CIS 449/549: Software Security

Anys Bacha

Slides from U. Shankar, M. Hicks, K. Du, D. Boneh, N. Zeldovich, A. Rahmati

What is a Buffer Overflow?

- The Bugs Framework (government entity that classifies bugs into distinct classes) defines a buffer overflow as

software accesses through an array of memory that is outside the boundary of the array

What is a Buffer Overflow?

- Buffer overflows (BOF) stem primarily from low level bugs written in C/C++

- In most cases buffer overflows cause crashes, but if maliciously crafted can result in:

    - Private data being stolen
    - Arbitrary code being executed
    - Critical information being corrupted

How Relevant Are BOF

- Performance is always at the top of the feature list
  - We like technology to always be fast

- Low level languages such as C/C++ are still very popular

- Systems software often written in C/C++ (operating systems, file systems, databases, compilers, network servers, command shells, etc.)

How Relevant Are BOF

- Many big companies still rely on C++ for their software including Google and Facebook (driven by performance)

- Internet of Things (IoT) software is primarily developed in C due to the limited hardware resources

- Compromises can result in significant damage
  - Arbitrary code execution

How Relevant Are BOF

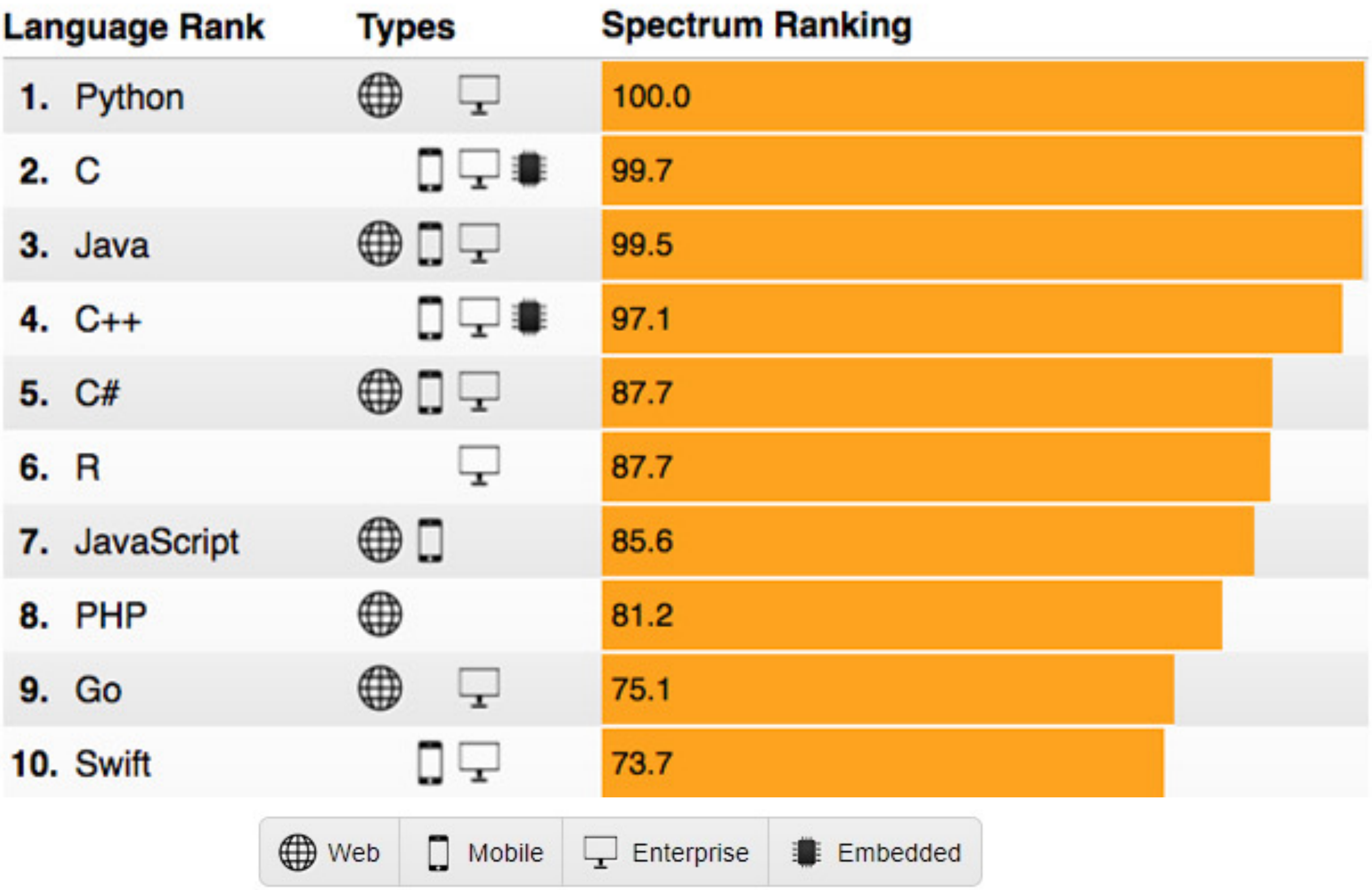
- Low level languages has the downside of exposing memory details
  - Exposes raw pointers to memory
  - Does not explicitly perform bounds-checking on arrays
  - Hardware doesn’t check this
  - We want to be as close to the hardware as possible

C/C++ Still Popular

Rank	Language	Type	Score
1	Python	Web, Enterprise, Embedded	100.0
2	Java	Web, Mobile, Enterprise	95.4
3	C	Mobile, Enterprise, Embedded	94.7
4	C++	Mobile, Enterprise, Embedded	92.4
5	JavaScript	Web	88.1
6	C#	Web, Mobile, Enterprise, Embedded	82.4
7	R	Enterprise	81.7
8	Go	Web, Enterprise	77.7
9	HTML	Web	75.4
10	Swift	Mobile, Enterprise	70.4

