Format String vulnerability and attack

TCS 591 Unit 2

What Are Format String Vulnerabilities ?

- Format strings are used in many programming languages to insert values into a text string. In some cases, this mechanism can be abused to perform buffer overflow attacks, extract information or execute arbitrary code
- The Format String exploit occurs when the submitted data of an input string is evaluated as a command by the application.
- In this way, the attacker could execute code, read the stack, or cause a segmentation fault in the running application, causing new behaviours that could compromise the security or the stability of the system

What Are Format String Vulnerabilities ?

- Format String attacks alter the flow of an application. They use string formatting library features to access other memory space.

- Vulnerabilities occurred when the user-supplied data is deployed directly as formatting string input for certain C/C++ functions (e.g., fprintf, printf, sprintf, setproctitle, syslog, ...).

- Format String attacks are related to other attacks in the Threat Classification: Buffer Overflows and Integer Overflows.

- All three are based on their ability to manipulate memory or its interpretation in a way that contributes to an attacker's goal.

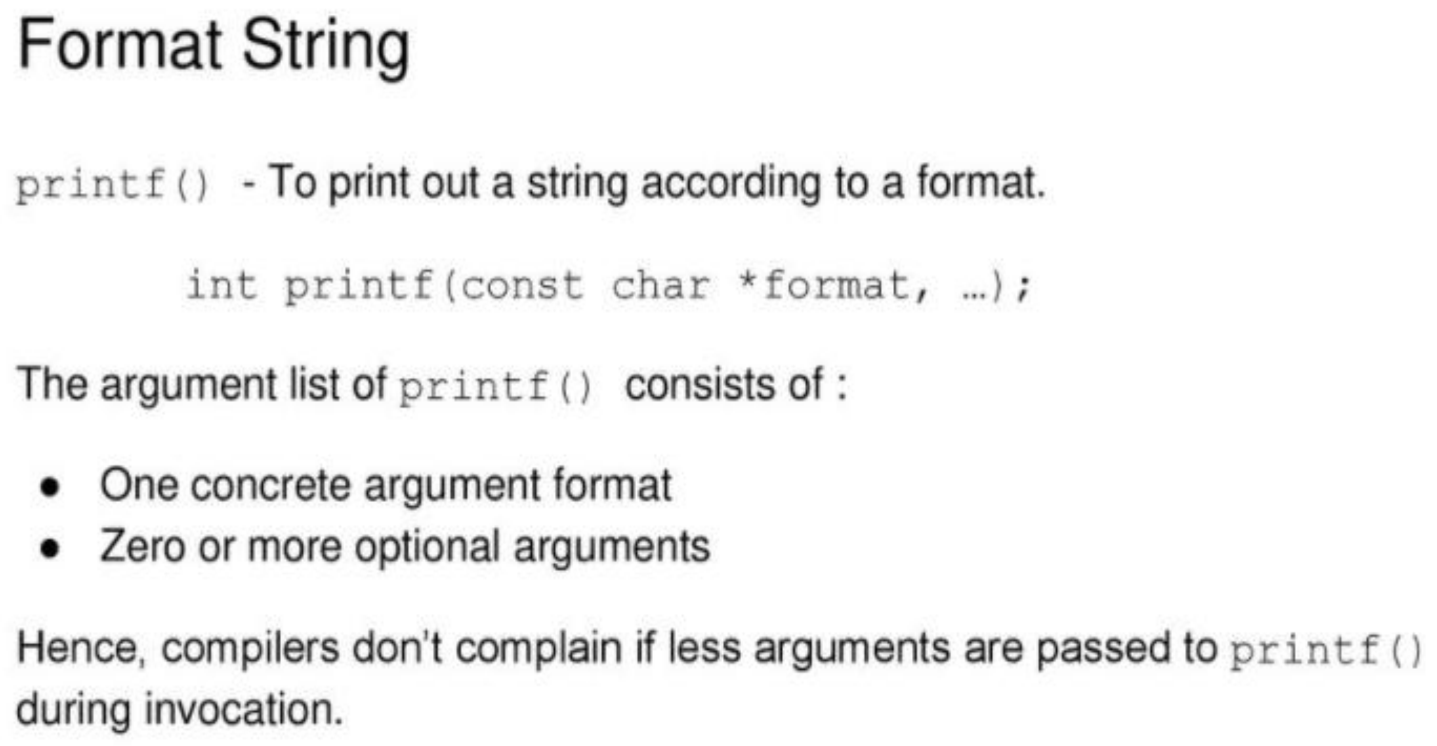
What Are Format String Vulnerabilities ?

