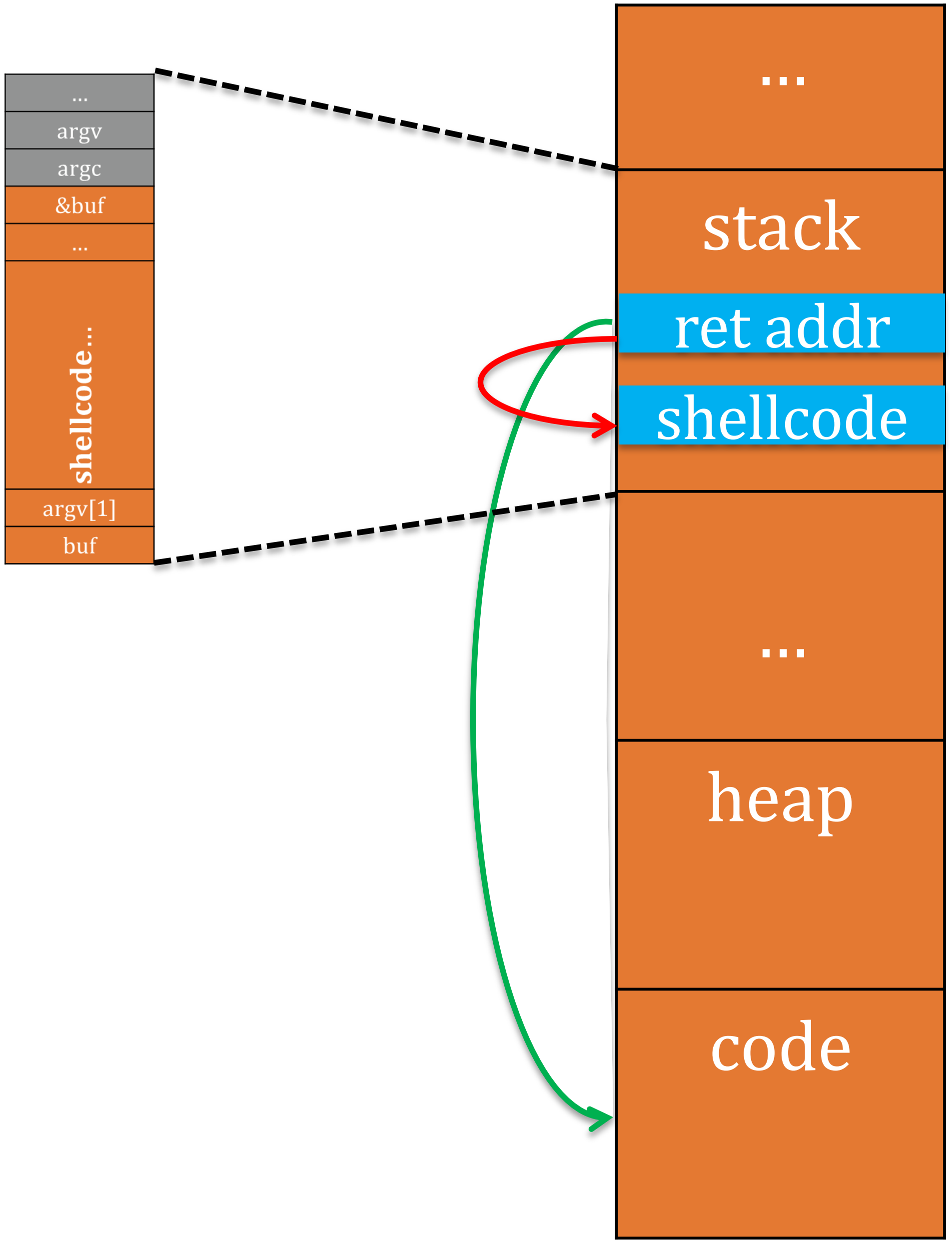
CS165 – Computer Security

Overflow Defenses and Return-Oriented Programming

October 10, 2025

Attacks Injecting Shellcode



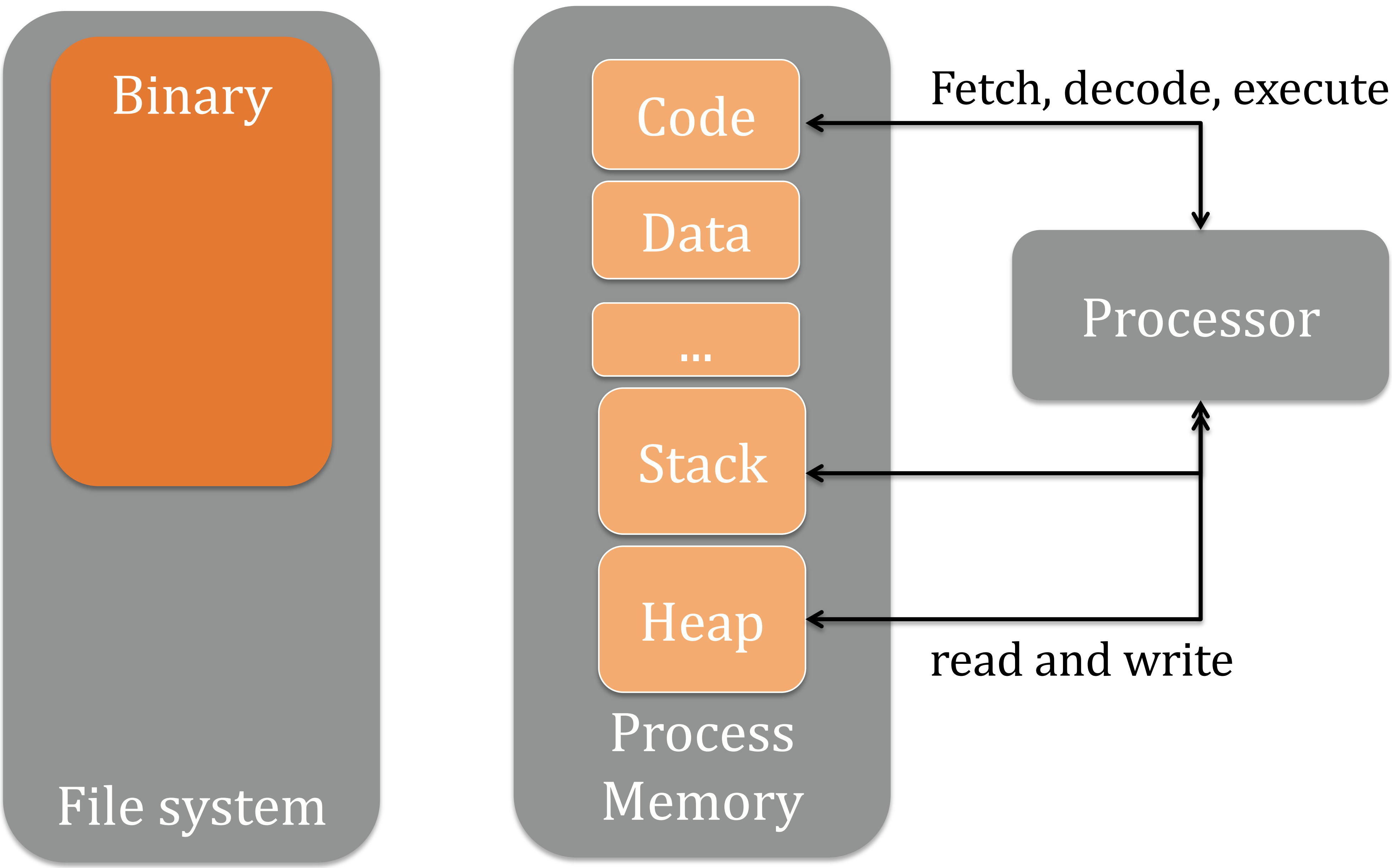
Injection Requirements

- What is required for a code injection attack?
  - Needed by the adversary…
  - That is not expected in practice?

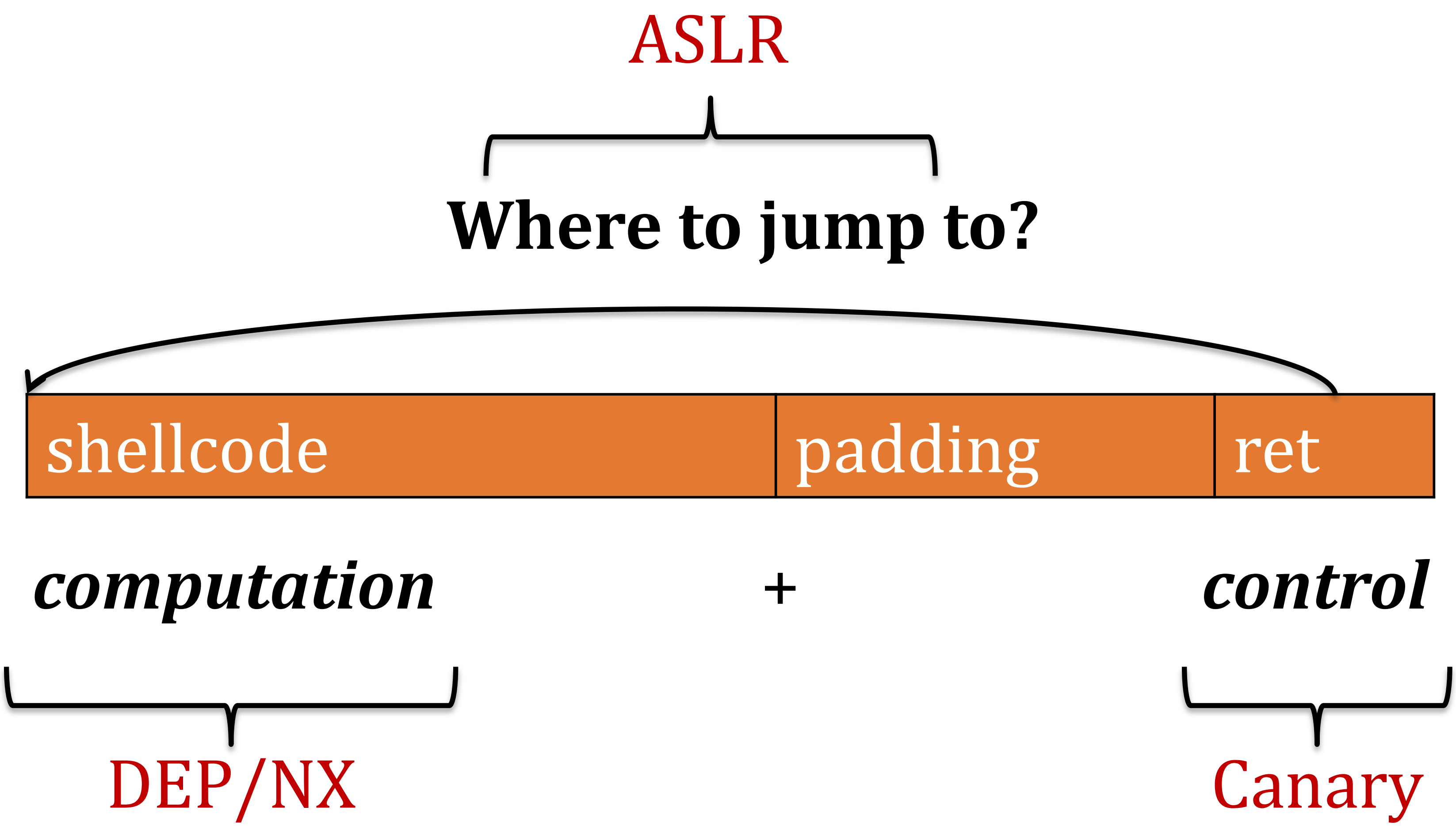
Injection Requirements

- What is required for a code injection attack?
  - Needed by the adversary…
  - That is not expected in practice?
- Answer: Execute stack memory
  - Code is injected in stack memory
  - Predict the location of the injected code
  - Set return address to location of injected code
- Must all memory be executable?
  - Recall page permissions

Basic Execution



Control-Flow Hijacking Defenses



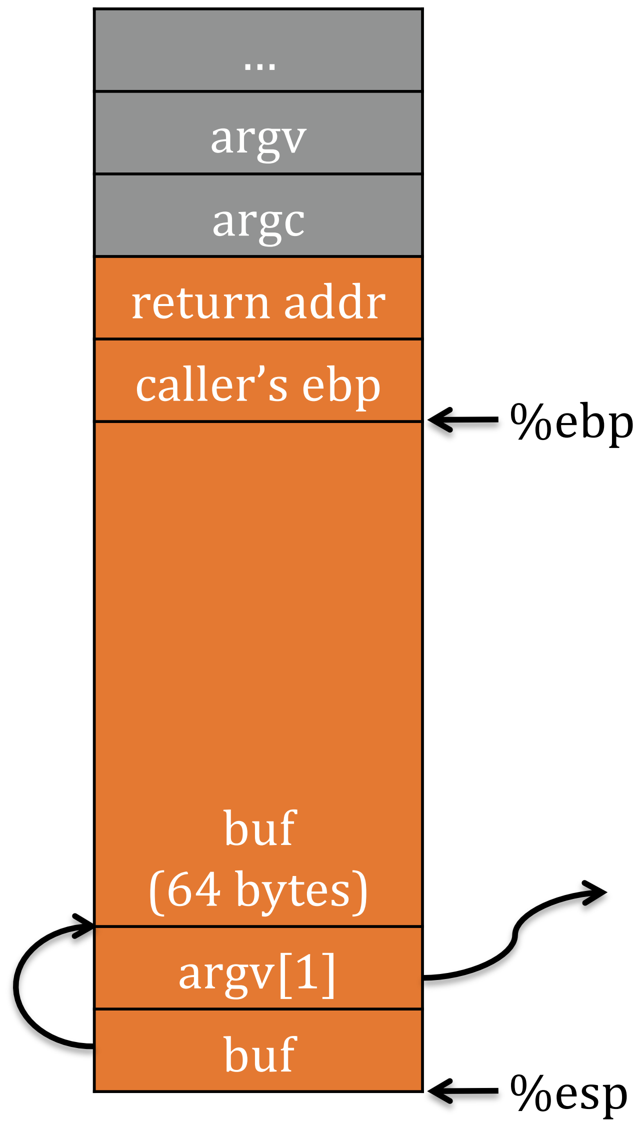
Wikipedia: “the historic practice of using canaries in coal mines, since they would be affected by toxic gases earlier than the miners, thus providing a biological warning system.”

