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

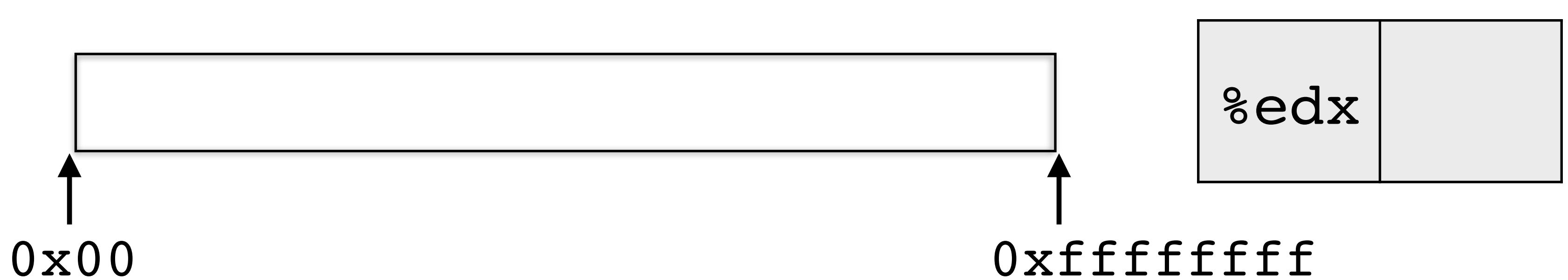
- Gadgets are instruction groups that end with **`ret`**

