CS426 Computer Security: Lab 3

Dave (Jing) Tian

March 31, 2021

1 Deadline

Sat Apr 10, AoE. (You know the late penalty.)

2 Goal

Basic binary analysis + shellcode injection on modern 64-bit architecture using Linux

3 Prerequisites

This assignment has been designed for **Ubuntu 20.04 LTS** on **x86-64** architecture. Please make sure your own Linux box has the right OS and architecture, or your VM use the right image if you are using VirtualBox, QEMU, etc. To make things easier for everyone, make sure you **disable ASLR** ahead of time before playing. Figure below has demonstrated the `uname` information of the machine I used and how to disable ASLR on Linux.

You are free to use whatever tools you like. Tools I used when designing this assignment include `strings`, `readelf`, `gdb`, `objdump`, `file`, `gcc`, and `python`. `gdb` will be super helpful here. So make sure you are making friend with her.

```
daveti@daveti-hp:~/git/compsec/labs/lab3$ uname -a
Linux daveti-hp 5.4.0-70-generic #78-Ubuntu SMP Fri Mar 19 13:29:52 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
daveti@daveti-hp:~/git/compsec/labs/lab3$
daveti@daveti-hp:~/git/compsec/labs/lab3$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
[sudo] password for daveti:
0
daveti@daveti-hp:~/git/compsec/labs/lab3$
```

4 Description

In the tar file, you should be able to find 3 files: this file, **64bit-overflow.pdf** and **sci**. The PDF provides a detailed work through of exploiting a buffer overflow for shellcode injection on 64-bit architecture. It should give you more understanding of what this assignment looks like beside the slides we discussed during the class. I highly recommend reading it, although I have never tried to replicate the example there.

The **sci** file is the target binary file that you will love or hate. It is an **suid** program, which means it is owned by root and you could get the power of root even if you are not root. (NOTE: please do NOT use root account on your Linux to play with this assignment.) If it is not an suid program in your local Linux system, use the commands below to enable suid again:

```
1 # sudo chown root sci
2 # sudo chmod u+s sci
```

What this program does is to ask you the password of *dave2*, whose password seems unbreakable in our first assignment. If you provide the right password, the program will output **Bingo!** and quit. Otherwise, it will output **Wrong password**.

This program can be run in 2 modes: interactive mode and silence mode. To run in interactive mode, you just run the program without any arguments, and the program will keep looping asking for the right password if your guess is not correct. In silence mode, you run the program with **one** argument, which is assumed to be the password. The program will tell the results without looping. To have a better understanding of this program, check out the figure below.

In the figure, you probably have noticed that the program also outputs the **address of a buffer**. Yes, this buffer is the exact buffer leading to stack overflow within this program. You will find the starting address of this buffer provided by the program super useful soon. In fact, let us talk about this program a bit more:

- It is an suid program.

```
daveti@daveti-hp:~/git/compsec/labs/lab3$ ll ./sci
-rwsrwxr-x 1 root daveti 17752 Mar 31 12:18 ./sci*
daveti@daveti-hp:~/git/compsec/labs/lab3$
daveti@daveti-hp:~/git/compsec/labs/lab3$ file ./sci
./sci: setuid ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter
daveti@daveti-hp:~/git/compsec/labs/lab3$
daveti@daveti-hp:~/git/compsec/labs/lab3$
daveti@daveti-hp:~/git/compsec/labs/lab3$ ./sci
Password for dave2>
hello
buffer=[0x7fffffffe370]
Wrong password
Password for dave2>
oops
buffer=[0x7fffffffe370]
Wrong password
Password for dave2>
^C
daveti@daveti-hp:~/git/compsec/labs/lab3$
daveti@daveti-hp:~/git/compsec/labs/lab3$
daveti@daveti-hp:~/git/compsec/labs/lab3$
daveti@daveti-hp:~/git/compsec/labs/lab3$ ./sci hello
buffer=[0x7fffffffe370]
Wrong password
daveti@daveti-hp:~/git/compsec/labs/lab3$
```

- It has implemented 2 functions (besides the ones provided by libc).
- It actually contains the password for *dave2*. (NOTE: do NOT do this in practice!)
- It prints out the starting address of the buffer that could be overflowed.
- The buffer is zero-out explicitly from the start.
- I have been told that the overflow is essentially enabled by `strcpy`.
- It is not stripped. (You are welcome.)
- It does not contain debugging information. (Oh yeah!)
- It does not contain stack protections and its stack is executable. (You are welcome again.)

Armed with all these useful information, you will need to find the secret password for poor Dave, and inject shellcode to exploit the stack overflow to create a shell. Let's have a detailed breakdown.

