CSE467: Computer Security

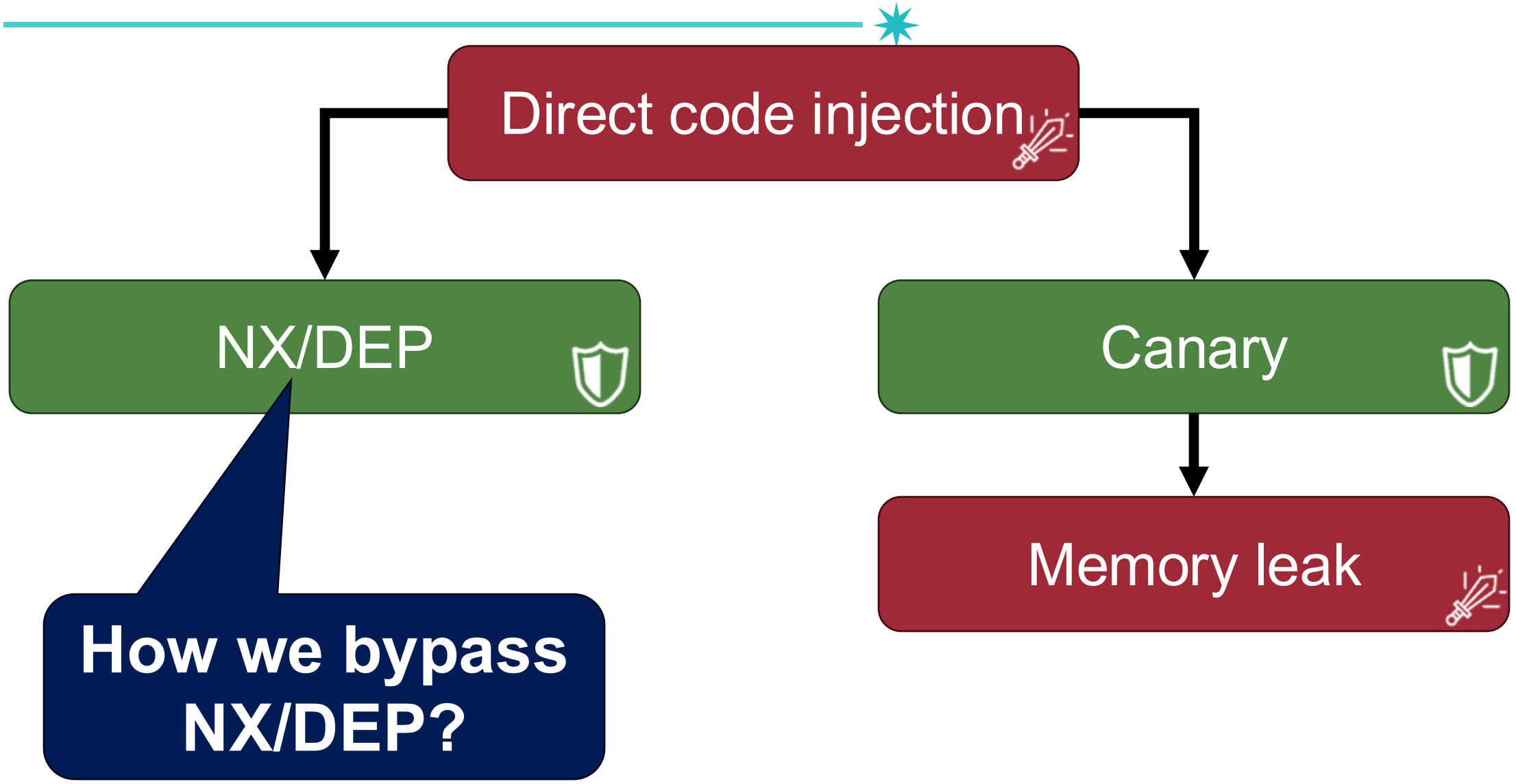
16. ROP

Seongil Wi

Department of Computer Science and Engineering

The slide is based on Prof. Sang Kil Cha's lecture slide

Motivation

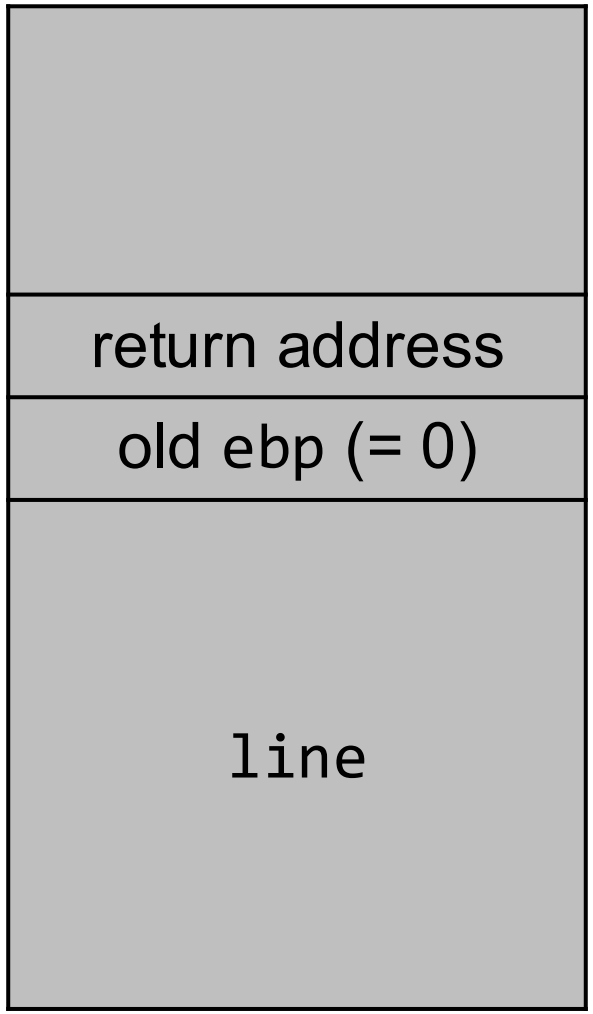

Direct code injection
NX/DEP
Canary
Memory leak
How we bypass NX/DEP?

Code-Reuse Attacks

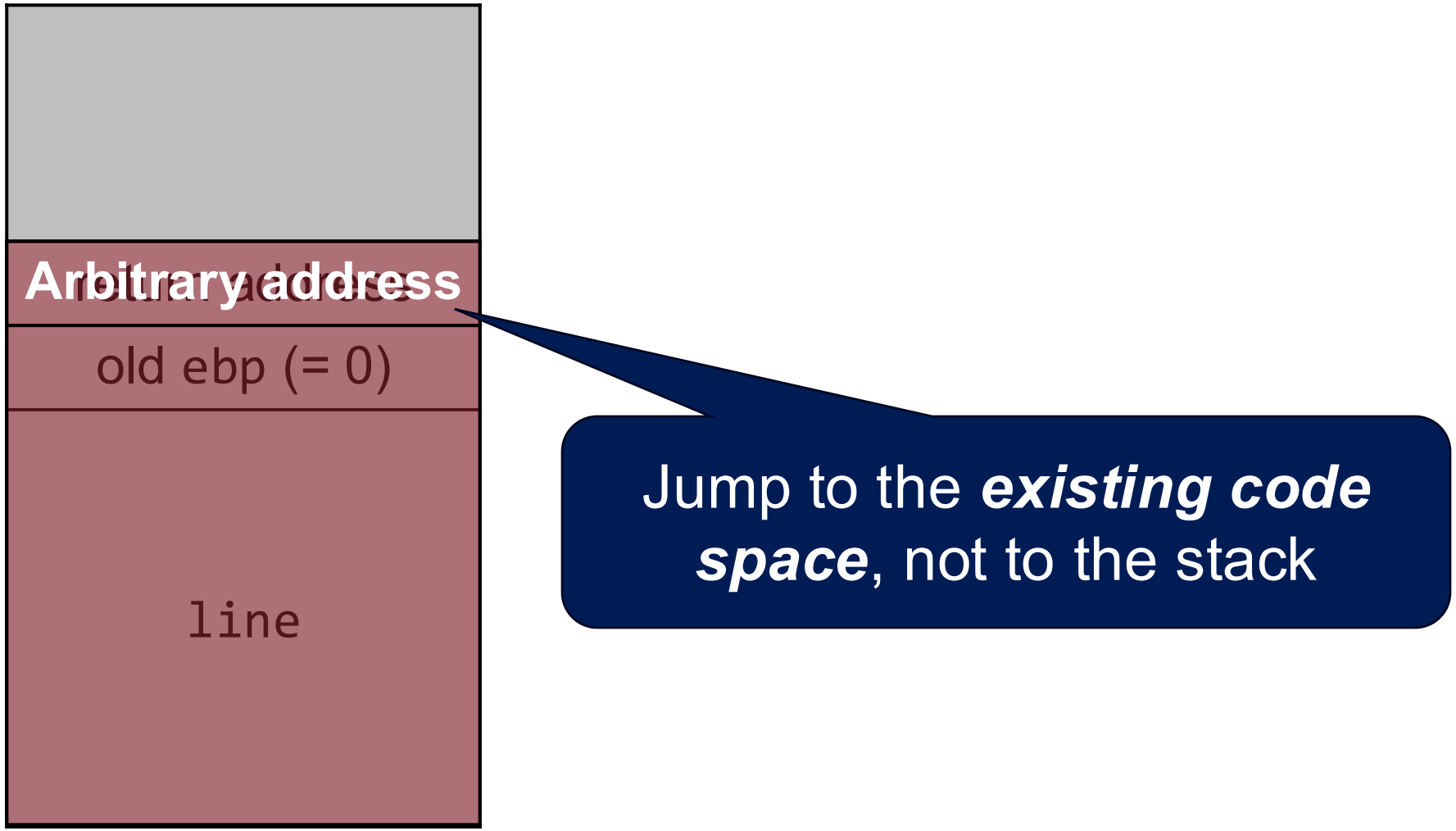
Bypassing DEP

- Return-to-stack exploit is disabled
- But, we can still jump to an arbitrary address of ***existing code*** **(= *Code Reuse Attack*)**

