Memory Corruption Attacks

Hyoungshick Kim

Department of Software

College of Software

Sungkyunkwan University

Content

- What is buffer overflow?
- Process memory layout
- Basic stack layout
- Buffer overflow attack

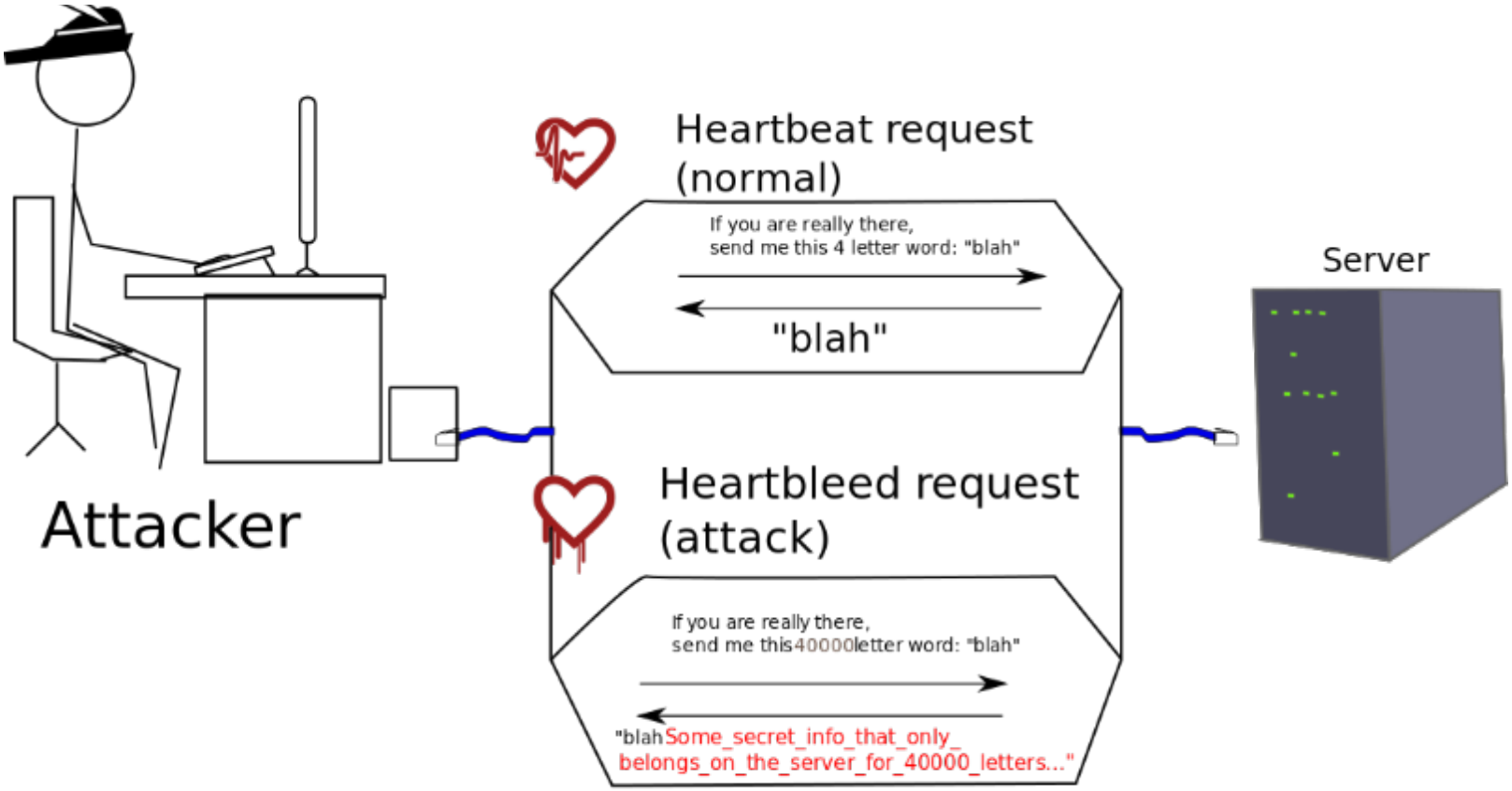
Buffer overflow

- A buffer overflow is a bug that affects low-level code, typically in C and C++ (because all strings are arrays of `char`'s)
- Buffer = contiguous memory associated with a variable or field
- A buffer overflow means **any access of a buffer outside of its allotted bounds**
  - ✓ Could be an over-*write* or over-*read*
- An attacker can perform buffer overflow attacks to do the followings:
  - ✓ Steal private information
  - ✓ Corrupt valuable information
  - ✓ Run code of the attacker's choice

Example: Heartbleed

- SSL/TLS is a core protocol for encrypted communications used by the web
- OpenSSL contains a function known as a heartbeat option. With it, a client periodically sends and receives messages to check whether both the client and the server are both still connected.
  - ✓ Discovered in March 2014. However, it has been in the released code since March 2012 (2 years old!)
- A carefully crafted packet causes OpenSSL to read and return portions of a vulnerable server's memory
  - ✓ Leaking passwords, keys, and other private information

Buffer over-read bug



https://xkcd.com/1354/

The extra data that is sent back is fetched from the server's memory, due to the bug. It could include passwords and private keys.

