Hardware Security

ECEn 522R

Memory Safety and Corruption Attacks

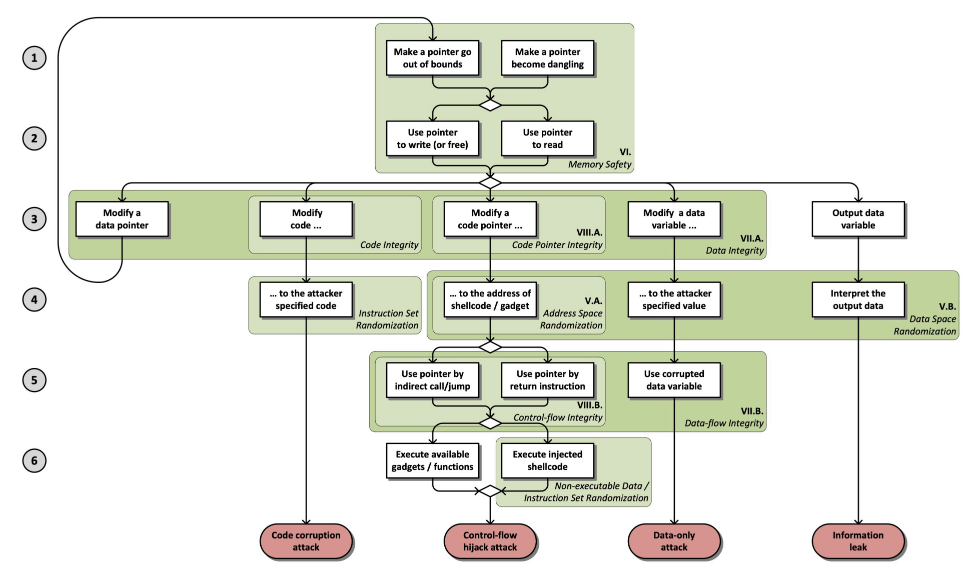


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

Stacey D. Son

18-Nov-2025

A brief history of buffer overflows

1972 — First public documentation in Computer Security Technology Planning Study (Some prior systems such as Multics recognized and solved the issue)

1988 — One of several exploits employed by the Morris worm

1993 — FreeBSD's first release

1996 — Smashing the Stack for Fun and Profit by Aleph One (Elias Levy)

1998 — Stack Canary Protection

2001 — Heap-based buffer overflows - Solar Desiigner (A. Peslyak)

2001 — DEP/W⊕X Protection

2001 — Return-to-Libc by Solar Designer (Alexander Peslyak)

2001 — Heap-based Buffer Overflows

2002 — Address-Space Randomization

2003 — PointGuard - Pointer Protection by C. Cowan

2007 — Return-Oriented Programming

2011 — Jump-Oriented Programming

2013 — "SoK: Eternal War in Memory" Attack Model

2018 — ARM Pointer Auth (PAC) and BTI instruction in iPhone

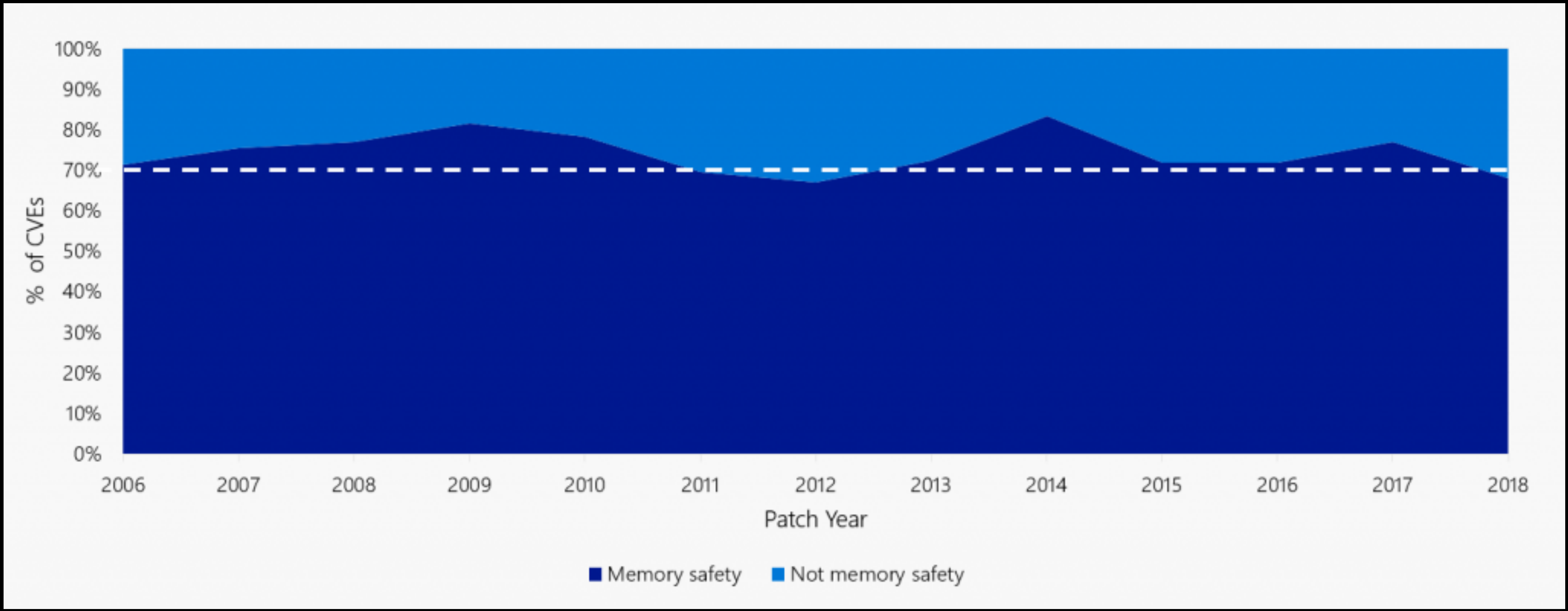
2021 — ARM Morello (CHERI) Hardware Available

2023 — ARM Memory Tag Ext. (MTE) in Google Pixel 8

Today

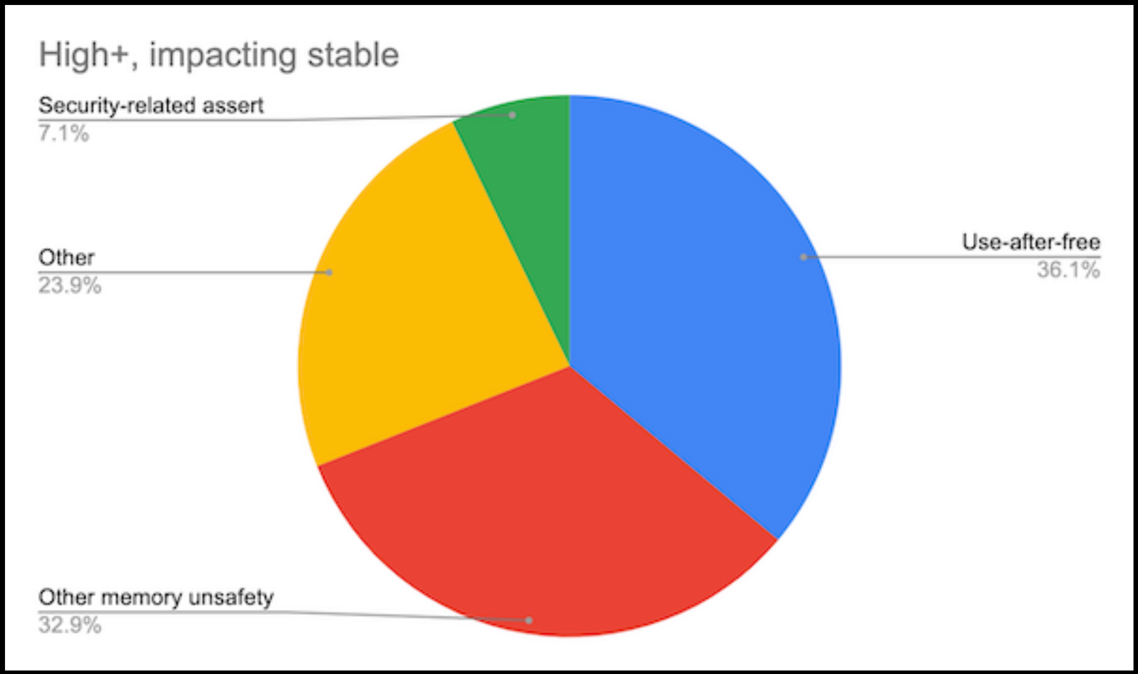
Complete Memory Safety: We are almost there!
(But we need more adoption.)

