CS 453/698: Software and Systems Security

Module: Operating Systems Security

Lecture: Malware, Systems Security, and Adversary Actions

Adam Caulfield

University of Waterloo

Spring 2025

This course so far…

Topics covered…

- Cryptography
- Compilers, the stack, the heap
- Memory errors (e.g., buffer overflow, use-after-free, format string)
- Software security mitigations (e.g., memory safety, ASLR, CFI)
- Race conditions, data races, atomicity violations
- Input sanitization, fuzzing, static analysis, symbolic execution

Software (part 1) and Systems (part 2) Security

This course so far...

Topics covered...

- Cryptography
- Compilers, the stack, the heap
- Memory errors (e.g., buffer overflow, use-after-free, format string)
- Software security mitigations (e.g., memory safety, ASLR, CFI)
- Race conditions, data races, atomicity violations
- Input sanitization, fuzzing, static analysis, symbolic execution

Software (part 1) and Systems (part 2) Security

Today!! → finish software security, intro to systems security

Outline

- A little about me…
- What is malware? What are the types of malware?
- Intro to systems security…. Reflections on Trusting Trust
- Adversarial Actions
  - What steps will be taken to compromise the system?

Outline

- A little about me...
- What is malware? What are the types of malware?
- Intro to systems security.... Reflections on Trusting Trust
- Adversarial Actions
  - What steps will be taken to compromise the system?

A little about me...

Currently a postdoctoral research scholar

- In Secure Systems Group (SSG) supervised by Prof. N. Asokan
- Jan 2025 - Ph.D. in Computing and Information Sciences
  - Rochester Institute of Technology
  - Advised by Dr. Ivan De Oliveira Nunes
- May 2019/2020 - B.S and M.S. in Computer and Electrical Engineering
  - University of Delaware

- I do systems security research, particularly in:
  - Computer Architecture, Embedded Systems, Trusted Execution Environments, Trusted Hardware Design, Program/Binary Analysis,
  - Dissertation Focus: *Runtime Security*
    - *Proofs of Execution, Control Flow Integrity, and Control Flow Attestation*
- I publish my work at Security and EDA conferences:
  - USENIX Security, IEEE S&P (Oakland), ACSAC, ICCAD, DAC
- Interested in research? Come chat after class any time

A little about me...

As it relates to this course:

- My expertise is systems security
- Particularly excited for OS, Mobile, Hardware security modules

Contacting me:

- Feel free to email me (acaulfie@uwaterloo.ca)
- Or tag me in Piazza

Office hours (same as Meng)

- Tuesday 11am-12pm in BBB
  - If you're coming, email me day before as a heads up...
- If that time doesn't work, email me

Outline

- A little about me…
- What is malware? What are the types of malware?
- Intro to systems security…. Reflections on Trusting Trust
- Adversarial Actions
  - What steps will be taken to compromise the system?

What is malware?

Definition: various forms of software written with malicious intent

Common Characteristic: need to be *executed in order* to cause harm

How might malware get executed?

- User action
  - Downloading and running malicious software
  - Viewing a web page containing malicious code
  - Opening an executable in an email attachment
  - Inserting CD/DVD or USB flash drive
- Exploiting existing flaw in the system
  - Memory vulnerability

Types of Malware

- Virus
- Worms
- Trojans
- Logical Bombs

Types of Malware - Virus

A virus is a specific type of malware that "infects" other files

- Traditionally, a virus could infect only executable programs
- But now, data documents can also contain executable code
  - Macros in .xlsx or javascript in .pdf

Upon opening an infected file, the virus will:

- Activate itself
- Copy itself into other files
- Execute its payload
- (sometimes) spread as far as it can: locally or to other machines

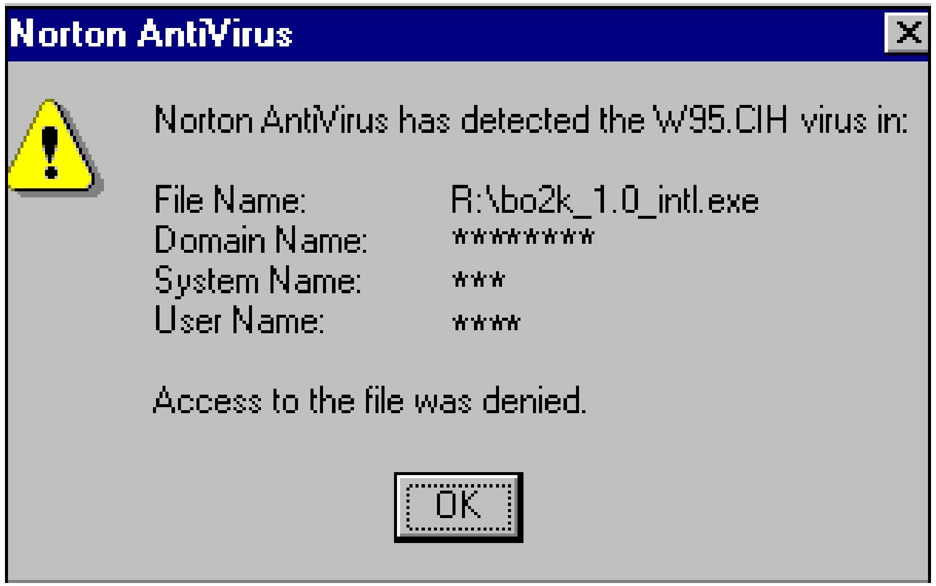
Types of Malware - Virus

Example virus: CIH “Chernobyl” virus, 1998

- Written by a student
- A challenge against antivirus software
- Spread under “portable executable file”

Once activated, it:

- Overwrite the first megabyte of the hard drive
- Attempt to corrupt the computer’s BIOS



Antivirus intercept CIH virus

Types of Malware - Worm

A worm is a self-contained piece of code that can replicate with little or no user involvement

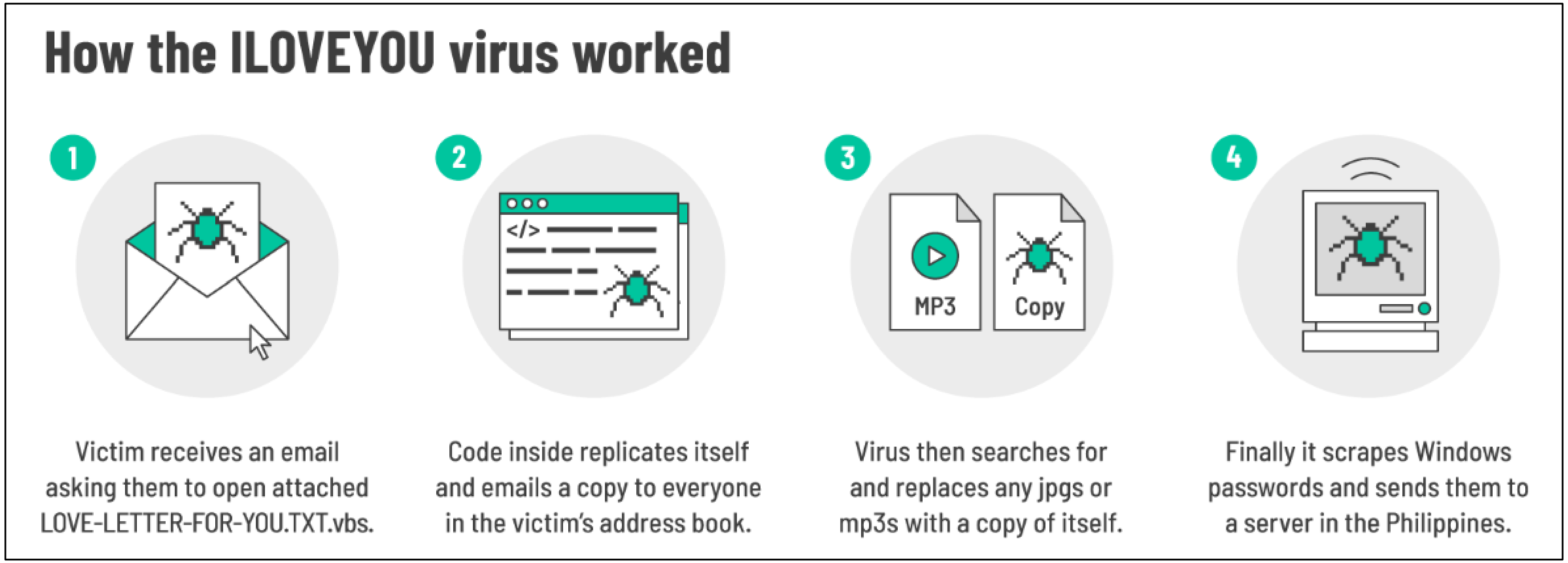
Worms often execute the following steps:

- Exploits security flaw to infect software
- Searches for other computers in local network or internet
- Perform some action (e.g., steal data, denial of service)

Types of Malware - Worm

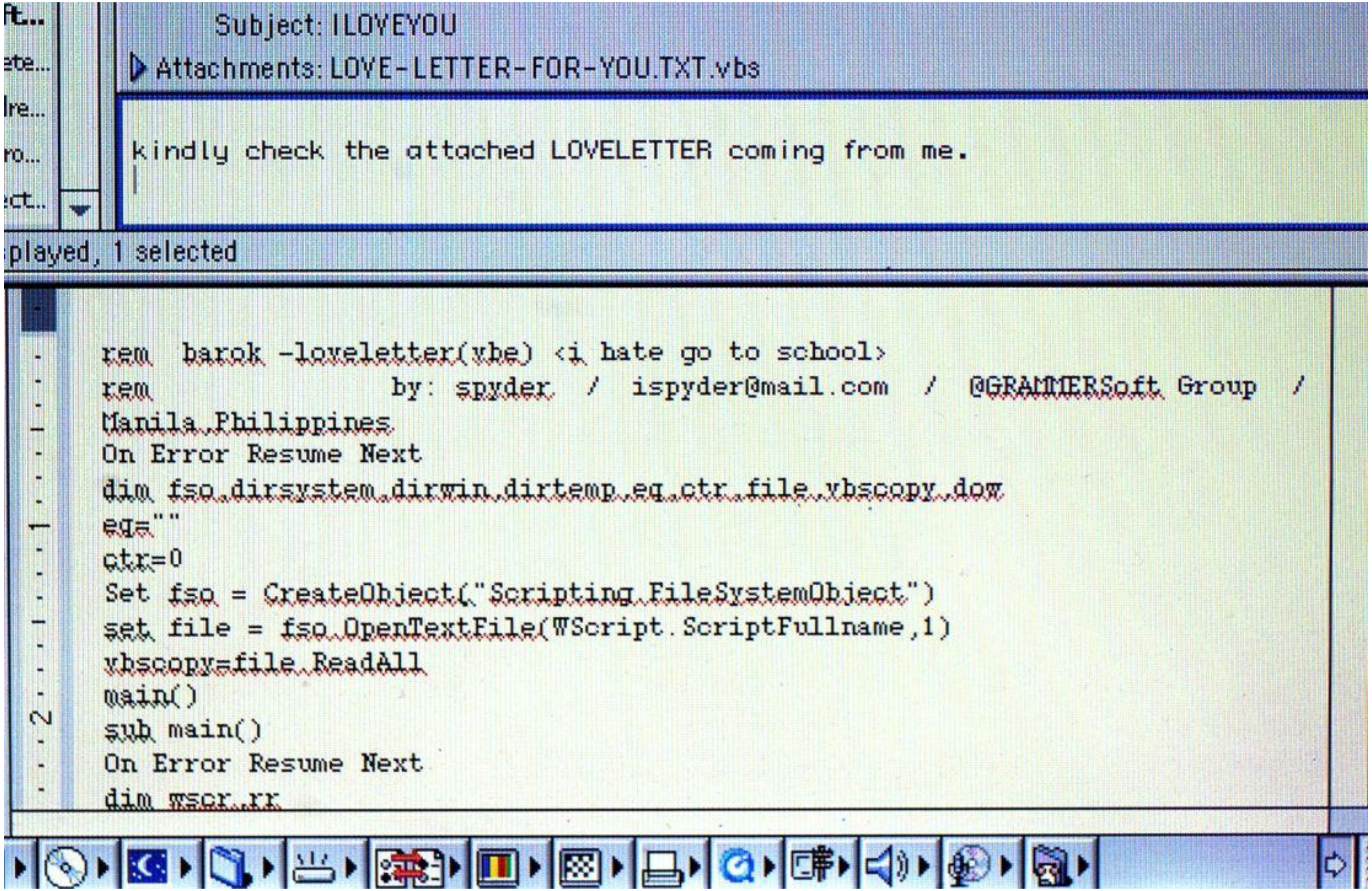
Example worm: ILOVEYOU

- One of the farthest-reaching worms, May 2000
- Infected almost every military base in the USA
  - USA Department of the Army: "12,010 manhours lost, estimated cost $79.2k"



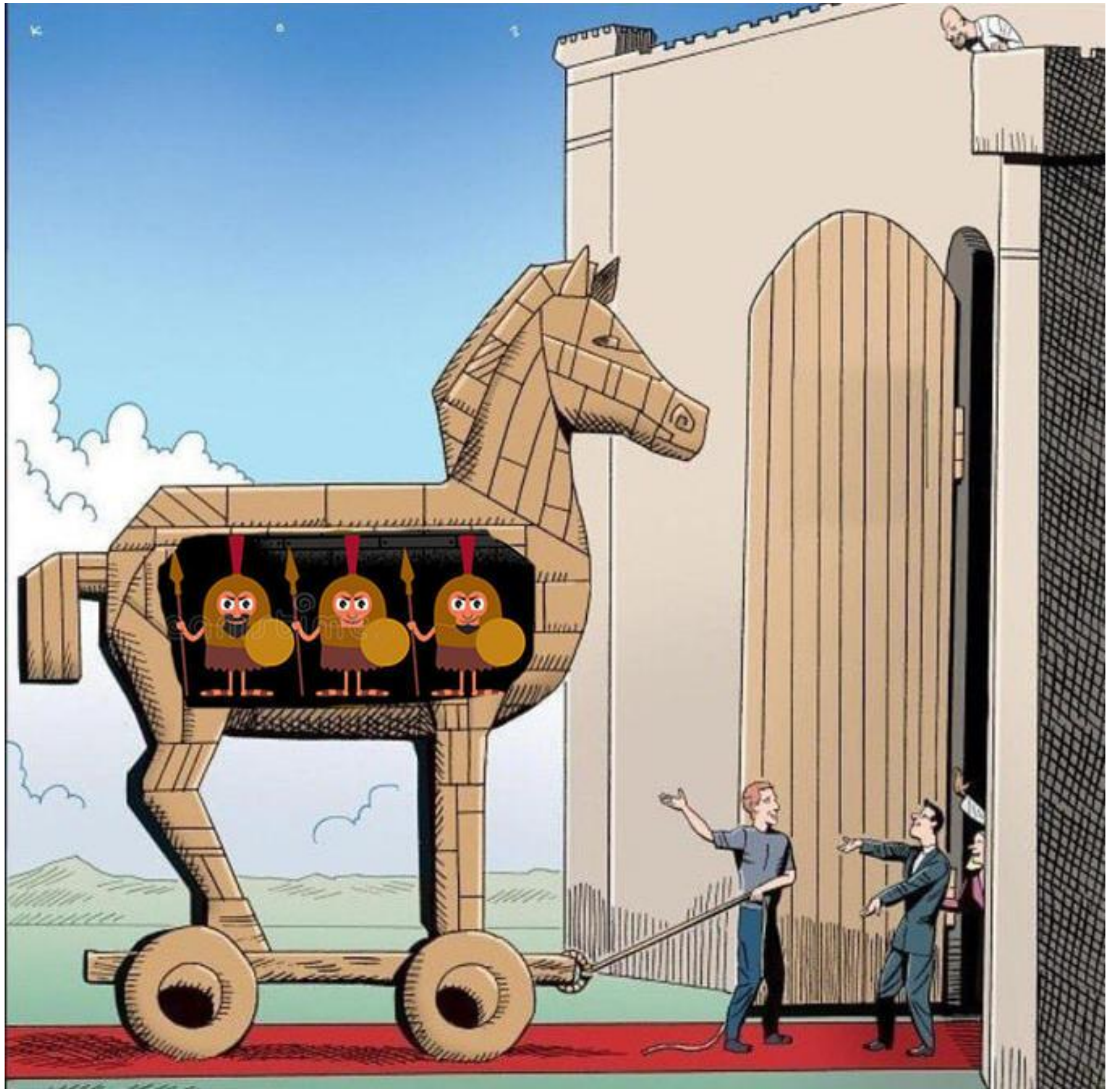
Steps of the ILOVEYOU worm

Types of Malware - Worm



Screenshot showing a copy of the ILOVEYOU worm

Types of Malware – Trojan Horses



Types of Malware – Trojan Horses

Trojan Horses (aka trojans) are programs which claim to do something innocuous while hiding malicious behavior.