Control flow hijacking

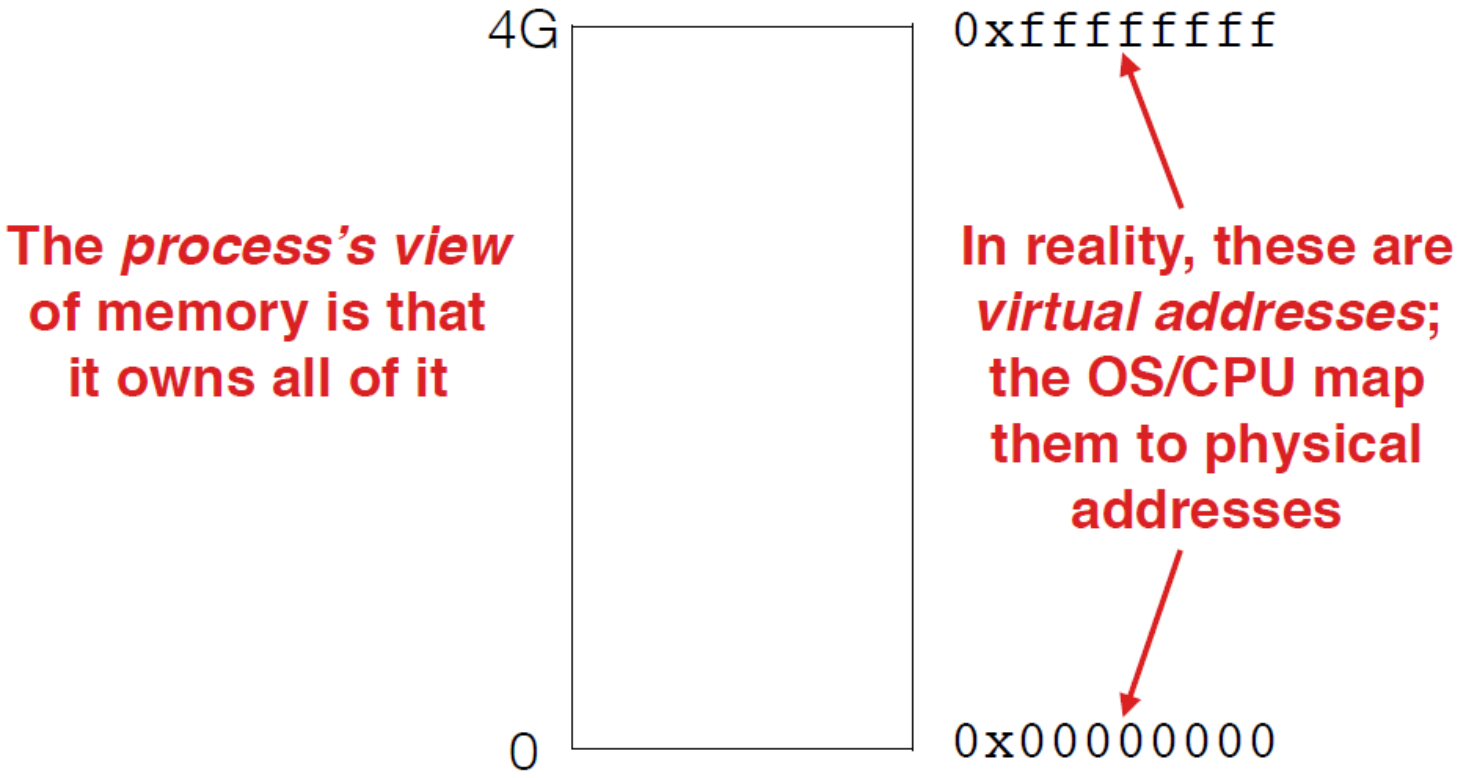
- The general idea is to overflow a buffer so that it overwrites the return address.
- When the function is done, it will jump to whatever address is on the stack.
- We put some code in the buffer and set the return address to point to it!

