Flying Under the Radar:

Maintaining Control of Kernel without Changing Kernel Code or Persistent Data Structures

Jinpeng Wei
Florida International University
weijp@cs.fiu.edu

Calton Pu
Georgia Institute of Technology
calton@cc.gatech.edu

Keke Chen
Wright State University
keke.chen@wright.edu

ABSTRACT

Cyber-spies rely on technologies such as rootkits to maintain a stealthy control of the victim kernel. Current techniques can detect changes to kernel code (e.g., SecVisor) and data (e.g., SBCFI), but have difficulties with transient kernel control flow attacks that insert execution requests into interrupt or kernel work queues (K-queues) without changing kernel code or data. Two examples implemented using Linux *tasklets* illustrate the effectiveness of K-queue attacks: key logger and CPU cycle stealer. Possible defenses to protect the kernel against K-queue attacks are outlined.

1. INTRODUCTION

As power grid, a critical national infrastructure, becomes more intelligent, the attacks against it moves into the cyber space. Compared with their physical world counterpart, cyber attacks against the power grid can be more elusive and dangerous. For example, U.S. officials worry that cyber-spies could use their demonstrated access to take control of power plants during a time of crisis or war [6].

One characteristic of such cyber-spies is they rely on stealthy malware (e.g., rootkits [4, 7]) to stay hidden before the actual strike (e.g., shutting down the grid). Therefore, if we are to defeat such cyber-spies, we have to better understand their capabilities, e.g., the technologies that they can leverage to hide themselves. For example, a botnet enlists a compromised computer into a network of similar computers under the control of a bot master. Such a command-and-control infrastructure seems ideal for an adversary that dispatches cyber-spies into the target space so that they can be commanded to launch an attack later on. As a matter of fact, botnets have already become a major threat. Malware development tools such as rootkits help attackers to break-in and to maintain control of victim nodes. The sheer size of successful botnets shows that: (1) they are able to break-in, and (2) they are able to provide useful work (for the attacker) while escaping detection and removal for a significant period of time after break-in. This paper focuses on a method to maintain stealthy control of the kernel (the second part).

We divide the malwares that attempt to maintain stealthy control of kernel (after successful break-in) into three broad classes. The first class modifies kernel code on disk or in memory. This class can be detected by virus scanners that scan memory and disk files for malware signatures or integrity monitors such as SecVisor [11] that detect unauthorized changes. The second class, called persistent kernel control flow attacks, does not change code, but makes persistent modifications to kernel data (e.g., function pointers in the system call table). Representative detectors for such modifications (such as SBCFI [9]) work by comparing the runtime value of the kernel data with known good values.

This paper investigates the third class, *transient* kernel control flow attacks, that are capable of achieving continual malicious function execution without changing either kernel code or persistent function pointers. This class of attacks exploits dynamic schedulable work units in modern multi-threaded kernels. For example, a device driver can request a dynamic soft timer to schedule the execution of a timed event handling callback function. At the specified time, the kernel invokes the callback function, trusting its code. We show that malware executing in kernel mode can insert a malicious callback function to maintain control of kernel and perform work for attackers.

The main contribution of this paper is a detailed description of transient kernel control flow attacks based on dynamic schedulable kernel queues (K-queues). This description includes: (1) an enumeration of dynamic kernel threads and kernel control flow transfers that can be used for maintaining stealthy kernel control, (2) a description of this kind of attacks, with two illustrative malware examples: a stealthy keylogger and a stealthy cycle stealer. A case study with proposed defense mechanism is outlined (details can be found in [12]).

The rest of the paper is organized as follows. Section 2 outlines the background information on kernel control flows and K-queues. Section 3 discusses an attack model that manipulates K-queues for persistent execution of the malware, and describes two illustrative malware examples that use tasklets. Section 4 discusses possible mitigations and defenses for the K-queue-driven attacks. Section 5 concludes the paper.

2. KERNEL CONTROL FLOWS AND SCHEDULABLE QUEUES

2.1 KERNEL CONTROL FLOWS

In this paper, we use Linux as a concrete and representative multi-threaded kernel. The Linux kernel can have a number of control flows (listed in Figure 1): exception handlers, interrupt service routines, Softirqs, and kernel threads such as work queues [3].