Unlike viruses or worms, do not try to inject themselves into other files

Trojans might embed themselves in:

- Email attachment
- Fake software updates
- “Free” movies or games

Types of Malware – Trojan Horses

Example: Zbot (Zeus Trojan), 2007

- Starts through a phishing email with a download to an attachment
  - Used to download the malware after a user executes it
- Local machine becomes part of the Zues Trojan botnet
  - Giving the owner control of the device
  - Use keylogging to get user's passwords, bank information
  - More information: https://www.proofpoint.com/us/threat-reference/zeus-trojan-zbot

Types of Malware – Logic Bomb

A logic bomb is malicious code hiding in the software that is already installed on a machine

- Then, it waits for a certain trigger to "go off" (execute its payload)

Example payloads in logic bombs:

- Erase data
- Corrupt data
- Encrypt data – charge you for decryption key (ransom)

Types of Malware – Logic Bomb

Where do logic bombs come from?

- Targeted planting
- From backdoors that
  - Developers forgot to remove
  - Were intentionally left for testing, maintenance, or legal purposes
  - Intentionally left for malicious purposes (insider attack)

Types of Malware – Logic Bomb

Example: Siemens logic bomb, 2019

- Insider attack – planted by former Siemens contractor
- Planted logic bombs inside spreadsheets
  - Inserted two years before they "triggered"
- Made all custom scripts in the spreadsheets crash
  - Siemens would only call the contractor, since he knew how to fix them
  - He would charge them a fee each time

- Story:
  - https://www.zdnet.com/article/siemens-contractor-pleads-guilty-to-planting-logic-bomb-in-company-spreadsheets/

Detecting Malware

When should we look for malware?

- As files are added to the system
  - Via Portable media, network channel
- Periodic scans of the entire computer
  - In hopes to catch anything we might have missed

General approaches:

- Signature-based protection
- Behavior-based protection

Detecting Malware

Signature-based Protection

- Keep a list of known malware

- For each malware in the list, store some characteristic feature
  - E.g., a ***signature*** of the malware

- Most use a feature of the malware code itself
  - The infection code
  - The payload code

- Can also try to identify other characteristics of malware
  - Hiding places within programs
  - Propagation characteristics

Detecting Malware

Signature-based Protection

- Limitations?
  - Can only scan for viruses that are in the list
  - New types of malware are constantly emerging
  - Some malware is ***polymorphic*** – does not make exact copies of itself

That's where ***behavior-based*** systems come in

- Does not search for static code fragments
- Detection is based on behavioral patterns

Detecting Malware

Examples:

Microsoft Defender Antivirus (Behavior-based detection)

Behavior monitoring is a critical detection and protection functionality of Microsoft Defender Antivirus.

Monitors process behavior to detect and analyze potential threats based on the behavior of applications, services, and files. Rather than relying solely on signature-based detection (which identifies known malware patterns), behavior monitoring focuses on observing how software behaves in real-time. Here's what it entails:

1. Real-Time Threat Detection:
   - Continuously observe processes, file system activities, and interactions within the system.
   - Defender Antivirus can identify patterns associated with malware or other threats. For example, it looks for processes making unusual changes to existing files, modifying or creating automatic startup registry (ASEP) keys, and other alterations to the file system or structure.
2. Dynamic Approach:

- Unlike static, signature-based detection, behavior monitoring adapts to new and evolving threats.
- Microsoft Defender Antivirus uses predefined patterns, and observes how software behaves during execution. For malware that doesn't fit any predefined pattern, Microsoft Defender Antivirus uses anomaly detection.
- If a program shows suspicious behavior (for example, attempting to modify critical system files), Microsoft Defender Antivirus can take action to prevent further harm, and revert some previous malware actions.

https://learn.microsoft.com/en-us/defender-endpoint/behavior-monitor

Norton Antivirus (combination of both)

How risks are detected

Applicable For: Windows

Norton uses several techniques to monitor and protect your devices from viruses, spyware, adware, and other security risks. The most common method is signature-based threat detection. Each time you run a virus scan, Norton obtains the virus definitions and performs a scan. It compares the contents of the files against the known threat signatures to identify threats.

Norton also uses heuristic detection to protect your device from threats for which signatures are unknown.

The following features in Norton continually protect your device from security threats:

- **Auto-Protect**

  Auto-Protect options let you customize the protection of your computer. The Auto-Protect feature does the following:
  - Monitors your computer for any unusual symptoms that might indicate an active security risk.
  - Loads into memory when Windows starts, providing constant protection while you work.
  - Checks for viruses, spyware, and other risks every time that you use software programs on your computer. It also checks every time when you insert any removable media, access the Internet, or use the document files that you receive or create.
- **Full Scan**

  Checks all the boot records, the files, and the programs that are running to protect your computer from viruses and spyware.
- **Quick Scan**

  The security risks that are running on your computer affect most of the areas of your computer. Quick Scan performs a fast scan of those affected areas. Run Quick Scan if you do not have time for a Full Scan but suspect that a security risk is running on your computer.

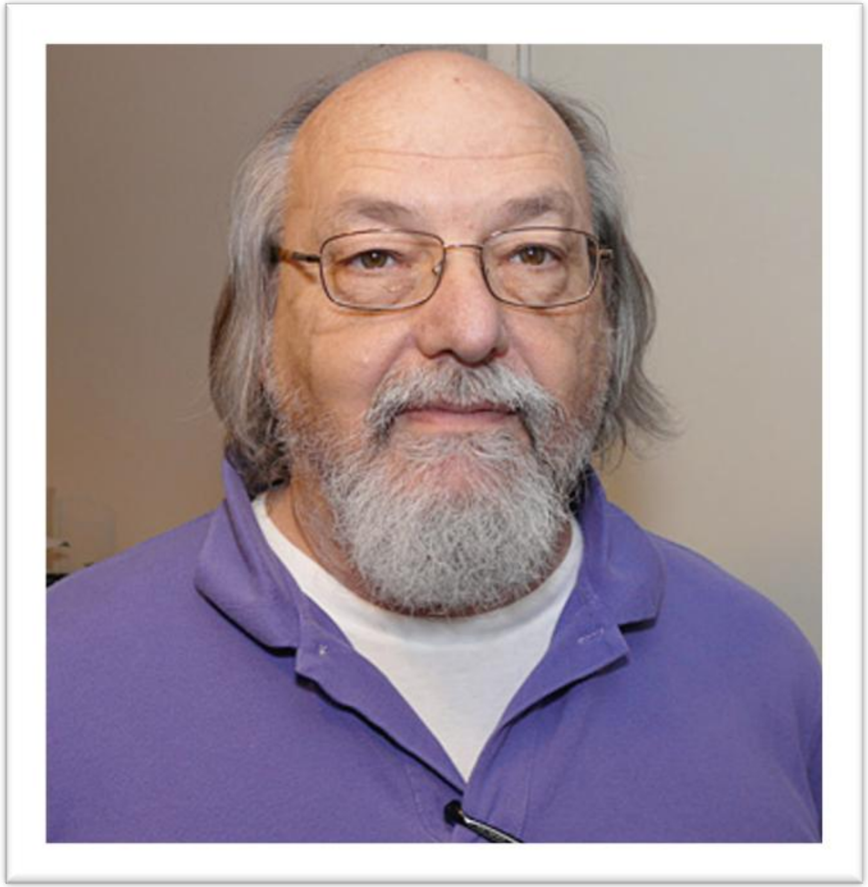
https://support.norton.com/sp/en/us/home/current/solutions/v20240108182054157?

Outline

- A little about me...
- What is malware? What are the types of malware?
- Intro to systems security.... Reflections on Trusting Trust
- Adversarial Actions
  - What steps will be taken to compromise the system?

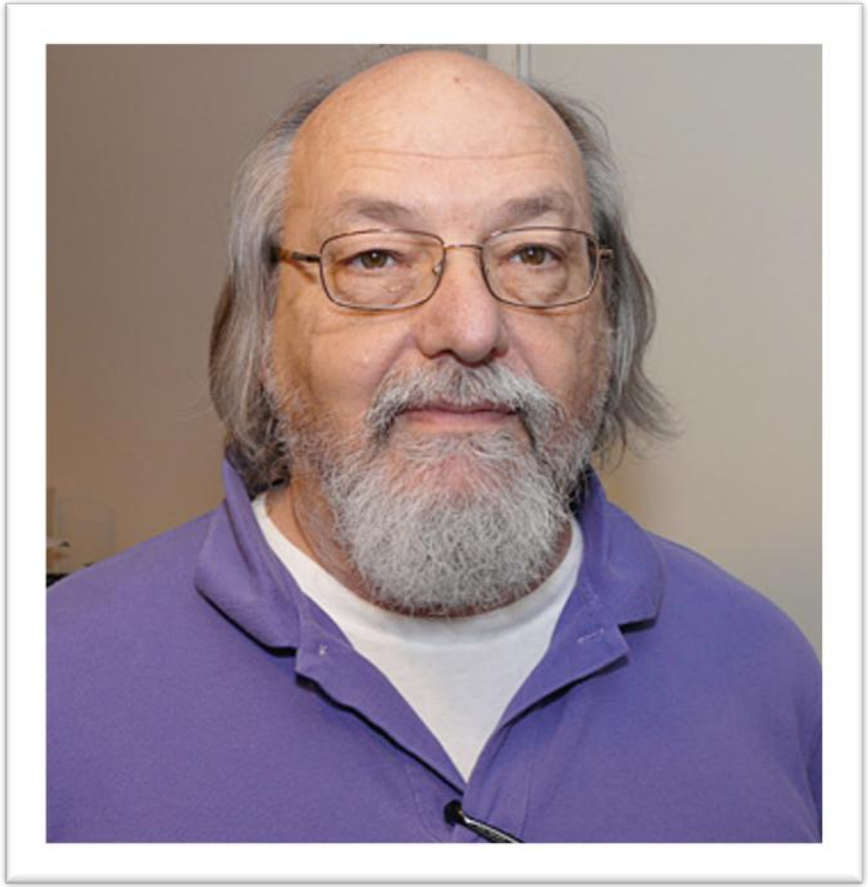
Reflections on Trusting Trust

Ken Thompson



Reflections on Trusting Trust

Ken Thompson



Unix, B/C co-designer, Go co-inventor, 1983
ACM Turing Award Recipient

Reflections on Trusting Trust

What code can we trust?

Consider “login” or “su” in Unix

- Why are these binaries “reliable” ?

Reflections on Trusting Trust

What code can we trust?

Consider “login” or “su” in Unix

- Why are these binaries “reliable” ?
- Is Ubuntu reliable? RedHat? Android?
- Does it send your password to someone?
- Does it have a backdoor for a “special” remote user?

- Can’t trust the binary
  - So, check source code or write your own, then recompile
  - Does this solve the problem?

Reflections on Trusting Trust

Can we trust the compiler itself?

- What if:
  - Compiler looks for source code that resembles the login process
  - Inserts a backdoor into the process
- Okay, so we can inspect the source code of the compiler
  - Looks good? Recompile the compiler!
- Does this solve the problem?

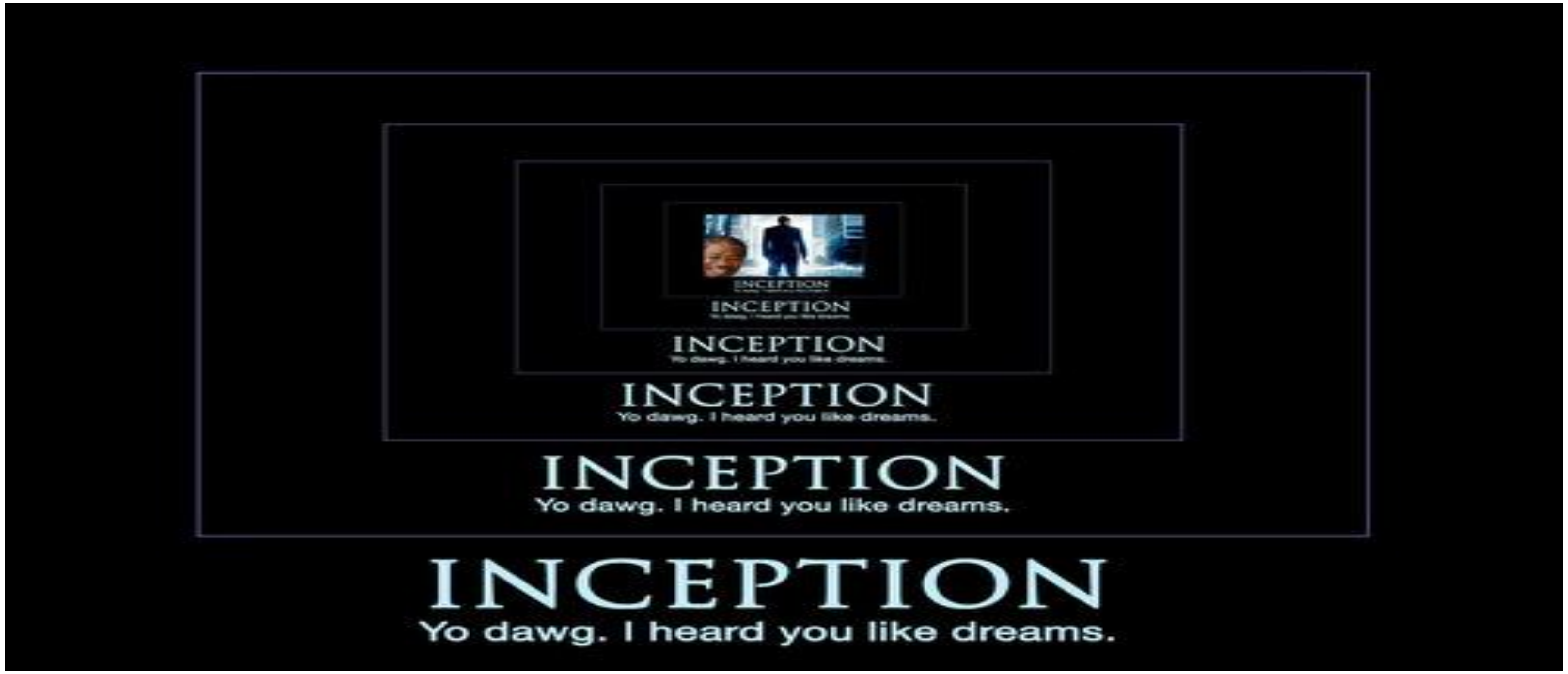
Reflections on Trusting Trust

- The compiler is written in C …

```
compiler(S) {
    if (match(S, "login-pattern")) {
        compile (login-backdoor)
        return
    }
    if (match(S, "compiler-pattern")) {
        compile (compiler-backdoor)
        return
    }
    .... /* compile as usual */
}
```

Reflections on Trusting Trust

“ The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code.”



Reflections on Trusting Trust

Moral of the story:

- We have to trust something
- Comprehend and minimize the amount of trust
- Trust is ***transitive***: if you trust something, you trust what it trusts

Systems Security:

- Understand the attacker's goal, entry point, capabilities
- Identify the **Root of Trust** for a system:
  - The parts that are relied upon (trusted) without verification
- Build and deploy defenses accordingly

Outline

- A little about me...
- What is malware? What are the types of malware?
- Intro to systems security…. Reflections on Trusting Trust
- Adversarial Actions
  - What steps will be taken to compromise the system?

Adversarial Actions

We must always look from the adversary point of view...

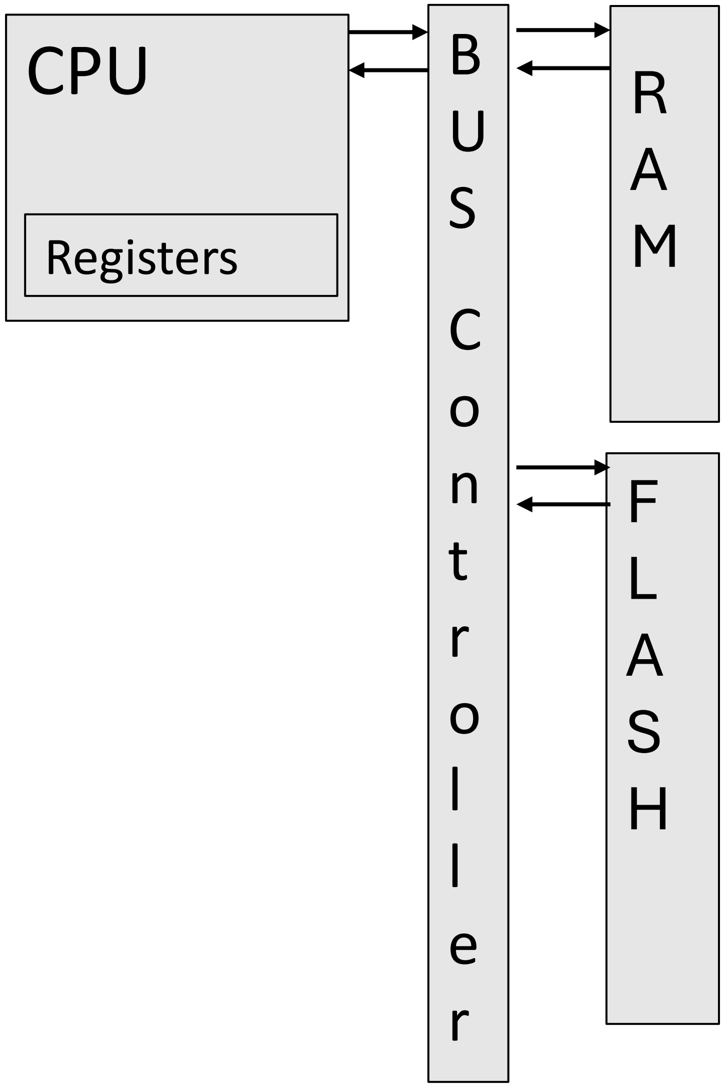
Types of malware describe **WHAT** an adversary's attack does

So, **HOW** does the adversary get there?

