Stack layout when calling functions

- What do we do when we *call* a function?
  - What data need to be stored?
  - Where do they go?

- How do we *return* from a function?
  - What data need to be *restored*?
  - Where do they come from?

Code examples

(see `~/UMD/examples/` in the VM)

Function call and return

```
int main() {
    ...
    func("Hey", 10, -3);
    ...
}
```

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

Function call and return

```
int main() {
    ...
    func("Hey", 10, -3);
    ...
}
```

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

- Caller
  - push args (in reverse)
  - push return addr (%eip + …)
  - jmp to func address
  - pop args
- Callee
  - push old frame ptr (%ebp)
  - set %ebp to stack top (%esp)
  - push local variables
  - . . .
  - restore old stack frame
    %esp = %ebp; pop ebp
  - jmp return addr: pop %eip

Function call and return

```
int main() {
    ...
    func("Hey", 10, -3);
    ...
}
```

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

- Caller
  - push args (in reverse)
  - push return addr (%eip + …)
  - jmp to func address
  - pop args
- Callee
  - push old frame ptr (%ebp)
  - set %ebp to stack top (%esp)
  - push local variables
  - . . .
  - restore old stack frame
    %esp = %ebp; pop ebp
  - jmp return addr: pop %eip

- Why point %ebp to stack top at start of callee body?
- Allows the compiled callee code to easily access args and local variables (via offsets from %ebp)
- Requires saving old %ebp

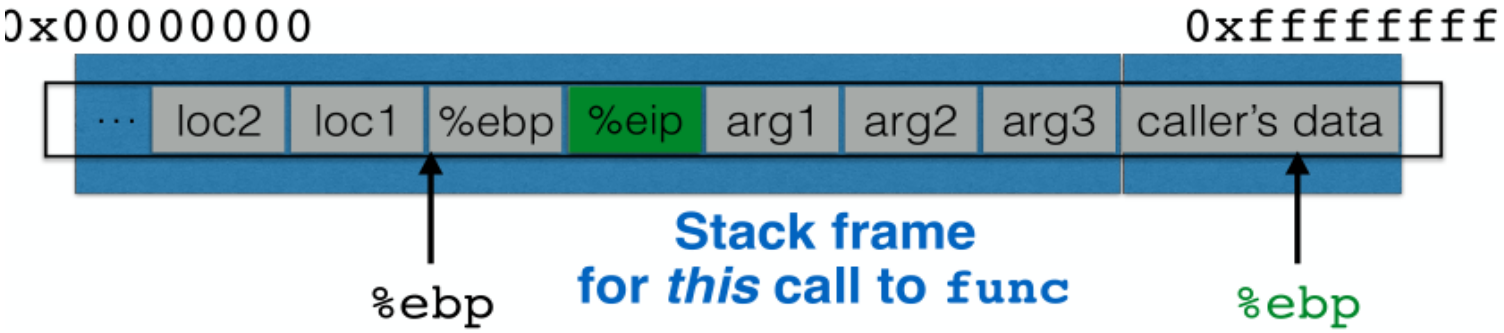
Function call and return

```
int main() {
    ...
    func("Hey", 10, -3);
    ...
}
```

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

- Caller
  - push args (in reverse)
  - push return addr (%eip + …)
  - jmp to func address
  - pop args

- Callee
  - push old frame ptr (%ebp)
  - set %ebp to stack top (%esp)
  - push local variables
  - . . .
  - restore old stack frame
    %esp = %ebp; pop ebp
  - jmp return addr: pop %eip



Stack and functions: Summary

Calling function:

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. **Jump to the function's address**

Called function:

4. **Push the old frame pointer** onto the stack: %ebp
5. **Set frame pointer** %ebp to where the end of the stack is right now: %esp
6. **Push local variables** onto the stack; access them as offsets from %ebp

Returning function:

7. **Reset the previous stack frame**: %ebp = (%ebp) /* copy it off first */
8. **Jump back to return address**: %eip = 4(%ebp) /* use the copy */

Buffer overflows

Buffer overflows from 10,000 ft

- Buffer =
  - Contiguous set of a given data type
  - Common in C
    - All strings are buffers of **`char`**'s
- Overflow =
  - Put more into the buffer than it can hold
- Where does the extra data go?
- Well now that you're experts in memory layouts…

A buffer overflow example

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

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

A buffer overflow example

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

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

A buffer overflow example

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

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

A buffer overflow example

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

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

A buffer overflow example

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

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

A buffer overflow example

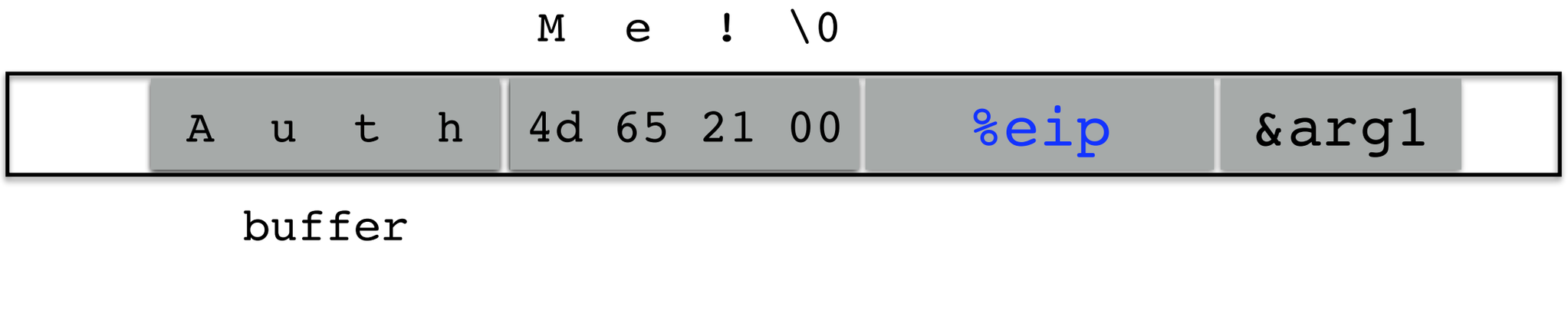
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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

A buffer overflow example

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

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

A buffer overflow example

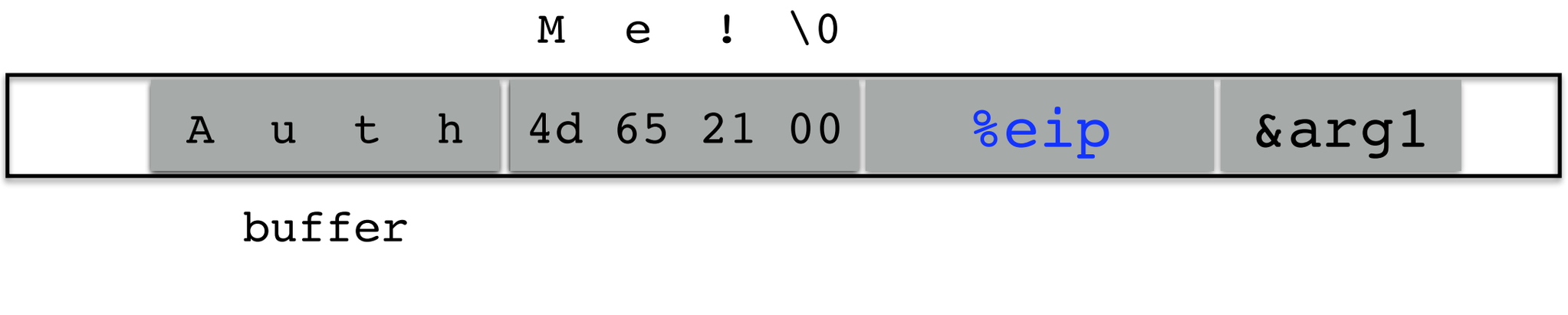
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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



A buffer overflow example

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

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

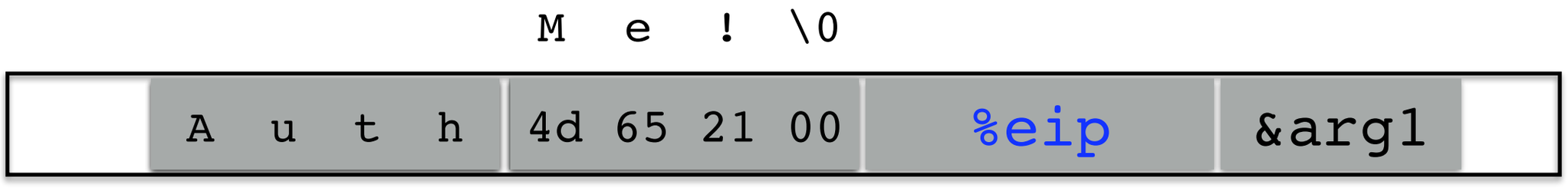
Upon return, sets `%ebp` to `0x0021654d`



