Exercises
Memory Errors: Buffer Overflow - Format String

Computer Security

Question

You are given two programs to analyze. Both are supposed acquire a string from the command line and echo it to the standard output.

Program 1	**Program 2**
`int main (int argc, char* argv[])`	`int main (int argc, char* argv[])`
`{`	`{`
`  char dst[256];`	`  char dst[256];`
`  sprintf(dst, argv[1]);`	`  snprintf(dst, 250, argv[1]);`
`  printf("%s", dst);`	`  printf(dst);`
`  return 0;`	`  return 0;`
`}`	`}`

Question

Program 1	**Program 2**
`int main (int argc, char* argv[])`	`int main (int argc, char* argv[])`
`{`	`{`
`  char dst[256];`	`  char dst[256];`
`  sprintf(dst, argv[1]);`	`  snprintf(dst, 250, argv[1]);`
`  printf("%s", dst);`	`  printf(dst);`
`  return 0;`	`  return 0;`
`}`	`}`

1) Focus on **Program 1** and explain which vulnerability/ies you find, **<u>describing how it/they work(s)</u>**.

Question

Program 1

```
int main (int argc, char* argv[])
{
  char dst[256];
  sprintf(dst, argv[1]);
  printf("%s", dst);

  return 0;
}
```

Program 2

```
int main (int argc, char* argv[])
{
  char dst[256];
  snprintf(dst, 250, argv[1]);
  printf(dst);

  return 0;
}
```

1) Focus on **Program 1** and explain which vulnerability/ies you find, **describing how it/they work(s)**.

There is a buffer overflow in the sprintf(). This allows an attacker that can control the argv[1] buffer to overwrite values on the main stack frame, leading to execution of arbitrary code. There is also a format string vulnerability in the same line.

Question

Program 1

```
int main (int argc, char* argv[])
{
  char dst[256];
  sprintf(dst, argv[1]);
  printf("%s", dst);

  return 0;
}
```

Program 2

```
int main (int argc, char* argv[])
{
  char dst[256];
  snprintf(dst, 250, argv[1]);
  printf(dst);

  return 0;
}
```

2) Now focus on **Program 2** and explain how the proposed fix prevents the vulnerability/ies you found.

Question

Program 1

```c
int main (int argc, char* argv[])
{
  char dst[256];
  sprintf(dst, argv[1]);
  printf("%s", dst);

  return 0;
}
```

Program 2

```c
int main (int argc, char* argv[])
{
  char dst[256];
  snprintf(dst, 250, argv[1]);
  printf(dst);

  return 0;
}
```

2) Now focus on **Program 2** and explain how the proposed fix prevents the vulnerability/ies you found.

The secure version of sprintf(), snprintf(), is used: it copies up to 250 bytes (which is below the size of the dst buffer).

Question

Program 1

```
int main (int argc, char* argv[])
{
  char dst[256];
  sprintf(dst, argv[1]);
  printf("%s", dst);

  return 0;
}
```

Program 2

```
int main (int argc, char* argv[])
{
  char dst[256];
  snprintf(dst, 250, argv[1]);
  printf(dst);

  return 0;
}
```

3) Is the fix in **Program 2** correct? Elaborate your answer in either case.

Question

Program 1	**Program 2**
`int main (int argc, char* argv[])`	`int main (int argc, char* argv[])`
`{`	`{`
`  char dst[256];`	`  char dst[256];`
`  sprintf(dst, argv[1]);`	`  snprintf(dst, 250, argv[1]);`
`  printf("%s", dst);`	`  printf(dst);`
`  return 0;`	`  return 0;`
`}`	`}`

3) Is the fix in **Program 2** correct? Elaborate your answer in either case.

No, the format string vulnerability is still present. Also, a further format string vulnerability has been introduced in the next line.

