This time

We will continue

Buffer overflows

and other memory safety vulnerabilities

By looking at

Overflow Defenses

- Everything you've always wanted to know about `gdb` but were too afraid to ask
- Overflow defenses
- Other memory safety vulnerabilities

gdb: your new best friend

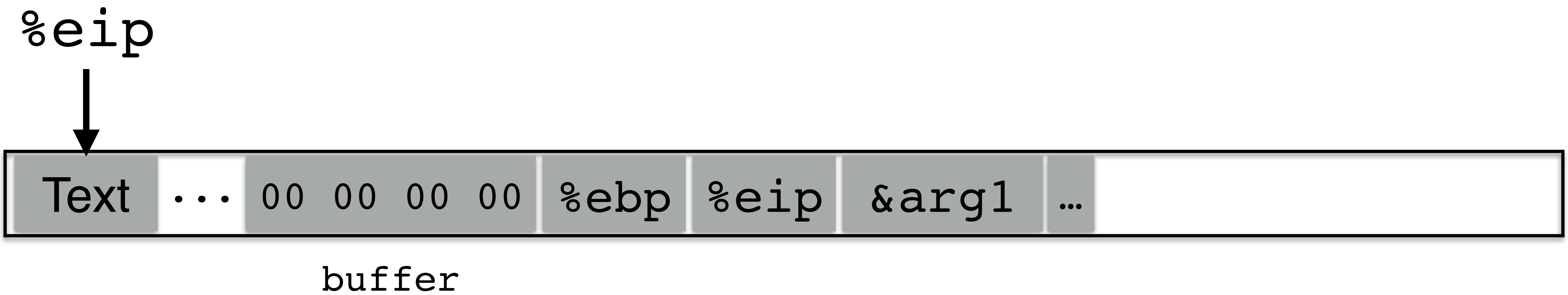
`i f`	Show **i**nfo about the current **f**rame (prev. frame, locals/args, %ebp/%eip)
`i r`	Show **info** about **reg**isters (%ebp, %eip, %esp, etc.)
`x/<n> <addr>`	E**x**amine <n> bytes of memory starting at address <addr>
`b <function>` `s`	Set a **b**reakpoint at <function> **s**tep through execution (into calls)

Recall our challenges

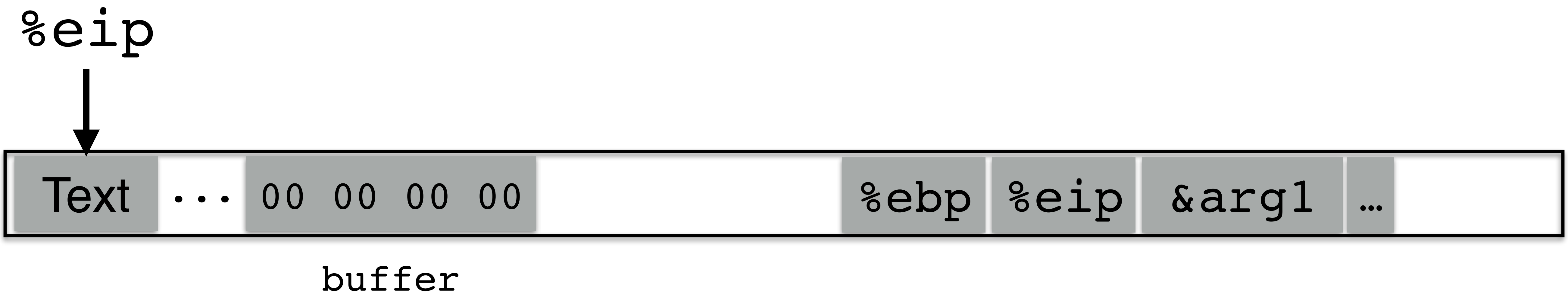
How can we make these even more difficult?

- Putting code into the memory (no zeroes)
- Getting %eip to point to our code (dist buff to stored eip)
- Finding the return address (guess the raw addr)

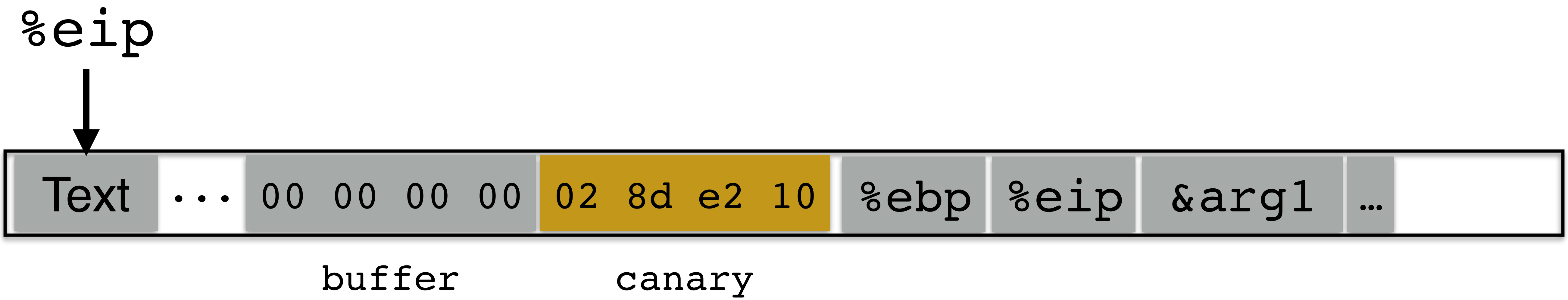
Detecting overflows with canaries



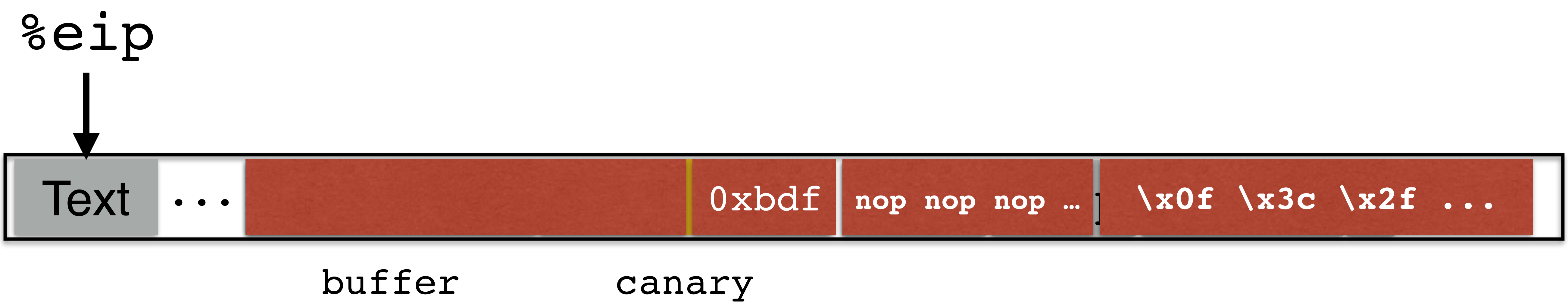
Detecting overflows with canaries



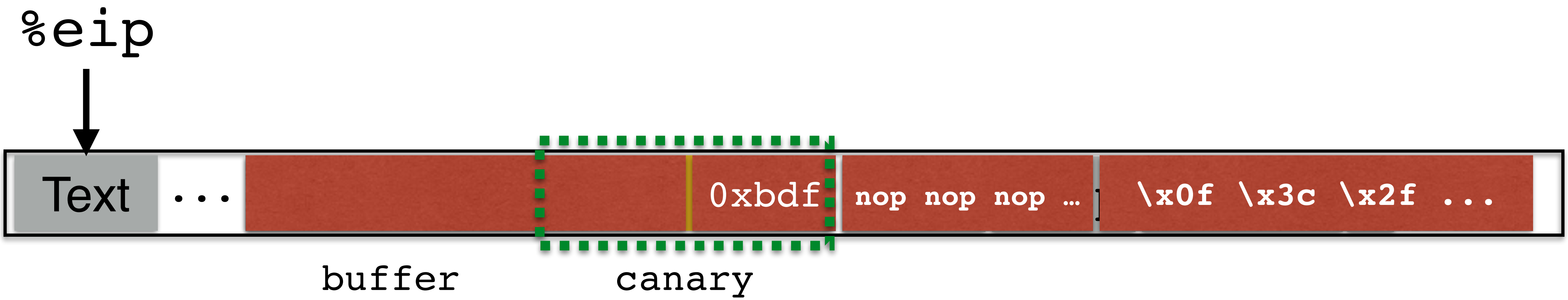
Detecting overflows with canaries



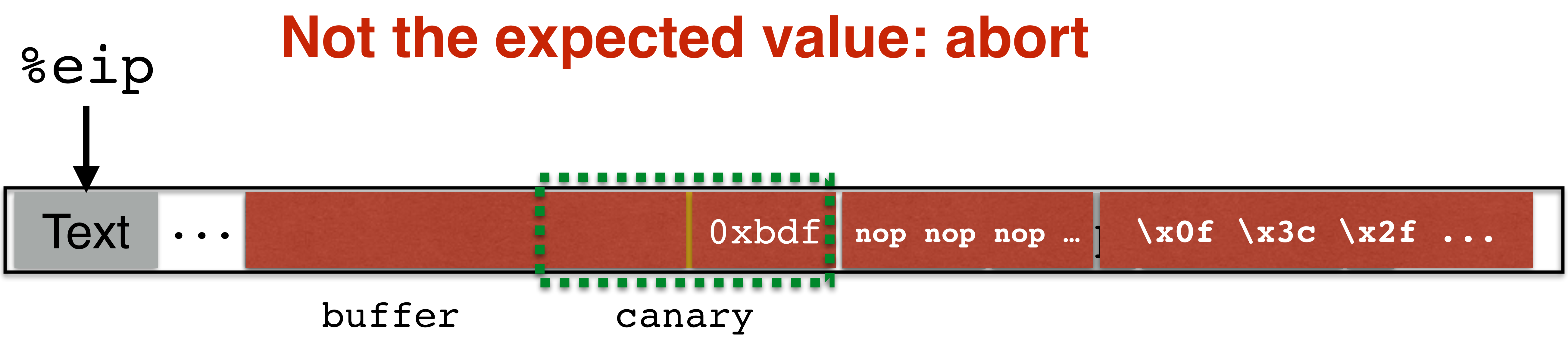
Detecting overflows with canaries



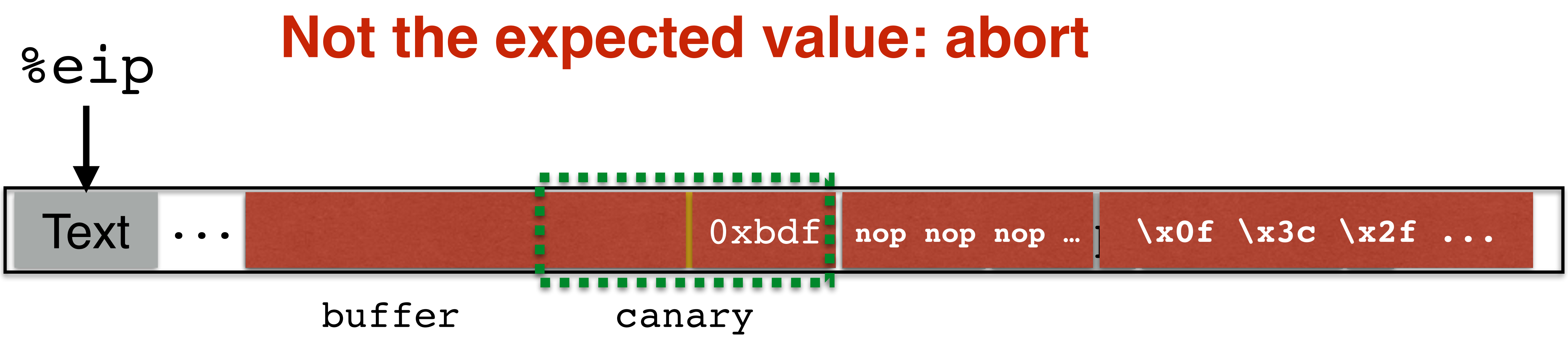
Detecting overflows with canaries



Detecting overflows with canaries



Detecting overflows with canaries



What value should the canary have?

Canary values

From StackGuard [Wagle & Cowan]

1. Terminator canaries (CR, LF, NULL, -1)
   - Leverages the fact that scanf etc. don't allow these
2. Random canaries
   - Write a new random value @ each process start
   - Save the real value somewhere in memory
   - Must write-protect the stored value
3. Random XOR canaries
   - Same as random canaries
   - But store canary XOR some control info, instead

Recall our challenges

How can we make these even more difficult?

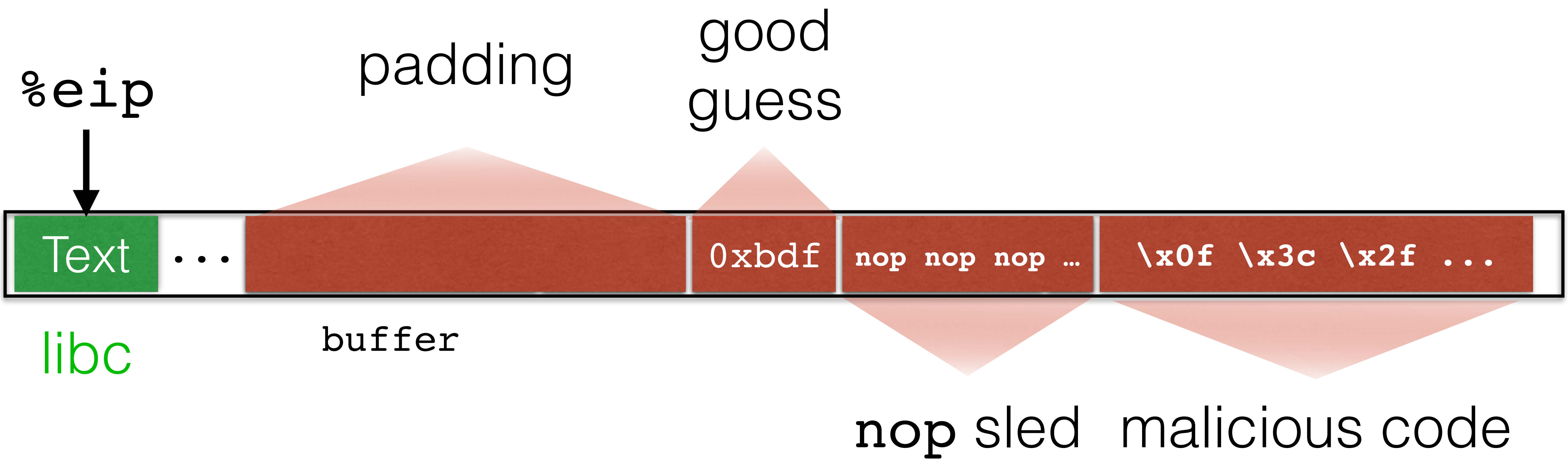
- Putting code into the memory (no zeroes)
  - Option: Make this detectable with canaries

- Getting %eip to point to our code(dist buff to stored eip)

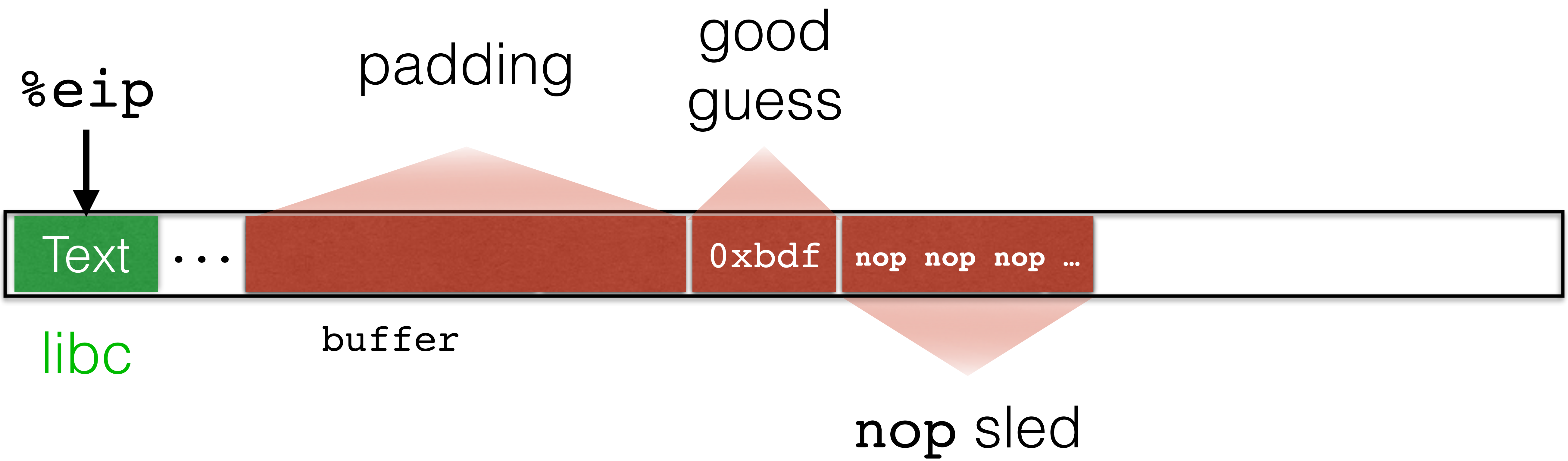
- Finding the return address (guess the raw addr)

