Other memory exploits

Heap overflow

- Stack smashing overflows a stack-allocated buffer
- You can also **overflow a buffer** allocated by `malloc`, which resides on the **heap**
  - What data gets overwritten?

Heap overflow

- Overflow a buffer allocated by malloc (on heap)
  - modify adjacent or nearby data
  - modify secret key to a known value
  - modify state to bypass authentication
  - modify interpreted strings used in commands
    - SQL injection
  - modify a function pointer
- Overwrite heap metadata
  - if p = malloc(...),
    then modifying *(p-1) corrupts heap header

Heap overflow example

```
typedef struct {
    char buff[LEN];
    int (*cmp)(char*, char*);
} xs;

int foo(xs *s, char *a1, char* a2) {
    strcpy( s->buff, a1 );
    strcat( s->buff, a2 );
    return s->cmp(s->buff, "file:foobar")
}
```

if ***strlen(a1) + strlen(a2) >= LEN***,
then ***s->cmp*** is overwritten

Heap read overflow

- Read data adjacent or nearby heap buffer
  - Leak secret info // eg, Heartbleed
  - Format string vulnerability

Format string vulnerabilities

Formatted I/O

- Recall: C's **`printf`** family of functions
- Format specifiers, list of arguments
  - Specifier indicates type of argument (%s, %i, etc.)
  - Position in string indicates argument to print

```
void print_record(int age, char *name)
{
  printf("Name: %s\tAge: %d\n",name,age);
}
```

What's the difference?

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

Attacker controls the format string

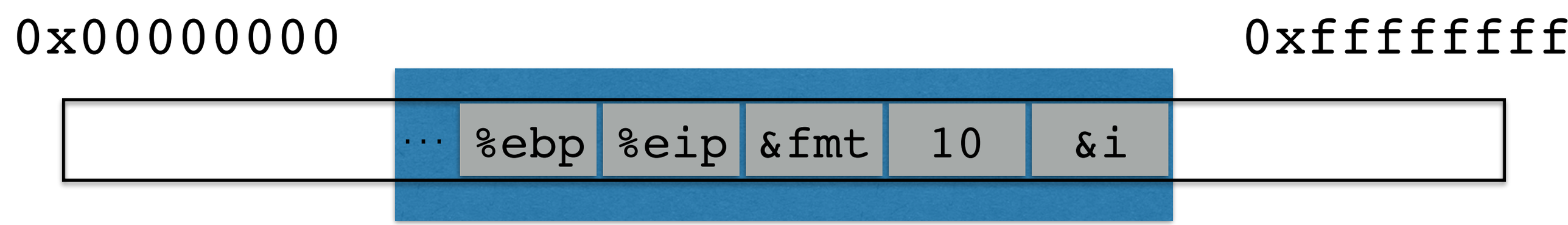
```
void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf("%s",buf);
}
```

`printf` format strings

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

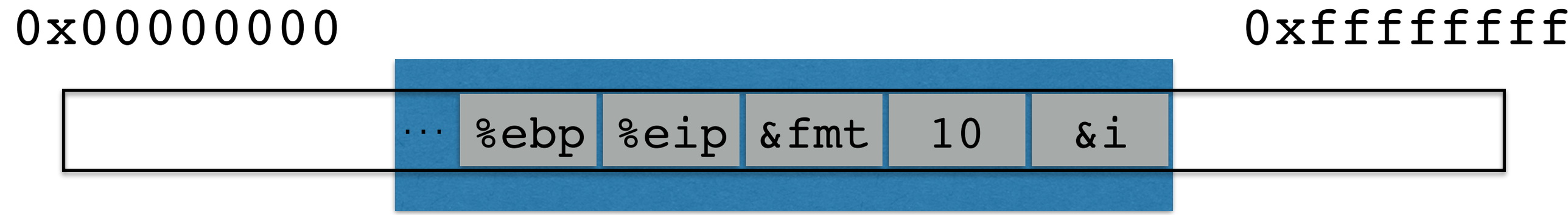
`printf` format strings

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



`printf` format strings

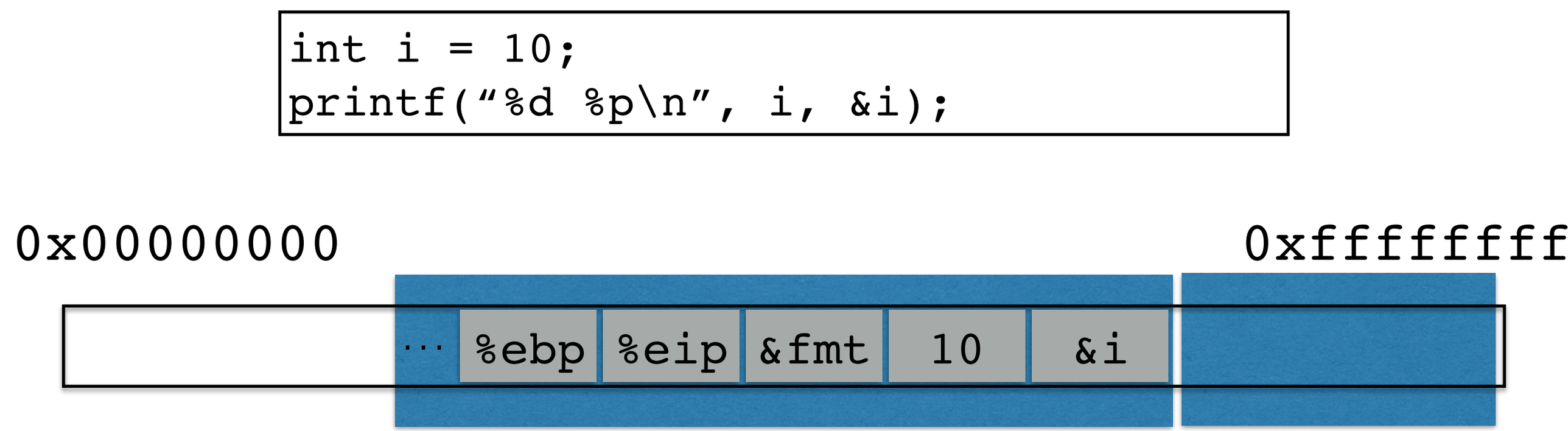
```
int i = 10;
printf(“%d %p\n”, i, &i);
```