Canary / Stack Cookies

http://en.wikipedia.org/wiki/File:Domestic_Canary_-_Serinus_canaria.jpg

“A”x68 . “\xEF\xBE\xAD\xDE”

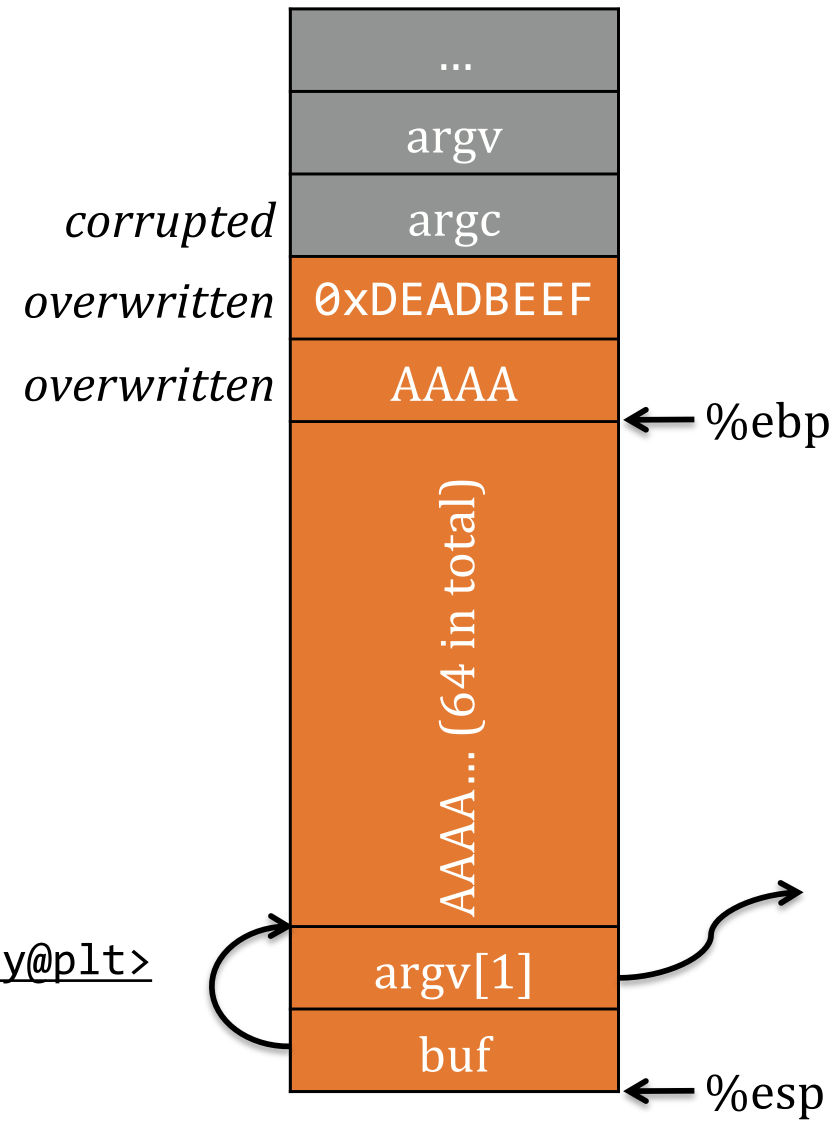
```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



“A”x68 . “\xEF\xBE\xAD\xDE”

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

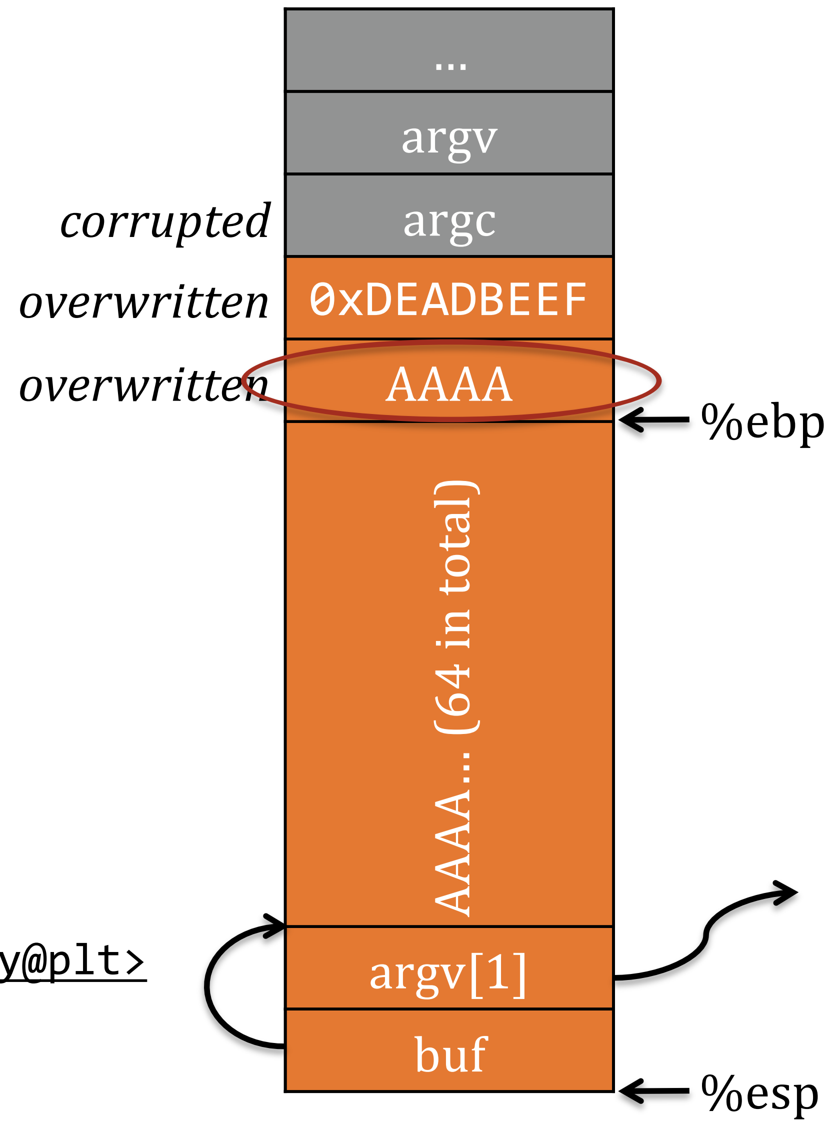
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



“A”x68 . “\xEF\xBE\xAD\xDE”

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

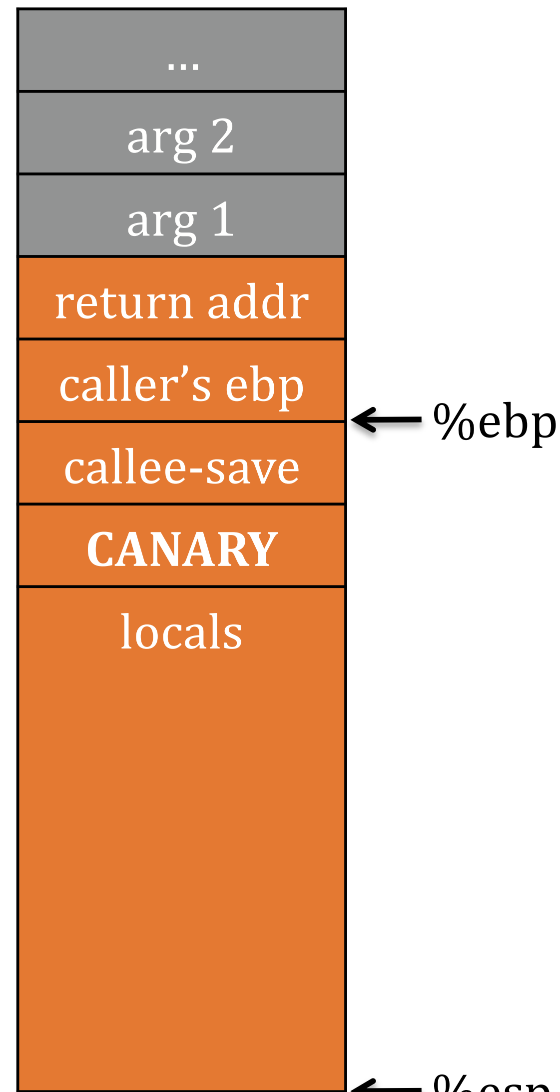
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



StackGuard [Cowen etal. 1998]

Idea:

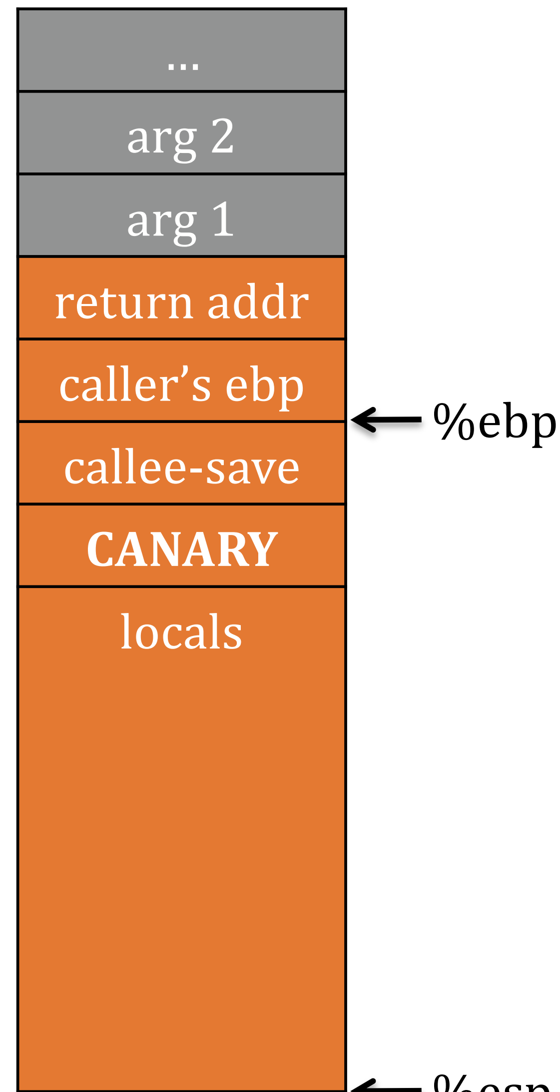
- prologue introduces a ***canary word*** between return addr and locals



StackGuard [Cowen etal. 1998]

Idea:

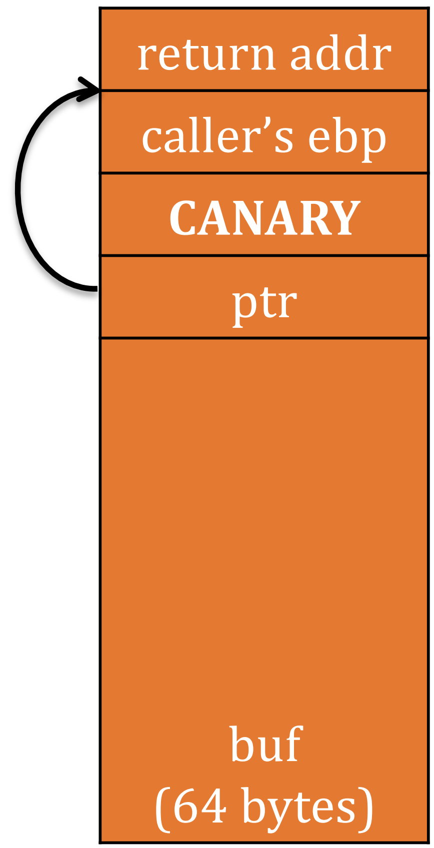
- prologue introduces a ***canary word*** between return addr and locals
- epilogue checks canary before function returns



Bypass: Data Pointer Corruption

Overwrite a data pointer *first*…

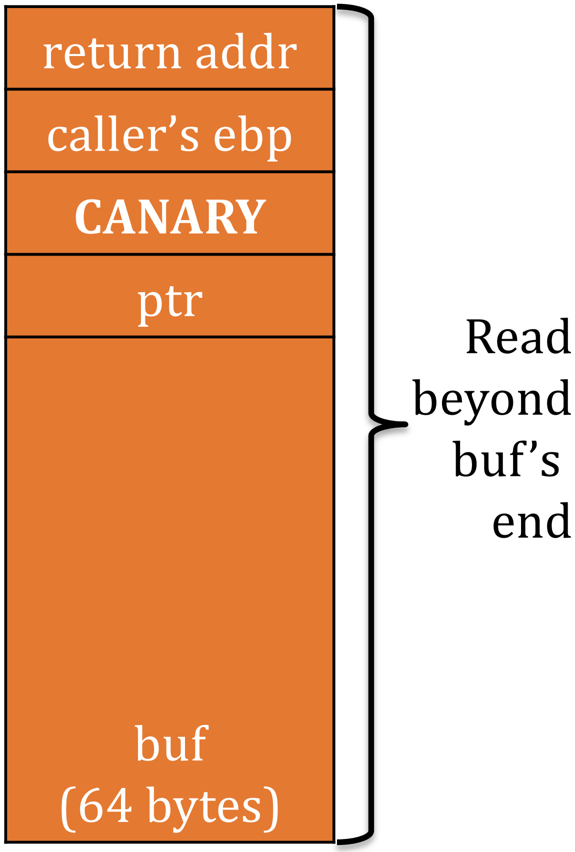
```
int *ptr;
char buf[64];
n = strlen(user1);
memcpy(buf, user1, n);
*ptr = user2;
```



Bypass: Buffer Over-Read (Disclosure)

Read beyond the end of buf to find the canary value

