

# **Lecture 1: Introduction and Overview**

**Sanchuan Chen**

[schen@auburn.edu](mailto:schen@auburn.edu)

8/16/2023



## Software Becomes Increasingly Sophisticated



# Software Becomes Increasingly Sophisticated



# Software Becomes Increasingly Sophisticated



# System Software Developed w/ Unsafe Languages

## C was designed in 1970s

- ① Less networked, less attacks
- ② Favor performance and flexibility
- ③ Lack of security features
  - ▶ No automatic memory management
  - ▶ No strong typing
  - ▶ No bounds checks
  - ▶ No overflow checks

# System Software Developed w/ Unsafe Languages

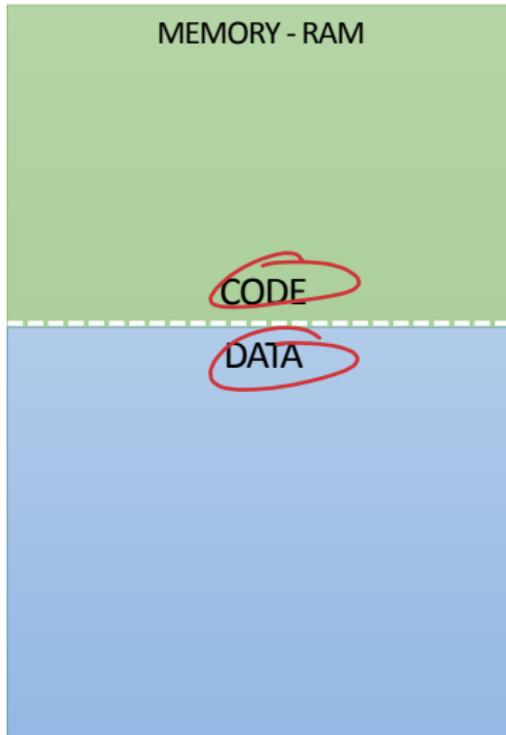
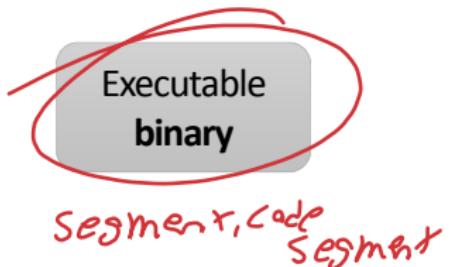
## C was designed in 1970s

- ① Less networked, less attacks
- ② Favor performance and flexibility
- ③ Lack of security features
  - ▶ No automatic memory management
  - ▶ No strong typing
  - ▶ No bounds checks
  - ▶ No overflow checks

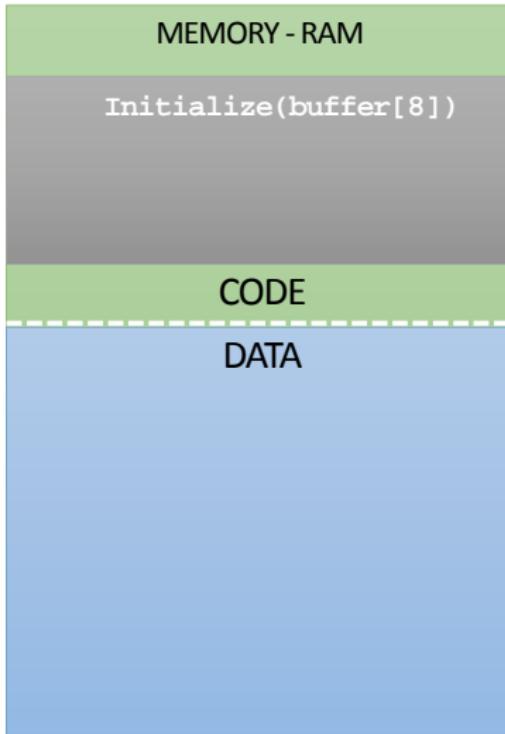
## Why still being used?

- ① Performance *harmon time faster*
  - ▶ C/C++/Object-C
  - ▶ Hand-written assembly
  - ▶ Optimal use of the hardware
- ② Backwards-compatibility *still used*
  - ▶ Legacy systems
  - ▶ Most of them were developed in C

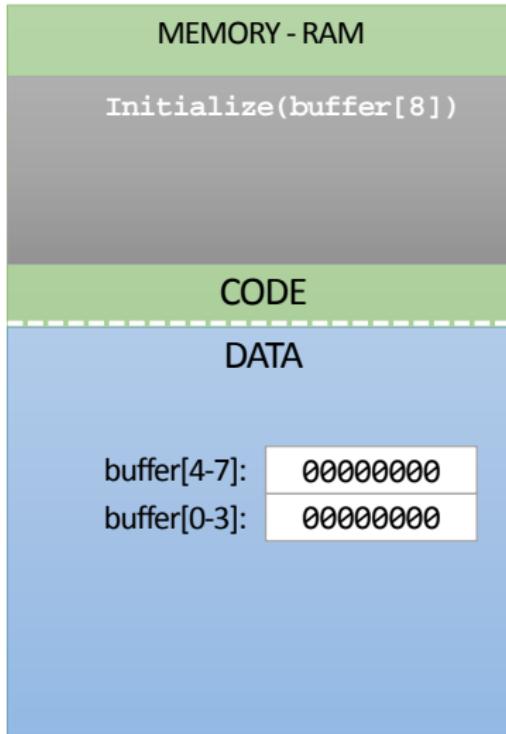
# The Memory Corruption Vulnerability *Mem corruption*



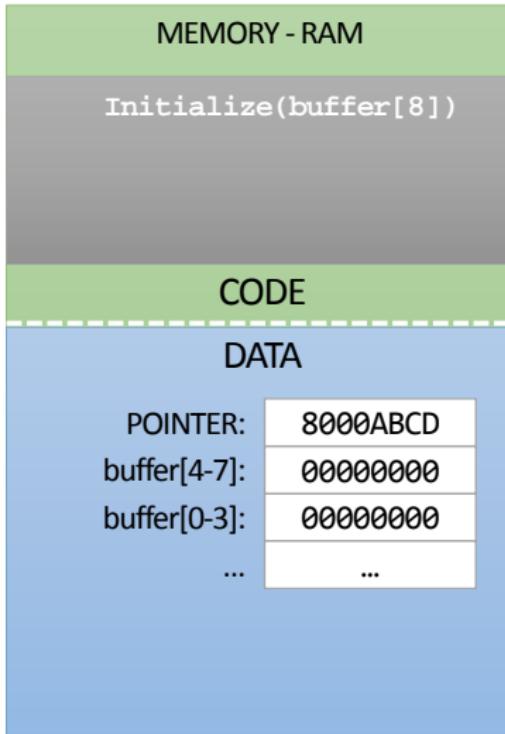
# The Memory Corruption Vulnerability *High level description*