Microsoft Report



https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

Google Chrome Report



https://www.chromium.org/Home/chromium-security/memory-safety/

Memory Safety Attack Model

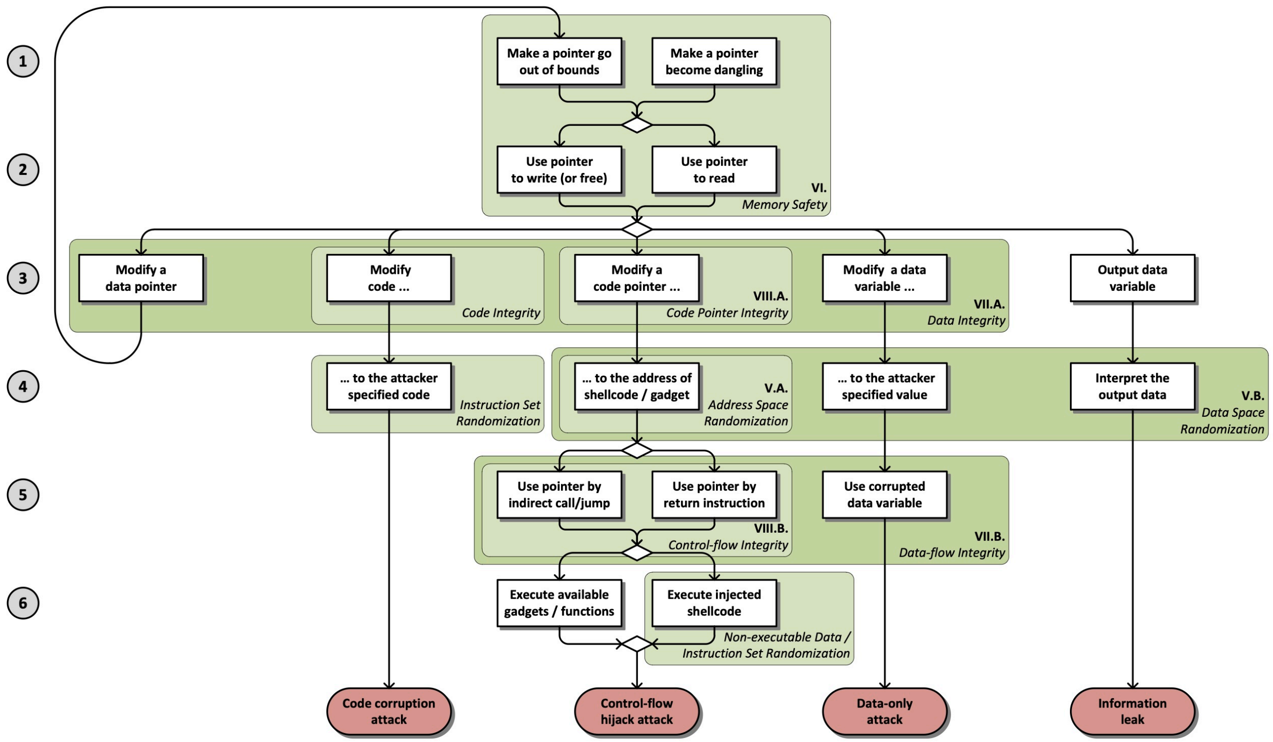


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

"SoK: Eternal War in Memory" https://people.eecs.berkeley.edu/~dawnsong/

Data Pointers: Pointers for modifying other memory

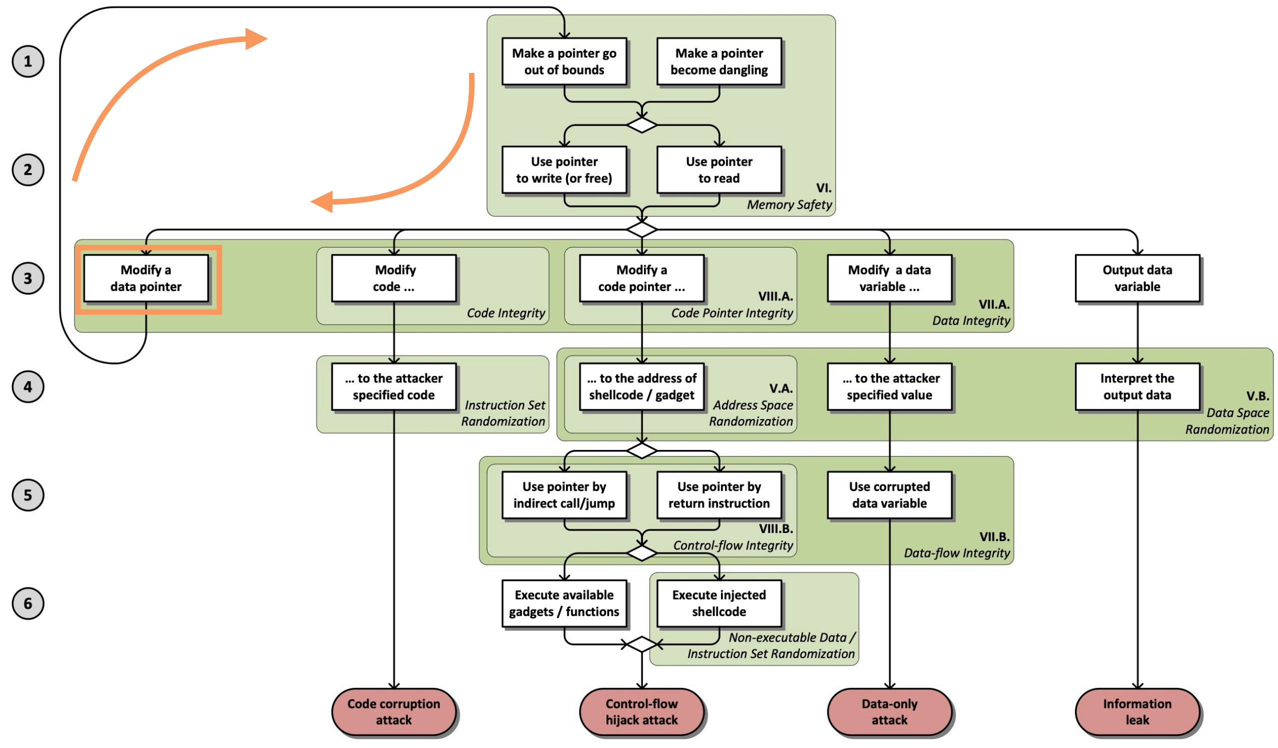


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

"SoK: Eternal War in Memory" https://people.eecs.berkeley.edu/~dawnsong/

Data Variable Modification

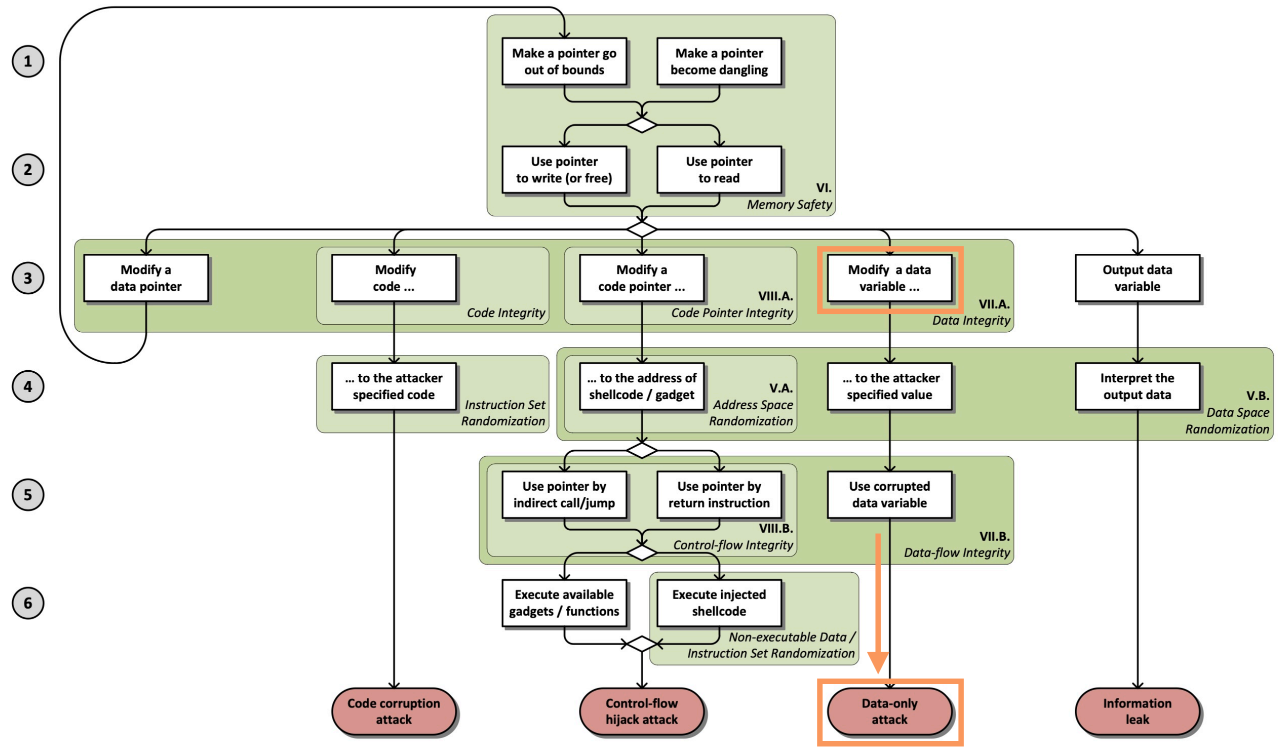


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

"SoK: Eternal War in Memory" https://people.eecs.berkeley.edu/~dawnsong/

Information Leak

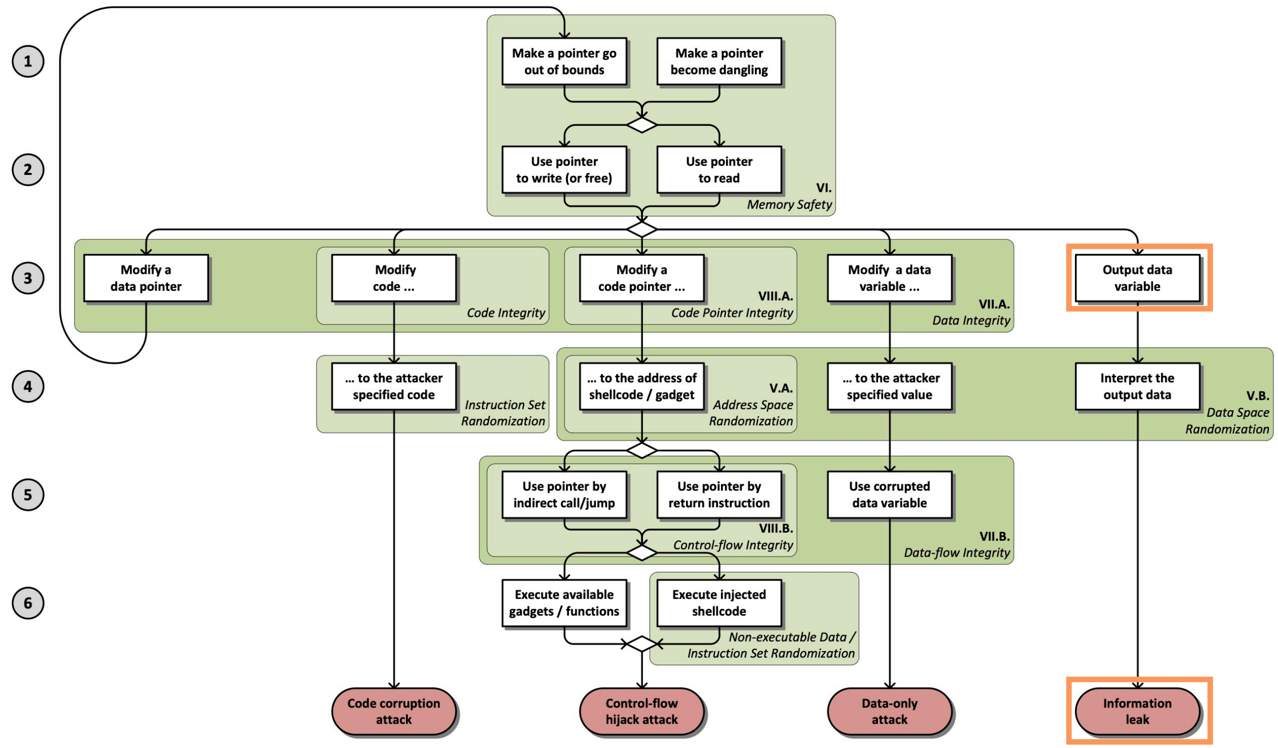
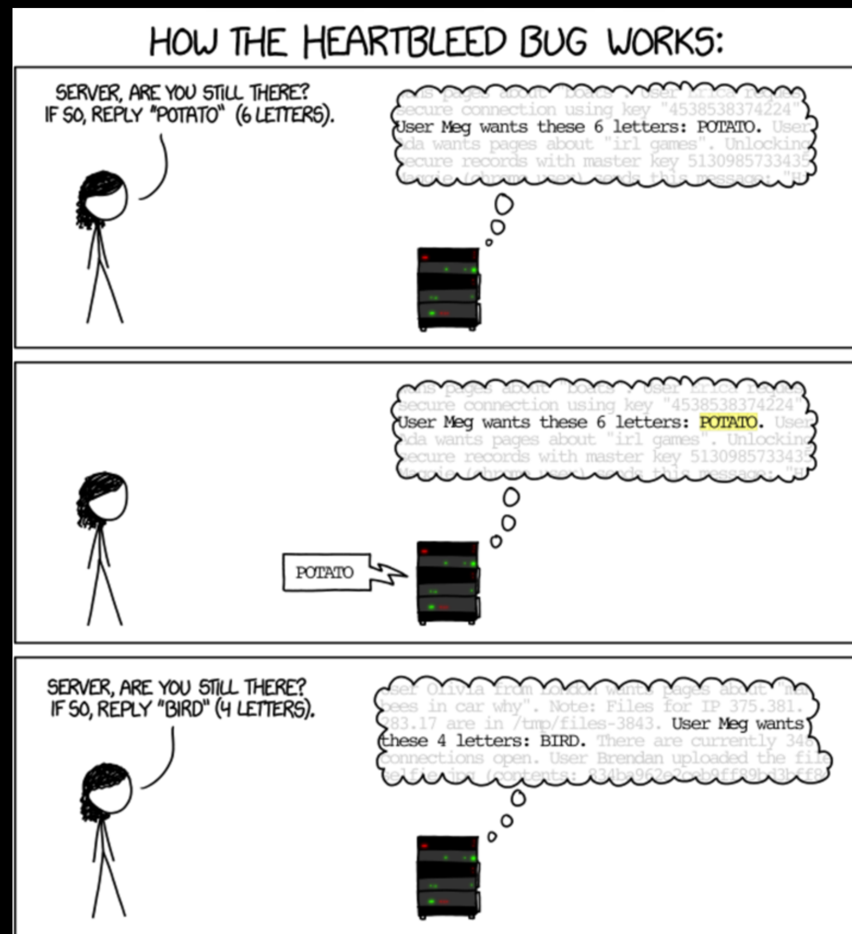


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

"SoK: Eternal War in Memory" https://people.eecs.berkeley.edu/~dawnsong/

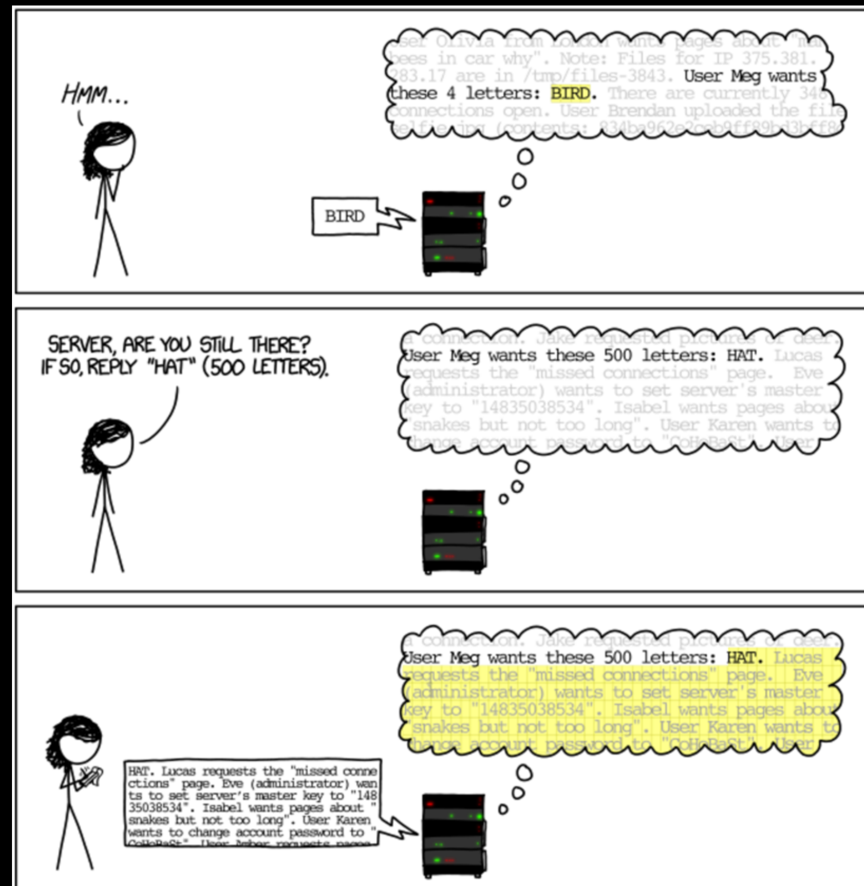
Information Leak “in the wild” 1/3

HeartBleed
(2014)

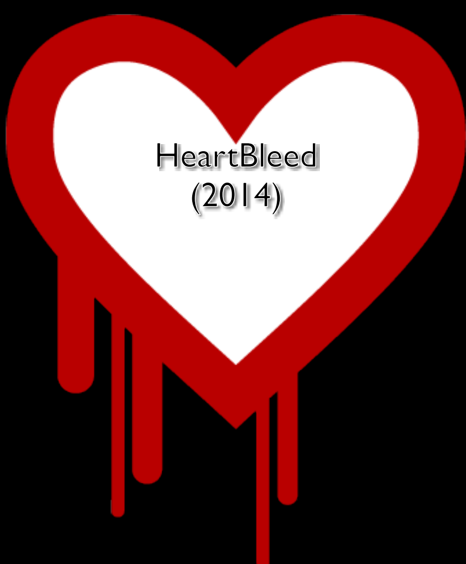


Information Leak “in the wild” 2/3

HeartBleed
(2014)