More next time…

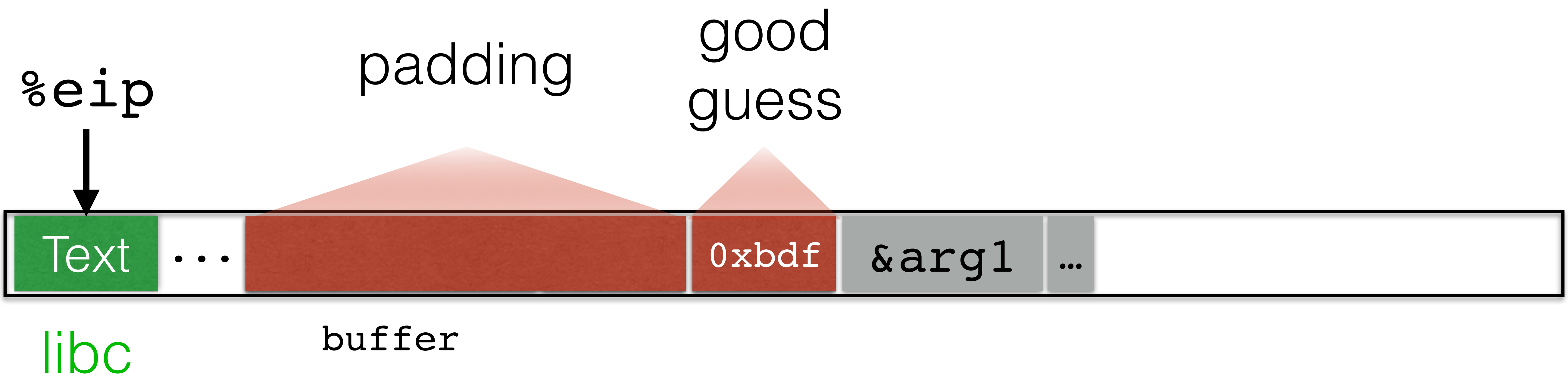
Return-to-libc



Return-to-libc

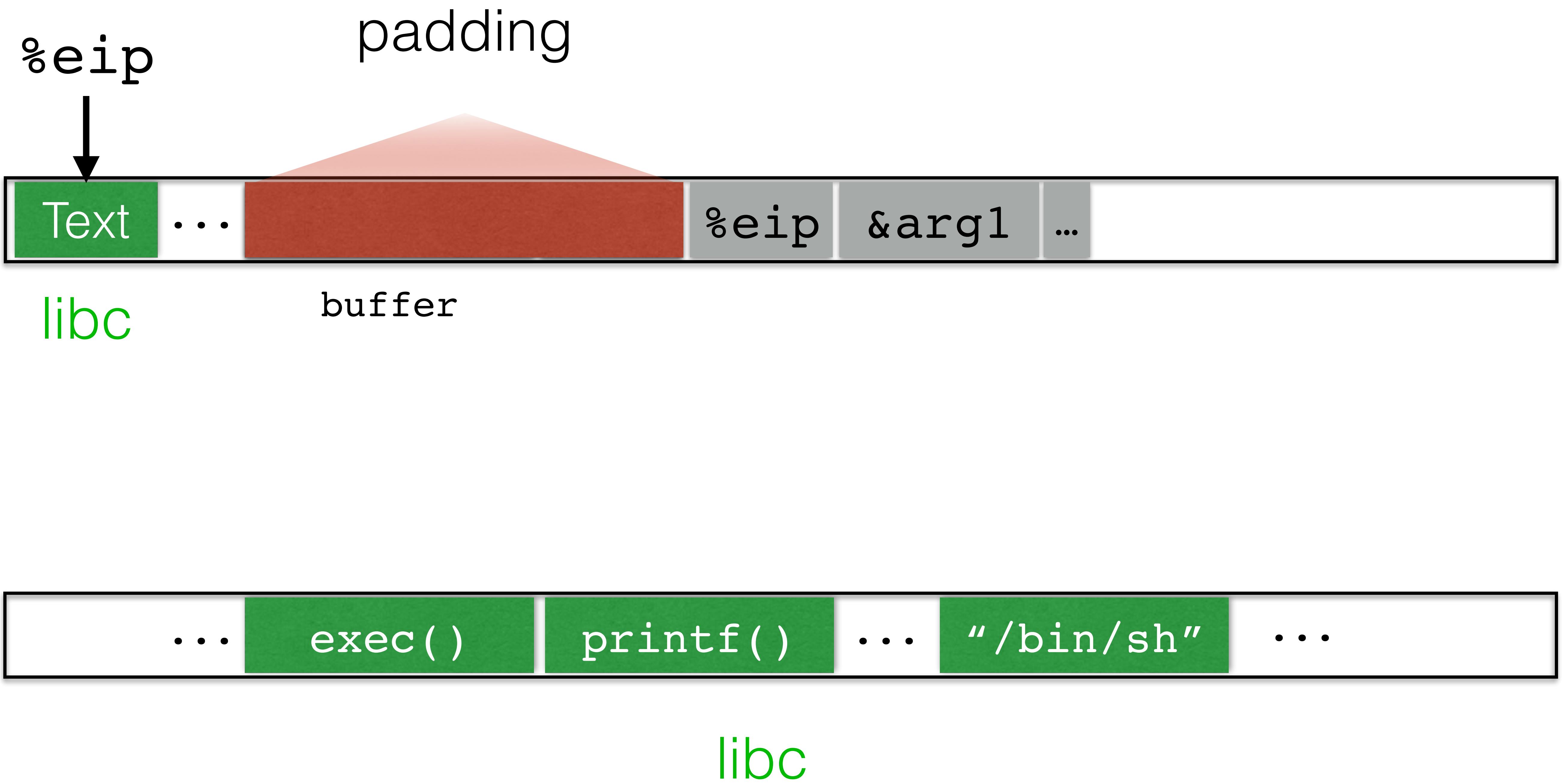


Return-to-libc

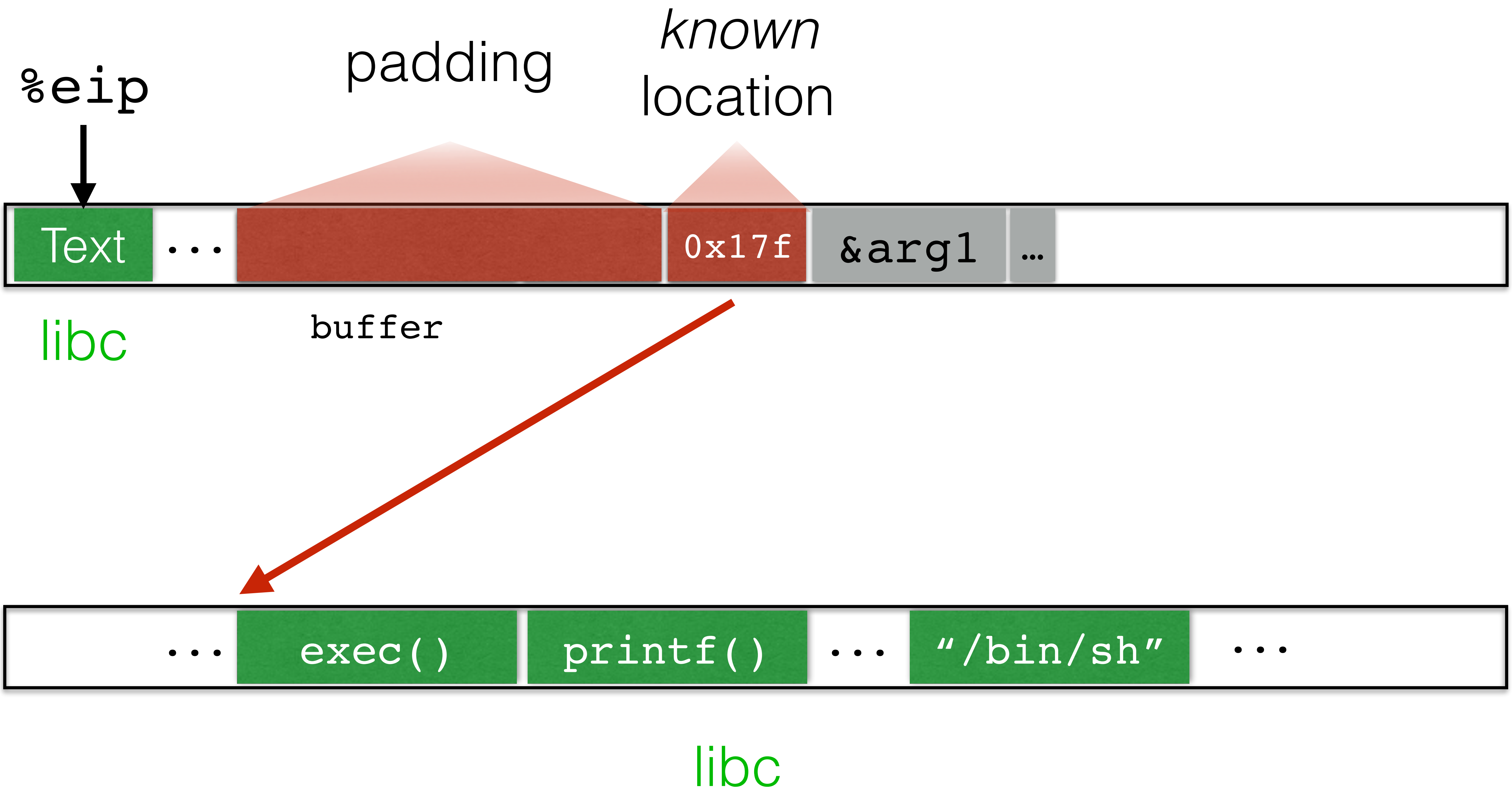


Return-to-libc

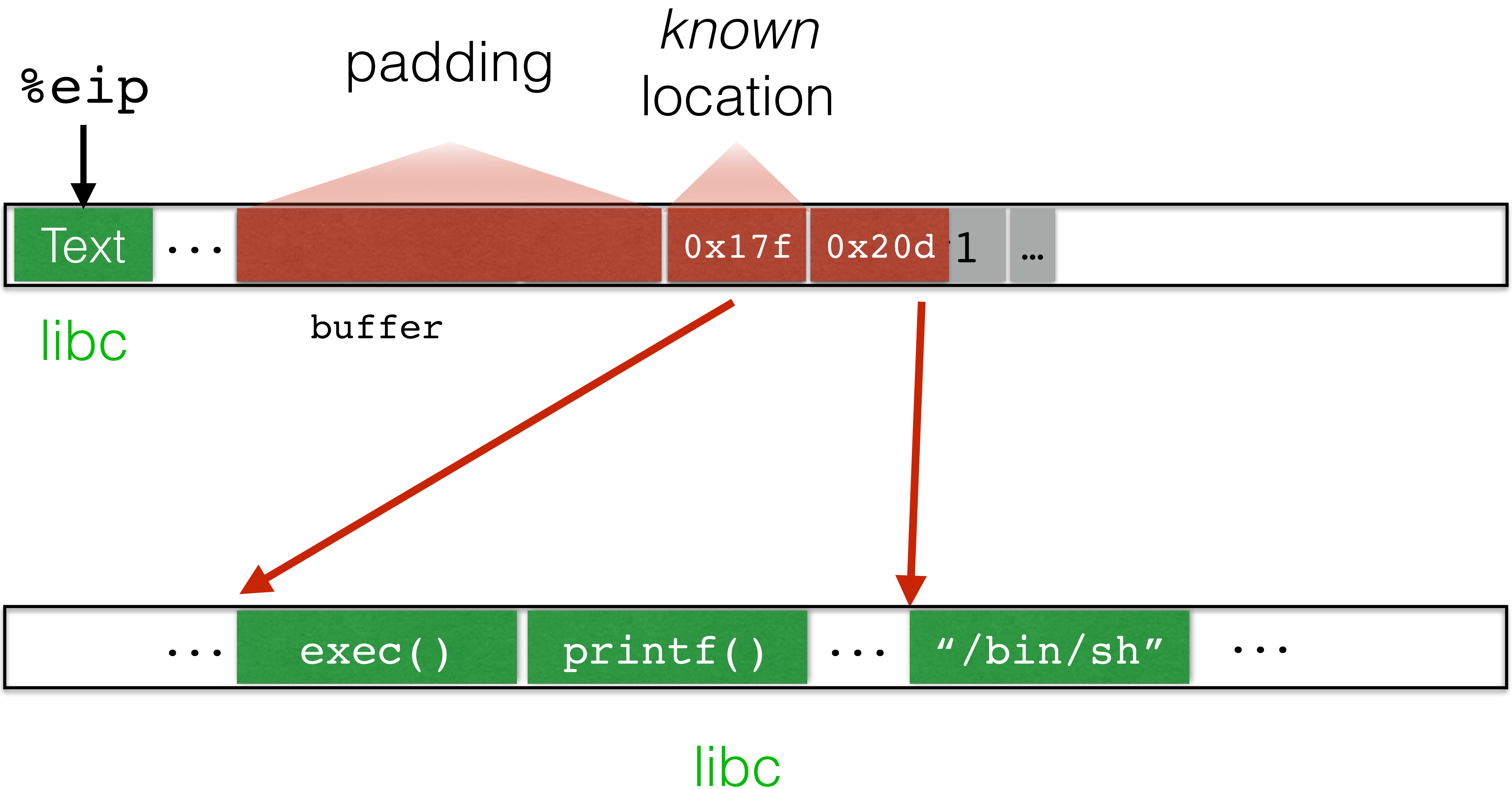
Return-to-libc



Return-to-libc



Return-to-libc



Recall our challenges

How can we make these even more difficult?

- Putting code into the memory (no zeroes)
  - Option: Make this detectable with canaries
- Getting %eip to point to our code (dist buff to stored eip)
  - Non-executable stack doesn't work so well
- Finding the return address (guess the raw addr)

Address Space Layout Randomization (ASLR)

- Basic idea: change the layout of the stack
- Slow to adopt
  - Linux in 2005
  - Vista in 2007 (off by default for compatibility with older software
  - OS X in 2007 (for system libraries), 2011 for all apps
  - iOS 4.3 (2011)
  - Android 4.0
  - FreeBSD: no

How would you overcome this as an attacker?

Overflow defenses summary

- Putting code into the memory (no zeroes)
  - Option: Make this detectable with **canaries**

- Getting %eip to point to our code (dist buff to stored eip)
  - **Non-executable stack** doesn’t work so well

- Finding the return address (guess the raw addr)
  - Address Space Layout Randomization (**ASLR**)
    - Many systems slow to adopt; also, how could you get around this?

