Re-establishing Trust in Compromised Systems: Recovering from Rootkits That Trojan the System Call Table

Julian B. Grizzard, John G. Levine, and Henry L. Owen

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250, USA
{grizzard,levine,owen}@ece.gatech.edu

Abstract. We introduce the notion of re-establishing trust in compromised systems, specifically looking at recovering from kernel-level rootkits. An attacker that has compromised a system will often install a set of tools, known as a *rootkit*, which will break trust in the system as well as serve the attacker with other functionalities. One type of rootkit is a kernel-level rootkit, which will patch running kernel code with *untrusted* kernel code. Specifically, current kernel-level rootkits replace trusted system calls with trojaned system calls. Our approach to recover from these type of rootkits is to extract the system call table from a *known-good* kernel image and reinstall the system call table into the running kernel. Building on our approach to current generation rootkits, we discuss future generation rootkits and address how to recover from them.

1 Introduction

Modern computer systems are vulnerable to a wide variety of attacks. As attackers develop methods to exploit these vulnerabilities, a large number of systems are compromised. Compromises are costly to individuals, companies, governments, and other organizations in terms of data breach, downtime, administration, and recovery. The number of new vulnerabilities discovered each year is growing, and as such we believe system compromises will continue to be a problem for the foreseeable future.

Much work has been done on preventing and detecting system compromises; however, system compromises continue to be a problem. To date we have seen little work done in terms of methods for recovering from system compromises. Conventional wisdom states that one should wipe the system clean, reinstall, and patch with the latest updates.

In this paper, we begin to explore alternatives to conventional wisdom in terms of recovering from system compromises. In certain cases, it may not be possible or desirable to shutdown the system to perform a fresh install. We study kernel-level rootkit modifications to compromised systems and present tools to recover from kernel-level rootkits. Our work focuses on the Linux kernel and Red

P. Samarati et al. (Eds.): ESORICS 2004, LNCS 3193, pp. 369–384, 2004.

Hat Linux distribution. The results of our work should be applicable to other operating systems, especially those on the *x86* architecture.

We specifically discuss one of the most common tactics of modern kernel-level rootkits: trojaning the system call table. When the system call table is trojaned, even known good system binaries will not report true information about the system. Our approach to recover from such attacks is to insert a trusted system call table from a known good kernel image into the running kernel. This approach gives control back to the system administrator and is the first step in recovering from a root compromise in which a kernel-level rootkit has been installed.

Future generation kernel-level rookits may trojan other types of kernel code instead of the system call table. We discuss possible directions for future rootkits and alternative kernel penetration techniques. Our recovery approach of bootstrapping trusted code into the kernel may be useful to recover from future generation rootkits.

1.1 Definition of Compromised System

When an attacker has gained some level of permissions on a computer system, the system is said to be *compromised.* If the attacker gains root access, the compromise is considered a root-level compromise. With root-level privileges, the attacker can change any state within the system. The attacker *owns* the system. The attacker can modify the system so that the original trusted reporting processes no longer report accurate information. Some level of trust must be restored to the system before all reporting information can be relied upon, depending on how trust is broken.

Compromised System – If a system is compromised, then the following conditions are true.

1. An attacker has gained some level of privileges on the system.
2. The attacker can read or modify some portion of the state within the system.

Root-level Compromised System – One specific class of compromised systems are *root-level* compromises. If a system is compromised with *root-level* access, then the following conditions are true.

1. An attacker has gained unlimited access to the system.
2. Any state within the system can be read or modified by the attacker.
3. Trust can be broken in the system.

1.2 Definition of Rootkit

A *rootkit* may be considered as a form of a *Trojan Horse* as discussed in [1]. Once an attacker has compromised a system, he or she often use a rootkit as a tool to covertly retain access to that system. A rootkit can contain utilities to allow the attacker to retain access, hide processes and activities, and break trust in the local system reporting and querying functions.

We classify rootkits into *user-level* and *kernel-level* rootkits. A user-level rootkit will alter operating system tools at the user level (which usually involves adding or modifying system binaries such as /bin/login). A kernel-level rootkit will alter or insert kernel-space executing code (e.g. system calls).

1.3 Definition of Trust

Trust can be defined as the level to which a user believes a computer system executes as specified and does nothing else. If a compromise occurs on that computer system and the user discovers it, the level at which the user trusts the system is significantly lessened. The lowered level of trust is understandable because, for example, a rootkit may be installed on the compromised system such that file listing commands hide certain files and thus not execute as specified.

1.4 Overview and Organization

The rest of our paper is outlined as follows. Section 2 discusses the problem of kernel-level rootkits and previous work. Section 3 discusses current generation rootkits that modify the system call table. Section 4 describes our approach for recovering from current generation rootkits. Section 5 shows results of applying our techniques to real-world rootkits. In Section 6 we look at future generation rootkits in terms of their penetration techniques and kernel targets to trojan. Further, we discuss a concept to strengthen our algorithm described in Section 4. Finally, we discuss our conclusions and future work in Section 7.