Information Leak “in the wild” 3/3



Code Modification or Injection

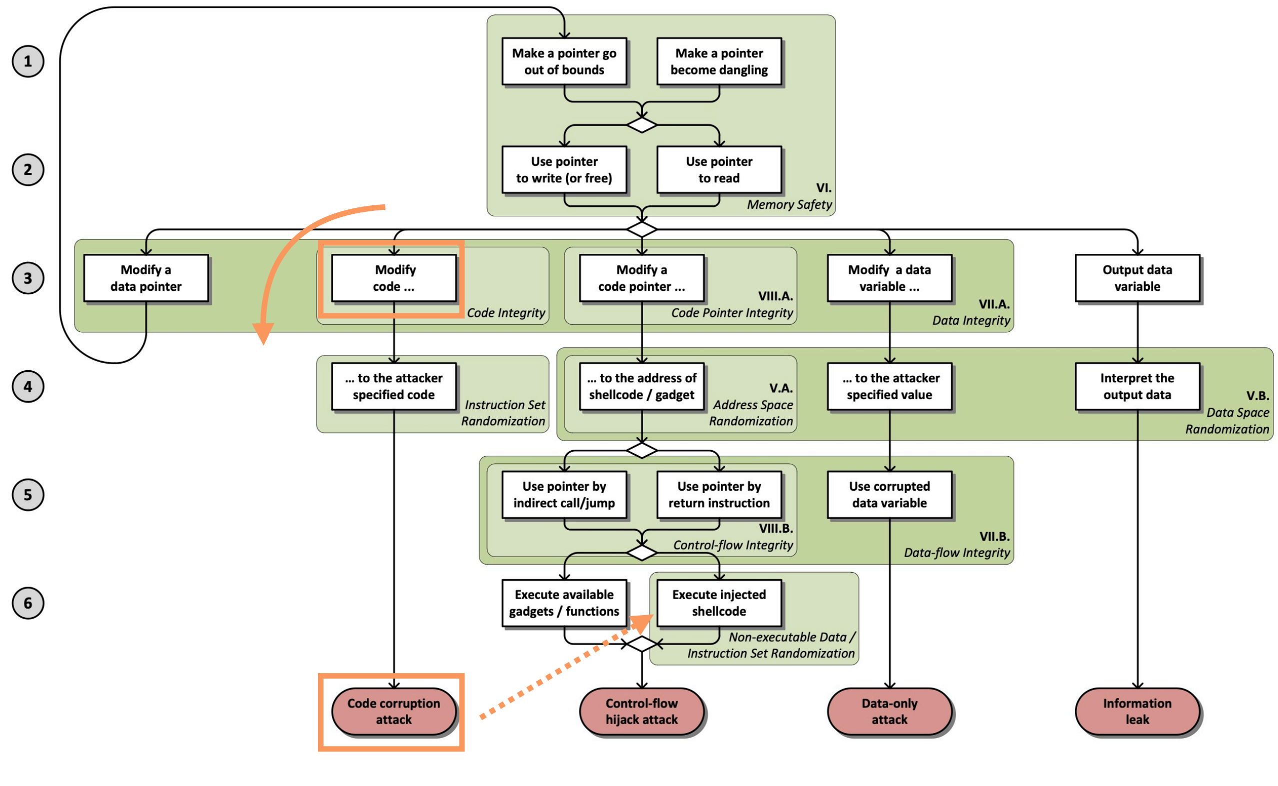


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

"SoK: Eternal War in Memory" https://people.eecs.berkeley.edu/~dawnsong/

Code Corruption Prevention with Hardware

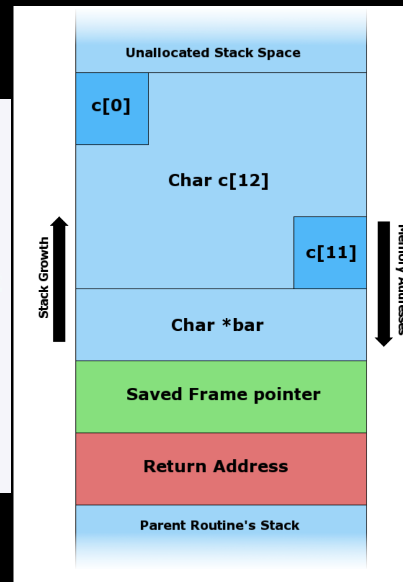
- Change the code:
  - Before it is loaded (prevented by code signing)
  - After it is loaded into memory (prevented by W⊕X / DEP? Implemented with AMD "NX", Intel "XD", ARM "XN", etc. hardware bits)
- Where W⊕X/DEP may not work:
  - Just-In-Time Compiling (e.g., Java Runtime and some other dynamic coding)
    - Modify the code in the JIT code buffer before it is executed
  - Code reuse attacks (will discuss later)

Code corruption attacks

- Self modifying code requires Write and Execute permissions (W⊕X)
- Just-In-Time (JIT) compilers (e.g, Java Script)
  - Modern implementations will write-protect the memory after the code is generated. Apple even added some HW support for this.
- The code may be corrupted before it is even loaded and used.
  - Protected with code signing and embedded keys in products.
  - e.g, SolarWinds Supply Chain Attack (2020)
    - Russian hackers compromised and gained access to SolarWinds' production environment and introduced malicious code into a network monitoring product.
- Code Injection

Simple Stack Overflow Example

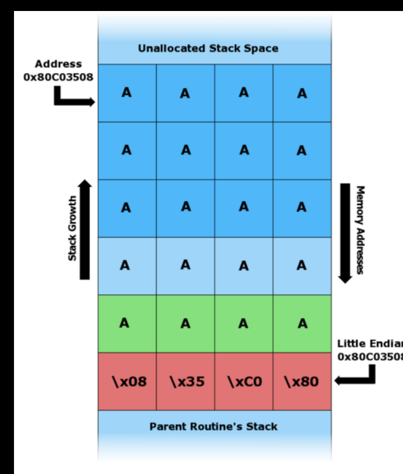
```c
#include <string.h>

void foo(char* bar) {
    char c[12];

    strcpy(c, bar); // no bounds checking
}

int main(int argc, char* argv[]) {
    foo(argv[1]);
    return 0;
}
```



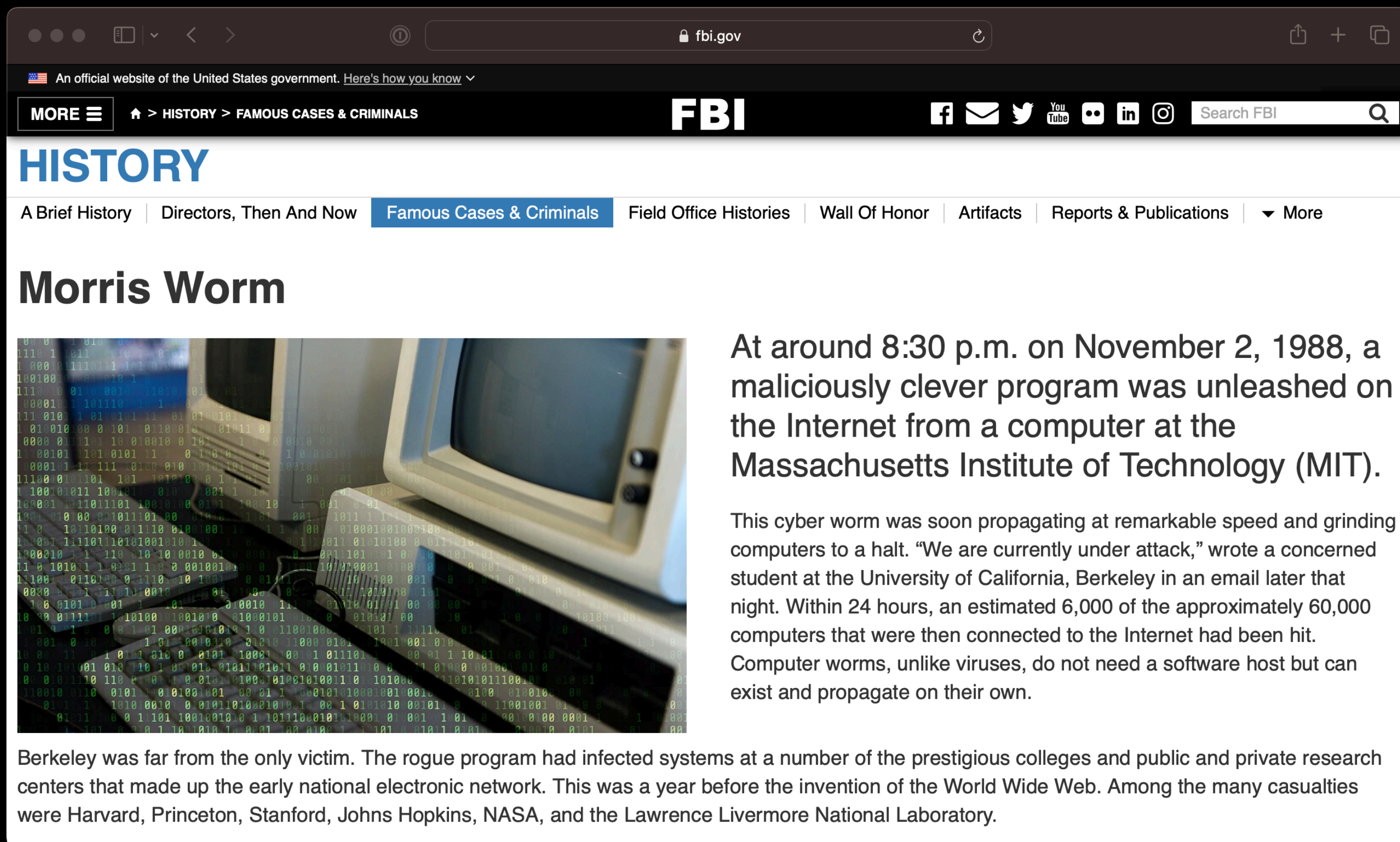
- Read some data from the command-line, Store it in a stack buffer. e.g., "AAAAAAAAAAAAAAAAAAAA\x08\x35\xC0\x80"

- The stack buffer is only 12 characters and strcpy() does not do bounds checking.

- Overwrites return address on stack to return to injected code (assumes executable stack)



“In the wild” Stack Overflow Example: Morris Worm



fbi.gov

An official website of the United States government. Here's how you know

MORE ☰ > HISTORY > FAMOUS CASES & CRIMINALS

FBI

Search FBI

HISTORY

A Brief History | Directors, Then And Now | Famous Cases & Criminals | Field Office Histories | Wall Of Honor | Artifacts | Reports & Publications | More

Morris Worm

At around 8:30 p.m. on November 2, 1988, a maliciously clever program was unleashed on the Internet from a computer at the Massachusetts Institute of Technology (MIT).

This cyber worm was soon propagating at remarkable speed and grinding computers to a halt. “We are currently under attack,” wrote a concerned student at the University of California, Berkeley in an email later that night. Within 24 hours, an estimated 6,000 of the approximately 60,000 computers that were then connected to the Internet had been hit. Computer worms, unlike viruses, do not need a software host but can exist and propagate on their own.

Berkeley was far from the only victim. The rogue program had infected systems at a number of the prestigious colleges and public and private research centers that made up the early national electronic network. This was a year before the invention of the World Wide Web. Among the many casualties were Harvard, Princeton, Stanford, Johns Hopkins, NASA, and the Lawrence Livermore National Laboratory.

- ‘fingerd’ service had a stack overflow vulnerability

Code Pointer Mod: Control-flow Hijacking

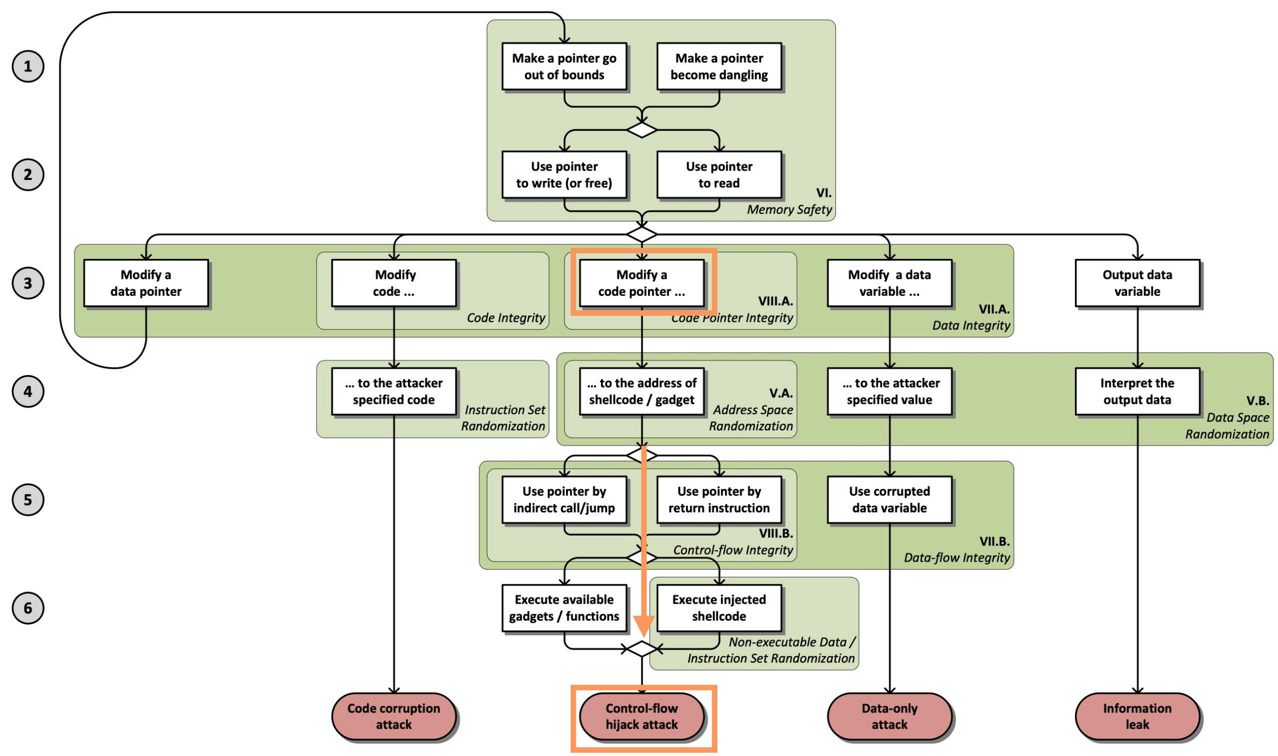


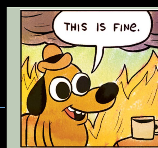
Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

"SoK: Eternal War in Memory" https://people.eecs.berkeley.edu/~dawnsong/

Buffer Overflow: Causes and Cures

- Typical memory exploit involves code injection
  - Put malicious code in a predictable location in memory, usually masquerading as data
  - Trick vulnerable program into passing control to it
    - e.g., Overwrite saved IP, function callback pointer, etc.
- Defense: prevent execution of untrusted code
  - Make stack and other data areas non-executable
  - Digital sign all code
  - Ensure that all control transfers are into a trusted, approved code image