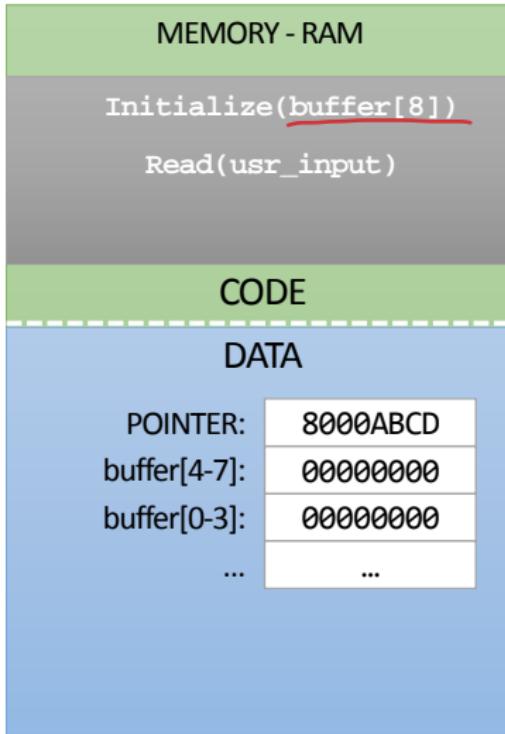
# The Memory Corruption Vulnerability



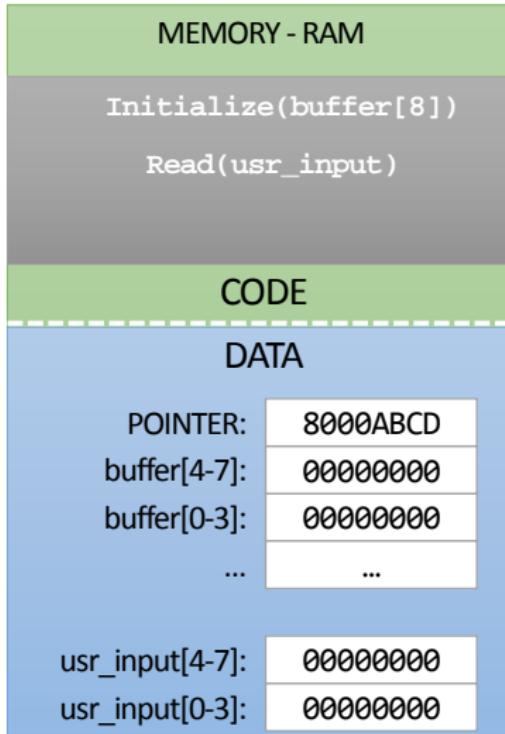
# The Memory Corruption Vulnerability



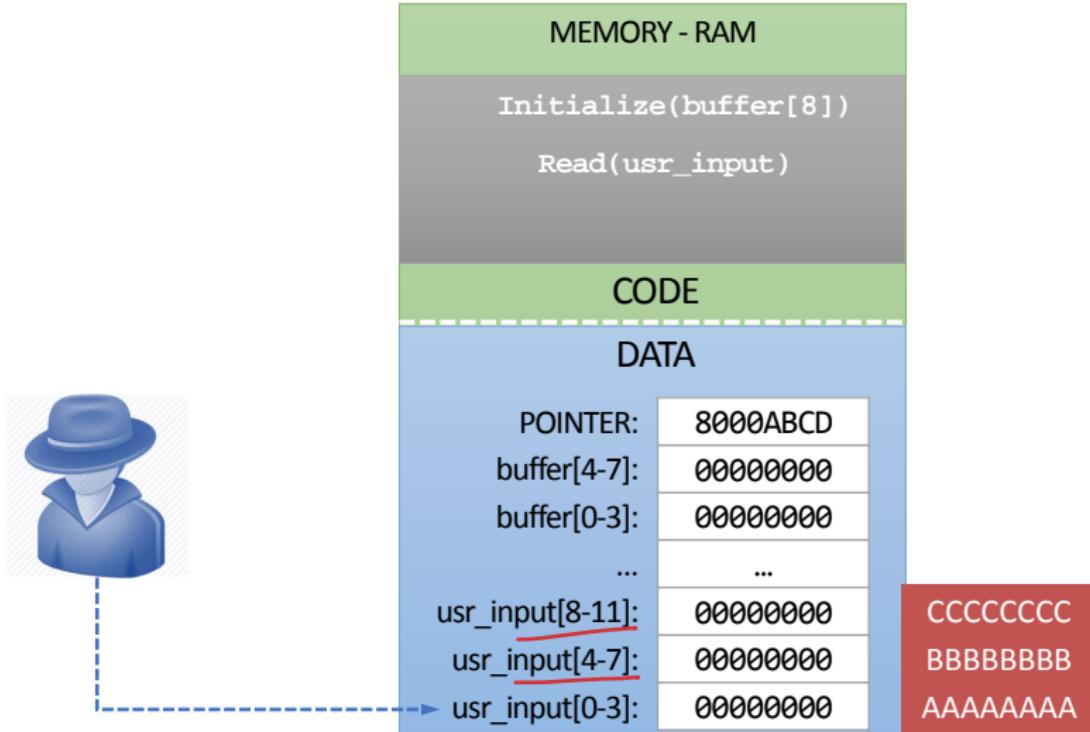
# The Memory Corruption Vulnerability



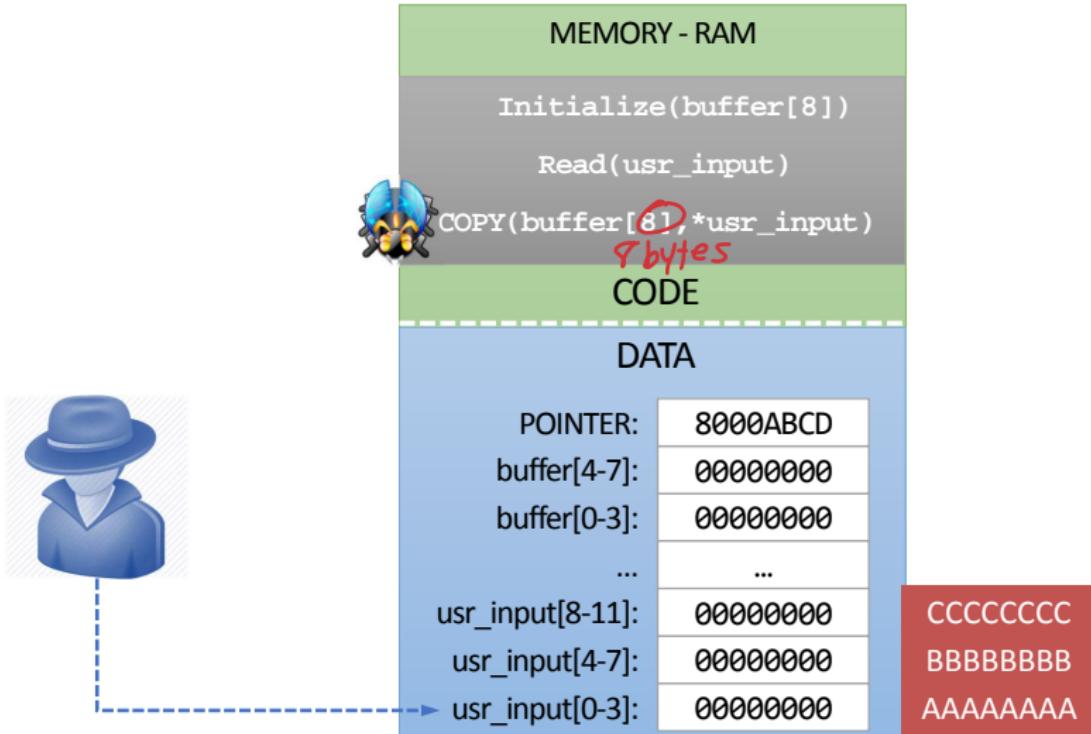
# The Memory Corruption Vulnerability



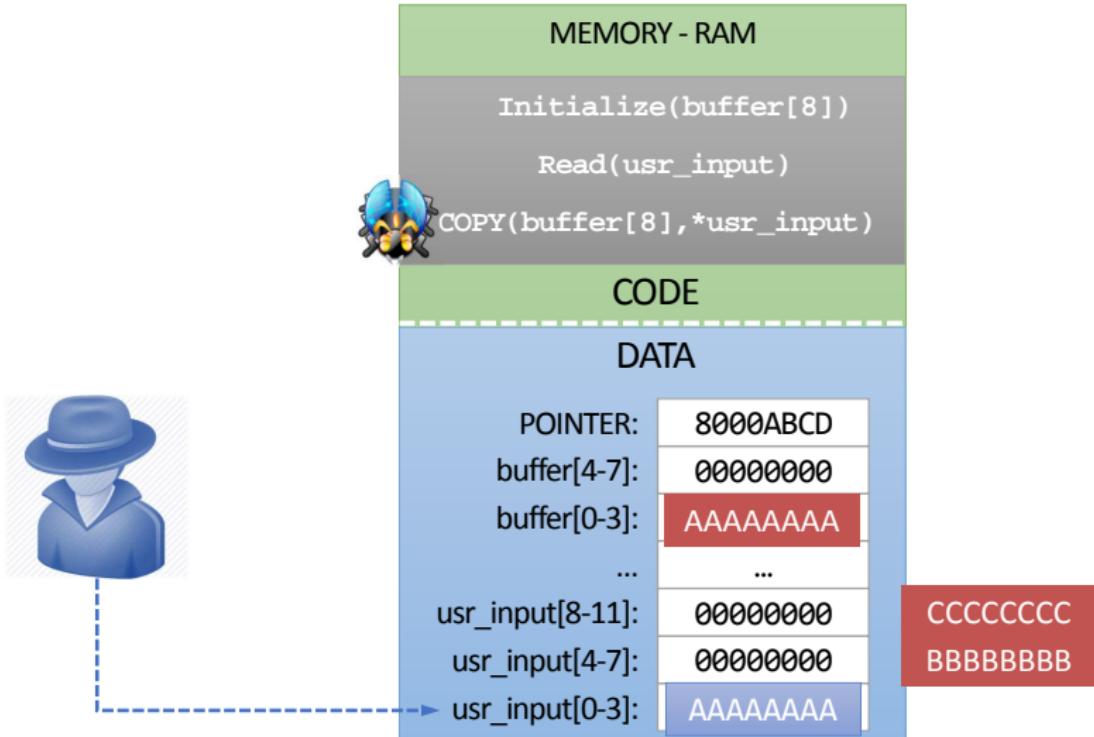
# The Memory Corruption Vulnerability



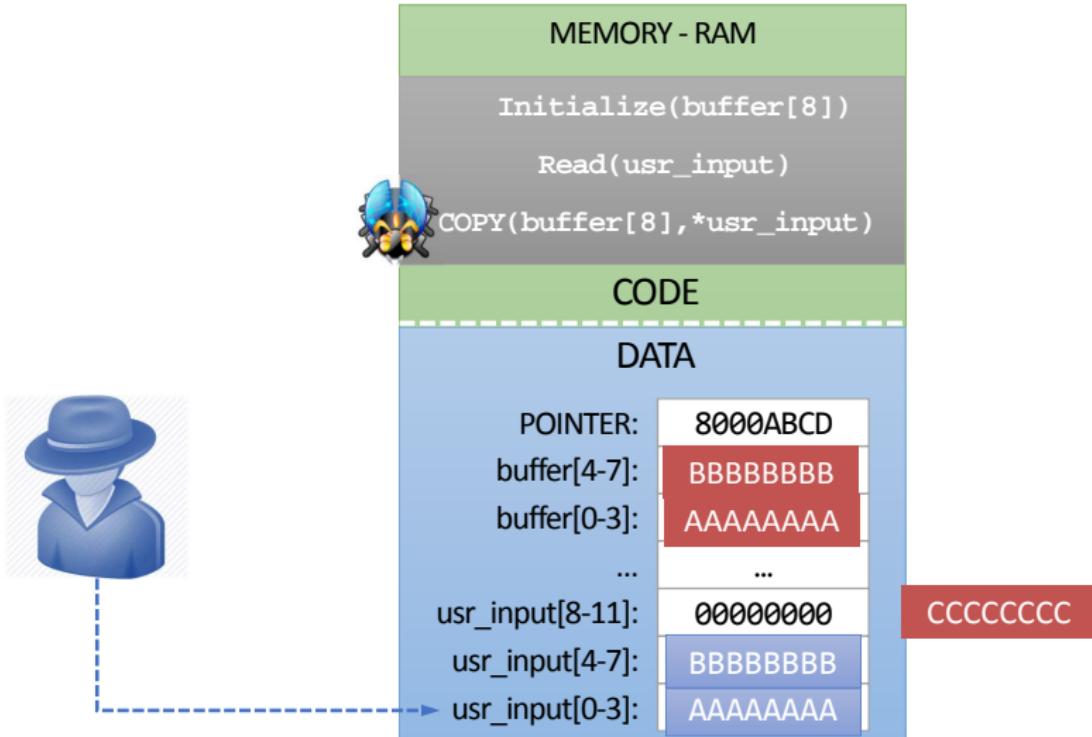
# The Memory Corruption Vulnerability



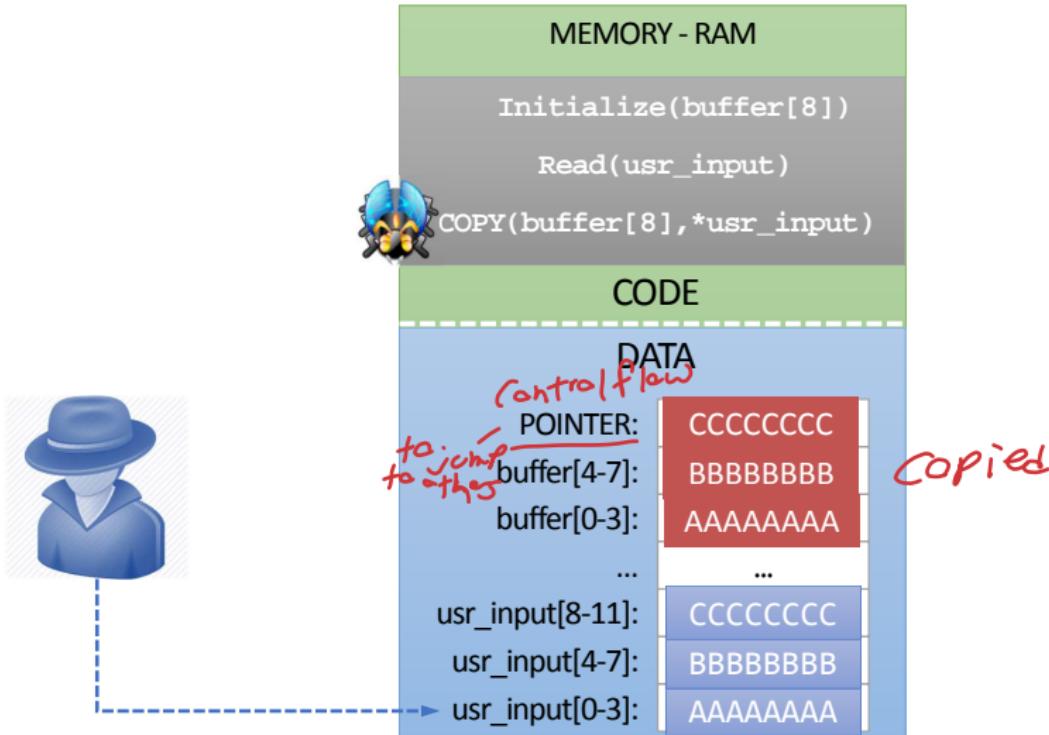
# The Memory Corruption Vulnerability



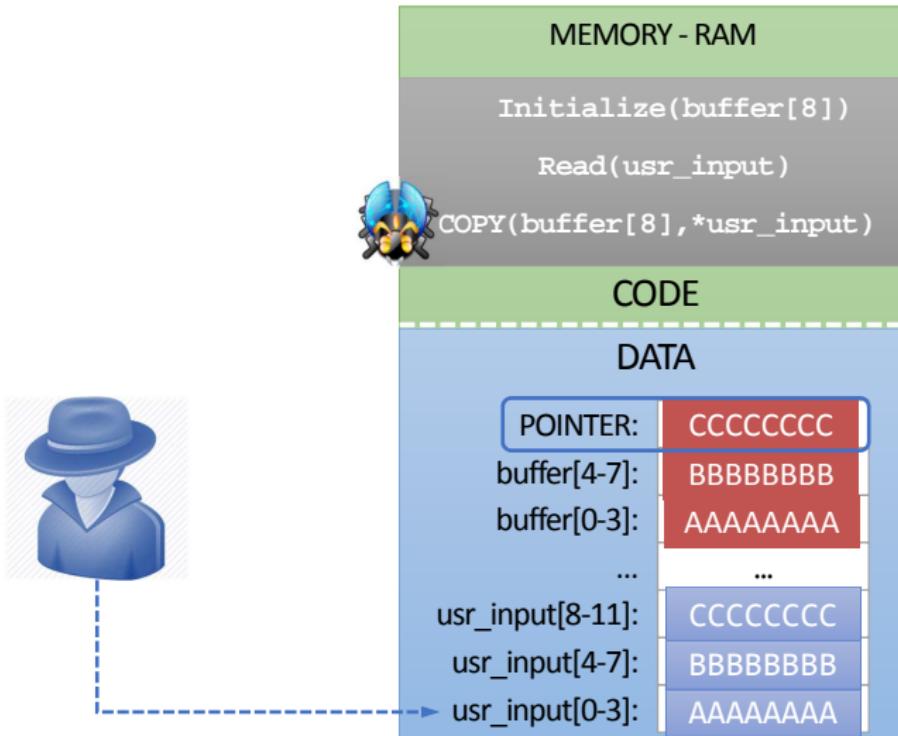
# The Memory Corruption Vulnerability



# The Memory Corruption Vulnerability



# The Memory Corruption Vulnerability



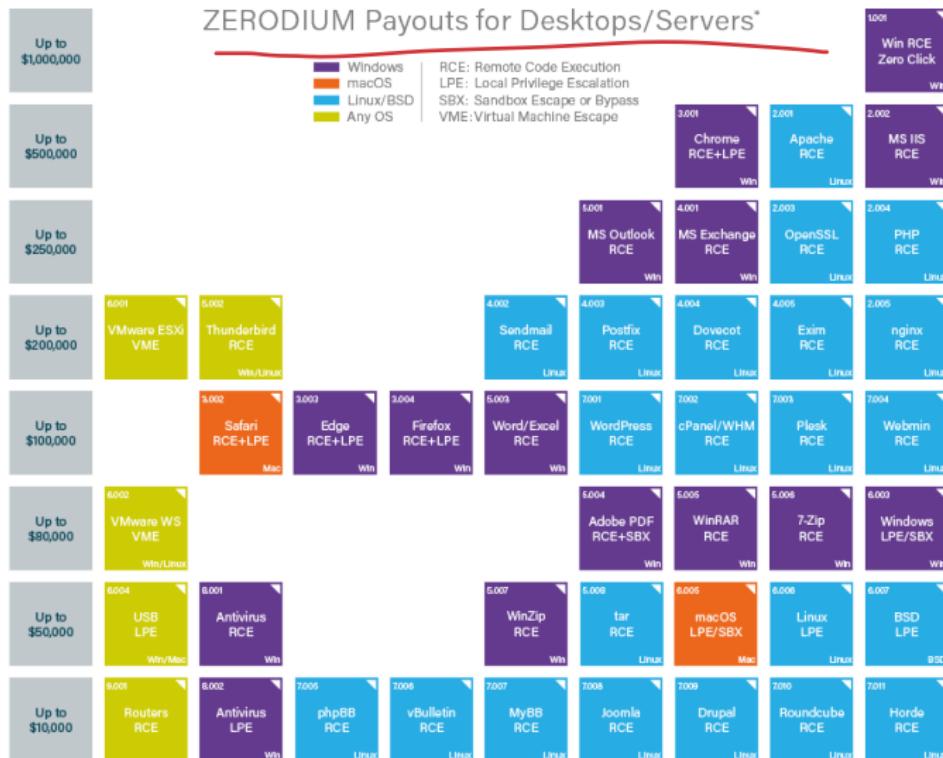
## Why It Matters

### Many High Impact Attacks

*attack browser*

- ① Web browsers repeatedly exploited in pwn2own contests
- ② Zero-day issues exploited in Stuxnet/Duqu
- ③ iOS jailbreak

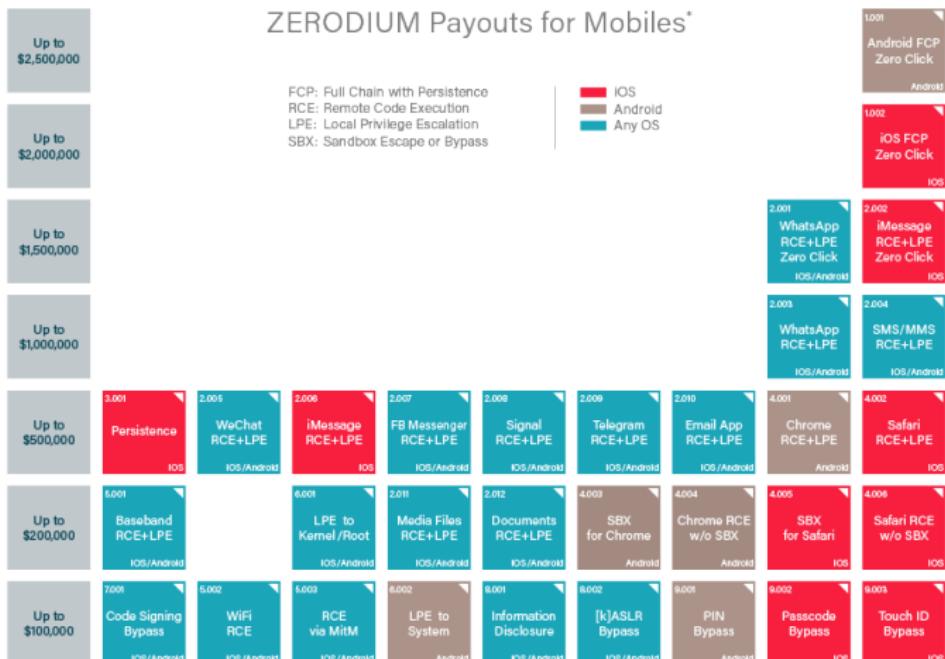
# Why It Matters



\* All payouts are subject to change or cancellation without notice. All trademarks are the property of their respective owners.

2019/01 © zerodium.com

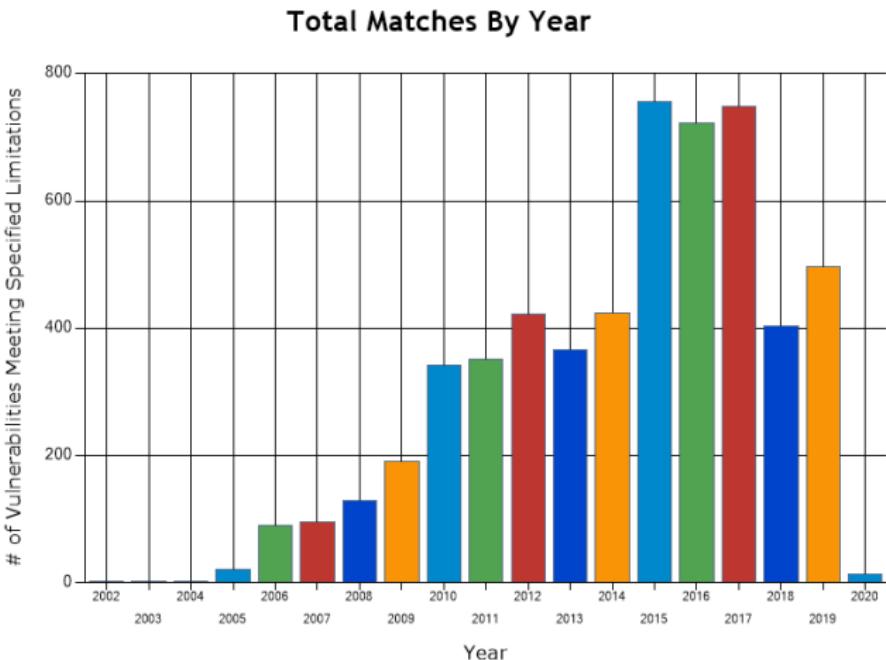
# Why It Matters



\* All payouts are subject to change or cancellation without notice. All trademarks are the property of their respective owners.

2019/09 © zerodium.com

# The Statistics of The Memory Corruption Vulnerabilities



<https://nvd.nist.gov/vuln/search/statistics>

Introduction  
oooooooooooo

Canary and DEP  
oooooo

Randomization  
oooooooo

Control Flow Integrity  
ooo

Discussion  
ooooo

Summary  
oo

# The Arm Race (Attack and Defense) w/ Memory Corruptions



# The Arm Race (Attack and Defense) w/ Memory Corruptions

Attacks

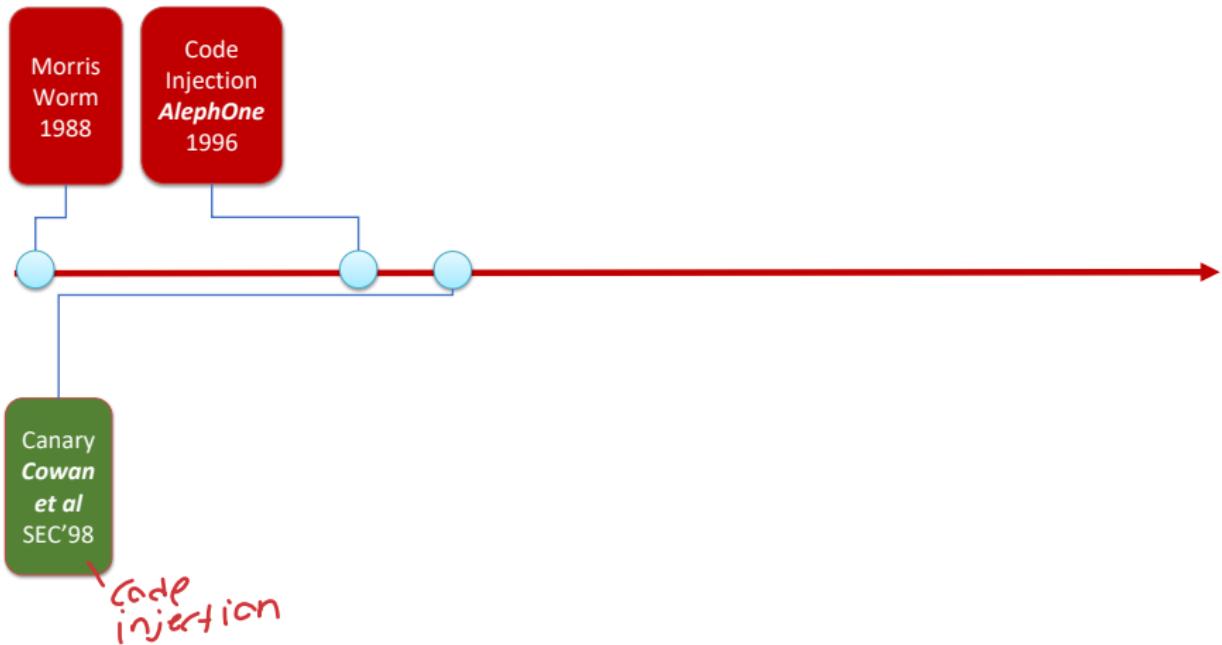


Defense

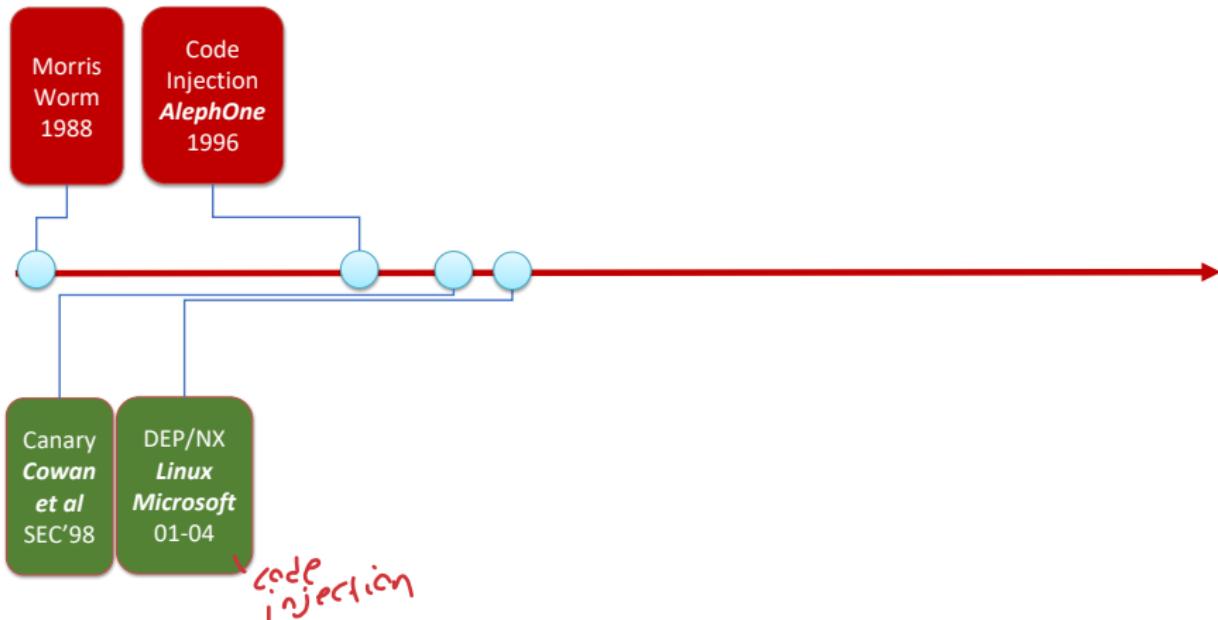
# The Arm Race (Attack and Defense) w/ Memory Corruptions



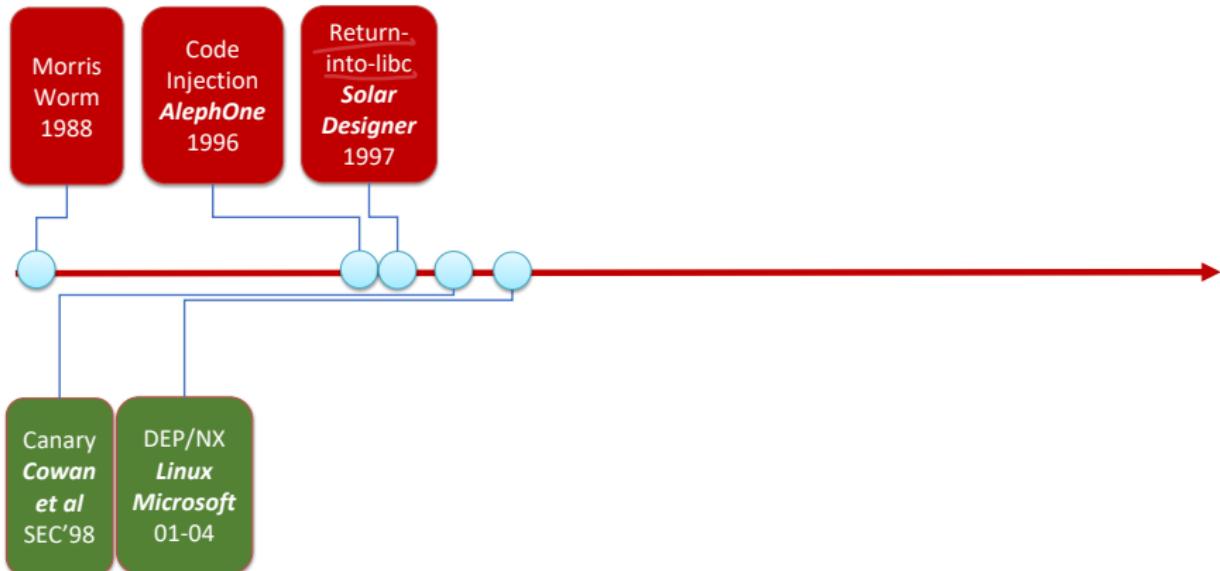
# The Arm Race (Attack and Defense) w/ Memory Corruptions



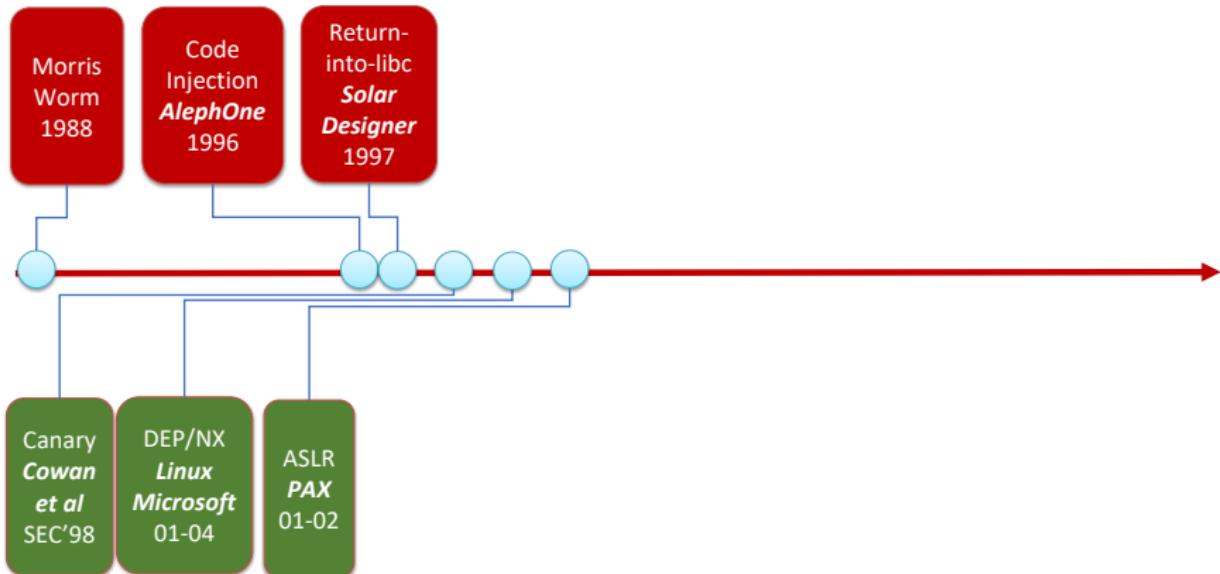
# The Arm Race (Attack and Defense) w/ Memory Corruptions



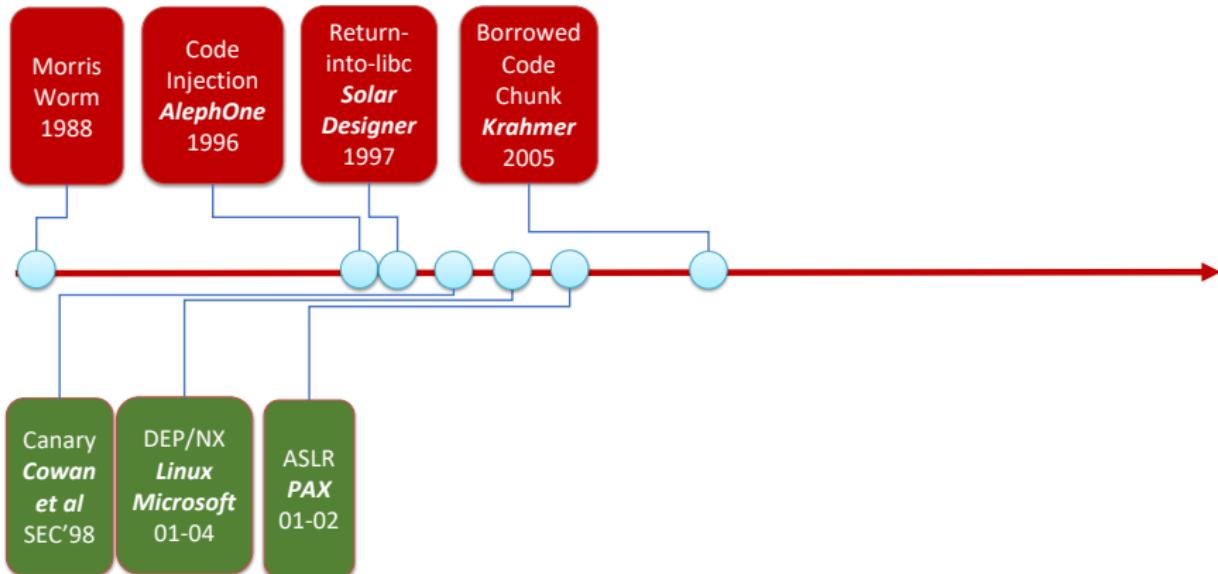
# The Arm Race (Attack and Defense) w/ Memory Corruptions



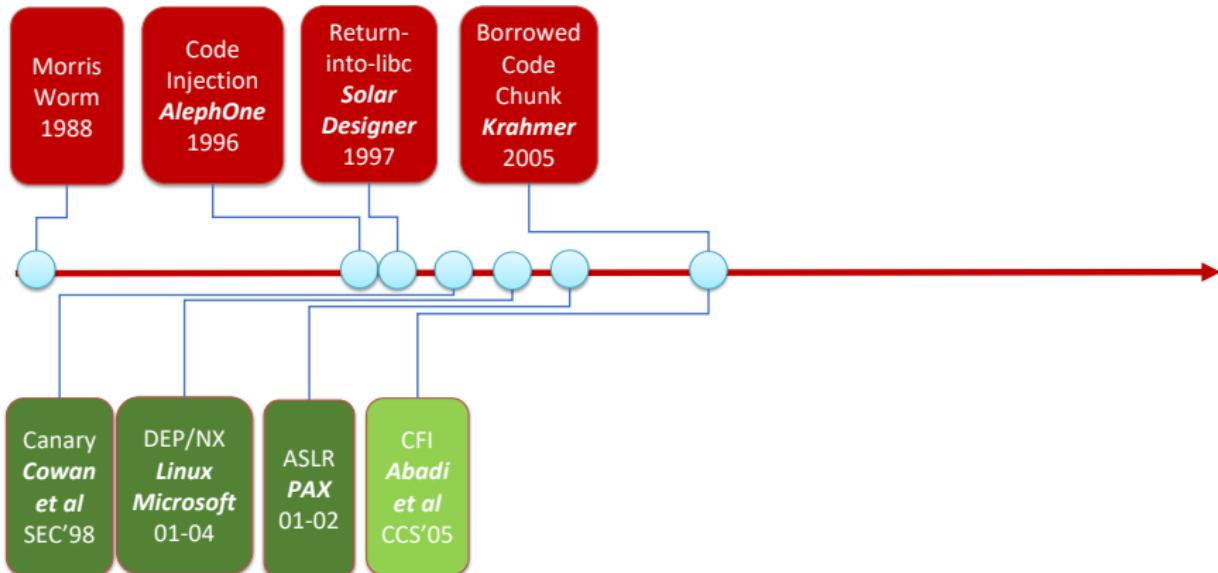
# The Arm Race (Attack and Defense) w/ Memory Corruptions



