keymap, they are queued at the *tty* buffer (2). As a major component of the Linux *tty* core, the *tty* buffer guarantees the stability in generating *key combinations*, which are readable characters. Key combinations may vary for different keyboard types, but most drivers can cover basic key combinations to make standard keyboards work properly.

Finally, a *tty* buffer related function is periodically called to get characters from the *tty* buffer and place them into a *tty* read queue (3). Normally user level applications use the `read()` system call to retrieve the queued characters (4).

Keyloggers can replace keyboard driver functions

and log keystrokes in three different ways: (I) intercepting handling scancodes or queuing key combinations, (II) collecting data at a receiving buffer, and (III) modifying a reading buffer function or reading system call.

(I) `handle_scancodes()` and `put_queue()`: a keylogger can replace either the `handle_scancodes()` function or the `put_queue()` function to directly log the scancodes. However, these replacements are uncommon. This is because the captured scancodes are unreadable, the module is platform dependent, and the keylogger is unable to log remote session keystrokes.

(II) *tty* buffer: a keylogger can replace the low level *tty* driver function `receive_buf()` to log received key combinations. The keylogger replaces this function by pointing the `ldisc.receive_buf` to a new function that includes a logging module. The new function can log all readable characters at the *tty* buffer and record remote session keystrokes. Because *tty* buffer is common in most platforms[27, 35], this method is the most effective and popular logging mechanism used in keyloggers.

(III) `tty_read()` and `sys_read()`: whenever a process reads a character from the *tty* core via `sys_read()` function, the function `tty_read()` is called. Hackers can replace these two functions with their modified ones. However, since `read_tty()` and `sys_read()` are heavily used by different processes, their replacements could significantly slow down a system, thereby making them easy to be detected.

Overall, logging at the *tty* buffer is the most reliable among all these different methods. The reason lies in that it enables an attacker to directly acquire readable key combinations while leaving few obvious footprints in the system [8, 11]. Therefore, our detection framework focuses on detecting keyloggers that employ this logging method. Note that the framework is also applicable to keyloggers that use other logging methods, albeit requiring certain extensions.

4 Framework Design

In this section, we give an overview of the framework, and then detail its functionalities at the OS level and the IDS level, respectively. Finally, we present the design flexibility of the framework.

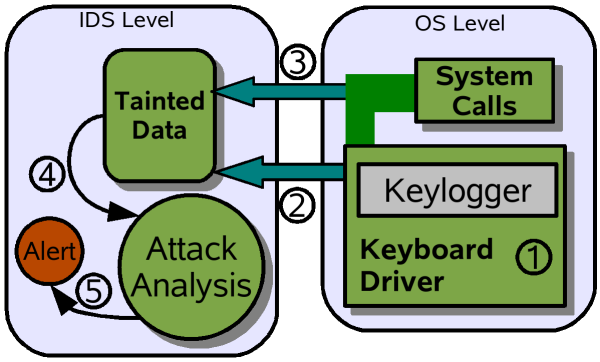

Figure 3. Framework overview

4.1. Framework Overview

As shown in Figure 3, the whole framework consists of two levels of functionalities: the OS level functionalities and the IDS level functionalities. The main detection components are located at the IDS level, while the OS level functionalities are in charge of passing keystroke data and related information to the IDS level.

As soon as keystroke data is entered into a keyboard driver **(1)**, an OS level trigger is activated to pass the keystroke data information from the OS level to the IDS level. At the IDS level, the data is labeled with a tag as tainted. A tainted-data record is created to monitor the data propagation **(2)**. After that, if any suspected operations perform on the tainted data, such as copying, moving, or writing, the OS level will send the suspected data information **(3)** to the IDS level for analysis **(4)**. Based on the tainted-data records, the IDS attack analysis component raises an alert on the suspected operations **(5)**.