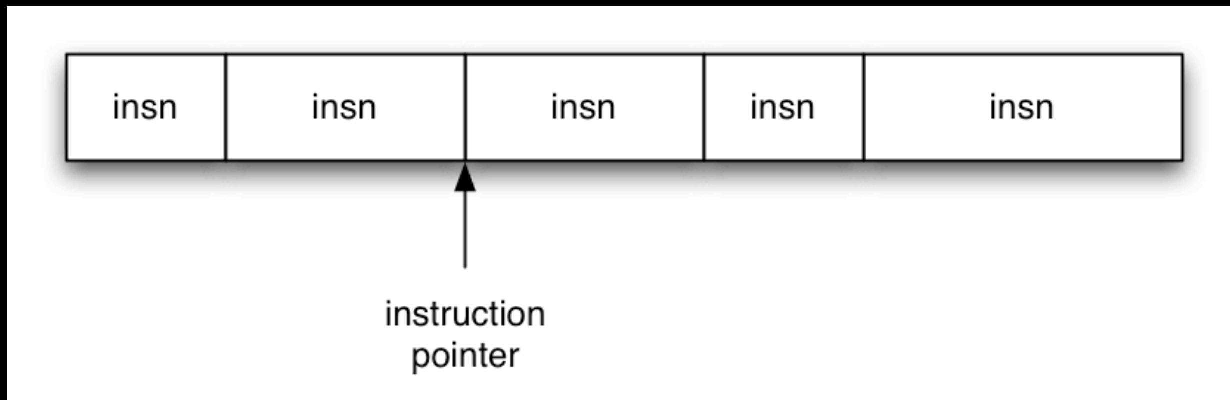
Software work-arounds were doing pretty well then ROP/JOP was introduced…

- 1972 — First public documentation in Computer Security Technology Planning Study (Some prior systems such as Multics recognized and solved the issue)
- 1988 — One of several exploits employed by the Morris worm
- 1993 — FreeBSD's first release
- 1996 — Smashing the Stack for Fun and Profit by Aleph One (Elias Levy)
- 1998 — Stack Canary Protection
- 2001 — Heap-based buffer overflows - Solar Desiigner (A. Peslyak)
- 2001 — DEP/W⊕X Protection
- 2001 — Return-to-Libc by Solar Designer (Alexander Peslyak)
- 2001 — Heap-based Buffer Overflows
- 2002 — Address-Space Randomization
- 2003 — PointGuard - Pointer Protection by C. Cowan
- 2007 — Return-Oriented Programming
- 2011 — Jump-Oriented Programming
- 2013 — "SoK: Eternal War in Memory" Attack Model
- 2018 — ARM Pointer Auth (PAC) and BTI instruction in iPhone
- 2021 — ARM Morello (CHERI) Hardware Available
- 2023 — ARM Memory Tag Ext. (MTE) in Google Pixel 8
- 

Today

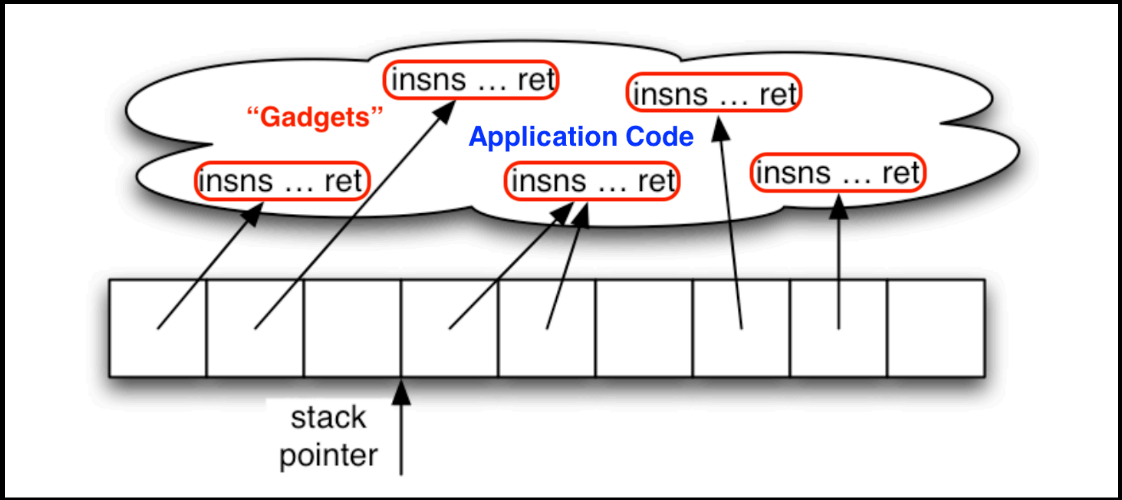
"We're going to need some bigger hardware"

The Ordinary Way of Programming



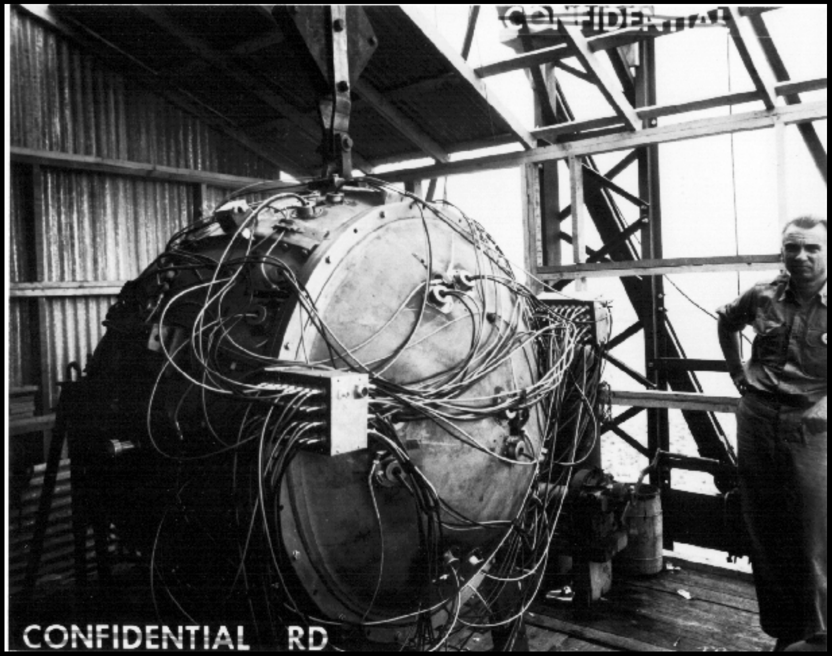
- Instruction pointer (IP) determines which instruction to fetch and execute
- Once processor has executed the instruction, it automatically increments IP to next instruction
- Control flow by changing value of IP: Jump, Call, Return, etc.

Return-Oriented Programming (ROP)



- Stack pointer (SP) determines which instruction sequence to fetch and execute
- Processor doesn't automatically increment the SP
  - But the RET at end of each instruction sequence does

Gadgets: Code Sequences Found in the Application or Libraries Code That End With a ‘RET’

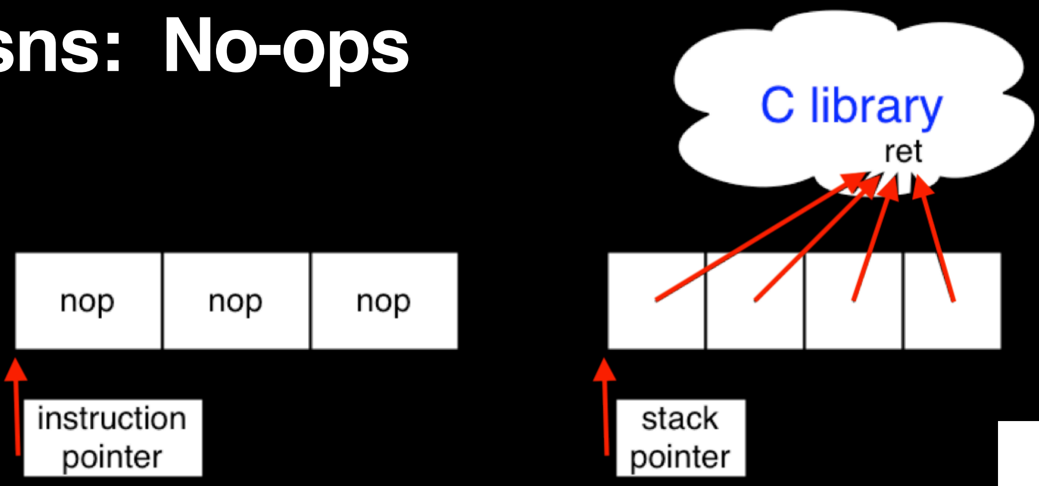


*

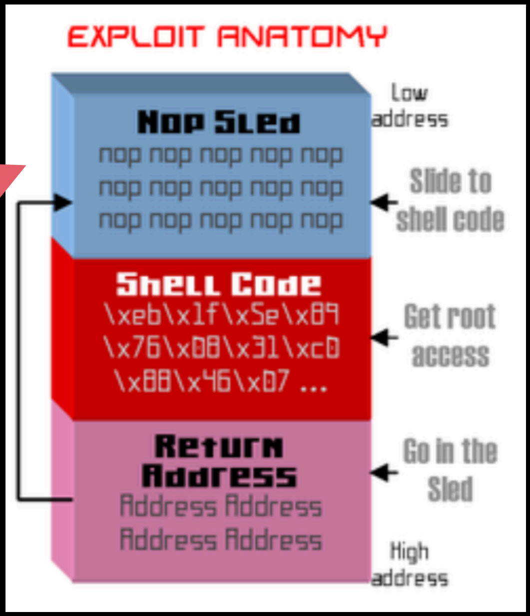
* Not to be confused with “THE Gadget”

Some examples of Gadgets….

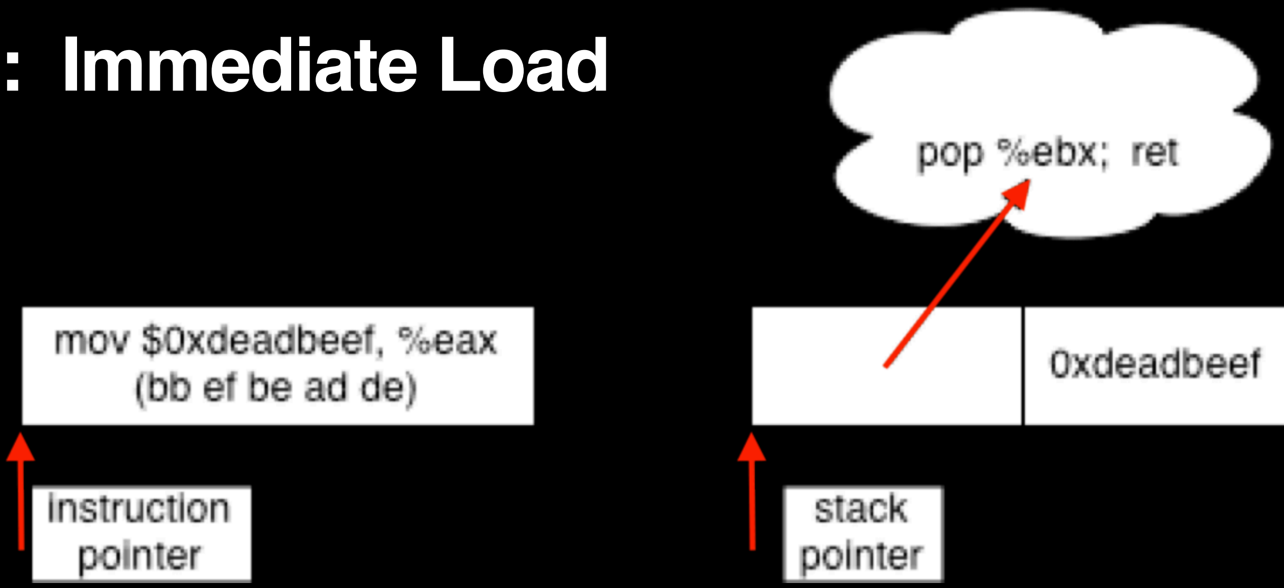
Gadget Insns: No-ops



- No-op instruction does nothing but advance the IP
- Return-oriented equivalent
  - Point to return instruction
  - Advances SP
- Useful in a “NOP sled”