2 Motivation

With the proliferation of exploits targeted to today's computer systems, an attacker has the ability to compromise a number of systems. Once an attacker has compromised a system, he or she will want to retain access to that system even if the original security hole is patched. In order to retain access to a compromised system, the attacker will often install a rootkit onto the target system. The rootkit will add a backdoor onto the target system that the attacker can use to reenter the system at a later time. We set up a Red Hat 6.2 system on the Georgia Tech honeynet [2], and within a matter of days an attacker had compromised the box and installed a kernel-level rootkit, *r.tgz*, on the system.

If the system administrator notices that an attacker has compromised the system, the administrator will immediately take pervasive actions to block the attacker from reentering the system. However, the attacker may have installed a rootkit to hide the attacker's activities, files, and backdoor entry point. To accomplish this goal, the rootkit will break trust in system reporting facilities (e.g. */bin/ls*, */usr/bin/top*, */sbin/lsmod*).

With a user-level rootkit, the system administrator can restore trust in the system by using known good utilities (e.g. */mnt/cdrom/ls*, */mnt/cdrom/top*, */mnt/cdrom/lsmod*). A kernel-level rootkit does not replace binaries but rather replaces running kernel code. We are not aware of any current methodology for restoring trust in a running system in which a kernel-level rootkit has been installed except for a complete reinstallation.

2.1 Related Work

Thimbleby, Anderson, and Cairns developed a mathematical framework to model Trojans and viruses [3]. They discuss a virus that could infect a system querying program in such a way that the querying program itself would be unable to detect that it was infected. This recursive infection leads to the idea behind kernel-level rootkits. When a kernel-level rootkit is installed, tools that check to see if a rootkit is installed are relying on an infected program, the kernel.

Recent research has been conducted developing a methodology for characterizing rootkits [1, 4, 5]. The methodology to characterize rootkits involves determining the Δ between a baseline system and a system compromised with a kernel-level rootkit. The Δ is used to characterize rootkits based on checksums, number of files replaced, number of files added, user level verses kernel level, penetration into the kernel, and so forth.

Government organizations have begun to investigate rootkits. The National Infrastructure Security Co-ordination Centre for the United Kingdom has recently published a report on Trojans and rootkits that discusses detection, remediation, and prevention of rootkits [6]. Their report describes Trojans as Remote Access Tools (RATs) that provide the attacker with a backdoor into the compromised system. The report discusses some of the functionality of RATs, which includes: monitoring system activities (i.e. watch users keystrokes and monitor users), monitor network traffic, use system resources, modify files, relay email (i.e. *spam*).

Other work has been conducted towards detecting and preventing kernel-level rootkits. Kim and Spafford show how a file system integrity checker, *tripwire*, can be used to monitor files for corruption, change, addition, and deletion [7]. In addition to other uses, tripwire can notify system administrators that system binaries have changed. Tripwire must establish a *baseline* for a known good file system. To establish a baseline, tripwire takes a hash (e.g. MD5, CRC, Snefru) of the files at a known good point. The baseline can be used for comparison at later points in time. A binary-level rootkit will replace system binaries, which will set off the "trip wire" and alert the administrator. However, a rootkit designer can counteract tripwire by breaking trust in the reporting tools upon which tripwire relies.

The open source and hacker communities have developed various tools to detect and prevent rootkits, which include: *chkrootkit* [8], *kern_check* [9], *CheckIDT* [10], and *Saint Michael* [11]. The chkrootkit tool is a script that checks systems for signs of rootkits. The chkrootkit script can detect many rootkits including both user-level rootkits and kernel-level rootkits, however some rootkits may evade detection. The kern_check tool is used to detect kernel-level rootkits. The kern_check tool compares the addresses of system calls as defined in the *System.map* file, generated at kernel compile time, to the current addresses of system calls. The CheckIDT tool is a user-level program that can read and restore the interrupt descriptor table, of which the *0x80th* entry points to the system call handler. Saint Michael is a kernel module that monitors the ktext (kernel code in memory) for modifications and attempts to recover from any modification to running kernel code. Saint Michael, however, must be installed prior to a system compromise and is not always successful.

2.2 Broader Scope

Intrusion prevention and intrusion detection have not slowed the growth of computer compromises to an acceptable rate. Research is drifting towards intrusion tolerance, and one element of intrusion tolerance is repair and recovery. In this paper, we begin to explore recovering from system compromises. There may be certain circumstances where the traditional format and reinstall is undesirable such as military systems, enterprise servers, or large clusters of machines.