4.1 Task 1: Crack the password

Since the program is able to tell if your password is correct or not, it is reasonable to assume that the correct one is saved somewhere, and further compared with an provided input using some typical libc function, e.g., `strncmp`

or something else. You might need to look into the assembly code to figure out how the whole password checking is done. Questions you might want to ask include:

- How do I find all the strings contained inside a binary? *Hint: strings*
- How do I see the assembly code of a binary? *Hint: objdump*
- How do I see the memory information of a program? *Hint: gdb*

4.2 Task 2: Inject the shellcode

To inject shellcode, we need to generate our shellcode first, explore how the overflow could be triggered, and finally come up with the right payload for buffer overflow.

4.2.1 Task 2.1: Generate the shellcode

The class slides have provided a minimum shellcode example using `execve` syscall on 32-bit architecture. You need to port it to 64-bit architecture. Since `int 0x80` was mainly designed for 32-bit architecture and the target program (sci) is 64-bit already, we could not reuse this instruction anymore. Instead, please use `syscall` instruction. Watch out for the differences between these 2 architectures and these 2 different syscall instructions.

That is being said, you probably need to write your own ASM file, e.g., **shellcode.s**, gets it compiled (*Hint: gcc*), and extract the machine code there (*Hint: objdump*). Note that your shellcode should NOT rely on libc, and it should NOT contain *NULL* bytes.

4.2.2 Task 2.2: Explore the overflow

The PDF provides an example on how to explore buffer overflow with the help of **python one liner**, which is essentially sending out a number of "A" as the input and waiting to see the crash. Questions you should consider include:

- Where is the buffer that could be overflowed? E.g., which function?
- What is the stack layout of that function? *Hint: objdump*

- How do I know if my "A"s overflow as expected? *Hint: gdb*
- How many "A"s do I need to overwrite the return address on the stack so that I could change the rip?

In the end, you should be able to answer all the questions above since now you are able to redirect the control flow of the program to something such as "AAAAAAAA", or to whatever address you want it to jump.

4.2.3 Task 2.3: Put everything together

Now you have your shellcode and you know how many bytes you need to overwrite the return address saved on the stack, it is time to put them together. The PDF again provides an example on how to do this. Remember the buffer **address** shown by the program? You should know why it is important and needed now. Because of the instrumentation from `gdb`, the memory layout from gdb is different from the one running without gdb, although in practice you could guess the "real" value based on the "gdb" value, e.g., by adding 0x50 or something similar. Note that the buffer address might also change based on the length of the input.

Your final shellcode payload might look like this – X-byte shellcode — Y-byte "A"s — Z-byte shellcode-address. You should know X, Y, and Z at this moment. Just watch out the endian issue of the Z-byte and you will be fine. As usual, your shellcode payload cannot contain any *NULL* bytes. Since your shellcode generated earlier does not contain zero bytes already, the only possible case where zero bytes might show up again would be the Z-byte. If it happens unfortunately, **what are you gonna do?**

4.2.4 Task 2.4: Create root shell (optional)

It is very likely that the shell you get in the end does not have the root permission. For instance, it was still created under your own UID rather than root (UID=0). As we mentioned earlier, this **sci** is an suid program and should have the power of root. So how exactly could we get a root shell? Here are some clues:

- Why does the program ask for password in the first place?
- What other functions have you seen from the objdump of the program other than `strcpy` and `strncmp`?

- What does the program do other than showing "Bingo!"?

All I can tell is that the password is important:) "Talk is cheap." Show me your shellcode payload leading to a root shell.

5 Deliverable

The final deliverable will be a technical report in PDF, generated from Latex or G-doc or whatever you like. It should contain enough instructions and screenshots to prove the integrity, validity, and reproducibility of your work. Specially, the report should include:

- Task 1: instructions and screenshots of how to get the password from the binary using certain tools.
- Task 2.1: source file of the shellcode, command used to compile the shellcode, and screenshot of extracting the machine code using certain tools.
- Task 2.2: screenshots of how to explore the buffer overflow, and instructions of the stack layout around the buffer with **exact** length information.
- Task 2.3: screenshots of your shellcode injection, and output of running `id` and `date` commands within the spawned shell.
- Task 2.4 (optional): same as Task 2.3

Missing of certain type of information or unclear instructions or screenshot will lead to penalty.

6 Grading

- Task 1: 50 pts
- Task 2.1: 20 pts
- Task 2.2: 10 pts
- Task 2.3: 20 pts
- Task 2.4 (optional): 50 pts

7 Others

Buffer overflow + shellcode injection is probably one of the most well-known offensive techniques. It is so easy to understand and yet hard to do in practice. Nevertheless, it is doable. Don't lose your patience easily and make progress everyday. You will love the fun of hacking low-level security stuffs and hopefully enjoy the moment when your shellcode works. Happy hacking.