# CS 366
# Intro to Computer Security

Dr. Stella Sun
EECS
University of Tennessee
Fall 2022

# Today's Class

- Buffer overflow defenses

# What Do Attacks Have in Common

- Attacker is able to input some data used by the program
- This causes unintentional access to some memory area in the program, e.g.,
  - past a buffer
  - arbitrary locations on the stack

# Defense Category 1: Making Exploitation Harder

- Recall the steps of a stack smashing
  - injecting attacker code into memory (no zeros)
  - overwrite %eip to point to and run attacker code
  - finding the return address (good guess)

# How Can We Make These Steps Harder

- Basic idea

  - complicating exploitation by changing compiler, libraries and/or OS (i.e., architectural design)

  - so that the application code doesn't need to be changed

- Let's look at some techniques

  - stack canaries

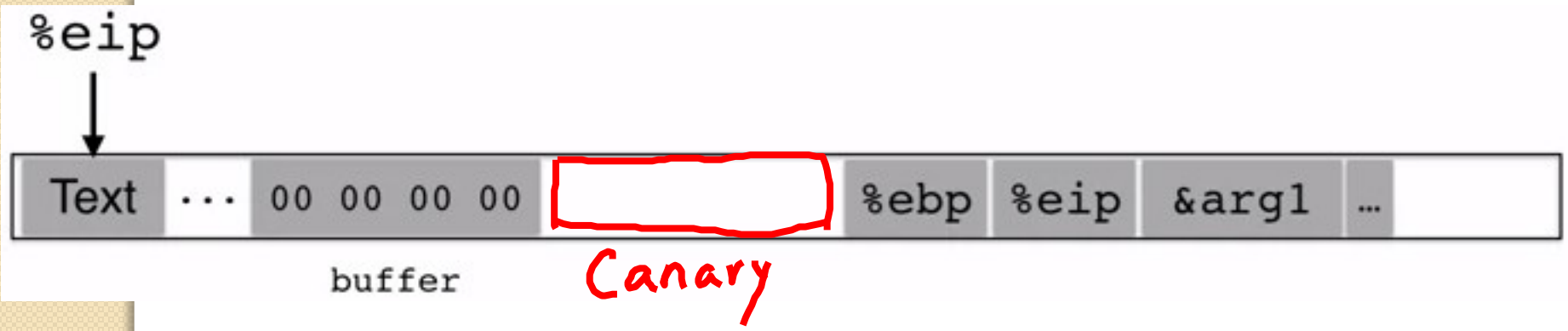
  - ASLR

# Stack Canaries

- Why called canary?

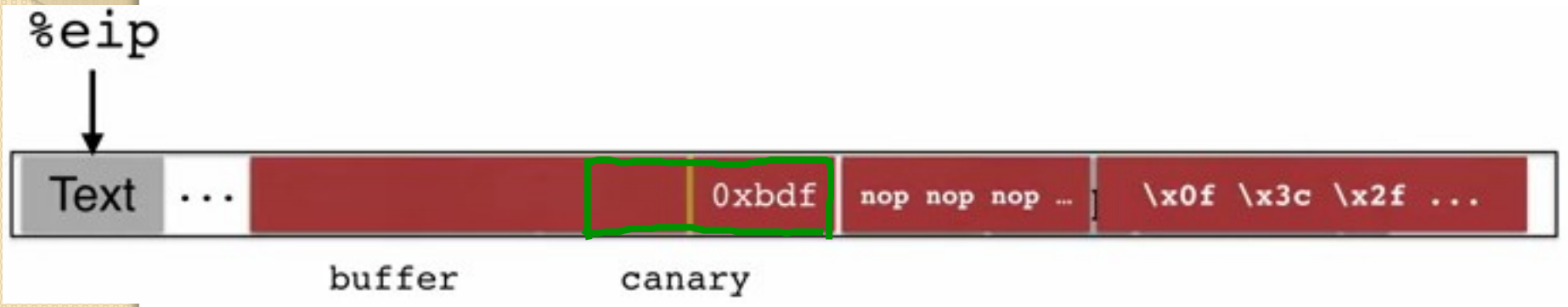
# Stack Canaries

- Detecting overflow with canaries

# Stack Canaries

- If the attacker overruns the buffer




- The attacker will also need to overrun the canary to overwrite %eip

- We can change the compiler such that before returning, it checks if the canary value is changed

# What Value Should The Canary Have

- Basic idea
  - choose one that's harder to guess or use

- Terminator canaries
  - NULL, CR, LF, etc.
  - leveraging the fact that string functions stop copying there
  - attackers cannot use string functions as attack vector

# What Value Should The Canary Have

- Random canaries

  - write a new random value at each process start

  - store this value in a global variable and write-protect it

  - insert into each stack frame

  - known only to the buffer overflow protection code

  - attackers need to be able to steal (read) the canary value

# What Value Should The Canary Have

- Random XOR canaries

  - random canaries XORed with control data (e.g., %eip)

  - slightly better because in some cases the %eip can be changed without overrunning the canary (e.g., when a pointer is overflowed to point to a %eip)

  - attackers needs to be able to steal (read) the canary value

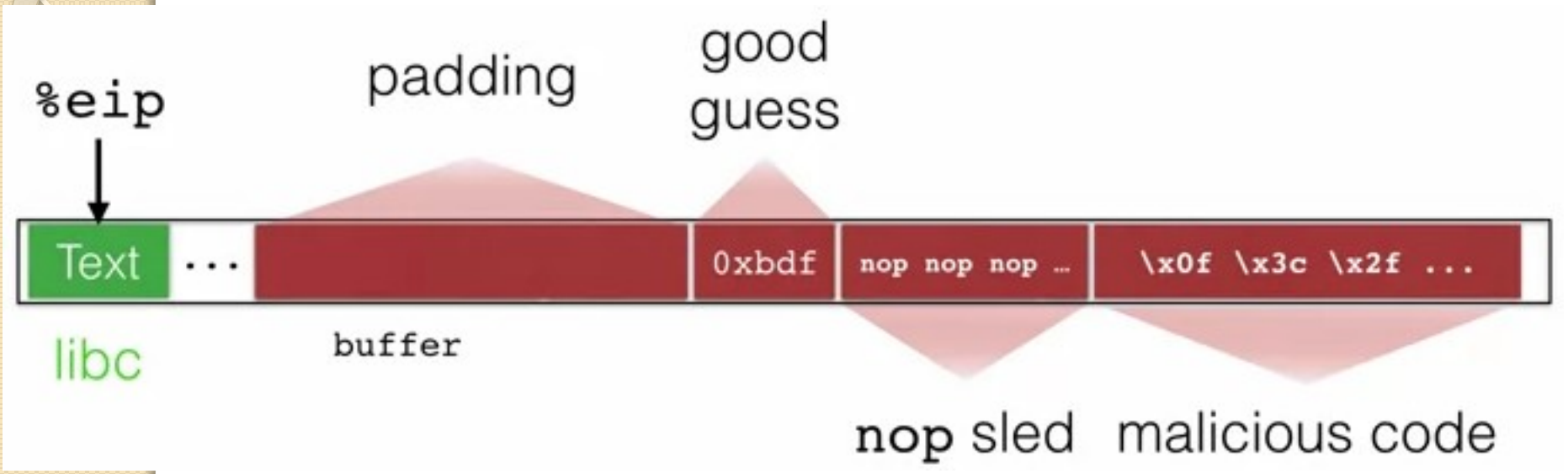
  - only protect control data (or whatever is being XORed), not other data or pointer itself

# What If Canaries Fail

- For example: attacker successfully modifies %eip without changing the canary
  - then attacker can still get his injected code to run

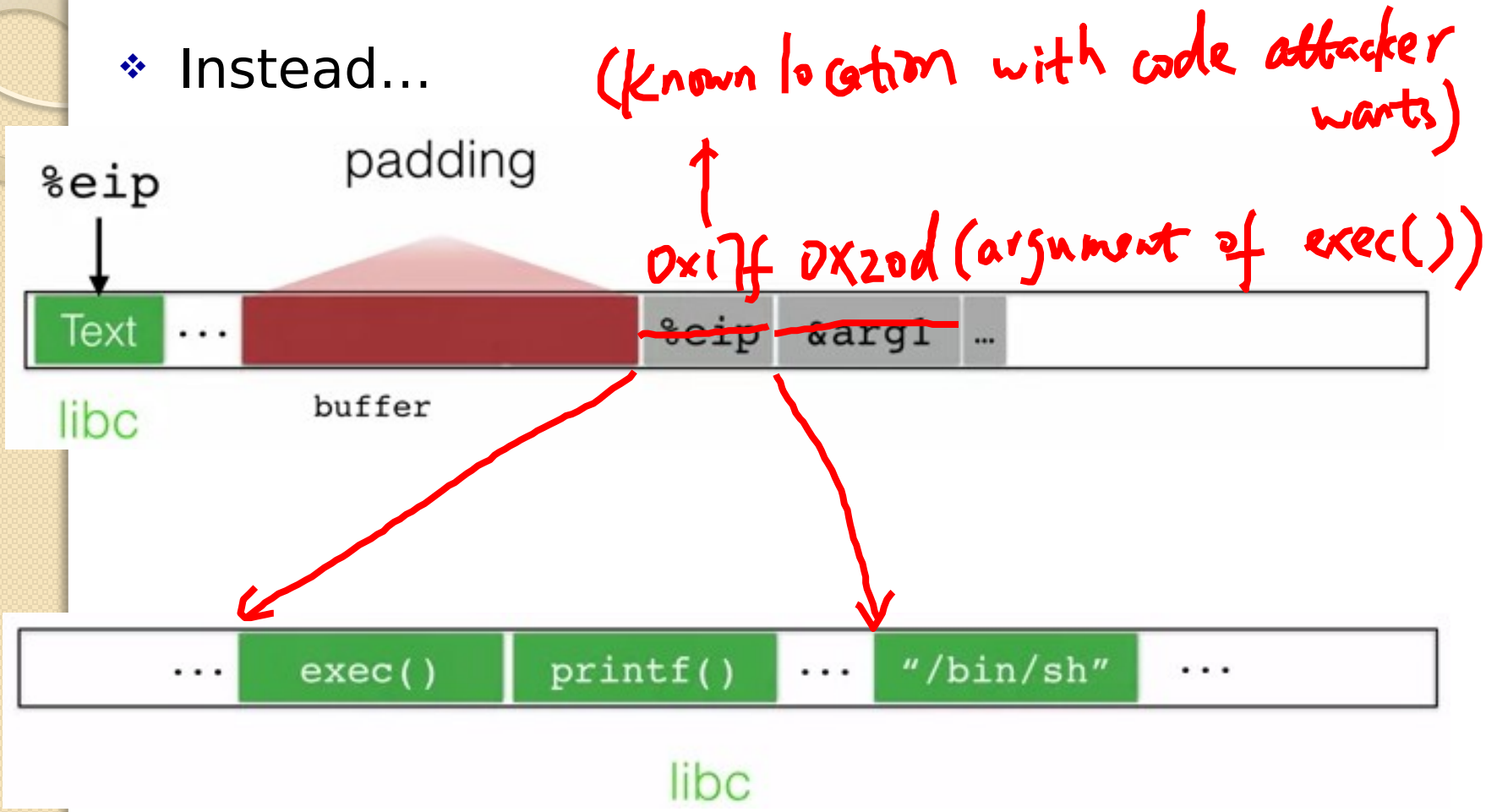
- Solution: make the stack non-executable
  - general idea: make everything other than the code (or text) region non-executable
  - called data execution protection (DEP) or W^X
  - very widely used memory protection