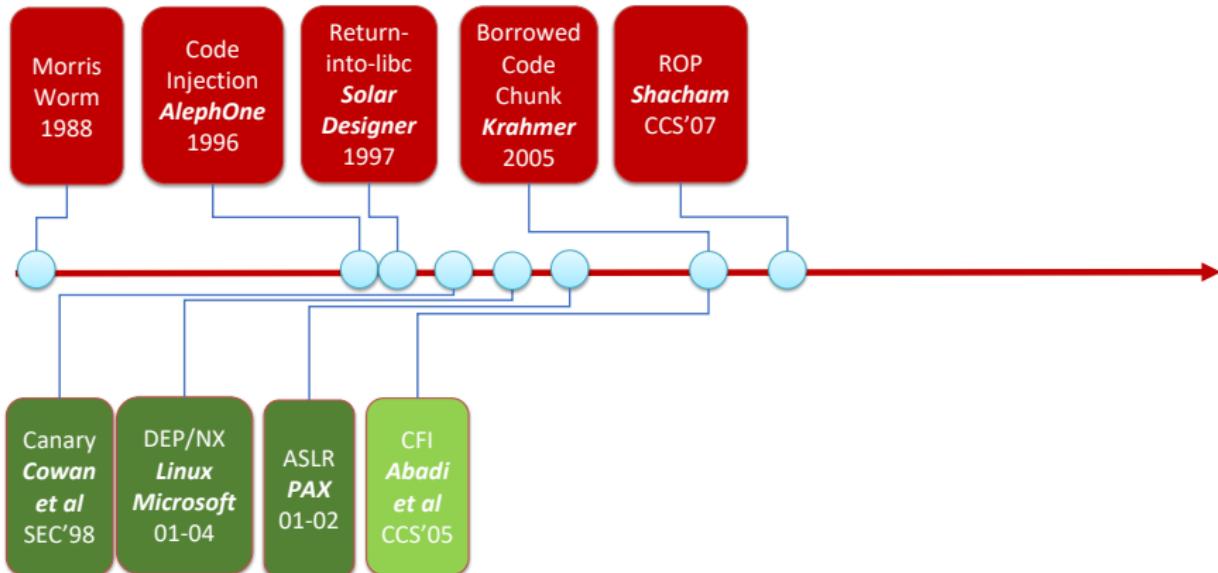
# The Arm Race (Attack and Defense) w/ Memory Corruptions



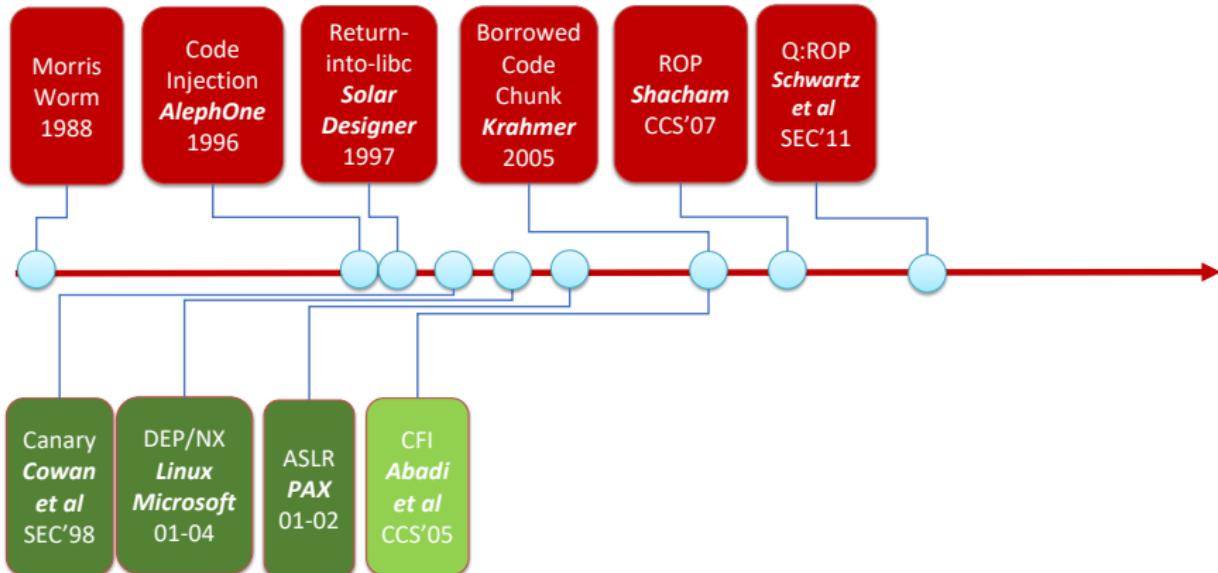
# The Arm Race (Attack and Defense) w/ Memory Corruptions



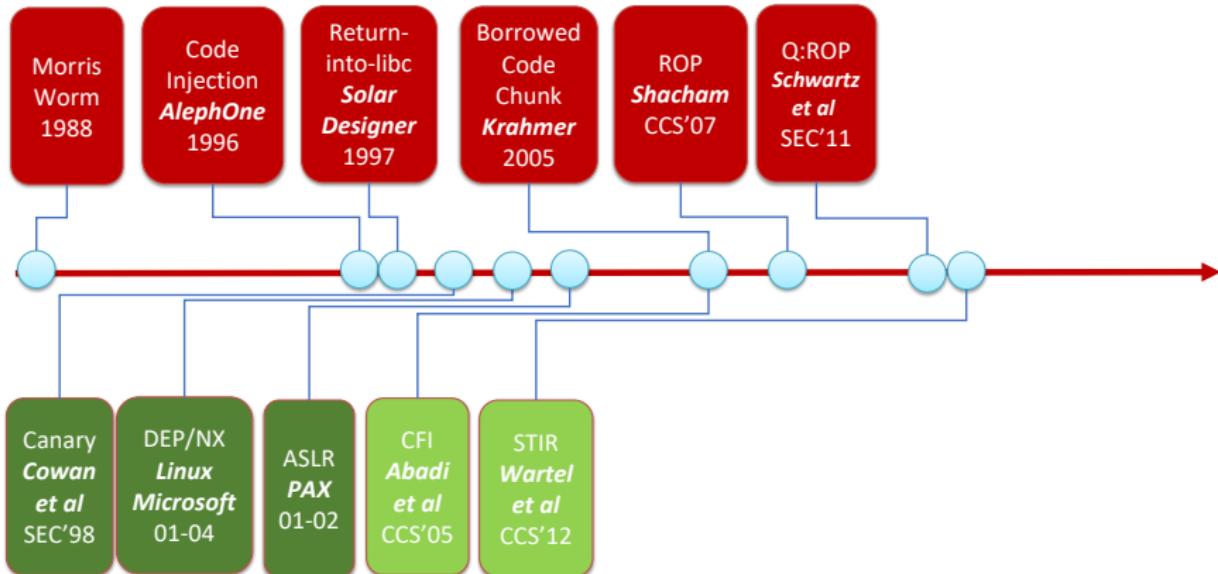
# The Arm Race (Attack and Defense) w/ Memory Corruptions



# The Arm Race (Attack and Defense) w/ Memory Corruptions

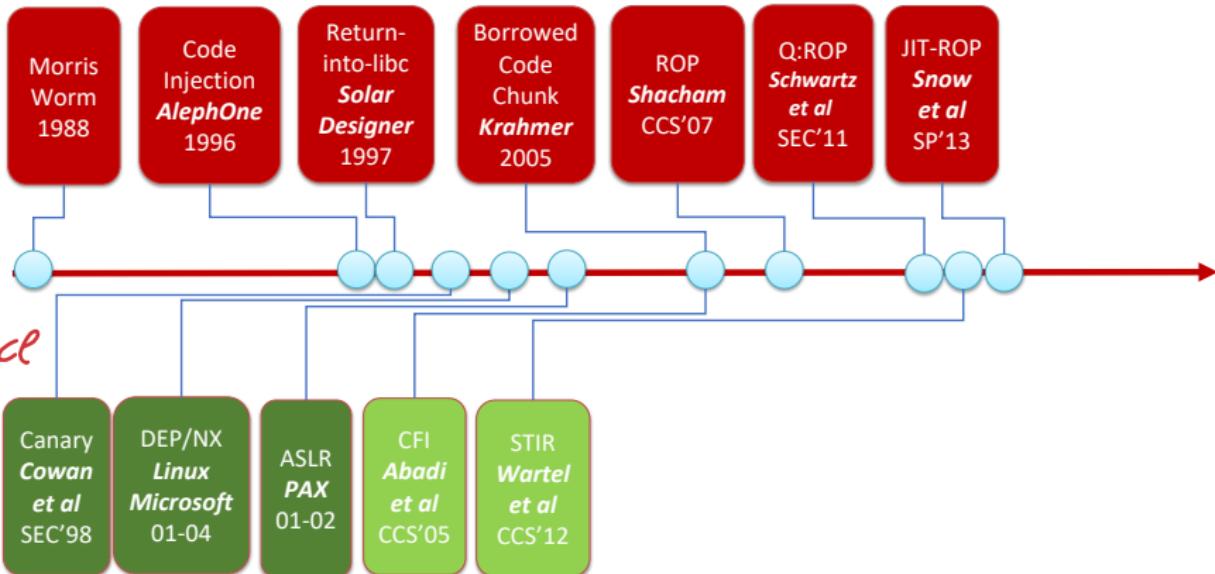


# The Arm Race (Attack and Defense) w/ Memory Corruptions



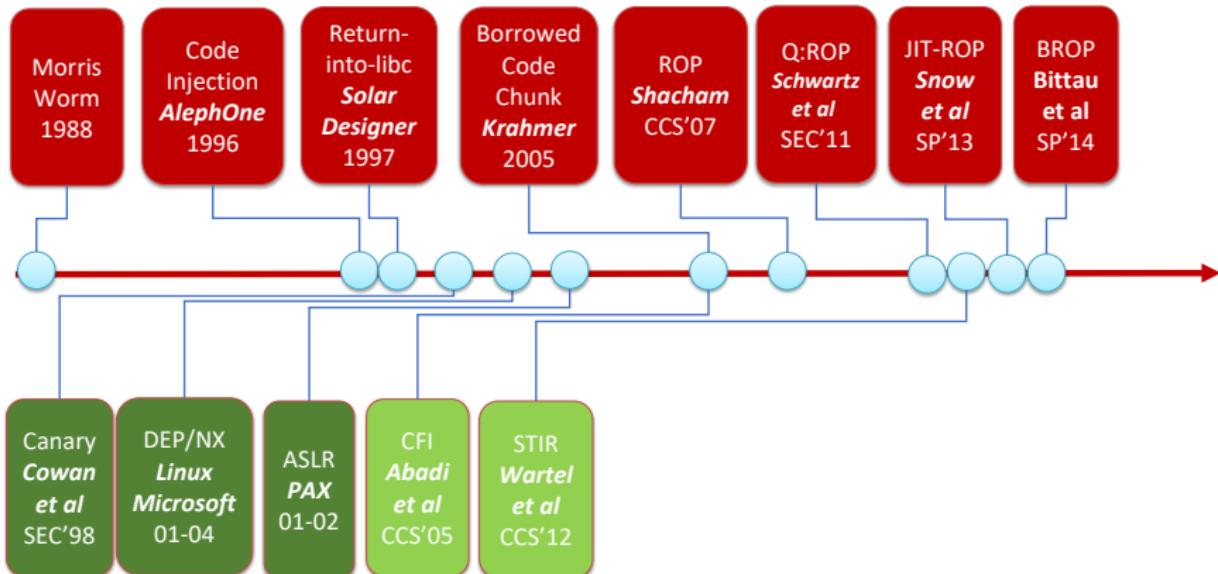
# The Arm Race (Attack and Defense) w/ Memory Corruptions

Attacks

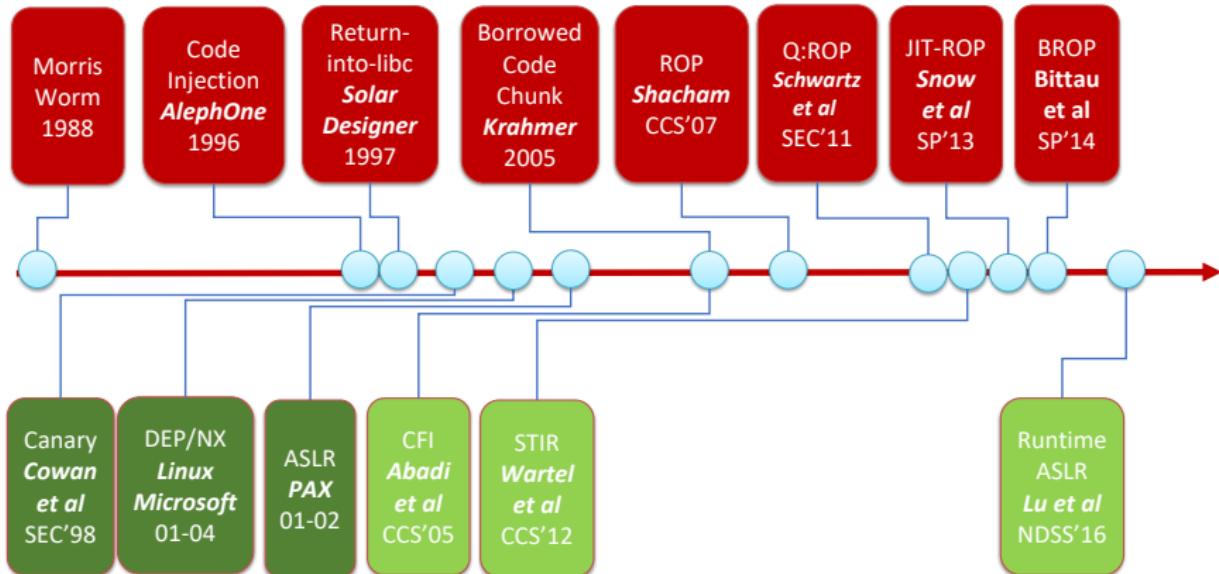


Defense

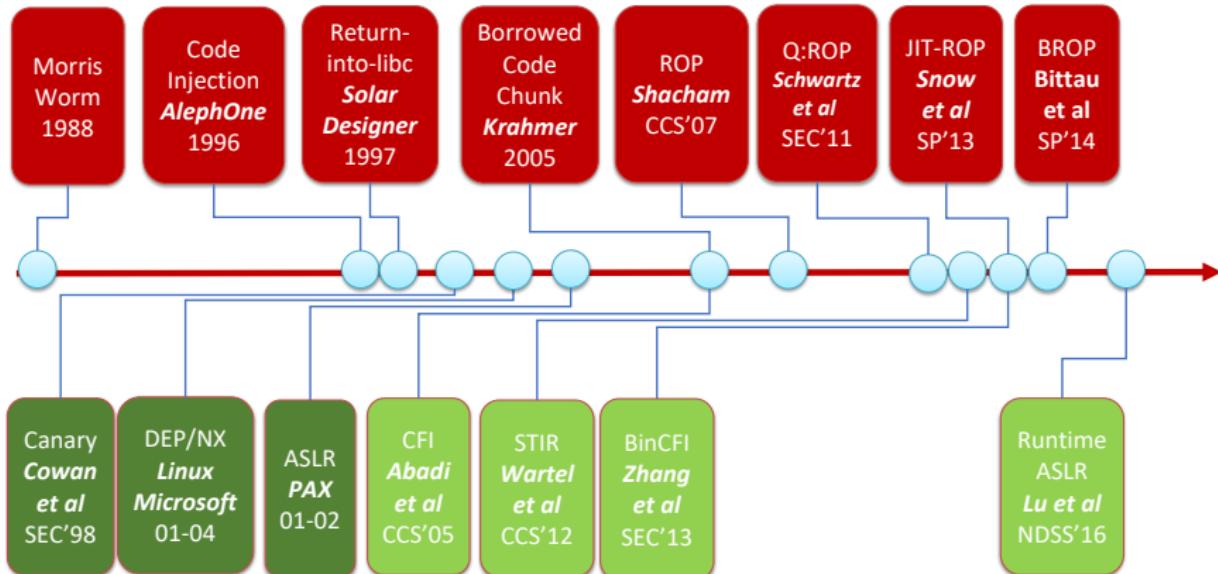
# The Arm Race (Attack and Defense) w/ Memory Corruptions



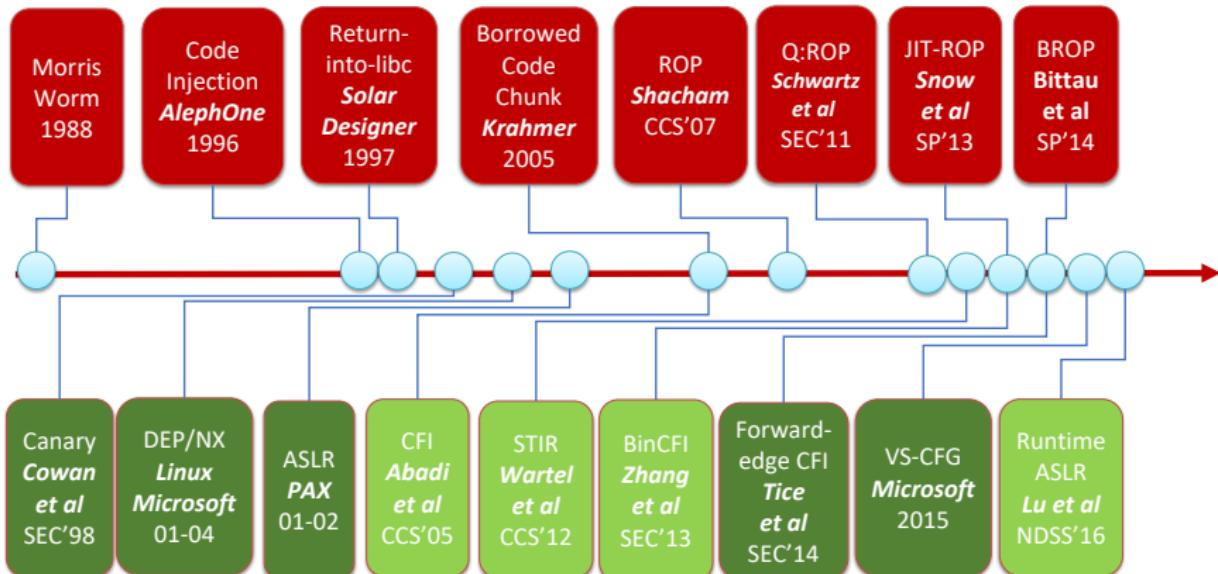
# The Arm Race (Attack and Defense) w/ Memory Corruptions



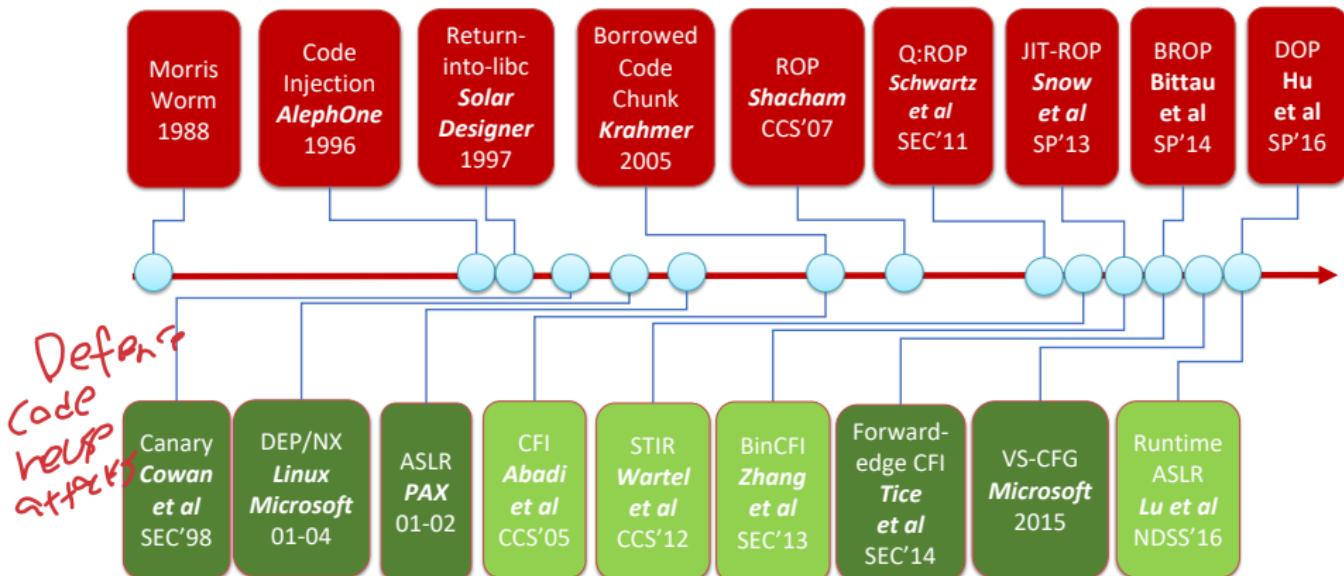
# The Arm Race (Attack and Defense) w/ Memory Corruptions



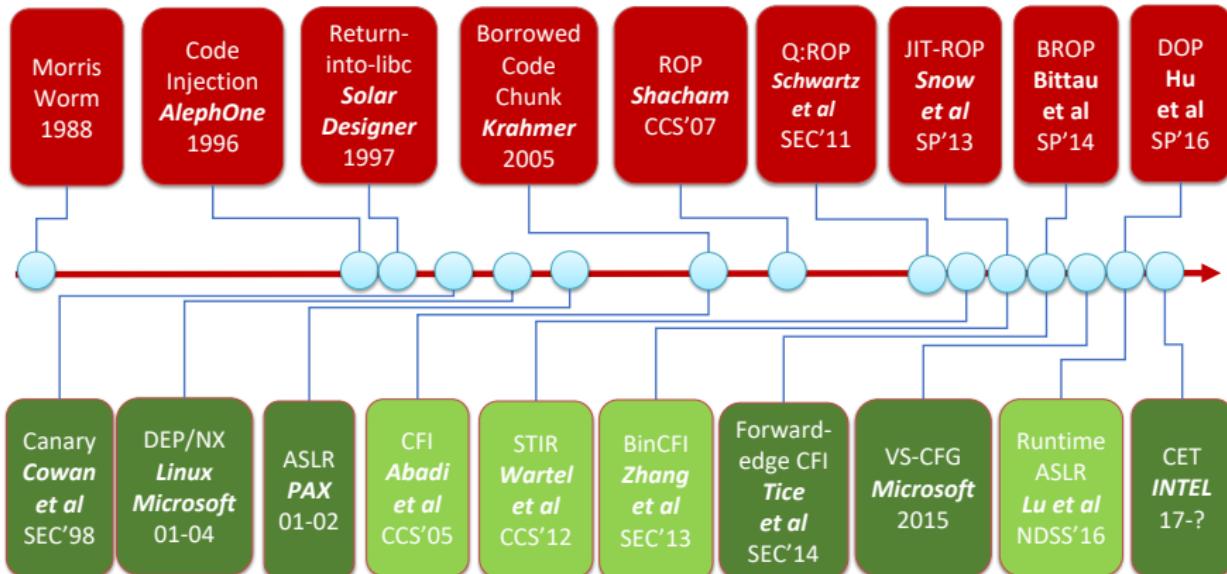
# The Arm Race (Attack and Defense) w/ Memory Corruptions