```
char user1[user2];
char buf[64];
memcpy(user1, buf, user2);
printf(”%s”, user1);
```

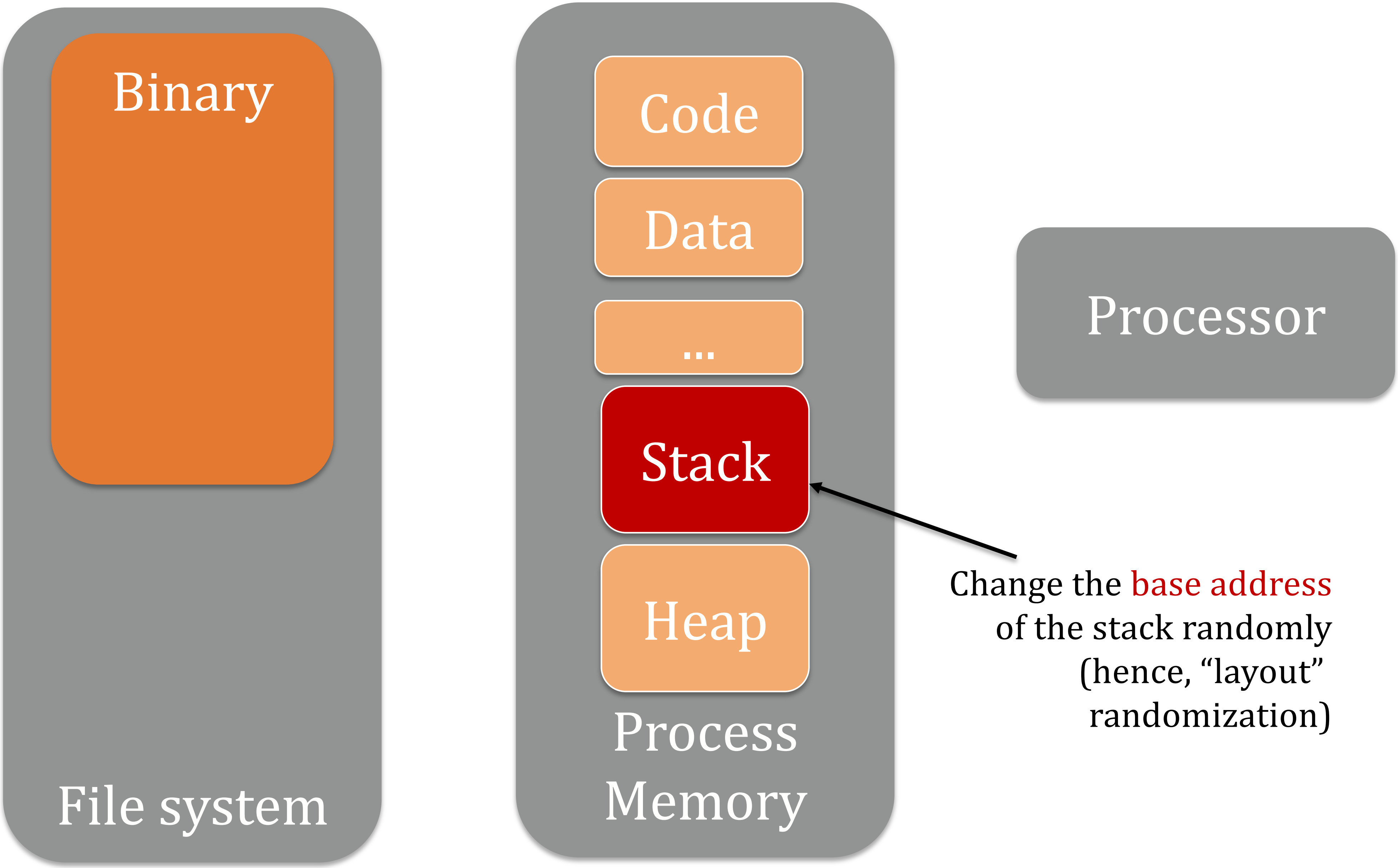


Canary Scorecard

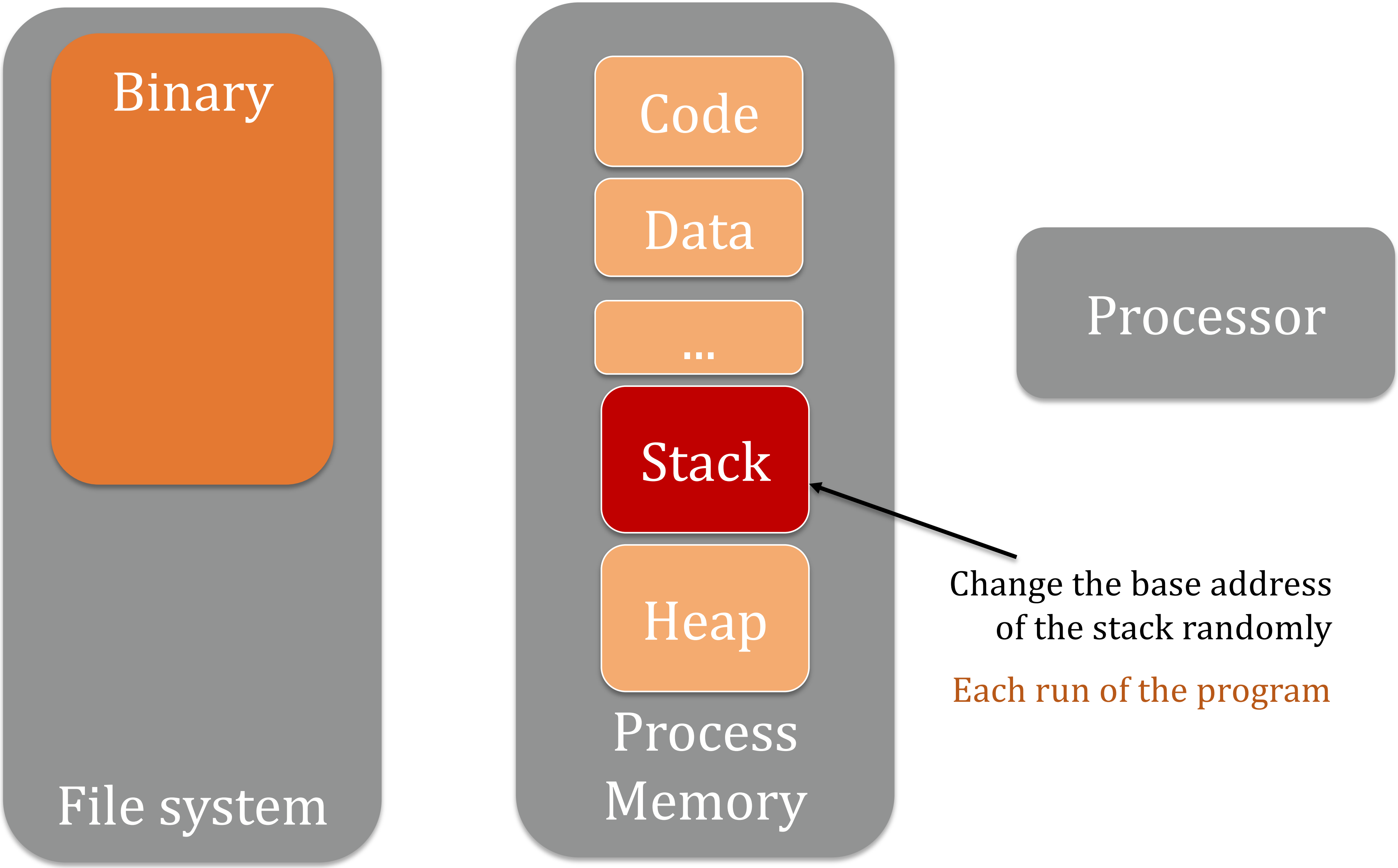
Aspect	Canary
Performance	• Some add'l instructions per function • time: a few percent on average • size: can optimize away in safe functions (but see MS08-067 *)
Deployment	• recompile suffices; no code change
Compatibility	• perfect—invisible to outside
Safety Guarantee	• *not really…*

* http://blogs.technet.com/b/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx

Address Space Layout Randomization



Address Space Layout Randomization

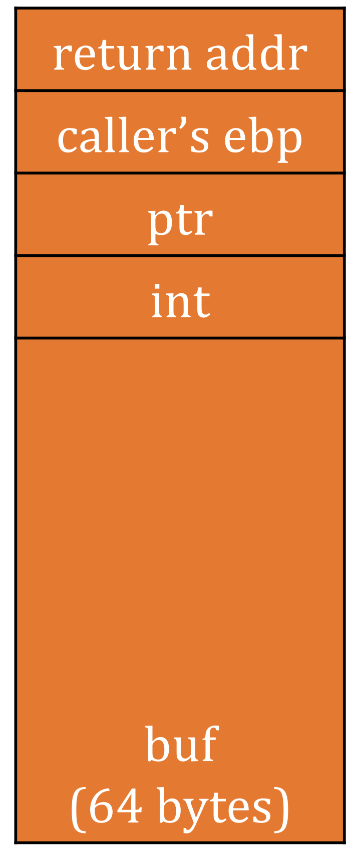


Stack Addresses

Normally, when you run a program, the stack is at the same memory location for every run

`So, buf is at the same address for each run`

`Why does that help for code injection attacks in buf?`

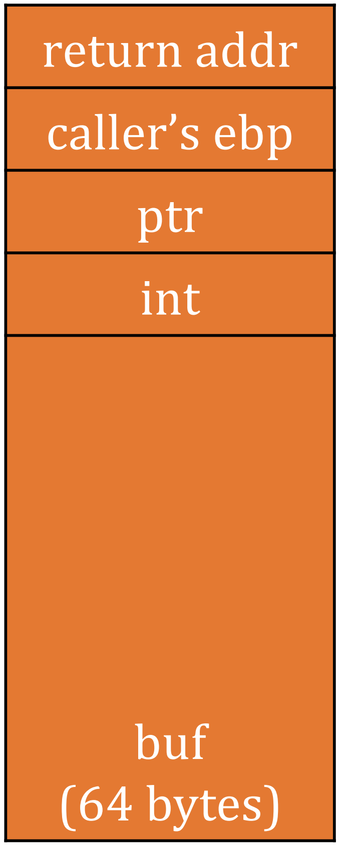


Randomize Base Addresses

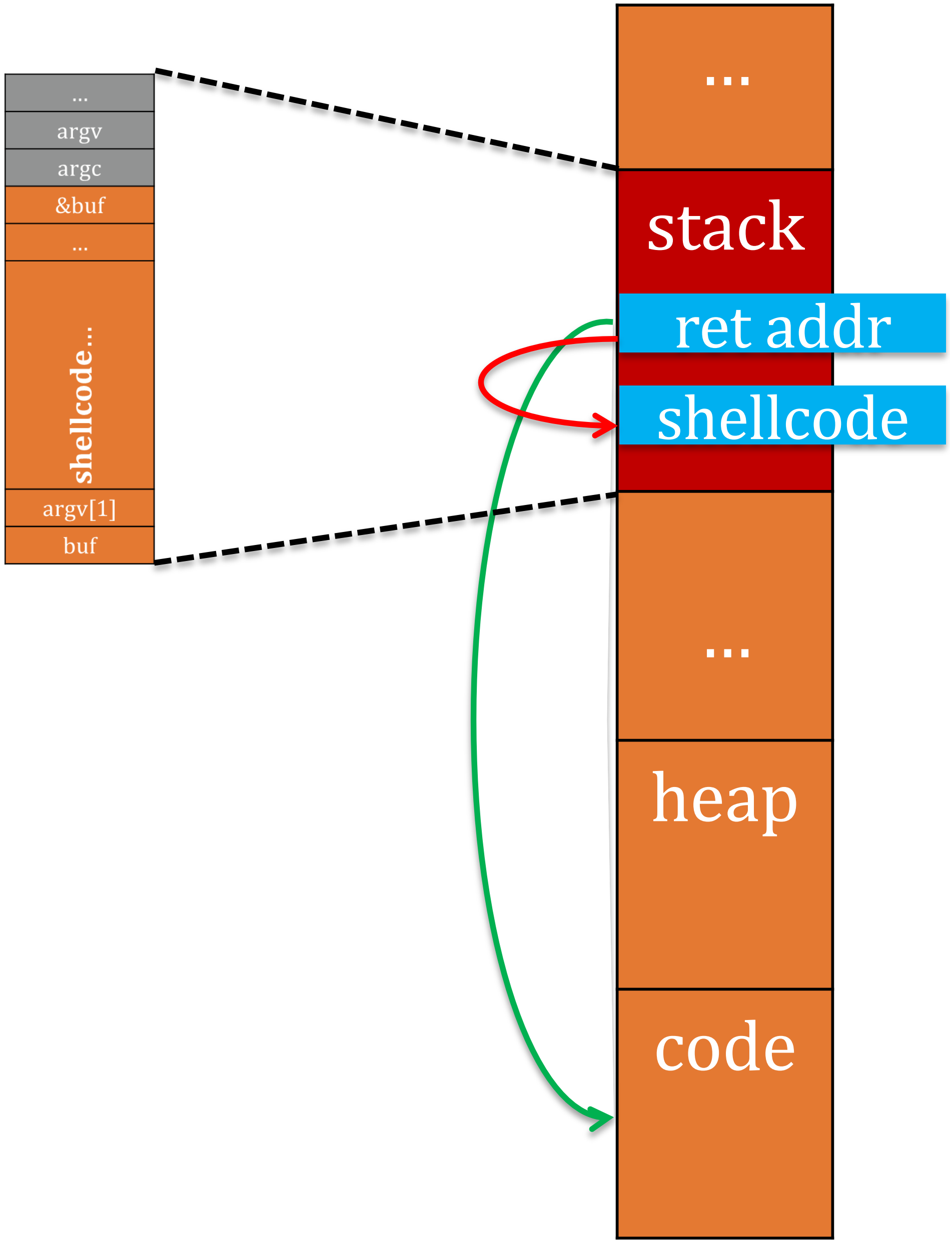
Instead choose a different address for the stack base each run – just the base is changed

Makes the location of buf in the stack unpredictable

Effectiveness is dependent on the number of possible base address locations – varies by system



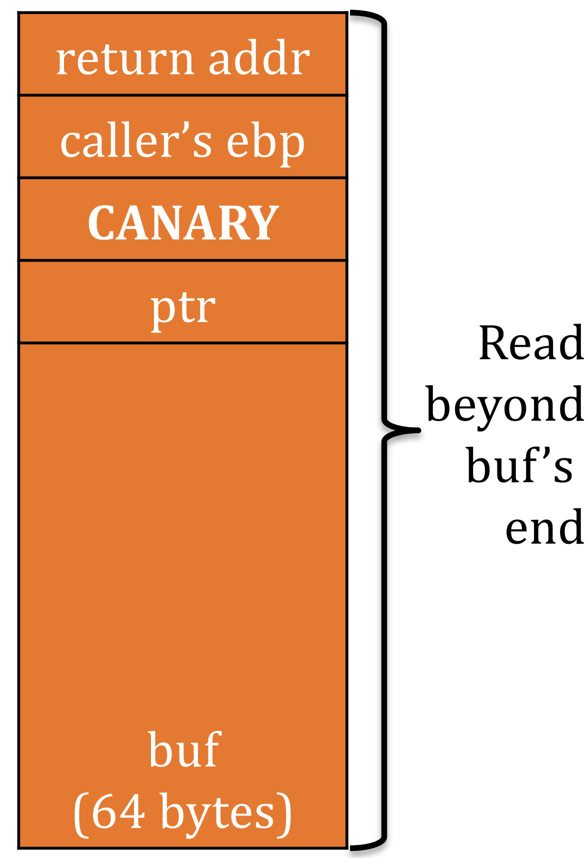
Thwarts Finding Shellcode



Bypass: Disclose Stack Pointer

Overread to read stack addresses to determine base address

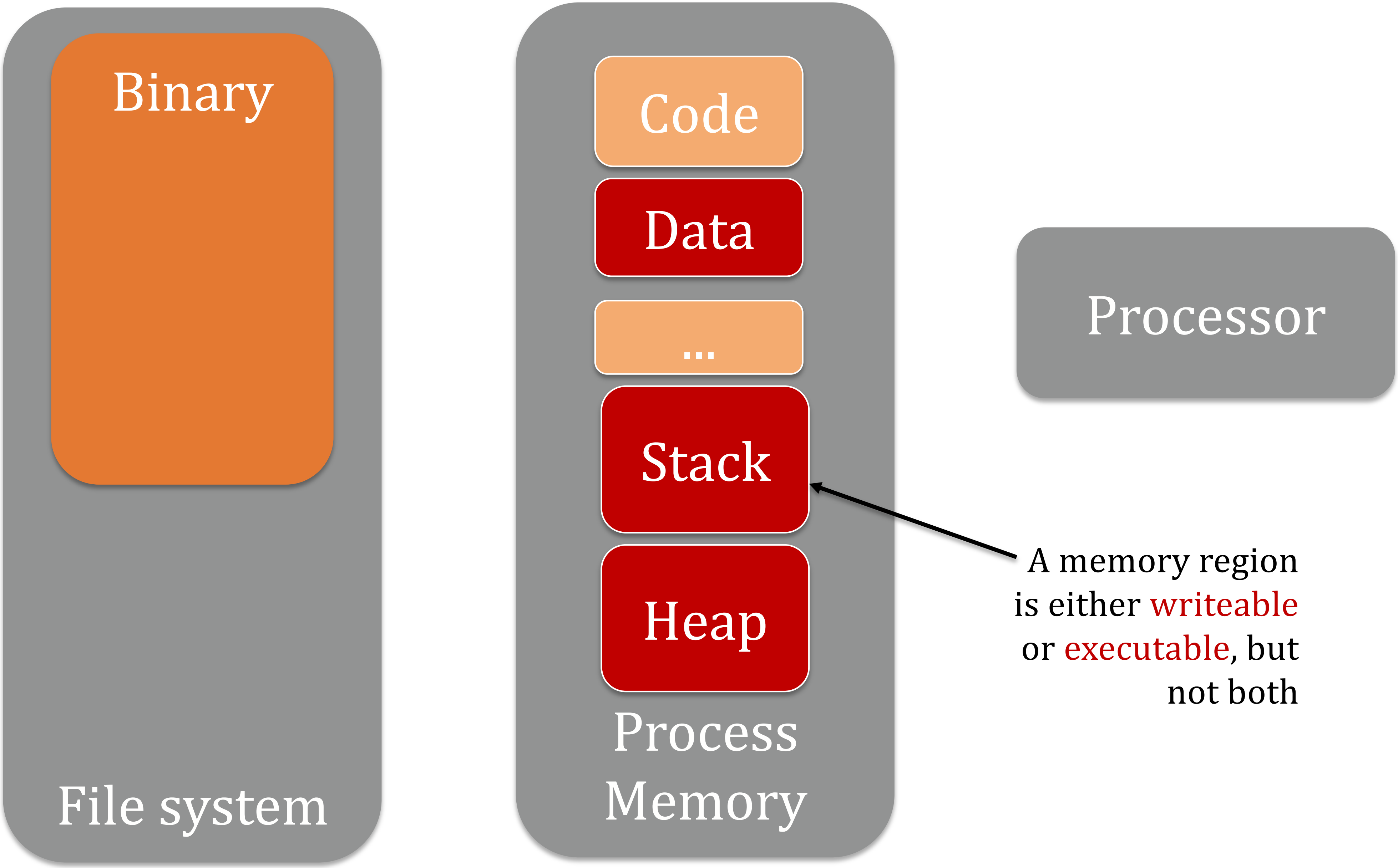
```
char user1[user2];
char buf[64];
memcpy(user1, buf, user2);
offset = &user1[64]-expect;
```



ASLR Scorecard

Aspect	ASLR
Performance	• few instructions when allocating stack • time: very little overhead • size: use more space for stack, but not that large in general (and 64-bit addrs)
Deployment	• recompile suffices; no code change
Compatibility	• perfect—invisible to outside
Safety Guarantee	• *not really…*

Data Execution Protection



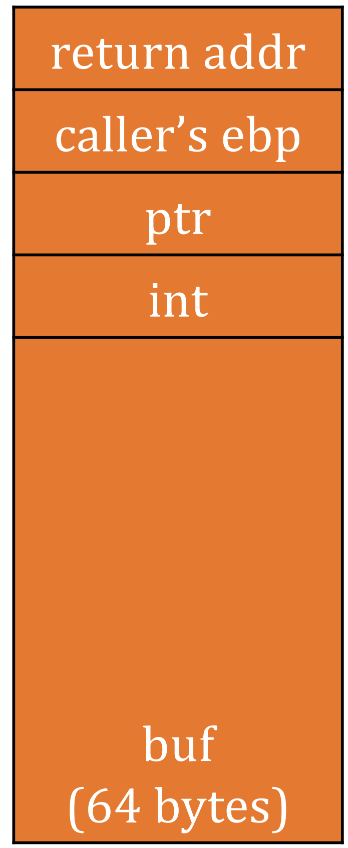
Non-Executable Stack

Traditionally, when you run a program, the stack memory region can be read, written, and executed

Allows us to load instructions onto the stack and execute them

NX defense makes the stack readable and writeable only

How is that done?



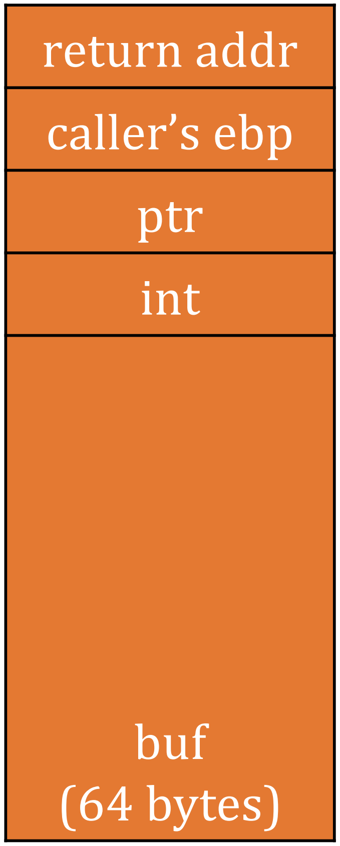
Non-Executable Stack

Traditionally, when you run a program, the stack memory region can be read, written, and executed

Allows us to load instructions onto the stack and execute them

NX defense makes the stack readable and writeable only