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

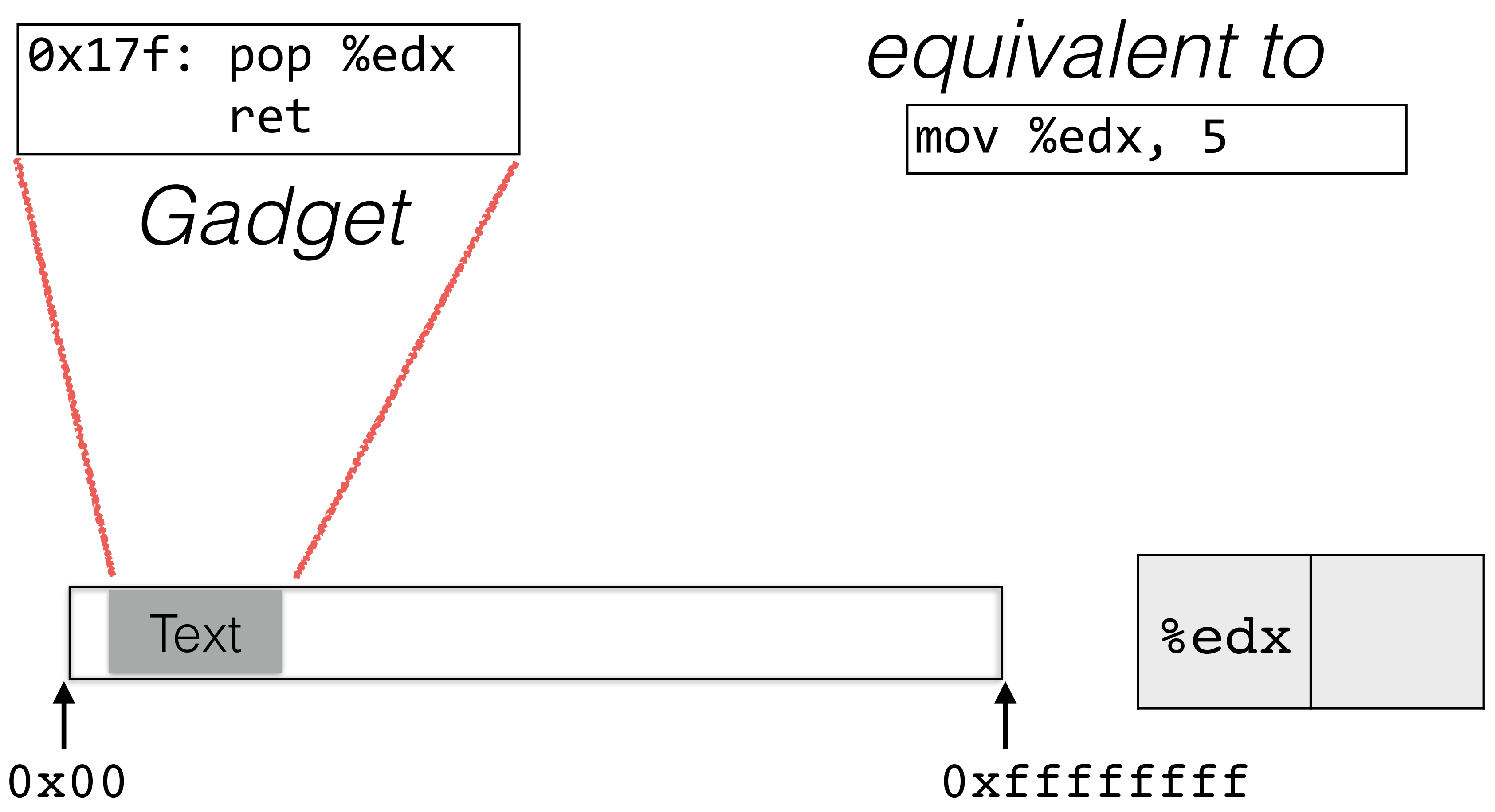
Simple example

equivalent to

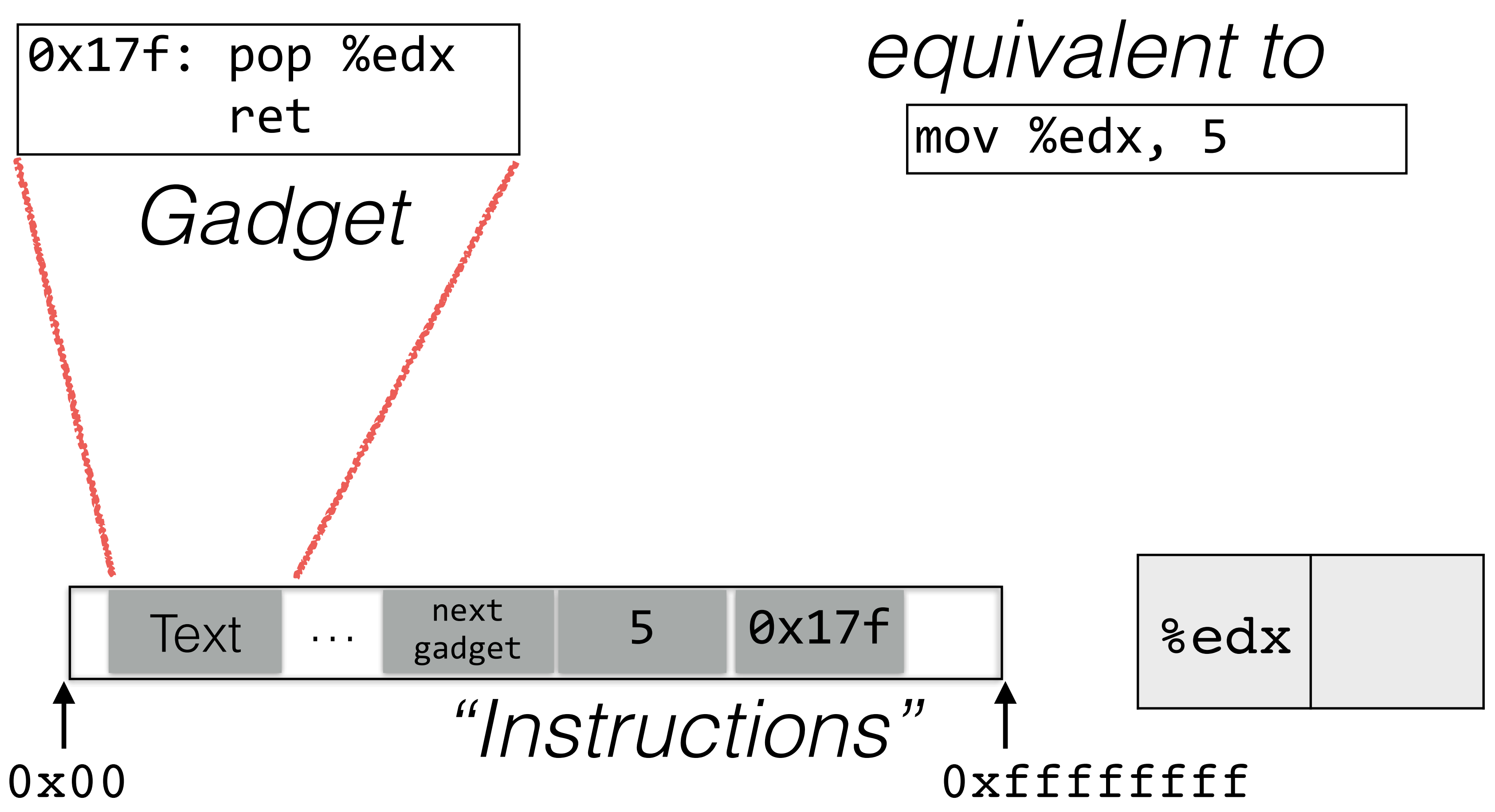
```
mov %edx, 5
```



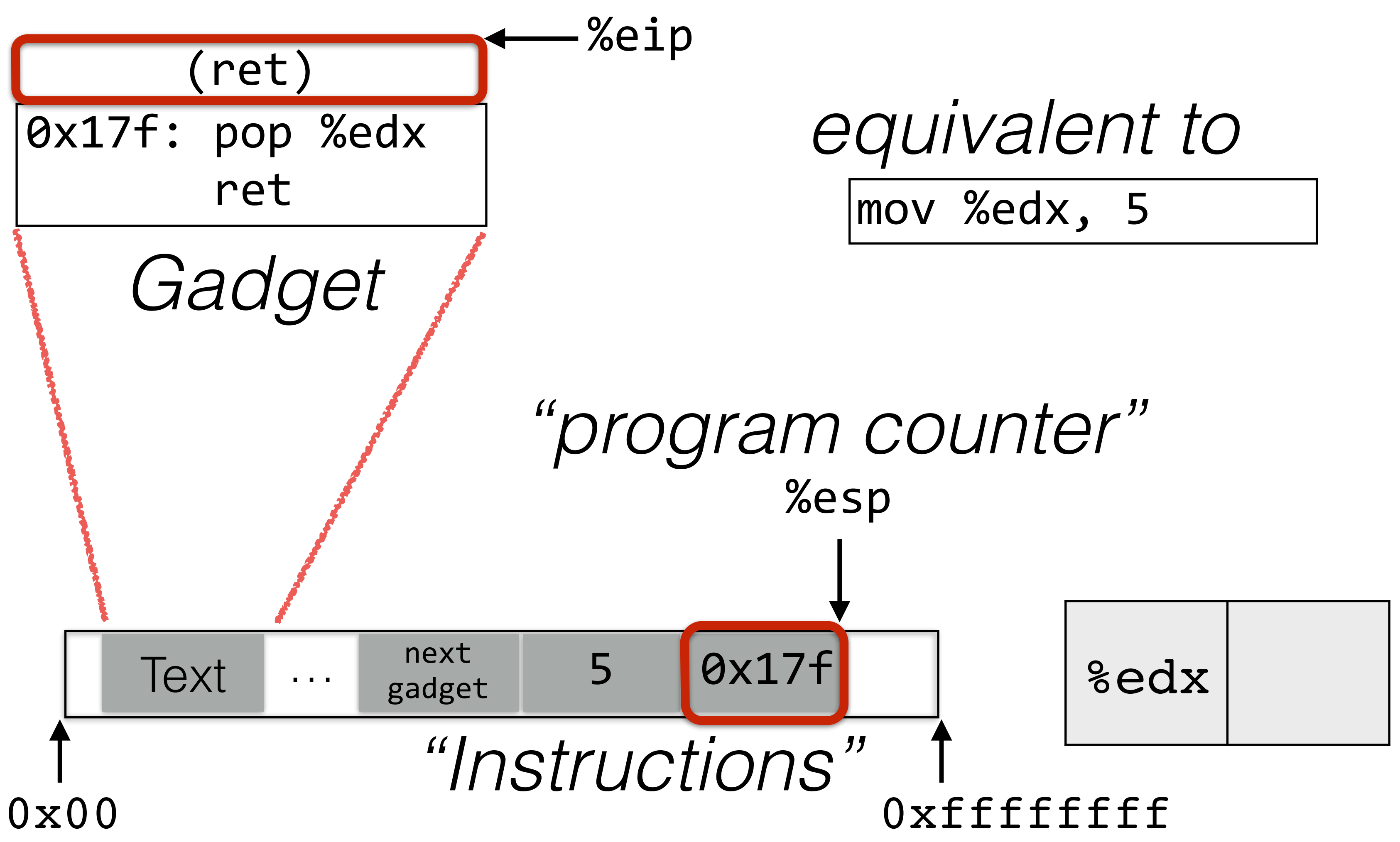
Simple example



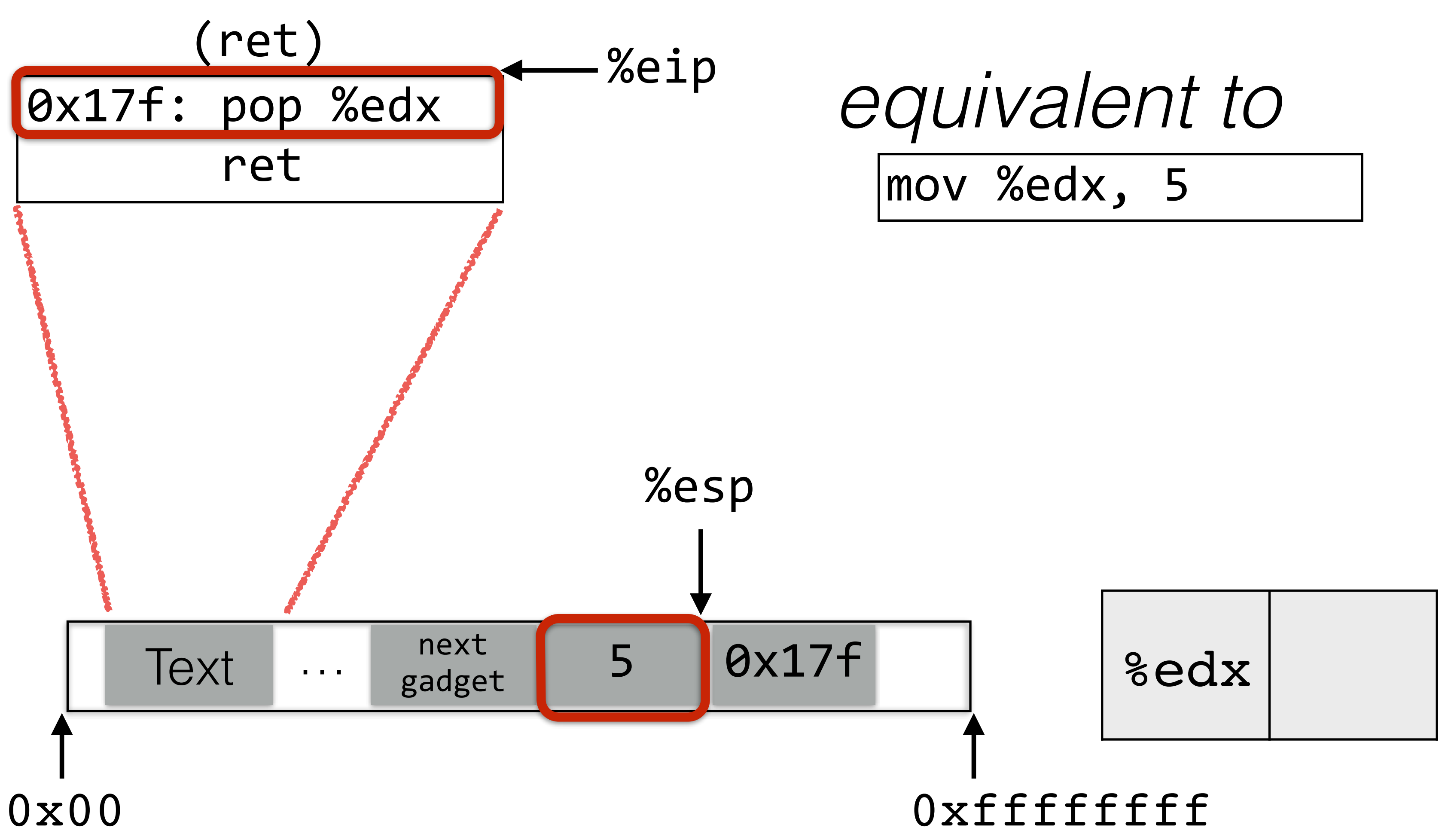
Simple example



Simple example



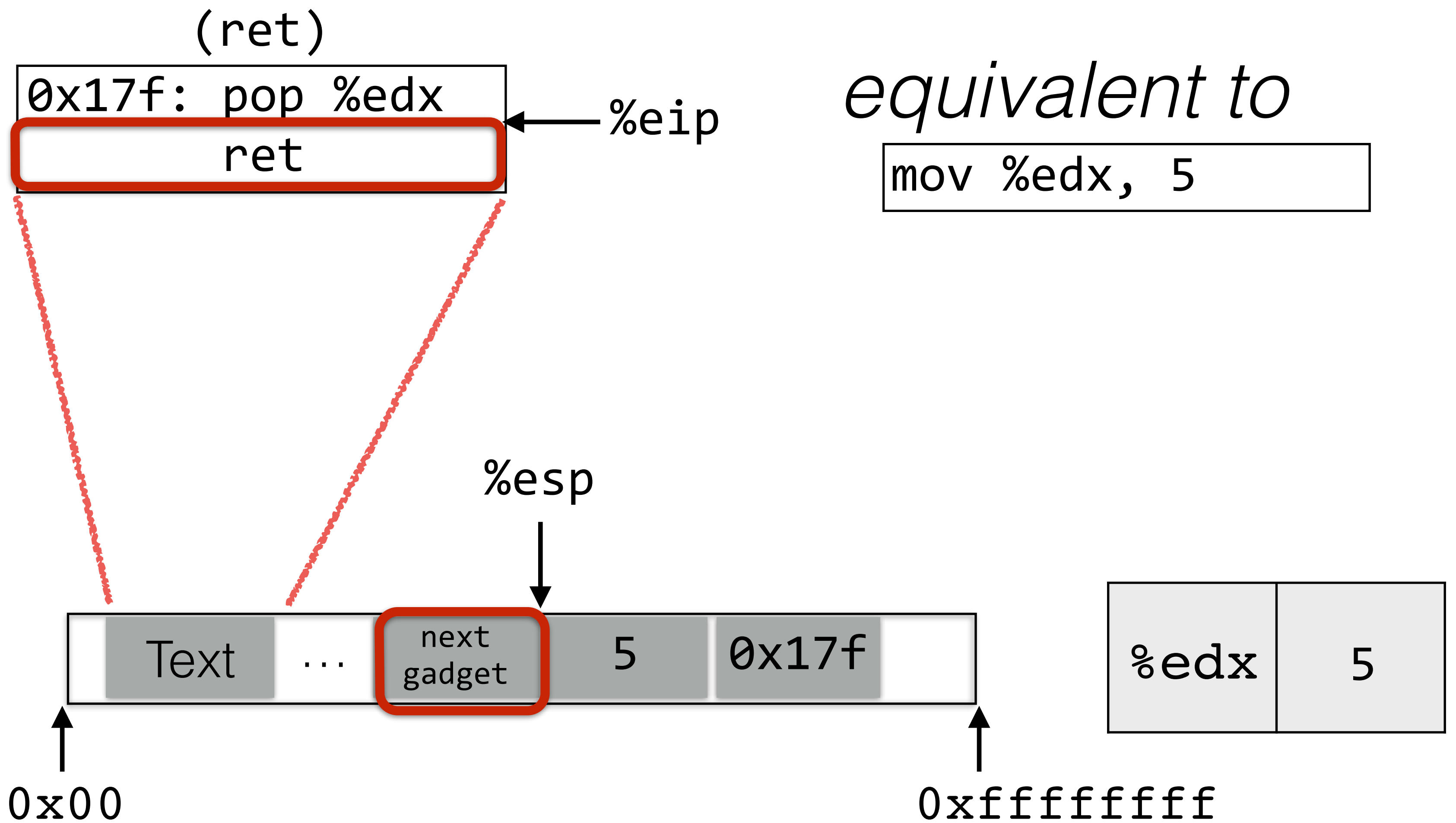
Simple example



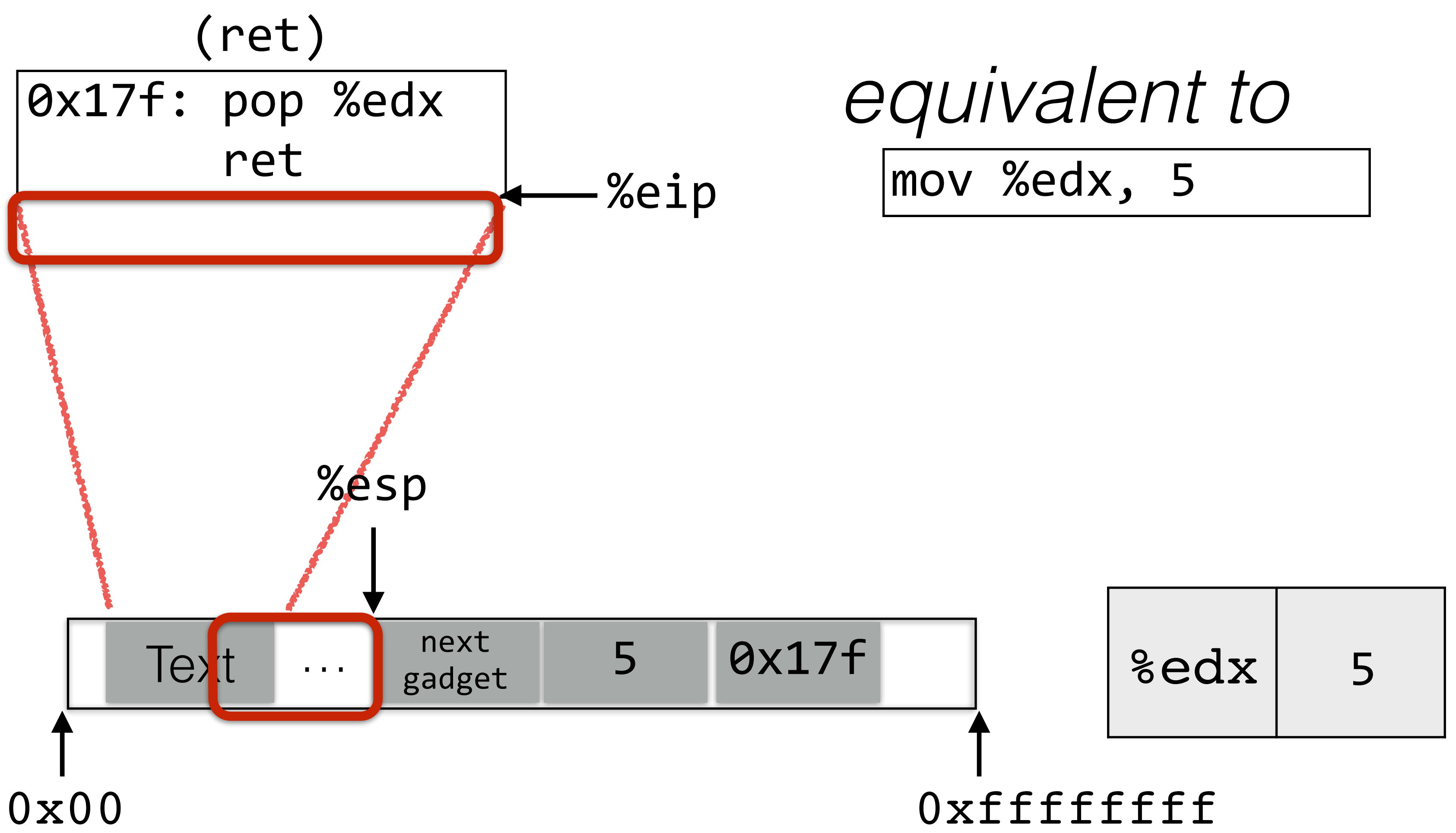
Simple example

