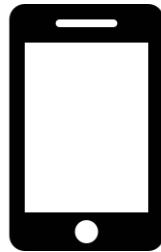


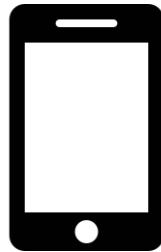
# Protecting Computer Systems through Eliminating or Analyzing Vulnerabilities

Byoungyoung Lee  
Georgia Institute of Technology

# Computers are everywhere



# Computers are everywhere



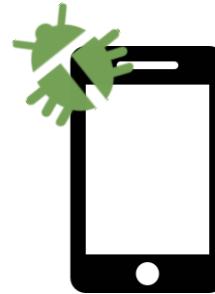
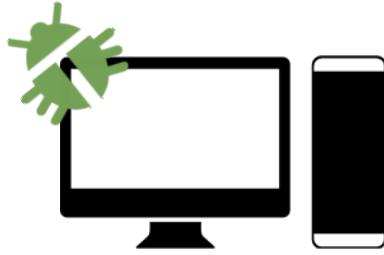
Affecting every aspect of our life



# Vulnerabilities are everywhere



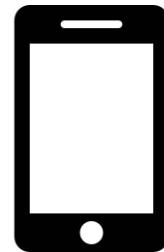
# Vulnerabilities are everywhere



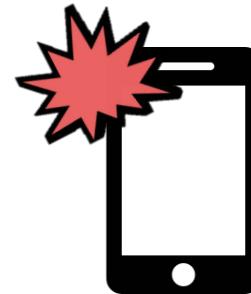
Human makes a mistake,  
and computers are made by human.



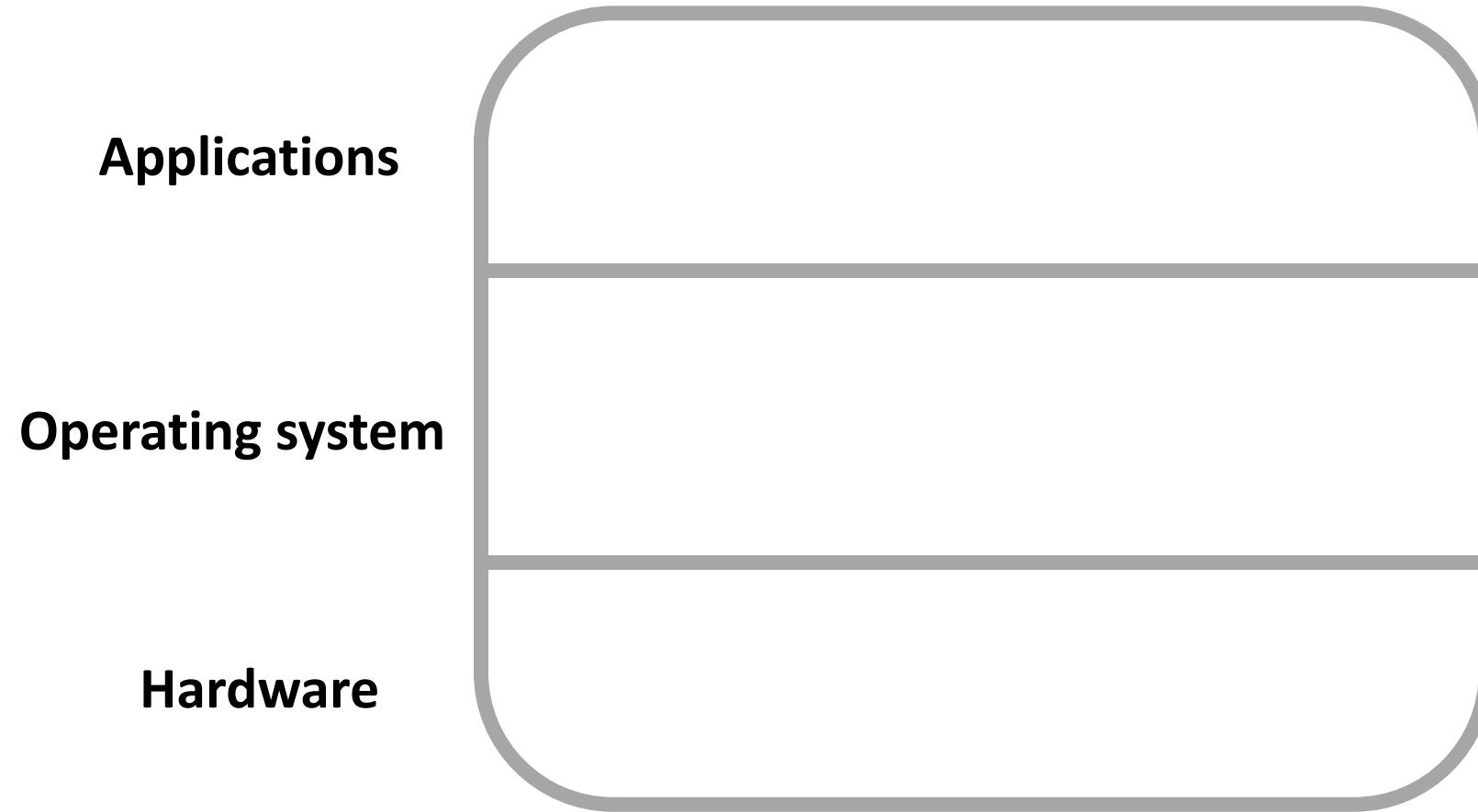
# Vulnerabilities are critical security problems



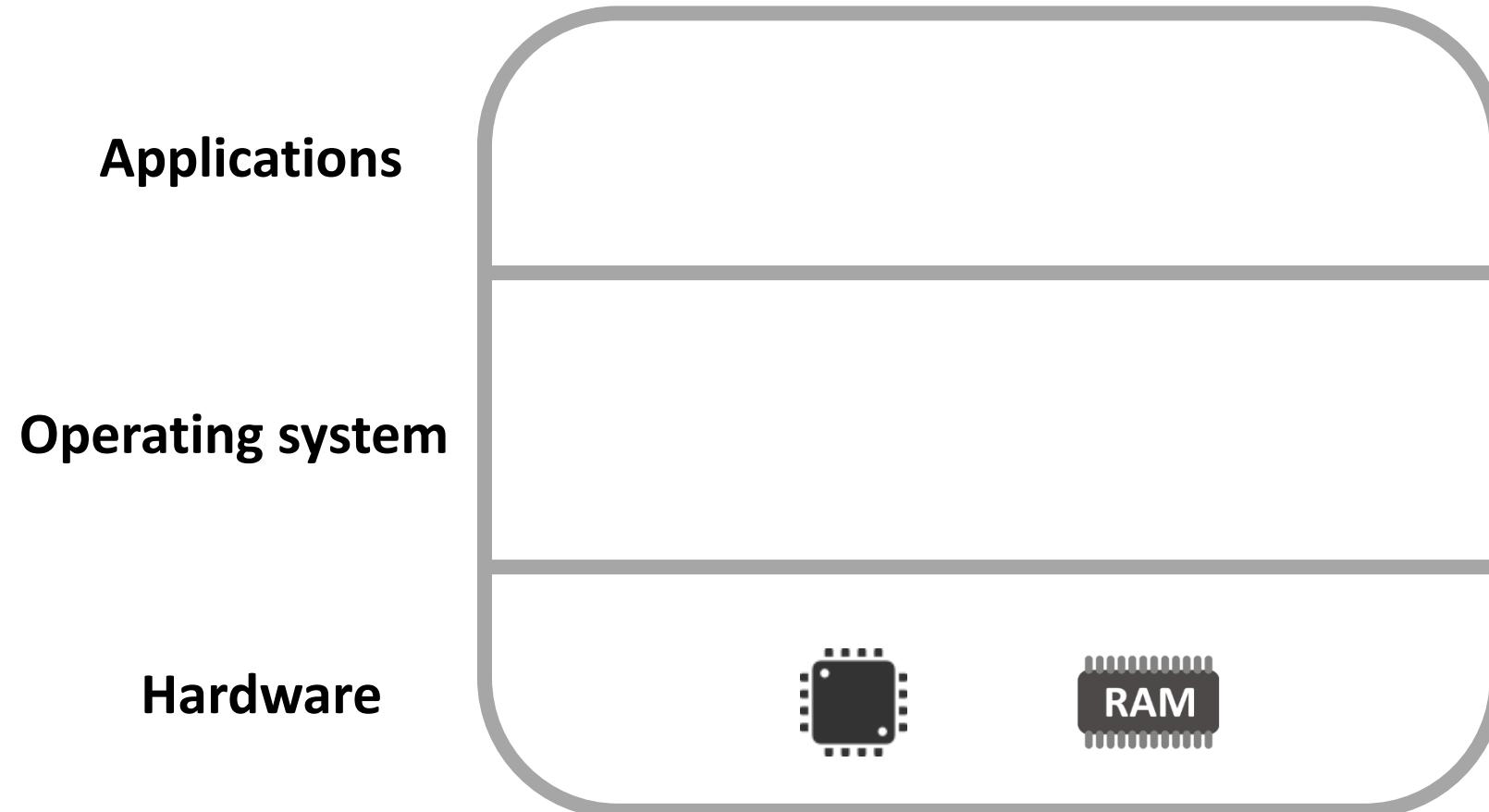
# Vulnerabilities are critical security problems