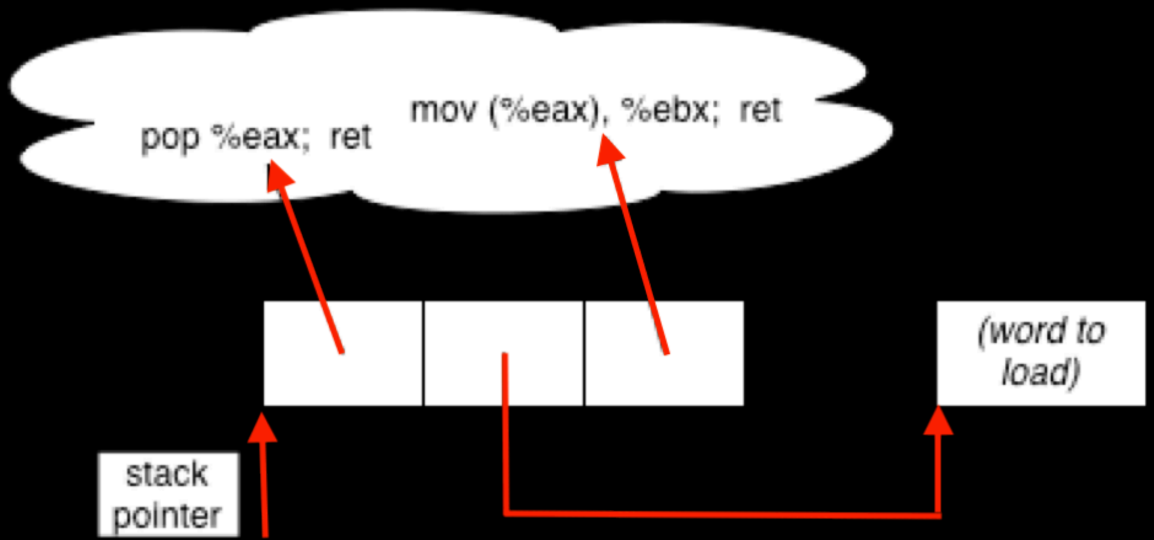


Gadget Insns: Immediate Load



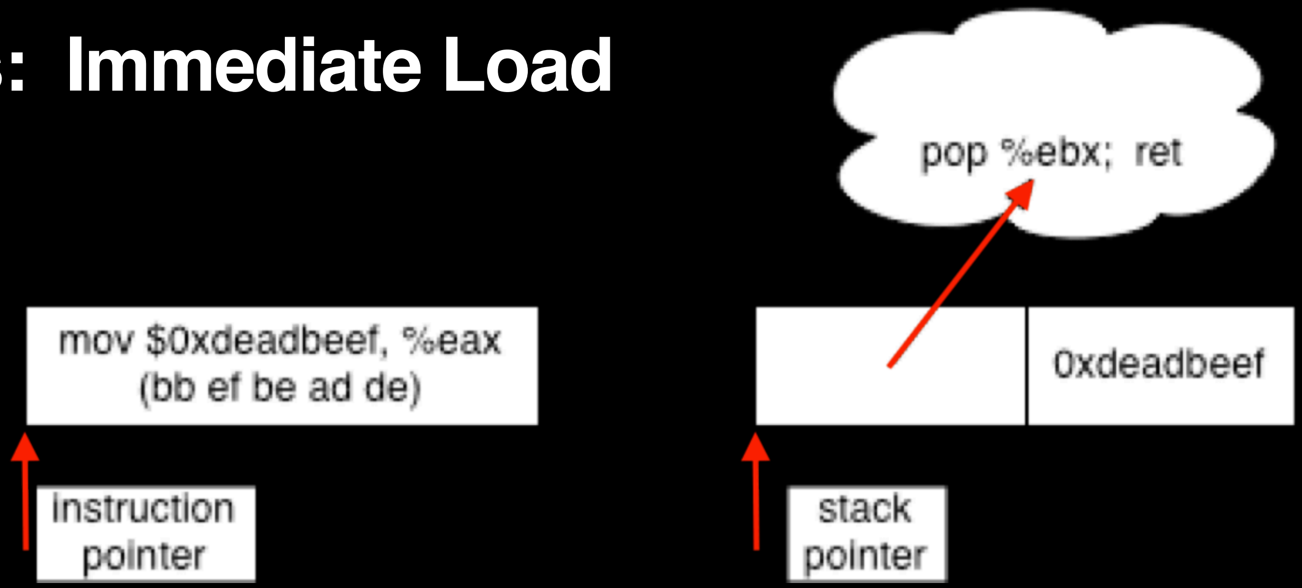
- Instructions can encode constants
- Return-oriented equivalent
  - Store on the stack
  - Pop into register to use

Gadget Insns: Multi-insn Sequences



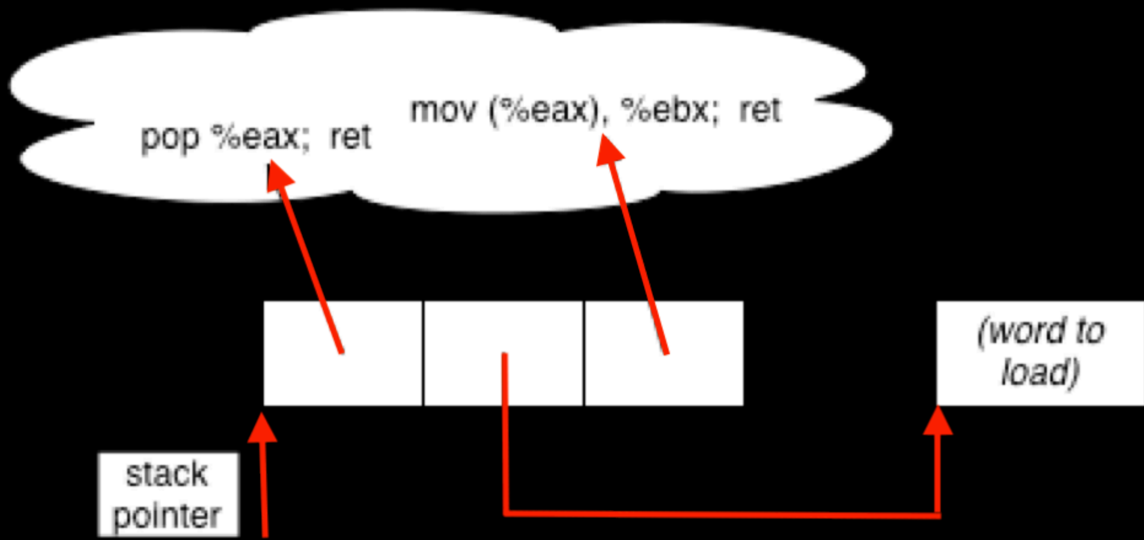
- Sometimes more than one instruction sequence needed to encode logical unit
- Example: load from memory into register
  - Load address of source word into EAX
  - Load memory at (EAX) into EBX

Gadget Insns: Immediate Load



- Instructions can encode constants
- Return-oriented equivalent
  - Store on the stack
  - Pop into register to use

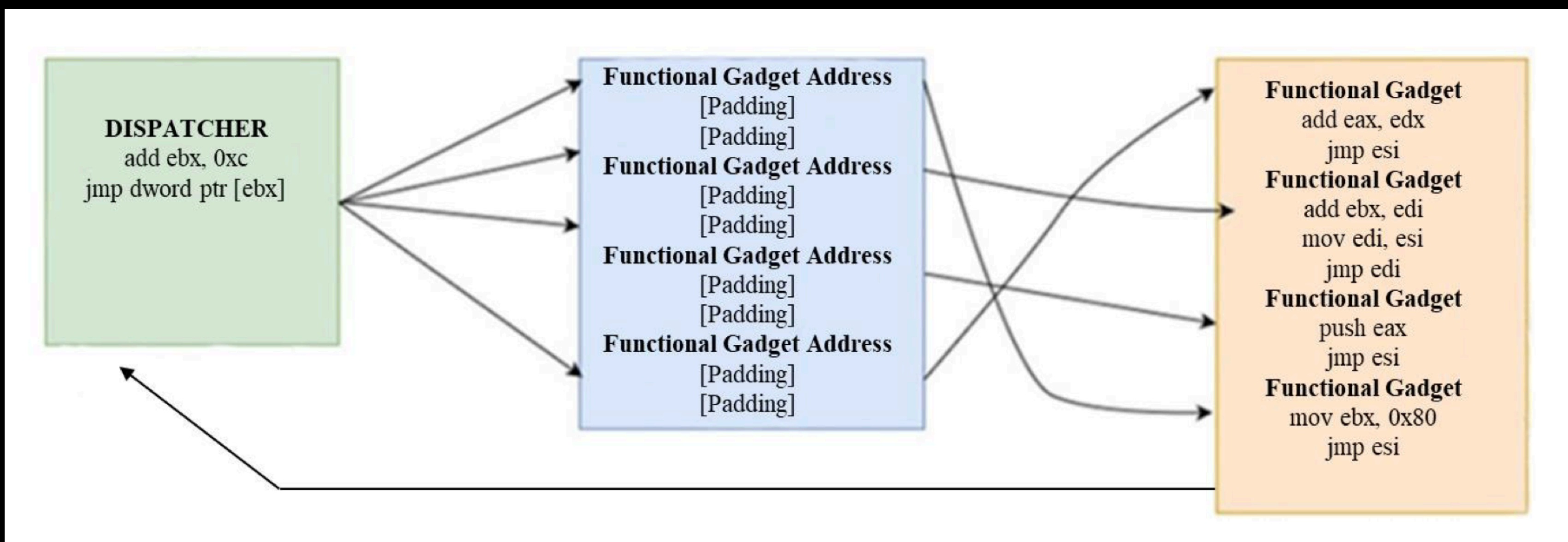
Gadget Insns: Multi-insn Sequences



- Sometimes more than one instruction sequence needed to encode logical unit
- Example: load from memory into register
  - Load address of source word into EAX
  - Load memory at (EAX) into EBX

Jump-Oriented Programming (JOP)

- JOP Gadgets end with BLR/BR (ARM) or JMP (Intel) instructions instead of RET
- Consists of A Dispatcher Gadget, Functional Gadget Table, and Functional Gadgets:



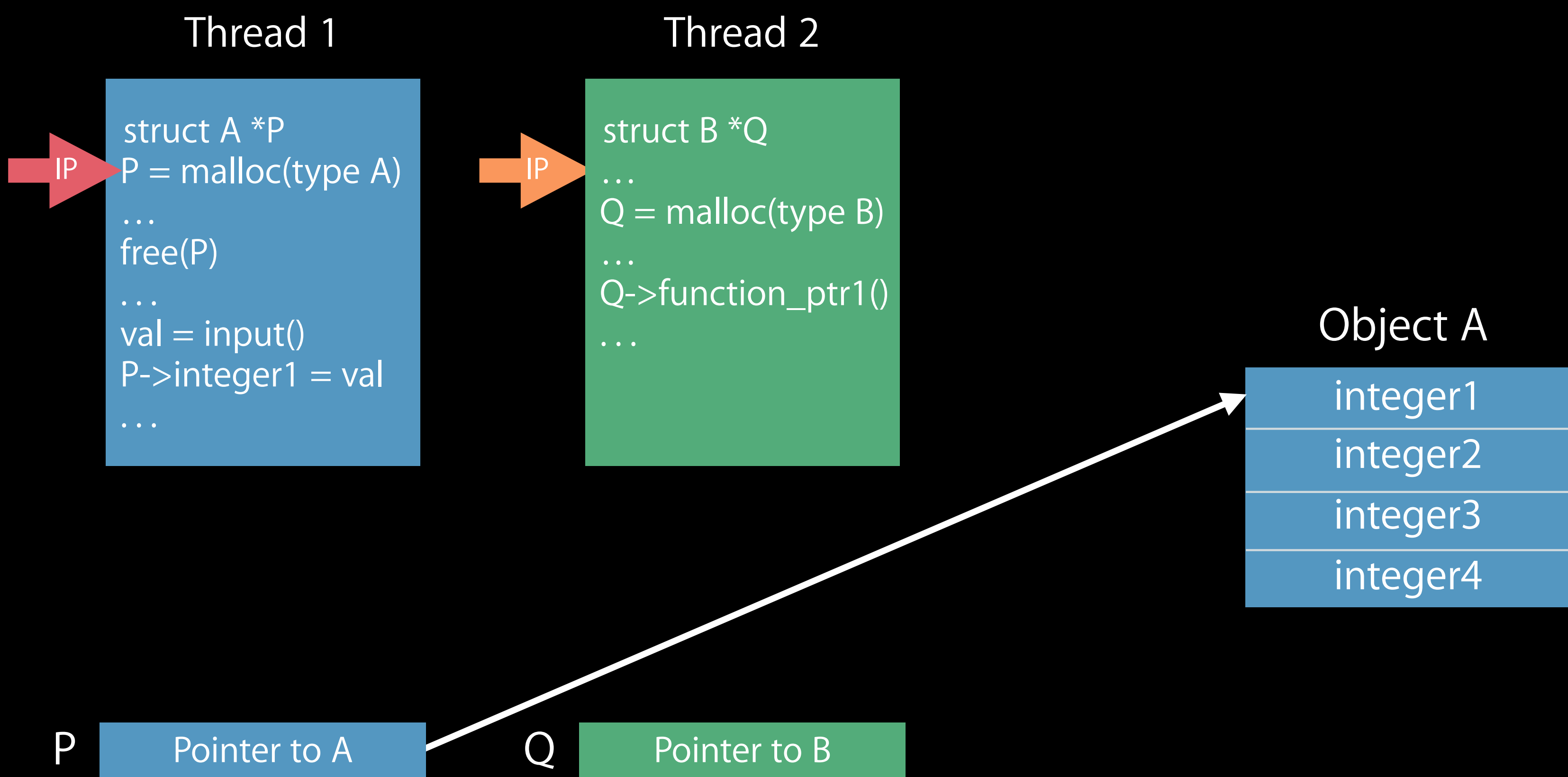
Migrations Against ROP and JOP

- Shadow Stacks (e.g., Intel Control-flow Enforcement Technology or CET)
  - Shadow stack stores a copy of the return address of each CALL in secure stack
  - On a RET, the processor checks if the return address stored in the normal stack and shadow stack are equal
- Branch Target Insructions e.g., ARM BTIs (For JOP Mitigation)
  - Indirect branches (BR and BLR) can only land on BTI instructions.
- Pointer Integrity
  - PointGuard (Software): Function pointers are XOR'ed with random value.
  - Pointer Signing/Authentication
    - CCFI: Cyptographically Enforced Control Flow Integrity
    - ARM Pointer Authentication Code (PAC) or CHERI for pointer integrity…