equivalent to

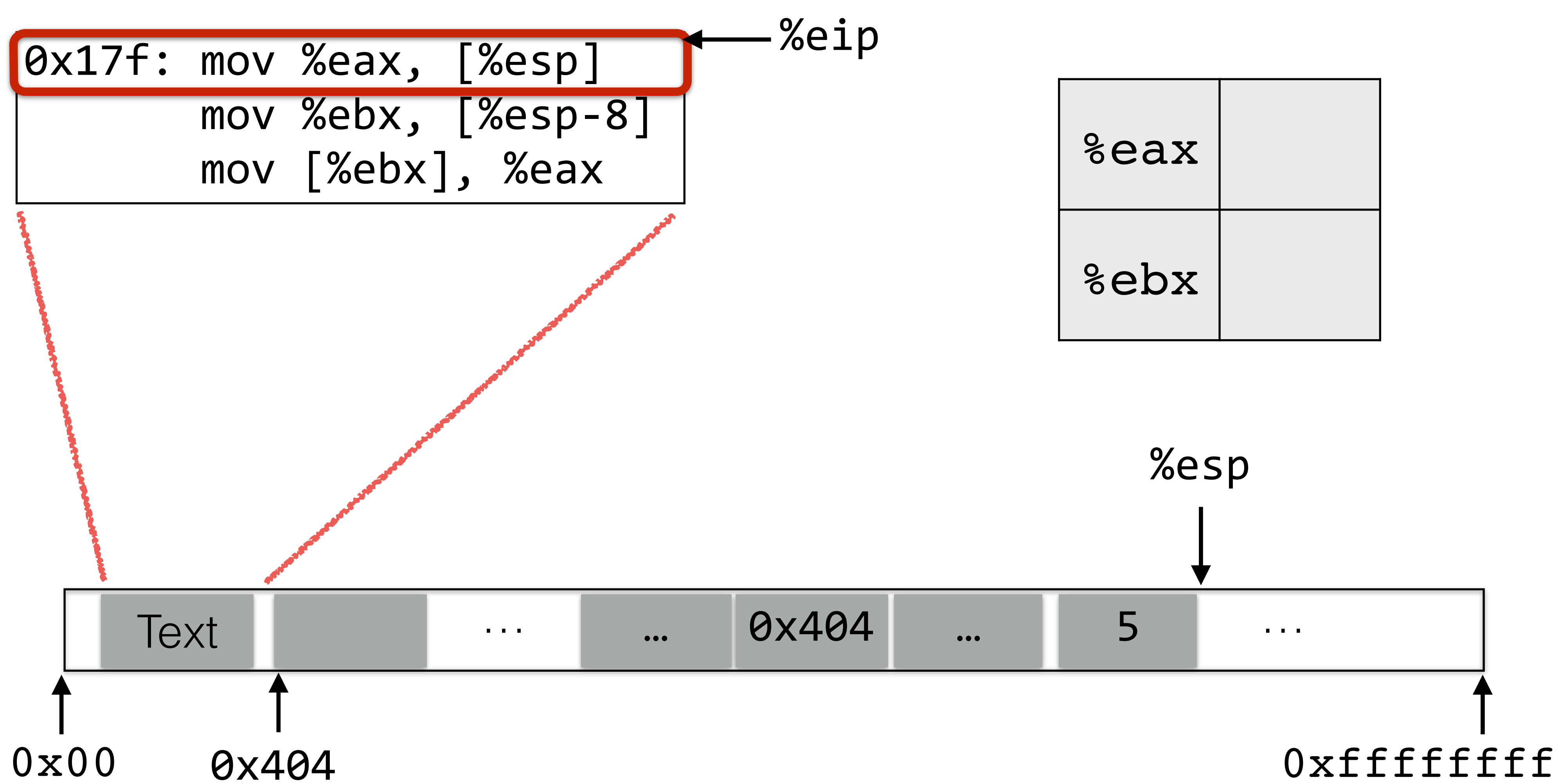
```
mov %edx, 5
```



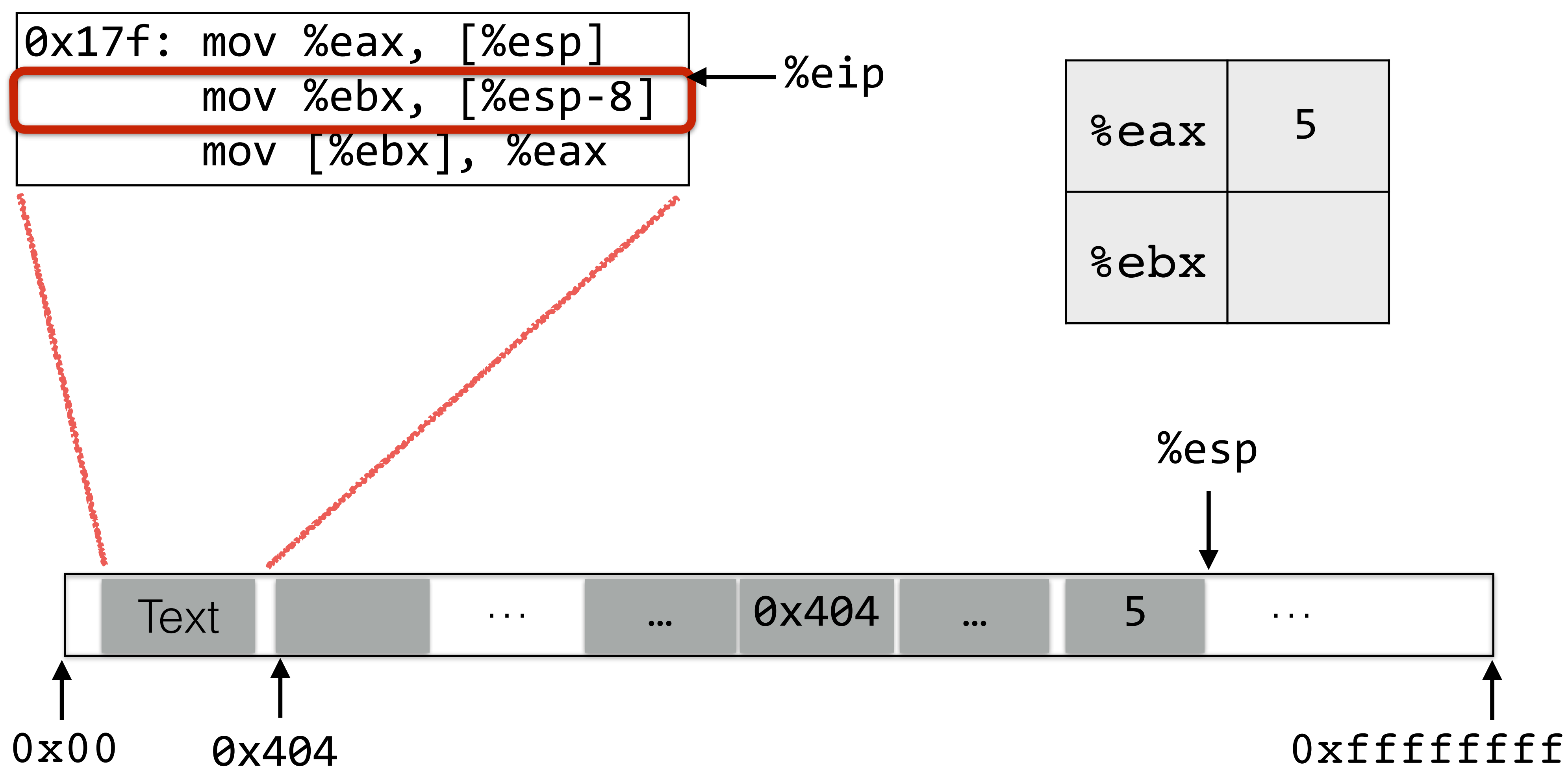
Simple example



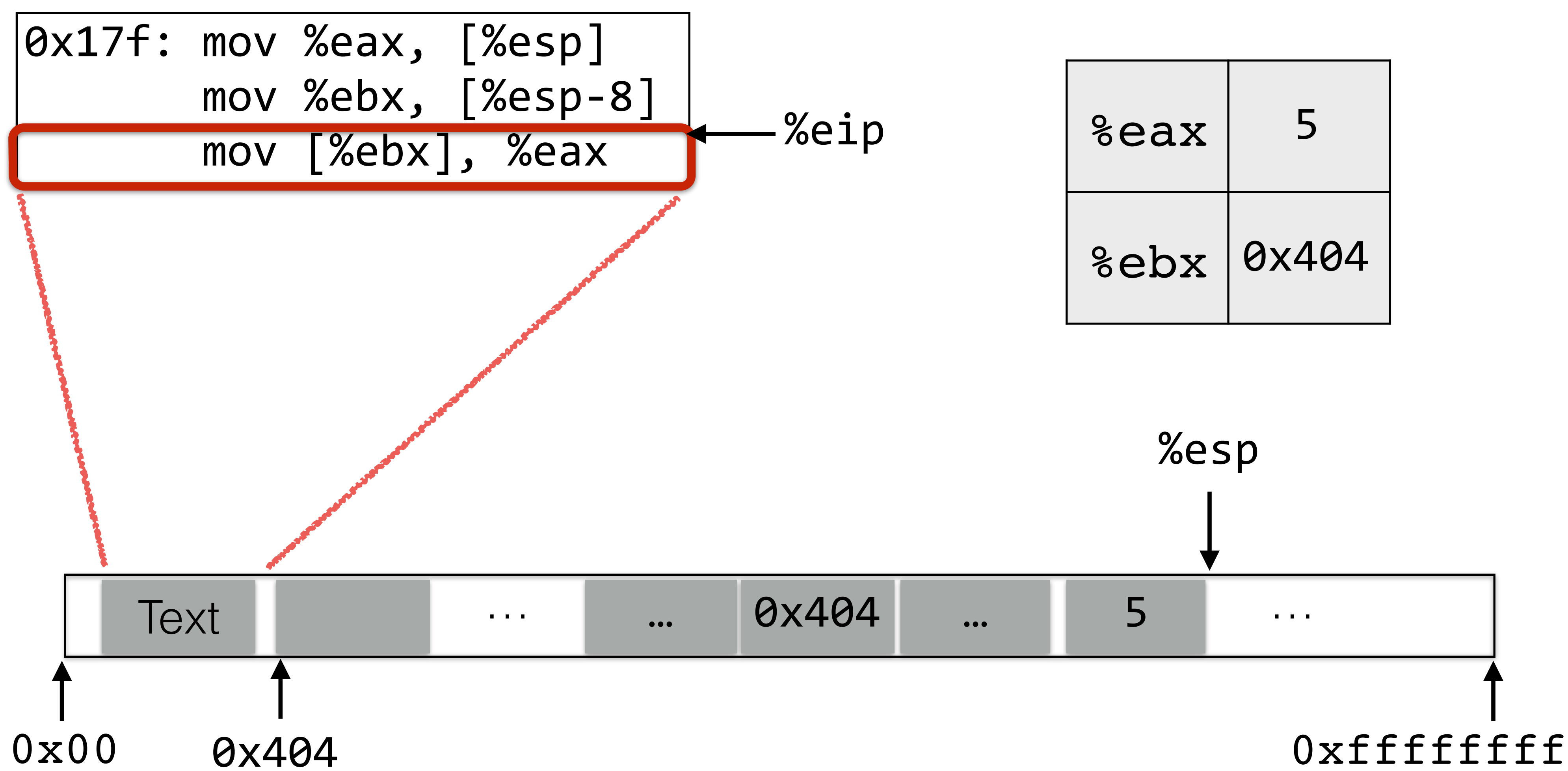
Code sequence



Code sequence

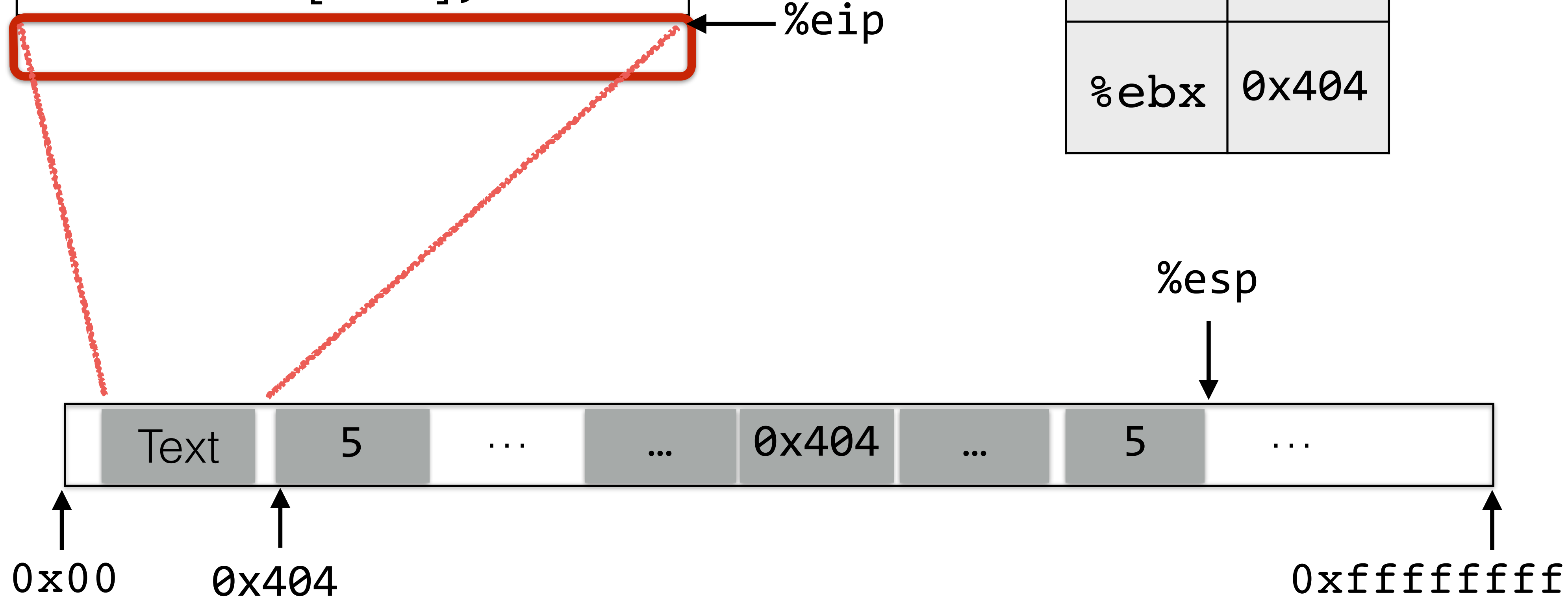


Code sequence

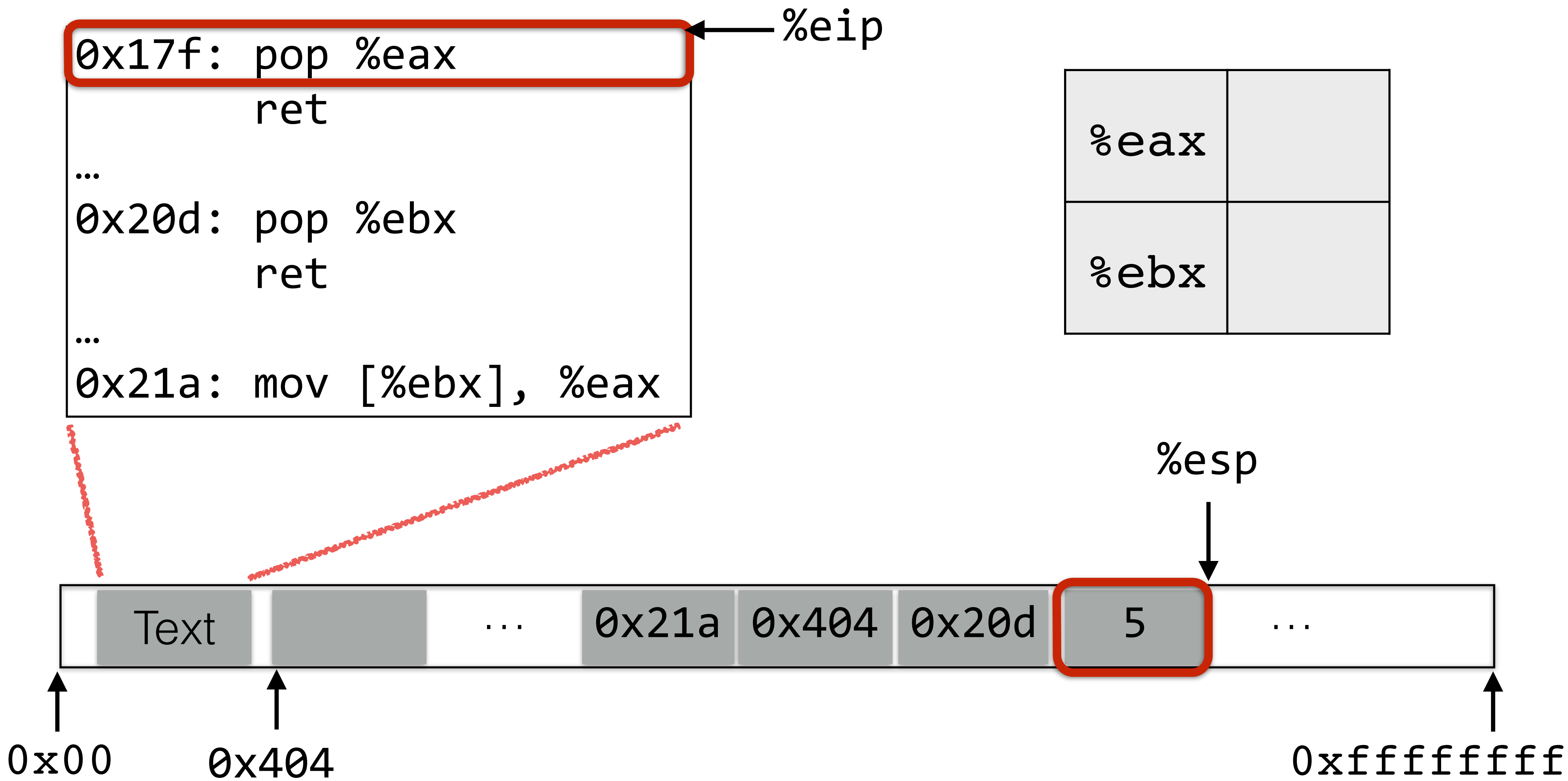


Code sequence

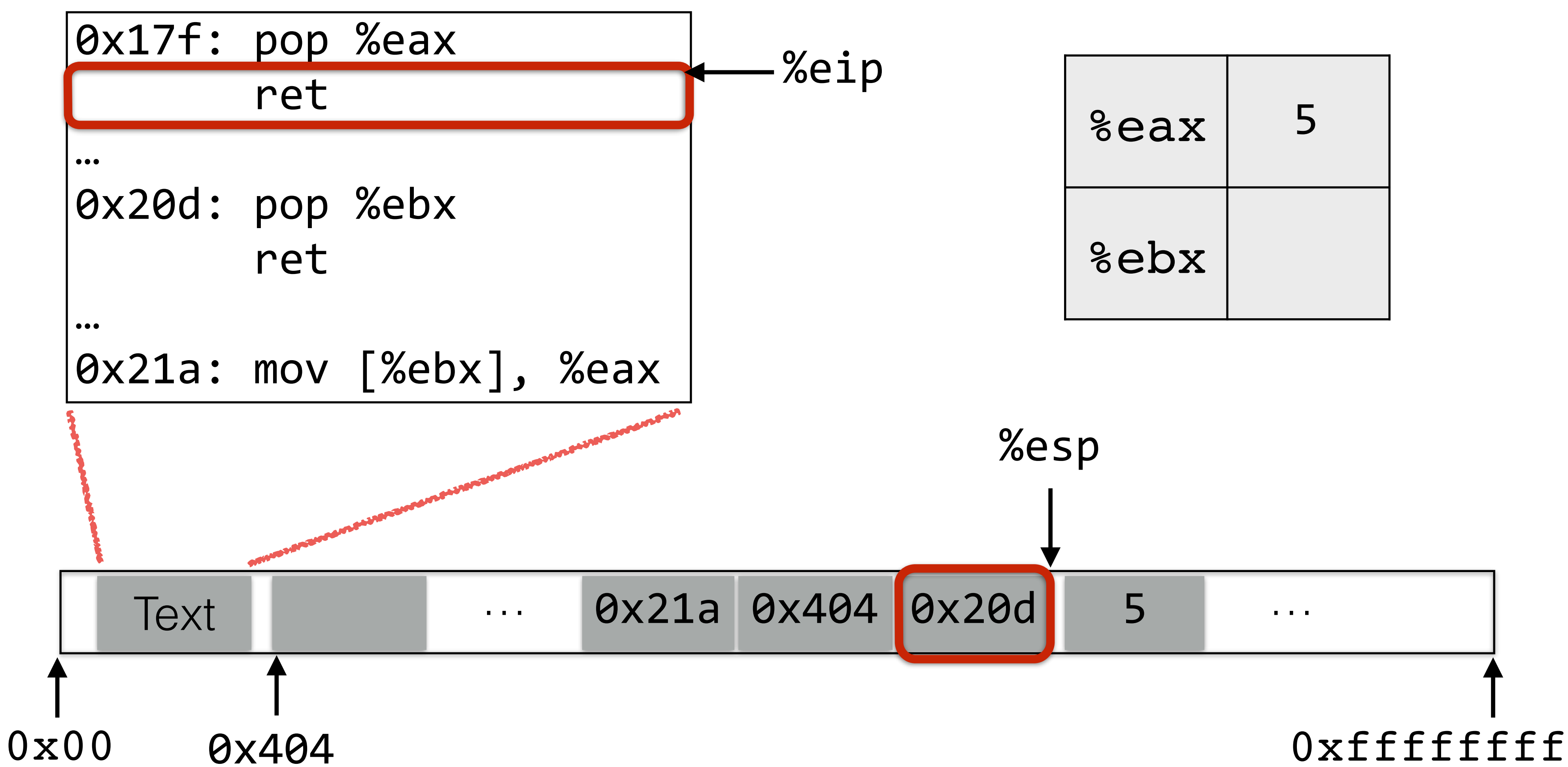
```
0x17f: mov %eax, [%esp]
       mov %ebx, [%esp-8]
       mov [%ebx], %eax
```