Of the various kinds of kernel control flows, exception and interrupt handlers execute at the highest priority, usually with interrupts disabled. Some exception and interrupt handler operations are interruptible and executed in Softirqs, for example, sending the keyboard line buffer to the terminal handler process. Softirqs are invoked in interrupt context (e.g. when the service routine for an I/O interrupt is finished), but with interrupt enabled. Furthest from hardware, kernel threads execute in process context and are therefore fully interruptible. They are interleaved with user processes, with the main difference being that kernel threads

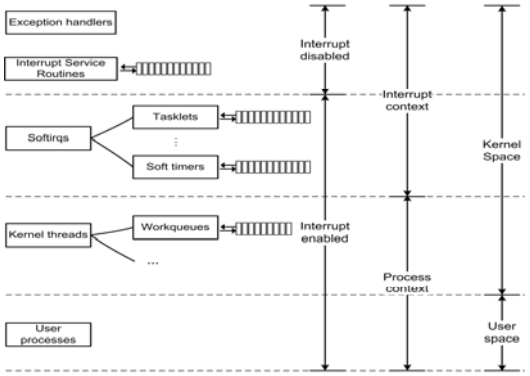

Figure 1: Kernel Control Flows with Schedulable Queues

execute in kernel context while user processes execute in user process context.

2.2 K-QUEUES IN THE LINUX KERNEL

The kernel control flows outlined in Figure 1 are executed by the kernel through *kernel schedulable queues* or K-queues for short. These K-queues are implemented as linked lists. In this section, we discuss four representative K-queues (with descending execution priorities).

2.2.1 IRQ Action Queues

When an interrupt happens, the Interrupt Descriptor Table (IDT) is used to find the corresponding Interrupt Service Routine (ISR), which may in turn delegate the interrupt handling to several *IRQ actions*. This is because multiple I/O devices can share an interrupt pin; therefore each of them may have its own way of handling the shared interrupt. The Linux kernel uses IRQ action queues to support such interrupt sharing. Each element of an IRQ action queue is a structure **irqaction** (Figure 2), which contains a *handler* field, a *dev_id* field, a pointer to the next element in the queue (the *next* field), and other information. The *handler* field is a function pointer to the handler routine, and the *dev_id* field is used to uniquely identify the device that provides the handler routine. When an interrupt happens, the ISR invokes all handler routine in the corresponding IRQ action queue.

2.2.2 Tasklet Queues

Compared to Interrupt Service Routines, tasklets are the preferred way to implement deferrable functions in I/O device drivers because executions of tasklets are interruptible; for example, tasklets are suitable for implementing the expansion of receive buffers by a gigabit network interface card driver, which can be time consuming due to allocation of more kernel memory.

As Figure 3 shows, a tasklet request contains a callback function pointer (in the *func* field) and a *data* pointer. In Linux, a tasklet request is inserted into one of two tasklet queues (based on whether **tasklet_schedule** or **tasklet_hi_schedule** is called), implemented by two Softirqs (numbers 0 and 5). When the **do_softirq** function comes across a tasklet structure (Figure 3) during the traversal of the two queues, it invokes the callback function and passes on the *data* field as the input parameter.

2.2.3 Soft Timer Queues

Dynamic soft timer is a well-established mechanism used by many kernel components to schedule the execution of timed event handling functions. In the Linux kernel, the requester of a soft timer first prepares an instance of soft timer interrupt request (STIR) of type **struct timer_list**, which contains information about the callback function (the *function* field), a data pointer (the *data* field), and the expiration time, among others. The **add_timer** function is invoked to add this instance of STIR into a linked list of pending timers: **tvec_bases**.

The soft timer queue is implemented by a Softirq (number 1) and STIRs executed in interrupt context (Figure 1). When a STIR in the linked list expires, it is removed from the list, its callback function is invoked, and the *data* pointer is passed along to the callback function as the input parameter. Typical callback functions also create the next STIR at the end of request processing.