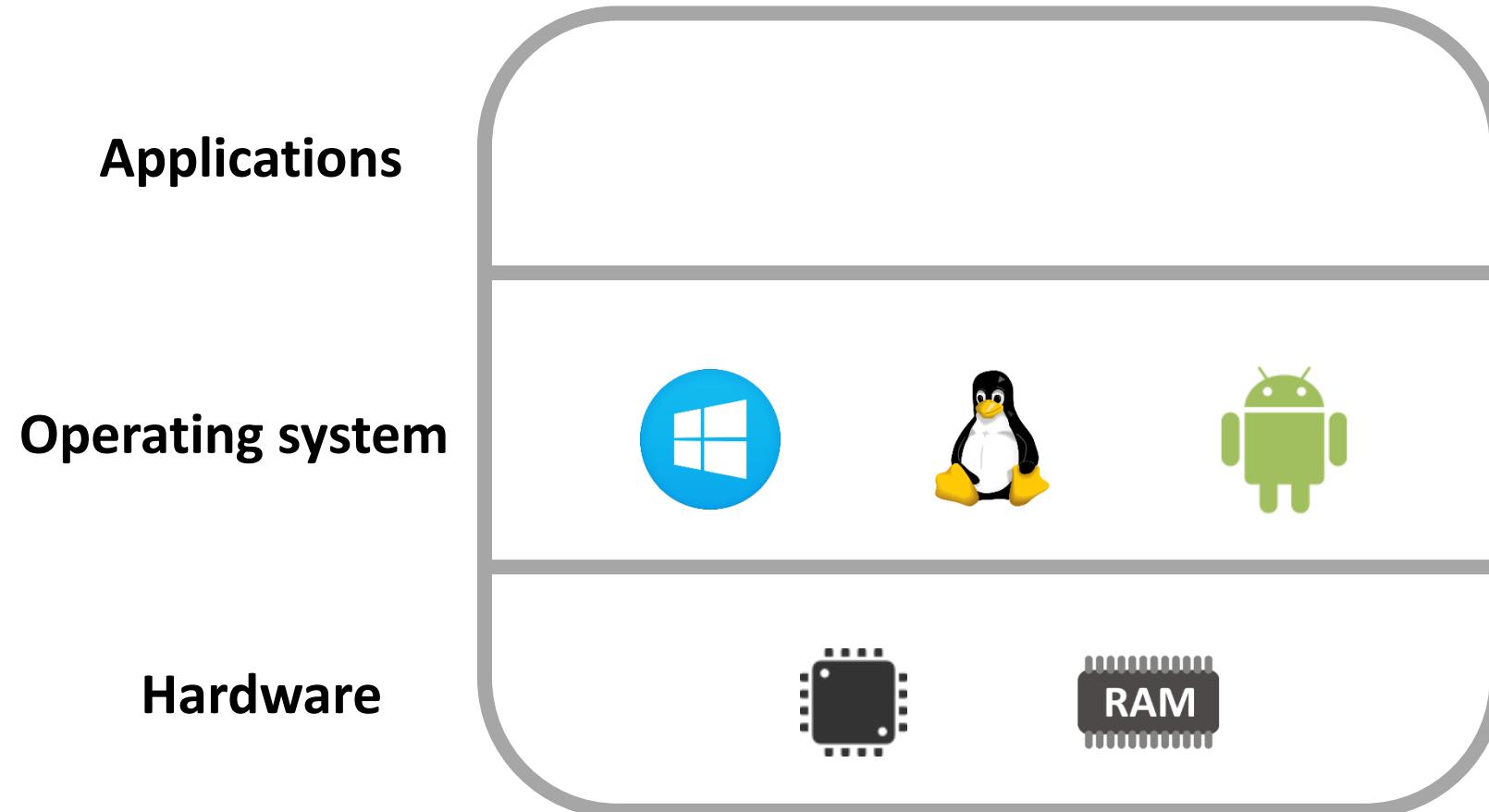
# Commodity computers are complex



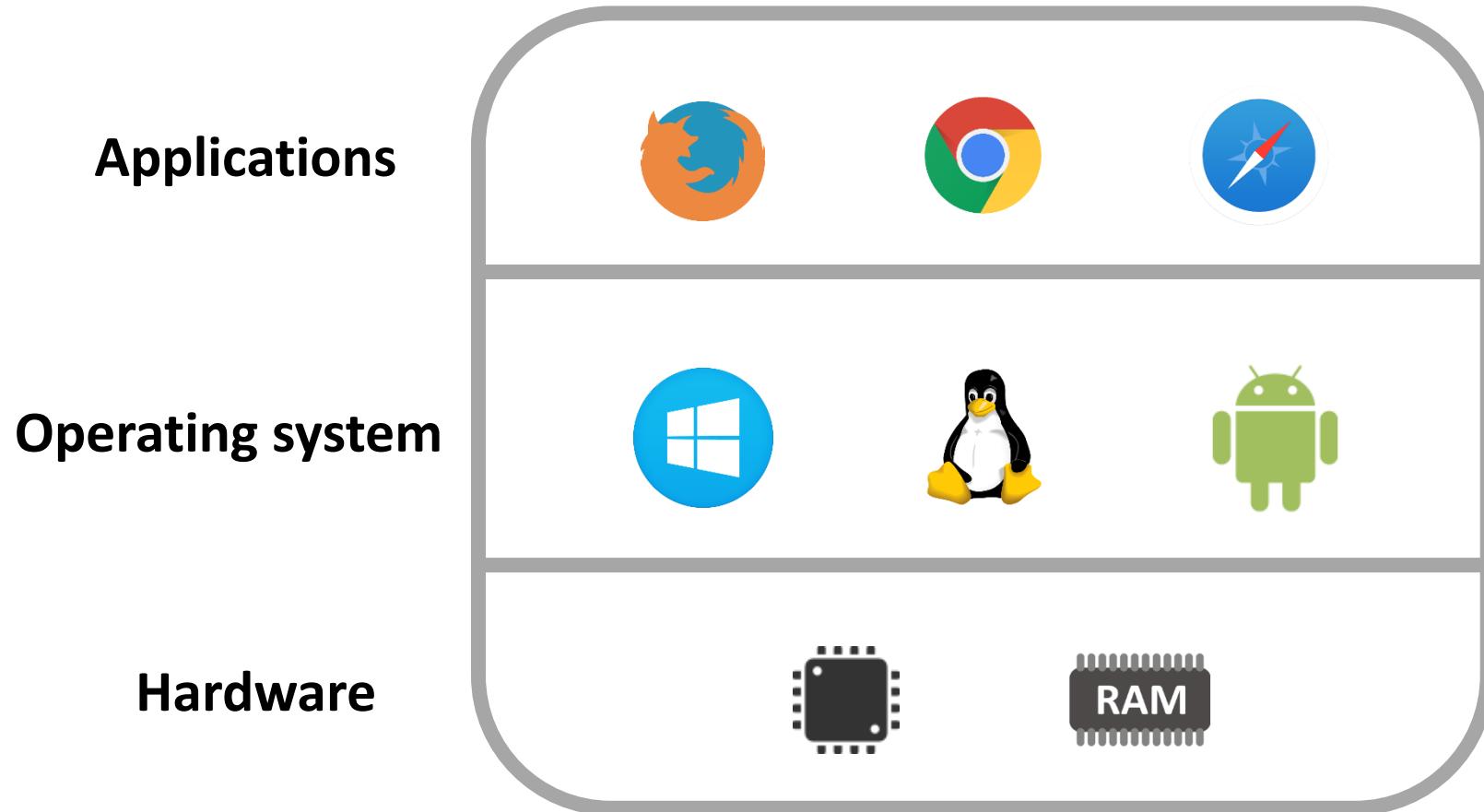
# Commodity computers are complex



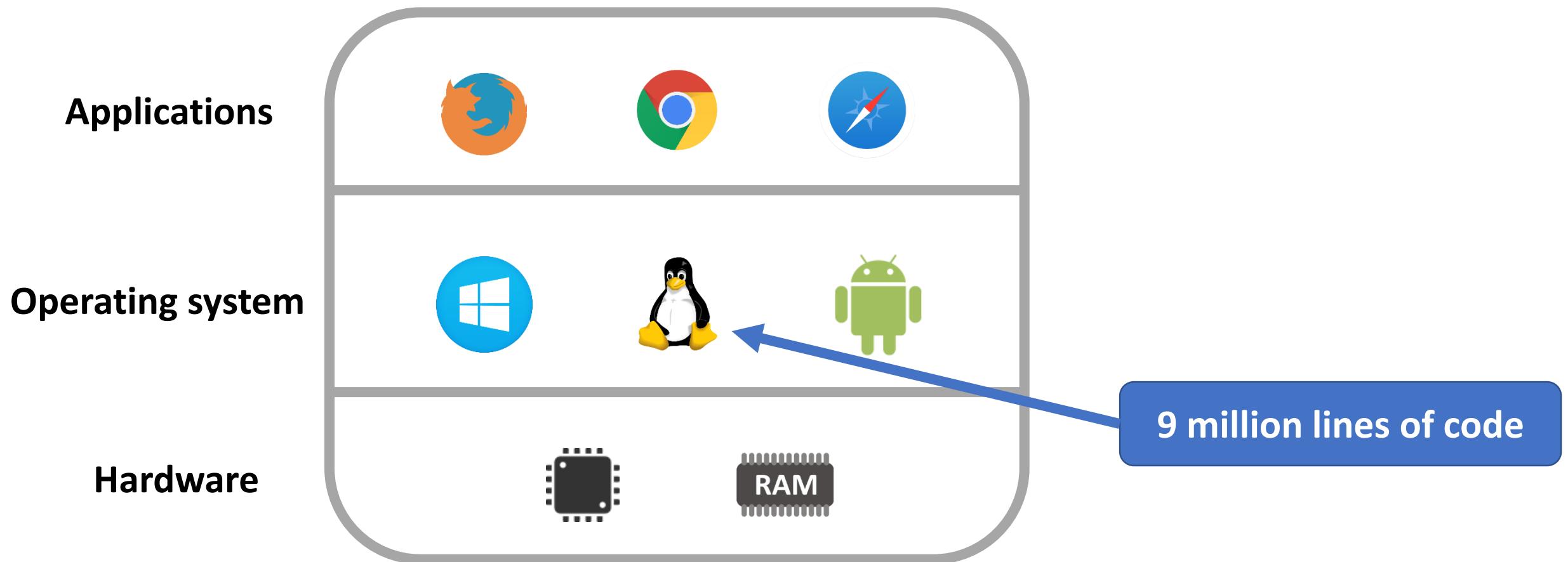
# Commodity computers are complex



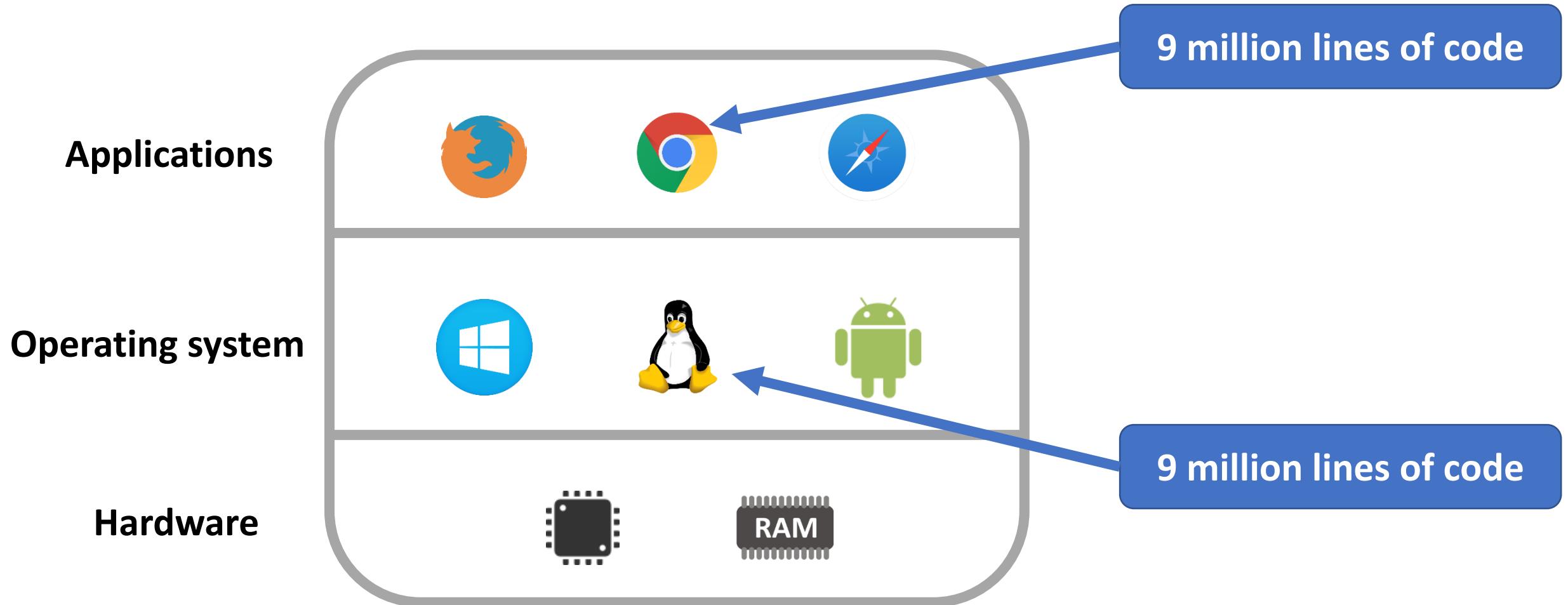
# Commodity computers are complex



# Commodity computers are complex



# Commodity computers are complex



# Many vulnerabilities

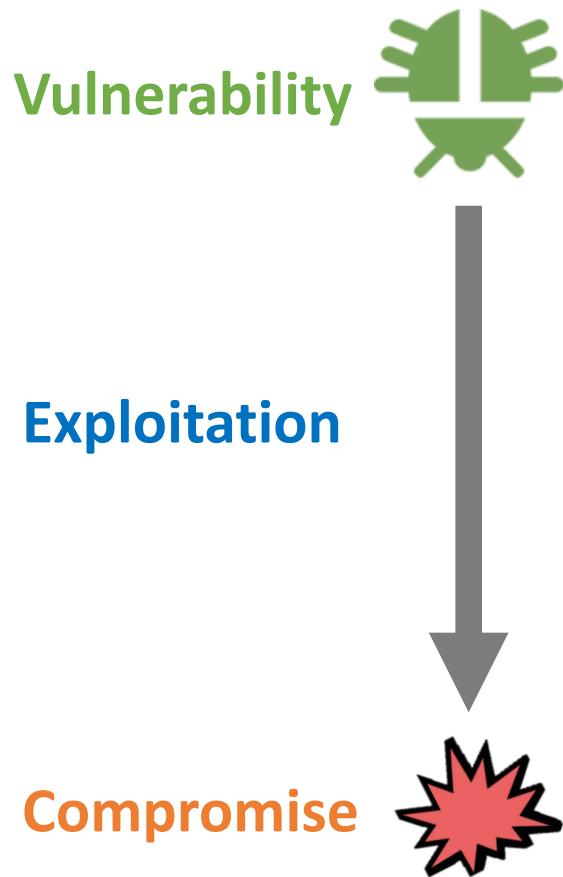
