New York University
Courant Institute of Mathematical Sciences
CSCI-GA 3033: Practical Computer Security

Memory War Lab

Due: Mar 17 2021, 11:59 PM (EST)

1 Lab Description

The learning objective of this lab is for students to gain the first-hand experience on an interesting variant of buffer-overflow attack; this attack can bypass an existing protection scheme currently implemented in major Linux operating systems. A common way to exploit a buffer-overflow vulnerability is to overflow the buffer with a malicious shellcode, and then cause the vulnerable program to jump to the shellcode that is stored in the stack. To prevent these types of attacks, some operating systems allow system administrators to make stacks non-executable; therefore, jumping to the shellcode will cause the program to fail. Unfortunately, the above protection scheme is not fool-proof; there exists a variant of buffer-overflow attack called the return-to-libc attack, which does not need an executable stack; it does not even use shell code. Instead, it causes the vulnerable program to jump to some existing code, such as the system() function in the libc library, which is already loaded into the memory. In this lab, students are given a program with a buffer-overflow vulnerability; their task is to develop a return-to-libc attack to exploit the vulnerability and finally to gain the root privilege. In addition to the attacks, students will be guided to walk through several protection schemes that have been implemented in Ubuntu to counter against the buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why.

You can read more about return-to-libc attacks that are better explained in these papers by c0ntex and Nergal.

Question 1: Please provide the names and NetIDs of your collaborator (up to 3). If you finished the lab alone, write None.

2 Lab Setup

There are two different parts in this lab. The format string(FMT) attack carries partial credit (70%), the Return-oriented Programming (ROP) attack gives rest of credit (30%). If you are unable to finish the full lab, at least submit FMT for partial credit. YOU ARE TO SUBMIT BOTH LABS FOR FULL CREDIT.

2.1 Lab File Part 1

This labs have 2 parts. Unzip the archive and move part 1 into a new directory and use the following commands to clone the `scripts` folder and the lab server folder from GitHub. Please check that Shang's public key is imported afterwards or your lab may be unable to be verified. Please note that you need 2 separate directories for the labs.

```
git clone git@github.com:nyupcs/GitCTF-scripts.git scripts
git clone git@github.com:nyupcs/pcs-sp21-lab3-partial-credit-server.git
```

Go to the folder `pcs-sp21-lab3-partial-credit-server` and use the following command to build the server container:

```
docker build . --file Dockerfile --tag pcs-sp21-lab3-partial-credit-server:latest
```

2.1.1 config.json

Open `config.json`. Replace `[GitHub Username]` with your `GitHub` username. Replace `[NetID]` with your NYU NetID Name. Replace `[Public Key ID]` with the key ID you generated in previous lab. A sample config.json is there for you to follow. This time, instead of writing just NetID, enter [NetID,Full Name] so that it is easier for us to match your submissions. Remember to remove the brackets as well. See sample config.json as reference.

2.2 Vulnerable Program Part 1

In the `service` folder inside `pcs-sp21-lab3-partial-credit-server`, you will find `echo.c` which is the vulnerable program used for this part of the lab.

```
/* service/echo.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>    //strlen
#include<unistd.h>    //write

void print(char *buf)
{
    printf(buf);
}

void interact()
{
    char buf[128];
    gets(buf);
    print(buf);
}

int main()
{
    while (1)
    {
        interact();
        fflush(stdout);
    }
    return 0;
}
```

Question 2: What is the vulnerability here in terms of format strings? Explain.

3 Lab Tasks

3.1 Play With the Echo Server

Use the following command to start the server container:

```
docker run -d -p 4000:4000 --name echo-server --cap-add=SYS_PTRACE pcs-sp21-lab3-partial-credit-server
nc 127.0.0.1 4000
```

In this lab, you will be using exploit.py as well to find the memory addresses using string format. First you find out the layout of the code so you can tell how the stack is like. Open CLI of the running docker container and use this.

```
objdump -d echo
```

Trace through <interact> to find old ebx, old ebp, return address, and buf. Hint: printf is the starting place of printing.

3.2 Using the exploit.py

Go to the `answer_template` folder. We have provided some files to start.

The `Dockerfile` contains instructions to build a client to perform the attack. Unless you choose to ignore `exploit.py` and write your own solution, you don't have to worry about this file.