Use-After-Free (UAF) Bug Attack

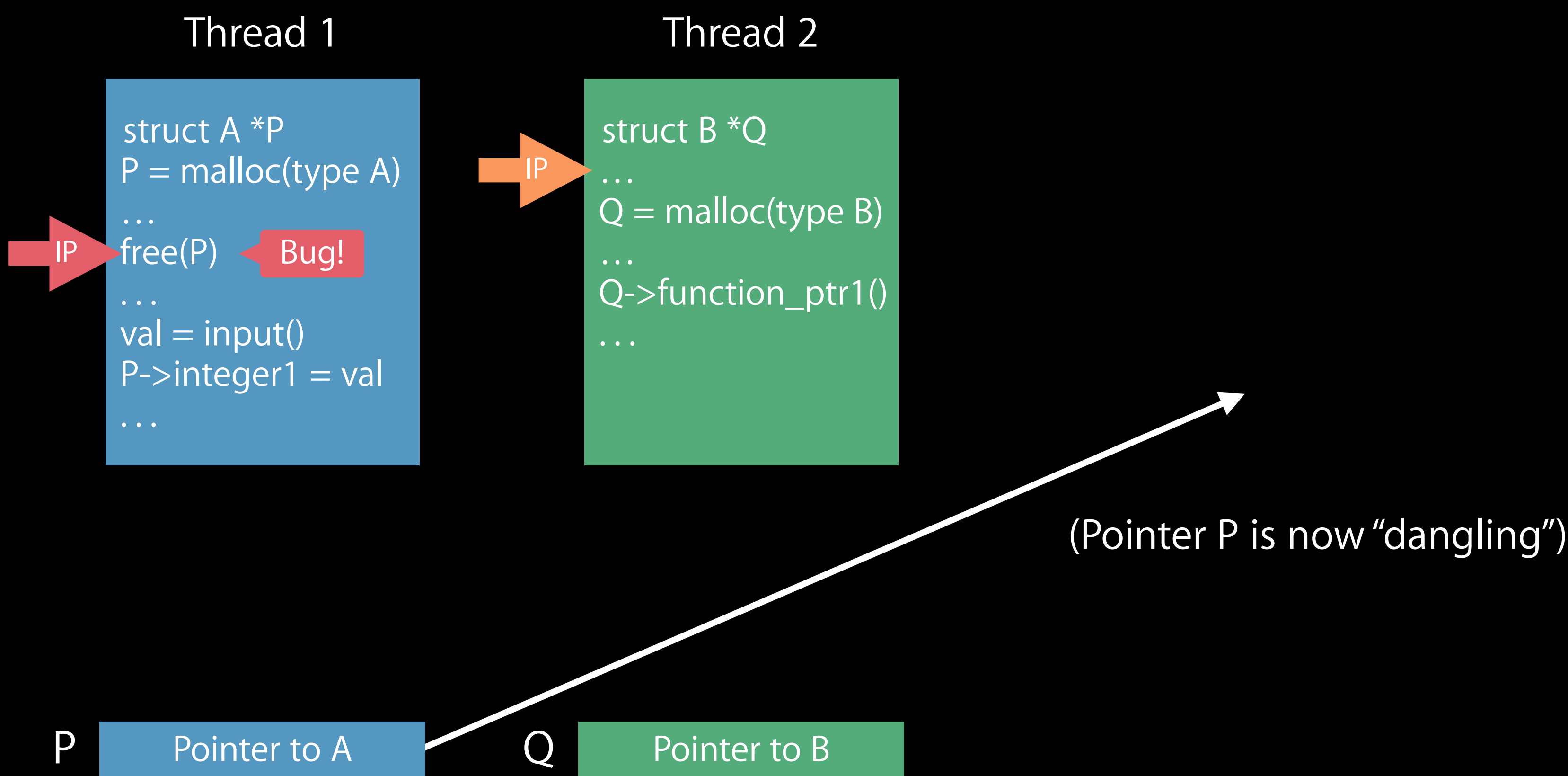
Use-After-Free Example

Thread 1 allocates a type A object in heap memory



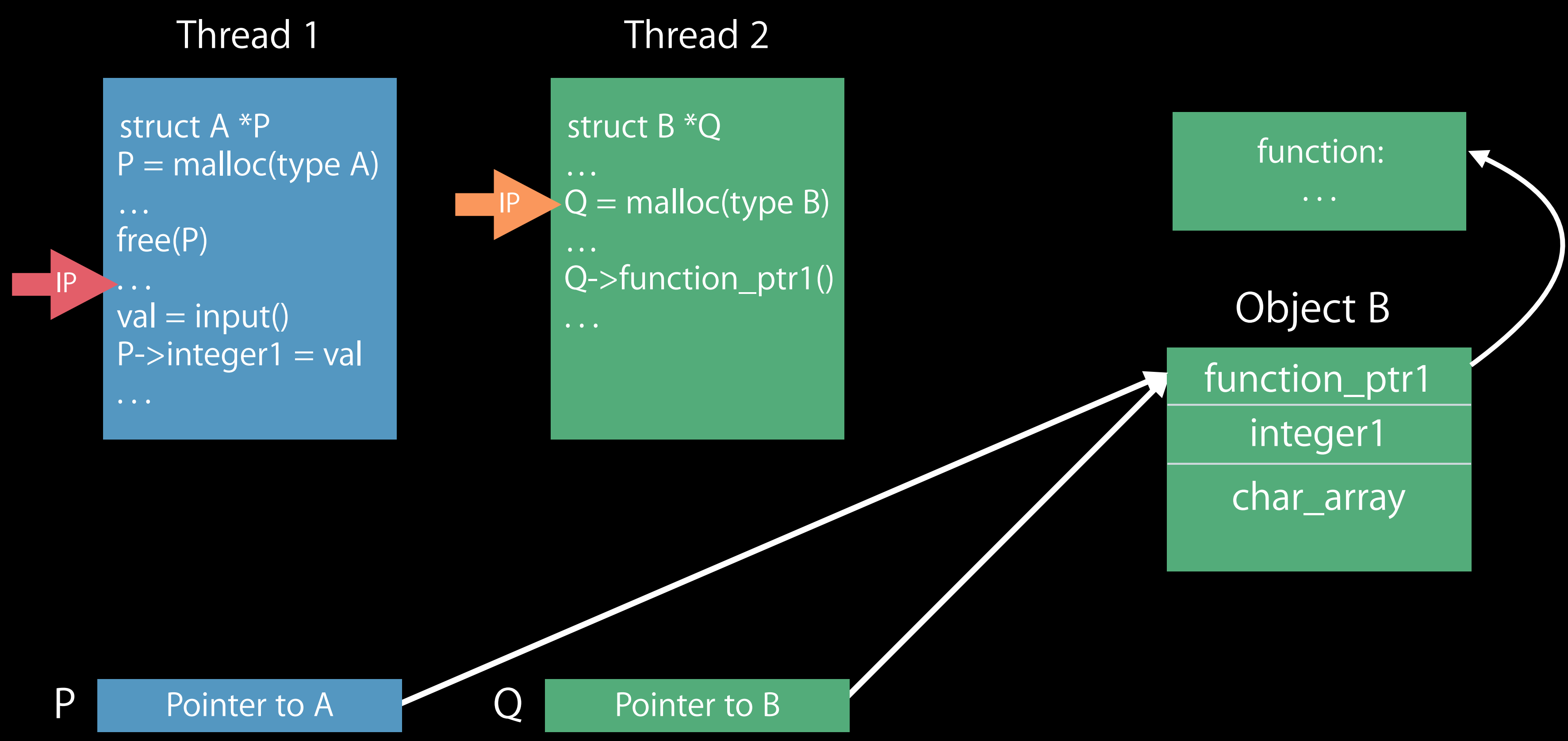
Use-After-Free Example

Thread 1 frees type A object in heap memory



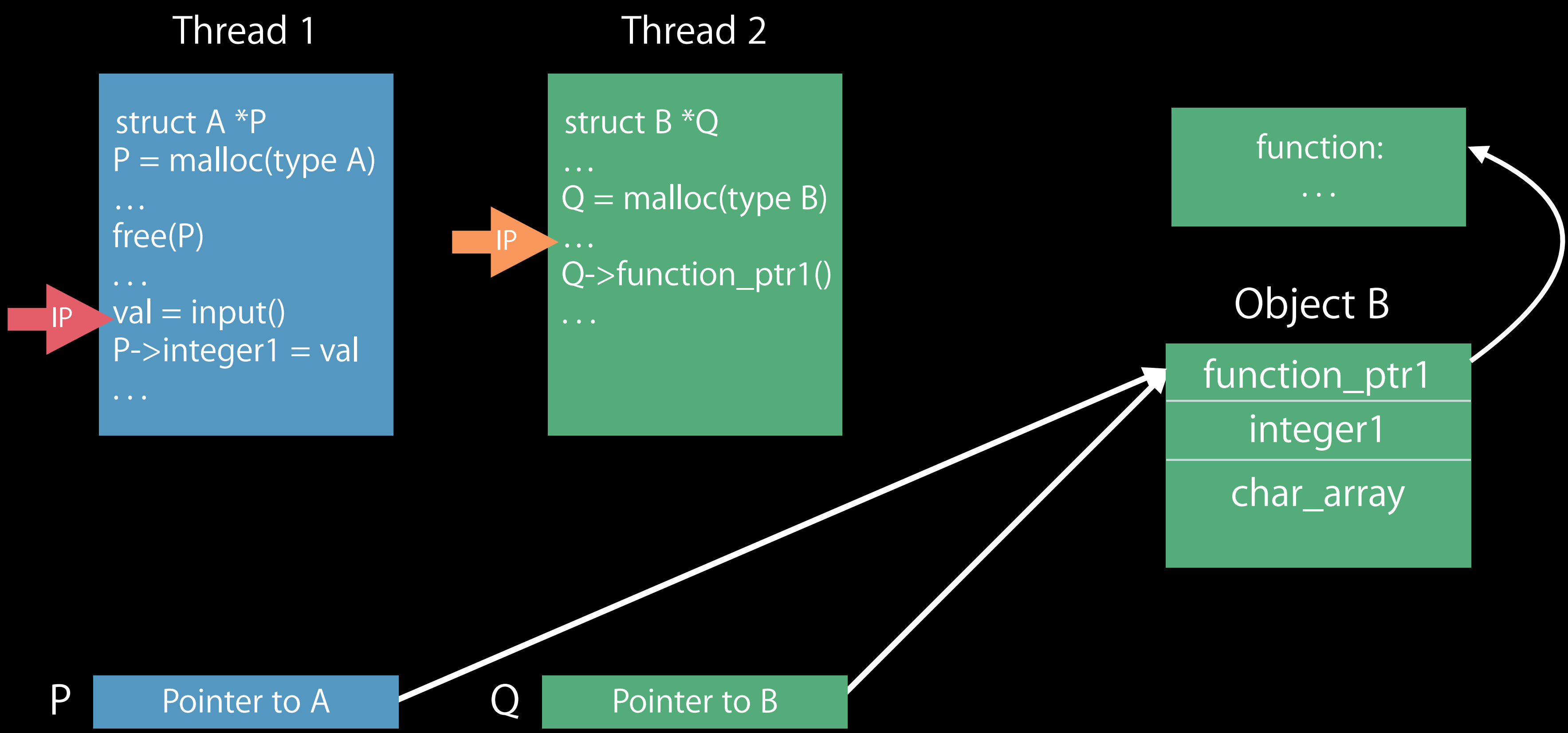
Use-After-Free Example

Thread 2 allocates type B object in heap (in same space that A used)



