

Exercises

Memory Errors: Buffer Overflow & Format String



Exercise 1

Consider the C program below, which **is affected by a typical buffer overflow vulnerability**.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  void vuln() {
6      char buf[32];
7      /*          <----- here          */
8      scanf("%s", buf);
9      if (strncmp(buf, "Knight_King!", 12) != 0) {
10         abort();
11     }
12 }
13
14 int main(int argc, char** argv) {
15     vuln();
16 }
```

Assume that the program runs on the usual IA-32 architecture (32-bits), with the usual “cdecl” calling convention. Also assume that the program is compiled **without any mitigation against exploitation** (ASLR is off, stack is executable, and stack canary is not present).

1. [2 points] Draw the stack layout **when the program is executing the instruction at line 7**, showing:

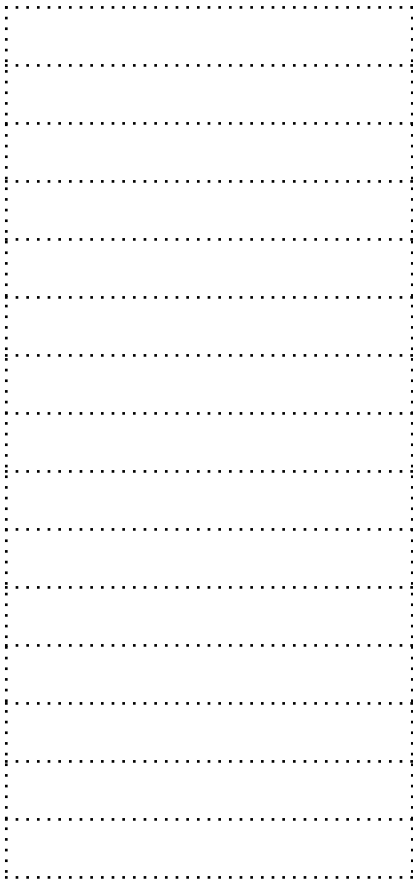
- a. Direction of growth and high-low addresses.
- b. The name of each allocated variable.
- c. The boundaries of frame of the function frames (main and vuln).

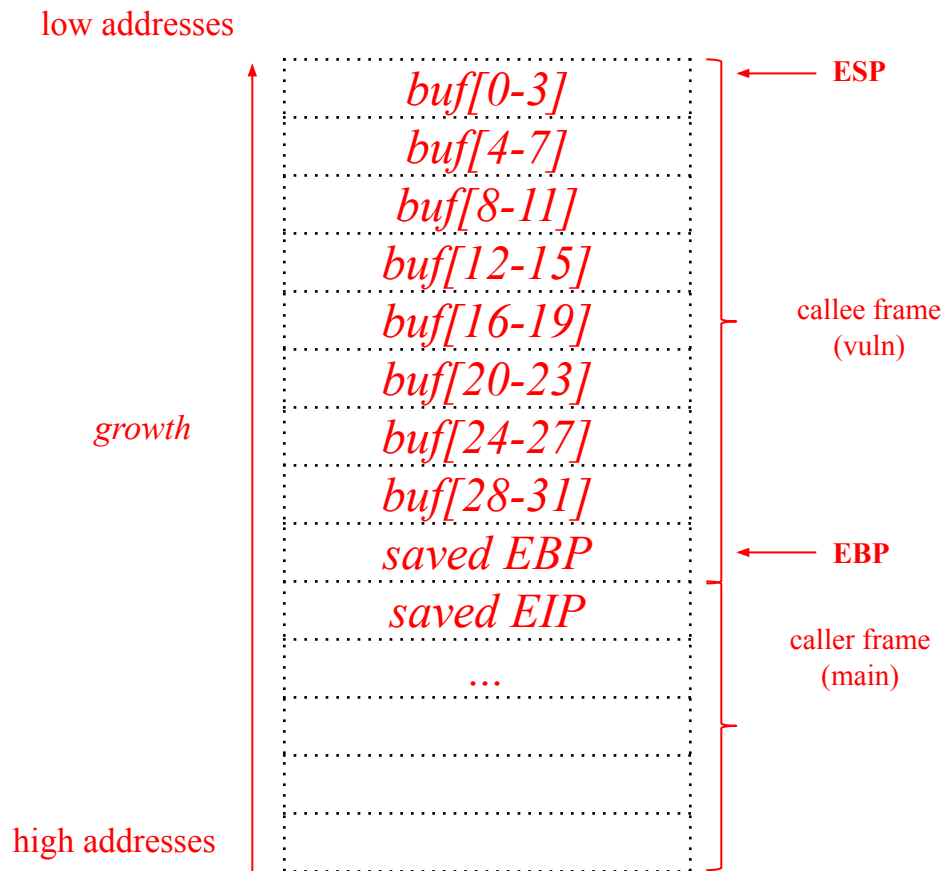
Show also the content of the caller frame (you can ignore the environment variables, just focus on what matters for the vulnerability and its exploitation).

1. Draw the stack layout **when the program is executing the instruction at line 7**, showing:

- a. Direction of growth and high-low addresses.
- b. The name of each allocated variable.
- c. The boundaries of frame of the function frames (main and vuln).

Show also the content of the caller frame (you can ignore the environment variables, just focus on what matters for the vulnerability and its exploitation).



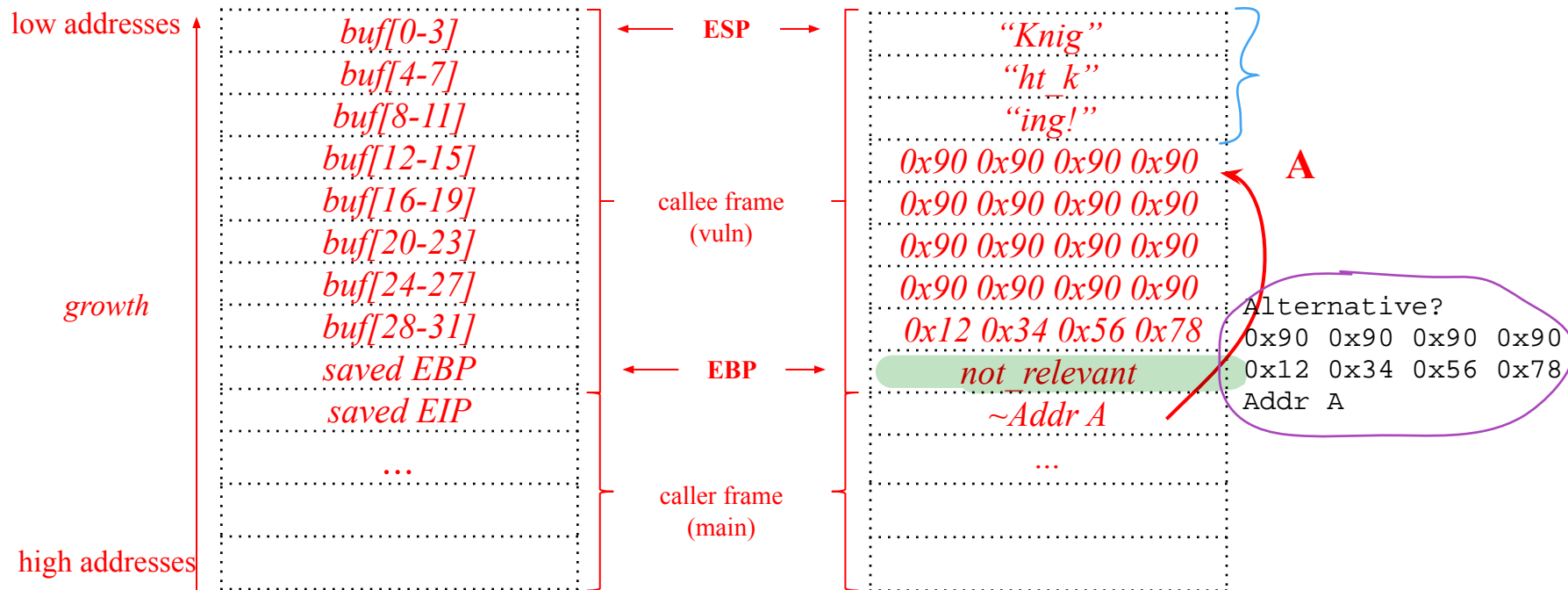


Write clearly all the steps and assumptions you need for the exploitation, and show the stack layout right after the execution of the scanf() during the program exploitation.