A buffer overflow example

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

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

Upon return, sets `%ebp` to `0x0021654d`



SEGFAULT (0x00216551)

A buffer overflow example

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

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

A buffer overflow example

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

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

A buffer overflow example

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

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

A buffer overflow example

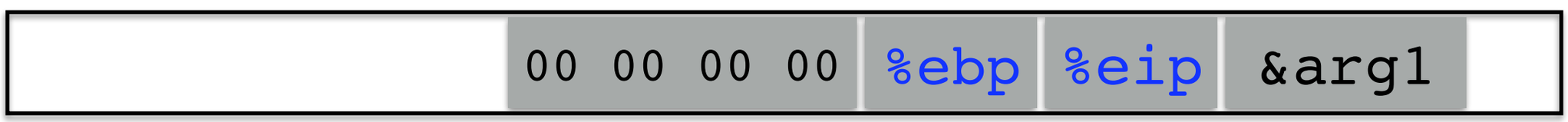
```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

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

A buffer overflow example

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

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

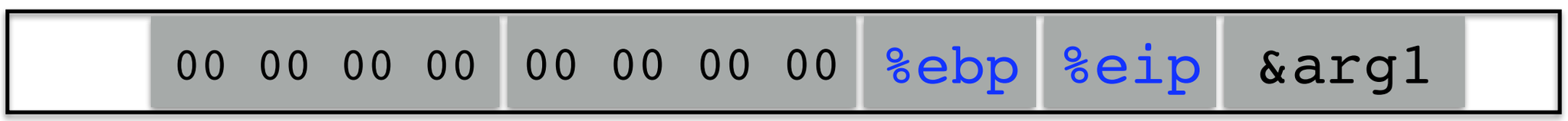
A buffer overflow example

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

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



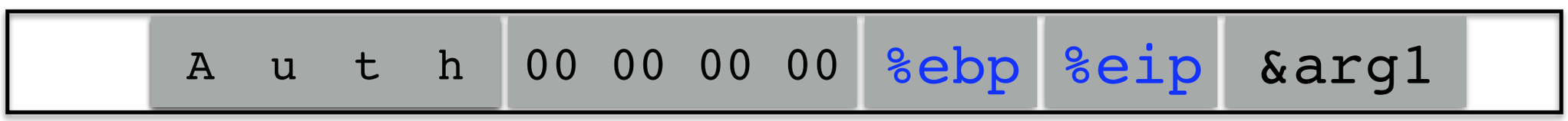
A buffer overflow example

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

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



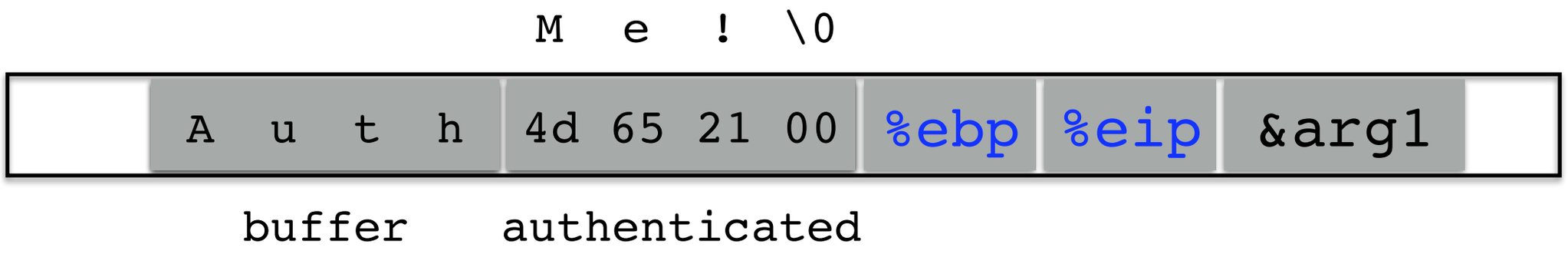
A buffer overflow example

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

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



A buffer overflow example

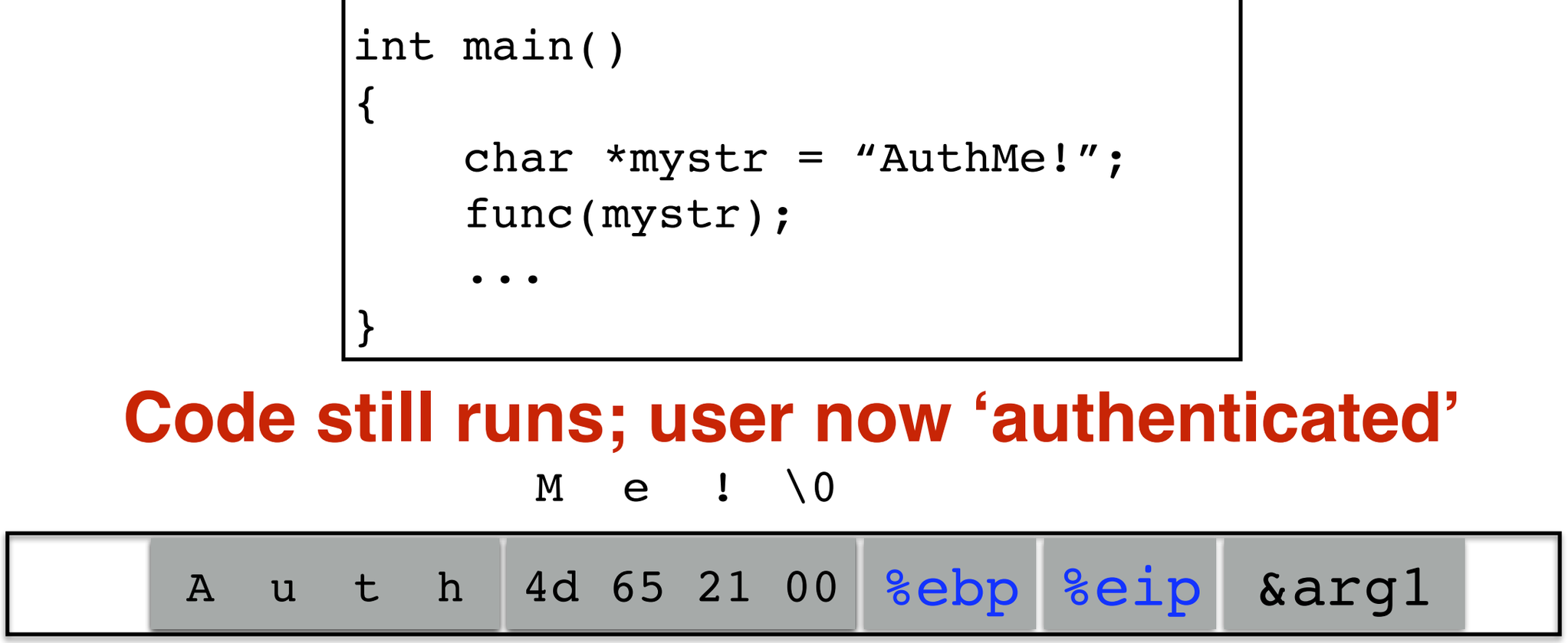
```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

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



A buffer overflow example

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

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

Code still runs; user now ‘authenticated’



```
void vulnerable()
{
    char buf[80];
    gets(buf);
}
```

```
void vulnerable()
{
    char buf[80];
    gets(buf);
}
```

```
void still_vulnerable()
{
    char *buf = malloc(80);
    gets(buf);
}
```

```
void safe()
{
    char buf[80];
    fgets(buf, 64, stdin);
}
```

```
void safe()
{
    char buf[80];
    fgets(buf, 64, stdin);
}
```

```
void safer()
{
    char buf[80];
    fgets(buf, sizeof(buf), stdin);
}
```

IE's Role in the Google-China War

By Richard Adhikari
TechNewsWorld
01/15/10 12:25 PM PT

The hack attack on Google that set off the company's ongoing standoff with China appears to have come through a zero-day flaw in Microsoft's Internet Explorer browser. Microsoft has released a security advisory, and researchers are hard at work studying the exploit. The attack appears to consist of several files, each a different piece of malware.

Computer security companies are scurrying to cope with the fallout from the Internet Explorer (IE) flaw that led to cyberattacks on Google and its corporate and individual customers.

The zero-day attack that exploited IE is part of a lethal cocktail of malware that is keeping researchers very busy.

"We're discovering things on an up-to-the-minute basis, and we've seen about a dozen files dropped on infected PCs so far," Dmitri Alperovitch, vice president of research at McAfee Labs, told TechNewsWorld.

The attacks on Google, which appeared to originate in China, have sparked a feud between the Internet giant and the nation's government over censorship, and it could result in Google pulling away from its business dealings in the country.