Web Mobile Enterprise Embedded

C/C++ Still Popular

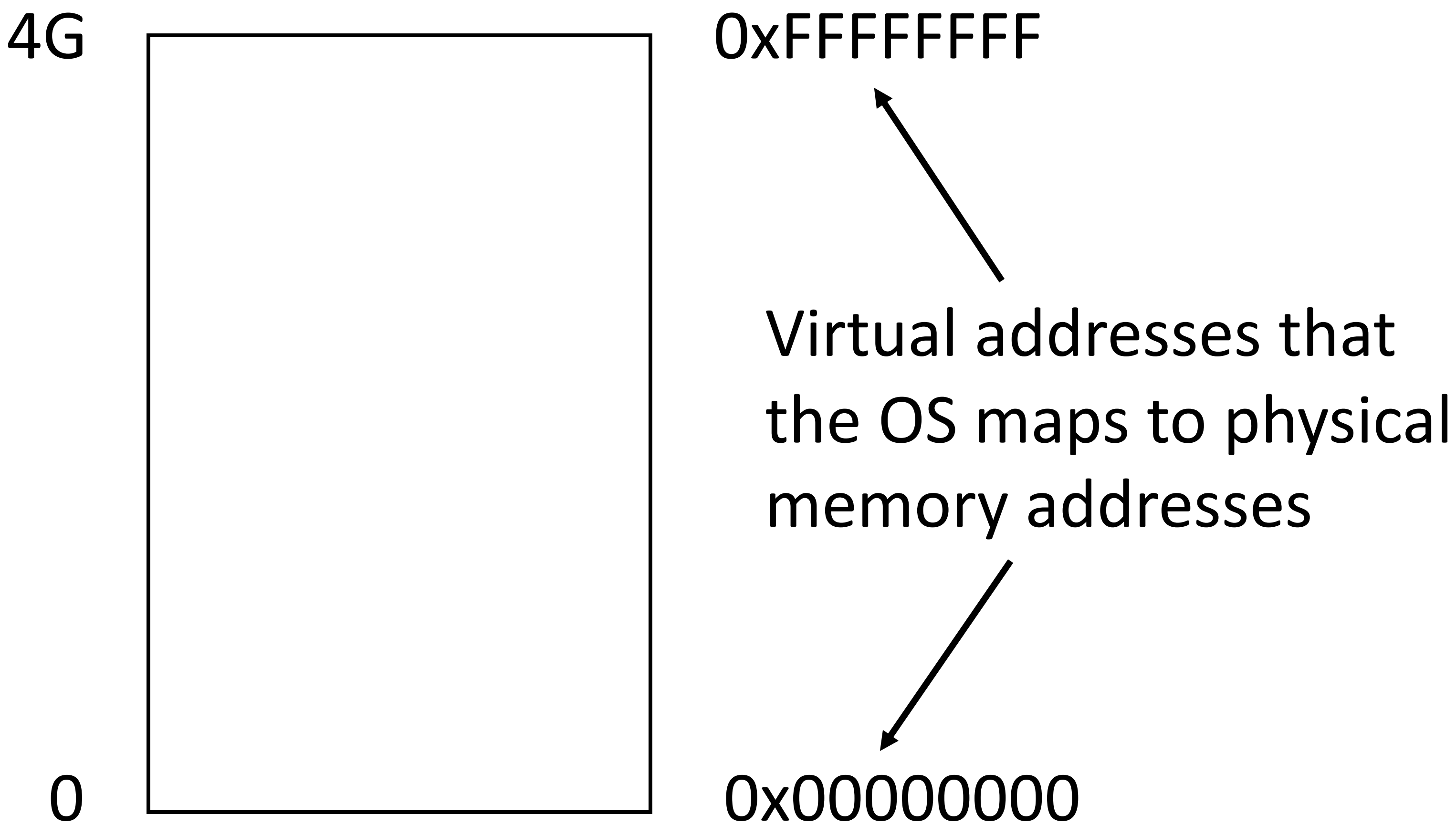


https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

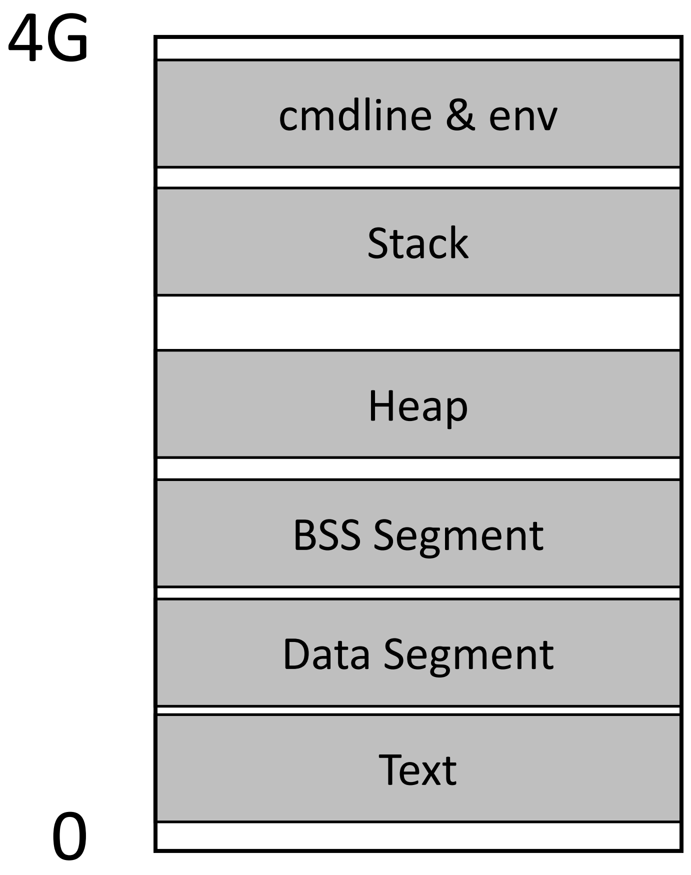
Memory Layout

Program Layout in Memory

Process thinks it
owns the entire range



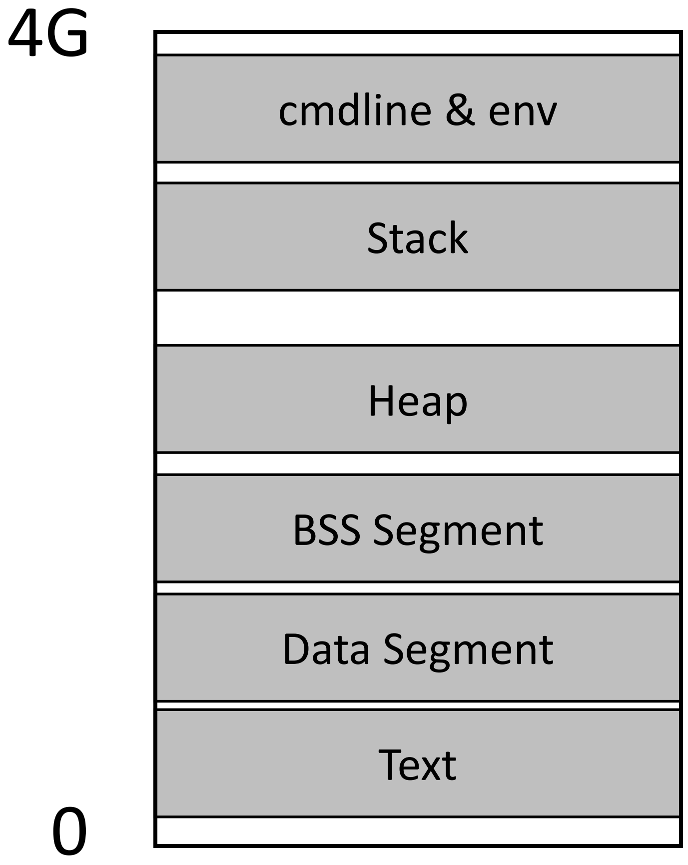
Program Layout in Memory




```c
int x = 100;
int main()
{

   int  a=2;
   float b=2.5;
   static y;


   int *ptr = (int *) malloc(2*sizeof(int));


   ptr[1]=5;
   ptr[2]=6;


   free(ptr)

   return 1;
}
```

Where would variables be located?

