Runtime Attacks and Defenses

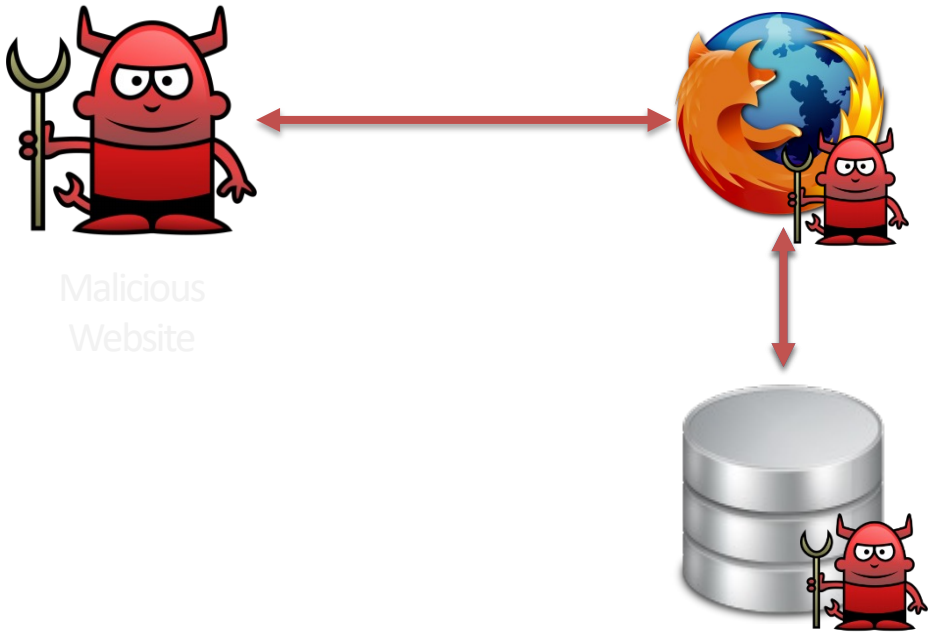
Based on Slides by Lucas Davi

https://www.syssec.wiwi.uni-due.de/team/lucas-davi/

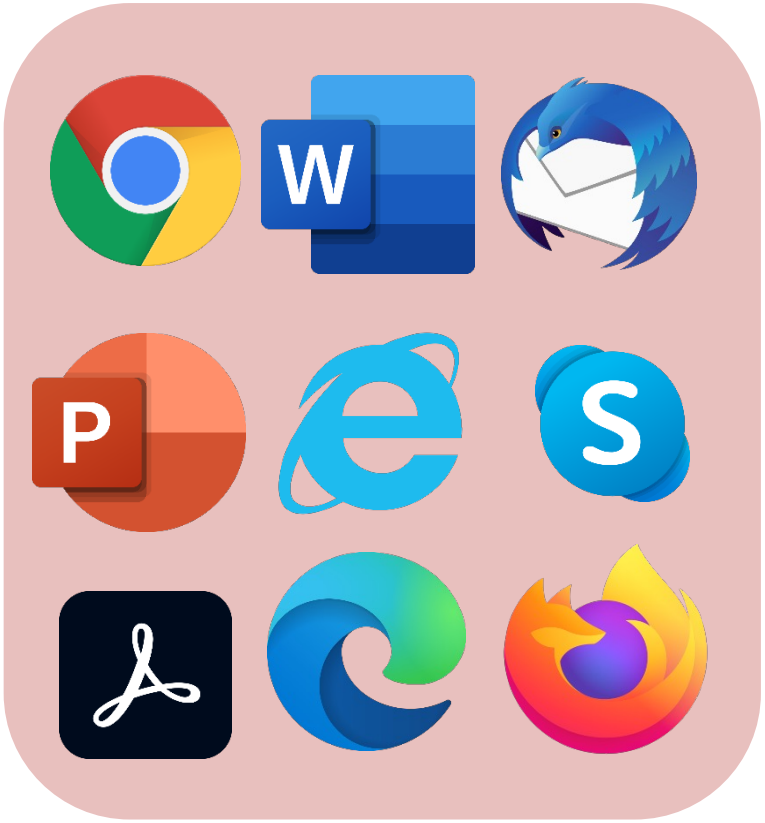
Motivational Example

A browser has access to:

- Passwords, Browsing History, Credit Card data …
- (Typically) All files on the computer



Motivation



- Sophisticated
- Millions of Lines of Code
- Have access to sensitive data
- Handle Untrusted Data

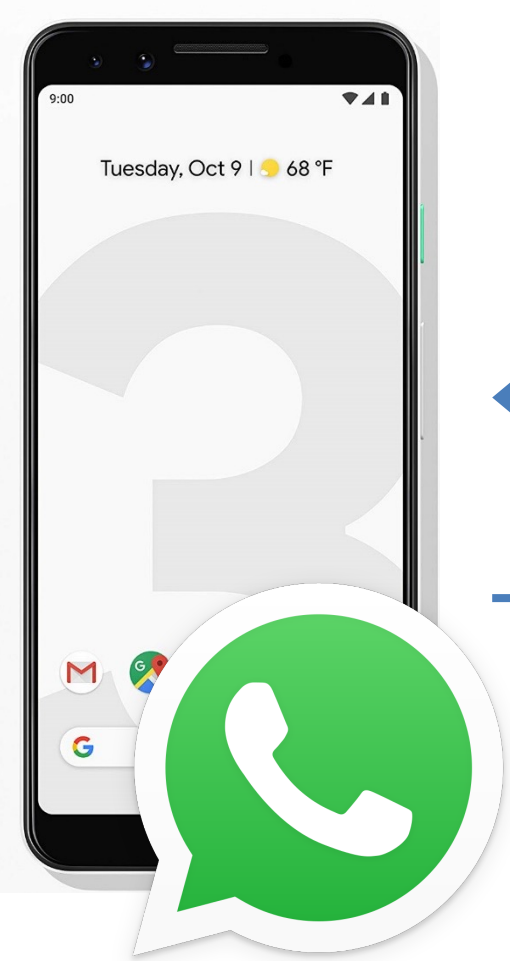
$\Rightarrow$ Good attack targets

Introduction

- Vulnerabilities
  - Programs continuously suffer from program bugs, e.g., a buffer overflow
  - Memory errors
  - CVE statistics; zero-day
- Runtime Attacks

Are runtime attacks really important?

WhatsApp remote code execution



Specially crafted voice call triggers vulnerability

Data leaked to adversary

https://www.ft.com/content/4da1117e-756c-11e9-be7d-6d846537acab

Some Recent Runtime Attacks

2017-10 Security Update for Adobe Flash Player for Windows 10 Version 1709 for x64-based Systems (KB4049179)
Last Modified: 10/16/2017
Size: 20.8 MB

Details:

Overview | Language Selection | Package Details | Install Resources

by

MSRC Number: n/a
MSRC severity Critical

Security

Here's a timeless headline: Adobe rushes out emergency Flash fix after hacker exploits bug

So much for that security-patch-free Octo

By Iain Thomson in San Francisco 16 Oct 2017 at 18:39

Security

It's 2017... And Windows PCs can be pwned via DNS, webpages, Office docs, fonts – and some TPM keys are fscked too

But at least there's no Flash update (not this week, anyway)

By Shaun Nichols in San Francisco 10 Oct 2017 at 22:22 59 SHARE ▼

Security

Sounds painful: Audio code bu users, apps get root on Linux

Cisco discusses Advanced Linux Sound Architecture mess before formal CVE release

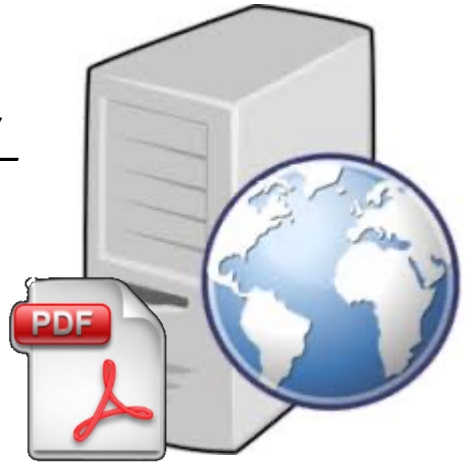
By Richard Chirgwin 15 Oct 2017 at 23:39 26 SHARE ▼

But runtime attacks have also some “good” side-effects

Apple iPhone Jailbreak

Disable signature verification and escalate privileges to root

Request
http://www.jailbreakme.com/_/iPhone3,1_4.0.pdf



1) Exploit PDF Viewer Vulnerability by means of **Return-Oriented Programming**
2) Start Jailbreak
3) Download required system files
4) Jailbreak Done

BASICS
What is happening in memory?

Outline of This Lecture

BASICS

- How does a program execute in memory?
- What is a runtime attack?

CODE INJECTION ATTACKS

- How do they work?
- How can they be prevented?