# Unfortunately, Can't Prevent return-to-libc



# Unfortunately, Can't Prevent return-to-libc

- Instead…

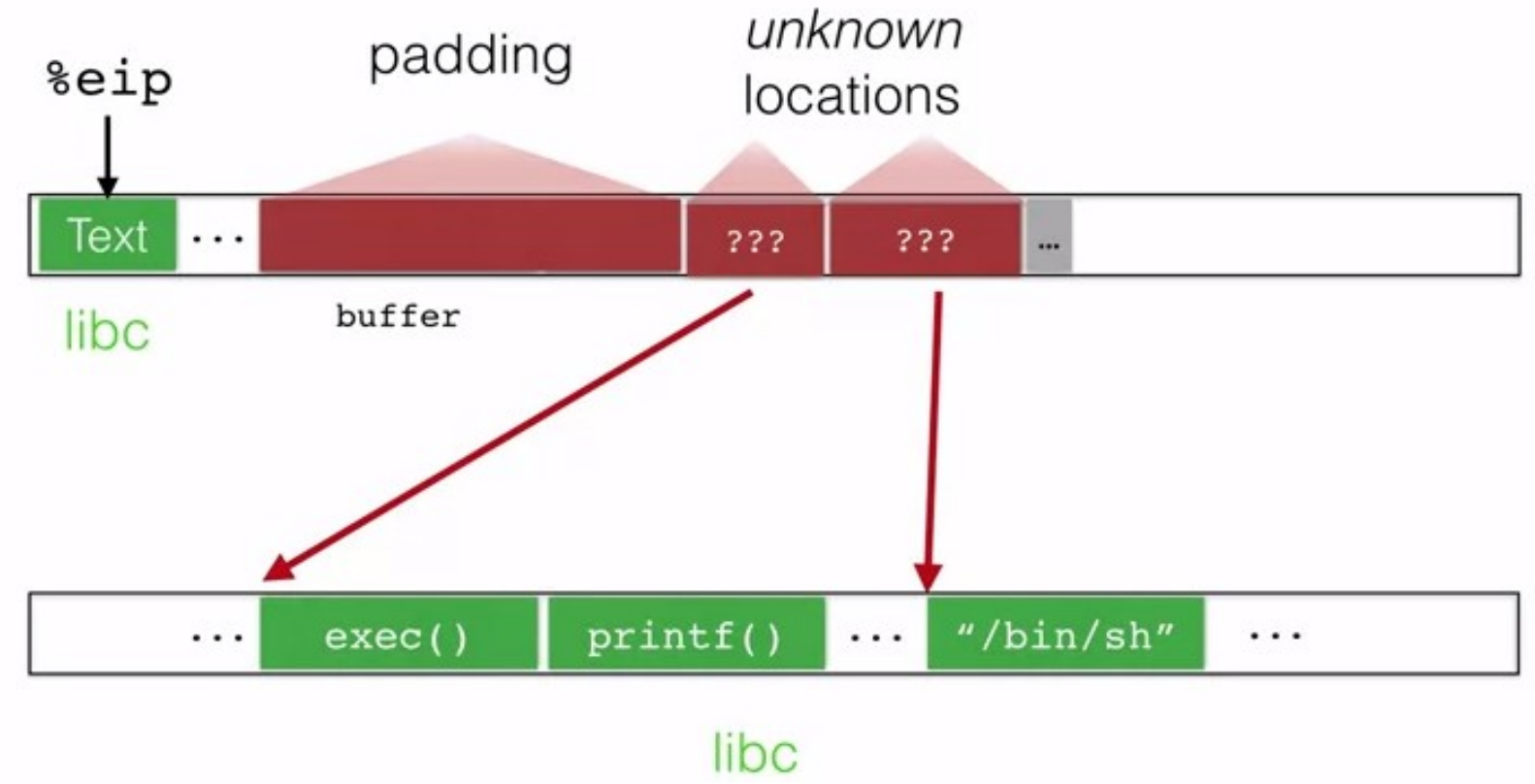



- Upon returning from the overrun function, the current process is turned into a shell

# Use Address Space Layout Randomization (A

- Basic idea:

  - randomly put standard libraries and other elements (e.g., stack) in memory to make it harder to guess



# ASLR

- Available on most modern OSs
- Downside

- only shifts the base of memory areas (not the locations within those areas)

- may not apply to program code, only libraries

- needs sufficient randomness, or can be brute forced

took 216 seconds to brute force ASLR on 32-bit systems [Shacham et al., “On the effectiveness of address- space randomization,” (CCS’04)]

- fortunately, 64-bits systems are hard to brute force

# Return-Oriented Programming (ROP)

- A special class of return-to-libc attack that is gaining popularity

[Schacham et al., The Geometry of Innocent Flesh on the Bone: Return-to-libc without Function Calls (on the x86), CCS'07]

- Basic idea:

  - Instead of the entire function, it pieces together code that's not intended to be there that ends with ret (called gadgets)

  - Harder to detect and avoid

  - Intuition: one can make out more words than were intentionally put there, e.g., "dress" is a suffix of "address", "head" can be made out from "the address"

  - This is possible with dense languages such as x86

# Dense x86 Instructions

- Suppose we have two instructions in libc:

```
f7 c7 07 00 00 00        test $0x00000007, %edi
0f 95 45 c3              setnzb -61(%ebp)
```

- Starting one byte later, the attacker instead obtains:

```
c7 07 00 00 00 0f        movl $0x0f000000, (%edi)
95                       xchg %ebp, %eax
45                       inc %ebp
c3                       ret
```

- How frequently does this happen?

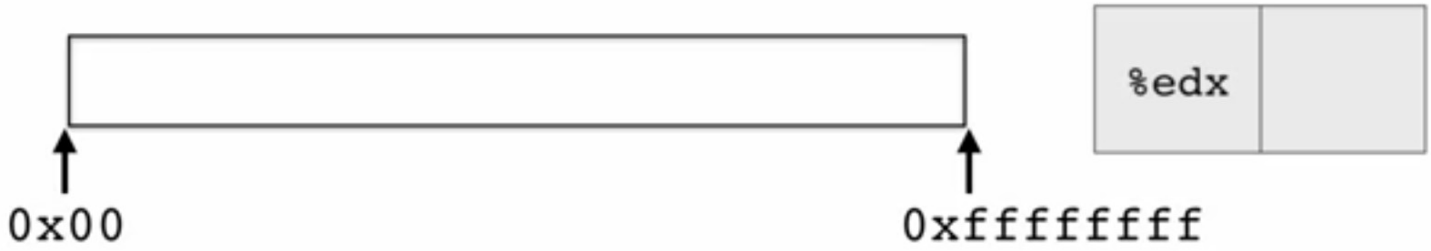
  - Very frequently with dense languages

  - The set of gadgets are Turing complete: ROP can do anything possible with x86 code