How is that done? Page perms



Disable DEP

- Can disable DEP
  - There's a system call to change page protections

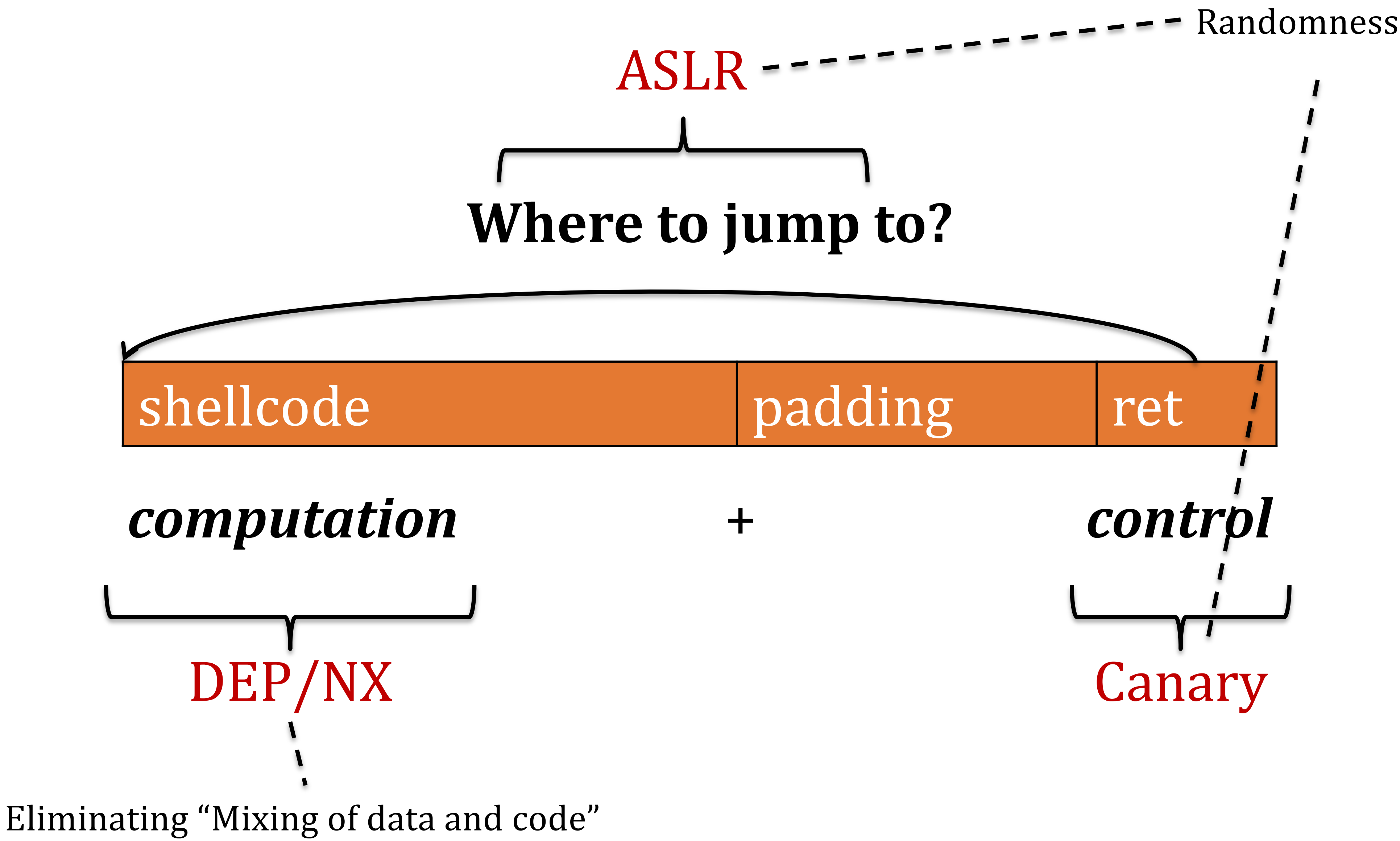
```
int mprotect(void *addr, size_t len, int prot);
```

  - Sets protection for region of memory starting at address
  - Invoke this system call to allow execution on stack and then start executing from the injected code
- But, need to get shellcode running to do this

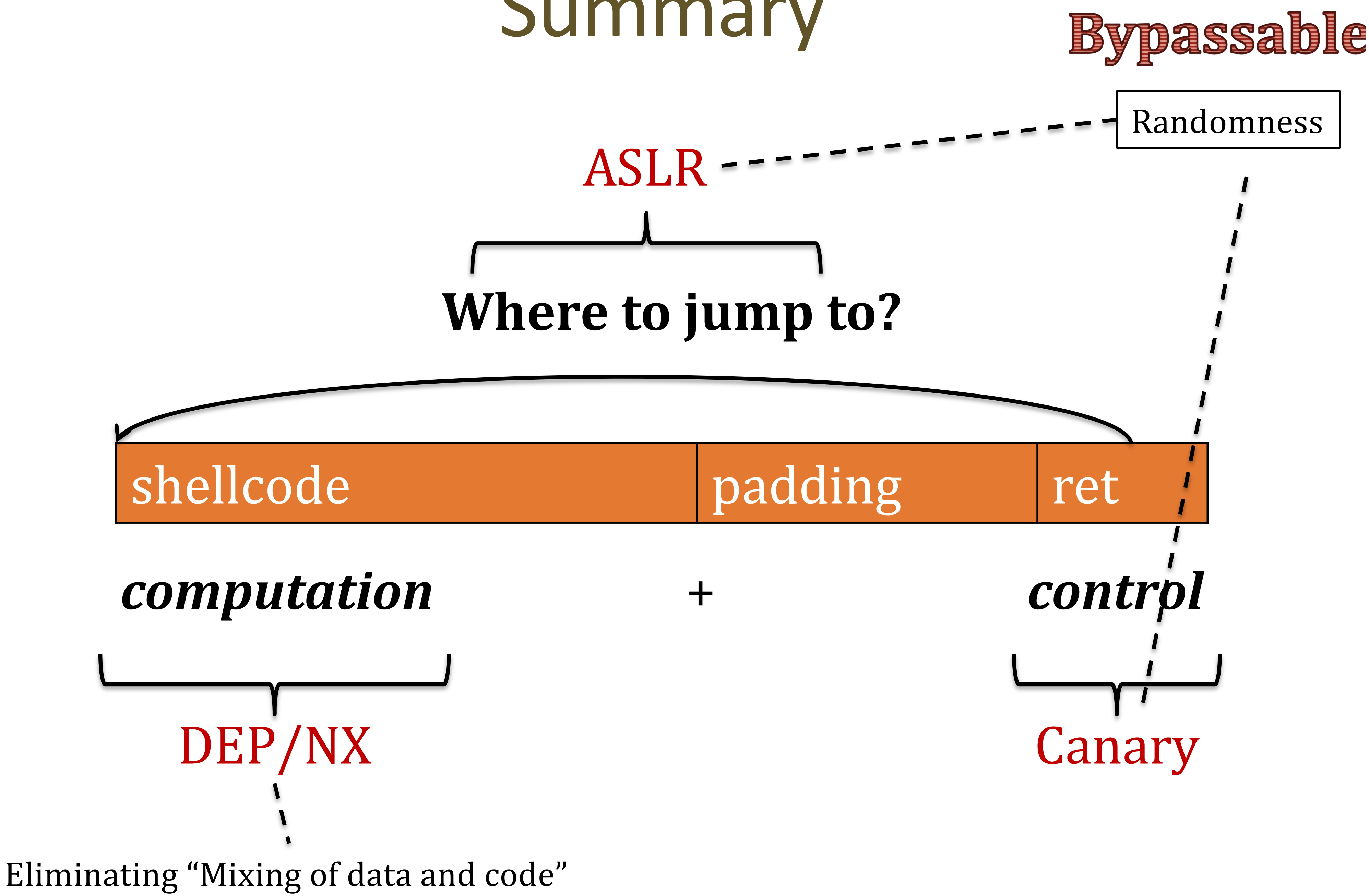
DEP Scorecard

Aspect	DEP
Performance	• already check page permissions • time: essentially none • size: one bit per page in page table
Deployment	• recompile suffices; no code change
Compatibility	• perfect—invisible to outside
Safety Guarantee	• *Better, but…*

Summary


Randomness
ASLR
Where to jump to?
shellcode
padding
ret
computation
+
control
DEP/NX
Canary
Eliminating "Mixing of data and code"

Summary



Bypassable
Randomness
ASLR
Where to jump to?
shellcode
padding
ret
computation
+
control
DEP/NX
Canary
Eliminating “Mixing of data and code”

Agenda

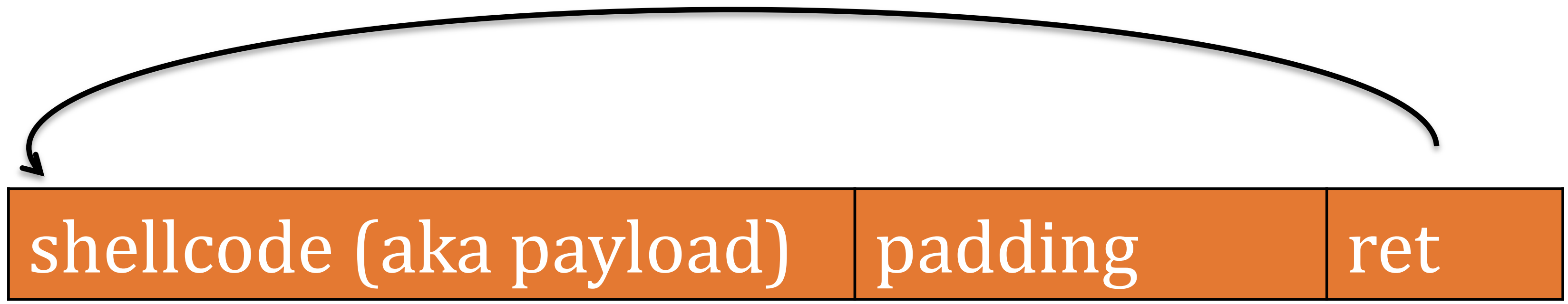
ROP Overview

Gadgets

Disassembling code

NEXT

Control Flow Hijack: Always control + computation

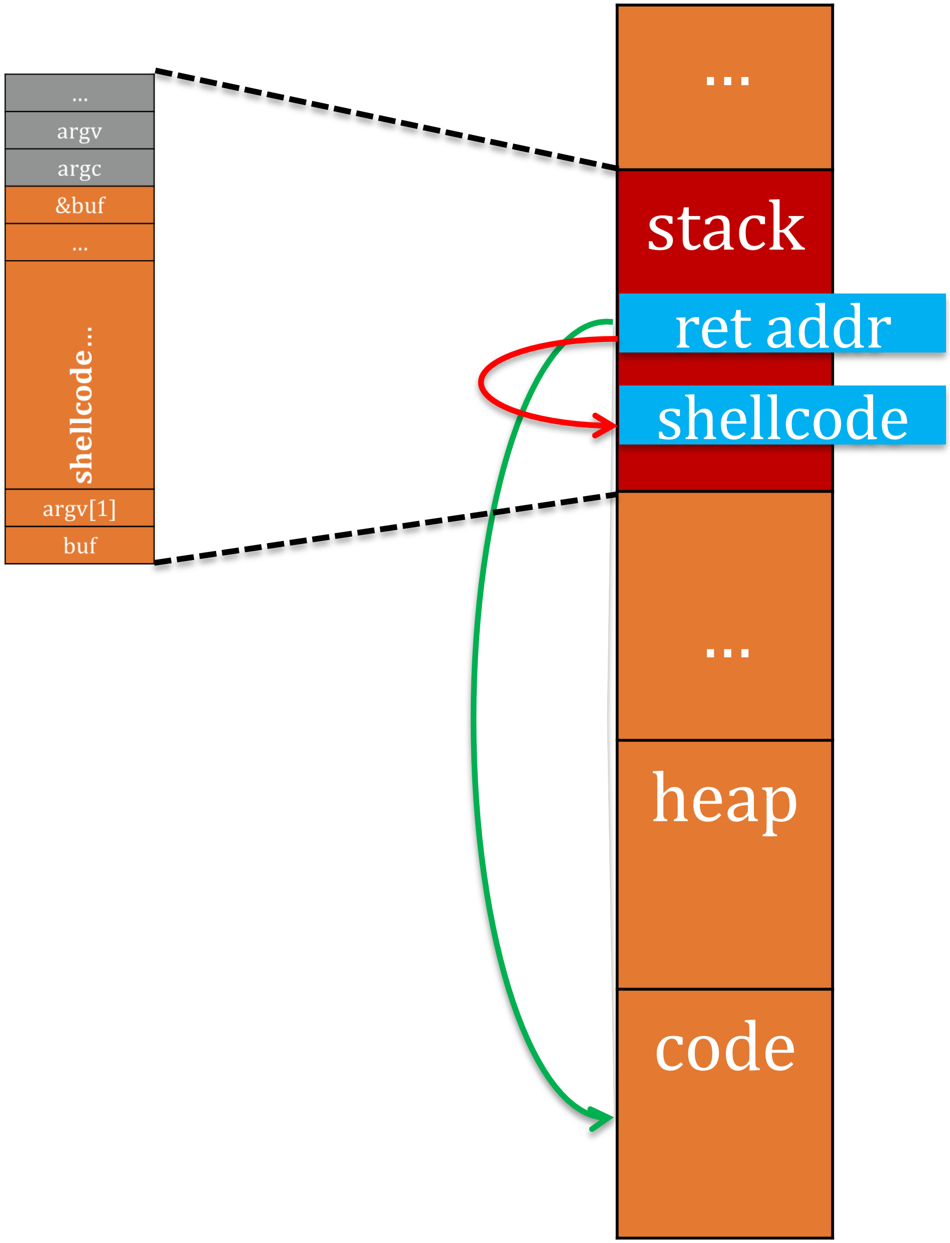


computation + ***control***

Return-oriented programming (ROP): shellcode without code injection

Acknowledgement: Some slides from David Brumley , Ed Schwartz, Kevin Snow, and Lucas Davi

Defenses Thwart Executing Shellcode



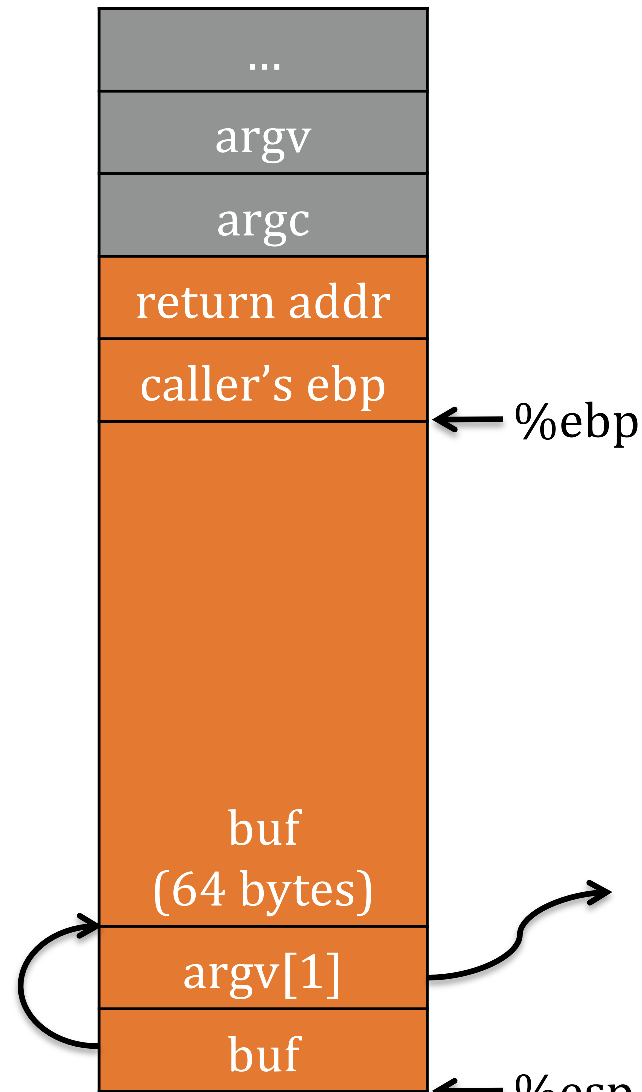
Motivation: Return-to-libc Attack

Bypassing DEP/NX!

Overwrite return address with address of libc function

- setup fake return address and argument(s)
- `ret` will "call" libc function

No injected code!

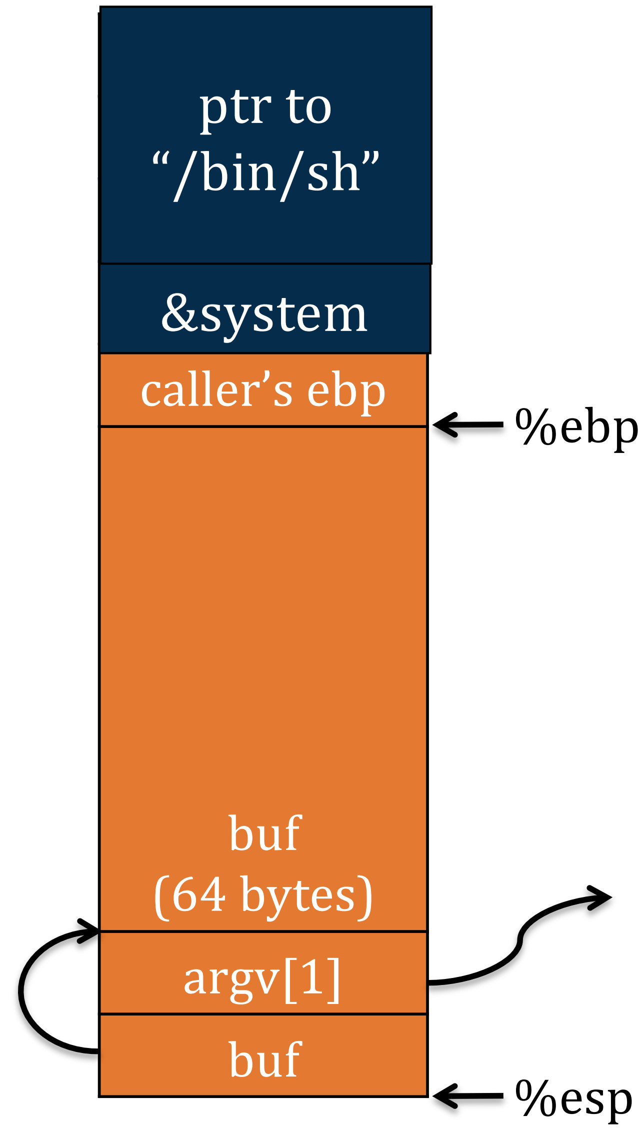


Motivation: Return-to-libc Attack

Overwrite return address with address of libc function

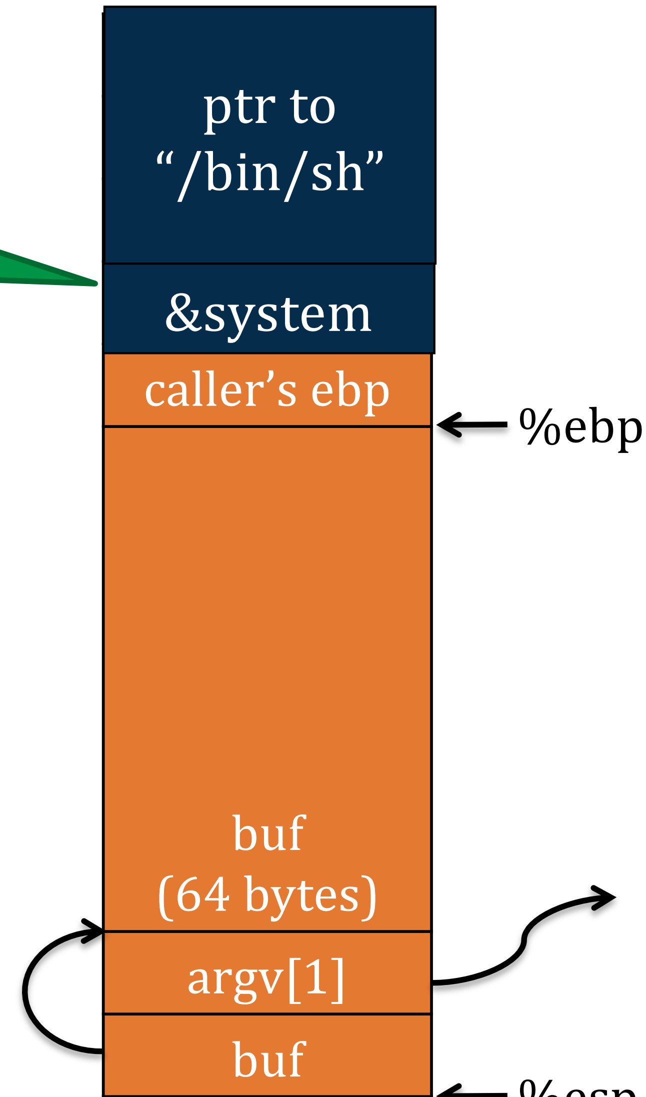
- setup fake return address and argument(s)
- `ret` will "call" libc function

No injected code!



Motivation: Return-to-libc Attack

ret transfers control to `system`, which finds arguments on stack



Overwrite return address with address of libc function

- setup fake return address and argument(s)
- `ret` will "call" libc function

No injected code!

Return Oriented Programming Techniques

Geometry of Innocent Flesh on the Bone

Shacham et al, ACM CCS 2007

The New Yo

Saturday, January 6, 2007

Daily Blog Tips awarded th