CODE REUSE ATTACKS

- Different types of attacks
- Possible defenses

Basic Program execution

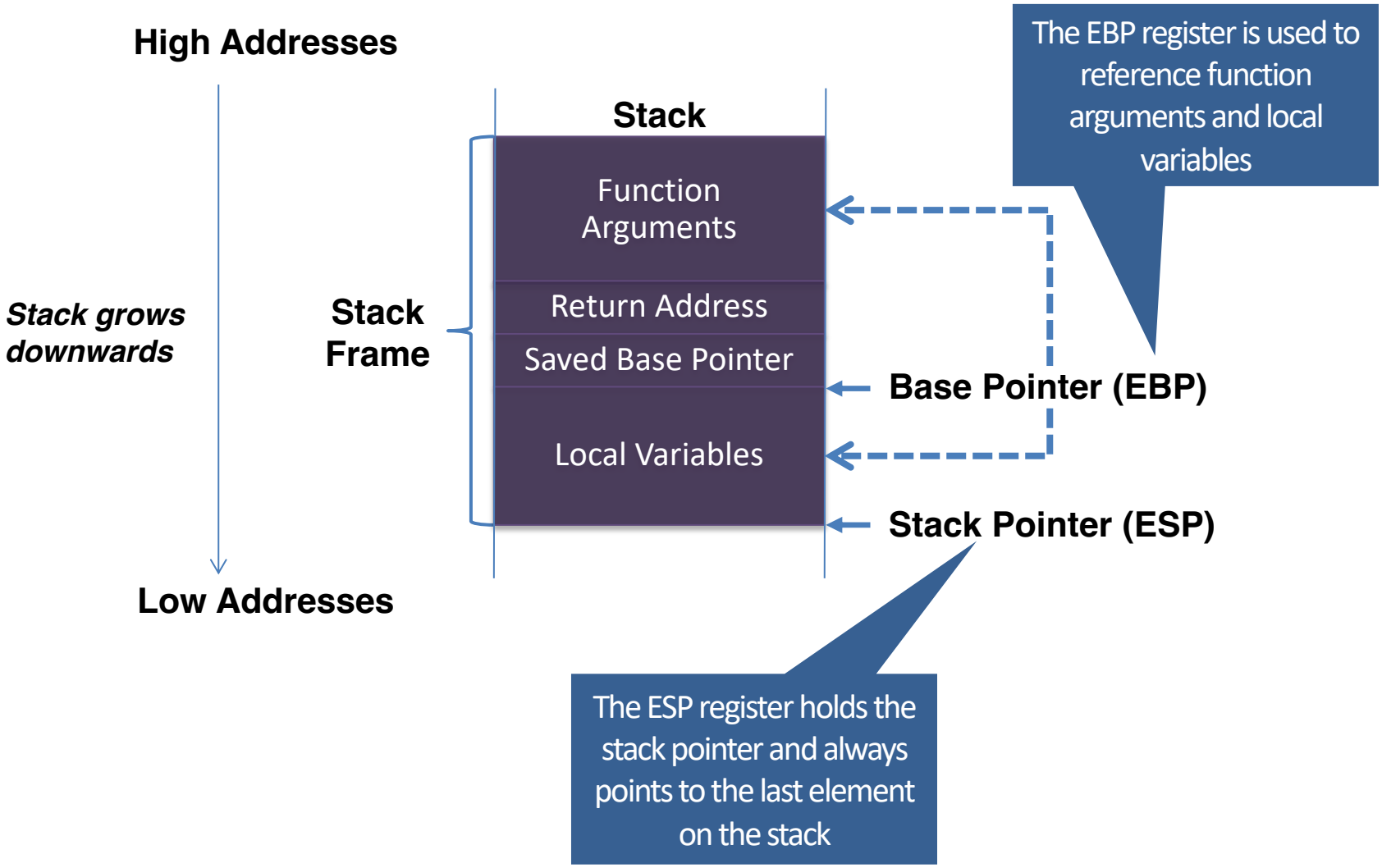
- Next instruction is in Instruction Pointer (EIP)
- Ability to overwrite own and previous stack frames:
  Local Variables => Modify program state
  Stored Instruction Pointers => Modify control flow

- When a function is called a new stack frame is created
  - Contains local variables
  - Contains location of previous stack frame (Saved Base Pointer)
  - Contains instruction pointer of calling function (Return Address)

  $\Rightarrow$ where to resume execution
- Pop = Move element from stack to register
- Push = Move register content on top of stack
- For more details, please review the exercise slides

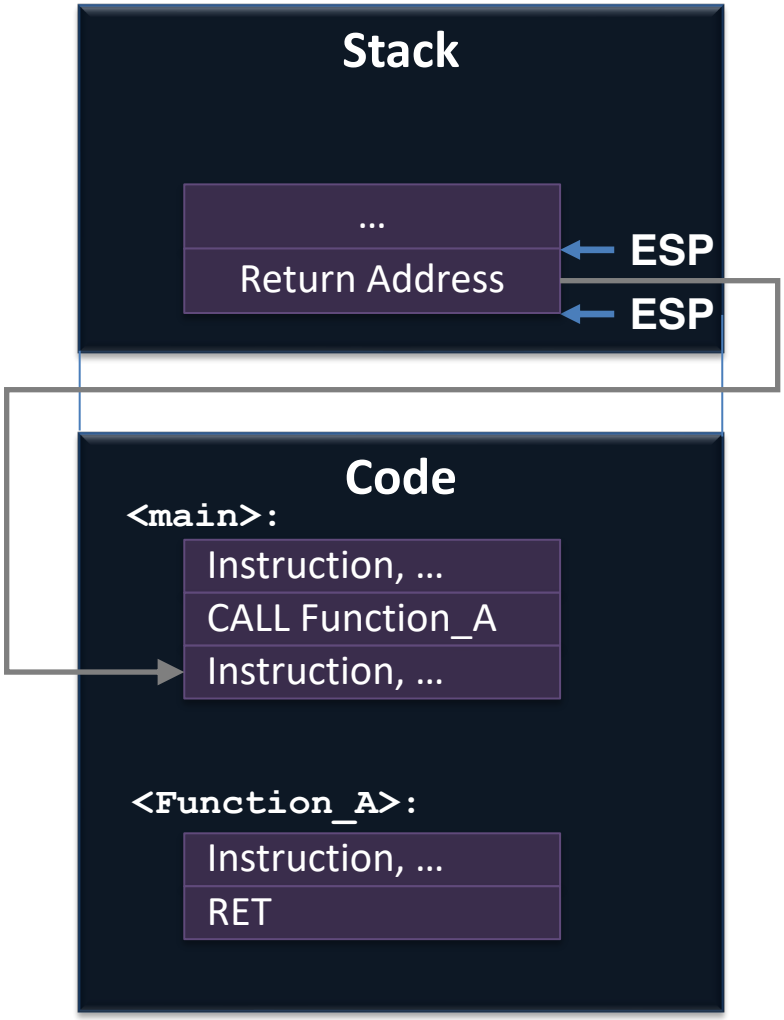
Stack Frame

Each function is associated with one stack frame on the stack



Calling Convention (on Intel x86)

- Function call performed via the x86 CALL instruction
  - E.g., CALL Function_A
  - The CALL instruction automatically pushes the return address on the stack, while the return address simply points to the instruction after the call
- Function return is performed via the x86 RET instruction
  - The RET instruction pops the return address off the stack and loads it into the instruction pointer (EIP)
  - Hence, the execution will continue in the main function



How is Stack Frame created?

- Every function needs to setup a stack frame

$\Rightarrow$ Compiler adds a function prologue and function epilogue

Function prologue:

- Save previous stack frame
- Create new stack frame

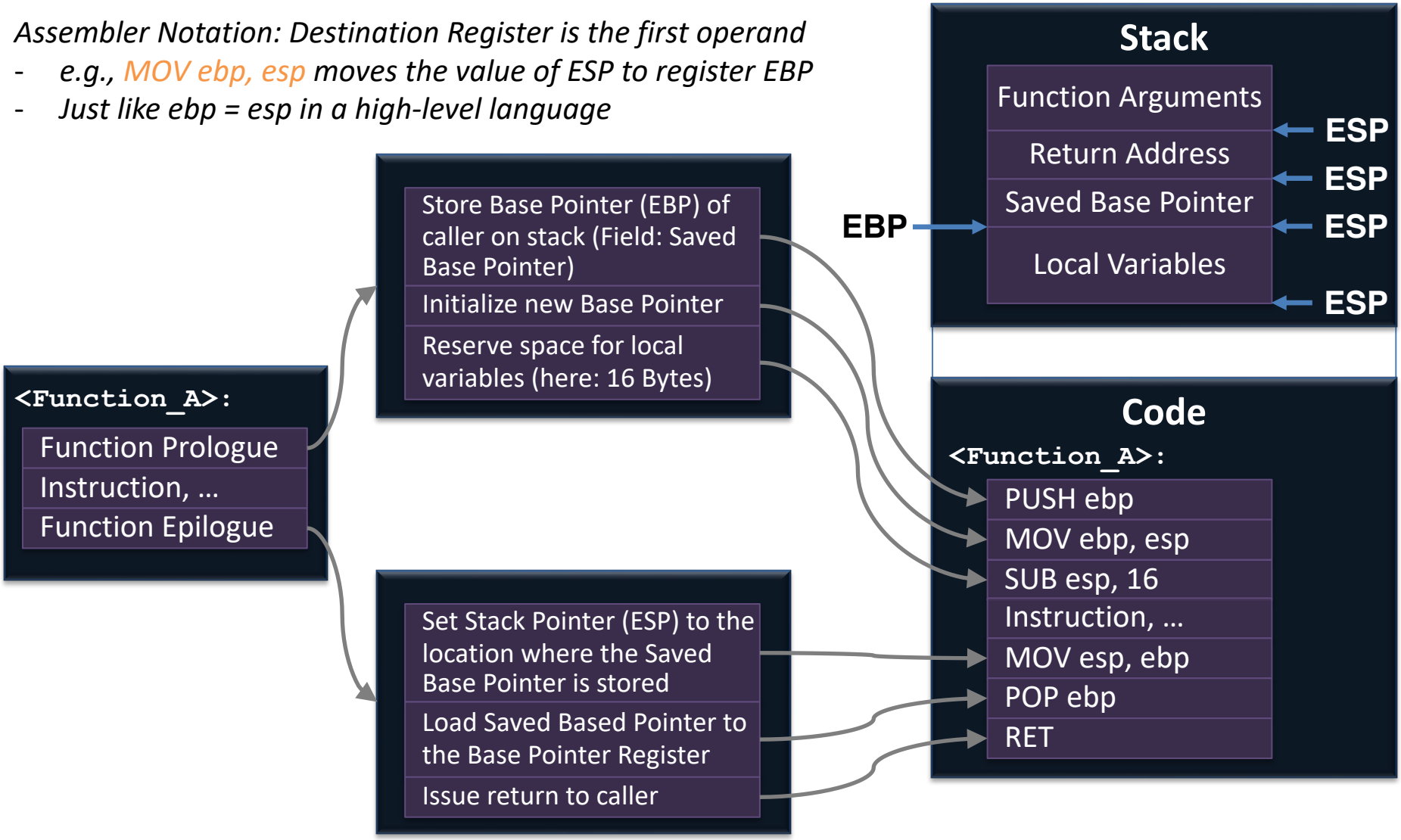
Function epilogue:

- Remove stack frame
- Restore previous stack frame

Function Prologue and Epilogue by Example

Assembler Notation: Destination Register is the first operand

- *e.g., MOV ebp, esp moves the value of ESP to register EBP*
- *Just like ebp = esp in a high-level language*



BASICS
What is a runtime attack ?