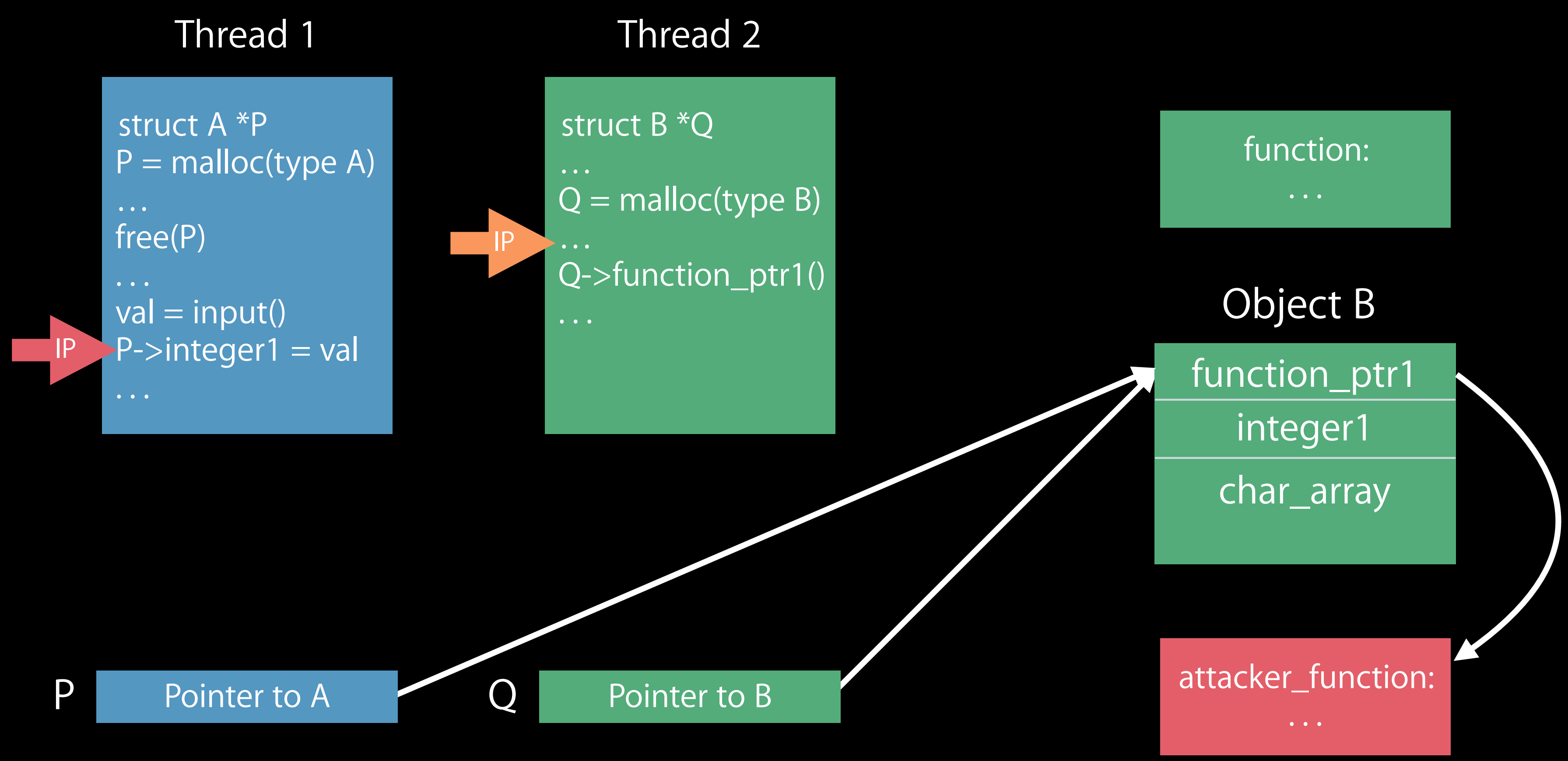
Use-After-Free Example

Attacker is able to input an value that is a valid pointer value



Use-After-Free Example

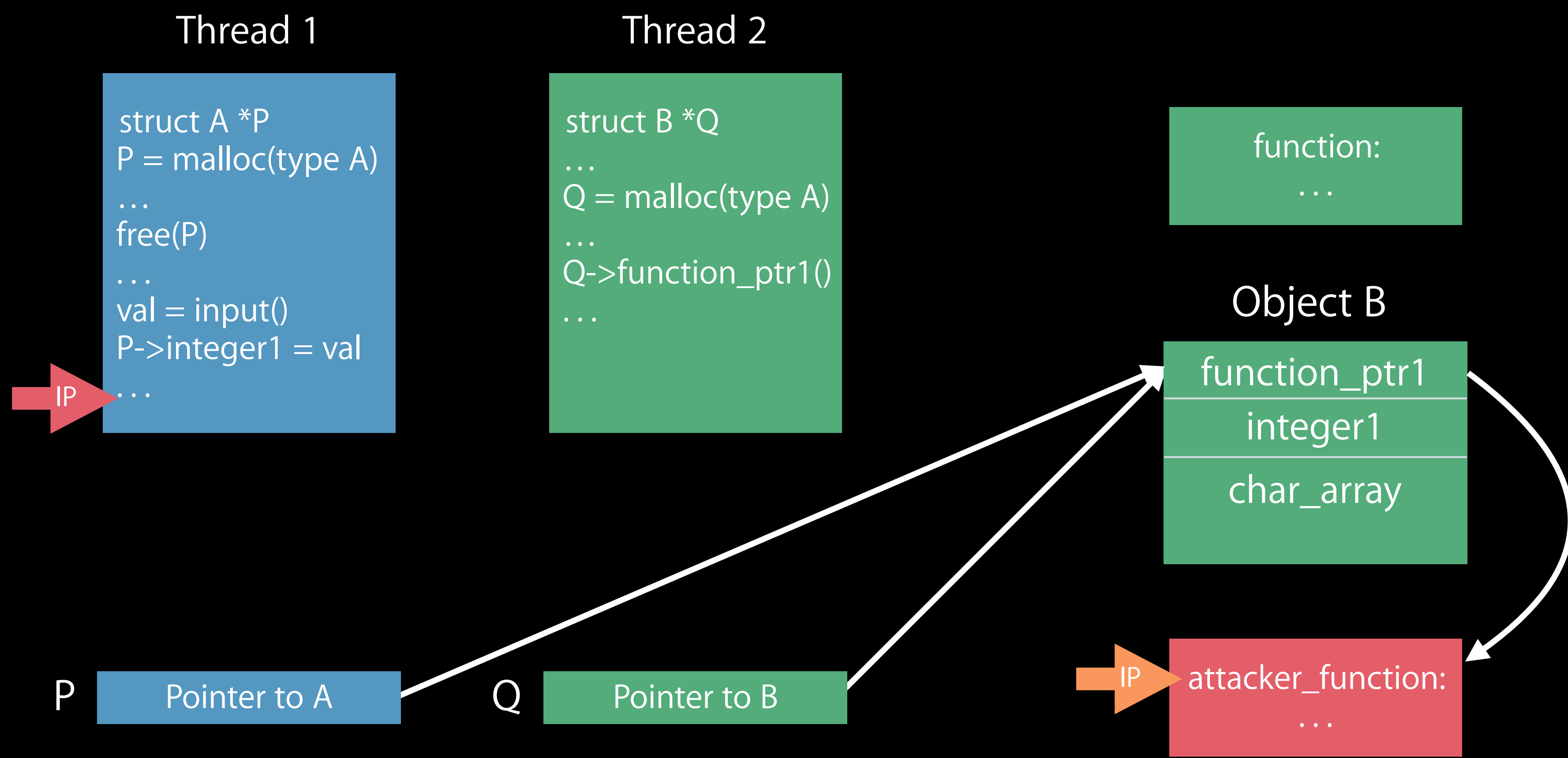
Thread 1 overwrites function pointer with pointer value from attacker



Stacey D. Son (sson at me dot com)

Use-After-Free Example

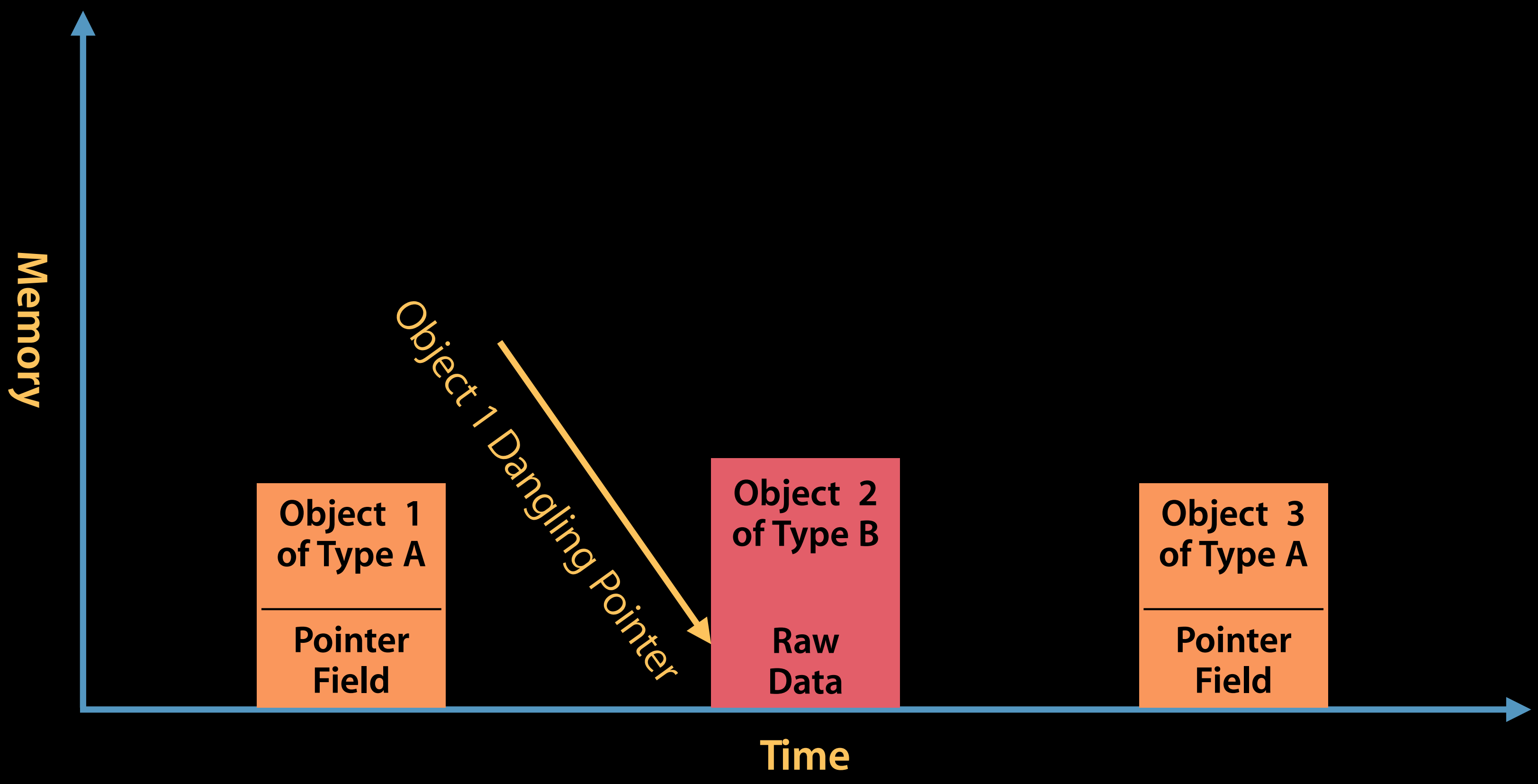
Thread 2 is now running attacker's code



Possible UAF Mitigation Ideas

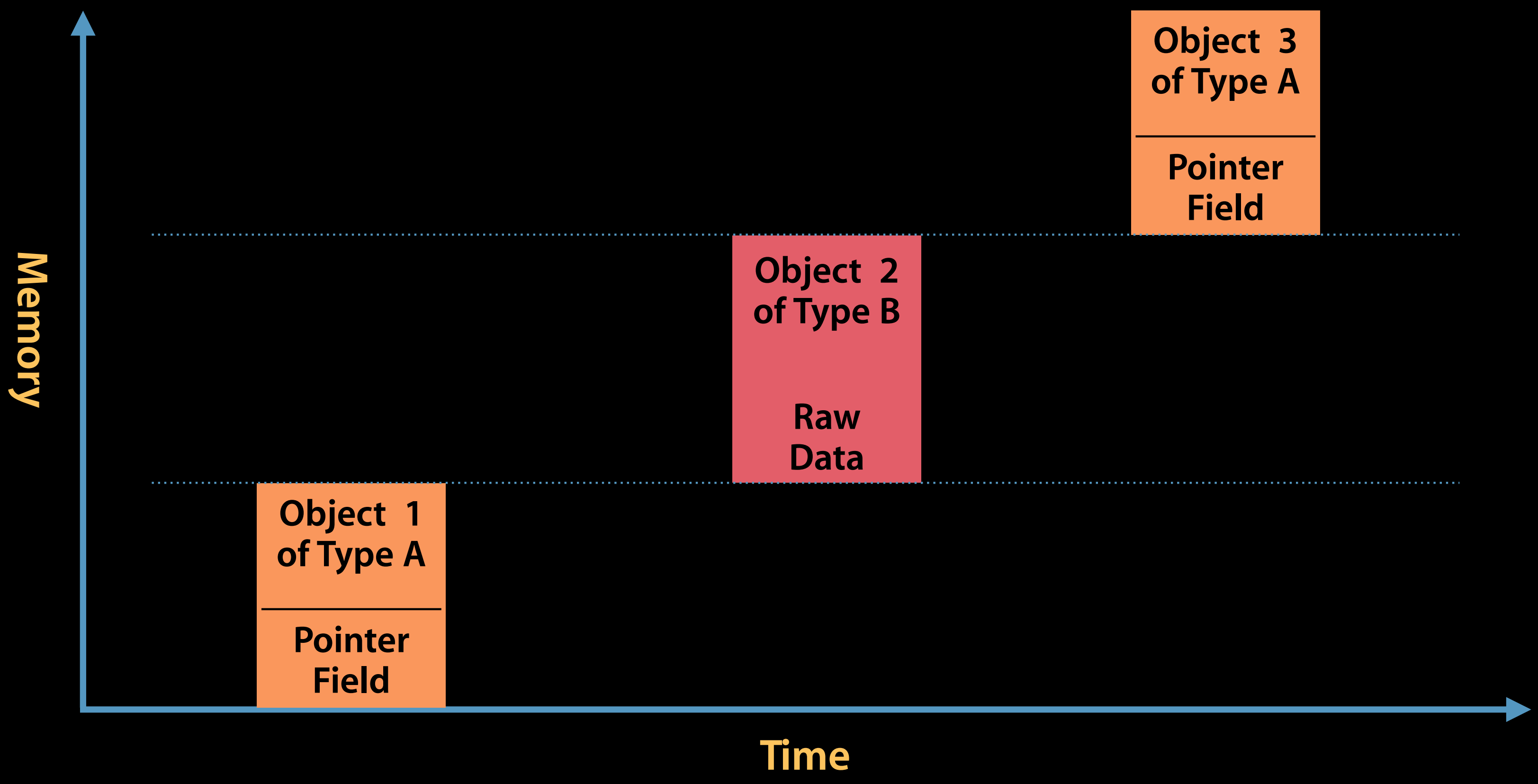
Temporal Memory Safety Violation

Use-After-Free bug shown on a Memory-Time Graph…



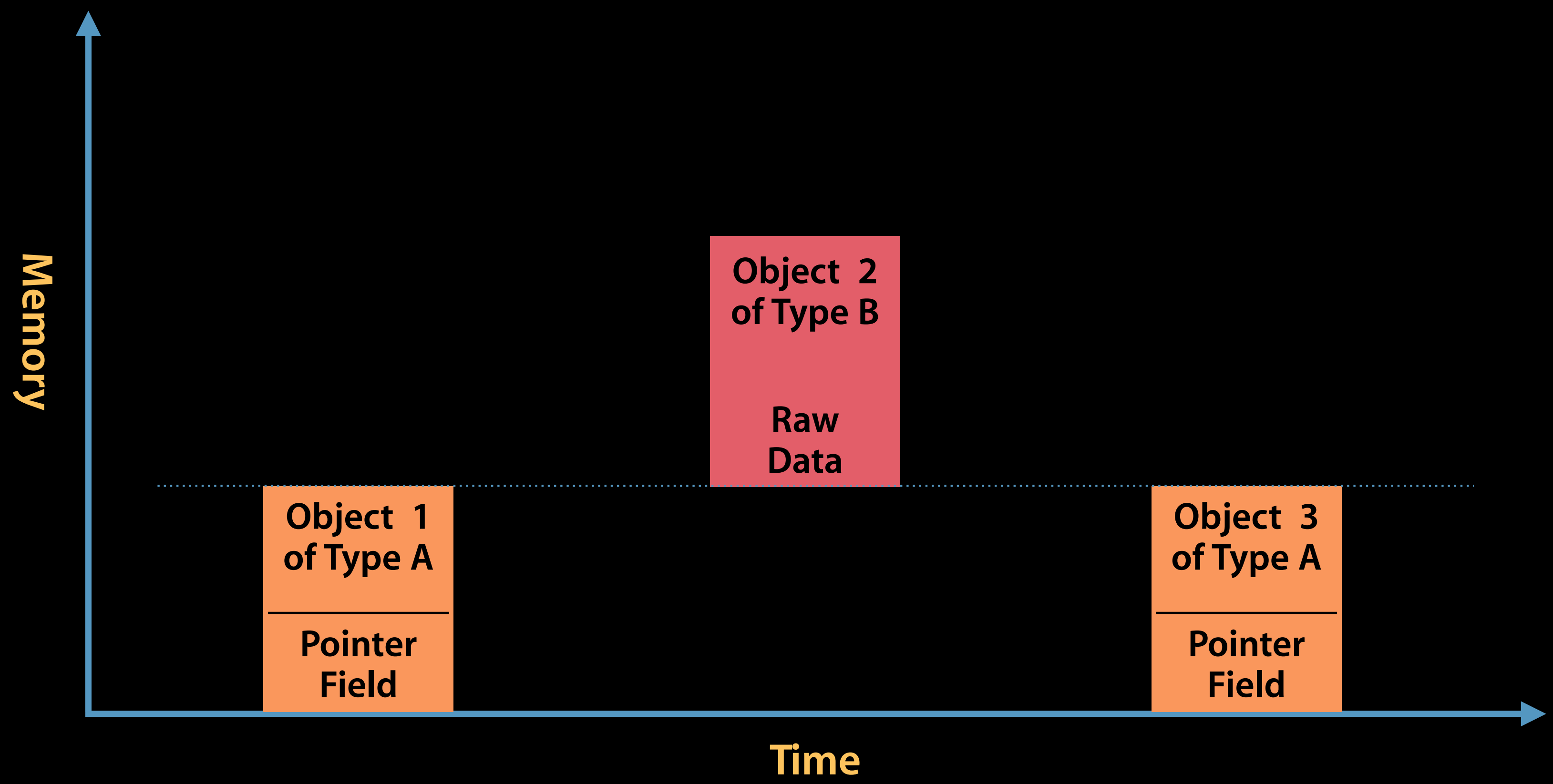
Non Virtual Memory Reuse Mitigation

Assumption: Virtual Memory Address Space is Cheap



Type-safe Memory Reuse Mitigation (AKA, "IsoHeaps")