Main Idea: Jump to Existing Code



Main Idea: Jump to Existing Code

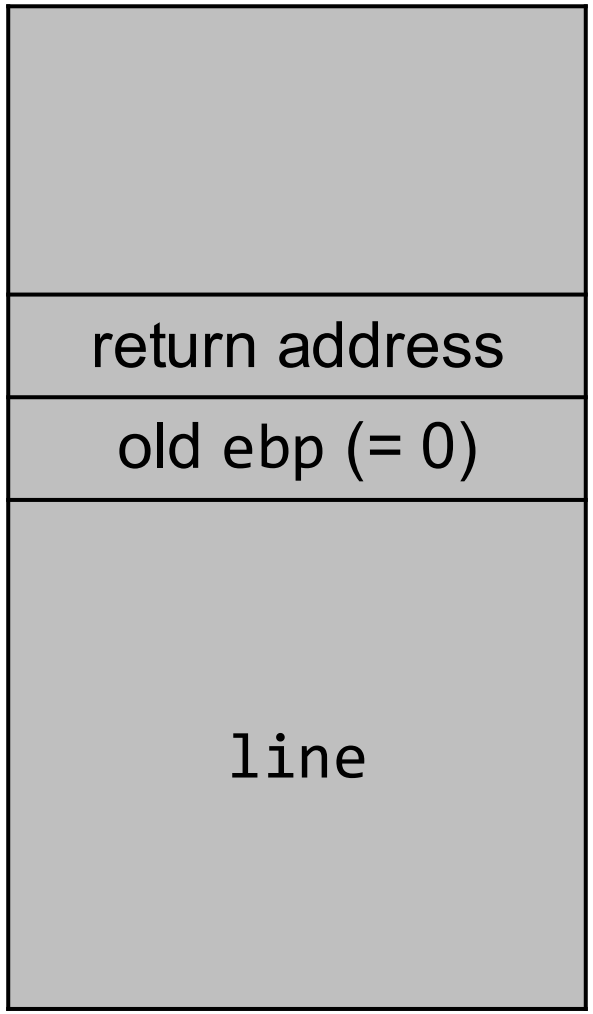


Code Reuse Attack #1: Return-to-Libc

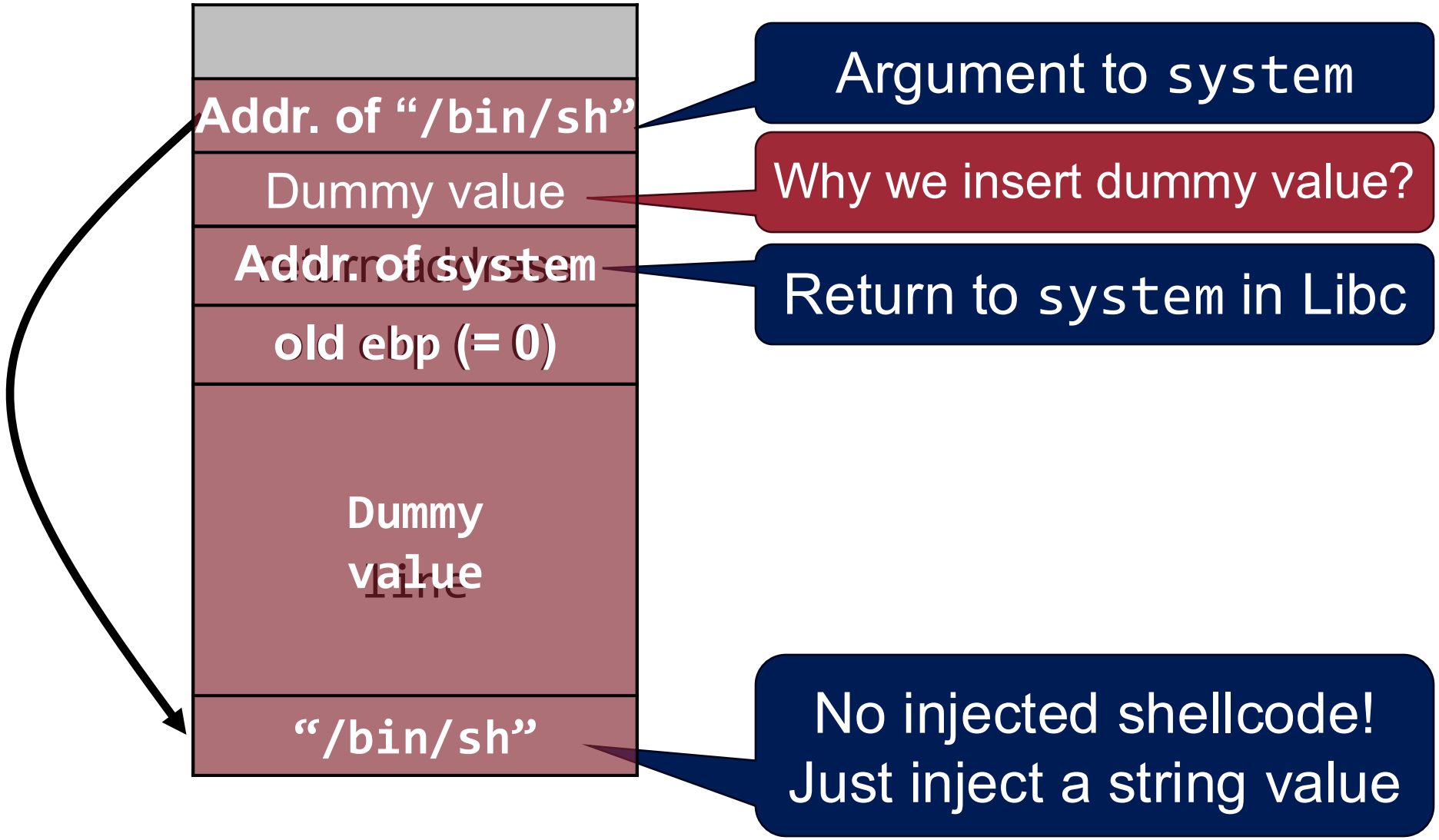
- LIBC (LIBrary C) is a standard library that most programs commonly use
  - For example, `printf` is in LIBC

- Many useful functions in LIBC to execute
  - `exec` family: `execl, execlp, execle, …`
  - `system`
  - `mprotect`
  - `mmap`

Code Reuse Attack #1: Return-to-Libc



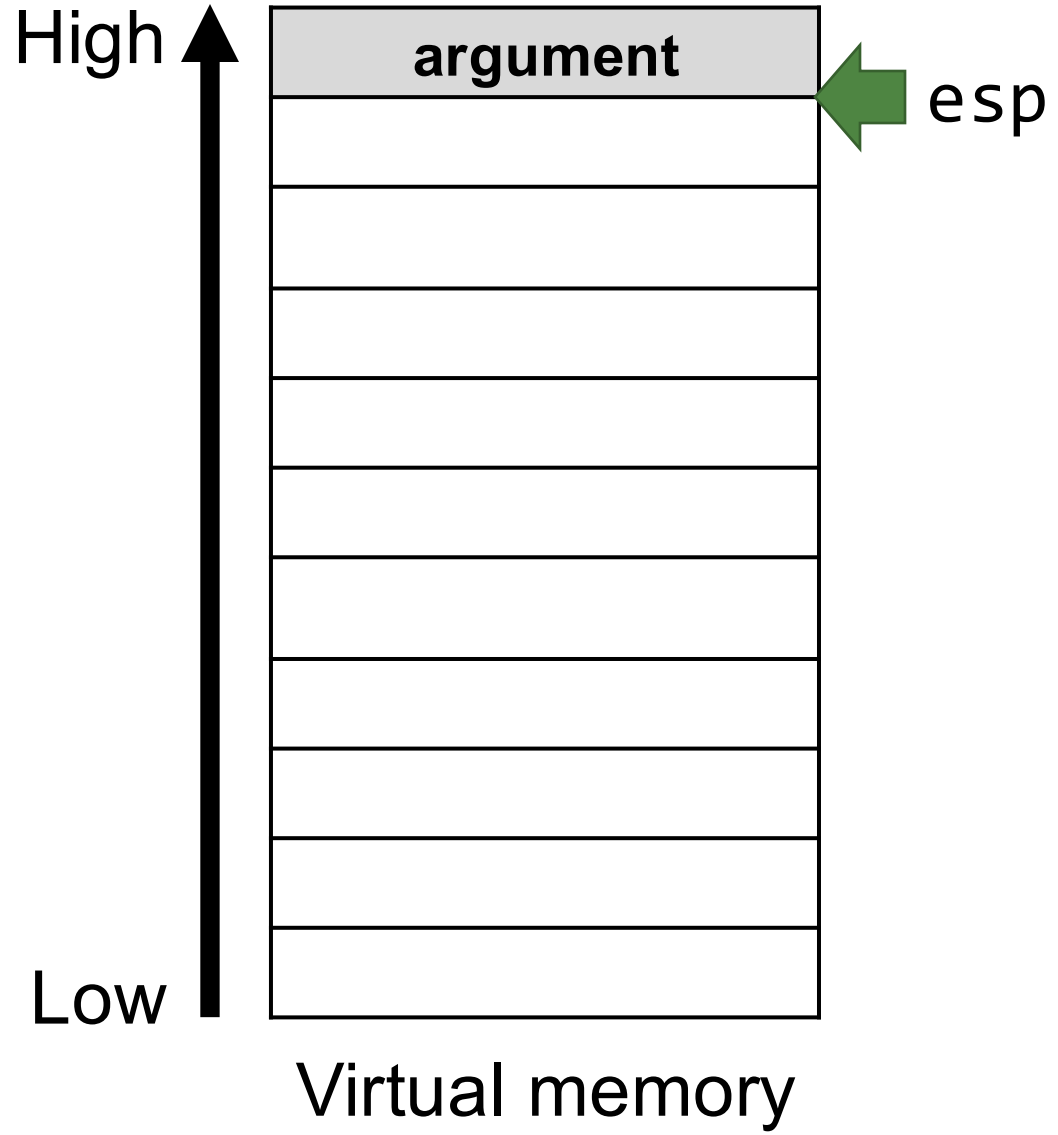
Code Reuse Attack #1: Return-to-Libc



Recap: Function Call (call)

```
    …
    call system
nextret:
    nop
    nop
    ...
system (in libc):
    nop
    nop
```

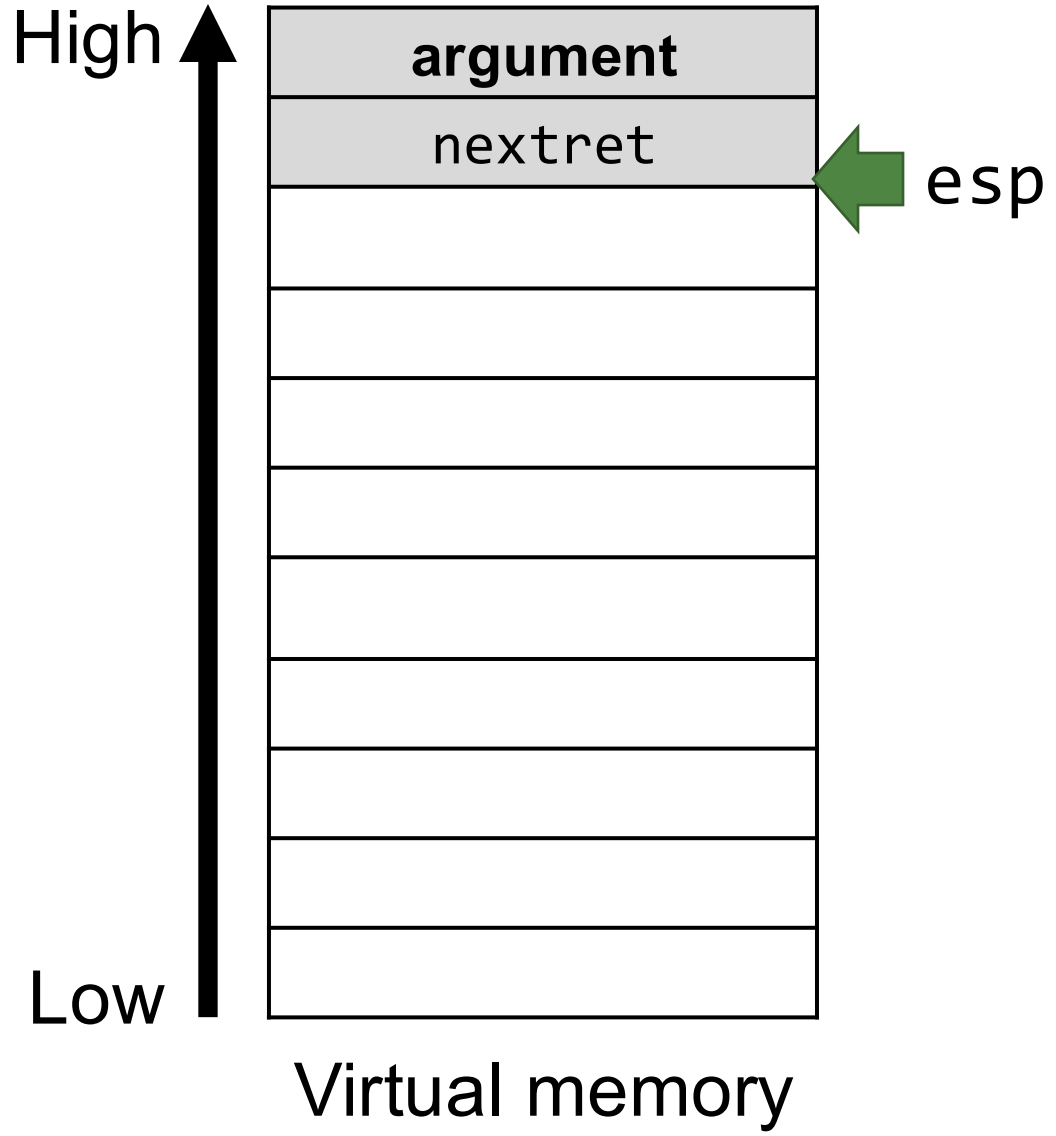
push nextret
jmp system



Recap: Function Call (call)

```
    …
    call system
nextret:
    nop
    nop
    …
system (in libc):
    nop
    nop
```

```
push nextret
jmp system
```