Last week Darren Rowse, from the famous Problogger blog, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among the Daily Blog Tips is attracting a vast audience of bloggers who are looking to improve their blogs. When asked about the success of his blog Daniel commented that

Ren
follc
imp

The
that
rela
the

The New Yo

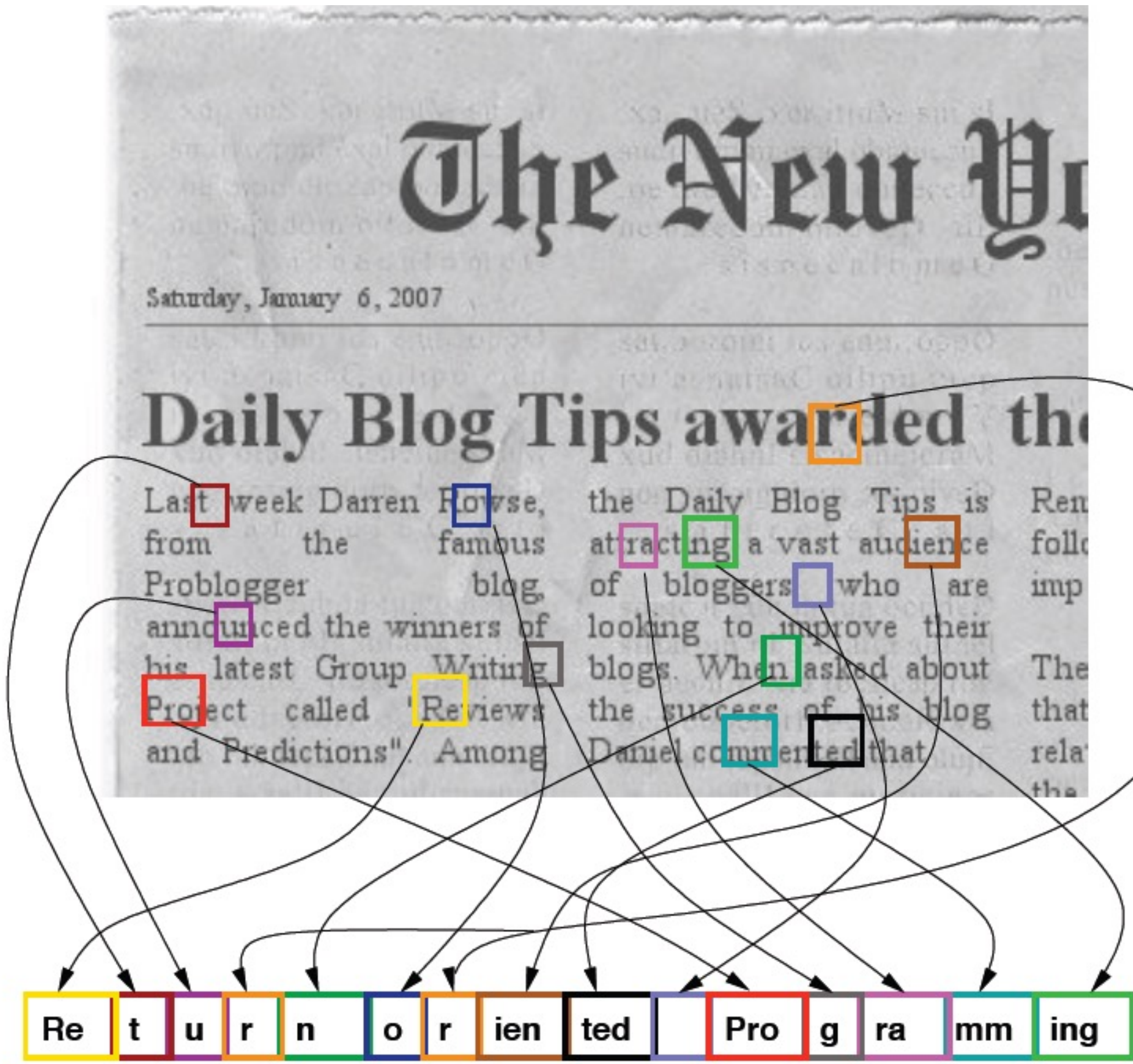
Saturday, January 6, 2007

Daily Blog Tips awarded the

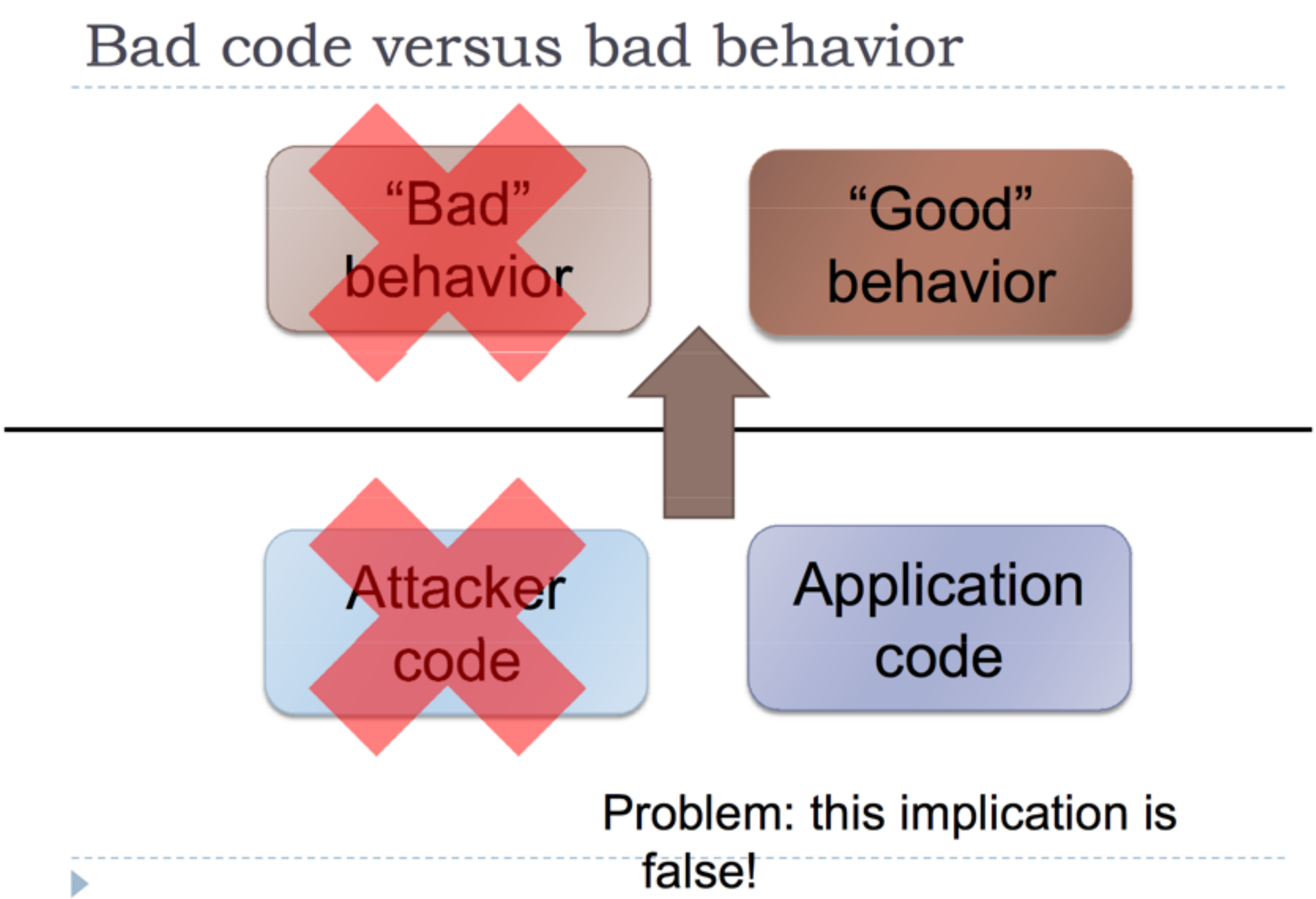
Last week Darren Rowse, from the famous Problogger blog, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among the Daily Blog Tips is attracting a vast audience of bloggers who are looking to improve their blogs. When asked about the success of his blog Daniel commented that

Ren
follo
imp

The
that
rela
the


The New Yo
Saturday, January 6, 2007
Daily Blog Tips awarded the
Last week Darren Rowse, from the famous Problogger blog, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among
the Daily Blog Tips is attracting a vast audience of bloggers who are looking to improve their blogs. When asked about the success of his blog Daniel commented that
Re
t
u
r
n
o
r
ien
ted
Pro
g
ra
mm
ing

ROP Programming



ROP Programming

attacker control of stack

arbitrary attacker computation and behavior
via return-into-libc techniques

(given any sufficiently large codebase to draw on)

ROP Programming: Key Steps

1. Disassemble code
2. Identify ***useful*** code sequences as gadgets
3. Assemble gadgets into desired shellcode

ROP Overview

- Idea: We forge shell code out of existing application logic gadgets

ROP Overview

- Idea: We forge shell code out of existing application logic gadgets

- Requirements:
  vulnerability + gadgets + some unrandomized code

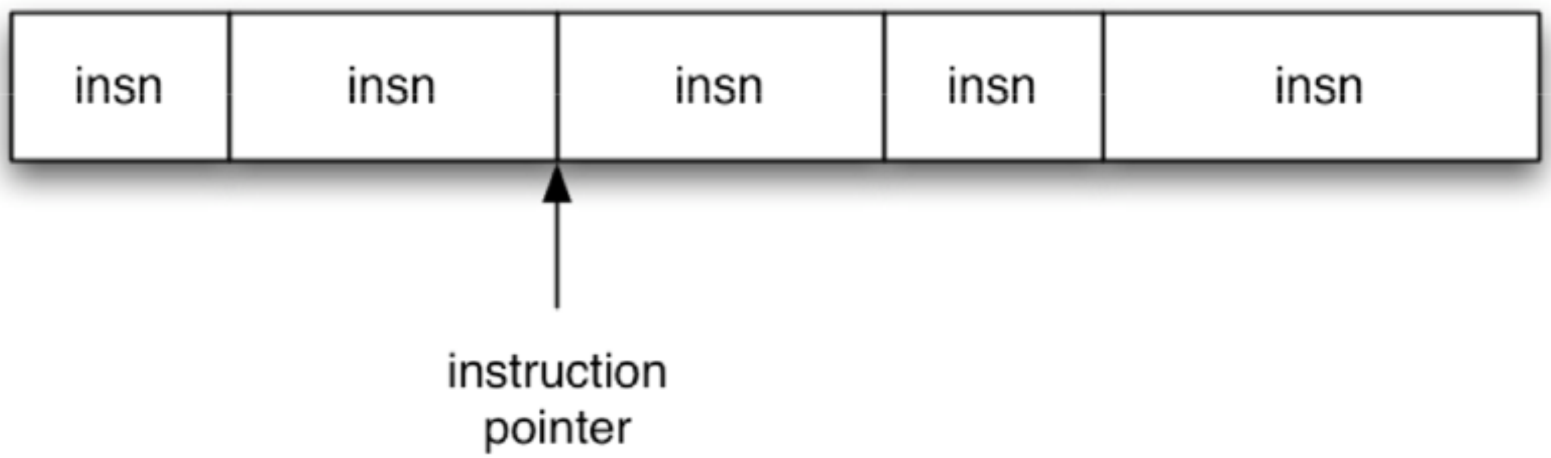
Agenda

ROP Overview ✓

Gadgets ← NEXT

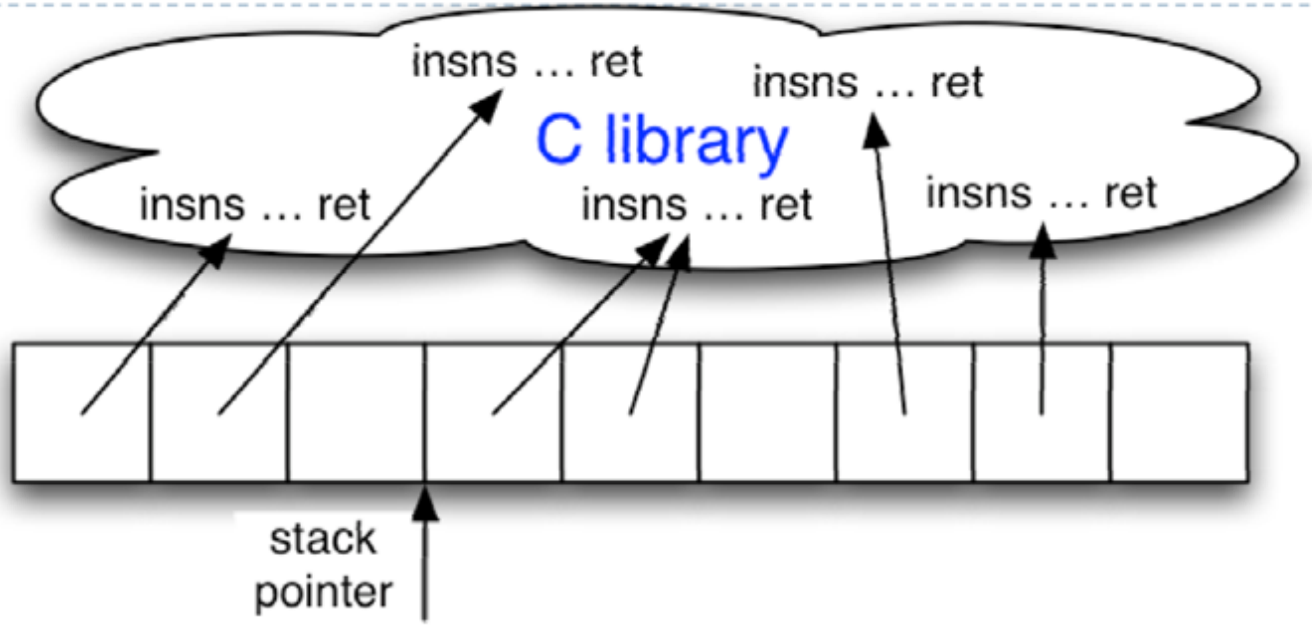
Disassembling code

Normal Execution



- The instruction pointer (%eip) determines which instruction to fetch and execute
- Once an instruction has executed, the processor automatically increments %eip to the next instruction

ROP Execution (On the Stack)



- The stack pointer (%esp) determines which instruction sequence to fetch and execute
- Once a sequence returns (executes ”ret”), the processor increments %esp to the next instruction sequence

Gadgets

A gadget is any instruction sequence ending with `ret`

Gadgets

`Mem[v2] = v1`

Desired Logic

Suppose a_3 and a_5 on stack

a_5
v_2
a_3
v_1 ← esp

Stack

eax	
ebx	
eip	a_1

Suppose ret address was set to a_1