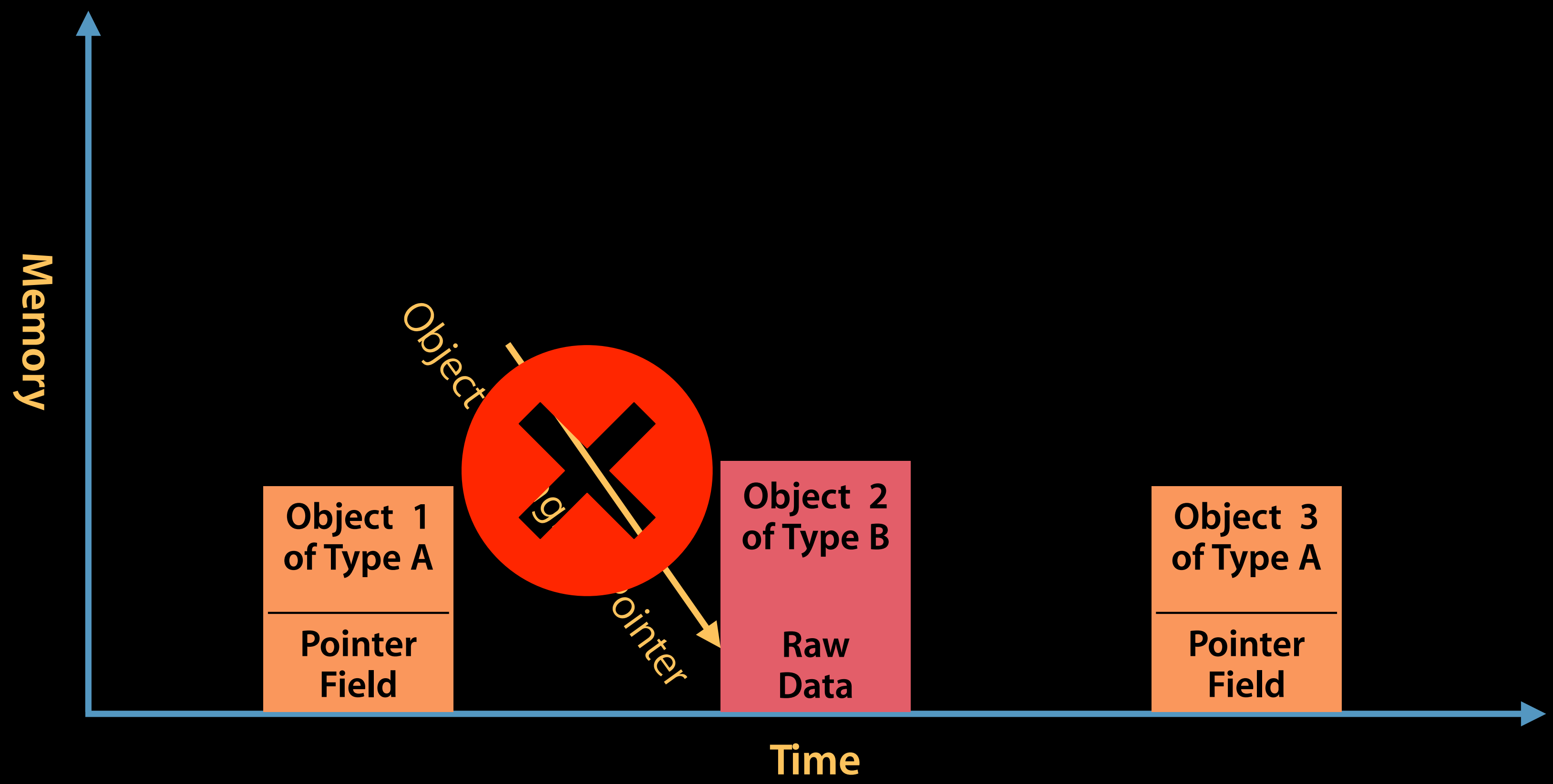
Assumption: Eliminating Type Confusion Solves the Problem



"Towards the next generation of XNU memory safety: kalloc_type" https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/

Sweep and Revoke Dangling Pointers

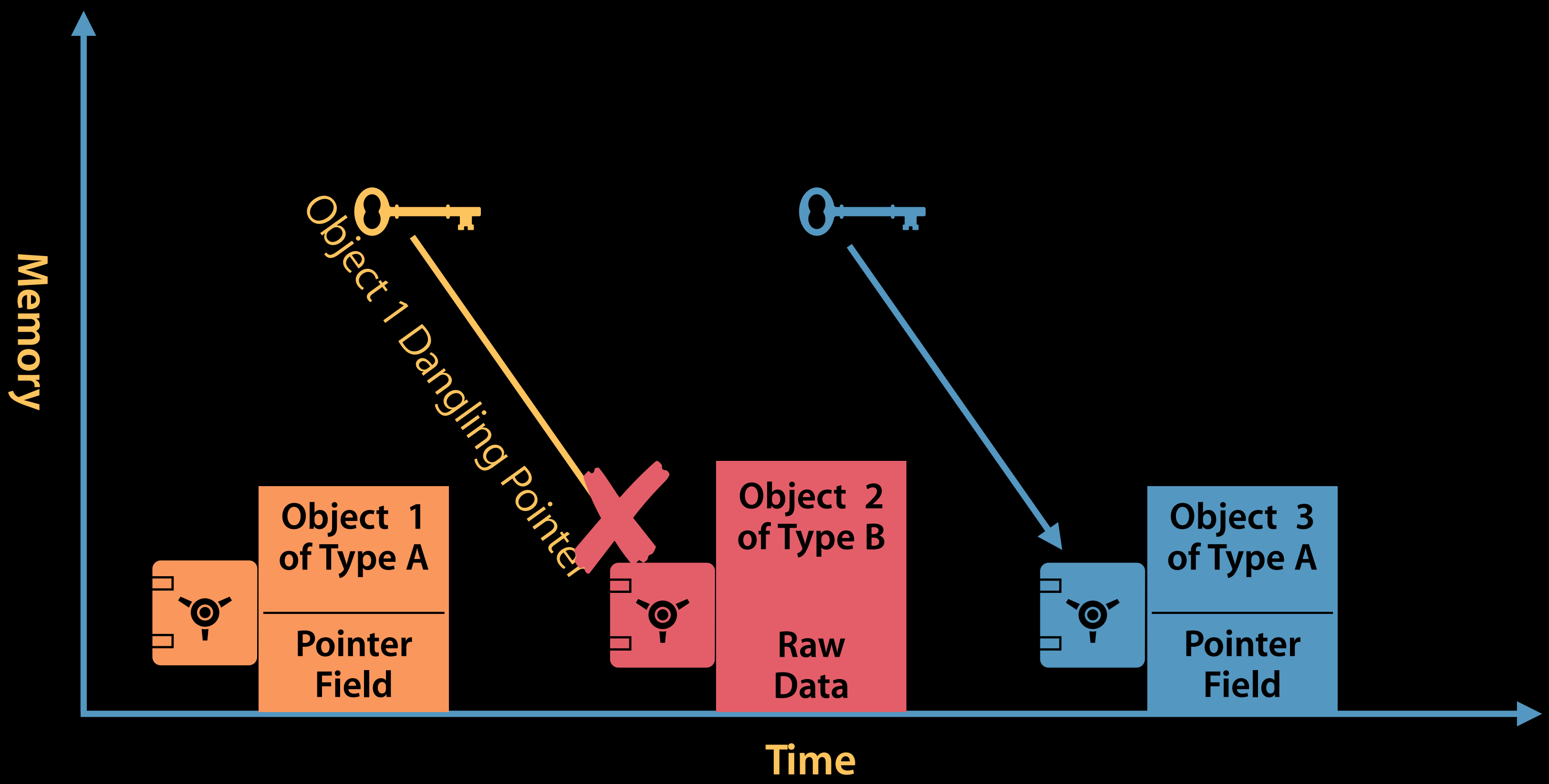
Assumption: Dangling pointers can be easily found and disabled



“Cornucopia: Temporal Safety for CHERI Heaps" https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2020oakland-cornucopia.pdf

Lock (Memory Object) and Keys (Pointers) Mitigation

Assumption: Change the Lock and Keys No Longer Work



"Introduction to the Memory Tagging Extension"

Real World Exploits - e.g, Jeff Bezos' phone

- Long exploit chains
  - Memory corruption exploit in WhatsApp
  - Memory corruption exploit to "jailbreak" (escape app sandbox)
  - Information leak in kernel to determine its memory offsets
  - Kernel memory corruption exploit to get kernel privileges
  - Download and installation of spyware

WIRED SECURITY POLITICS THE BIG STORY BUSINESS SCIENCE CULTURE REVIEWS

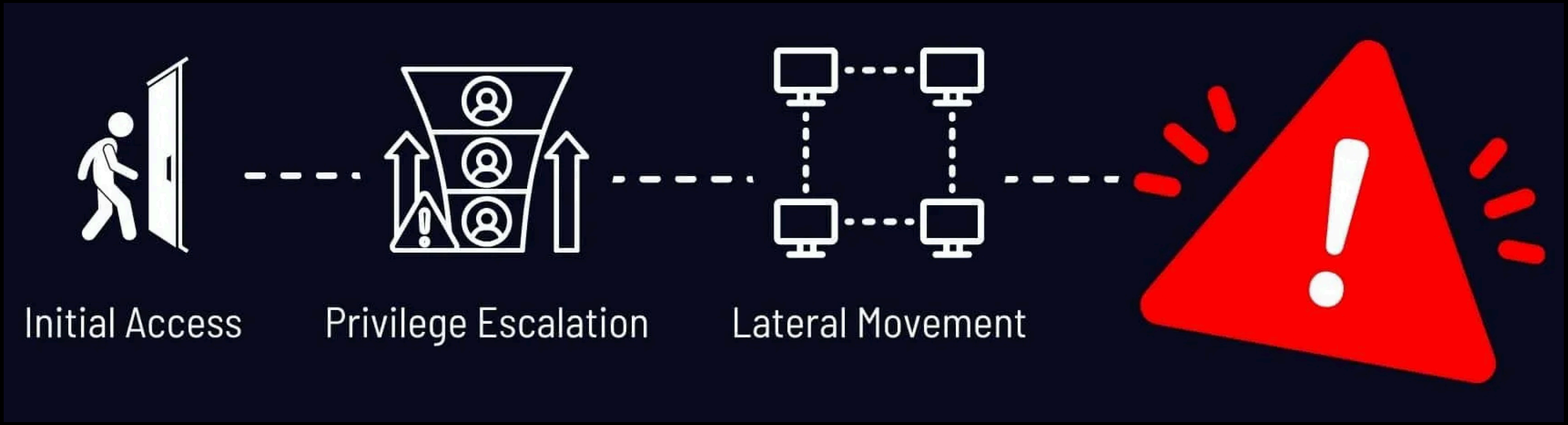
Everything We Know About the Jeff Bezos Phone Hack

A UN report links the attack on Jeff Bezos' iPhone X directly to Saudi Arabian Crown Prince Mohammed bin Salman.

Investigators say that a video file sent from the Saudi crown prince to Jeff Bezos over WhatsApp was loaded with malware. PHOTOGRAPH: ANDREW HARRER/BLOOMBERG/GETTY IMAGES

ON NOVEMBER 8, 2018, Amazon CEO Jeff Bezos received an unexpected text message over WhatsApp from Saudi Arabian leader Mohammed bin Salman. The two had exchanged numbers several months prior, in April, at a small dinner in Los Angeles, but weren't in regular contact; Bezos had previously

Exploit Chains



- Initial Access or “grappling hook” (e.g., WhatsApp)
- Sandbox escape, “root”, and kernel privilege escalations
- Payload download and installations. Modify base OS software. Add spyware. etc.

With good memory safety, most exploit chains are broken.. Unfortunately, not all bugs are memory safety related

Memory Integrity Enforcement vs. real-world exploit chains

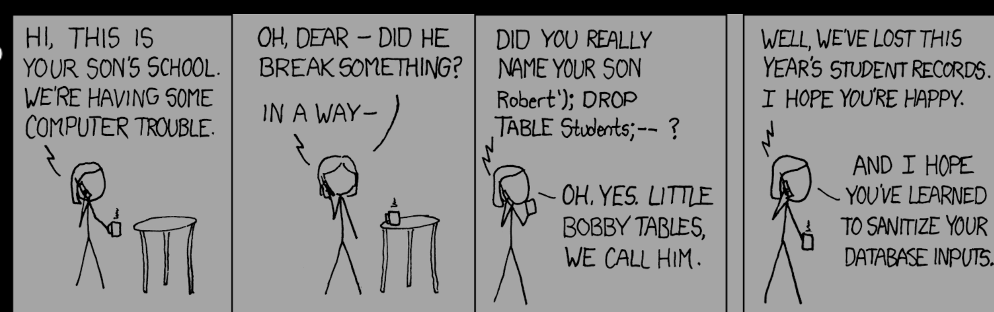
Blocked by secure allocators

Blocked by EMTE

Blocked by secure allocators and EMTE

Surviving step

Logical step



Coming up on “The World of Memory Safety”…

- Thursday’s Episode: “Modern CPU Extensions for Memory Safety (Part 1): ARM PAC, BTI, and MTE”
- Next Tuesday’s Episode: “Modern CPU Extensions for Memory Safety (Part 2): CHERI and CHERIoT”