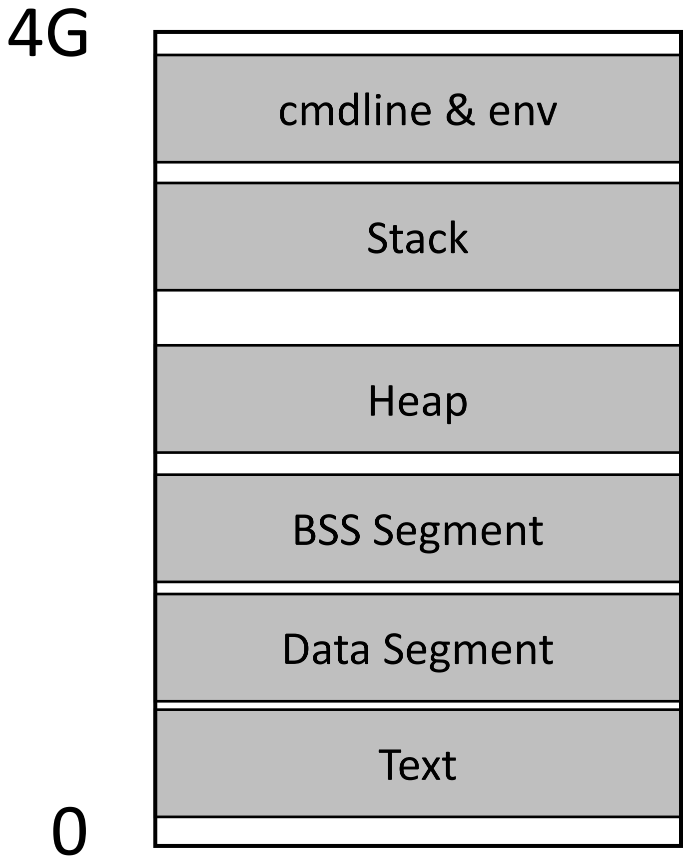
Program Layout in Memory




```c
int x = 100;
int main()
{
   int  a=2;
   float b=2.5;
   static y;

   int *ptr = (int *) malloc(2*sizeof(int));

   ptr[1]=5;
   ptr[2]=6;

   free(ptr)

   return 1;
}
```

Where would variables be located?

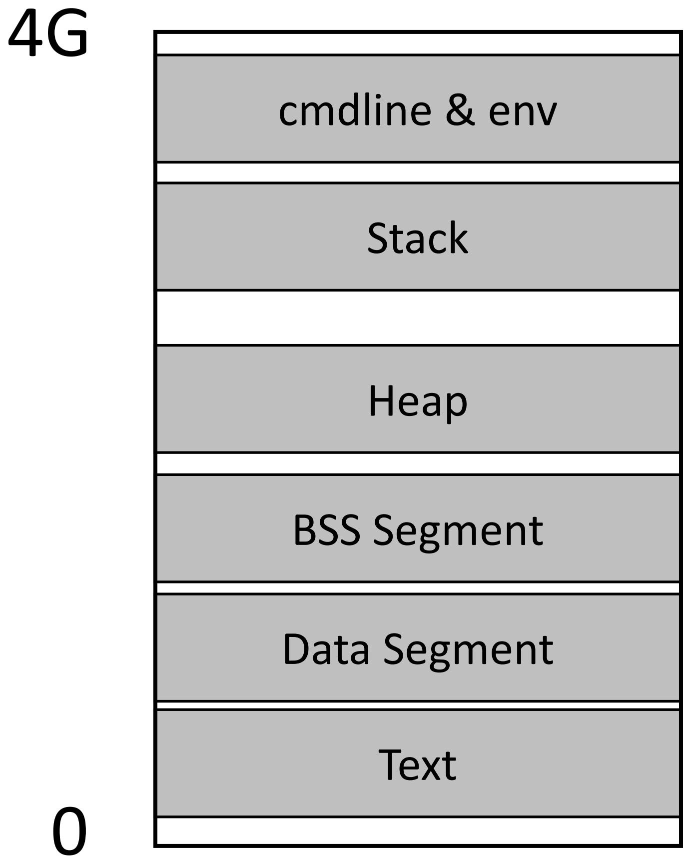
Program Layout in Memory




```
int x = 100;
int main()
{
  int  a=2;
  float b=2.5;
  static y;

  int *ptr = (int *) malloc(2*sizeof(int));

  ptr[1]=5;
  ptr[2]=6;

  free(ptr)

  return 1;
}
```

Where would variables be located?

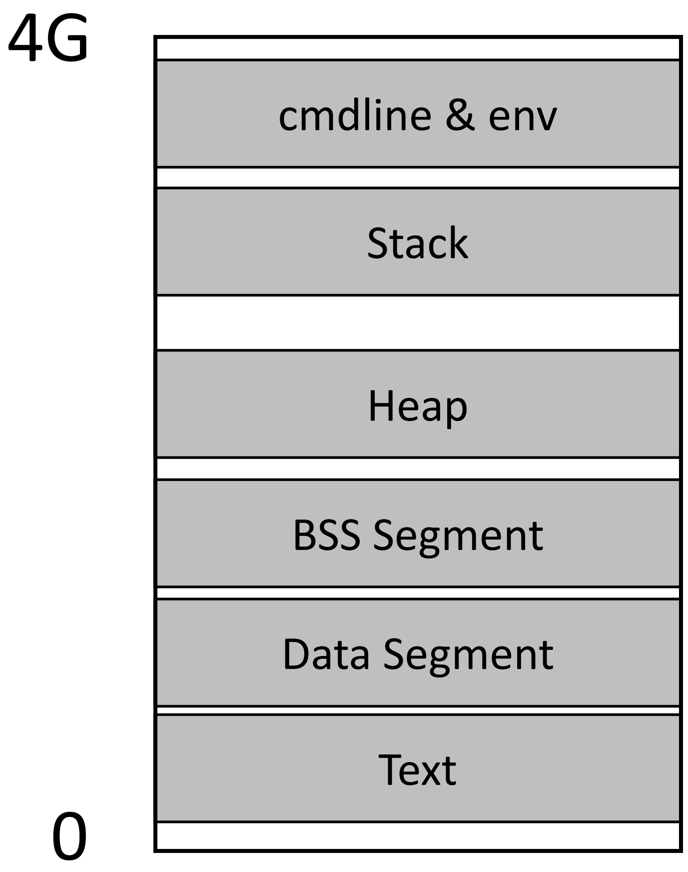
Program Layout in Memory




```
int x = 100;
int main()
{

   int  a=2;
   float b=2.5;
   static y;


   int *ptr = (int *) malloc(2*sizeof(int));


   ptr[1]=5;
   ptr[2]=6;


   free(ptr)

   return 1;
}
```

Where would variables be located?