Year	# of vulnerabilities (CVE)
2012	5,297
2013	5,191
2014	7,946
2015	6,412

National Vulnerability Database, NIST

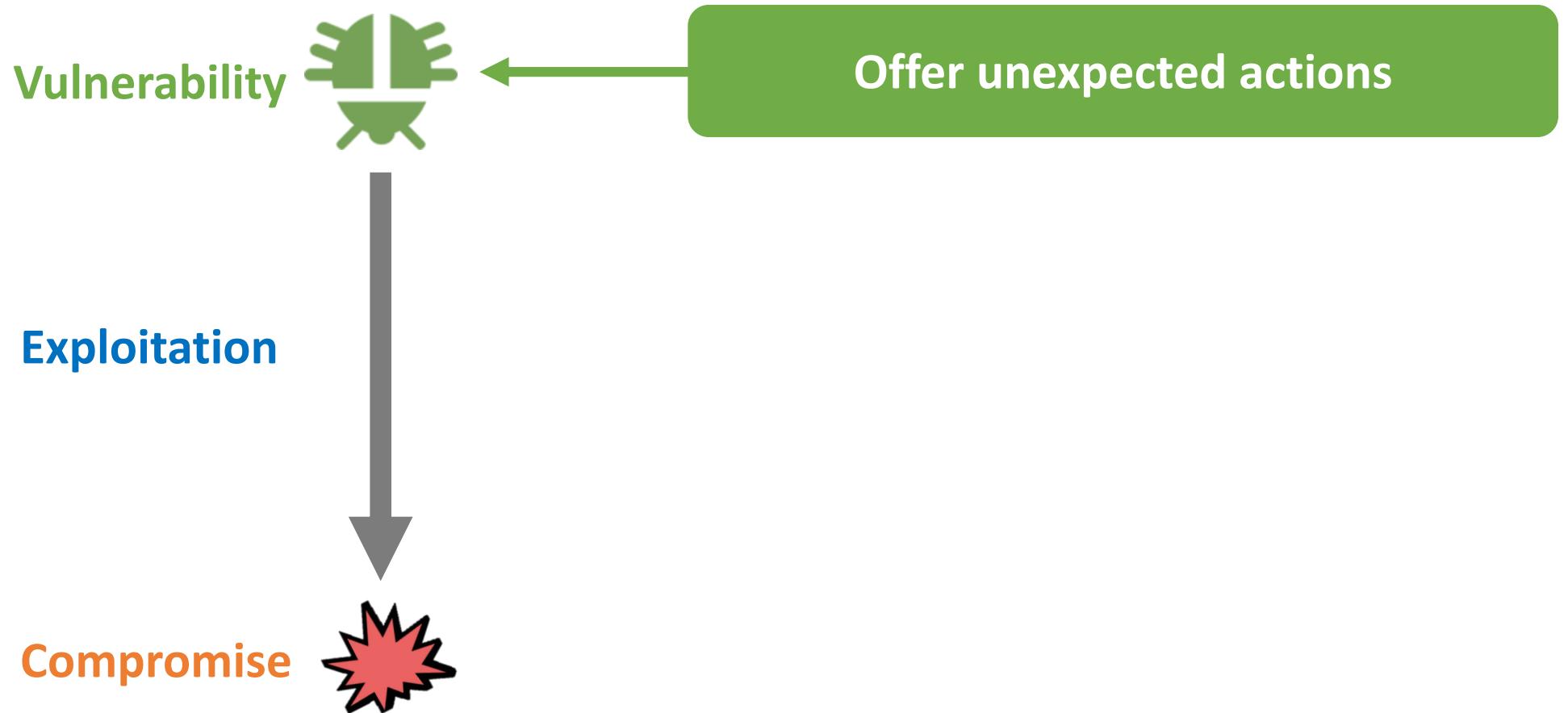
# Thesis focus and approaches

- Thesis focus
  - Protecting computer systems from vulnerabilities
- Approaches
  - Comprehensive understanding on both systems and vulnerabilities
  - Design practical security solutions for commodity systems