This image shows a blank sheet of primary-ruled paper. It features a series of horizontal dotted lines spaced evenly down the page. A single vertical dashed line runs down the center, creating two equal-width columns. The paper is otherwise white and contains no other markings or text.[illegible]

2. Write an exploit for the buffer overflow vulnerability in the above program. Your exploit should execute the following simple shellcode, composed only by 4 instructions (4 bytes): 0x12 0x34 0x56 0x78.

Write clearly all the steps and assumptions you need for the exploitation, and show the stack layout right after the execution of the scanf() during the program exploitation.



3. If address space layout randomization (ASLR) is active, is the exploit you just wrote still working without modifications? Why?

3. If address space layout randomization (ASLR) is active, is the exploit you just wrote still working **without modifications? Why?**

*No, because the **address of the stack** would be **randomized** for every execution, and we must have to have a **leak** in order to exploit it successfully.*

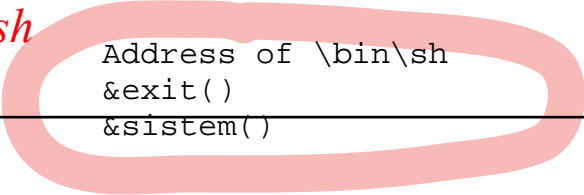
4. If the **stack is made non executable** (i.e., NX, W^X), is the exploit you just wrote still working *without modifications*? If not, propose an alternative solution to exploit the program.

4. If the **stack is made non executable** (i.e., NX, W^X), is the exploit you just wrote still working *without modifications*? If not, propose an alternative solution to exploit the program.

*No, we can use ret to **libc** technique in order to execute the **system** function in the libc and obtain a **shell**.*

In order to do that, we have to prepare the stack to:

- change the value of **A** and let it point to the **address of system***
- write in the buffer **/bin/sh***
- write just above the **overwritten sEIP** the return address of the called library and the address of the string **/bin/sh***