Let's look at the several phases of the attack

1. Identifying the memory vulnerability
2. Violating integrity
3. Identifying exploit payload
4. Dispatching the exploit
5. Executing the exploit
6. Achieving the attack

System Model

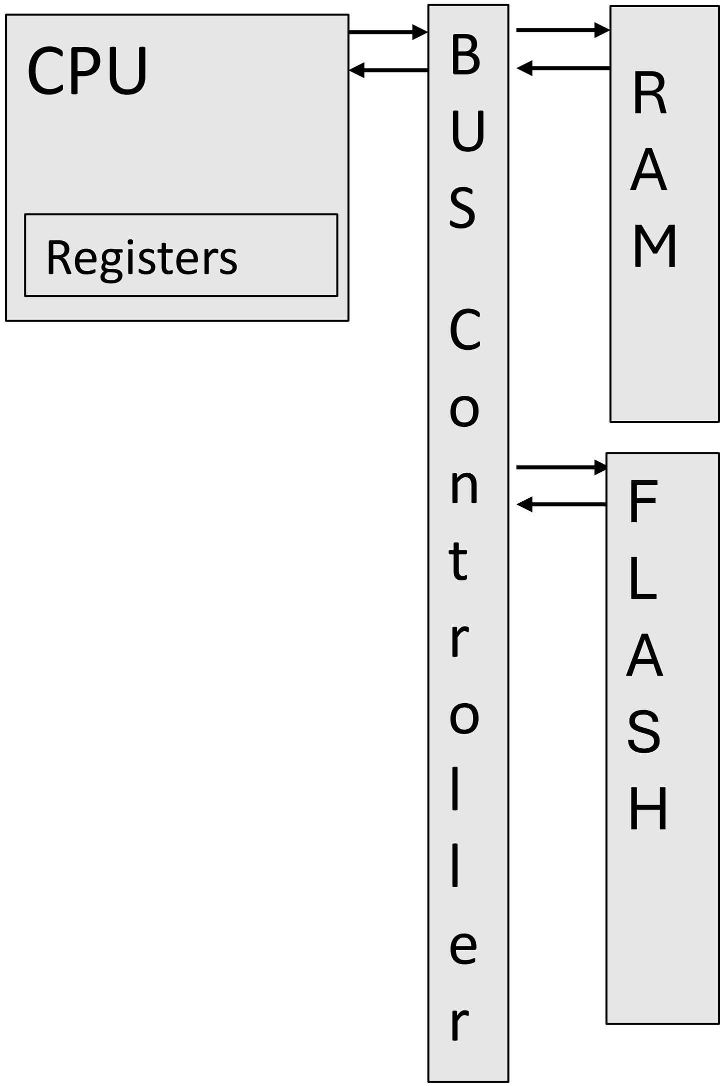


First, lets define a “simple computer”

Core components:

- CPU + Registers
- Bus controller
- Memory
  - Volatile – RAM
  - Non-volatile -- FLASH

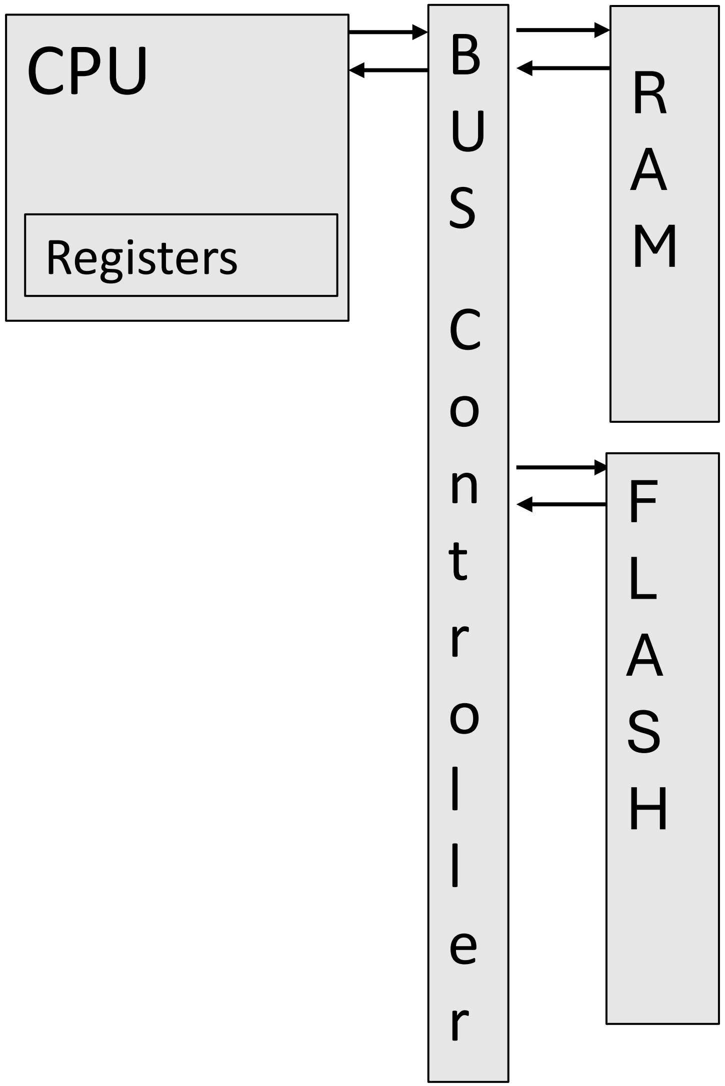
System Model



CPU

- Reads from Memory to Registers
- Writes from memory to Registers
- Registers are internal to CPU
- Manipulates registers for operations
  - R1 = R2 + R3
  - R6 = R7 xor R3
  - Etc…

System Model



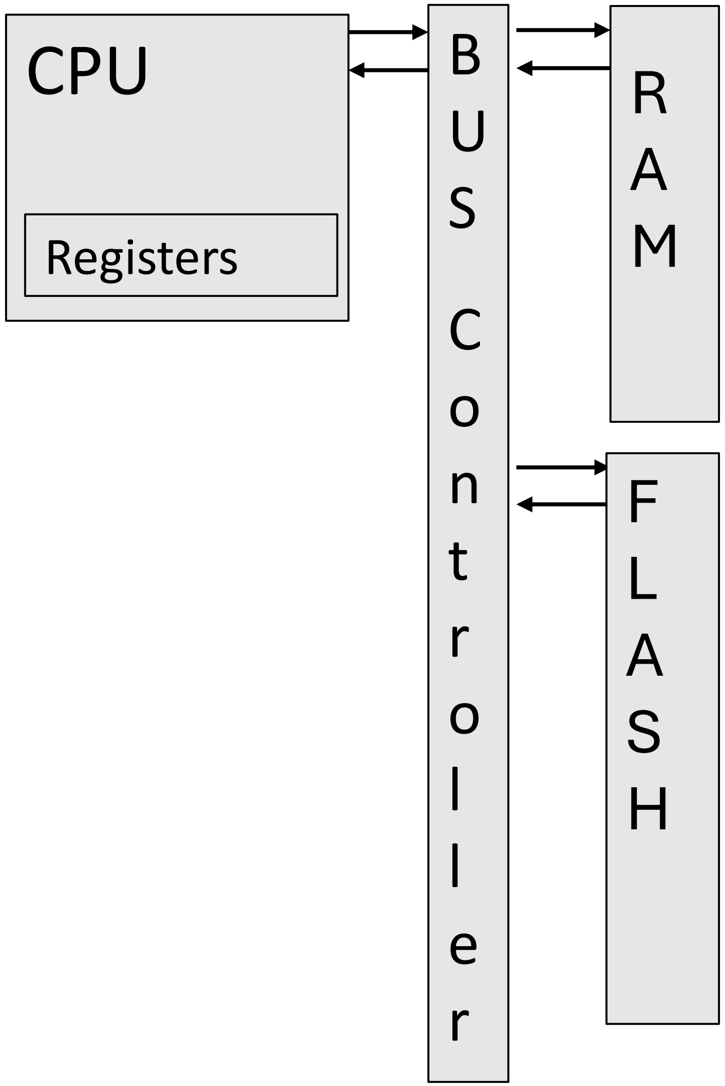
BUS Controller

- Access memory as instructed by CPU
- load 0x1234 R3
- store r5 0xe400
- etc…

Facilitate each read/write from/to memory location X

- X can be in either RAM or FLASH

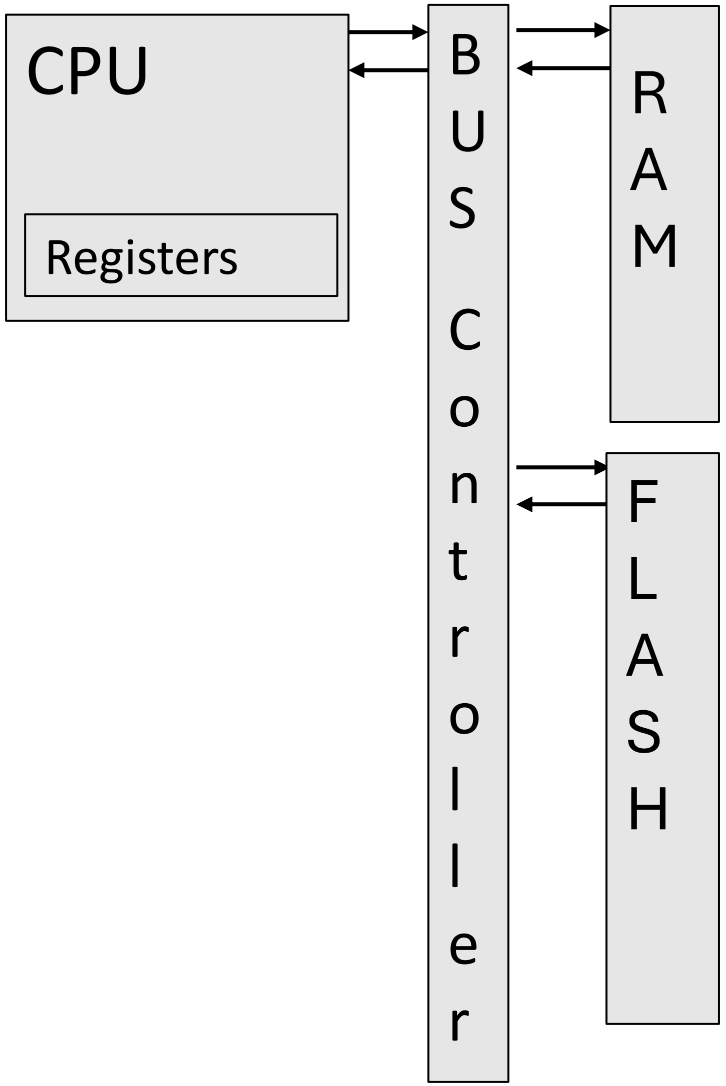
System Model



RAM

- Volatile (erased when powered off)
- Used to store intermediate computation results
- Aka **<u>data memory</u>**

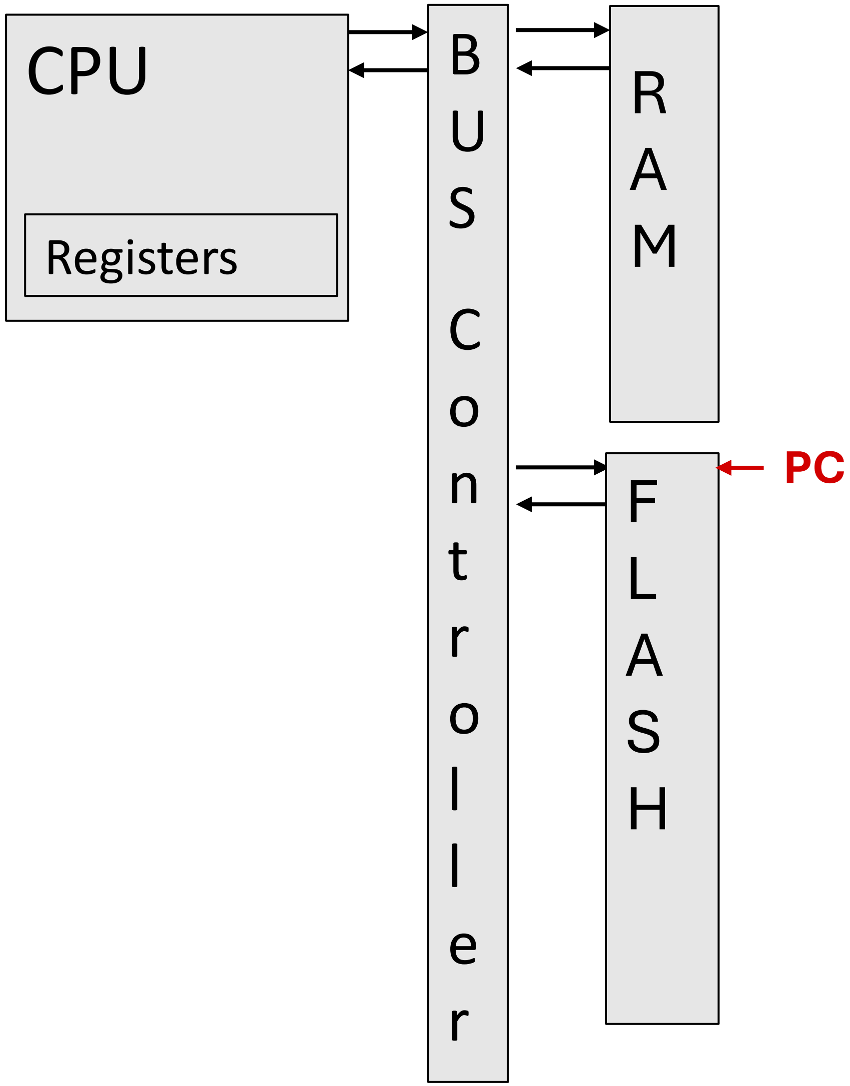
System Model



FLASH

- Non Volatile (persistent across power cycles)
- **<u>Program Memory</u>**
- Stores instructions

System Model



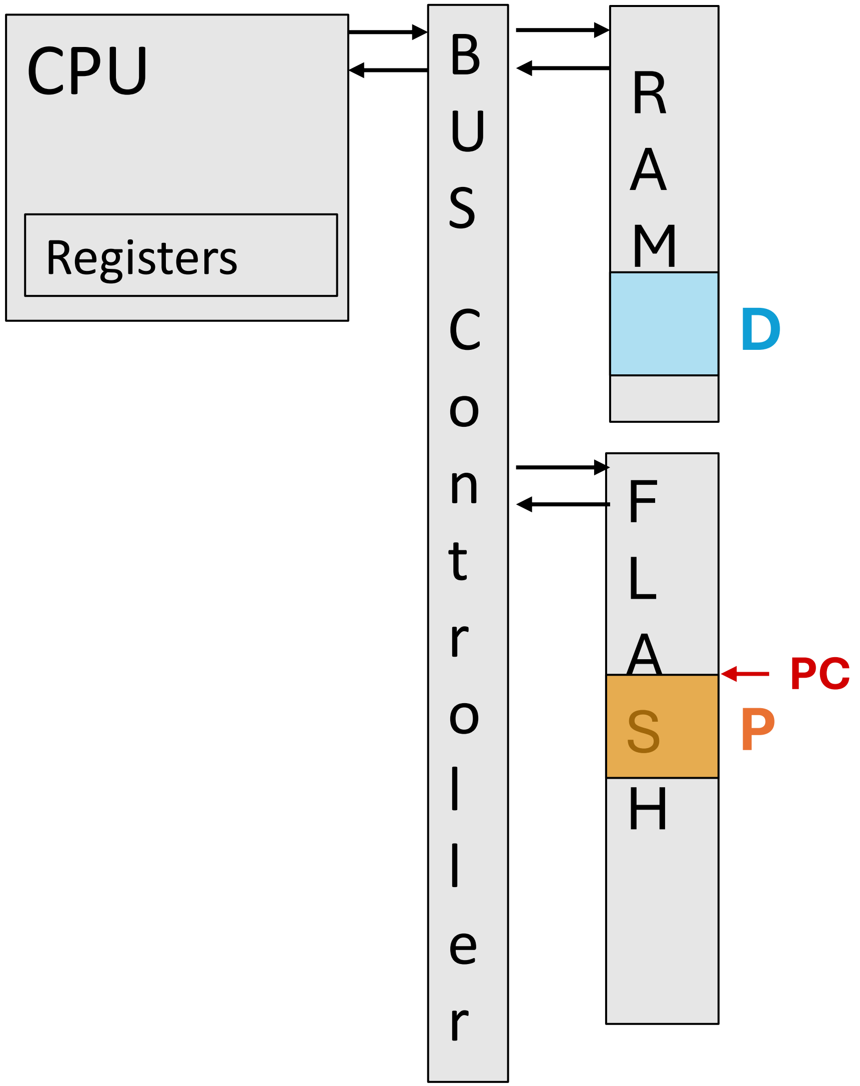
Execution

- At each given time (clock cycle), the CPU executes an instruction
- It executes the instruction stored in a special register
    - **<u>Program Counter (PC):</u>** points to address of the instruction that is currently executing