Big Picture – Buffer Overflow

Source Code
C

*COPY (buffer[8], *usr_input)*

Copies

Copies User Input
into fixed-size buffer

Compile

Executable
binary

MOV …
MOV …
MOV …

reg0 → buffer[8]

reg1 → usr_input

Buffer Overflow: Program Execution

Executable **binary**

MEMORY - RAM

Initialize buffer[8]

Get usr_input

COPY (buffer[8], *usr_input)

CODE

DATA

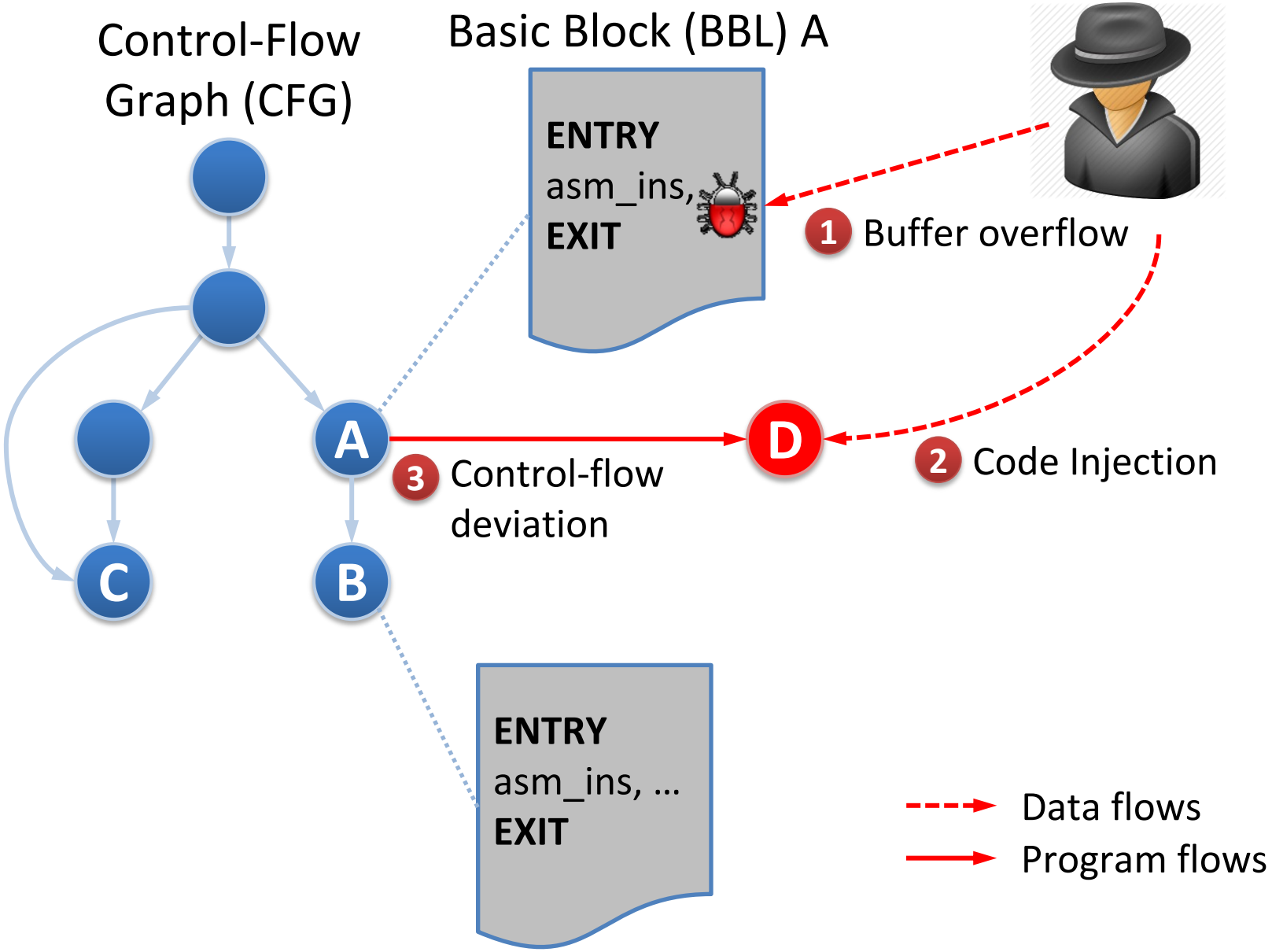
POINTER:	8000ABCD
buffer[4-7]:	00000000
buffer[0-3]:	00000000
...	...
usr_input[8-11]:	CCCCCCCC
usr_input[4-7]:	BBBBBBBB
usr_input[0-3]:	AAAAAAAA

Observations

- There are several observations
  1. A programming error leads to a program-flow deviation
  2. Missing bounds checking
     - Languages like C, C++, or assemby do not automatically enforce bounds checking on data inputs
  3. An adversary can provide inputs that influence the program flow

- What are the consequences?

General Principle of Code Injection Attacks



Code Injection Attacks

- Attacker changes control flow

$\Rightarrow$ Program does what attacker wants

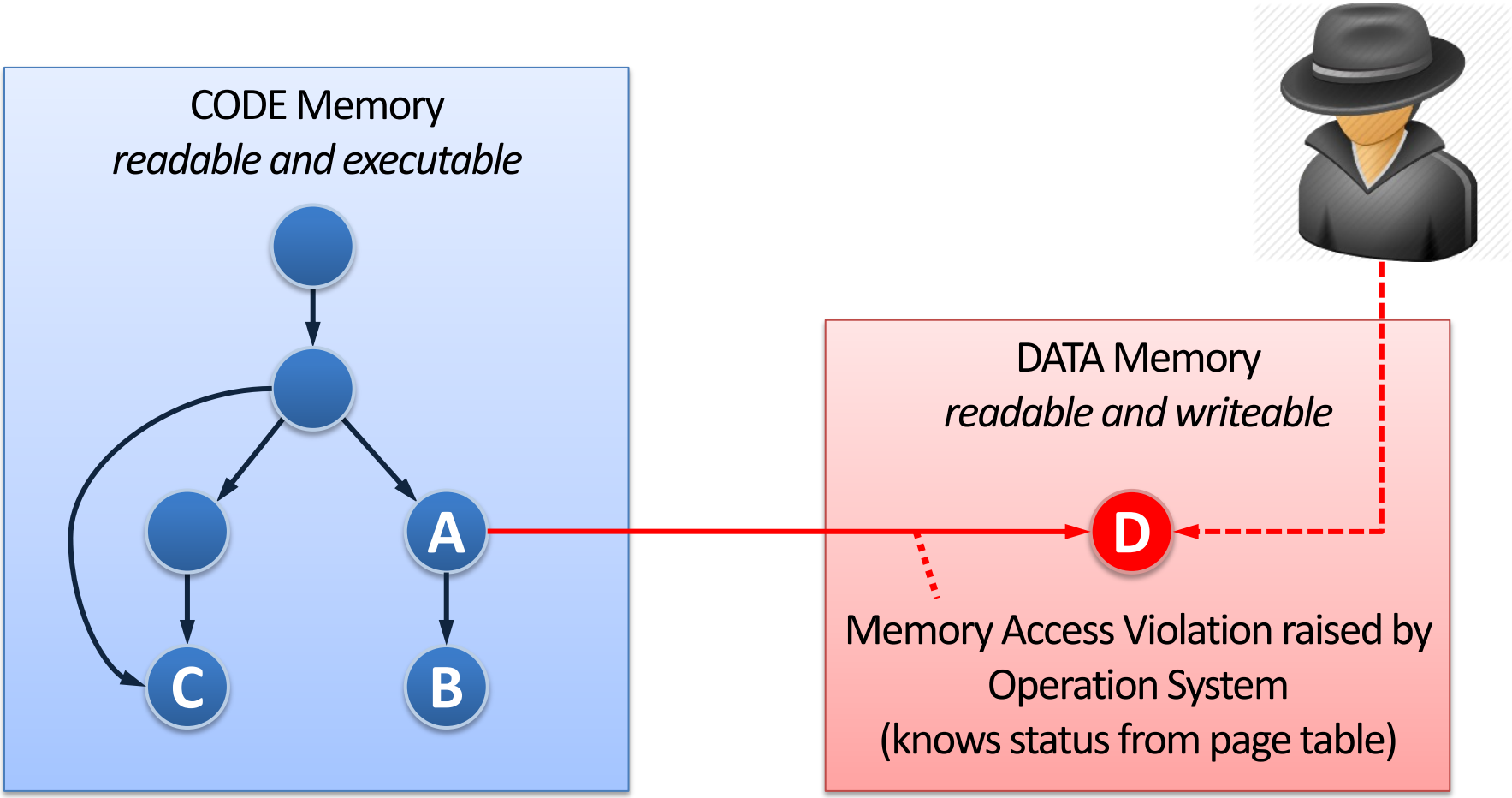
- Very powerful
- Attacker-injected code

$\Rightarrow$ Power to run arbitrary code

- Typically the code is written directly to the stack
- How do we defend against this?

Data Execution Prevention (DEP)

- Idea
  - Prevent execution from a writeable memory (data) area



Data Execution Prevention (DEP) cntd.