# The Arm Race (Attack and Defense) w/ Memory Corruptions

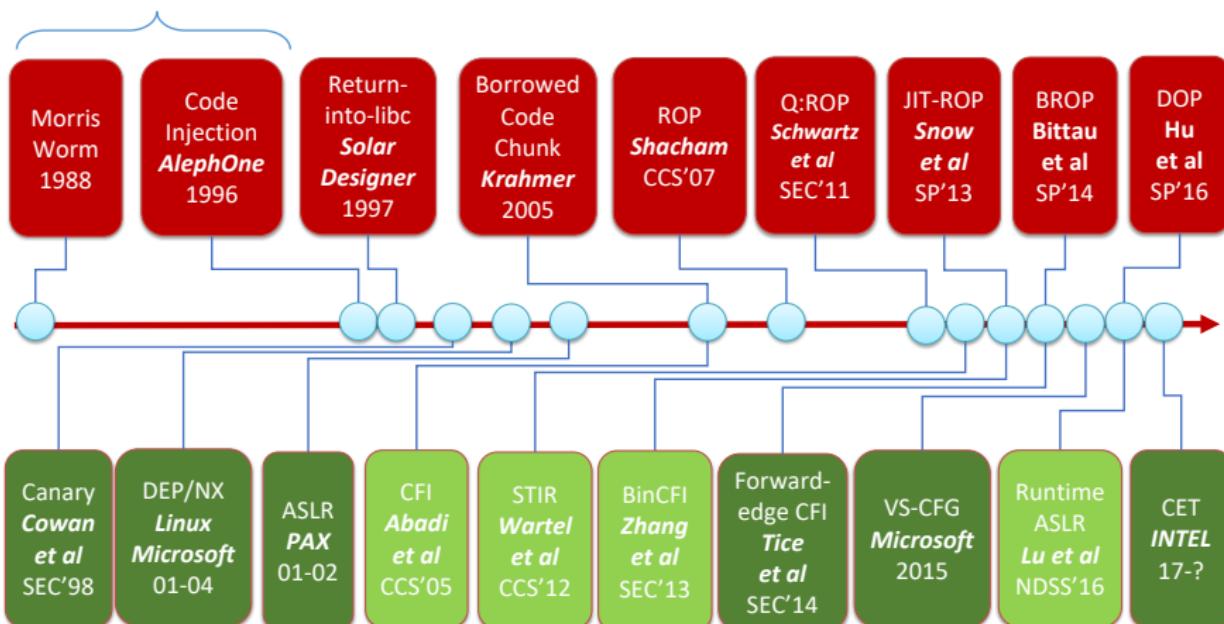


# The Arm Race (Attack and Defense) w/ Memory Corruptions

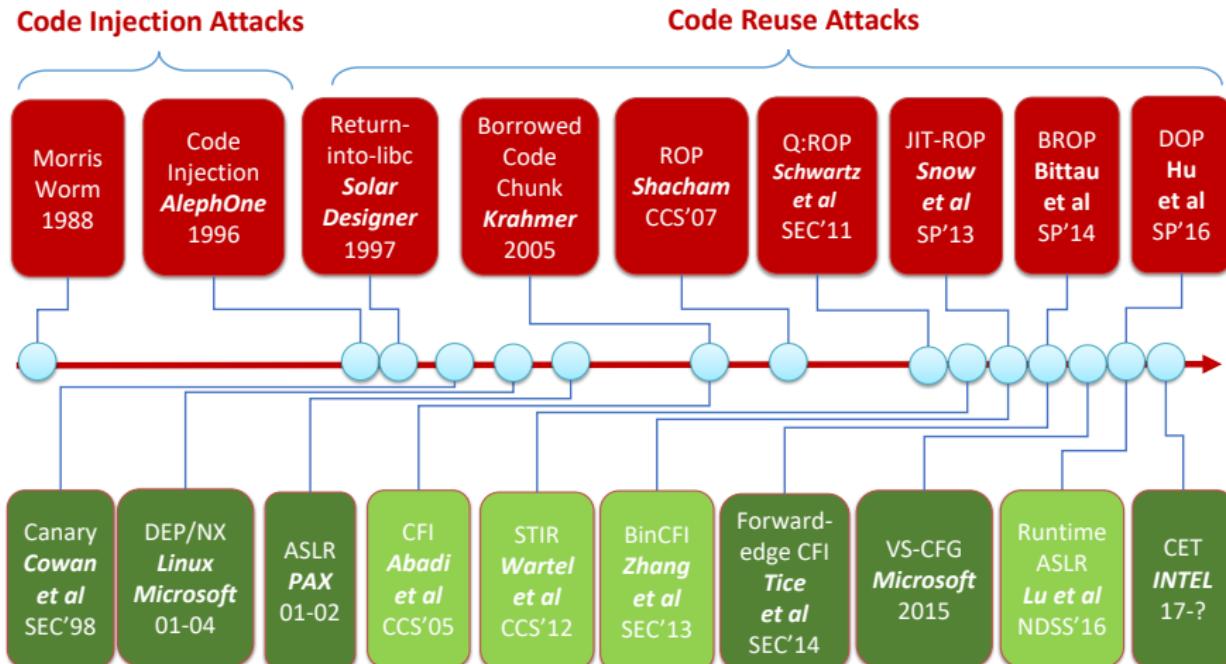


# The Arm Race (Attack and Defense) w/ Memory Corruptions

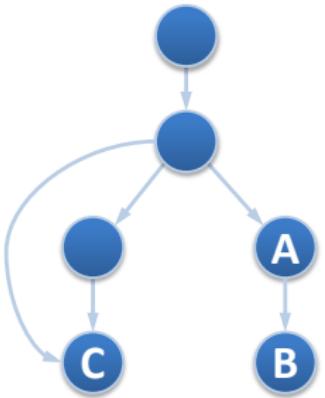
## Code Injection Attacks



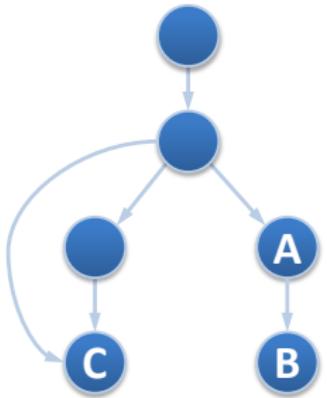
# The Arm Race (Attack and Defense) w/ Memory Corruptions



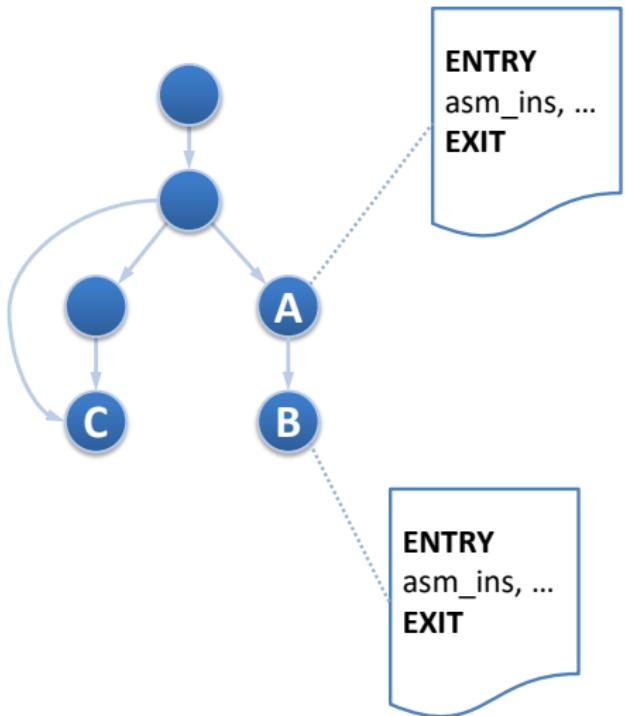
## General Principle of Code Injection Attacks



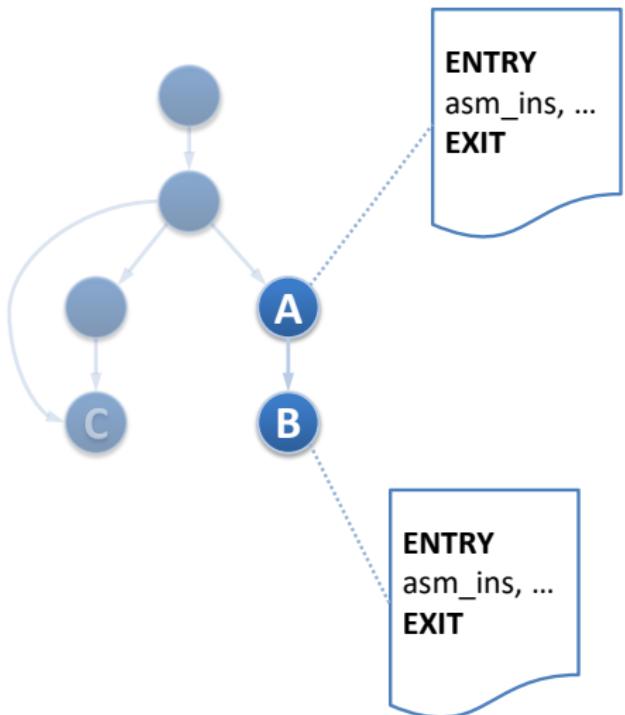
# General Principle of **Code Injection Attacks**



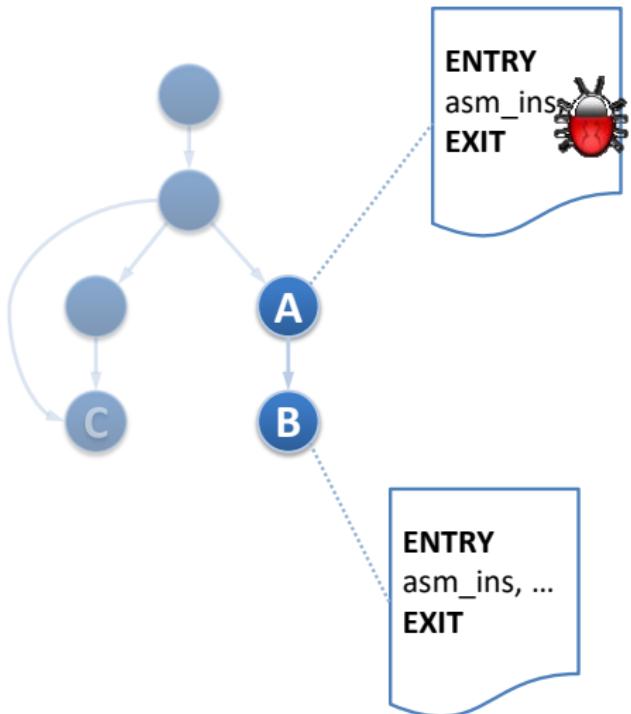
# General Principle of Code Injection Attacks



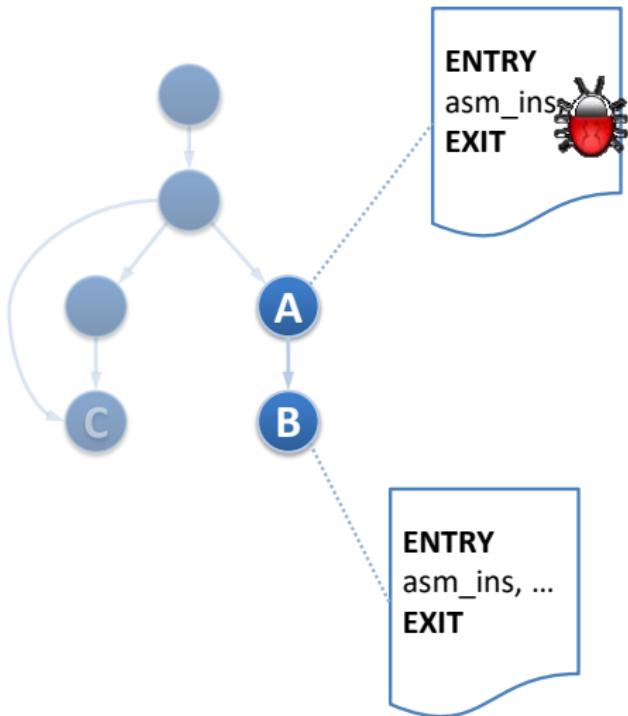
# General Principle of Code Injection Attacks



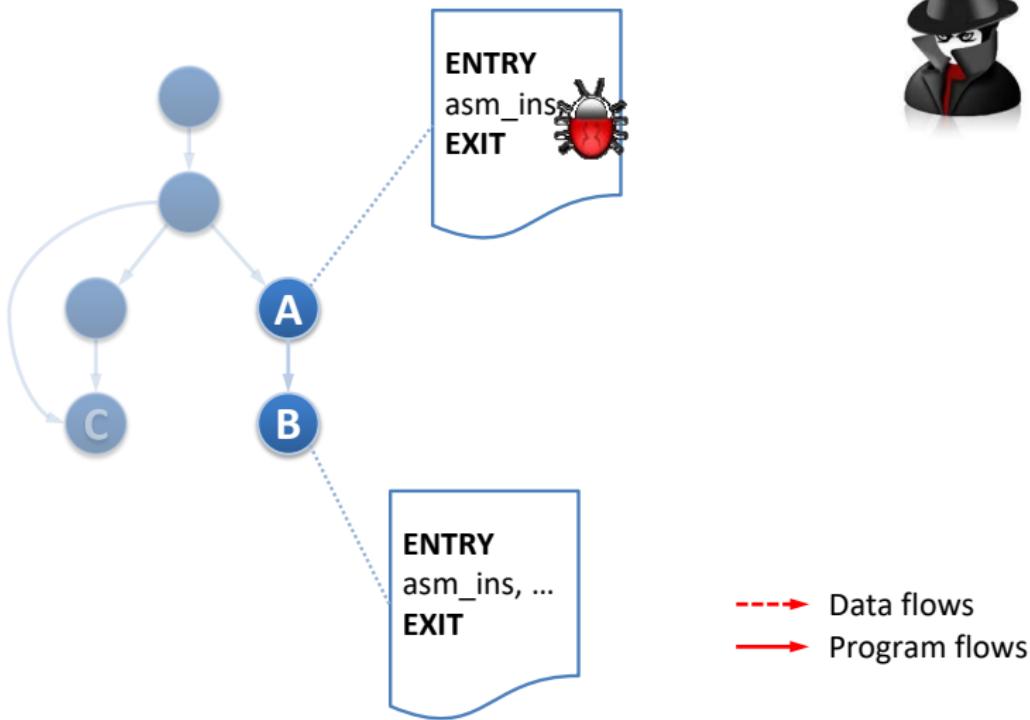
# General Principle of Code Injection Attacks



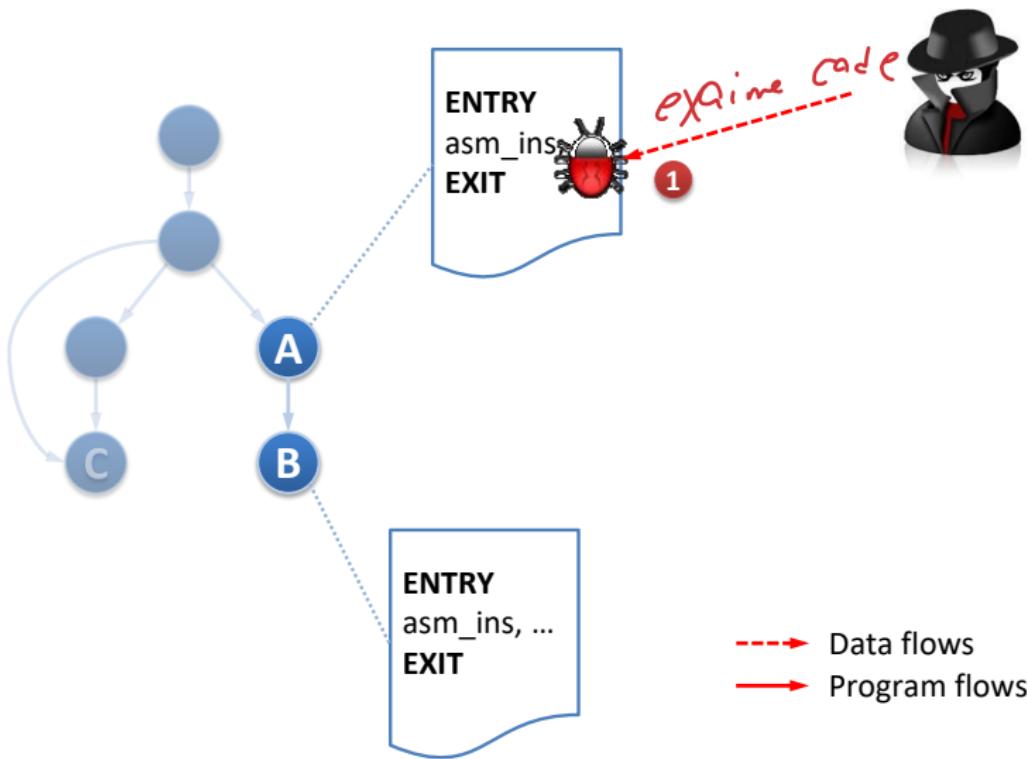
# General Principle of **Code Injection Attacks**



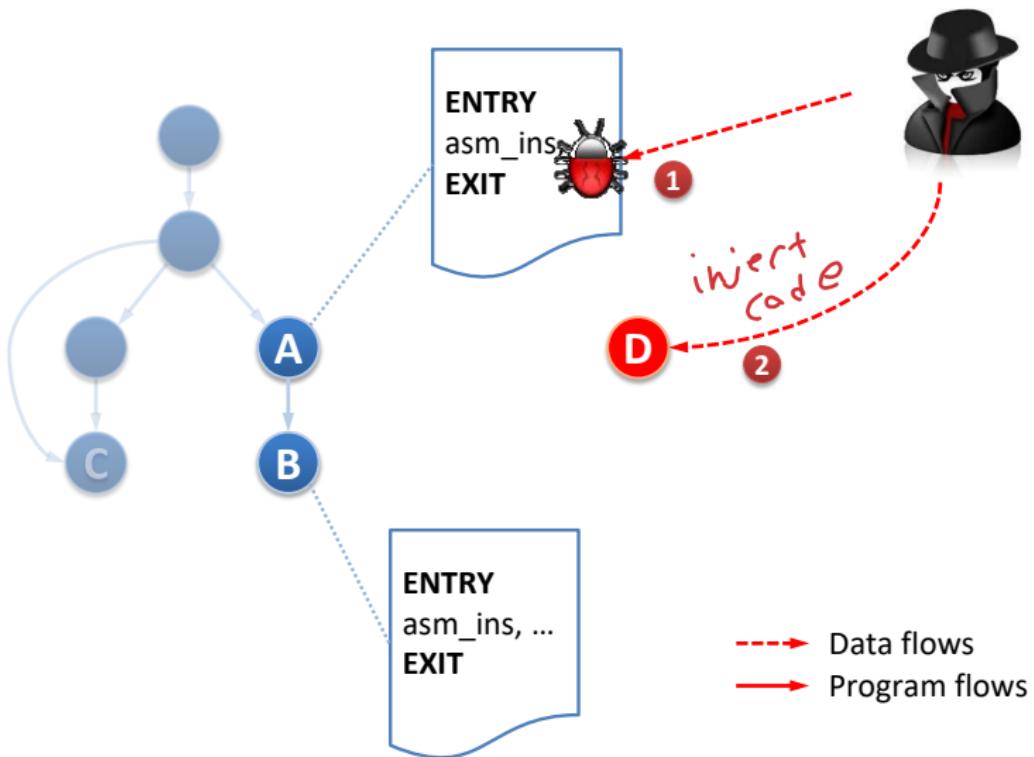
# General Principle of Code Injection Attacks



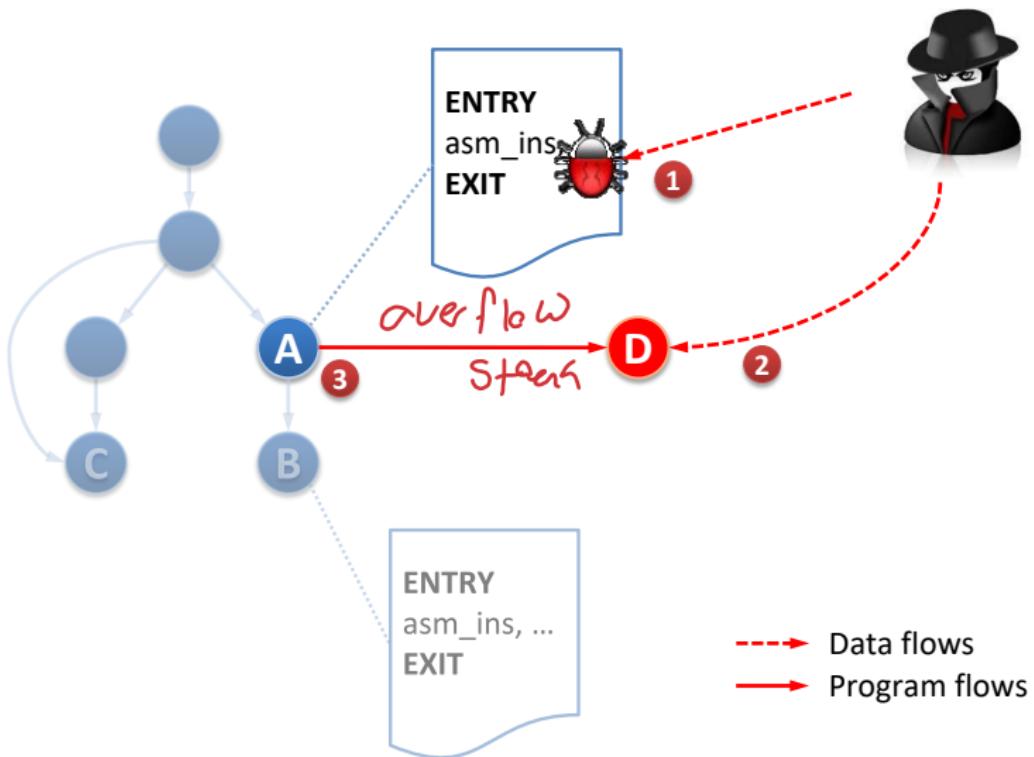
# General Principle of Code Injection Attacks



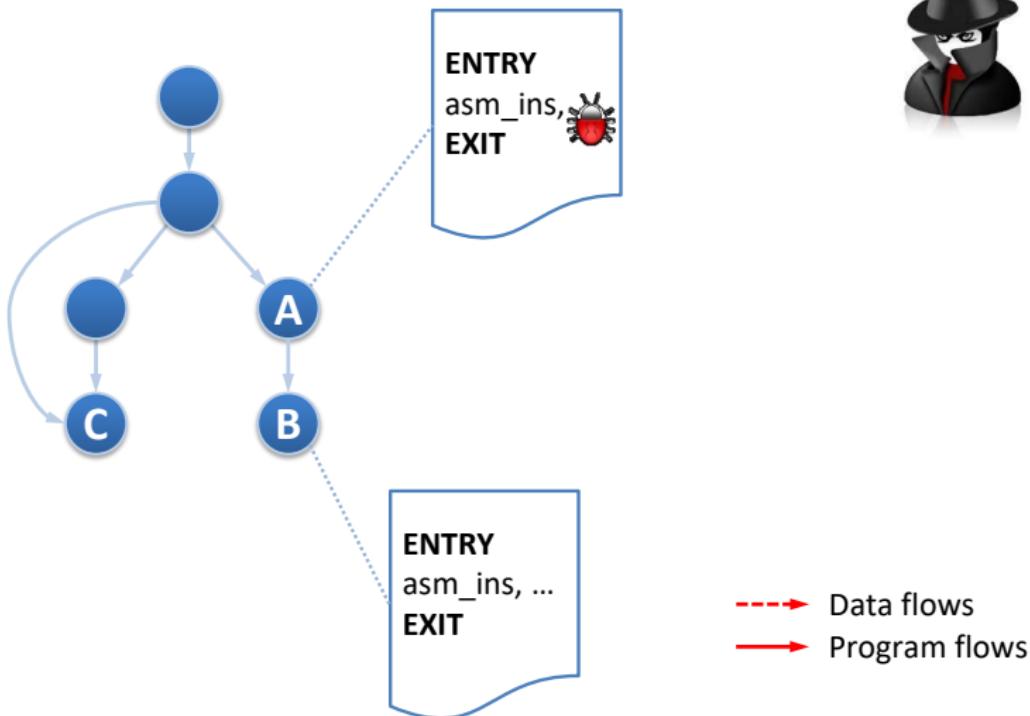
# General Principle of Code Injection Attacks