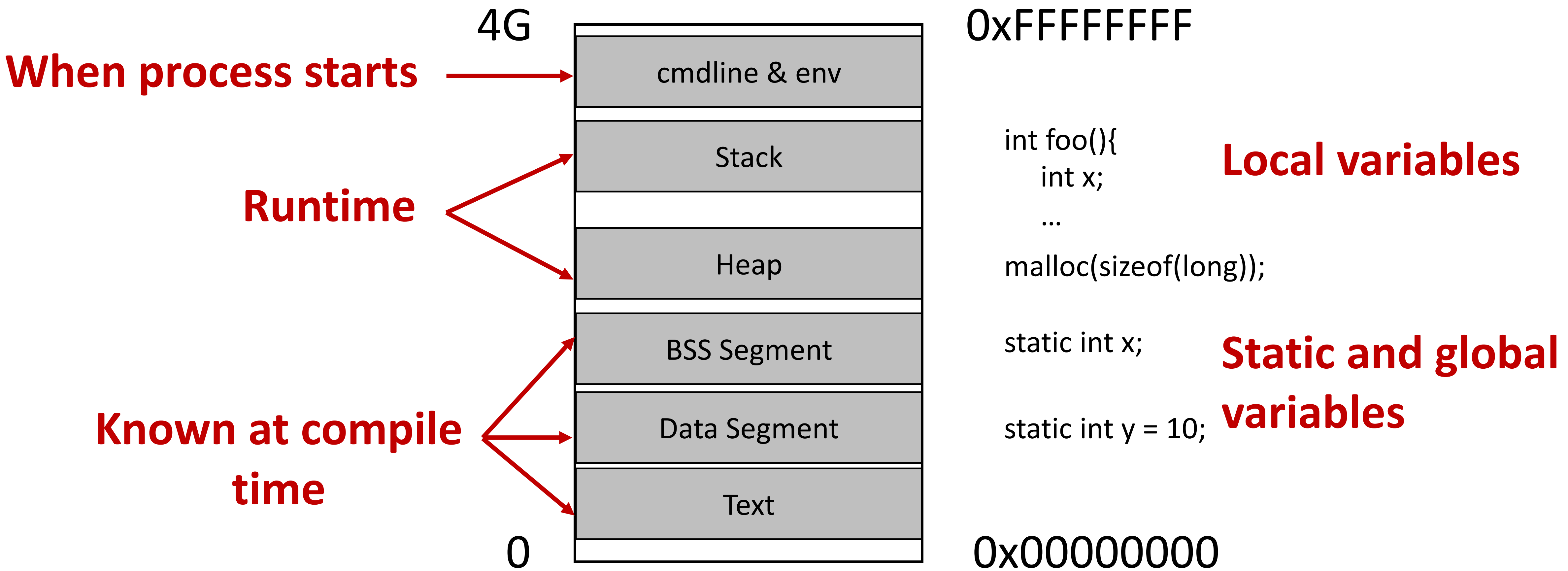
Program Layout in Memory




```
int x = 100;
int main()
{

   int  a=2;
   float b=2.5;
   static y;


   int *ptr = (int *) malloc(2*sizeof(int));


   ptr[1]=5;
   ptr[2]=6;


   free(ptr)

   return 1;
}
```

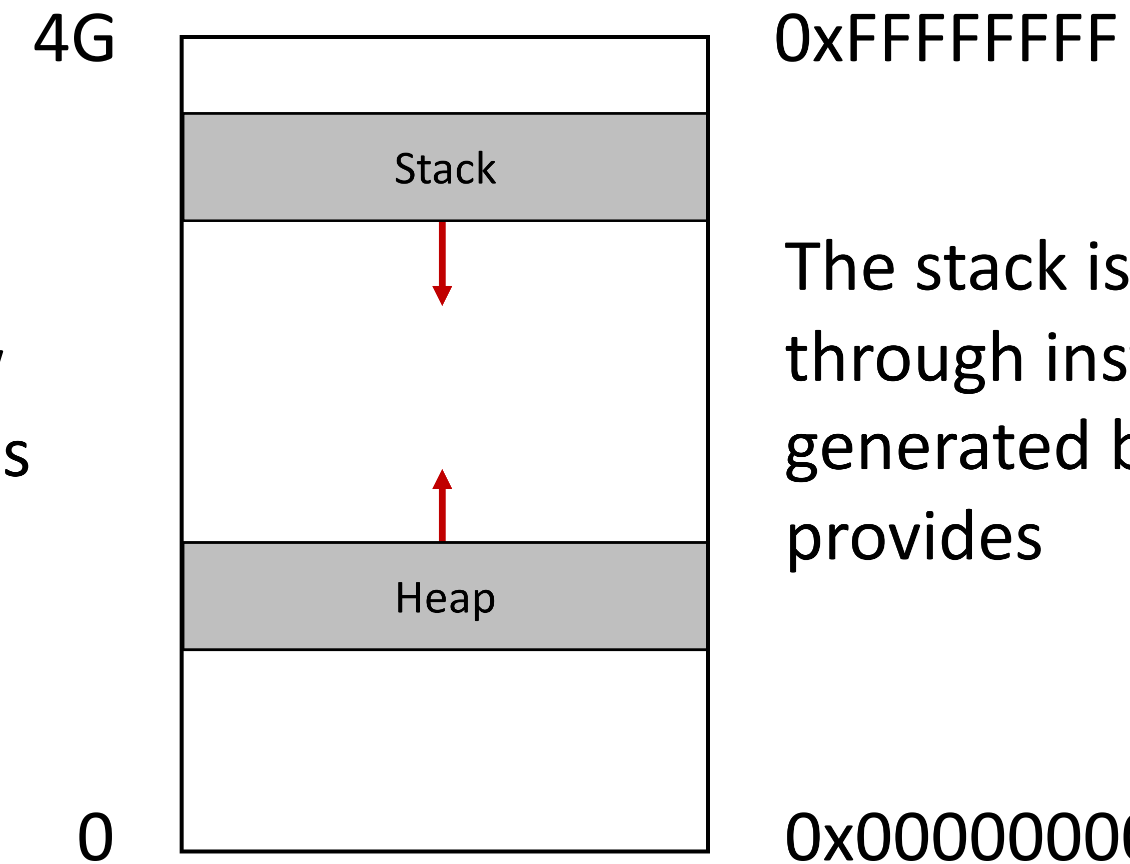
Where would variables be located?

