ELEC 377 – Operating Systems
Lab 5 – Software Security

Lab Date: Nov 20/21, and Nov 27/28 2019 Due: Dec 2, 2019 at 0900

Objectives

• Compromise a server program using a standard buffer overflow

Introduction

The purpose of this lab is to examine a common security vulnerability, the stack overflow. One way of testing the security of applications is to try and break into them in a controlled environment. Many companies have "tiger" teams whose job it is to test the security of their own systems. To do so, many organizations have both the blue team (defensive) and the red team (which actively attacks the network trying to find security holes).

Ethics

The purpose of learning the details of attacks is twofold:

1. Understanding how attacks happen so that you will be better able to design and implement security code.
2. Use security attacks in a controlled environment to test the security of a particular program or application.

Occasionally when security is covered in undergraduate courses, some students not only get curious, but decide to try out some of what is discussed. This is reasonable, as long as it is done in a controlled manner. However, if you attempt it on a public network you will find yourself in a great deal of trouble. The era of breaking into a company's network and having them pay you to fix it are long gone. Be especially careful if you ever work with virus or worm code since it may accidentally escape from you. The authorities will take the view that if you are work with worms and viruses, you are responsible for the results. Be warned.

PreLab

In order to complete the lab in two weeks it is important that you read the documentation on the stack layout and the x86 architecture. Please read the documentation and answer the following questions. They are worth 2 marks for a total score of 12 marks for this lab. The questions are due at the beginning of this weeks lab(Nov 20/21). Hand in on paper with your name and student number.

1. Move the value 0xb5 into the c register.
2. Assume that there are three local variables a,b and c that have been allocated on the stack at offsets 0, -4 and -8 bytes respectively from the base pointer (EBP). Write the single instruction to load the lower 8 bits of the variable c into the a register.
3. Load the address of the variable b into the b register.
4. Store the value in register a into the address in register b.

Setup and Environment

The files you need for this lab are:

Makefile - The makefile to compile the programs. When Linux downloads these files, it sometimes removes the tabs. Each compile command must start with a tab

selfcomp.c - This is a file that can execute the exploit internally. This will be used to develop the exploit.

server - The vulnerable server. This is a binary executable file. There is no source, so you will have to find the vulnerability without looking at the code.

client.c A client you will use to compromise the server program.

There is a tutorial on the web site with much of the information you need to conduct the lab. Part of the lab is examining core dump files. But you must tell the slackware distribution that the kernel is allowed to create dump files. This requires that you execute the following command:

```
ulimit -c unlimited
```

Do not log in as root for this lab. It has no kernel modules to write or install. If you are using the csh shell for some reason, then you use the command

```
limit coredumpsize unlimited
```

These commands remove the limit on the size of the core dump file that may be created. The default limit is 0 (no dump files allowed). A dump file is created when a process encounters an error. It contains a dump of the state of memory of the process as well as the CPU registers. When you execute the command, core dumps are only enabled for the shell that you gave the command to. If you execute the exploit in a different window from the above, or if you log out and log back in, the core dump file will *not* be created.

For those of you that want to do this lab at home, there is no particular requirement to use the Virtual PC environment. However, there is a restriction on which versions of Linux may be used to perform the lab. The buffer overflow is the most common exploit, and some versions of Linux have added various types of protection. The most common is to move the stack in memory each time a process is started. Since the version of buffer overflow we are exploring here requires that you use an address on the stack, this will make it difficult to execute. The version of Slackware used in the lab does not contain this protection and thus is vulnerable to the buffer exploit. A simple way to test your version of Linux is with the following program:

```
#include <stdio.h>
int main()
{
       int a;
       printf("%x\n",&a);
}
```

You will have to have the 32 bit compatibility libraries installed, and compile in 32 bit mode. Compile and run it several times. If the address that is printed out changes, then you will have a difficult time completing this lab on your version of Linux. In this case, we have to turn off this protection for the approach in this lab to work. Executing the following two lines as root will fix this.

```
echo "kernel.randomize_va_space = 0" >> /etc/sysctl.conf
/sbin/sysctl -p /etc/sysctl.conf
```

You do not have to do this to the virtual machines in the lab. They already have the kernel stack offset disabled. After the lab, you will want to comment out the line in `/etc/sysctl.conf` in your machine at home to re-enable the protection.

Newer versions of Linux also include stack protection in the compiler. Using the flag `-fno-stackprotector` during compilation will disable the generation of the stack protection code.