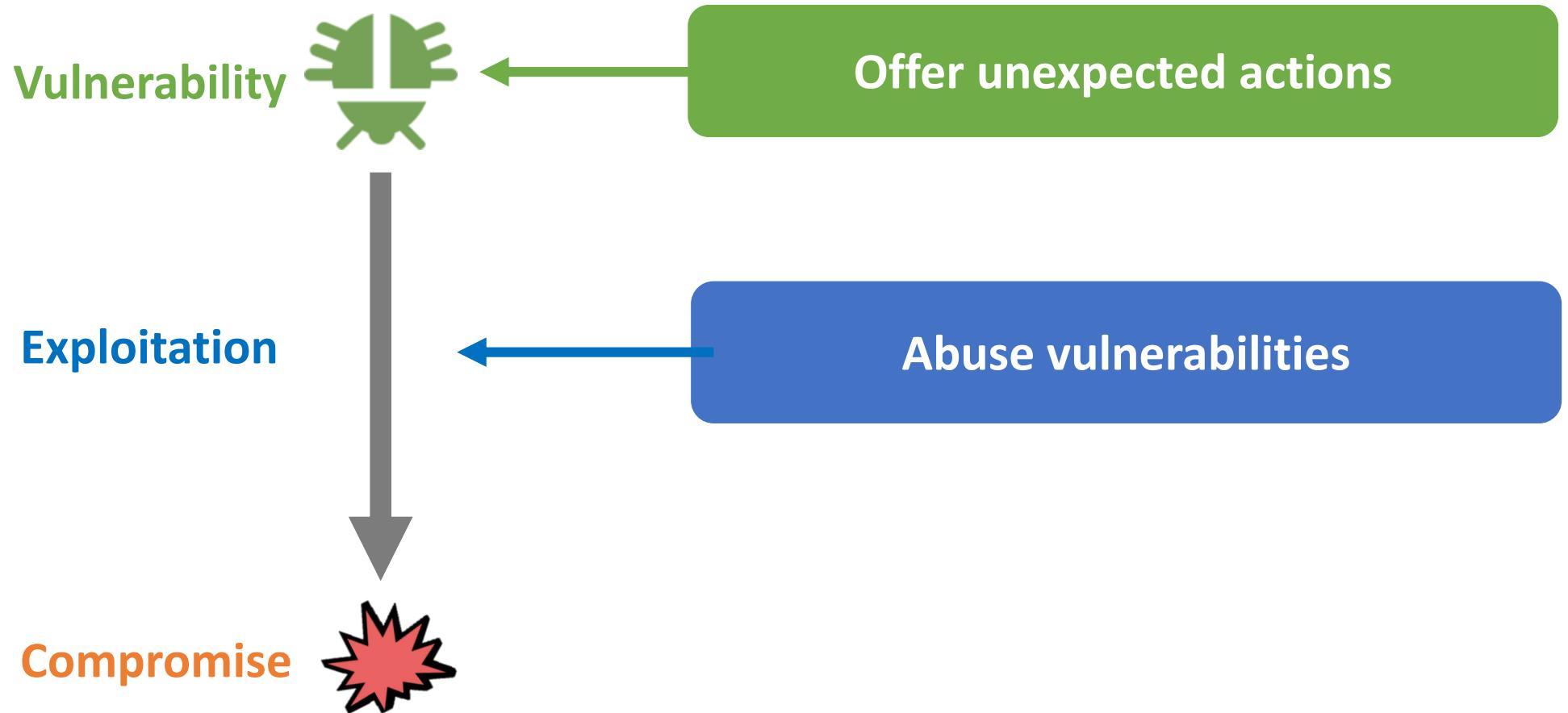
# Attacker's view on vulnerabilities



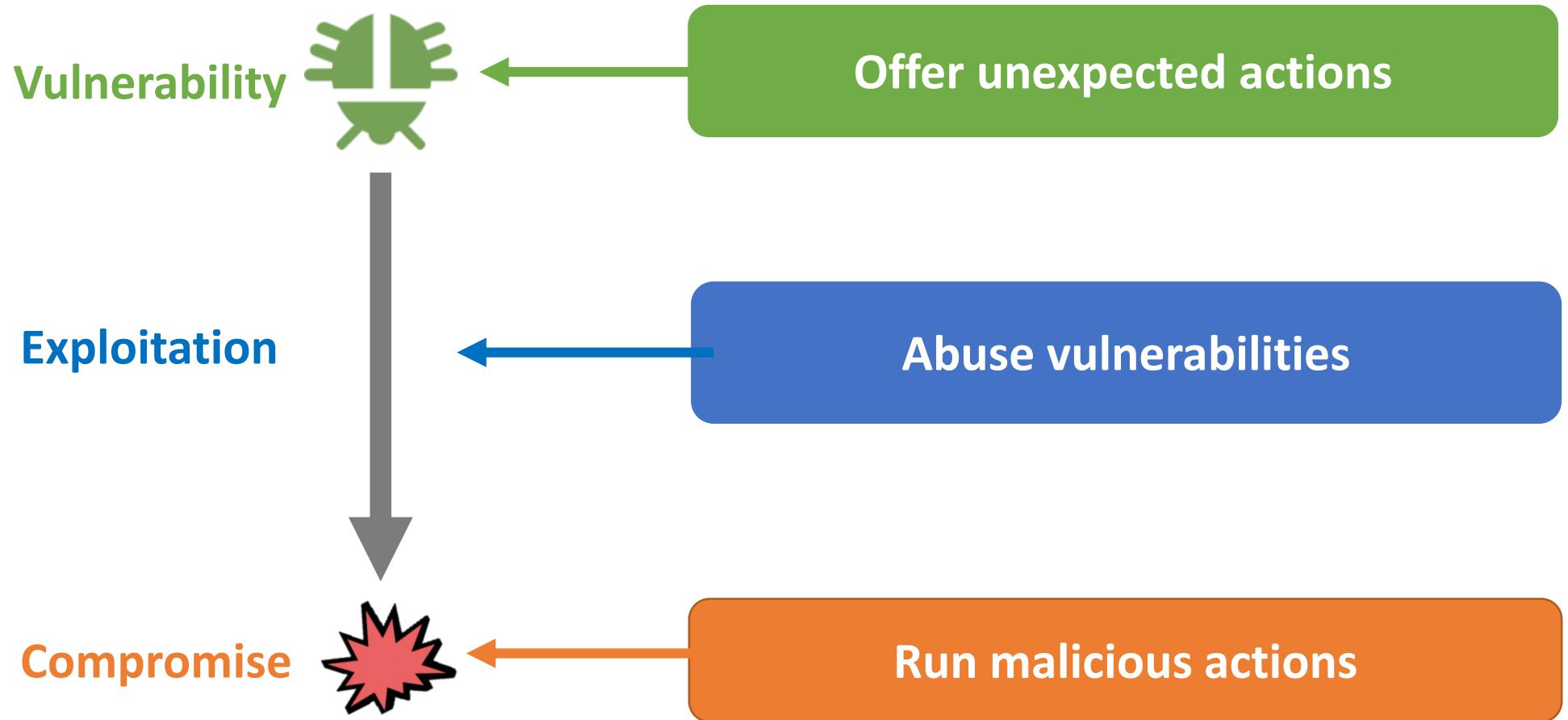
# Attacker's view on vulnerabilities



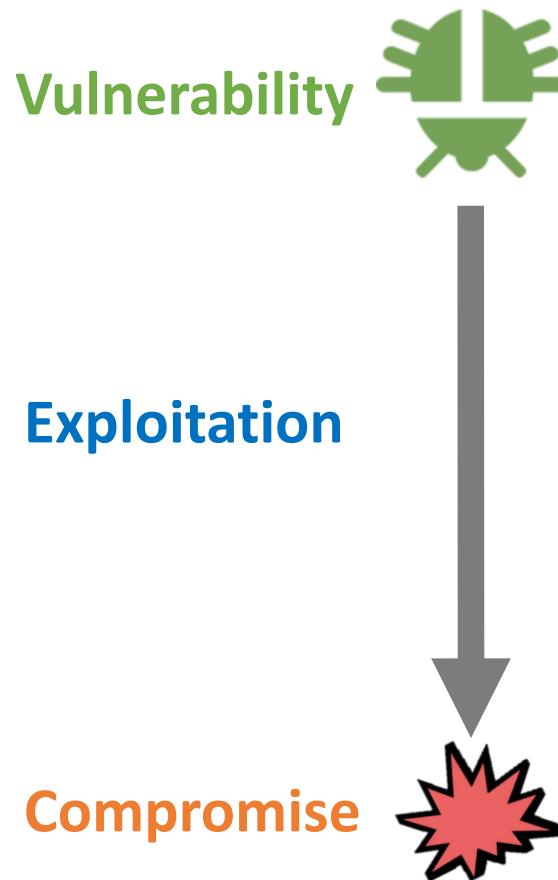
# Attacker's view on vulnerabilities



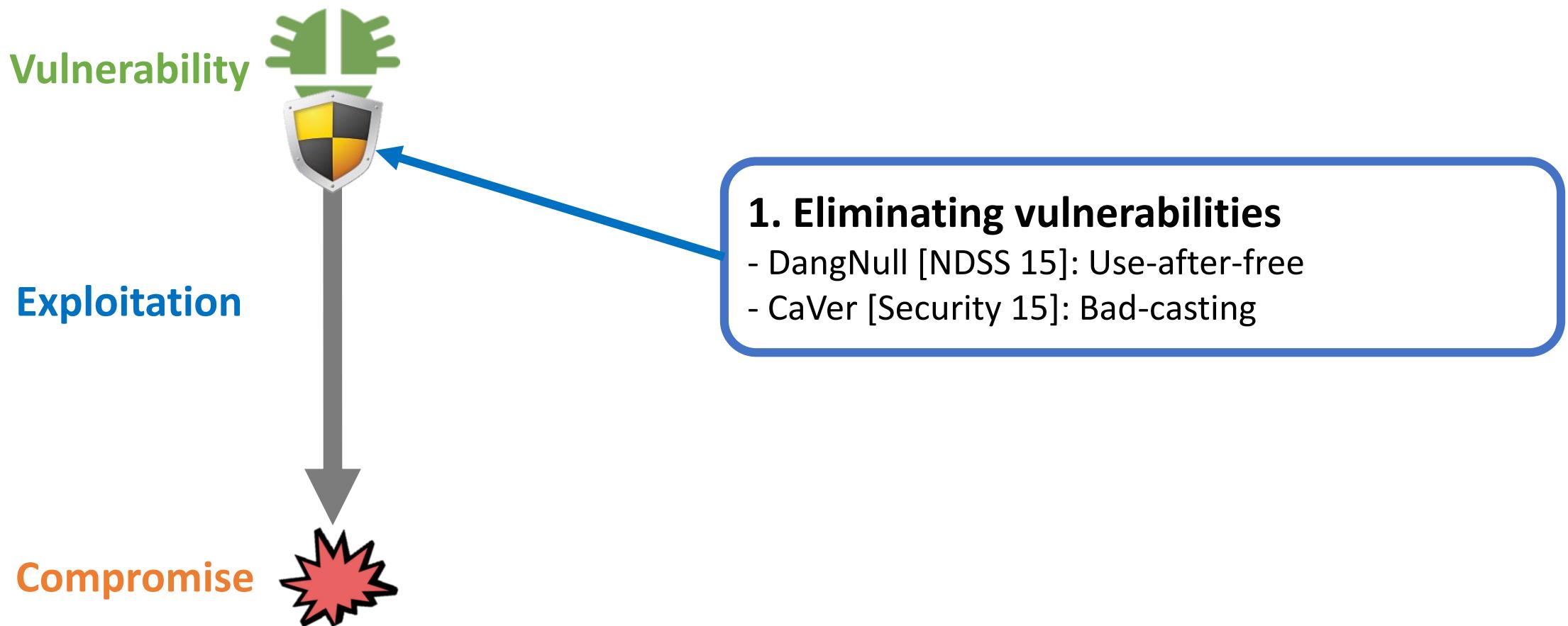
# Attacker's view on vulnerabilities



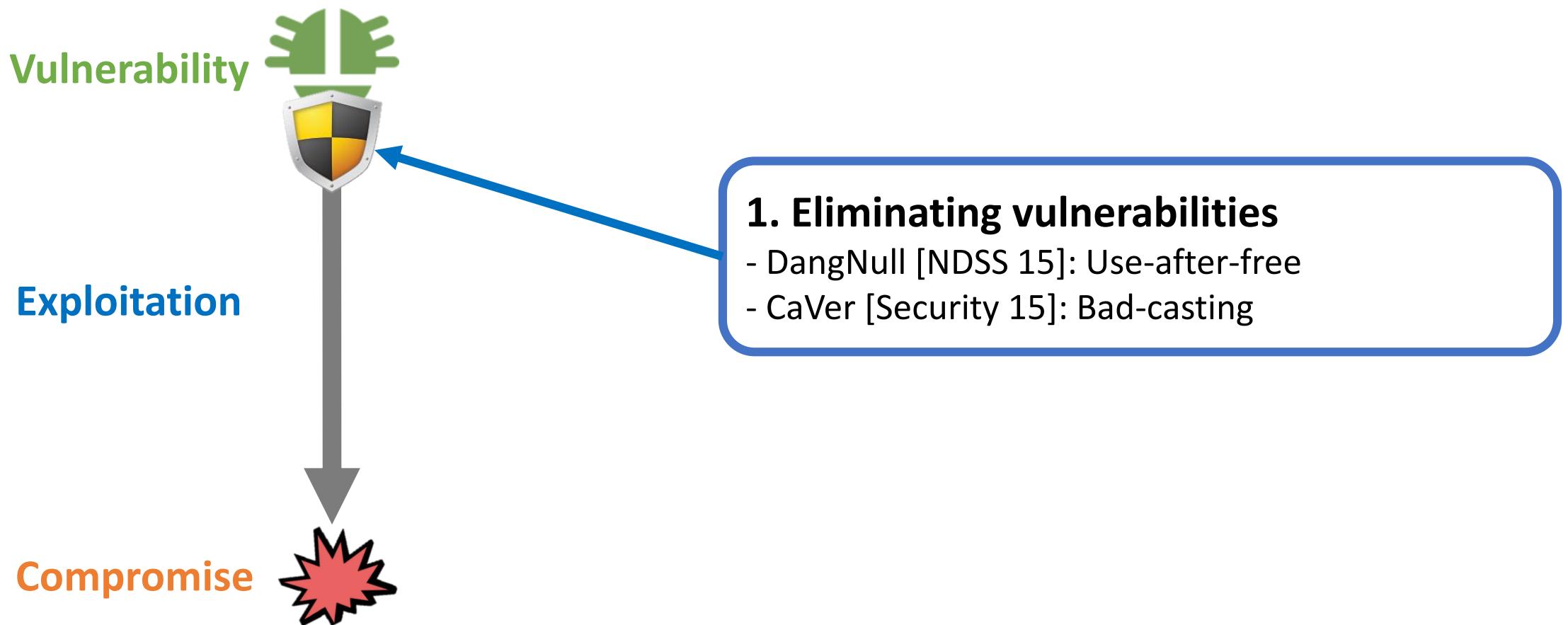
# Thesis topics



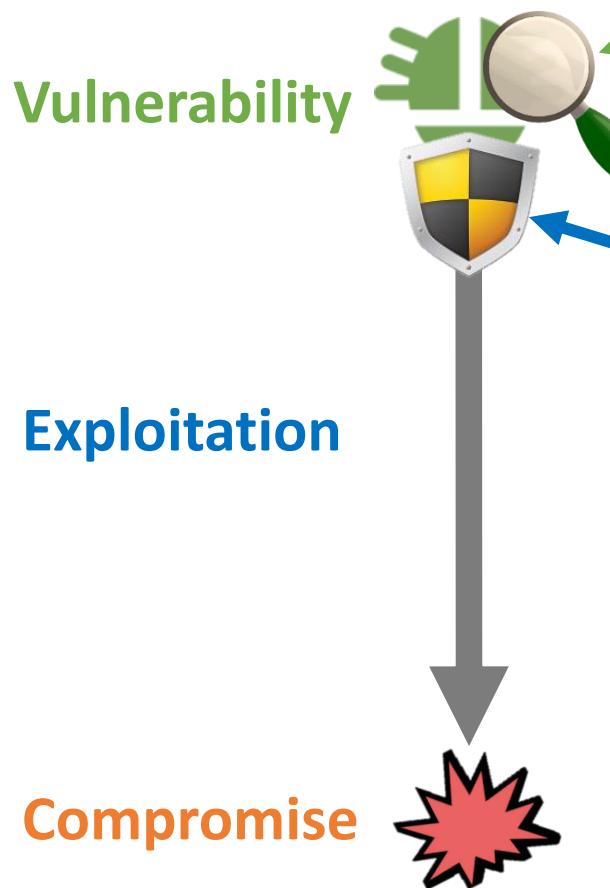
# Thesis topics



# Thesis topics



# Thesis topics



## 2. Analyzing vulnerability

- SideFinder: Timing-channels in hash tables