`printf`’s stack frame

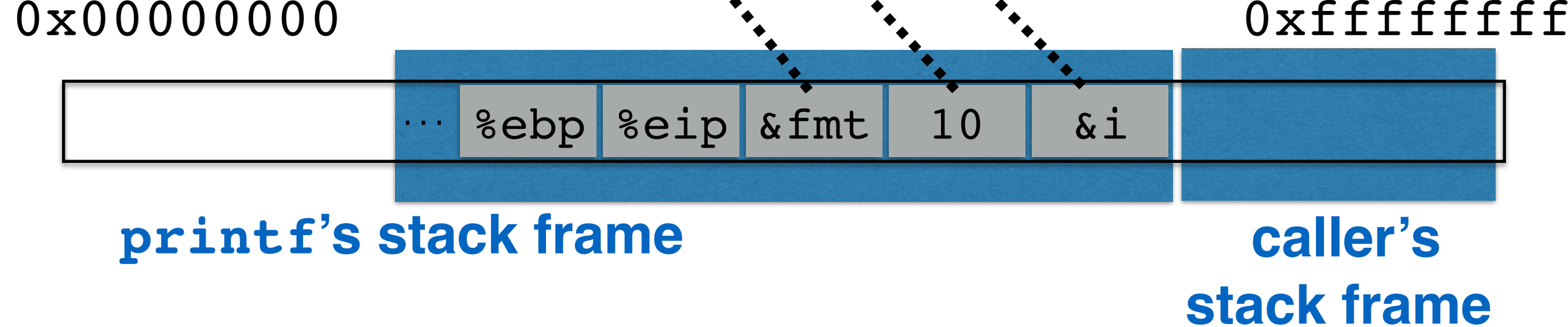
`printf` format strings

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



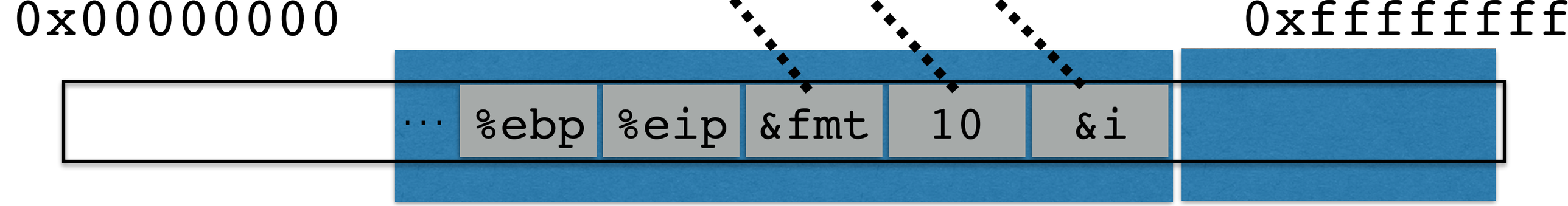
`printf` format strings

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



`printf` format strings

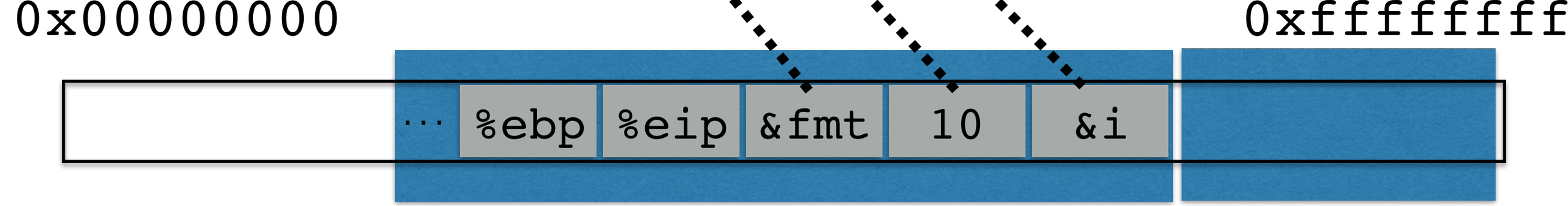
```
int i = 10;
printf("%d %p\n", i, &i);
```



- 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

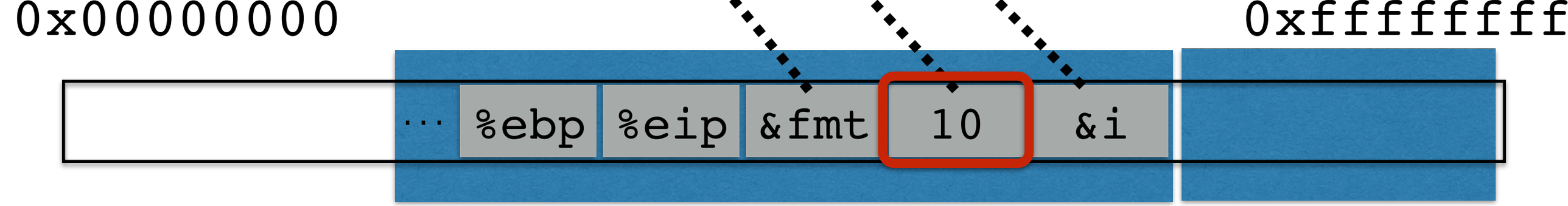
```
int i = 10;
printf("%d %p\n", i, &i);
```



- 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

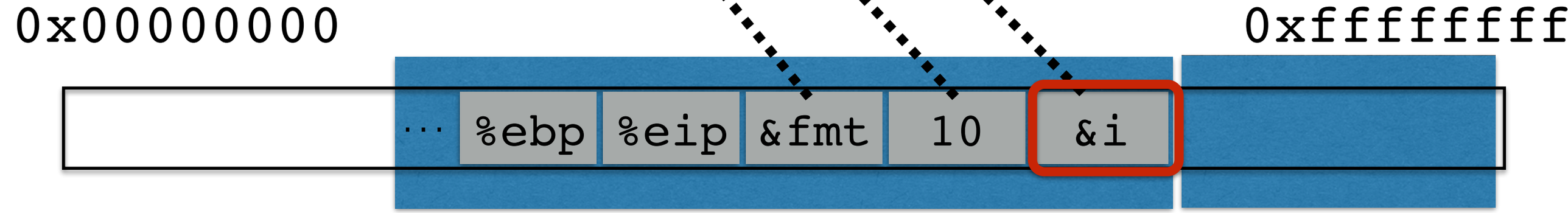
```
int i = 10;
printf("%d %p\n", i, &i);
```



- 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

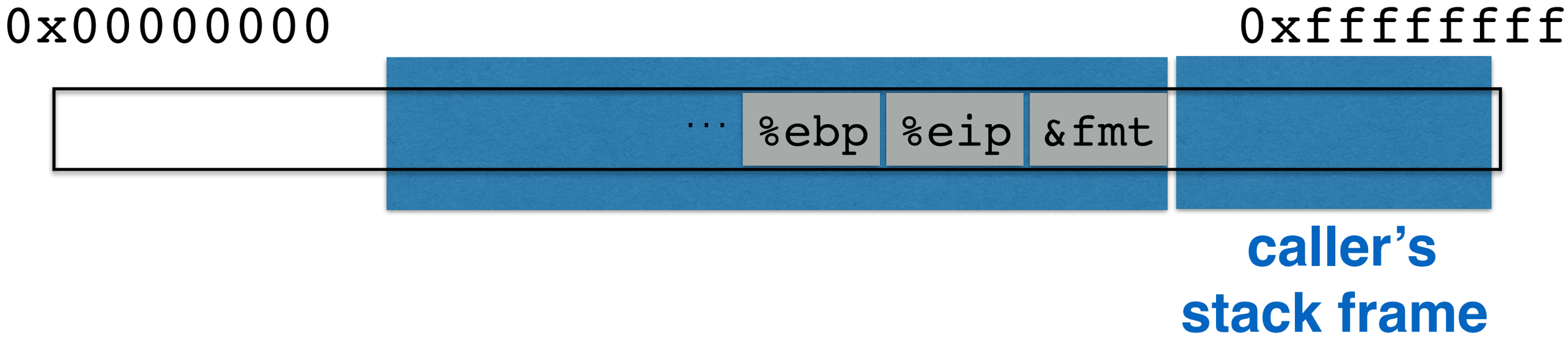
```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

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

“%d %x"

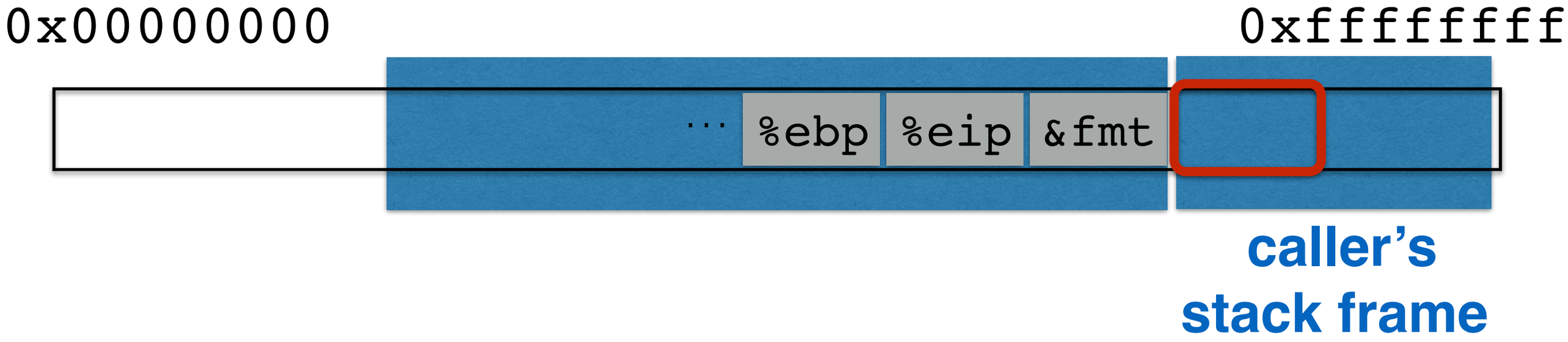
```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