Pointing to the Flaw

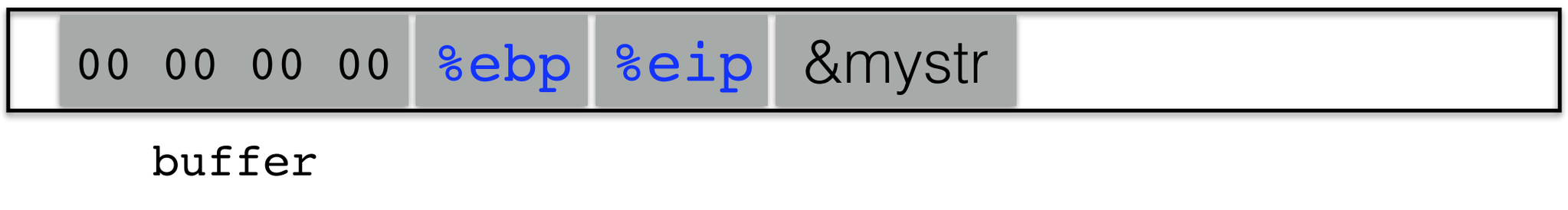
The vulnerability in IE is an invalid pointer reference, Microsoft said in security advisory 979352, which it issued on Thursday. Under certain conditions, the invalid pointer can be accessed after an object is deleted, the advisory states. In specially crafted attacks, like the ones launched against Google and its customers, IE can allow remote execution of code when the flaw is exploited.

User-supplied strings

- In these examples, we were providing our own strings
- But they come from users in myriad aways
  - Text input
  - Network packets
  - Environment variables
  - File input…

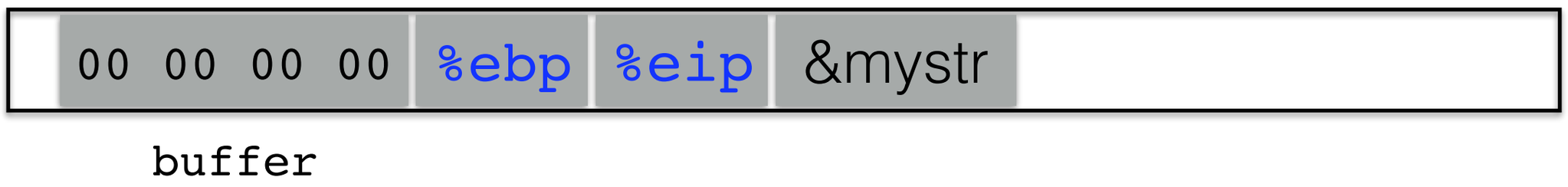
What's the worst that could happen?

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



What's the worst that could happen?

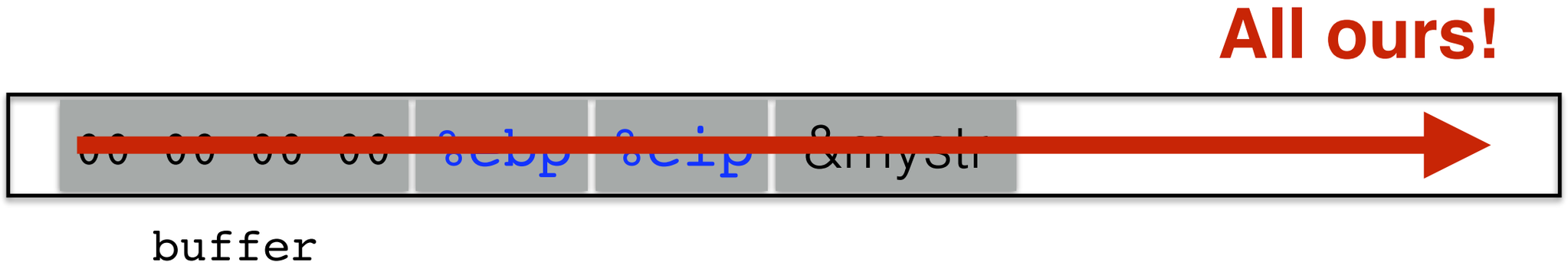
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



`strcpy` will let you write as much as you want (til a '\0')

What's the worst that could happen?

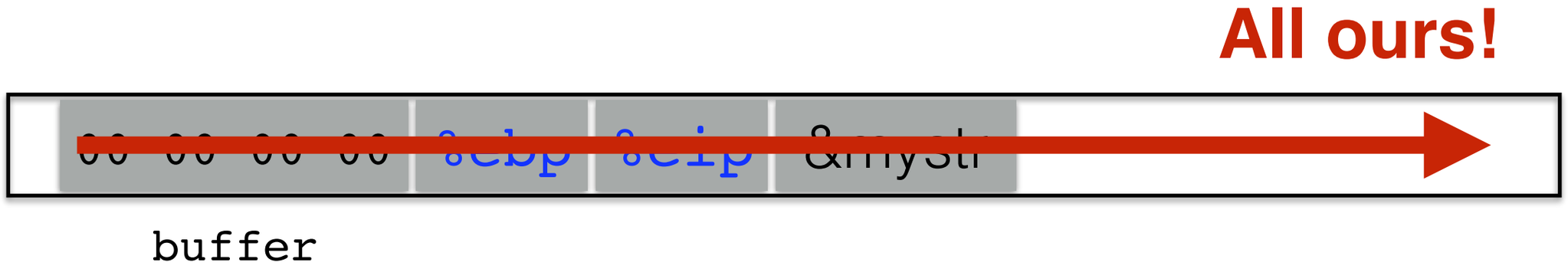
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



`strcpy` will let you write as much as you want (til a '\0')

What's the worst that could happen?

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



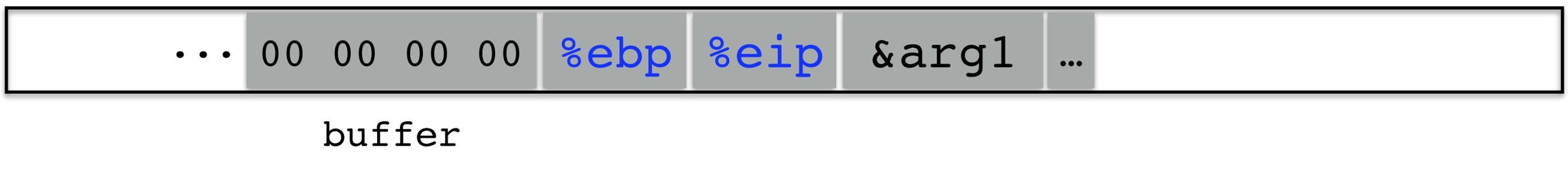
`strcpy` will let you write as much as you want (til a '\0')

What could you write to memory to wreak havoc?