```
Address of \bin\sh
&exit()
&system()
```

Exercise 2

Consider the C program below:

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <string.h>
04
05 void vuln(int n) {
06     struct {
07         char buf[16];
08         char tmp[16];
09         int sparrow = 0xBAAAAAAD;
10     } s;
11
12     if (n > 0 && n < 16) {
13         fgets(s.buf, n, stdin);
14         if(strncmp(s.buf, "H4CK", 4) != 0 || s.buf[14] != "X") {
15             abort();
16         }
17         scanf("%s", s.tmp);
18         if(s.sparrow != 0xBAAAAAAD) {
19             printf("Goodbye!\n");
20             abort();
21         }
22     }
23 }
24
25 int main(int argc, char** argv) {
26     vuln(argc);
27     return 0;
28 }
```

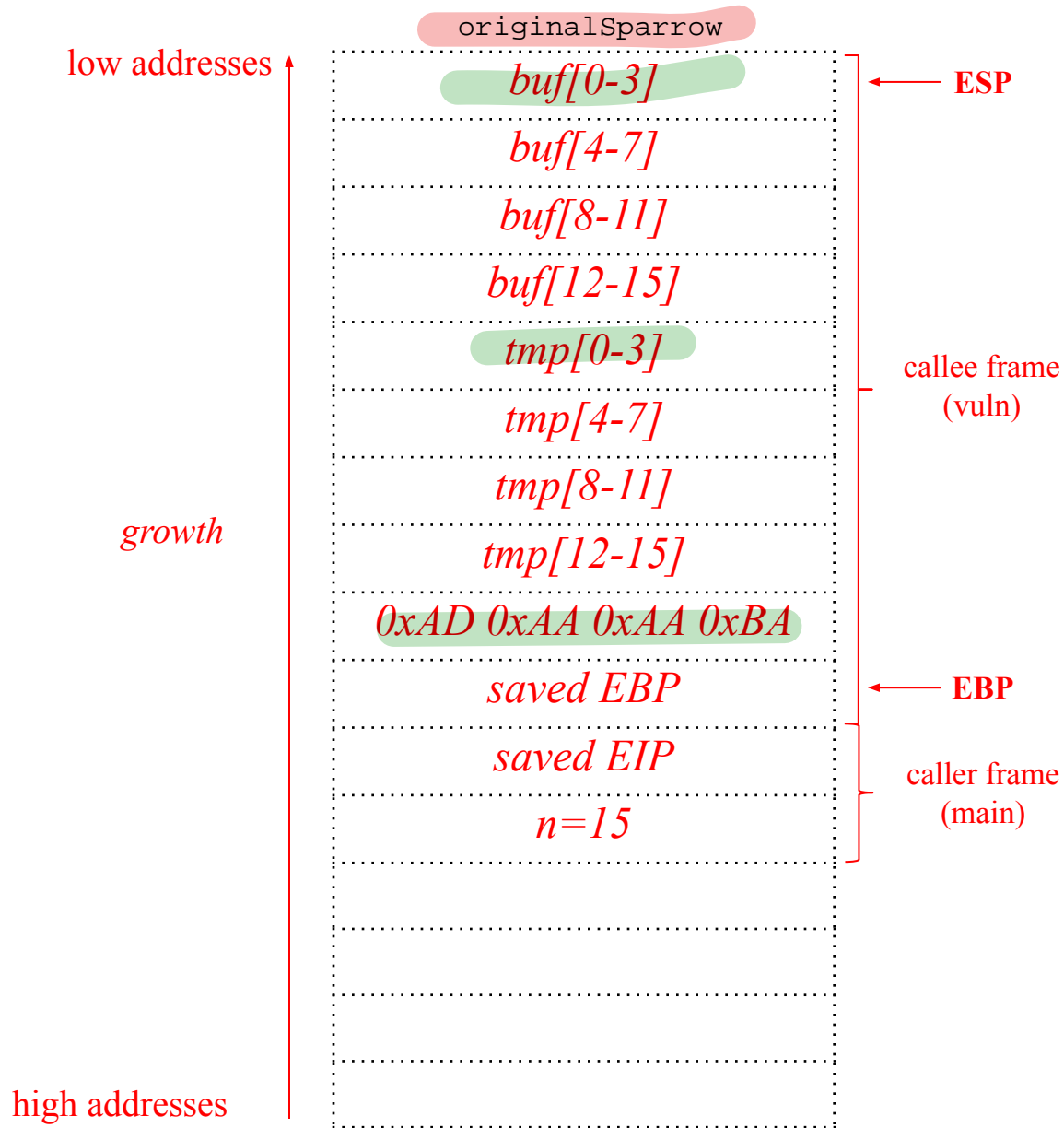
1. Assume the usual IA-32 architecture (32-bits), with the usual “cdecl” calling convention. Assume that the program is compiled without any mitigation against exploitation (the address space layout is not randomized, the stack is executable, and there are no stack canaries).

Draw the stack layout when the program is executing the instruction at line 12, showing:

- a. Direction of growth and high-low addresses;
- b. The name of each allocated variable;
- c. The boundaries of frame of the function frames (main and vuln).

Show also the content of the caller frame (you can ignore the environment variables, just focus on what matters for the vulnerability and its exploitation).

[illegible]



2. The program is affected by a buffer overflow vulnerability. Complete the following table.

Vulnerability	Line(s)	Motivation
Buffer Overflow		

2. The program is affected by a buffer overflow vulnerability. Complete the following table.

Vulnerability	Line(s)	Motivation
Buffer Overflow	17	<i>Scanf("%s") reads unlimited characters</i>

3. The underlined lines of code attempt to mitigate the exploitation of the buffer overflow vulnerability you just found.

3.a. Shortly describe what is the implemented technique, and how it works in general.

3.b. Describe the weaknesses in this specific implementation, and how you would fix them.

3. The underlined lines of code attempt to mitigate the exploitation of the buffer overflow vulnerability you just found.

3.a. Shortly describe what is the implemented technique, and how it works in general.

The underlined code implements a stack canary as a mitigation against stack-based buffer overflows. When writing past the end of a stack-allocated buffer, one would overwrite the variable x before overwriting the saved EIP. Before returning from the function, the content of the variable x is checked against the original value: if they differs, a buffer overflow is detected and the program aborts without returning from the function (i.e., without triggering the exploit).

3.b. Describe the weaknesses in this specific implementation, and how you would fix them.

In this case, the stack canary is a static value (0xBAAAAAAD): if the binary is available, it is enough to retrieve this value by reverse engineering the program; then, during the exploitation it is enough to overwrite the stack canary with this (known) value. To fix this issue, the stack canary should be randomized at the program startup and placed in a register.

