

Computer Science 530 - Lab Assignment #4 -- Fall 2021

Due: Friday October 22, 2021, 4:30 p.m.

Overview

Infrastructure for Lab

There are three different Virtual Machines that you will use for the two parts of this lab. You will use the linux or windows scripts (batch files) to configure the virtual machines, therefore you should run these VM's within VirtualBox.

Location of files








The ova file for the three appliances are available in the CSci530 google drive in the folder for Lab 4 [here](#).

Please note that you may need to login to google drive with your USC account in order to access these files.

At least two of these files are the same as you used in lab 2 and 3, but I have included them again in the Lab 4 folder for ease of access. Let me also add a note of caution regarding the VM from lab 2: Because we changed the date on that Virtual machine, the system might incorrectly believe that the disk might be corrupt and try to run fsck to repair the disk. This may be fine, but if it does not work for you, just delete the VM and reinstall from the ova file.

The files in the google drive for lab 4 are:

My Drive > 2021 Lab 4 ▾

Name	Owner	Last mo
 stackoverflow.windows	me	8:37 PM
 heartbleed.windows	me	8:37 PM
 stackoverflow.linapp	me	8:40 PM
 heartbleed.linapp	me	8:40 PM
 Snort-on-Centos.ova	me	8:24 PM
 f19-heartbleeder.ova	me	8:45 PM
 kali-linux1.0.7.ova	me	8:46 PM

You will note that there is a directory with scripts (or BATCH files) for each of the parts of this lab, stackoverflow and heartbleed. For each, there is a directory for windows machines, and another for Linux and apple systems. Download the scripts from the two directories that are relevant to your machine. The scripts in these directories are used to clone the virtual machines (populate), start them (poweron), configure the network between them (construct network, this script is only present for the heartbleed part of the lab), power them off, and get rid of them when you are done with the lab.

You will run these scripts at the appropriate time for each of the experiments below.

Some notes on this instance of fedora Linux

We have already loaded most of the programs you will need for this lab into the virtual appliance. When you start the virtual machine you will be asked to login. For this lab you will be logging in to the root account and the Passwords for the account is "c\$I@bLinuX". The third character is the letter "l" as in lab.

Setting up this experiment

Scripts are used. A set of them is provided for each experiment. Please see the "Virtual Machine Usage" section of [this document](#). A summary of their use and intended execution order follows.

"vmconfigure-populate" is first, creates machines
 "vmconfigure-construct-network" is next, if it exists
 "vmconfigure-guestOS-internal-settings" is next, it makes the internal settings after powering the machines on;
 so if this one exists "vmconfigure-poweron" is not needed
 "vmconfigure-poweron" is next, if there is no "vmconfigure-guestOS-internal-settings"

Now you can use your VM(s). When you are finished:

"vmconfigure-poweroff"
 "vmconfigure-destroy" if you want to delete the machines, but not if you plan to come back to them later

If you do come back to them later,

"vmconfigure-guestOS-internal-settings" should be repeated, if present
 "vmconfigure-powern" if there is not "vmconfigure-guestOS-internal-settings" present

Application Security

This lab is divided in two parts:

1 - [stack overflow mechanics](#) - we will not go so far as to overflow the stack with any particular attack data or code, but will examine its structure and operation in detail to grasp what overflowing it means.

2 - [heartbleed exploit](#) demonstration

These are chosen from the endless myriad of "soft spots" in software. There are too many different kinds of vulnerabilities to mention. And that doesn't even address the ones nobody knows about. These two, chosen randomly because they are feasible, representative, fun, and might fit in an hour and a half, should give you some idea of the variety of ways software can be weak.

Part one: stack overflow mechanics

As advance preparation for this exercise, read though page 8 of Hackin9 magazine article [Overflowing the stack on Linux x86](#) by Piotr Sobolewski. This exercise derives from it.

Use the "stackoverflowVM" virtual machine. (Expect the exercise not to work in other environments; don't try to use them.) You will trace the execution of some small programs in gdb, the GNU debugger.