Instructions

- Load/store from memory
- Operate on registers
- <u>Mandate the next PC value</u>

System Model



Assumption:

- The system has already booted and initialized
- It is currently running program (P)
- It reads/writes from data (D)

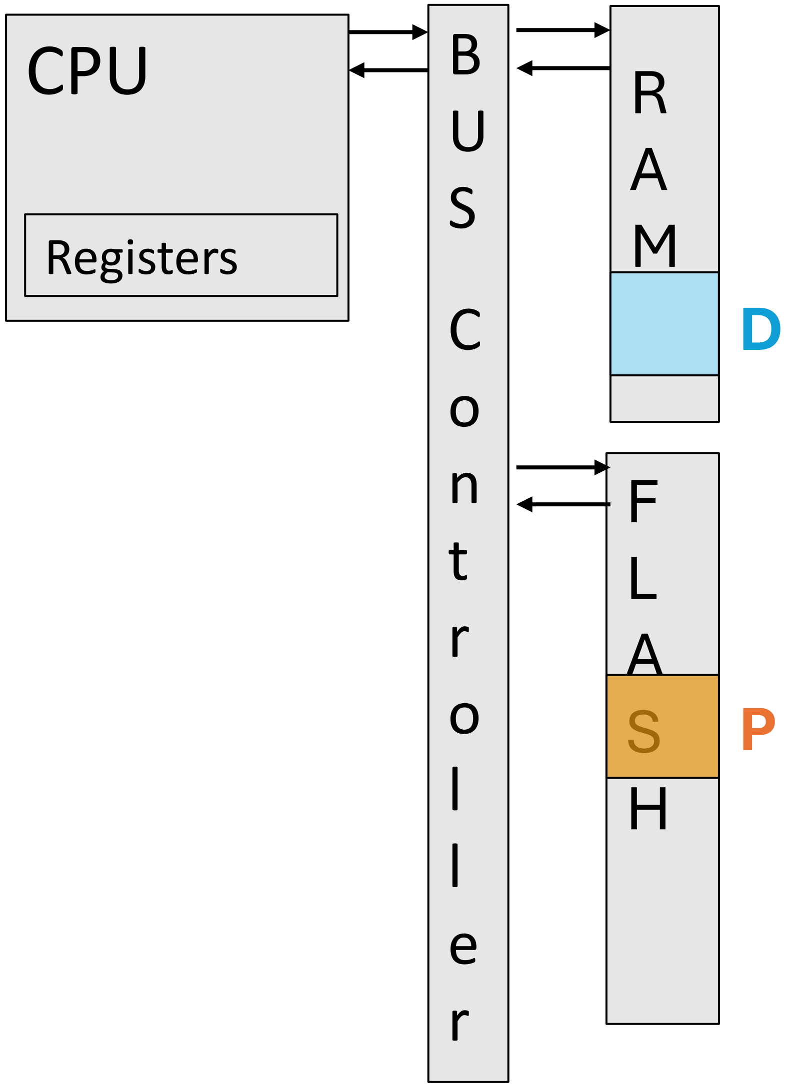
Malware: information leak or malicious execution

Where does an adversary start to attack P or D?

Memory Vulnerability

Steps taken by an adversary

1 **Memory Vuln.** 2 Integrity Violation 3 Exploit Payload 4 Exploit Dispatch 5 Execute Exploit

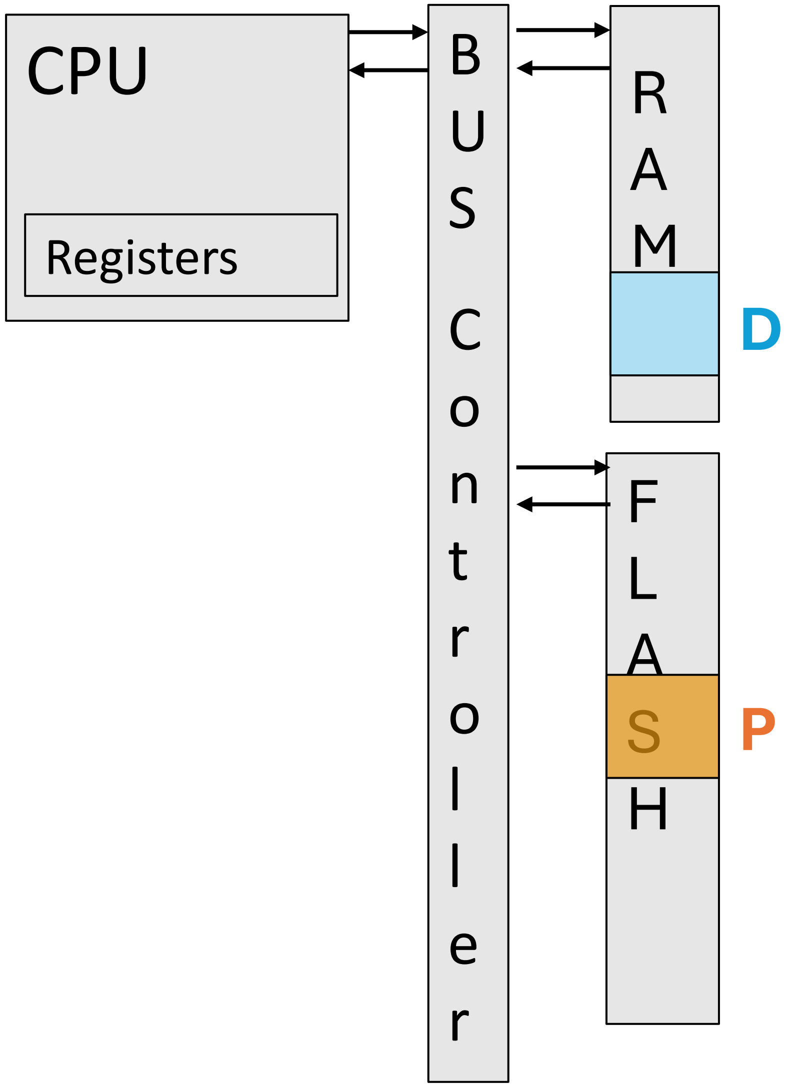


Step 1: Identify a Memory Vulnerability in P

- Out of bounds pointer
  - Buffer overflows (stack, heap)
  - Integer overflows
- Dangling pointer
  - Use-after-free
  - Double free
- Format string vulnerability

Steps taken by an adversary

1 **Memory Vuln.** 2 Integrity Violation 3 Exploit Payload 4 Exploit Dispatch 5 Execute Exploit



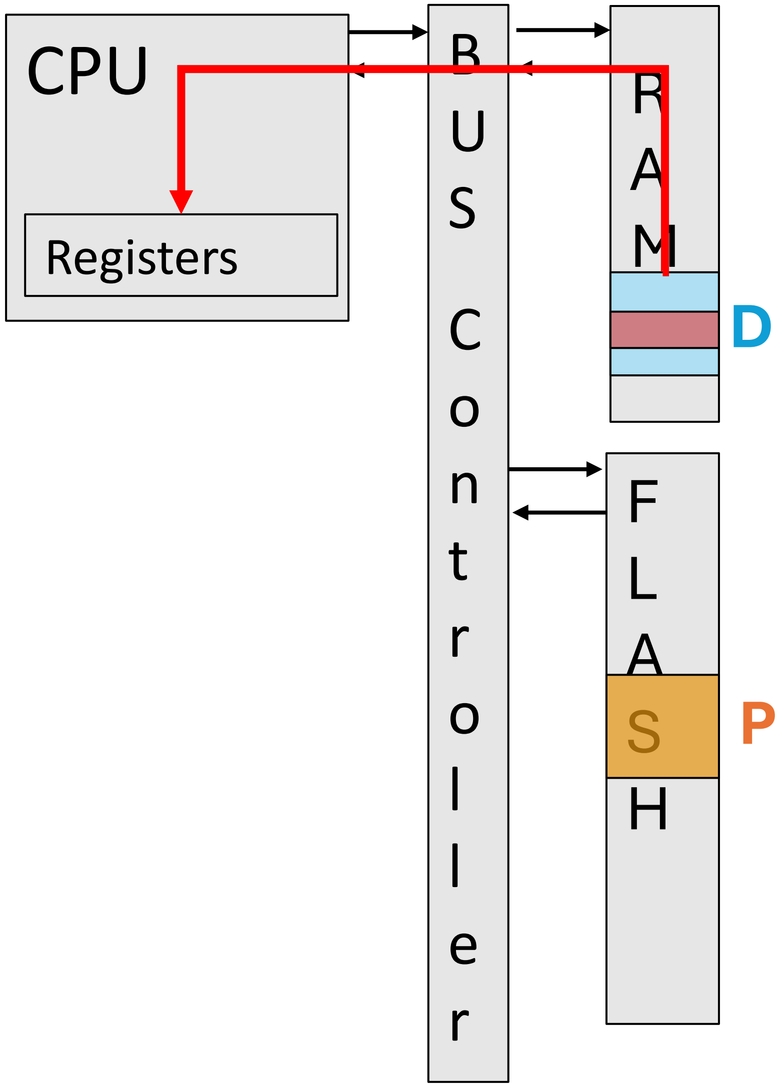
Step 1: Identify a Memory Vulnerability in P

Leads to a useful gadget for our adversary...

- Unintended read
- Unintended write

Steps taken by an adversary

1 Memory Vuln. **2 Integrity Violation** 3 Exploit Payload 4 Exploit Dispatch 5 Execute Exploit

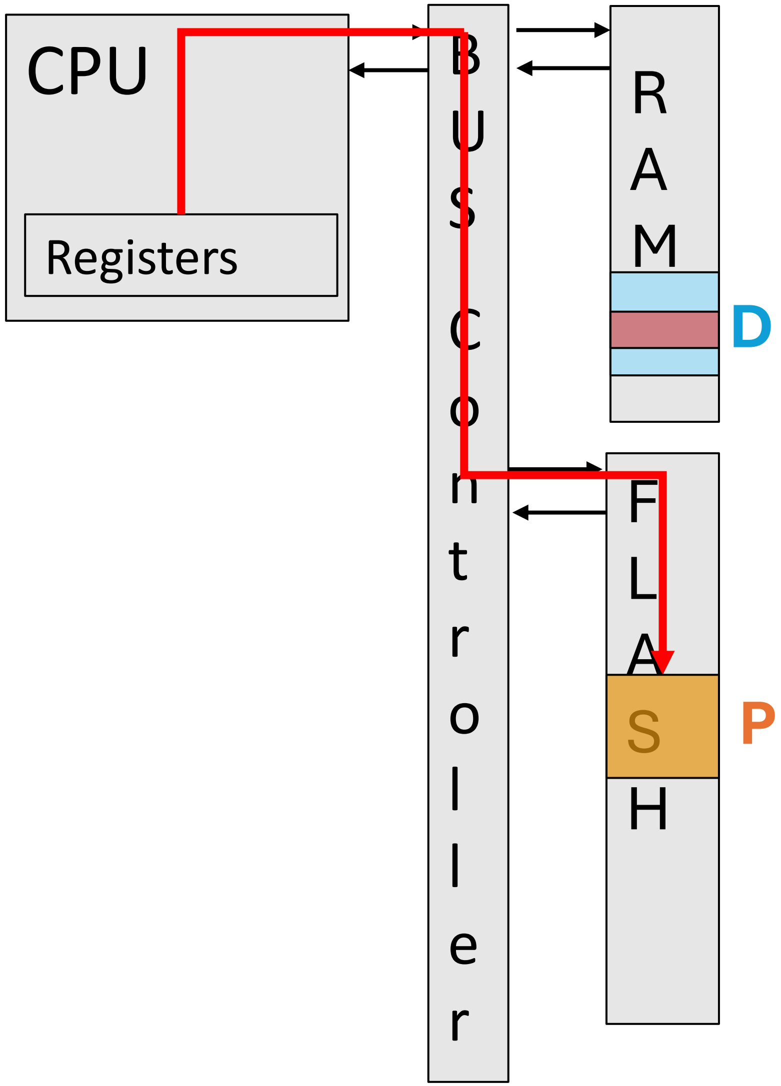


Step 2: Cause an Integrity Violation

- Unintended read:
  - Exfiltrates data from memory to registers

Steps taken by an adversary

1 Memory Vuln. 2 Integrity Violation **3 Exploit Payload** 4 Exploit Dispatch 5 Execute Exploit



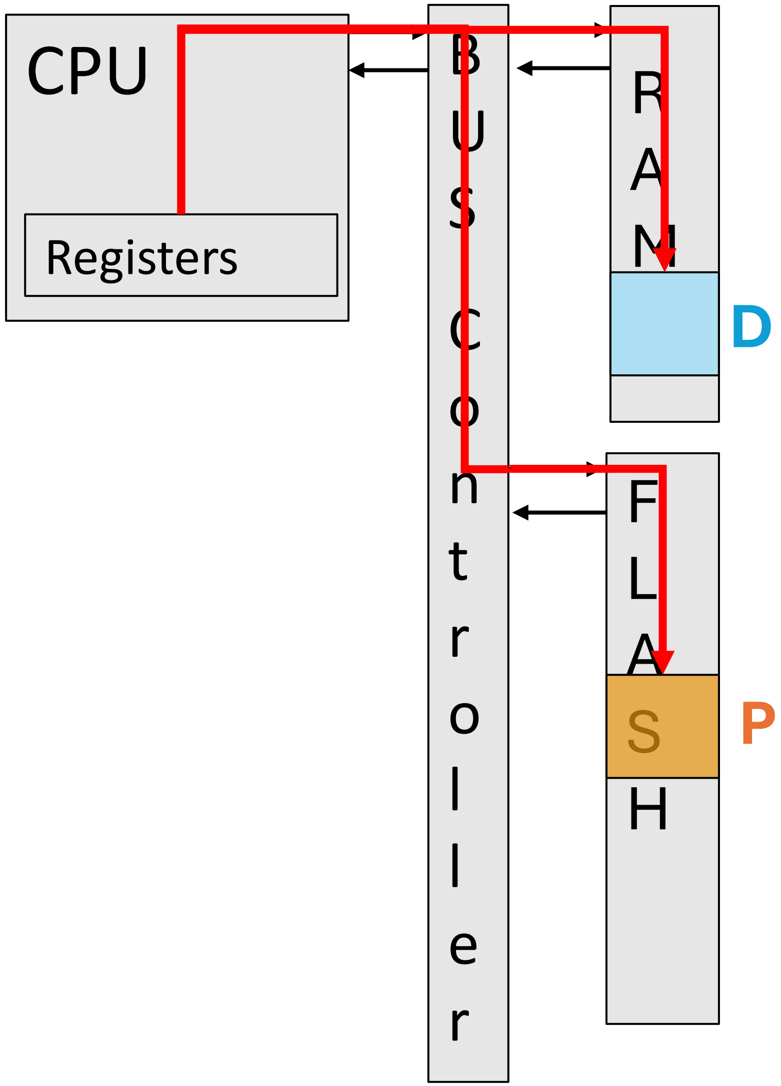
Step 3: Construct Payload

Unintended Read:

- Interpret/send the exfiltrated data
- Done! **Information Leakage**

Steps taken by an adversary

1 Memory Vuln. 2 **Integrity Violation** 3 Exploit Payload 4 Exploit Dispatch 5 Execute Exploit



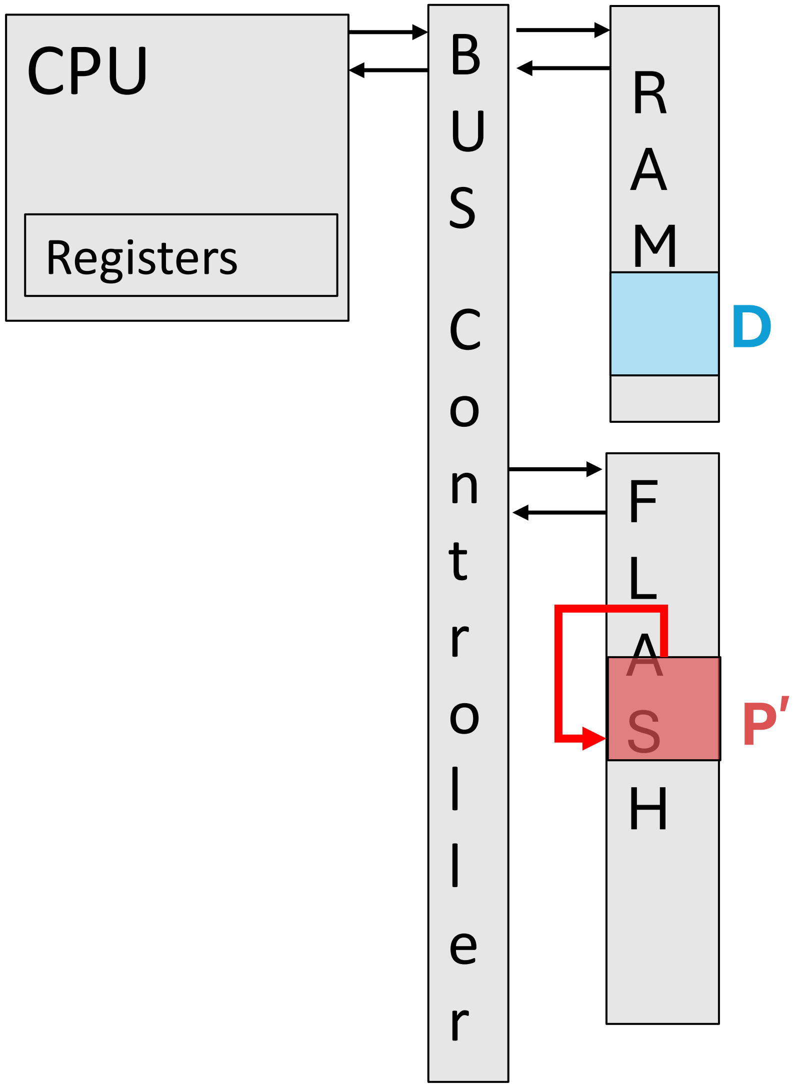
Step 2: Cause an Integrity Violation

Unintended write:

- Modifies code or data
- Enables malicious execution

Steps taken by an adversary

1 Memory Vuln. 2 **Integrity Violation** 3 Exploit Payload 4 Exploit Dispatch 5 Execute Exploit



Step 2: Cause an Integrity Violation

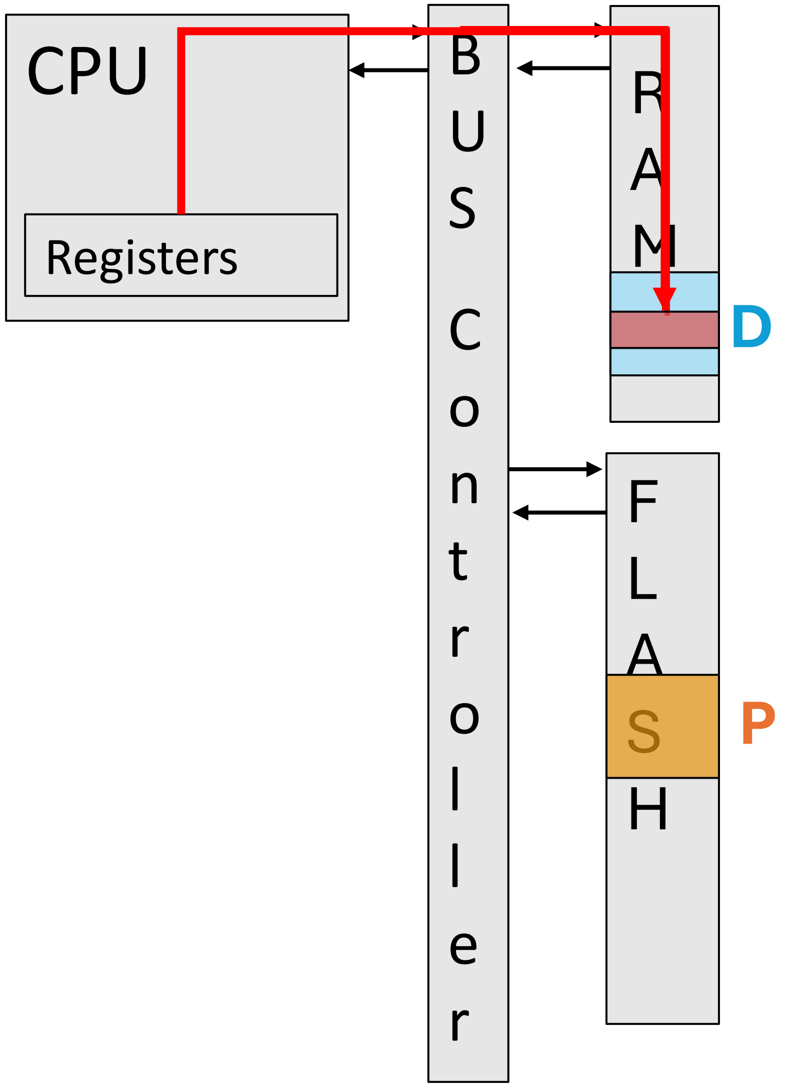
Unintended Writes that enable malicious execution

Enable modification of...

- The program itself

Steps taken by an adversary

1 Memory Vuln. **2 Integrity Violation** 3 Exploit Payload 4 Exploit Dispatch 5 Execute Exploit



Step 2: Cause an Integrity Violation

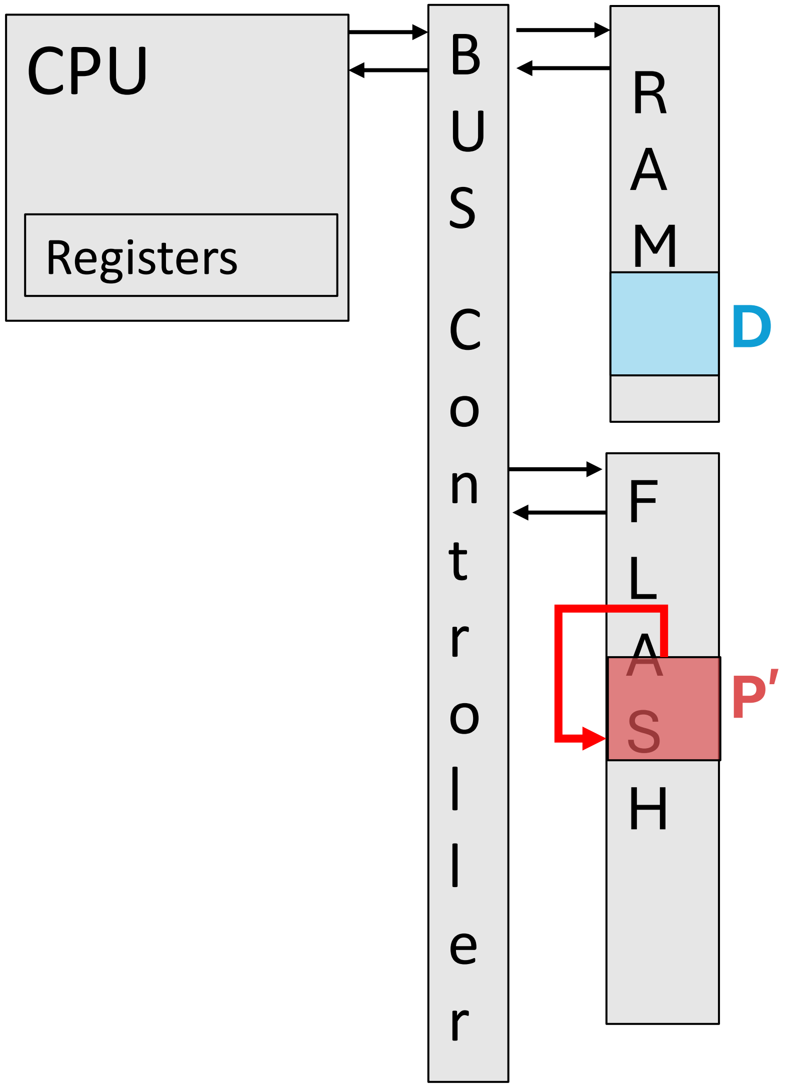
Unintended Writes that enable malicious execution

Enable modification of…

- The program itself
- The program's ***control data***
    - Return address, function pointer
- The program's ***non-control data***
    - Data that affects the execution path
- The available integrity violation itself is used for the ***payload***

Steps taken by an adversary

1 Memory Vuln. 2 Integrity Violation **3 Exploit Payload** 4 Exploit Dispatch 5 Execute Exploit



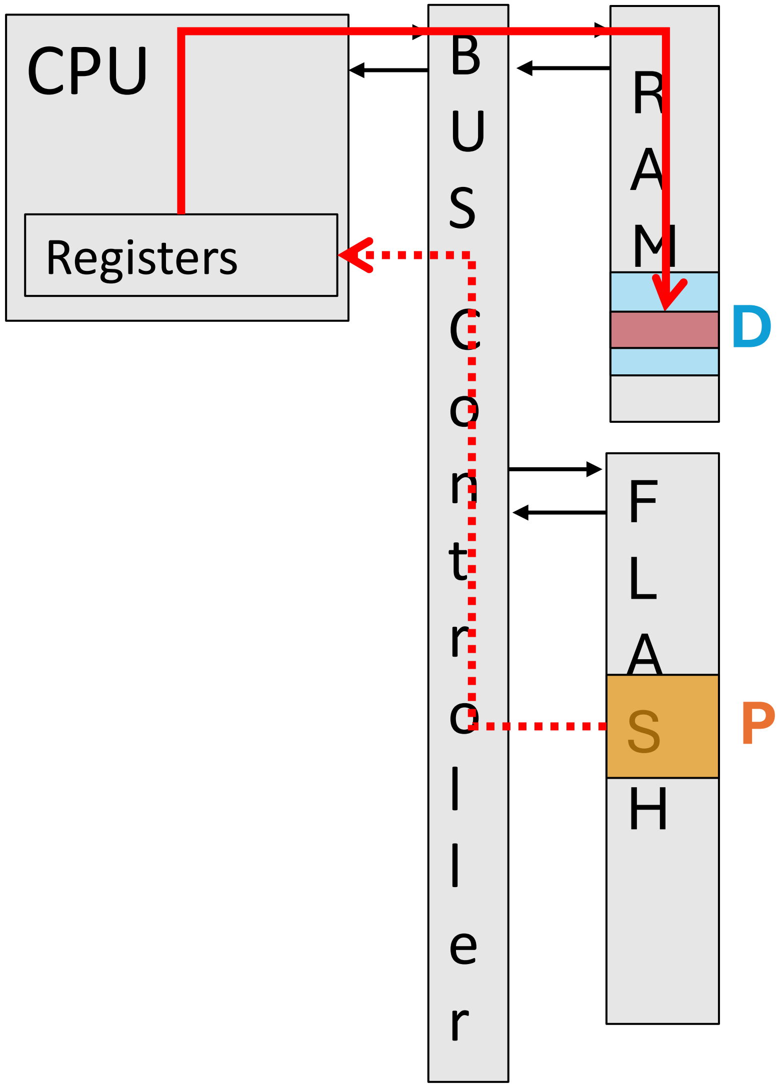
Step 3: Construct Payload

Unintended Writes that enable malicious execution

- Modify program (fragment) → inject attacker-controlled code

Steps taken by an adversary

1 Memory Vuln. 2 Integrity Violation **3 Exploit Payload** 4 Exploit Dispatch 5 Execute Exploit



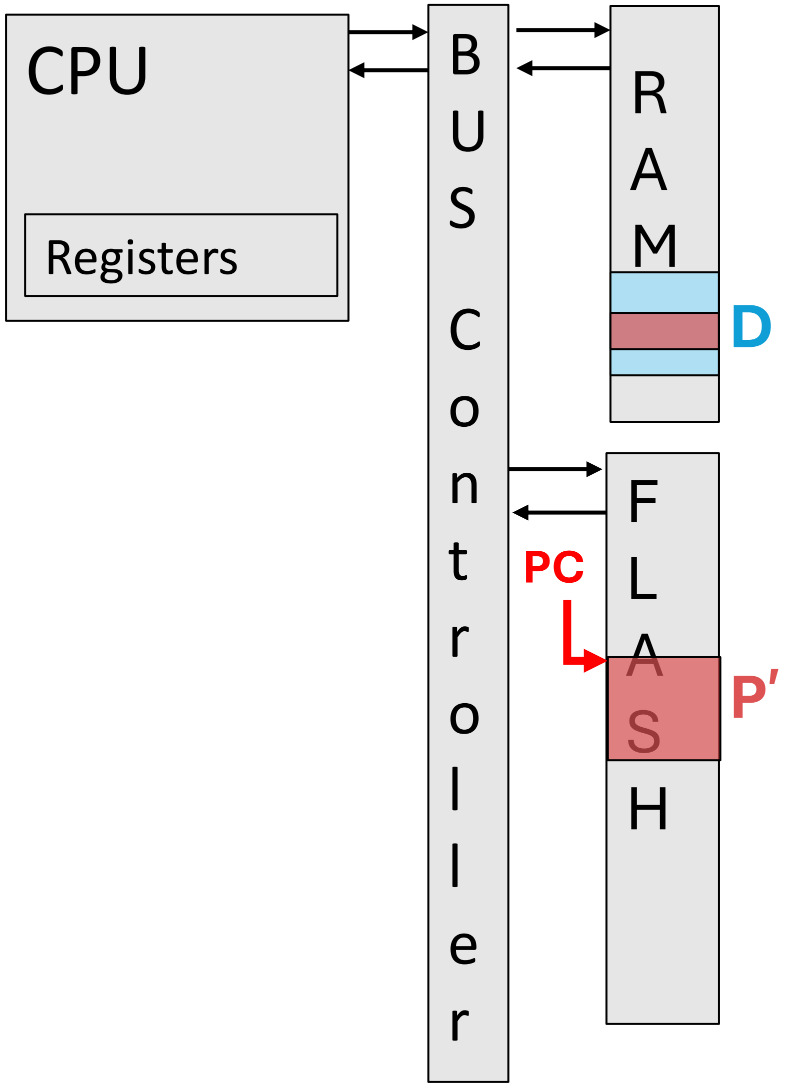
Step 3: Construct Payload

Unintended Writes that enable malicious execution

- Modify program (fragment) → inject attacker-controlled code
- Modify ***control data*** → inject attacker-controlled address as…
  - Return address
  - Function pointer
- Modify ***non-control data*** → inject attacker-controlled data into
  - Data that affects the execution path
    - Variables used in if-else blocks, switch statements, loops

Steps taken by an adversary

1 Memory Vuln. 2 Integrity Violation 3 Exploit Payload **4 Exploit Dispatch** 5 Execute Exploit



Step 4: Dispatch the exploit

Unintended Writes that enable malicious execution

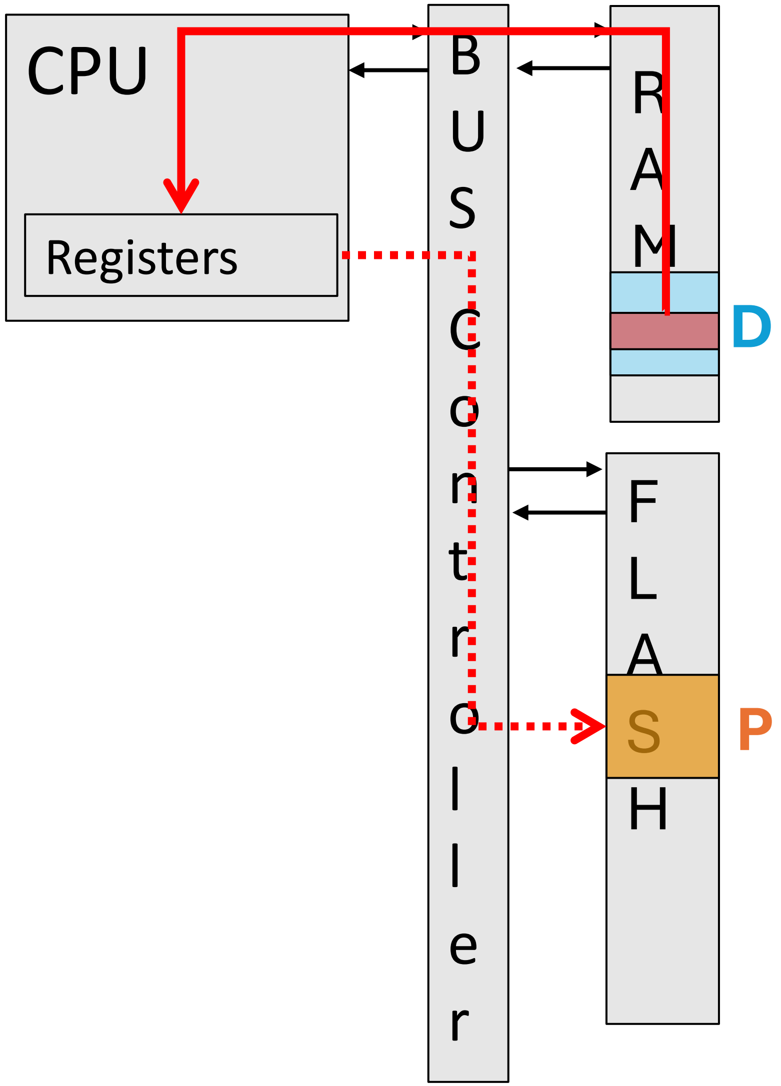
Abuse benign program behavior that operates on adversary-controlled input

Call/jump/return to

- The modified program (fragment)