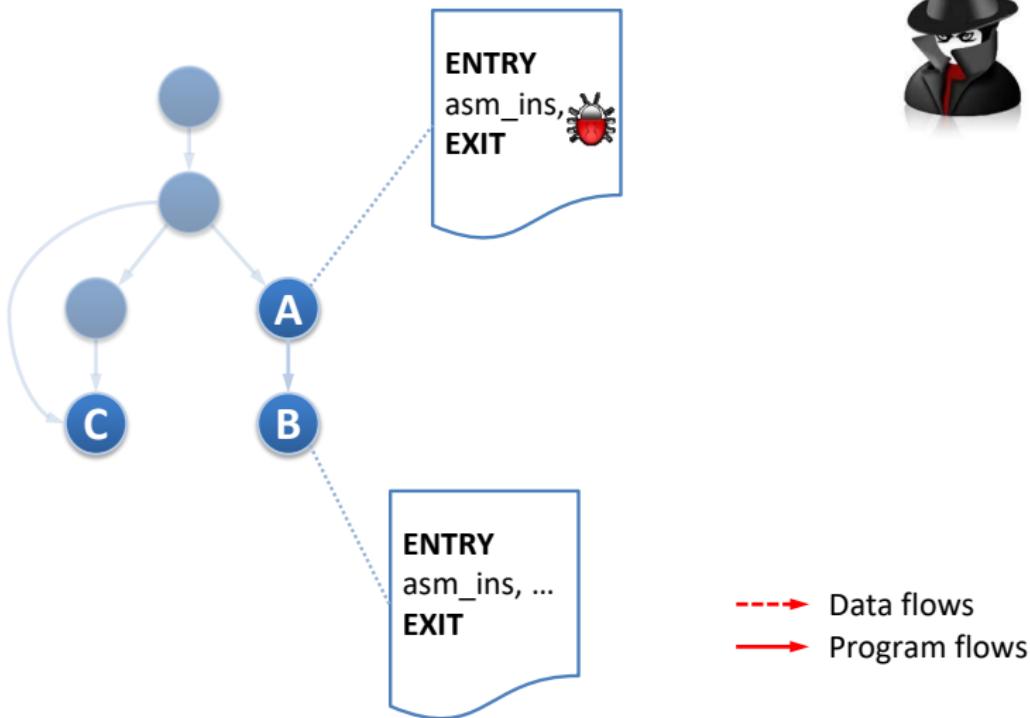
# General Principle of Code Injection Attacks



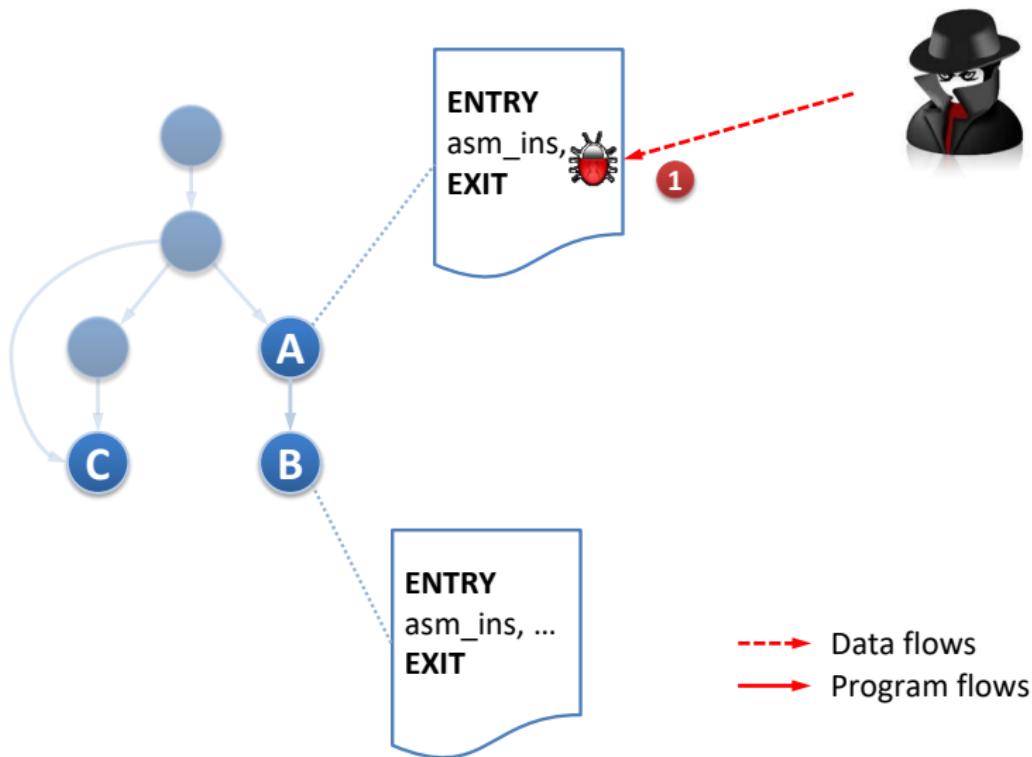
## General Principle of Code Reuse Attacks



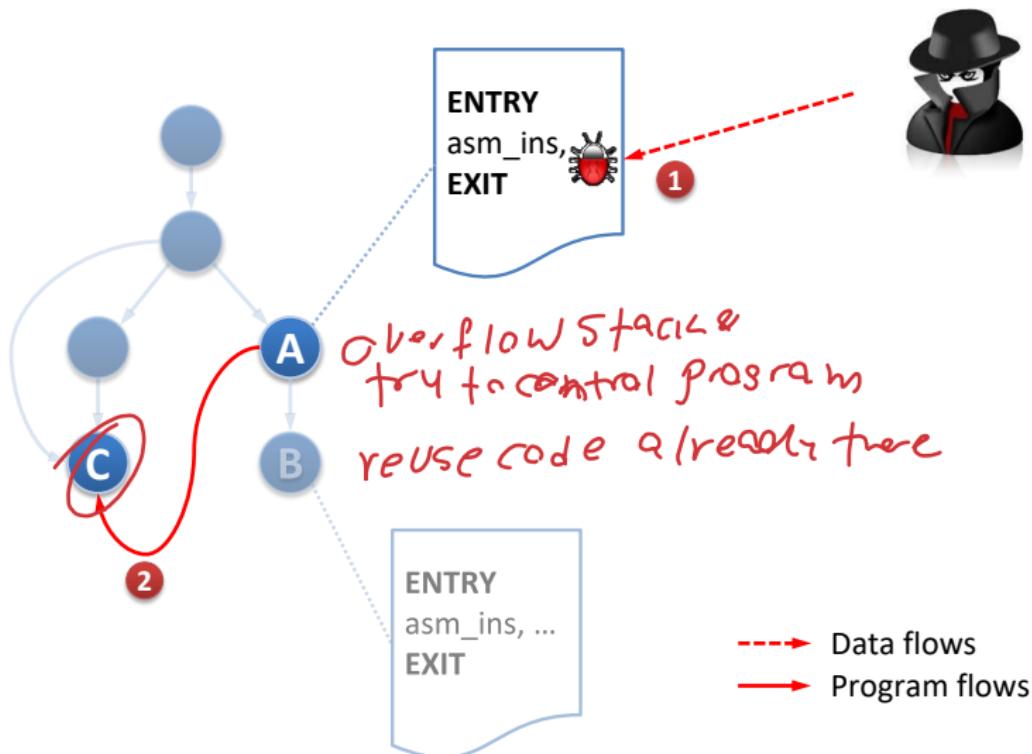
# General Principle of **Code Reuse Attacks**



# General Principle of Code Reuse Attacks

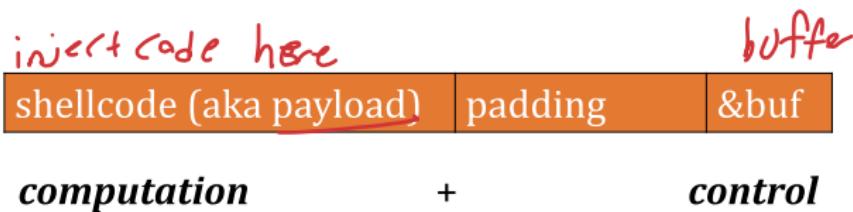


# General Principle of Code Reuse Attacks



# Memory Corruption → Control Flow Hijack

*overwritten pointers*



## Steps

- ① Leveraging memory corruption vulnerabilities
- ② Constructing special input (exploit):
  - ▶ Performing certain **computation**
  - ▶ Gaining the **control** of the program