- To understand the attack, it’s necessary to understand the components that constitute it.

The **Format Function** is an ANSI C conversion function, like **printf, fprintf**, which converts a primitive variable of the programming language into a human-readable string representation.

The **Format String Parameter**, like **%x %s** defines the type of conversion of the format function.

Format String



Format String

`printf()` - To print out a string according to a format.

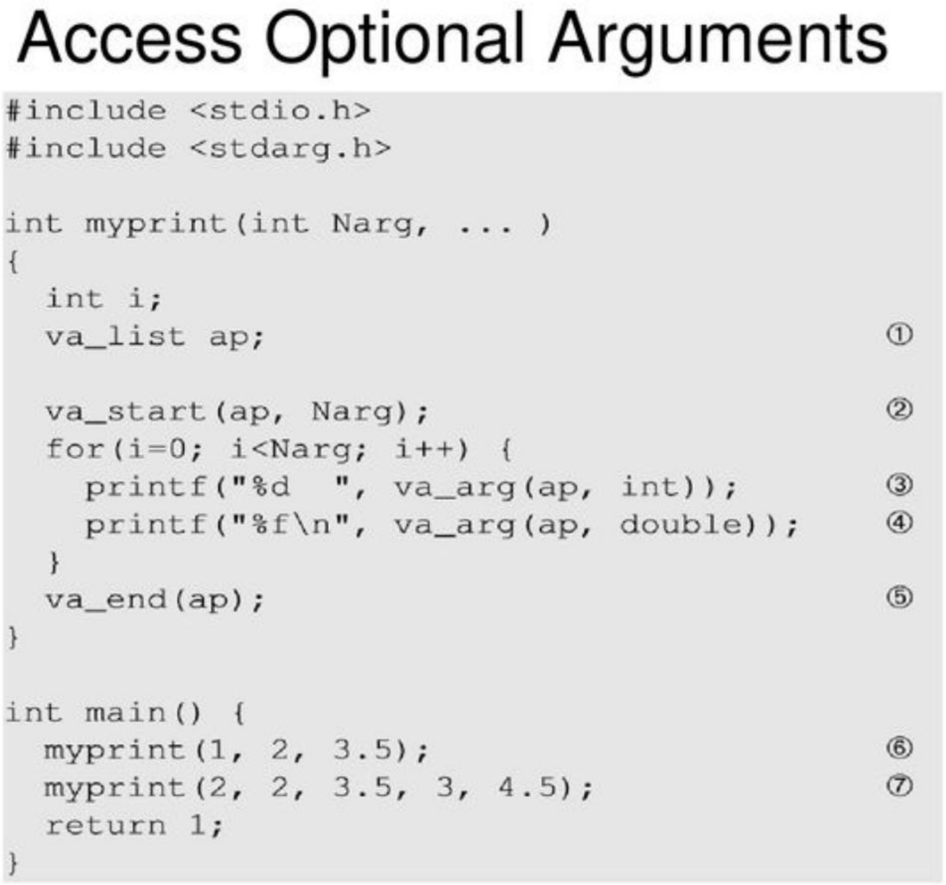
```
        int printf(const char *format, …);
```

The argument list of `printf()` consists of :

- One concrete argument format
- Zero or more optional arguments

Hence, compilers don't complain if less arguments are passed to `printf()` during invocation.