- Good coding practices

Last time

We continued

Buffer overflows

and other memory safety vulnerabilities

By looking at

Overflow Defenses

- Finish overflow attacks & other vulnerabilities
- Overflow defenses
- Everything you've always wanted to know about `gdb` but were too afraid to ask

This time

Continuing with
Software Security

Writing & testing for
Secure Code

Required reading:

"StackGuard: Simple Stack Smash Protection for GCC"

Optional reading:

"Basic Integer Overflows"

"Exploiting Format String Vulnerabilities"

http://nob.cs.ucdavis.edu/bishop/secprog/robust.html

Cat and mouse

Cat and mouse

- **Defense: Make stack/heap non-executable** to prevent injection of code

Cat and mouse

- **Defense: Make stack/heap non-executable** to prevent injection of code
  - **Attack response: Return to libc**

Cat and mouse

- **Defense: Make stack/heap non-executable** to prevent injection of code
  - **Attack response: Return to libc**
- **Defense: Hide the address of desired libc code** or **return address** using ASLR

Cat and mouse

- **Defense: Make stack/heap non-executable** to prevent injection of code
  - **Attack response: Return to libc**
- **Defense: Hide the address of desired libc code** or **return address** using ASLR
  - **Attack response: Brute force search** (for 32-bit systems) or **information leak** (format string vulnerability: later today)

Cat and mouse

- **Defense: Make stack/heap non-executable** to prevent injection of code
  - **Attack response: Return to libc**
- **Defense: Hide the address of desired libc code** or **return address** using ASLR
  - **Attack response: Brute force search** (for 32-bit systems) or **information leak** (format string vulnerability: later today)
- **Defense: Avoid using libc code entirely** and use code in the program text instead

Cat and mouse

- **Defense: Make stack/heap non-executable** to prevent injection of code
  - **Attack response: Return to libc**
- **Defense: Hide the address of desired libc code** or **return address** using ASLR
  - **Attack response: Brute force search** (for 32-bit systems) or **information leak** (format string vulnerability: later today)
- **Defense: Avoid using libc code entirely** and use code in the program text instead
  - **Attack response:** Construct needed functionality using **return oriented programming (ROP)**

Return oriented programming (ROP)

Return-oriented Programming

Return-oriented Programming

- Introduced by Hovav Shacham in 2007
    - *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)*, CCS'07

Return-oriented Programming

- Introduced by Hovav Shacham in 2007
    - *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)*, CCS'07

- Idea: rather than use a single (libc) function to run your shellcode, **string together pieces of existing code, called *gadgets***, to do it instead

Return-oriented Programming

- Introduced by Hovav Shacham in 2007
    - *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)*, CCS'07

- Idea: rather than use a single (libc) function to run your shellcode, **string together pieces of existing code, called *gadgets***, to do it instead

- Challenges
    - **Find the gadgets** you need
    - **String them together**

Approach

Approach

- Gadgets are instruction groups that end with `ret`

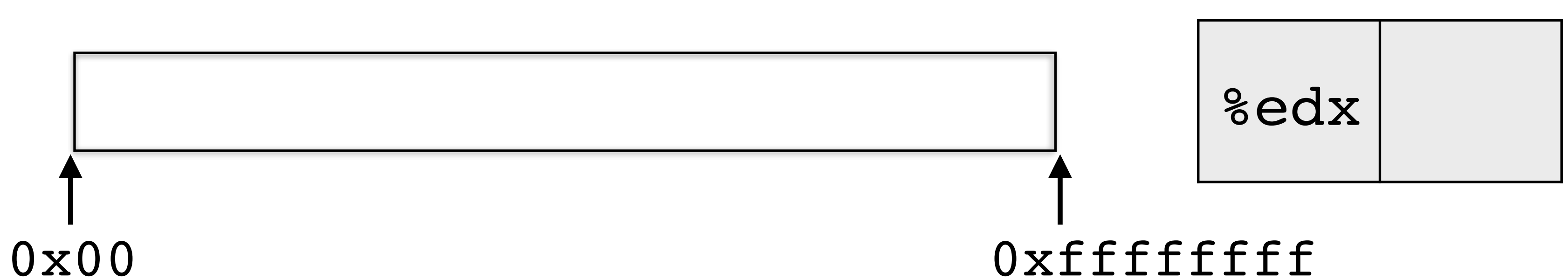
Approach

- Gadgets are instruction groups that end with **`ret`**
- Stack serves as the code
  - **`%esp`** = program counter
  - Gadgets invoked via **`ret`** instruction
  - Gadgets get their arguments via **`pop`**, etc.
    - Also on the stack