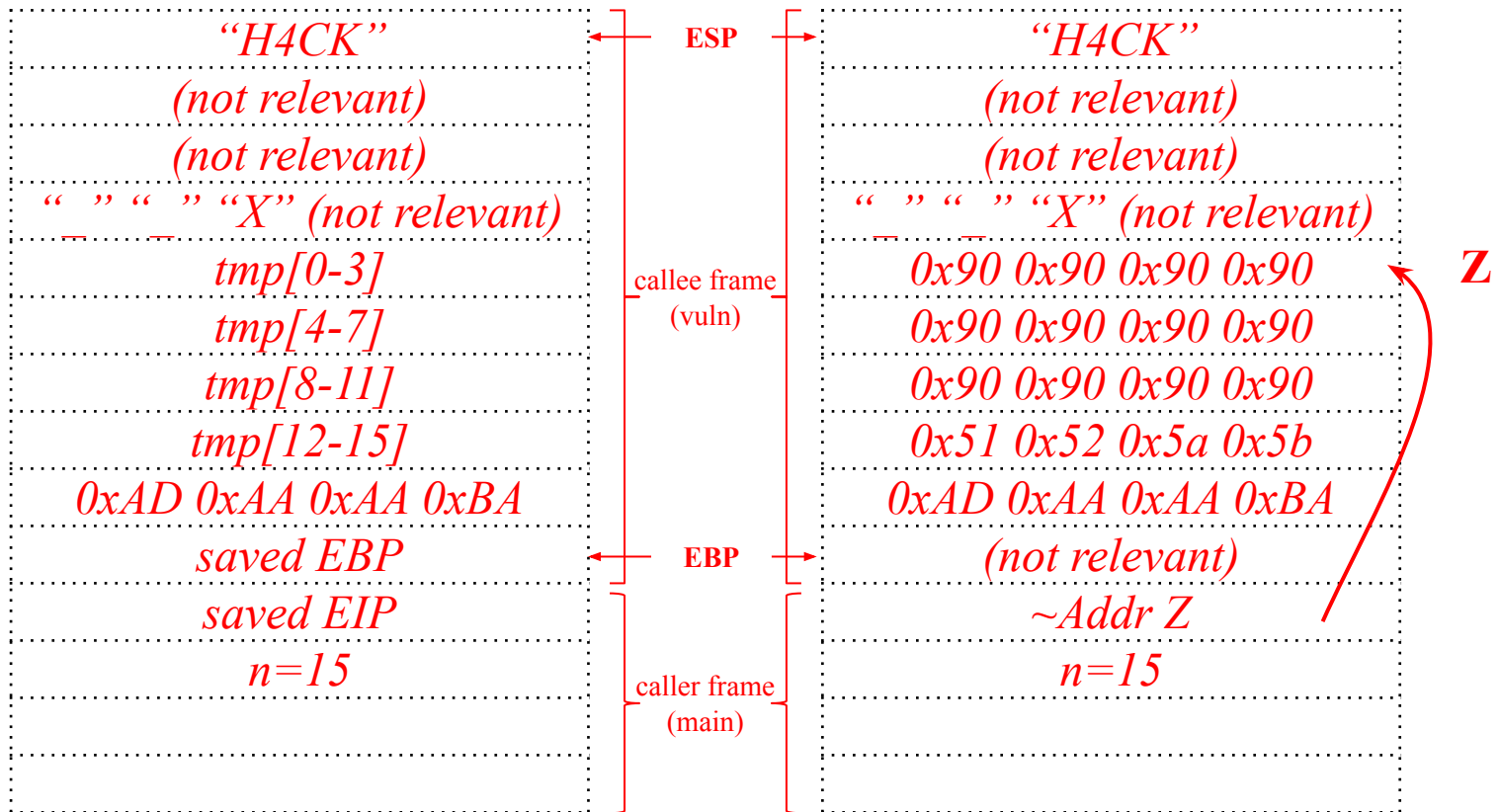
4. Write an exploit for the buffer overflow vulnerability in the above program to execute the following simple shellcode, composed only by 4 instructions (4 bytes): **0x51 0x52 0x5a 0x5b**. Make sure that you show how the exploit will appear in the process memory with respect to the stack layout right before and after the execution of the detected vulnerable line during the program exploitation. Ensure you include all of the steps of the exploit, ensuring that the program and the exploit execute successfully. Include also any assumption on how you must call the program (e.g., the values for the command-line arguments required to trigger the exploit correctly, environment variables).

[illegible][illegible]

low addresses

growth

high addresses



5. Assuming that W^X is enabled (i.e., non-executable stack):

5.a. Is the exploit you just wrote still working? Why?

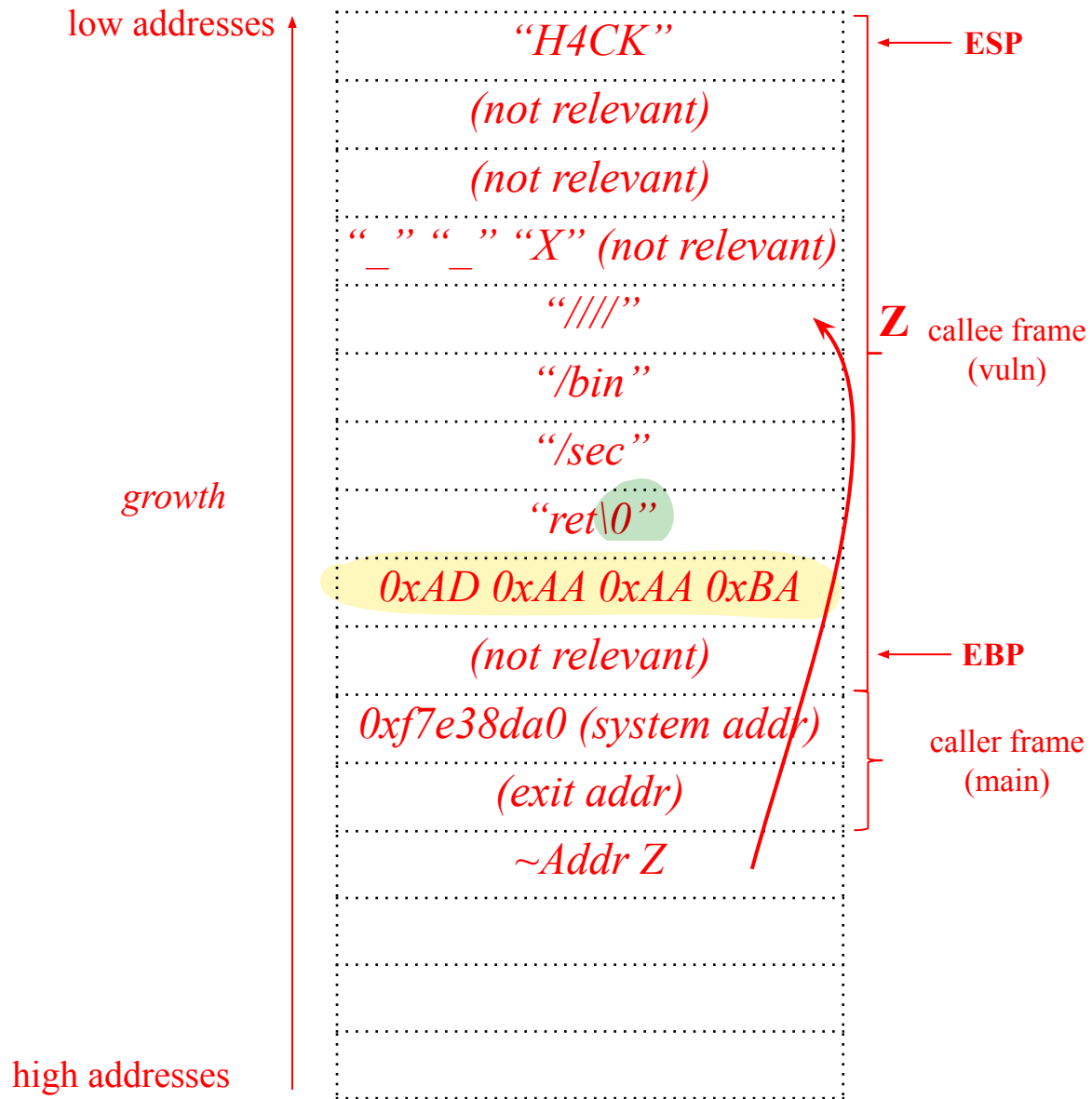
5. Assuming that W^X is enabled (i.e., non-executable stack):