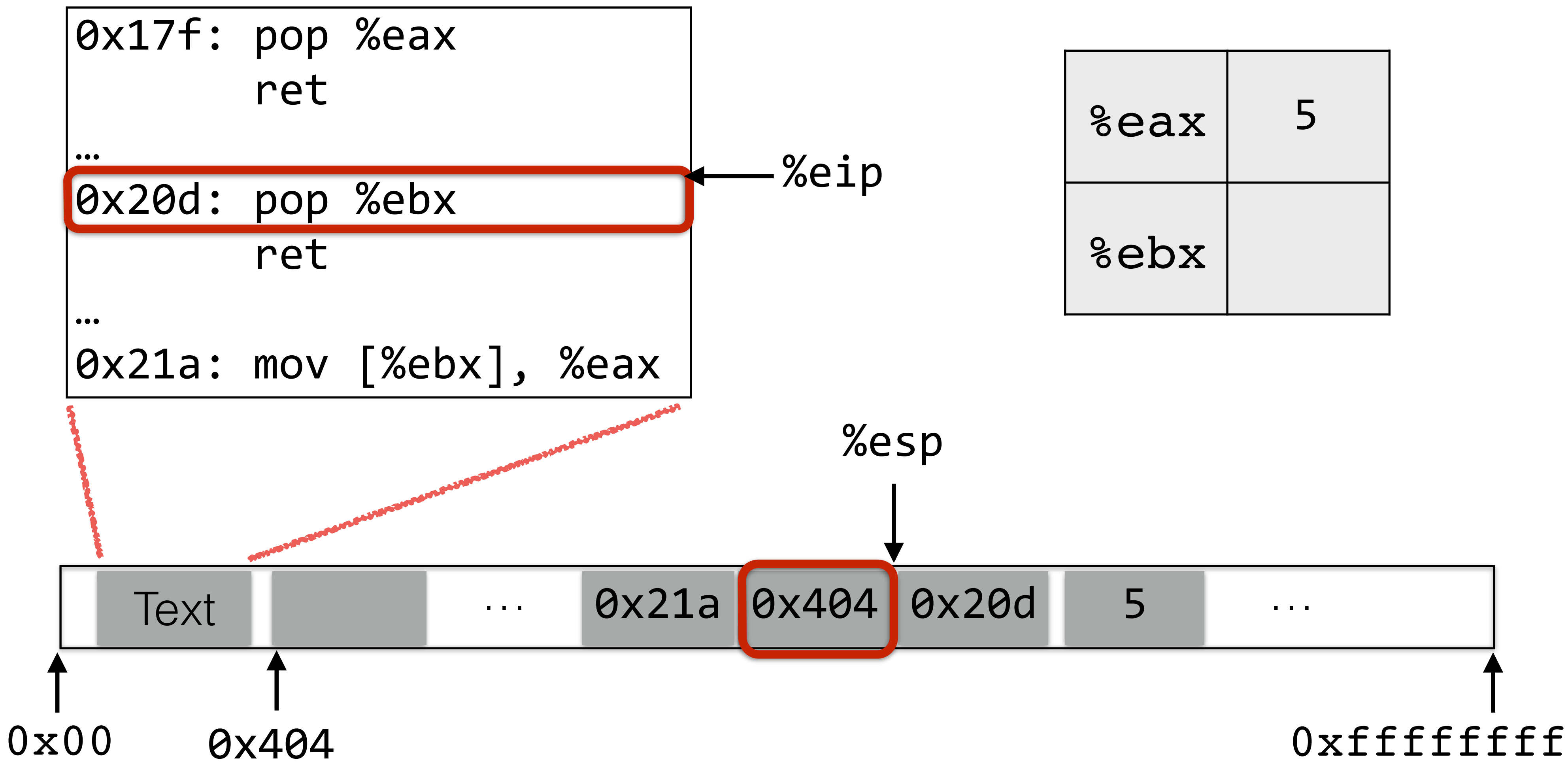
Equivalent ROP sequence



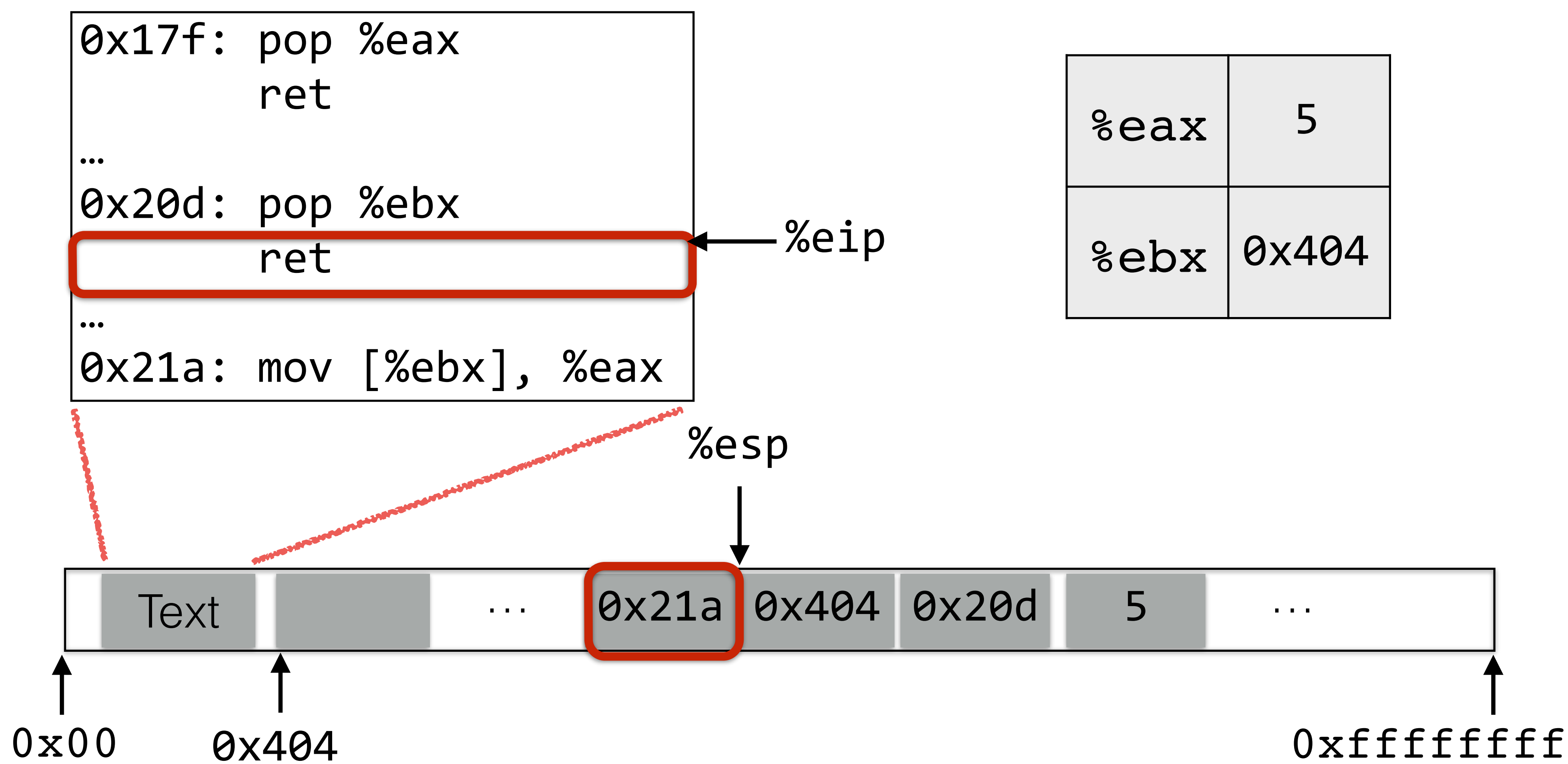
Equivalent ROP sequence



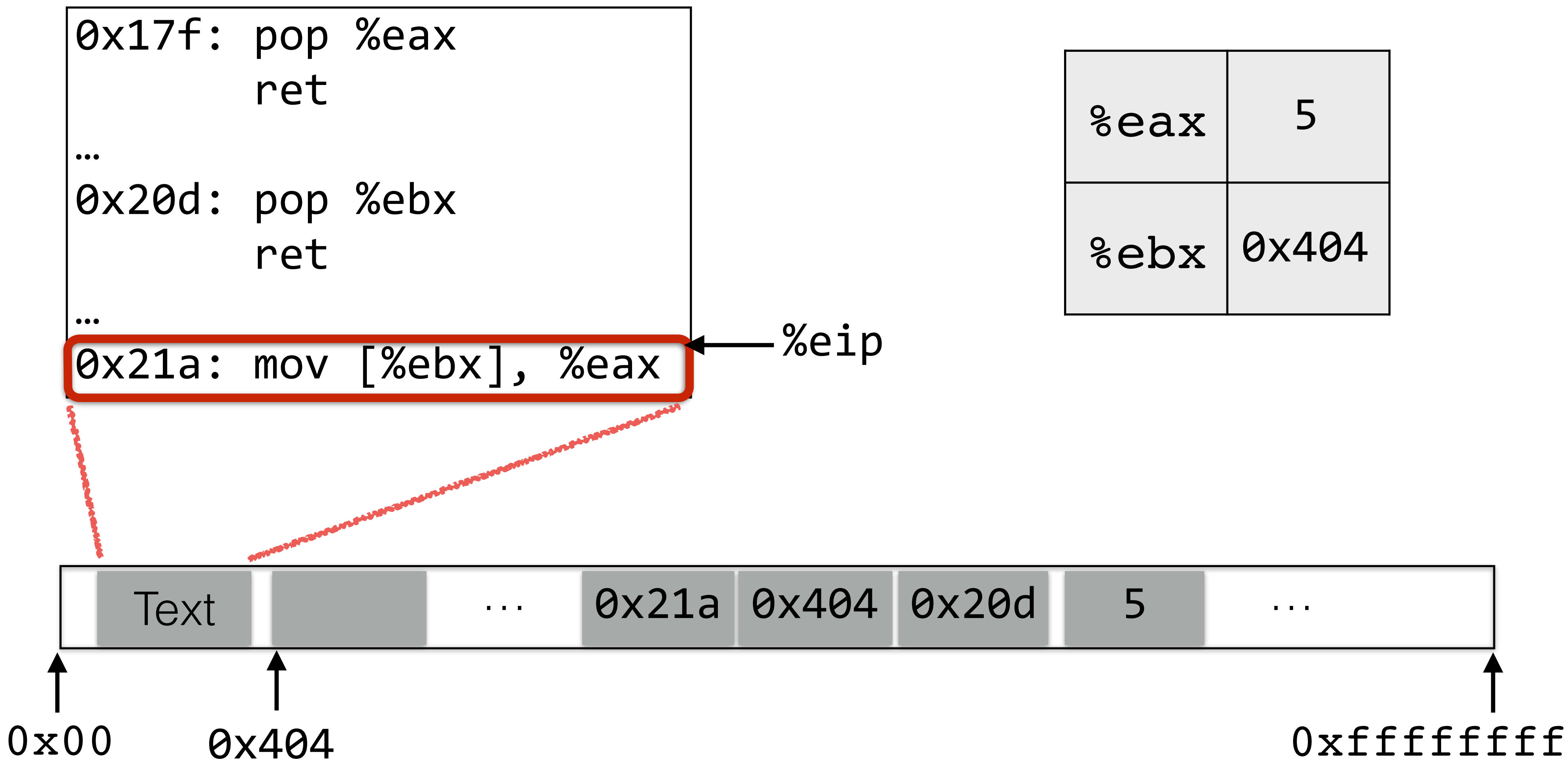
Equivalent ROP sequence



Equivalent ROP sequence



Equivalent ROP sequence



Equivalent ROP sequence

```
0x17f: pop %eax
       ret
…
0x20d: pop %ebx
       ret
…
0x21a: mov [%ebx], %eax
```

%eax	5
%ebx	0x404

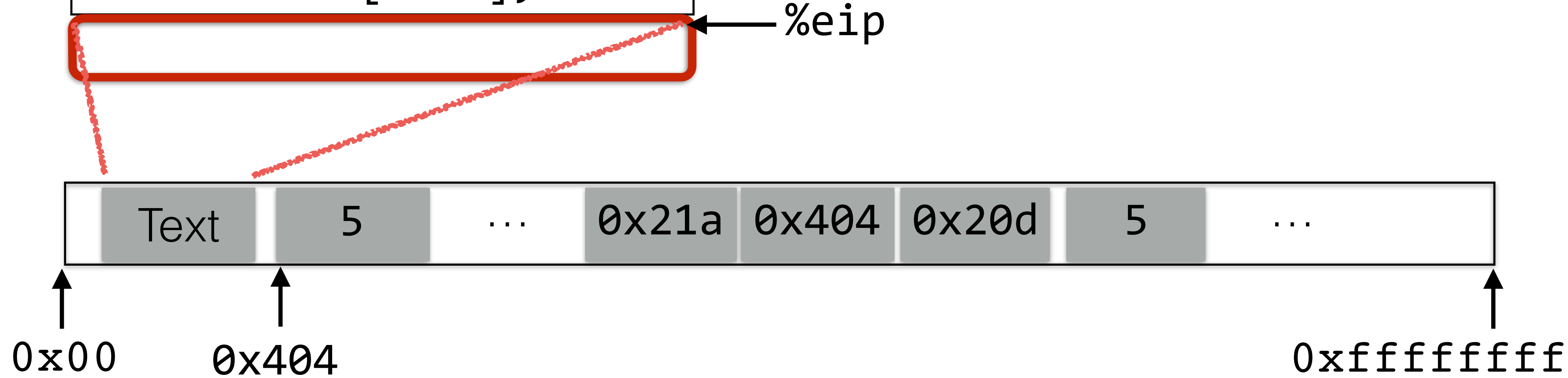


Image by Dino Dai Zovi

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);
}
```

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

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

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

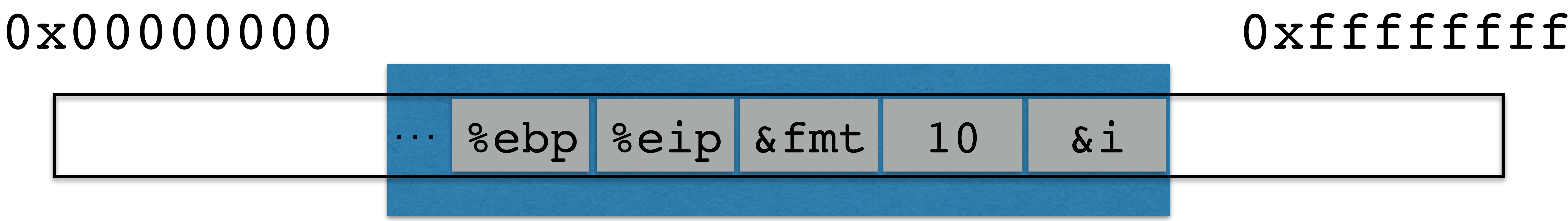
Format string vulnerabilities

`printf` format strings

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

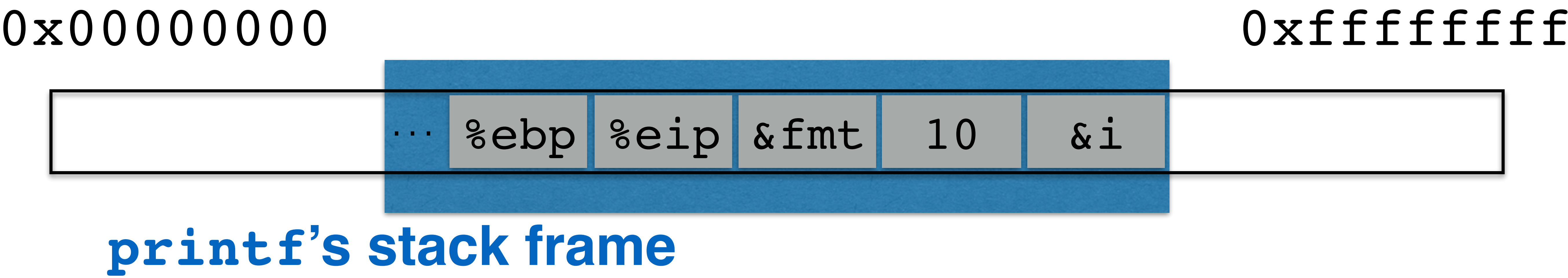
`printf` format strings

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



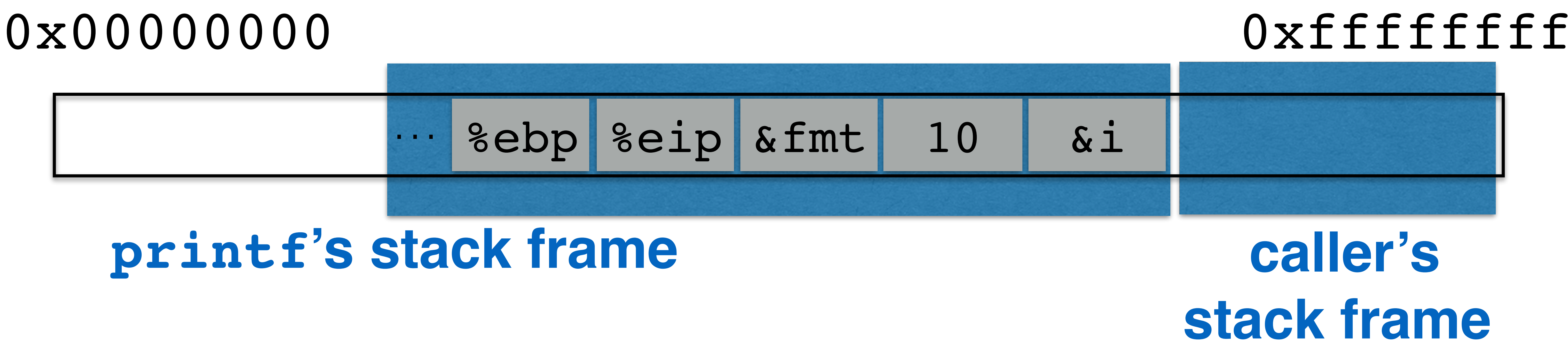
`printf` format strings

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



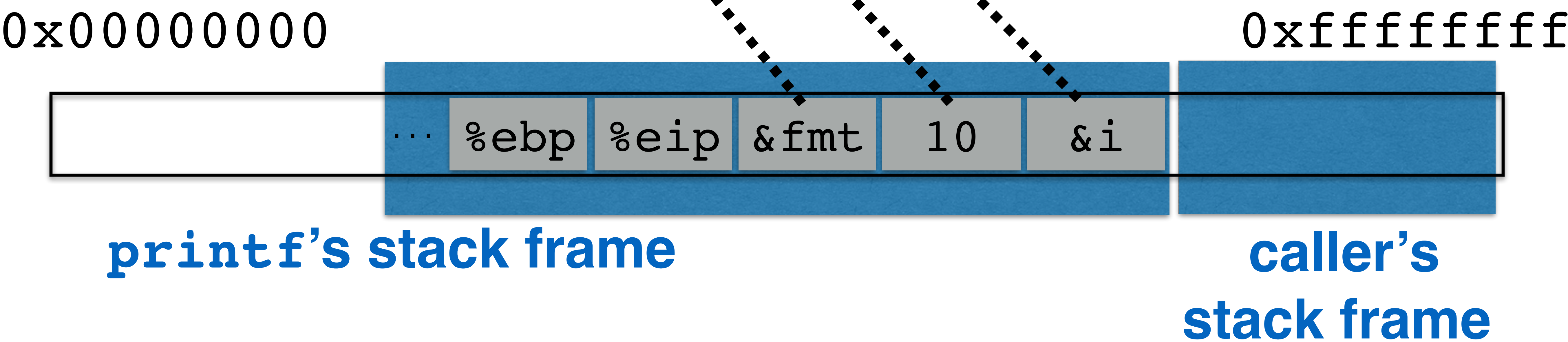
`printf` format strings

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



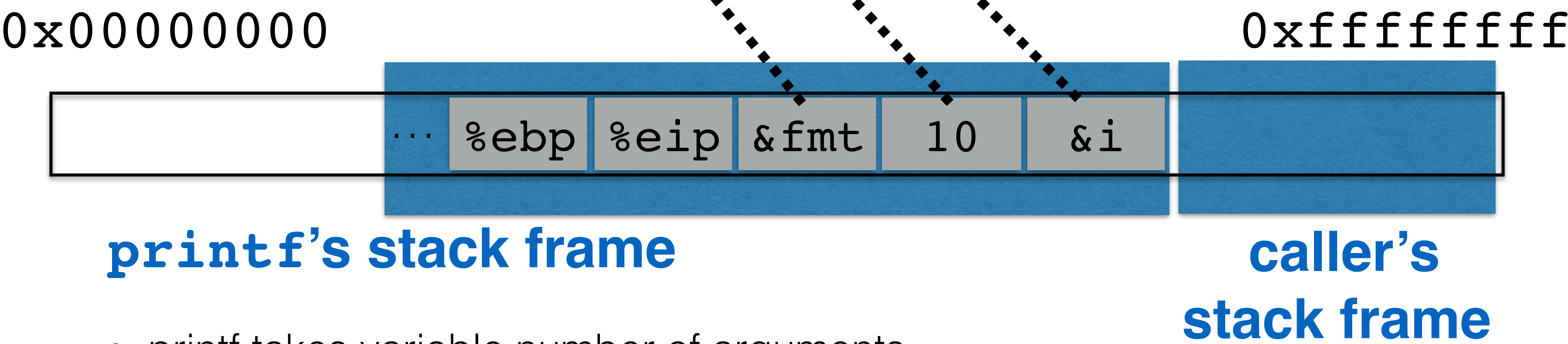
`printf` format strings

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



`printf` format strings

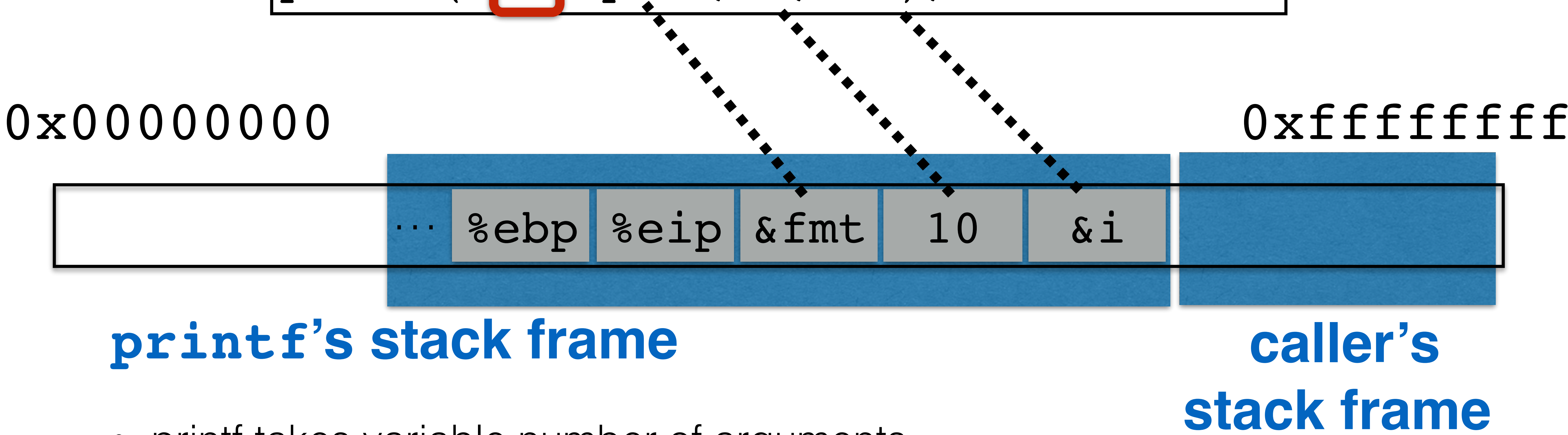
```
int i = 10;
printf("%d %p\n", i, &i);
```



- 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

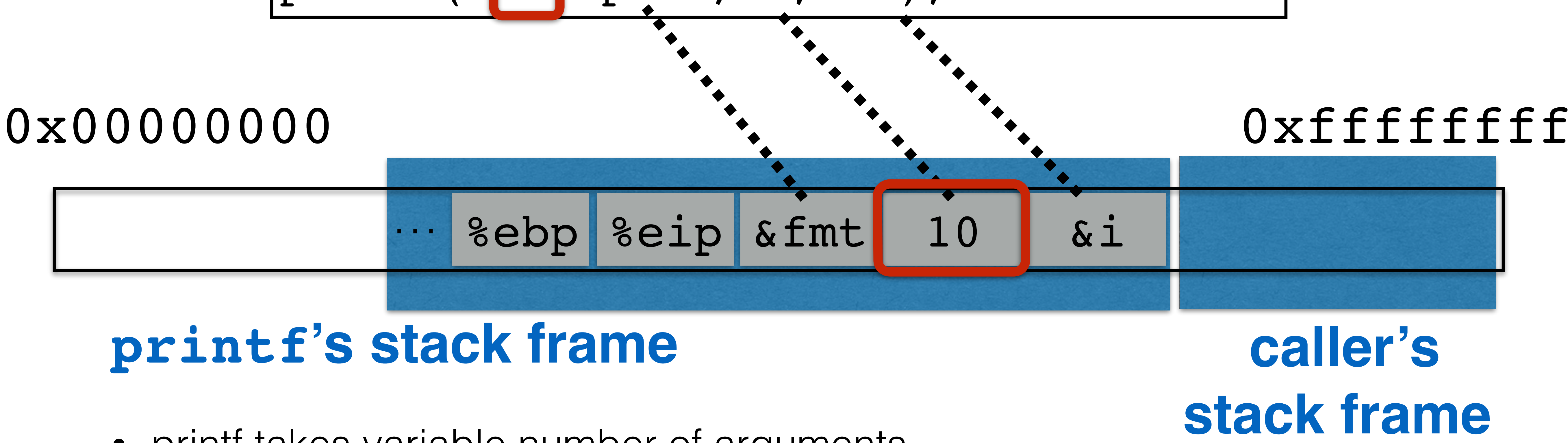
```
int i = 10;
printf("%d %p\n", i, &i);
```



- 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

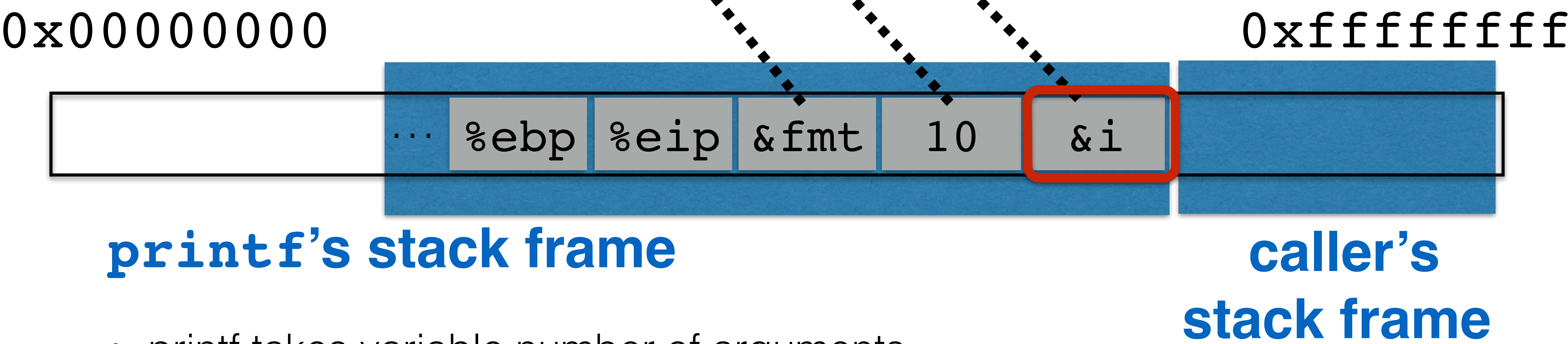
```
int i = 10;
printf("%d %p\n", i, &i);
```



- 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

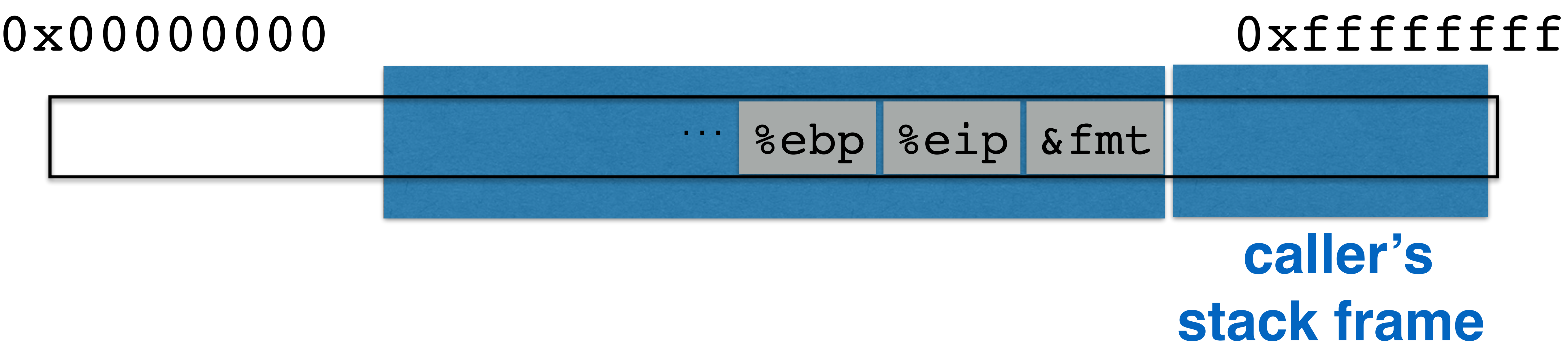
```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