Problem

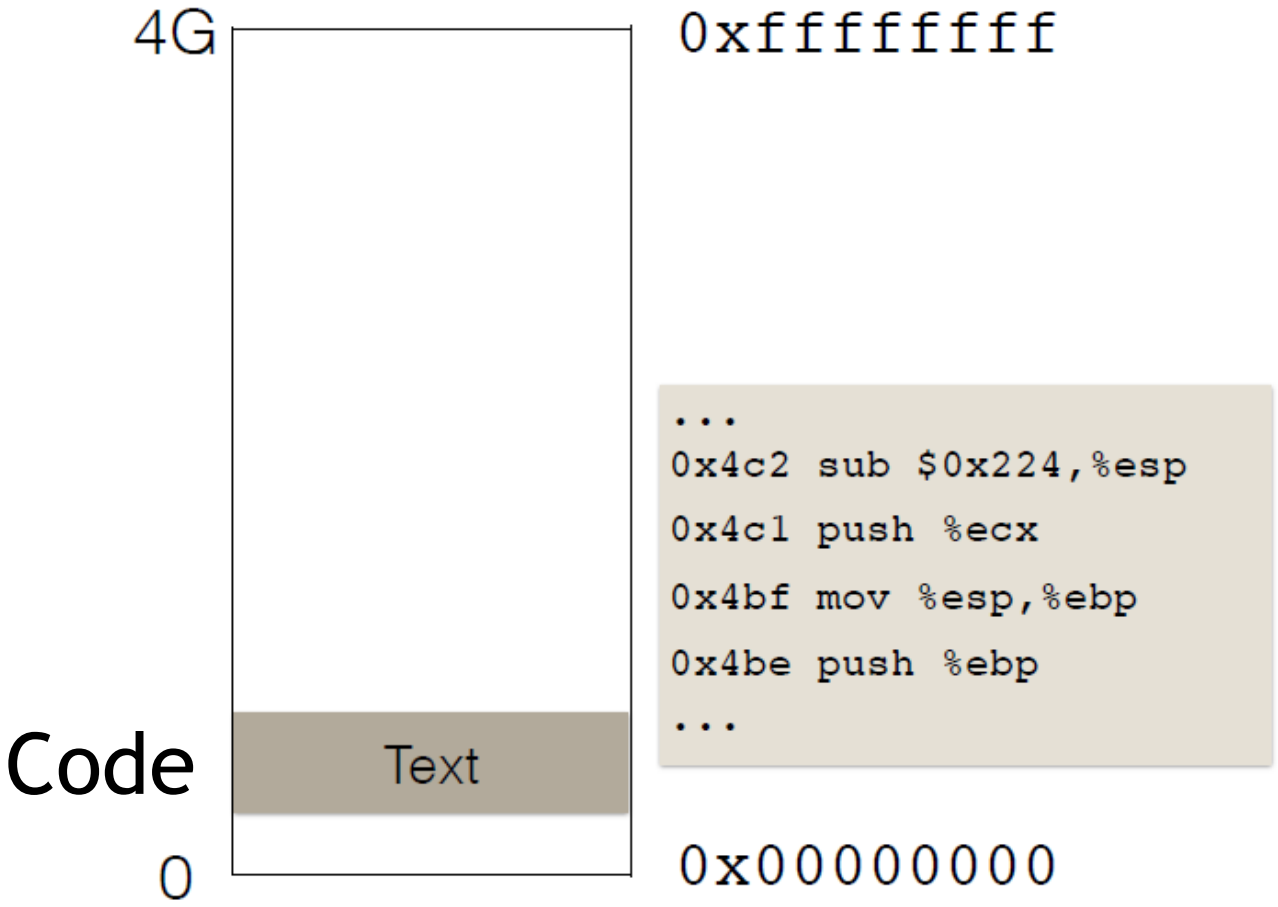
```
void foo(char *s) {
  char buf[10];
  strcpy(buf,s);
  printf("buf is %s\n",s);
}

…
foo("thisstringistolongforfoo");
```

Q. What happened? A. Segmentation fault

All programs are in memory



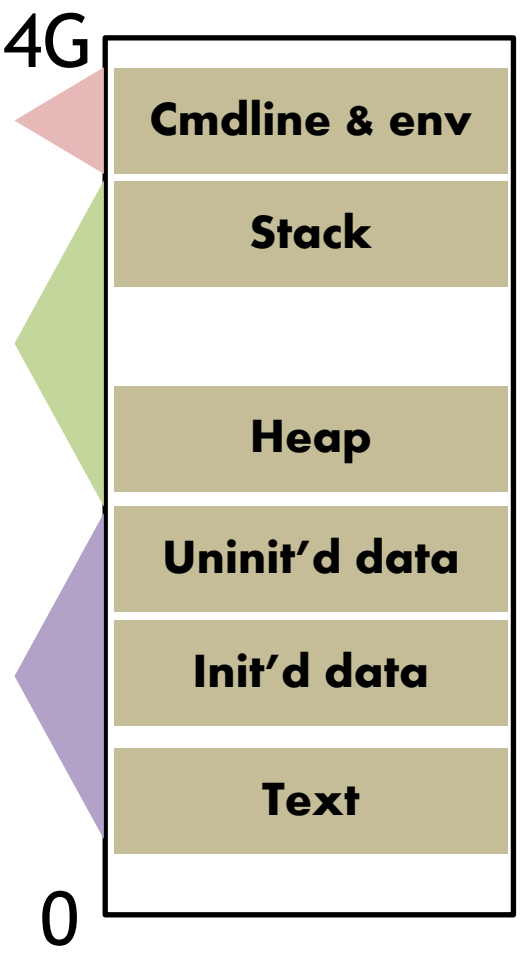
The instructions themselves are in memory