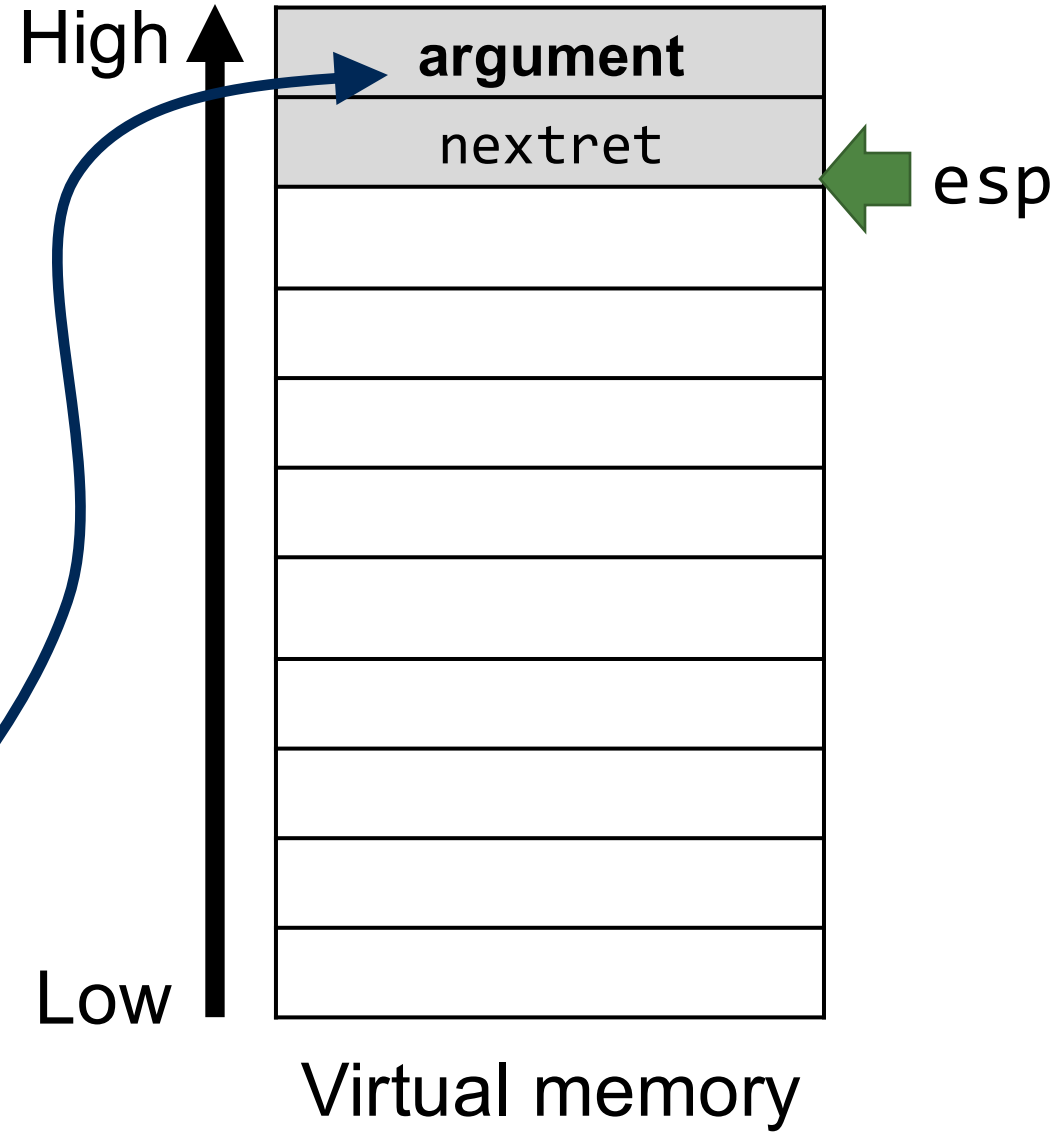


Recap: Function Call (call)


```
    …
    call system
nextret:
    nop
    nop
    …
system (in libc):
    nop
    nop
```

push nextret
jmp system

Recognize esp+0x4
as first parameter



Recap: LIBC provides System Call Wrapper


```
08049162 <main>:
 8049162: 55                            push   ebp
 8049163: 89 e5                         mov    ebp,esp
 8049165: 83 ec 08                      sub    esp,0x8
 8049168: c7 45 f8 08 a0 04 08          mov    DWORD PTR [ebp-0x8],0x804a008
 804916f: c7 45 fc 00 00 00 00          mov    DWORD PTR [ebp-0x4],0x0
 8049176: 6a 00                         push   0x0
 8049178: 8d 45 f8                      lea    eax,[e
 804917b: 50                            push   eax
 804917c: 68 08 a0 04 08                push   0x804a008
 8049181: e8 c7 29 02 00                call   806c4b0 <__execve>
```

First argument of execve

You are actually calling a wrapper function around the syscall

Recap: LIBC provides System Call Wrapper

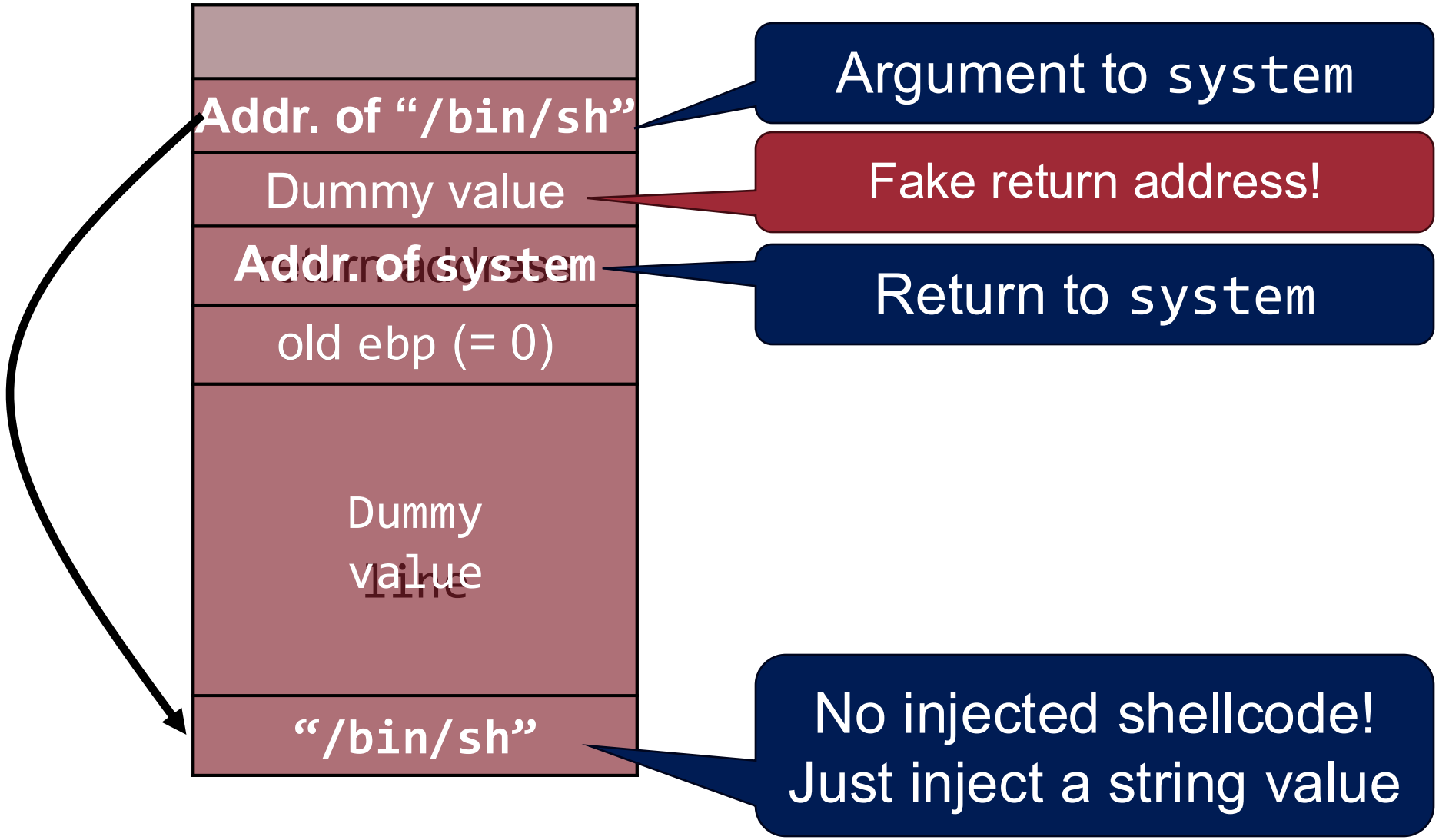
LIBC code

```
0806c4b0 <__execve>:
 806c4b0:   53                      push   ebx
 806c4b1:   8b 54 24 10             mov    edx,DWORD PTR [esp+0x10]
 806c4b5:   8b 4c 24 0c             mov    ecx,DWORD PTR [esp+0xc]
 806c4b9:   8b 5c 24 08             mov    ebx,DWORD PTR [esp+0x8]
 806c4bd:   b8 0b 00 00 00          mov    eax,0xb
 806c4c2:   cd 80                   int    0x80
```

Get first argument

System Call!

Code Reuse Attack #1: Return-to-Libc



Motivation of Return-oriented Programming

Return-to-LIBC requires LIBC function calls, but …☹

- Different versions of LIBC

```
attacker_local@environment:~$ ldd --version
ldd (Ubuntu GLIBC 2.31-0ubuntu9.17) 2.31
```

```
victim@environment:/# ldd --version
ldd (Ubuntu GLIBC 2.27-3ubuntu1) 2.27
```

Motivation of Return-oriented Programming

Return-to-LIBC requires LIBC function calls, but …☹

- Different versions of LIBC
- LIBC may not be used at all
- Some functions in LIBC can be excluded

```
attacker_local@environment:~$ ldd --version
ldd (Ubuntu GLIBC 2.31-0ubuntu9.17) 2.31
```

```
victim@environment:/# ldd --version
ldd (Ubuntu GLIBC 2.27-3ubuntu1) 2.27
```

Motivation of Return-oriented Programming

Return-to-LIBC requires LIBC function calls, but …☹

- Different versions of LIBC
- LIBC may not be used at all
- Some functions in LIBC can be excluded

```
attacker_local@environment:~$ ldd --version
ldd (Ubuntu GLIBC 2.31-0ubuntu9.17) 2.31
```

Can we spawn a shell without the use of LIBC functions?