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

“%d %x"

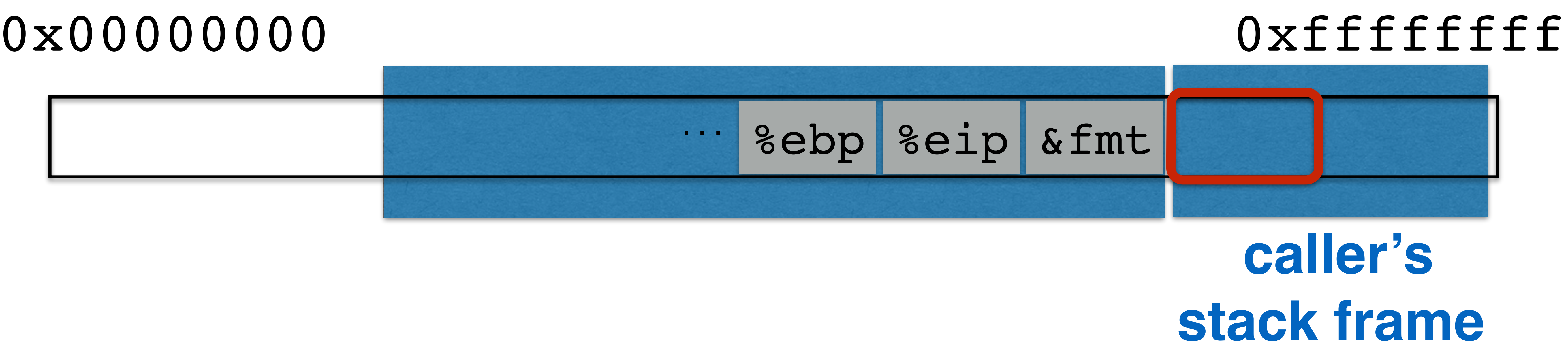
```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

“%d %x"



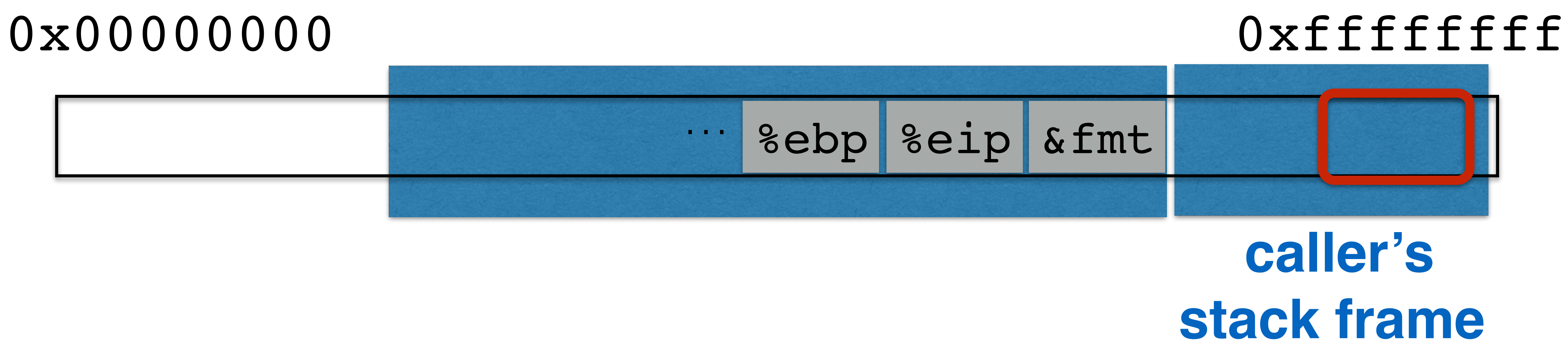
```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

"%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

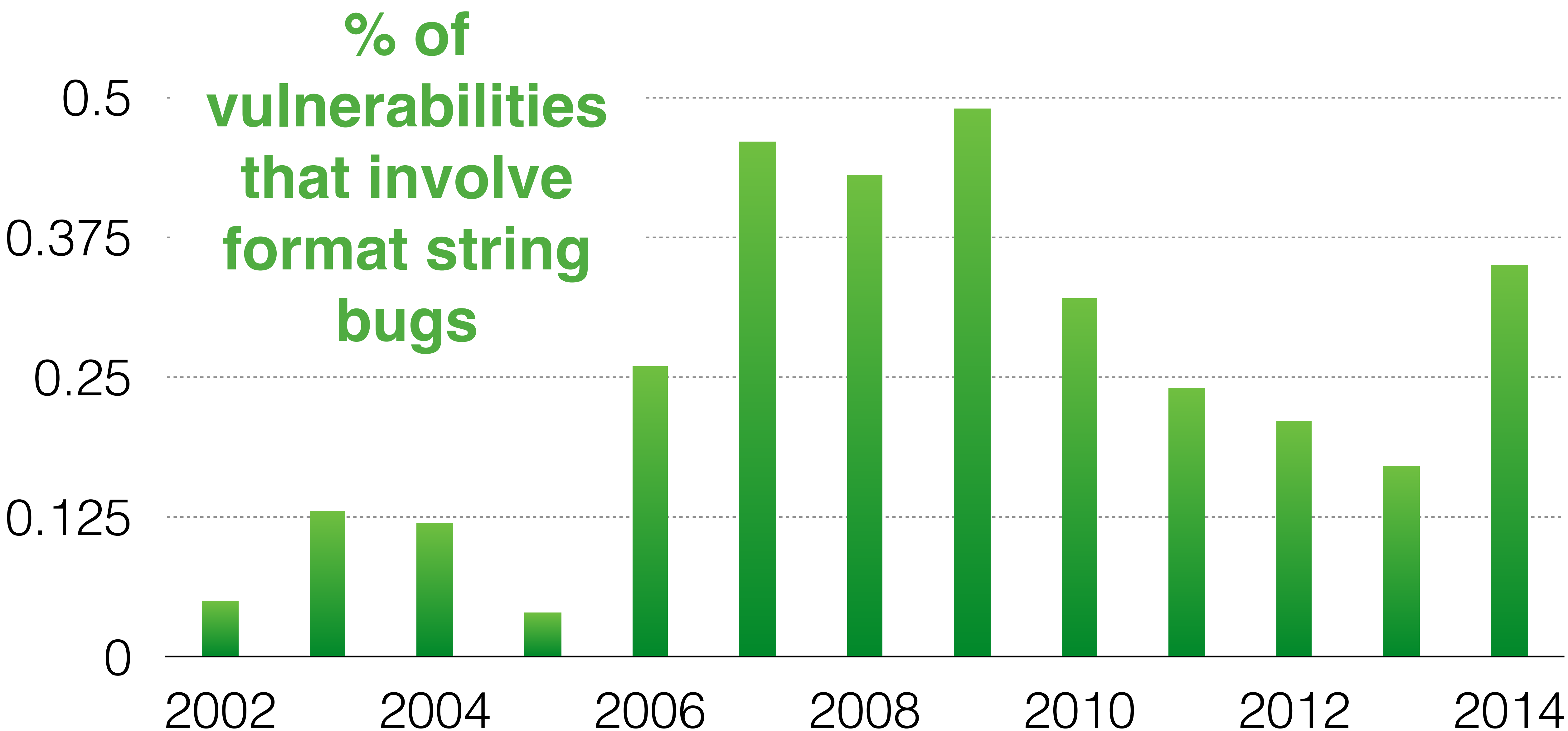
Format string vulnerabilities

- `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()
{   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()
{
    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?

HUGE

```
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?

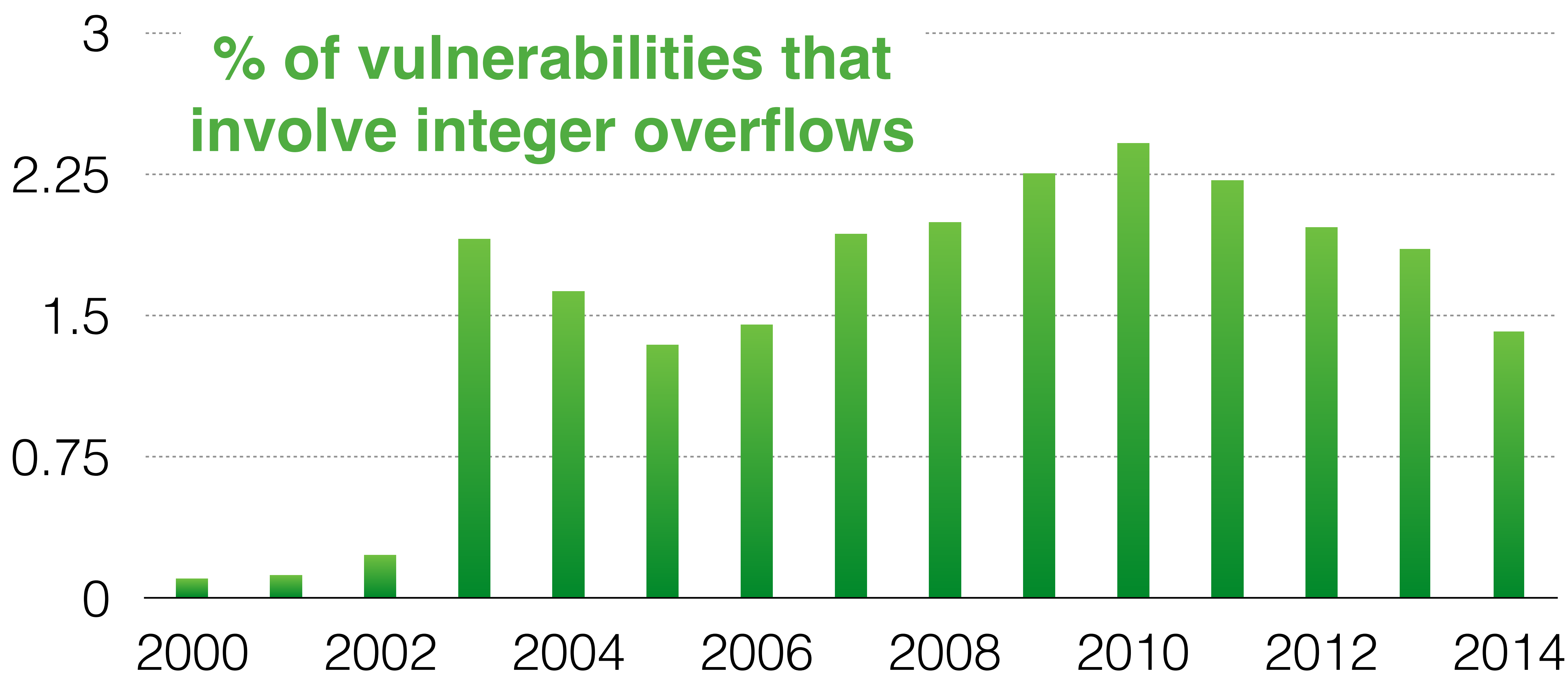
```
void vulnerable()
{
    size_t len;
    char *buf;
  HUGE
    len = read_int_from_network();
    buf = malloc(len + 5); Wrap-around
    read(fd, buf, len);
    ...
}
```

What's wrong with this code?

```
void vulnerable()
{
    size_t len;
    char *buf;
  HUGE
    len = read_int_from_network();
    buf = malloc(len + 5); 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

What’s wrong with this code?

Suppose that it has higher privilege than the user

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf(“cannot access file\n”);
        exit(-1);
    }

    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf(“%s\n”, buf);
    return 0;
}
```

What’s wrong with this code?

Suppose that it has higher privilege than the user

uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }

    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

What's wrong with this code?

Suppose that it has higher privilege than the user

uid

euid

~attacker/mystuff.txt

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }

    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

What's wrong with this code?

Suppose that it has higher privilege than the user

~attacker/mystuff.txt

uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }

    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

What's wrong with this code?

Suppose that it has higher privilege than the user

uid

euid

```
int main() {
    char buf[1024];
    ...                ~attacker/mystuff.txt
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
ln -s /usr/sensitive ~attacker/mystuff.txt
    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

What’s wrong with this code?

Suppose that it has higher privilege than the user

```
int main() {
    char buf[1024];
    ...                ~attacker/mystuff.txt
    if(access(argv[1], R_OK) != 0) {
        printf(“cannot access file\n”);
        exit(-1);
    }
ln -s /usr/sensitive ~attacker/mystuff.txt
    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf(“%s\n”, buf);
    return 0;
}
```

uid

euid

“Time of Check/Time of Use” Problem (TOCTOU)

Avoiding TOCTOU

uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }



    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);

    printf(buf);
}
```

Avoiding TOCTOU

```
     int main() {
         char buf[1024];
         ...
uid      if(access(argv[1], R_OK) != 0) {
             printf("cannot access file\n");
             exit(-1);
         }



euid     file = open(argv[1], O_RDONLY);
         read(file, buf, 1023);
         close(file);

         printf(buf);
     }
```

Avoiding TOCTOU

uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
    euid = geteuid();
    uid = getuid();
    seteuid(uid);   // Drop privileges
    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);

    printf(buf);
}
```

Avoiding TOCTOU

uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
    euid = geteuid();
    uid = getuid();
    seteuid(uid);   // Drop privileges
    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    seteuid(euid);  // Restore privileges
    printf(buf);
}
```

Memory safety attacks

- Buffer overflows
  - Can be used to read/write data on stack or heap
  - Can be used to inject code (ultimately root shell)
- Format string errors
  - Can be used to read/write stack data
- Integer overflow errors
  - Can be used to change the control flow of a program
- TOCTOU problem
  - Can be used to raise privileges

General defenses against memory-safety

Defensive coding practices

- Think defensive driving
  - Avoid depending on anyone else around you
  - If someone does something unexpected, you won't crash (or worse)
  - It's about *minimizing trust*

- Each module takes responsibility for checking the validity of all inputs sent to it
  - Even if you "know" your callers will never send a NULL pointer…
  - …Better to throw an exception (or even exit) than run malicious code

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

How to program defensively

- Code reviews, real or imagined
  - Organize your code so it is obviously correct
  - Re-write until it would be self-evident to a reviewer

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

- Remove the opportunity for programmer mistakes with better languages and libraries
  - Java performs automatic bounds checking
  - C++ provides a safe `std::string` class

Secure coding practices

```
char digit_to_char(int i) {
    char convert[] = “0123456789”;
    return convert[i];
}
```

Think about all potential inputs, no matter how peculiar

Secure coding practices

```
char digit_to_char(int i) {
    char convert[] = “0123456789”;
    return convert[i];
}
```

Think about all potential inputs, no matter how peculiar

```
char digit_to_char(int i) {
    char convert[] = “0123456789”;
    if(i < 0 || i > 9)
        return ‘?’;
    return convert[i];
}
```

Secure coding practices

```
char digit_to_char(int i) {
    char convert[] = "0123456789";
    return convert[i];
}
```

Think about all potential inputs, no matter how peculiar

```
char digit_to_char(int i) {
    char convert[] = "0123456789";
    if(i < 0 || i > 9)
        return '?';
    return convert[i];
}
```

Enforce rule compliance at runtime

Rule: Use safe string functions

- Traditional string library routines assume target buffers have sufficient length

Rule: Use safe string functions

- Traditional string library routines assume target buffers have sufficient length

```
char str[4];
char buf[10] = “good”;
strcpy(str,”hello”); // overflows str
strcat(buf,” day to you”); // overflows buf
```

Rule: Use safe string functions

- Traditional string library routines assume target buffers have sufficient length

```
char str[4];
char buf[10] = “good”;
strcpy(str,”hello”); // overflows str
strcat(buf,” day to you”); // overflows buf
```

- Safe versions check the destination length

```
char str[4];
char buf[10] = “good”;
strlcpy(str,”hello”,sizeof(str)); //fails
strlcat(buf,” day to you”,sizeof(buf));//fails
```

Rule: Use safe string functions

- Traditional string library routines assume target buffers have sufficient length

```
char str[4];
char buf[10] = “good”;
strcpy(str,”hello”); // overflows str
strcat(buf,” day to you”); // overflows buf
```

- Safe versions check the destination length