Set when process starts

Runtime

Known at compile time



0xffffffff

```
int f() {
    int x;
    …
```

```
mallloc(sizeof(long));
```

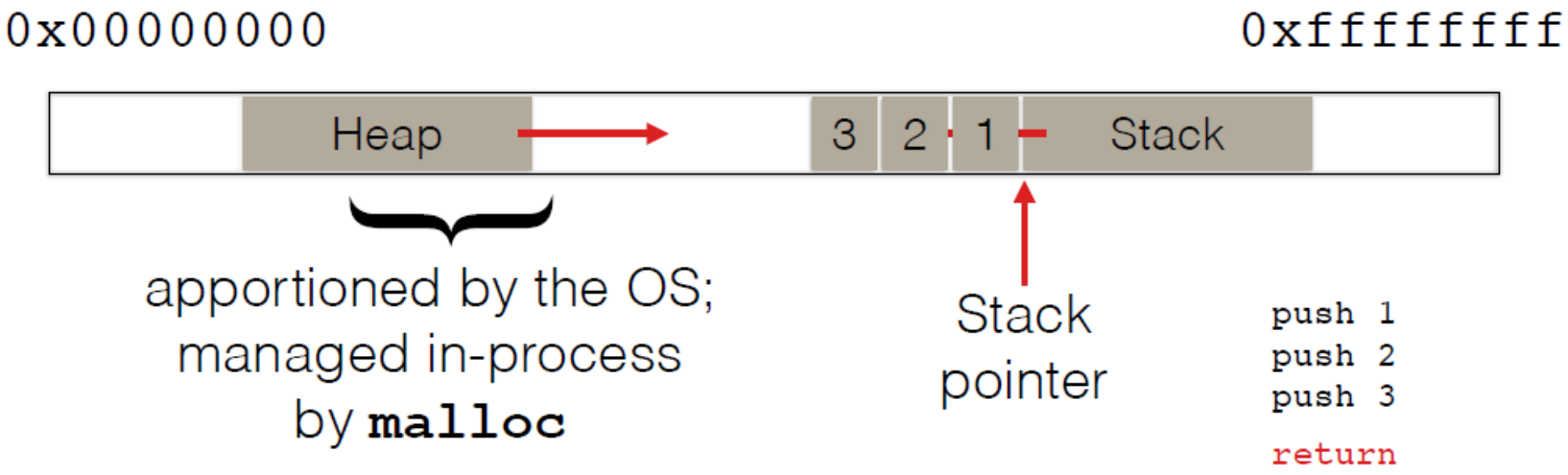
```
static int x;
```

```
static const int y = 10;
```

0x00000000

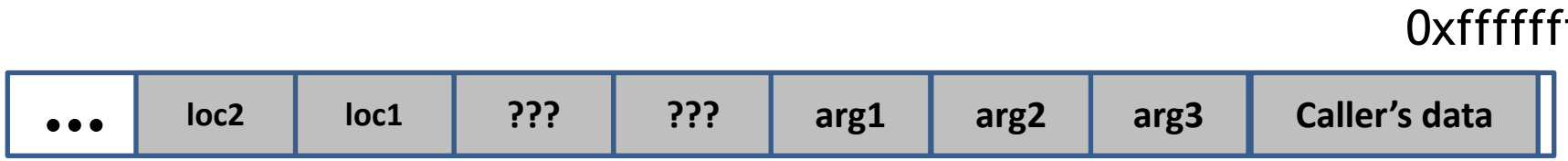
Stack and heap grow in opposite directions