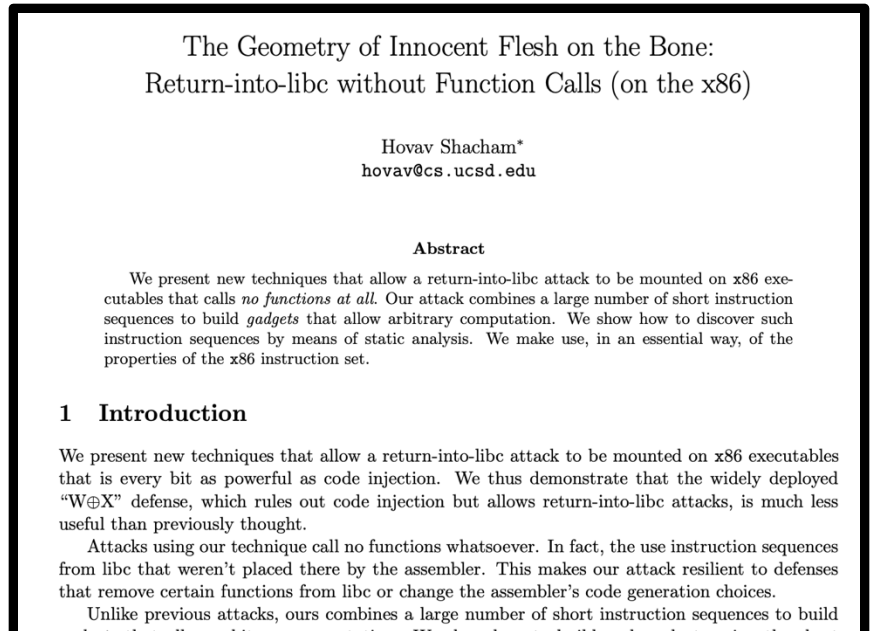
Return-oriented Programming (ROP)

Code Reuse Attack #2: ROP

Generalized Code Reuse Attack

Formally introduced by Hovav in CCS 2007

"The Geometry of Innocent Flesh on the Bone: Return-to-libc without Function Calls (on the x86)"



The Geometry of Innocent Flesh on the Bone:
Return-into-libc without Function Calls (on the x86)

Hovav Shacham*
hovav@cs.ucsd.edu

Abstract

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls *no functions at all*. Our attack combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set.

1 Introduction

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that is every bit as powerful as code injection. We thus demonstrate that the widely deployed "W⊕X" defense, which rules out code injection but allows return-into-libc attacks, is much less useful than previously thought.

Attacks using our technique call no functions whatsoever. In fact, the use instruction sequences from libc that weren't placed there by the assembler. This makes our attack resilient to defenses that remove certain functions from libc or change the assembler's code generation choices.

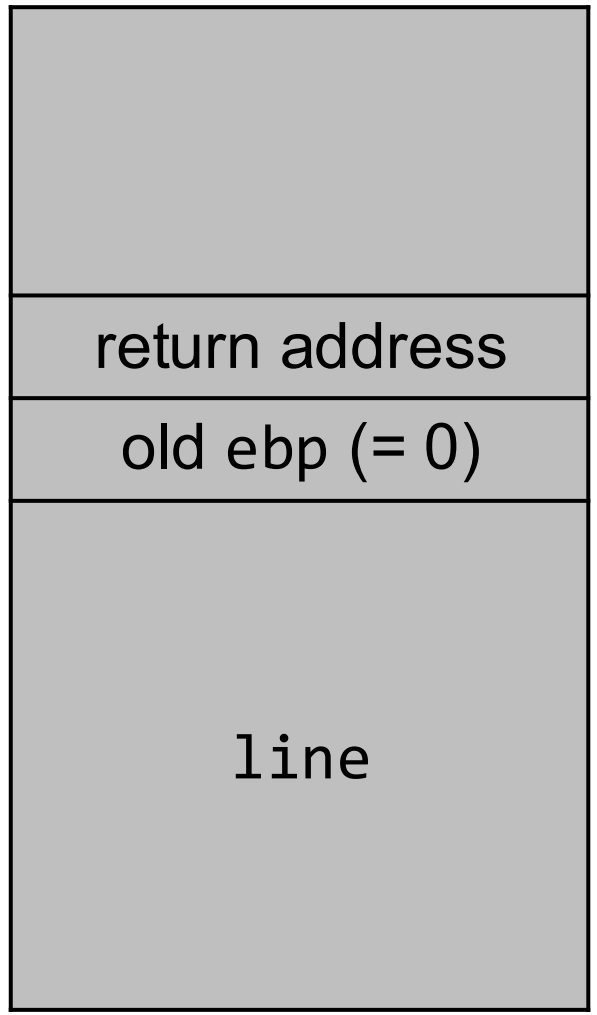
Unlike previous attacks, ours combines a large number of short instruction sequences to build

Main Idea: Return (`ret`) Chaining

Attacker's goal:

execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```

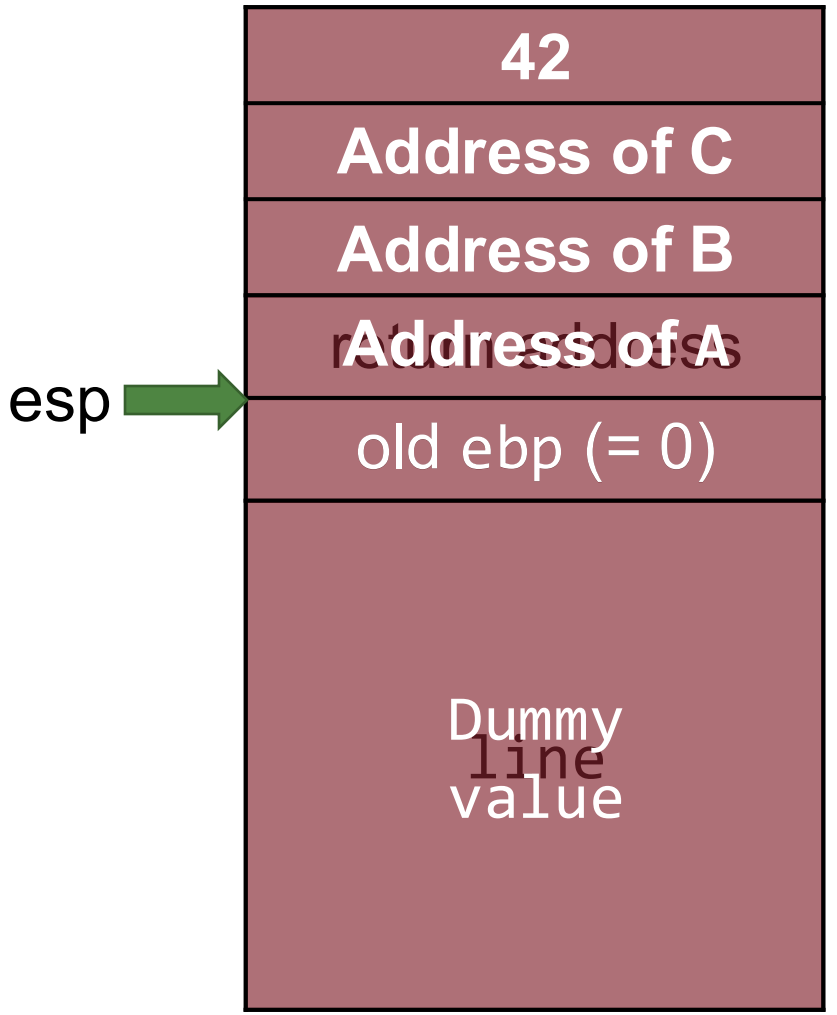


Main Idea: Return (ret) Chaining

Attacker's goal:

execute following instructions

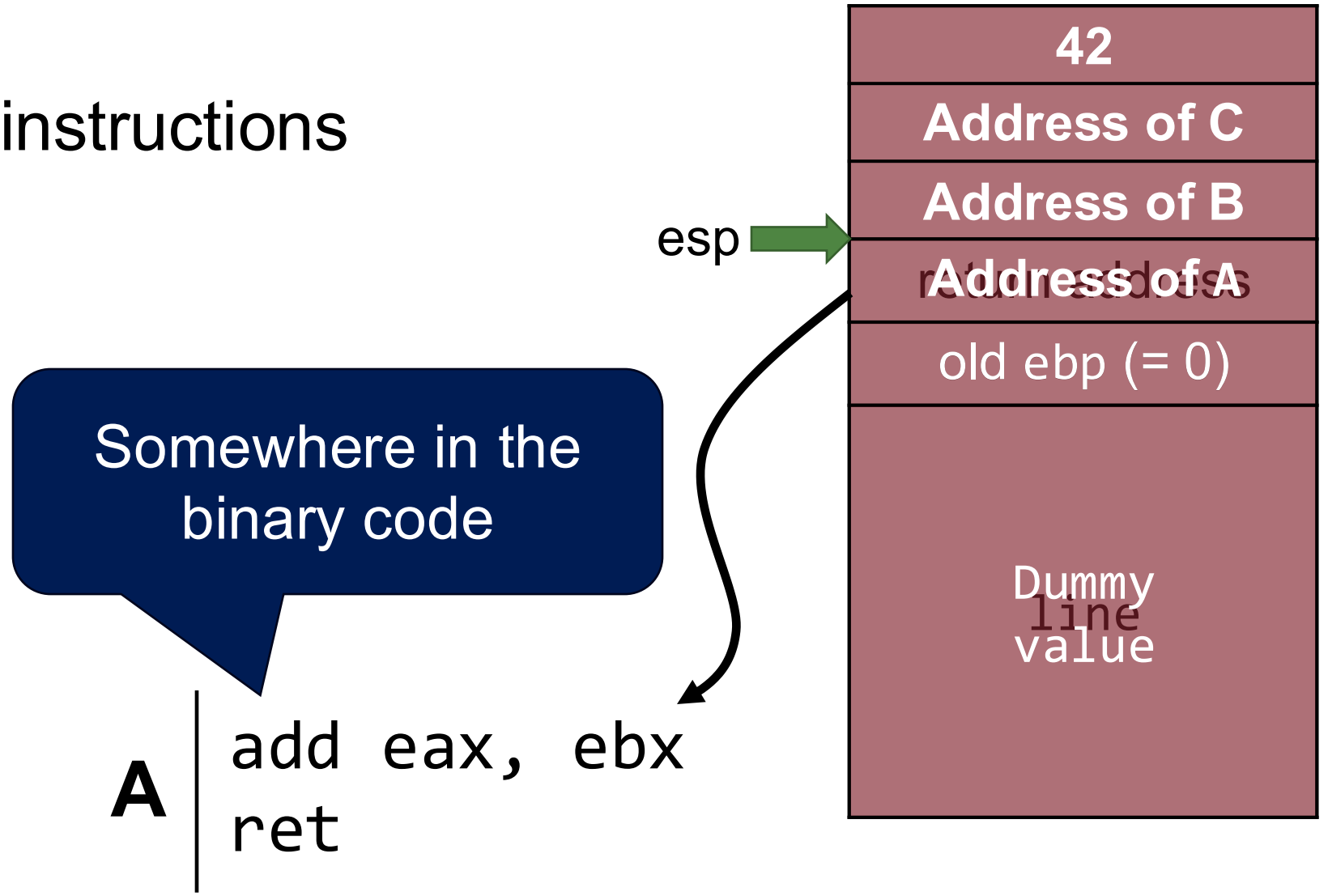
```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```



Main Idea: Return (ret) Chaining

Attacker's goal:
execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```



Main Idea: Return (ret) Chaining

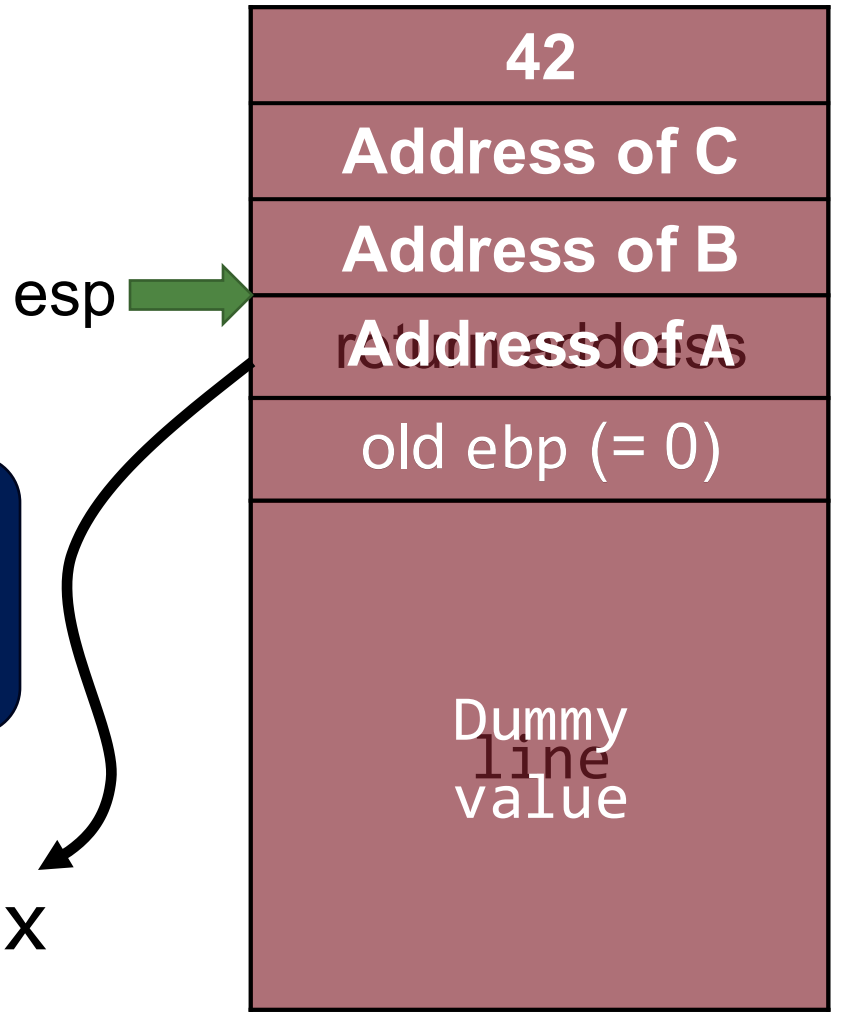
Attacker's goal:
execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```

ROP Gadget:
Instruction sequence that ends with `ret`

A

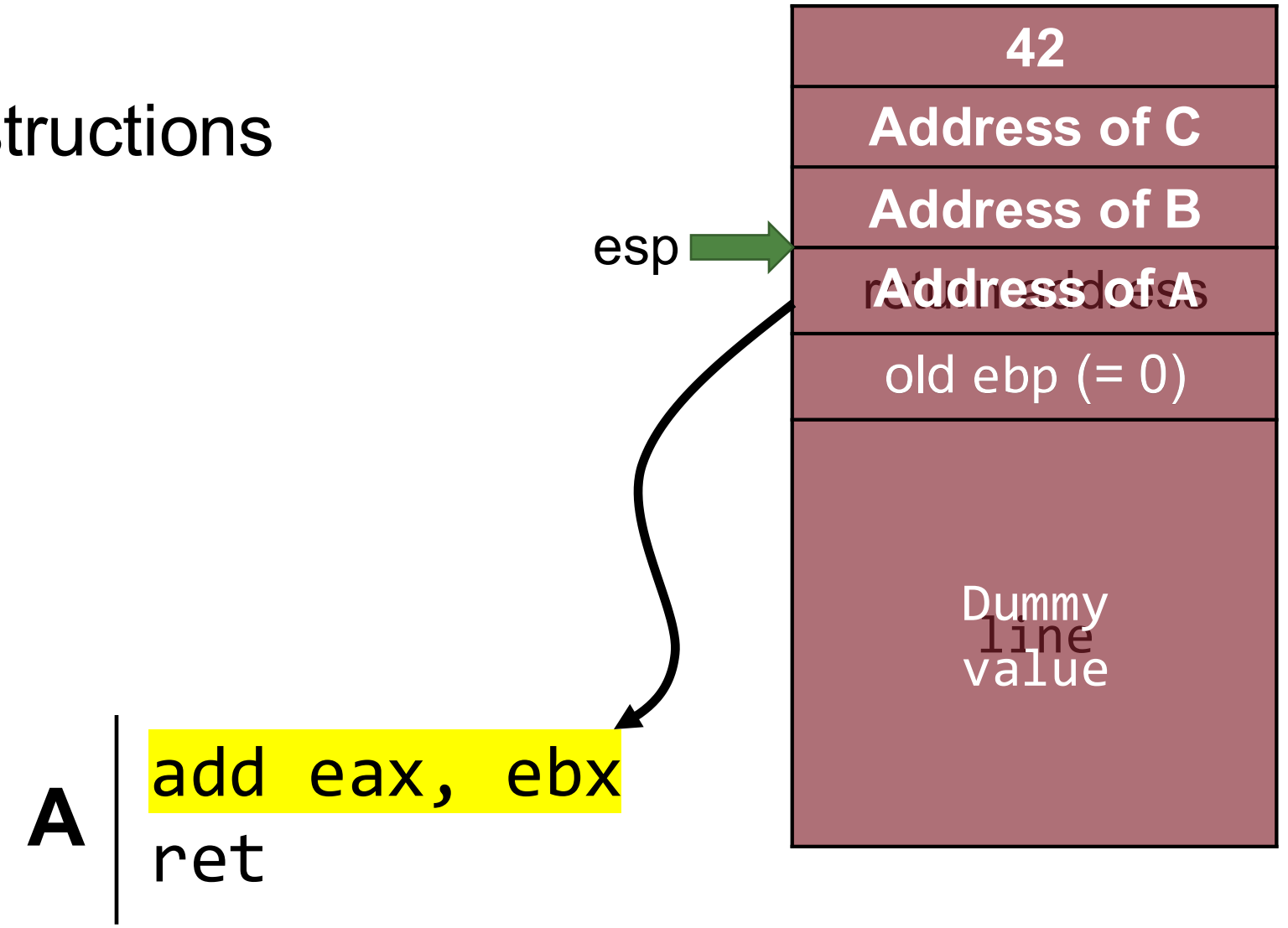
```
add eax, ebx
ret
```



Main Idea: Return (ret) Chaining

Attacker's goal:
execute following instructions

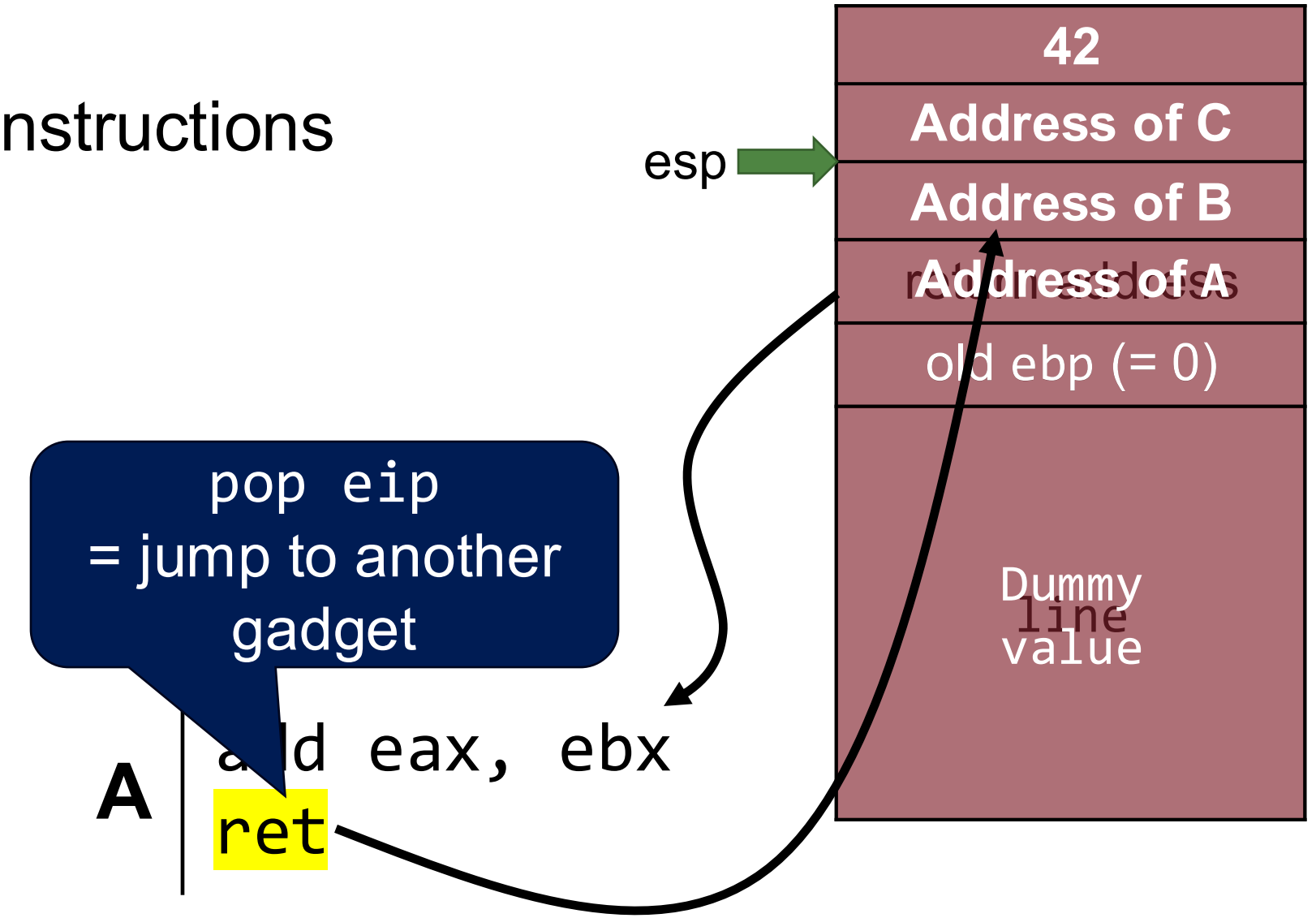
```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```