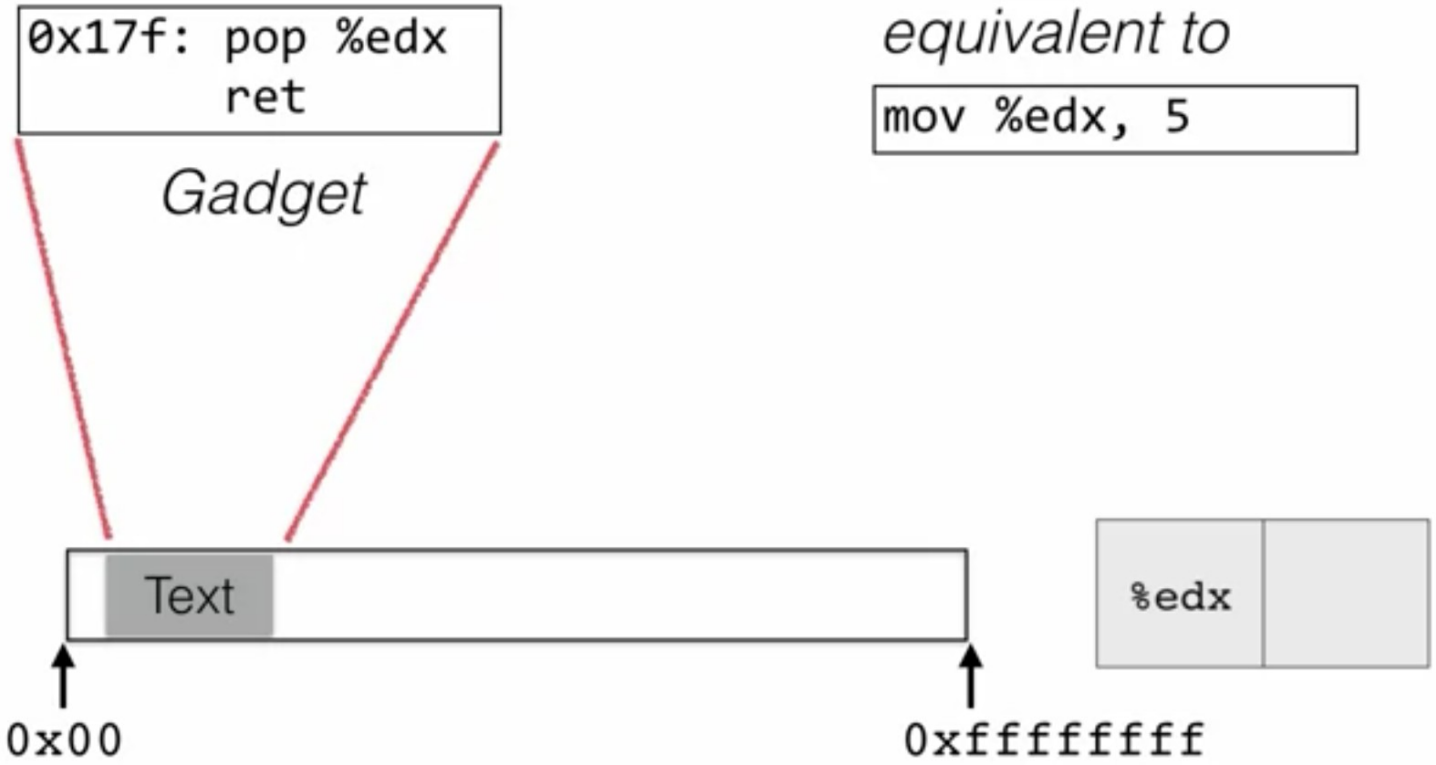
# Simple ROP Example

*equivalent to*

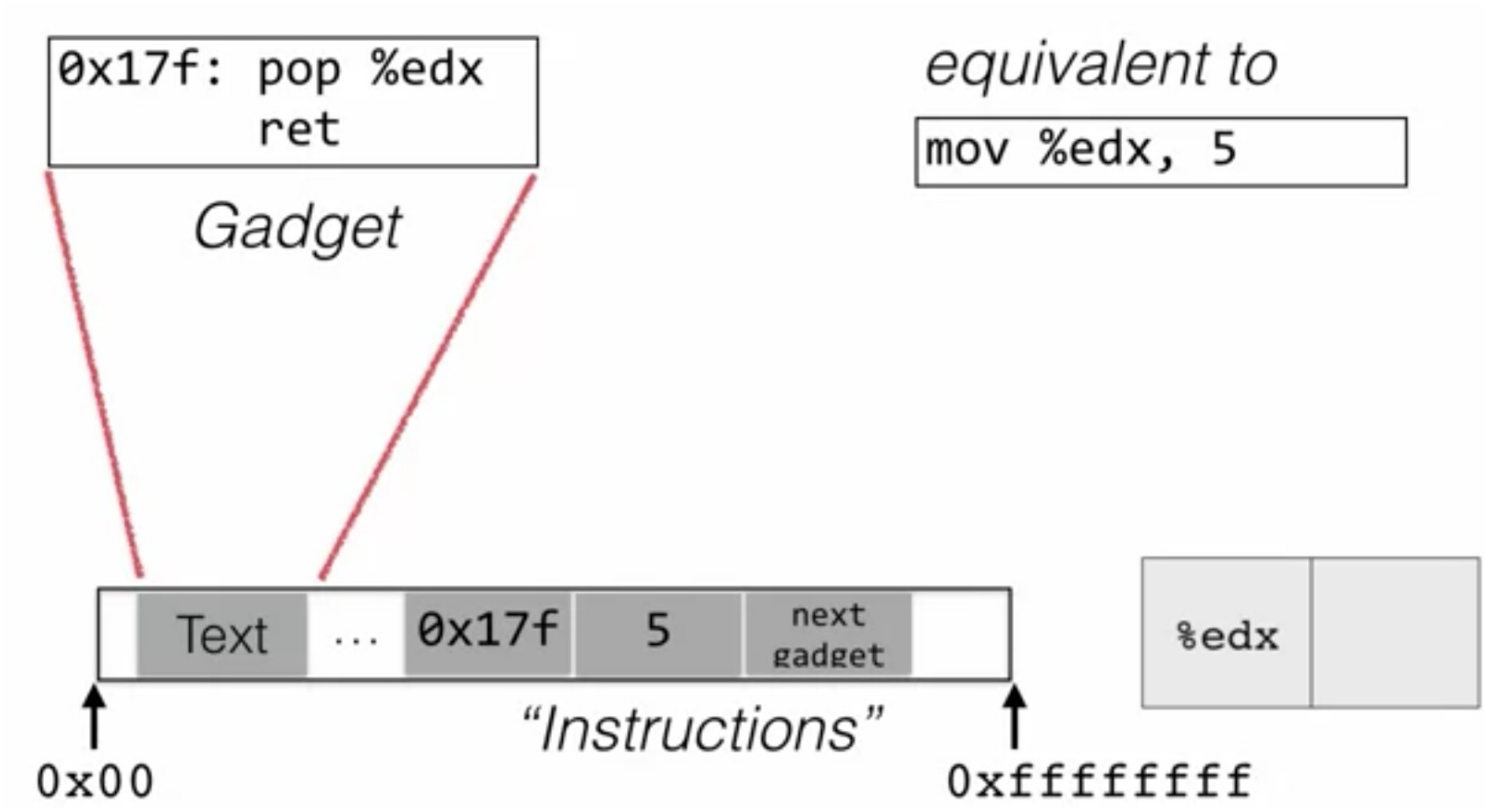
```
mov %edx, 5
```



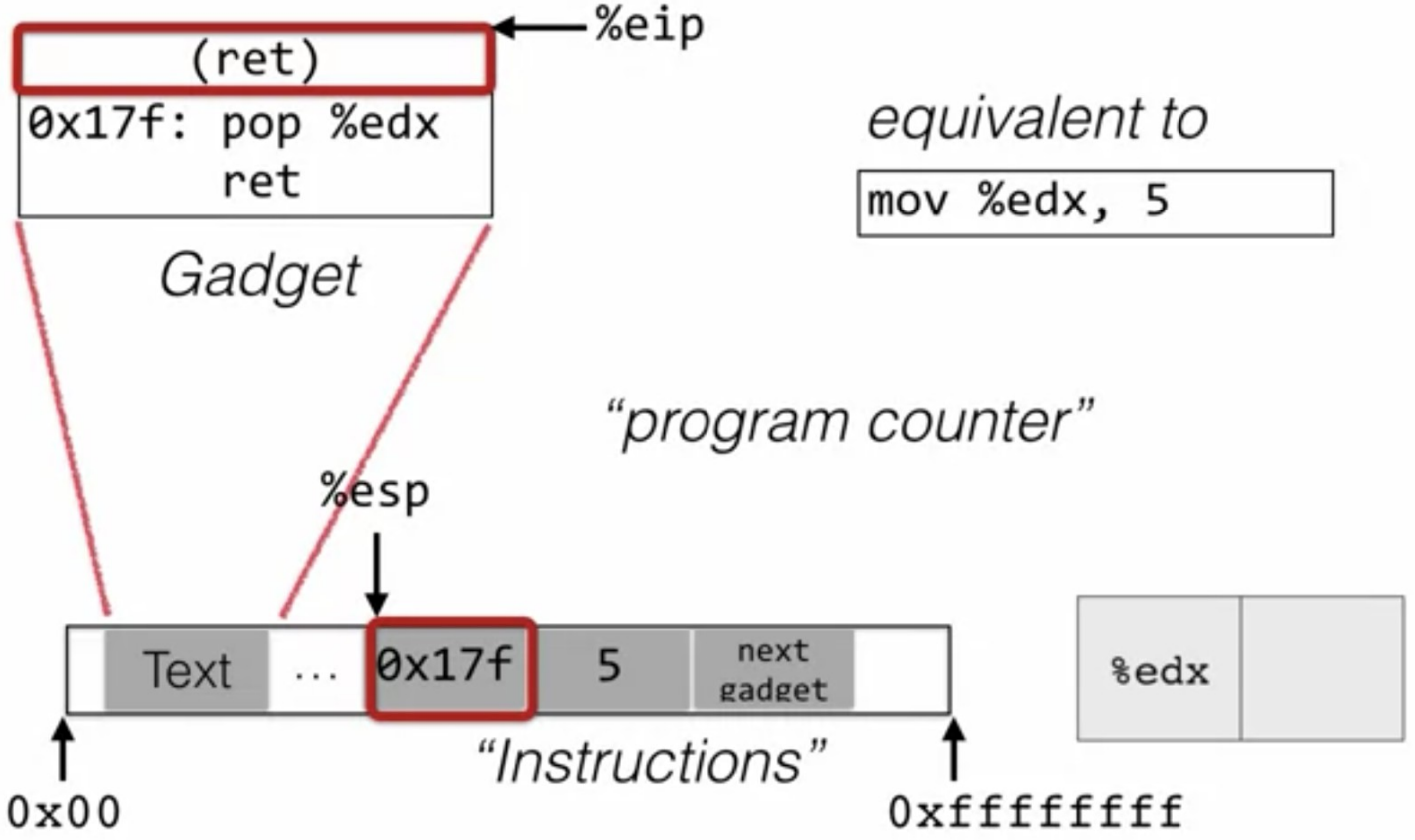
# Simple ROP Example



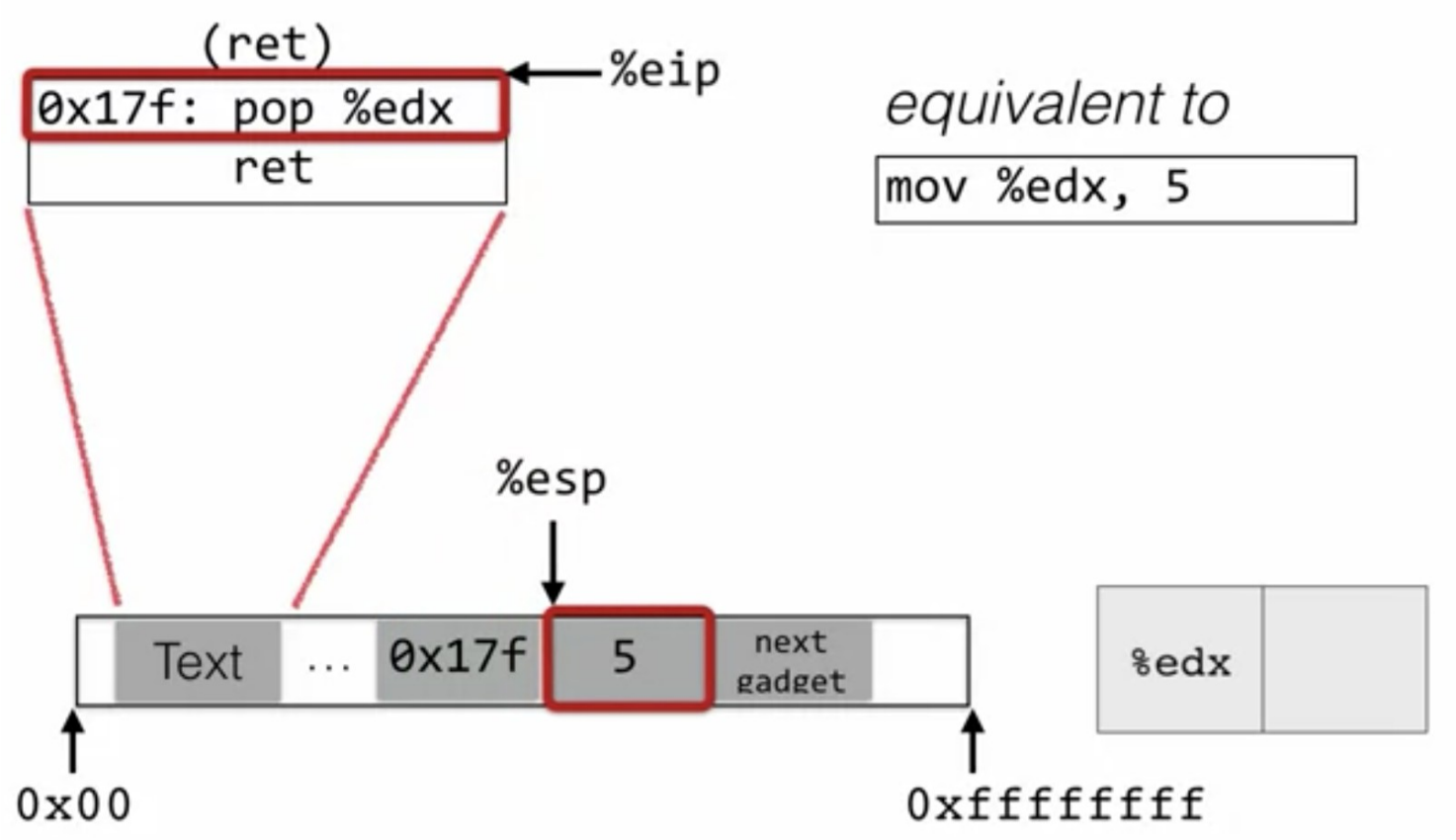
# Simple ROP Example



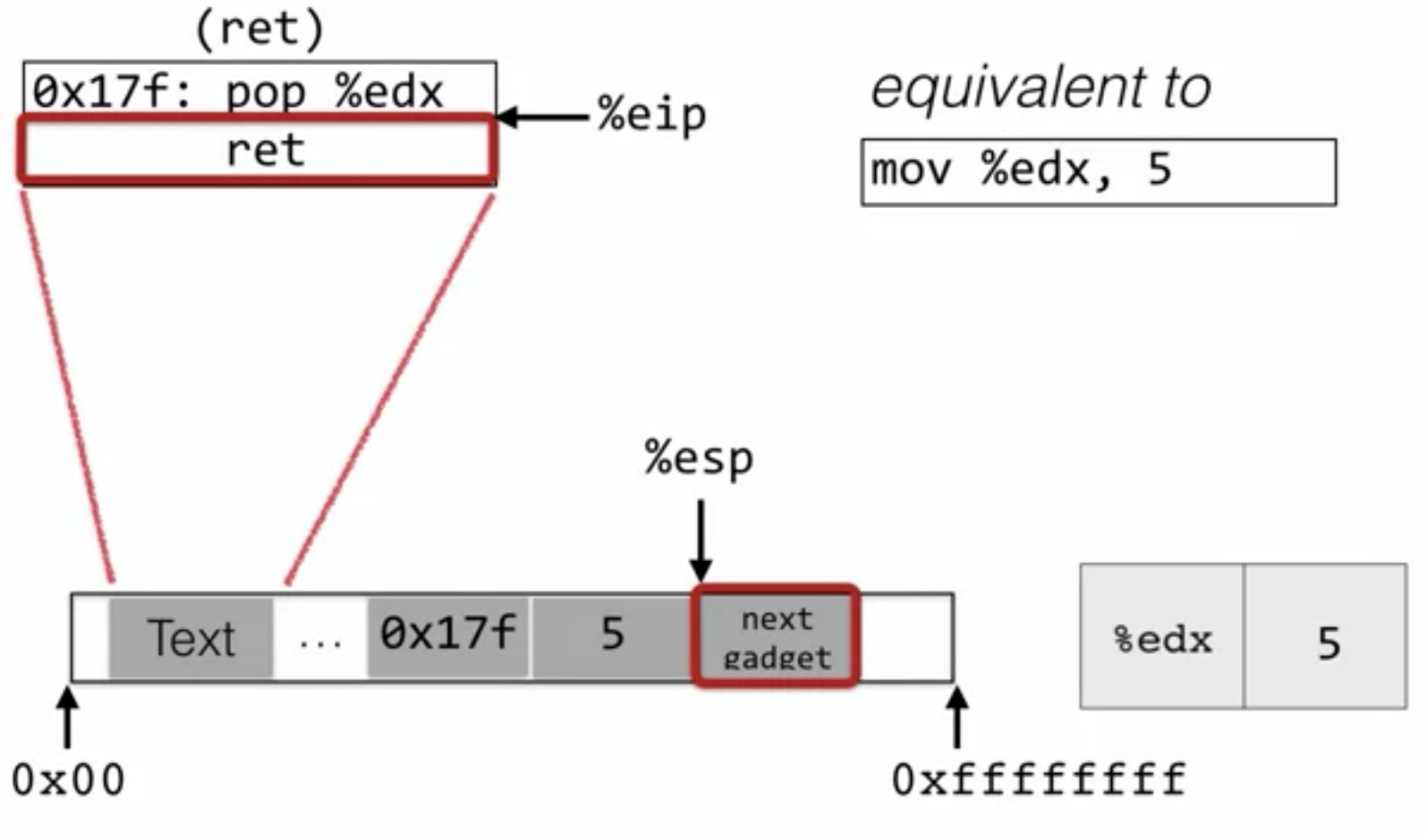
# Simple ROP Example



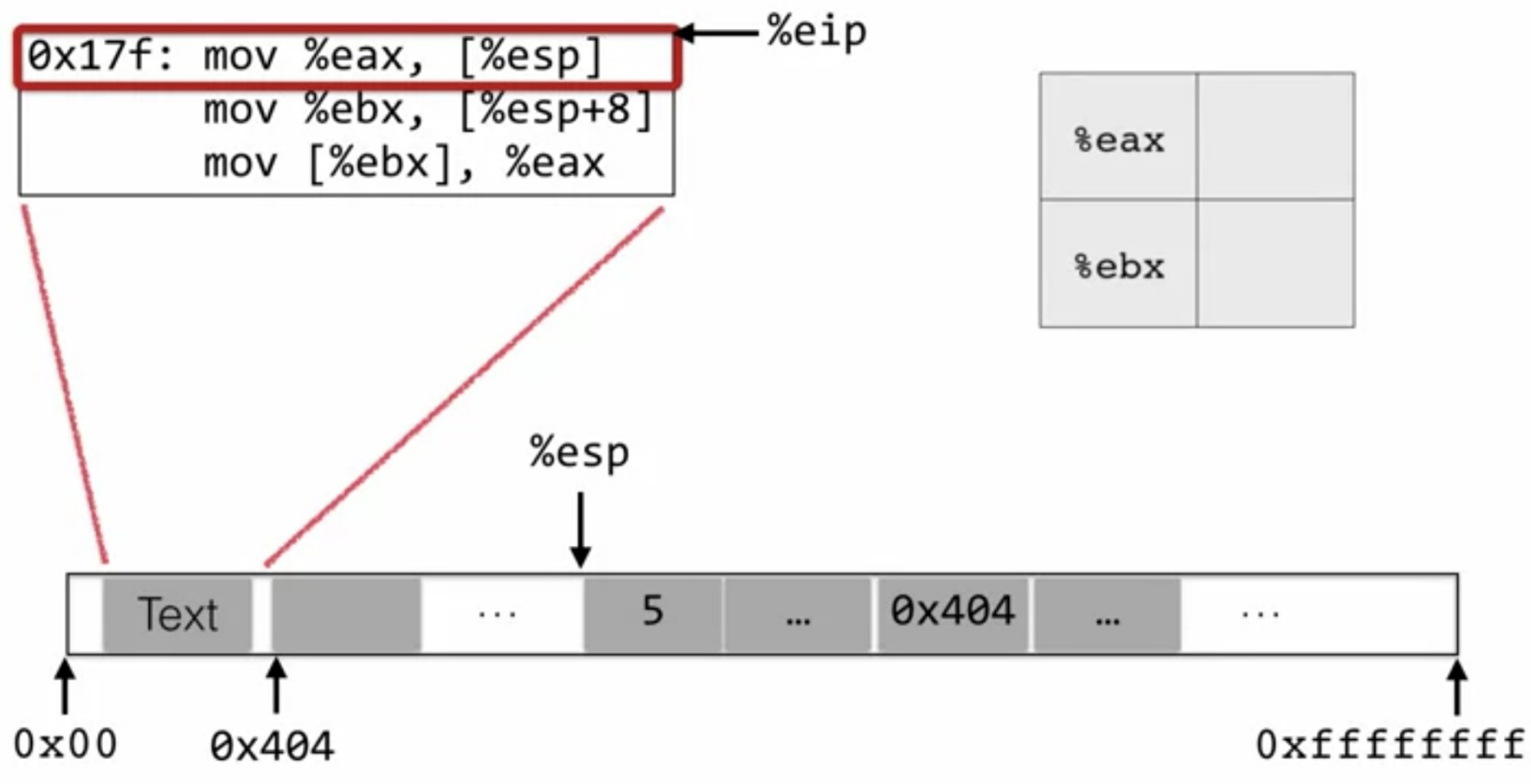
# Simple ROP Example



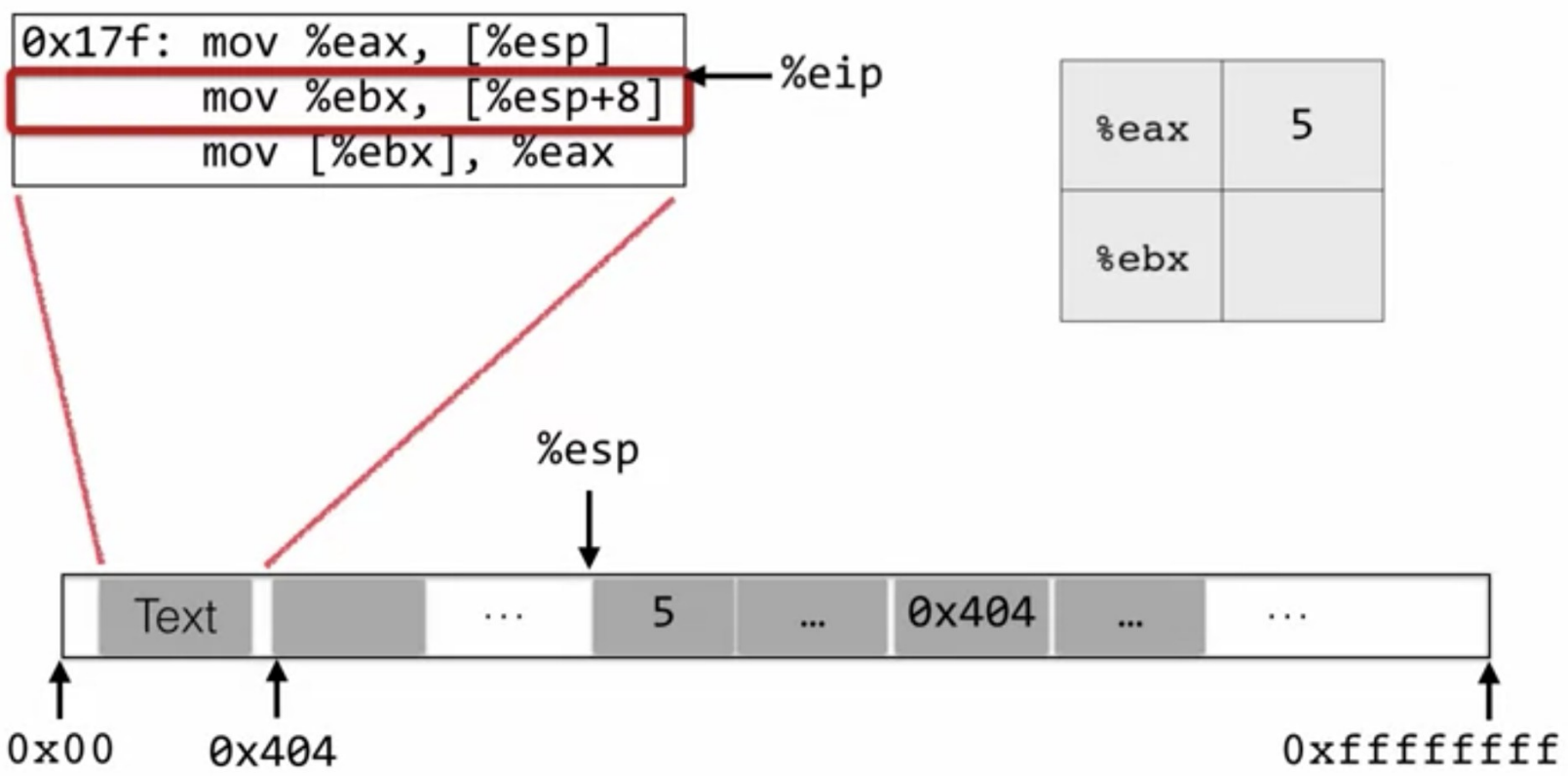