Program Layout in Memory



Focus on Stack-based Attacks

Stack and heap grow in opposite directions



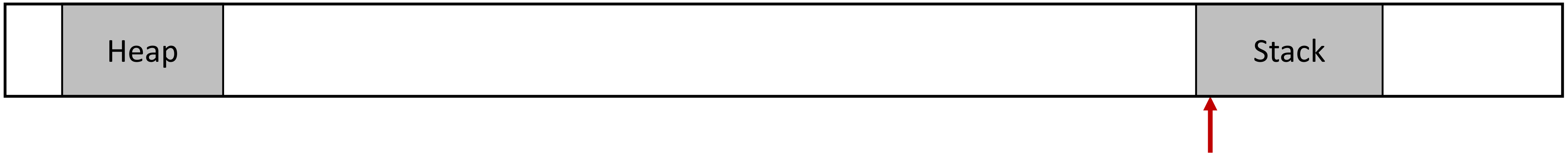
The stack is adjusted through instructions generated by the compiler provides

Focus on Stack-based Attacks

0x00000000

The stack is adjusted through instructions generated by the compiler provides

0xFFFFFFFF



Stack pointer

push 1
push 2
push 3

Focus on Stack-based Attacks

0x00000000

The stack is adjusted through instructions generated by the compiler provides

0xFFFFFFFF

The heap is allocated by the OS and managed by the process through malloc()

Stack pointer

push 1
push 2
push 3

Function Calls

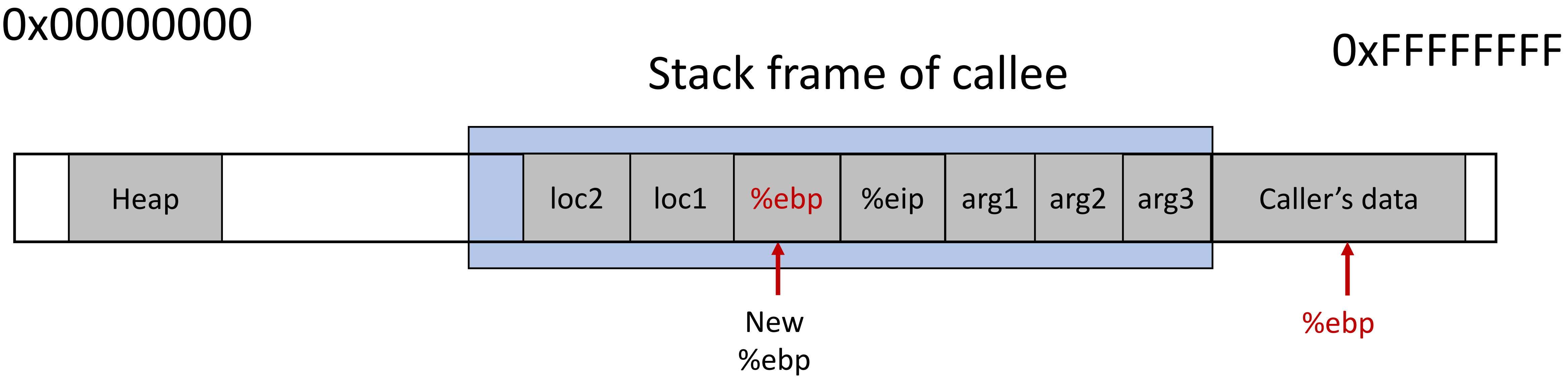
```
int main() {
  …
  foo(1, 2, 3);
  …
}
```

```
void foo(int arg1, int arg2, int arg3) {
  char loc1[4];
  int loc2;
  …
}
```

- Caller:
  - Push arguments onto stack in reverse order
  - Push return address
    - %eip + sizeof(curr inst.)
  - Branch to function address
  - Restore stack by popping arguments

- Callee:
  - Push old frame pointer (%ebp)
  - Set %ebp to top of stack (where old %ebp stored)
  - Push local variables
  - …
  - Restore old stack frame
    - %esp = %ebp; pop %ebp
  - Branch to return address: pop %eip

Function Calls



Summary of Function Calls

- Calling function:
  - Push arguments onto the stack in reverse order
  - Push the return address of the next instruction to be run in the calling function
    - %eip + sizeof(current instruction)
  - Branch to the function’s address
- Called function:
  - Push the old frame pointer onto the stack (%ebp)
  - Set the new frame pointer %ebp to where the old %ebp was pushed
  - Push local variables onto the stack

Summary of Function Calls

- Returning to calling function:
  - Reset the previous stack frame
    - %ebp = (%ebp)
    - Need to copy %ebp into another register first
  - Jump back to the return address
    - %eip = 4(%ebp)
    - Need to use copied value of ebp (current stack frame)

Stack Layout Example

- Stack Frame

```
void foo(int a, int b) {
  int x, y;
  x = a+b;
  y = a – b;
}
```

foo(5, 6);

What does the stack frame look like?

Stack Layout Example

- Stack Frame

```
void foo(int a, int b) {
  int x, y;

  x = a+b;
  y = a – b;
}
```

foo(5, 6);

How do we reference a, b, x, y?

	y	x	%ebp	%eip	a=5	b=6	Caller’s data	

Binary code is generated during compilation stage!

Stack Layout Example

- Stack Frame

```
void foo(int a, int b) {
  int x, y;

  x = a+b;
  y = a – b;
}
```

foo(5, 6);

How do we reference a, b, x, y?

- Frame Pointer

```
movl 12(%ebp), %eax
movl 8(%ebp), %edx
addl %edx, %eax
movl %eax, -4(%ebp)
```

	y	x	%ebp	%eip	a=5	b=6	Caller's data	

Binary code is generated during compilation stage!

Compiler uses offsets relative to ebp

Copying Data to a Buffer

```
int main() {
  …
  char src[40] = “Hello world \0 Extra string”;
  char dest[40];

   strcpy(dest, src);

   return 0;
}
```

A buffer overflow involves copying data to a buffer

Copying Data to a Buffer

```
int main() {
  …
  char src[40] = “Hello world \0 Extra string”;
  char dest[40];

   strcpy(dest, src);

   return 0;
}
```

What is this?

A buffer overflow involves copying data to a buffer

Copying Data to a Buffer

```
int main() {
  …
  char src[40] = “Hello world \0 Extra string”;
  char dest[40];

   strcpy(dest, src);

   return 0;
}
```

What is this?

Tells compiler to insert 0x0 in binary

A buffer overflow involves copying data to a buffer

Copying Data to a Buffer

```
int main() {
  …
  char src[40] = "Hello world \0 Extra string";
  char dest[40];

   strcpy(dest, src);

   return 0;
}
```

What is this?