5.a. Is the exploit you just wrote still working? Why?

No, because we can't jump to our shellcode (the stack is non-executable).

5.b. Let's assume that the C standard library is loaded at a known address during every execution of the program, and that the (exact) address of the function `system()` is `0xf7e38da0`. Explain how you can exploit the buffer overflow vulnerability to launch the program `/bin/secret`.

We write in `tmp` the string `/bin/secret`, and we overwrite the saved EIP with the address of the `system`. We then overwrite the next 4 bytes with the address of the `exit()`, so that when the system finishes, it will call the `exit()` function (optional). Finally, we overwrite the next 4 bytes with the address of the `tmp` variable (required), so that the parameter of the `system()` is `/bin/secret`.



6. Assuming that ASLR is enabled (i.e., randomized memory layout): Is the exploit you just wrote still working? Why?

No, because with ASLR enabled, we don't know neither the address of system() in the libc, nor the address of the string /bin/secret written in the buffer (both the stack and the shared libraries are randomized)

7. Assume now that the program is compiled without any mitigation against exploitation (the address space layout is not randomized, the stack is executable, and there are no stack canaries). Propose the simplest **modification to the C code provided** (i.e., patch) that **solves** the **buffer overflow vulnerability** detected, motivating your answer.

7. Assume now that the program is compiled without any mitigation against exploitation (the address space layout is not randomized, the stack is executable, and there are no stack canaries). Propose the simplest **modification to the C code provided** (i.e., patch) that **solves** the **buffer overflow vulnerability** detected, motivating your answer.

7. Assume now that the program is compiled without any mitigation against exploitation (the address space layout is not randomized, the stack is executable, and there are no stack canaries). Propose the simplest **modification to the C code provided** (i.e., patch) that **solves** the **buffer overflow vulnerability** detected, motivating your answer.

```
scanf("%15s", s.tmp);  
fgets(s.tmp, 15, stdin);
```

+ *motivation*

Exercise 3

```
00 #include <stdio.h>
01 #include <stdlib.h>
02 #include <string.h>
03 #include <stdint.h>
04
05 typedef struct {
06     char buf[40];
07     char sel;
08 } data_t;
09
10 void reverse_string(data_t * data) {
11     char prefix[8];
12
13     scanf("%16s", prefix);
14     sprintf(data->buf, "%s\n", strrev(data->buf));
15     if (strncmp(data->buf, "DAVINCI:", 8) == 0) {
16         printf("%s", prefix);
17         printf(data->buf);
18         return;
19     }
20     else {
21         abort();
22     }
23 }
24
25 void main(int argc, char** argv) {
26     data_t data;
27
28     clearenv(); // Clear environment variables
29
30     scanf("%40s", data.buf);
31     scanf("%c", &data.sel);
32
33     if (data.sel == 'r') {
34         reverse_string(&data);
35     } else {
36         printf("%s", data.buf);
37     }
38 }
```

2. The program is affected by a typical buffer overflow and a format string vulnerability. Complete the following table, focusing on a vulnerability per row.

Vulnerability	Line	<u>Motivate and Modify the line to solve the detected vulnerability</u>
Buffer Overflow	13	It can fill the buffer with 16 characters, but the buffer can just contain 8 characters <code>scanf("%8s", prefix);</code>
Format String	17	It doesn't use placeholders. <code>printf("%s", data->buf);</code>

2. The program is affected by a typical buffer overflow and a format string vulnerability. Complete the following table, focusing on a vulnerability per row.

Vulnerability	Line	<u>Motivate and Modify the line to solve the detected vulnerability</u>
Buffer Overflow	13	[see slides]
Format String	17	[see slides]

[2 points] 2. Focus only on the stack-based buffer overflow(s) you found. Write an exploit for this vulnerability that **must execute the following shellcode**, composed of 8 bytes, which opens a shell: **0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38**.

2.a) Describe **all the steps and assumptions** required to successfully exploit the vulnerability. **Include also any assumption on how you must call and run the program:** e.g., the values for the command-line arguments required to trigger the exploit correctly and/or environment variables (if any), the input provided during the execution, if multiple executions are necessary.

[2 points] 2. **Focus only on the stack-based buffer overflow(s) you found.** Write an exploit for this vulnerability that **must execute the following shellcode**, composed of 8 bytes, which opens a shell: **0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38**.

2.a) Describe **all the steps and assumptions** required to successfully exploit the vulnerability. **Include also any assumption on how you must call and run the program:** e.g., the values for the command-line arguments required to trigger the exploit correctly and/or environment variables (if any), the input provided during the execution, if multiple executions are necessary.

- *Insert the (reversed) exploit (NOP+shellcode) in buf that must end with :ICNIVAD (when reversed DAVINCI:)*
- *select 'r'*
- *overflow prefix to overwrite sEIP with addr of ~buf+12*

ENV VARIABLE are cleaned at each execution -> placing shellcode in the env is not feasible

ALT SOL: directly exploit prefix

2.b) Make sure that you show how the exploit will appear in the process memory with respect to the stack layout right **before and after the execution of the vulnerable line** during the program exploitation showing:

- Direction of growth and high-low addresses;
- The name of each allocated variable;
- The content of relevant registers (i.e., EBP, ESP);
- The functions stack frames.