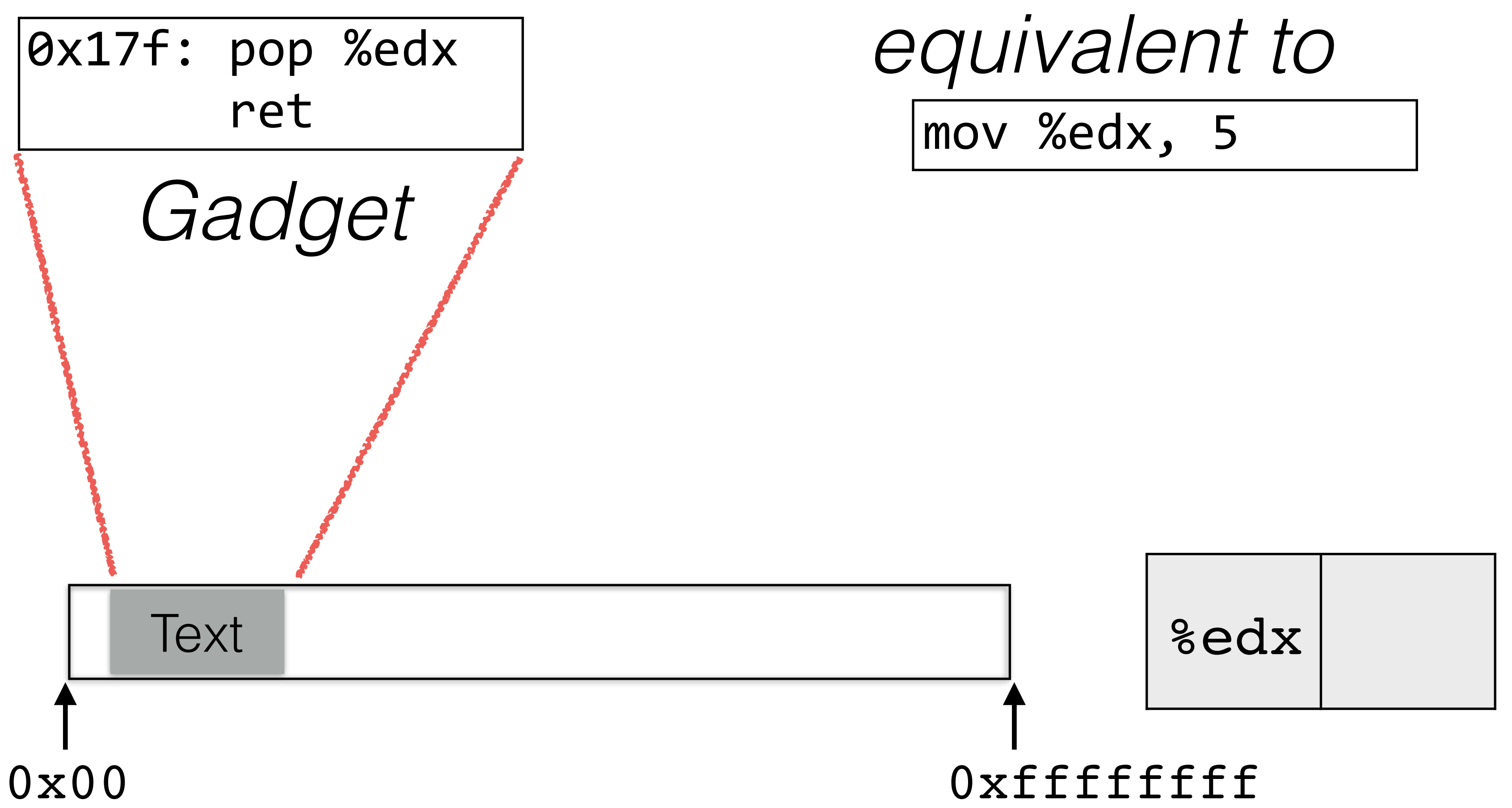
Simple example

equivalent to

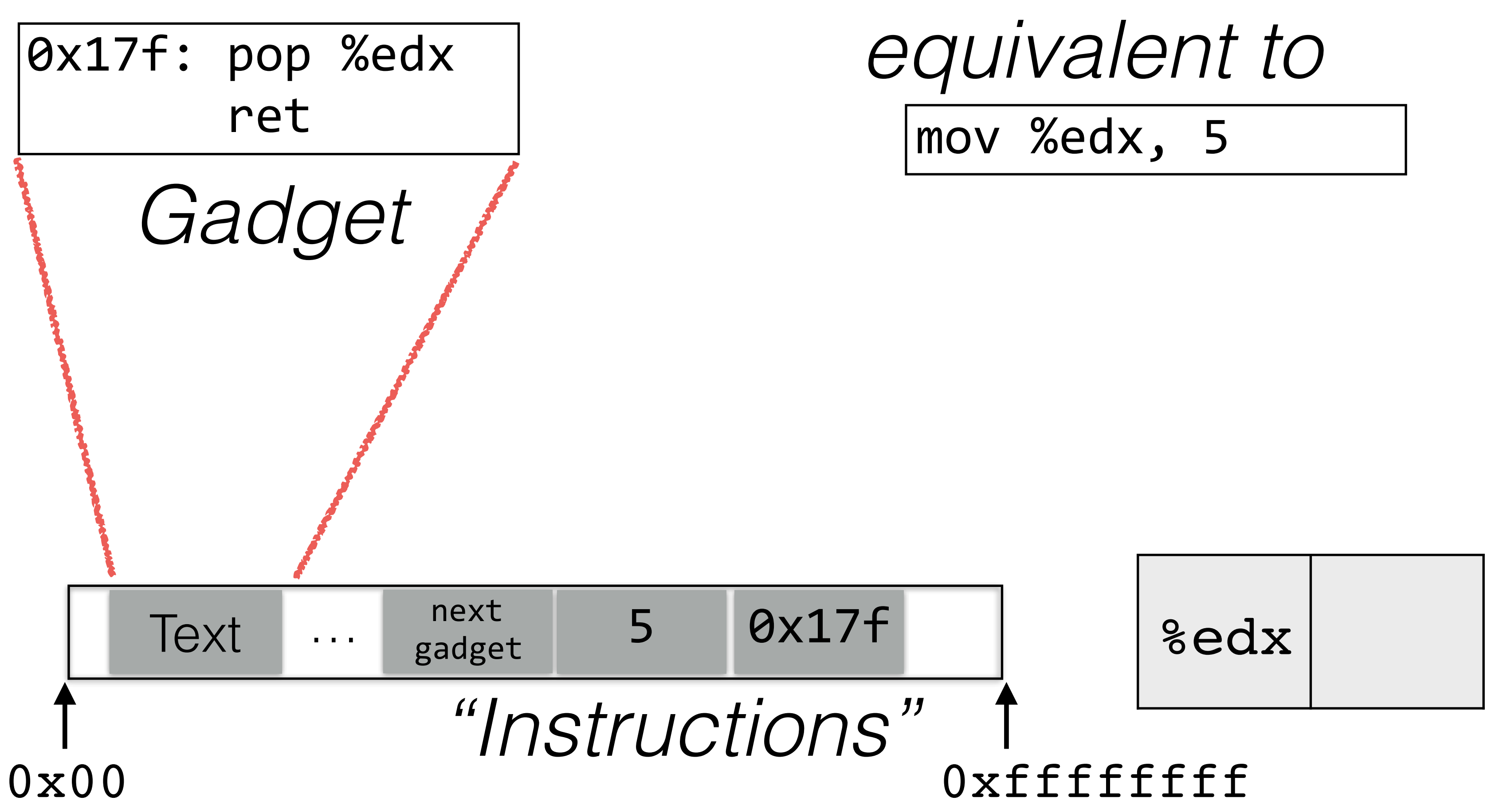
```
mov %edx, 5
```



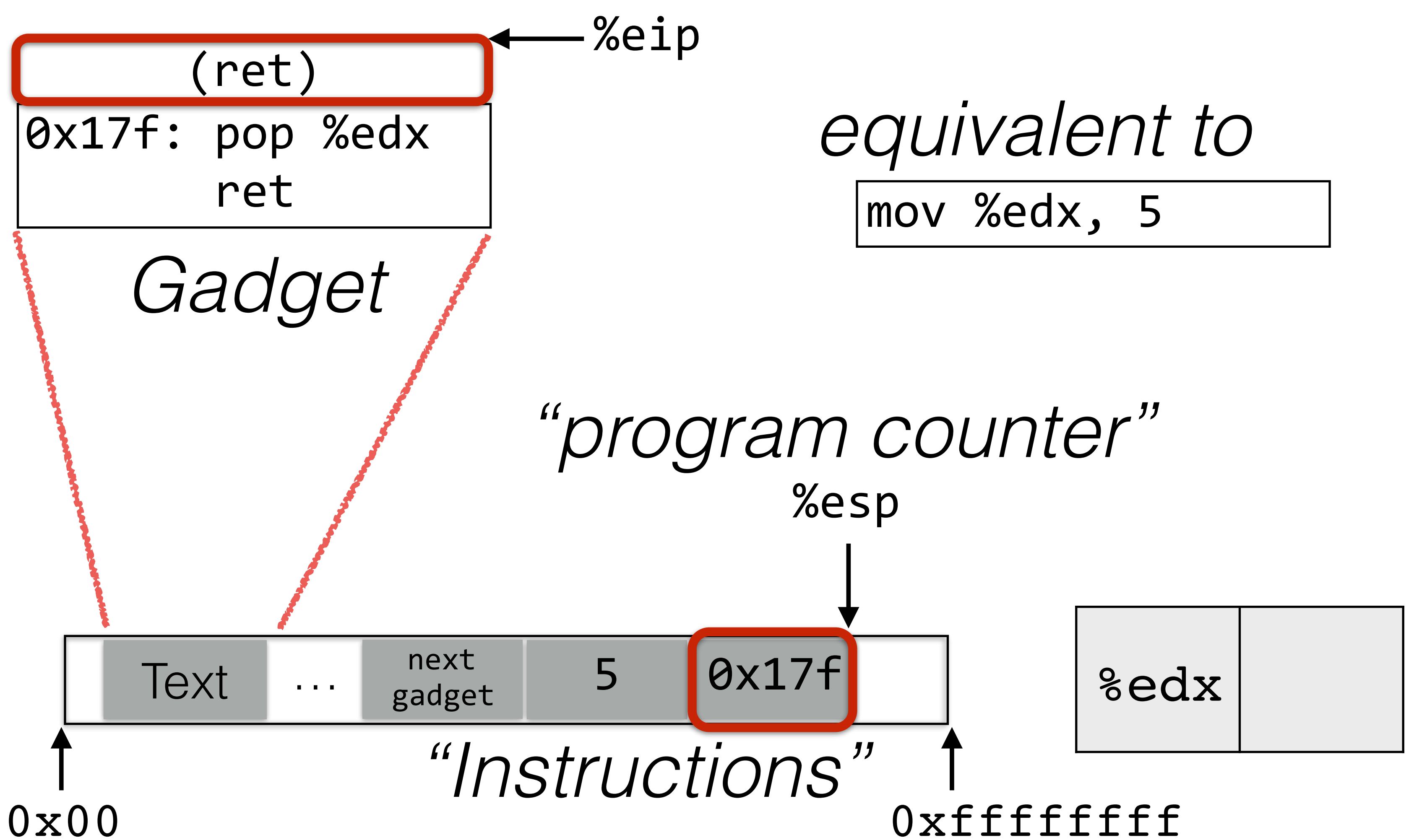
Simple example



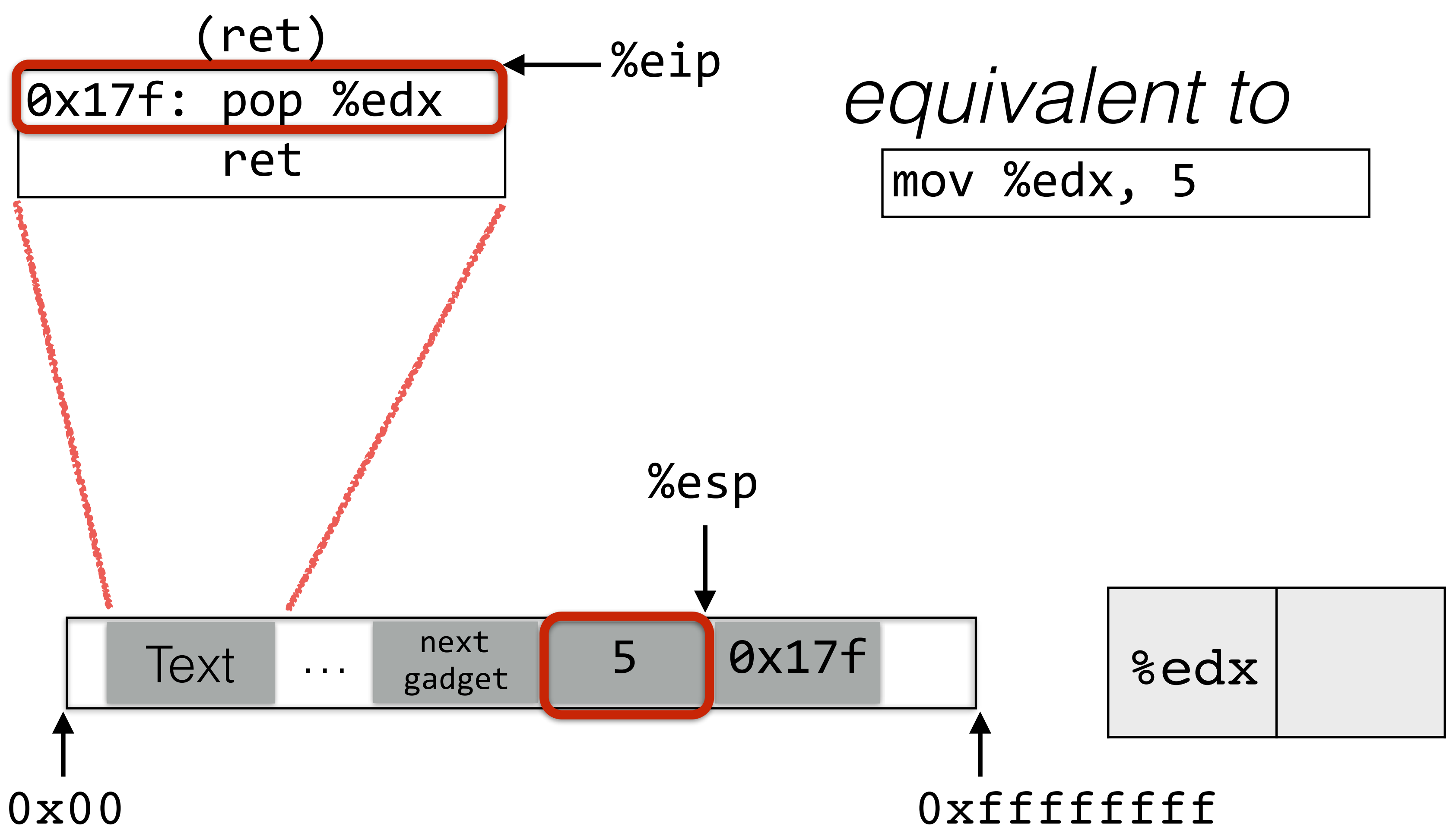
Simple example



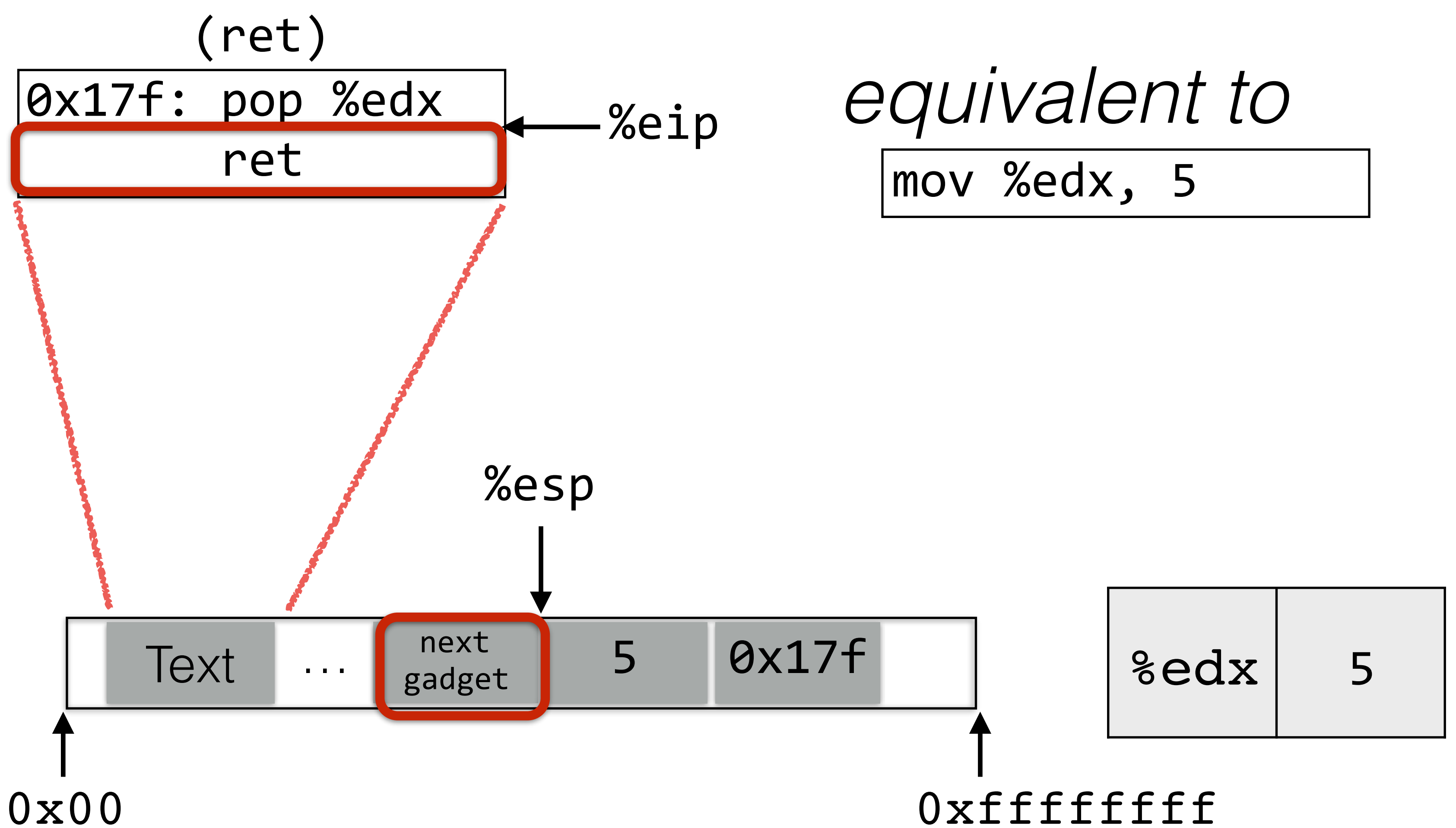
Simple example



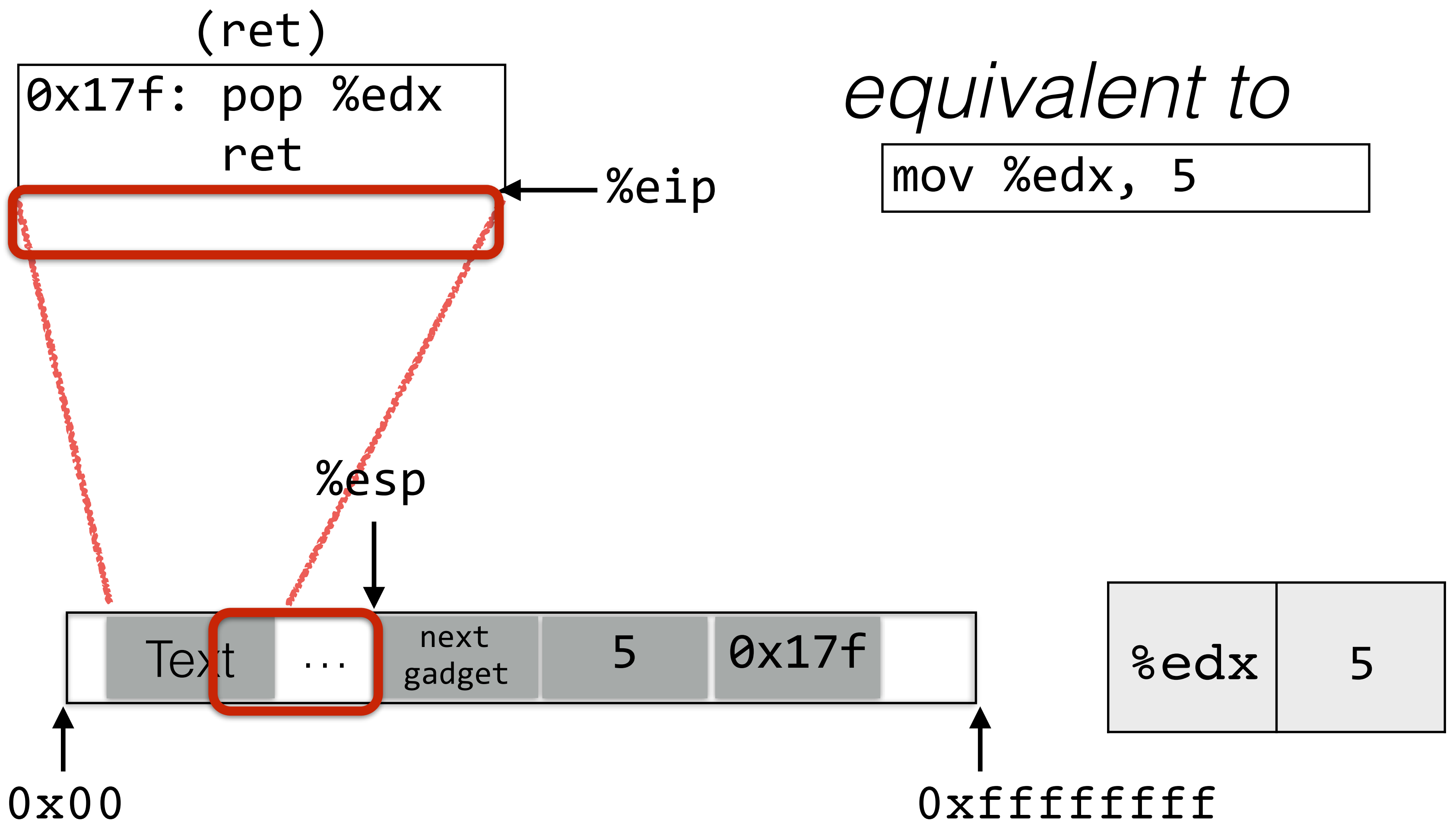
Simple example



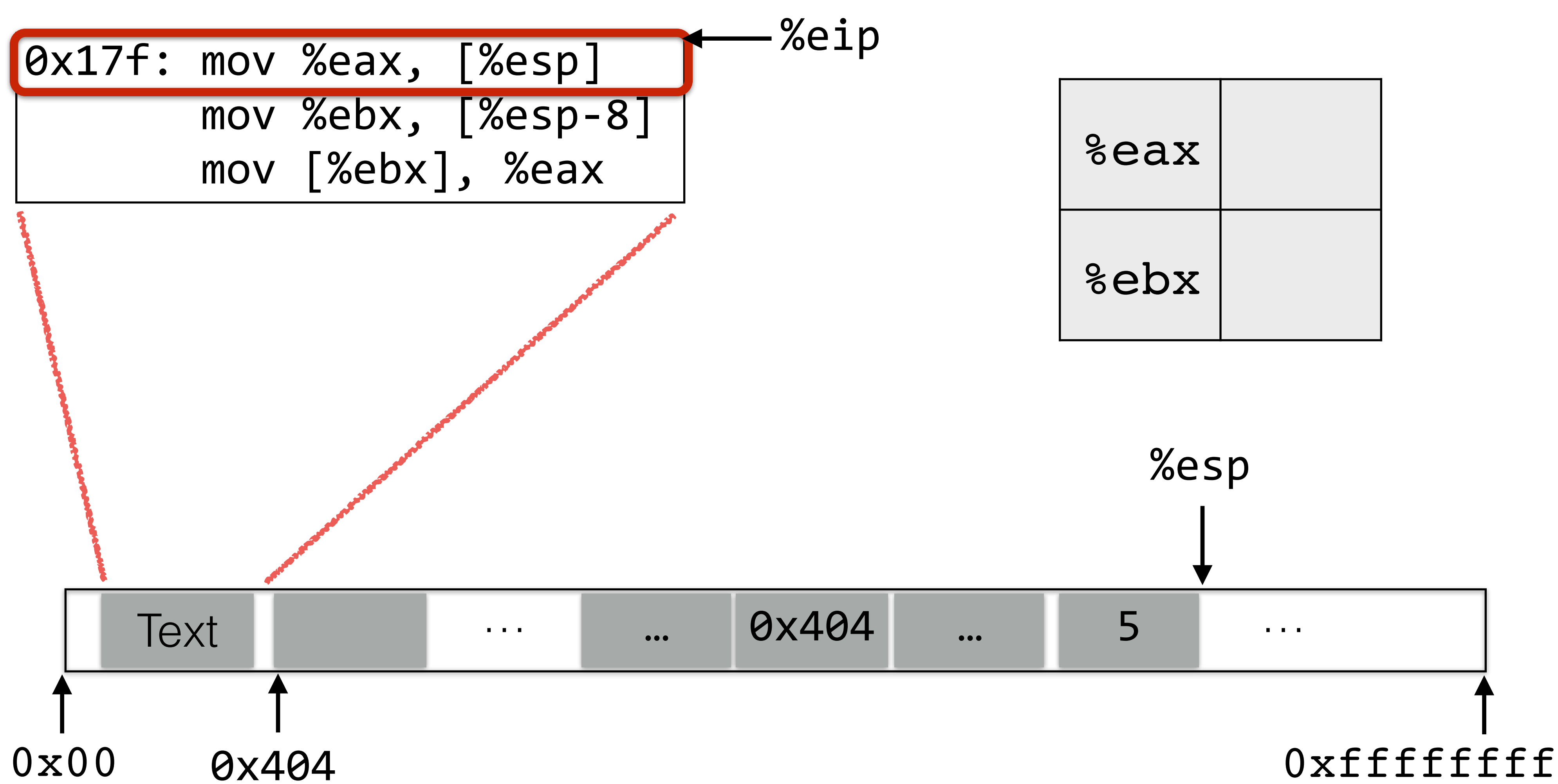
Simple example



Simple example

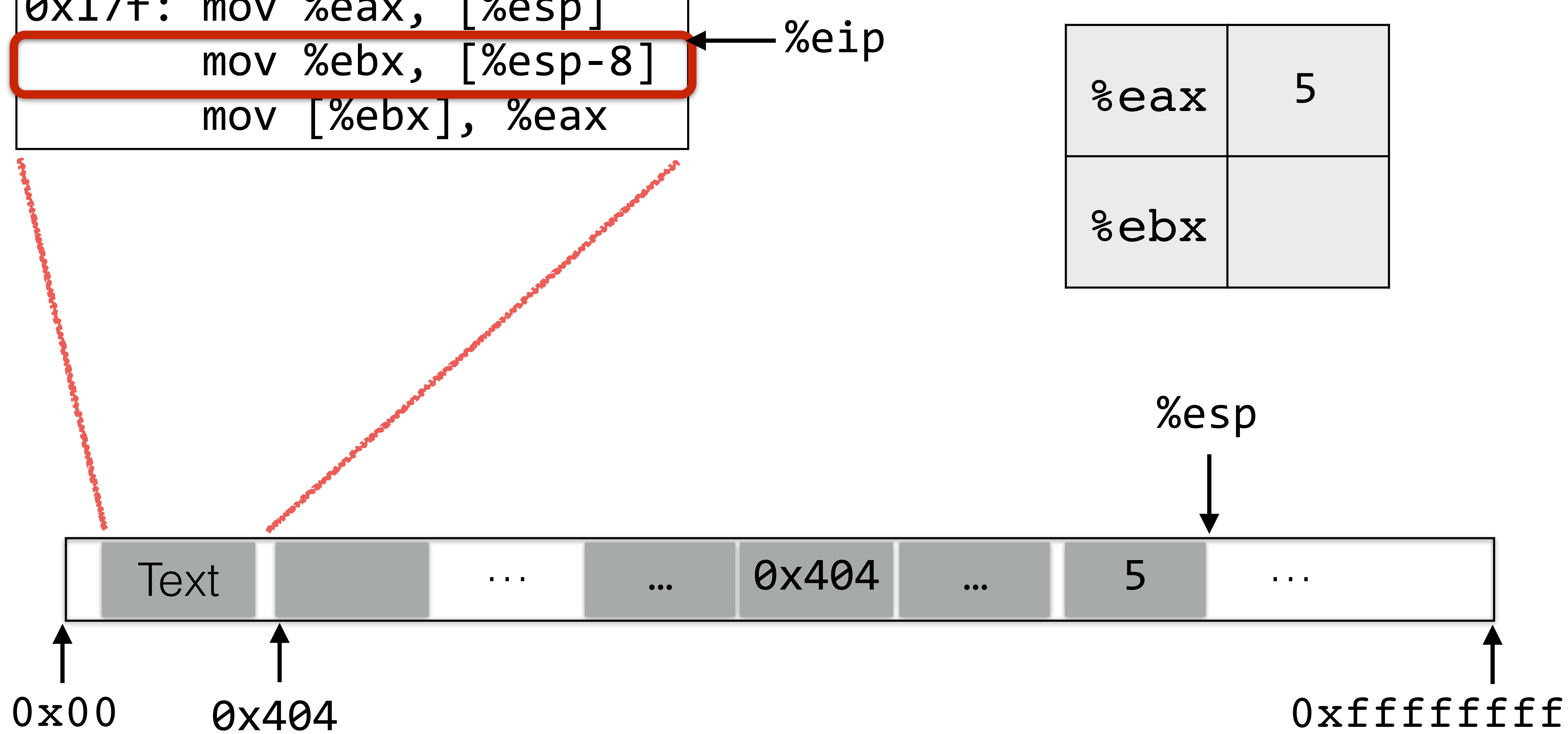


Code sequence



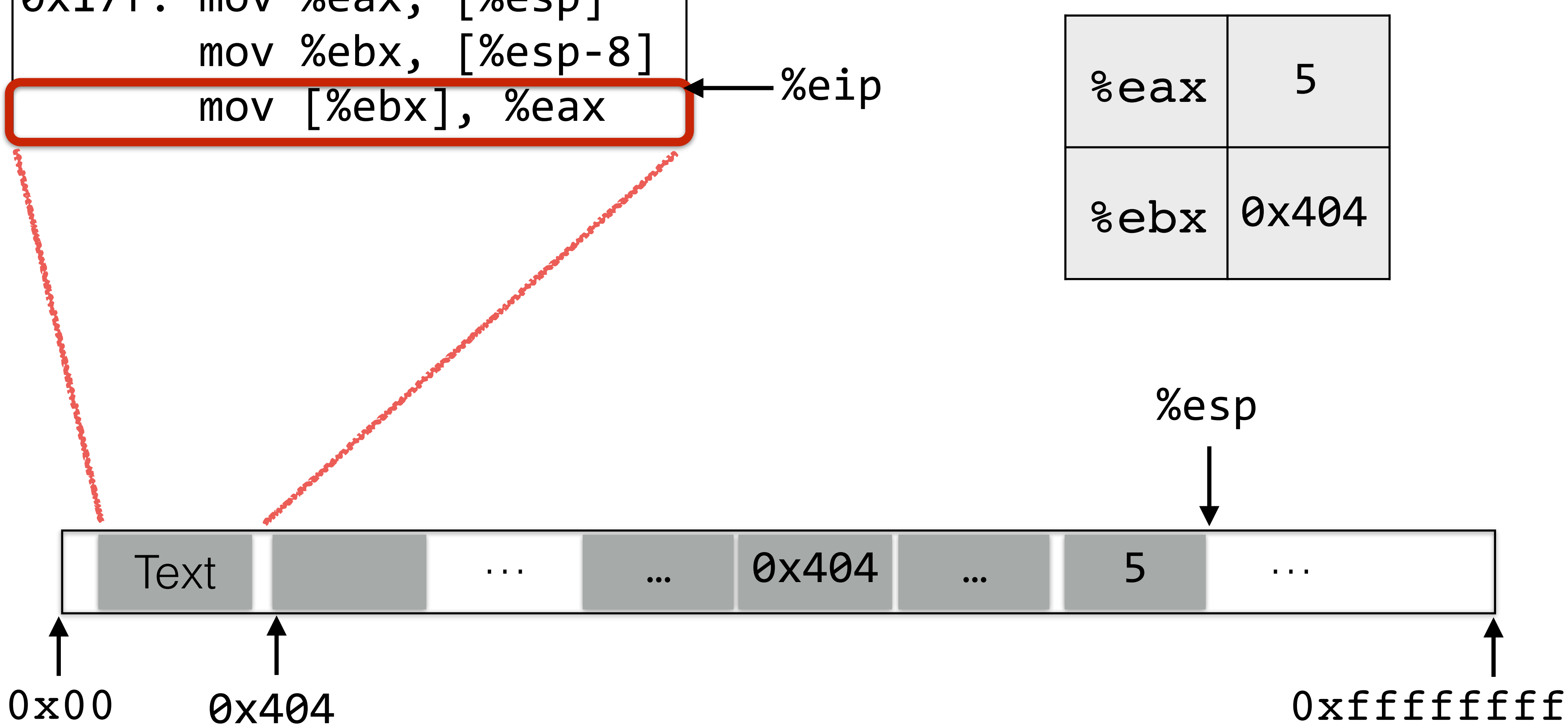
Code sequence

```
0x17f: mov %eax, [%esp]
       mov %ebx, [%esp-8]
       mov [%ebx], %eax
```