- Implementations
  - Modern OSes enable DEP by default (Windows, Linux, iOS, Android, Mac OSX)
  - Intel, AMD, and ARM feature a special No-Execute bit to facilitate deployment of DEP

  $\Rightarrow$ Hardware-Support makes it more efficient
- Side Note
  - There are other notions referring to the same principle
    - W $\oplus$ X – Writeable XOR eXecutable
    - Non-executable memory
    - NX-Bit

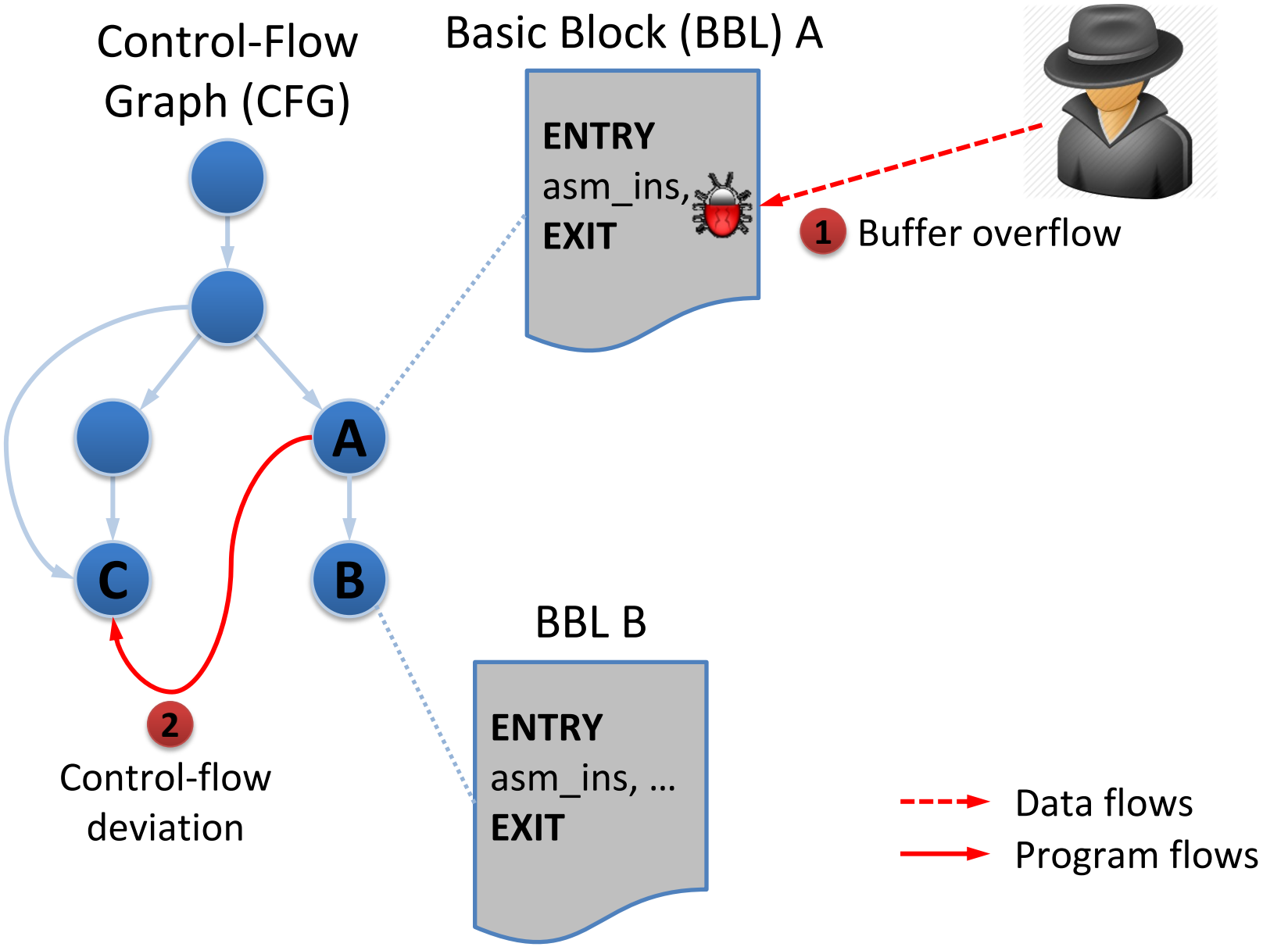
What can the attacker do now?

- Attacker cannot inject own code due to DEP

$\Rightarrow$ Reuse existing code

$\Rightarrow$ Code Reuse Attacks

General Principle of Code Reuse Attacks



What code to reuse?

- Attacker cannot inject own code
- What code can the attacker reuse?
- Goals:
  - Available in all programs
  - Allows execution of arbitrary commands

⇒ **Libraries (in particular libc)**

- **Included in almost all programs**
- **Mapped into memory during execution**

⇒ **Return-to-libc attack (will be done in exercise)**

return-to-libc

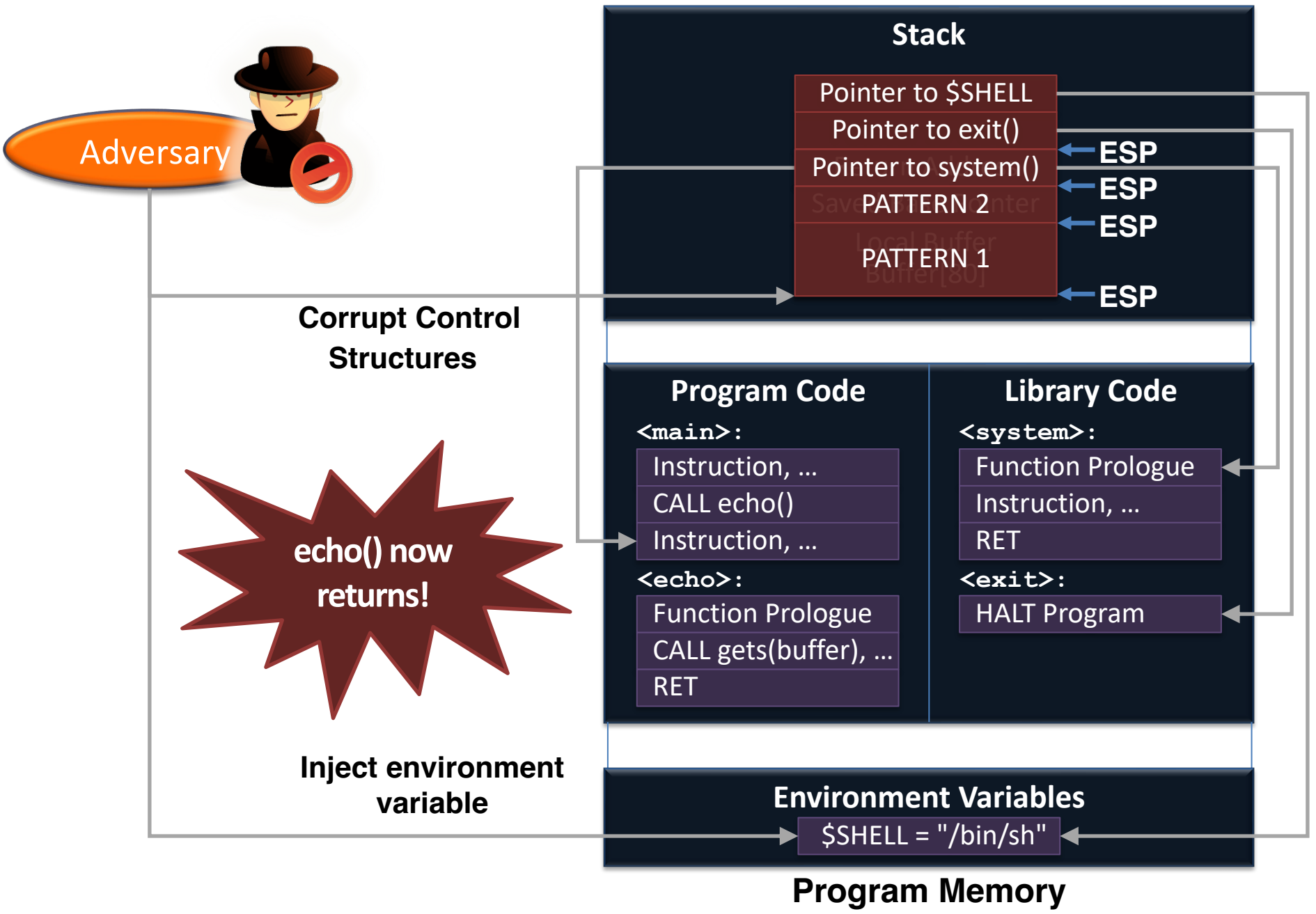
- Basic idea of return-to-libc
  - Redirect execution to functions in shared libraries
  - Main target is UNIX C library libc
    - Libc is linked to nearly every Unix program
    - Defines system calls and other basic facilities such as open(), malloc(), printf(), system(), execve(), etc.
  - Attack example: system (“/bin/sh”), exit()