```
char str[4];
char buf[10] = “good”;
strlcpy(str,”hello”,sizeof(str)); //fails
strlcat(buf,” day to you”,sizeof(buf));//fails
```

Again: new your system’s/language’s semantics

Replacements

Replacements

- … for string-oriented functions
  - `strcat` ⇒ `strlcat`
  - `strcpy` ⇒ `strlcpy`
  - `strncat` ⇒ `strlcat`
  - `strncpy` ⇒ `strlcpy`
  - `sprintf` ⇒ `snprintf`
  - `vsprintf` ⇒ `vsnprintf`
  - `gets` ⇒ `fgets`

Replacements

- … for string-oriented functions
  - `strcat` ⇒ `strlcat`
  - `strcpy` ⇒ `strlcpy`
  - `strncat` ⇒ `strlcat`
  - `strncpy` ⇒ `strlcpy`
  - `sprintf` ⇒ `snprintf`
  - `vsprintf` ⇒ `vsnprintf`
  - `gets` ⇒ `fgets`

- Microsoft versions different
  - `strcpy_s`, `strcat_s`, `...`

Replacements

- … for string-oriented functions
  - `strcat` ⇒ `strlcat`
  - `strcpy` ⇒ `strlcpy`
  - `strncat` ⇒ `strlcat`
  - `strncpy` ⇒ `strlcpy`
  - `sprintf` ⇒ `snprintf`
  - `vsprintf` ⇒ `vsnprintf`
  - `gets` ⇒ `fgets`
- Microsoft versions different
  - `strcpy_s`, `strcat_s`, …

Note: None of these in and of themselves are "insecure." They are just commonly misused.

(Better) **Rule**: Use safe string library

- Libraries designed to ensure strings used safely
  - **Safety first**, despite some performance loss

- Example: Very Secure FTP (**vsftp**) **string library**

http://vsftpd.beasts.org/

```
struct mystr; // impl hidden

void str_alloc_text(struct mystr* p_str,
                    const char* p_src);
void str_append_str(struct mystr* p_str,
                    const struct mystr* p_other);
int str_equal(const struct mystr* p_str1,
              const struct mystr* p_str2);
int str_contains_space(const struct mystr* p_str);
…
```

- Another example: **C++** **`std::string`** safe string library

Rule: Understand pointer arithmetic

Rule: Understand pointer arithmetic

```
int x;
int *pi = &x;
char *pc = (char*) &x;
```

Rule: Understand pointer arithmetic

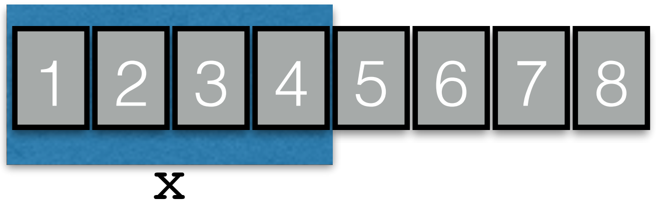
```
int x;
int *pi = &x;
char *pc = (char*) &x;
```

`(pi + 1) == (pc + 1)` ???

Rule: Understand pointer arithmetic

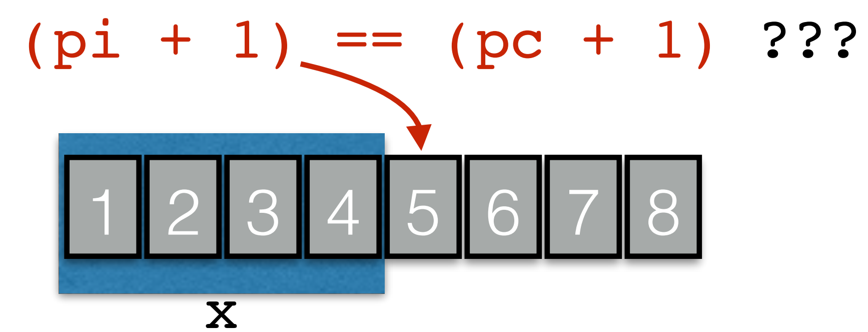
```
int x;
int *pi = &x;
char *pc = (char*) &x;
```

```
(pi + 1) == (pc + 1) ???
```



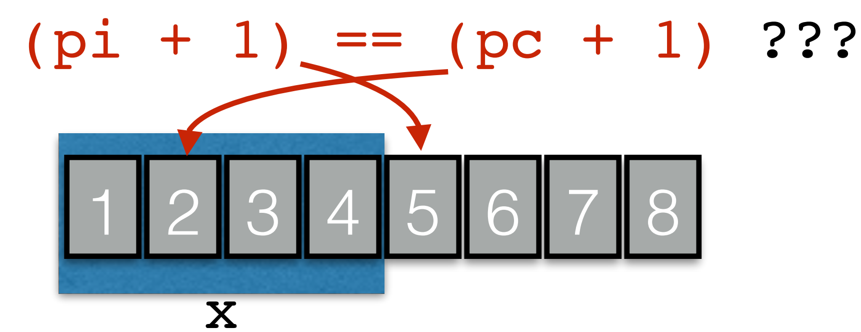
Rule: Understand pointer arithmetic

```
int x;
int *pi = &x;
char *pc = (char*) &x;
```



Rule: Understand pointer arithmetic

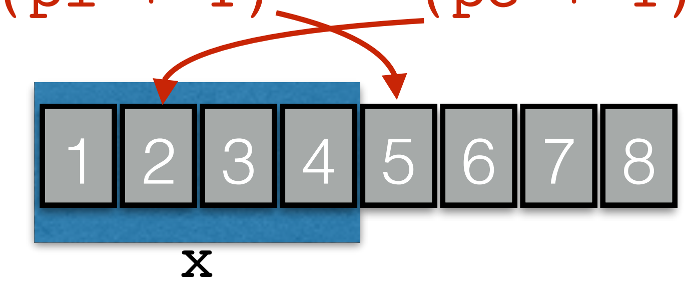
```
int x;
int *pi = &x;
char *pc = (char*) &x;
```



Rule: Understand pointer arithmetic

```
int x;
int *pi = &x;
char *pc = (char*) &x;
```

```
(pi + 1) == (pc + 1) ???
```

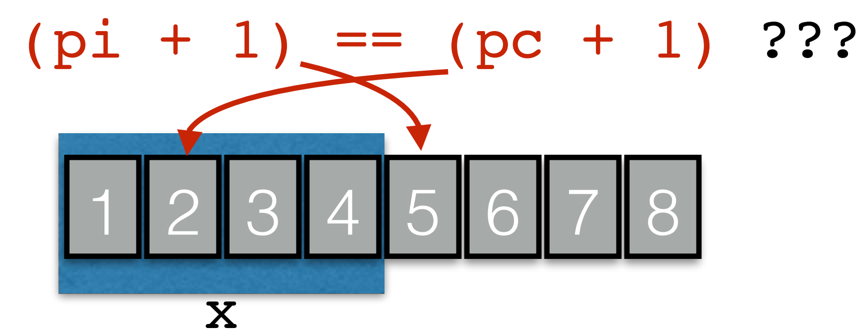


- `sizeof()` returns number of *bytes*, but pointer arithmetic multiplies by the `sizeof` the type

```
int buf[SIZE] = { … };
int *buf_ptr = buf;

while (!done() && buf_ptr < (buf + sizeof(buf))) {
  *buf_ptr++ = getnext(); // will overflow
}
```

Rule: Understand pointer arithmetic

```
int x;
int *pi = &x;
char *pc = (char*) &x;
```



- `sizeof()` returns number of *bytes*, but pointer arithmetic multiplies by the `sizeof` the type

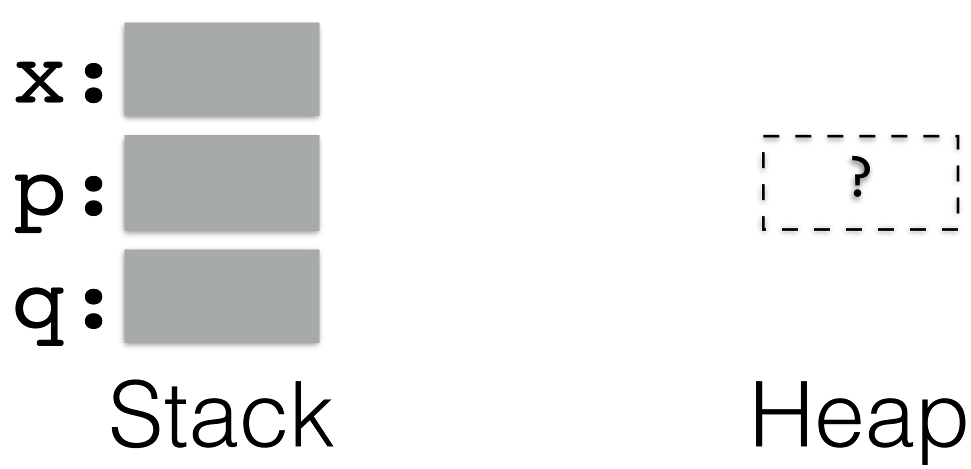
```
int buf[SIZE] = { … };
int *buf_ptr = buf;

while (!done() && buf_ptr < (buf + sizeof(buf))) {
  *buf_ptr++ = getnext(); // will overflow
}
```

- So, **use the right units**

```
while (!done() && buf_ptr < (buf + SIZE)) {
  *buf_ptr++ = getnext(); // stays in bounds
}
```

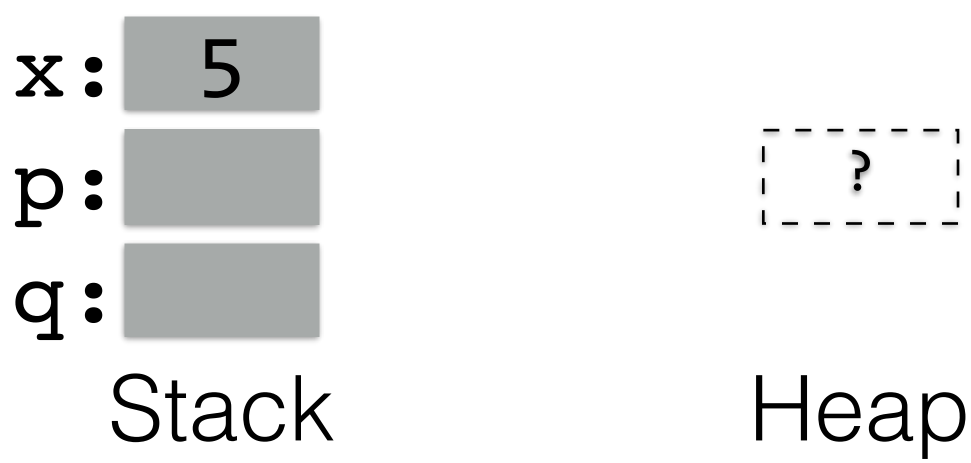
Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5;
**q = 3; //crash (or worse)!
```



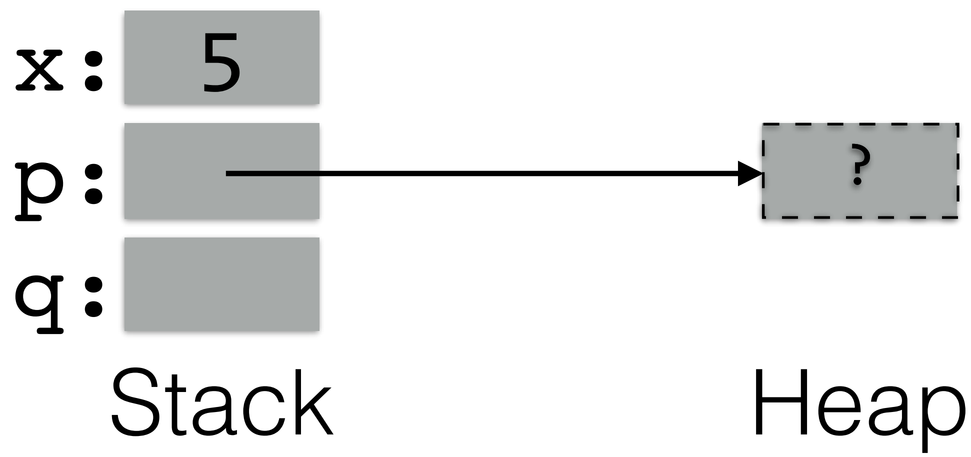
Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5;
**q = 3; //crash (or worse)!
```



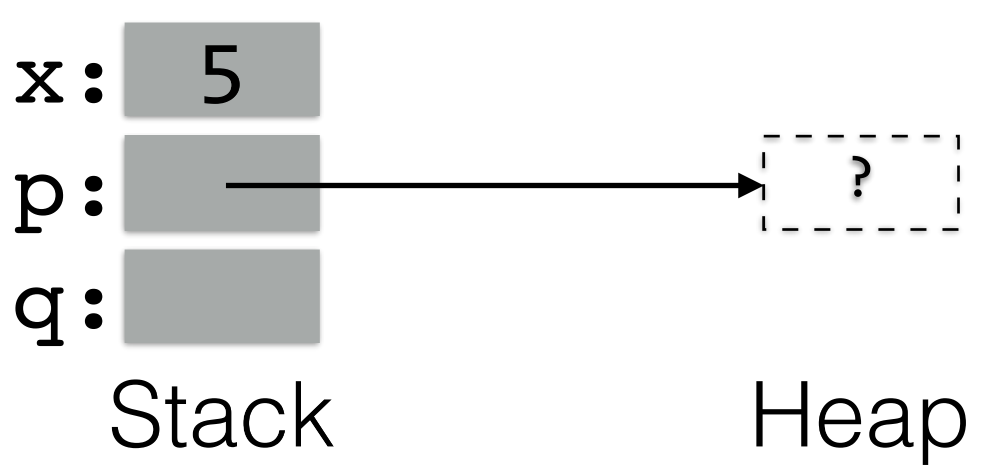
Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5;
**q = 3; //crash (or worse)!
```



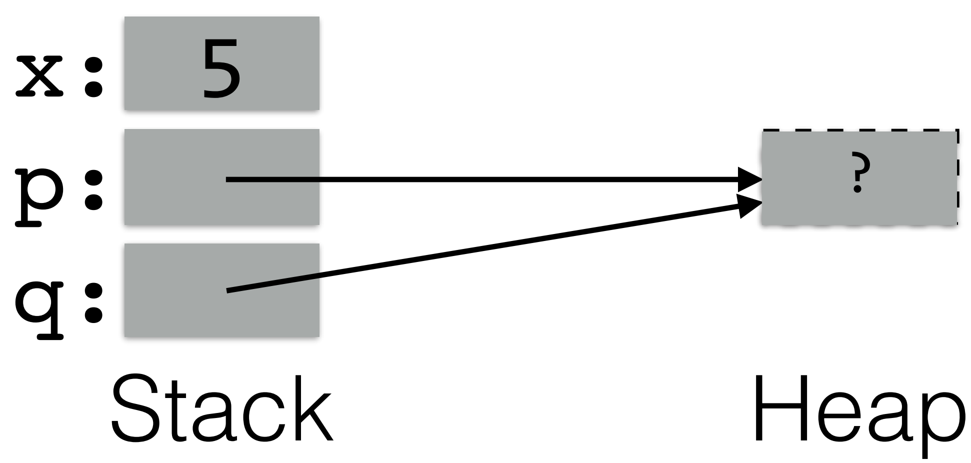
Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5;
**q = 3; //crash (or worse)!
```



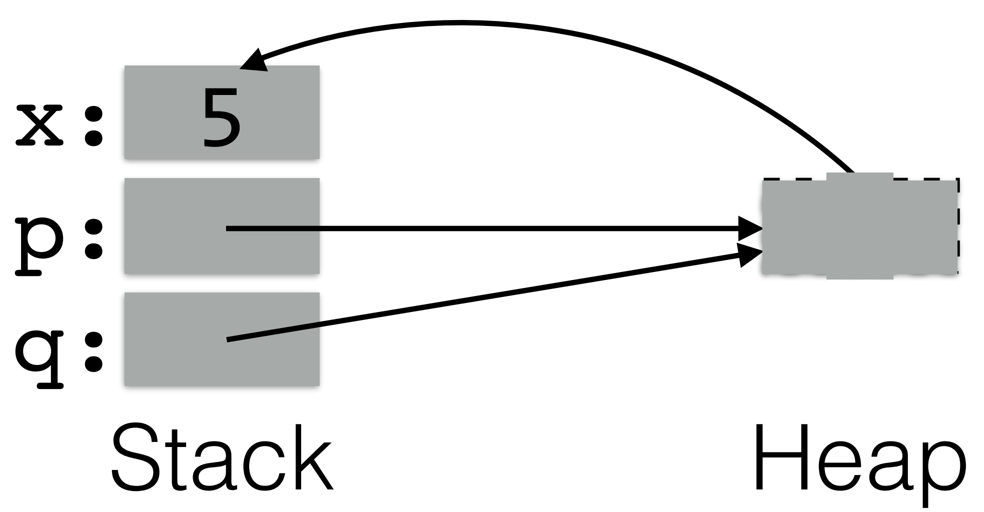
Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5;
**q = 3; //crash (or worse)!
```



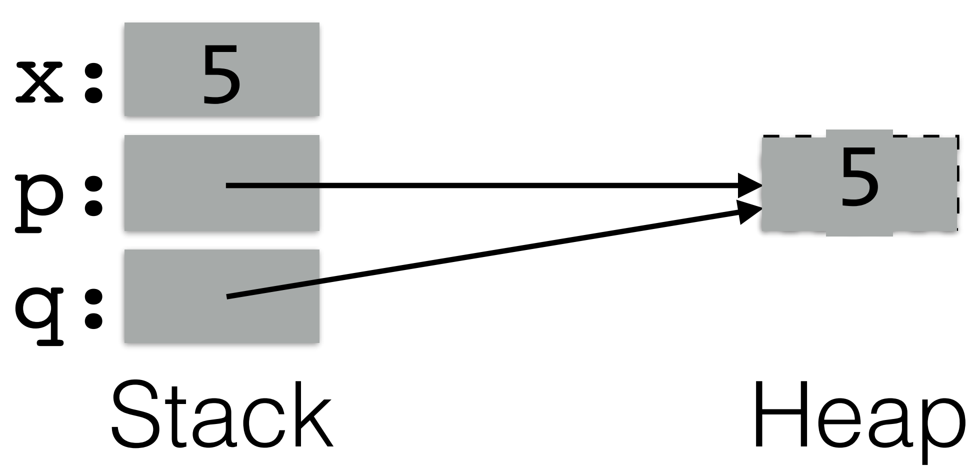
Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5;
**q = 3; //crash (or worse)!
```



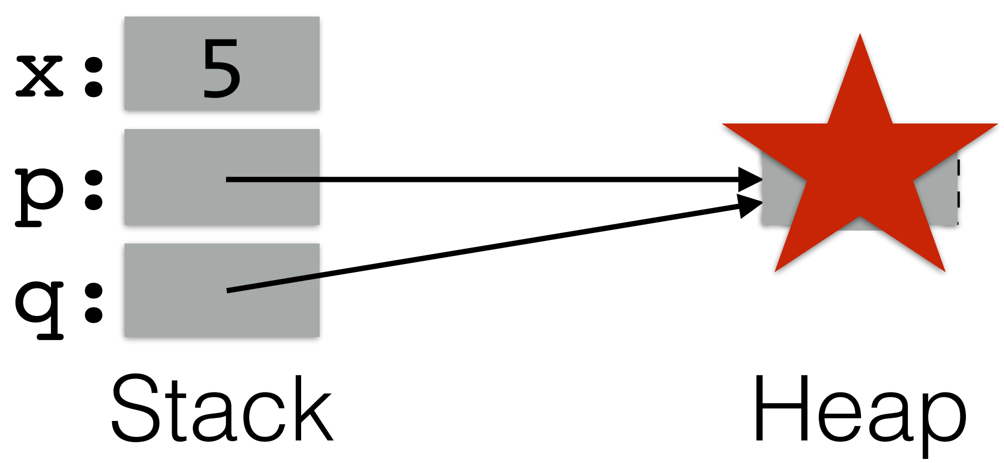
Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5;
**q = 3; //crash (or worse)!
```