"%d %x"



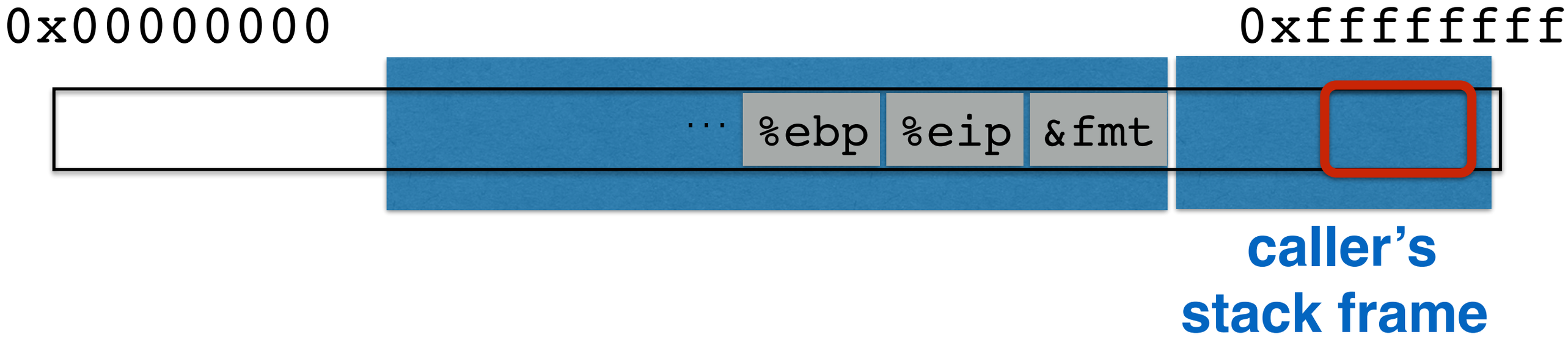
```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

“%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

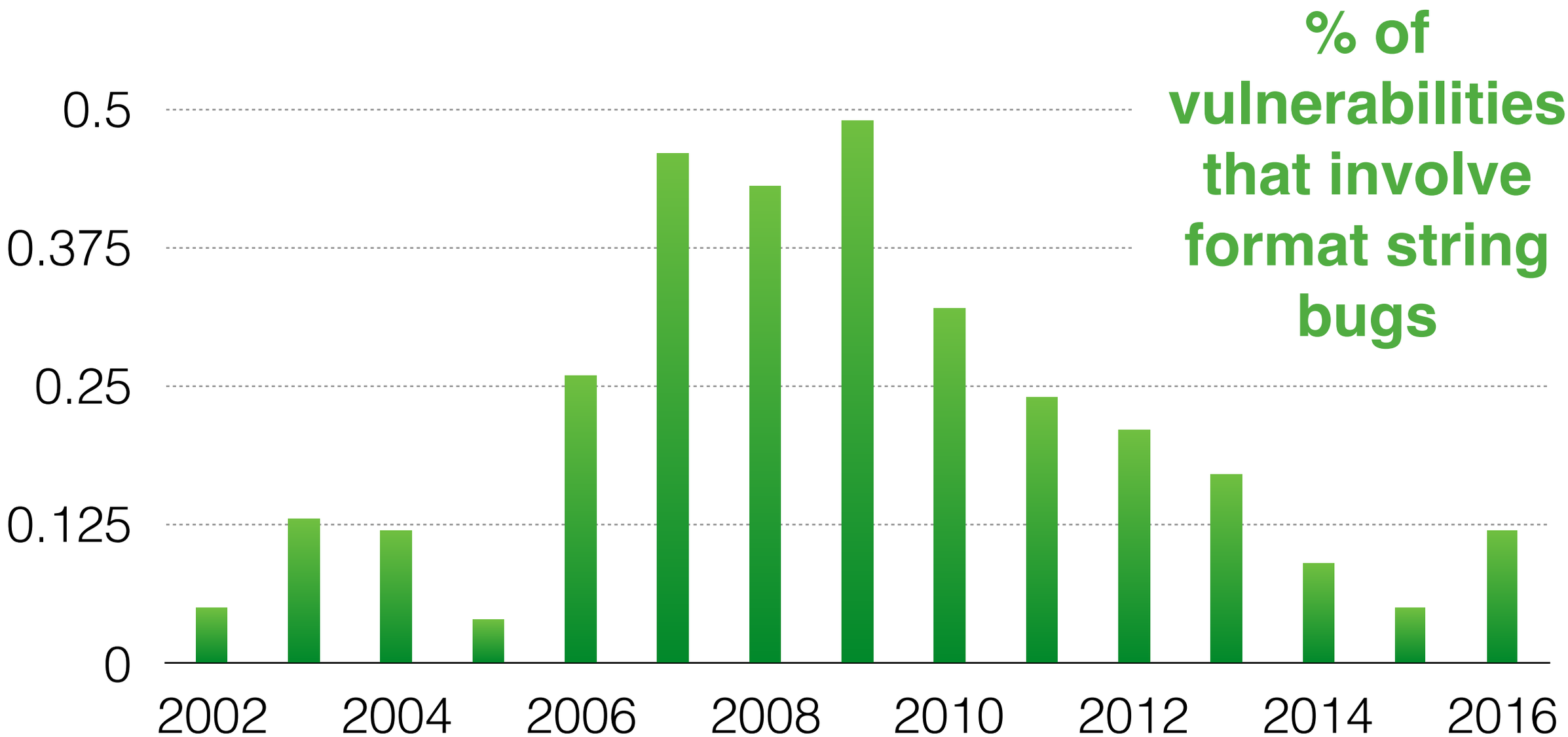
Format string vulnerabilities

- `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()
{
    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);
}
```

Negative

```
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()
{
    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);
}
```

Negative

Ok

```
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()
{
    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);
}
```

Negative

Ok

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;