In order to get some gdb experience, play with vars.c. Examine its source code. Compile it with debugging data in the output, then bring up the output binary "vars" in the debugger gdb:

```
gcc vars.c -o vars -ggdb
gdb vars
```

Press ctrl-L to clear the screen. Run these commands in succession within the debugger:

```
list
break 5
run
print $esp
print $ebp
```

This runs the program up to (but not including) the first assignment, in line 5. Then it examines the values in the stack pointer and base pointer (these are registers, not memory locations). You should note that the base pointer exceeds the stack pointer by 0x28, or 40. The 40 bytes between the 2 addresses are "the stack," which we would like to examine. The command for you to execute is:

```
x/10 $esp
```

meaning, examine the 10 4-byte words starting with the address given by the stack pointer (going through the one at the address just before the base pointer). You'll see 40 not too meaningful bytes. Now fire the next line, "a=1;"

```
next
```

And re-display the stack:

```
x/10 $esp
```

The change in the stack is interesting. Note that the final 4-byte word now holds 0x00000001 while before it was something else. If you now run the next 2 assignments you will see the stack evolve to contain them:

```
next
x/10 $esp
next
x/10 $esp
```

(You can use command history in gdb just like the shell, where the up-arrow key recalls recent commands to the command line without your having to re-type them.) Issue "next" a couple more times to bring the program to completion.

vars.c shows one of the things the stack contains-- *local variables* for a function. rvals.c shows what happens when there is more than one function. The stack is also utilized to hold the *return addresses* in the calling function to which each called function is supposed to return control when it finishes. Since each function has its own stuff-- variables, return values-- the stack is divided internally into "sub-stacks." There's a separate, private little sub-stack for every function. They are called frames. So... another thing that goes in the frame for each function is the *previous frame pointer*. It points back within the stack to the calling function's frame. (This isn't the same as the return address. Both the return address and the previous frame pointer belong to the calling function, but the return address points into its code area of memory while the frame pointer points into the stack.) rvals.c shows the appearance of these new items in the stack in addition to the variables we've already seen there. The use of rvals.c in the debugger to show this is documented in the [slides for this lab](#). Review their depiction of what happens when rvals.c runs.

Repeat the rvals run shown in the slides' screenshots

"Do" the screenshots by mimicking them on your machine to "bring it alive." rvals.c called a function but passed it nothing. If at any point needed information has scrolled off the screen, shift-pgup will scroll the display backward a screenful at a time for you.

The next thing that interests us is the stack manifestation of the parameters passed into receiving arguments during a function call. That's what stack_2.c demonstrates.

Repeat the stack_2.c run shown in the slides' screenshots

by mimicking it on your machine.

Now let's shift our attention to a program that passes parameters, `stack_1.c`.

```
gcc stack_1.c -o stack_1 -g -gdb
gdb stack_1
```

In the debugger

```
list
break 8
break 3
break 4
run AAAAAAAAAA
x/6 $esp
next
x/24 $esp
next
x/24 $esp
```

The breaks are 1) in main just before the function call, 2) in the function just before filling the buffer (which, as a variable, is in the stack) with what the user typed, and 3) just after filling it. When you do the first "x/24 \$esp" you'll see a block of text. Find embedded in it the smaller block of text you got before when you did "x/6 \$esp". The new stuff is the "stack growth" and is the current function's stack frame laid onto the main function's stack frame. The stack as a whole has the two frames now. After you do the second "x/24 \$esp" compare the resulting block of text with the one before. It should show the "AAAAAAAAAA" has inserted itself. The A's appear as their hex code value 41. Find the ten 41's.

Now locate the return address in the stack. The function is depending on it for later use. When the function terminates, the return address will guide the program back into the main function from whence it came. The return address is important. To find the return address:

```
print $ebp
```

add 4 to the number that results and treat the sum as an address indexing into a place in the stack. At that place, the number you see is the return address. Where are your ten A's? How far away from the return address are they? If you'd typed 11 A's on the command line, your A's would be one byte closer to the return address. What if you typed 12 or 13? What would happen to the return address if you typed 100 A's?