## 1. Eliminating vulnerabilities

- DangNull [NDSS 15]: Use-after-free
- CaVer [Security 15]: Bad-casting

## **1. Eliminating vulnerabilities**

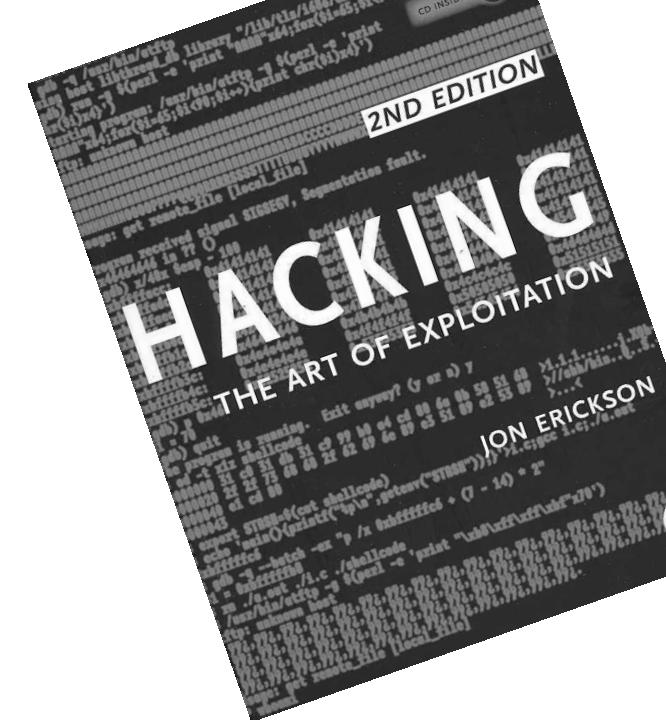
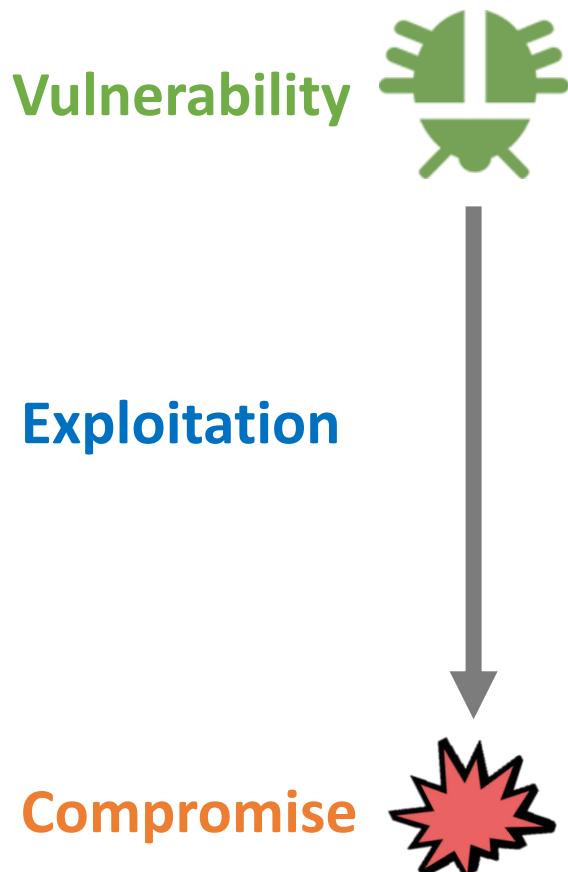
**DangNull [NDSS 15]: Eliminating use-after-free vulnerabilities**

**CaVer [Security 15]: Eliminating bad-casting vulnerabilities**

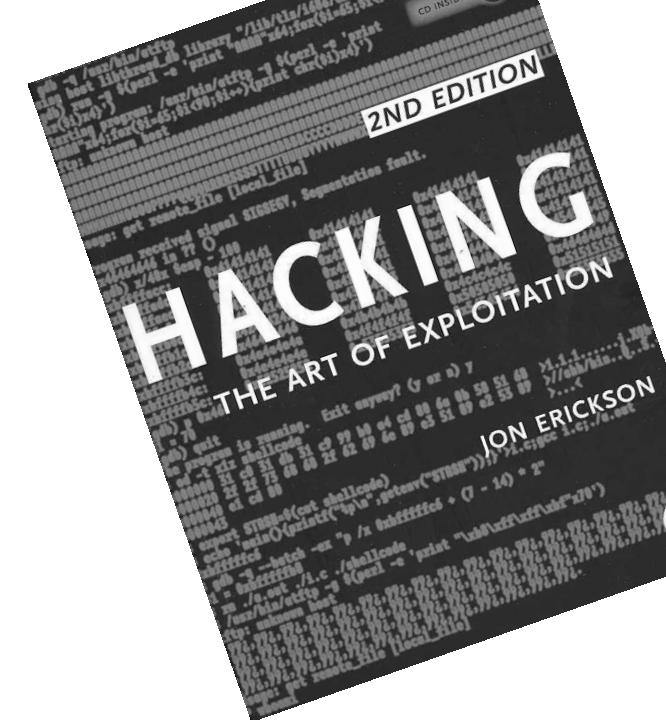
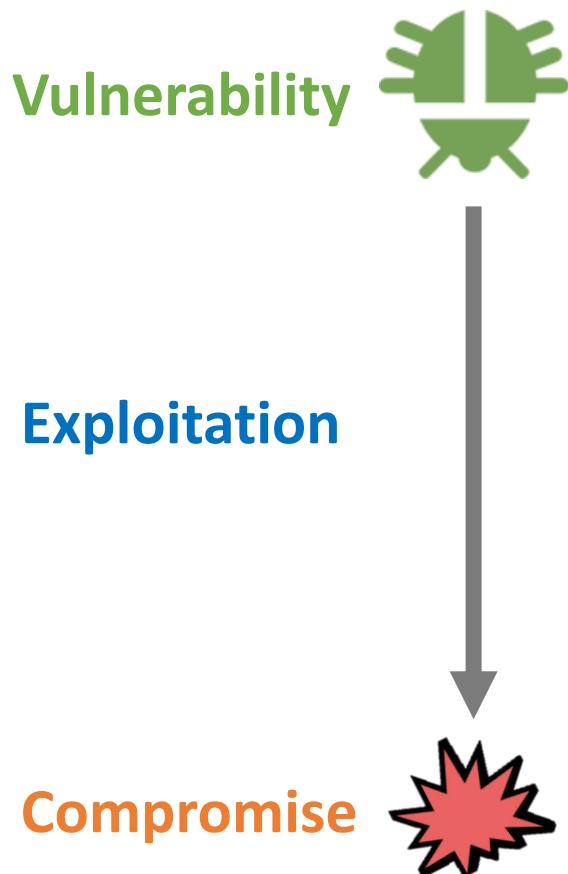
## **2. Analyzing vulnerabilities**