Compiler emits instructions
adjust the size of the stack at run-time



Focusing on the stack for now

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    ...
}
```

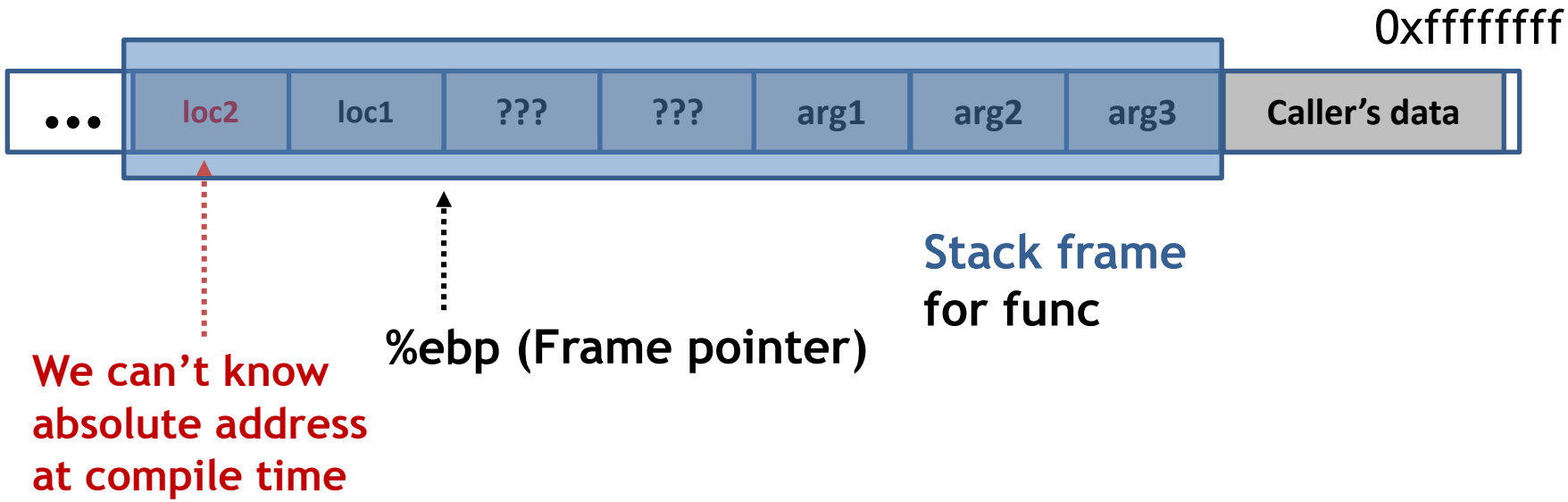


Local variables pushed in the same order as they appear in the code

Arguments pushed in reverse order of code

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    ...
}
```