Steps taken by an adversary

1 Memory Vuln. 2 Integrity Violation 3 Exploit Payload 4 **Exploit Dispatch** 5 Execute Exploit



Step 4: Dispatch the exploit

Unintended Writes that enable malicious execution

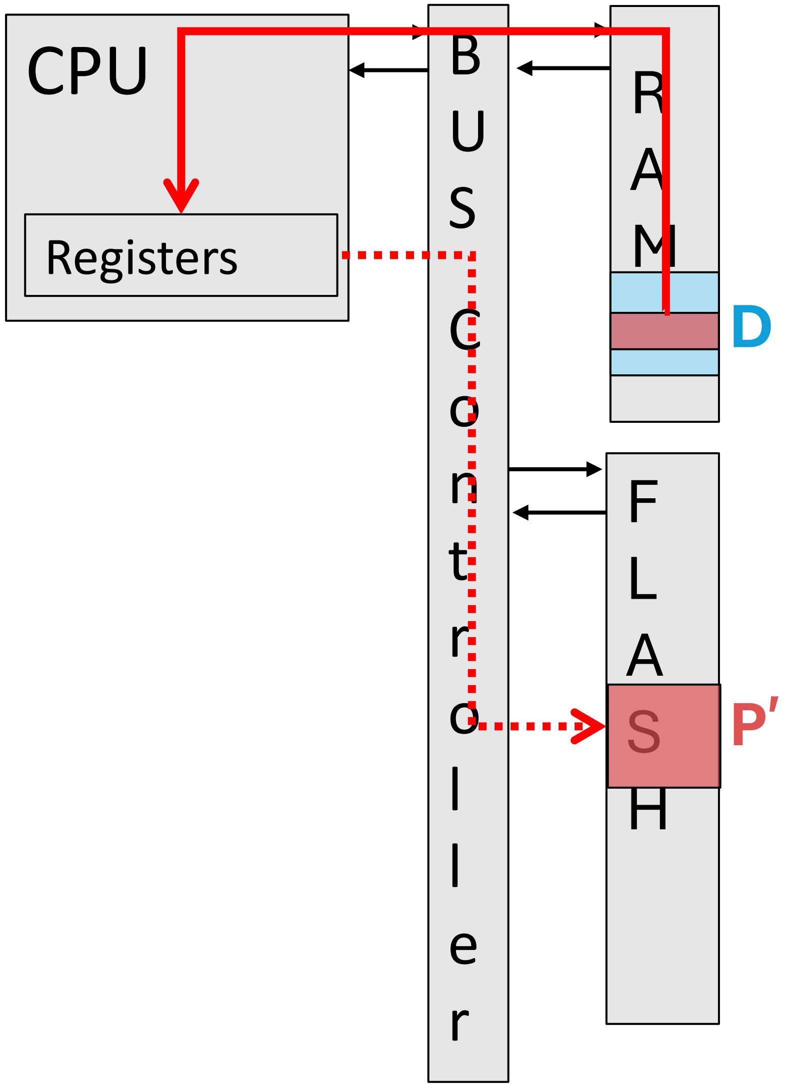
Abuse benign program behavior that operates on adversary-controlled input

Call/jump/return to

- The modified program (fragment)
- The gadget at the adv-controlled address (***control data***)
- The gadget using the ***non-control data***

Steps taken by an adversary

1 Memory Vuln. 2 Integrity Violation 3 Exploit Payload 4 **Exploit Dispatch** 5 Execute Exploit



Step 4: Dispatch the exploit

Unintended Writes that enable malicious execution

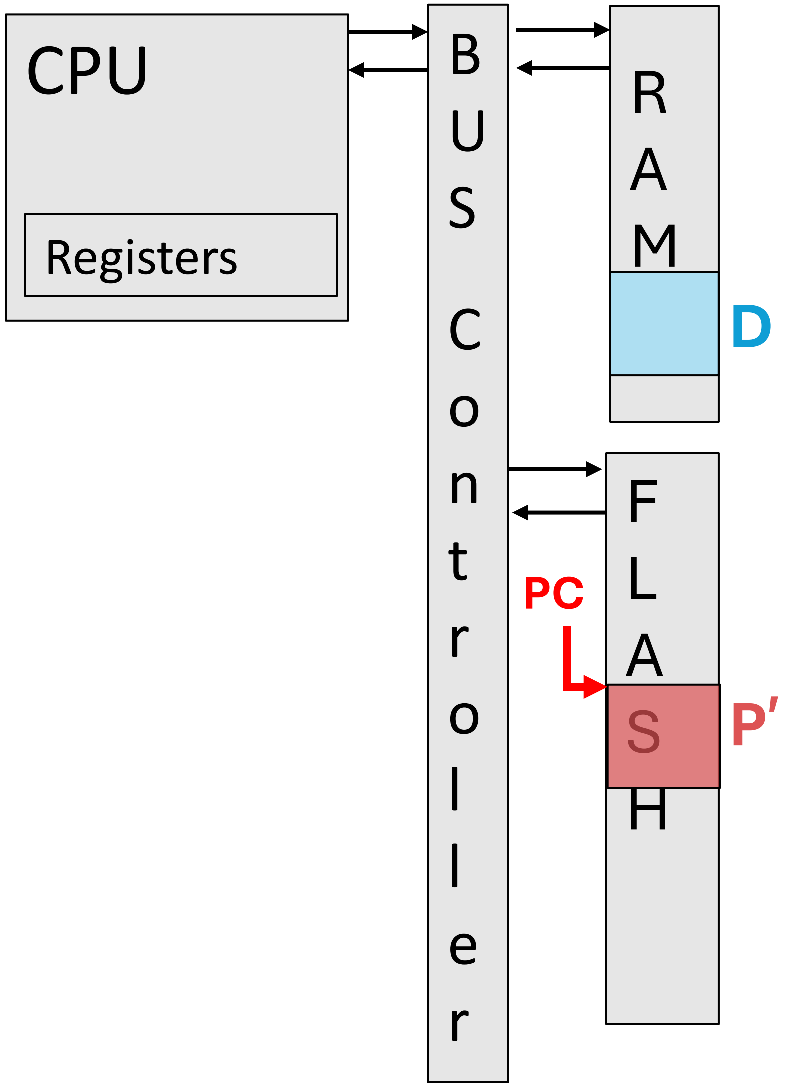
Abuse benign program behavior that operates on adversary-controlled input

Call/jump/return to

- The modified program (fragment)
- The gadget at the adv-controlled address (***control data***)
- The gadget using the ***non-control data***

Steps taken by an adversary

1 Memory Vuln. 2 Integrity Violation 3 Exploit Payload 4 Exploit Dispatch **5 Execute Exploit**



Step 5: Execute the exploit

At this point, the attacker has compromised the program

Achieved **malicious execution**

- Either through their own inserted instructions
- Through stitching of unmodified instructions

Steps taken by an adversary

1. Memory Vulnerability
2. Integrity Violation
3. Exploit Payload
4. Exploit Dispatch
5. Exploit Execution
6. Attack

Steps taken by an adversary

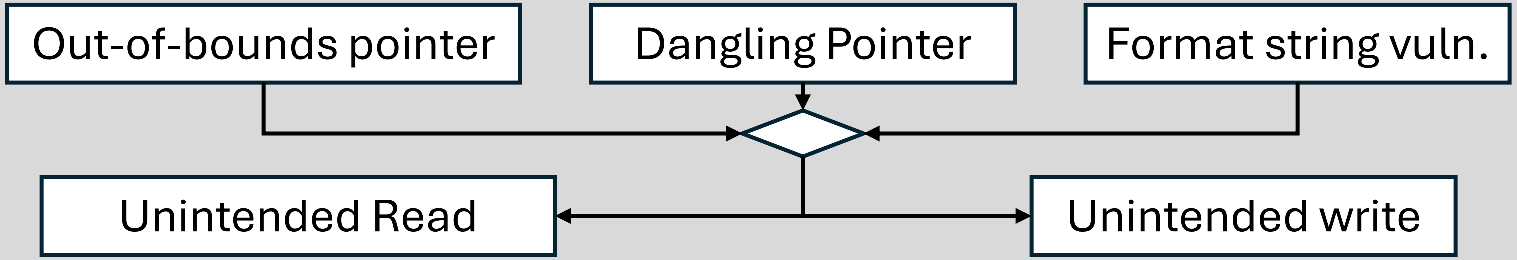
1. Memory Vulnerability
2. Integrity Violation
3. Exploit Payload
4. Exploit Dispatch
5. Exploit Execution
6. Attack