**SideFinder: Analyzing timing-channel vulnerabilities**

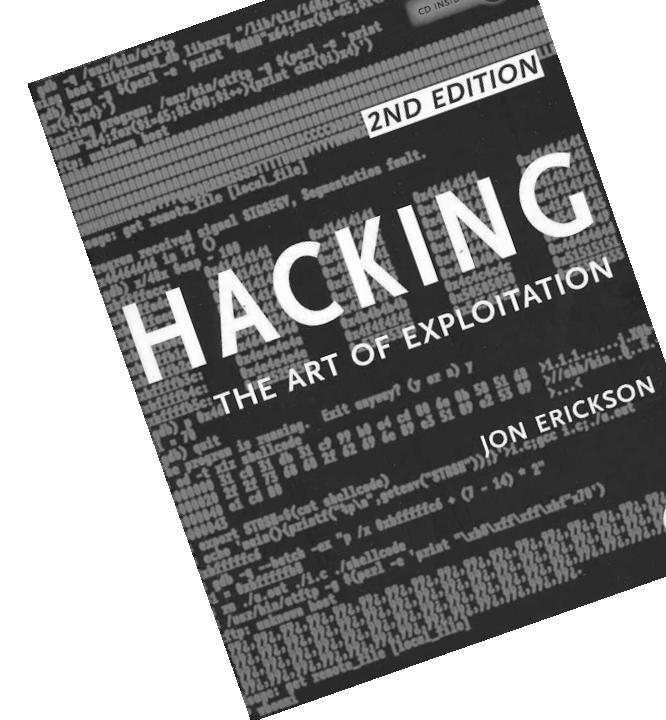
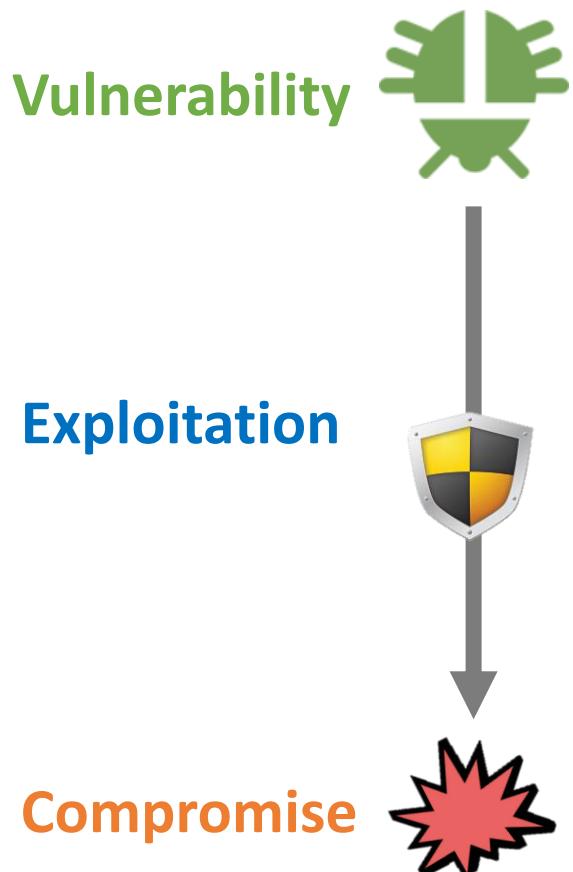
# Hacking: the art of exploitation



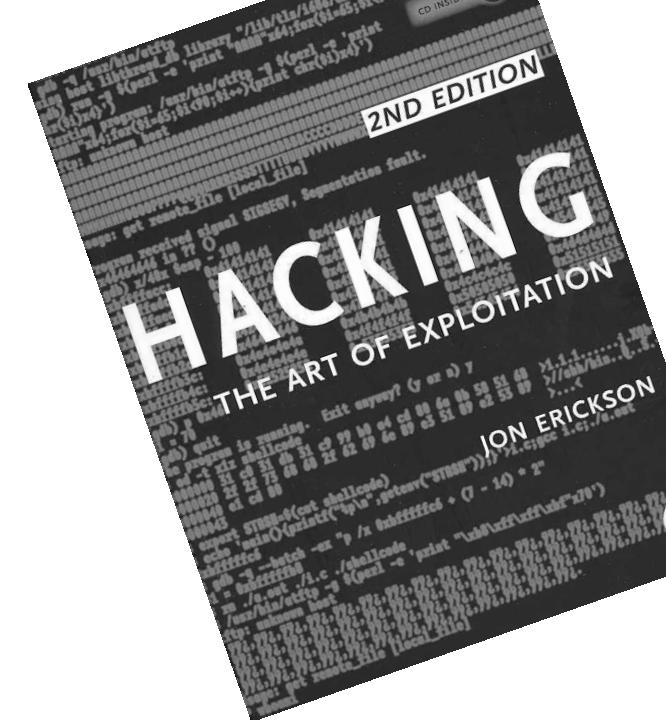
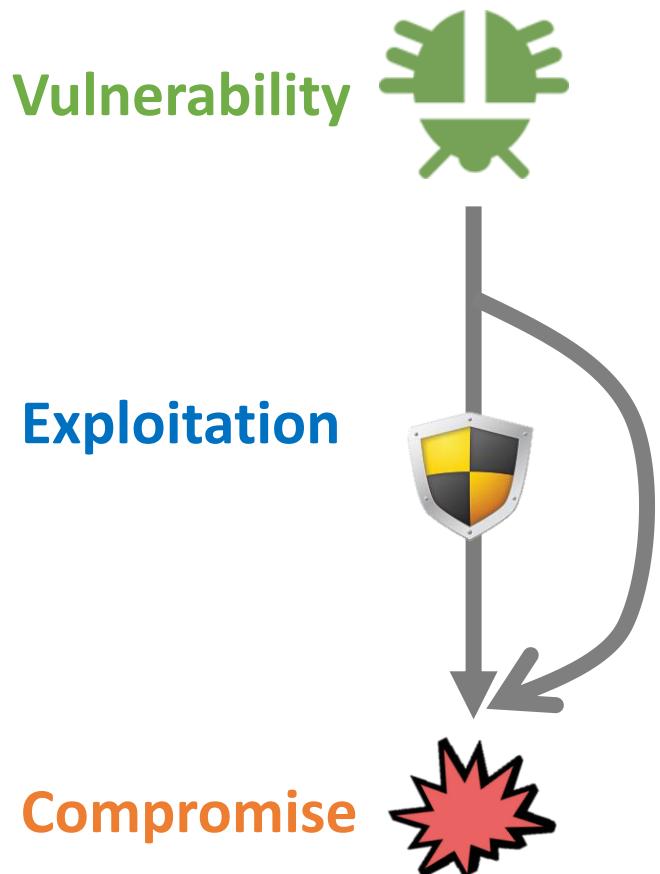
# Hacking: the art of exploitation



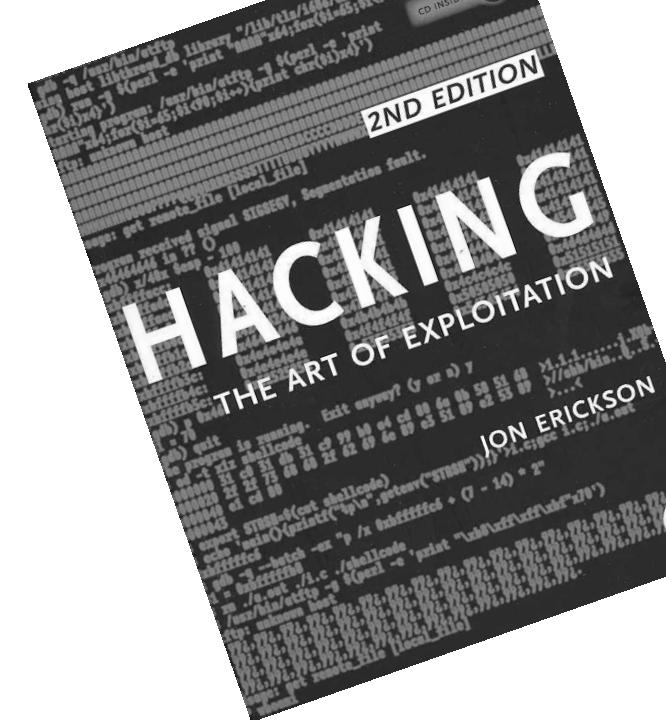
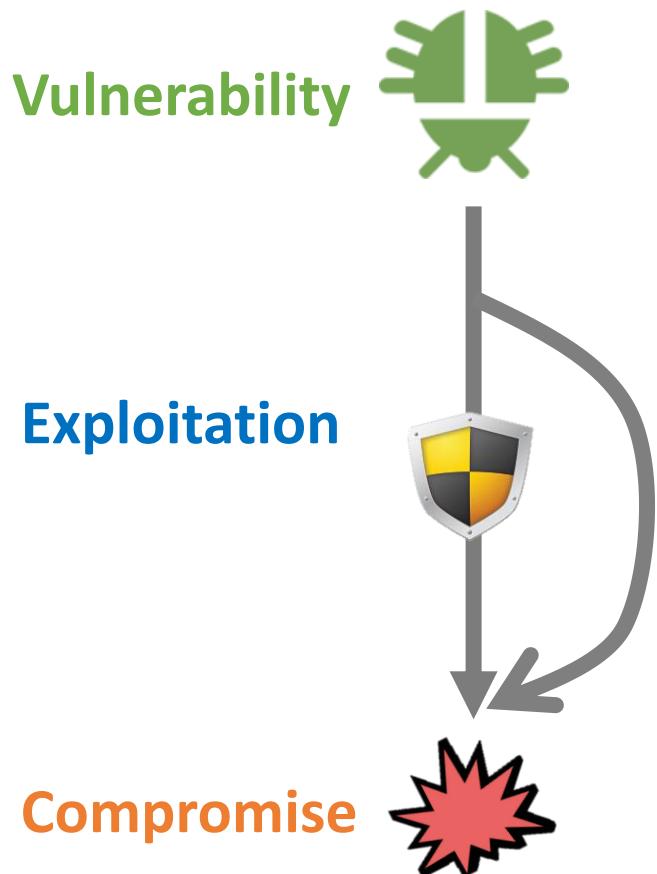
# Hacking: the art of exploitation