Exercise 1

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

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

void vuln() {
  char buf[32];
  /*              <---------------- here       */
  scanf("%s", buf);
  if (strncmp(buf, "Knight_King!", 12) != 0) {
     abort();
  }
}

int main(int argc, char** argv) {
  vuln();
}
```

Exercise 1

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** (ARSL 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:

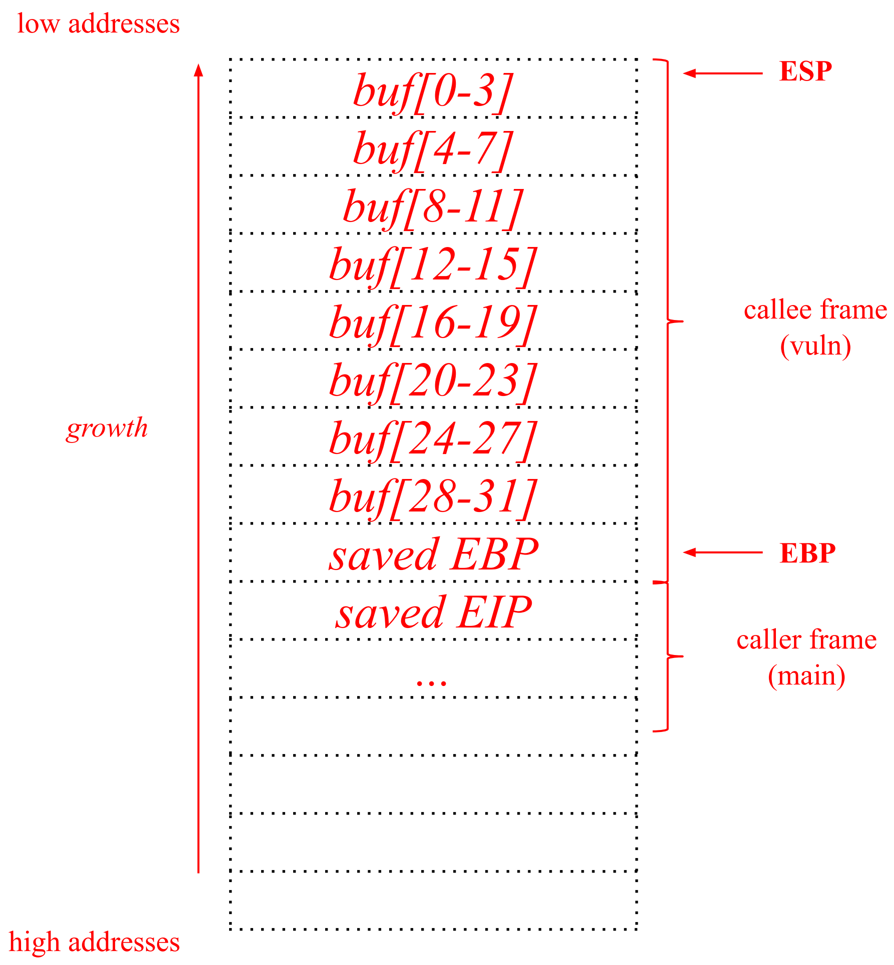
a. Direction of growth and high-low addresses.
b. The name of each allocated variable.
c. The boundaries of frame of the function frames (main and vuln).

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

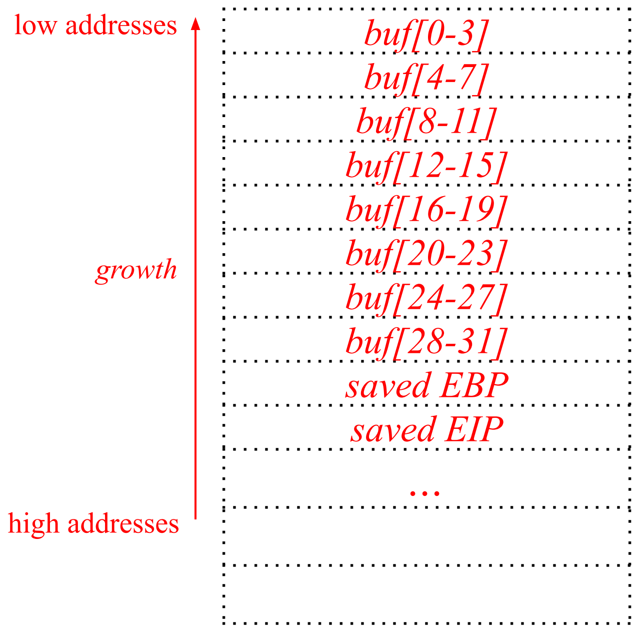
a. Direction of growth and high-low addresses.
b. The name of each allocated variable.
c. The boundaries of frame of the function frames (main and vuln).

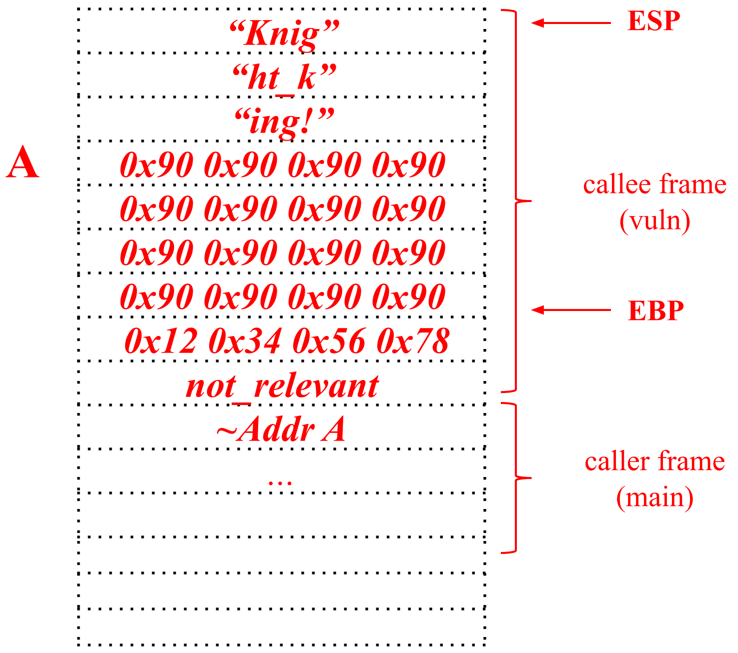
Show also the content of the caller frame (you can ignore the environment variables, just focus on what matters for the vulnerability and its exploitation).


low addresses
buf[0-3]
buf[4-7]
buf[8-11]
buf[12-15]
buf[16-19]
buf[20-23]
buf[24-27]
buf[28-31]
saved EBP
saved EIP
...
ESP
EBP
callee frame
(vuln)
caller frame
(main)
growth
high addresses

2. [3 points] 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.

2. [3 points] 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. [1 point] If **address space layout randomization (ASLR)** is active, is the exploit you just wrote still working **without modifications**? Why?

3. [1 point] 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. [1 point] 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. [1 point] 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 change the value of **B** and let it point to the **address of system**.*

Exercise 2

BlaBlaStack is a new stack protection mechanism that is designed to protect against local attackers (e.g., users that can have a shell and launch binaries).

BlaBlaStack works by inserting instructions in the prologue that push a value to the stack. This value is generated at compile time and is hardcoded in the binary.

Then, during the epilogue, this value is popped from the stack and an appropriate instruction compares it with the hardcoded, expected one.

In case of match, the program goes on, otherwise it is aborted.

- Why this mechanism does not provide proper protection against stack overflows?
- How would you fix *BlaBlaStack* in order to protect from local attackers?

This is a compiler based protection mechanism, it is a variant of the stack canary mechanism seen in class. The attacker can read the binary, for example using a disassembler, obtain the hardcoded value and include it as part of the exploit.

Instead of hardcoding the canary value, the instructions in the prologue should generate a random value and push it on a general purpose register. During the epilogue, the register is used to compare the valued popped from the stack.