Code injection

High-level idea

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



High-level idea

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

(1) Load our own code into memory

High-level idea

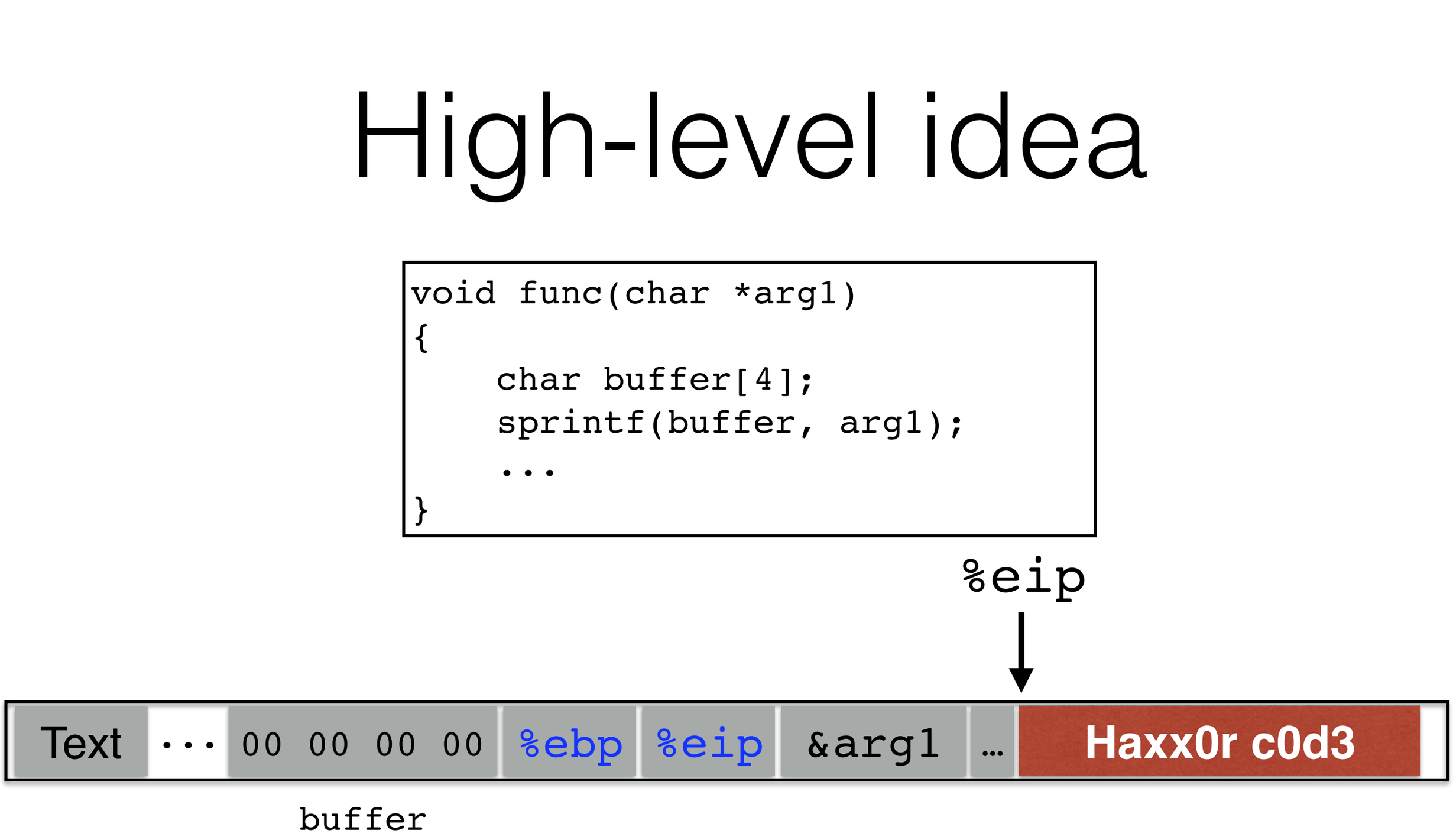
```
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```

(1) Load our own code into memory

(2) Somehow get `%eip` to point to it

High-level idea

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



(1) Load our own code into memory

(2) Somehow get `%eip` to point to it

High-level idea

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

(1) Load our own code into memory

(2) Somehow get `%eip` to point to it

This is nontrivial

- Pulling off this attack requires getting a few things really right (and some things sorta right)

- Think about what is tricky about the attack
  - The key to defending it will be to make the hard parts really hard

Challenge 1
Loading code into memory

- It **must be the machine code** instructions (i.e., already compiled and ready to run)

- We have to be careful in how we construct it:
  - It **can't contain** any **all-zero bytes**
    - Otherwise, sprintf / gets / scanf / … will stop copying
    - How to write assembly to never contain a full zero byte?
  - It **can't use the loader** (we're injecting)
    - How to find addresses we need?

What code to run?

- One goal: **general-purpose shell**
  - Command-line prompt that gives attacker **general access to the system**
- The code to launch a shell is called **shellcode**
- Other stuff you could do?

Shellcode

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

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
...
```

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
...
```

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

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”
...
```

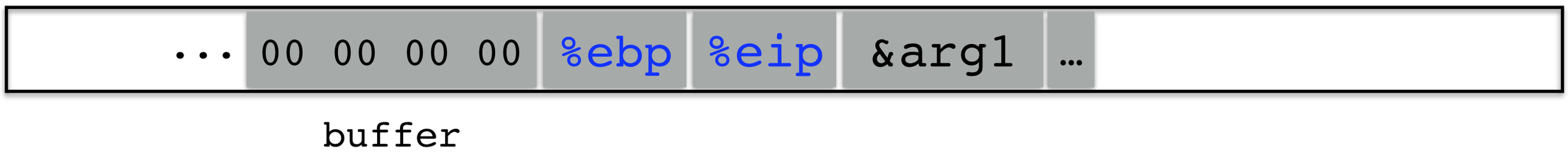
Machine code

(Part of) your input

Challenge 2

Getting our injected code to run

- ***All we can do is write to memory from `buffer` onward***
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running



Thoughts?

Challenge 2

Getting our injected code to run

- ***All we can do is write to memory from `buffer` onward***
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running

Thoughts?

Challenge 2

Getting our injected code to run

- ***All we can do is write to memory from `buffer` onward***
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running

Thoughts?

Challenge 2

Getting our injected code to run

- ***All we can do is write to memory from `buffer` onward***
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running

Thoughts?

Challenge 2

Getting our injected code to run

- ***All we can do is write to memory from `buffer` onward***
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running

Thoughts?

Challenge 2

Getting our injected code to run

- ***All we can do is write to memory from `buffer` onward***
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running

Thoughts?

Stack and functions: Summary

Calling function:

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. **Jump to the function's address**

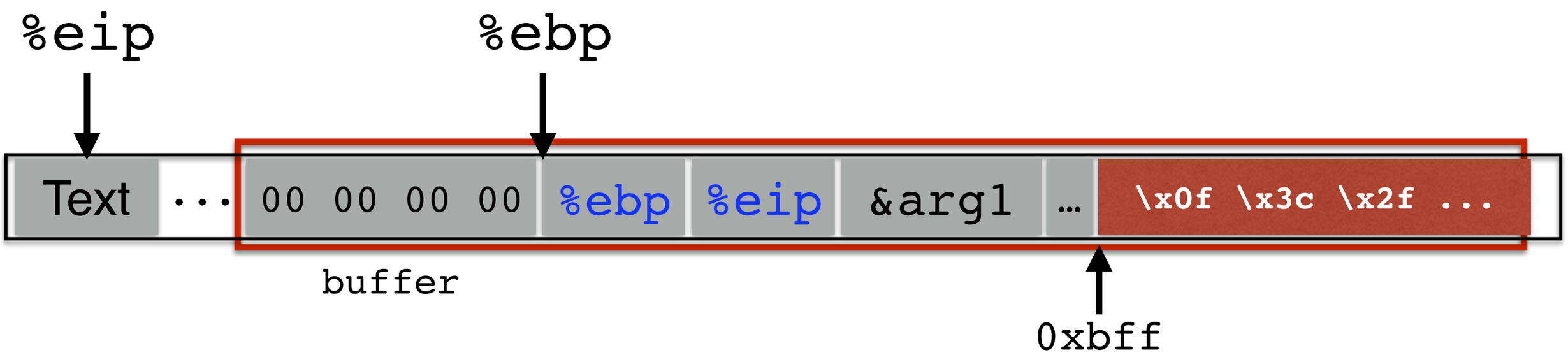
Called function:

4. **Push the old frame pointer** onto the stack: %ebp
5. **Set frame pointer** %ebp to where the end of the stack is right now: %esp
6. **Push local variables** onto the stack; access them as offsets from %ebp

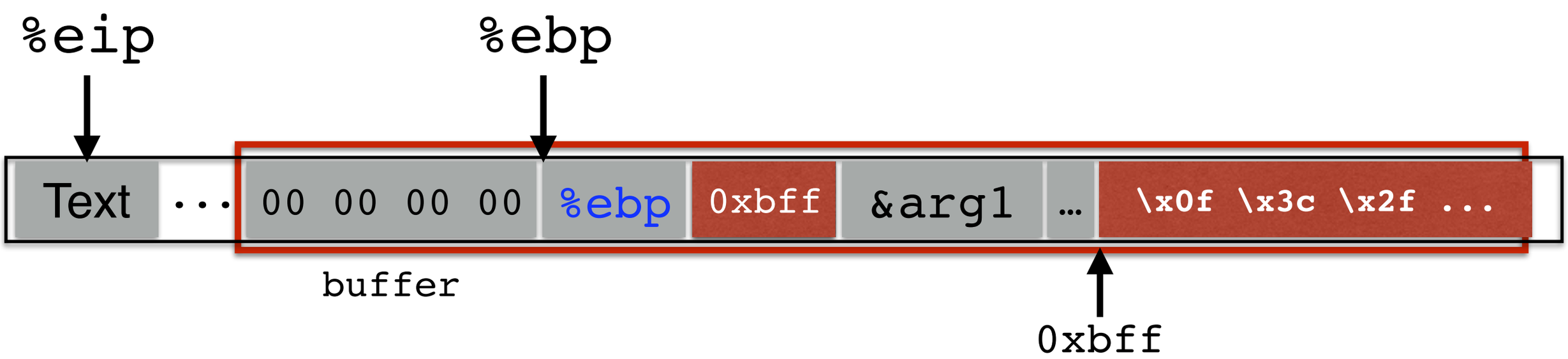
Returning function:

7. **Reset the previous stack frame**: %ebp = (%ebp)
8. **Jump back to return address**: %eip = 4(%ebp)