## Same principle, different mechanism

- ① Code injection
- ② Code reuse

# The Run-time Defenses Against Memory Corruptions

**Stack Canaries**

*guardian stack*

**Data Execution Prevention (DEP)**

**Code and Data Randomization**

**Control-Flow Integrity**

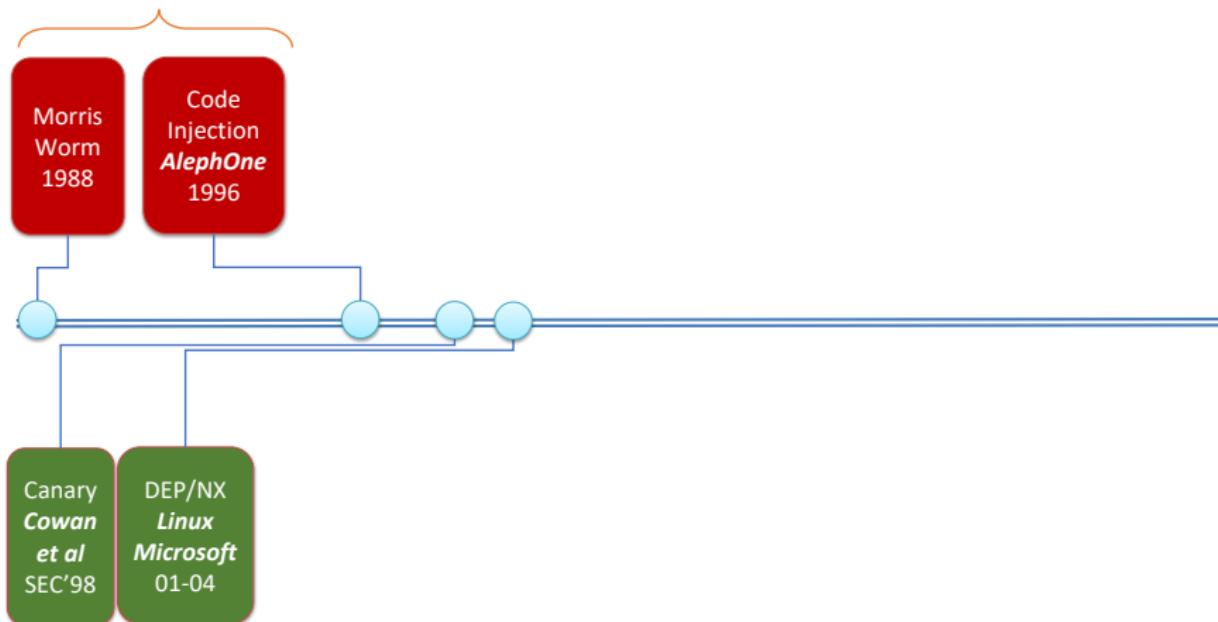
# Defense Against Code Injection

## Code Injection Attacks



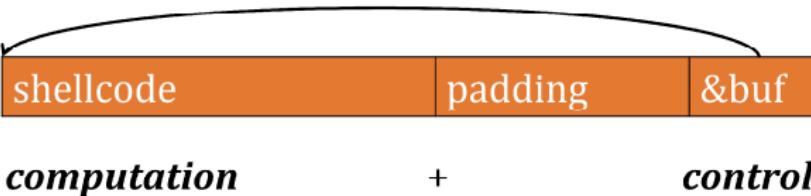
# Defense Against Code Injection

## Code Injection Attacks

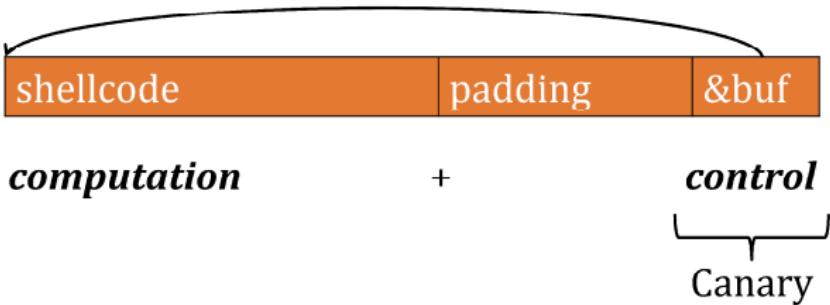


# How Canary Defending Against Control Flow Hijacks

Overflow buffer



# How Canary Defending Against Control Flow Hijacks



# Canary Defense

*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



Warning

Introduction  
oooooooooo

Canary and DEP  
○○●○○

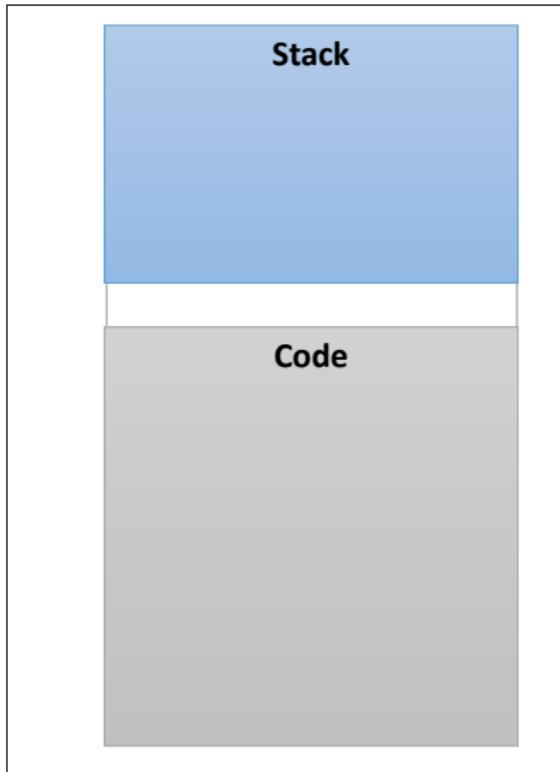
Randomization  
oooooo

Control Flow Integrity  
ooo

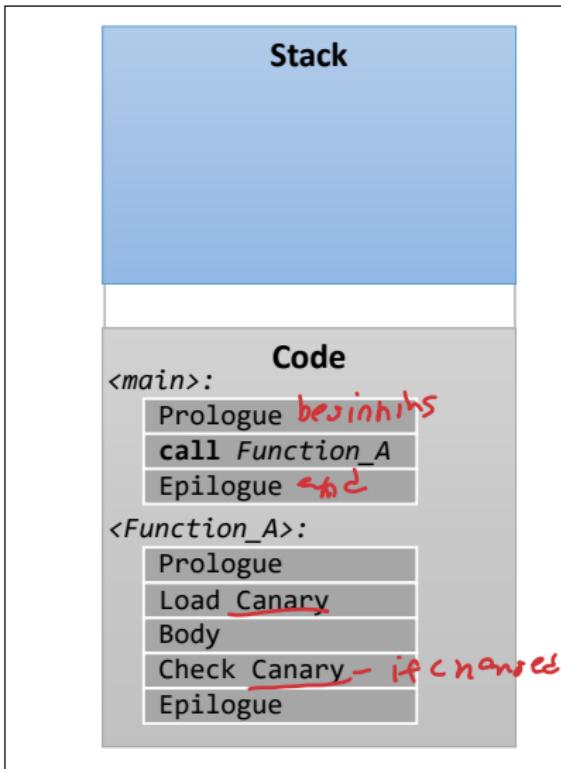
Discussion  
ooooo

Summary  
oo

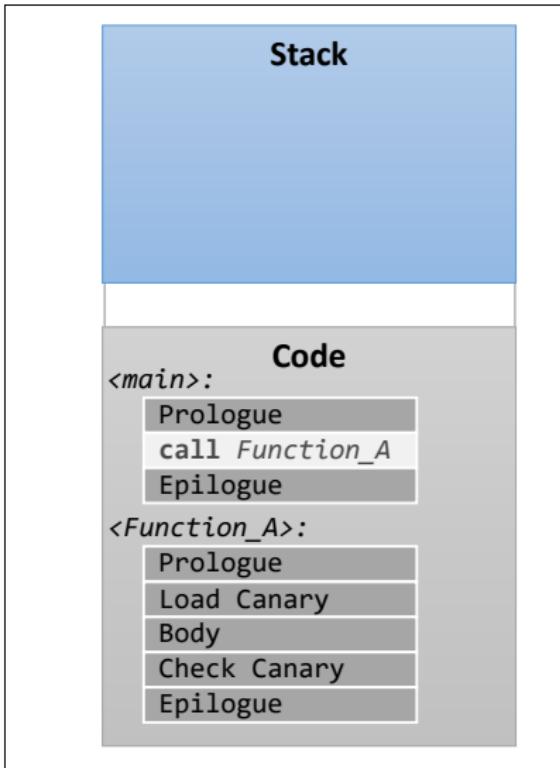
# Canary Defense



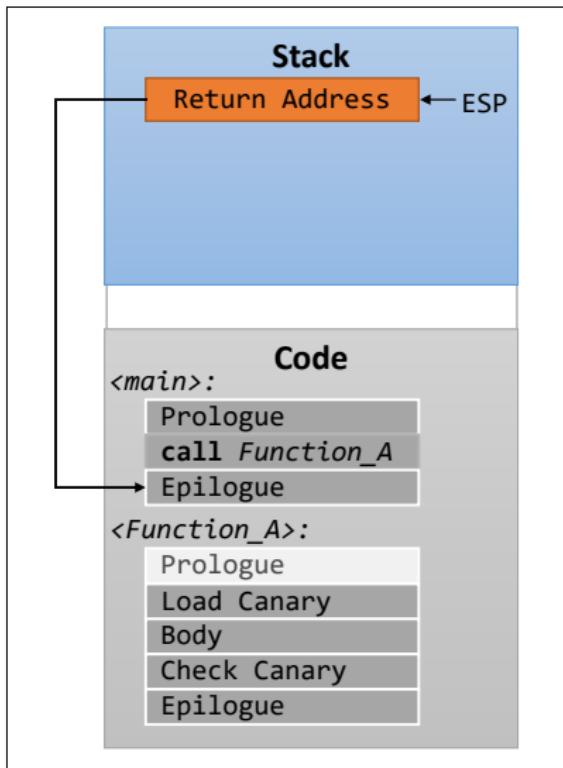
# Canary Defense



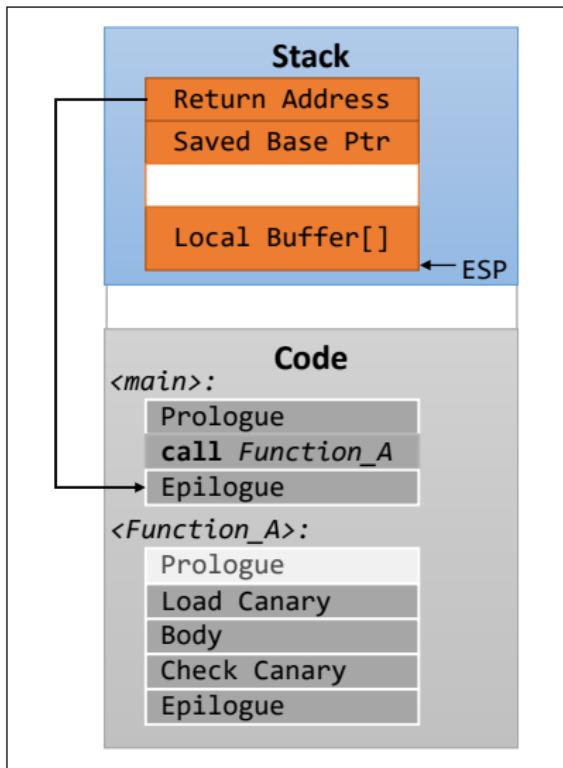
# Canary Defense



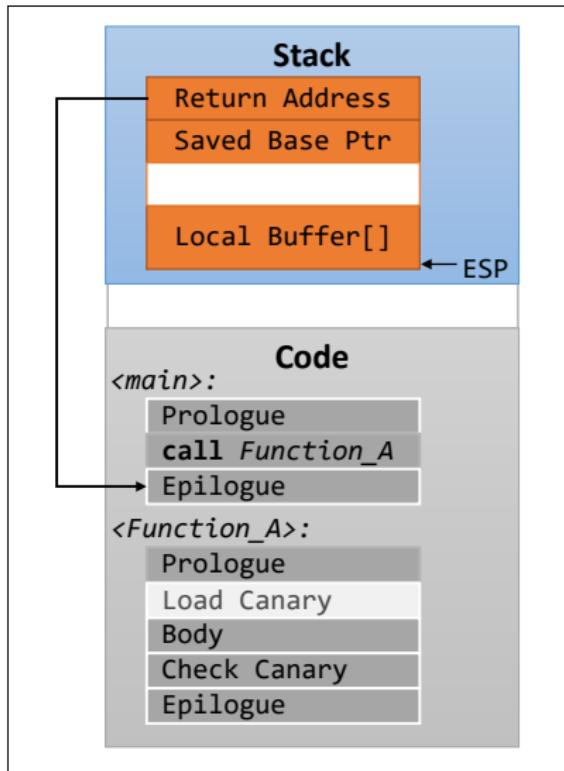
# Canary Defense



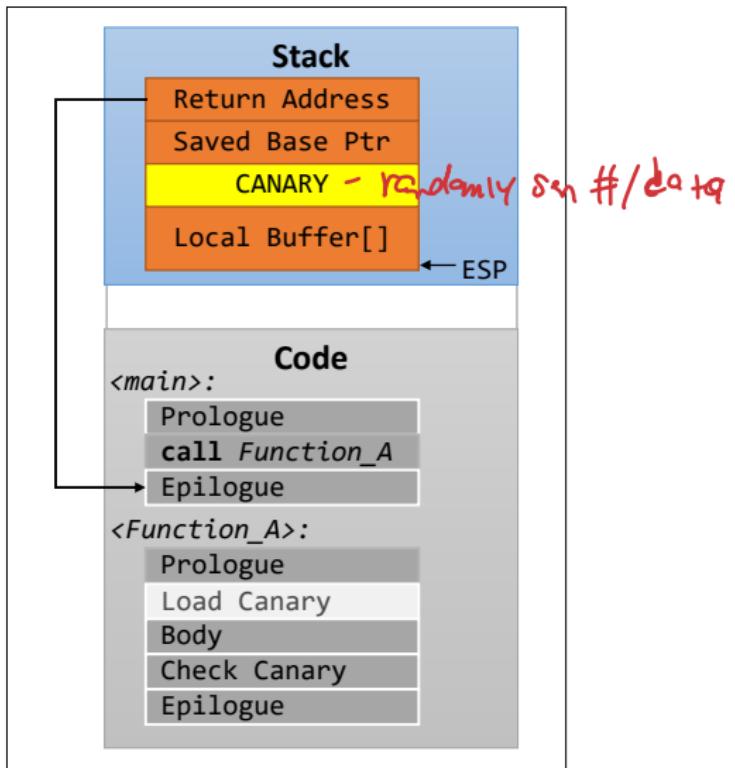
# Canary Defense



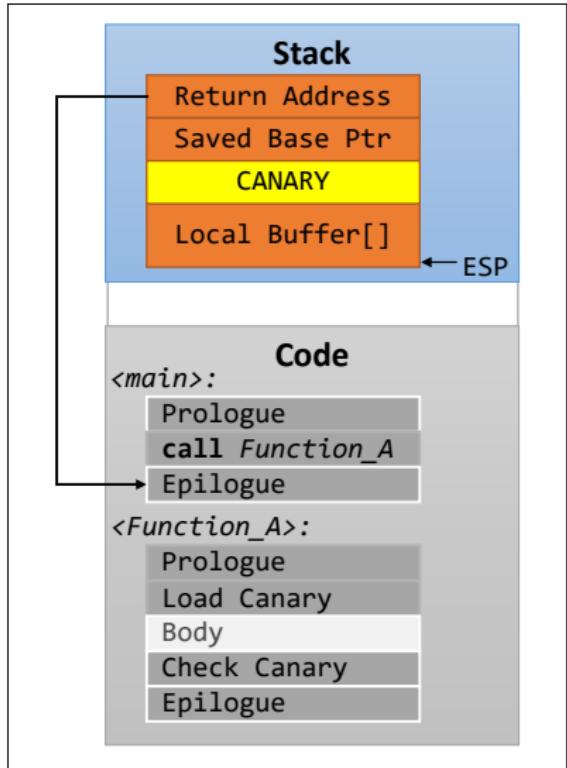
# Canary Defense



# Canary Defense

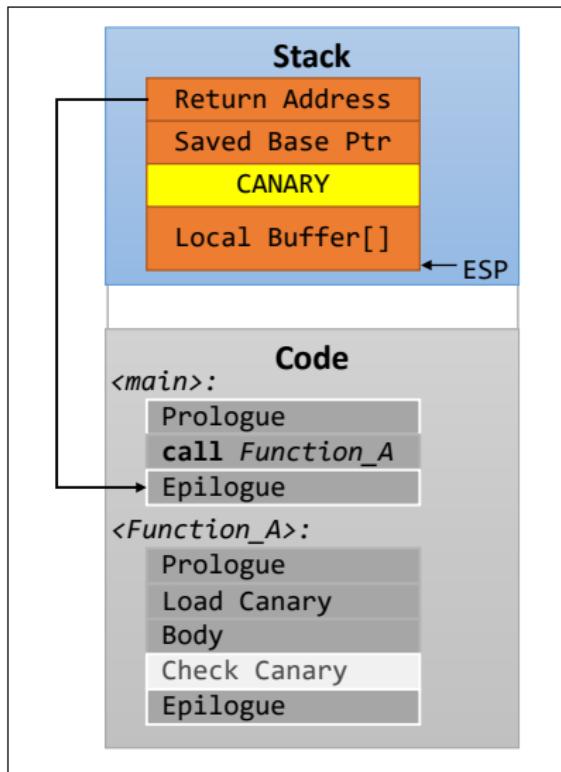


# Canary Defense

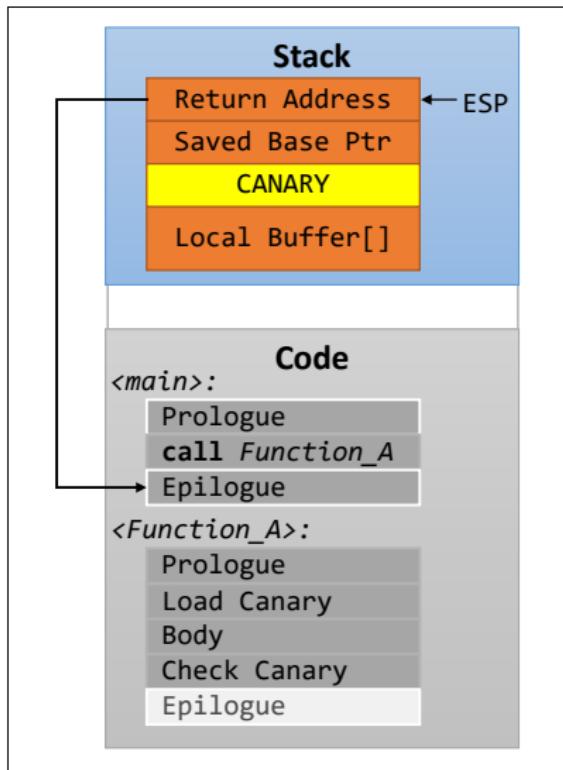


B/C randomly generated  
if the attacker override the  
generated data the  
program will know

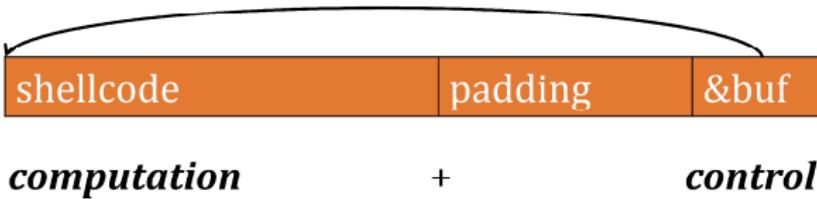
# Canary Defense



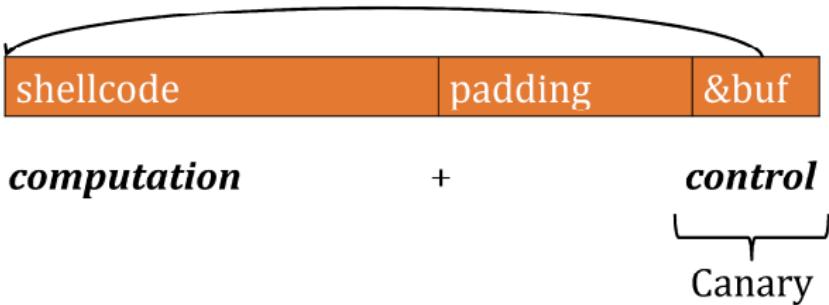
# Canary Defense



# How to defend against control flow hijacks



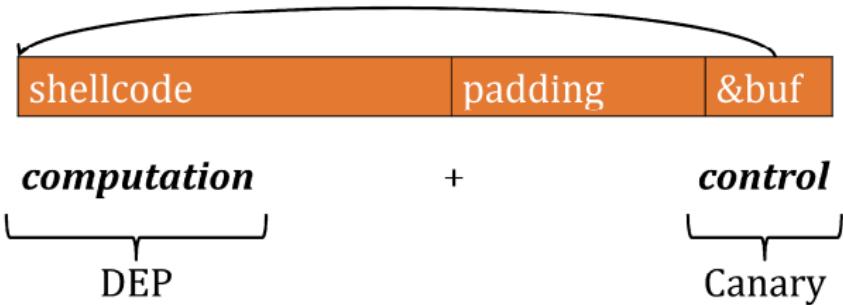
# How to defend against control flow hijacks



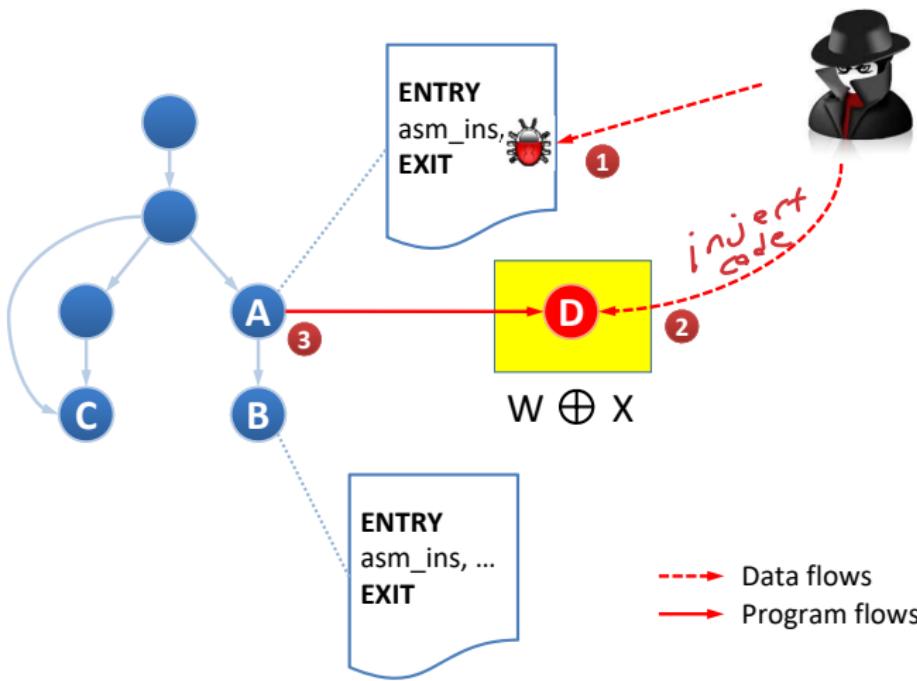
## How to defend against control flow hijacks

We can stop people injecting code into memory

## Data Execution Prevention

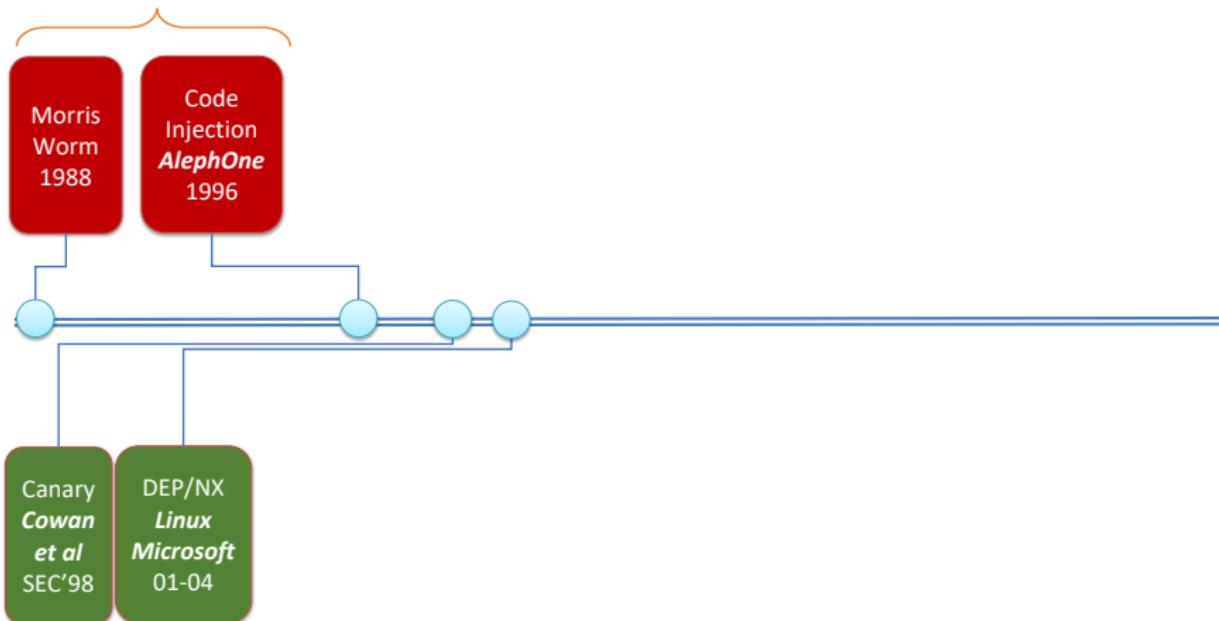


# Data Execution Prevention



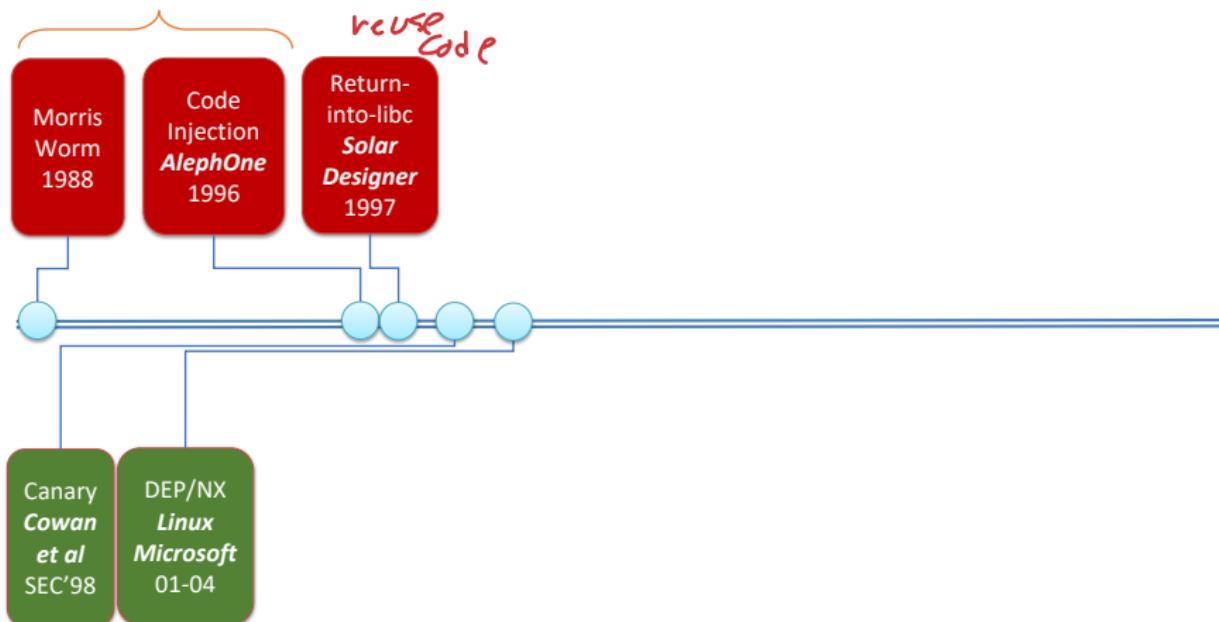
# Defense Using Randomization

## Code Injection Attacks



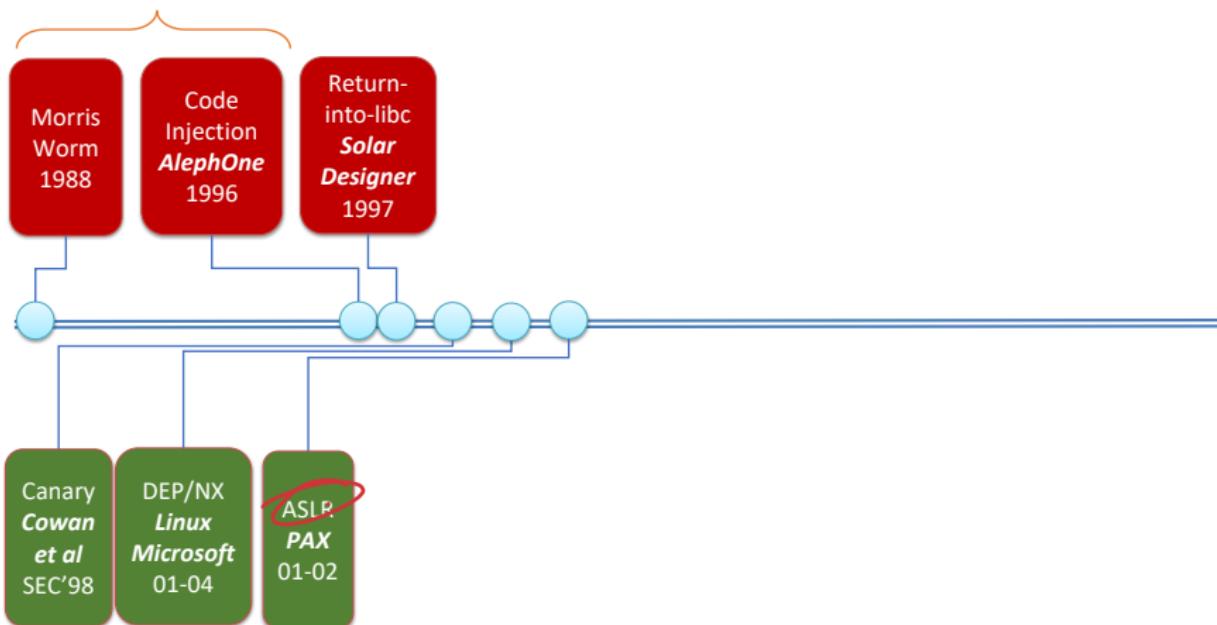
# Defense Using Randomization

## Code Injection Attacks

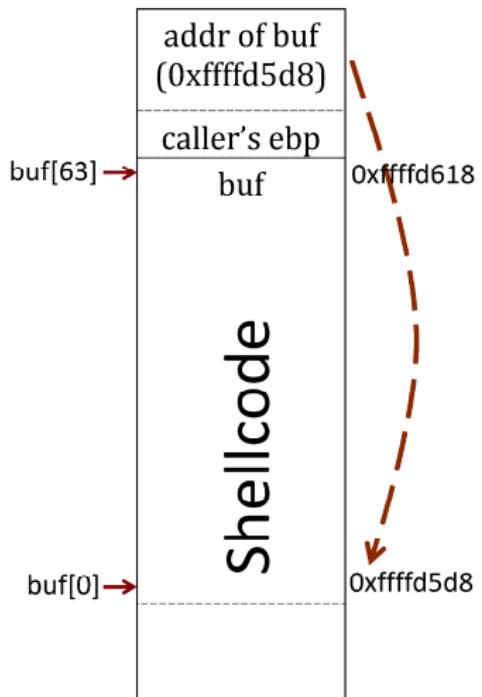


# Defense Using Randomization

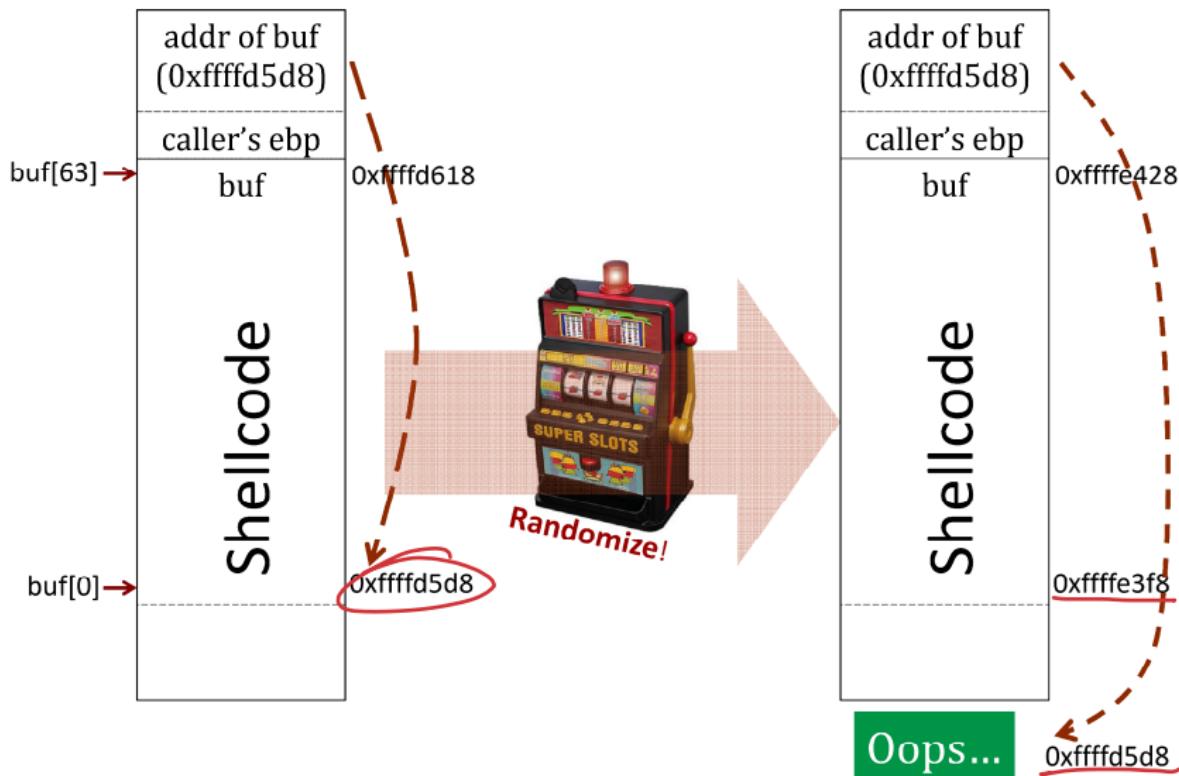
## Code Injection Attacks



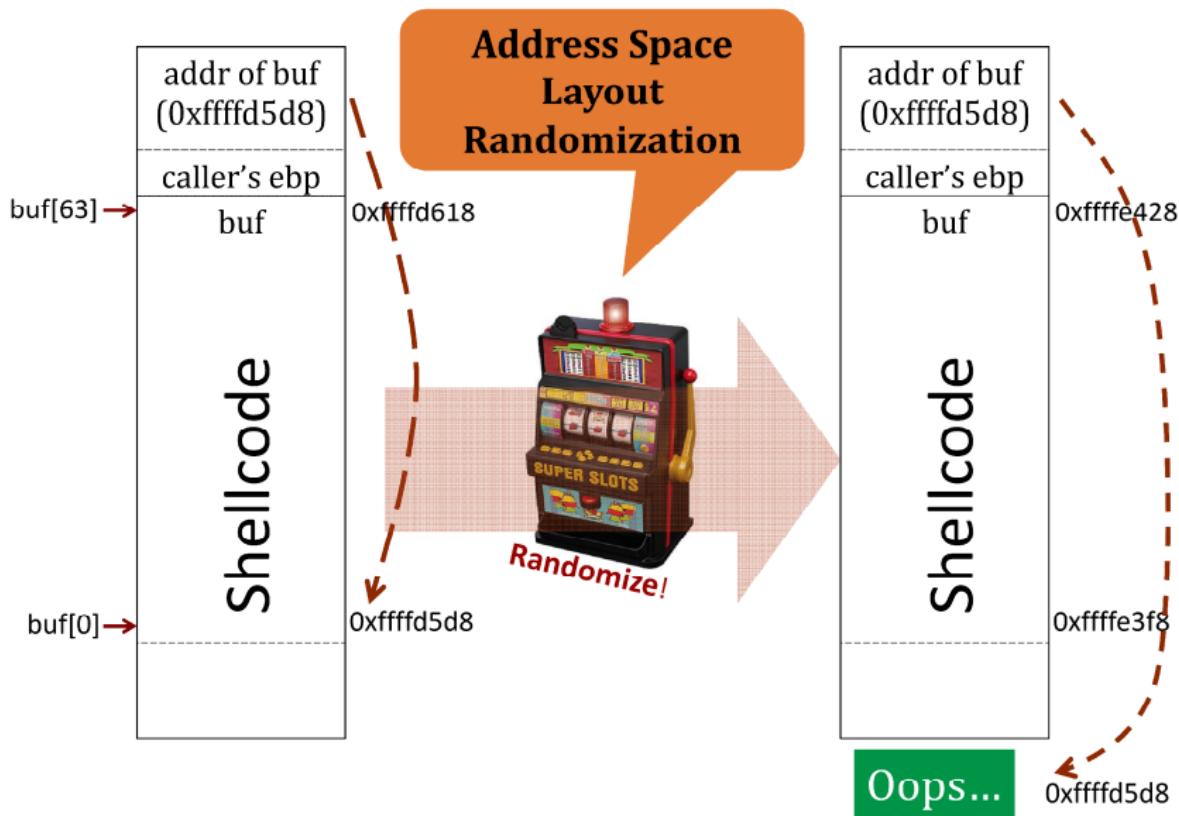
# Address Space Layout Randomization (ASLR)