Part two: heartbleed exploit demonstration

The OpenSSL project issued a [heartbleed vulnerability announcement](#) on April 7, 2014. Reaction was immediate. Within hours patches to fix it were applied, attacks upon it spiked, and detailed explanations were published.

This exercise uses two machines, kali-linux1.0.7 and f19-heartbleeder. The former plays the attacker role and the latter the victim role. f19-heartbleeder contains version 1.0.1e of OpenSSL, which has the heartbleed vulnerability. It also has apache, plus a web page containing a web form in which user-defined wording can be entered (in hopes of being seen later in the victim's RAM). kali linux contains code to exploit heartbleed and thus return content out of the victim's RAM. This code is part of the Metasploit framework, which in turn is an installed part of kali linux.

Log into both machines as root. In heartbleeder, set a hostname and take note of the name of the interface and its IP:

```
hostnamectl set-hostname heartbleeder
ifconfig -a
```

In hearteater similarly, in a terminal window:

```
hostname hearteater
date 090112002015 [ sets the date back to 2015, required for this exercise ]
ifconfig -a
```

The hostname setting is a convenience, so that the prompt in your terminal window will keep you mindful which machine you are on. But the prompt only picks up the newly set hostnames when you get a new terminal window. So at this point you might wish to "exit" your current one and launch a new one (by logging in afresh on heartbleeder or using the terminal window icon on hearteater).

You have no further business on heartbleeder. It is just the passive victim; there is nothing you need to do. It is already running apache, which utilizes OpenSSL. If you are curious, on heartbleeder:

```
openssl version -a
netstat -pant
```

The openssl command reveals openssl's version, listed [here](#) as a vulnerable one. The netstat command shows httpd (apache) listening on ports 80 and 443. In particular, on port 443 apache uses the https protocol which relies on ssl/tls as implemented by our openssl 1.0.1e. So talking to apache on 443 is how to interact with the vulnerable code and exploit it.

Switch to hearteater. Let's verify heartbleeder's open ports by nmap's empirical test, from outside heartbleeder, the same info we just got from netstat inside it. On hearteater:

```
nmap -sT <heartbleeder's IP>
```

This is nmap's classic tcp port scan. It shows protocol http on port 80 and https on port 443 are in play. Do it again, with the -sV option to probe the identity and version of the software employing those port numbers:

```
nmap -sV <heartbleeder's IP>
```

Let's make an empirical determination whether our heartbleeder is vulnerable. nmap has a scripting engine, and **there's an nmap script to detect the heartbleed vulnerability**.

```
nmap -sV --script=ssl-heartbleed <heartbleeder's IP>
```

This may take a bit of time. It reports that the victim is vulnerable. Now invoke the Metasploit console:

`msfconsole`

This may take some time, then eventually yields a prompt. Metasploit also has a scripting engine, and **there's a metasploit script to exploit the heartbleed vulnerability** (i.e., to obtain some randomly located chunk of victim RAM). Run that script:

`use auxiliary/scanner/ssl/openssl_heartbleed`

It gives another prompt. It's interactive.

`info`

We are told, "This module implements the OpenSSL Heartbleed attack. The problem exists in the handling of heartbeat requests, where a fake length can be used to leak memory data in the response." Point it at the victim and run it:

`set RHOSTS < heartbleeder's IP >`

`set RPORT 443`

`exploit`

It reports detection of "Heartbeat response with leak." Is this its cryptic way of telling us it got hold of some client RAM? We want to see it. Ask it to show us:

`set verbose true`

then again:

`exploit`

This time it shows you a screenful of stuff, spattered victim memory. It's encoded in a pure-ASCII form so is legible, and semi-intelligible. I noticed what looked like http, and apparent certificate header fields. If you run it again you'll get different stuff (do it, several times). The data obtained from within victim memory each time is randomly chosen. You're fishing, and you can't choose what you'll catch.

Maybe there's a way to put something in memory that would be recognizable if you captured it. Maybe you'd like to control it. While that's asking for more than you deserve, it just might be possible.