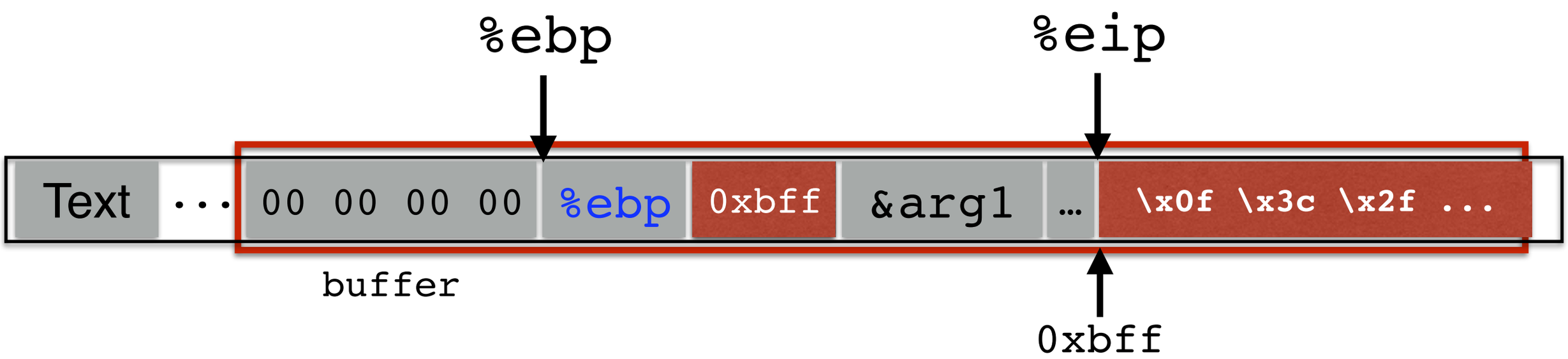
Hijacking the saved `%eip`



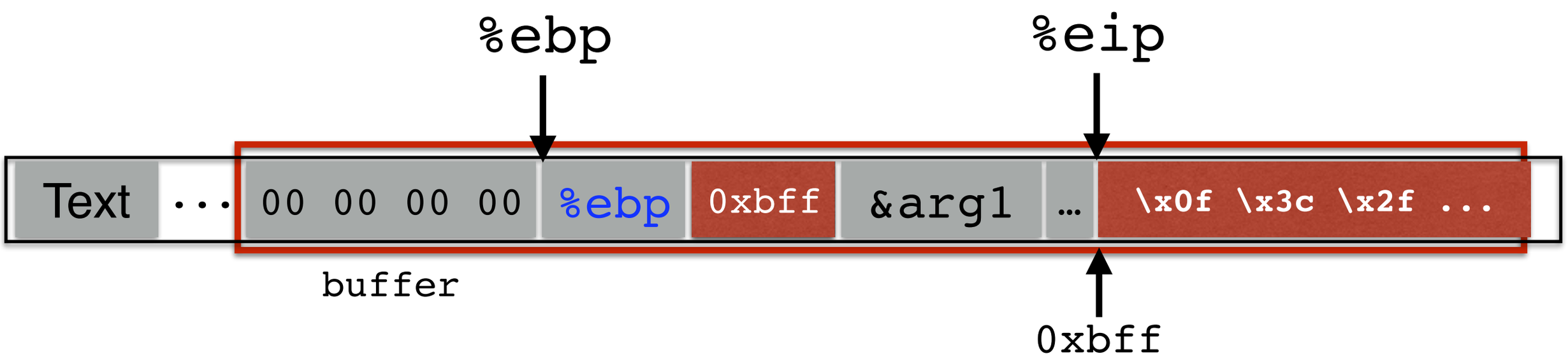
Hijacking the saved `%eip`



Hijacking the saved `%eip`



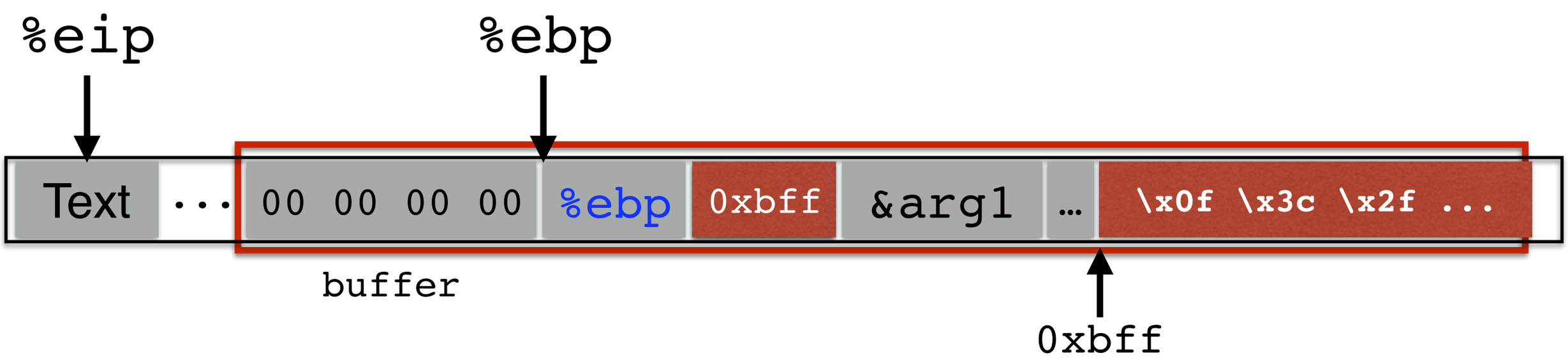
Hijacking the saved `%eip`



But how do we know the address?

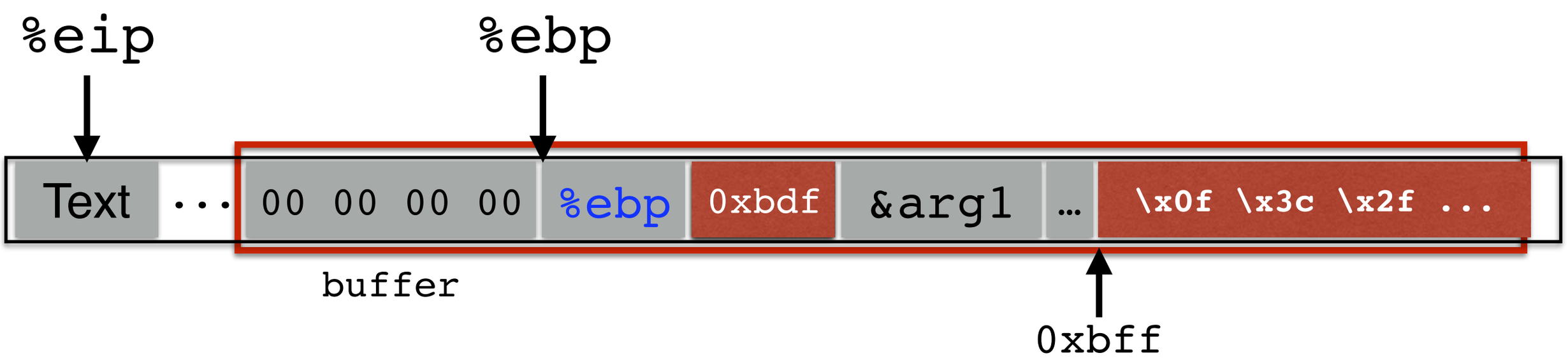
Hijacking the saved `%eip`

What if we are wrong?



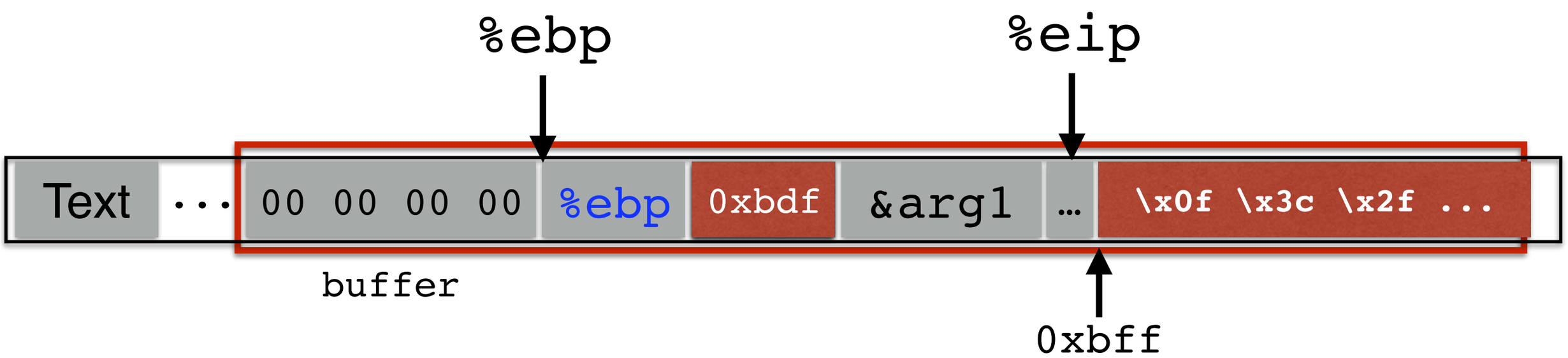
Hijacking the saved `%eip`

What if we are wrong?