Main Idea: Return (ret) Chaining

Attacker's goal:
execute following instructions

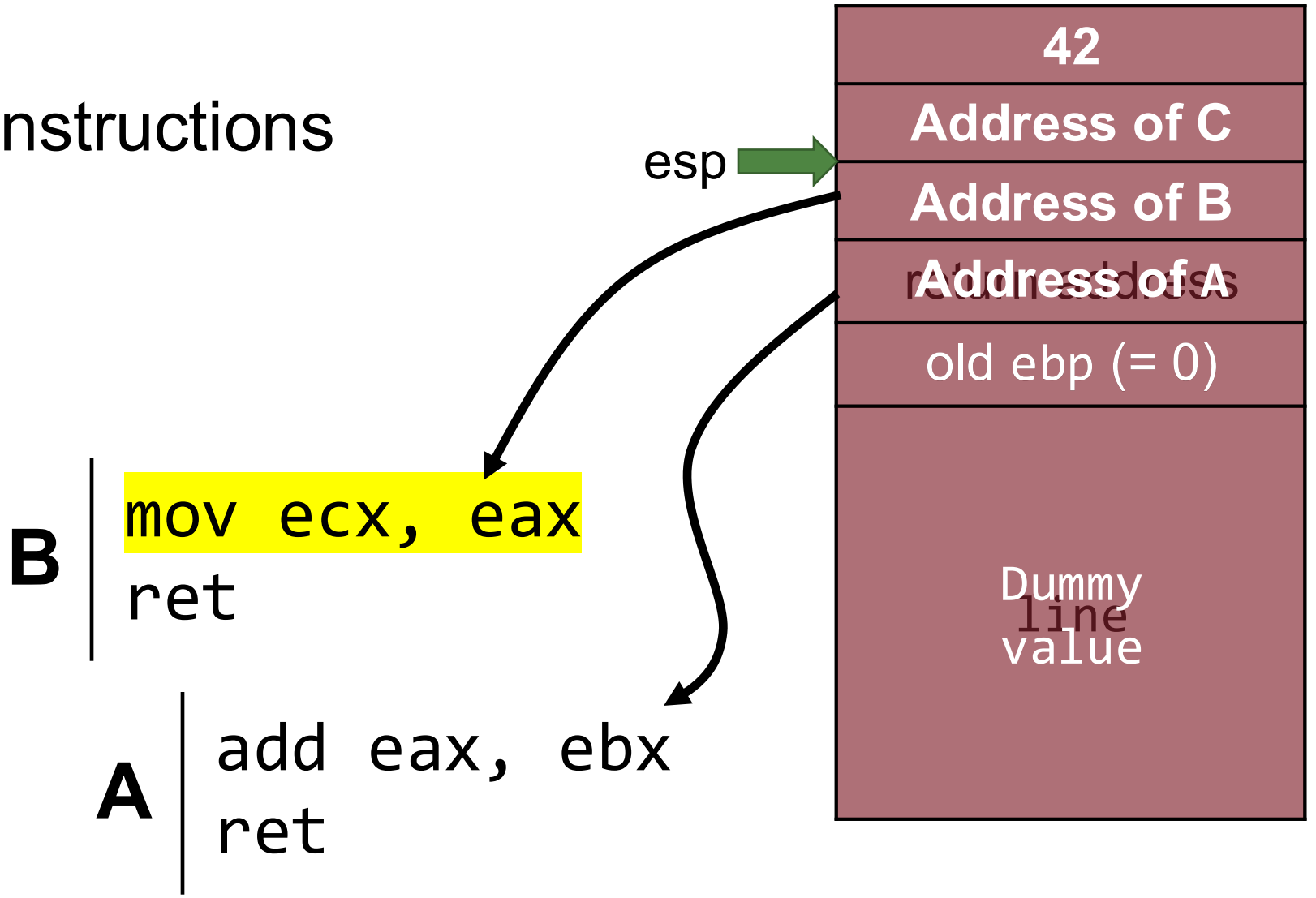
```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```



Main Idea: Return (ret) Chaining

Attacker's goal:
execute following instructions

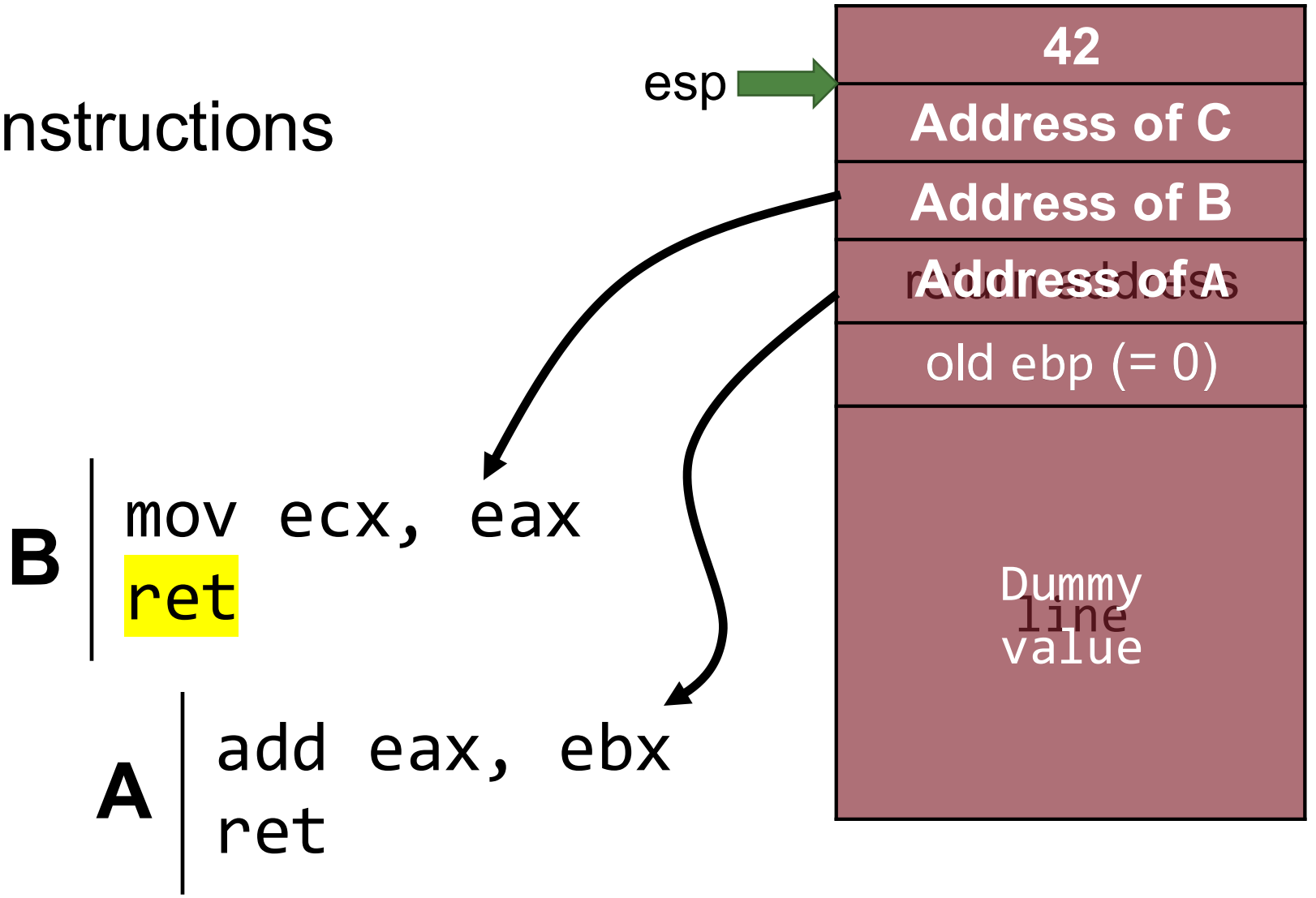
```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```



Main Idea: Return (ret) Chaining

Attacker's goal:
execute following instructions

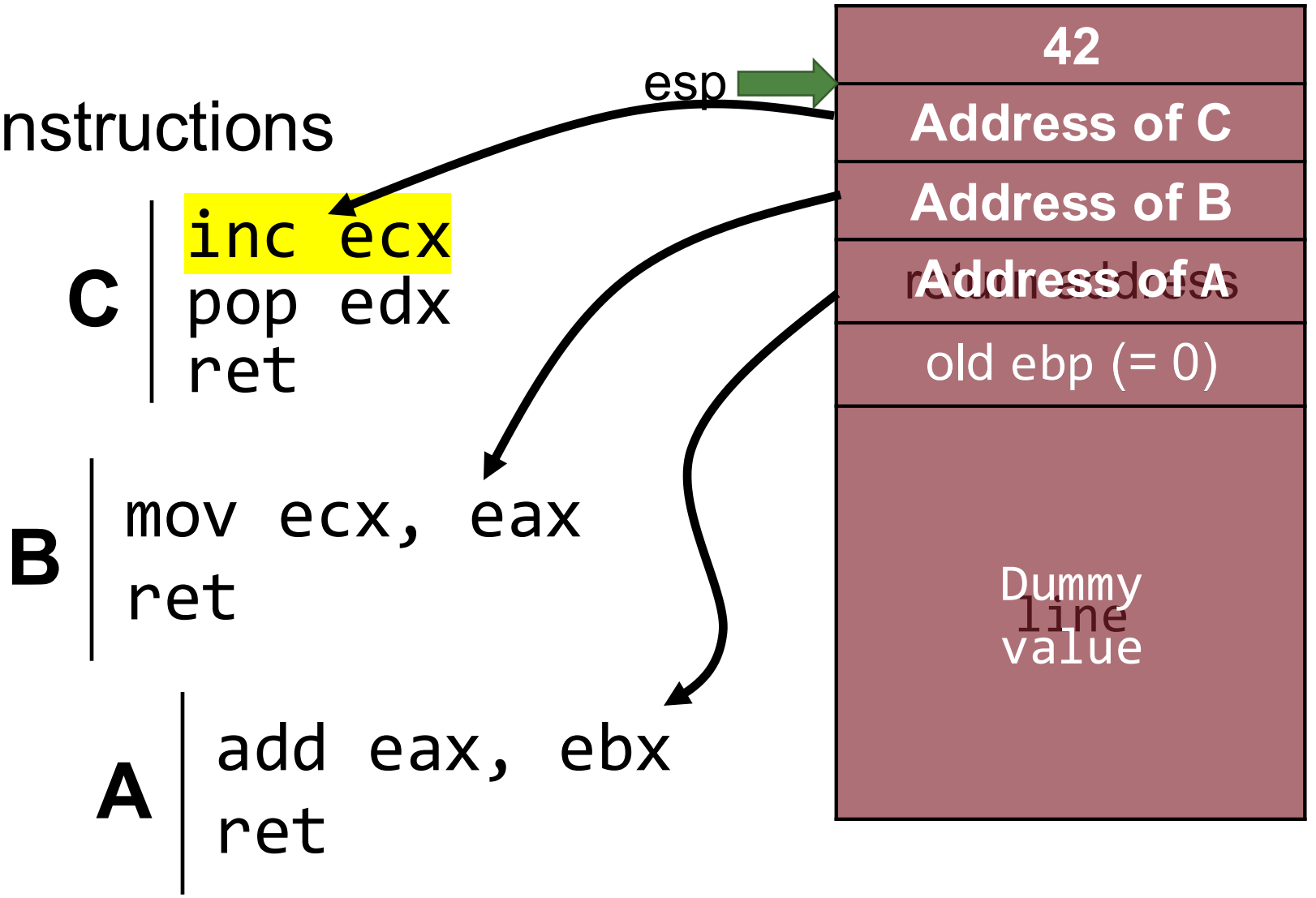
```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```



Main Idea: Return (ret) Chaining

Attacker's goal:
execute following instructions

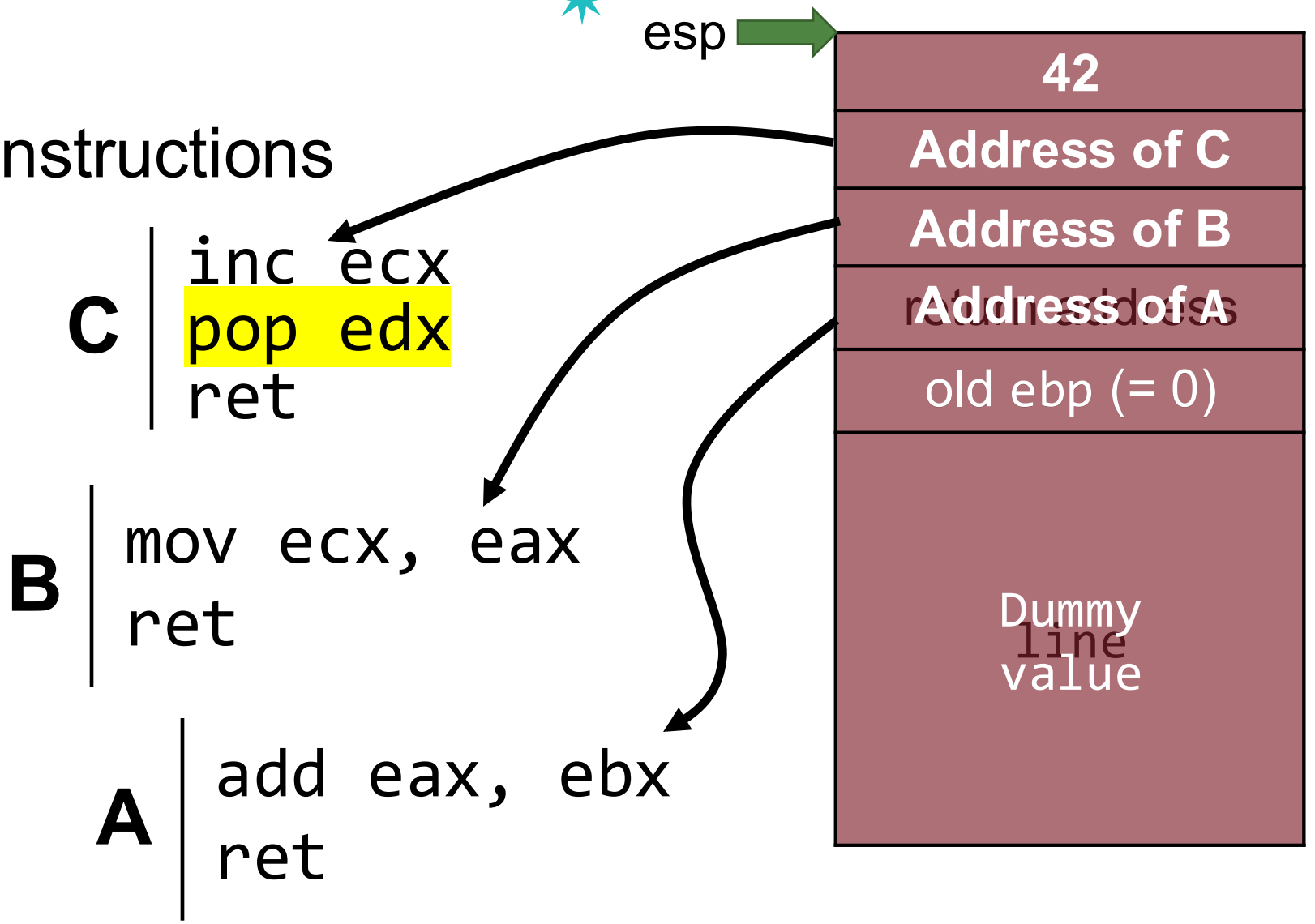
```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```