2.b) Make sure that you show how the exploit will appear in the process memory with respect to the stack layout right **before and after the execution of the vulnerable line** during the program exploitation showing:

- Direction of growth and high-low addresses;
- The name of each allocated variable;
- The content of relevant registers (i.e., EBP, ESP);
- The functions stack frames.

Before the vulnerable line				After the vulnerable line		ALTERNATIVE
Low addresses						
^			Function frames			
	prefix[0-3]	<- ESP	reverse_string()	not_relevant		0x20 0x30 0x40 0x50 <-X
	prefix[4-7]		reverse_string()	not_relevant		0x60 0x70 0x80 0x90
	sEBP	<- EBP	main()	not_relevant		not_relevant
	sEIP		main()	~X		~X
	&data		main()	not_relevant		not_relevant
	buf[0-3]		main()	not_relevant		not_relevant
	buf[4-7]		main()	not_relevant		not_relevant
	buf[8-11]		main()	not_relevant		not_relevant
	buf[12-15]		main()	not_relevant		not_relevant
	buf[16-19]		main()	not_relevant		not_relevant
	buf[20-23]		main()	not_relevant		not_relevant
	shellcode		main()	shellcode		not_relevant
	reverse		main()	reverse	<-X	not_relevant
	:ICN		main()	:ICN		:ICN
	IVAD		main()	IVAD		IVAD
	sel (r)		main()	sel (r)		sel (r)
	sEBP		main()	not_relevant		not_relevant
	sEIP			not_relevant		not_relevant
	argc			not_relevant		not_relevant
	argv			not_relevant		not_relevant
High addresses						

3.a) Write an exploit for this vulnerability that executes the previous shellcode, assuming that you have already prepared the memory (the shellcode has been positioned in a place under the control of the attacker) with the correct arguments. Assume that:

- The address of the return address (*saved EIP*) of the function exploited is:
0x8da0fee4 (i.e., where to write)
- The address of the first instruction of the **shellcode** is at **0xd3f4e2d0** (i.e., what to write)
- The displacement on the stack of the vulnerable function's argument is: **7**

Knowing that $dec(0xd3f4) = 54260$ and $dec(0xe2d0) = 58064$, write the exploit clearly, describe **all the steps, assumptions and the components of the format string** required to successfully exploit the vulnerability. Include also any assumption on how you must call and run the program: e.g., the values for the command-line arguments required to trigger the exploit correctly and/or environment variables (if any), the input provided during the execution.

- *Insert the (reversed) FS exploit in buf and must end with :ICNIVAD (when reversed DAVINCI:)*
- *select 'r'*
- *Exploit executed to overwrite sEIP with addr of shellcode*
- *EXPLOIT AS SEEN AT LECTURE -> + 8 (DAVINCI:) char already printed!*

dec(0xd3f4) = 54260 < dec(0xe2d0)=58064

Swap required

**<target+2><target>%<lower_part-16>c%pos+2\$n<higher_part-low_part>c%pos+2+1\$n
+ [Substitute the values and make computation]**

3.b) Make sure that you show how the exploit will appear in the process memory with respect to the stack layout right **before and after the execution of the vulnerable line** during the program exploitation showing:

- Direction of growth and high-low addresses;
- The name of each allocated variable;
- The content of relevant registers (i.e., EBP, ESP);
- The functions stack frames.

Show also the content of the caller frame.

- the reversed FS exploit will fill more buffer cells

prefix[0-3]	not_relevant
prefix[4-7]	not_relevant
sEBP	not_relevant
sEIP	0xd3f4e2d0
&data	not_relevant
buf[0-3]	not_relevant
buf[4-7]	not_relevant
buf[8-11]	not_relevant
buf[12-15]	not_relevant
buf[16-19]	not_relevant
buf[20-23]	not_relevant
reverse	reverse
FS exploit	FS exploit
:ICN	:ICN
IVAD	IVAD
sel (r)	sel (r)
sEBP	not_relevant
sEIP	not_relevant
argc	not_relevant
argv	not_relevant

[1 point] 4. Consider ONLY the correctly implemented “Random Terminator Canary” mitigation technique (i.e, randomly generated at each execution with ‘\0’ in its value). Is it effective to prevent your exploit at point 2. And 3. ? If the mitigation technique is effective, explain why and describe how you would bypass the mitigation in order to **execute the same shellcode of the previous question.**

It is effective to prevent the exploit at point 2 but not the exploit at point 3.

+ motivation [see slides].

The only way to bypass the mitigation is to use the FS. With the BO it is not possible since it is limited to 16 bytes.

Consider the C program below.

Exercise 4

```
00 #include <stdio.h>
01 #include <stdlib.h>
02 #include <string.h>
03
04 int guess(char *user) {
05     struct {
06         int n;
07         char usr[17];
08         char buf[16];
09     } s;
10     snprintf(s.usr, 16, "%s", user);
11     do{
12         scanf("%s", s.buf);
13         if (strncmp(s.buf, "DEBUG", 5) == 0) {
14             scanf("%d", &s.n);
15             for(int i = 0; i < s.n; i++) {
16                 printf("%x", s.buf[i]);
17             } else {
18                 if(strncmp(s.buf, "pass", 4) != 0 {
19                     abort();
20                 } else {
21                     if(s.usr[0] == '_'){
22                         return 1;
23                     } else {
24                         printf("Sorry User: ");
25                         printf(s.usr);
26                         printf("\nThe secret is wrong! \n");
27                         return 0;
28                     }
29                 }
30             } while(strncmp(s.buf, "DEBUG", 5) == 0);
31     }
32     int main(int argc, char** argv) {
33         guess(argv[1]);
34     }
```

