CYB2200 - Lab 2
Format String Attack

Format string attack lab

- **Due:** Friday, Sep 19th, 11:59 pm.
- **Turn in:** This lab report.
- **Points:** 30 points + 10 extra points
- **Objective:** Understand format string vulnerability (topic relates to Metacharacters in class). And launch a format string attack to see what value you can reveal in the program.

- Available code for this lab:
  - `vul_prog_easy.c` and `vul_prog_hard.c`

Note - before you start:

- For this lab, Do not modify the source code.
  - You only need to compile and execute the program with specific input, and you will be able to exploit the format string vulnerability.
- The invalid input for this lab is totally different from lab 1.
  - We are not using extra long input, or different types of input for this lab.
- You only need to turn in the lab report.
  - Make sure you answer the questions with details and good explanations, to prove you understand the format string vulnerability and attack.

Function and arguments

- Most of the functions take a fixed number of arguments.

- eg:
  - int sum(int var_1, int var_2);
  - double avg(double x, double y, double z);

printf() in C

- Some functions accept any number of arguments, they are called variadic function.

- E.g: printf()
  - printf(“hello world!\n”); ← 1 argument
  - printf ("a has value %d\n", a); ← 2 arguments
  - printf ("a has value %d, b has value %d\n", a, b); ← 3 arguments
  - printf ("a has value %d, b has value %d, c is at address: %x\n", a, b, &c);

    ↑ 4 arguments

printf function

- `int printf(const char *format,…)`
  - The … indicates that zero or more optional arguments can be provided when the function is invoked.

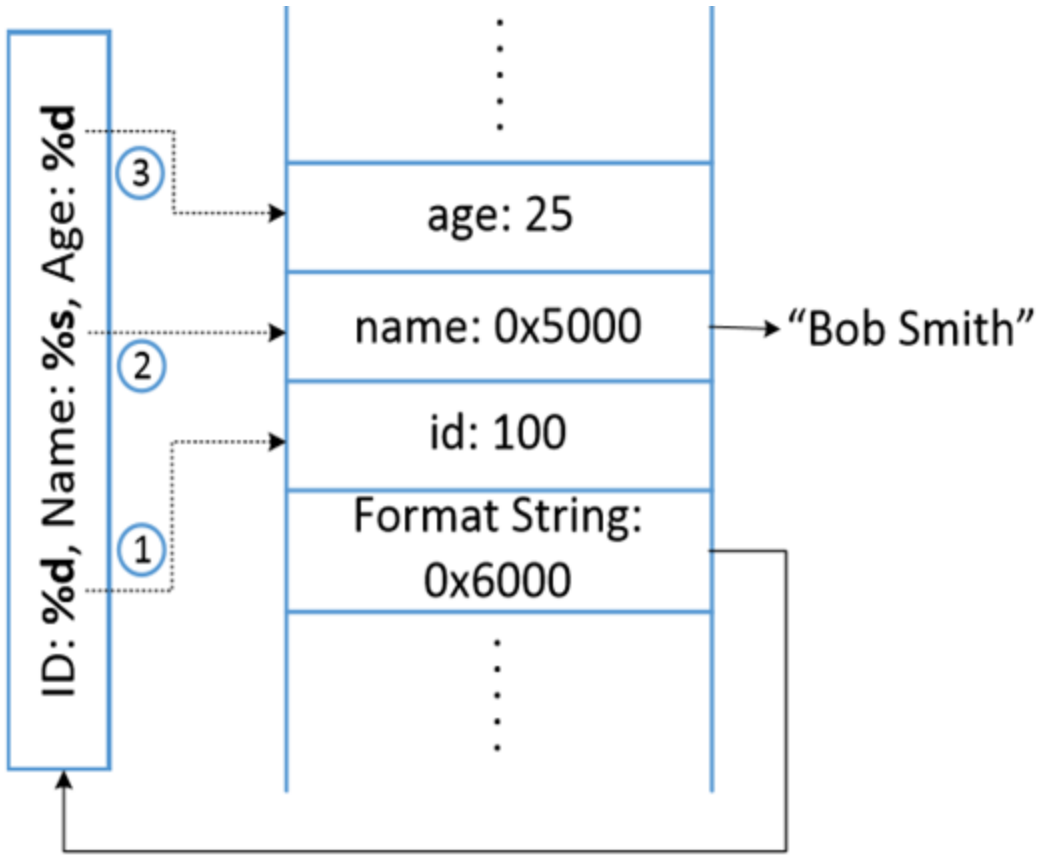
- **format string** – This is the string that contains the text to be written to stdout. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested.
- The printf function uses its first argument to determine how many arguments will follow and of what types they are.

Format tags/format specification

Parameter	Meaning	Passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

Format tags determine the output format.

Stack layout




```
int main()
{
   int id = 100;
   int age = 25;
   char *name = "Bob Smith";

    printf("ID: %d, Name: %s, Age: %d\n", id, name, age);
}
```

What if there is a mismatch?

What if there is a mismatch?