Tells compiler to insert 0x0 in binary

Different ways to copy data

strcpy()

How does strcpy do the copy?

memcpy()

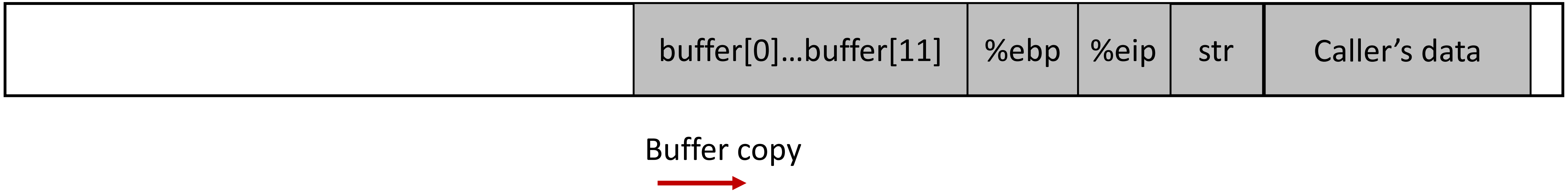
Needs size

A buffer overflow involves copying data to a buffer

Buffer Overflow

```
void foo (char *str) {
  char buffer[12];
  strcpy(buffer, str);
}
int main() {
  char *str = “This is definitely longer than 12”;
  foo(str);
  return 0;
}
```

What will happen after this?



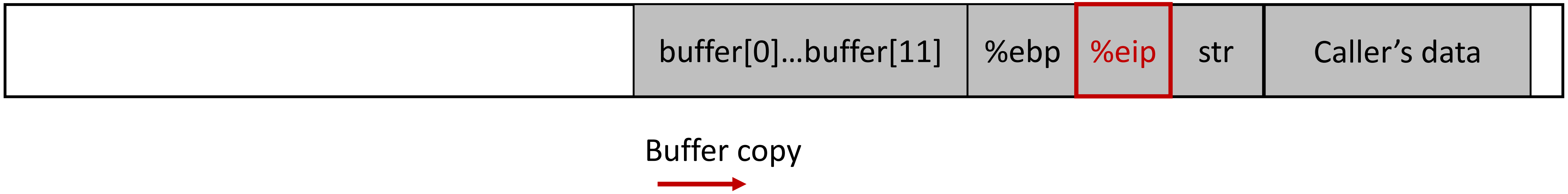
Buffer Overflow

```
void foo (char *str) {
  char buffer[12];
  strcpy(buffer, str);
}
int main() {
  char *str = “This is definitely longer than 12”;
  foo(str);
  return 0;
}
```

Execute unmapped address

Jump to protected place

Invalid instruction

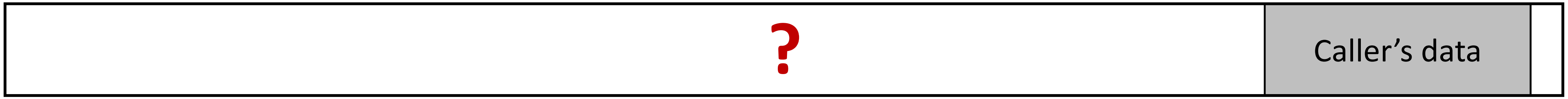


Buffer Overflow Example 1

```
void foo (char *arg1) {
  char buffer[4];
  strcpy(buffer, arg1);
  …
}
int main() {
  char *str = “AuthMe!”;
  foo(str);
  …
}
```

What will this code do?

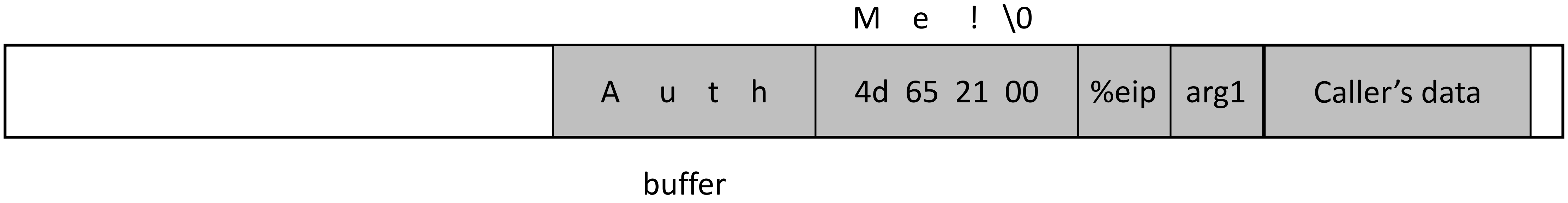
Describe the stack layout after foo() is called?



Buffer Overflow Example 1

```
void foo (char *arg1) {
  char buffer[4];
  strcpy(buffer, arg1);
  …
}
int main() {
  char *str = “AuthMe!”;
  foo(str);
  …
}
```

What will happen to the program?

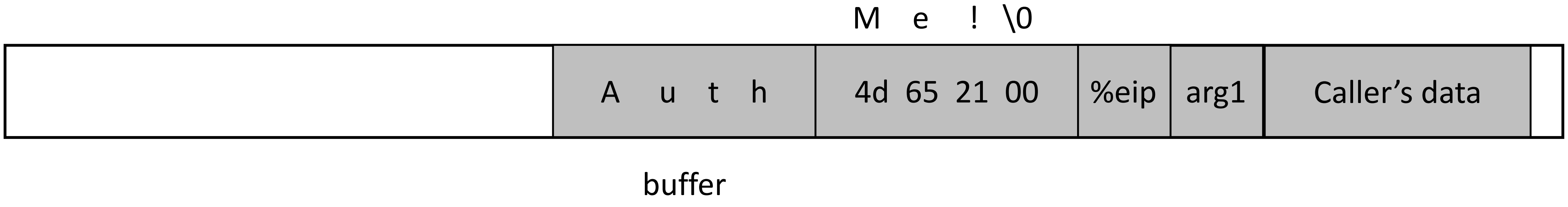


Buffer Overflow Example 1

```
void foo (char *arg1) {
  char buffer[4];
  strcpy(buffer, arg1);
  …
}
int main() {
  char *str = “AuthMe!”;
  foo(str);
  …
}
```

What will happen to the program?

Crash with SEGFAULT due to bad %ebp

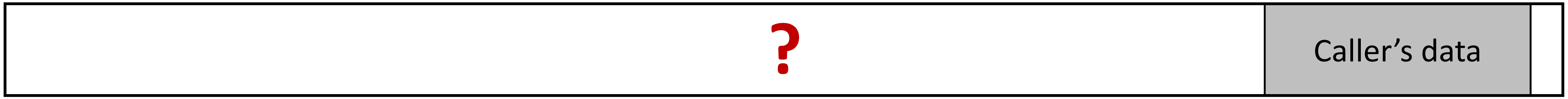


Buffer Overflow Example 2

```
void foo (char *arg1) {
  int authenticated = 0;
  char buffer[4];
  strcpy(buffer, arg1);
  if(authenticated) {…}
}
int main() {
  char *str = “AuthMe!”;
  foo(str);
  return 0;
}
```

What will this code do?