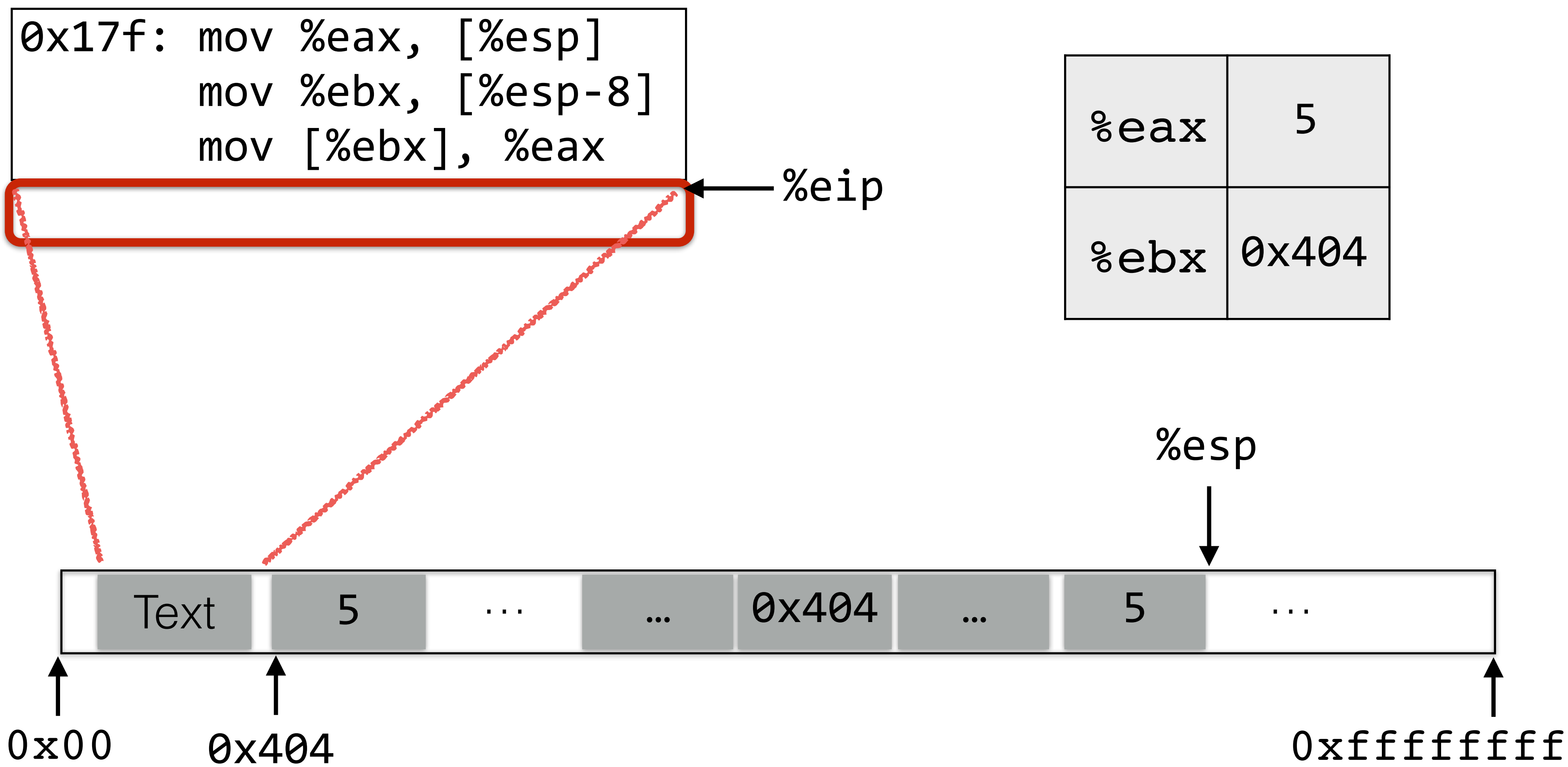


Code sequence

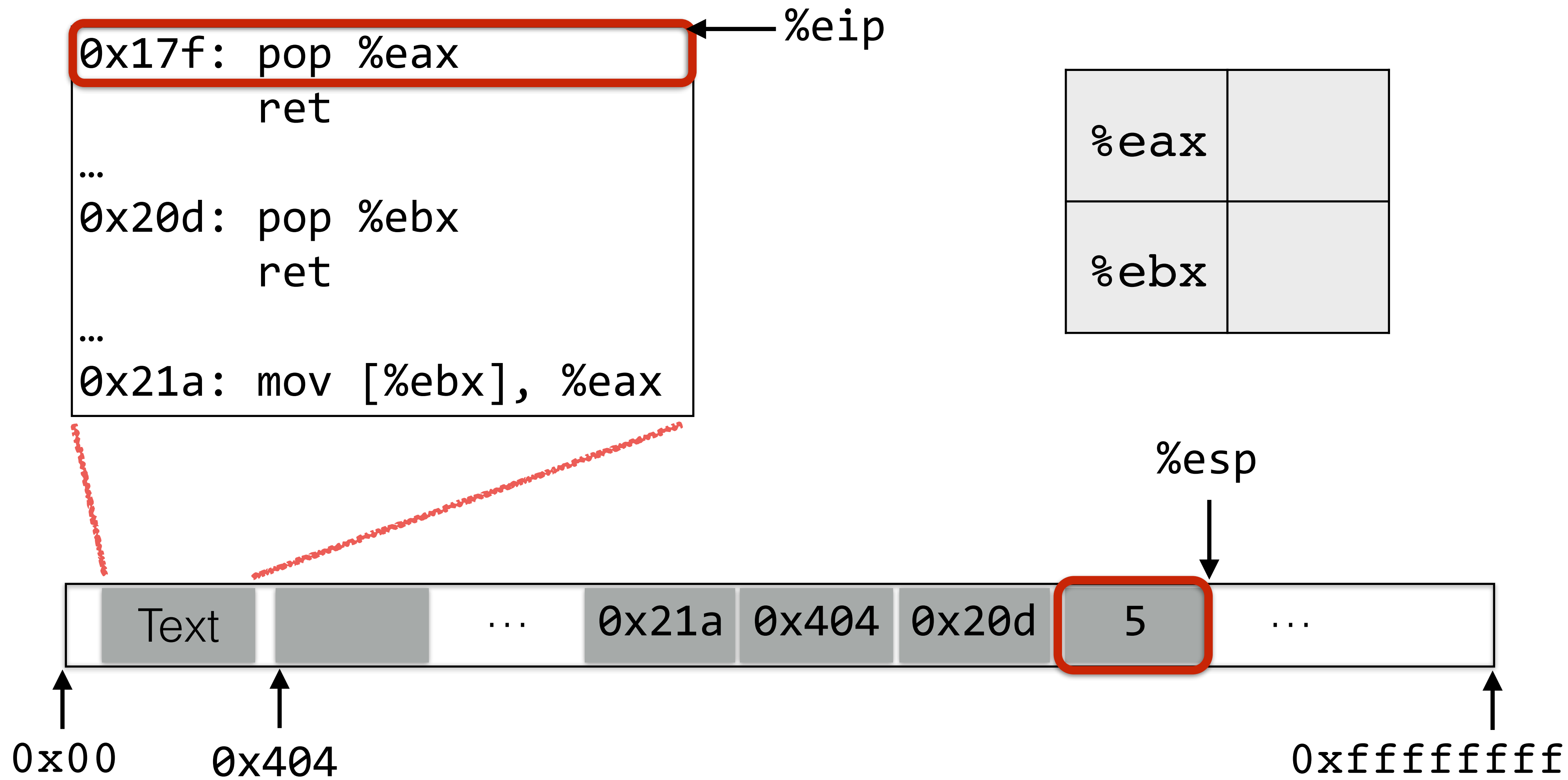
```
0x17f: mov %eax, [%esp]
       mov %ebx, [%esp-8]
       mov [%ebx], %eax
```