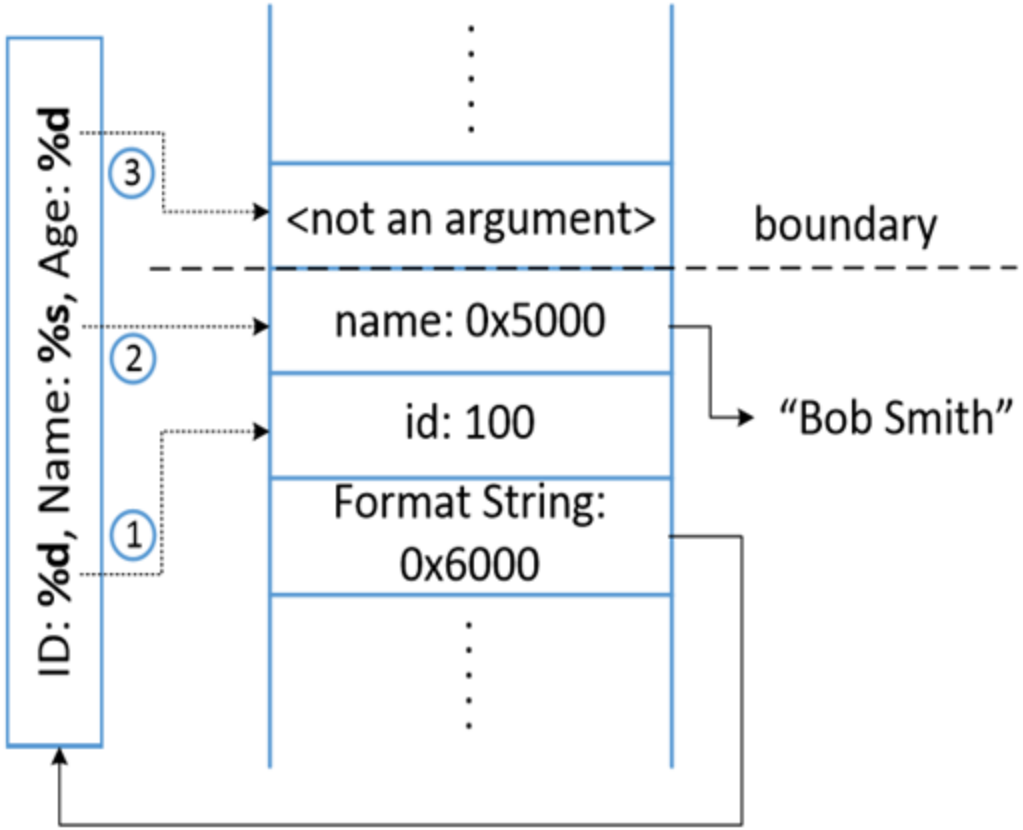
- For regular functions:
  - If a function's definition has three arguments, but two are passed to it during the invocation, compilers will catch this as an error.

- For printf():
  - Because it is defined in a special way (with …)
  - Compilers never complain about printf()
  - `printf("ID: %d, Name: %s, Age: %d\n", id, name);`
    - It continues fetching data from the stack!
  - This leads to the format string attack.

What if there is a mismatch?

```
int main()
{
   int id = 100;
   int age = 25;
   char *name = "Bob Smith";

   printf("ID: %d, Name: %s, Age: %d\n", id, name);
}
```



- Now take a look at the `vul_prog_easy.c`
- Understand the program
- Identify where the format string vulnerability is.

This is the second half of the vul_prog_easy.c

```
// decimal integer needed, be careful, not in hex
    printf("\nPlease enter a decimal integer\n");
    scanf("%d", &int_input[0]);  /* getting an input1 from user */
    printf("Please enter a decimal integer for integer2\n");
    scanf("%d", &int_input[1]);  /* getting an input2 from user */
    printf("Please enter a string\n");
    scanf("%s", user_input); /* getting a string from user */

    /* Vulnerable place */
    printf(user_input);
    printf("\n");

    return 0;
}
```

"Today is Friday!"

"Today is %s Friday %d"

Tasks 1 - crash the program

- Please use `vul_prog_easy.c` for the following tasks:
- (8 points) Task 1 - crash the program
  - Note: The crash in this lab should be triggered by exploiting the format string vulnerability. Specifically, the `print(user_input);` (on line 42). The crash should not be caused by using very long input or other invalid input we tried in lab 1.
  - [Screenshot here] to show your program crashed.
    - In addition to the program output. Make sure the screenshot shows the input/commands you typed.
  - Please explain in detail why the program crashed and which part of the user input triggered the crash.

Task 1: Crash the Program

- %s%s%s%s…

```
$ ./vul
......
Please enter a string: %s%s%s%s%s%s%s%s
Segmentation fault (core dumped)
```

- For each `%s`, it fetches a value from the stack.
- `%s` – get the data stored at the address.
- As we give `%s`, `printf()` treats the value as address and fetches data from that address. If the value is not a valid address, the program crashes.

Task 2 - print out 20 values on the stack

- %s – get the data stored at the address.
- %x – print integer value in hex.

- Use user input: `%x%x%x%x%x%x%x%x%x%x%x%x`

```
$ ./vul
......
Please enter a string: %x.%x.%x.%x.%x.%x.%x.%x
63.b7fc5ac0.b7eb8309.bffff33f.11223344.252e7825.78252e78.2e78252e
```

Take a look at the values on the stack. Any value you recognize?

Task 3

- Task 3 – Format string attack: Use a format tag to print out at least one secret value. The screenshot should have the letter ‘C,’ a letter ‘S,’ or both letters ‘C’ and ‘S’ shown.
  - [Screenshot here]
  - Please explain what you did and why.

See next slide for an example output.

Task 4 – extra credits

This take is not required.

Please use `vul_prog_hard.c` for this task!

Hint: This program uses dynamic memory, memory allocated with malloc(), which is stored on the heap, not on the stack. Exploiting a format string vulnerability only shows the values on the stack.

- (10 pts) Task 4 – Format string attack: Use format tags to print out at least one secret value.

  Assume you don't know the two letters. Manually writing the value to the stack doesn't count.

  - (5 pts) The screenshot shows the letter 'E'
  - (5 pts) The screenshot shows both letters 'E' and 'V'.

e.g.

```
34c0.6b6e34b8.6b6e34e0.471ff24.6b6e34f4.E.444.222.6b6e37d0.1.V.80004
```