Q. Where is this loc2?



But, we can know the relative address

- loc2 is always 8bytes before ???s

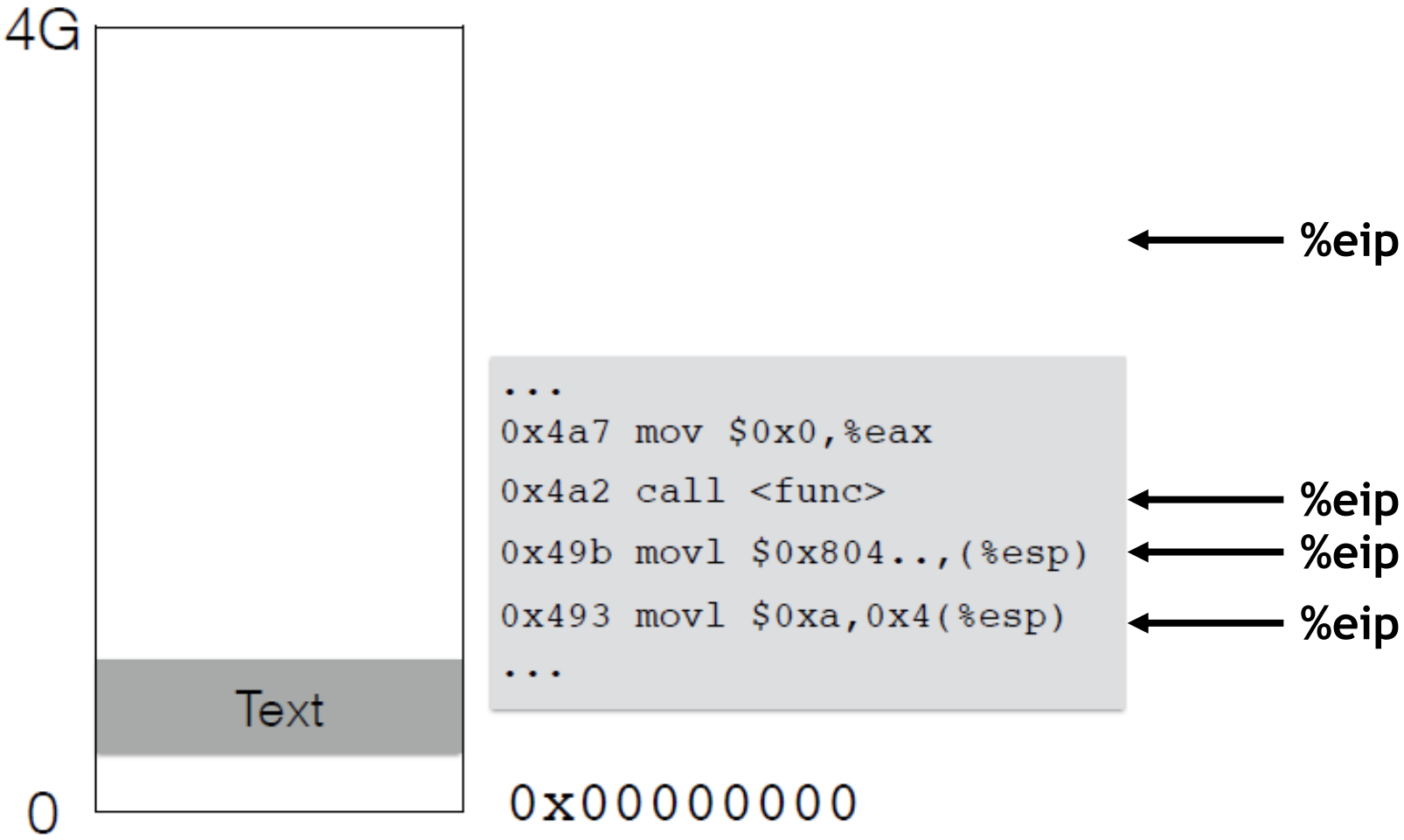
Special registers

- Frame (or Base) Pointer
  - ✓ %ebp
  - ✓ Points to 'bottom' of stack frame
  - ✓ The value of %ebp is pushed on the stack (and later restored) by the **callee** on function entry

- Stack Pointer
  - ✓ %esp
  - ✓ Points to 'top' of stack frame

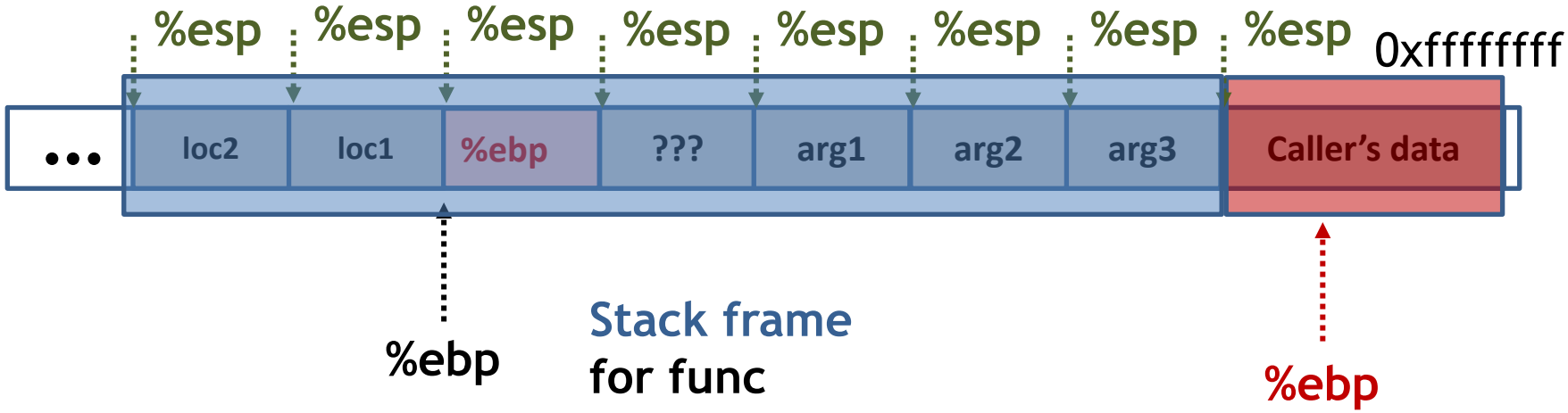
- Instruction Pointer (or program counter)
  - ✓ %eip
  - ✓ Points to the next instruction to be executed

Instructions in memory



Stack frame

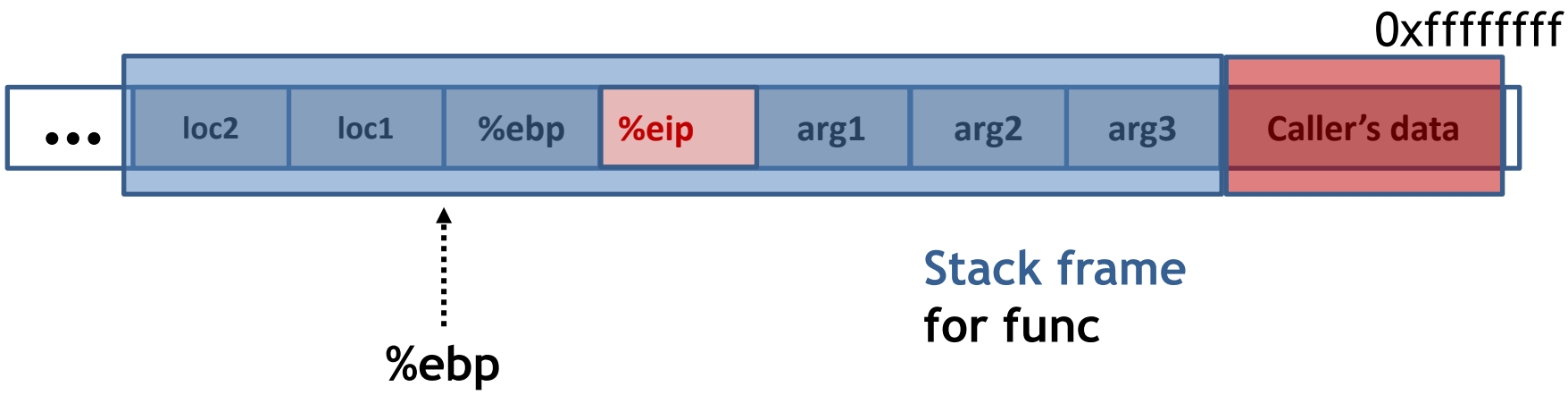
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    ...
}
```



- We need to store ??? to return
- We need to store old %ebp
- We set %ebp to current (%esp)

What is ???

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    ...
}
```