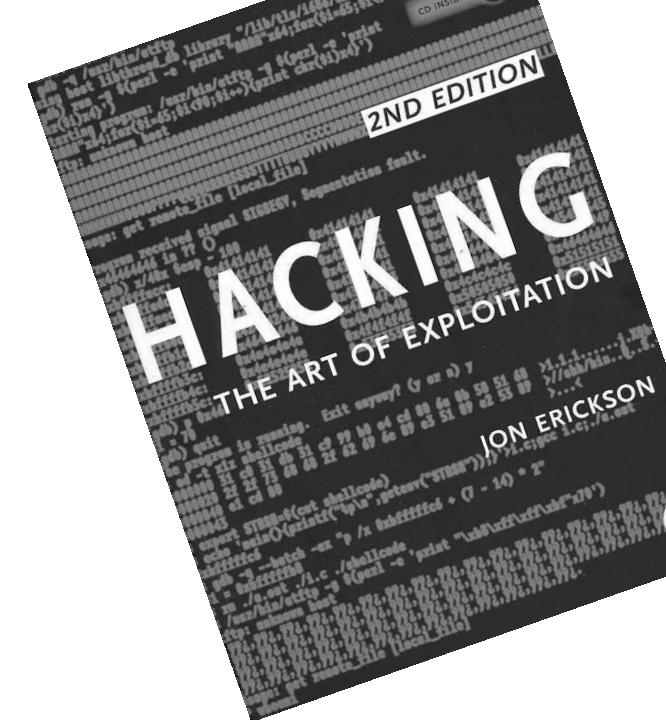
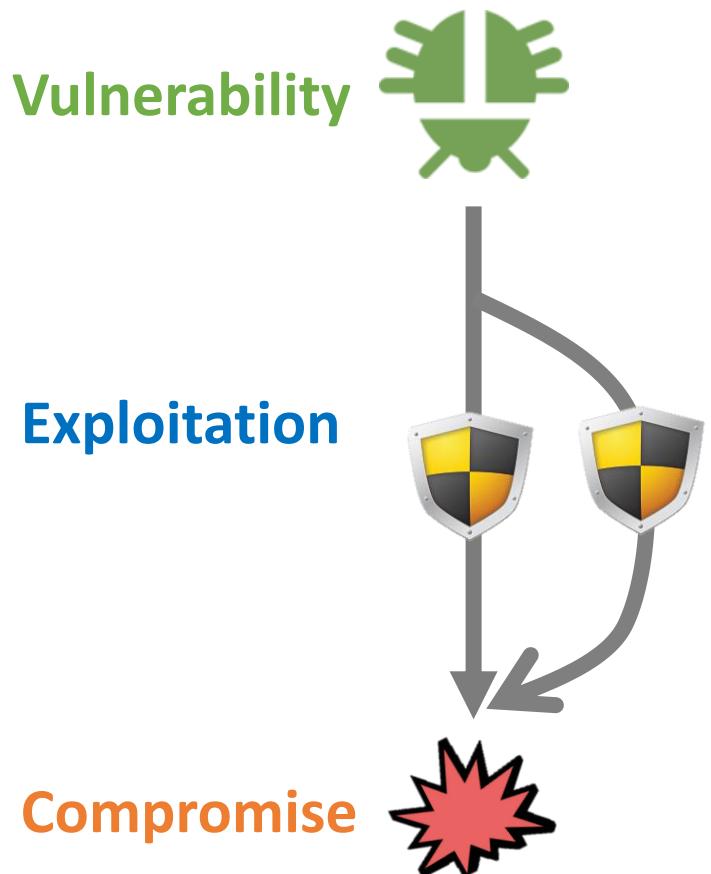
# Hacking: the art of exploitation



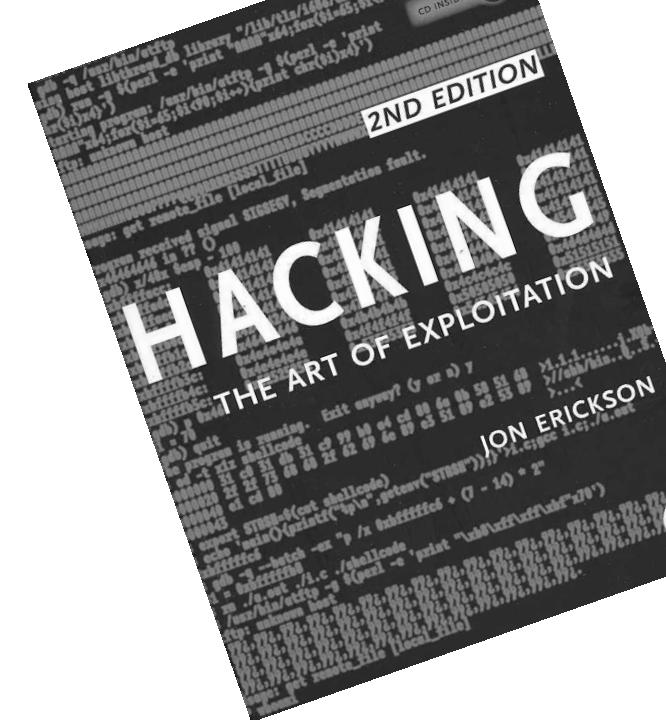
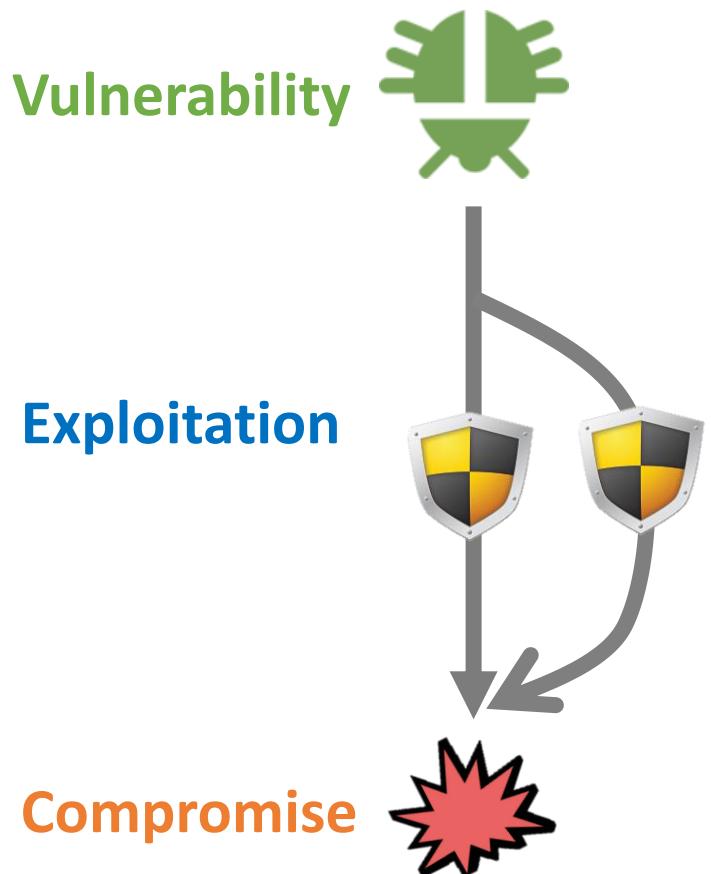
# Hacking: the art of exploitation



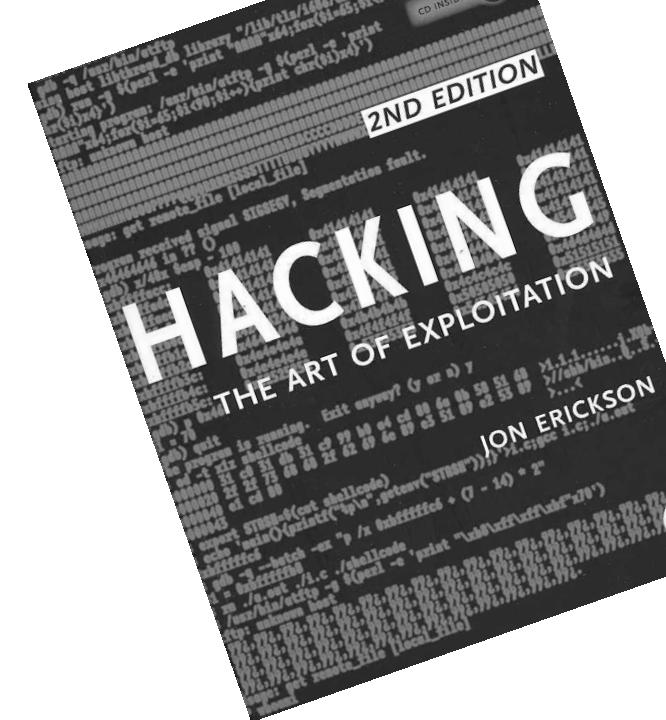
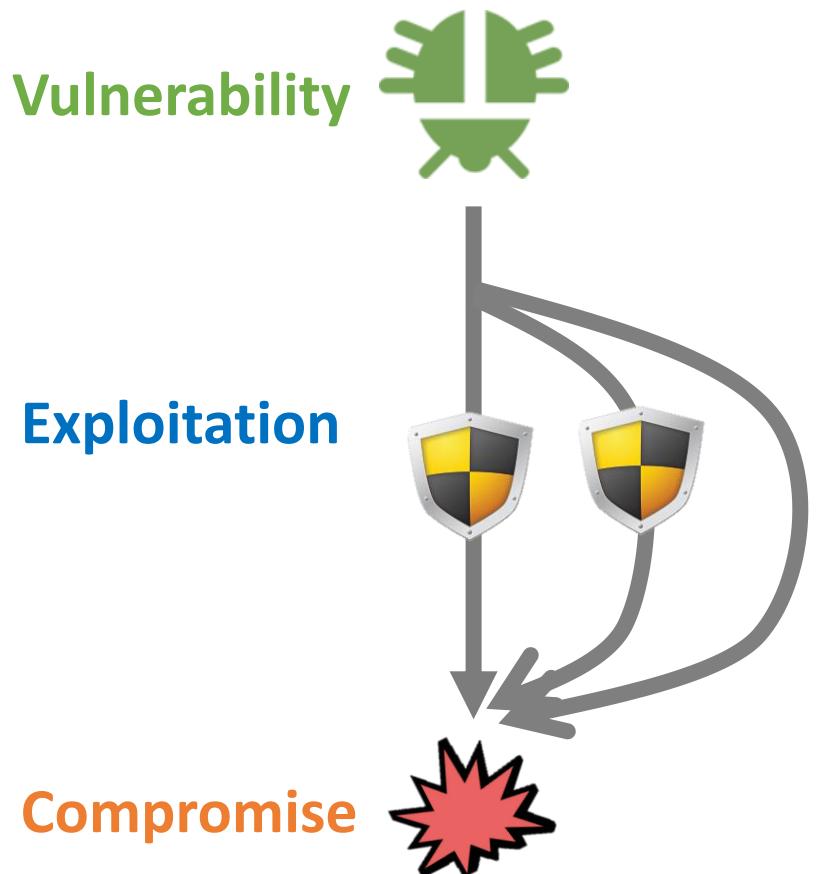
# Hacking: the art of exploitation