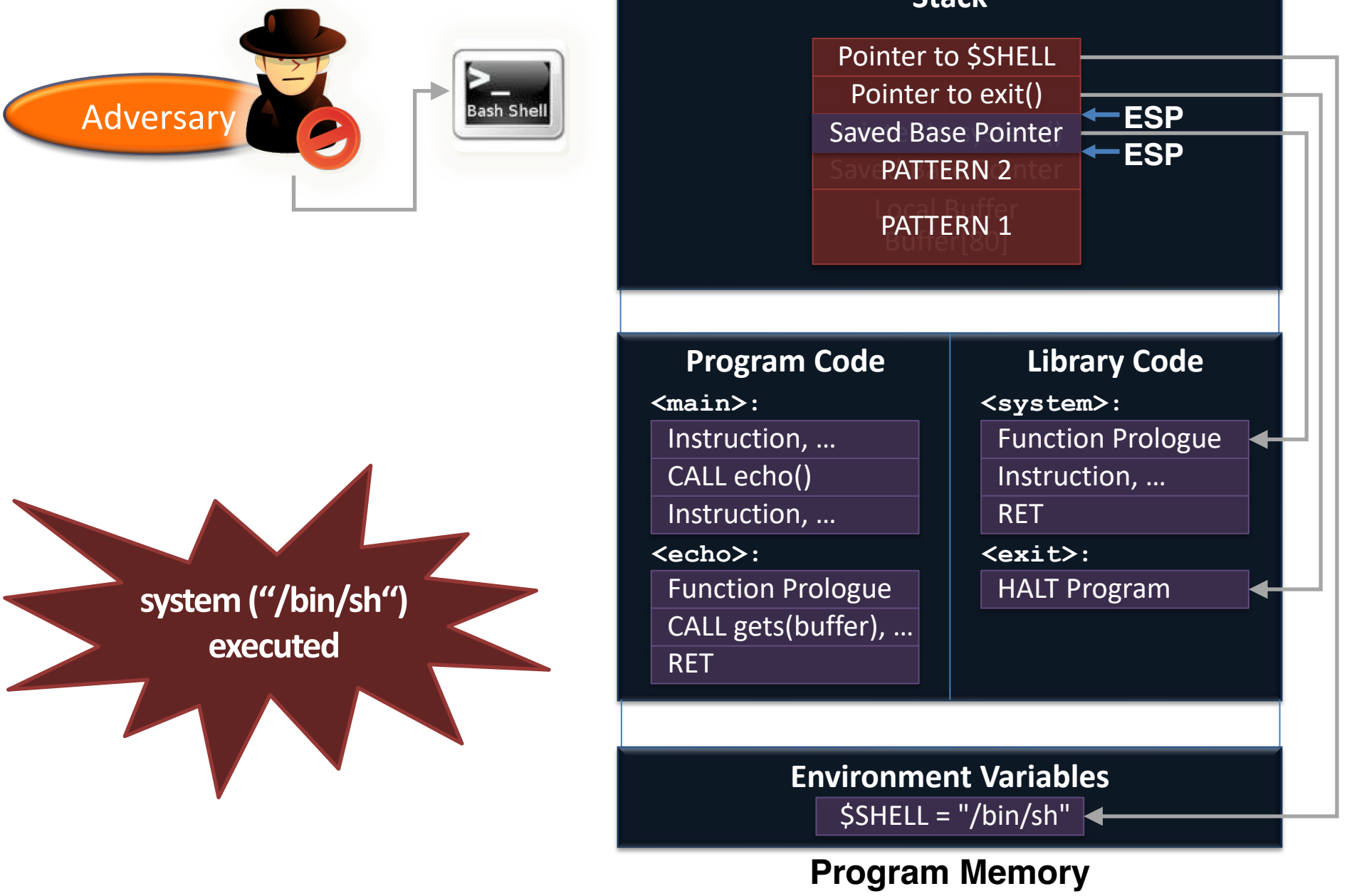
Execute a shell => Use permissions and privileges of the exploited program

Running Example

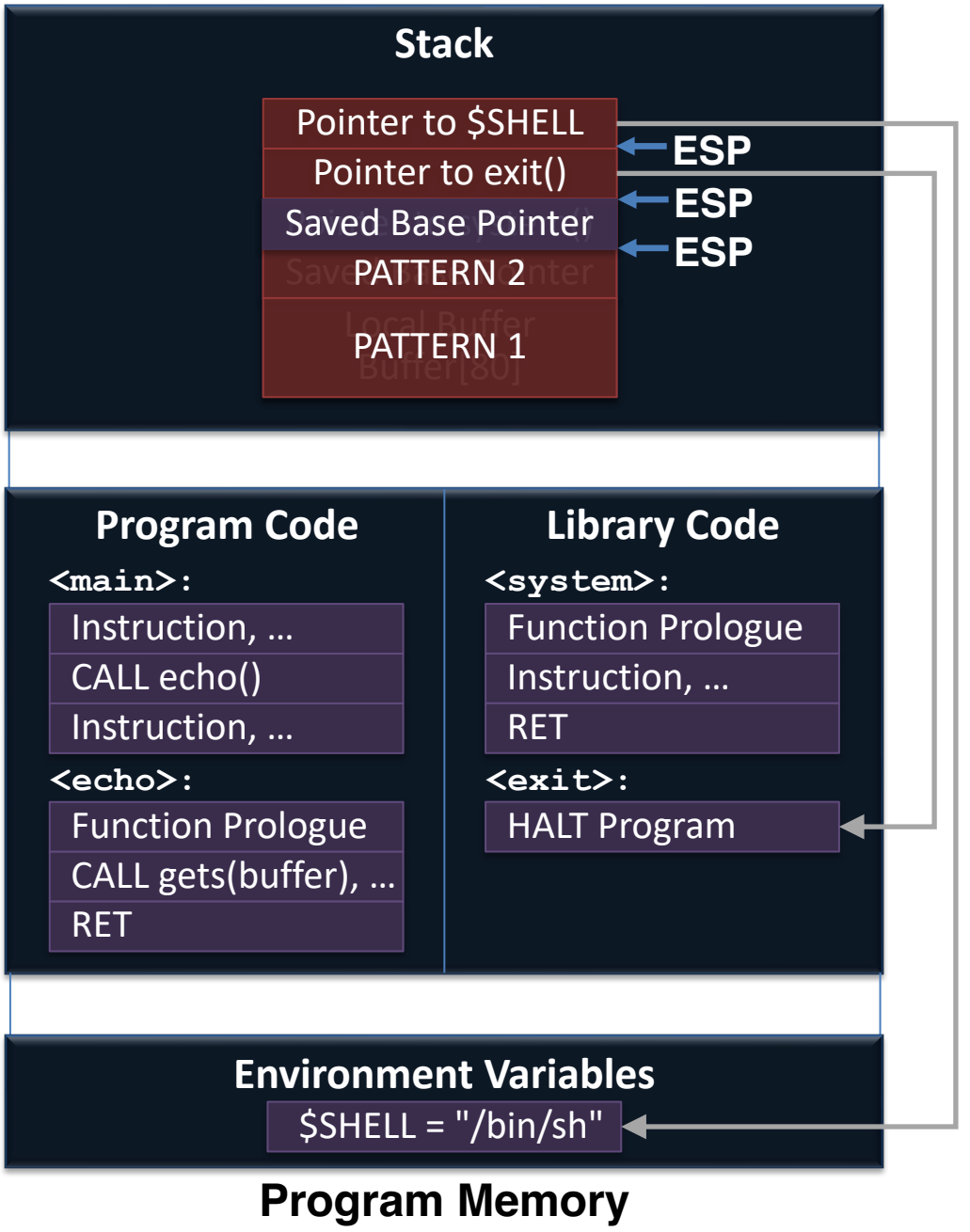
Vulnerable: Unlimited input into fixed-sized array (buffer)

```
#include <stdio.h>
void echo()
{
    char buffer[80];
    gets(buffer);
    puts(buffer);
}
int main ()
{
    echo();
    printf("Done");
    return 0;
}
```


Adversary
Stack
Pointer to $SHELL
Pointer to exit()
Pointer to system()
PATTERN 2
PATTERN 1
ESP
ESP
ESP
ESP
Corrupt Control Structures
Program Code
<main>:
Instruction, …
CALL echo()
Instruction, …
<echo>:
Function Prologue
CALL gets(buffer), …
RET
Library Code
<system>:
Function Prologue
Instruction, …
RET
<exit>:
HALT Program
echo() now returns!
Inject environment variable
Environment Variables
$SHELL = "/bin/sh"
Program Memory


Adversary
Bash Shell
Stack
Pointer to $SHELL
Pointer to exit()
Saved Base Pointer
PATTERN 2
PATTERN 1
ESP
ESP
Program Code
<main>:
Instruction, …
CALL echo()
Instruction, …
<echo>:
Function Prologue
CALL gets(buffer), …
RET
Library Code
<system>:
Function Prologue
Instruction, …
RET
<exit>:
HALT Program
Environment Variables
$SHELL = "/bin/sh"
Program Memory
system ("/bin/sh") executed