# Simple ROP Example



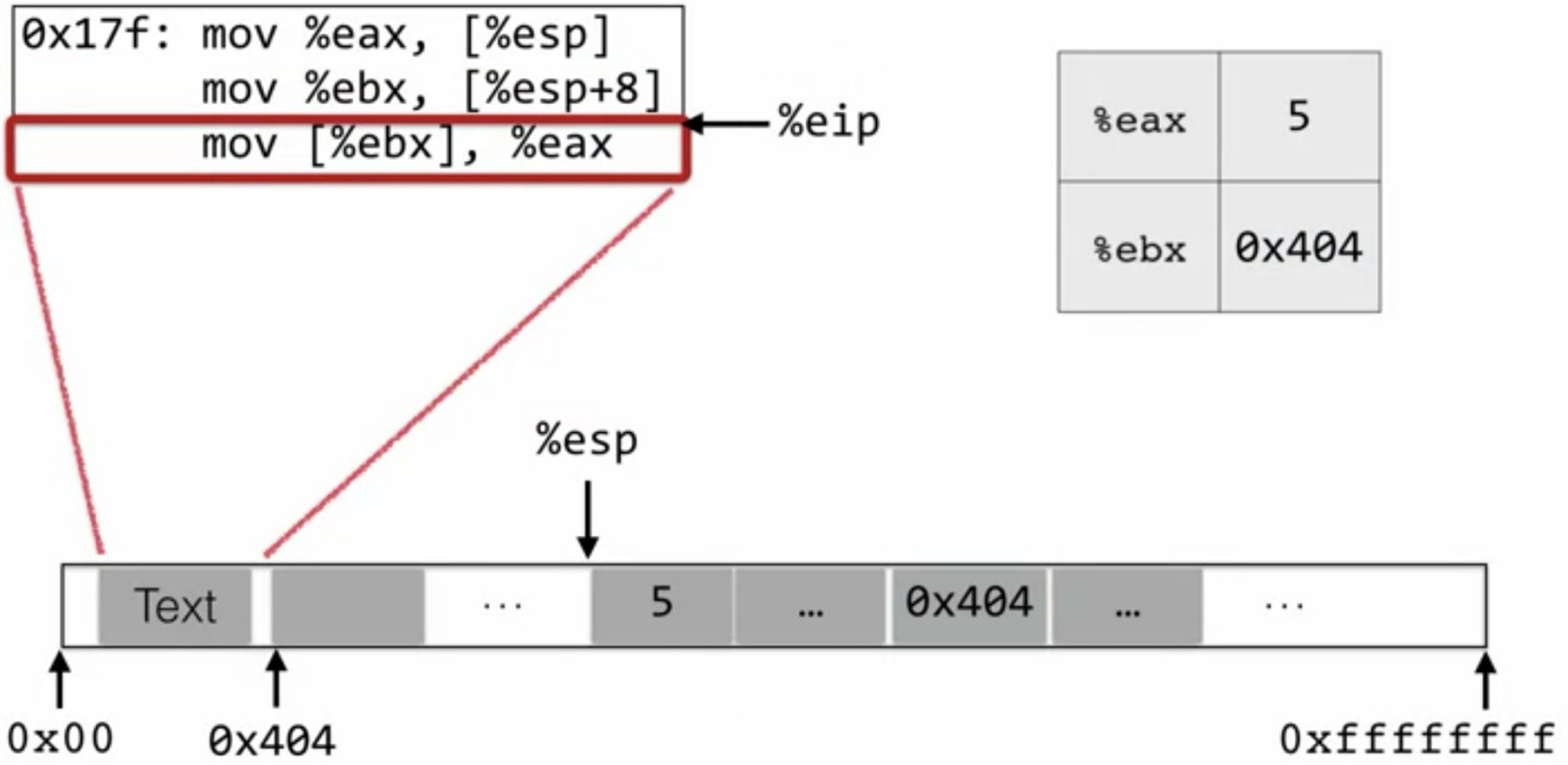
# Another Example – Code Seque



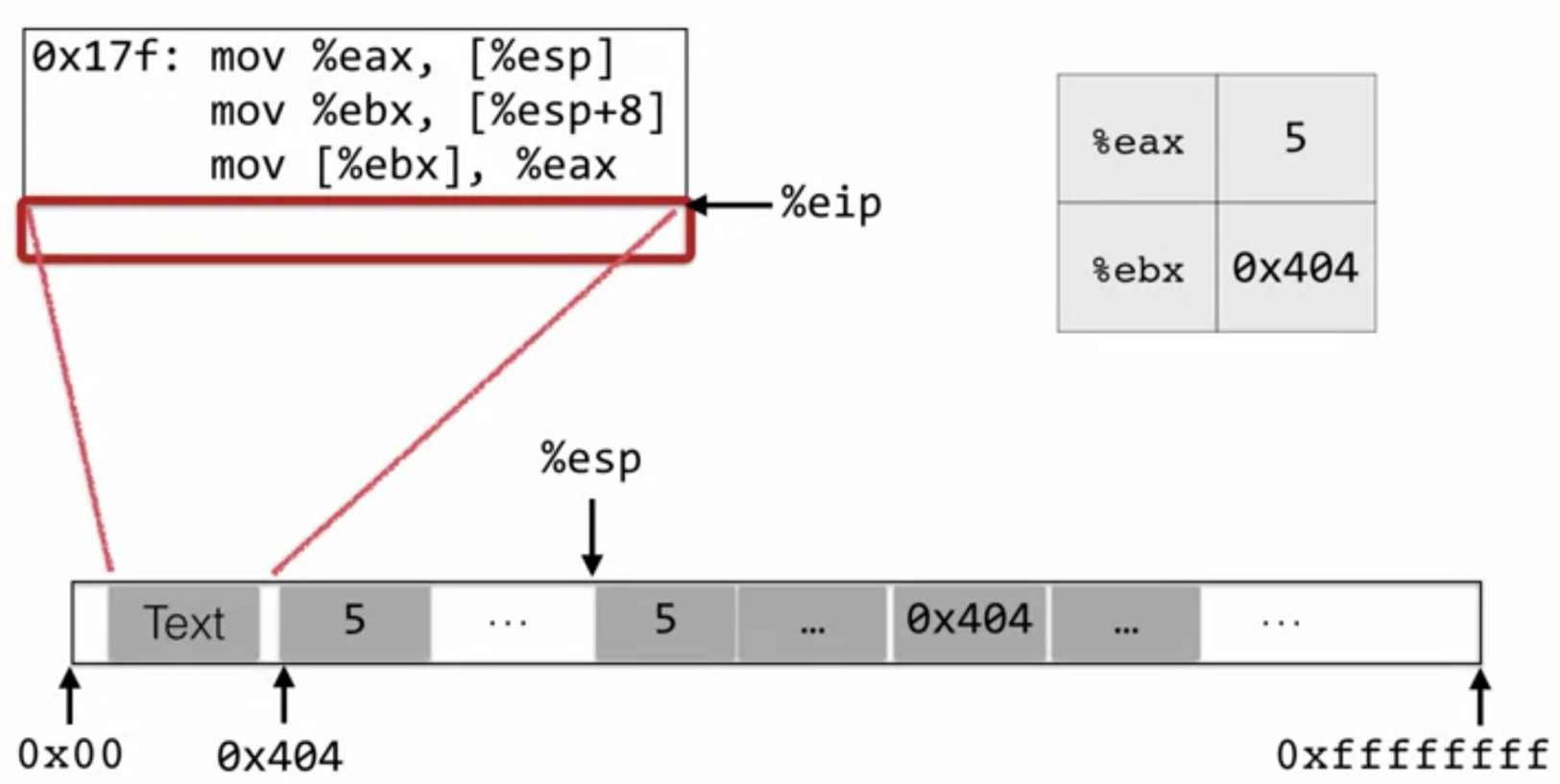
# Another Example – Code Sequen



# Another Example – Code Seque

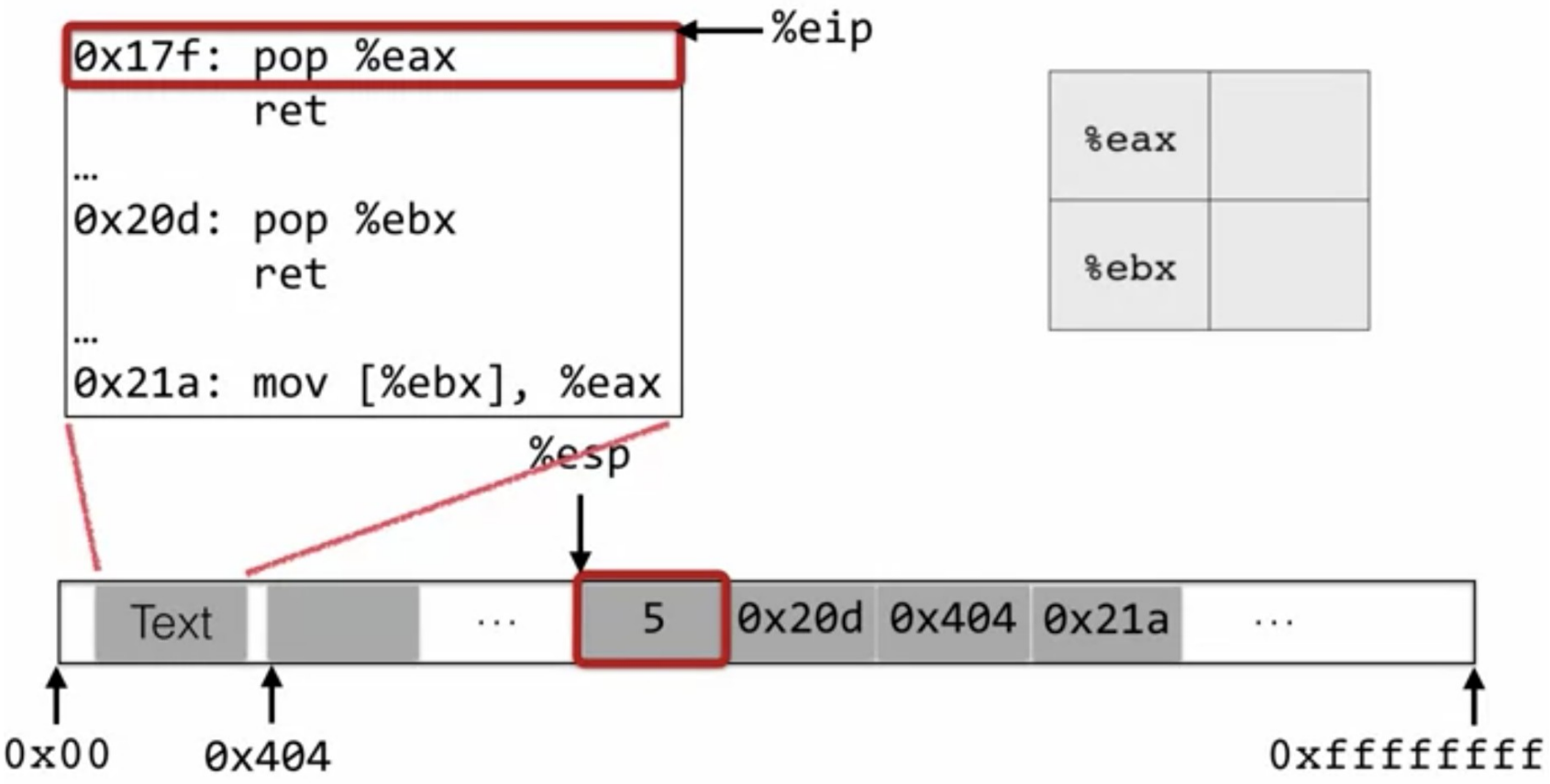


# Another Example – Code Seque

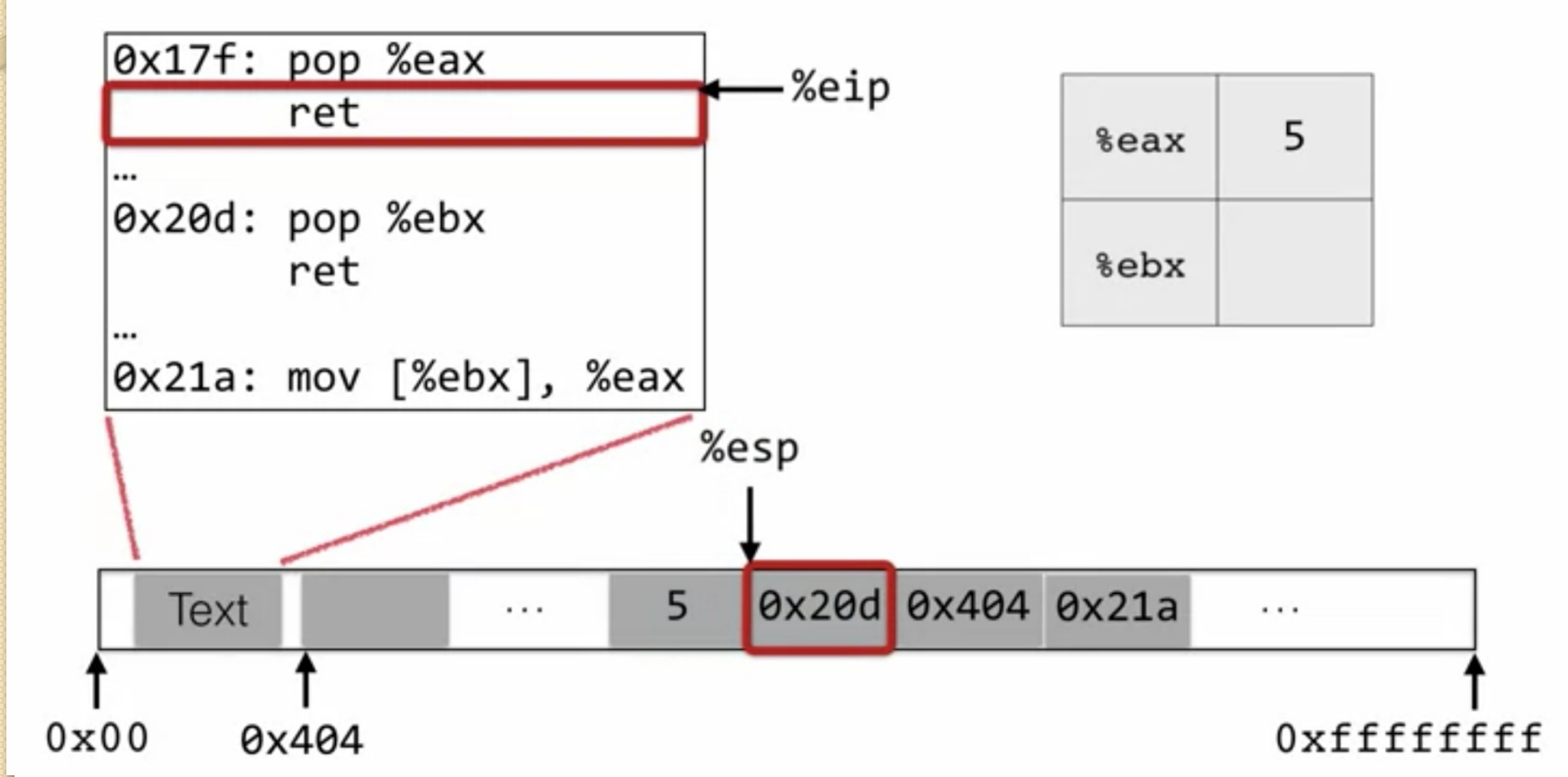


# Another Example – ROP

~~Equivalence~~

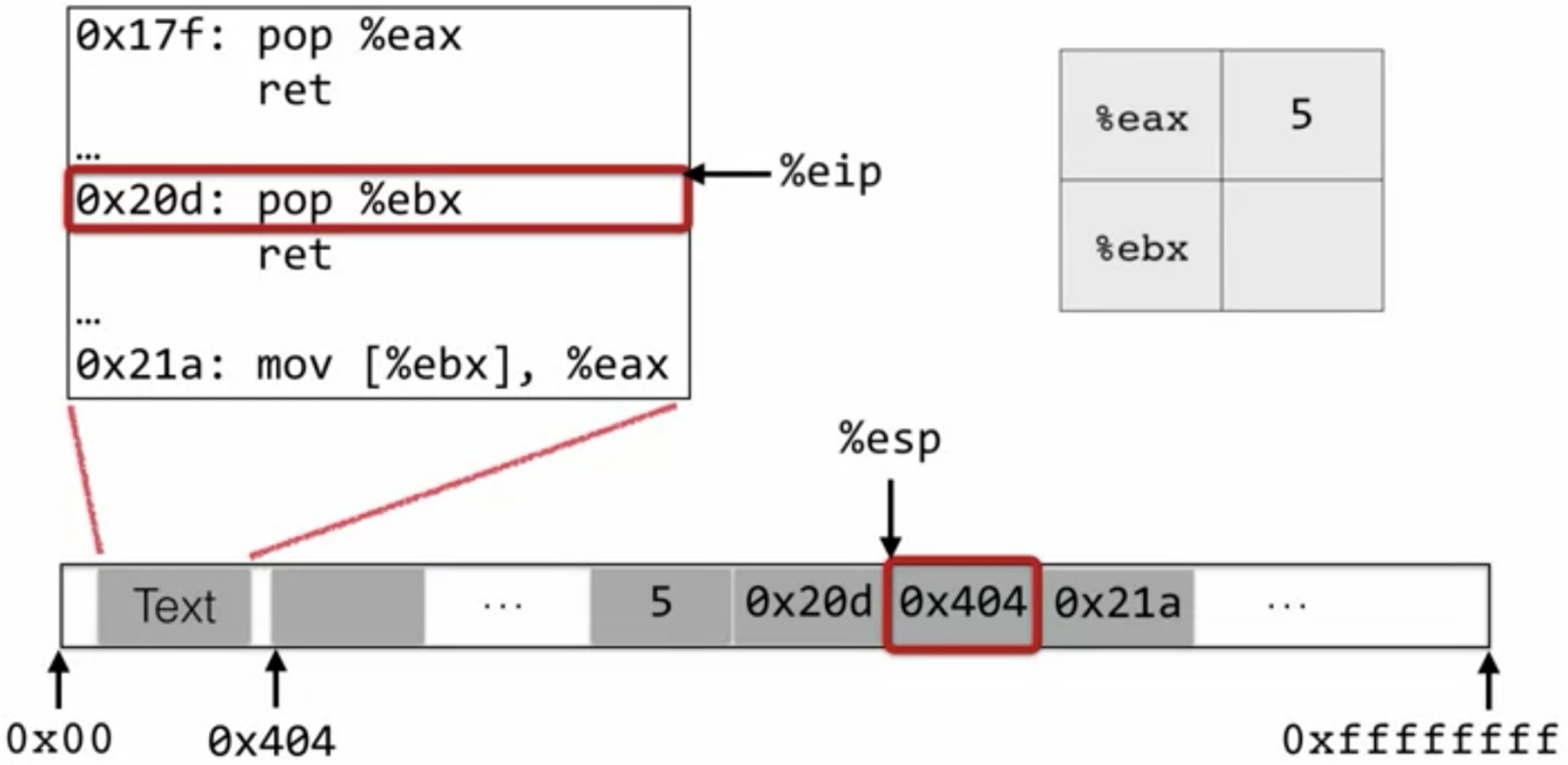