```
function_prologue:
    pushl $0x0000d00d
    pushl %ebp
    mov %esp, %ebp
    subl $4, %esp

function_foo:
    (functionbody)

function_epilogue:
    movl %ebp, %esp
    popl %ebp
    cmpl
$0x0000d00d,(%esp)
    jne function_bar
    addl $4, %esp
    ret

function_bar:
    (functionbody)
```

Consider the following assembly code

- (2 points) Describe in detail what the assembly code does and the name of the technique that is being implemented.
- (1 point) Describe the weakness in this implementation.
- (1 point) Describe, in common words, how would you fix the vulnerability.

Exercise 3

Consider the C program below:

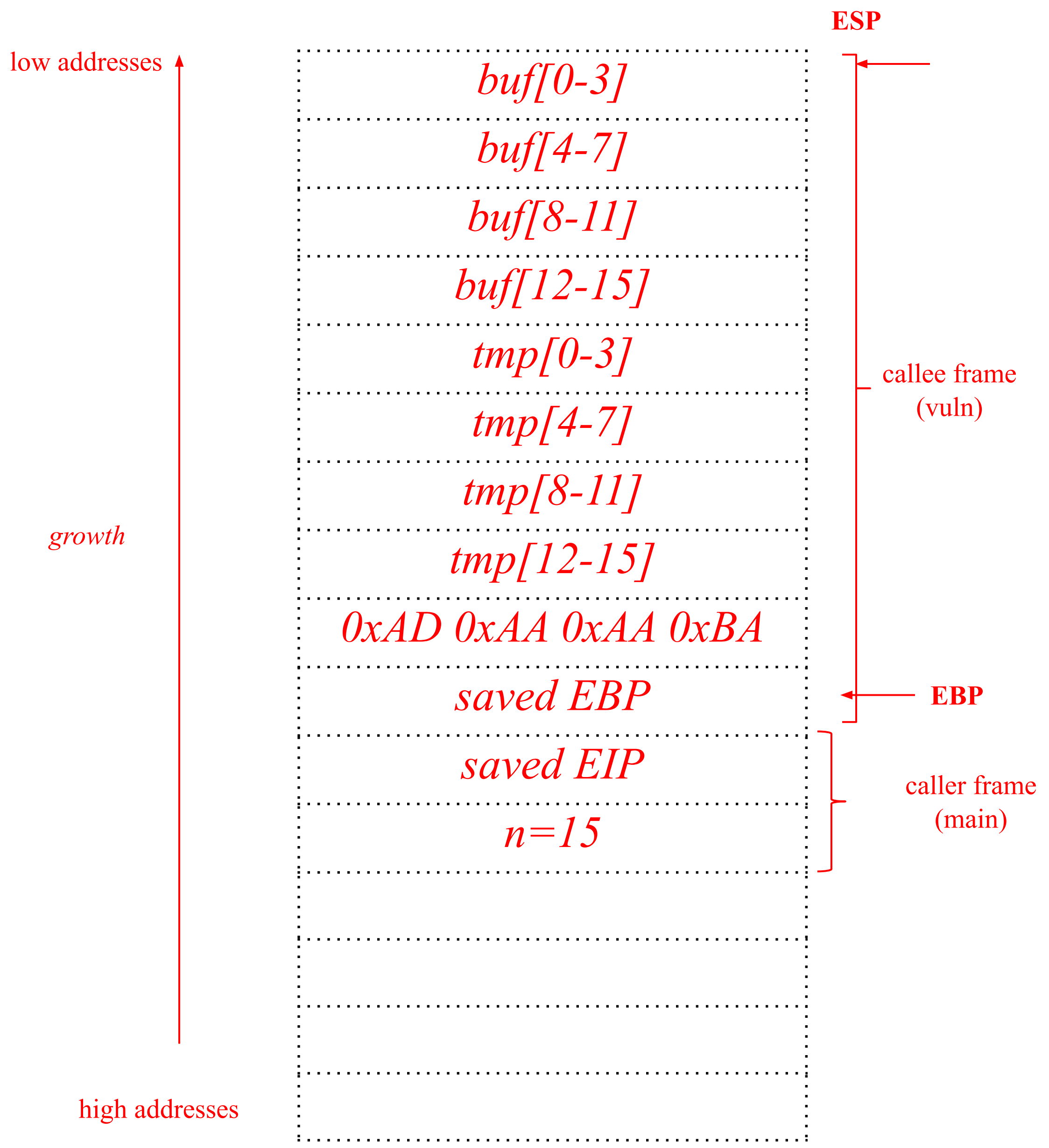
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void vuln(int n) {
    struct {
        char buf[16];
        char tmp[16];
        int sparrow = 0xBAAAAAAD;
    } s;

   if (n > 0 && n < 16) {
         fgets(s.buf,n,stdin);
         if(strncmp(s.buf, "H4CK", 4) != 0 && s.buf[14] != “X”) {
               abort();
         }
         scanf("%s", s.tmp);
         if(s.sparrow != 0xBAAAAAAD) {
               printf("Goodbye!\n");
               abort();
         }
   }
}

int main(int argc, char** argv) {
   vuln(argc);
   return 0;
}
```

1. [1 point] 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).


ESP
low addresses
buf[0-3]
buf[4-7]
buf[8-11]
buf[12-15]
tmp[0-3]
tmp[4-7]
tmp[8-11]
tmp[12-15]
0xAD 0xAA 0xAA 0xBA
saved EBP
saved EIP
n=15
callee frame (vuln)
EBP
caller frame (main)
growth
high addresses

2. [1 point] The program is affected by a buffer overflow vulnerability. Complete the following table.