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

HUGE

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);
    ...
}
```

HUGE

Wrap-around

What's wrong with this code?

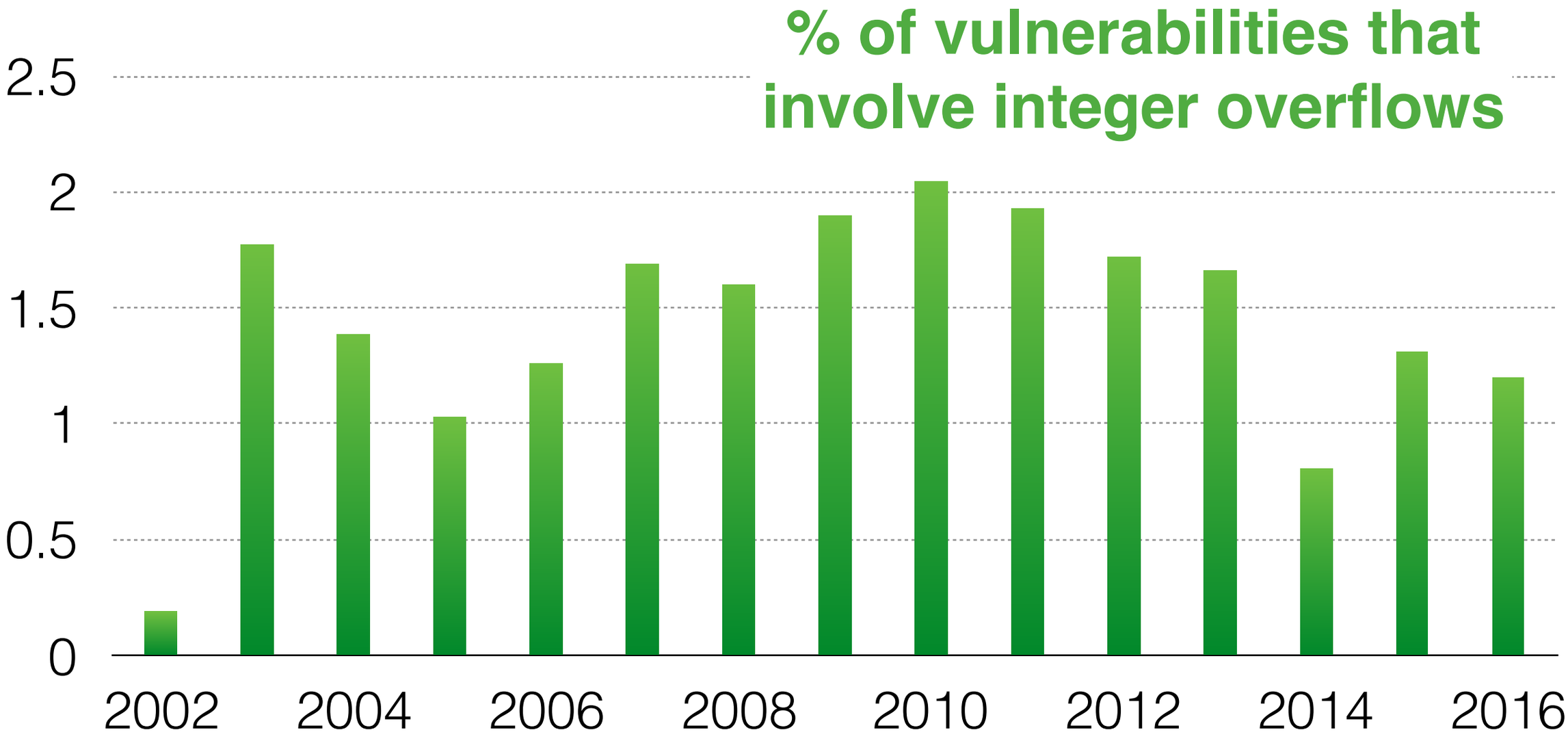
```
void vulnerable()
{
    size_t len;
    char *buf;

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

HUGE

Wrap-around

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

Defenses

Recall our challenges

How can we make these even more difficult?

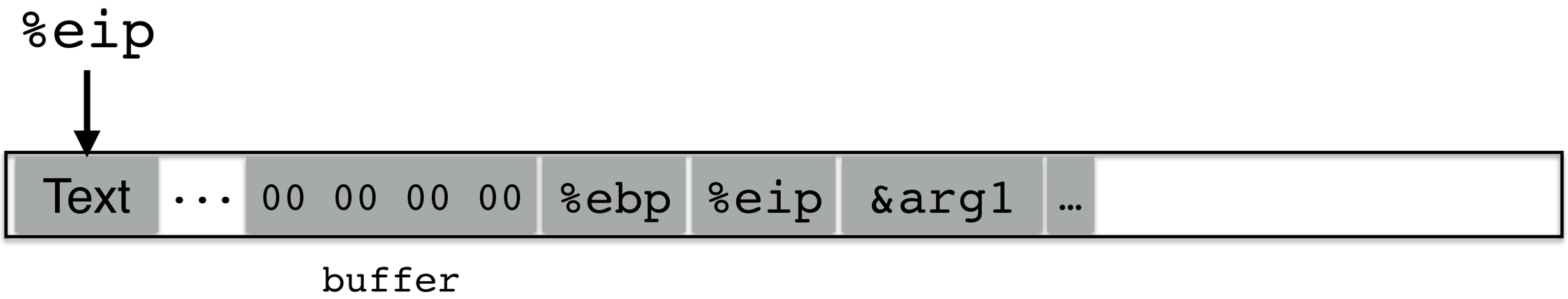
- Putting code into the memory (no zeroes)
- Getting %eip to point to our code (dist buff to stored eip)
- Finding the return address (guess the raw addr)

Defense: Canaries in stack

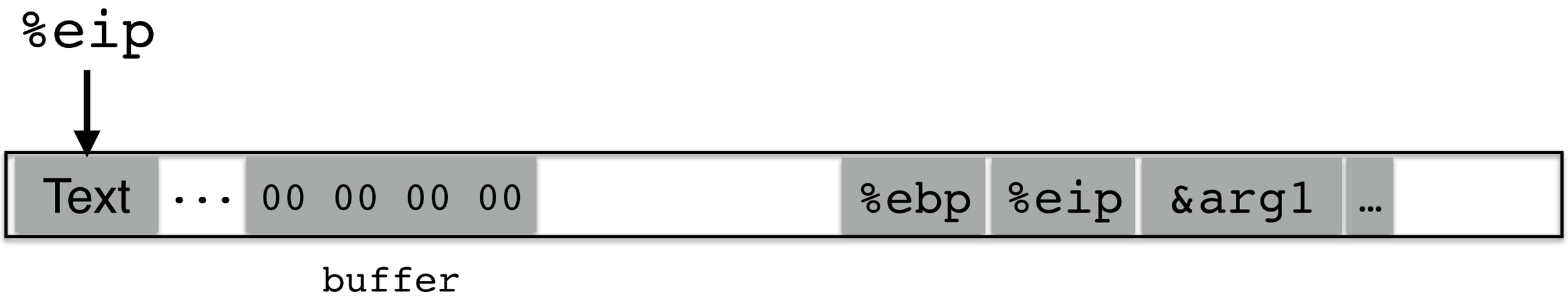
- Compiler inserts some extra code in function
  - At entry: after pushing ebp, push an unlikely bit pattern (canary)
  - At return: before popping ebp, check the canary
    - terminate program if canary has changed

- Counter-attack: rewrite the canary in overflow
  - guess the canary
  - read the canary (via, eg, printf vulnerabilities)