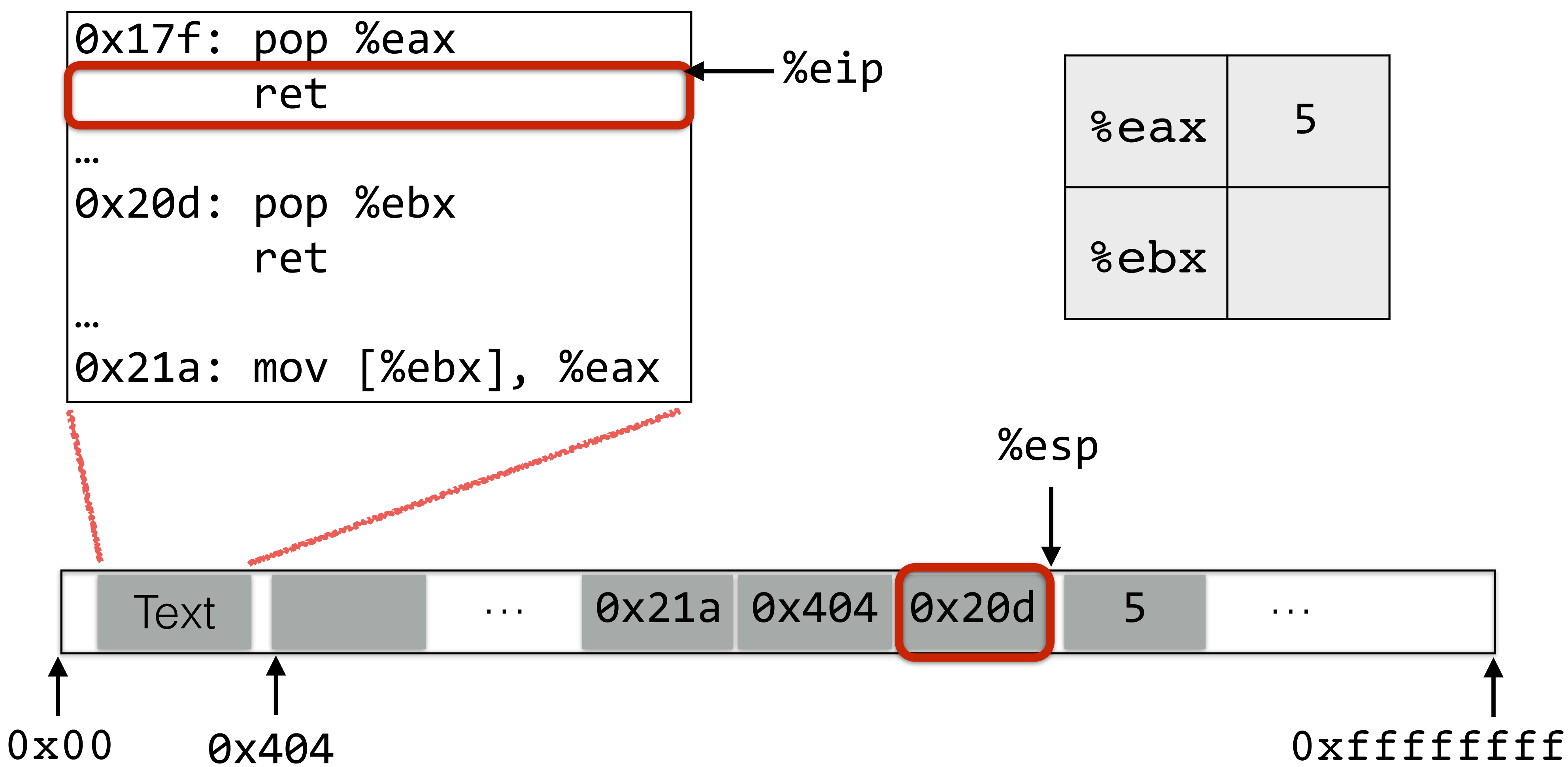
Code sequence



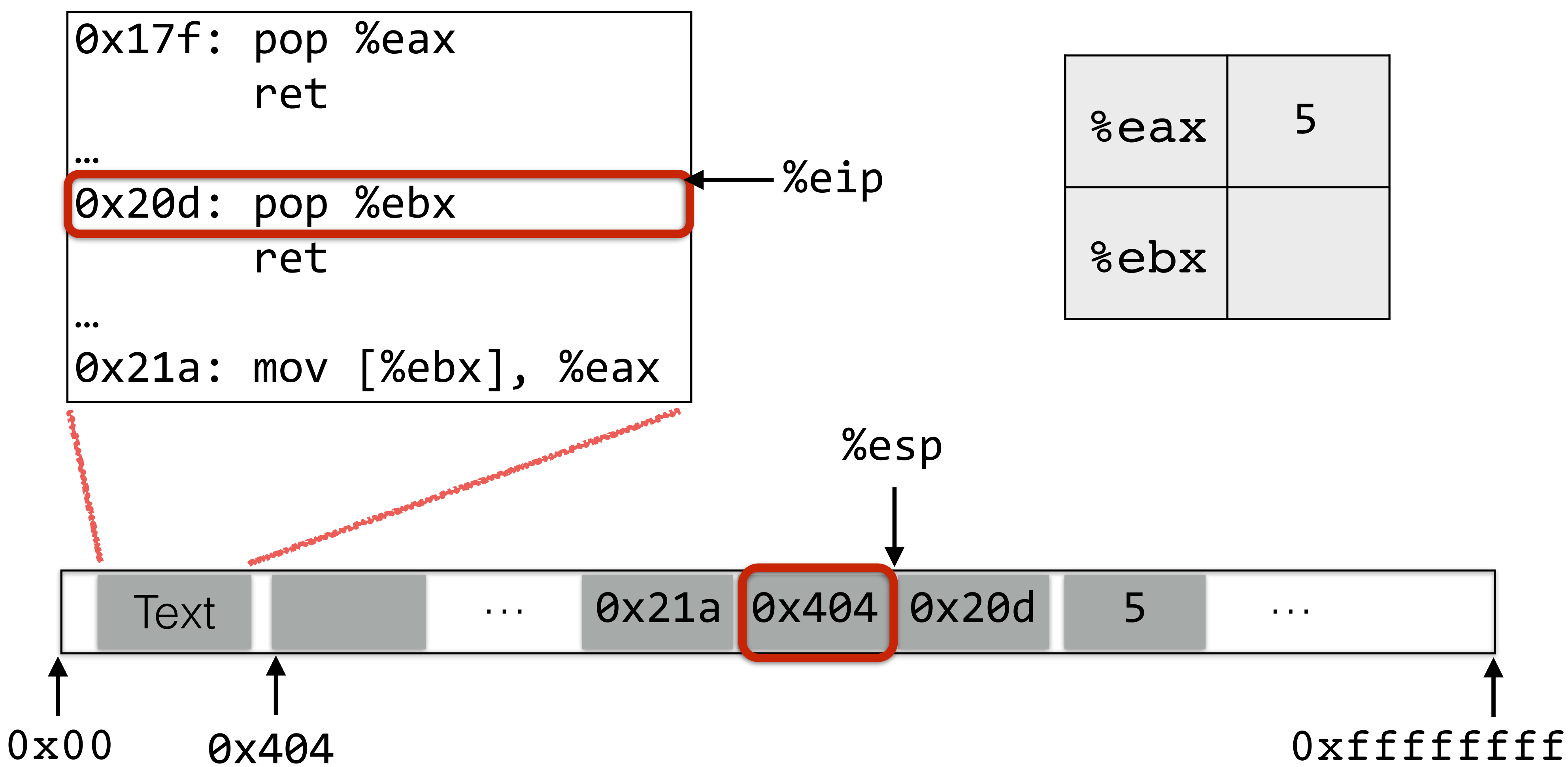
Equivalent ROP sequence



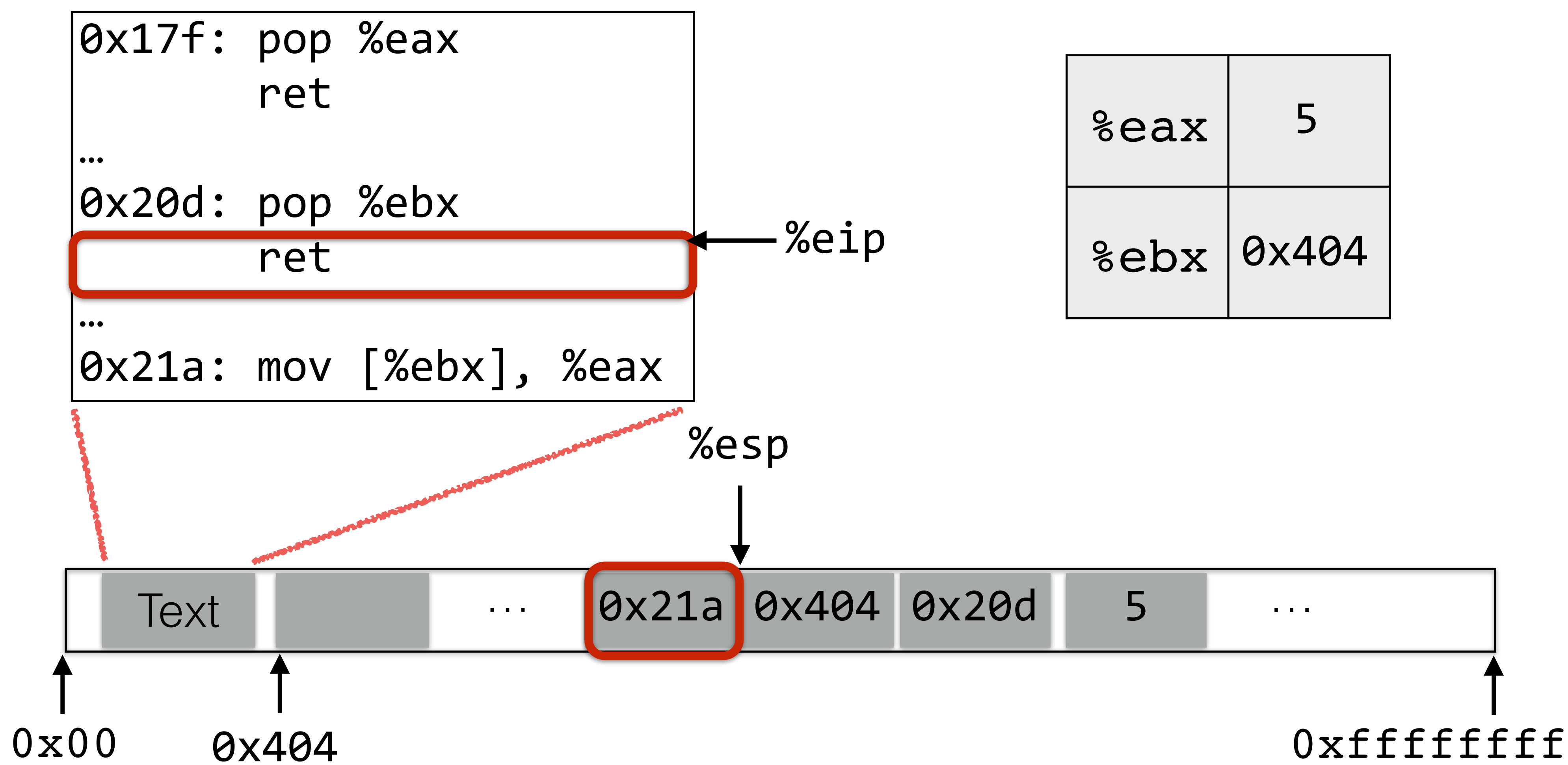
Equivalent ROP sequence



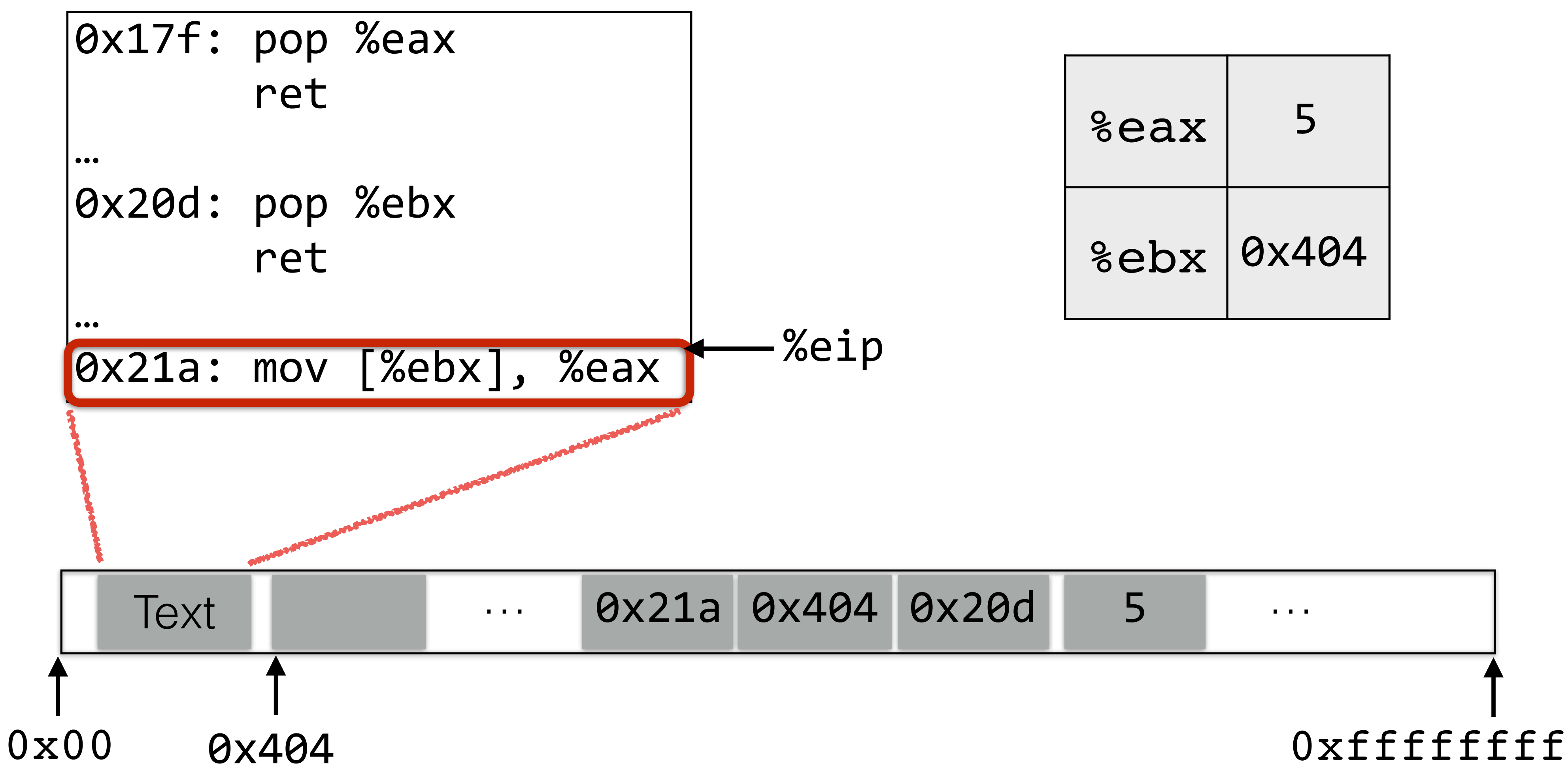
Equivalent ROP sequence



Equivalent ROP sequence



Equivalent ROP sequence



Equivalent ROP sequence

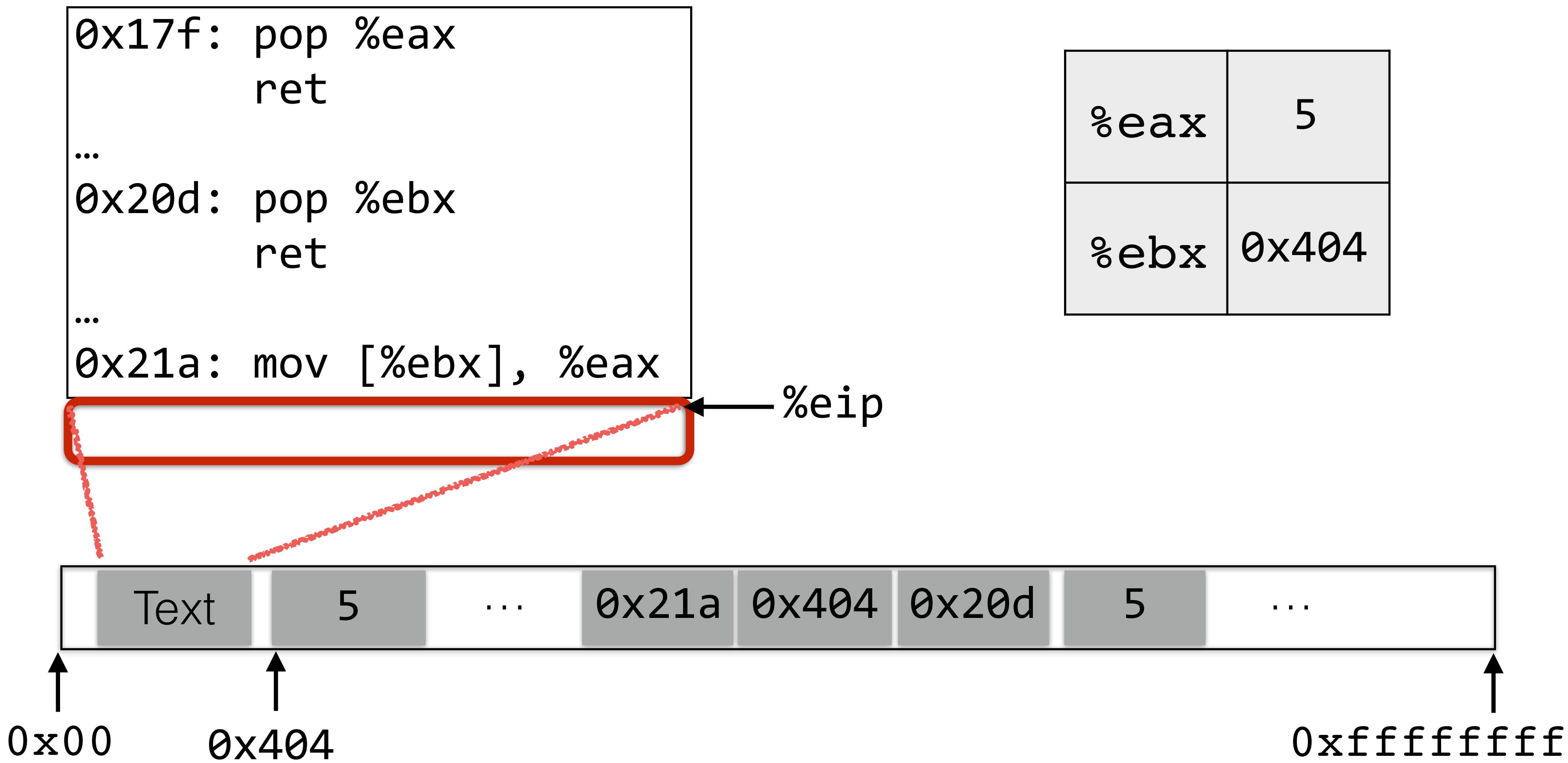


Image by Dino Dai Zovi

Whence the gadgets?

Whence the gadgets?

- How can we find gadgets to construct an exploit?

Whence the gadgets?

- How can we find gadgets to construct an exploit?
  - **Automate a search of the target binary for gadgets** (look for `ret` instructions, work backwards)
    - Cf. https://github.com/0vercl0k/rp

Whence the gadgets?

- How can we find gadgets to construct an exploit?
  - **Automate a search of the target binary for gadgets** (look for `ret` instructions, work backwards)
    - Cf. https://github.com/0vercl0k/rp
- Are there sufficient gadgets to do anything interesting?

Whence the gadgets?

- How can we find gadgets to construct an exploit?
  - **Automate a search of the target binary for gadgets** (look for `ret` instructions, work backwards)
    - Cf. https://github.com/0vercl0k/rp

- Are there sufficient gadgets to do anything interesting?
  - Yes: Shacham found that for significant codebases (e.g., libc), **gadgets are Turing complete**
    - Especially true on x86's dense instruction set

Whence the gadgets?

- How can we find gadgets to construct an exploit?
  - **Automate a search of the target binary for gadgets** (look for `ret` instructions, work backwards)
    - Cf. https://github.com/0vercl0k/rp
- Are there sufficient gadgets to do anything interesting?
  - Yes: Shacham found that for significant codebases (e.g., libc), **gadgets are Turing complete**
    - Especially true on x86's dense instruction set
  - Schwartz et al (USENIX Security '11) have automated gadget shellcode creation, though not needing/requiring Turing completeness

Blind ROP

Blind ROP

- **Defense: Randomizing the location of the code** (by compiling for position independence) on a 64-bit machine makes attacks very difficult

Blind ROP

- **Defense: Randomizing the location of the code** (by compiling for position independence) on a 64-bit machine makes attacks very difficult
  - Recent, published attacks are often for 32-bit versions of executables

Blind ROP

- **Defense: Randomizing the location of the code** (by compiling for position independence) on a 64-bit machine makes attacks very difficult
  - Recent, published attacks are often for 32-bit versions of executables

- **Attack response**: **Blind ROP**

Blind ROP

- **Defense: Randomizing the location of the code** (by compiling for position independence) on a 64-bit machine makes attacks very difficult
    - Recent, published attacks are often for 32-bit versions of executables
- **Attack response**: **Blind ROP**

  If server restarts on a crash, but does not re-randomize:

Blind ROP

- **Defense: Randomizing the location of the code** (by compiling for position independence) on a 64-bit machine makes attacks very difficult
  - Recent, published attacks are often for 32-bit versions of executables

- **Attack response**: **Blind ROP**

  If server restarts on a crash, but does not re-randomize:

1. Read the stack to **leak canaries and a return address**

Blind ROP

- **Defense: Randomizing the location of the code** (by compiling for position independence) on a 64-bit machine makes attacks very difficult
  - Recent, published attacks are often for 32-bit versions of executables

- **Attack response: Blind ROP**

  If server restarts on a crash, but does not re-randomize:

  1. Read the stack to **leak canaries and a return address**
  2. Find gadgets (at run-time) to **effect call to `write`**

Blind ROP

- **Defense: Randomizing the location of the code** (by compiling for position independence) on a 64-bit machine makes attacks very difficult
  - Recent, published attacks are often for 32-bit versions of executables

- **Attack response**: **Blind ROP**

  If server restarts on a crash, but does not re-randomize:

1. Read the stack to **leak canaries and a return address**
2. Find gadgets (at run-time) to **effect call to `write`**
3. **Dump binary to find gadgets for shellcode**

http://www.scs.stanford.edu/brop/

Defeat!

- The blind ROP team was able to **completely automatically**, only **through remote interactions**, develop a **remote code exploit for nginx**, a popular web server

Defeat!

- The blind ROP team was able to **completely automatically**, only **through remote interactions**, develop a **remote code exploit for nginx**, a popular web server
    - The exploit was carried out on a 64-bit executable with full stack canaries and randomization

Defeat!

- The blind ROP team was able to **completely automatically**, only **through remote interactions**, develop a **remote code exploit for nginx**, a popular web server
  - The exploit was carried out on a 64-bit executable with full stack canaries and randomization
- Conclusion: **give an inch, and they take a mile?**

Defeat!