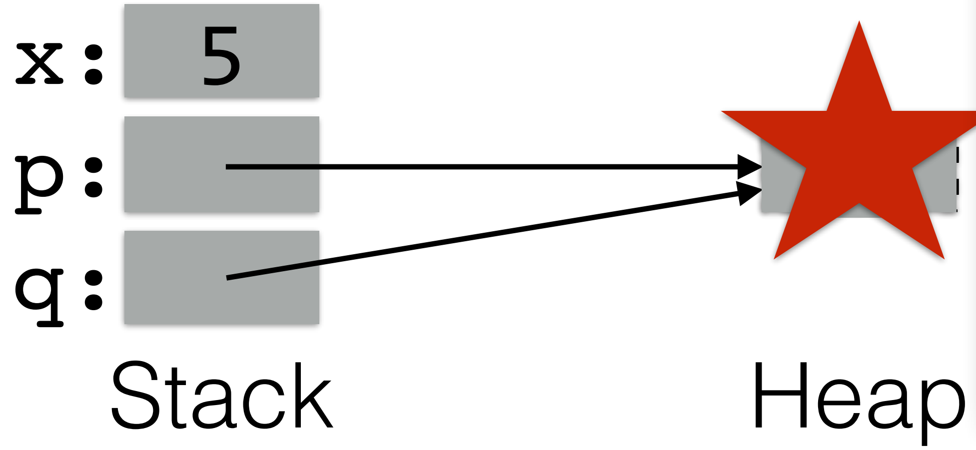
Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5;
**q = 3; //crash (or worse)!
```



Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5;
**q = 3; //crash (or worse)!
```



IE's Role in the Google-China War

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

Text Size
Print Version
E-Mail Article

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.

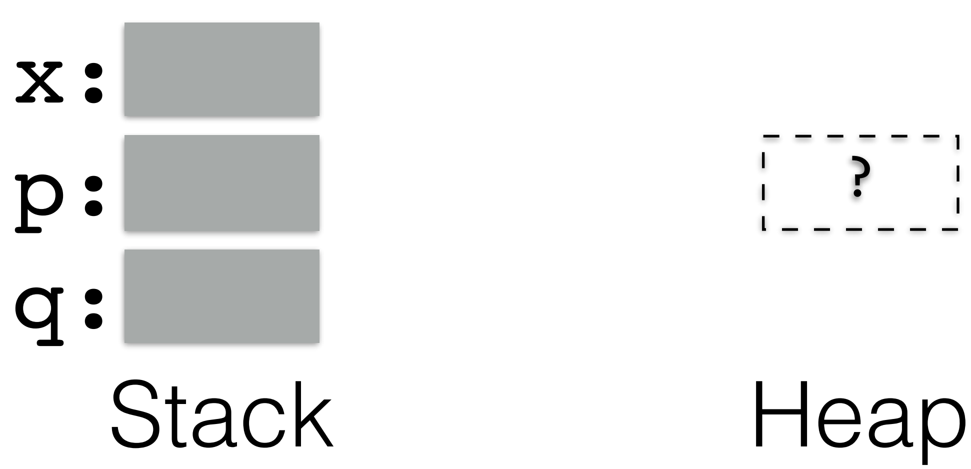
The attacks on Google, which appeared to originate in China, have sparked a feud between the Internet giant and the nation's government over censorship, and it could result in Google pulling away from its business dealings in the country.

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.

Rule: Use NULL after free

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; //defend against bad deref
int **q = malloc(sizeof(int*)); //reuses p’s space
*q = &x;
*p = 5; //(good) crash
**q = 3;
```

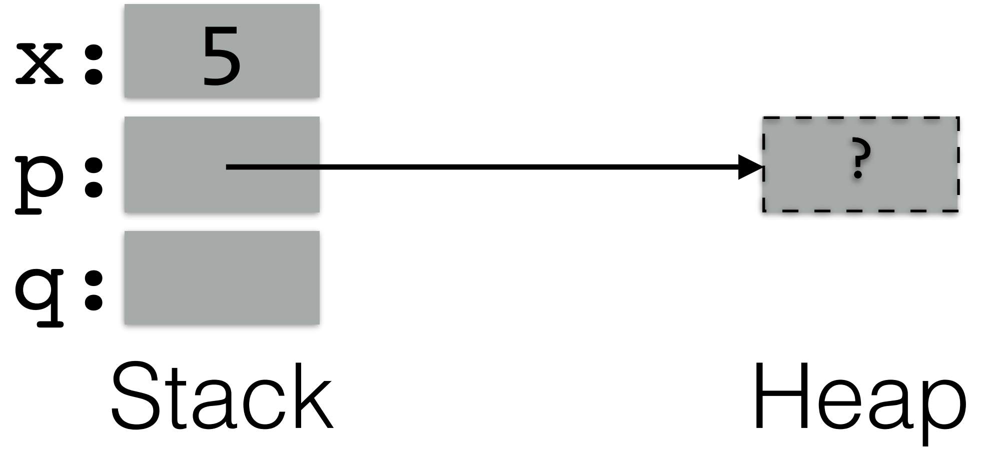


Rule: Use NULL after free

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; //defend against bad deref
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5; //(good) crash
**q = 3;
```

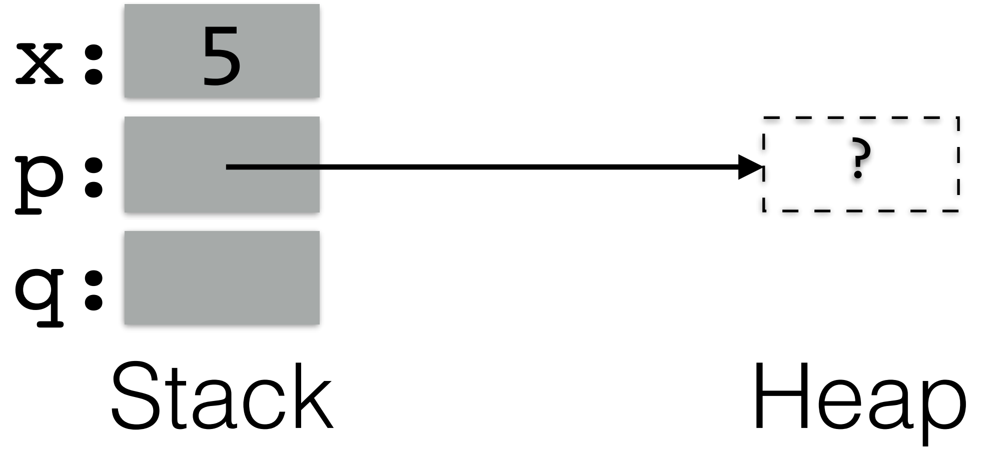
Rule: Use NULL after free

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; //defend against bad deref
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5; //(good) crash
**q = 3;
```



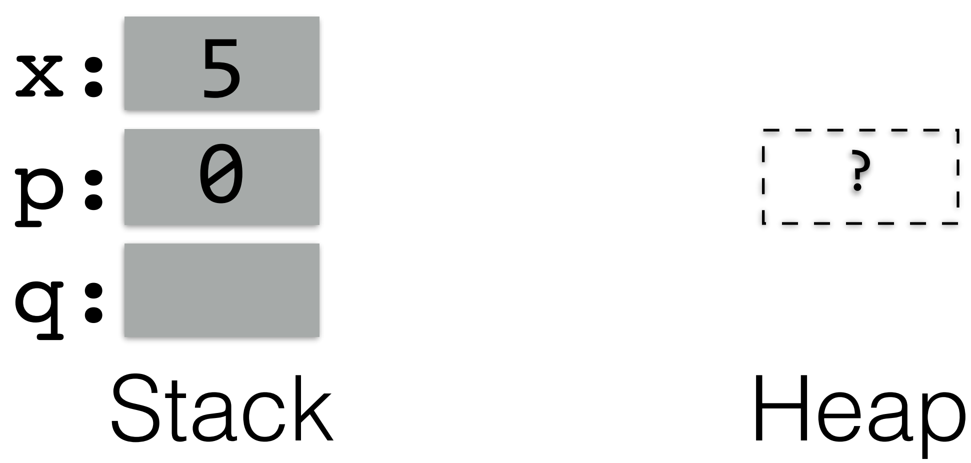
Rule: Use NULL after free

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; //defend against bad deref
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5; //(good) crash
**q = 3;
```



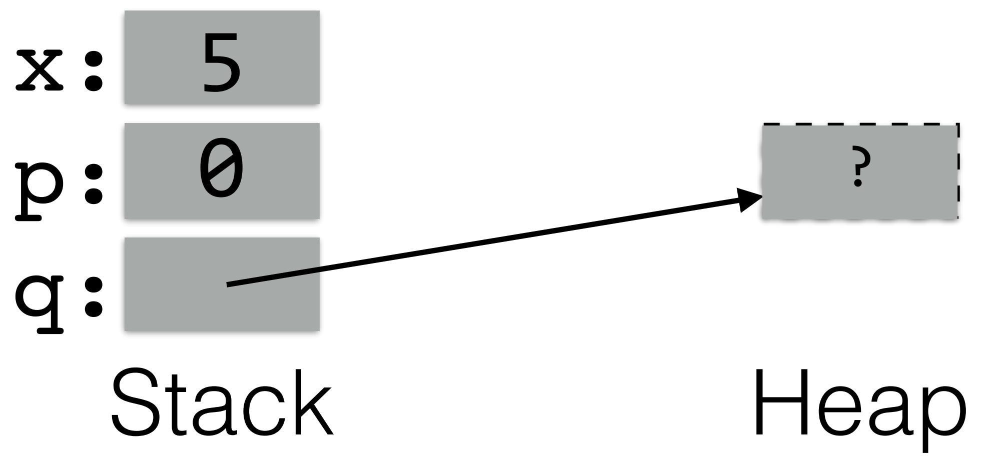
Rule: Use NULL after free

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; //defend against bad deref
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5; //(good) crash
**q = 3;
```



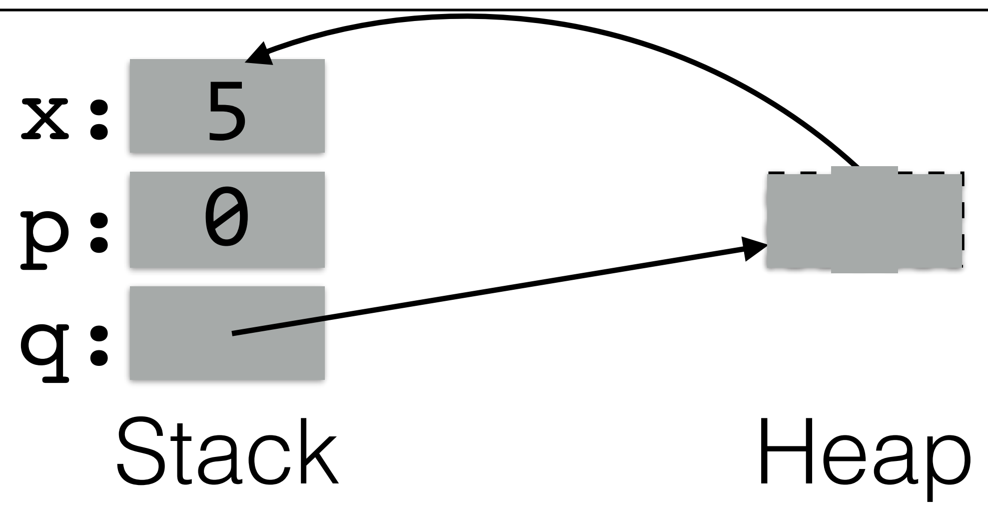
Rule: Use NULL after free

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; //defend against bad deref
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5; //(good) crash
**q = 3;
```



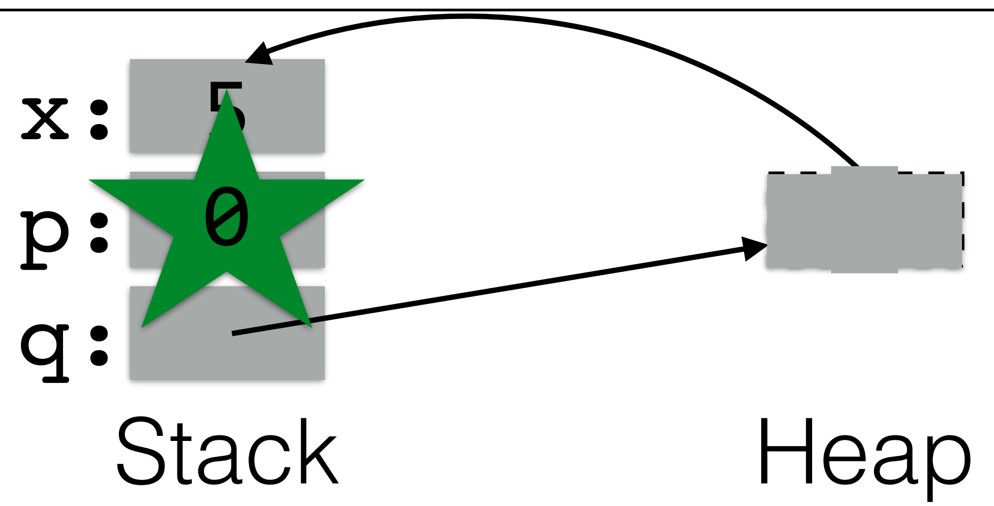
Rule: Use NULL after free

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; //defend against bad deref
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5; //(good) crash
**q = 3;
```



Rule: Use NULL after free

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; //defend against bad deref
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5; //(good) crash
**q = 3;
```



Manage memory properly

```
int foo(int arg1, int arg2) {
  struct foo *pf1, *pf2;
  int retc = -1;

  pf1 = malloc(sizeof(struct foo));
  if (!isok(arg1)) goto DONE;
  …
  pf2 = malloc(sizeof(struct foo));
  if (!isok(arg2)) goto FAIL_ARG2;
  …
  retc = 0;

 FAIL_ARG2:
  free(pf2); //fallthru
 DONE:
  free(pf1);
  return retc;
}
```

- Common approach in C: *goto chains* to avoid duplicated or missed code
  - Like try/finally in languages like Java
- Confirm your logic!…

What’s wrong with this code?

```
{
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
. . .

fail:
    SSLFreeBuffer(&hashCtx);

    return err;
}
```

What's wrong with this code?

```
{
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
. . .

fail:
    SSLFreeBuffer(&hashCtx);

    return err;
}
```

Basically, this is checking a signature (more on this later)

What’s wrong with this code?

```
{
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
. . .

fail:
    SSLFreeBuffer(&hashCtx);

    return err;
}
```

Will *always* jump to ‘fail’

Basically, this is checking a signature (more on this later)

Rule: Use a safe allocator

- ASLR challenges exploits by making the base address of libraries unpredictable

- **Challenge heap-based overflows** by making the **addresses** returned by `malloc` **unpredictable**
  - Can have some negative performance impact

- Example implementations:
  - **Windows Fault-Tolerant Heap**
    - http://msdn.microsoft.com/en-us/library/windows/desktop/dd744764(v=vs.85).aspx
  - **DieHard** (on which fault-tolerant heap is based)
    - http://plasma.cs.umass.edu/emery/diehard.html

Rule: Favor safe libraries

- **Libraries** encapsulate **well-thought-out design**.
  Take advantage!