```
a1: pop eax;
    ret
a3: pop ebx;
    ret
a5: mov [ebx], eax
    ret
```

Gadgets

```
Mem[v2] = v1
```

Desired Logic

a_5	
v_2	
a_3	
v_1	← esp

Stack

eax	v_1
ebx	
eip	a_1

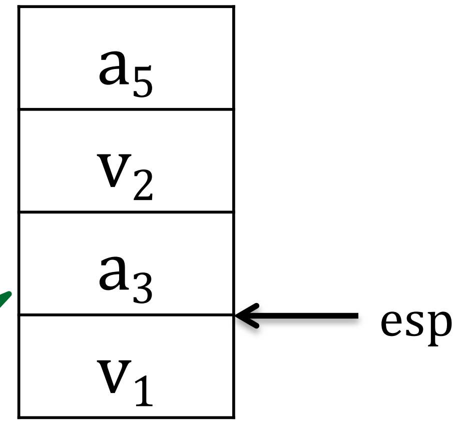
```
a1: pop eax;
    ret
a3: pop ebx;
    ret
a5: mov [ebx], eax
    ret
```

Gadgets

`Mem[v2] = v1`

Desired Logic

What happens next?



Stack

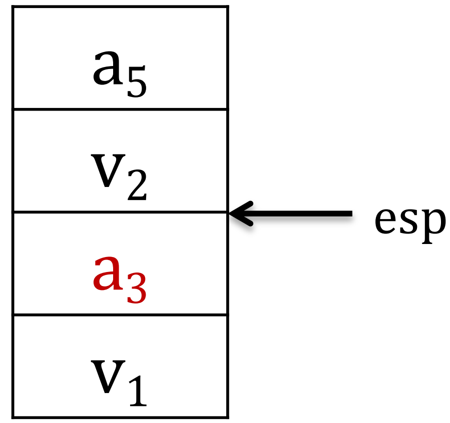
eax	v_1
ebx	
eip	a_2

