COEN225 Lab 3: Controlling the Flow

Overview

The learning objective of this lab is for students to understand how a security vulnerability such as a buffer overflow can not only corrupt program data but also how a process's control flow data can be manipulated to execute code, ultimately providing a malicious attacker the ability to take over a running program. In addition, students will explore defensive measures implemented in operating system and modern compilers to mitigate these types of attacks.

Lab Submission

Perform the tasks below and submit your responses to questions noted in the task submission sections. Also include any source code you used to generate exploits or other tests.

The lab work and submission must be completed individually but working with other students is allowed. The Camino message board is the preferred method for asking questions so that the instructor and students can respond and allow all students to learn from it.

Lab Tasks

Initial Setup

We will use the same virtual machine setup from previous labs. Modern Linux distributions have implemented several security mechanisms to make buffer overflow attacks more difficult. In order to focus on the buffer overflow itself and avoid spending time building workarounds using other techniques, we will explicitly disable these mechanisms for the lab. Later, we will enable them one at a time to understand the effects.

Address Space Randomization. Many stack based buffer overflow exploits rely on a user controlled buffer to always be at a known location in memory. Modern Linux systems uses address space randomization (ASLR) to randomize the starting location of various memory segments including the heap and stack. Randomizing the starting locations of memory segments forces the attacker to guess addresses for the attack, or find the memory address by other methods.

To disable ASLR, use the following command (must be root):

```
$ su
Password: (enter root password)
# sysctl -w kernel.randomize_va_space=0
```

To enable ASLR, use the following command (must be root):

```
$ su
```

```
Password: (enter root password)
# sysctl -w kernel.randomize_va_space=2
```

Note: The ASLR setting is *temporary* and will be re-enabled if the system is rebooted. Remember to disable the setting again before starting.

Stack Protector. The GCC compiler implements a security mechanism called Stack Protector to detect buffer overflow attacks. Programs compiled using GCC with these protections enabled (default in recent versions) are difficult to exploit. Stack Protector can be enabled or disabled using command line flags at compile time.

To disable stack protector:

```
$ gcc -fno-stack-protector test.c
```

To enable stack protector:

```
$ gcc -fstack-protector test.c
```

Non-Executable (NX) Stack. Executables mark their memory sections with flags, one of which is the executable flag. In modern versions of GCC, the .text section (where the program code is) is allowed to be executable and the stack is set to be non-executable. As we will be injecting our code into the stack, we will need to allow the stack to be executable.

For executable stack:

```
$ gcc -z execstack test.c
```

For non-executable (protected) stack:

```
$ gcc -z noexecstack test.c
```

Task 1: Overwriting the return address

Compile lab3a.c. While we are learning how to develop the attack, disable all protections including ASLR, NX, and stack protector.

```
$ su
Password: (enter root password)
# sysctl -w kernel.randomize_va_space=0
# exit
$ gcc  -o lab3a -g3 -z execstack -fno-stack-protector lab3a.c
```

Run the program. Notice that entering a short input will print out "Returned to main" and cleanly exit. The objective is to cause `unused_function()` to run, even though it is never called in the program. This is possible by manipulating the instruction pointer register (eip) to point at the memory address where `unused_function()` is located. The eip is normally incremented as it executes instructions, but when it encounters a return instruction, eip will be loaded with the return address located in the stack. We can use the vulnerable `gets()` function to overwrite the return address with an address of our choosing.

First, we must find the address of `unused_function()`. Use the objdump tool's disassembly option to get the addresses of the code.

```
$ objdump -d lab3a
```

Now that we have the address of the function, we can now write exploit code to overwrite the return address with it.

Although you can provide input to the program using your keyboard, memory addresses have a wide range of byte values and is difficult to type using a keyboard, especially if there is no printable ASCII equivalent character for that value. Instead we will write a exploit generator to produce our exploit file. A sample is provided in gen_exploit_a.c or you may write your own.

Note: You do not have to compile this program with protections disabled because this is only a tool to help generate the exploit code you will use to attack the vulnerable program. We are not looking for vulnerabilities in this code.

The gen_exploit_a.c will take the contents of the data in the array buffer and write it out to a file. Note that the \x notation can be used to enter arbitrary byte vales in hex format, which solves the challenge of entering non-printable values. The array should end with a newline (\n) character as the gets function looks for the newline to signal the end of input.

Run the program to generate your exploit file, then use the xxd command to view the contents.

```
$ xxd exploit
```

Rather than typing your input to the lab3a program by hand, use the redirection operator to feed the exploit file as your input.

```
$ ./lab3a < exploit
```

Now let's figure out what it takes to corrupt the `func()` function's stack frame. Create an exploit file that is long enough to fill, but not overflow, the `gets()` buffer. Run the program and feed the exploit file again to ensure that it is working properly. Now slowly add 'A's to the input and re-run the program. Don't forget to re-compile and re-run the exploit generator every time you modify your code. When you see a segmentation fault, this tells you that the saved ebp is corrupted. Continue adding to the size of the input until you don't see the "Returned to main" message anymore. This is where the return address is corrupted. Make note of the size of the input required to corrupt the return address, as this will help you later.