4.2. OS Level

The OS level passes information related to the keystroke data to the IDS level for tainting and analyzing. The related information includes a data content and its correspondent memory address, and an occurred operation. The data content is the keystroke data. To record the keystroke data, keyloggers use some specific operations to export data to file systems or networks. Therefore, we also need to monitor the occurrences of these operations.

```
struct passed_data
{
 dt_content; /* Data content, ASCII characters */
 dt_addr; /* Correspondent memory address */
 pro_id; /* Current process ID that invokes the data passing */
 is_written; /* Flag to notice occurred writing operations */
};
```

Figure 4. Passed data between two levels

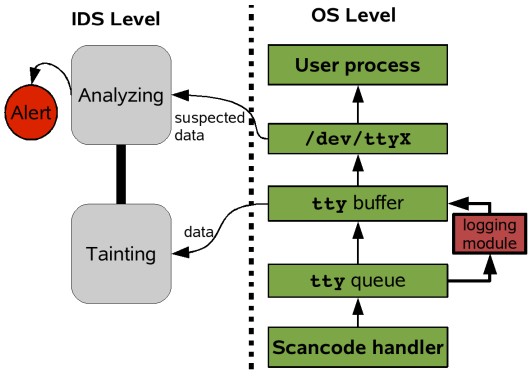

Figure 5. The framework with the logging module at the OS level, and the analysis modules (tainting and analyzing) at the IDS level

The *tty* buffer is a character buffer whose size is 512 bytes. This buffer is a part of the *tty* device. Under both Linux kernel 2.4 and 2.6, the *tty* device is represented by a structure `tty_struct`. We add a trigger in the `n_tty_receive_buf()` function to get the original keystroke data. Note that this function is invoked by the terminal driver to store a keystroke data into the *tty* buffer. Through the `tty_struct` in that function, we can obtain the keystroke data and its corresponding memory address in the *tty* buffer [27]. To pass these data to the IDS level for tainting, we use a new system call that is invoked by the trigger.

Figure 4 depicts the data information that is passed from the OS level to the IDS level. In the `passed_data`, we use the `is_written` flag and the current process ID (PID) to monitor the operations performed on the data. The flag `is_written` is set when the data is touched by a *write* operation. To get a PID, we use `current->pid`, instead of the system call `getpid()`. The PID here refers to the process that invokes the function `n_tty_receive_buf()` to store the keystroke data into the *tty* buffer. This PID is the same as the PID of the user-level process that controls the keystroke data from the *scancode handler* to the *user level*.

A general host-based IDS can monitor not only keyboard data, but other types, such as data from other devices, networks, or user level applications. Therefore, in our framework, the keyboard data needs to be sent to the IDS level in a particular way. Under the Linux kernel, IRQs and registers are used to exchange data.

- IRQs In the Linux kernel, IRQs can be used to synchronize events among kernel components. In our framework, a readiness of the keystroke data in the *tty* buffer is considered as an event. When such a event happens, we use an IRQ to trigger the sending data.The selection of this IRQ should be conflict-free with other used IRQs. For instance, the IRQ `0x80` is used for regular primitive system calls. We use the IRQ `0x82` in the framework because of its availability.

- Registers Registers are used to store data exchanged among kernel components. For example, to store system call parameters, the following registers of Linux kernel are used in an increasing order: `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, and `%ebp`. Since the host-based IDS needs to capture the exchanging data via system calls, we assign specific registers for the purpose. Note that because of a limited number of registers in the 80x86 architecture, the assignment has to satisfy two conditions addressed by Bovet *et al*. [14]: each parameter length cannot exceed the length of a register (32 bits in the 80x86 processor) and the number of parameters cannot exceed six. In our framework, the system call number is passed through `%eax`, while the other `passed_data` elements are stored in `%ebx`, `%ecx`, `%edx`, `%esi`, and `%edi`.

As shown in Figure 5, suspicious data is passed from the OS level to the IDS level.The suspected data may be used by either memory operations or a *write* operation. In the following, we describe the two scenarios in detail.