```
a1: pop eax;
a2: ret
a3: pop ebx;
    ret
a5: mov [ebx], eax
    ret
```

Gadgets

`Mem[v2] = v1`

Desired Logic



Stack

eax	v_1
ebx	
eip	a_3

```
a1: pop eax;
     ret
a3: pop ebx;
     ret
a5: mov [ebx], eax
     ret
```

Gadgets

`Mem[v2] = v1`

Desired Logic

Stack	
a_5	
v_2	← esp
a_3	
v_1	

Stack

eax	v_1
ebx	
eip	a_3

```
a1: pop eax;
     ret
a3: pop ebx;
     ret
a5: mov [ebx], eax
     ret
```

Gadgets

```
Mem[v2] = v1
```

Desired Logic

a_5
v_2 ← esp
a_3
v_1

Stack

eax	v_1
ebx	v_2
eip	a_3

```
a₁: pop eax;
    ret
a₃: pop ebx;
    ret
a₅: mov [ebx], eax
    ret
```

Gadgets

`Mem[v2] = v1`

Desired Logic

What happens next?

Stack	
a_5	← esp
v_2	
a_3	
v_1	

Stack

eax	v_1
ebx	v_2
eip	a_4

a_1: `pop eax;`
`ret`
a_3: `pop ebx;`
a_4: `ret`
a_5: `mov [ebx], eax`
`ret`

Gadgets

Mem[v2] = v1

Desired Logic

a_5
v_2
a_3
v_1

esp

Stack

eax	v_1
ebx	v_2
eip	a_5

```
a₁: pop eax;
     ret
a₃: pop ebx;
     ret
a₅: mov [ebx], eax
     ret
```

Gadgets

`Mem[v2] = v1`

Desired Logic

What is the value at [v2]?

a_5	← esp
v_2	
a_3	
v_1	

Stack

eax	v_1
ebx	v_2
eip	a_5

```
a1: pop eax;
    ret
a3: pop ebx;
    ret
a5: mov [ebx], eax
    ret
```

Gadgets

`Mem[v2] = v1`

Desired Logic

Stack	
a_5	← esp
v_2	
a_3	
v_1	

Stack

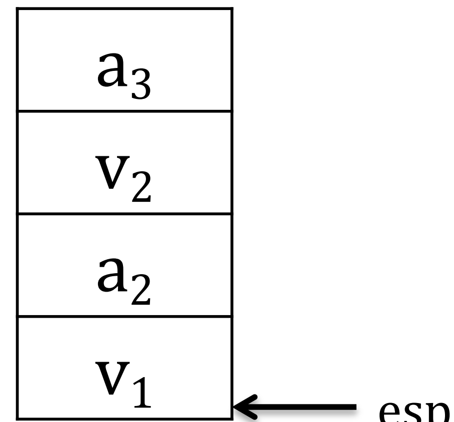
eax	v_1
ebx	v_2
eip	a_5

```
a1: pop eax;
     ret
a3: pop ebx;
     ret
a5: mov [ebx], eax
     ret
```

Equivalence

`Mem[v2] = v1`

Desired Logic



Stack

“Gadgets”

a_1: `pop eax; ret`

a_2: `pop ebx; ret`

a_3: `mov [ebx], eax; ret`

Implementation 2

Equivalence

`Mem[v2] = v1`

Desired Logic

Stack
a_3
v_2
a_2
v_1 ← esp

Stack

semantically equivalent

"Gadgets"

Implementation 1		Implementation 2
a_1: `mov eax, [esp]; ret`	⟷	a_1: `pop eax; ret`
a_2: `mov ebx, [esp+8]; ret`	⟷	a_2: `pop ebx; ret`
a_3: `mov [ebx], eax; ret`	⟷	a_3: `mov [ebx], eax; ret`

Implementation 1 **Implementation 2**

Equivalence

`Mem[v2] = v1`

Desired Logic

a_3
v_2
a_2
v_1

Stack

No need to be contiuous!

```
a1: pop eax; ret
...
a2: pop ebx; ret
...
a3: mov [ebx], eax
```

```
a1: pop eax; ret
a2: pop ebx; ret
a3: mov [ebx], eax
```

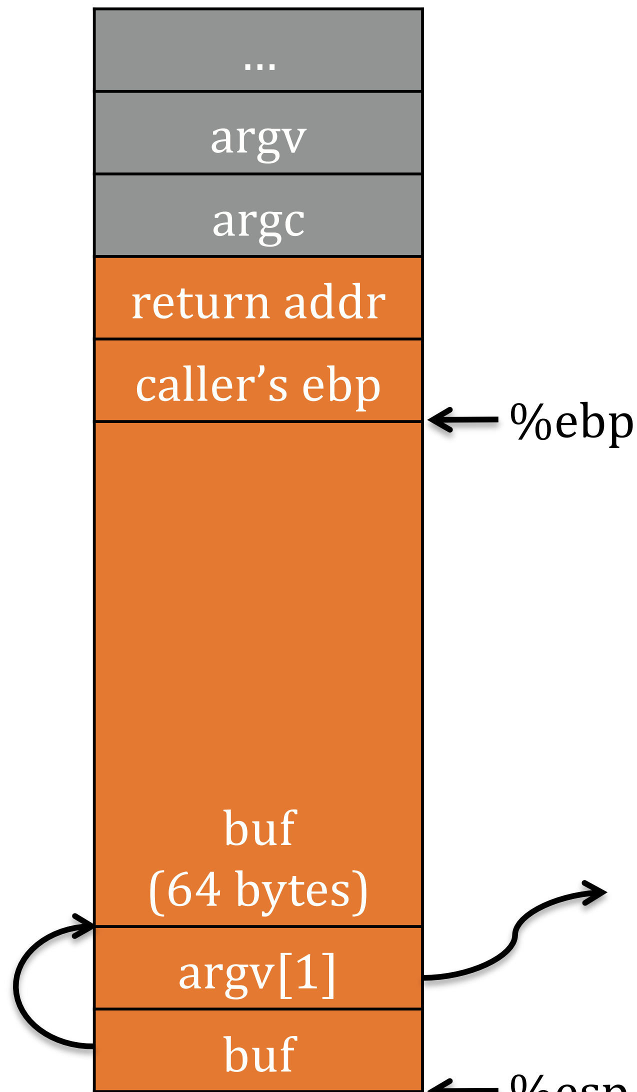
Implementation 2

Return-Oriented Programming (ROP)