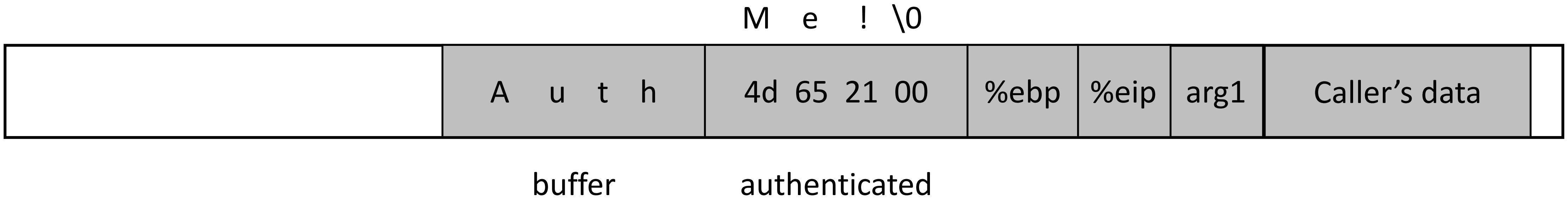
Describe the stack layout after foo() is called?



Buffer Overflow Example 2

```
void foo (char *arg1) {
  int authenticated = 0;
  char buffer[4];
  strcpy(buffer, arg1);
  if(authenticated) {…}
}
int main() {
  char *str = "AuthMe!";
  foo(str);
  return 0;
}
```

The user is now authenticated without any crashes



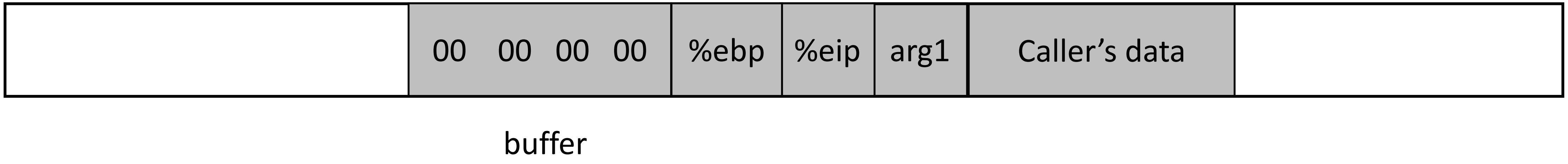
Most Programs Process User Input

- Previous examples used hardcoded strings
- Most useful programs require some level of interaction with the user
- Users can supply input through a multitude of mechanisms including text input, packets over the networks, environment variables, and file input

What Can We Do with User Input?

```
void foo (char *arg1) {
  char buffer[4];
  strcpy(buffer, arg1);
  …
}
```

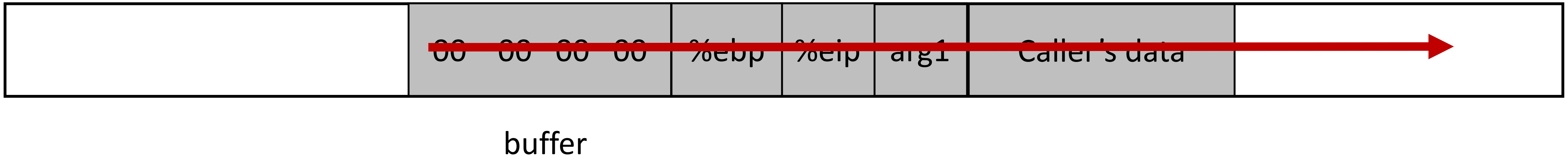
What can we do with user input to make this more interesting?



What Can We Do with User Input?

```
void foo (char *arg1) {
  char buffer[4];
  strcpy(buffer, arg1);
  …
}
```

What can we do with user input to make this more interesting?



strcpy() allows you to overwrite memory until \0 is encountered

What can you do with this knowledge?

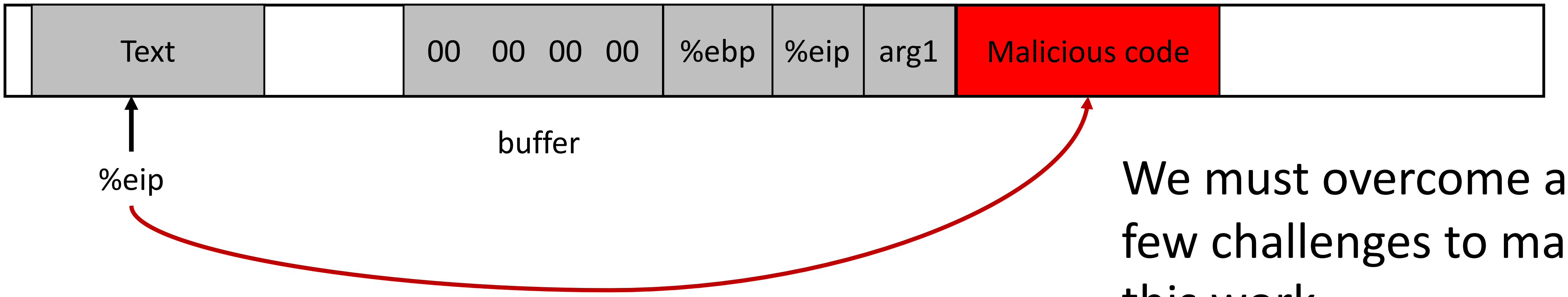
Code Injection

Overview

```
void foo (char *arg1) {
  char buffer[4];
  sprintf(buffer, arg1);
  …
}
```

Goal:

- Use input as attack surface
- Insert user supplied code into memory
- Set %eip to point to user code



We must overcome a few challenges to make this work

Challenge 1

- Must directly load machine code into memory (instructions we want to see executed)

- The machine code must not contain any zeros
  - Zeros would cause sprintf(), gets(), scanf() to stop copying

- Need to run a general purpose shell that provides attacker with easy access to system resources

Shellcode

```
int main() {
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Shellcode is code that spawns a shell

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

Write code in assembly

Assembler

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

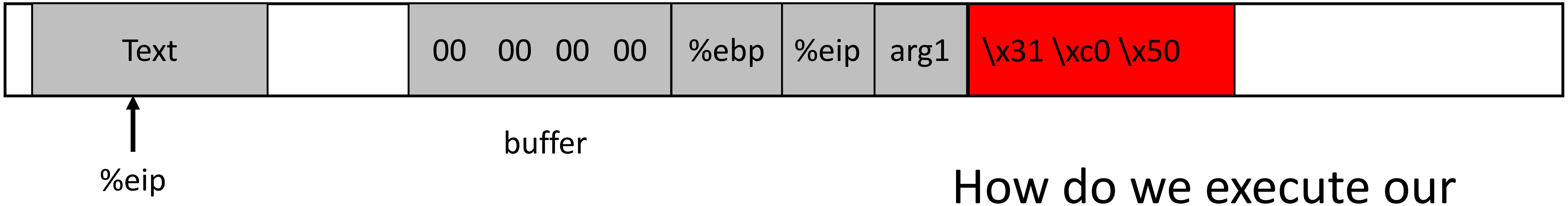
Machine code

Shellcode Example