Main Idea: Return (ret) Chaining

Attacker's goal:
execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```



Main Idea: Return (ret) Chaining

Attacker's goal:
execute following instructions

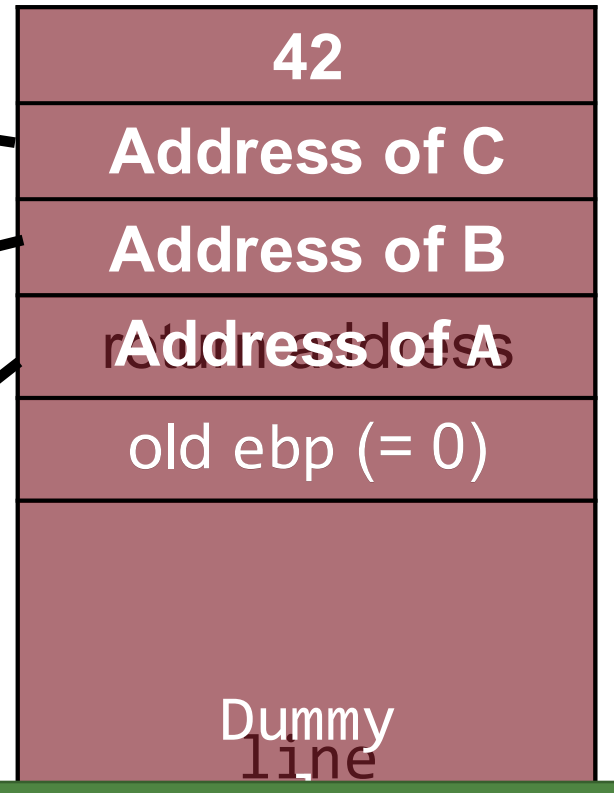
```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```

C

```
inc ecx
pop edx
ret
```

B

```
mov ecx, eax
ret
```



Return chaining with ROP gadgets allows **arbitrary computation**!

ROP Workflow

1. Disassemble binary

2. Identify useful instruction sequences (i.e., gadgets)
   - E.g., an instruction sequence that ends with `ret` is useful
   - E.g., an instruction sequence that ends with `jmp reg` can be useful
     (`pop eax; jmp eax`)

3. Assemble gadgets to perform some computation
   - E.g., spawning a shell

Challenge: Gathering as **many gadgets** as possible

Many Gadgets in Regular Binaries?

x86 instructions have their lengths ranging from 1 byte to 18 bytes, i.e., it uses ***variable-length encoding***

x86 instructions have variable lengths

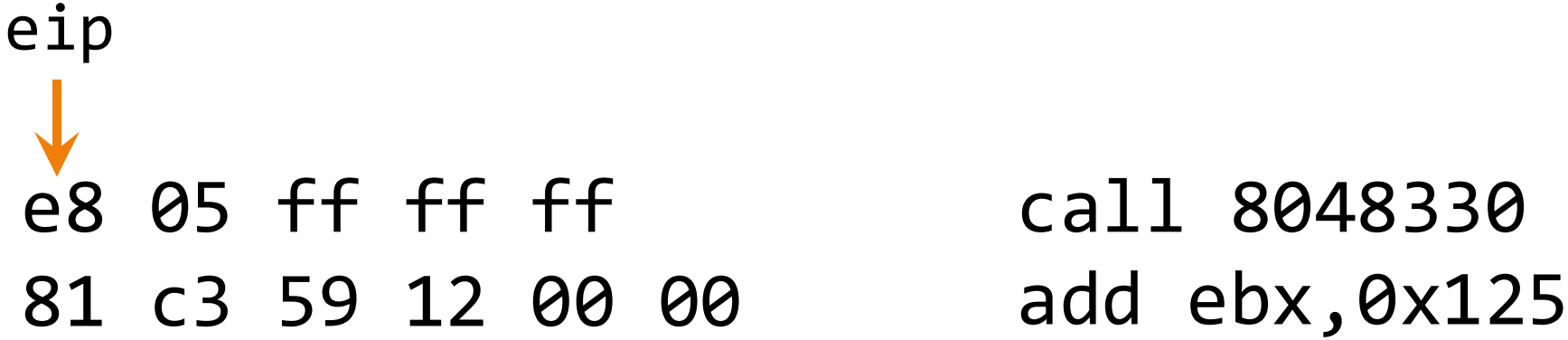
```
08048aac <main>:
 8048aac:       8d 4c 24 04             lea    ecx,[esp+0x4]
 8048ab0:       83 e4 f0                and    esp,0xfffffff0
 8048ab3:       ff 71 fc                push   DWORD PTR [ecx-0x4]
 8048ab6:       55                      push   ebp
 8048ab7:       89 e5                   mov    ebp,esp
 8048ab9:       51                      push   ecx
 8048aba:       83 ec 14                sub    esp,0x14
 8048abd:       c7 45 f0 88 ad 0a 08    mov    DWORD PTR [ebp-0x10],0x80aad88
 8048ac4:       c7 45 f4 00 00 00 00    mov    DWORD PTR [ebp-0xc],0x0
 8048acb:       83 ec 04                sub    esp,0x4
 8048ace:       6a 00                   push   0x0
 8048ad0:       8d 45 f0                lea    eax,[ebp-0x10]
 8048ad3:       50                      push   eax
 8048ad4:       68 88 ad 0a 08          push   0x80aad88
 8048ad9:       e8 02 39 01 00          call   805c3e0 <__execve>
 ...
```

Many Gadgets in Regular Binaries?

x86 instructions have their lengths ranging from 1 byte to 18 bytes, i.e., it uses ***variable-length encoding***

Therefore, there can be both **intended** and **unintended gadgets** in x86 binaries

Disassembling x86



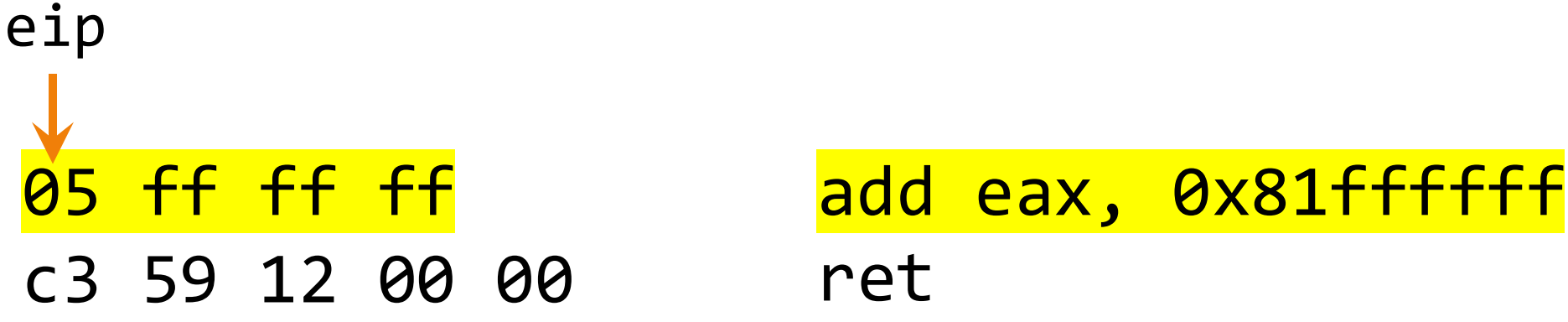
What if we disassemble the code from the second byte (`05`)?

Unintended `ret` Insturction


```
e8 05 ff ff ff          add eax, 0x81ffffff
81 c3 59 12 00 00       ret
```

Totally different, but still valid instructions!

Unintended ret Insturction



Unintended `ret` Insturction