You may notice that there can be some extra space between the end of the buffer in func() until you start overwriting the saved ebp. A compiler is not required to use the minimum amount of space possible.

Since the program now crashes and it doesn't return to main(), we have no insight into what is happening. So let's launch GDB and see what we can find out.

```
$ gdb ./lab3a
(gdb) run < ./exploit
```

GDB will catch the segmentation fault and print the address of the code that it was trying to execute. You may notice that it is fully or partially part of your exploit data. This is the piece that overwrote the return address in memory, which was then loaded into eip and executed. But because there wasn't valid code there, it crashes.

Now adjust your exploit code so that you know exactly which bytes overwrite the return address. Then replace it with the memory address of `unused_function()` that you got earlier and try again. **Tip**: Don't forget endianness!

If successful, you should see the output from unused function (and then likely crash as the stack is corrupted). Congratulations! You have successfully controlled a program's execution flow simply by putting in too much data.

Task 1 Submission

What was your final exploit code that caused the `unused_function()` to execute? You may include your exploit generator code, and/or print out the contents of your exploit file by using the `xxd` command.

Task 2: Code Injection

In the previous task, we were able to change a program's flow to execute code that should not have been executed. However, this limits the attacker's control only to code that is already in the program. Ideally an attacker would want to run any code of their choosing, which brings us to code injection.

What is Shellcode

First we need some shellcode. Shellcode is machine code created by an attacker that they want to execute on the target system. In many cases, the ideal result is to spawn a shell for the attacker to allow them to run any command they want (thus the name shellcode). The shellcode must be injected into memory, such as the stack, and then the instruction pointer (EIP) must be pointed at that location so that it is executed. So what would shellcode look like? Consider the following code:

```
#include <stdio.h>
int main(void) {
  char *cmd[2];
  cmd[0] = "/bin/sh";
  cmd[1] = NULL;
  execve(cmd[0], cmd, NULL);
}
```

The above C code will execute a shell, but it can't be used as-is. As we are injecting code into a process that is already running, so shellcode must be machine code for the target architecture (the output of the compiler and assembler). Additionally, because the address of library functions, such as execve() from the standard C library, are patched in by the linker before the program executes, our shellcode will not be able to find library functions on its own when injected. This means that we must write shellcode in a way that doesn't use libraries and can run independently. The byte string below is a slightly modified version of the above code assembled for the x86 architecture. The code uses system calls so it does not rely on any library code to work. As system calls are specific to an operating system, this shellcode will only work on Linux.

```
"\x31\xc0"       /* xorl %eax,%eax     */
"\x50"           /* pushl %eax         */
"\x68""//sh"     /* pushl $0x68732f2f  */
"\x68""/bin"     /* pushl $0x6e69622f  */
"\x89\xe3"       /* movl %esp,%ebx     */
"\x50"           /* pushl %eax         */
"\x53"           /* pushl %ebx         */
"\x89\xe1"       /* movl %esp,%ecx     */
"\x99"           /* cdq                */
"\xb0\x0b"       /* movb $0x0b,%al     */
"\xcd\x80"       /* int $0x80          */
```

In the context of this lab, spawning a shell on a system that you already have shell access to doesn't make a lot of sense. However, if you consider that many programs today are connected to networks, getting a remote shell is an attacker's dream.

As the purpose of this lab is to understand how to cause shellcode to run rather than building the shellcode itself, alternative shellcode has been prepared in gen_exploit_b.c that will run the command `cat /etc/passwd` which is easy to see when you are successful in getting it to run. This demonstrates that you are able to not only print out sensitive information from the system, but you are able to run any command if you want as long as you can write the proper shellcode for it.

The Vulnerable Program
Compile lab3b.c. Don't forget to disable protections.

```
$ su
Password: (enter root password)
# sysctl -w kernel.randomize_va_space=0
# exit
$ gcc  -o lab3b -g3 -z execstack -fno-stack-protector lab3b.c
```

Before running the program, look at the source code. This program reads a file from disk and copies it into a buffer. Create a test file called "exploit_b" and test how this works.

This program is similar to Task 1 in that a buffer overflow vulnerability is present in the vuln() function where the user's input is copied into a buffer that is too small. This allows you to corrupt vuln()'s stack frame and overwrite the return address pointing back at main(). This time, instead of overwriting this with a function already present in the code, you must put in the address of where your shellcode will be in virtual memory when the program runs and loads the file. This is trickier than before, as unlike the code in the .text section which is static, your shellcode will be somewhere in the stack but it will differ depending on the environment. Use GDB to figure out where the buffer is located (and your shellcode).

Note: Unlike lab3a.c, lab3b.c loads your exploit code from a file rather than standard input. You can run lab3b.c without a redirect.

This exploit generator for this lab is incomplete. It creates a file called "exploit_b" which can then be read by the lab3b program.