Example

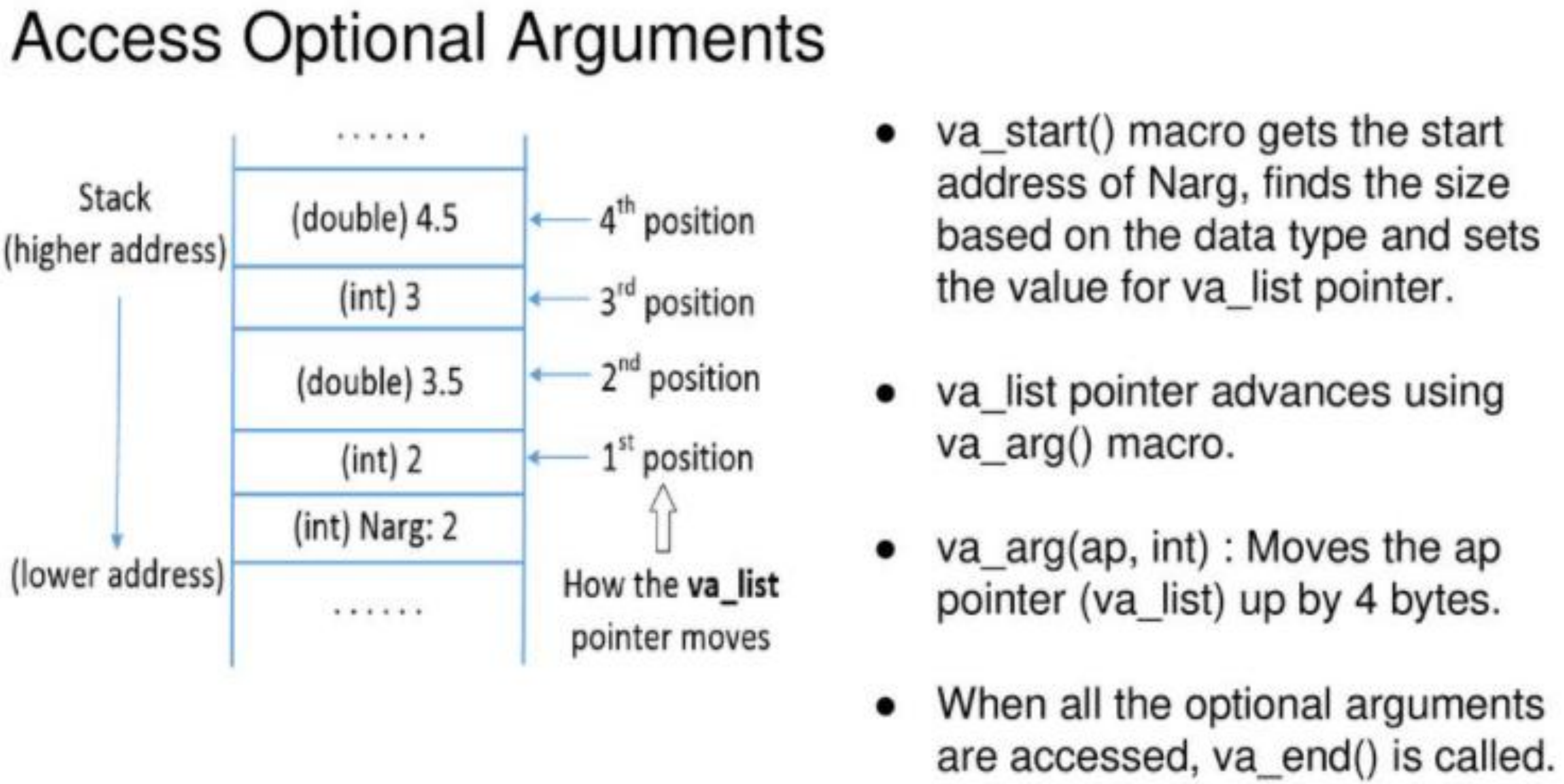


Access Optional Arguments

```
#include <stdio.h>
#include <stdarg.h>

int myprint(int Narg, ... )
{
  int i;
  va_list ap;                                   ①

  va_start(ap, Narg);                           ②
  for(i=0; i<Narg; i++) {
    printf("%d  ", va_arg(ap, int));            ③
    printf("%f\n", va_arg(ap, double));         ④
  }
  va_end(ap);                                   ⑤
}

int main() {
  myprint(1, 2, 3.5);                           ⑥
  myprint(2, 2, 3.5, 3, 4.5);                   ⑦
  return 1;
}
```

- myprint() shows how printf() actually works.
- Consider myprintf() is invoked in line 7.
- va_list pointer (line 1) accesses the optional arguments.
- va_start() macro (line 2) calculates the initial position of va_list based on the second argument Narg (last argument before the optional arguments begin)