- **Smart pointers**
  - Pointers with only safe operations
  - Lifetimes managed appropriately
  - First in the Boost library, now a C++11 standard

- **Networking**: Google protocol buffers, Apache Thrift
  - For dealing with network-transmitted data
  - Ensures input validation, parsing, etc.
  - Efficient

Defensive coding practices

- Think defensive driving
  - Avoid depending on anyone else around you
  - If someone does something unexpected, you won't crash (or worse)
  - It's about *minimizing trust*

- Each module takes responsibility for checking the validity of all inputs sent to it
  - Even if you "know" your callers will never send a NULL pointer…
  - …Better to throw an exception (or even exit) than run malicious code

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

Automated testing techniques

- **Code analysis**
  - Static: Many of the bugs we've shown could be easily detected (but we run into the Halting Problem)
  - Dynamic: Run in a VM and look for bad writes (valgrind)

- **Fuzz testing**
  - Generate **many random inputs**, see if the program fails
    - Totally random
    - Start with a valid input file and mutate
    - Structure-driven input generation: take into account the intended format of the input (e.g., "string int float string")
  - Typically involves *many many* inputs (clusterfuzz.google.com)

Penetration testing

- Fuzz testing is a form of "**penetration testing**" (pen testing)
- Pen testing assesses security by *actively trying to find explotable vulnerabilities*
  - Useful for both attackers and defenders
- Pen testing is useful at many different levels
  - Testing programs
  - Testing applications
  - Testing a network
  - Testing a server…

Kinds of fuzzing

Kinds of fuzzing

- **Black box**
  - The tool knows nothing about the program or its input
  - **Easy to use** and get started, but will **explore only shallow states** unless it gets lucky

Kinds of fuzzing

- **Black box**
  - The tool knows nothing about the program or its input
  - **Easy to use** and get started, but will **explore only shallow states** unless it gets lucky

- **Grammar based**
  - The tool generates input informed by a grammar
  - **More work to use**, to produce the grammar, but **can go deeper** in the state space

Kinds of fuzzing

- **Black box**
  - The tool knows nothing about the program or its input
  - **Easy to use** and get started, but will **explore only shallow states** unless it gets lucky
- **Grammar based**
  - The tool generates input informed by a grammar
  - **More work to use**, to produce the grammar, but **can go deeper** in the state space
- **White box**
  - The tool generates new inputs at least partially informed by the code of the program being fuzzed
  - Often **easy to use**, but **computationally expensive**

Fuzzing inputs

Fuzzing inputs

- **Mutation**
  - Take a **legal input and *mutate* it**, using that as input

Fuzzing inputs

- **Mutation**
  - Take a **legal input and *mutate* it**, using that as input
  - Legal input might be human-produced, or automated, e.g., from a grammar or SMT solver query
    - Mutation might also be forced to adhere to grammar

Fuzzing inputs

- **Mutation**
  - Take a **legal input and *mutate* it**, using that as input
  - Legal input might be human-produced, or automated, e.g., from a grammar or SMT solver query
    - Mutation might also be forced to adhere to grammar
- **Generational**
  - **Generate** input from scratch, e.g., from a **grammar**

Fuzzing inputs

- **Mutation**
  - Take a **legal input and *mutate* it**, using that as input
  - Legal input might be human-produced, or automated, e.g., from a grammar or SMT solver query
    - Mutation might also be forced to adhere to grammar
- **Generational**
  - **Generate** input from scratch, e.g., from a **grammar**
- **Combinations**

Fuzzing inputs

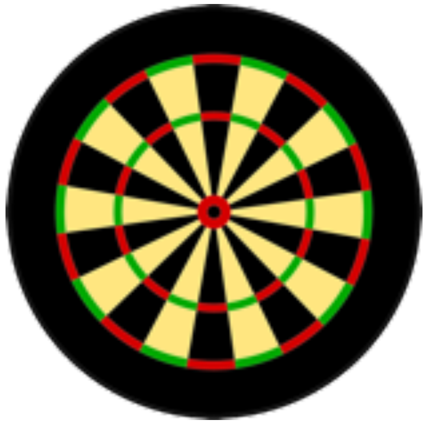
- **Mutation**
  - Take a **legal input and *mutate* it**, using that as input
  - Legal input might be human-produced, or automated, e.g., from a grammar or SMT solver query
    - Mutation might also be forced to adhere to grammar
- **Generational**
  - **Generate** input from scratch, e.g., from a **grammar**
- **Combinations**
  - Generate initial input, mutate^N, generate new inputs, …

Fuzzing inputs

- **Mutation**
  - Take a **legal input and *mutate* it**, using that as input
  - Legal input might be human-produced, or automated, e.g., from a grammar or SMT solver query
    - Mutation might also be forced to adhere to grammar
- **Generational**
  - **Generate** input from scratch, e.g., from a **grammar**
- **Combinations**
  - Generate initial input, mutate^N, generate new inputs, …
  - Generate mutations according to grammar

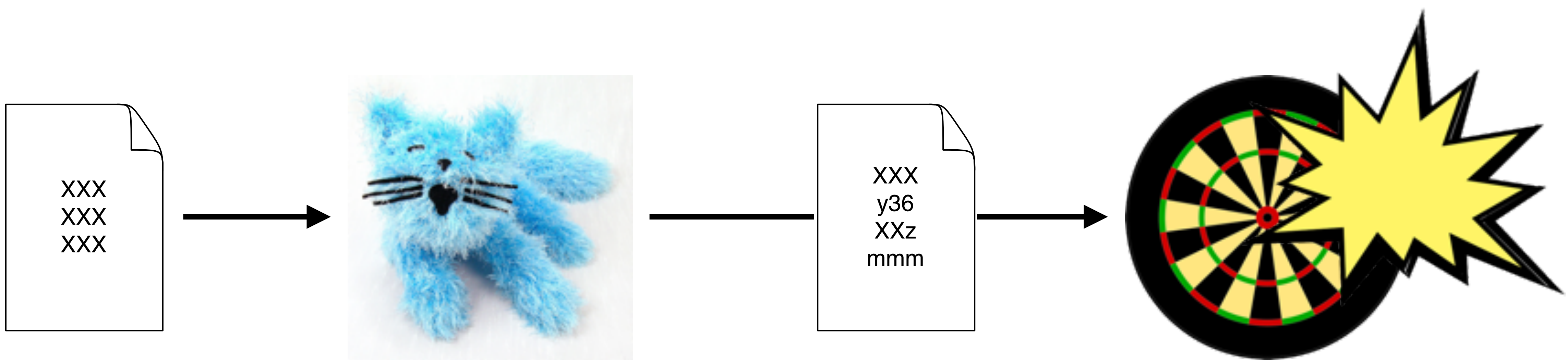
File-based fuzzing

- **Mutate** or **generate** inputs
- **Run the target program** with them
- See **what happens**



File-based fuzzing

- **Mutate** or **generate** inputs
- **Run the target program** with them
- See **what happens**



Examples: Radamsa and Blab

Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
```

Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
```

Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
% echo … | radamsa --seed 12 -n 4 | bc -l
```

Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
% echo … | radamsa --seed 12 -n 4 | bc -l
```

- **Blab** *generates* inputs according to a grammar (*grammar-based*), specified as regexps and CFGs

https://code.google.com/p/ouspg/wiki/Radamsa https://code.google.com/p/ouspg/wiki/Blab

Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
% echo … | radamsa --seed 12 -n 4 | bc -l
```

- **Blab** *generates* inputs according to a grammar (*grammar-based*), specified as regexps and CFGs

```
% blab -e '(([wrstp][aeiouy]{1,2}){1,4} 32){5} 10'
```

https://code.google.com/p/ouspg/wiki/Radamsa https://code.google.com/p/ouspg/wiki/Blab

Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
% echo … | radamsa --seed 12 -n 4 | bc -l
```

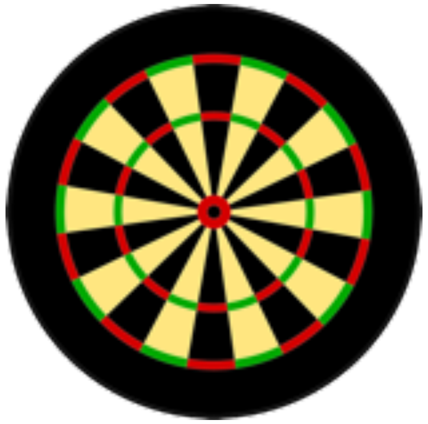
- **Blab** *generates* inputs according to a grammar (*grammar-based*), specified as regexps and CFGs

```
% blab -e '(([wrstp][aeiouy]{1,2}){1,4} 32){5} 10'
soty wypisi tisyro to patu
```

https://code.google.com/p/ouspg/wiki/Radamsa https://code.google.com/p/ouspg/wiki/Blab

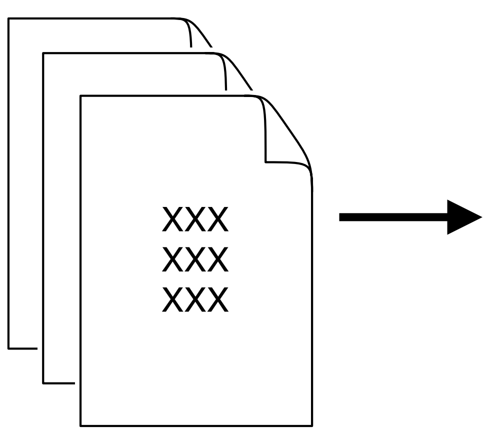
Network-based fuzzing

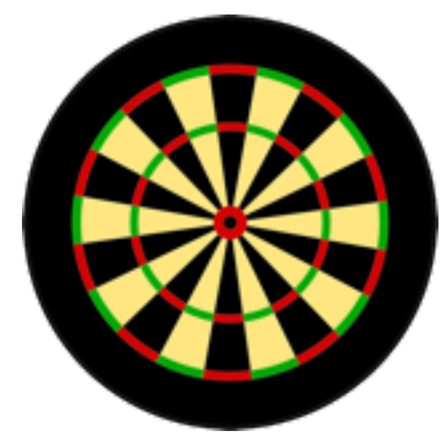
- **Act as 1/2 of a communicating pair**
  - Inputs could be produced by replaying previously recorded interaction, and altering it, or producing it from scratch (e.g., from a protocol grammar)



Network-based fuzzing

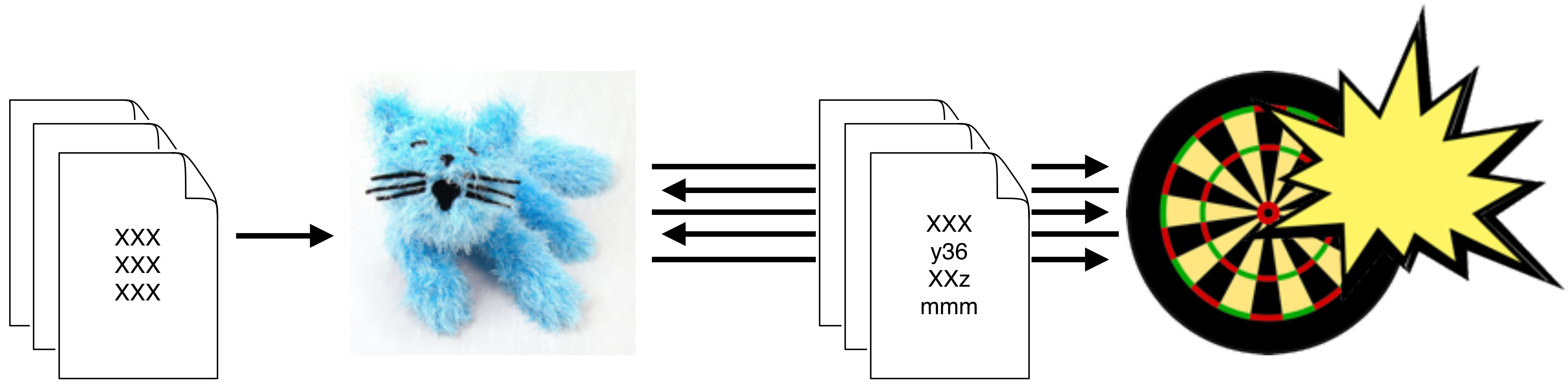
- **Act as 1/2 of a communicating pair**
  - Inputs could be produced by replaying previously recorded interaction, and altering it, or producing it from scratch (e.g., from a protocol grammar)





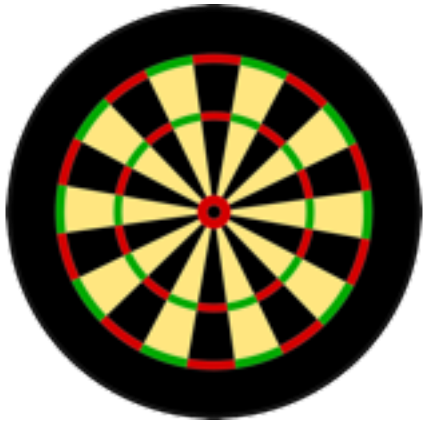
Network-based fuzzing

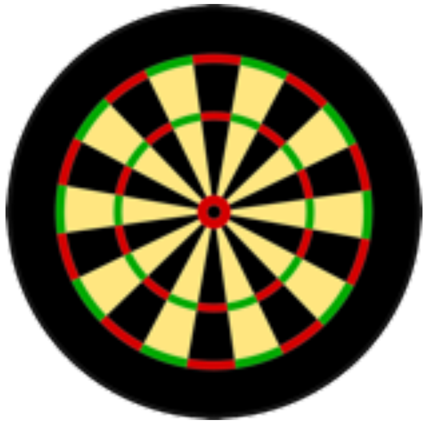
- **Act as 1/2 of a communicating pair**
  - Inputs could be produced by replaying previously recorded interaction, and altering it, or producing it from scratch (e.g., from a protocol grammar)



Network-based fuzzing

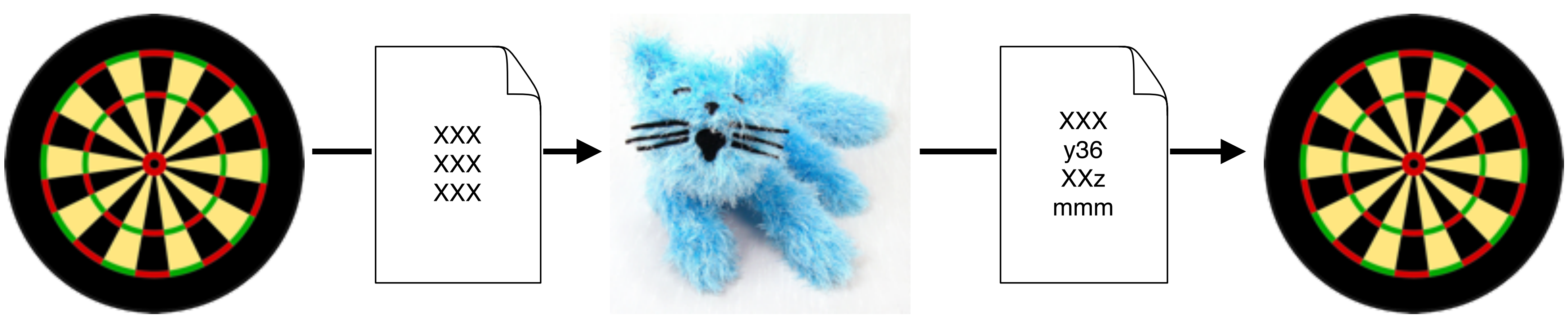
- **Act as a "man in the middle"**
  - mutating inputs exchanged between parties (perhaps informed by a grammar)





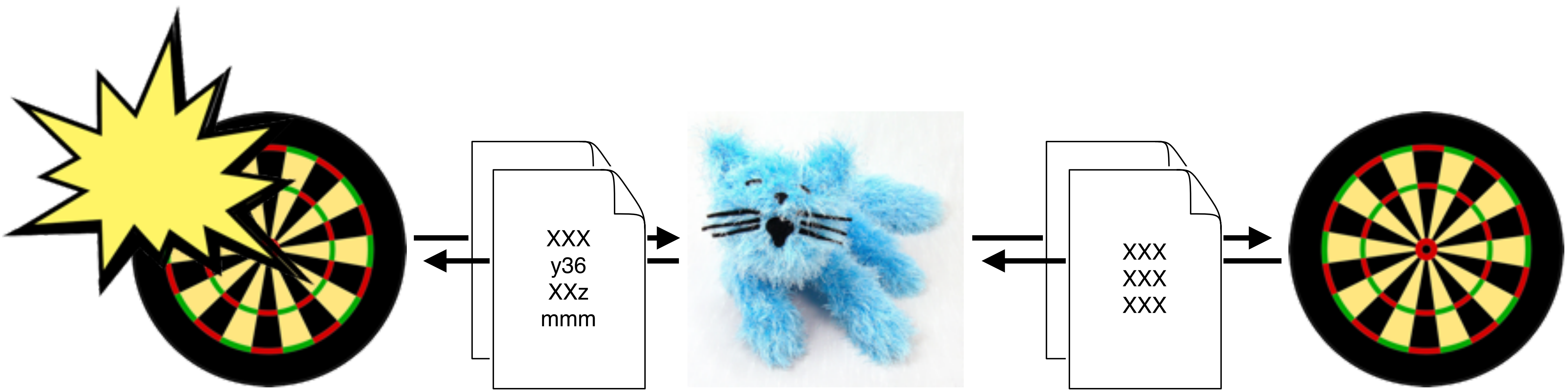
Network-based fuzzing

- **Act as a "man in the middle"**
    - mutating inputs exchanged between parties (perhaps informed by a grammar)



Network-based fuzzing

- **Act as a “man in the middle”**
    - mutating inputs exchanged between parties (perhaps informed by a grammar)



There are many many fuzzers

- American Fuzzy Lop
  - Mutation-based white-box buzzer
- SPIKE
  - A library for creating network-based fuzzers
- Burp Intruder
  - Automates customized attacks against web apps
- And many more… (BFF, Sulley, …)

You fuzz, you crash. Then what?

You fuzz, you crash. Then what?

Try to find the **root cause**

You fuzz, you crash. Then what?

Try to find the **root cause**

Is there a smaller input that crashes in the same spot? (Make it easier to understand)

You fuzz, you crash. Then what?

Try to find the **root cause**

Is there a smaller input that crashes in the same spot? (Make it easier to understand)

Are there multiple crashes that point back to the same bug?

You fuzz, you crash. Then what?

Try to find the **root cause**

Is there a smaller input that crashes in the same spot? (Make it easier to understand)

Are there multiple crashes that point back to the same bug?

Determine if this crash represents an **exploitable vulnerability**

You fuzz, you crash. Then what?

Try to find the **root cause**

Is there a smaller input that crashes in the same spot? (Make it easier to understand)

Are there multiple crashes that point back to the same bug?

Determine if this crash represents an **exploitable vulnerability**

In particular, is there a buffer overrun?

Finding memory errors

Finding memory errors

1. **Compile** the program with **Address Sanitizer** (ASAN)
    - Instruments accesses to arrays to check for overflows, and use-after-free errors
    - https://code.google.com/p/address-sanitizer/

Finding memory errors

1. **Compile** the program with **Address Sanitizer** (ASAN)
   - Instruments accesses to arrays to check for overflows, and use-after-free errors
   - https://code.google.com/p/address-sanitizer/

2. **Fuzz it**

Finding memory errors

1. **Compile** the program with **Address Sanitizer** (ASAN)
   - Instruments accesses to arrays to check for overflows, and use-after-free errors
   - https://code.google.com/p/address-sanitizer/

2. **Fuzz it**

3. Did the program **crash with an ASAN-signaled error**? Then worry about exploitability

Finding memory errors

1. **Compile** the program with **Address Sanitizer** (ASAN)
   - Instruments accesses to arrays to check for overflows, and use-after-free errors
   - https://code.google.com/p/address-sanitizer/

2. **Fuzz it**

3. Did the program **crash with an ASAN-signaled error**? Then worry about exploitability

- Similarly, you can *compile with other sorts of error checkers* for the purposes of testing
  - E.g., `valgrind` memcheck http://valgrind.org/

Defensive coding summary

- **Understand** the systems and libraries you use
  - printf("%n"); !!
  - This is the root cause of most crypto implementation errors (next section)
- **Assume the worst**; code and design defensively
  - Think defensive driving
- Use ***pen testing*** tools to help automate
  - Assume that attackers will; seek to have at least as much information as they!

This time

Continued with
Software Security

Writing & testing for
Secure Code

- Return oriented programming
- Format string & integer overflow vulnerabilities
- Defenses via good code & automated pen testing

Next time

Continuing with
Software Security

Getting sick with
Malware

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