Information leak

Malicious execution

Steps taken by an adversary

1 **Memory Vulnerability**



2 Integrity Violation

3 Exploit Payload

4 Exploit Dispatch

5 Exploit Execution

6 Attack

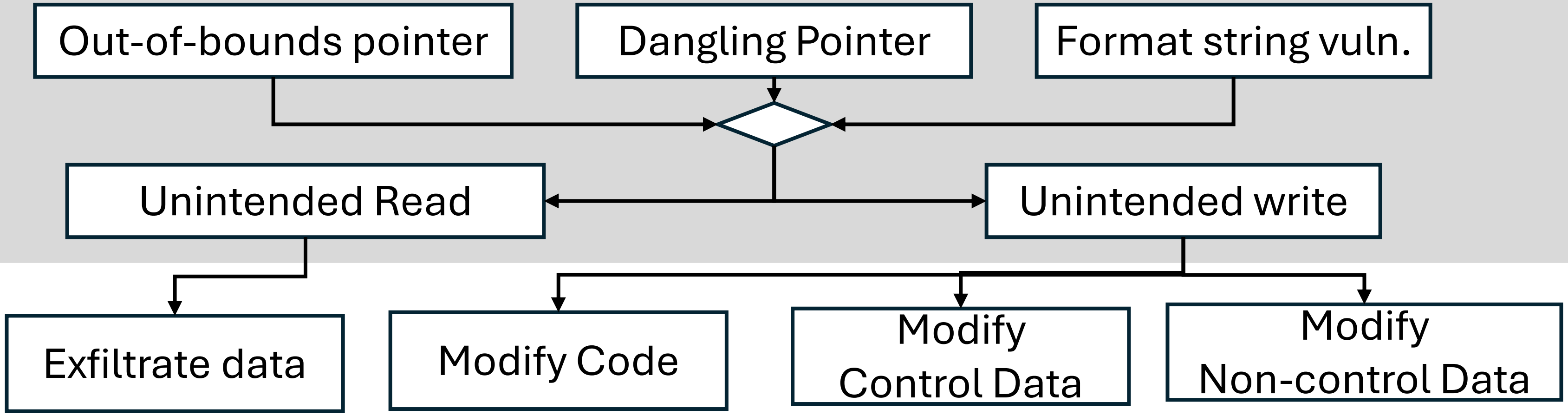
Information leak

Malicious execution

Steps taken by an adversary

1 Memory Vulnerability

2 **Integrity Violation**



3 Exploit Payload

4 Exploit Dispatch

5 Exploit Execution

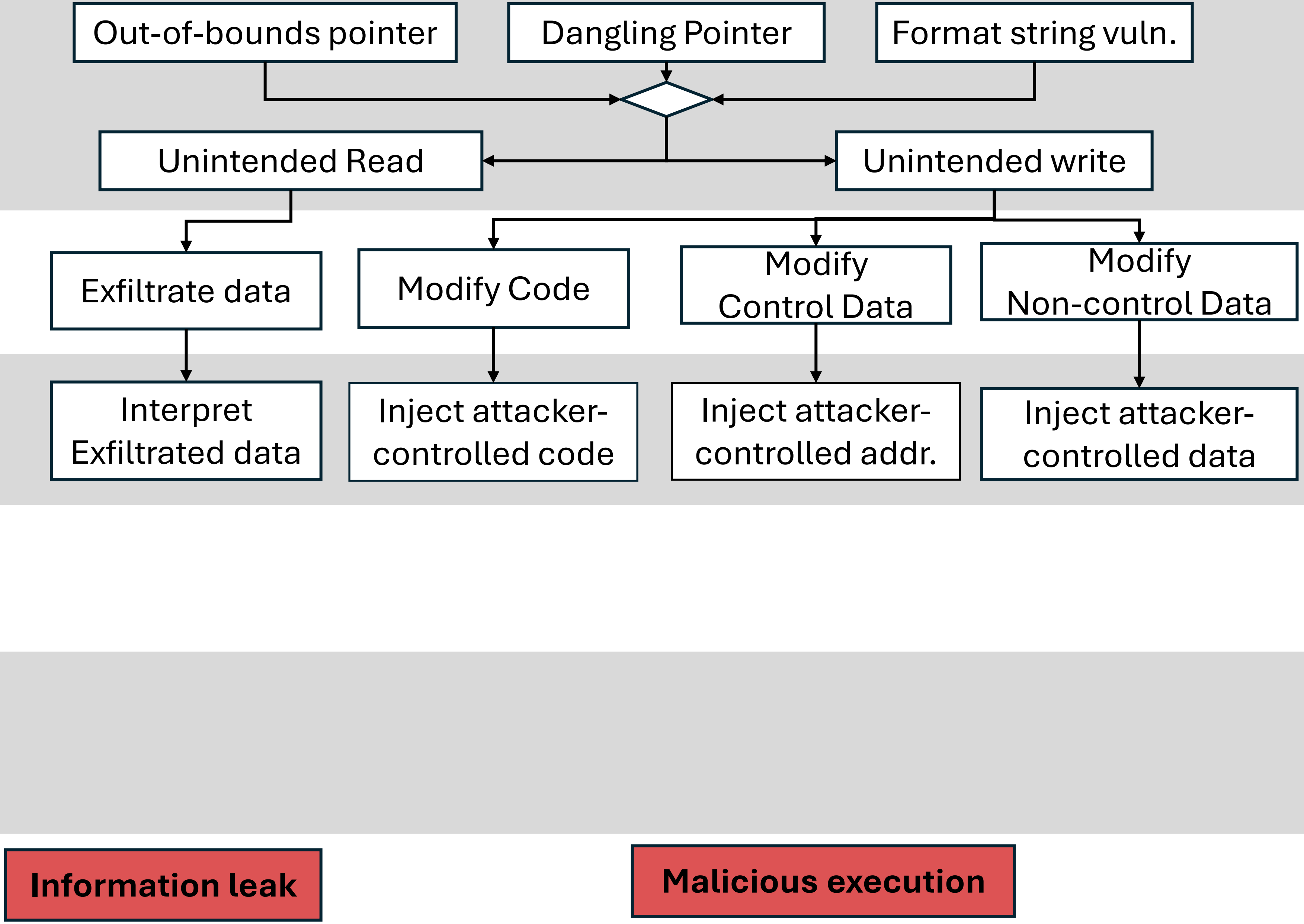
6 Attack

Information leak

Malicious execution

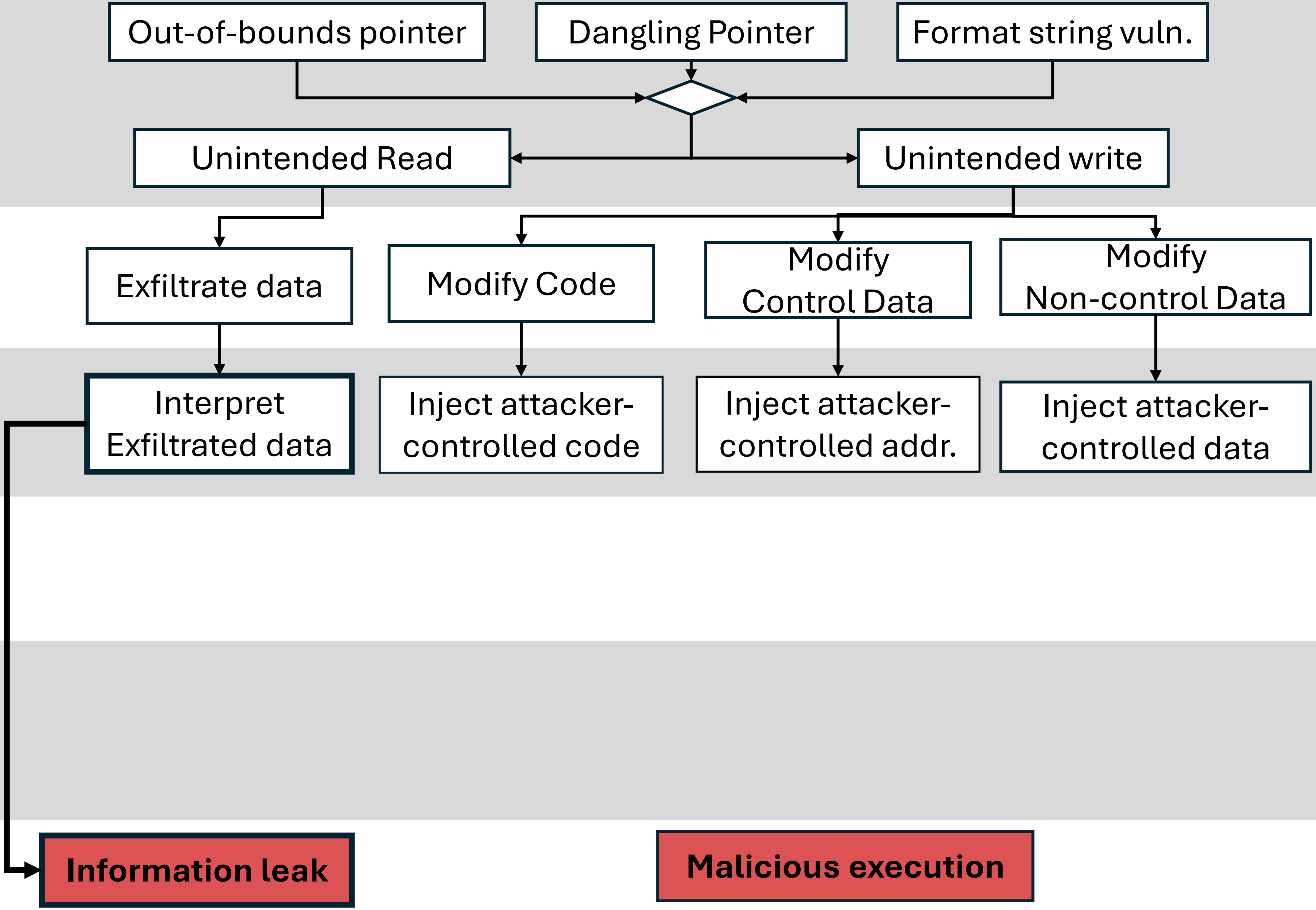
Steps taken by an adversary

1. Memory Vulnerability
2. Integrity Violation
3. **Exploit Payload**
4. Exploit Dispatch
5. Exploit Execution
6. Attack

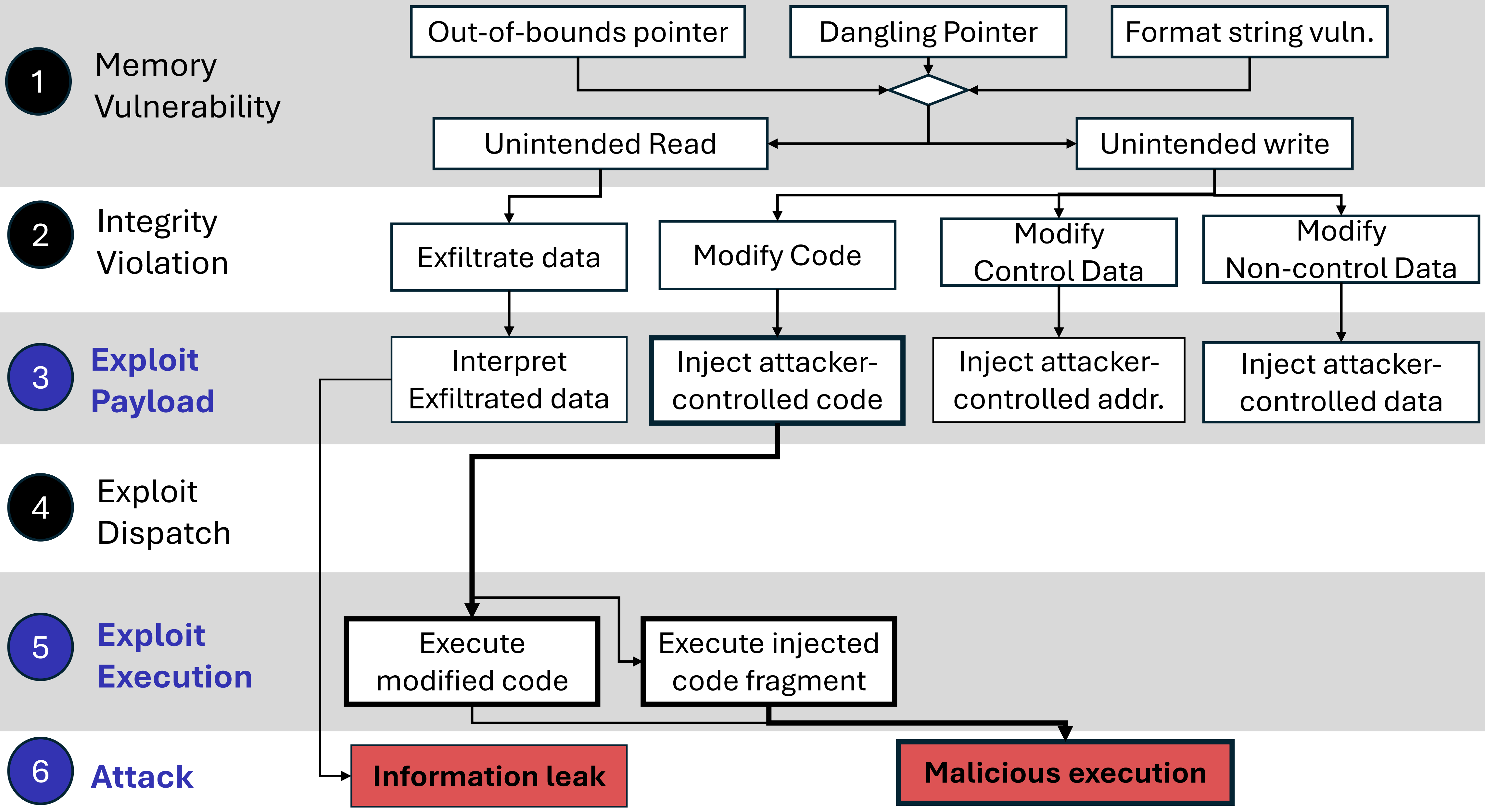


Steps taken by an adversary

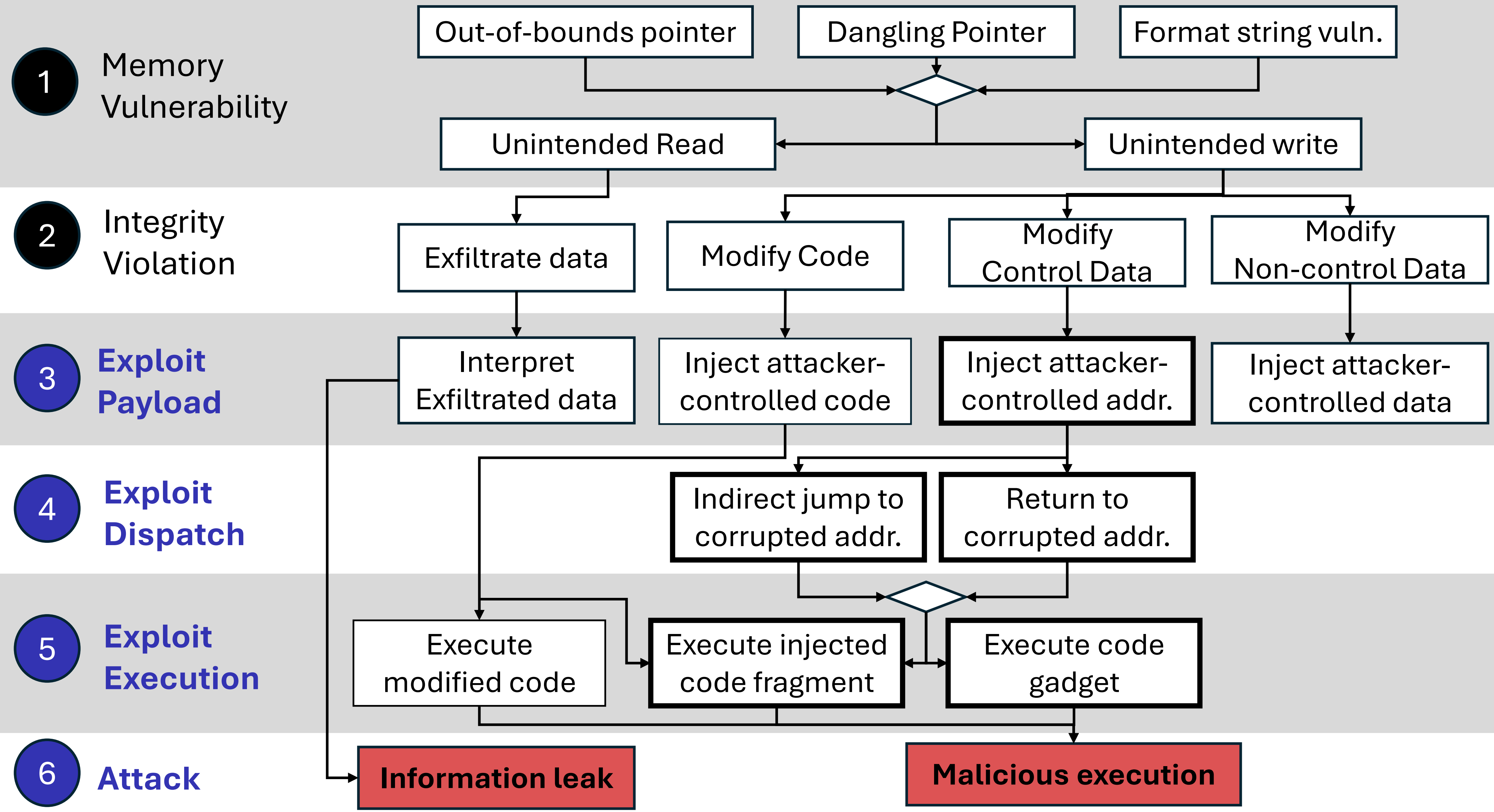
1 Memory Vulnerability
2 Integrity Violation
3 **Exploit Payload**
4 Exploit Dispatch
5 Exploit Execution
6 **Attack**