First, the keystroke data in the *tty* buffer may be used by other kernel function calls. These function calls can copy or move the keystroke data to another memory location by using primitive kernel macro functions, such as `*memcpy()`, `*memmove()`. To re-

lease these allocated kernel memory areas, the function `kfree()` is invoked. Suppose that keyloggers know the memory location and can record the data, thus, we need to monitor the possible data propagation. To monitor the propagated data, we use triggers to deliver its related information to the IDS level for analysis. Note that these functions can be used by other processes, therefore, the IDS level needs to check whether the data has been tainted before. To help the IDS level monitor the data propagation, we need to specify the source addresses of three memory functions. To differentiate the three memory functions, we introduce a function indicator number. Thus, both the source address and the function indicator number are added into the `passed_data`.

Second, to record the keystroke data, the keylogger may use one of two writing functions: *tty* core `write()` and `sys_write()`. The former is used to write the tainted data to other *tty* devices, and the latter is used to export the tainted data to the file system or network I/O. To intercept these *write* operations that may perform on the tainted data, we use a trigger to send the tainted-data related information to the IDS for analysis. To help the IDS level differentiate the suspected data from the original keystroke data that needs to be tainted, the `is_written` flag is set in the `passed_data`.

Figure 6 depicts the high level function definition of the OS level triggering. Two new system calls `sys_passing()` and `sys_propgt()` primitively invoke the routine `_sysKBCall()`. The function `sys_passing()` is activated based on two events: the readiness of a keystroke data in the *tty* buffer and the occurrence of a writing operation performed on the suspected data. The function `sys_propgt()` is executed based on the memory operations of the propagated data. The source address and the function indicator number are represented by `sr_addr` and `m_id`.

4.3. IDS Level

The host-based IDS works as an analysis component to process the keystroke data in two steps: tainting and analyzing.

```
asmlinkage int sys_passing(data, addr){
 /*Passing passed_data and triggering for tainting or analyzing*/
 ...
 passing(data, addr, current_id, is_written);
}

asmlinkage int sys_propgt(data, addr, sr_addr, m_id){
 /* Passing propagated data for tainting*/
 ...
 propgt(data,addr,current_id,sr_addr,m_id);
}

 /* Passing data using the kernel wrapper routine */
_sysKBCall(type, name, data, addr, id, arg4, arg5){\
 long __res; \ __asm__ volatile ("int $0x82" \
  :  "=a" (__res) \
  :  "" (__NR_##name), "b" (data), \
  "c" (addr), "d" (id), "S" (arg4), "D" (arg5)); \
}
```

Figure 6. Triggering at the OS level

4.3.1 Tainting

Based on the OS level triggering, the tainting component taints the keystroke data and maintains the tainted data in a tainted-data table. The IDS level structures the tainted data table based on the `passed_data`. As a multi-row table, each row contains the data content and its address, the corresponding PID, and the tag. As an assigned one-bit value, the tag is set to one indicating that the data is tainted. The tainting component examines the received data to identify whether it is an original keystroke data or the propagated data. The detailed procedure is described as follows.

First, the tainting component checks the `passed_data` received through the trigger `sys_passing`. This data needs to be tainted if it is not used by *write* operations. Then, the tainting component checks the suspected data received through the trigger `sys_propgt`. Based on the source address, we verify whether the data is tainted or not. This data is ignored if its source address is not in the tainted-data table. If the source address in the table, the tainting component differentiates the received data based on the function indicator. For the `*memcpy()` function, the received data is tainted and stored in the tainted-data table. For the `*memmove()` function, the data at the new memory location is tainted, and the one at the old memory location is untainted. Finally,

for the `kfree()` function, we need to untaint the tainted data by removing its tag in the tainted-data table.

4.3.2 Analyzing

The analyzing component first checks whether or not the suspected data is tainted. If not tainted, we ignore the data since it belongs to a legitimate writing process. If tainted, we need to verify whether it is maliciously used for the *write* operations. To detect a malicious *write* operation, the checking component compares the PID values received from the `passed_data` and those in the tainted-data table. After the comparisons, if these PID values are mismatched, it means that this *write* operation is performed by a malicious kernel logging process. Then, the IDS level exports an alarm. The alarm includes the PID of the malicious logging process, the keystroke data, and the correspondent memory address. Otherwise, if these PID values are matched, it means that the *write* operation belongs to an user-level process. Then, the IDS level just exports a notice. The contents of a notice are similar to those of an alarm. However, a notice contains the user-level PID of the application.

