Lab 8.2: Advanced Buffer Overflows
IN618 Security

April 7, 2016

Introduction

Last time we looked at a fairly simple example of a buffer overflow. From it, I hope you got a basic idea of how a buffer overflow works. You did not, however, see the full power of a successful overflow works. You also didn't see how to prepare a realistic overflow attack.

In this lab you will execute a "real" buffer overflow attack. It will not be easy. There are many steps and a lot of little details. It will take some time. It's also not a lab you can do a portion of and then come back and complete later - at least not on a lab machine. But if you stick with it you will do something that very few people have actually done. In fact, very few people even understand how to do it.

You can do this lab on your own machine. It shouldn't cause any problems. However, I recommend that you use a lab machine.

1 Setup

Find the `bufferlab` folder on the I drive and copy it onto the `C:` drive of your lab machine. This will be your primary working directory

Run the nasm-2.11.08 installer. You will need this later to prepare your exploit code.

Once the MinGW installer completes, compile and run `vulnerable.c`:

- Open a `cmd` window.
- `cd C:\bufferlab`
- `gcc -fno-stack-protector -o vulnerable vulnerable.c`
- Run the program by typing `vulnerable.exe aaaabbbbcccc...yyyyzzzz`.

The program should crash, showing that it's vulernable to a buffer overflow. Click "Close the program" in the error dialog.

2 Prepare the attack

We are going to have to analyse the vulernable program using a special debugger, OLLYDBG. Extract the `odbg110.zip` file and run the OLLYDBG debugger that is unpacked. (Click "yes" to the dll message.)

- Go to File →open.
- Choose vulnerable.exe.

- In the Arguments box, enter aaaabbbbcccc... yyyyzzzz, then click "open".
- Push F9 once or twice to advance the debugger until the top left window goes blank.

Look in the "Registers" window for the value of the EIP register. In mine I found 71707070 - hex for "qppp". Now we know what part of our string is overwriting the return address. See figure1.png for an example.

If we count the characters in the argument string, we see that we write 65 of them before we overflow the buffer. This means that we have 61 bytes in which we can enter our malicious code. After that, we will use the next four bytes to change the return address to the base address of our malicious code.

Now, in your `cmd` window, run the `arwin` utility to find the address of two Windows functions.

- Run `arwin.exe kernel32.dll WinExec`
  I found 0x75462ff1.
- Run `arwin.exe kernel32.dll ExitProcess`
  I found 0x753e79d8.

The addresses you find will be different. Note the addresses. Our malicious code is written in assembly language in the file exploit.asm. Edit that file, placing the addresses you just found in the appropriate places. You should be able to tell where from the comments.

Now we will compile the exit code and then partially disassemble it to get the actual bytes of the executable code. The steps to do this in your cmd window are:

- `cd C:\Program Files (x86)\nasm`
- `nasm.exe -f elf C:\bufferlab\exploit.asm`
- `cd C:\bufferlab`
- `ld -o exploit.bin exploit.o`
- `objdump -d exploit.bin`

From the output we get the bytes of our malicious code that we will write into the buffer. See figure2.png for an example. Before we begin the attack, we need to find the address of the buffer in memory. Run `vulnerable.exe` again in OLLYDBG to find the address. Click F9 about twice until you can find the hex digits of our long input sting in memory and not the address. See figure3.png for an example. In this example, the buffer is at 0028fef0. There's actually one byte - the first 'a' - in the word above. When we write these bytes in our buffer overflow we will reverse the byte order and write the address as f0 fe 28, omitting the zeroes.

Now we have enough information to prepare our attack. We know:

- The number of bytes we need to write to overflow the buffer.
- The bytes of malicious code that we will place in the buffer.
- The value of the return address we will write into the bufer.

We will write a short C program to launch our attack. That program is already in the file `attack.c`, but you will need to adjust some values to match the information you found. In particular, you'll need to change the `ret` address to the one that you found, and you will need to change the `code` string to match your exploit code.

In `attack.c` notice that the exploit code has been padded with a number of
`90` bytes. This is because the exploit code is only 42 bytes long, and we need to write more bytes to overflow the buffer.
`90` is a no-op instruction. You may need to do a little trial and error to get the right number of bytes, but I expect you will need 19 of these no-op codes.

Compile and run this code to execute your attack.`gcc -o attack attack.c` is the command to compile it, then just enter the command `attack` to execute it.