The exploit is missing 3 things:

- Where the shellcode should go in the exploit file (replace the ???)
- The memory address of your shellcode when it is loaded by the vulnerable program (use gdb to find it)
- Write the address of the shellcode to your exploit file in a way that it will overwrite the vuln() function frame's return address

The file is initially filled with the 0x90 byte, which is a NOP instruction on x86. The NOP is a do-nothing instruction where the CPU will simply use a cycle without doing anything and move onto the next instruction. This can make it easier to successfully jump to your shellcode because if your shellcode comes after a series of NOPs, then the return address that you use can be your shellcode exactly or any NOP instructions before it. If EIP points to any one of the NOP instructions, the CPU will simply do nothing until it eventually gets to your shellcode. This widens the target area and can make it easier to guess a memory location that works. This is known as a **NOP sled**.

At this point, it's up to you to build an exploit that causes the shellcode to run. You are encouraged to try things and research as needed to move forward. If you have questions, you are encouraged to make use of the discussion board. There is some additional guidance below if you need suggestions for where to start.

If successful, you should see the contents of the /etc/passwd file printed to the screen. Good luck!

Guidance

The first step is figuring out how to overwrite the return address of the vuln() function. You can use the technique described in Task 1 to determine which bytes in the exploit file will overwrite the return address. Make sure you can reliably overwrite the return address with a 4-byte value of your choosing before moving forward. If you can't overwrite the return address, it doesn't matter where or how good your shellcode is because it will never run!

Finding the memory address of where your shellcode will be in virtual memory is likely the trickiest part. You know that the exploit is being copied into local variables on the stack. Try setting a breakpoint at the vuln function and examine the stack to get a rough idea of where your shellcode will go. You can access the location of where a variable is located by using the reference operator (adding & before the variable name). See Lab 2 if you don't remember how this works.

It may take several attempts before you get the right memory address, but using the NOP sled technique properly should help. Use it to your advantage when you are positioning your shellcode within the exploit file. Guessing addresses is rather common when developing an exploit.

Don't forget endian-ness!

Task 2 Submission

Submit your completed gen_exploit_b.c and the contents of the exploit file: `xxd exploit_b`

Answer the following questions about your exploit:

Where did you put your shellcode within your exploit file? The beginning, middle, or end? Why?

If you modify lab3.c so that the size of small_buffer is 20 instead of 12, does your exploit work? It shouldn't, but why not? What change would you need to make to your exploit to make it work? Make the changes to your code and try it to verify your solution.

Task 3 Stack Protector

In the previous tasks, we disabled the Stack Protector mechanism in GCC when compiling the programs. In this task, we will consider repeating task 2 with this defense enabled. To do so, compile the program using the `-fstack-protector` option.

```
$ gcc -g3 -fstack-protector -z execstack -o lab3b lab3b.c
```

Run your attack again and observe the result.

Task 3 Submission

Answer the following questions: What happened when you ran your attack? Why was this output displayed? Explain how the stack protector prevents your attack from succeeding.

Task 4 Non-executable Stack

In the previous tasks, we instruct the compiler to flag the stack section as an executable region of memory, even though the stack does not need to be executable for the program to function. In this task, recompile the vulnerable program using the noexecstack option, and repeat the attack in Task 2. We will disable the stack protector again to see how the non-executable stack affects the attack.

```
$ gcc -g3 -fno-stack-protector -z noexecstack -o lab3b lab3b.c
```

Task 4 Submission

Answer the following questions: What happened when you ran your attack? Why was this output displayed? Explain how the non-executable stack prevents your attack from succeeding.

A non-executable stack only makes it impossible to run shellcode on the stack, but it does not prevent a buffer overflow attack in general because there are other ways to run malicious code after exploiting a buffer overflow vulnerability. The return-to-libc attack is one example that we will cover in a future class.

Task 5 Address Space Layout Randomization

For this task, we will enable the Address Space Layout Randomization feature. To observe only the effects of ASLR, re-compile the program with the other protections disabled.

```
$ gcc -fno-stack-protector -z execstack -o lab3b lab3b.c
```

Next, enable ASLR as root. Drop back to the normal user after doing so.

```
$ su
Password: (enter root password)
# sysctl -w kernel.randomize_va_space=2
# exit
```

Note: If you developed your exploit code to run in GDB, you will also need to enable ASLR within GDB as well. Programs executed from within GDB will have ASLR disabled by default to make it easier to debug programs. To enable ASLR in GDB, use the command below (don't be thrown off by the double-negative):

```
(gdb) set disable-randomization off
```

Now run the same attack developed in Task 2.

Task 5 Submission

Answer the following questions: What happened when you ran your attack? Why was this output displayed? Explain how ASLR prevents your attack from succeeding.

It is possible to pull off a successful attack even with ASLR disabled with a well crafted exploit and a bit of luck (and maybe many tries). How is this possible? How does the NOP sled technique help?

END OF LAB