- We also need to store old %eip Push next %eip on the stack before calling

```
void func(char *arg1)
{
     int authenticated = 0;
     char buffer[4];
     strcpy(buffer, arg1);
     if(authenticated) { ...
}

int main()
{
     char *mystr = "AuthMe!";
     func(mystr);
     ...
}
```

```
void foo(char *s) {
      char buf[4];
      strcpy(buf,s);
      …
}
```

All ours!

buffer

strcpy will let you write as much as you want

```
void foo(char *s) {
        char buf[4];
        strcpy(buf,s);
        …
}
```

1. How can you load your own code into memory?
2. How can we get %eip to point to it?

Example of malicious codes

```
#include <stdio.h>

char *args[] = {"/bin/ls", NULL};

void main(void) {
  execv("/bin/ls",args);
  printf("I'm not printed\n");
}
```

- In many cases, attacker's goal is to run a **general-purpose shell**
  - ✓ Command-line prompt that gives attacker general access to the system
- The code to launch a shell is called **shellcode**

Shellcode

```
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Assembly

```
xorl %eax, %eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp,%ebx
pushl %eax
...
```

```
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
...
```

Machine code

(Part of) your input

Loading code into memory

- It **must be the machine code** instructions (i.e., already complied and ready to run)
- You can take a code, and generate machine language.
  - It can't contain any all-zero bytes. Why?
- Copy down the individual byte values and build a string.

Sample program/string

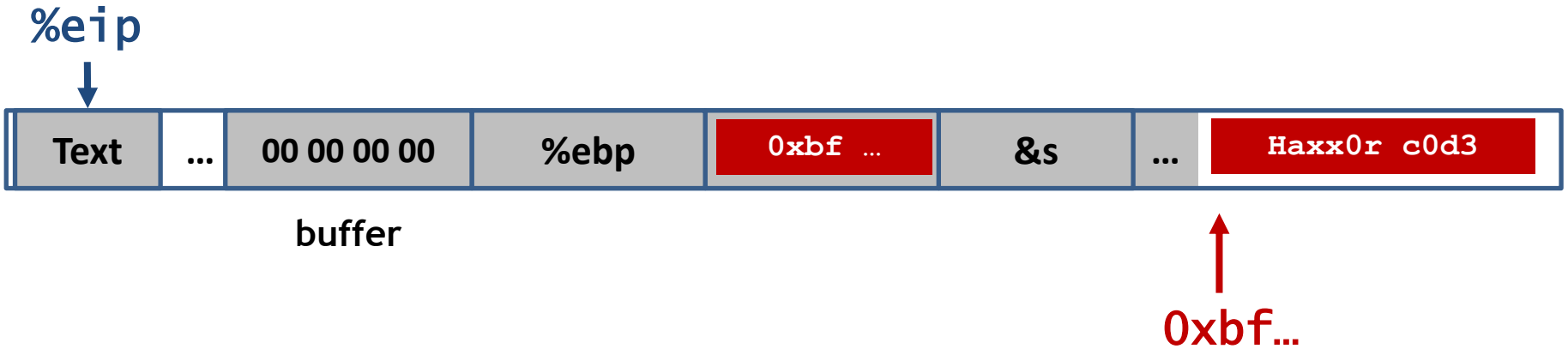
```
unsigned char cde[] =
\x31\xc0\x50\x68\x76\x08\x31\xc0\x88\x46\x07
  \x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x
  8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd
  \x80\xe8\xdc\xff\xff\xff/bin/ls
```

We use this string for buffer overflow!

Sample overflow program

```
unsigned char cde[] = "\x31\xc0\…

void foo(char *s) {
  char buf[100];
  strcpy(buf,s);
  printf(“buf is %s\n”,s);
}