Vulnerability	**Line(s)**	**Motivation**
Buffer Overflow		

2. [1 point] The program is affected by a buffer overflow vulnerability. Complete the following table.

Vulnerability	**Line(s)**	**Motivation**
Buffer Overflow	17	*[See slides]*

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

3.a. [1 point] Shortly describe what is the implemented technique, and how it works in general.

3.b. [1 point] 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.

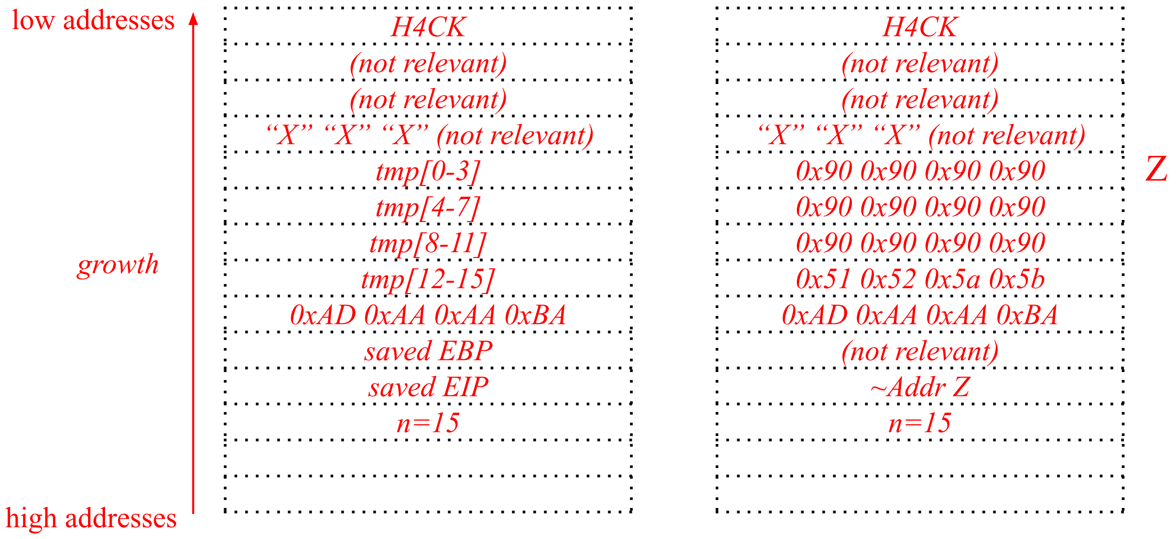
3.a. [1 point] Shortly describe what is the implemented technique, and how it works in general.

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. [1 point] 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. [2 points] 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).


low addresses
growth
high addresses
H4CK
(not relevant)
(not relevant)
"X" "X" "X" (not relevant)
tmp[0-3]
tmp[4-7]
tmp[8-11]
tmp[12-15]
0xAD 0xAA 0xAA 0xBA
saved EBP
saved EIP
n=15
H4CK
(not relevant)
(not relevant)
"X" "X" "X" (not relevant)
0x90 0x90 0x90 0x90
0x90 0x90 0x90 0x90
0x90 0x90 0x90 0x90
0x51 0x52 0x5a 0x5b
0xAD 0xAA 0xAA 0xBA
(not relevant)
~Addr Z
n=15
Z

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

5.a. [1 point] Is the exploit you just wrote still working? Why?