Example



Access Optional Arguments

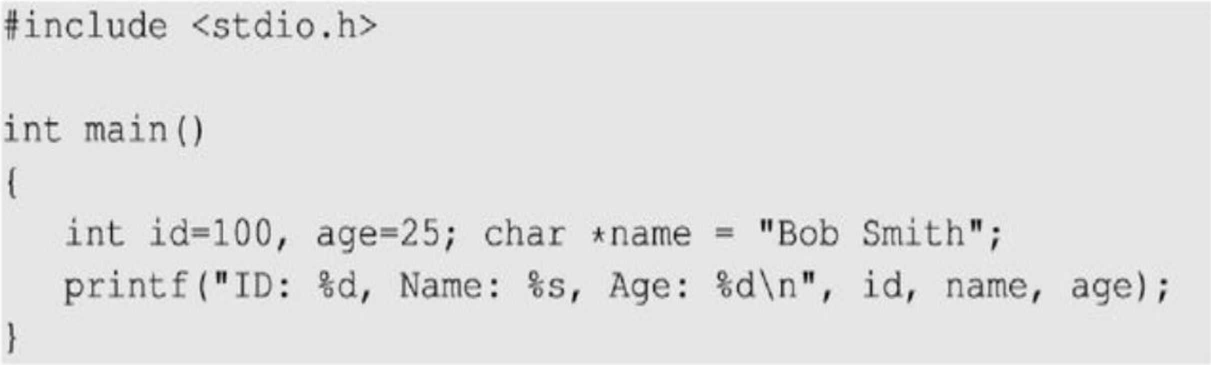
- va_start() macro gets the start address of Narg, finds the size based on the data type and sets the value for va_list pointer.

- va_list pointer advances using va_arg() macro.

- va_arg(ap, int) : Moves the ap pointer (va_list) up by 4 bytes.

- When all the optional arguments are accessed, va_end() is called.