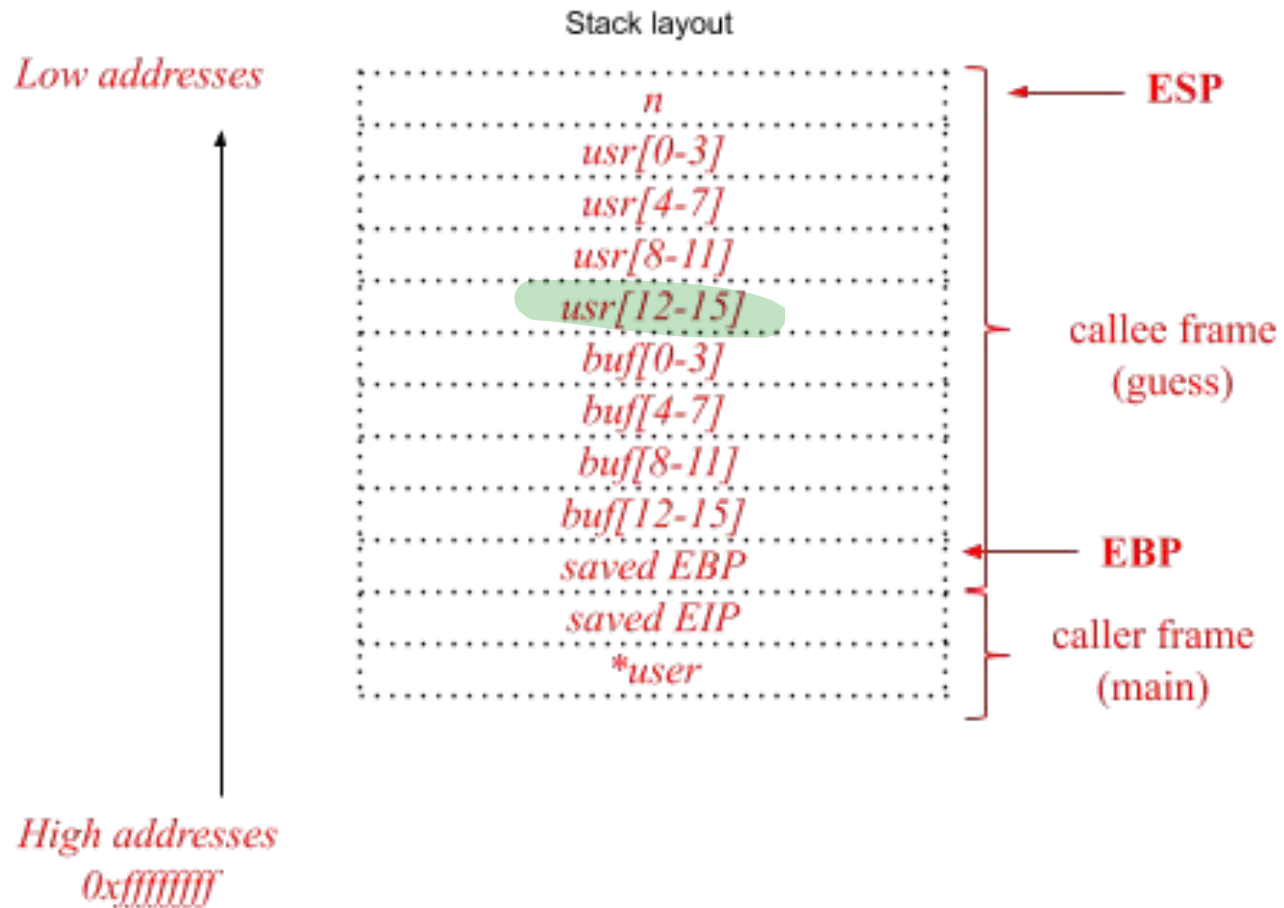
1. Assuming that the program is compiled and run for the usual IA-32 architecture (32-bits), with the usual cdecl calling convention, draw the stack layout just before the execution of line 11 showing:

- Direction of growth and high-low addresses;
- The name of each allocated variable;
- The boundaries of the function stack frames (main and **guess**)

Show also the content of the caller frame (you can ignore the environment variables: just focus on what matters for the exploitation of typical memory corruption vulnerabilities).

Assume that the program has been properly invoked with a single command line argument.

[illegible]



2. The program is affected by a typical buffer overflow and a format string vulnerability. Complete the following table, focusing on a vulnerability per row.

Vulnerability	Line	Motivation
Buffer Overflow		
Format String		

2. The program is affected by a typical buffer overflow and a format string vulnerability. Complete the following table, focusing on a vulnerability per row.

Vulnerability	Line	Motivation
Buffer Overflow	<i>Line 12</i>	<i>the scanf reads an user-supplied string of arbitrary length and copies it in a stack buffer</i>
Format String	<i>Line 25</i>	<i>printf(s.usr) where the format string, s.usr, is directly supplied by the (untrusted) user from CLI argument.</i>

3. [3 points] Assume that the program is compiled and run with no mitigation against exploitation of memory corruption vulnerabilities (**no canary, executable stack, environment with no ASLR** active).

Focus on the buffer overflow vulnerability. Write an exploit for the buffer overflow vulnerability in the above program to execute the following simple shellcode, composed only by 4 instructions (4 bytes): **0x58 0x5b 0x5a 0xc3**.

Make sure that you show how the exploit will appear in the process memory with respect to the stack layout right before and after the execution of the detected vulnerable line during the program exploitation.

Ensure you include all of the steps of the exploit, ensuring that the program and the exploit execute successfully. Include also any assumption on how you must call the program (e.g., the values for the command-line arguments required to trigger the exploit correctly).