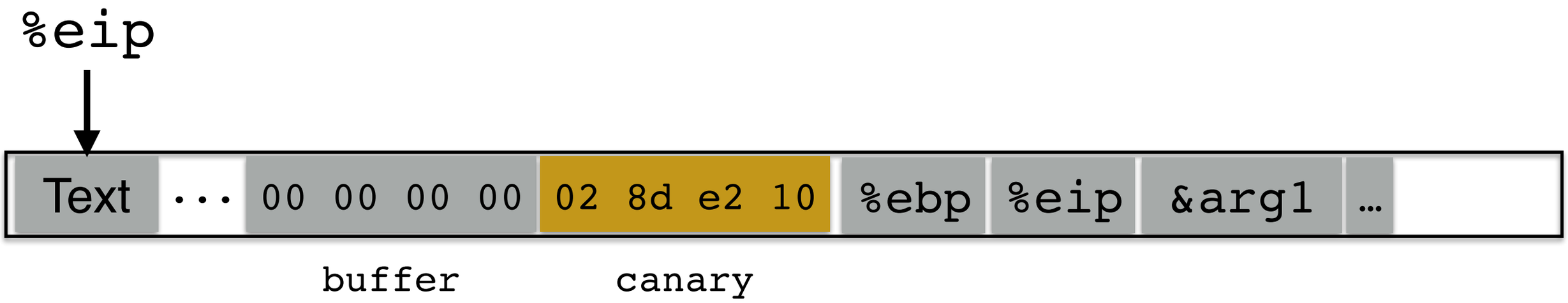
Detecting overflows with canaries



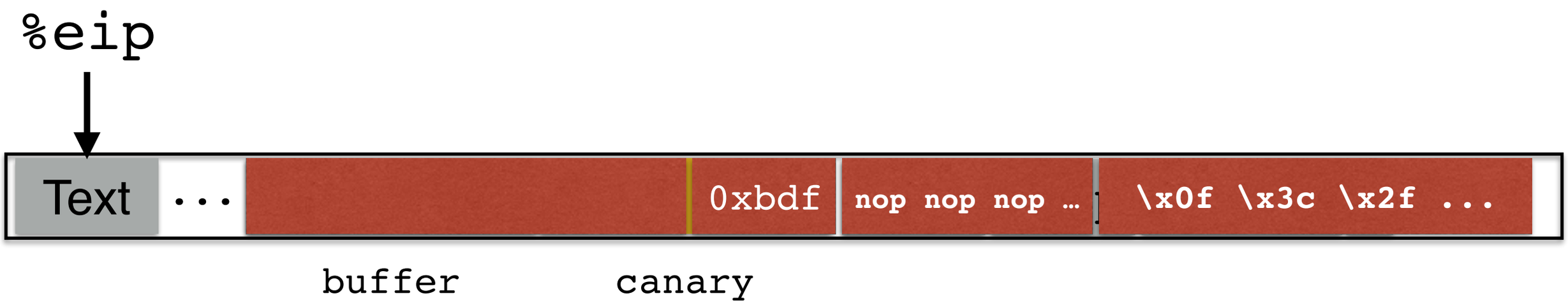
Detecting overflows with canaries



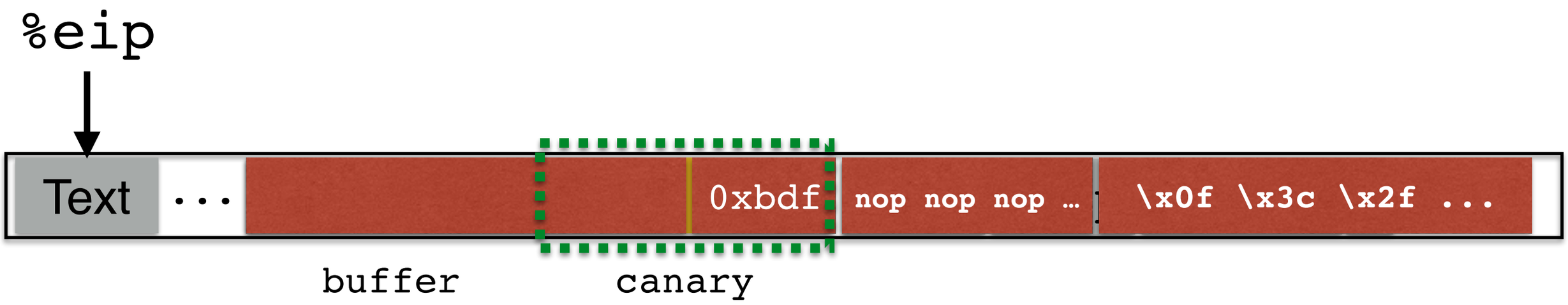
Detecting overflows with canaries



Detecting overflows with canaries

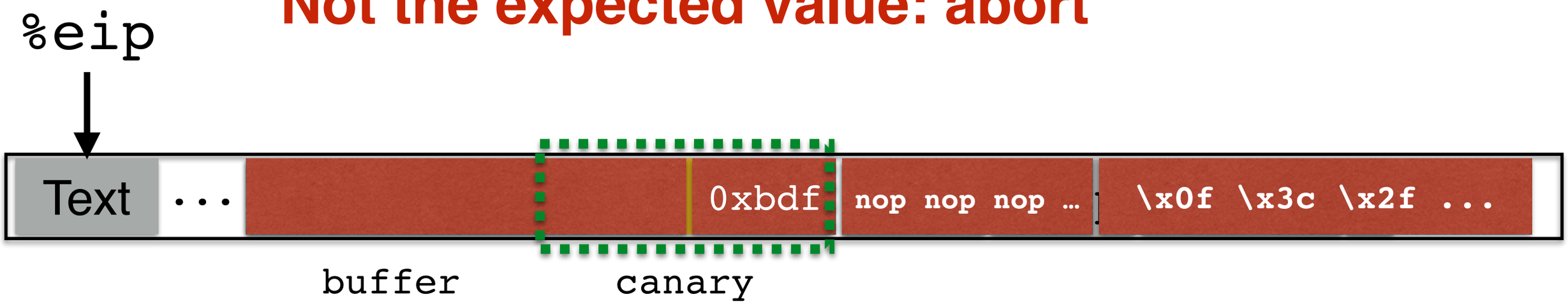


Detecting overflows with canaries

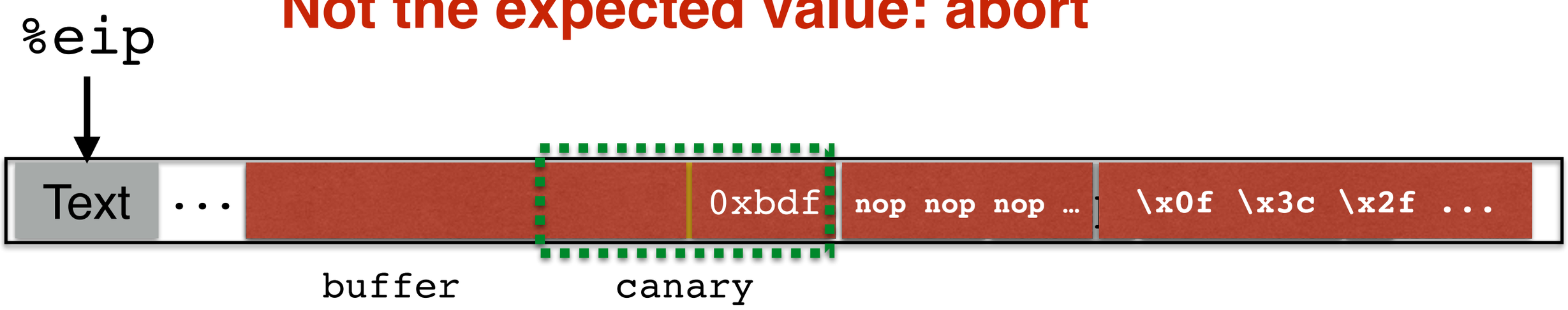


Detecting overflows with canaries

Not the expected value: abort



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

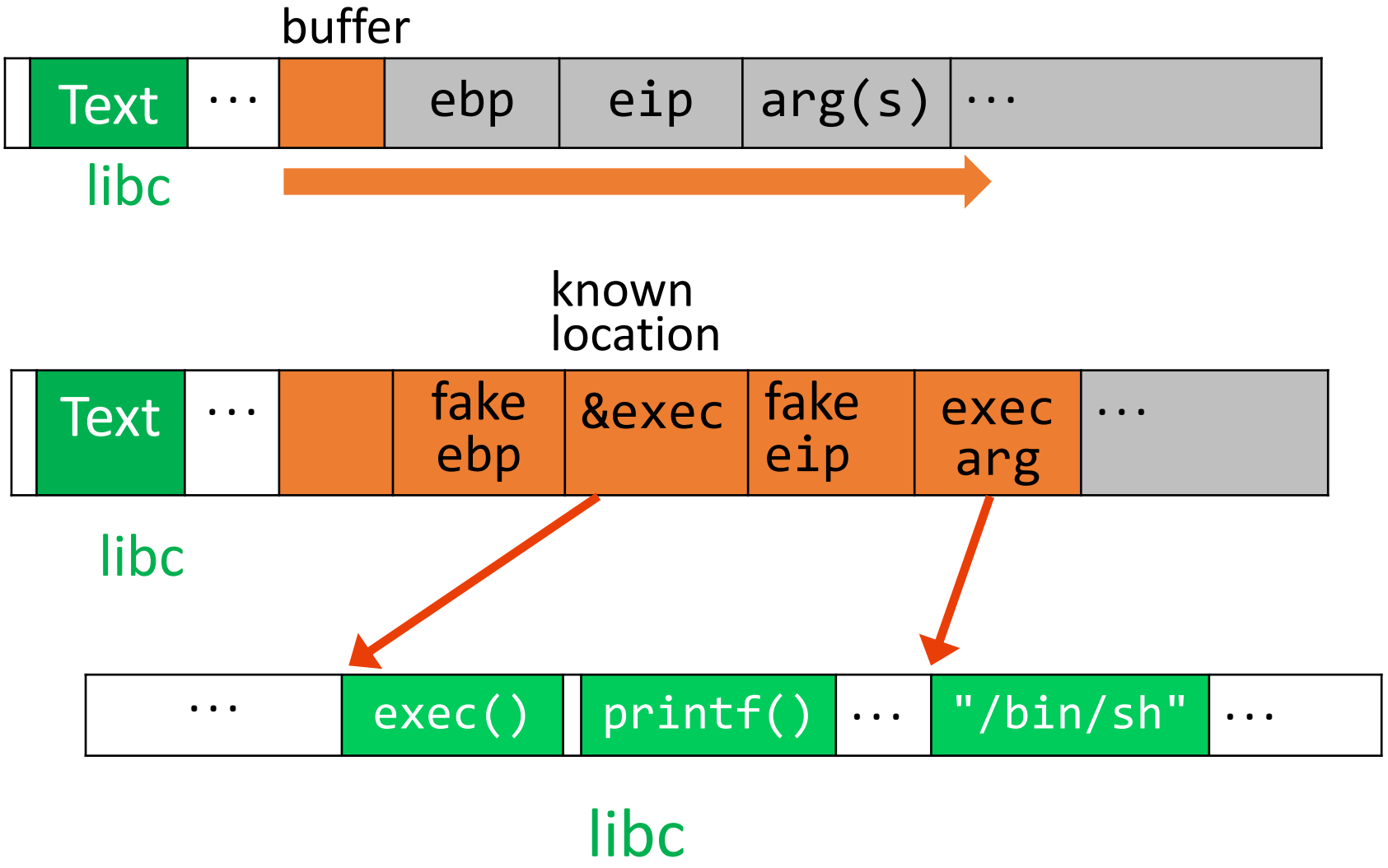