```
e8 05 ff ff ff          add eax, 0x81ffffff
81 c3 59 12 00 00       ret
```

Many Gadgets in Regular Binaries?

Also, program size may matter!

Larger code ⇒ More chance to get useful gadgets

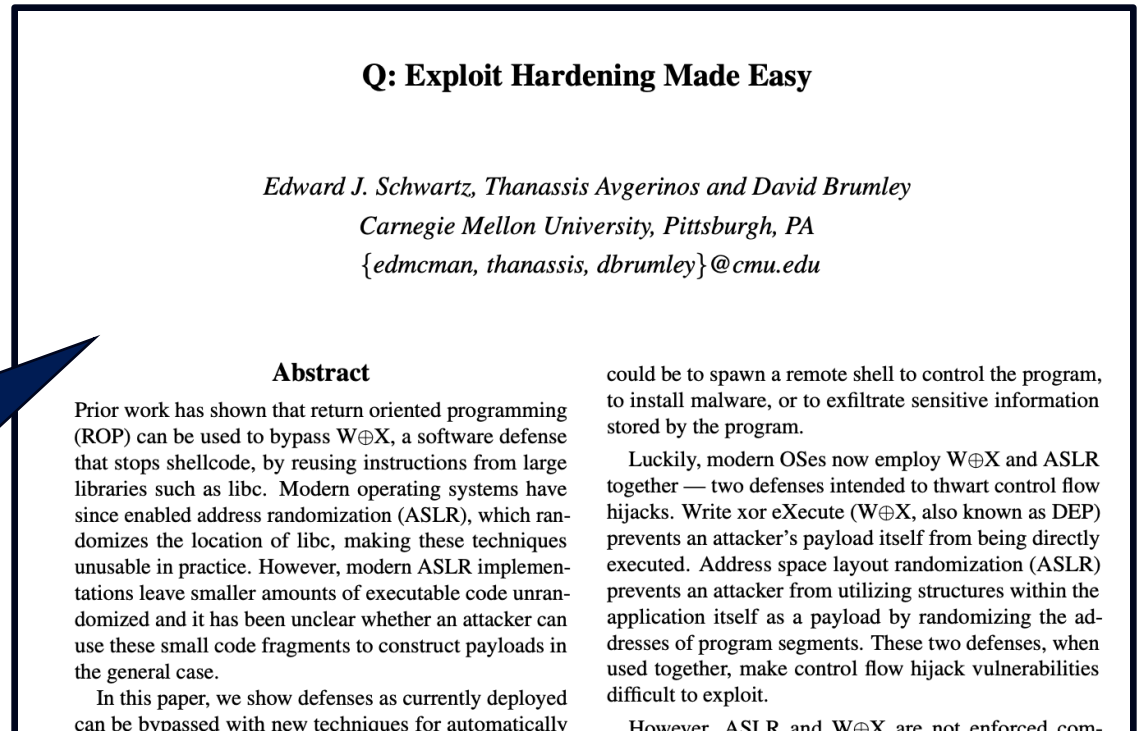
Many Gadgets in Regular Binaries?

Also, program size may matter!

Larger code ⇒ More chance to get useful gadgets

Exploit Hardening Made Easy,
USENIX Security 2011

Show that 100KB was enough to successfully create exploits for 80% of the binaries in /usr/bin



Q: Exploit Hardening Made Easy

Edward J. Schwartz, Thanassis Avgerinos and David Brumley
Carnegie Mellon University, Pittsburgh, PA
{*edmcman, thanassis, dbrumley*}*@cmu.edu*

Abstract

Prior work has shown that return oriented programming (ROP) can be used to bypass W⊕X, a software defense that stops shellcode, by reusing instructions from large libraries such as libc. Modern operating systems have since enabled address randomization (ASLR), which randomizes the location of libc, making these techniques unusable in practice. However, modern ASLR implementations leave smaller amounts of executable code unrandomized and it has been unclear whether an attacker can use these small code fragments to construct payloads in the general case.

In this paper, we show defenses as currently deployed can be bypassed with new techniques for automatically

could be to spawn a remote shell to control the program, to install malware, or to exfiltrate sensitive information stored by the program.

Luckily, modern OSes now employ W⊕X and ASLR together — two defenses intended to thwart control flow hijacks. Write xor eXecute (W⊕X, also known as DEP) prevents an attacker's payload itself from being directly executed. Address space layout randomization (ASLR) prevents an attacker from utilizing structures within the application itself as a payload by randomizing the addresses of program segments. These two defenses, when used together, make control flow hijack vulnerabilities difficult to exploit.

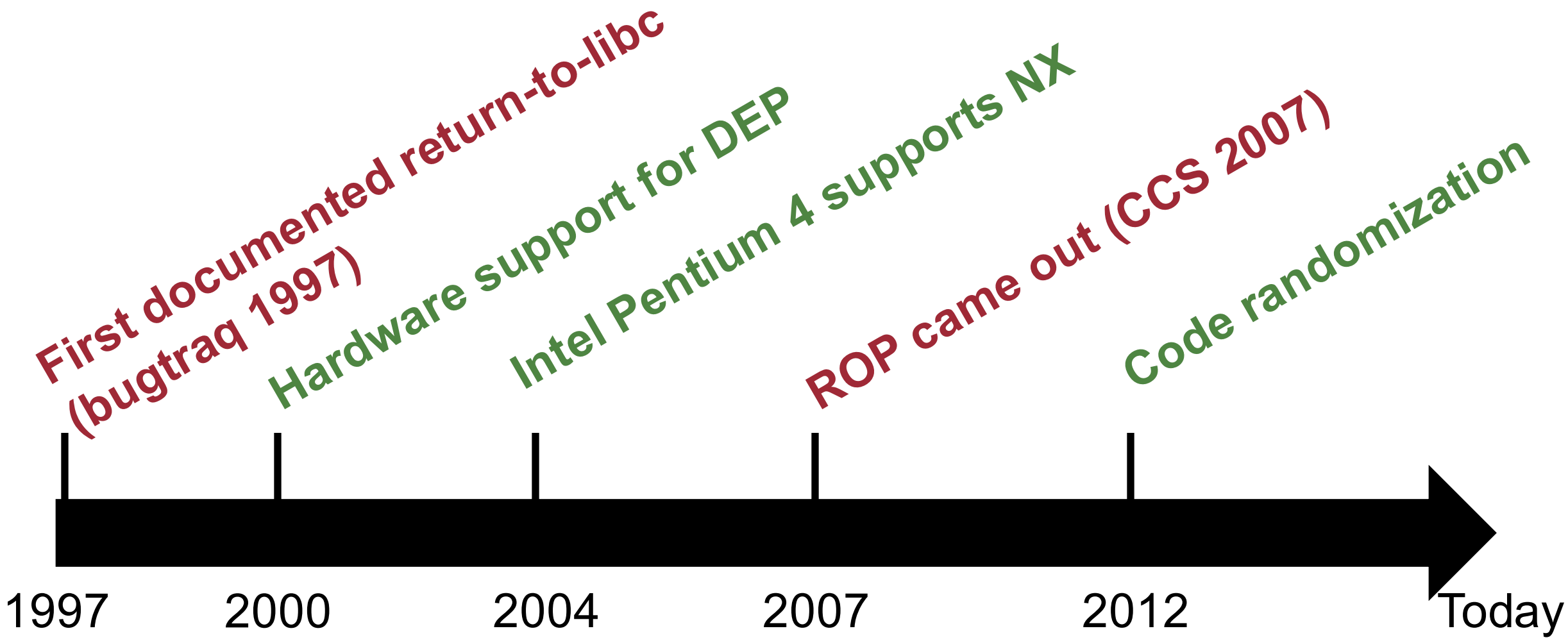
However, ASLR and W⊕X are not enforced com-

Question

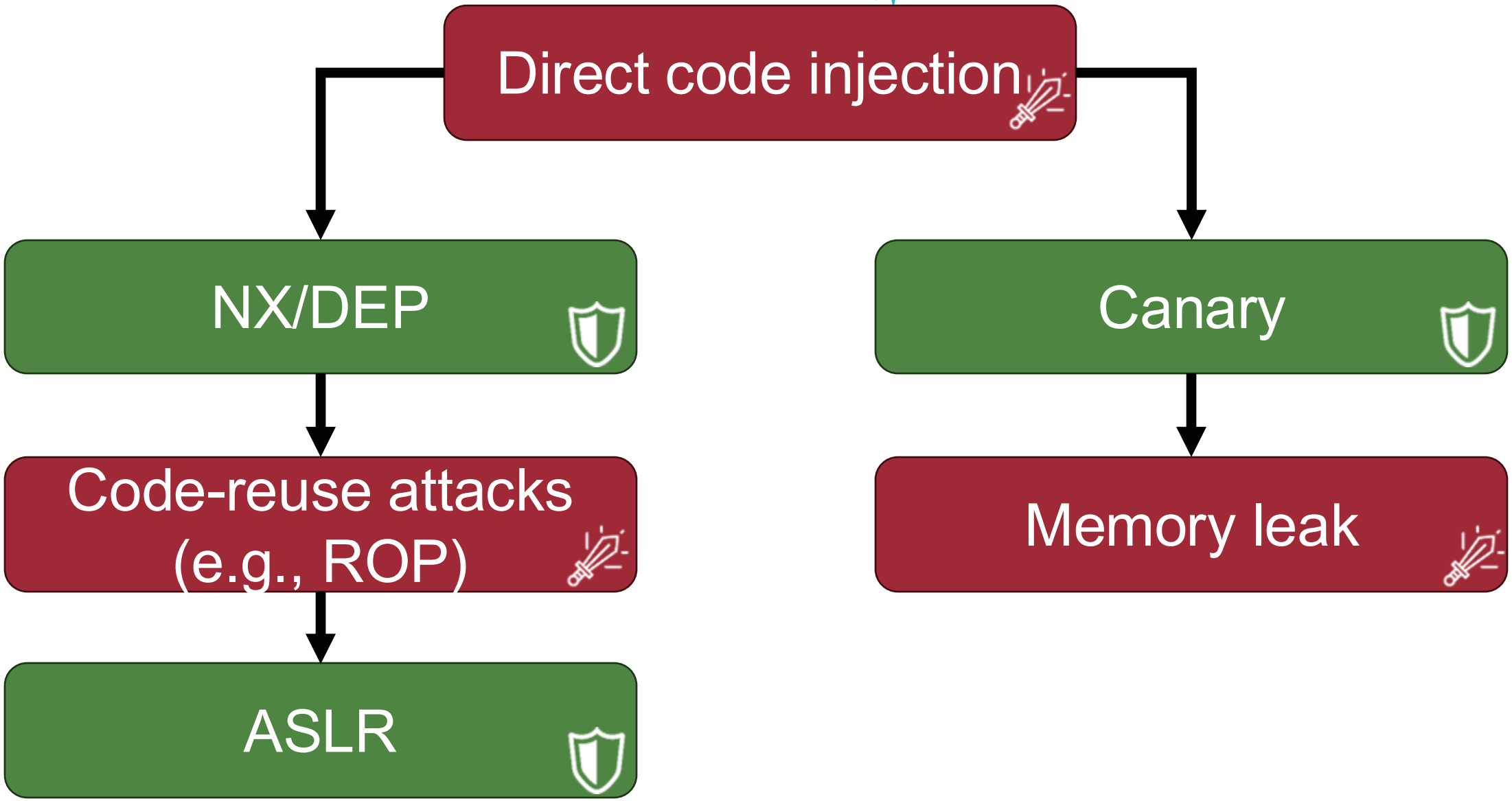
How can we mitigate code reuse attacks (ROP)?

Address randomization (ASLR)!
(next lecture)

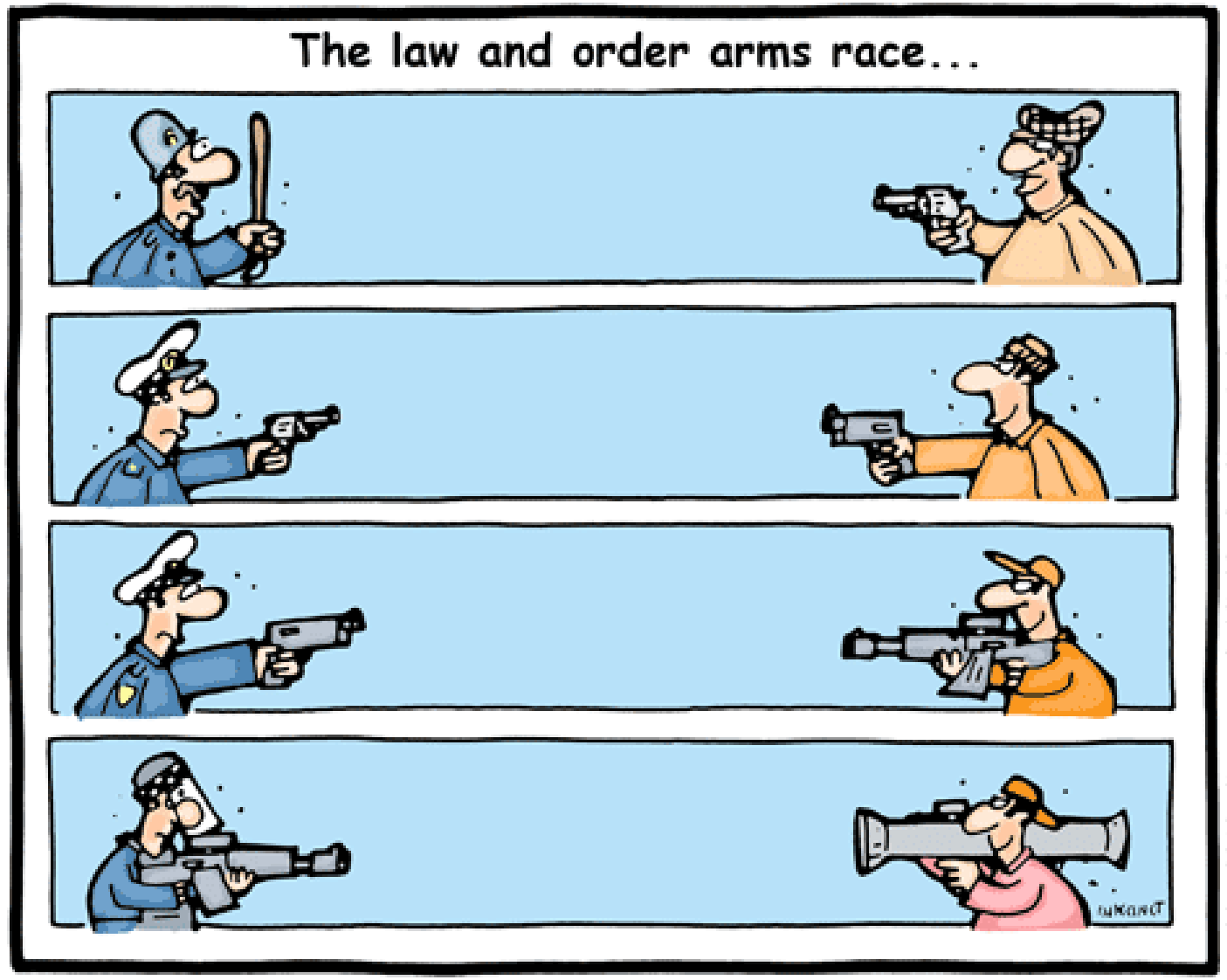
DEP and Code Reuse Attacks



Control Hijack Attack / Defense So Far



Arms Race in Security



Summary

- Code reuse attacks allow an attacker to bypass DEP
- Many mitigation techniques are proposed for code reuse attacks, which will be covered next

Question?