Lab 8: Advanced Exploitation Techniques

CS 460 Spring 2015: Security Lab

Due 11:59 PM Thursday 03/19/15

Overview:

This lab will introduce you to techniques involved in exploiting binaries running on systems with modern protections. You will want to be comfortable with the concept of ASLR and DEP, as well as the premise of ROP chains. We discussed them in lecture, the wikipedia page for each topic is a good place to read if you don't feel confident in your research. This binary was taken from a capture the flag event (CTF) and was a 200 point challenge. (Out of a max of 500)

The goal of the binary is to to print out a flag on the server. Course Staff has prepared a similar setup as last lab, where you will practice your exploit on a local machine or VM, and then you will exploit the grading server to get a flag. Again, if you cannot figure out the exploit, put the work you did in answer.txt so that graders can try to give you partial credit. To run the binary, Course Staff recommends using socat, which just makes it so that you dont have to execute the binary everytime you crash it.

```
socat TCP-LISTEN:PORT,reuseaddr,fork EXEC:./exploitlab
```

When you want to write to socat, you should use your favorite programming language to open a socket to the port to send and recieve data, although you can also try to use netcat or telnet.

Hints:

There's a shorter list of things you have to do at the bottom.

1. If you run this binary and feed it a string, youll notice it just spits out WIN, and then exits. That's all the functionality that this binary has. It's up to you to figure out the exploit, although given that it's taking in a string as input, it shouldnt be too hard to find. Afterwards, try to overwrite the return address, and crash the program. Things to try:

    - Run `objdump -D` on it to generate a disassembly of the source code, and see if you can read through the disassembly to see where a buffer overflow might occur.
    - Put in a lot of "A"s and see if you can crash it.
    - Run Capstone on it for another dissassembly, or run Triad decompiler on it to try to generate C source code for the original program.

2. If the terms, GOT and PLT mean nothing to you, look over the slides from the "Advanced Exploitation" lecture and read this.

    - If you didnt notice when looking through the binary before, there are hardcoded addresses. What does that mean about the protections on this binary? What addresses could you overwrite to gain access to the libc read and write functions? Remember: parameters are pulled off of the stack, so you can store your arguments you want to send to either function!

3. Now you can read and write arbitrarily, but that's not winning yet. You want to execute shellcode! If you try just throwing shellcode on the stack, as in the previous lab, you'll notice it doesnt work. This is because the binary is compiled with DEP. If you want proof, you can use a tool called "readelf" to verify this. `readelf -l ./exploitlab`, will show `GNU_STACK` as `RW` only, no execute bit.

    So there are 2 possible solutions for what you can do. There's the classic return to libc

attack, or you can map a section of the stack as executable using mmap in libc and store your shellcode in that section.(Remember you have access to read and write!) Either works.

Decide which you want to do, and you'll now need to hunt for the offset. ASLR is enabled, so libc is not hardcoded in memory. But we do have access to a couple handy tools that will give us an information leak.

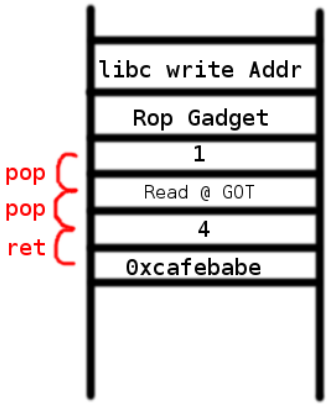
```
write(1, -Address of Read in GOT-, 4)
```

Use this to find the address of read in libc!

4. Now locate read in libc and find the offset of either mmap or system from it. (Math!) This can be done with a disassembler on your libc file. (For example, you can run `objdump -D | less` on the libc attached in your svn and then search for `__read` and `__libc_system`).

5. So now you can find the address of everything needed for your exploit, this is the tricky part of the lab. Youre going to need to find some gadget that can clear the stack for you. What I mean by this, is a sequence of POPs followed by a RET. This way, you can clear your parameters that you sent to the write function off of the stack, and ret will return to the next address in your chain of functions.

   For example, your stack layout should look like follows:



This way, the return will actually go to cafebabe like you planned it to. You will want to use a dissasembler (like objdump), or ROP finding tool (like ROPgadget) to find such a chain of instructions.

6. All your pieces are now able to be assembled. Now you can craft your exploit which uses the info leak from step 3, and the gadget from the previous step. Ill leave this part open to think about, but you need to figure out what sequence to write values to the stack. (PROTIP: The stack layout in the last step is the first step.)

Short Version:

1. Locate Vulnerability, and overwrite the return address.
2. Figure out how you would call Read and Write.
3. Find the address of Read in libc
4. Find the offset of either mmap or system.
5. Locate a POP POP POP RET gadget.
6. Write your Rop chain.

Extra Credit:

In previous years, there was a separate part where you would write a metasploit module to exploit the binary for this lab. Course Staff does not want to assign too much homework in one lab, so this part has been made optional. Course Staff **strongly** recommends to anyone interested in doing security as a profession to do the extra credit for this lab, as writing this metasploit module will give you experience with metasploit internals, and force you to work with an industry standard tool. Also it will give you something to talk about in interviews.

Hand in:

1. Turn in your flag via svn as usual.

2. If you do the extra credit, turn in your metasploit module in extra_credit.rb - metasploit modules will be graded manually.