- The blind ROP team was able to **completely automatically**, only **through remote interactions**, develop a **remote code exploit for nginx**, a popular web server
  - The exploit was carried out on a 64-bit executable with full stack canaries and randomization
- Conclusion: **give an inch, and they take a mile?**
- Put another way: **Memory safety is really useful!**

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

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

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

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

```c
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

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

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

```c
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

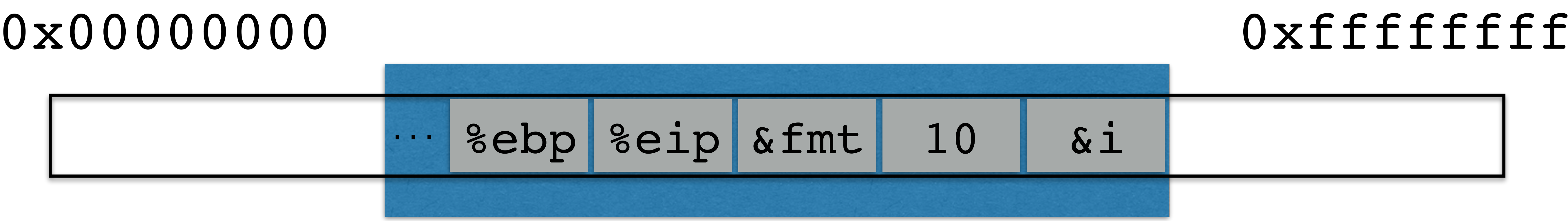
Format string vulnerabilities

`printf` format strings

```
int i = 10;
printf(“%d %p\n”, i, &i);
```

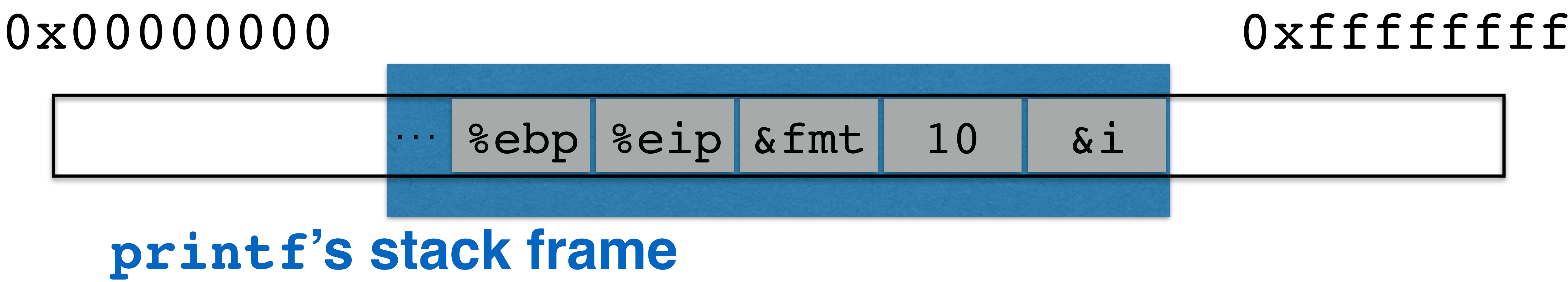
printf format strings

```
int i = 10;
printf("%d %p\n", i, &i);
```



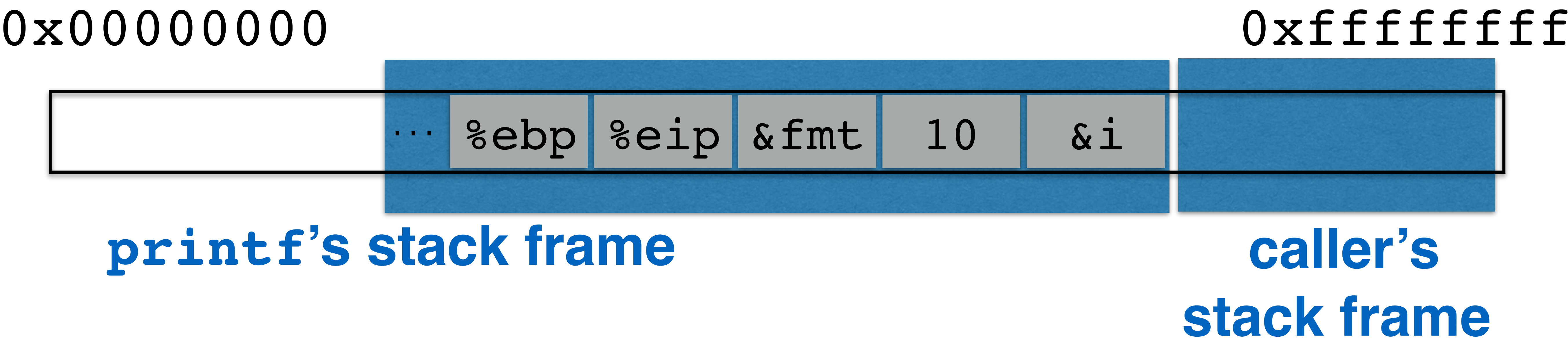
`printf` format strings

```
int i = 10;
printf("%d %p\n", i, &i);
```



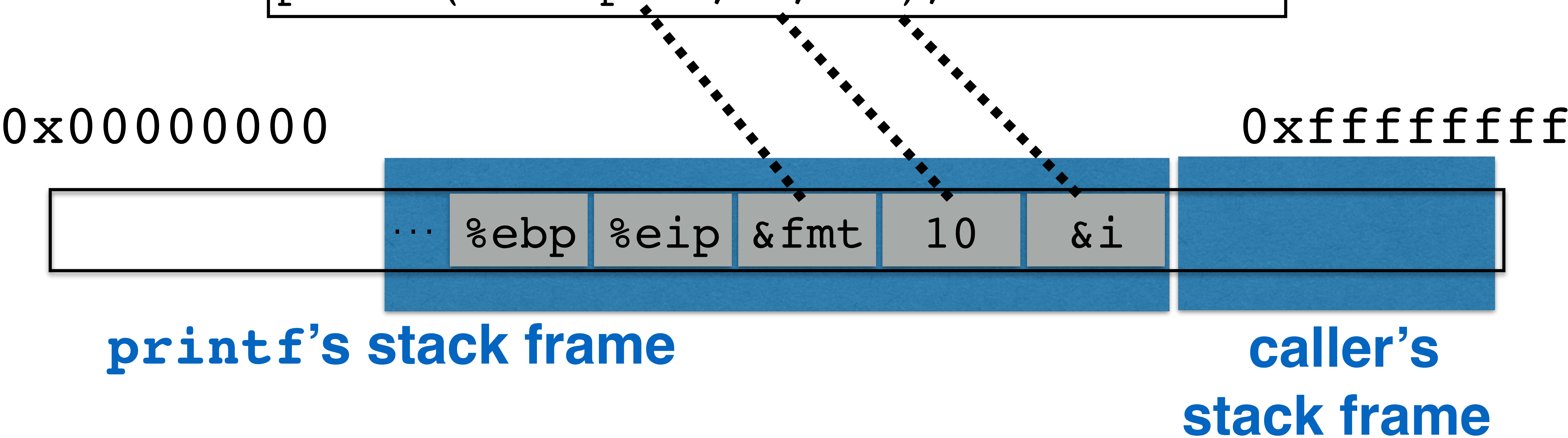
`printf` format strings

```
int i = 10;
printf(“%d %p\n”, i, &i);
```



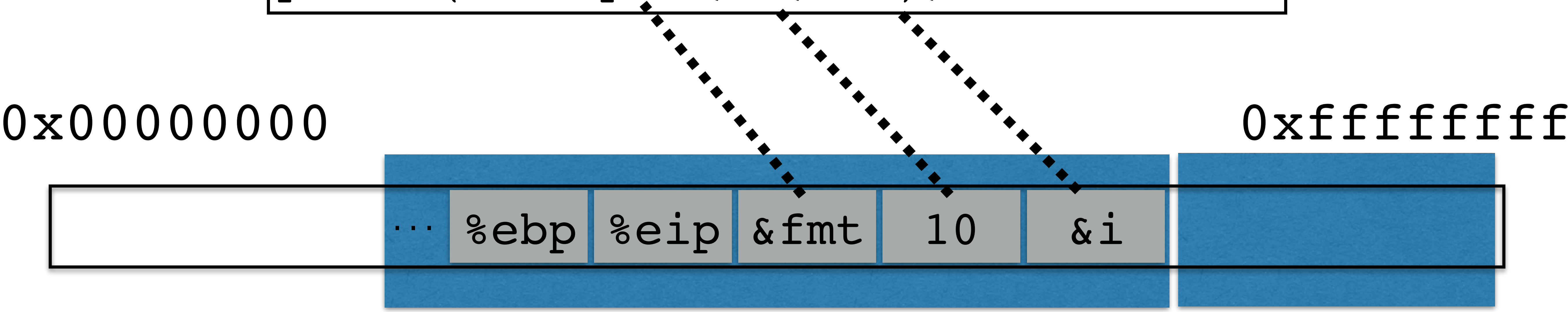
`printf` format strings

```
int i = 10;
printf("%d %p\n", i, &i);
```



`printf` format strings

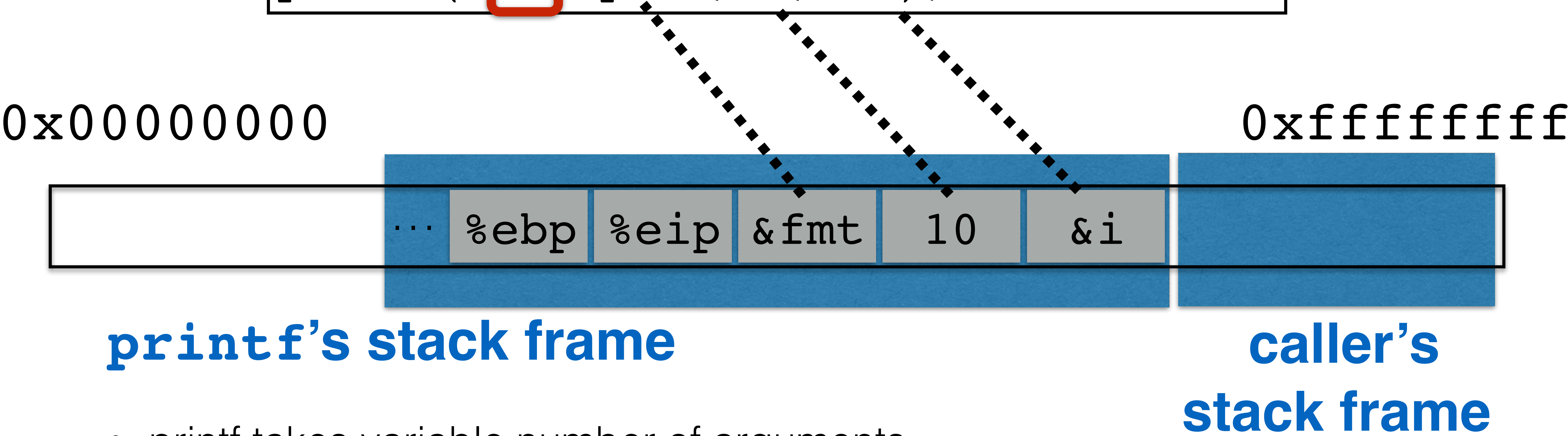
```
int i = 10;
printf("%d %p\n", i, &i);
```



- printf takes variable number of arguments
- printf pays no mind to where the stack frame "ends"
- It presumes that you called it with (at least) as many arguments as specified in the format string

`printf` format strings

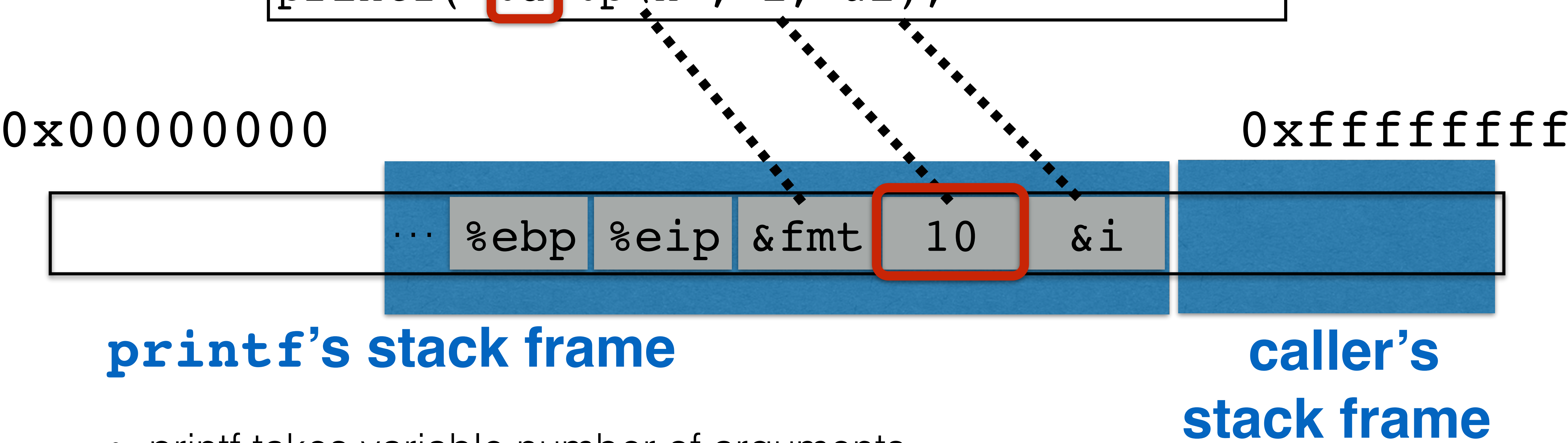
```
int i = 10;
printf("%d %p\n", i, &i);
```



- printf takes variable number of arguments
- printf pays no mind to where the stack frame "ends"
- It presumes that you called it with (at least) as many arguments as specified in the format string

`printf` format strings

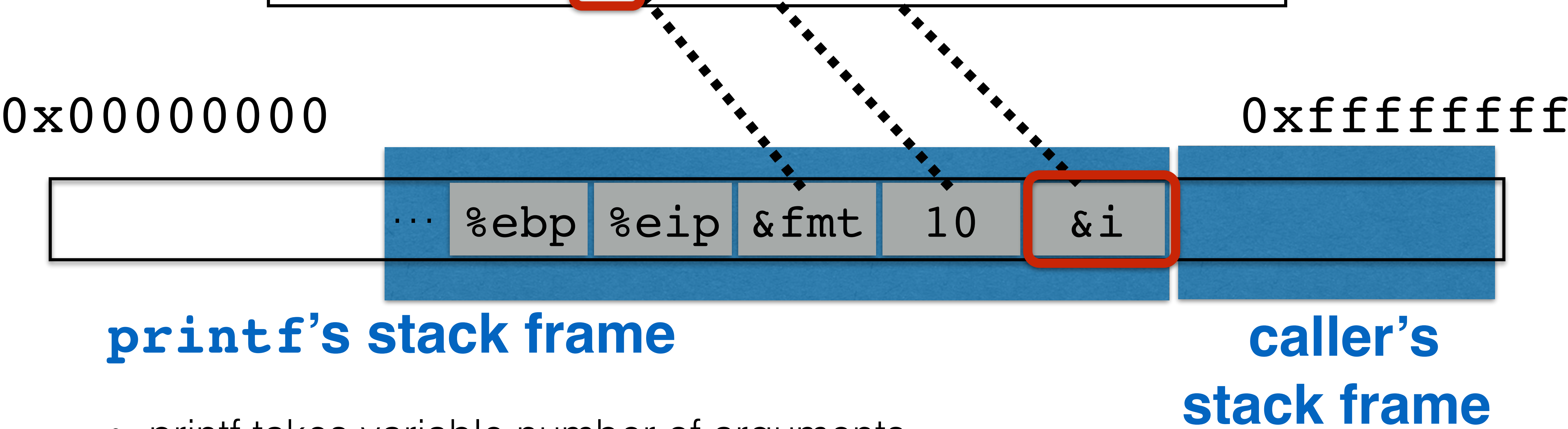
```
int i = 10;
printf("%d %p\n", i, &i);
```



- printf takes variable number of arguments
- printf pays no mind to where the stack frame "ends"
- It presumes that you called it with (at least) as many arguments as specified in the format string

`printf` format strings

```
int i = 10;
printf("%d %p\n", i, &i);
```



- printf takes variable number of arguments
- printf pays no mind to where the stack frame "ends"
- It presumes that you called it with (at least) as many arguments as specified in the format string