We are also motivated by the need to perform forensics analysis on compromised systems. When a system is compromised, it is important to gather evidence that can be used for legal purposes[1]. It is important to understand the attack in order to prevent future attacks. Much of the evidence in a compromised system might only be resident in memory, so the evidence must be recovered before powering off the machine. In order to retrieve accurate information in the system, trust must be restored.

Although our work focuses on methods to recover operating system structures from system compromises, in many cases the most damaging part of a compromise is the data on the system that was compromised. This data can include passwords, credit cards numbers, keys, or other sensitive information. Our work does not solve the problem of data compromise, but we think it is another step in that direction. We envision self-healing systems that automatically detect system compromises and halt all attacker activity as quickly as possible in order to minimize the damage done.

3 Analysis of Current Generation Kernel-Level Rootkits

Kernel-level rootkits are rootkits that modify or insert code that runs in kernel mode. These types of rootkits may include user-level components but must have some functionality that resides at the kernel level. From our experience of examining rootkits, we characterize kernel-level rootkits based on two additional characteristics: *Penetration* into the kernel and *Modification* of the system call table.

3.1 Penetration

In terms of *Penetration*, we classify current generation kernel-level rootkits into two types based on their technique used for modifying kernel code. The subclassifications of kernel-level rootkits are:

– *Module* – Kernel-level rootkit that enters malicious code into the kernel by way of a loadable kernel module (LKM). The LKM, once inserted, will usually hide itself from system reporting facilities (i.e. */sbin/lsmod*). We consider these type of rootkits generation I kernel-level rootkits.

[1] From discussions with Office of Information Technology personnel at Georgia Tech.

- *User* – Kernel-level rootkit that patches running kernel code with malicious code from a user-space process. Usually, this type of rootkit will access kernel memory through the */dev/kmem* file. The Linux kernel provides access to kernel memory to user-space processes through the */dev/kmem* file. We consider these type of rootkits generation II kernel-level rootkits.

3.2 Modification

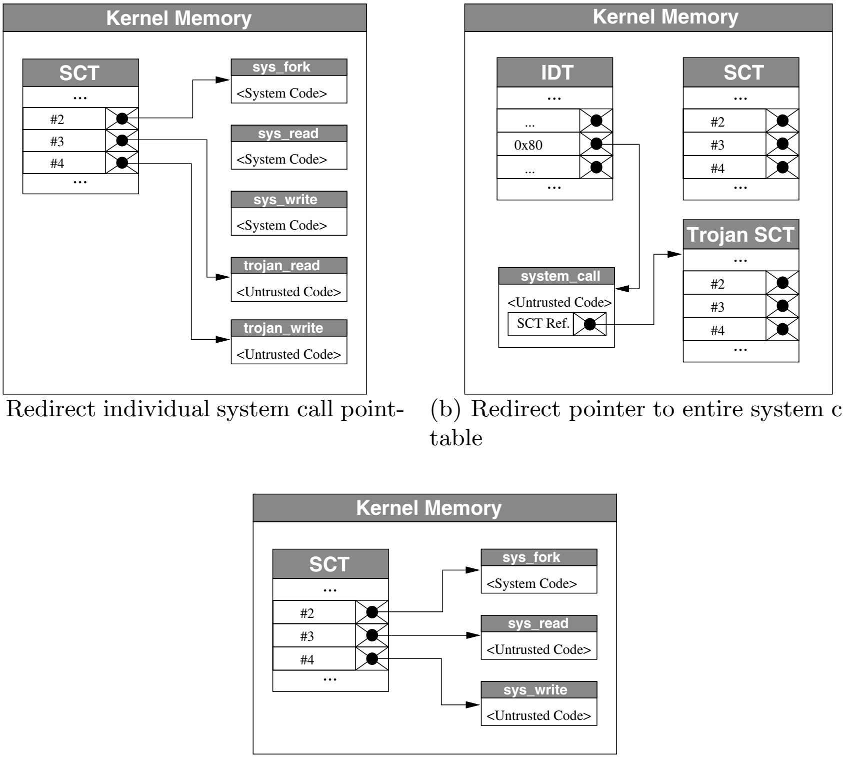
In addition, to classifying kernel-level rootkits in terms of penetration, we also classify rootkits in terms of how they modify the system call table, denoted *Modification*. Below are the subclassifications of *Modification*:

- *Entry Redirection* – Redirects individual system calls within the system call table. Modifies original system call table.
- *Entry Overwrite* – Overwrites individual system call code. Does not modify original system call table.
- *Table Redirection* – Redirects the entire system call table. Does not modify original system call table.