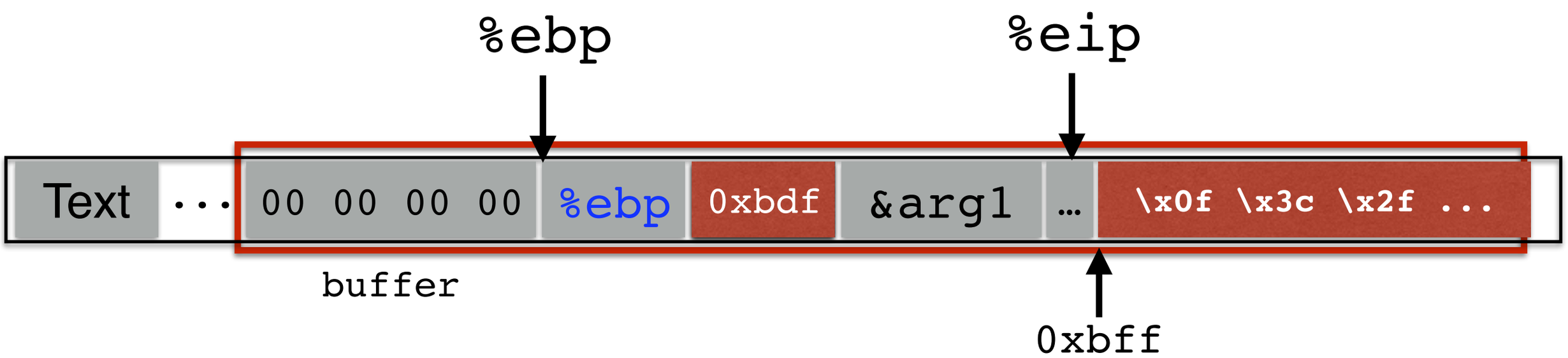
Hijacking the saved `%eip`

What if we are wrong?



Hijacking the saved `%eip`

What if we are wrong?



This is most likely data, so the CPU will panic (Invalid Instruction)

Challenge 3

Finding the return address

Challenge 3

Finding the return address

- If we don't have access to the code, we don't know how far the buffer is from the saved **`%ebp`**

Challenge 3

Finding the 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!

Challenge 3

Finding the 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

Challenge 3

Finding the return address

- If we don't have access to the code, we don't know how far the buffer is from the saved **`%ebp`**

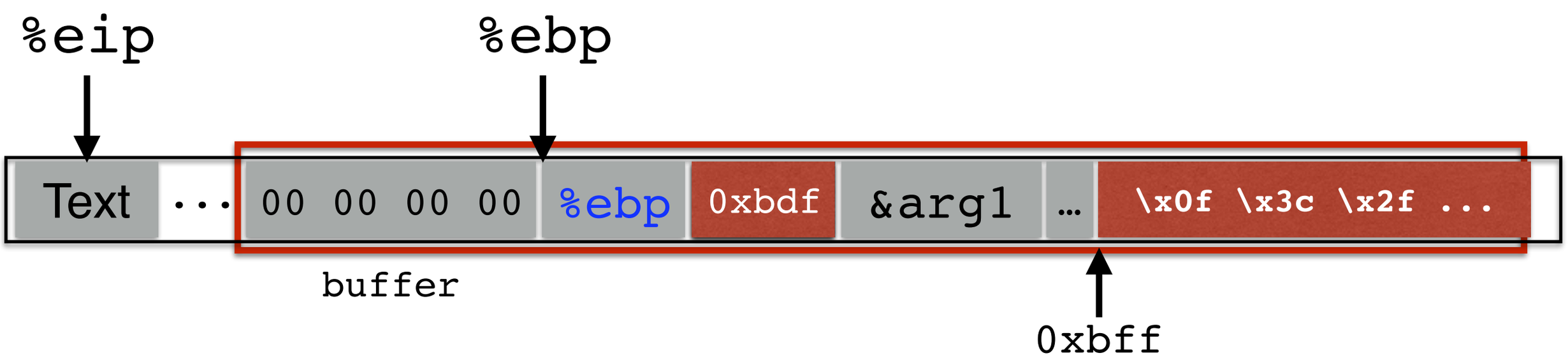
- One approach: just try a lot of different values!

- Worst case scenario: it's a 32 (or 64) bit memory space, which means 2^{32} (2^{64}) possible answers

- But without address randomization:
  - The stack always starts from the same, fixed address
  - The stack will grow, but usually it doesn't grow very deeply (unless the code is heavily recursive)