2.2.4 Work Queues

Work queues are used to schedule kernel threads that interleave with user processes. Compared to tasklets that execute in interrupt context, work queues execute kernel threads in kernel context.

A work queue is a linked list of work requests (Figure 4), dynamically inserted through functions such as **queue_work**. Similar to a tasklet, each work request has a callback function (the *func* field) and a *data* field. The server for a work queue is a kernel thread such as *events/0*, which executes each element in the list by invoking its callback function with the *data* field passed on as the input parameter.

3. CASE STUDIES OF K-QUEUE-DRIVEN ATTACKS

3.1 Malware Architecture

A stealthy malware can exploit the dynamic nature of K-queues to maintain stealthy control of kernel. We adopt an informal architecture of malwares that execute useful work for botnet owner/renter. We divide such malwares into three steps: (1) break-in, (2) connect to the kernel control flow, and (3) continual execution of malicious functionality. Break-in methods (step 1) such as buffer overflow [5] are well known and omitted here. After break-in, persistent kernel control flow attacks (e.g., the rootkits listed in [9]) change kernel data structures such as

```
struct irqaction {
    irqreturn_t (*handler)(int, void *,
struct pt_regs *);
    unsigned long flags;
    void *dev_id;
    struct irqaction *next;
    int irq;
};
```

Figure 2: The Definition of irqaction

```
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

Figure 3: The Definition of tasklet_struct

```
struct work_struct {
    unsigned long pending;
    struct list_head entry;
    void (*func)(void *);
    void *data;
    void *wq_data;
    struct timer_list timer;
};
```

Figure 4: The Definition of work_struct

permanent function pointers (step 2) so the kernel would regularly jump to malicious functionality and satisfy step 3.

In contrast to persistent kernel modifications, transient kernel control flow attacks insert a malicious request into a K-queue (e.g., by supplying malicious callback function or data) in step 2. The continual execution of malware (step 3) is achieved by inserting a new malicious request into the K-queue at the end of the callback function.

Advanced malware is actively misusing K-Queues to their advantage. For example, the Rustock.C spam bot relies on two Windows kernel timers to check whether it is being debugged/traced [8], and the Storm/Peacomm spam bot invokes **PsSetLoadImageNotifyRoutine** to register a malicious callback function that disables security products [2]. Given such concrete use cases, one interesting question is to what extent transient and short execution units can carry out useful work for botnet owners/renters. To answer this question, some illustrative designs of transient kernel control flow attacks are described in this section. For concreteness, the malicious functionality part uses tasklets (Section 2.2.2) to achieve significant work, including effective violation of confidentiality, integrity, and availability of a running kernel.

In these examples, malicious functionality is implemented as a Linux loadable kernel module with an initialization function that requests the first tasklet. At break-in (details omitted due to the changing methods of step 1), the malware is loaded, the kernel invokes its initialization function, and the first tasklet is inserted (step 2). The continual execution of malware (step 3) is achieved by each malware tasklet scheduling a new tasklet at the end of the callback function (e.g., using a timer as shown in Figure 5).

3.2 Stealthy Key Logger

A typical class of malware steals sensitive information from the host node. A straightforward malware implementation intercepts the kernel functions that process such sensitive information. For example, a key logger [10] can replace the keyboard interrupt handler with a malicious one that records the keyboard input. The following implemented example shows that persistent kernel modifications are unnecessary for such malicious functionality.

A tasklet-based key logger keeps kernel code and interrupt-related data structures intact. It periodically peeps into various buffers in the kernel, where the keyboard input information is stored. As Figure 6 shows, when a key is pressed, the keyboard hardware generates an interrupt. The keyboard interrupt handler fetches the key stroke information and temporarily stores it in the TTY flip buffer before transferring it into the TTY line discipline buffer. Finally, when a user-level application reads from the input device, the key stroke information is copied into the user's buffer.