[illegible][illegible]

Low addresses

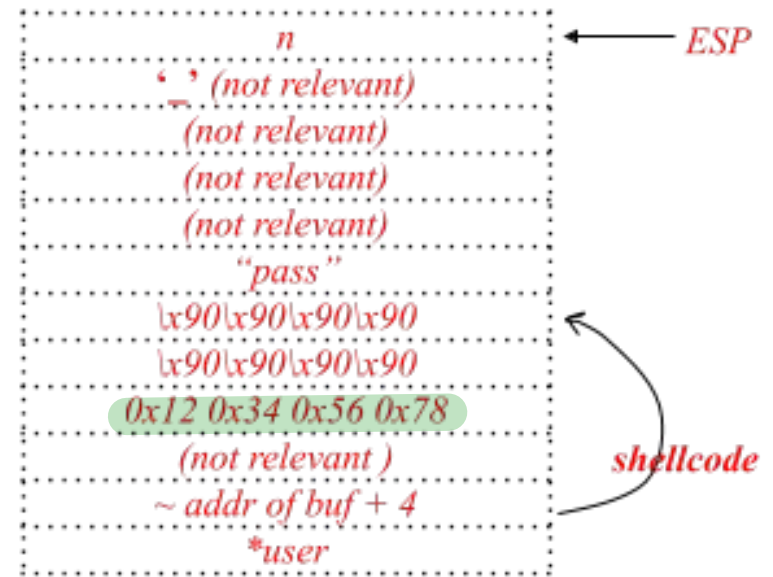
Stack layout before vulnerable line



High addresses

`0xffffffff`

Stack layout after vulnerable line



4. Now focus on the format string vulnerability you identified. We want to exploit this vulnerability to overwrite the saved EIP with the address of the environment variable \$EGG that contain your executable shellcode. Assuming we know that:

- The address of \$EGG (i.e., what to write) is $0x44674234$ ($0x4467$ (hex) = 17511 (dec), $0x4234$ (hex) = 16948 (dec))
- The target address of the EIP (i.e., the address where we want to write) is $0x42414515$
- The displacement on the stack of your format string is equal to 7

write an exploit for the format string vulnerability to execute the shellcode in the environment variable EGG.

Write the exploit clearly, detailing all the components of the format string, and detailing all the steps that lead to a successful exploitation.

The command line argument is directly fed as a format string to snprintf, and the format string itself is on the stack with a given displacement: we just need to construct a standard format string exploit.

- We need to write 0x44674234 to 0x42414515(<tgt>)*
- 0x4467 > 0x4234 -> we write 0x4234 first*

The value of the command line argument will have the form

<tgt><tgt+2>%<N1>c%<pos>\$hn%<N2>c%<pos+1>\$hn

where

<tgt> = 0x42414515

<tgt+2> = 0x42414517

<pos> = 7

<N1> = dec(0x4234) - 8 (bytes already written) = 16948 - 8 = 16940

<N2> = dec(0x4467) - 16948 (bytes already written) = 17511 - 16948 = 563

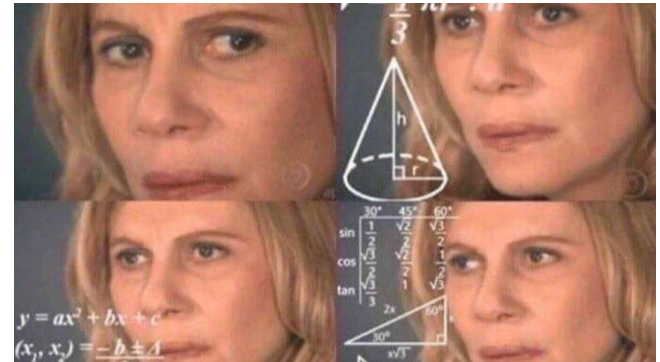
thus the final string is

\x15\x45\x41\x42\x17\x45\x41\x42%16940c%7\$hn%563c%8\$hn

Consider the C program below.

```
00 #include <stdio.h>
01 #include <stdlib.h>
02 #include <string.h>
03
04 int guess(char *user) {
05     struct {
06         int n;
07         char usr[17];
08         char buf[16];
09     } s;
10     snprintf(s.usr, 16, "%s", user);
11     do{
12         scanf("%s", s.buf);
13         if (strncmp(s.buf, "DEBUG", 5) == 0) {
14             scanf("%d", &s.n);
15             for(int i = 0; i < s.n; i++) {
16                 printf("%x", s.buf[i]);
17             } else {
18                 if(strncmp(s.buf, "pass", 4) != 0 {
19                     abort();}
20                 else {
21                     if(s.usr[0] == '_'){
22                         return 1;
23                     } else {
24                         printf("Sorry User: ");
25                         printf(s.usr);
26                         printf("\nThe secret is wrong! \n");
27                         return 0;}
28                 }
29             }
30     } while(strncmp(s.buf, "DEBUG", 5) == 0);
31 }
32 int main(int argc, char** argv) {
33     guess(argv[1]);
34 }
```

Warning !!!



4. Now consider that the program is compiled enabling the exploit mitigation technique known as stack canary. Assume that the compiler and the runtime correctly implement this technique with a random value changed at every program execution. Are the two exploits you wrote still working *without modifications*? Why? If not working modify the exploit so that it works reliably in this setting (i.e., enabled stack canaries). Please describe precisely how you would modify the above exploit, and any further assumption you need to make it work. If your exploit needs to perform multiple iteration of any loop, please describe what you would write to the standard input at every iteration of the loop(s). If the exploit is working without modification, please describe precisely why the vulnerability is still exploitable.

First exploit

Second exploit

First exploit

No. the first exploit would overwrite the stack canary (placed in the stack before the return address). The code placed by the compiler in the function epilogue would detect the canary overwrite and abort the execution of the program without jumping to the saved return address, and without executing the attacker's shellcode. We need to make the program perform two iterations of the loop.

First iteration: write an argv[1] that starts with '_' (e.g., '_dbg!\n') and 'DEBUG' at line 16; the code at lines 17-21 will print n words from the stack. Assuming that n is sufficiently high to print also the word containing the stack canary (or, if the attacker controls the command line arguments, passing a sufficiently high n), this would leak the stack canary value for that specific execution of the program.

Then, as `strncmp(s.buf, "DEBUG", 5) == 0`, the `do...while` loop will not exit.

Second iteration: at this point, we know the value of the stack canary (read during the first loop iteration). We write to stdin (i.e., we put in the buffer) the same exploit I wrote at point 2, changing it slightly so that the stack canary gets overwritten by the value leaked during the first iteration. Also, we need to ensure that `strncmp(s.buf, "DEBUG", 5) == 1` so that the loop ends, the function returns, and the shellcode is executed.

Second exploit

Yes, the second exploit still work without modification...