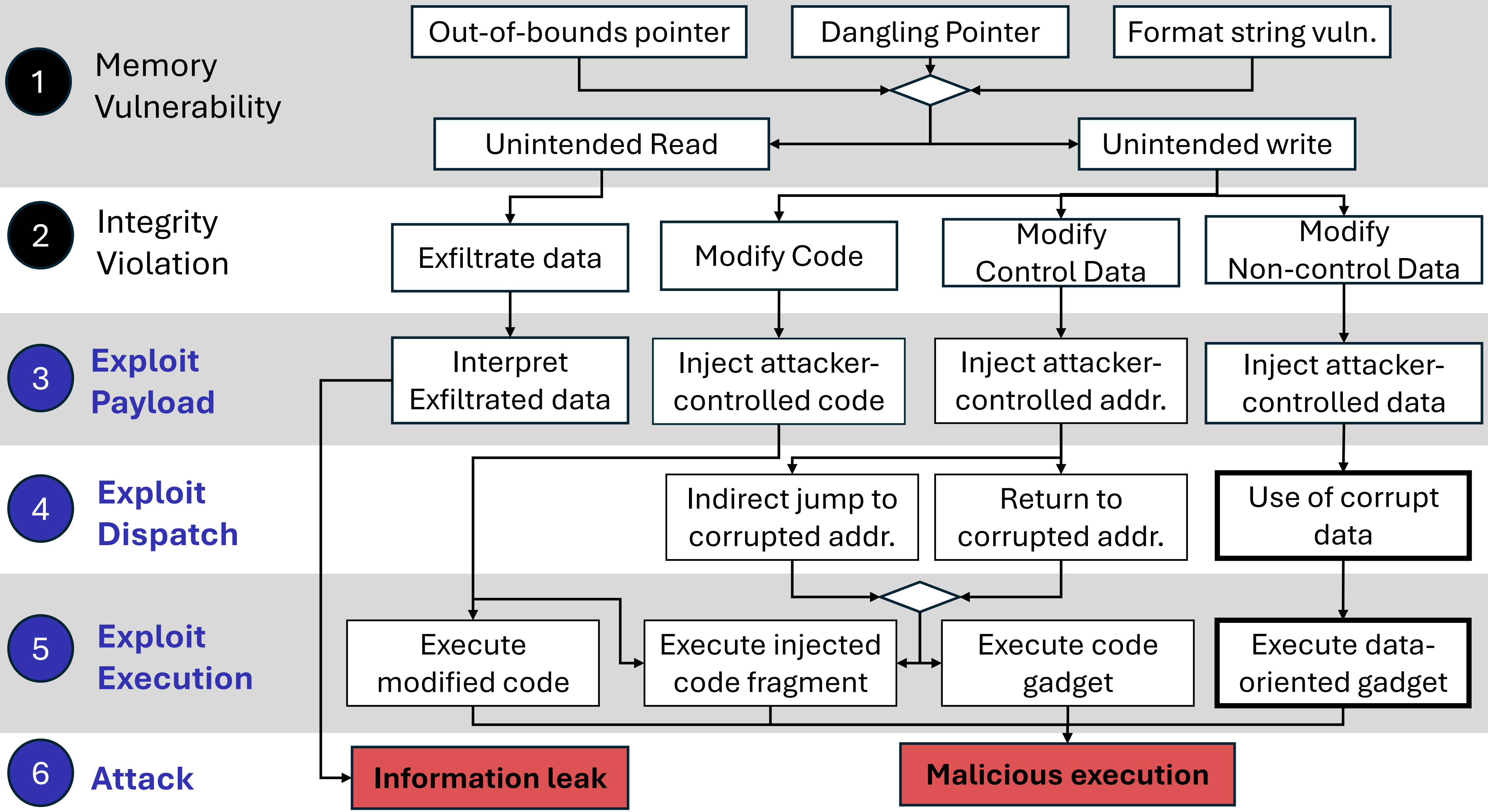
Steps taken by an adversary



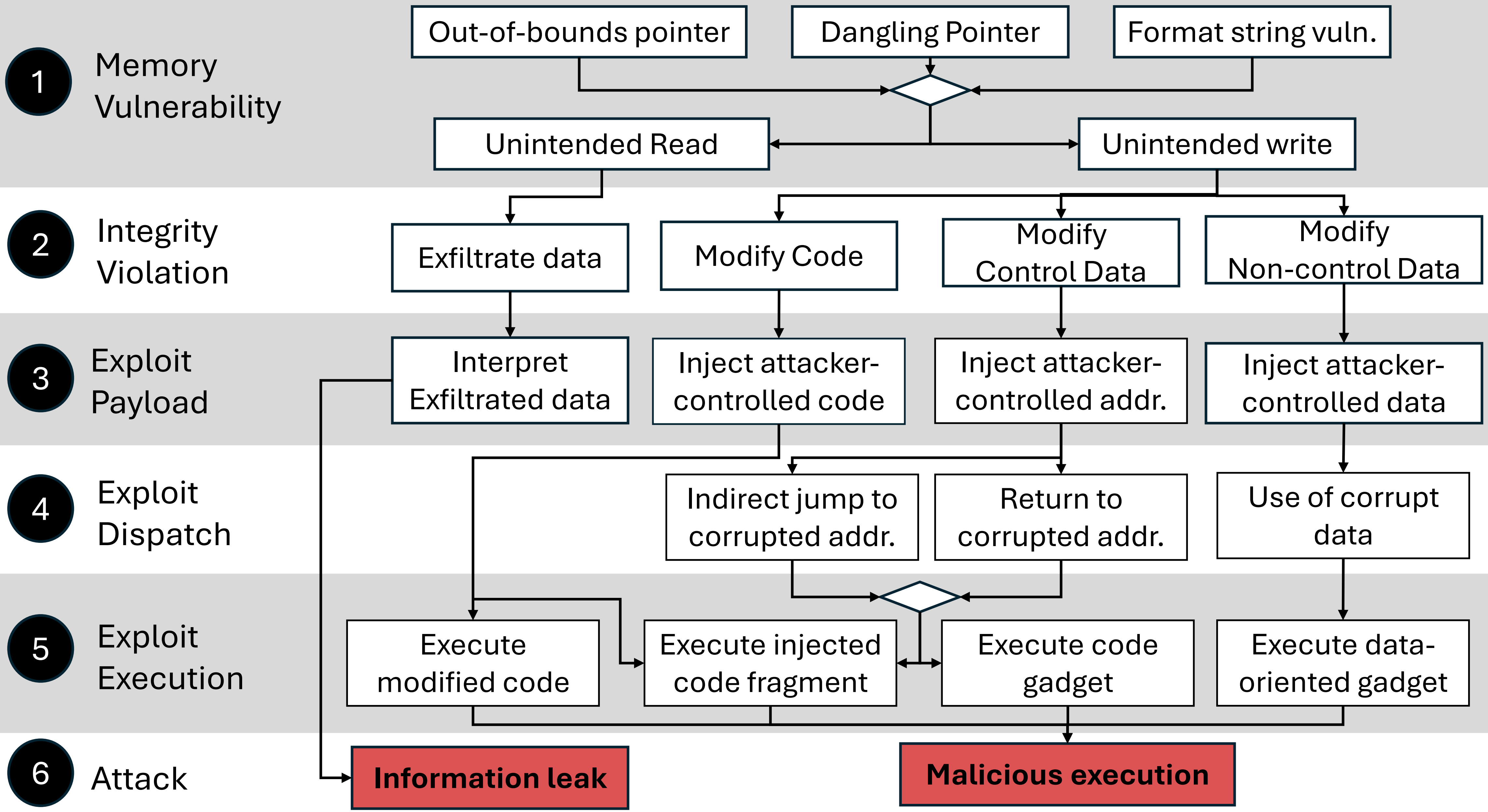
Steps taken by an adversary



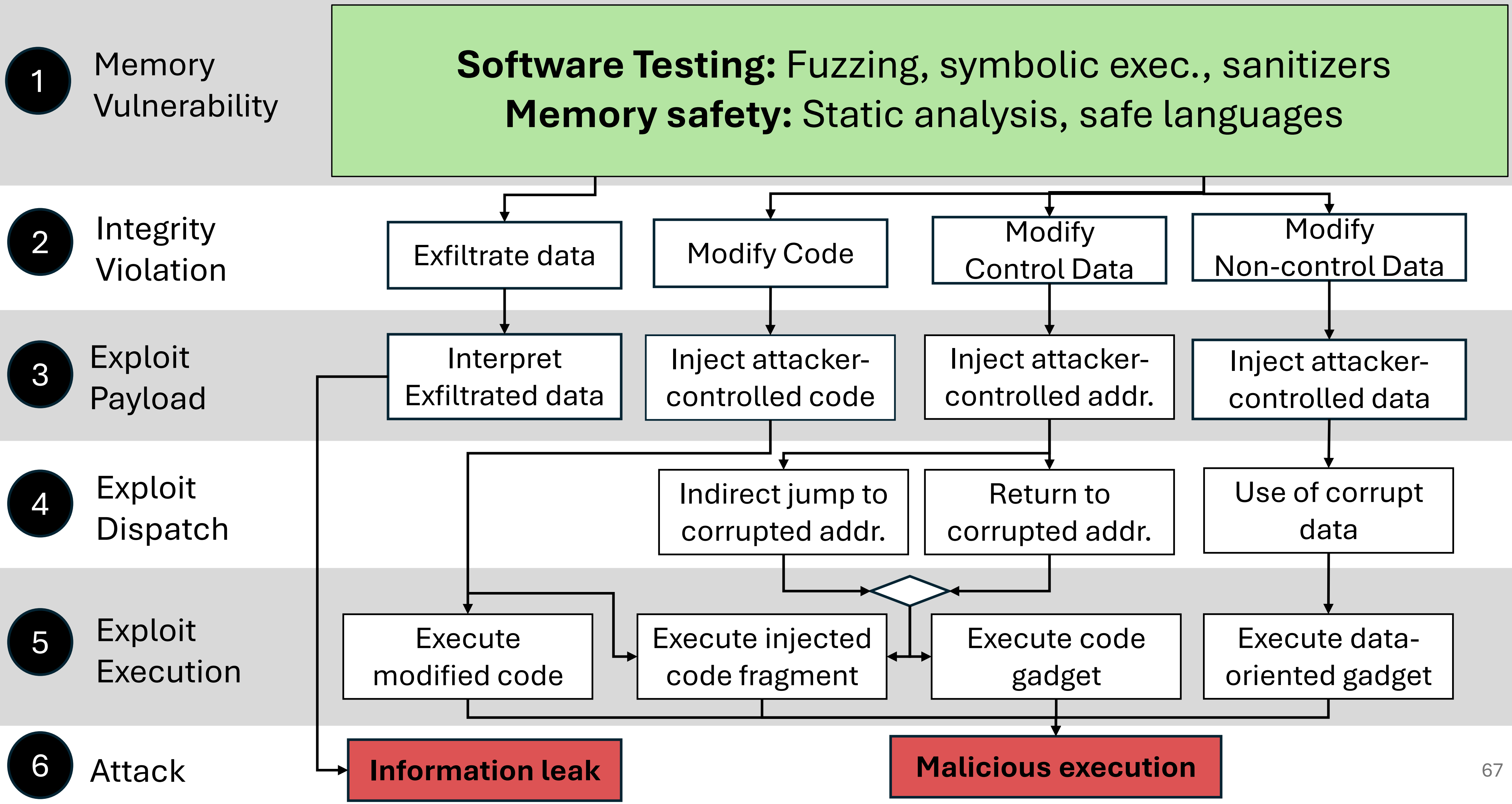
Steps taken by an adversary



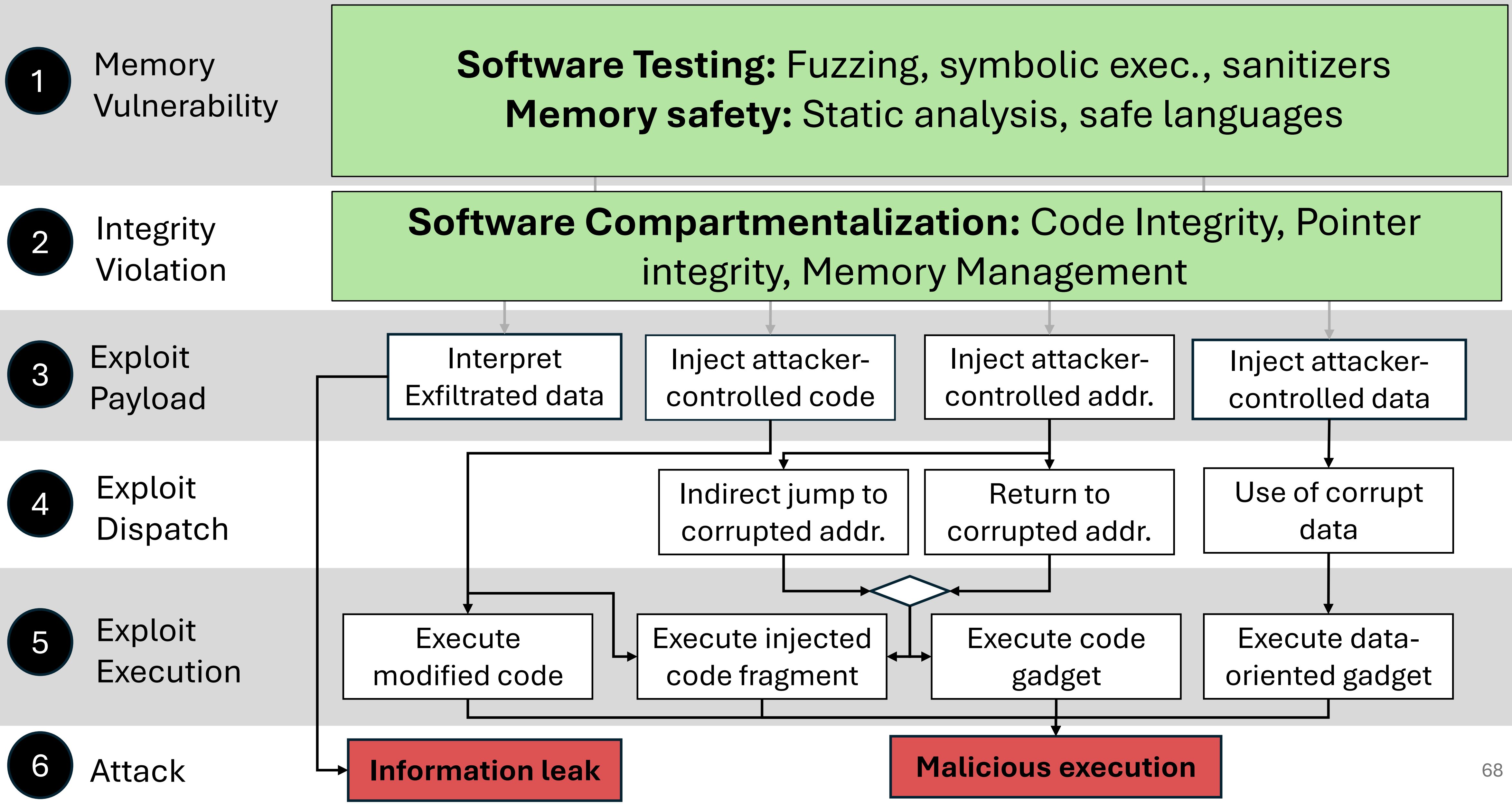
Steps taken by an adversary



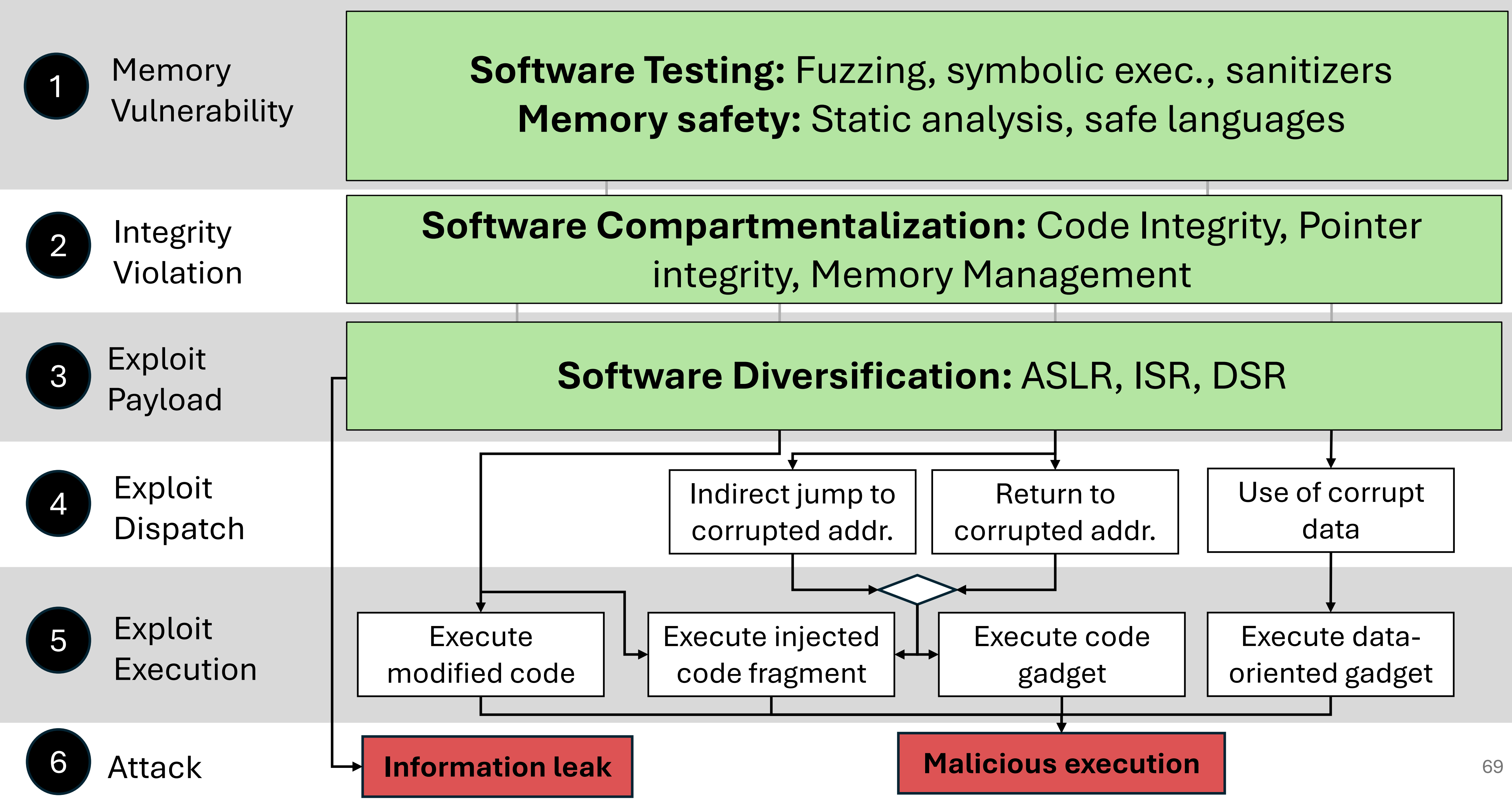
Steps taken by an adversary



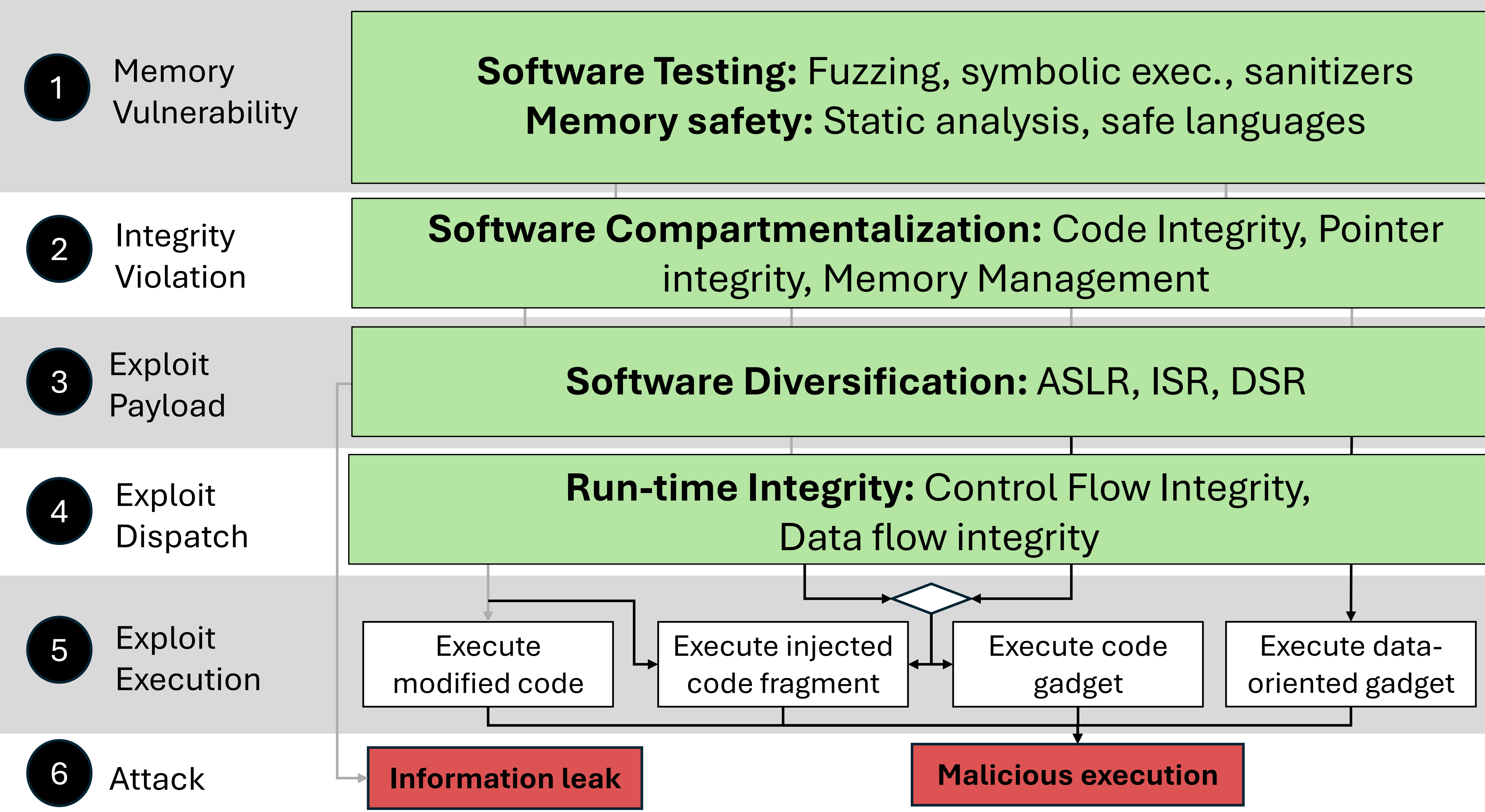
Steps taken by an adversary



Steps taken by an adversary



Steps taken by an adversary



Steps taken by an adversary

Step	Adversary step	Defense
1	Memory Vulnerability	**Software Testing:** Fuzzing, symbolic exec., sanitizers **Memory safety:** Static analysis, safe languages
2	Integrity Violation	**Software Compartmentalization:** Code Integrity, Pointer integrity, Memory Management
3	Exploit Payload	**Software Diversification:** ASLR, ISR, DSR
4	Exploit Dispatch	**Run-time Integrity:** Control Flow Integrity, Data flow integrity
5	Exploit Execution	**Last line of defense:** W+X, DEP, Static/Run-time Attestation
6	Attack	**Information leak** / **Malicious execution**

Steps taken by an adversary

1. Memory Vulnerability
2. Integrity Violation
3. Exploit Payload
4. Exploit Dispatch
5. Exploit Execution
6. Attack

Software Testing: Fuzzing, symbolic exec., sanitize
Memory safety: Static analysis, safe languages

Software Compartmentalization: Code Integrity, Poi
integrity, Memory Management

Software Diversification: ASLR, ISR, DSR

Run-time Integrity: Control Flow Integrity,
Data flow integrity

Last line of defense:
W+X, DEP, Static/Run-time Attestation

Root of Trust (RoT)

Secure Boot

OS's

TPMs

TEEs

Information leak

Malicious execution

Coming up…

- Operating Systems Security
  - Memory management units (MMUs), virtualization
  - Compartmentalization / Sandboxing
  - Access control, capabilities
- “Usable” Security
  - Authentication & attestation
  - Software supply chain – attacks and defenses (detection)
- Mobile & Hardware Security
  - Android security
  - Trusted Platform Modules (TPMs)
  - Trusted Execution Environments (TEEs)
  - Side Channel attacks & prevention
- Non-technical security aspects
  - Ethical and legal issues
  - Proving Compliance in systems

Reminders & Resources

Reminders:

- A2 is due June 20

Resources:

- Dynamic Malware Analysis in the Modern Era—A State of the Art Survey
- Reflections on Trusting Trust
- SoK: Eternal War in Memory