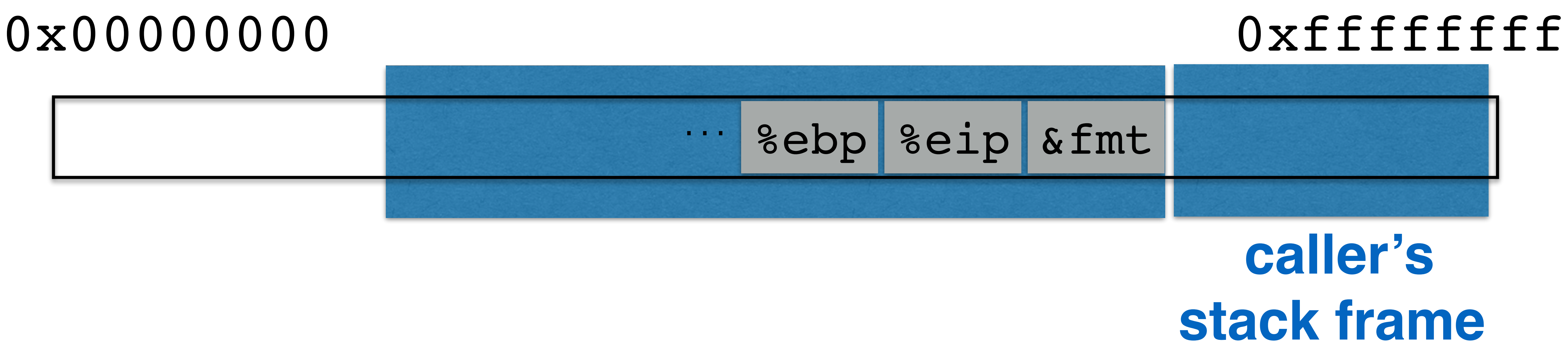
```c
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

```c
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

“%d %x"

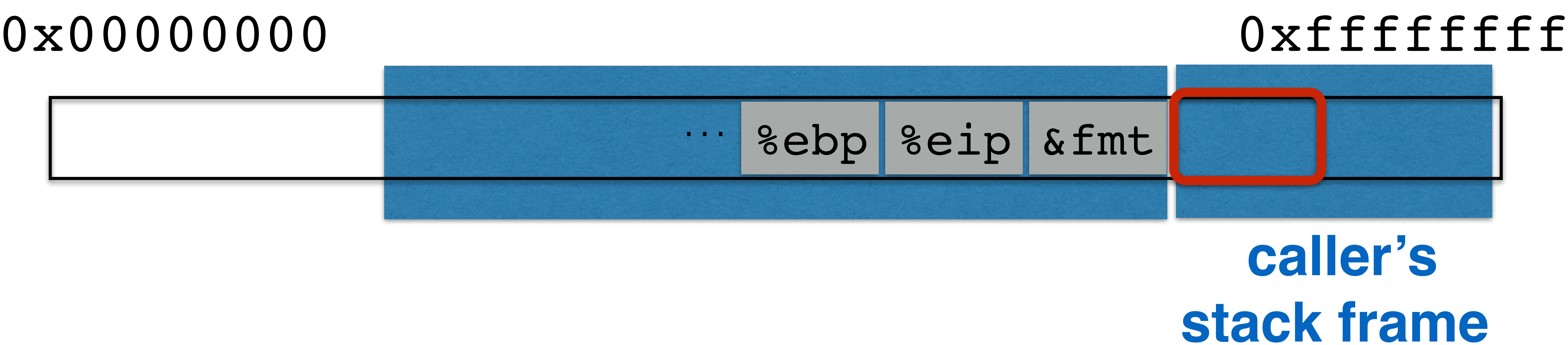
```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

“%d %x"



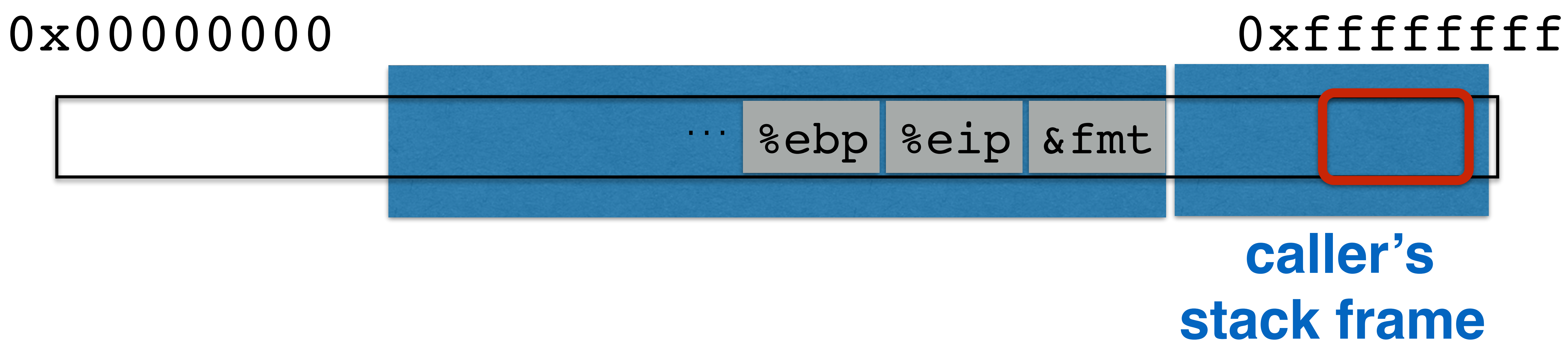
```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

“%d %x"



```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

“%d %x"



Format string vulnerabilities

Format string vulnerabilities

- `printf(“100% dml”);`

Format string vulnerabilities

- `printf("100% dml");`
  - Prints stack entry 4 byes above saved %eip

Format string vulnerabilities

- `printf(“100% dml”);`
  - Prints stack entry 4 byes above saved %eip
- `printf(“%s”);`

Format string vulnerabilities

- `printf(“100% dml”);`
  - Prints stack entry 4 byes above saved %eip
- `printf(“%s”);`
  - Prints bytes *pointed to* by that stack entry

Format string vulnerabilities

- `printf(“100% dml”);`
    - Prints stack entry 4 byes above saved %eip
- `printf(“%s”);`
    - Prints bytes *pointed to* by that stack entry
- `printf(“%d %d %d %d …”);`

Format string vulnerabilities

- `printf("100% dml");`
  - Prints stack entry 4 byes above saved %eip
- `printf("%s");`
  - Prints bytes *pointed to* by that stack entry
- `printf("%d %d %d %d …");`
  - Prints a series of stack entries as integers

Format string vulnerabilities

- **`printf(“100% dml”);`**
  - Prints stack entry 4 byes above saved %eip
- **`printf(“%s”);`**
  - Prints bytes *pointed to* by that stack entry
- **`printf(“%d %d %d %d …”);`**
  - Prints a series of stack entries as integers
- **`printf(“%08x %08x %08x %08x …”);`**

Format string vulnerabilities

- `printf("100% dml");`
  - Prints stack entry 4 byes above saved %eip
- `printf("%s");`
  - Prints bytes *pointed to* by that stack entry
- `printf("%d %d %d %d …");`
  - Prints a series of stack entries as integers
- `printf("%08x %08x %08x %08x …");`
  - Same, but nicely formatted hex

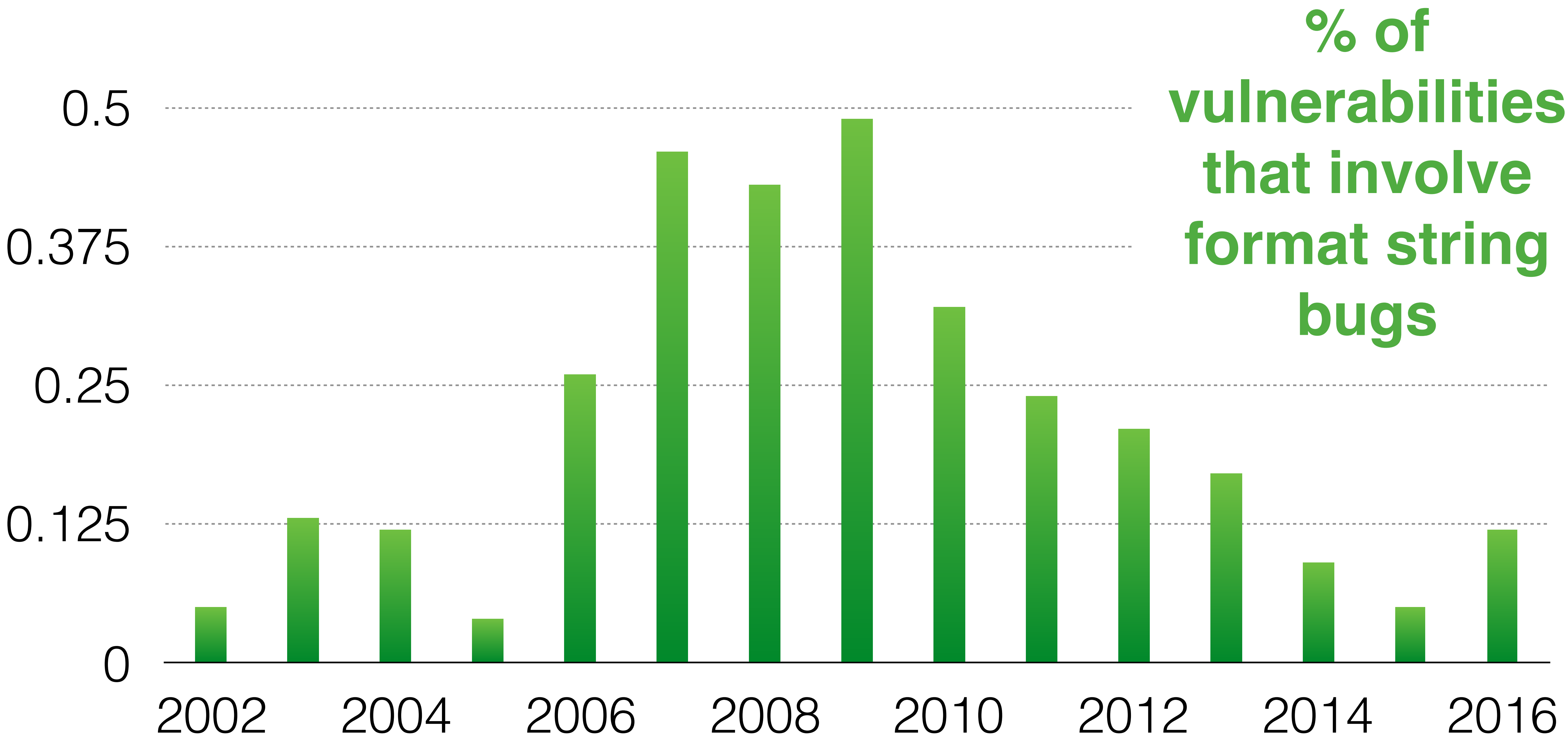
Format string vulnerabilities

- `printf("100% dml");`
  - Prints stack entry 4 byes above saved %eip
- `printf("%s");`
  - Prints bytes *pointed to* by that stack entry
- `printf("%d %d %d %d …");`
  - Prints a series of stack entries as integers
- `printf("%08x %08x %08x %08x …");`
  - Same, but nicely formatted hex
- `printf("100% no way!")`

Format string vulnerabilities

- `printf(“100% dml”);`
  - Prints stack entry 4 byes above saved %eip
- `printf(“%s”);`
  - Prints bytes *pointed to* by that stack entry
- `printf(“%d %d %d %d …”);`
  - Prints a series of stack entries as integers
- `printf(“%08x %08x %08x %08x …”);`
  - Same, but nicely formatted hex
- `printf(“100% no way!”)`
  - ***WRITES*** the number 3 to address pointed to by stack entry

Format string prevalence



http://web.nvd.nist.gov/view/vuln/statistics

What’s wrong with this code?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf(“Too large\n”);
        return;
    }
    memcpy(buf, p, len);
}
```

What's wrong with this code?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }
    memcpy(buf, p, len);
}
```

```
void *memcpy(void *dest, const void *src, size_t n);
```

What’s wrong with this code?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf(“Too large\n”);
        return;
    }
    memcpy(buf, p, len);
}
```

```
void *memcpy(void *dest, const void *src, size_t n);
typedef unsigned int size_t;
```

What's wrong with this code?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{   Negative
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }
    memcpy(buf, p, len);
}
```

```
void *memcpy(void *dest, const void *src, size_t n);
typedef unsigned int size_t;
```

What’s wrong with this code?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{   Negative
    int len = read_int_from_network();
    char *p = read_string_from_network();
 Ok if(len > BUF_SIZE) {
        printf(“Too large\n”);
        return;
    }
    memcpy(buf, p, len);
}
```

```
void *memcpy(void *dest, const void *src, size_t n);
typedef unsigned int size_t;
```

What's wrong with this code?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{   Negative
    int len = read_int_from_network();
    char *p = read_string_from_network();
 Ok if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }
    memcpy(buf, p, len);
}                   Implicit cast to unsigned
```

```
void *memcpy(void *dest, const void *src, size_t n);
typedef unsigned int size_t;
```

Integer overflow vulnerabilities

What's wrong with this code?

```
void vulnerable()
{
    size_t len;
    char *buf;

    len = read_int_from_network();
    buf = malloc(len + 5);
    read(fd, buf, len);
    ...
}
```

What’s wrong with this code?

```
void vulnerable()
{
    size_t len;
    char *buf;
  HUGE
    len = read_int_from_network();
    buf = malloc(len + 5);
    read(fd, buf, len);
    ...
}
```

What's wrong with this code?

HUGE

```
void vulnerable()
{
    size_t len;
    char *buf;

    len = read_int_from_network();
    buf = malloc(len + 5); 
    read(fd, buf, len);
    ...
}
```

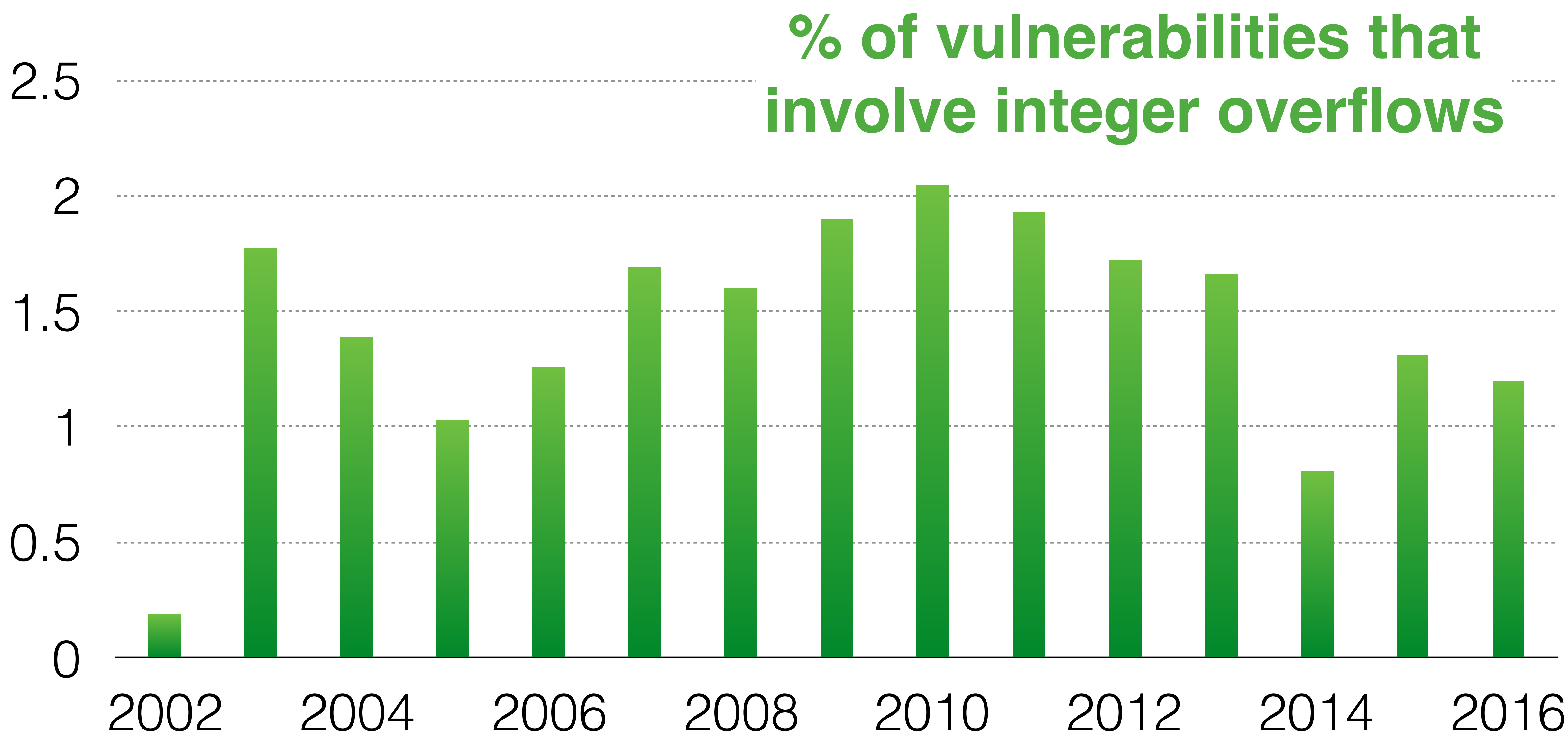
Wrap-around

What’s wrong with this code?

```
void vulnerable()
{
    size_t len;
    char *buf;
  HUGE
    len = read_int_from_network();
    buf = malloc(len + 5); Wrap-around
    read(fd, buf, len);
    ...
}
```

Takeaway: You have to know the semantics of your programming language to avoid these errors

Integer overflow prevalence



http://web.nvd.nist.gov/view/vuln/statistics