The Lab

The objective of the lab is to copy the */etc/passwd* file remotely. Despite the name of the file, it does not actually contain the passwords. Those are stored in */etc/shadow*. However, only root can access the shadow file. However the passwd file still contains interesting and useful information to an intruder. It includes information about what account names are valid and where they are located. This information could be sold to spammers, or could be used to formulate another more sophisticated attack.

Start by using the `selfcomp.c` program to become aware of the tools. The first thing you should do is to compile and run `selfcomp.c`. It should do nothing. If you look at it, it copies bytes from a global variable in to a local buffer. Modify `selfcomp.c` by changing the two references to the variable *compromise* in the procedure *doTest* to refer to the variable *compromise1* and adding and removing 'x' characters from the variable *compromise1* until running the program causes the program to crash and dump core. Use the command

```
gdb selfcomp core
```

to examine the core file. Do not run any of the programs under debugger supervision. This changes the offset of the stack and will cause confusion. Use the debugger after to look at the core dump file. The symbolic debugging will be of little use since the stack has been compromised and the program counter points nowhere near the code of the program. However, the debugger command

```
info registers
```

will give you what you need. The two registers we are interested in are *esp* (the stack pointer) and *eip* (the program counter). You want to adjust the number of 'x' characters until the contents of the program counter is 'WXYZ' in hex (look it up using *man ascii*). You will now have two pieces of information: The first is the length of the compromise (the length of *compromise1*), the second is the address of the return address on the stack. When the function returns, it pops the return address into the program counter. The program immediately crashes. Therefore, the stack pointer points to the byte directly above the return address.

Write the compromise in nasm based on the tutorial on the web site. You assemble the x86 code using the command:

```
nasm -l file.lst -f bin file.nasm
```

where *file.nasm* is the name of your exploit file. The list file (*file.lst*) shows the listing of your shell code. The binary version is in the file with the same name as your file but with no extension. Use the command "`od -t x1 filename`" to see the assembled code. Ensure that there are no null bytes (0x00) or newlines (0x0A). Sometimes a newline may occur because of the offsets between data (12 bytes apart). Insert an extra dummy byte or noop instruction in the appropriate place to eliminate the troublesome value. Copy those bytes into the compromise array in *selfcomp.c*. You will have to pad the beginning of the array with enough no-op instructions (0x90) so that the new return address is aligned with the existing return address. You must also calculate the beginning address of your code based on the length of you code and the value given to you by the debugger and insert that into the array. More information is in the tutorial on the web site. You can then compile and execute the program. If it copies the contents of `/etc/passwd`, then your exploit is working.

Once you have the compromise working in the `selfcomp.c` file, repeat the exercise with the client and server program. Start the server program in one window. It takes a port number as an argument. This is a number between 1024 and 65536. I tend to use 10000 when testing. The server program accepts a single line of input from the network and echoes it back. In another window run the client program. This program sends a string to the server and waits for a response. The network code has already been written for you. As with the `selfcomp.c` program, add or remove 'x' characters to the variable *compromise* until the ***server*** crashes. Run the debugger on the core generated by the server to determine the value of the stack and the length of the compromise string. You can then copy your exploit code into client.c. Change

the *fprintf* call in the *PutLine* function to use your array of code. You will probably have to calculate the number of padding chars and the return address in the compromise to fit the server program. You will also have to add a newline and a null character just after the new return address. The reason is that the server uses the *gets* function call which reads up to a new line character. So if you do not send a newline character, the *gets* function will not return. The null character is needed so that *fprintf* knows when to stop sending characters. This is also the reason you cannot have any newlines or nulls in the middle of your exploit since then the entire exploit will not be sent or read.

One problem you may encounter has to do with the TCP/IP stack. Sometimes when a program crashes, the kernel does not release the port. The symptoms are that when you restart the server for the next test, the client will connect and nothing will happen. If this occurs, just kill the server (Ctrl-C) and restart with a different port number (e.g. 10001).

Testing

Your testing must show both selfcomp and client retrieving the /etc/passwd file. This way you will get partial marks if you get one, but not the other working.

Documentation

You should document as much as you can. This should include documenting how long is the string used to compromise the server and the location of the return address on the stack. You should describe how the attack works. You must also include the NASM source code for your compromise as well as the source code of the `selfcomp.c` and `client.c` programs.