Note that the PIDs stored in the tainted-data table are associated with user-level applications to control the keystroke data flow. During the taint propagation, the memory operation functions, `*memcpy()` and `*memmove()`, can be invoked by the processes referred by the PIDs above. These processes are different from the *write* operation process of the keylogger.

The high level function definition of the trigger at the IDS level is shown in Figure 7. Supposed that 253 and 254 are the system call numbers of the two triggering system calls `sys_passing()` and `sys_propgt()`, through the handler `handle_keyboard_syscalls()`, the IDS level activates the tainting and analyzing components upon these numbers.

```
handle_keyboard_syscalls(NR_sysalls) {
switch (NR_syscalls)
   case 253: /*Passing: Tainting and Analyzing*/
      /*Converting from a virtual to physical address*/
      phyaddr = get_phy_page(env, env->regs[R_ECX]);
      /*Receiving passed_data*/
      ...
      if (!is_written) /*Tainting the keystroke data*/
         tag_set_keyboard(ECXTAG, phyaddr,...);
      else /*Occurred writing operation*/
         /*Analyzing tainted data */
         if (map_istainted(phyaddr & PAGE_MASK)& ...)
              /* Data is tainted, checking the process ID*/
              if (matchedPID())
                    notice(env, PID, ...);
              else
                    alert(env, ALERT_CI, PID, ...);
              /*Untainting*/
              tag_clear(...);
         else
              /*Ignoring this data*/

   case 254: /*Taint propagating*/
      /*Receiving passed_data*/
      ...
      /*Checking tainted data*/
      if (map_istainted(phyaddr & PAGE_MASK)& ...)
         if (m_id == memcpy)
              /*Tainting the current data*/
         else if (m_id == memove)
              /*Tainting the new data, untainting the old one*/
         else if (m_id == kfree)
              /*Untainting the current data*/
      else
         /*Ignoring this data*/
   default:
      /* Notice error in passing */
}
```

Figure 7. Triggering at the IDS level

4.4. Flexibility

With minor changes, the proposed framework is able to detect other types of kernel keyloggers. Besides the `tty`-buffer-based keyloggers, there are two other approaches to stealing the keystroke data: (1) via handling scancodes or queuing combinations, and (2) via reading buffer function or reading system call. In the following, we briefly describe the necessary modifications for detecting those two kinds of kernel keyloggers.

handling scancodes or queuing combinations: To monitor the replacement of these functions, after the `handle_scancode()` function is performed, the *scancodes* and *keydown flag* data are tainted and monitored. We need to implement a trigger inside the `handle_scancode()` to monitor the propagated data. Similarly, an attack can be detected based on *write* operations, which perform on the tainted *scancode* and *keydown flag* data.

reading buffer function or reading system call: The keylogger may attempt to intercept and record the output of the `sys_read()` system call or the `read()` function. To detect these illegitimate operations, we need to taint the input of these functions before they are used. Note that the keyboard data is the input of these functions. Similarly, we also need to verify whether or not the tainted data is used by *write* operations.

5 Evaluation

In this section, we first describe the experimental setup including the Argos-based IDS and Vlogger, a typical kernel level keylogger. We then present the experimental results in terms of the detection accuracy and CPU utilization.

5.1. Experimental Setup

Portokalidis *et al*. [26] developed a containment environment named Argos for defending against worms and human orchestrated attacks. Argos is built upon QEMU [12], a fast x86 emulator. It can track network data throughout execution to identify their illegal uses as jump targets, function addresses, or instructions. Argos works as a host-based IDS, which employs dynamic taint analysis to detect exploits and protect unmodified OSes or processes.

Based on the dynamic taint analysis provided by Argos, we have built a prototype of the proposed framework in the Linux system. In the prototype implementation, we have modified Argos and Linux kernel so that the function calls and triggers can work properly to pass data for tainting and checking. For the experiments, we use the Argos version 0.2.3 to monitor a Slackware Linux system. The Argos is hosted on a Pentium 4 machine with a 1.3Ghz CPU and 512MB memory, running Linux 2.6.17.