# Another Example – ROP ~~Equivalence~~

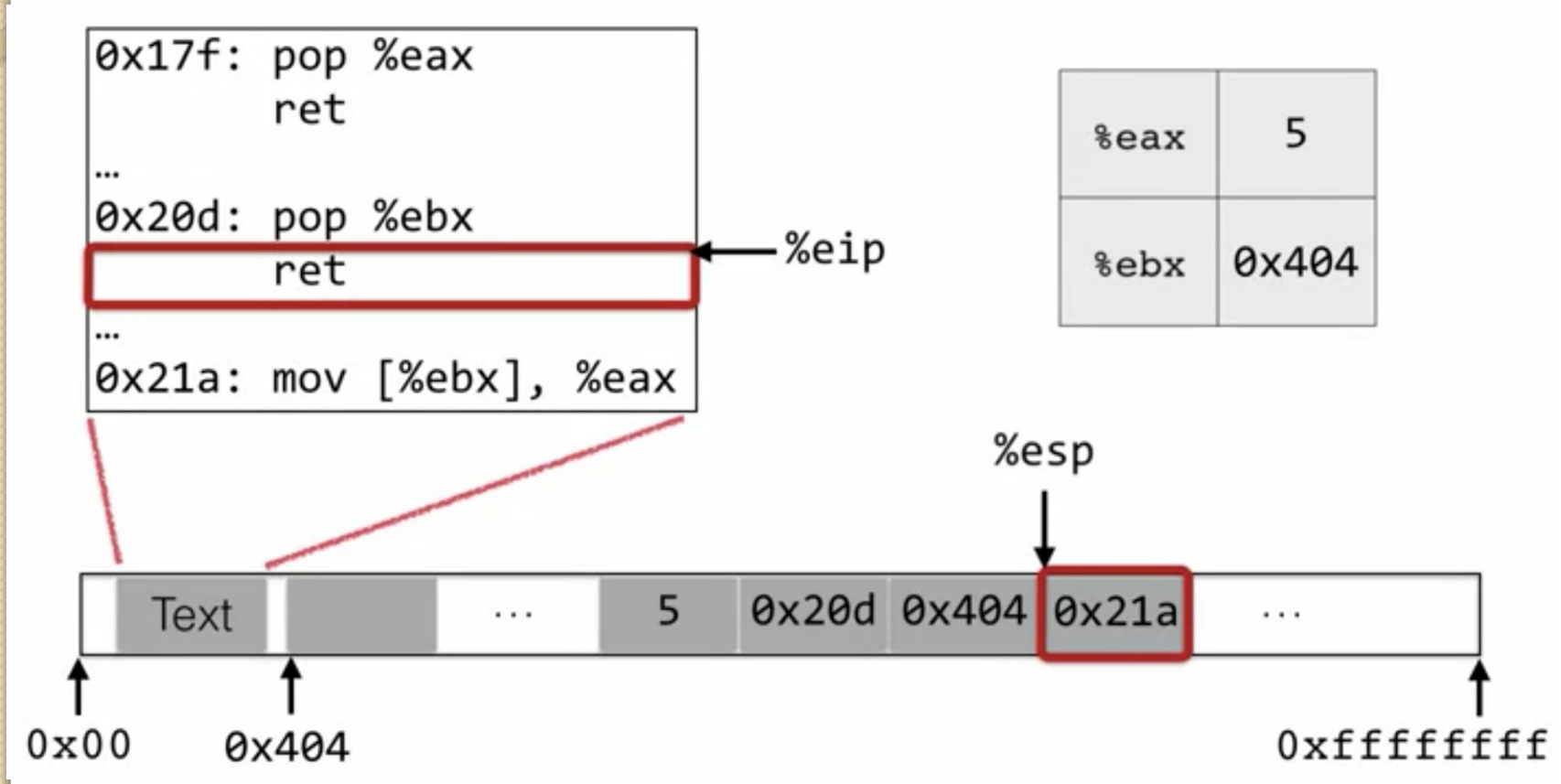


# Another Example – ROP

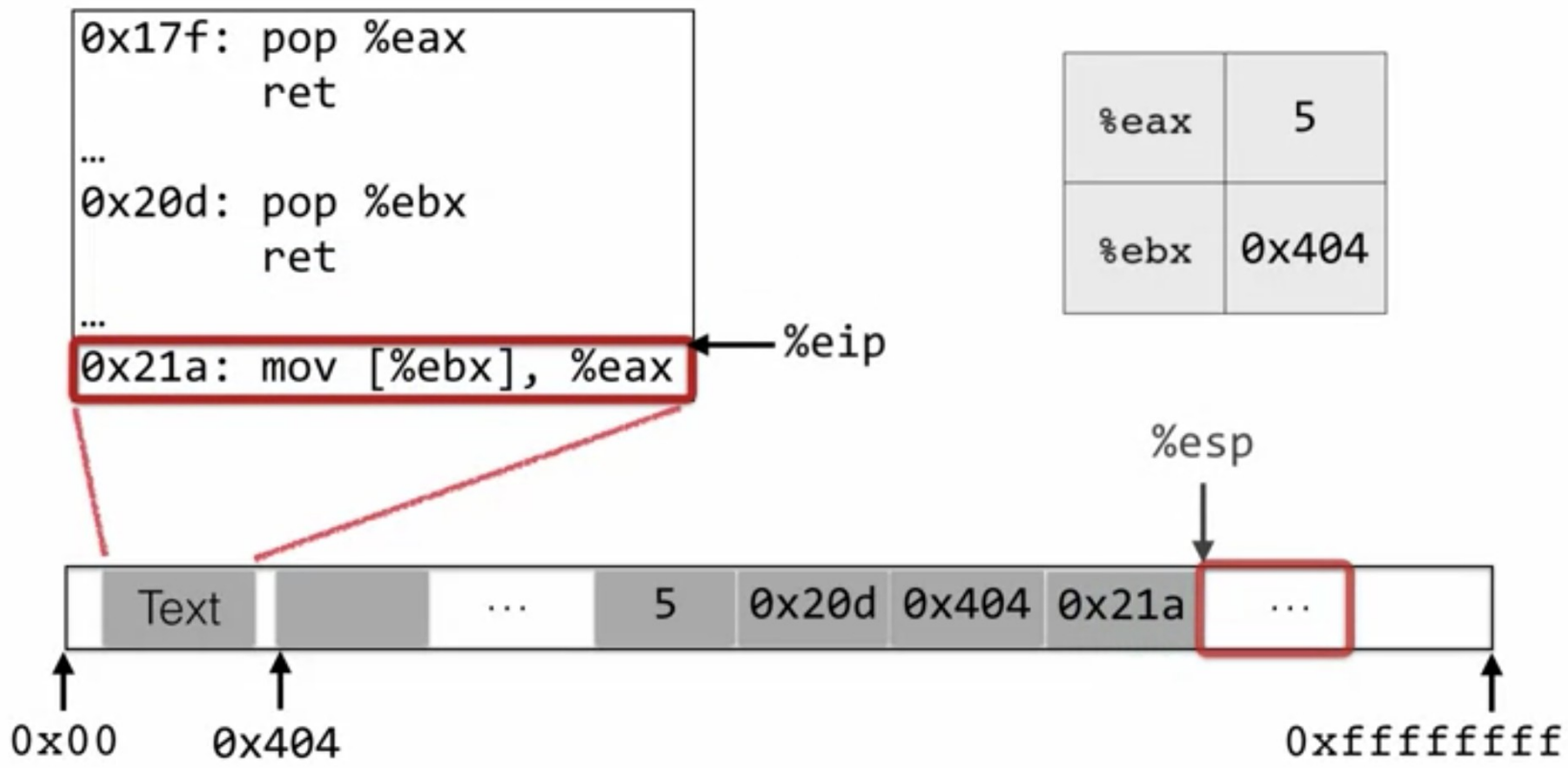
~~Equivalence~~



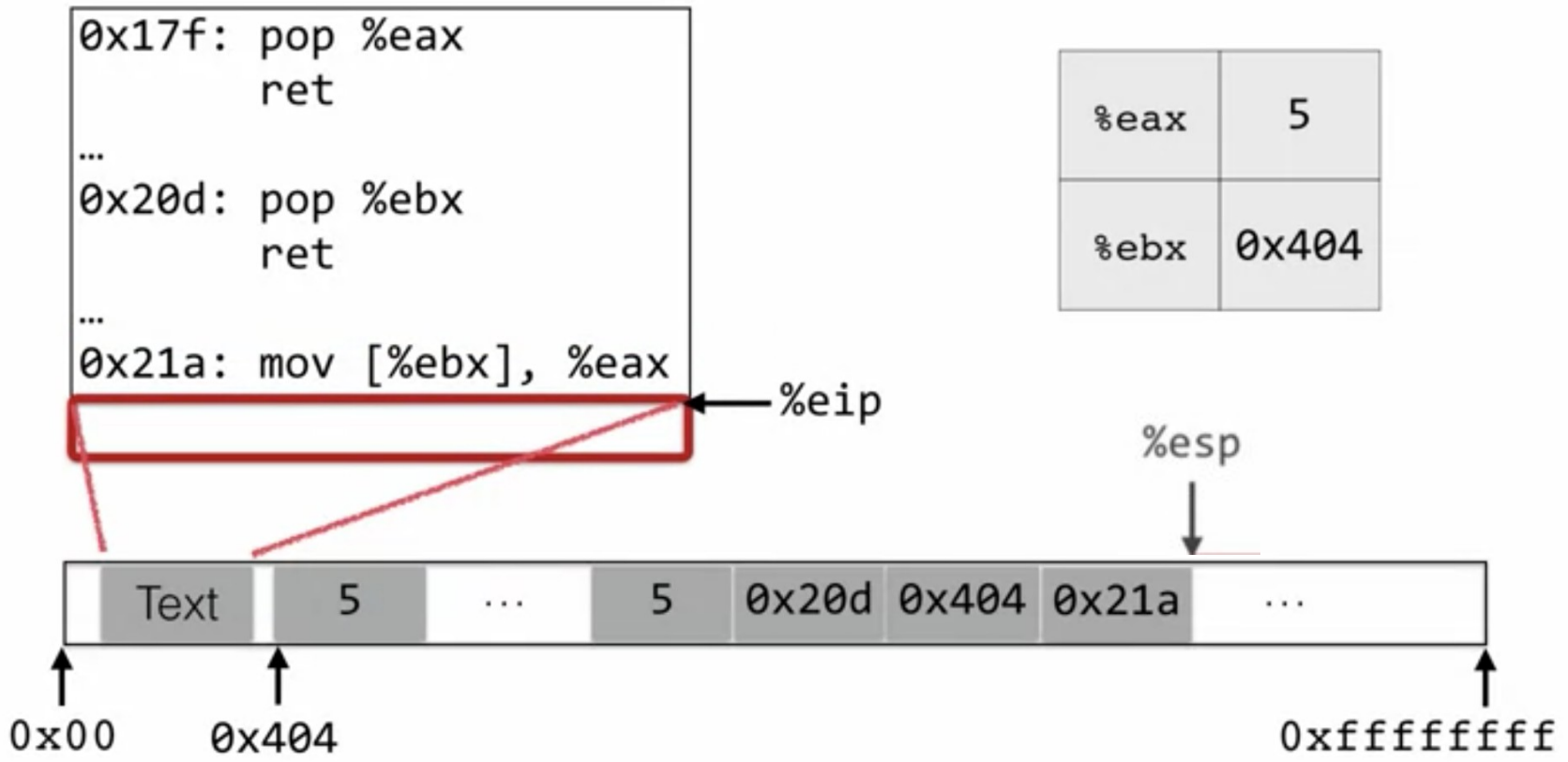
# Another Example – ROP ~~Equivalence~~



# Another Example – ROP Equivalence



# Another Example – ROP Equivalence



# Gadgets and How to Find Them

- Gadgets perform well-defined operations, e.g., load, xor, jump
- Each gadget makes use of one or more sequences of instructions found in text segment and ends with ret
- To find the sequences: first find a ret and store it in the root of a trie, then scan backwards to find children
- The “child-of” relation in the trie means that the child instruction immediately precedes the parent instruction in text segment
- Example: If in a trie, a node representing pop %eax is a child of the root node (ret), then we have discovered the sequence  pop %eax; ret