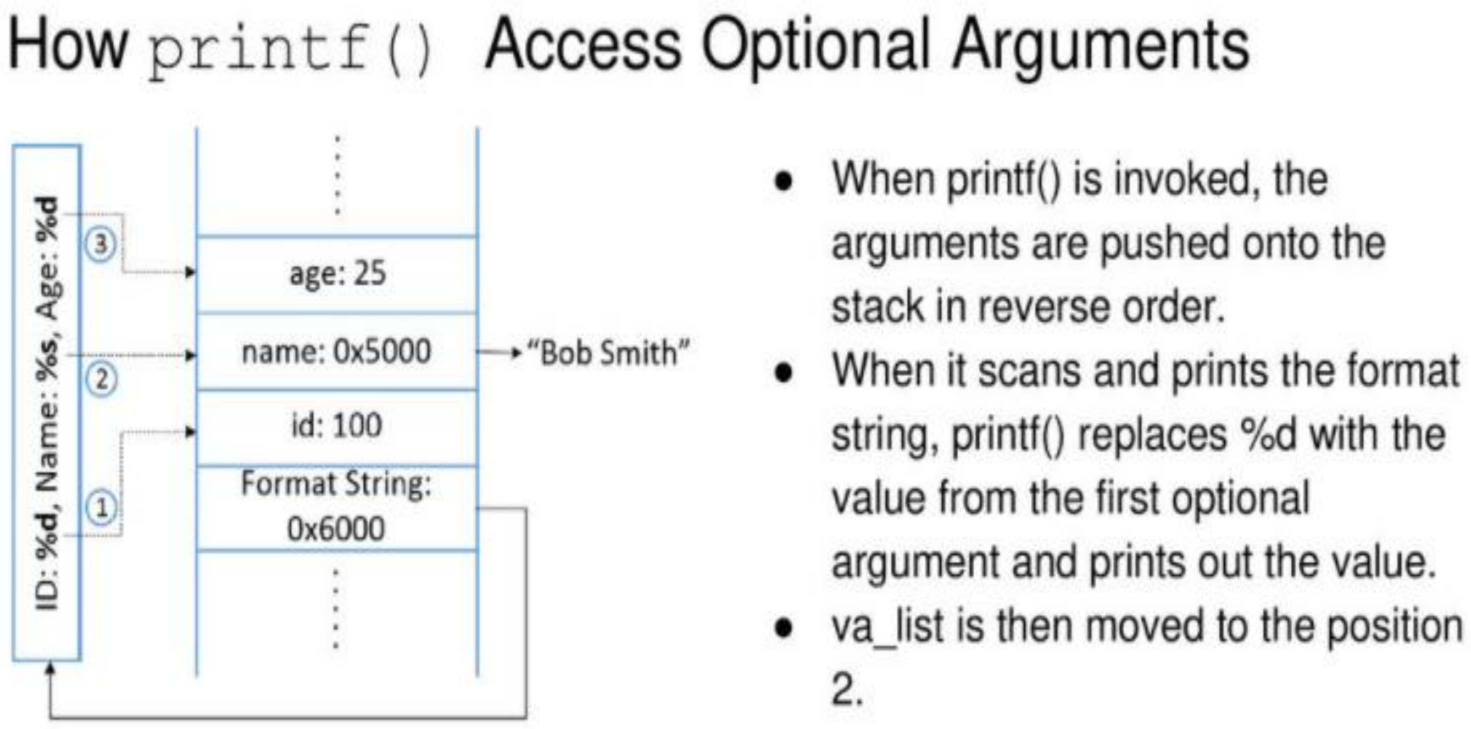
How `printf()` Access Optional Arguments



```
#include <stdio.h>

int main()
{
   int id=100, age=25; char *name = "Bob Smith";
   printf("ID: %d, Name: %s, Age: %d\n", id, name, 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.
- `printf()` calls **va_arg()**, which returns the optional argument pointed by **va_list** and advances it to the next argument.

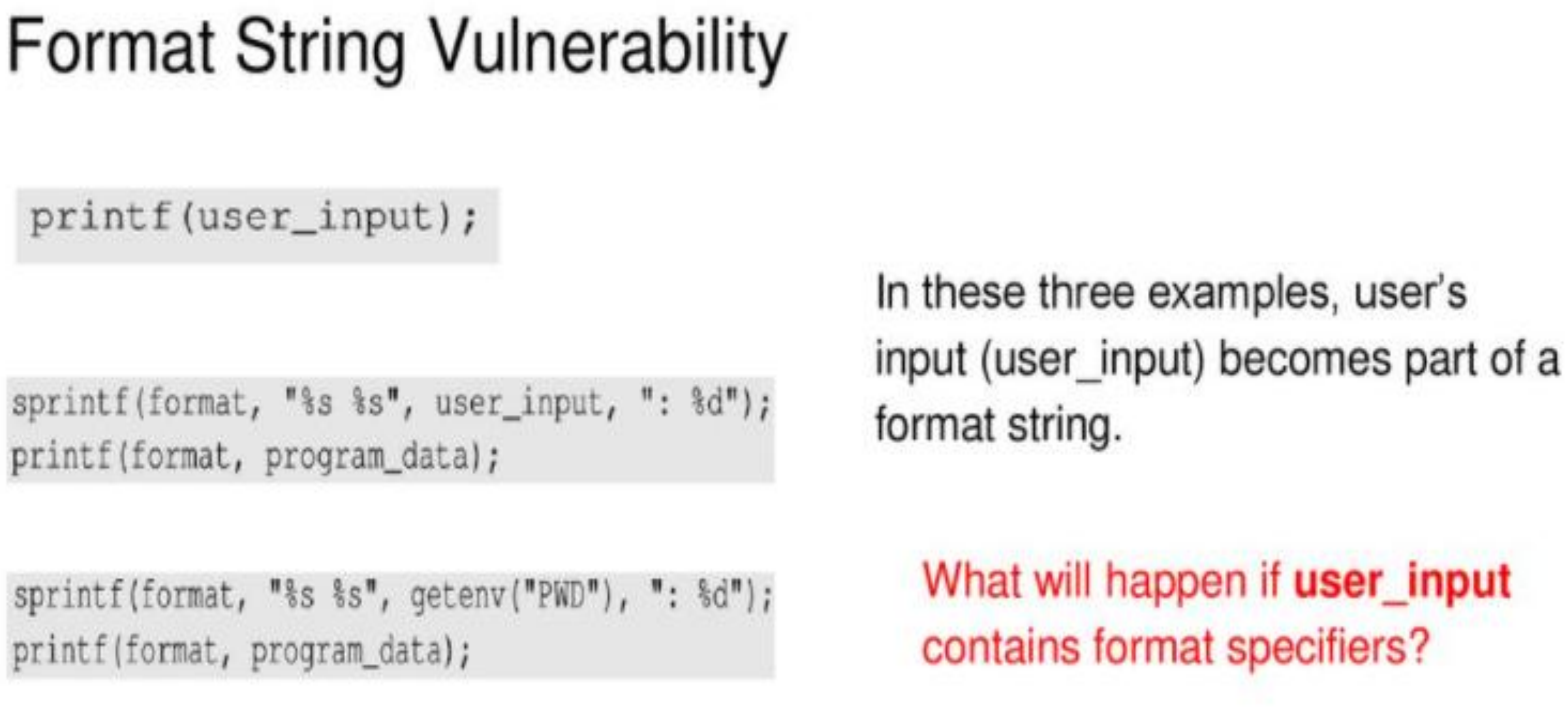
Example



How printf() Access Optional Arguments

- When printf() is invoked, the arguments are pushed onto the stack in reverse order.
- When it scans and prints the format string, printf() replaces %d with the value from the first optional argument and prints out the value.
- va_list is then moved to the position 2.

Format String Vulnerability



Format String Vulnerability