We validate the efficacy of our prototype implementation in an environment that contains a representative kernel level keylogger, Vlogger [8], and a collection of 11 benign applications. Vlogger is an advanced Linux kernel-based keylogger, which can log keystrokes of both administrators and users via console, serial and remote sessions (Telnet, SSH). In comparison with other well-known kernel-based keyloggers such as *ttyrpld* [7] or *Sebek* [5], Vlogger is more generic, more powerful, and harder to detect. The benign applications include three web browsers (Konqueror, Opera, and Firefox), four editors (Vim, Nedit, KWord, and OpenOffice writer), and four instant messengers (Gaim, Skype, aMSN, and Kopete). They all induce various keyboard interactions. We use these benign applications to verify that the normal operations on keystroke data will not be mis-classified as the activities of a kernel keylogger.

5.2. Detection Accuracy

We first verify that without our detection framework, the original Argos is unable to detect the existence of the Vlogger, no matter whether benign applications are running or not. This result is expected because Argos is designed to detect the malicious use of data as jump targets, function addresses, and instructions, but not to detect the stealing of sensitive keystrokes. We then evaluate the detection accuracy of our framework under three different settings: only with the Vlogger, only with benign applications, with both the Vlogger and benign applications.

Only with the Vlogger: when only the Vlogger is activated in the Linux system, our detection framework can always detect the existence of the Vlogger, which maliciously copies the data in the `tty` buffer of a keyboard drive and writes to a log file. We flag alarms on the Vlogger's logging activities whenever keyboard buttons are pressed. For each logging action, we generate an alarm record that contains the Vlogger kernel process ID, the buffer content, and the related memory addresses. To further evaluate the detection capability of our framework, we also execute different remote SSH sessions in the system. In general, an SSH client connects to an SSH server through a *pseudo-tty* handler, which works as an interface between remote hardware devices and real *tty* drivers. Thus, the remote session keystrokes are passed through the *tty* buffer, and can be logged by the Vlogger. Still, our detection framework is able to detect all the logging activities of the Vlogger in these SSH sessions.

Only with benign applications: when the Vlogger is inactivated and only benign applications run in the Linux system, the possible low level operations on keystroke data caused by these applications can always

Scenario	Keylogger Detection	Mis-Classification
Vlogger Only	100%	0%
Benign Only	N/A	0%
Vlogger+Benign	100%	0%

Table 1. Framework effectiveness summary

be correctly identified. In the extensive tests of all these 11 benign applications with different keyboard activities, our detection framework does not issue any alarms or notices.

With both the Vlogger and benign applications: when the Vlogger is activated and benign applications run in the Linux system, our detection framework can always detect the logging activities of the Vlogger and issue alarms. In other words, in such a typical setting, suspected actions on keystroke data can always be detected by our detection framework and be accurately classified as either alarms or notices. Table 1 summarizes the detection accuracy of our detection framework under the three experimental settings.

5.3. CPU Utilization

Portokalidis *et al*. [26] claimed that the performance of the original Argos in the fastest configuration is at most two times slower than the vanilla QEMU. This number is almost unchanged when our kernel keylogger detection framework is applied, due mainly to the following two reasons. First, the size of the keystroke data is small, thus our detection framework can efficiently pass, taint, check, and untaint it. Second, the keystroking frequency is relatively low, and hence, the overall detection task is not heavy. For example, the study conducted by Karat *et al*. [21] shows that on average only 33 words are typed by a user in a minute.

To have a concrete view of the performance impact of our detection framework upon the original Argos system, we also measure the kernel level CPU utilization during both booting time and running time. For booting, the average startup time is 120 seconds and the CPU utilization is sampled per second. Figure 8 shows the dynamics of the CPU utilization with and without our detector during the booting time. We can see that our detector slightly increases the CPU utilization. On average, the CPU utilization is increased by 18% and the detector does not affect the overall system performance during the booting time.