3. Random XOR canaries
   - Same as random canaries
   - But store canary XOR some control info, instead

Defense: No write-execute memory

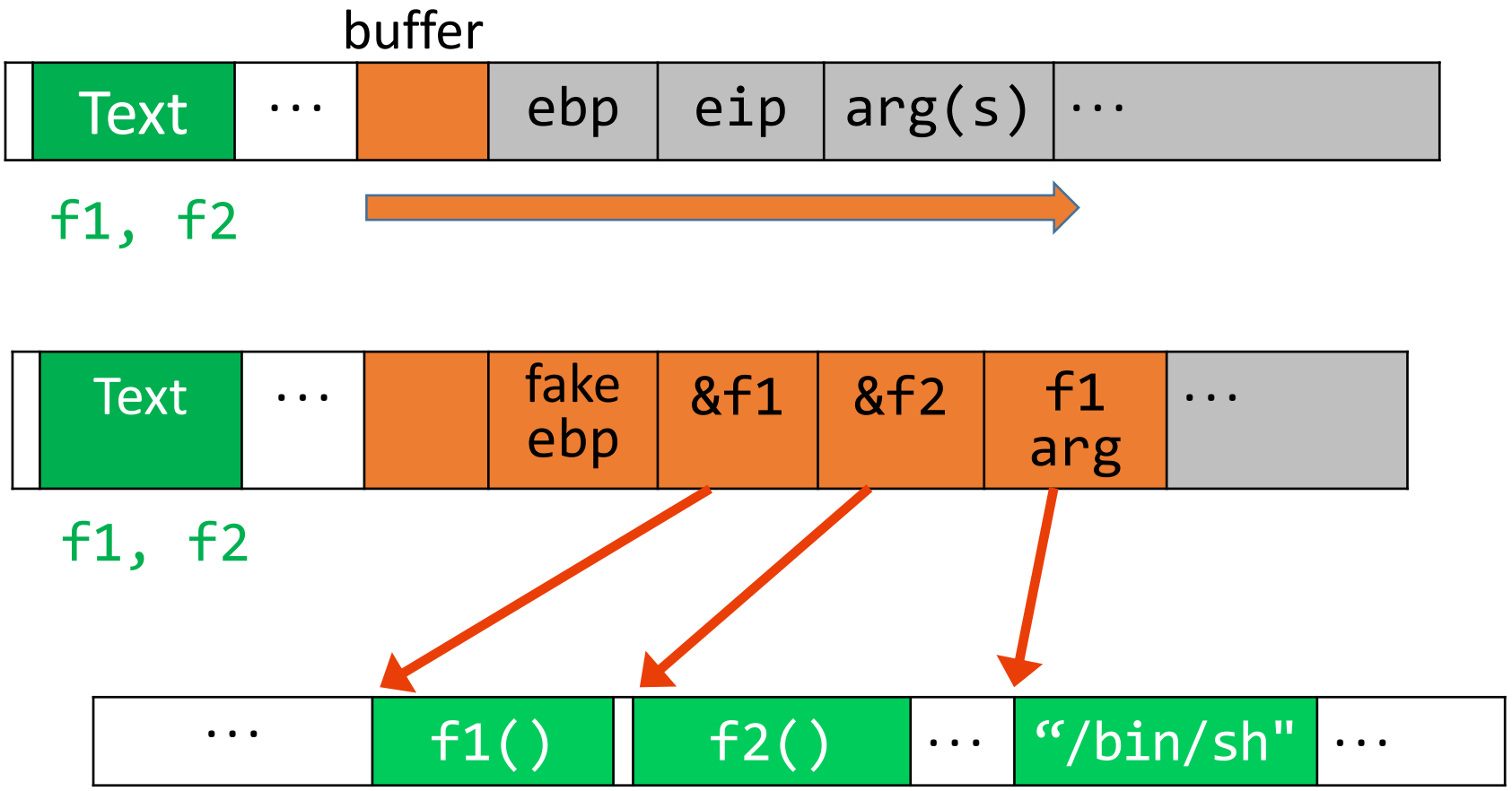
- Make the stack non-executable
  - Set stack page map entries to non-executable
  - Attempt to execute from stack results in exception
  - Stops code injection into stack
- Generalize: No write-execute memory (W-xor-X)
- Counter-attack: attack with existing code
  - return-to-libc, return-oriented programming

Return-to-libc

Return-to-libc: exec("/bin/sh")