int main(void) {
  printf("Running foo\n");
  foo(cde);
  printf("foo returned\n");
}
```

Hijacking the saved %eip

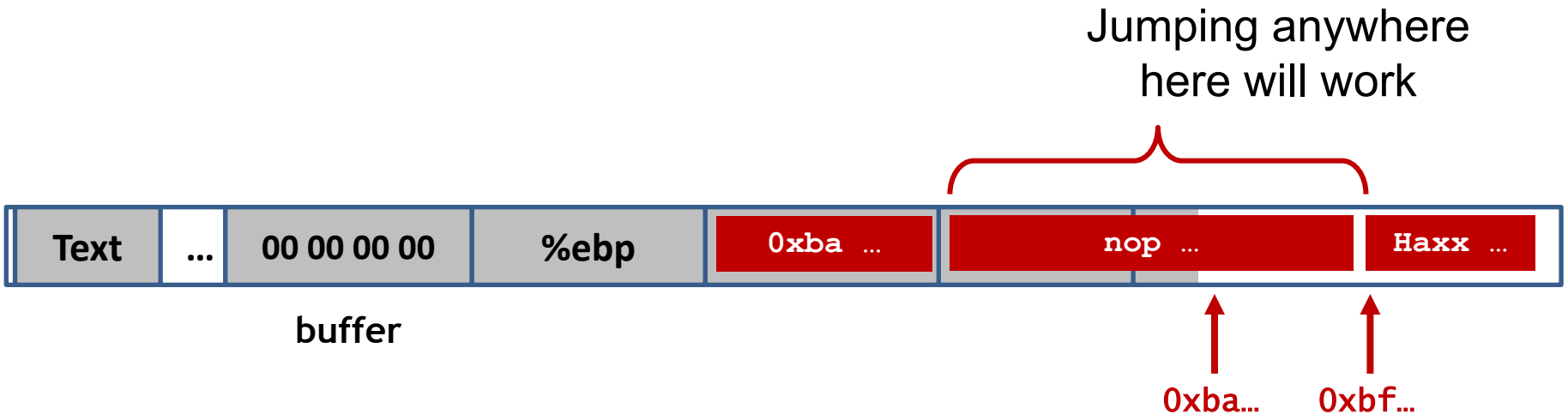


However, if we point to wrong address (i.e., 0xbd … invalid instruction), the CPU will panic

How to guess the correct return address?

- If we don't have access to the code, we don't know how far the buffer is from the saved %ebp
- One approach: just try a lot of different values!
  - Worst case scenario: it's a 32 (or 64) bit memory space, which means 2^{32} (2^{64}) possible answers
- Without address randomization:
  - The **stack always starts** from the same fixed address

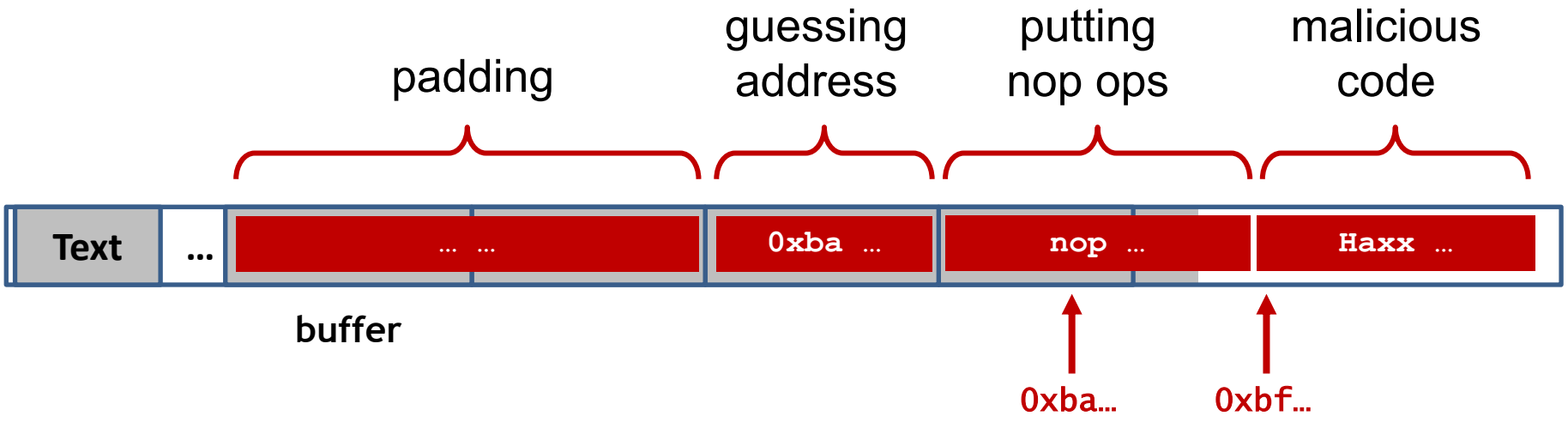
Improving our chances: nop sleds



nop is a single-byte instruction (just moves to the next instruction)

Now we improve our chances of guessing by a factor of #nops

Putting it all together



Questions?


Sungkyunkwan
Campus
Social
Graduate
Professors
English
Confucian
campus
Zone
Sciences
University
Global
Education
Seoul
Natural
Suwon
Korean
School
university
established
edit
also
Humanities
Science
schools
Administration
ASLE-Korea
Academic
Advanced
constructed
Building
Engineering
located
Joseon
Anniversary
Institute
now
two
Hall
Course
lecturers