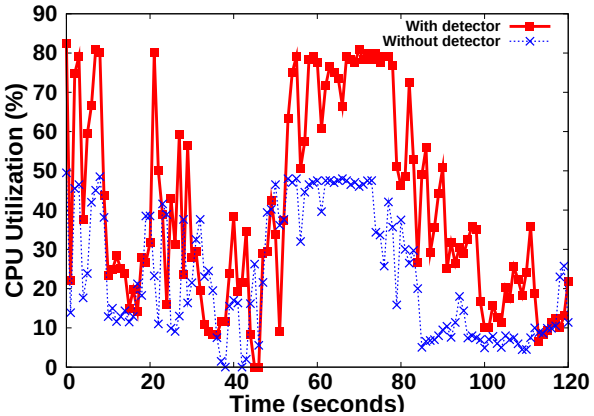

Figure 8. Booting time kernel level CPU utilization comparison

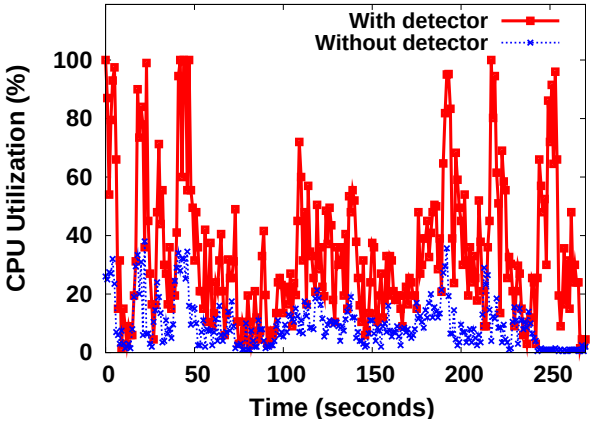

Figure 9. Running time kernel level CPU utilization comparison

In the running time, we measure the CPU utilization when both the Vlogger is activated and benign applications are running. The system is kept running for almost 5 minutes and the CPU utilization is sampled per second. Figure 9 shows the dynamics of the CPU utilization with and without our detector during the running time. On average, the CPU utilization is increased by 24% when our detection framework is activated. Only at few points, the CPU utilization reaches 100%. Therefore, the overhead incurred by our detector is very moderate.

6 Related Work

In this section, we first survey two main classes of keylogger detectors: application-based and kernel-based [10]. Then, we review the dynamic taint analysis technique. Finally, we describe other related work on defending against keyloggers.

Application keylogger detector: Since 93% personal computers world wide use the MS Windows [31], most application keylogger detectors aim to detect spyware keyloggers running on MS Windows. These keyloggers can be installed in a computer via different ways, such as email, web browser, or some remote access methods. To detect them, two different techniques, signature-based and hook-based, have been proposed.

- *Signature-based:* Based on a database of keylogger behavior, this technique monitors files, dynamic linked libraries, or registry entries that are modified or added maliciously into a system by keyloggers. Therefore, any application keyloggers whose signatures are in the database are always detectable. This technique has been applied in some commercial software, such as Prevx [4] and TrendMicro [6], but it is not effective in detecting keyloggers whose signatures are unknown.

- *Hook-based:* This technique takes advantage of the `SetWindowsHookEx()` function to monitor the keyboard status before and after passing the keystroke data between two hook procedures. Therefore, a keylogger's intercept between hook procedures can be detected. Compared to the *signature-based* technique, this technique is more effective and has been widely used. For example, in System Virginity Verifier [28] and HookFinder [36], this technique has been extended to detect hook-based malwares. Some commercial software is also equipped with this technique, such as VICE [16], Microsoft Antispyware, Ad-Aware [1], and PestPatrol [3].

Kernel keylogger detector: Since the behaviors of kernel keyloggers are stealthy, an effective solution to resolve this problem is challenging [22, 10]. A common solution is to use *anti-rootkits* to detect malicious intercepts at kernel level [2]. *Rootkits* are programs designed to corrupt the legitimate control of an OS. Some kernel keyloggers use rootkits to intercept the system calls. The *anti-rootkits* generally stay in the memory and scan the entire processes, modules, services, and loaded drivers to find suspected activities. However, because kernel keyloggers can often effectively hide their behaviors, they may not to be detected by *anti-rootkits*. In addition, *anti-rootkits* may generate high overhead by scanning many low level components of a system. Our framework uses the dynamic taint analysis technique and keeps its focus on keystroke data, thus it can be more accurate and efficient.