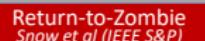
# Address Space Layout Randomization (ASLR)



# Address Space Layout Randomization (ASLR)



# Recent Efforts on Code Reuse Attack

1997	ret2libc <i>Solar Designer</i>
2001	Advanced ret2libc <i>Nergal</i>
2005	Borrowed Code Chunk Exploitation <i>Krahmer</i>
2007	ROP on x86 <i>Shacham (CCS)</i>
2008	ROP on SPARC <i>Buchanan et al (CCS)</i> ROP on Atmel AVR <i>Francillon et al (CCS)</i>
2009	ROP Rootkits <i>Hund et al (USENIX)</i> ROP on PowerPC <i>FX Lindner (BlackHat)</i> ROP on ARM/iOS <i>Miller et al (BlackHat)</i>
2010	ROP without Returns <i>Checkoway et al (CCS)</i> Practical ROP <i>Zovi (RSA Conference)</i> Pwn2Own (iOS/IE) <i>Iozzo et al / Nils</i>
2011/ 2012	JIT-ROP <i>Snow et al (IEEE S&amp;P)</i>
2013	     Real-World Exploits
2014	Blind ROP <i>Bittau et al (IEEE S&amp;P)</i> Out-Of-Control <i>Göktas et al (IEEE S&amp;P)</i> Stitching Gadgets <i>Davi et al (USENIX)</i> Flushing Attacks <i>Schuster et al (RAID)</i> ROP is Dangerous <i>Carlini et al (USENIX)</i>
2015	 Losing Control <i>Liebchen et al (CCS)</i> Control-Flow Bending <i>Carlini et al (USENIX)</i>
2016	
2017	Address Oblivious Code reuse <i>Rudd et al (NDSS)</i> Code-reuse attacks for the Web <i>Lekeis et al (CCS)</i>

Introduction  
oooooooooooo

Canary and DEP  
ooooo

Randomization  
ooo●oo

Control Flow Integrity  
ooo

Discussion  
oooo

Summary  
oo

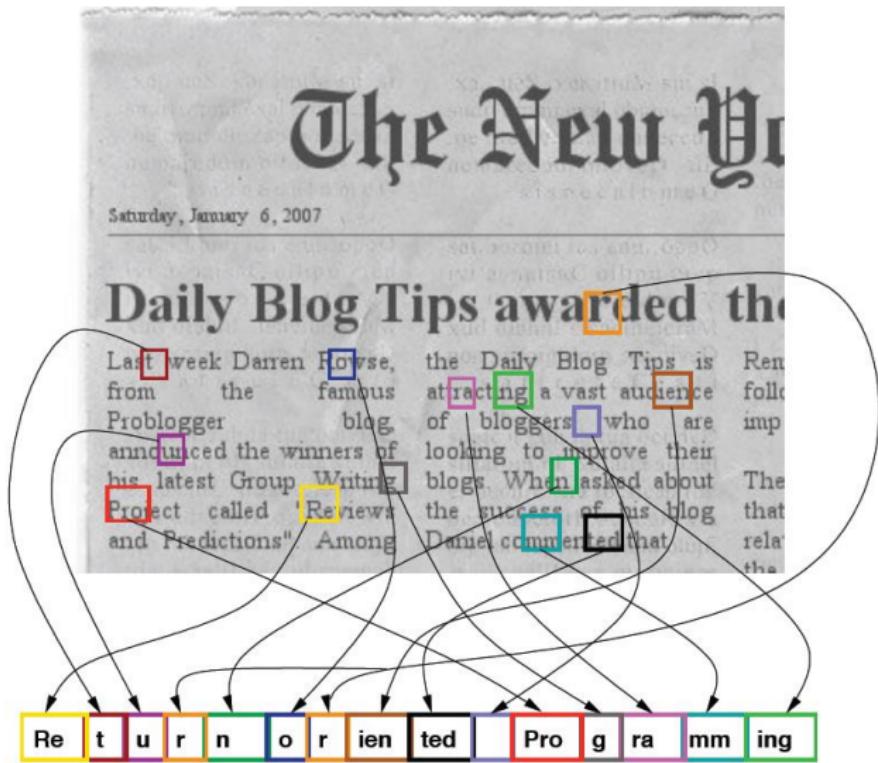
# Return Oriented Programming (ROP)



# Return Oriented Programming (ROP)

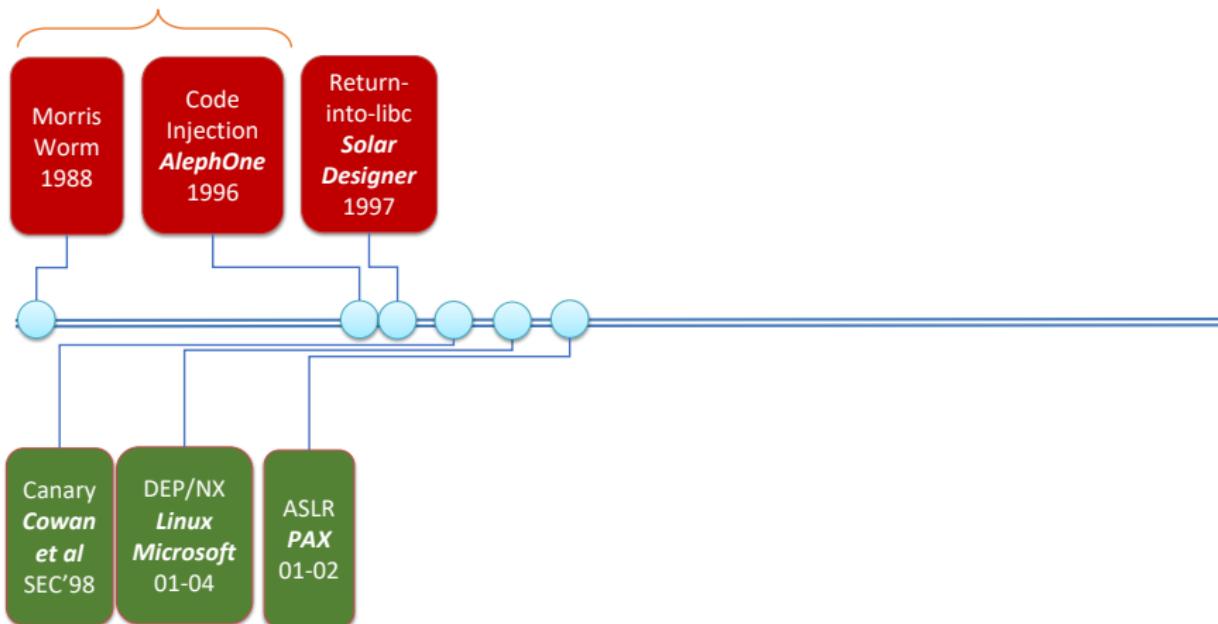


# Return Oriented Programming (ROP)

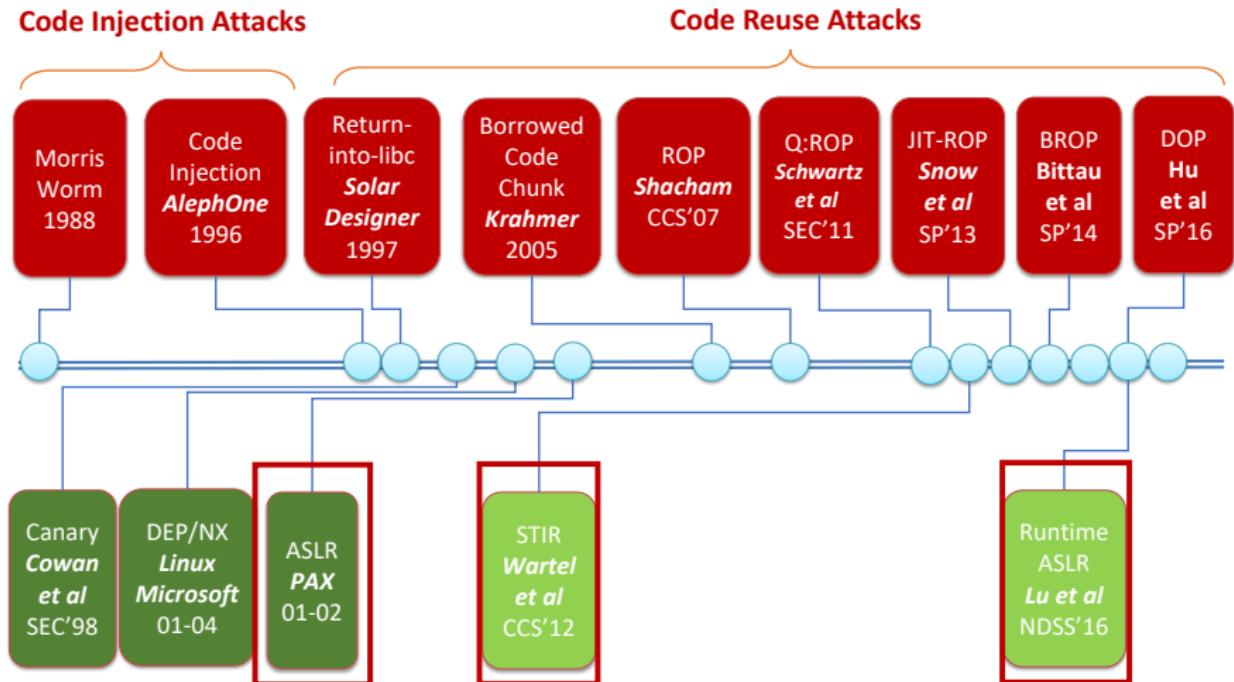


# More Defenses Using Randomization

## Code Injection Attacks



# More Defenses Using Randomization



# From Coarse-Grained to More Fine-Grained ASLR

Application Run 1



Program Memory

# From Coarse-Grained to More Fine-Grained ASLR

Application Run 1



Application Run 2

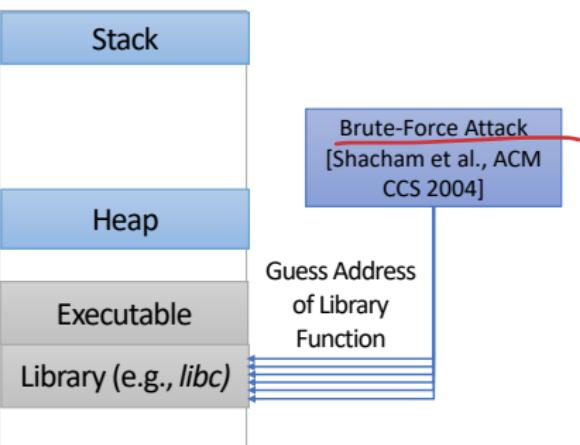


# From Coarse-Grained to More Fine-Grained ASLR

Application Run 1



Application Run 2



# From Coarse-Grained to More Fine-Grained ASLR

## Application Run 1

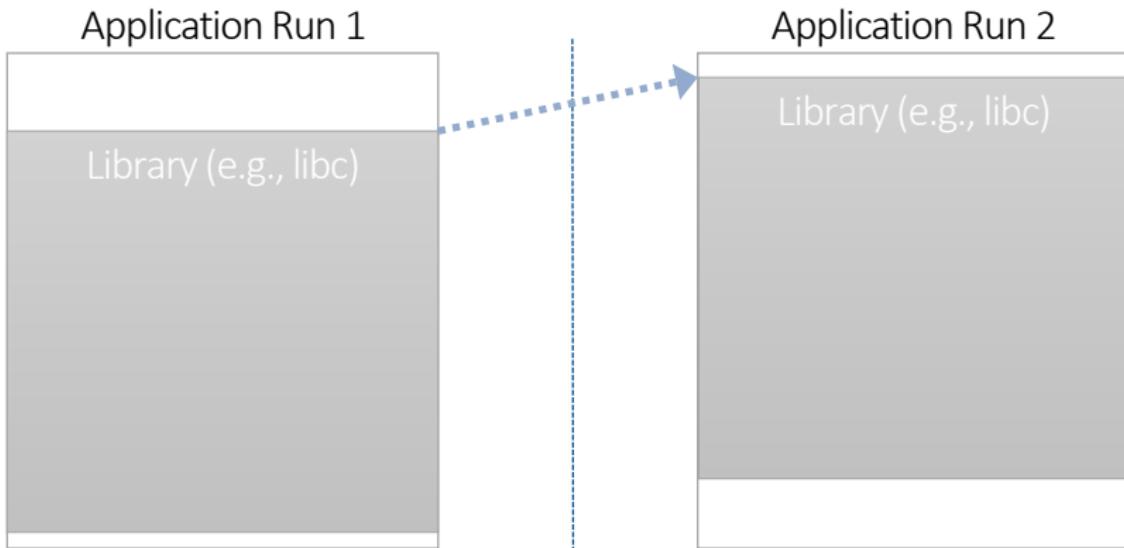


# From Coarse-Grained to More Fine-Grained ASLR

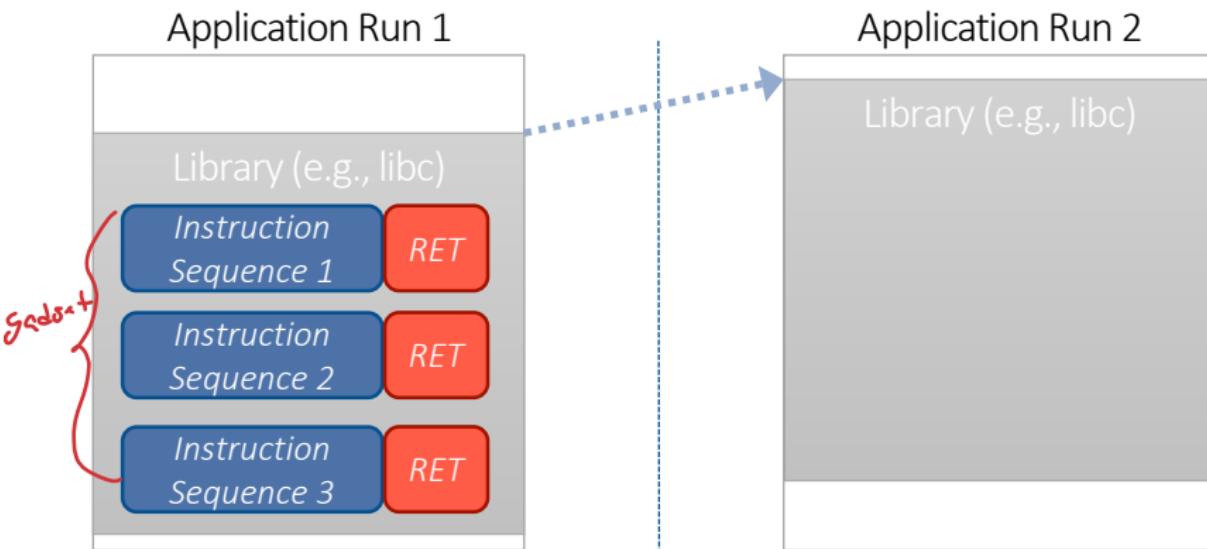
## Application Run 1



# From Coarse-Grained to More Fine-Grained ASLR



# From Coarse-Grained to More Fine-Grained ASLR



Introduction  
oooooooooo

Canary and DEP  
ooooo

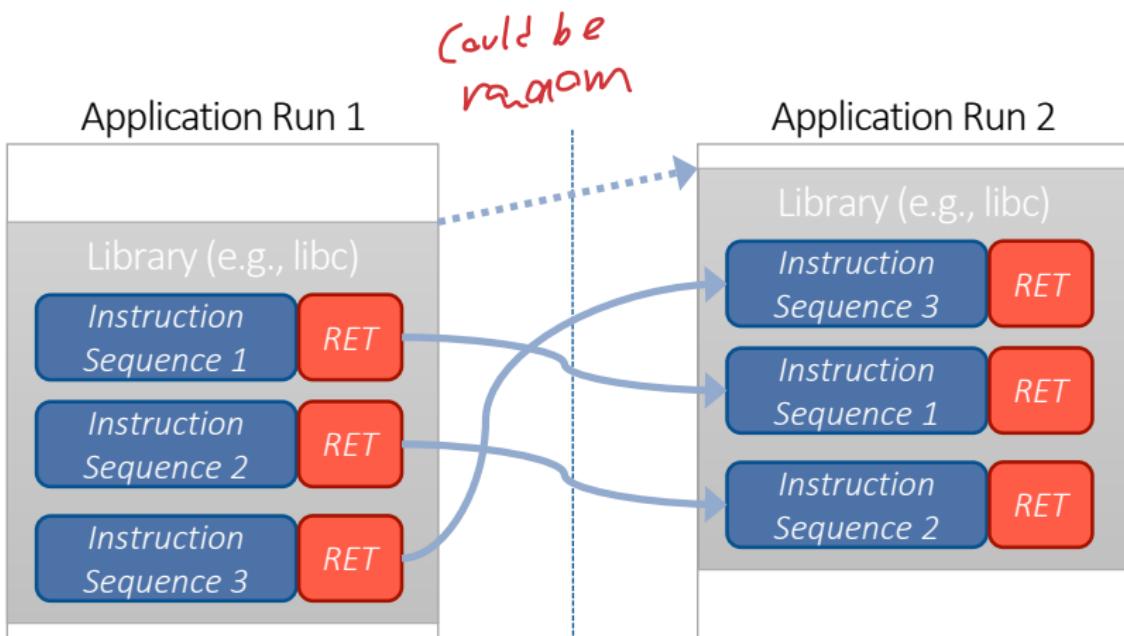
Randomization  
ooooo●

Control Flow Integrity  
ooo

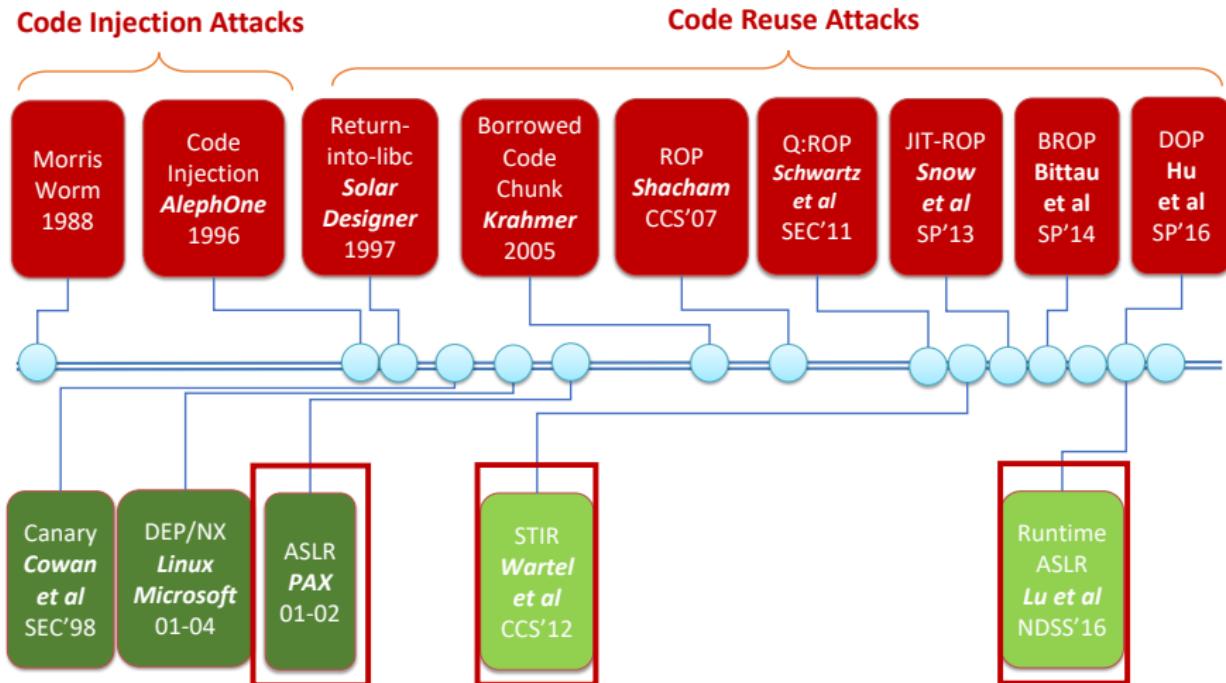
Discussion  
oooo

Summary  
oo

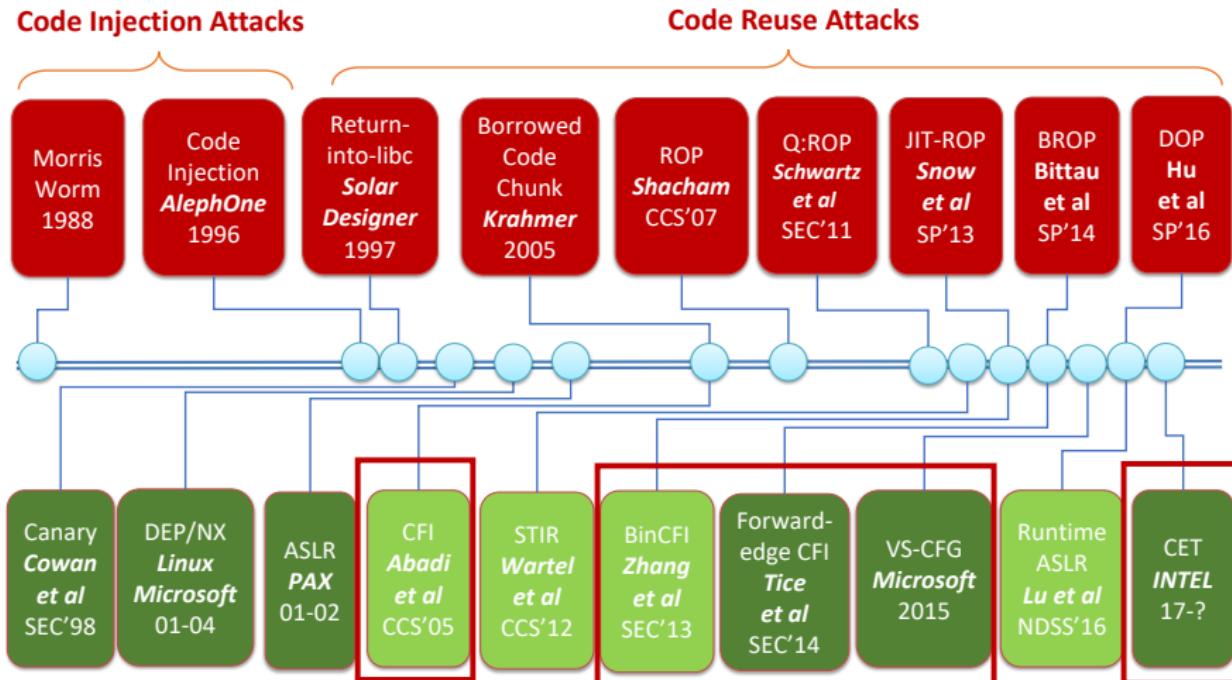
## From Coarse-Grained to More Fine-Grained ASLR