system ("/bin/sh") returning



Limitations of code-reuse

- No branching, i.e., no arbitrary code execution

- Critical functions can be eliminated or wrapped

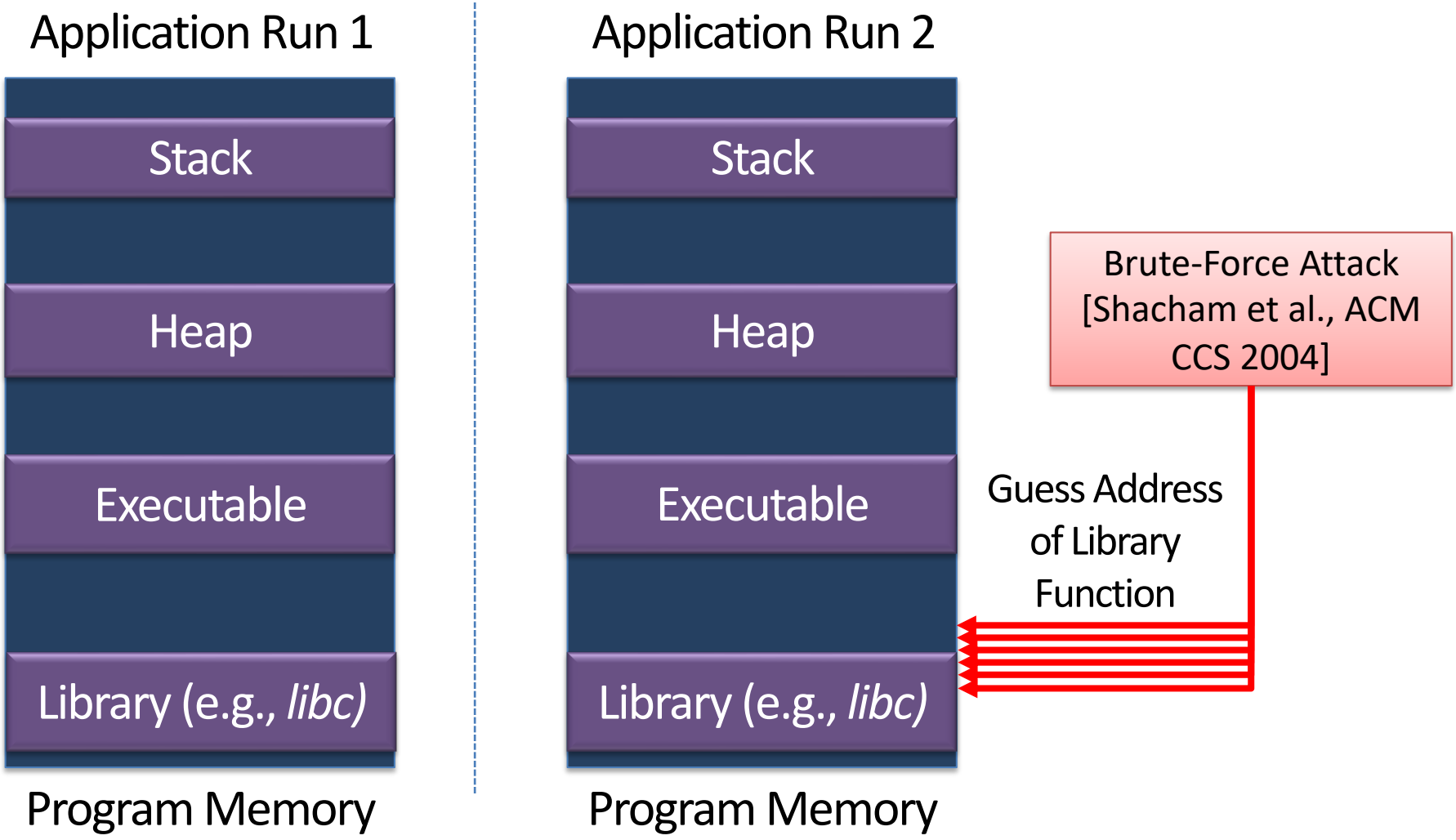
$\Rightarrow$ Then not usable for attack any longer

How can we generally defend against return-to-libc?

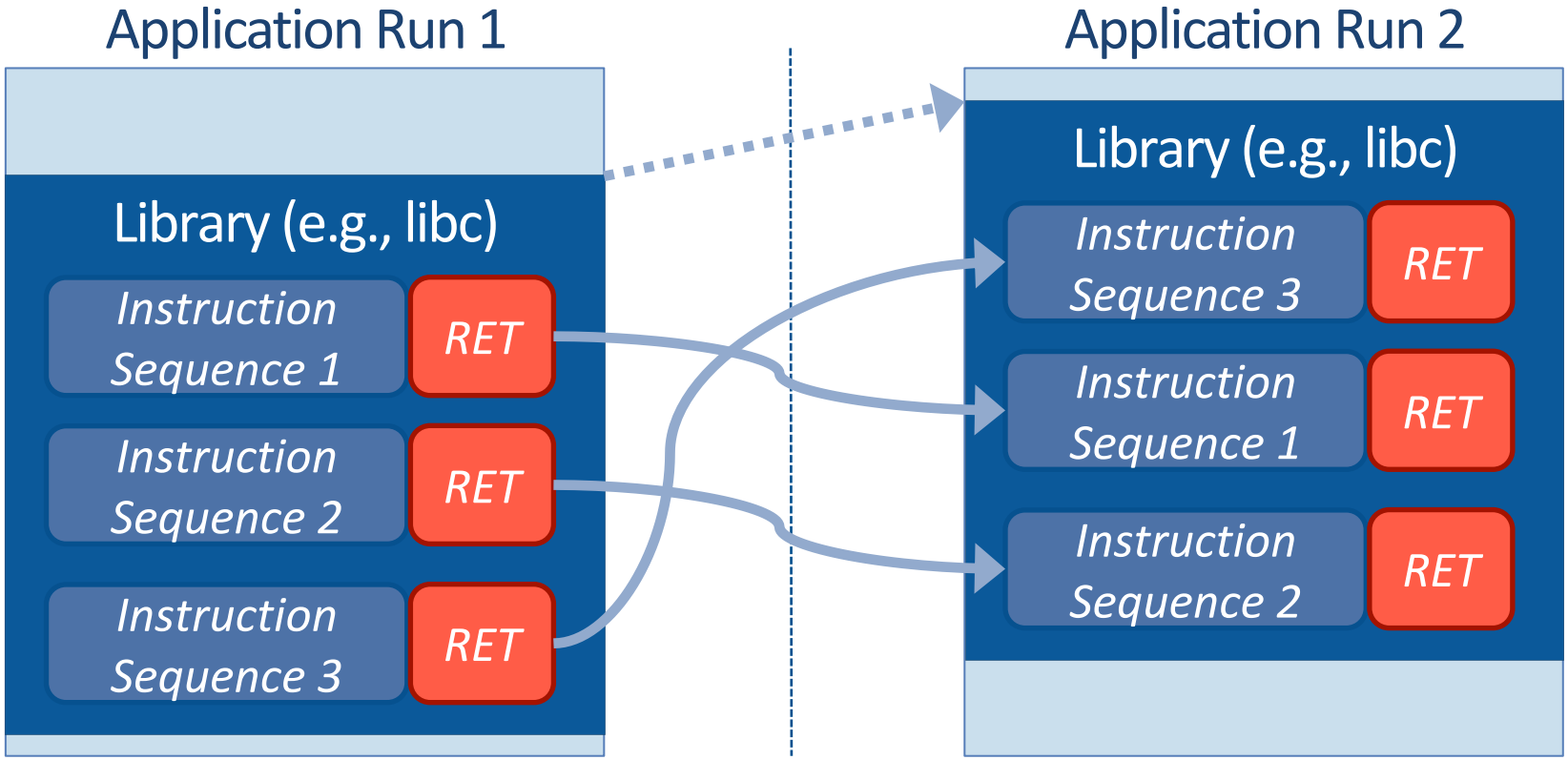
ASLR – Address Space Layout Randomization

Basics of Code Randomization

- ASLR randomizes the base address of code/data segments



Fine-Grained ASLR



- **ORP** [Pappas et al., IEEE S&P 2012]: Instruction reordering/substitution within a BBL
- **ILR** [Hiser et al., IEEE S&P 2012]: Randomizing each instruction‘s location
- **STIR** [Wartell et al., ACM CCS 2012] & **XIFER** [with Davi et al., AsiaCCS 2013]: Permutation of BBLs

How can ASLR be defeated?

- Cannot use library addresses

  (unless we find the library addresses first)
- Want to execute any code

$\Rightarrow$ Reuse small parts (gadgets) of program or libraries

- How do we know addresses?
  - Address leak
  - Program code location not randomized (efficiency reasons)