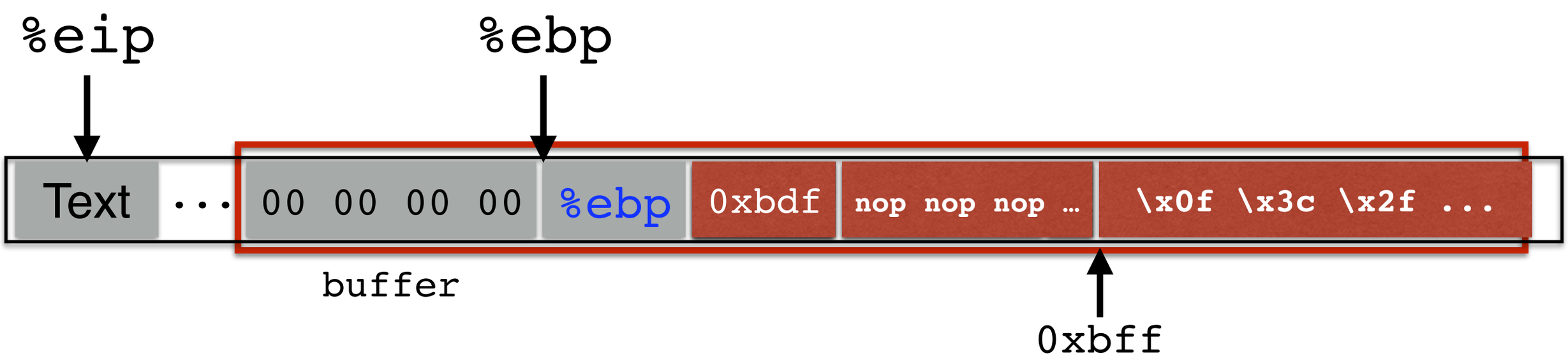
Improving our chances: `nop` sleds

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



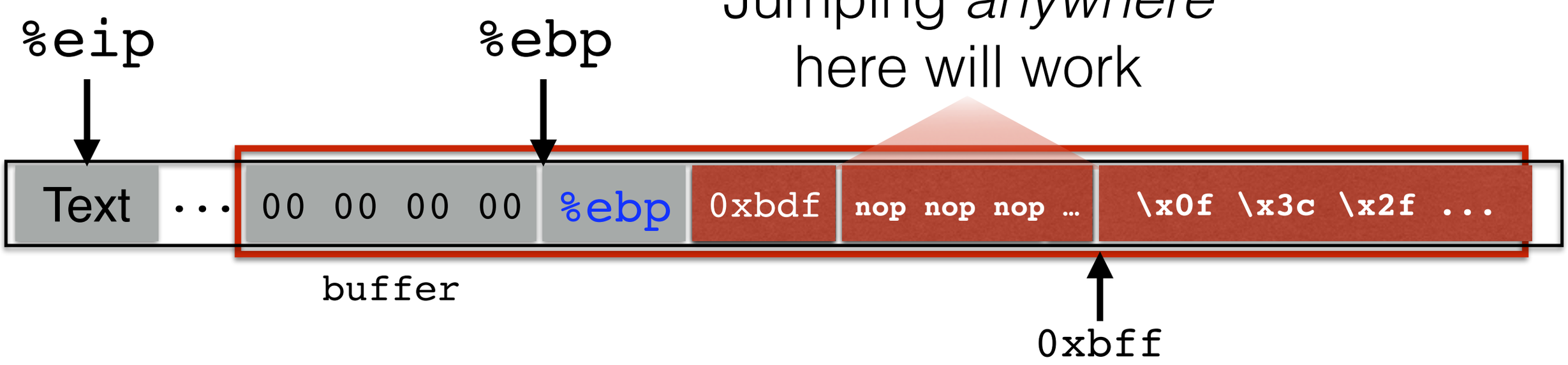
Improving our chances: `nop` sleds

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



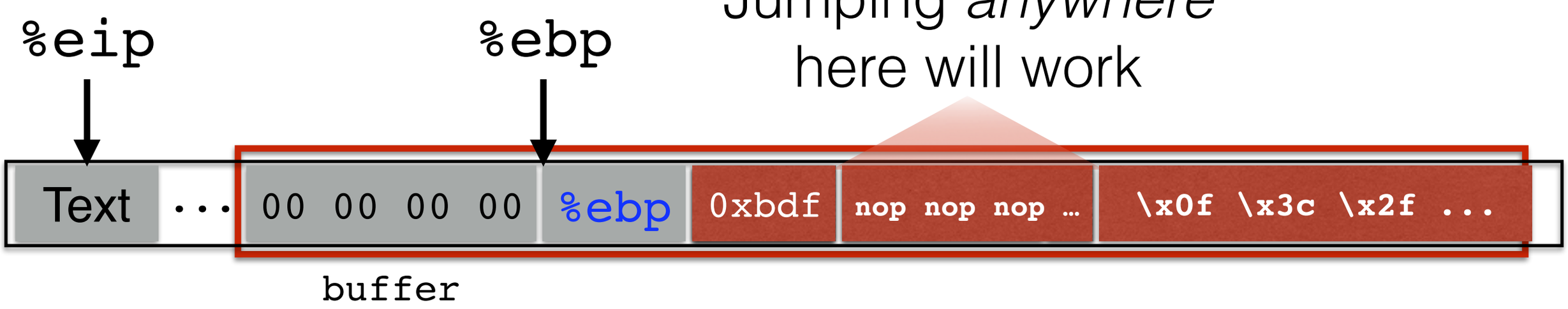
Improving our chances: `nop` sleds

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



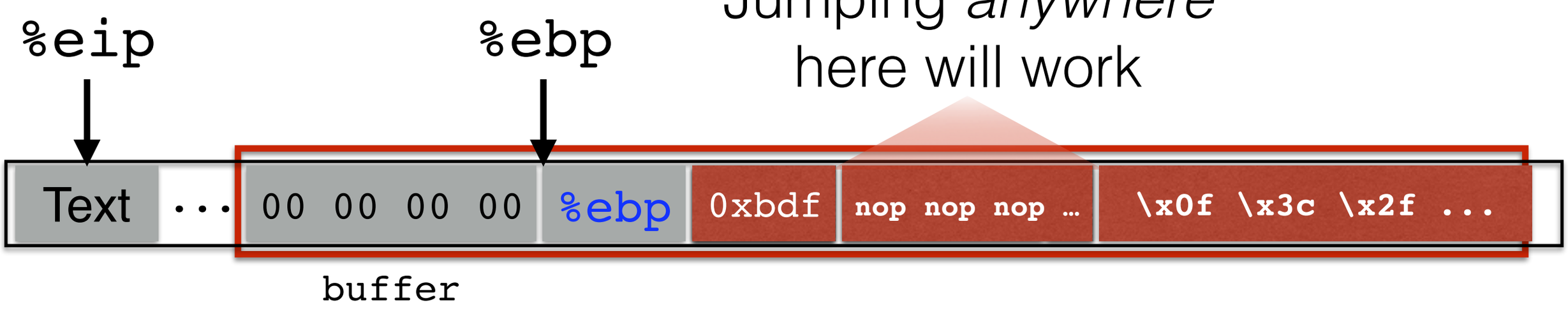
Improving our chances: `nop` sleds

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



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

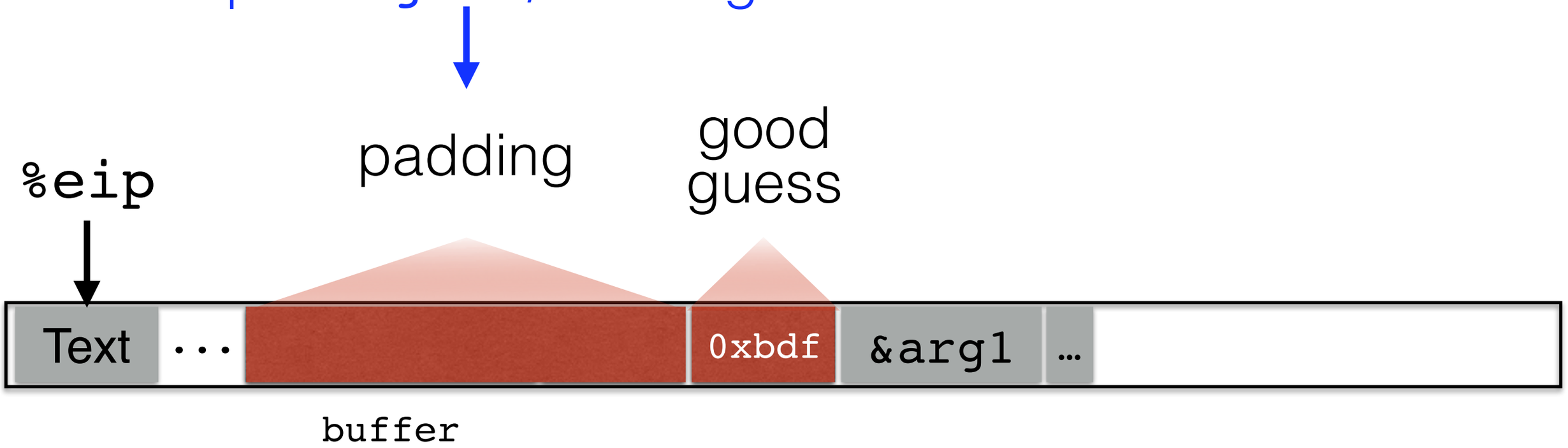
Putting it all together

Putting it all together

But it has to be *something*;
we have to start writing wherever
the input to **`gets`**/etc. begins.

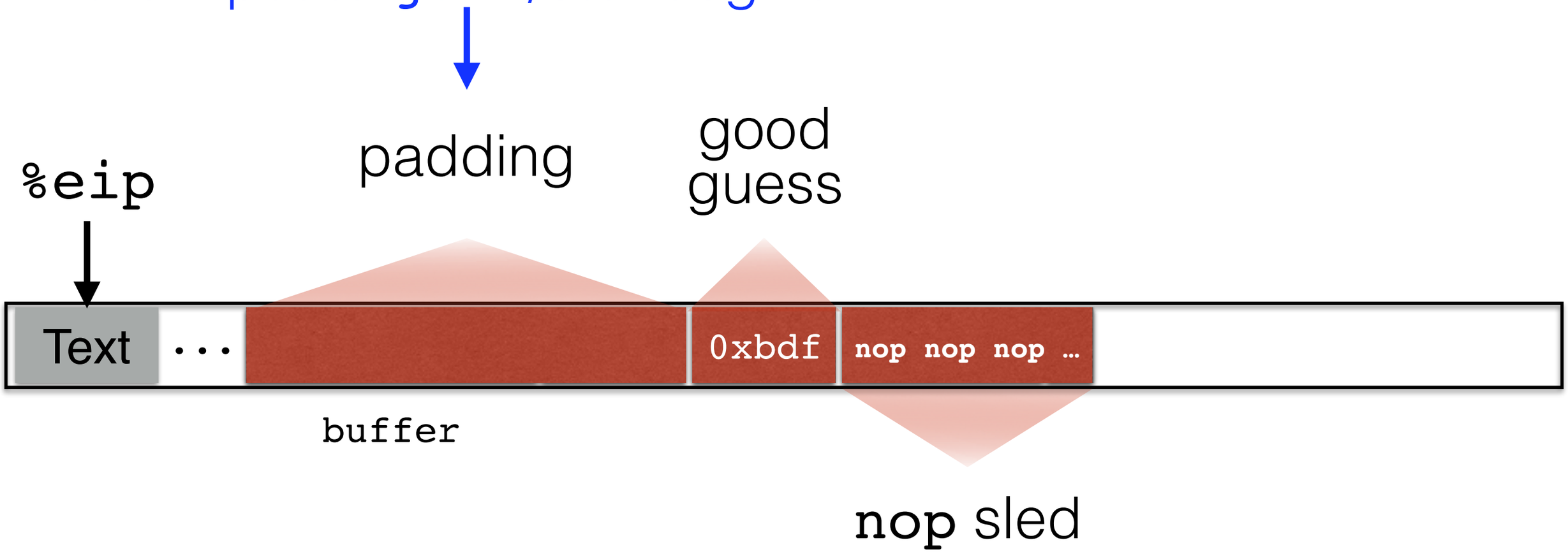
Putting it all together

But it has to be *something*;
we have to start writing wherever
the input to **`gets`**/etc. begins.