```
printf(user_input);
```

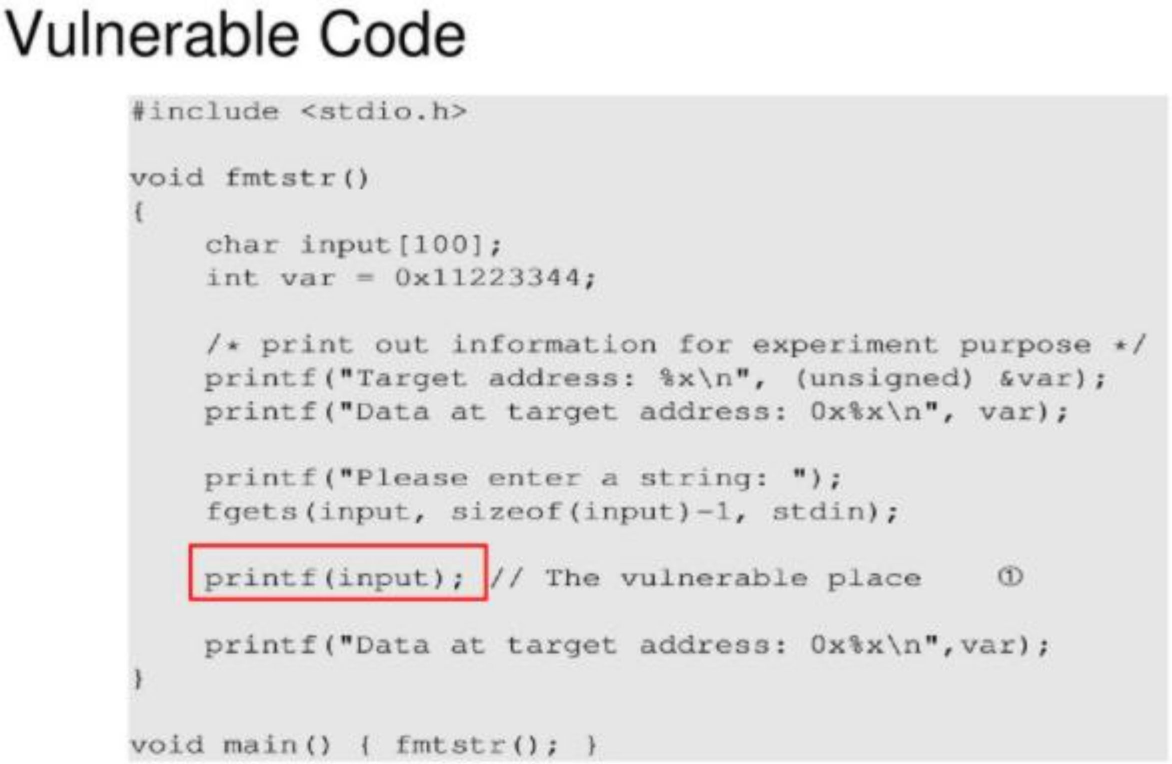
```
sprintf(format, "%s %s", user_input, ": %d");
printf(format, program_data);
```

```
sprintf(format, "%s %s", getenv("PWD"), ": %d");
printf(format, program_data);
```

In these three examples, user's input (user_input) becomes part of a format string.

What will happen if **user_input** contains format specifiers?

Vulnerable Code



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(); }
```

Difference between Buffer Overflow and Format String exploits

- In buffer overflow, the programmer fails to keep the user input between bounds, and attackers exploit that to overflow their input to write to adjacent memory locations.
- But in format string exploits, user-supplied input is included in the format string argument. Attackers use this vulnerability and control the location where they perform arbitrary writes.