Figure 1(a) shows how a kernel-level rootkit can redirect individual system calls within the system call table (SCT). The picture represents kernel memory after a kernel-level rootkit with *Entry Redirection* has been installed on the system. In Figure 1(a), the *sys_fork* system call is unmodified. Notice, however, that system calls number three and number four point to Trojan system calls. The trusted *sys_read* and *sys_write* are still resident in memory, but there are no references to them. The system call table now points to *trojan_read* and *trojan_write*. Any binary executable that relies upon the system calls *sys_read* and *sys_write* will receive untrusted information from the trojaned system calls.

Figure 1(c) represents kernel memory after a rootkit with *Entry Overwrite* has been installed. Again, the *sys_fork* system call is unaltered. Notice, however, that the two system calls *sys_read* and *sys_write* have been overwritten. The actual code for the system calls has been overwritten as opposed to the corresponding table entry that references the system calls. The system call table itself is unaltered with this type of rootkit. We have not seen this type of rootkit but speculate that one could be constructed. The advantage of this type of rootkit is that a program such as *kern_check* [9] would not be able to detect the presence of the rootkit as *kern_check* only checks the system call table, but that is only a short-lived advantage as new tools are developed.

Figure 1(b) represents kernel memory after a rootkit with *Table Redirection* has been installed. The picture depicts kernel memory for the i386 architecture and the Linux kernel. Within the Linux kernel code exists a table called the Interrupt Descriptor Table (IDT) that points to kernel handlers for each interrupt. The *0x80th* vector is a software interrupt that points to the system call table. All user processes invoke a software interrupt *0x80* in order to call a system call [12]. When software interrupt *0x80* is invoked, the interrupt handler for interrupt *0x80* is called, which is the system call handler. The system call handler



(a) Redirect individual system call pointers

(b) Redirect pointer to entire system call table

(c) Overwrite individual system call code

Fig. 1. Current rootkit methods to trojan system call table

takes arguments from a user-space process and invokes the requested system call. The system call handler contains a reference to the system call table, which is used to lookup requested system calls. This reference can be changed in order to redirect the entire system call table.

As Figure 1(b) shows, the entire system call table has been redirected to a *Trojan* system call table. The trojan system call table usually contains many of the same entries as the original system call table but with a few key system calls replaced with trojan system calls. We have not shown how the Trojan system call table points to system calls in Figure 1(b) as it is similar to Figure 1(a).

3.3 Sample Rootkits

Table 1 shows a sample listing of kernel-level rootkits that we have classified in terms of their characteristics. We show three rootkits that penetrate kernel space through a *Module* and use *Entry Redirection* to trojan the system call table. The *heroin* rootkit is one of the earliest known kernel-level rootkits and is simply a kernel module that redirects a few key system calls. The *knark* and

Table 1. Sample classification of kernel-level rootkits

Rootkit	Penetration	Modification
heroin	Module	Entry Redirection
knark	Module	Entry Redirection
adore	Module	Entry Redirection
sucKIT	User	Table Redirection
zk	User	Table Redirection
r.tgz	User	Table Redirection

adore rootkits are other module based rootkits that redirect system call table entries.

The second group of rootkits listed are *sucKIT*, *zk*, and *r.tgz*. These rootkits all use table redirection and access kernel memory through the */dev/kmem* file. The *sucKIT* rootkit appears to be one of the pioneering rootkits for *Table Redirection*. The *r.tgz* rootkit was captured on a honeynet [13].

We have not seen any kernel-level rootkits that use *Table Redirection* and are also kernel modules. Similarly, we have not seen any kernel-level rootkits that penetrate the kernel from user space and also use *Entry Redirection*. We speculate that different combinations of rootkit characteristics are possible but see no motivation to build them. In addition, we also speculate that future kernel-level rootkits may redirect the software interrupt handler or the entire interrupt descriptor table, but have not seen any rootkits to date that use this technique. Finally, we have not seen any rootkits that use *Entry Overwrite* to trojan system calls.

4 Recovery by Bootstrapping Trust into the Running Kernel

Since kernel-level rootkits modify the system call table, the system call table must be repaired in order to recover from a kernel-level rootkit. Kernel-level rootkits overwrite portions of the kernel memory, so some information is lost. However, all of the kernel code can be found elsewhere. In Linux based systems, all that is needed is a copy of the kernel image, *vmlinux*. The kernel image contains the system call table and system calls.

Our approach to bootstrap trust into the running kernel is to, essentially, build a whitehat kernel-level rootkit. Our techniques is similar to sucKIT derivative rootkits. We bootstrap a trusted system call table into the running kernel and redirect the entire system call table to our trusted system call table. We strip out a trusted system call table from a known good kernel image, which is on known good media. Below, we discuss our algorithm, implementation, and tools.

4.1 Algorithm

The algorithm has five steps. We use some of the techniques of the sucKIT rootkit.