# Limitation of ROP

- Bypasses W^X
- But not stack canaries or ASLR
- Remains a real remote code injection threat

# More Advanced ROP

- Blind ROP
- PIC ROP

# Control Flow Integrity (CFI)

- Behavior-based detection

[Abadi et al., “Control-Flow Integrity,” (CCS’05)]

- Recall: stack canaries, W^X, and ASLR make one of the steps in stack smashing harder, but may not fully stop the attack
- Idea of CFI: observe the program’s behavior – Is it as expected?
- Challenges:
  - Define “expected behavior”
  - Detect deviations from the behavior efficiently
  - Avoid compromise of detector

# Control Flow Integrity (CFI)

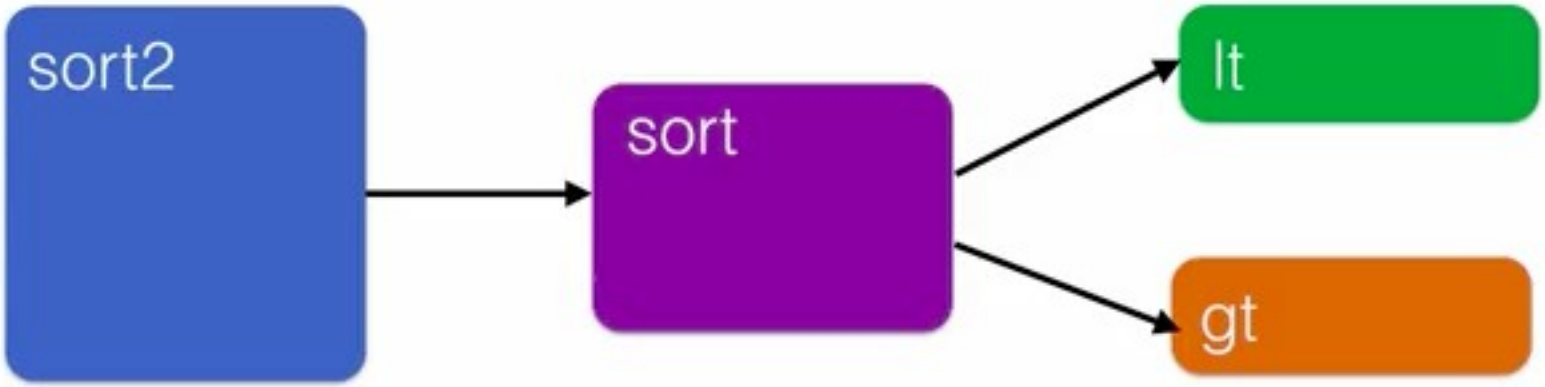
- Define “expected behavior”
  - control flow graph (CFG)
- Detect deviations from the behavior efficiently
  - Inline reference monitor (IRM)
- Avoid compromise of the detector
  - Sufficient randomness of labels; immutable code; non- executable data

# Control Flow Graph (CFG)

- Call graph: which function calls which other functions

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```

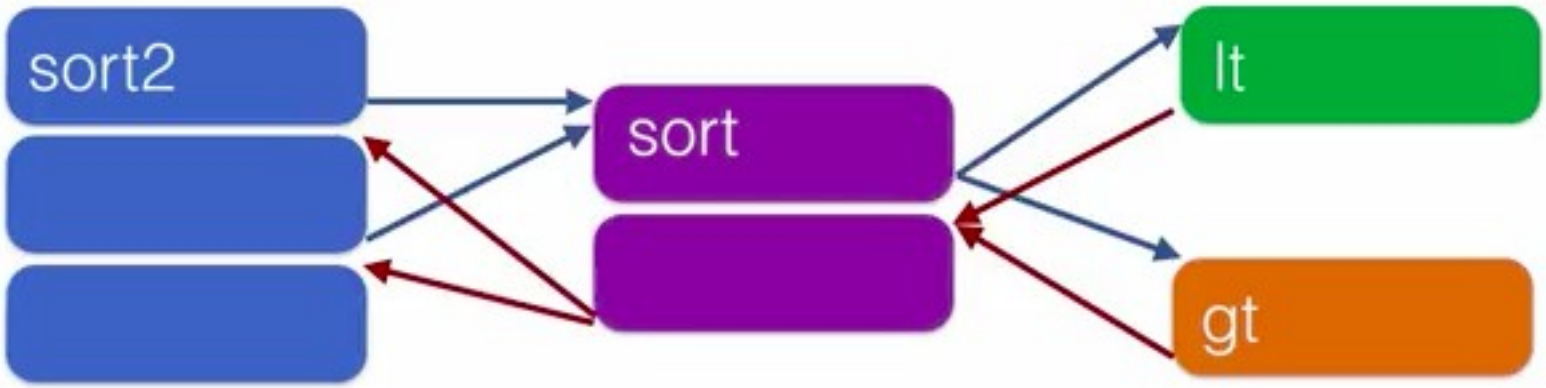


# Control Flow Graph (CFG)

- CFG: breaks into basic blocks of machine code instructions
- Distinguish call from return
- Each block ends with some control transfer such a

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(a, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```

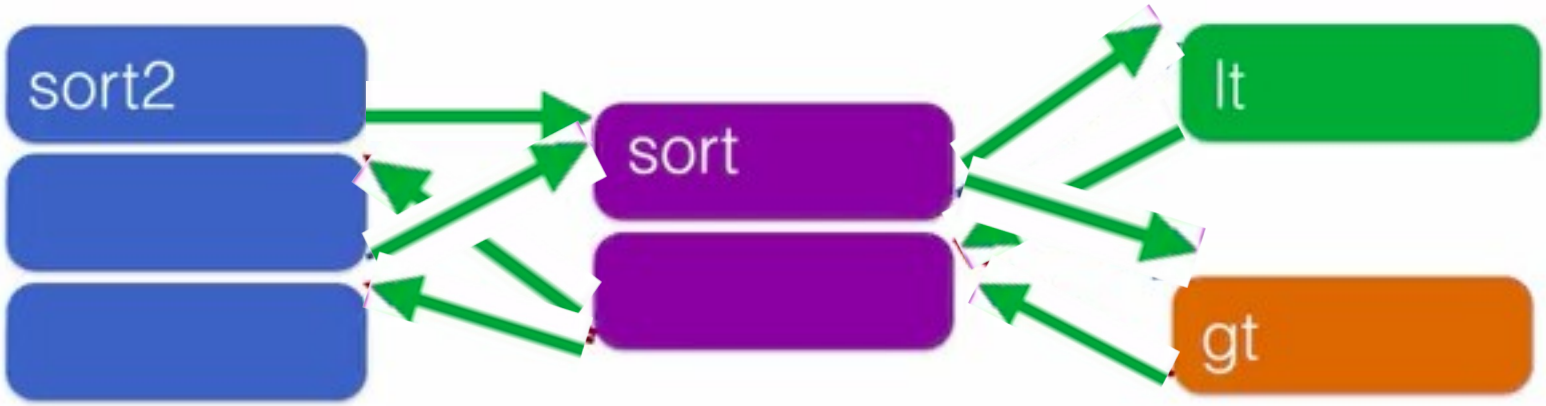


# Control Flow Graph (CFG)

- Computed the call/return CFG in advanced
- Using program analysis (source code or binary)

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(a, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```

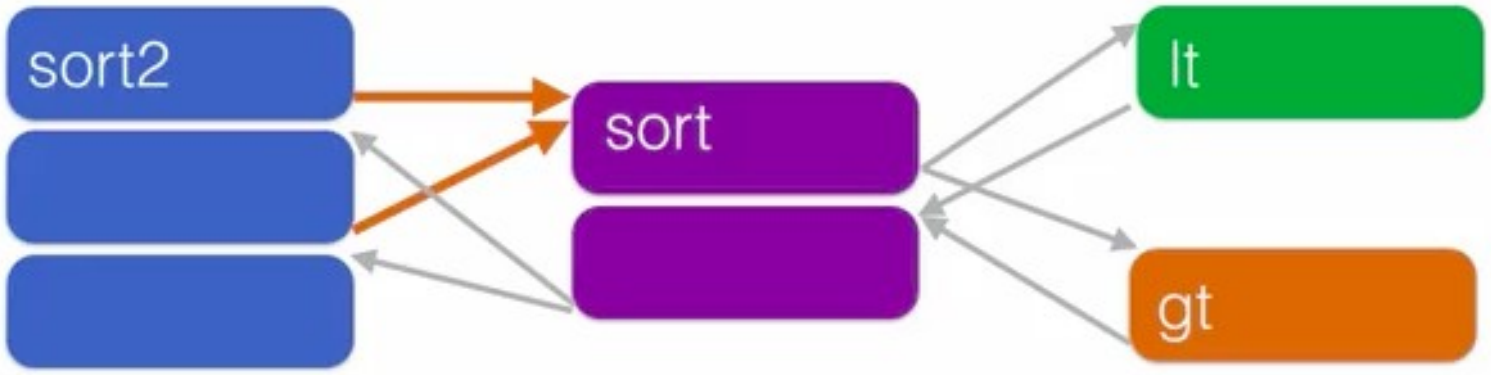


# Inline Reference Monitor (IRM)

- Checks for compliance with CFG
- No need to monitor direct calls: target address cannot be changed assuming code is immutable

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(a, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



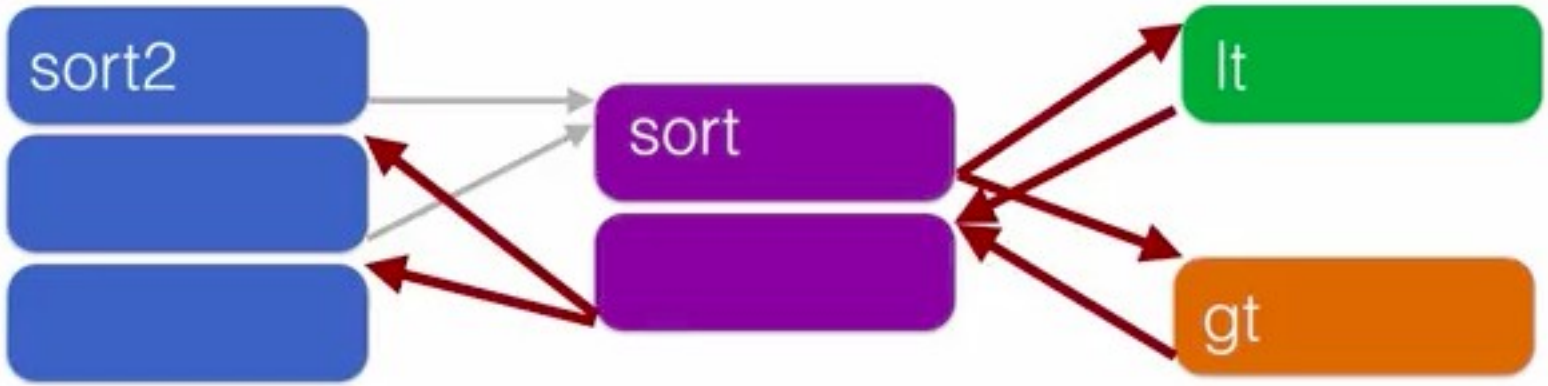

***Direct calls*** *(always the same target)*

# Inline Reference Monitor (IRM)

- Only monitors indirect transfer to improve efficiency: jmp, call via register, return