$\Rightarrow$ Small Parts must end with return

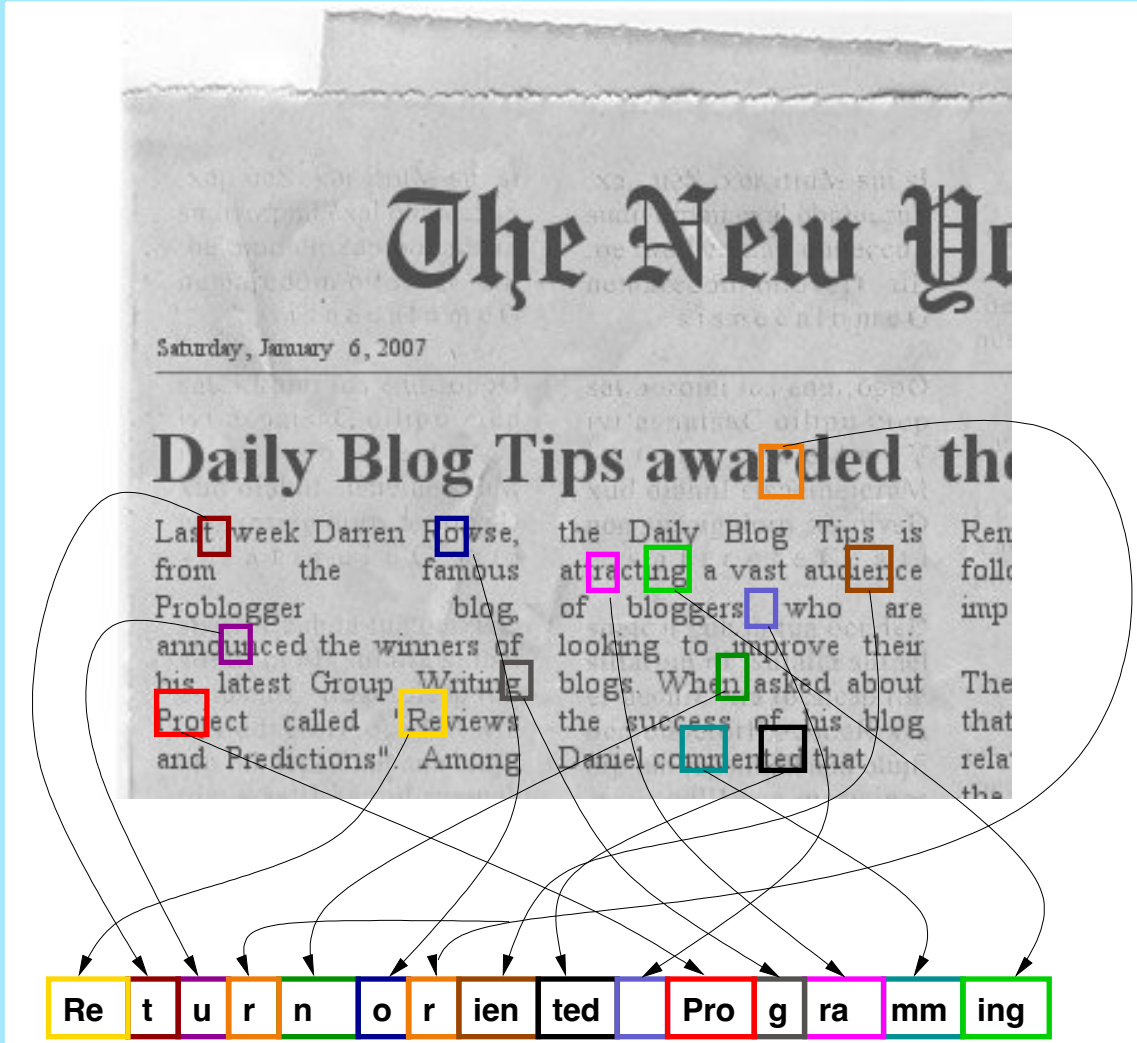
- Allows attacker to specify next part

  $\Rightarrow$ Build Gadgets

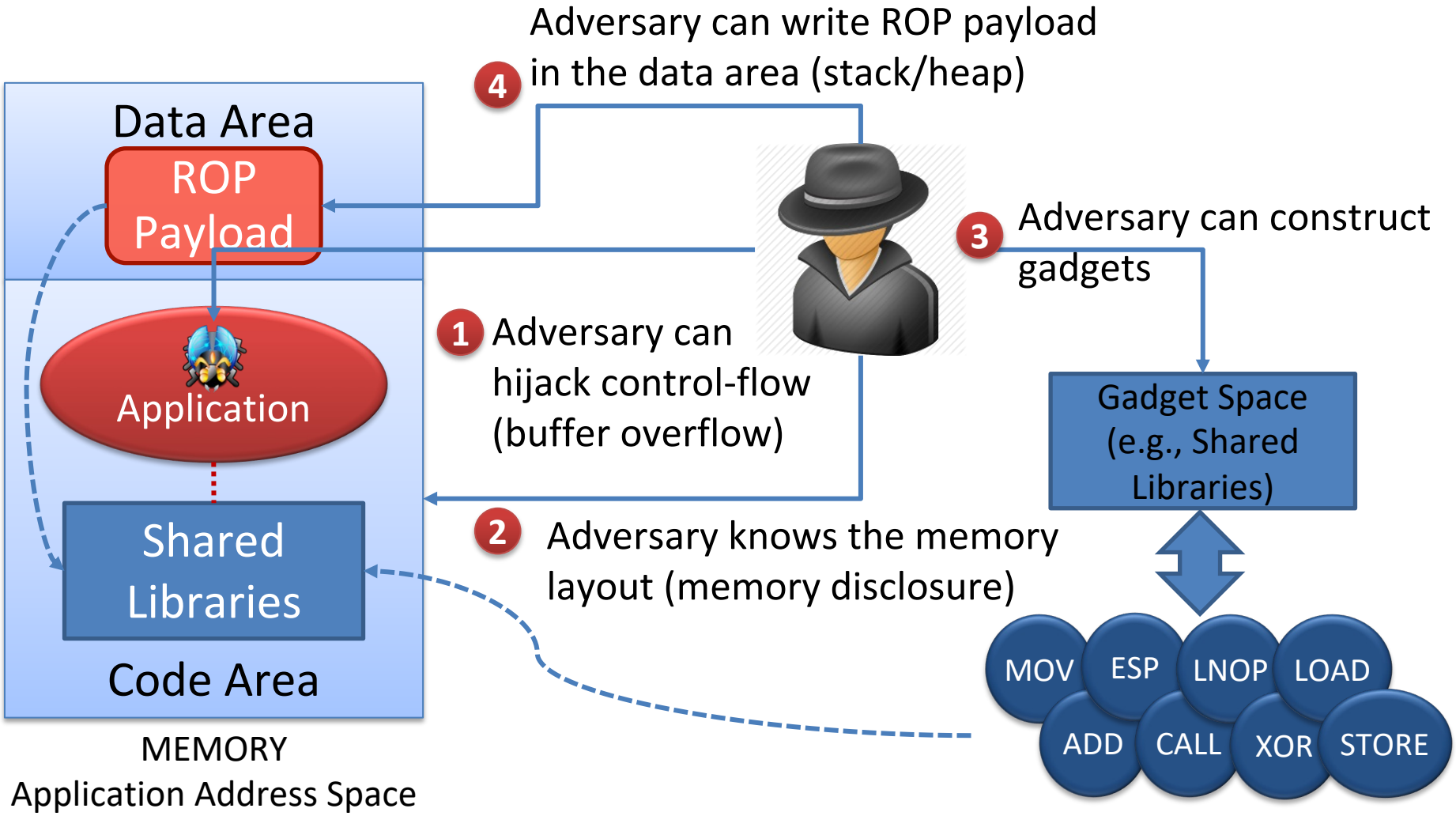
$\Rightarrow$ <u>Return-Oriented Programming</u>

(simplified version in the exercise)

ROP: The Big Picture



ROP Adversary Model/Assumption



ROP Attack Technique: Overview

Corrupt Control Structures

Program Stack

- Return Address 3
- Value 2
- Value 1
- Return Address 2
- Return Address 1

SP

Program Code

- Sequence 1
  - asm_ins
  - RET
- Sequence 2
  - POP REG1
  - POP REG2
  - RET
- Sequence 3
  - asm_ins
  - RET

...

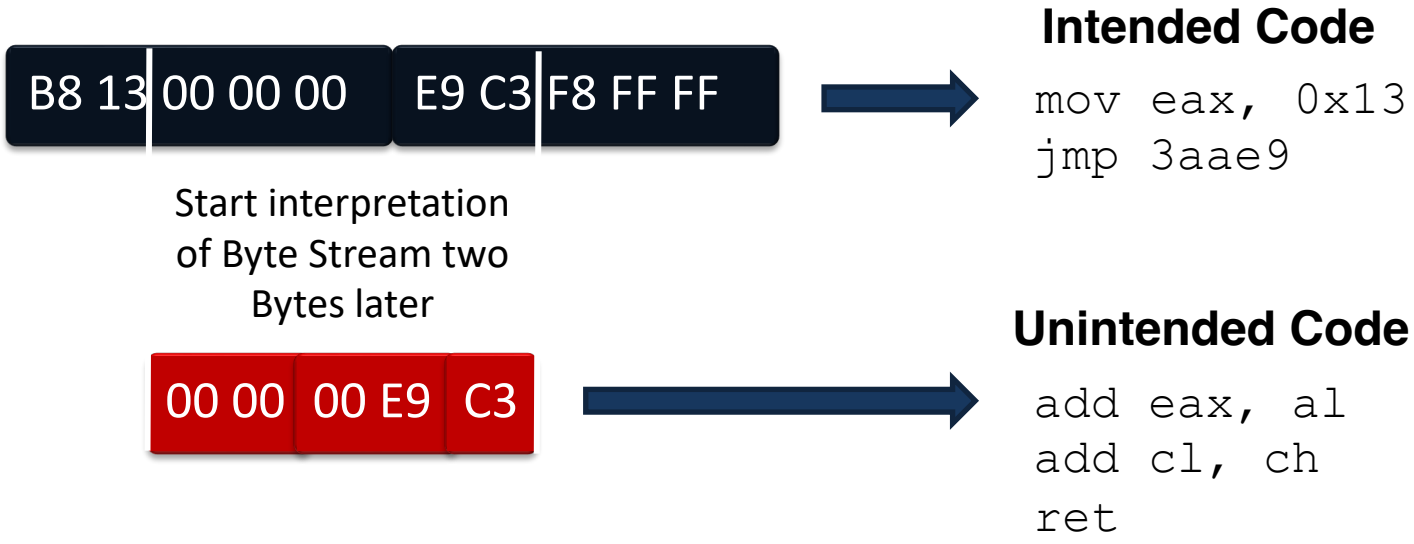
REG1:	Value 1
REG2:	Value 2

Summary of Basic Idea

- Perform arbitrary computation with gadgets
- Approach
  - Use small instruction sequences (e.g., of libc) instead of using whole functions
  - Instruction sequences range from 2 to 5 instructions
  - All sequences end with a return instruction
  - Instruction sequences are chained together to a gadget
  - A gadget performs a particular task (e.g., load, store, xor, or branch)
  - Afterwards, the adversary enforces his desired actions by combining the gadgets

Unintended Instruction Sequences on x86

- Instruction sequences never intended by the programmer
  - Get new code out of existing code
- Possible due to
  - Unaligned Memory Access
  - Variable-Length Instructions