1. For each system call, allocate kernel memory for the system call and copy a trusted version of the system call into the allocated space. The offset for x86 *call* instructions within each system call must be adjusted when copying the system call to a new location in memory.
2. Allocate kernel memory for the system call table and set the entries of the system call table to point to the trusted system calls from Step 1.
3. Allocate kernel memory for the system call handler and copy a trusted system call handler into the memory. Note that the system call handler should reference the newly allocated trusted system call.
4. Query the *idtr* register to locate the interrupt descriptor table.
5. Set the *0x80th* entry in the interrupt descriptor table to the newly allocated system call handler.

Note that the trusted system calls will come from a trusted image of the kernel. In addition to the sucKIT rootkit's redirection of the entire system call table, we also redirect the *0x80th* entry of the interrupt descriptor table, the system call handler. The reason for this redirection is that we speculate future rootkits may redirect the system call handler and our goal is to rely on as little trust in the system as possible.

It is interesting to note that machine code in the Linux kernel cannot simply be copied from one memory location to another byte by byte. Kernel code compiled with the gcc compiler has many x86 *call* instructions. One form of the *call* instruction specifies a relative offset to the target. When moving kernel code around in memory, these *call* instructions must be modified by adjusting the relative offset. This depends entirely on where the *call* instruction and target are located in memory. Additionally, a known good hash of the code being copied will no longer be valid after modifying the offset value.

4.2 Implementation

We have developed our tools for the i386 architecture. The target system for development is Red Hat 8.0 with the default Linux kernel 2.4.18-4. The installation includes the normal development tools and the Linux kernel sources. Our implementation is a whitehat kernel-level rootkit that can be classified as a *User* rootkit that performs *Table Redirection*. Below we describe a few aspects of the implementation.

In order to strip the system calls out of a Linux kernel image, we use code from the gdb debugger. The gdb debugger has the ability to parse binaries and strip out functions, which in our case are system calls. Our implementation strips all of the system calls from the given kernel image, vmlinux-2.4.18-14, and feeds them to our whitehat kernel-level rootkit, *recover_kkit*.

Our code uses *Table Redirection* in order to bootstrap trusted code into the running kernel. We use sucKIT's technique to locate the address of the system call handler. Once we have the address of the system call handler, we can parse the system call handler code and locate the reference to the system call table. By replacing the reference to the system call table so that it points to a trusted

```
struct idtr idtr;
struct idt idt80;
ulong old80;
/* Pop IDTR register from CPU */
asm("sidt %0" : "=m" (idtr));
/* Read kernel memory through /dev/kmem */
rkm(fd, &idt80, sizeof(idt80), idtr.base +
    0x80 * sizeof(idt80));
/* Compute absolute offset of
 * system call handler for kmem */
old80 = idt80.off1 | (idt80.off2 << 16);
```

Fig. 2. Source code to find address of system call handler

```
#define  rr(n, x) ,n ((ulong) x)
#define __NR_oldolduname  59
#define  OURSYS __NR_oldolduname
#define syscall2(__type, __name, __t1, __t2) \
    __type __name(__t1 __a1, __t2 __a2)      \
{                                            \
   ulong __res;                              \
   __asm__ volatile                          \
   ("int $0x80"                              \
   : "=a" (__res)                            \
   : "0" (__NR_##__name)                     \
     rr("b", __a1)                           \
     rr("c", __a2));                         \
   return (__type) __res;                    \
}
#define  __NR_KMALLOC OURSYS
static inline syscall2(ulong, KMALLOC, ulong, ulong);
```

Fig. 3. Source Code - Kmalloc as a System Call

system call table, trust can be re-established. The code for locating the system call table can be seen in Figure 2. The key line is the assembly instruction

```
{asm("sidt %0" : "=m" (idtr));}
```

This assembly instruction copies the contents of the idtr register into the *idtr* variable. The absolute offset of the interrupt descriptor table can be calculated to locate the interrupt descriptor table. The *0x80th* entry of the interrupt descriptor table points to the system call handler.

Since our implementation is a *User* type implementation, a tricky part of the implementation becomes allocating kernel memory. We use the same technique that the sucKIT rootkit uses. Figure 3 shows the source code used to wrap *kmalloc()*, the kernel memory allocator, into a system call. In the Figure, KMALLOC is the virtual address of the *kmalloc()* function within kernel space. Our code first locates the current system call table by reading the reference to the current table from the system call handler. Then an unused system call, *sys_olduname*, is taken over and replaced with a system call that we will call *sys_kmalloc*. Now a user-space program can allocate kernel memory simply by issuing the system call *sys_kmalloc*.

Using the techniques described above, we have implemented a whitehat rootkit called *recover_kkit*. Our implementation follows the algorithm described above. Below we discuss our tools.