The `exploit.py` contains base to start. It already set up the TCP connection for you and the shellcode is already done for you. Now you need to change the [Edit here]s to correct variable and/or value using information from previous section. Once you receive the flag, print the flag to the `stdout`.

```
#!/usr/bin/env python3

from socket import *
import sys
import time

HOST = sys.argv[1]
PORT = int(sys.argv[2])
BUFSIZE = 1024
ADDR = (HOST, PORT)

s = socket(AF_INET, SOCK_STREAM)
try:
    s.connect(ADDR)
except Exception as e:
    print('Cannot connect to the server.')
    sys.exit()


s.send('[edit format string attack here]')
#Parse and use receive string here

shellcode = b"\x31\xc0\x31\xdb\x31\xc9\x31\xd2"+\
    b"\xeb\x32\x5b\xb0\x05\x31\xc9\xcd"+\
    b"\x80\x89\xc6\xeb\x06\xb0\x01\x31"+\
    b"\xdb\xcd\x80\x89\xf3\xb0\x03\x83"+\
    b"\xec\x01\x8d\x0c\x24\xb2\x01\xcd"+\
    b"\x80\x31\xdb\x39\xc3\x74\xe6\xb0"+\
    b"\x04\xb3\x01\xb2\x01\xcd\x80\x83"+\
    b"\xc4\x01\xeb\xdf\xe8\xc9\xff\xff"+\
    b"\xff"+b"/var/ctf/flag"

payload = '[edit here]'

s.send(payload)
time.sleep(2)
flag = s.recv(BUFSIZE).decode('utf-8')
flag = flag.rstrip()

print('Got the following flag:')
sys.stdout.write(flag)
```

Question 3: Use format string exploit to find the memory addresses. Take a screenshot of you obtaining the addresses and include it in lab report. What is the distance between buff address and the return address?

Now use the addresses obtained to edit the payload.

Question 4: Copy and paste your exploit.py in report

When you finished, remember to stop and remove the docker container. Do the same when you are doing part 2 of the lab.

```
docker rm -f echo-server
```

Attention: Before submitting your code, make sure you have stopped and removed the docker container. Otherwise, the submission script will not be able to verify your code because of port conflict.

4 Submission Part 1

Rename the folder `answer_template` to `answer` and the file `exploit.py` to `exploit`, and use the following command to submit your code:

```
python3 scripts/gitctf.py submit --exploit answer
                                 --service-dir pcs-sp21-lab3-partial-credit-server
                                 --target lab3 --branch main
```

The script will verify your solution locally. If the code passes the test, it will be uploaded to the lab repo as an issue.

During the upload stage, check `yes` when asked to use Shang and PCS's public key. When the script will ask you for your GitHub password, enter the personal access token that you generated in previous lab. `Please check that you have successfully generated an issue and your NetID,Full Name has been recorded.`

`Please note that you team and player_team is netid,full name with a comma in the middle and NO BRACKETS`

5 Understanding the stack

5.1 Sample vulnerable program

Below is a sample C program to help you understand how stacks work.

```c
/* foobar.c */
#include<stdio.h>
void foo(int x)
{
printf("Hello world: %d\n", x);
}
int main()
{
foo(1);
return 0;
}
```

Below is the assembly analysis of the C program.

```
......
8 foo:
9 pushl %ebp
10 movl %esp, %ebp
11 subl $8, %esp
12 movl 8(%ebp), %eax
13 movl %eax, 4(%esp)
14 movl $.LC0, (%esp) : string "Hello world: %d\n"
15 call printf
16 leave
17 ret
......
21 main:
22 leal 4(%esp), %ecx
23 andl $-16, %esp
24 pushl -4(%ecx)
25 pushl %ebp
26 movl %esp, %ebp
```

```
27 pushl %ecx
28 subl $4, %esp
29 movl $1, (%esp)
30 call foo
31 movl $0, %eax
32 addl $4, %esp
33 popl %ecx
34 popl %ebp
35 leal -4(%ecx), %esp
36 ret
```

5.2 Stack layout

We will use line numbers instead of instruction addresses in the explanation.