Turing-Completeness

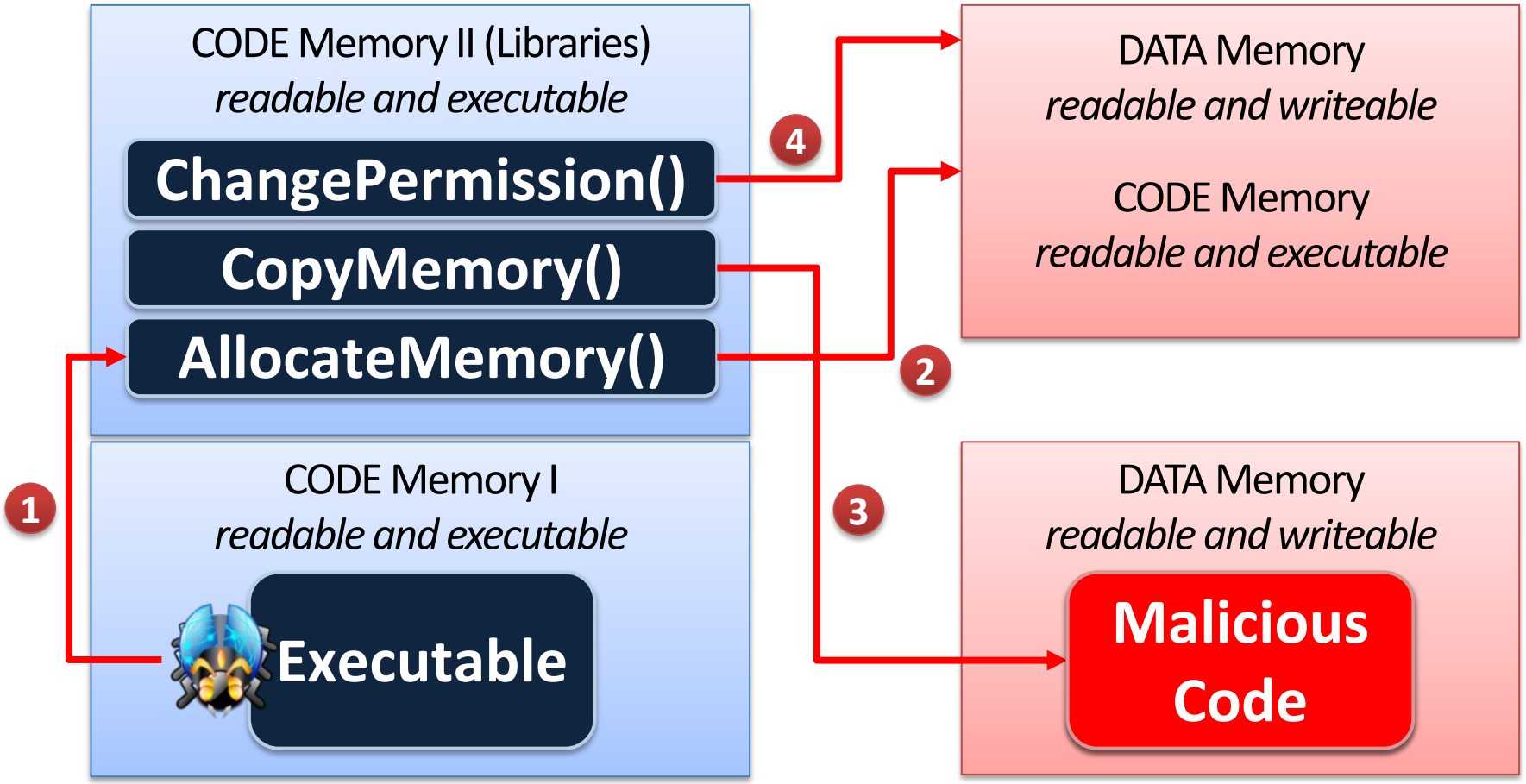
- Turing-Completeness allows arbitrary computation
- Minimum gadget set required (theoretically) [Shacham CCS 2007]
  - Memory load and store
  - (Un)Conditional branch - *the most challenging gadget*
- Practically additional gadgets for ROP payload needed:
  - Data processing (for moving values between registers)
  - Arithmetic and logical operations
  - Function and system call

Code Injection vs. Code Reuse

- Code Injection – *Adding a new node to the CFG*
  - Adversary can execute arbitrary malicious code
    - open a remote console (classical shellcode)
    - exploit further vulnerabilities in the OS kernel to install a virus or a backdoor
- Code Reuse – *Adding a new path to the CFG*
  - Adversary is limited to the code nodes that are available in the CFG
  - Requires reverse-engineering and static analysis of the code base of a program

Hybrid Exploits

- Today‘s attacks combine code reuse with code injection



Stack Canaries

Idea:

- Protect important stack values

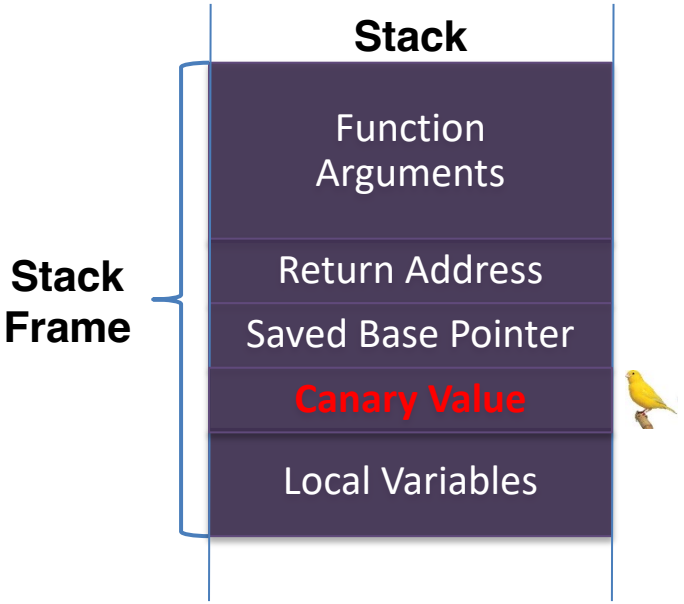
During function prologue:

- Write a random value onto stack
- Copy stored in memory “far away”

During function epilogue:

- Check random value against copy
- Stop execution on mismatch

- Compiler puts these instructions in
- Extra work => Runtime Overhead



Not only ROP (1/2)

- The control flow of a program can be diverted:
  - By overwriting the **return address**
  - By overwriting a **variable** used for a control flow decision
  - By overwriting a **function pointer** to point to a different function

- Example: Modify a variable

```
int buffer[10];
int authorized = 0;

strcpy(buffer, userinput);
if(authorized)
    printf("yes");
else
    printf("no");
```

User overflow to overwrite authorized to 1

Not only ROP (2/2)

- Example: Modify a function pointer
  - What is a function pointer?

```
void func1(void){ doThis(); }
void func2(void){ doThat(); }
void (*function_ptr)(void);
function_ptr = func1;
function_ptr() // calls func1
```

  - Example:

```
int buffer[10];
void (*func_ptr)(void)
func_ptr = func1;
strcpy(buffer, userinput);
```

User overflow to overwrite func_ptr to point to func2

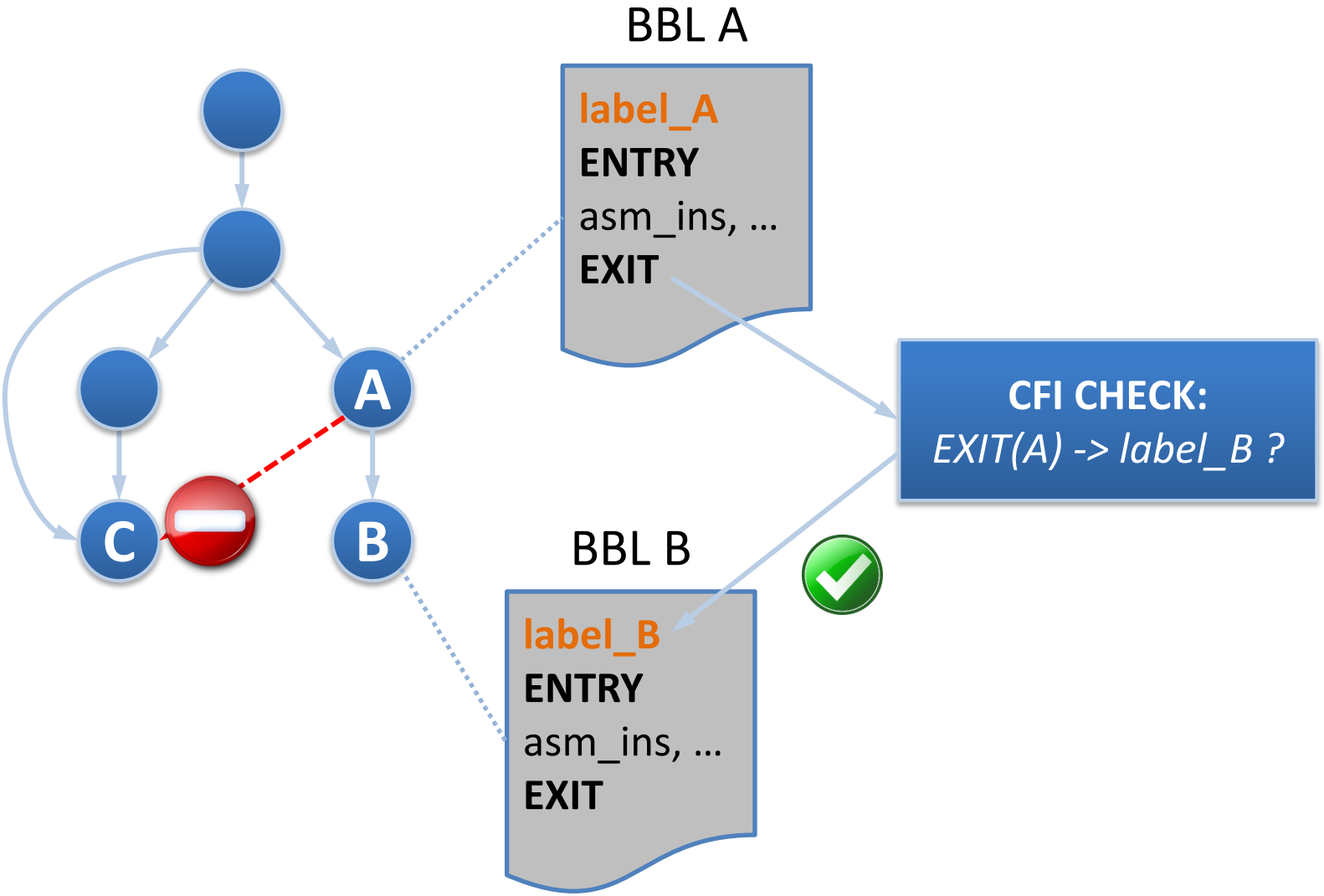
Generic ROP Defense: Control-Flow Integrity (CFI)

Restricting indirect control-flow targets to a pre-defined control-flow graph

$\Rightarrow$ Usually requires compiler support
$\Rightarrow$ Know which control flows are intended
$\Rightarrow$ Checks requires extra work during runtime
$\Rightarrow$ Performance issues (partially solved recently)

Original CFI Label Checking

[Abadi et al., CCS 2005 & TISSEC 2009]



Shadow Stack

- Has been proposed for a while
- Idea: CPU protects return addresses in special, separate memory
- Checks them before using them
- Included in Tiger Lake Intel Processors: https://software.intel.com/content/www/us/en/develop/articles/technical-look-control-flow-enforcement-technology.html