Buffer Overflow Exercise

1. Overview

The learning objective of this lab is for you to gain the first-hand experience on buffer-overflow vulnerability. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

Lab environment: Use the pre-built Ubuntu machine that has been provided for this class. **Note:** If you are using a shared folder between your host machine and your VM, make sure to not run the codes from inside the shared folder. Copy them to some other location on Ubuntu and run from there.

Submission: You need to submit a detailed lab report to describe what you have done and what you have observed. Follow the tasks and for each task answer the **Q#** specifically in your report. You may provide an explanation of the observations that are interesting or surprising. **You should always add any code that you modified and screenshots of what you have observed to your report.** You are encouraged to pursue further investigation, beyond what is required by the lab description. Only submit typed reports electronically. No handwritten reports are accepted.

2. Tasks

Task 1: Setting up the environment by disabling countermeasures

As previously discussed in class, Operating systems and compilers have implemented several security mechanisms to make the buffer overflow attack more difficult. In this section, in order to make the attack easier, we will first disable these countermeasures.

- **Address Space Randomization.** In order to disable address randomization temporarily use the following command:

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

  Remember that the changes we make are temporary, so if you reboot your system you will need to run this command again. To view the current settings for ASLR on your system you can use this command:

  ```
  $ sysctl -a --pattern randomize
  ```

  If the value is 2, it means full randomization is on, and if it is 0 it means that randomization is disabled.

- **StackGuard protection** scheme is a method used by GCC compiler to prevent buffer overflow. We will disable this feature by using the `-fno-stack-protector` option while compiling the program.

- **Nonexecutable stack.** The most recent version of gcc automatically marks the binary version of program to indicate that this program does not require executable stack. In order to disable this countermeasure we use the option `-z execstack` when compiling the program to make the stack executable.

- **Configuring /bin/sh** As discussed in Lab1, the shell in Ubuntu20.04 VM has a countermeasure that prevents itself from being executed in a Set-UID process. In this lab, our victim program is a Set-UID program, and our attack is trying to run /bin/sh, therefore this countermeasure makes our attack more difficult. We will make our attack easier by linking /bin/sh to another shell that does not have such a countermeasure. Later we will see that it is not very hard for the attacker to defeat this countermeasure as part of the attack. You've already done this part in Lab1. In order to see if your shell is /bin/zsh type in:

```
$ ls -la /bin/sh
```

If it shows /bin/zsh you are good to go, otherwise run the following two commands:

```
$ sudo rm /bin/sh
$ sudo ln -s /bin/zsh /bin/sh
```

Task 2: The shellcode

Before doing the attack, let's get familiar with the shellcode. A shellcode is a code that launches a shell. This is usually the malicious code injected by the attacker to the stack.

A program called `shellcodetest.c` is given to you. Take a look at this file. `code[]` contains the program to execute a shell in bytecode. The C version of this code is provided below. It is as if you write a program to execute a shell in C and then compile and extract the bytecode from it.

```
#include <stdio.h>
int main() {
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

`shellcodetest.c` program is written to test this bytecode to make sure it will launch a shell. Compile this program using the following gcc command.

```
$ gcc -m32 -z execstack -o shellcodetest shellcodetest.c
```

Q1: Run the program and describe your observations. Add a screenshot of what you observed.

Task 3: The vulnerable program

You are provided with a program called `unsafe.c`. We are assuming as the attacker you have access to the executable file for this program which is also a root-owned Set-UID program. This program has a buffer overflow vulnerability and your goal in this lab is to exploit this vulnerability to spawn a shell with root privilege.

Compile this vulnerable program by including the `-fno-stack-protector` and `-z execstack` options to turn off the StackGuard and the non-executable stack protections as shown below.

```
$ gcc -m32 -z execstack -fno-stack-protector -o unsafe unsafe.c
```

After the compilation, you need to make the program root-owned Set-UID. Remember the commands to do this from Lab1?

This program has a buffer overflow vulnerability as indicated in the code. It first reads an input from a file called `inputfile`, and then passes this input to another buffer in the function `copyfunc()`. The original input can have a maximum length of 527 bytes, but the buffer in `copyfunc()` is only BUFFSIZE bytes long. Because `strcpy()` does not check boundaries, buffer overflow will occur. Since this program is a root-owned Set-UID program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a shell with root privileges. Remember that this program receives its input from a file and this file is under user's control. Your goal is to create the contents for *inputfile*, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

Q2: Take a look at the unsafe.c program, assuming we are at the first line inside of copyfun() draw the stack frame in your report. This means your diagram will include the stack frames for foofunc() and copyfunc(). Also remember any function calls that have been completed is no longer on the stack.

Task 4: Exploiting the vulnerability, the real attack

To exploit the buffer-overflow vulnerability in the unsafe.c program, we need to prepare a payload (malicious code that is designed to execute a specific action on a target system), and save it inside *inputfile*. You are given a partially completed exploit code called `exploit.py`. The goal of this code is to construct contents that will be written in `inputfile`, and will later get copied into the buffer when the `unsafe` program is running and the buffer overflow vulnerability results in invoking a shell. You need to fill out parts of the code with '#Need to change' comment.

Your task is to (1) find out the values needed to complete `exploit.py` by debugging the unsafe code as described below. (2) finish `exploit.py` program and run it by typing `python3 exploit.py`. This will generate the `inputfile`. **troubleshoot:** If you receive a permission denied error when running `exploit.py`, you may need to give it execute permission. Try to change the permission to rwx–x–x, for example.

After you created the inputfile by running the `exploit.py`, (3) run the vulnerable program `unsafe`. If your exploit is implemented correctly, you should be able to get a root shell with # sign. Once you spawn the shell you can check your effective uid and real uid by typing `id` on the screen.

In order to be able to complete this task you need to refer to the explanation I have provided in class in regard to this assignment. To exploit the buffer-overflow vulnerability, the most important thing to know is the distance between the buffer's starting position and the place where the return-address is stored. We will use a debugging method to find it out. The next section ”Help with gdb” is useful in finding the addresses.

Help with gdb

You can use gdb to get more information about the vulnerable program. Compile the code to debug the program:

```
$ gcc unsafe.c -m32 -o unsafe_gdb -g -z execstack -fno-stack-protector
```

Create an empty inputfile

```
$ touch inputfile
```

Run the debugger

```
$ gdb unsafe_gdb
```

Inside the debugger, create a breakpoint at the copyfunc()

```
gdb-peda$ b copyfunc
```

Then run the code inside the debugger:

```
gdb-peda$ run
```

At this point gdb stops before the ebp register is set to point to the current stack frame, so if we print out the value of ebp here, we will get the caller's ebp value. We need to use next to execute a few instructions and stop after the ebp register is modified to point to the stack frame of the copyfunc() function.

```
gdb-peda$ next
```

You can print the address of the `buffer` using the command:

```
gdb-peda$ p &buffer
```

You can print the ebp address using the command:

```
gdb-peda$ p $ebp
```

Use Control+D or type quit to exit from GDB at any time.

Now that you have the $ebp and &buffer values, look at the diagram you drawn for the stack frame and find the distance from the buffer's starting position to the beginning of return-address (find this distance in decimal, not hex). This is the value of *offset* in exploit.py code. Find the other values that need to be changed in the Python code.

Q3: Deliverables: (1) Show a screenshot of your exploit.py code. (2) Write the values of &buffer and $ebp and explain how you came up with the *offset* value and content[offset+0 to 3]. It is very important to clearly explain your approach in finding these values. (3) Show a screenshot of the result when you run the `unsafe` program, and the result of `id` from the shell you invoked.

Task 5: Defeating shell's countermeasure

As explained before, the shell in Ubuntu 20.04 drops privileges when it detects that the effective UID does not equal to the real UID. This is what we saw in Lab1, and in order to make our attack work, we changed the shell we are using in Ubuntu to a version that does not have this countermeasure. In this task, we want to see that even if we don't disable this countermeasure, the attacker is able to defeat this. In order to be able to test this, you first need to change the shell back to a version with the countermeasure by using the following commands:

```
$ sudo rm /bin/sh
$ sudo ln -s /bin/dash /bin/sh
```

One approach for the attacker to defeat this countermeasure is to change the real user ID of the victim process to zero before invoking the shell program. This way, when the shell compares the effective uid and the real uid they both will be equal to zero. This is because the effective uid is 0 as a result of the program being a root-owned Set-UID program. We can achieve this by calling `setuid(0)` before invoking `/bin/sh` in the shellcode. The bytecode for calling `setuid(0)` is as follows. All you need to do is to add the following code to the beginning of the shellcode in `exploit.py`:

```
    "\x31\xc0"              /* xorl    %eax,%eax             */
    "\x31\xdb"              /* xorl    %ebx,%ebx             */
    "\xb0\xd5"              /* movb    $0xd5,%al             */
    "\xcd\x80"              /* int     $0x80                 */
```

Q4: Try the attack from Task 4 again and see if you can get a root shell. Check the uid by typing `id` and report it. Show screenshots. Please describe your results.

Task 6: Defeating Address Randomization

As mentioned in class, on 32-bit Linux machines, stacks randomization space is not very high and can be brute forced. In this task we will try to defeat the address randomization countermeasure by brute forcing. First we need to turn on the Ubuntu's address randomization using the following command.

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

Q5: Run the same attack as in Task 4 and report your observation. Show screenshots.

We now try to attack the vulnerable program repeatedly, hoping that the address we put in the `inputfile` can eventually be correct. You can use the following shell script to run the vulnerable program in an infinite loop. If the attack succeeds, the script will stop, and you will get a shell. This may take a while, try it as long as it is possible depending on your system. You might want to let it run overnight if needed.

```
#!/bin/bash

SECONDS=0
value=0
while [ 1 ]
    do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(($duration / 60))
    sec=$(($duration % 60))
    echo "It has been $min:$sec (mins:secs)"
    echo "The program has been running $value times so far."
    ./unsafe
    echo ""
done
```

Q6: Run this code and describe your observations. Show a screenshot of how long you have run the script. If you did not get to successfully run the attack after a reasonable time, first take a screenshot of your screen, then stop the program and report your result. In order to stop the program you can use Ctrl-C, and if it doesn't work, just close the terminal. Show a screenshot of how long you ran the script. For this question, make sure you write down in your report whether you are using Amazon AWS or the virtual box.

Note for those who are not familiar with writing shell scripts: Write the provided code in a file, suppose you name it `myattack`. Then run this file: `./myattack`. If you get a permission denied error, give the file execute permission and run it again.