# Defense Using Control Flow Integrity



# Defense Using Control Flow Integrity



Introduction  
oooooooooo

Canary and DEP  
ooooo

Randomization  
oooooo

**Control Flow Integrity**  
○●○

Discussion  
oooo

Summary  
oo

# ASLR and CFI

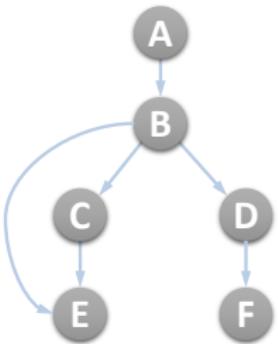
**(Fine-grained) Code  
Randomization**

**Control-Flow Integrity  
(CFI)**

# ASLR and CFI

## (Fine-grained) Code Randomization

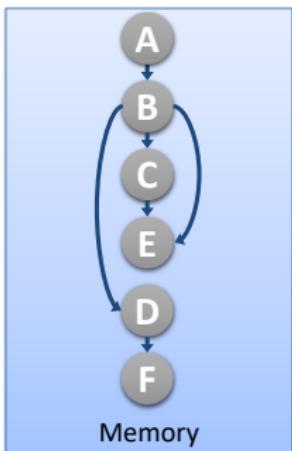
## Control-Flow Integrity (CFI)



# ASLR and CFI

## (Fine-grained) Code Randomization

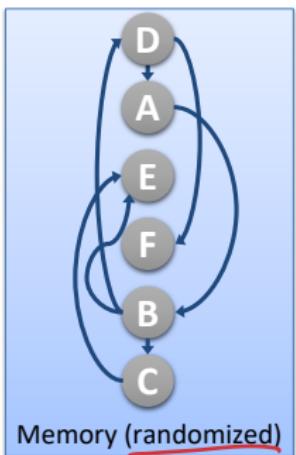
## Control-Flow Integrity (CFI)



# ASLR and CFI

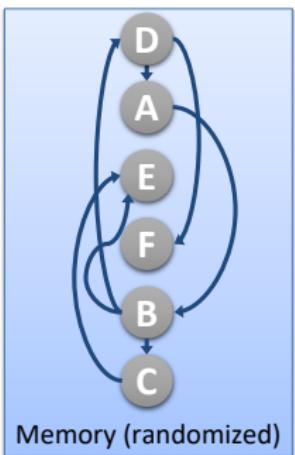
## (Fine-grained) Code Randomization

## Control-Flow Integrity (CFI)

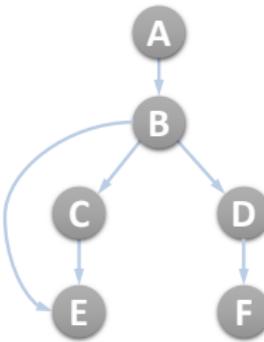


# ASLR and CFI

## (Fine-grained) Code Randomization

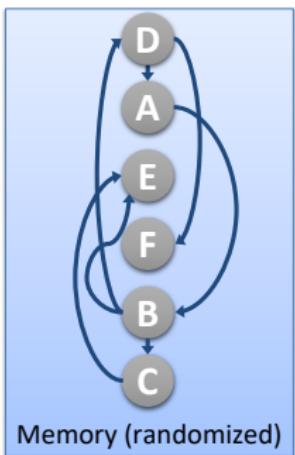


## Control-Flow Integrity (CFI)

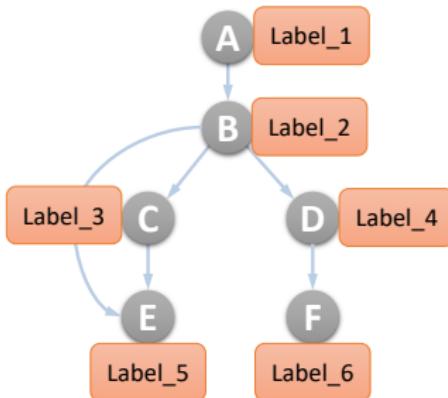


# ASLR and CFI

## (Fine-grained) Code Randomization

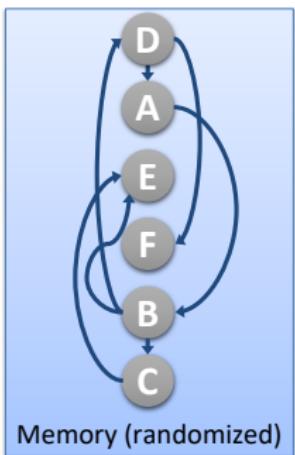


## Control-Flow Integrity (CFI)

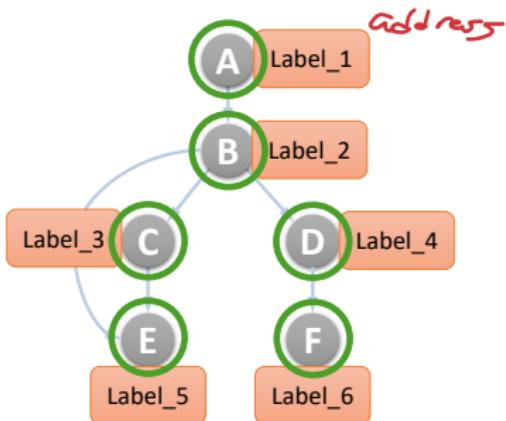


# ASLR and CFI

## (Fine-grained) Code Randomization

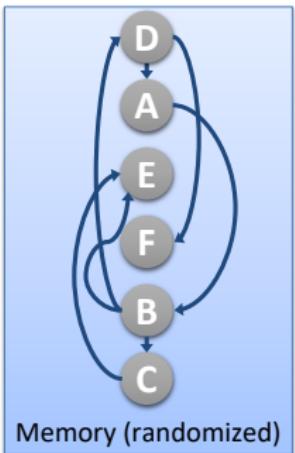


## Control-Flow Integrity (CFI)

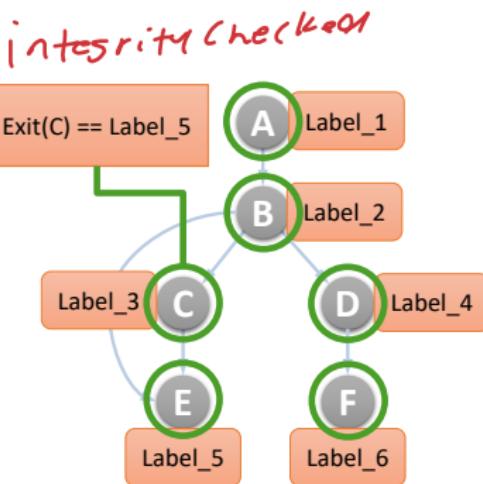


# ASLR and CFI

## (Fine-grained) Code Randomization



## Control-Flow Integrity (CFI)



# Recent Efforts on CFI Research

2002	Program Shepherding <i>Kiriansky et al. (USENIX Sec.)</i>									
2005	Control-Flow Integrity (CFI) <i>Abadi et al. (CCS 2005)</i>									
2006	XFI <i>Abadi et al. (OSDI)</i>		Architectural Support for CFI <i>Budiu et al. (ASID)</i>							
2010	EMET <i>Microsoft</i>		HyperSafe <i>Wang et al. (IEEE S&amp;P)</i>							
2011	CFI and Data Sandboxing <i>Zeng et al (CCS)</i>		Control-Flow Locking <i>Bletch et al. (ACSAC)</i>		ROPdefender <i>Davi et al. (AsiaCCS)</i>					
2012	Branch Regulation <i>Kayaalp et al (ISCA)</i>			Mobile CFI <i>Davi et al. (NDSS)</i>						
2013	Control-Flow Restrictor <i>Pewny et al (ACSAC)</i>		kBouncer <i>Pappas et al. (USENIX Sec.)</i>		bin-CFI <i>Zhang et al. (USENIX Sec.)</i>					
2014	ROPecker <i>Cheng et al. (NDSS)</i>		Forward-Edge CFI <i>Tice et al. (USENIX Sec.)</i>		SAFEDISPATCH <i>Jang et al. (NDSS)</i>					
2015	Control-Flow Guard <i>Microsoft</i>		Per-input CFI <i>Niu et al. (CCS)</i>		CCFI <i>Mashtizadeh et al. (CCS)</i>					
2016	Vtrust <i>Zhang et al. (NDSS)</i>		Protecting Vtables <i>Bounov et al. (NDSS)</i>		HAFIX++ <i>Sullivan et al. (DAC)</i>					
2017	PTCFI <i>Gu et al. (CODASPY)</i>		GRIFFIN <i>Ge et al. (ASPLOS)</i>		FlowGuard <i>Liu et al. (HPCA)</i>					
2018	CFIXX <i>Burow et al. (NDSS)</i>									
	CET <i>Intel</i>									

# The Practical Run-time Defenses Against Memory Corruptions

**Stack Canaries**

**Data Execution Prevention (DEP)**

**Code and Data Randomization**

**Control-Flow Integrity**

Introduction  
oooooooooo

Canary and DEP  
ooooo

Randomization  
oooooo

Control Flow Integrity  
ooo

Discussion  
o●ooo

Summary  
oo

# How to Implement the Defense Mechanisms (or Software in General)

# How to Implement the Defense Mechanisms (or Software in General)



Binary  
Instrumentation

## How to Implement the Defense Mechanisms (or Software in General)

Binary  
Instrumentation

Binary analysis is limited  
→ Defense will be incomplete

## How to Implement the Defense Mechanisms (or Software in General)

Binary  
Instrumentation

Binary analysis is limited  
→ Defense will be incomplete

Compiler  
Extensions

## How to Implement the Defense Mechanisms (or Software in General)

Binary  
Instrumentation

Binary analysis is limited  
→ Defense will be incomplete

Compiler  
Extensions

Requires recompilation of applications  
→ Legacy applications remain vulnerable

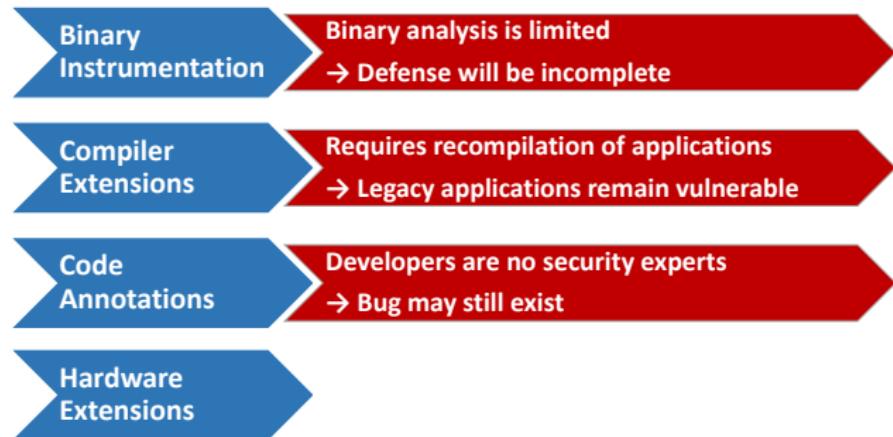
# How to Implement the Defense Mechanisms (or Software in General)



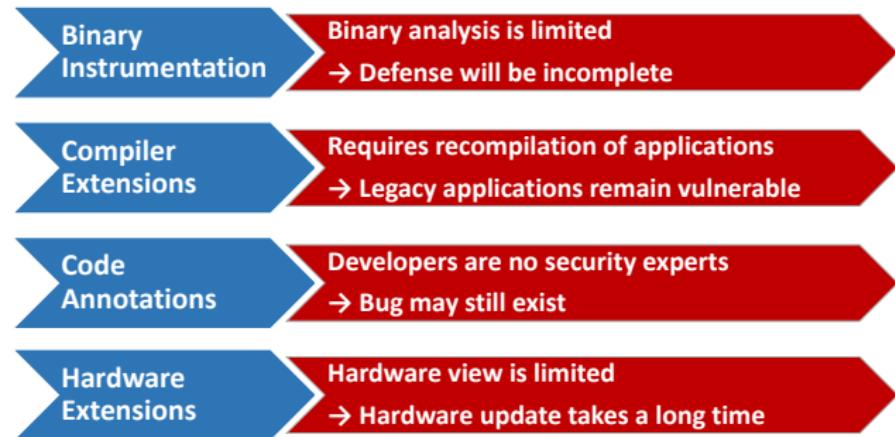
# How to Implement the Defense Mechanisms (or Software in General)



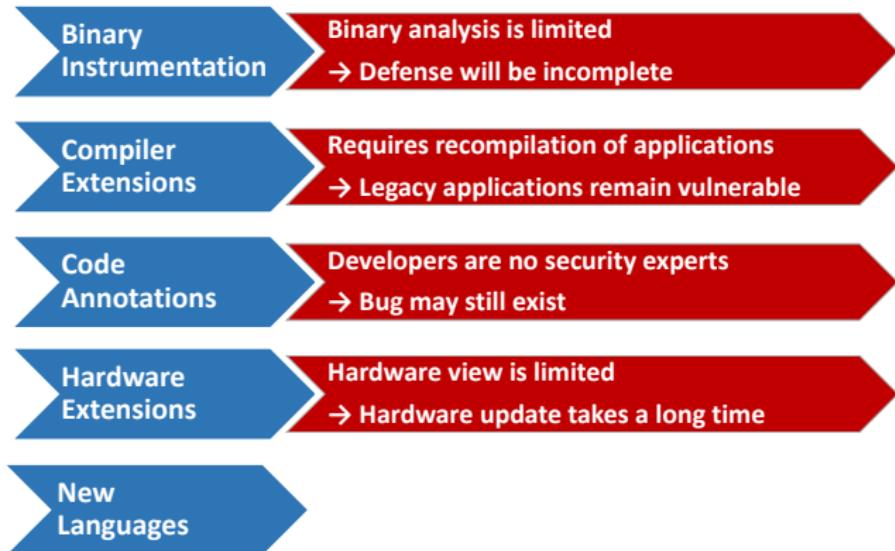
# How to Implement the Defense Mechanisms (or Software in General)



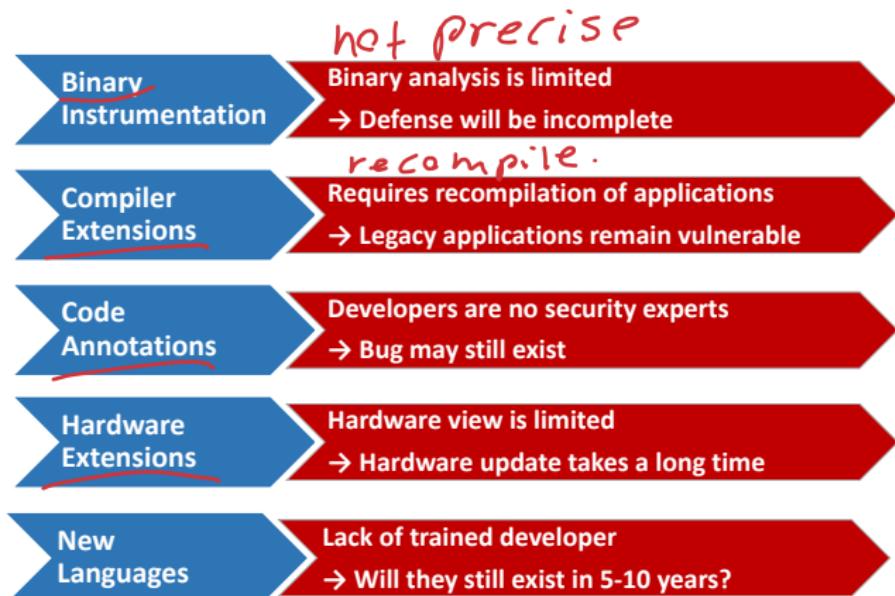
# How to Implement the Defense Mechanisms (or Software in General)



# How to Implement the Defense Mechanisms (or Software in General)



# How to Implement the Defense Mechanisms (or Software in General)



# Other Approaches: Proactive Vulnerability Identification

## Fuzzing

### ① Identifying vulnerabilities before attackers

- ▶ Blackbox (dumb) fuzzing
- ▶ Generational aka grammar-based fuzzing
- ▶ Whitebox fuzzing with SAGE
  - ▶ Looking at symbolic execution of the code
- ▶ Evolutionary fuzzing with afl
  - ▶ Grey-box, observing execution of the (instrumented) code

# Other Approaches: Proactive Vulnerability Identification

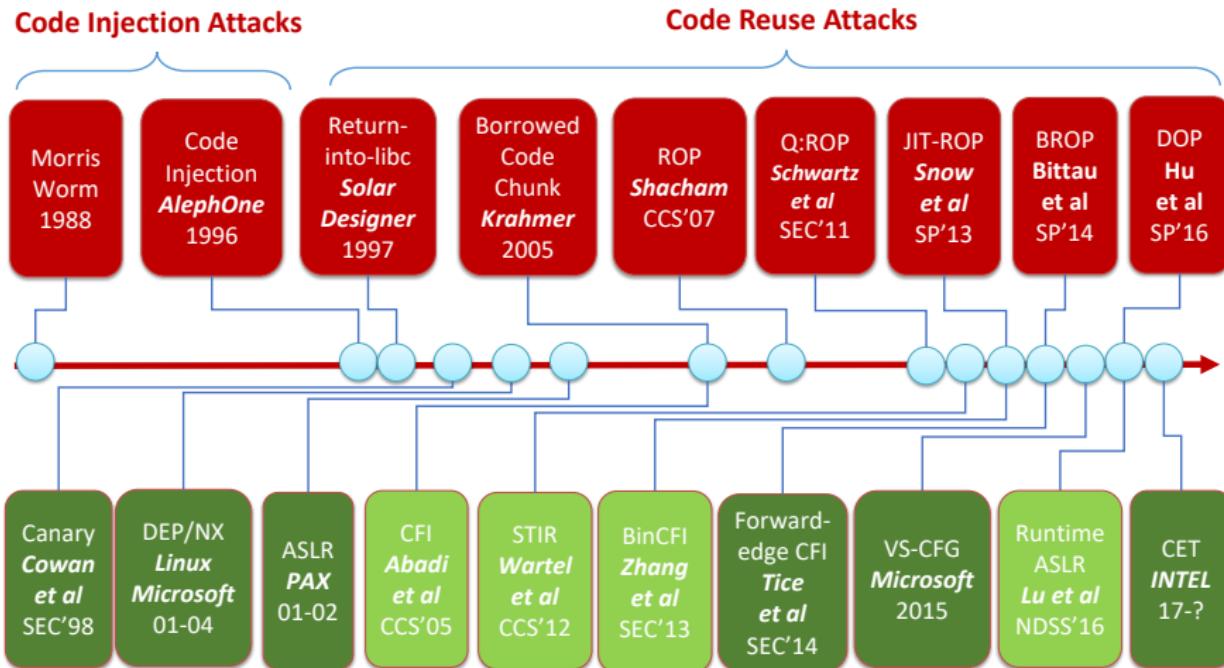
## Fuzzing

- ① Identifying vulnerabilities before attackers
  - ▶ Blackbox (dumb) fuzzing
  - ▶ Generational aka grammar-based fuzzing
  - ▶ Whitebox fuzzing with SAGE
    - ▶ Looking at symbolic execution of the code
  - ▶ Evolutionary fuzzing with afl
    - ▶ Grey-box, observing execution of the (instrumented) code

## AFL (American Fuzzy Lop)

- ① Support software w/
  - ▶ Source code (using a compiler flag)
    - ▶ C/C++/Object-C
    - ▶ Hand-written assembly
  - ▶ Binary (executed in an emulator)
- ② Identified many 0-day vulnerabilities
- ③ Highly practical

# The Arm Race Between Offense and Defense



# Practical Defenses

## When Developing New Software

- ① Turning on compiler flag
  - ▶ Canary (/gs, -fstack-protector)
  - ▶ CFI (/cfg, -fsanitize=cfi-icall)
  - ▶ CET (gcc 8.0)
  - ▶ ASLR (-pie -fPIE)
- ② Using type safe language
  - ▶ Rust

# Practical Defenses

## When Developing New Software

- ① Turnning on compiler flag
  - ▶ Canary (/gs, -fstack-protector)
  - ▶ CFI (/cfg, -fsanitize=cfi-icall)
  - ▶ CET (gcc 8.0)
  - ▶ ASLR (-pie -fPIE)
- ② Using type safe language
  - ▶ Rust

## When Deploying Old Software

- ① Turnning on kernel DEP
- ② Turnning on ASLR
  - ▶ Load-time
  - ▶ Microsoft EMET
- ③ Keeping software patched
- ④ Penetration testing
  - ▶ Using fuzzing

# Software Security

## Software is Not Secure

- ① Memory unsafe code (written in C/C++, asm, ...)
- ② Prevalent software defects
  - ▶ Stack/Heap smashing
  - ▶ Format string bugs
  - ▶ Pointer errors

## Defenses

- ① Stack/Heap Canaries
- ② ASLR, DEP, CFI
- ③ RELRO, BIND\_NOW
- ④ BPF\_SECCOMP,  
FORTIFY\_SRC

# Software Security

## Software is Not Secure

- ❶ Memory unsafe code (written in C/C++, asm, ...)
- ❷ Prevalent software defects
  - ▶ Stack/Heap smashing
  - ▶ Format string bugs
  - ▶ Pointer errors

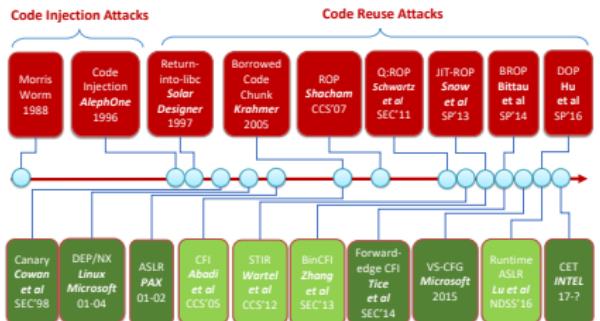
## Defenses

- ❶ Stack/Heap Canaries
- ❷ ASLR, DEP, CFI
- ❸ RELRO, BIND\_NOW
- ❹ BPF\_SECCOMP,  
FORTIFY\_SRC

## Offenses

- ❶ Code injection
- ❷ Code Reuse
  - ▶ Return-to-libc (ret2libc)
  - ▶ Return-oriented prog.  
(ROP)
  - ▶ Just-In-Time ROP  
(JIT-ROP)

# Thank You



1

# Q&A

schen@auburn.edu  
schuan.github.io

<sup>1</sup>Instructor appreciates the help from Prof. Zhiqiang Lin.