To try that bring up a browser, Kali linux has one, Iceweasel, under the Applications/Internet submenu. Point it at your victim:

`https://< heartbleeder's IP >`

Tell the browser to "Add exception" for this untrusted connection. You get a silly page that lets you enter something in a text field, then prints a second page containing that "something" (similar to entry fields for passwords and credit card numbers on the web). Go back and forth between pages, giving authors "Mark Twain", "Jack Frost", "Mickey Mouse" and such. Do it a few of times, hoping to place known-quantity content into the server's RAM. Then run the exploit. If you're lucky you may see those names within the data it returns. The data's whatever is floating around in memory, so it could also have been a password or a key in settings where those are involved. You have to be lucky, you can't control it. But the vulnerability is that you *can* be lucky, and can try as many times as you want.

Understand what is happening. There was nothing insecure (i.e., unconfidential, legible, or cleartext) about the ssl conversation. ssl fully and successfully did its usual job (watching with Wireshark will convince you). It protected the transmittal between attacker and victim. But remember that the ultimate fate we want for encrypted material is its decryption, in the good hands of the destination's intended recipient. Here, the destination is the victim's apache process with its calls to openssl through its mod_ssl module. After he gets the decrypted incoming conversation, what does he do with it? He parks it somewhere in memory for his use thereafter. Heartbleed is a post-conversation attack. It does not attack the conversation. It breaks into the memory where the conversation's content was afterward placed. What does the attacker care which copy he gets, or from where, as long as he gets one?

After you have performed the above lab components, answer the following questions.

1. In `stack_1.c` the separation in the stack between the beginning of the argument `buf` and the beginning of the stored return address is 28 bytes. See the screenshot in the slide entitled "Stack separation between argument and return address," also the article's page 7. In the screenshot, the argument starts at `0xbfffd530`. And the return address (always at `ebp` plus 4 bytes, and `ebp` holds `0xbfffd548`) is at `0xbfffd54c`. The separation between their starting points is therefore

`0xbfffd54c - 0xbfffd530 = 0x0000001c`

or twenty-eight bytes.

Change the code in `stack_1.c` to provide a buffer length of 2 instead of 10. That is, change line 2 from `"char buf[10];"` to `"char buf[2];"`. If you are not comfortable with the vi editor, this stream editor (sed) command can be used:

`sed -i 's/10/2/' stack_1.c`

Recompile and investigate (break/interrupt just after the stack buffer has been given data). Determine where exactly the argument starts by executing `"print &buf"`. Determine where the return address lies by executing `"print $ebp"` and adding 4. What is the separation now?

2. An easier, more casual way to observe the buffer overflow is to run `stack_1` from the command line instead of the debugger, supplying the variable's content as a command-line argument (note the program is written with formal parameters to receive it). While you can get away with passing a few more bytes than the buffer's designed for, without error, at a certain point and beyond you'll experience "Segmentation fault" error messages. Segmentation being a memory management technique, and you having screwed up memory pointers, there's no surprise in that terminology. Recompile `stack_1.c` with its original 10-byte `buf` instead of 2. Run `stack_1.c` and cheat by a byte, by passing it 11 characters (using "12345678901..." lets you visually keep track). Pass it increasing numbers of characters until you get the "Segmentation fault". Note the lowest/first value at which you reach that problem.

a. What is that value?

b. in broad generality, why is more than 10 OK up to a point?

c extra credit: I expected the lowest encounter with the problem at 1 byte more than it actually was. Why was I off by 1 byte and what's the reason for the problem appearing when it does? What is getting clobbered at that particular point?

3. There are both similarities and differences between the stack buffer and tls heartbeat flaws you exploited. Name one of each. That is, identify something they have in common that makes them the same, plus something else that distinguishes them as different. Any intelligent (and accurate) similarity and difference will be fine.

4. How would an attacker who wished to optimize heartbleed's value to him do it? Unlike transferring an entire database or shadow file that can net useful sensitive data from a system in great volume, this exploit is so random and modest that it can't really be controlled to advantage. Running it gets a mere few thousand bytes and

most is uninteresting garbage. What would you do to refine it and extract some metal from this raw ore?