```
DECLARE_TASKLET(keylogger_tasklet, log_it, 0);
struct timer_list keylogger_timer =
   TIMER_INITIALIZER(sched_me, 0, 0);
static void sched_me(void){
   tasklet_schedule(&keylogger_tasklet);   return;
}
static void log_it(unsigned long arg){
   dump_keybuffer();
   keylogger_timer.expires = jiffies + (HZ);
   add_timer(&keylogger_timer);
   return;
}
```

Figure 5: Skeleton of the stealthy key logger

A critical question about this sampling-based tasklet-driven key logger is whether it can capture every key stroke, since it depends on an effective sampling rate. The key logger can glean key stroke information from the TTY flip buffer, the TTY line discipline buffer, or the user's buffer. The TTY flip buffer has a very short retention time relative to the TTY line discipline buffer, which is a circular buffer of significant size (by default 4,096 bytes). Since each key stroke generates 2 bytes of information, the TTY line discipline buffer can keep information on up to 2,048 key strokes. Since it takes minutes to hours for the user to fill up the line discipline buffer, the key logger malware only needs to sample from time to time (e.g., once per minute should be good enough) to collect all keystrokes from the line discipline buffer.

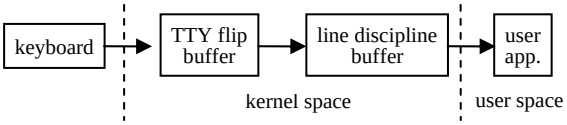

Figure 6: Flow of Keyboard Input Information in Linux

This sampling key logger has been implemented on Linux to collect key strokes on an X-window desktop. Figure 5 shows the skeleton of this key logger, which leverages a soft timer to periodically schedule new tasklets. Due to space limit, we do not show the details of the function **dump_keybuffer**. Our key logger captures keystrokes entered into X-window applications such as the Firefox web browser. These applications handle many interesting key strokes such as username and password in online banking and credit card numbers in online shopping.

In case more frequent sampling is required, the key logger will request faster soft timers (e.g., to collect information from the TTY flip buffer). In this case, techniques for hiding the higher resource consumption should be employed (see Section 3.3) to keep the key logger stealthy.

3.3 Stealthy Denial of Service Attack (CPU Cycle Stealer)

A second kind of typical attacks is denial of service (DoS, or lowered quality of service). The attack becomes effective when the DoS malware is able to hide itself and its effects from detection for a significant amount of time. In a tasklet-based attack, the callback function can perform computationally intensive work to slow down the legitimate applications such as a web server. Such a CPU cycle stealer has been implemented by inserting a program to compute the factorial of a given number in the callback function. By adjusting the value of the number and the tasklet's frequency, different slowdown factor can be obtained. For example, when there is no other competing workload and the tasklet is inserted once per second, if the factorial of 41 is computed in the callback function, about 33% of total CPU time is consumed by the malware. As the input number increases, more and more CPU time are consumed by the malware, and the CPU is saturated when the number reaches 48. This experiment is performed on an Intel Xeon at 2.93GHz with 196MB memory and 6GB hard disk.

One problem with such DoS attacks is that the CPU wastage can be noticed by system tools such as *top*, because the kernel maintains performance accounting information for different sources of computation. For example, the CPU time consumed by the above malicious callback function is attributed to "software interrupt". To hide the CPU wastage, the malicious callback function further manipulates the kernel accounting data so that the CPU time wasted by the malware factorial program is attributed towards the idle CPU time. Therefore, it is not immediately obvious why the system performance is degrading.

The CPU cycle stealer violates the availability of CPU resources and the integrity of the performance accounting information. However, since the performance accounting information is dynamic, there is no easy notion of what is normal. Besides, there can be many reasons for slowdown of the service (network congestion, server overload, etc), so it is hard to locate the true cause of the problem.

4. POSSIBLE DEFENSE AGAINST K-QUEUE-DRIVEN MALWARE

What distinguishes K-queue-driven malwares (see the illustrative examples in Section 3) is that they preserve kernel code and data control flow integrity. The ability to insert K-queue requests suffices for these attacks to work. An effective defense needs to prevent such unauthorized executions.