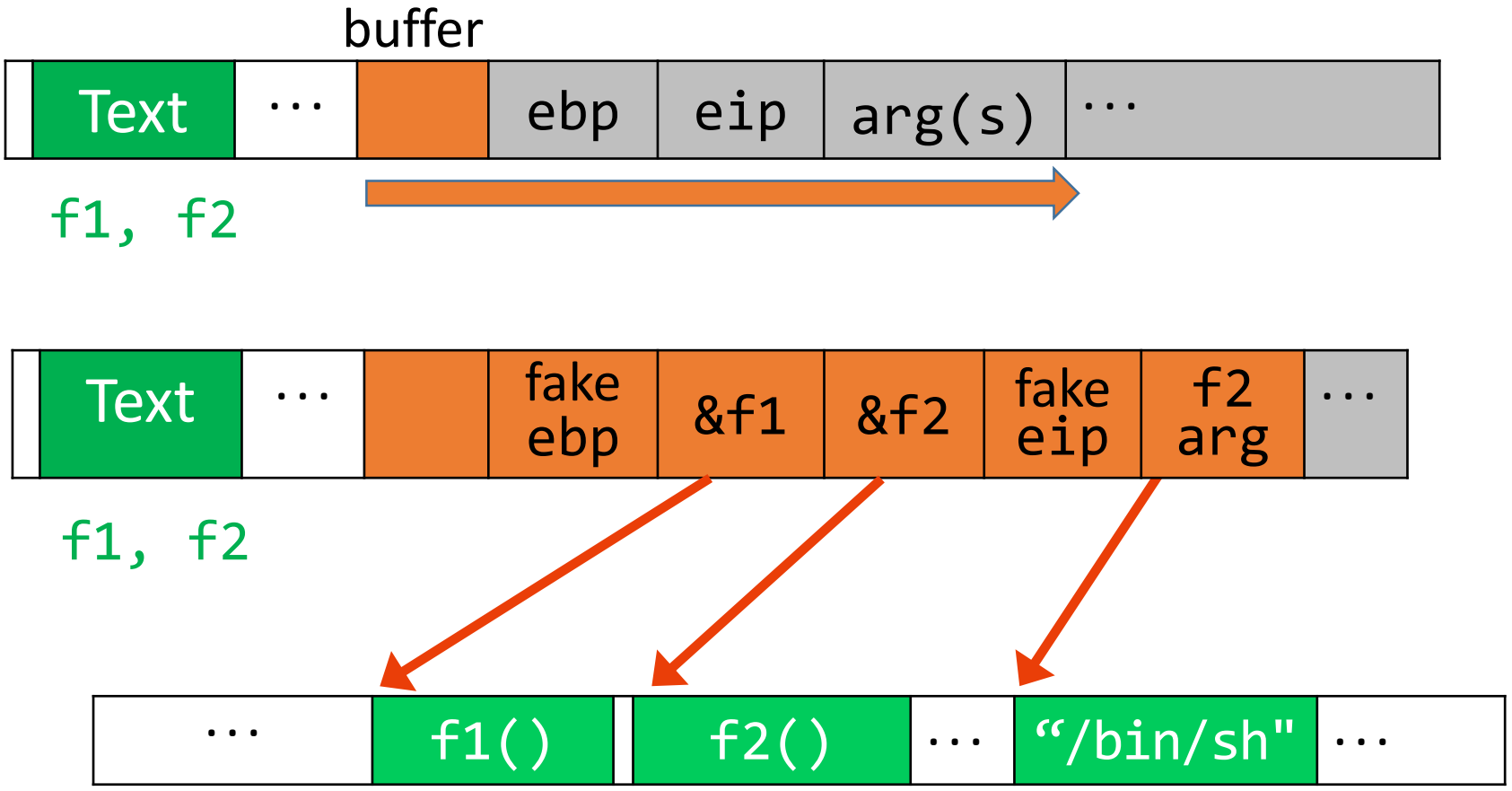


Return-to-libc: `f1("/bin/sh"); f2();`

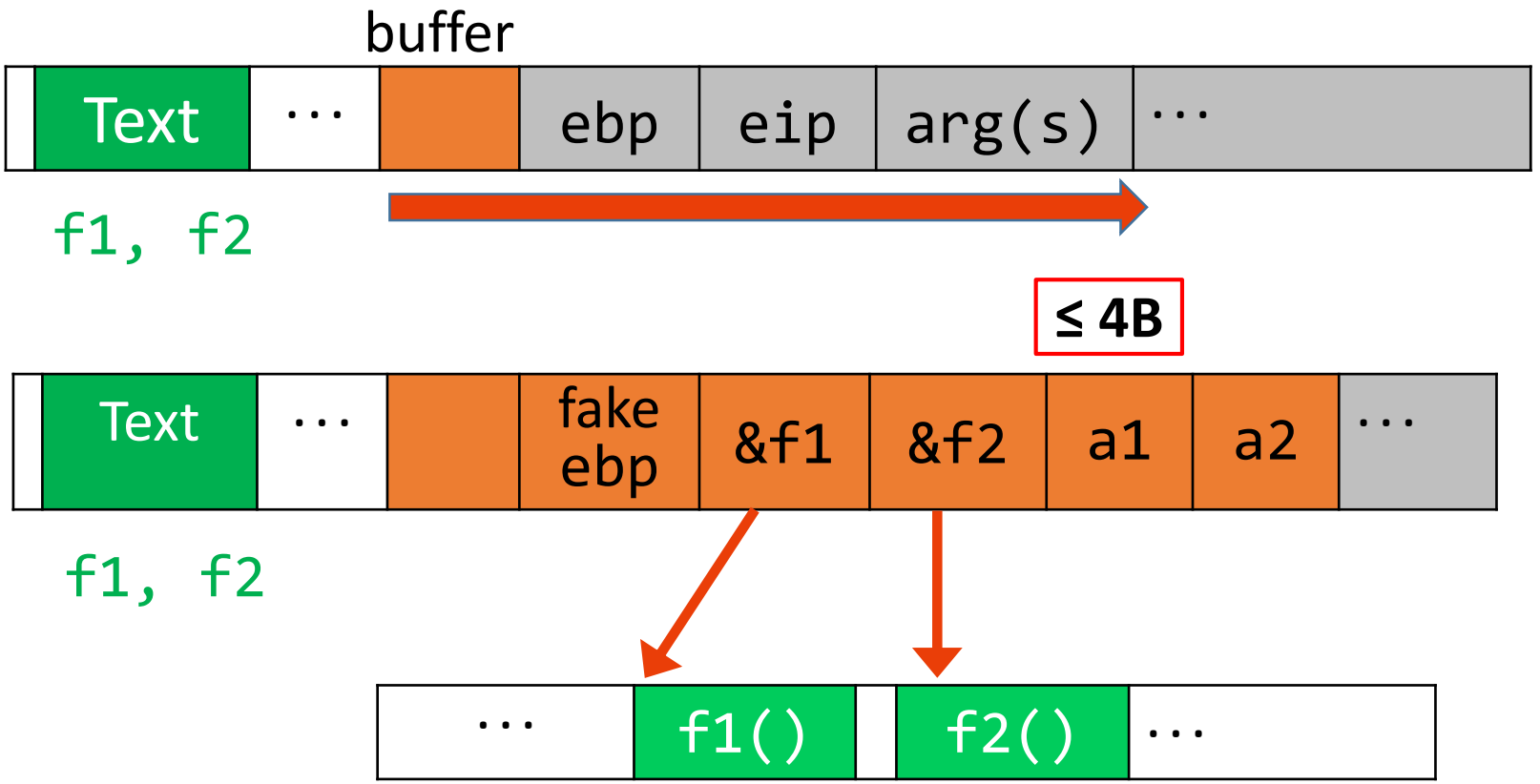


Note that `f1()` and `f2()` need not be in `libc`

Return-to-libc: `f1(); f2("/bin/sh");`



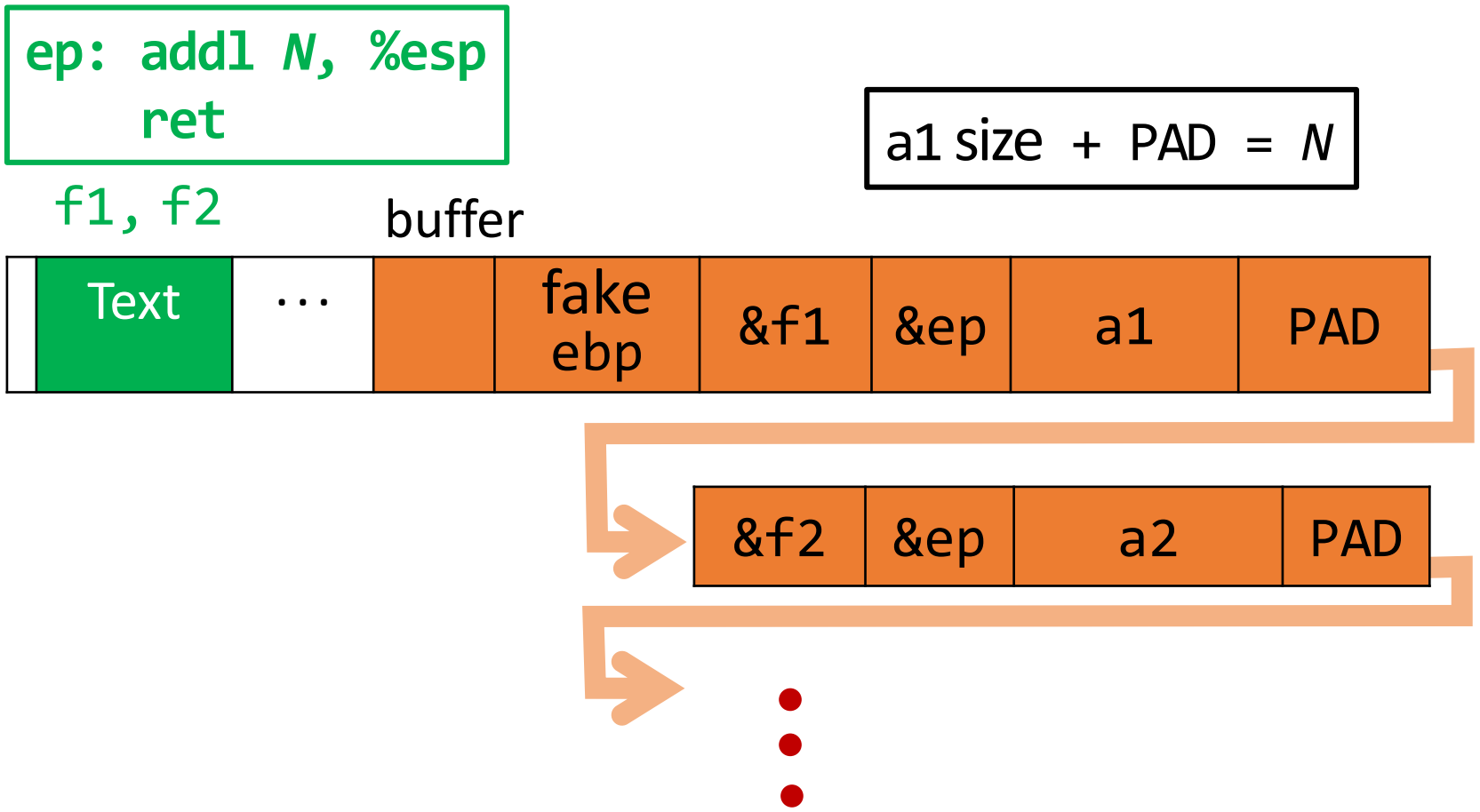
Return-to-libc: `f1(a1); f2(a2);`



- Cannot do `f1(a1); f2(a2); f3();` // unless `a1 == &f3`
- Also `a1` must fit in 4 bytes (fake eip space)

Return-to-libc: `f1(a1); f2(a2);...` arg size ≤ *N*

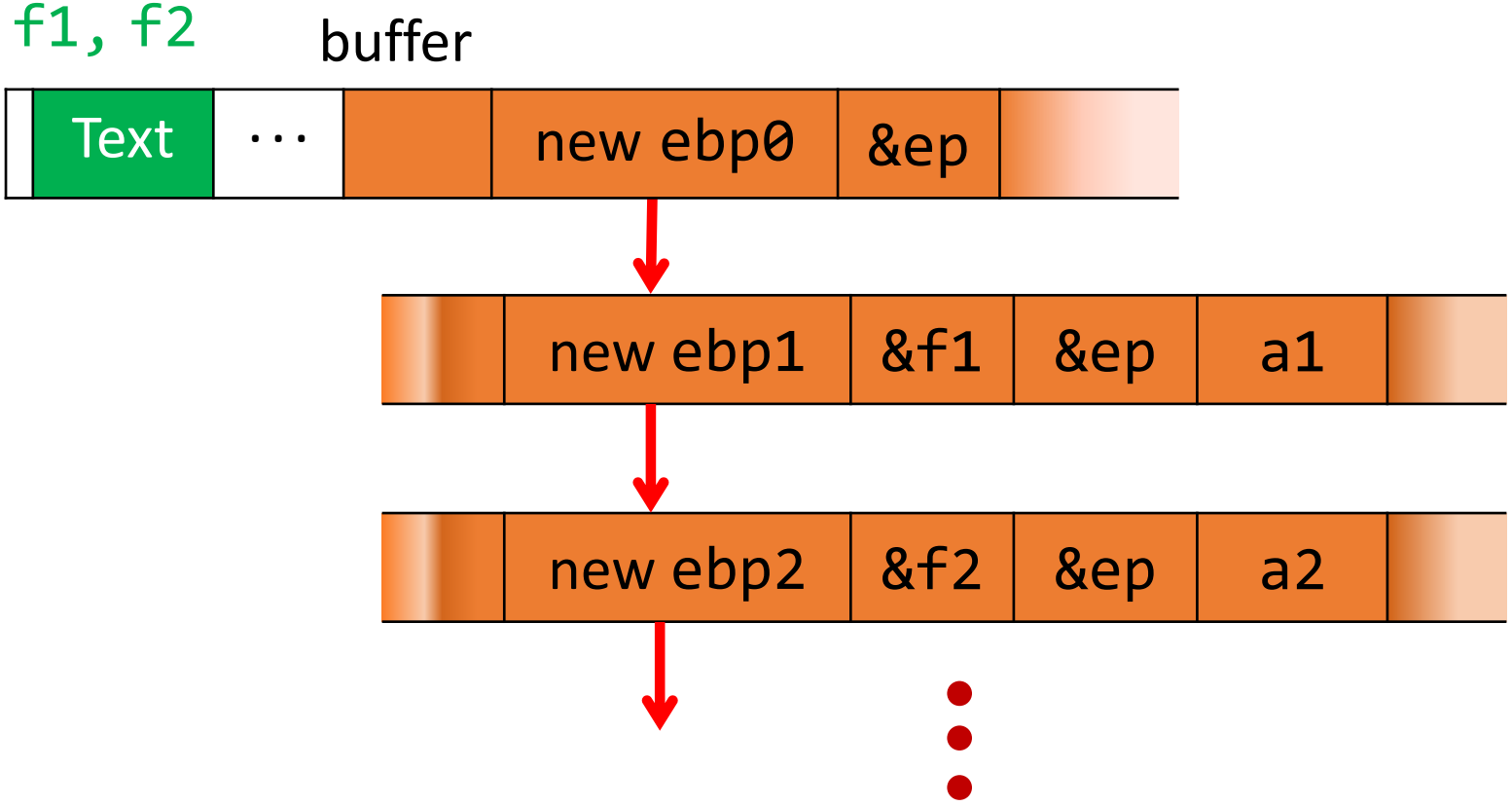
Nergal 2001, Advanced return-into-libc exploits



Return-to-libc: `f1(a1); f2(a2);...` any arg size

Nergal 2001

```
ep: esp←ebp
    pop ebp
    ret
```



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?

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
  - **Attack response:** Construct needed functionality using **return oriented programming (ROP)**

Return oriented programming (ROP)

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

- 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

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

- **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/