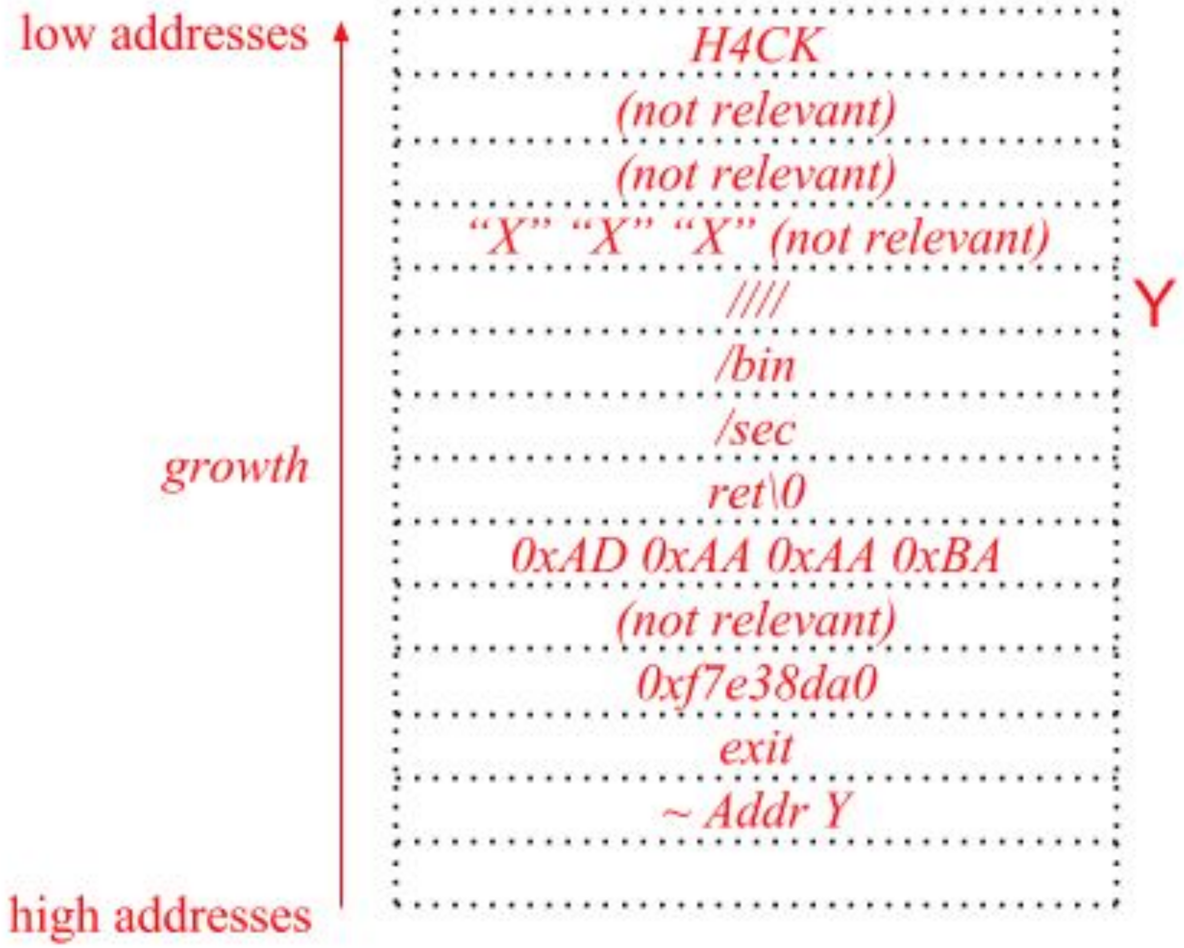
5. Assuming that W^X is enabled (i.e., non-executable stack):

5.a. [1 point] 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. [2 points] 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 system(). We then overwrite 8 bytes more for the system() EIP (to exit) and for the pointer to ~ Y so that, when jumping into the system() function, this pointer will be where system() expects the parameter and will complete the execution.



6. [1 points] Assuming that ASLR is enabled (i.e., randomized memory layout): Is the exploit you just wrote still working? Why?

6. [1 points] <u>Assuming that ASRL is enabled</u> (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. [2 point]. 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. [2 point]. 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*

Consider the C program below.

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

int guess(char *user) {
    struct {
        int n;
        char usr[16];
        char buf[16];
    } s;

    snprintf(s.usr, 16, "%s", user);

    do{
        scanf("%s", s.buf);
        if (strncmp(s.buf, "DEBUG", 5) == 0) {
            scanf("%d", &s.n);
            for(int i = 0; i < s.n; i++) {
                printf("%x", s.buf[i]);
            }
        } else {
            if(strncmp(s.buf, "pass", 4) == 0 && s.usr[0] == '_') {
                return 1;
            } else {
                printf("Sorry User: ");
                printf(s.usr);
                printf("\nThe secret is wrong! \n");
                abort();
            }
        }
    } while(strncmp(s.buf, "DEBUG", 5) == 0);
}