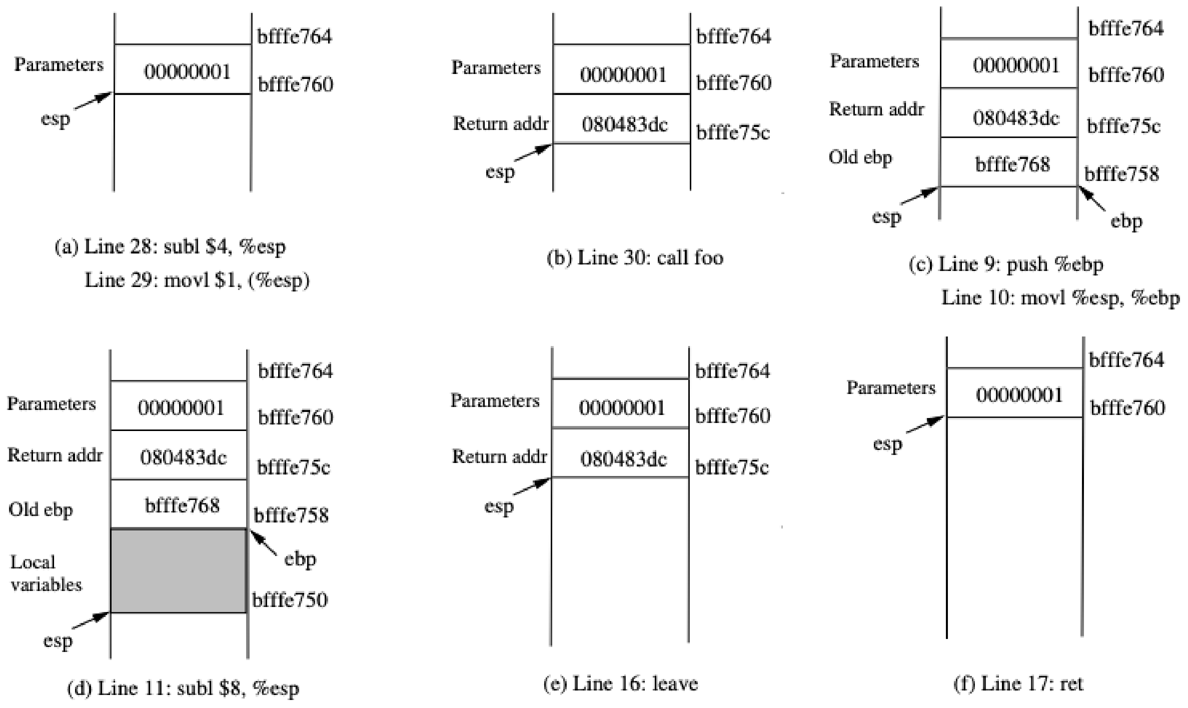

Figure 1: Entering and Leaving `foo()`

Entering and leaving foo() • `Line 28-29`: These two statements push the value 1, i.e. the argument to the foo(), into the stack. This operation increments %esp by four. The stack after these two statements is depicted in Figure 1(a).

• `Line 30: call foo:` The statement pushes the address of the next instruction that immediately follows the call statement into the stack (i.e the return address), and then jumps to the code of foo(). The current stack is depicted in Figure 1(b).

• `Line 9-10:` The first line of the function foo() pushes %ebp into the stack, to save the previous frame pointer. The second line lets %ebp point to the current frame. The current stack is depicted in Figure 1(c).

• `Line 11: subl $8, %esp:` The stack pointer is modified to allocate space (8 bytes) for local variables and the two arguments passed to printf. Since there is no local variable in function foo, the 8 bytes are for arguments only. See Figure 1(d).

5.3 Leaving foo()

Now the control has passed to the function foo(). Let us see what happens to the stack when the function returns.

• `Line 16: leave:` This instruction implicitly performs two instructions (it was a macro in earlier x86 releases, but was made into an instruction later):

mov %ebp, %esp
pop %ebp

The first statement releases the stack space allocated for the function; the second statement recovers the previous frame pointer. The current stack is depicted in Figure 1(e).

• `Line 17: ret:` This instruction simply pops the return address out of the stack, and then jump to the return address. The current stack is depicted in Figure 1(f).

• `Line 32: addl $4, %esp:` Further restore the stack by releasing more memories allocated for foo. As you can see that the stack is now in exactly the same state as it was before entering the function foo (i.e., before line 28).