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(a, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```

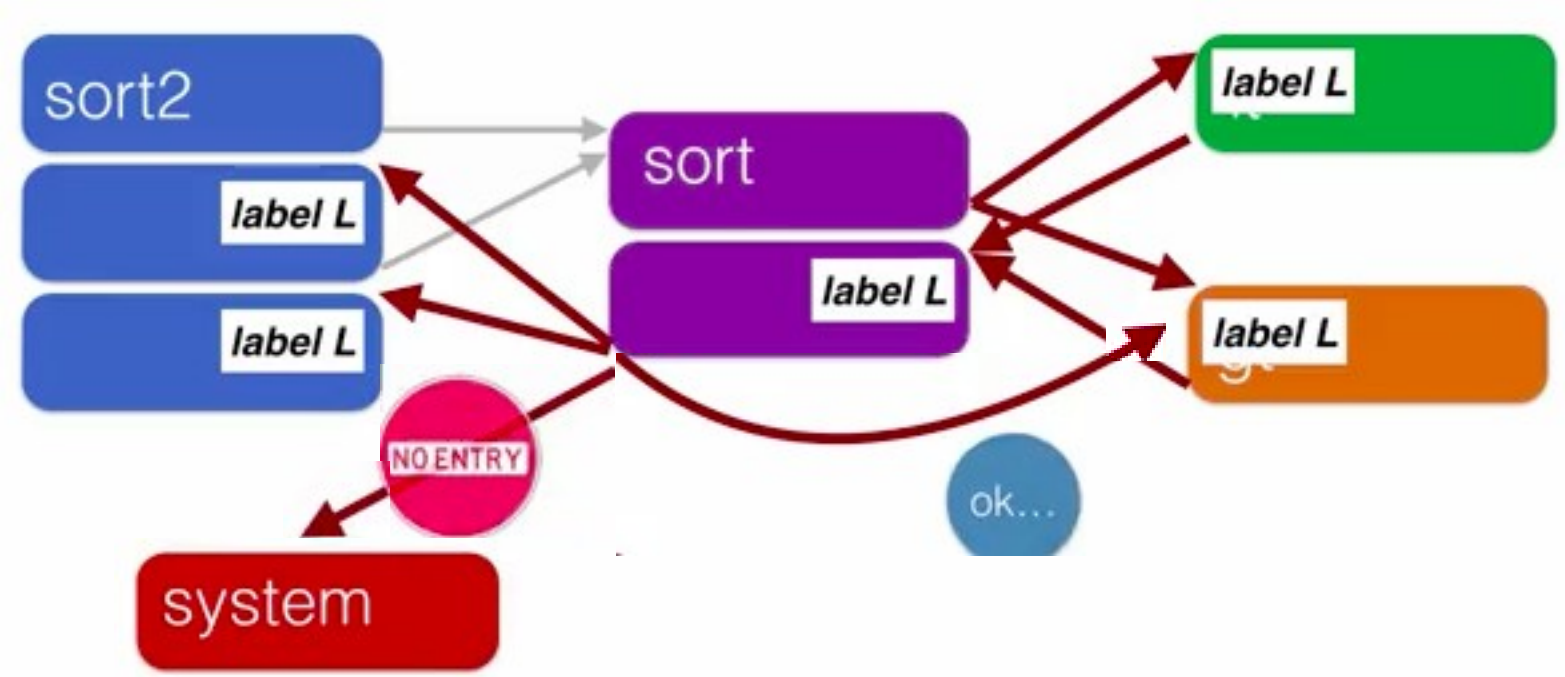



***Indirect transfer*** (`call` *via register, or* `ret`)

# Inline Reference Monitor (IRM)

- Implement IRM inline via machine code rewriting
- Insert a label just before the target address of an indirect transfer
- Insert code to check the label of the target at the source of the indirect transfer

# Simple Labeling

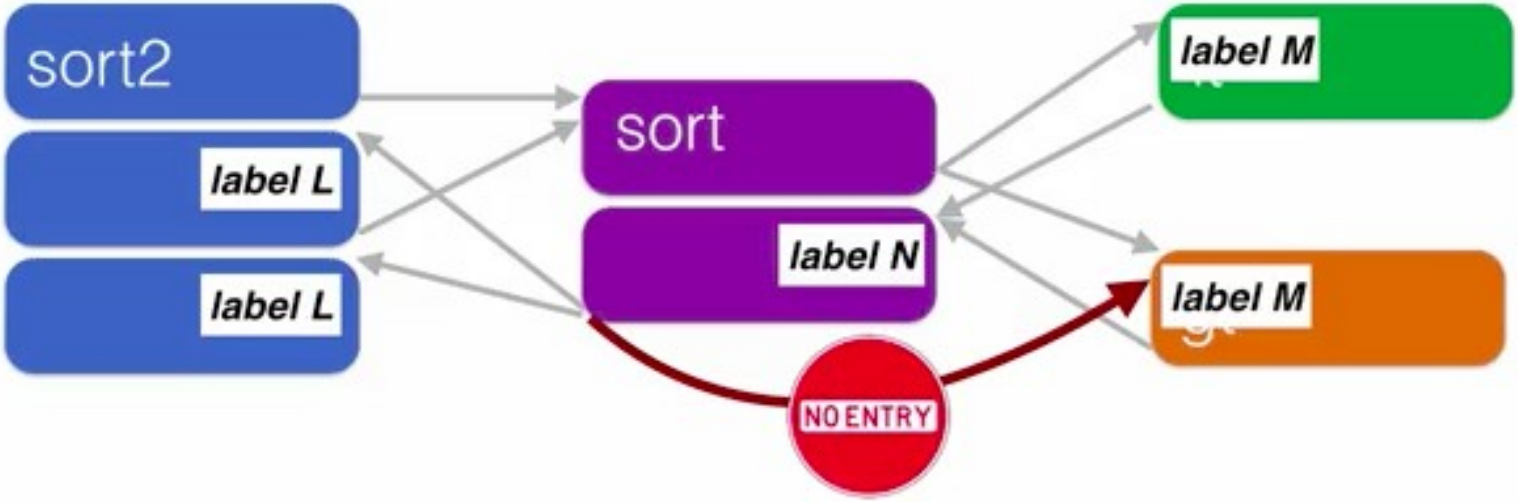



***Use the same label at all targets***

***Blocks return to the start of direct-only call targets***

***but not incorrect ones***

# Detailed Labeling




Constraints:

- return sites from calls to `sort` must share a label (*L*)
- call targets `gt` and `lt` must share a label (*M*)
- remaining label unconstrained (*N*)

# CFI Instrumentation

| Source | | | | Destination | | | |
|---|---|---|---|---|---|---|---|
| Opcode bytes | Instructions | | | Opcode bytes | Instructions | | |
| FF E1 | jmp | ecx | ; computed jump | 8B 44 24 04 | mov | eax, [esp+4] | ; dst |
| | | | | ... | | | |

can be instrumented as (a):

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 81 39 78 56 34 12 | cmp | [ecx], 12345678h | ; comp ID & dst | 78 56 34 12 | ; data 12345678h | | ; ID |
| 75 13 | jne | error_label | ; if != fail | 8B 44 24 04 | mov | eax, [esp+4] | ; dst |
| 8D 49 04 | lea | ecx, [ecx+4] | ; skip ID at dst | ... | | | |
| FF E1 | jmp | ecx | ; jump to dst | | | | |

# Can We Defeat CFI?

- Inject code that has a legal label
  - won't work because we assume non-executable data
- Modify code labels to allow desired control flow
  - won't work because we assume code is immutable
- Modify stack during a check to make it seem to pass
  - won't work because we assume attacker cannot change the registers where we load relevant data (no TOCTOU)

# Performance of CFI

- Overhead due to insertion of labeling and instructions that check for CFI compliance
  - classic CFI: 16% overhead average, 45% worst case
  - modular CFI: 5% average, 12% worst case
- MCFI can eliminate 95.75% of ROP gadgets by ruling their use non-compliant with CFG
- CFI achieves >99% average indirect-target reduction (AIR): percentage of possible targets of indirect jumps CFI rules out

# Limitations of CFI

- CFI defeats control-flow modifying attacks
  - remote code injection; return-to-libc/ROP, etc.
- Does not prevent manipulation of flow allowed by CFG
- Does not prevent data leaks or corruptions (no control flow hijacking)
  - Heartbleed bug
  - “authenticated” flag

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

- But it is very effective against remote code injection and ROP

# Recap: Cat and Mouse

Attack: stack smashing

Defense: stack canaries (e.g., StackGuard), W^X

Attack: guess or bypass stack canaries, return-to-libc/ROP

Defense: ASLR (against return-to-libc/ROP)

Attack: brute force or information leak on ASLR, blin
ROP

Defense: CFI

CFI not mature, not effective against non-code
injection

...

# Defense Aategory 2: Avoid The Flaw Entirely

- Secure coding in C

  - Since the language provides few guarantees, developers must use discipline

  - In general, follow rules (good security practices) for good C coding

  - Good reference guide: CERT C coding standard

- Combined with advanced code review and testing

  - e.g., program analysis, fuzzing

- Use memory-safe languages

  - e.g., avoid C and C++, make C and C++ memory safe

# Rule: Enforce Input Compliance

```
int main() {
  char buf[100], *p;
  int i, len;
  while (1) {
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   len = atoi(p);
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   for (i=0; i<len; i++)
    if (!iscntrl(buf[i])) putchar(buf[i]);
    else putchar('.');
   printf("\n");
 }}
```

***Recall earlier example***

- Read integer
- Read message
- Echo back (partial) message

# Rule: Enforce Input Compliance

```
int main() {
  char buf[100], *p;
  int i, len;
  while (1) {
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   len = atoi(p);
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   for (i=0; i<len; i++)
    if (!iscntrl(buf[i])) putchar(buf[i]);
    else putchar('.');
   printf("\n");
}}
```

***May exceed actual message length!***

***Recall earlier example***

- Read integer
- Read message
- Echo back (partial) message

# Rule: Enforce Input Compliance

```
int main() {
  char buf[100], *p;
  int i, len;
  while (1) {
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   len = atoi(p);
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   len = MIN(len,strlen(buf));
   for (i=0; i<len; i++)
    if (!iscntrl(buf[i])) putchar(buf[i]);
    else putchar('.');
   printf("\n");
}}
```

***Recall earlier example***

**Sanitizes** input to be compliant

# Rule: Enforce Input Compliance

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


}
```

# Rule: Enforce Input Compliance

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

- Unfounded trust in received input is a recurring source of vulnerability

# Rule: Use Safe String Functions

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

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

# 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`, …

# Rule: Don’t Forget NUL Terminator

```
char str[3];
strcpy(str,"bye"); // write overflow
int x = strlen(str); // read overflow
```

```
char str[3];
strlcpy(str,"bye",3); // blocked
int x = strlen(str); // returns 2
```

# Rule: Understand Pointer Arithmetic

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

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

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

# (Better) Rule: Favor Safe Libraries

- Libraries encapsulates well-thought-out design
- Networking: Google protocol buffers, Apache Thrift

  - for dealing with network transmitted data

  - ensure input validation, parsing, etc.

  - efficient

# Defense Summary

- Defense category 1 and 2 are complementary

# Other Programming Oversights

- TOCTOU
- Off-by-one error
- Undocumented entry points in programs
- …
- Not all programming oversights have security implications

# TOCTOU (Race Condition)

- Time-of-Check to Time-of-Use
  - Between access check and use (race window), data can be altered

```
function withdraw($amount)
{
    $balance = getBalance();
    if($amount <= $balance) {
        $balance = $balance - $amount;
        echo "You have withdrawn: $amount";
        saveBalance($balance);
    }
    else {
        echo "Insufficient funds.";
    }
}
```

# TOCTOU (Race Condition)

- Time-of-Check to Time-of-Use
  - Between access check and use (race window), data can be altered

```
void silly_function(char *pathname) {
  struct stat f, we;
  int rfd, wfd;
  char *buf;
  stat(pathname, &f);
  stat("/what/ever", &we);
  if (f.st_dev == we.st_dev && f.st_ino == we.st_ino) {
    return;
  }
  rfd = open(pathname, O_RDONLY);
  buf = malloc(f.st_size - 1);
  read(rfd, buf, f.st_size - 1);
  close(rfd);

  stat(pathname, &f);
  if (f.st_dev == we.st_dev && f.st_ino == we.st_ino) {
    return;
  }
  wfd = open(pathname, O_WRONLY | O_TRUNC);
  write(wfd, buf, f.st_size - 1);
  close(wfd);
  free(buf);
}
```