4.3 Tools

Including our whitehat rootkit, we have implemented a suite of tools that can be used to check for and recover from kernel-level rootkits. Our tools can be found on our website [14]. The *read_sctp* tool reads the address of the current system call table and can be used to compare the actual system call addresses to the ones found in the *System.map* file. Our approach differs from *kern_check's* method in that our program looks up the actual system call table as referenced in the running system call handler. Another tool we created is called *ktext*. The *ktext* tool can be used to capture portions of kernel memory in the running kernel. We have used the *ktext* tool to determine a Δ for kernel-level rootkits [5]. Other tools provide the ability to dump system call table entries to a file and write individual system call table entries to kernel memory. Finally, the *recover_kkit* tool can be considered a whitehat kernel-level rootkit that can be used to recover from blackhat kernel-level rootkits.

5 Results on Current Generation Rootkits

In order to test our *whitehat* kernel-level rootkit, we have selected three *blackhat* kernel-level rootkits to recover from. We have chosen to test *knark*, *sucKIT*, and *r.tgz*. These three rootkits represent kernel-level rootkits that penetrate the kernel from both user space and from a kernel module. Also, our tools are tested against both *Entry Redirection* and *Table Redirection* type rootkits. Finally, we also test our tool against *r.tgz* because it represents a rootkit that was captured in the wild on the Georgia Tech Honeynet. Recovering from the *r.tgz* demonstrates how our research can be applied to real-world scenarios. Figure 4 shows the results of our testing.

5.1 Recovering from Knark

In our first scenario, we have installed *knark* on a Red Hat 8.0 system with a Linux 2.4.18 kernel. The results can be seen in Figure 4(a). The first step is to install *knark*. Since knark is loaded as a kernel module, we insert knark with the *insmod* command. The kernel prints a message warning that *knark.o* does not have an agreeable license. The second step is to hide a binary, which we have placed in the */bin* directory, called *rootme*. The *rootme* binary is part of the knark rootkit and is used to execute binaries with root-level permissions from a regular user account. The *hidef* utility is part of the knark rootkit and is used to hide utilities. In the third step, we list files in the */bin* directory that begin

```
[root@h1 cd]# insmod ./knark.o
Warning: loading knark.o will taint the
kernel: no license See
http://www.tux.org/lkml/#export-
tainted
for information about tainted modules
Module knark loaded, with warnings
[root@h1 cd]# ./hidef /bin/rootme
hidef.c by Creed @ #hack.se 1999 <creed
@sekure.net> Port to 2.4 by Cyberwinds
#Irc.openprojects.net 2001
[root@h1 cd]# ./ls /bin/root*
ls: /bin/root*:
No such file or directory
[root@h1 cd]# ./recover_kkit
Trust has been Re-established!
[root@h1 cd]# ./ls /bin/root*
/bin/rootme
```

(a) Recovering from knark

```
[root@h2 cd]# ./sk
/dev/null RK_Init: idt=0xc037d000,
sct[]=0xc0302c30,
kmalloc()=0xc0134fa0, gfp=0x0
Z_Init: Allocating kernel-code
memory... Done,
12747 bytes, base=0xc8090000
BD_Init: Starting backdoor daemon...
Done, pid=1435
[root@h2 cd]# ./ls /sbin/init*
/sbin/init
/sbin/initlog
[root@h2 cd]# ./recover_kkit
Trust has been Re-established!
[root@h2 cd]# ./ls /sbin/init*
/sbin/init
/sbin/initlog
/sbin/initsk12
```

(b) Recovering from sucKIT

```
[root@h3 cd]# ./all
[===== INKIT version 1.3a, Aug 20 2002 <http://www.usg.org.uk> =====]
[====== (c)oded by Inkubus inkubus@hushmail.com> Anno Domini, 2002 ======]
RK_Init: idt=0xc027a000, sct[]=0xc0248928, kmalloc()=0xc0121b88, gfp=0x15 Z_Init:
Allocating kernel-code memory...Done,
13147 bytes, base=0xc9498000 BD_Init: Starting backdoor daemon...Done, pid=1213
[root@h3 cd]# ./ps -p 1213
PID TTY TIME CMD
[root@h3 www]# ./recover_kkit
Trust has been Re-established!
[root@h3 cd]# ./ps -p 1213
PID TTY TIME CMD
1213 ? 00:00:00 all
```

(c) Recovering from r.tgz

Fig. 4. Testing recover_kkit Tool on Three Kernel-Level Rootkits

with *root*. No files are shown indicating that our system cannot be trusted. The fourth step is to install our trusted system call table with our tool *recover_kkit*. We use a read-only cdrom to run our tools. Now notice that upon listing files again, the file *rootme* is seen. Trust has been re-established in the compromised host.