6 Vulnerable Program Part 2

Create a different directory from part 1 of the lab and import the service files.

```
git clone git@github.com:nyupcs/GitCTF-scripts.git scripts
git clone git@github.com:nyupcs/pcs-sp21-lab3-full-credit-server.git
```

Go to the folder `pcs-sp21-lab3-full-credit-server` and use the following command to build the server container:

```
docker build . --file Dockerfile --tag pcs-sp21-lab3-full-credit-server:latest
```

In the `service` folder inside `pcs-sp21-lab3-full-credit-server`, you will find `echo.c` which is the vulnerable program used for this part of the lab. In this server, the stack is not executable, so you cannot reuse the exploit from part 1. Below is the difference in the respective Makefiles.

```
#in part 1 of lab
$(CC) $(TARGET).c -z execstack -fno-stack-protector -o $(TARGET)
#in part 2 of lab
$(CC) $(TARGET).c -fno-stack-protector -o $(TARGET)
```

By default, newer versions of GCC prevents stack from being executable. Part 1 of the lab was made executable to show how vulnerable programs in the fast were to buffer overflow attacks. In the current cat-and-mouse game between hackers and security professionals, protection features are more common place but more new buffer vulnerabilities are discovered as time passes.

```
/* service/echo.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>    //strlen
#include<unistd.h>    //write

void print(char *buf)
{
    printf(buf);
}

void interact()
{
    char buf[128];
    gets(buf);
    print(buf);
}

void call_system(char *cmd) {
    system(cmd);
}

int main()
{
    call_system("echo 'Welcome to echo server!'");
    while (1)
    {
        interact();
        fflush(stdout);
    }
    return 0;
}
```

Question 5: What is the vulnerability here in terms of return function? Explain.

7 Lab Tasks

Your goal is to override the variable 'buf' in the function <interact> in a way that will let the function return to the function <call_system> instead of <main>.

Use the following command to start the server container:

```
docker run -d -p 4000:4000 --name echo-server --cap-add=SYS_PTRACE pcs-sp21-lab3-full-credit-server
nc 127.0.0.1 4000
```

Run objdump again after connecting to the docker container. This time it is different. You have to find the return address to exploit in <call system> as well to direct back to the payload.

```
#!/usr/bin/env python3

from socket import *
import sys
import time

HOST = sys.argv[1]
PORT = int(sys.argv[2])
BUFSIZE = 1024
ADDR = (HOST, PORT)

s = socket(AF_INET, SOCK_STREAM)
try:
    s.connect(ADDR)
except Exception as e:
    print('Cannot connect to the server.')
    sys.exit()


s.send(b'[Edit here]')
time.sleep(2)
#Parse and use string here

call_system_addr = "Edit here"

payload = "Edit here" + b'cat /var/ctf/flag\n'

s.send(payload)
time.sleep(2)

flag = s.recv(BUFSIZE).decode('utf-8')
flag = flag.rstrip()

print('Got the following flag:')
sys.stdout.write(flag)
```

You will notice there is no shellcode here. You are supposed to direct system call to a string command.

Question 6: Use format string exploit to find the memory addresses. Take a screenshot of you obtaining the addresses and include it in lab report.

Now use the addresses obtained to edit the payload.

Question 7: Copy and paste your exploit.py in report

8 Submission Part 2

8.1 Submit Your Code

Rename the folder `answer_template` to `answer` and the file `exploit.py` to `exploit`, and use the following command to submit your code:

```
python3 scripts/gitctf.py submit --exploit answer
                                 --service-dir pcs-sp21-lab3-full-credit-server
                                 --target lab3 --branch main
```

Question 8: What is your issue URL? Include both labs here.

8.2 Submit Your Lab Report

You need to submit a lab report to answer all the questions above with screenshots to describe what you have done and what you have observed; you also need to provide brief explanations to the observations that are interesting or surprising.

Late submissions are accepted with 50% grading penalty within 24 hours of the due. Submissions that are late for more than 24 hours will NOT be accepted.

Please submit your solution in PDF or image on https://www.gradescope.com/courses/225188, using Entry Code: **86EDWX**. And kindly choose the right page for your answer to every question.

Collaboration Policy

You can optionally form study groups consisting of up to 3 people per group (including yourself).

Everybody should write individually by themselves and submit the lab reports separately. DO NOT copy each other's lab reports, or show your lab reports to anyone other than the course staff, or read other students' lab reports.

9 Acknowledgements

This lab is adopted from the Git-based CTF project. The original project is here and the paper is here.

This PDF was created on 2021-03-10 02:59:40Z.