```
Line 1: xorl %eax,%eax
Line 2: pushl %eax             # push 0 into stack (end of string)
Line 3: pushl $0x68732f2f      # push "//sh" into stack
Line 4: pushl $0x6e69622f      # push "/bin" into stack
Line 5: movl %esp,%ebx         # %ebx = name[0]
Line 6: pushl %eax             # name[1]
Line 7: pushl %ebx             # name[0]
Line 8: movl %esp,%ecx         # %ecx = name
Line 9: cdq                    # %edx = 0
Line 10: movb $0x0b,%al
Line 11: int $0x80             # invoke execve(name[0], name, 0)
```

Challenge 2

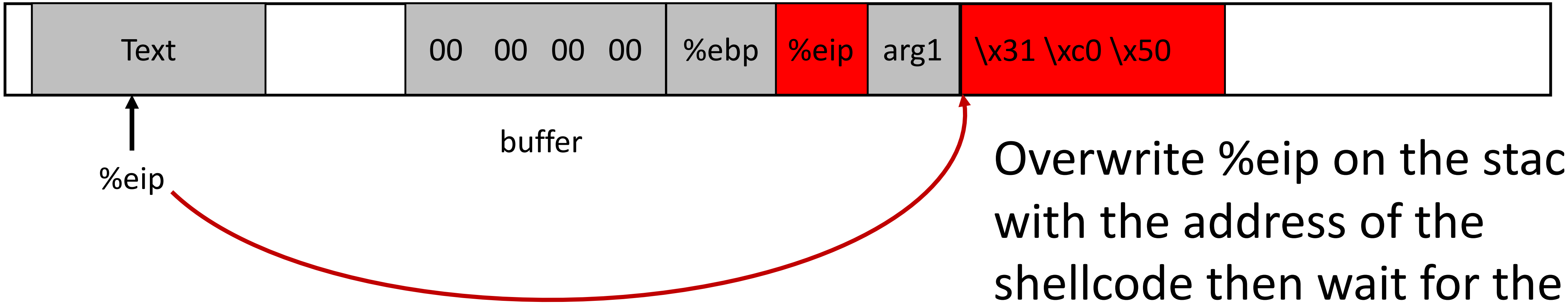
- We can only write to memory sequentially
- We need to have a way to execute code from code that's already executing



How do we execute our shellcode?

Challenge 2

- We can only write to memory sequentially
- We need to have a way to execute code from code that's already executing

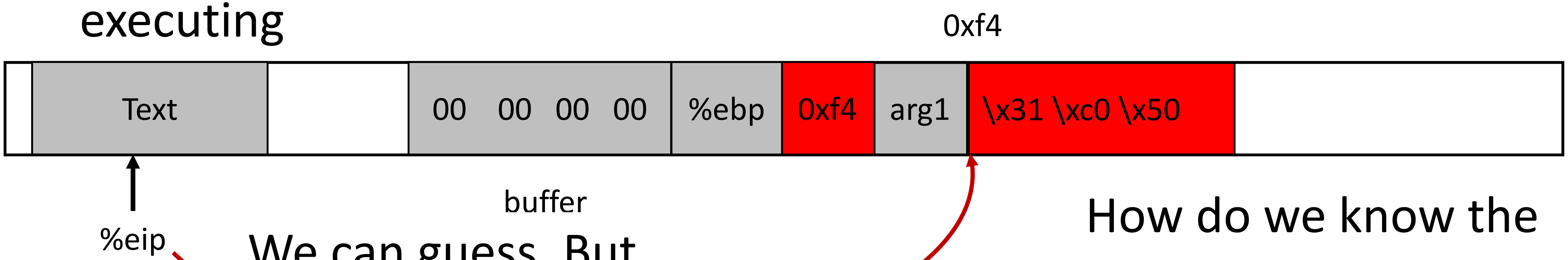


Overwrite %eip on the stack with the address of the shellcode then wait for the function call to return

Challenge 2

- We can only write to memory sequentially (cannot skip specific regions)

- We need to have a way to execute code from code that's already executing



We can guess. But what if we're wrong?

How do we know the address to use?

Possibly panic if invalid instruction (i.e. data)

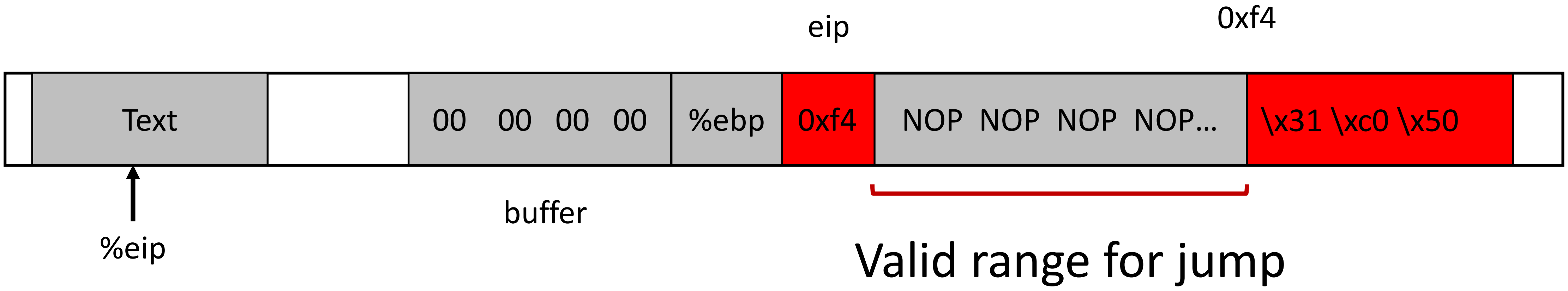
Challenge 3

- We need to determine the location of the return address on the stack
  - Where %eip is saved
  - We don’t know how far %ebp is from the buffer

- We could brute force the address space and try all 2^32 addresses on a 32-bit machine
- Can be done more efficiently if address space layout randomization (ASLR) is disabled
  - The stack will always start from a fixed location
  - Most programs don’t have a deep stack

NOP Sleds

- Inserting NOPs in the malicious code can improve our chances
- A NOP will just increment the value of the %eip and move to the next instruction
- Chance of succeeding improves according to the number of inserted NOPs



END