5.2 Recovering from sucKIT

In our second scenario, we have installed *sucKIT* on a Red Hat 8.0 system. The results can be seen in Figure 4(b). The steps are similar to that of knark. We install the rootkit, show that some files are hidden when running the *ls* utility, restore trust, and finally show that the hidden files appear. The *sucKIT* rootkit hides files that have a certain extension, in our case “sk12”. The initsk12 file is used in coordination with the init file to load sucKIT upon a reboot. Trust has been re-established in a system that has been compromised with a kernel-level rootkit that performs *Table Redirection*.

5.3 Recovering from r.tgz

In our third scenario, we have installed *r.tgz* on a Red Hat 6.2 system. The results can be seen in Figure 4(c). This rootkit is an example of a real-world scenario. In our scenario, an attacker has compromised the system and starts a Trojan process with the *all* utility. The *all* utility is part of the *r.tgz* rootkit. Initially, the process is hidden, as seen by the first *ps* execution. Then, we install our trusted system call table and issue the *ps* command again. You can see that this time the hidden process shows up. We have successfully re-established trust in a compromised host that was compromised in the wild.

6 Future Generation Rootkits and Recovery

6.1 Possible Penetration Techniques

We have discussed current generation rootkit kernel penetration techniques in Section 3.1. In this section, we discuss kernel penetration techniques that we have not seen in current rootkits while studying existing rootkits. Based on our experience, we speculate that future generation rootkits may use these techniques as more security features are added to kernels to prevent current generation rootkits (i.e. do not allow module loading or access via /dev/mem). Some of these techniques have been discussed in hacker communities; perhaps the techniques are already in use, but we have not seen any evidence to sustain such claims.

- *DMA* – These type of kernel-level rootkits could patch running kernel code with malicious code by programming an attached hardware device to use direct memory access (DMA) to modify kernel code. The concept was introduced in [15], but we have not seen any implementations.
- *Swapped-out Pages* – With root-level access, the attacker has raw access attached hard disks. Memory pages are swapped to the hard disk when memory becomes full. An attacker could use raw hard disk I/O to modify swapped out pages in order to penetrate the kernel. Normally the kernel code is never swapped to the disk, but an attacker could use indirect means to penetrate the kernel through swapped out pages.
- *Local Image* – The kernel image resides as a binary file on the file system. The attacker can modify the kernel image on disk and replace trusted code with trojaned code. The next time the system is rebooted, the trojaned kernel image will be loaded into memory, thus accomplishing the attacker's goal without modifying the running kernel.
- *Distributed Image* – The beginning of the chain of trust starts at the source code and binary distributors. An attacker could compromise a kernel image before it is ever installed on the system (i.e. replace code or binary files with trojans before the kernel is distributed). As Thompson points out, one must "trust the people who wrote the software," or in this case trust the people who distribute the kernel [16].

6.2 Kernel Targets for Kernel-Level Rootkits

The first kernel-level rootkits developed have focused on trojaning the system call table. The system call table is the gateway from user space to kernel space, and so is a natural target and easily trojaned. Tools are being developed to detect and counter these types of rootkits including our tools that allow recovery from a certain class of kernel-level rootkits. As such developments continue, the arms race is escalated. Attackers will continue to develop new means of trojaning the kernel. Below we outline such targets for kernel-level rootkits.

- *System Call Table and Interrupts* – Section 3 gives an extensive discussion of how the system call table is trojaned. Many widely examined rootkits use this means of trojan when targeting the kernel. The interrupt subsystem is a general target of the kernel as interrupts are often serviced on behalf of processes.
- *Redirecting Core Kernel Functionality* – Core kernel functionality is a target of kernel-level rootkits. Examples include the scheduler, process handler, authorization mechanisms, and the virtual file system mechanisms. The latest adore rootkit, adore-ng, targets the virtual file system [17].
- *Redirecting Extremity Functionality* – Extremity functionality includes subsystems of the kernel such as the network drivers, hard disk controllers, network stack, and so forth. For example, a rootkit may want to modify the network stack so that the kernel listens for incoming requests from the attacker, unbeknownst to the system administrator.
- *Modifying Kernel Data Structures* – Finally, the attacker may modify the kernel data structures in addition or instead of modifying the kernel code. For example, a kernel module can be hidden from the *lsmod* command by removing it from the linked list that contains currently loaded kernel modules. This specific technique is already in use today.

6.3 Using a Trusted Immutable Kernel Extension for Recovery

Our algorithm described in Section 4 works well for recovering from system call table modifications but relies on one assumption that must be addressed. The algorithm assumes that a core level of trust remains intact in the system that would allow our program to function as expected. As long as the rootkit installation is well understood and known to be in accordance with our assumption, the method is valid. However, we also address the case in which the full extent of the rootkit is unknown.

Our solution to this problem is a Trusted Immutable Kernel Extension (TIKE) as introduced in [18]. TIKE is an enabling extension that can be used to ensure a trusted path exists within the system even if a kernel-level rootkit is installed. One approach to building TIKE is through virtualization. The production guest system is isolated from the host operating system. The production system may be attacked, but we assume the host operating system is inaccessible from the guest operating system. Therefore, our recovery algorithm can be carried out on the host system, with some modifications, in order to incontestably re-establish trust in the compromised system.