Runtime kernel control flow integrity checking: One possibility is to use defenses originally designed for persistent kernel control flow attacks (CFI [1] and SBCFI [9]) for transient kernel control flow attacks such as K-queue-driven malwares. For example, SBCFI performs a garbage-collection style traversal of kernel data structures to validate function pointers at runtime. However, it is non-trivial to generalize the original SBCFI proposal to cover K-queues. First, the dynamic scanning must include all the K-queue data structures and the transitive closure of their data [12]. Second, the frequency of scanning must be increased to capture the transient K-queue requests, augmenting the execution cost of this approach.

Complete mediation of K-queues: Another possible defense is to verify and validate each K-queue request (e.g, STIR) before transferring control to its callback function [12]. The validation, for example, can check the address of the callback function against a white list of legitimate callback functions. First, the determination of the white list can be achieved through automated static analysis of the whole kernel source code (including the device drivers). Second, a reference monitor can be added to the K-queue scheduling logic to validate a request for execution, using the white list. For soft timers in the Linux 2.6.16 kernel, an analysis of 3,688 source files found 365 legitimate STIR callback functions. We also implemented a checker for soft timer driven malwares based on virtualization [12].

5. CONCLUSION

With the continued deployment of cyber infrastructures for the national power grid, maintaining a stealthy control of the kernels (with the help of stealthy malware) in the power grid cyber space has become an important strategy for the adversaries. Malwares that change kernel code or data can be detected by currently known techniques such as virus scanners (code changes) and SBCFI (data changes). In this paper, we describe a third class of malware, transient kernel control flow attacks, which manipulate dynamic schedulable kernel queues (K-queues) to achieve continual malicious function execution while remaining undetected by existing detection tools, because they do not modify either kernel code or persistent function pointers. As a result, such malware poses a significant threat to power grid infrastructure security.

A concrete implementation of transient kernel control flow attacks is to insert a malware execution request into one of kernel interrupt handling or work queues (a K-queue attack). Two illustrative examples of such attacks have been implemented using tasklets in Linux: a key logger and a CPU cycle stealer. These examples show the feasibility and potential effectiveness of K-queue attacks.

Since K-queue attacks use facilities that are supported by all multi-threaded modern operating system kernels, they have the potential to become significant threats for systems such as the power grid. Potential defenses against K-queue attacks are non-trivial, since the defense must prevent unauthorized execution of K-queue requests by distinguishing legitimate requests from malware threats. This paper includes an outline of such a defense, details of which have been published.

6. REFERENCES

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*.

[2] Boldewin, F. 2007. Peacomm.C - Cracking the nutshell. *Anti Rootkit*. http://www.antirootkit.com/articles/eye-of-the-storm-worm/Peacomm-C-Cracking-the-nutshell.html.

[3] Bovet, D. and Cesati, M. 2002. Understanding the Linux Kernel, Second Edition. O'Reilly. ISBN: 0-596-00213-0.

[4] Brumley, D. 1999. Invisible intruders: rootkits in practice. *USENIX login*.

[5] Cowan, C., Pu, C., et al. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. of the 7th USENIX Security Symposium*.

[6] Gross, G. 2009. Cybercriminals Can Shut Down U.S. Electrical Grid. http://www.cio.com/article/488716/Cybercriminals_Can_Shut_Down_U.S._Electrical_Grid

[7] Hultquist, S. 2007. Rootkits: The next big enterprise threat? http://www.infoworld.com/d/security-central/rootkits-next-big-enterprise-threat-781

[8] Kwiatek, L. and Litawa, S. 2008. Yet another Rustock analysis... *Virus Bulletin*.

[9] Petroni, N. and Hicks, M. 2007. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proc. of the 14th ACM Conference on Comp. and Comm. Security*.

[10] Phrack Inc. 2002. Writing Linux Kernel Keylogger. *Phrack* Volume 0x0b, Issue 0x3b, Phile #0x0e of 0x12.

[11] Seshadri, A., Luk, M., Qu, N., and Perrig, A. 2007.SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of SOSP*.

[12] Wei, J., Payne, B. D., Giffin, J., and Pu, C. 2008. Soft-timer driven transient kernel control flow attacks and defense. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC)*.