Dynamic taint analysis: The dynamic taint analysis technique has been widely applied to solve different security problems. To discover the zero-day worms, Crandall *et al*. [18] used this technique to monitor the information related to kernel timers. Kong *et al*. [23] applied a generic instruction-level runtime taint checking architecture for handling non-control data attacks. To detect malicious accesses, Castro *et al*. [17] monitored the tainted data flow by using a data-flow graph. Based on the notion of the pointer taintness and source code corruption point, Xu *et al*. [34] proposed an architectural technique to defeat both control and non-control data attacks. Sezer *et al*. [29] developed MemSherlock, an automated debugging tool that can identify unknown memory corruption vulnerabilities. To solve general user-level malware problems, the tainting technique has been used in some recent research projects at CMU [25, 37, 15]. Our framework differs from these previous works at two points: tainting source and triggering detection. First, instead of considering different inputs from network sockets or stdin as untrusted, our framework focuses on the data in the *tty* buffer and the propagated data. Second, to detect illegitimate use of the tainted keystroke data, our framework only focuses on the *write* operations, due to the "logging" nature of keyloggers.

Other related work: There are a few other related work on defending against keyloggers. For example, Wang *et al*. [33] designed and implemented Strider GhostBuster, which can effectively detect resource hidden malware including keyloggers. Florencio and Herley [19] proposed to use a shared-secret proxy to enter passwords on computers that may have

keyloggers installed.

7 Discussions and Future Works

In this section, we discuss potential evasions to our detection framework and our future works.

Breaking the tainting propagation and detection: We need to answer a question: *can a kernel keylogger break either the taint propagation or the detection of the framework?* A kernel malware author may design a keylogger, which not only records keystroke data from the *tty* buffer but also modifies other driver and kernel components. As mentioned in Section 2, such malicious kernel modifications are possible.

For the tainting propagation, we rely on an assumption that kernel keyloggers cannot intercept the three memory macro functions: `memmove()`, `memcpy`, and `kfree()`. In a real system, this assumption is reasonable because a host-based intrusion prevention system (HBIP) can be used to prevent malicious changes on these macro functions. For this reason, we argue that it is difficult for a keylogger to break the tainting propagation of our detection framework.

To evade the detection, keyloggers can either intercept the original *write* operations or use its own *write* functions. To prevent an interception of the original *write* functions, we can still use an HBIP system. For this case, kernel keyloggers cannot break the detection. The use of other *write* operations by kernel keyloggers may make the detection more challenging. But the problem is still tractable by correspondingly extending the detection points of our framework.

Future works: In general, taint-analysis-based systems have high overhead for monitoring the taint propagation. For example, the original Argos [26], HookFinder [36], and Valgrind [25], with additional instrumentations, are normally two times slower than vanilla frameworks. Although the overhead of our framework is moderate, there is still space to further improve the efficiency by integrating the techniques proposed in some recent works such as VMscope [20] and Panorama [37]. We consider this improvement as a part of our future works. We also plan to evaluate the design flexibility of our detection framework by implementing prototypes and conducting experiments on different OS kernel architectures.

8 Conclusion

In this paper, we developed a framework based on the dynamic taint analysis to detect kernel level keyloggers that intercept user inputs from a keyboard driver, specifically the `tty` buffer. The framework taints and monitors the keyboard data flow, and then detects any illegal access on the tainted keystroke data. We built a prototype of the proposed framework in the Linux system. To validate the detection efficacy of the prototype, we employed Argos emulator as the underlying host-based IDS and used Vlogger as testing keylogger. The experimental results show that our prototype can accurately detect kernel keylogging activities and identify their root causes.

References

[1] Ad-aware anti-spyware. `www.lavasoftusa.com/software/`.