Techniques similar to our recovery method for system call tables can be used for many classes of future generation kernel-level rootkits. Our approach is summarized as follows: For the given kernel function redirection, copy a known good function from a known good kernel image and redirect the running kernel function to the known good function. Furthermore, since the level of trust that is broken may be unknown, the recovery should take place through a mechanism such as TIKE. The technique must be applied to the entire chain of trust in order to be certain that trust has been restored. This technique does not cover all possibilities, but does work for a given class of compromises. For example, rootkits that modify kernel data structures are more difficult to recover from.

7 Conclusions and Future Work

We have studied how trust can be broken in a system, specifically when a kernel-level rootkit is installed. We have applied a methodology to characterize current generation kernel-level rootkits in order to determine how to recover from them. Kernel-level rootkits can be classified in terms of their *Penetration* method and in terms of their system call table *Modification* method. Modern kernel-level rootkits can *Penetrate* the kernel from user space and use *Table Redirection* in order to install a trojaned system call table.

After providing an understanding of kernel-level rootkits, we introduced tools that can be used to recover from kernel-level rootkits. Our tool strips a known good system call table from the provided kernel image and bootstraps the trusted system call table into the running kernel. We then looked at future generation rootkits, further strengthened our algorithm with TIKE, and introduced a methodology to recover from future generation rootkits.

We have begun to explore the notion of re-establishing trust in compromised systems. We have shown that trust can be restored to a system, even if a kernel-level rootkit has been installed. Continued work will include applying our algorithm to more real-world compromises on the Georgia Tech honeynet to help validate the approach. We will also extend our work to cover more than just the system call table towards the entire system in order to establish techniques for self-healing computer systems. Our current work has focused on the Linux operating system, but future work will look into how our methods can be applied to other widely used operating systems.

References

1. Levine, J., Culver, B., Owen, H.: A methodology for detecting new binary rootkit exploits. In: Proceedings IEEE SoutheastCon 2003, (Ocho Rios, Jamaica)
2. Georgia Tech honeynet research project. `http://users.ece.gatech.edu/~owen/Research/HoneyNet/HoneyNet_home.htm` (2004)
3. Thimbleby, H., Anderson, S., Cairns, P.: A framework for modelling trojans and computer virus infection. The Computer Journal **41** (1998) 445–458

4. Levine, J., Grizzard, J., Owen, H.: A methodology to detect and characterize kernel level rootkit exploits involving redirection of the system call table. In: Proceedings of Second IEEE International Information Assurance Workshop, IEEE (2004) 107–125
5. Levine, J.G., Grizzard, J.B., Owen, H.L.: A methodology to characterize kernel level rootkit exploits that overwrite the system call table. In: Proceedings of IEEE SoutheastCon, IEEE (2004) 25–31
6. Trojan horse programs and rootkits. Technical Report 08/03, National Infrastructure Security Co-Ordination Centre (2003)
7. Kim, G.H., Spafford, E.H.: The design and implementation of tripwire: A file system integrity checker. In: ACM Conference on Computer and Communications Security. (1994) 18–29
8. The chkrootkit website. `http://www.chkrootkit.org/` (2004)
9. kern_check.c. `http://la-samhna.de/library/kern_check.c` (2003)
10. kad (pseudo): Handling interrupt descriptor table for fun and profit, issue 59, article 4. `http://www.phrack.org` (2002)
11. WWJH.NET. `http://wwjh.net` (2003)
12. Bovet, D., Cesati, M.: Understanding the Linux Kernel. O'Reilly&Associates, Sebastopol, CA (2003)
13. Levine, J.G., Grizzard, J.B., Owen, H.L.: Application of a methodology to characterize rootkits retrieved from honeynets. In: Proceedings of 5th IEEE Information Assurance Workshop. (2004) 15–21
14. Re-establishing trust tools. `http://users.ece.gatech.edu/~owen/Research/trust_tools/trust_tools.htm` (2003)
15. sd (pseudo), devik (pseudo): Linux on-the-fly kernel patching without lkm, issue 58, article 7. `http://www.phrack.org` (2001)
16. Thompson, K.: Reflections on trusting trust. Commun. ACM **27** (1984) 761–763
17. Labs, S.: Subverting the kernel. `http://la-samhna.de/library/rootkits/basics.html` (2004)
18. Grizzard, J.B., Levine, J.G., Owen, H.L.: Toward a trusted immutable kernel extension (TIKE) for self-healing systems: a virtual machine approach. In: Proceedings of 5th IEEE Information Assurance Workshop. (2004) 444–445