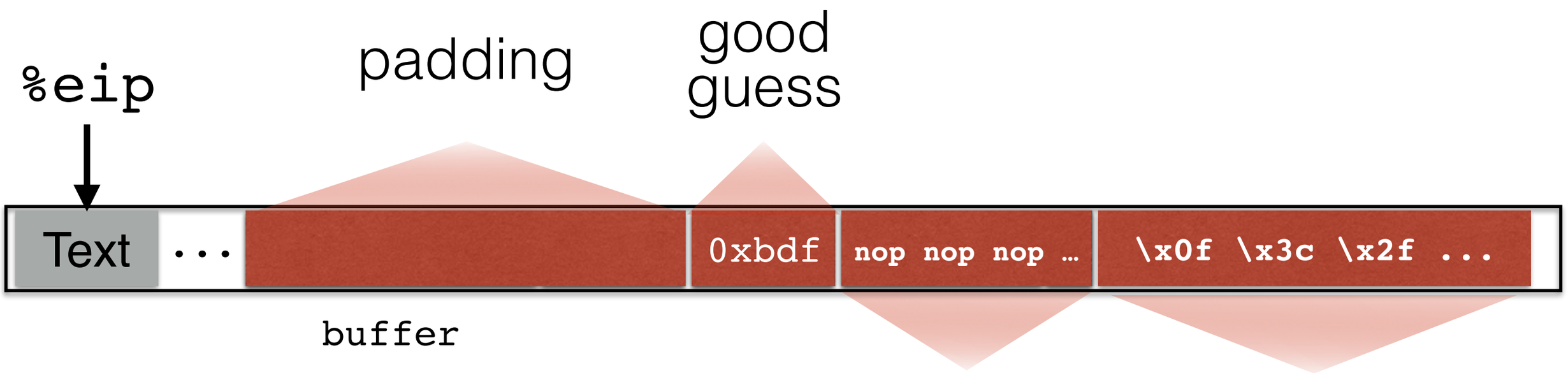
Putting it all together

But it has to be *something*;
we have to start writing wherever
the input to **`gets`**/etc. begins.



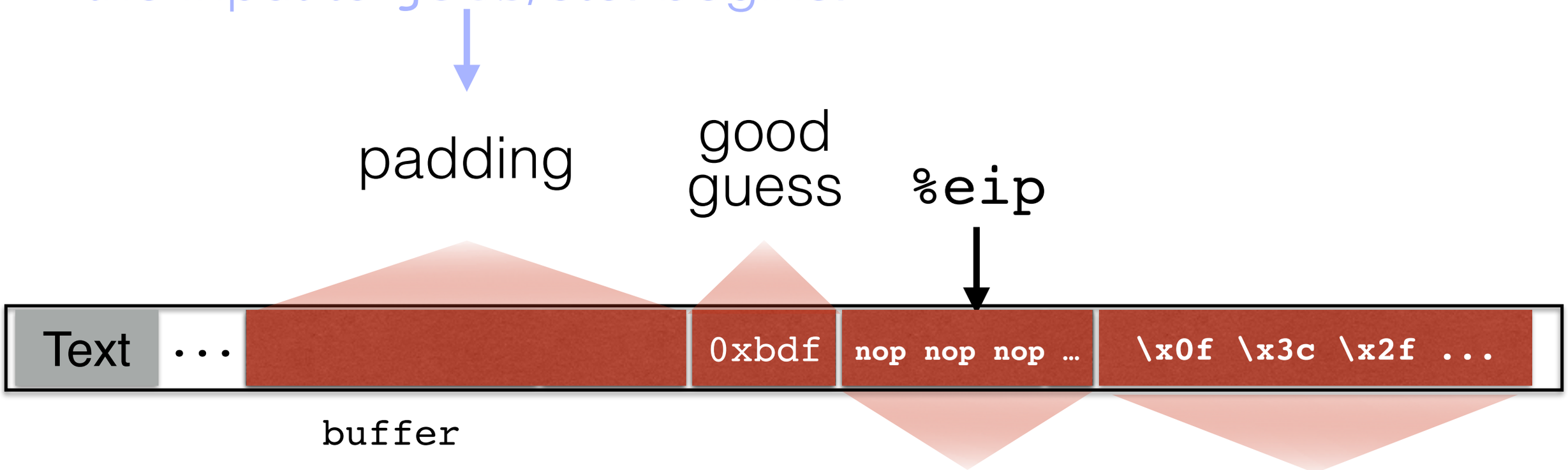
Putting it all together

But it has to be *something*;
we have to start writing wherever
the input to **`gets`**/etc. begins.



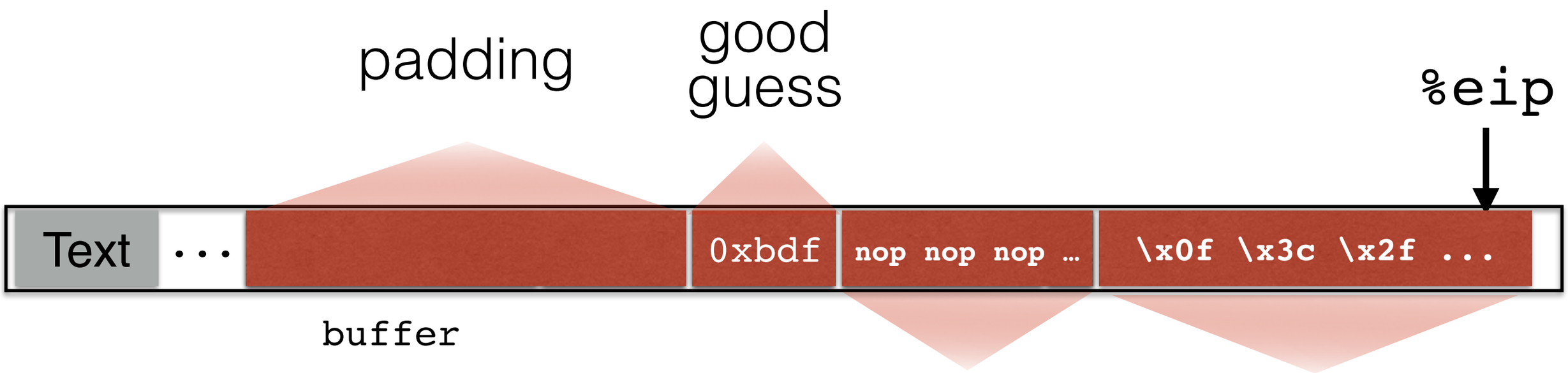
Putting it all together

But it has to be *something*;
we have to start writing wherever
the input to `gets`/etc. begins.



Putting it all together

But it has to be *something*;
we have to start writing wherever
the input to `gets`/etc. begins.



Setuid-root vulnerable code

- Let C be the vulnerable code (being attacked)
- Let P be the process executing C (when attacked)
- The attack, if successful, results in the attacker getting a shell that executes with the same privileges that P had in C
- Ideally the attacker wants a root-level shell
  - This happens if P is a root process
  - It also happens if P is a non-root process but C is a "setuid-root" executable.
    Why? Read on.

Privilege escalation

- An execution platform invariably has shared resources
  - cpus, memory, io devices, page maps, password files, etc
- Ordinary users need to access these resources, but only via special functions
  - cpu schedulers, device drivers, executable loaders, etc
- Ordinary-user code should be stopped at runtime from accessing these resources in any other way
  - eg, directly calling instructions that access the resources
- This is achieved by privilege-escalation mechanisms

Privilege escalation in hardware

- Cpu has different modes: kernel, user, ...
- Instructions (and addresses) are partitioned into privileged and nonprivileged
- Kernel code is executed with cpu in kernel mode
- User code is executed with cpu in user mode
- To use kernel-level resources, user code calls kernel functions via so-called system calls (aka syscalls)
  - syscall n: sets cpu mode to kernel; target address is fixed
- Return from system-call sets cpu to user mode

Permissions in Unix

- Each user has a userid (uid)
  - root user usually has uid 0
- Each process has a uid
  - set by the login process, and inherited after that
- Each file has:
  - owner: uid of the process that created the file
  - rwx (read/write/execute) attributes for owner
  - " " " " " for group
  - " " " " " for others
- Adequate to protect a user's files from other users
- Not adequate for accessing shared resources

Privilege escalation in Unix

- Each process has an effective userid (euid)
  - in addition to the previous uid (aka real userid (ruid))
  - euid initially equals ruid, but it can change temporarily
- Each executable file has a setuid bit
  - in addition to the previous owner uid and rwx attributes
  - if setuid is set: when a non-owner process executes the file, its euid is set to the owner id during execution of the file
    - euid is set to owner id when entering the executable
    - euid is set back to ruid when returning from the executable
- setuid-root: short for owned by root, setuid bit set
- So a setuid-root executable file can be executed with root permissions by a non-root user
  - Note analogy to a system call (at hardware level)

Privilege escalation example

- Let pwds be the file of [userid, password] entries
  - root is the owner of pwds and has rwx access to it
  - other users have no access
- Let passwd be an executable file that manages pwds
  - passwd's owner is root
  - *all* users have x-access
  - setuid bit is set
- passwd executed by root can access pwds because the executing process has ruid of root
- passwd executed by a non-root user *u* can access pwds because the executing process has euid of root

gdb: your new best friend

```
i f
```

Show **i**nfo about the current **f**rame
(prev. frame, locals/args, %ebp/%eip)

```
i r
```

Show **info** about **reg**isters
(%ebp, %eip, %esp, etc.)

```
x/<n> <addr>
```

E**x**amine <n> bytes of memory
starting at address <addr>

```
b <function>
```

Set a **b**reakpoint at <function>

```
s
```

step through execution (into calls)

- run, breakpoint, step, next, continue
  - stepi, nexti
- print, x
- backtrace, info [frame, registers, args, locals]
- list, disas