[2] Freeware antirootkits. `http://wiki.castlecops.com/Lists_of_freeware_antirootkit`.

[3] Pestpatrol home edition anti-spyware. `www.pestpatrol.com/Products/PestPatrolHE/`.

[4] Prevx csi scanner. `http://info.prevx.com`.

[5] Sebek homepage - a part of the honeynet project. `http://www.honeynet.org/tools/sebek`.

[6] Trend micro. `www.TrendMicro.com`.

[7] ttyrpld - tty logging daemon. `http://ttyrpld.sourceforge.net`.

[8] Vlogger at the hacker's choice. `www.thc.org`.

[9] Keyloggers become cyber criminal tool of choice, 2007. `http://www.vnunet.com/vnunet/news/2186806/keyloggers-become-cyber`.

[10] Keystroke logging, 2007. `http://en.wikipedia.org/wiki/Keystroke_logging`.

[11] R. at vnsecurity.net. Writing linux kernel keylogger, June 2002.

[12] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[13] M. bm63p@yahoo.com. Kernel based keylogger, 2005. `www.packetstormsecurity.org/UNIX/security/kernel.keylogger.txt`.

[14] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[15] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 2–16, Washington, DC, USA, 2006. IEEE Computer Society.

[16] J. Butler and G. Hoglund. Vice - catch the hookers - (plus new rootkit techniques), July 2004. http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-0%4-butler.pdf.

[17] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association.

[18] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 235–248, New York, NY, USA, 2005. ACM.

[19] D. Florencio and C. Herley. Klassp: Entering passwords on a spyware infected machine using a shared-secret proxy. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 67–76, Washington, DC, USA, 2006. IEEE Computer Society.

[20] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138, New York, NY, USA, 2007. ACM.

[21] C.-M. Karat, C. Halverson, D. Horn, and J. Karat. Patterns of entry and correction in large vocabulary continuous speech recognition systems. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 568–575, New York, NY, USA, 1999. ACM Press.

[22] K. Kasslin. Kernel malware: The attack from within. In *The 9th Annual AVAR International Conference*, 2008.

[23] J. Kong, C. C. Zou, and H. Zhou. Improving software security via runtime instruction-level taint checking. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 18–24, New York, NY, USA, 2006. ACM Press.

[24] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware on the web. In *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium (NDSS 2006)*, San Diego, CA, February 2006.

[25] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[26] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev.*, 40(4):15–27, 2006.

[27] A. Rubini and J. Corbet. *Linux Device Drivers, 2nd Edition*. Oreilly & Associates Inc, 2001.

[28] J. Rutkowska. System virginity verifier defining the roadmap for malware detection on windows system. http://www.invisiblethings.org/papers/hitb05_virginity_verifier.ppt.

[29] E. C. Sezer, P. Ning, C. Kil, and J. Xu. Memsherlock: an automated debugger for unknown memory corruption vulnerabilities. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 562–572, New York, NY, USA, 2007. ACM.

[30] S. Shetty. Introduction to spyware keyloggers, 2005. www.securityfocus.com/infocus/1829.

[31] S. Shetty. Operating system market share, May 2007. http://marketshare.hitslink.com/report.aspx?qprid=2\&qpmr=15\&qpdt=1\&qpct=3%\&qptimeframe=M\&qpsp=100.

[32] K. Subramanyam, C. E. Frank, and D. H. Galli. Keyloggers: The overlooked threat to computer security. In *MCURCSM*, 2003.

[33] Y.-M. Wang, D. Beck, B. Vo, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 368–377, Washington, DC, USA, 2005. IEEE Computer Society.

[34] J. Xu, N. Nakka, S. Chen, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 378–387, Washington, DC, USA, 2005. IEEE Computer Society.

[35] M. Xu, B. Salami, and C. Obimbo. How to protect personal information against keyloggers. In *IMSA*, pages 275–280, 2005.

[36] H. Yin, Z. Liang, and D. Song. Hookfinder: Identifying and understanding malware hooking behavior. In *NDSS - Proceedings of the 15th Annual Network and Distributed System Security Symposium*, February 2008.

[37] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.