```
Mem[v2] = v1
```

Desired *Shellcode*

- Find needed instruction gadgets at addresses a_1, a_2, and a_3 in *existing* code
- Overwrite stack to execute a_1, a_2, and then a_3



Return-Oriented Programming (ROP)

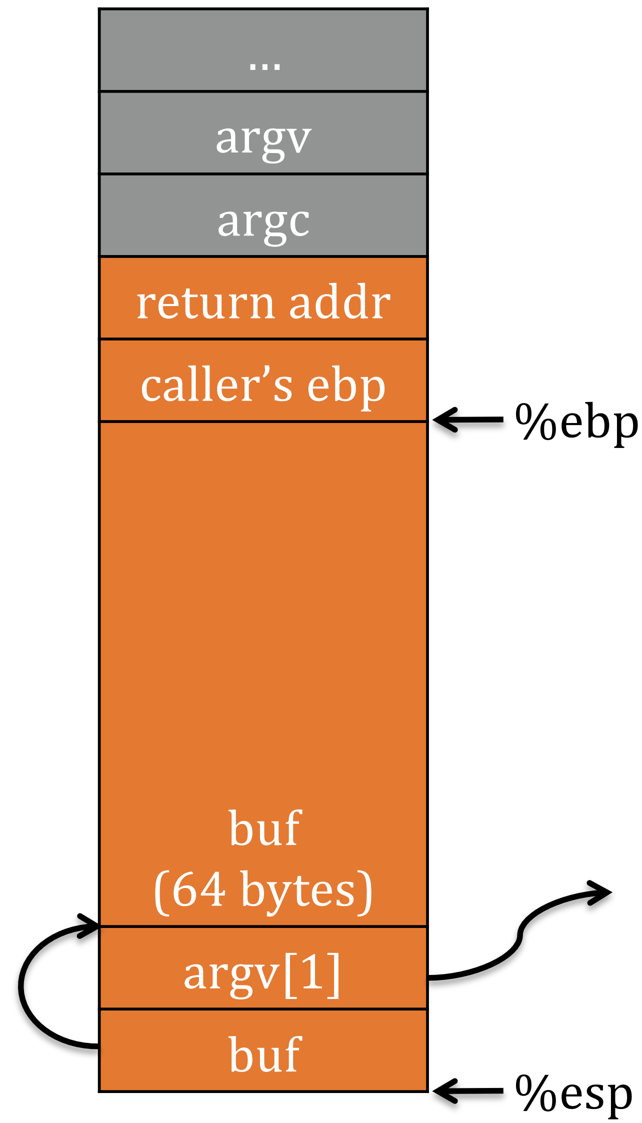
`Mem[v2] = v1`

Desired *Shellcode*

a_1: `pop eax; ret`

a_2: `pop ebx; ret`

a_3: `mov [ebx], eax; ret`

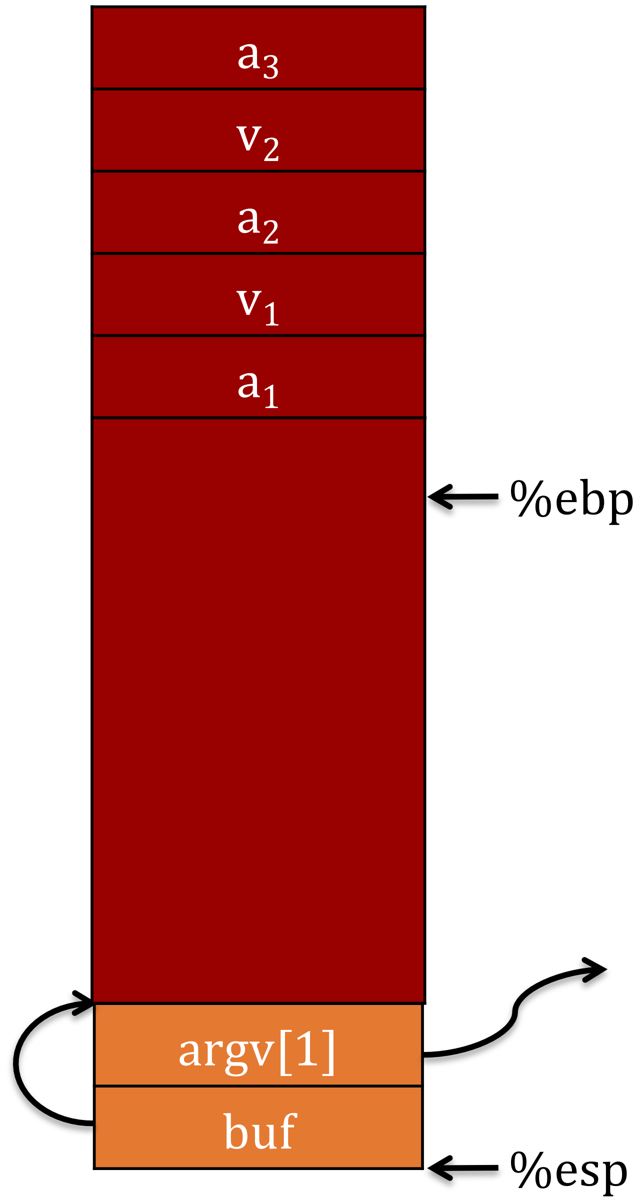


Return-Oriented Programming (ROP)

`Mem[v2] = v1`

Desired *Shellcode*

```
a1: pop eax; ret
a2: pop ebx; ret
a3: mov [ebx], eax; ret
```



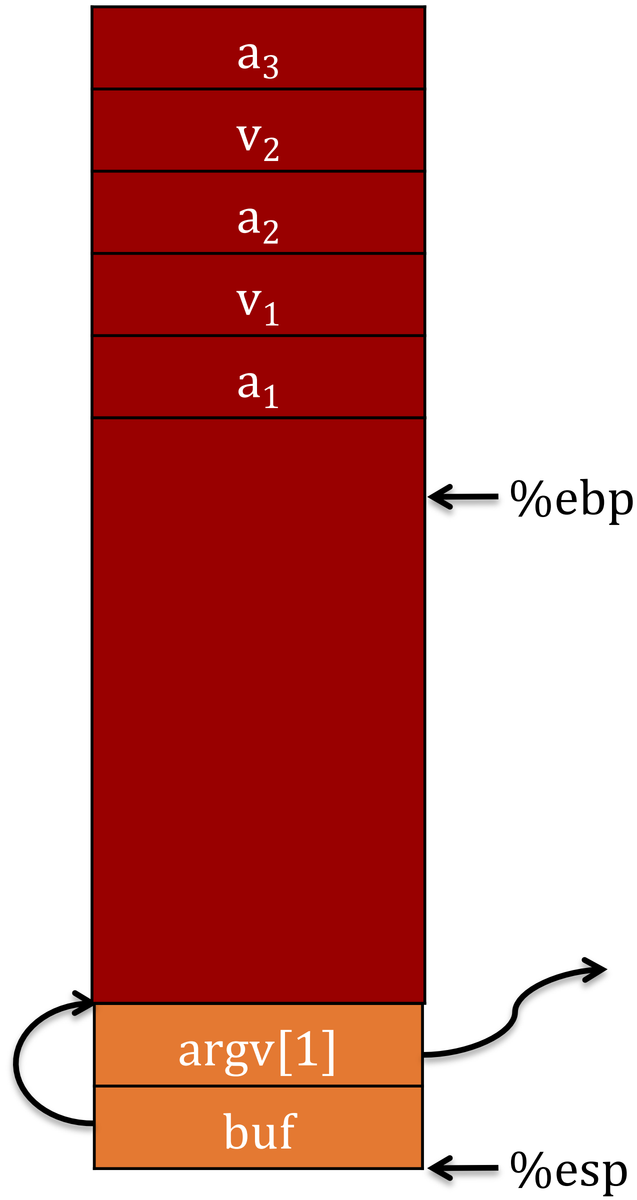
Return-Oriented Programming (ROP)

`Mem[v2] = v1`

Desired *Shellcode*

```
a₁: pop eax; ret
a₂: pop ebx; ret
a₃: mov [ebx], eax; ret
```

Desired store executed!



Finding Gadgets

- How do we build a complete exploit from available code?
  - Must find gadgets in that code
- How do you think one finds ***all*** the gadgets in a code region?

Finding Gadgets

- How do we build a complete exploit from available code?
  - Must find gadgets in that code
- How do you think one finds ***all*** the gadgets in a code region?
  - Find sequence of instructions until a ”ret” is reached
    - Find “a, b, c, ret” – where a, b, and c are other instructions

Finding Gadgets

- How do we build a complete exploit from available code?
  - Must find the gadgets that are available in that code
- How do you think one finds ***all*** the gadgets in a code region?
  - Start from a "ret" byte "0xc3" at any memory location and work backwards to find the longest useful sequence of instructions for a gadget
    - Find "a, b, c, ret" – find "c, ret", then "b, c, ret", then…

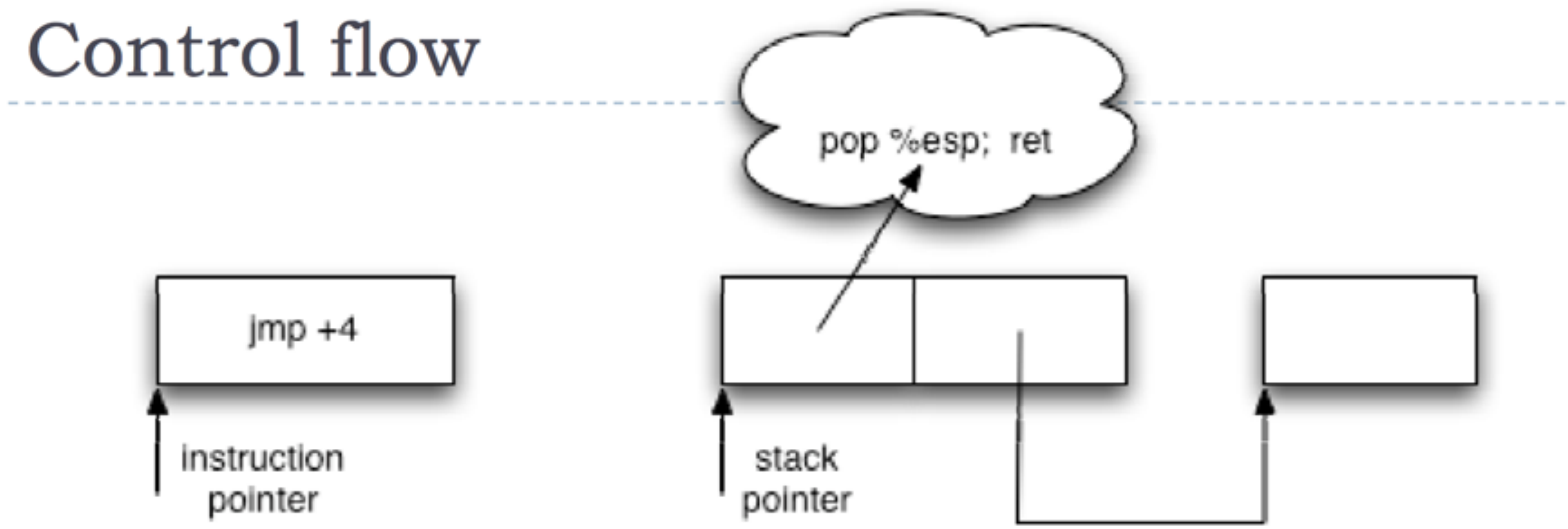
Expressability

- What can we do with return-oriented programming?
  - Anything any other program can do
  - How do we know?

Expressability

- What can we do with return-oriented programming?
  - Anything any other program can do
  - How do we know? Turing completeness
- A language is Turing complete if it has (loosely)
  - Conditional branching
  - Can change memory arbitrarily
- Both are possible with ROP

Expressability



- Cause a control flow change by popping a value from the stack into %esp
  - Executing from that location going forward

Conclusions

- New defenses make code injection impractical
  - Especially DEP/NX (e.g., non-executable stack)
- Another attack option for adversaries
  - Return-oriented programming: Use the code that is already there rather than injecting new code
- If there is enough code, ROP is just as powerful as code injection attacks
  - Turing complete computation
- More defenses are needed (later)

Quiz

```
void foo(char *input){
    char buf[512];
    ...
    strcpy (buf, input);
    return;
}
```

Draw a stack diagram for a ROP exploit to **pop a value 0xBBBBBBBB into eax and add 0x80 to it.**

a_1: add eax, 0x80; pop ebp; ret

a_2: pop eax; ret

Known Gadgets

Quiz

```
void foo(char *input){
    char buf[512];
    ...
    strcpy (buf, input);
    return;
}
```

Draw a stack diagram for a ROP exploit to **pop a value 0xBBBBBBBB into eax and add `0x80` to it.**

a_1: add eax, 0x80; pop ebp; ret

a_2: pop eax; ret

Known Gadgets

Stack: a_2; 0xBBBBBBBB; a_1

Quiz

a_1: pop ebx; ret

a_2: pop eax; ret

a_3: mov eax, (ebx); ret

a_4: mov ebx, (eax); ret

a_5: add eax, (ebx); ret

a_6: push ebx; ret

a_7: pop esp; ret

Known Gadgets

Draw a stack diagram for a ROP exploit to **store the value `0xBBBBBBB` into address `0xAAAAAAA`.**

Quiz

a_1: pop ebx; ret

a_2: pop eax; ret

a_3: mov eax, (ebx); ret

a_4: mov ebx, (eax); ret

a_5: add eax, (ebx); ret

a_6: push ebx; ret

a_7: pop esp; ret

Known Gadgets

Draw a stack diagram for a ROP exploit to **store the value `0xBBBBBBB` into address `0xAAAAAAA`.**

a1; 0xBBBBBBB; a2; 0xAAAAAAA; a4 or
a2; 0xBBBBBBB; a1; 0xAAAAAAA; a3

Quiz

a_1: pop ebx; ret

a_2: pop eax; ret

a_3: mov eax, (ebx); ret

a_4: mov ebx, (eax); ret

a_5: add eax, (ebx); ret

a_6: push ebx; ret

a_7: pop esp; ret

Known Gadgets

Draw a stack diagram for a ROP exploit to **store the value `0xBBBBBBB`+1 into address `0xAAAAAAA`**

Quiz

a_1: pop ebx; ret

a_2: pop eax; ret

a_3: mov eax, (ebx); ret

a_4: mov ebx, (eax); ret

a_5: add eax, (ebx); ret

a_6: push ebx; ret

a_7: pop esp; ret

Known Gadgets

Draw a stack diagram for a ROP exploit to **store the value `0xBBBBBBB+1` into address `0xAAAAAAA`**

a2; 0xBBBBBBB; a1; 0xAAAAAAA; a3; a2; 1; a5

Quiz

a_1: pop ebx; ret

a_2: pop eax; ret

a_3: mov eax, (ebx); ret

a_4: mov ebx, (eax); ret

a_5: add eax, (ebx); ret

a_6: push ebx; ret

a_7: pop esp; ret

Known Gadgets

Draw a stack diagram for a ROP exploit to **store the value `0xBBBBBBB+1` into address `0xAAAAAAA`** – **then execute from `0xBBBBBBB+1`**

Quiz

a_1: pop ebx; ret

a_2: pop eax; ret

a_3: mov eax, (ebx); ret

a_4: mov ebx, (eax); ret

a_5: add eax, (ebx); ret

a_6: push ebx; ret

a_7: pop esp; ret

Known Gadgets

Draw a stack diagram for a ROP exploit to **store the value `0xBBBBBBB+1` into address `0xAAAAAAA` – then execute from `0xBBBBBBB+1`**

Assume 0xBBBBBBB+1 is stored at 0xAAAAAAA
a7; 0xAAAAAAA or a6; a7; (if ebx has 0xAAAAAAA)

Questions