int main(int argc, char** argv) {
    guess(argv[1]);
}
```

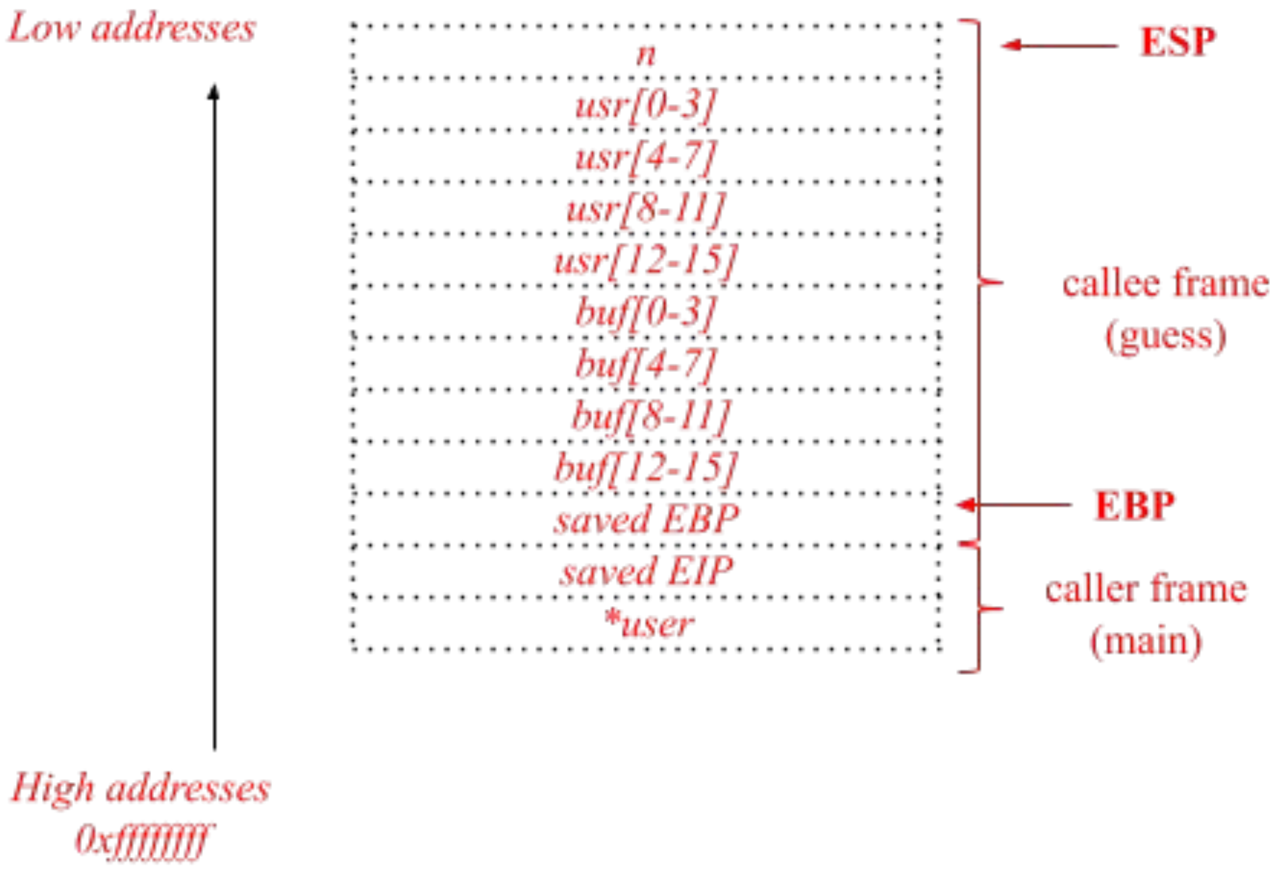
1. [2 points]. Assuming that the program is compiled and run for the usual IA-32 architecture (32-bits), with the usual cdecl calling convention, draw the stack layout just before the execution of line 11 showing:

- 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.

Stack layout



2. [1 points] 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		

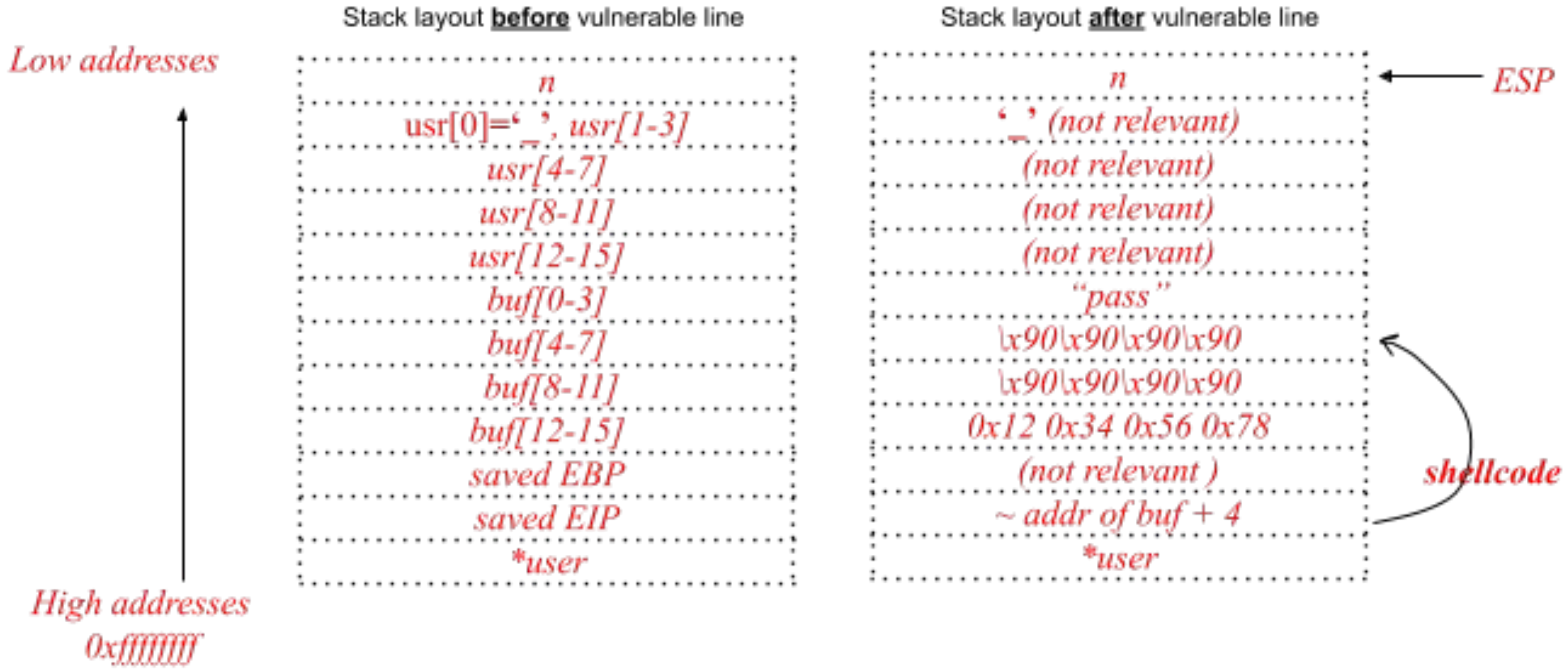
2. [1 points] The program is affected by a typical buffer overflow and a format string vulnerability. Complete the following table, focusing on a vulnerability per row.

Vulnerability	**Line**	**Motivation**
Buffer Overflow	*Line 15*	*the scanf reads an user-supplied string of arbitrary length and copies it in a stack buffer*
Format String	*Line 26*	*printf(s.usr) where the format string, s.usr, is directly supplied by the (untrusted) user from CLI argument.*

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).


Stack layout **before** vulnerable line
Stack layout **after** vulnerable line
Low addresses
High addresses
0xffffffff
n
usr[0]='_', usr[1-3]
usr[4-7]
usr[8-11]
usr[12-15]
buf[0-3]
buf[4-7]
buf[8-11]
buf[12-15]
saved EBP
saved EIP
*user
n
'_' (not relevant)
(not relevant)
(not relevant)
(not relevant)
"pass"
\x90\x90\x90\x90
\x90\x90\x90\x90
0x12 0x34 0x56 0x78
(not relevant)
~ addr of buf + 4
*user
ESP
shellcode

4. [2 points] 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
```

4. [2 points] 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 to the stdin '`_dbg!\n`*' as argv[1] 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 <u>same</u> 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...

The End