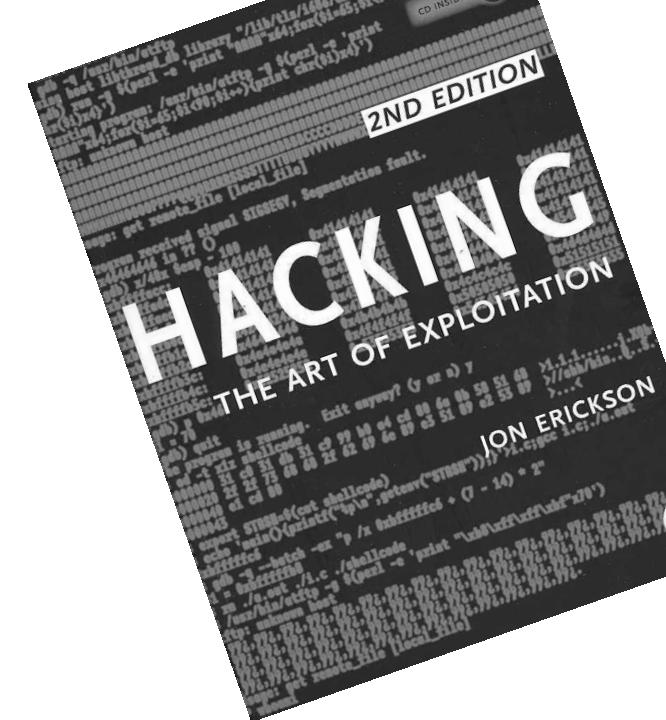
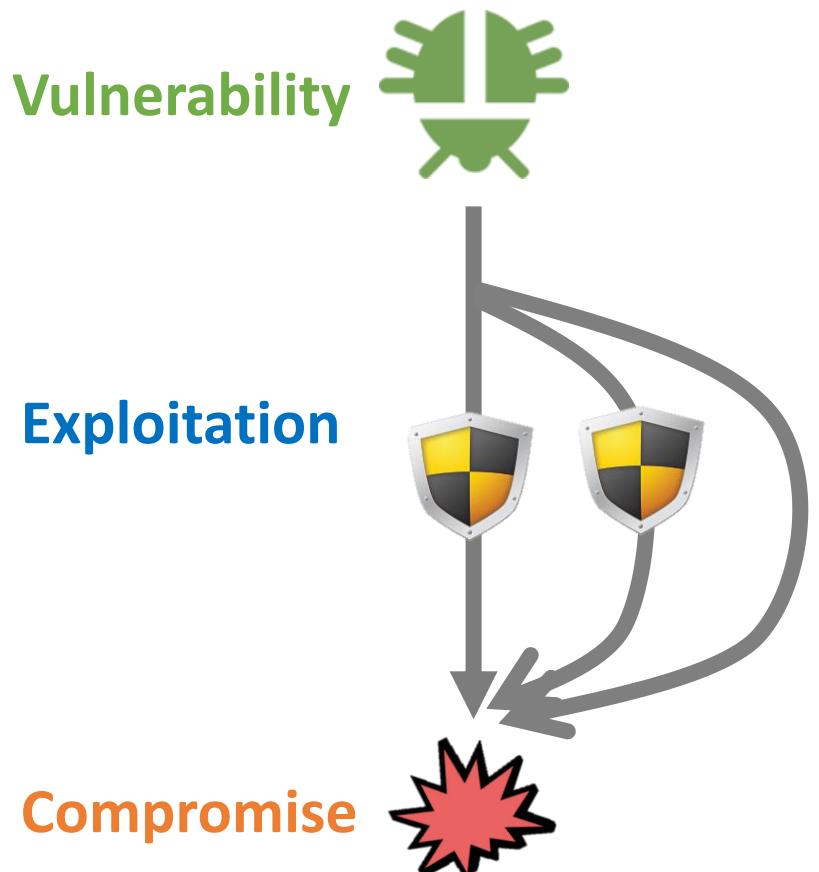
# Hacking: the art of exploitation



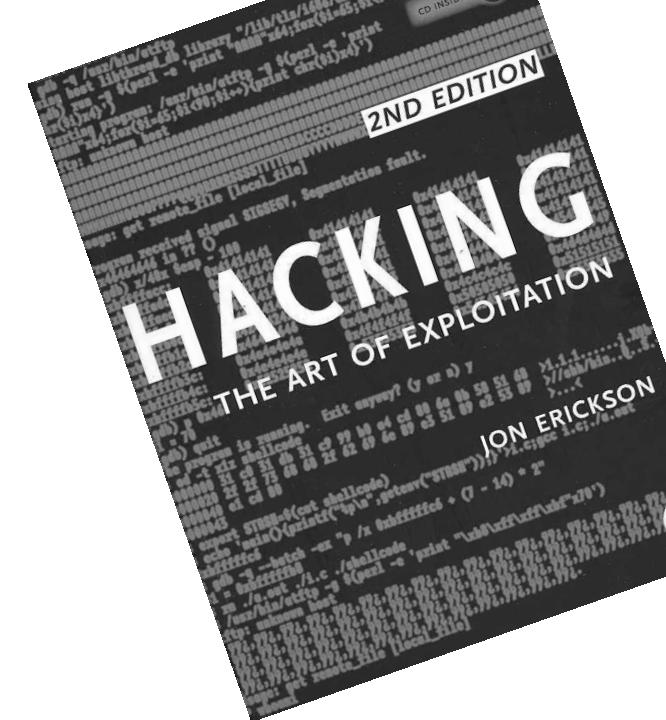
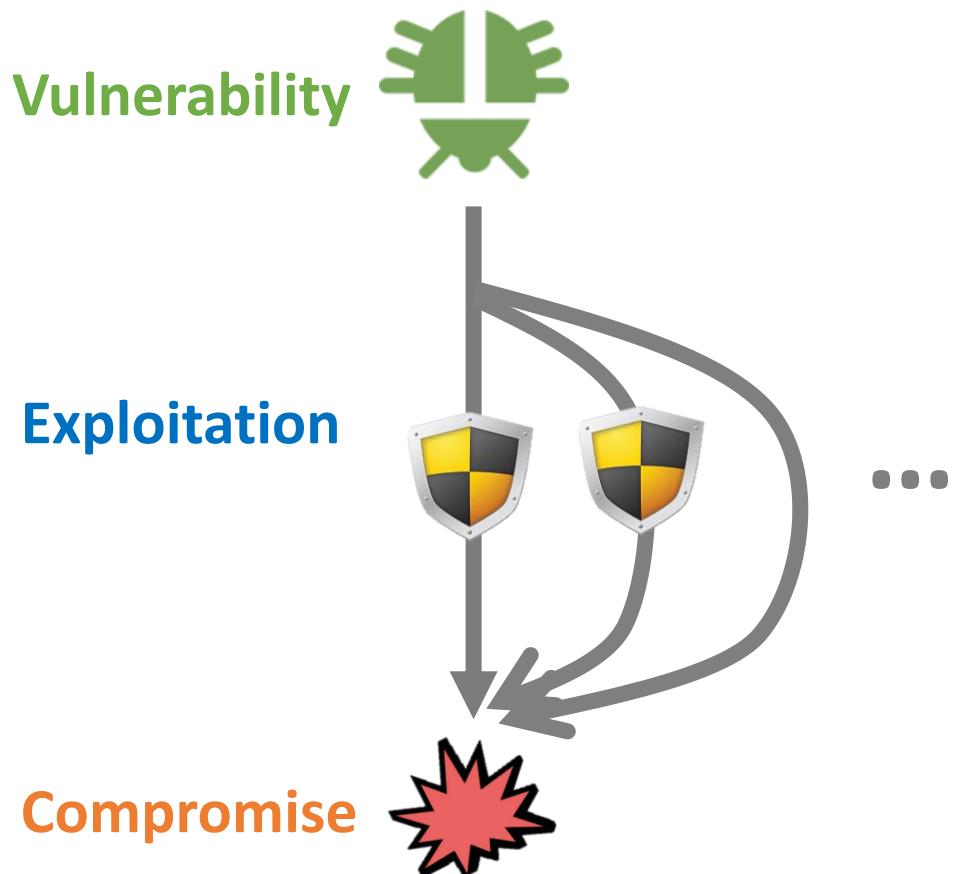
# Hacking: the art of exploitation



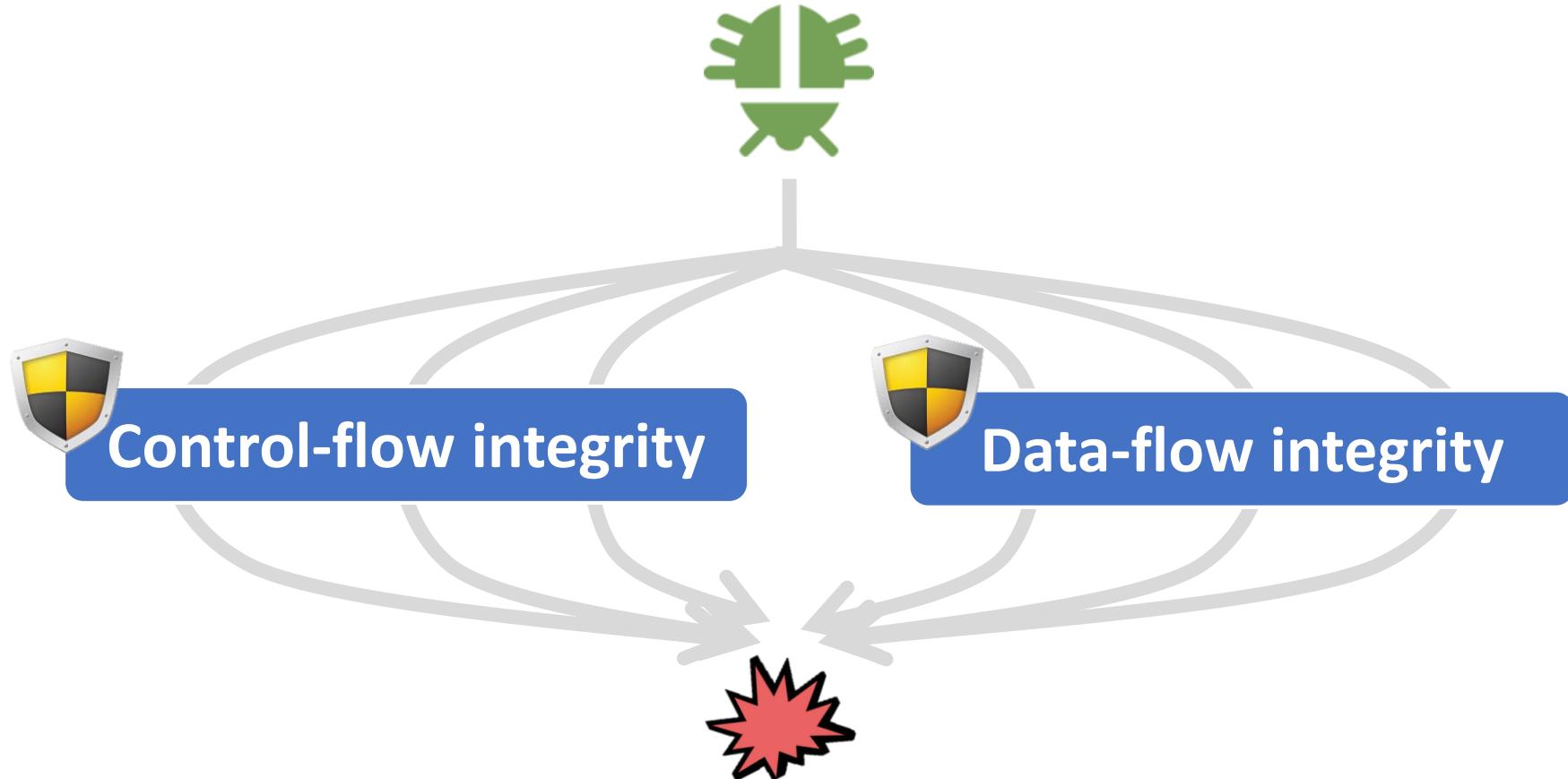
# Hacking: the art of exploitation



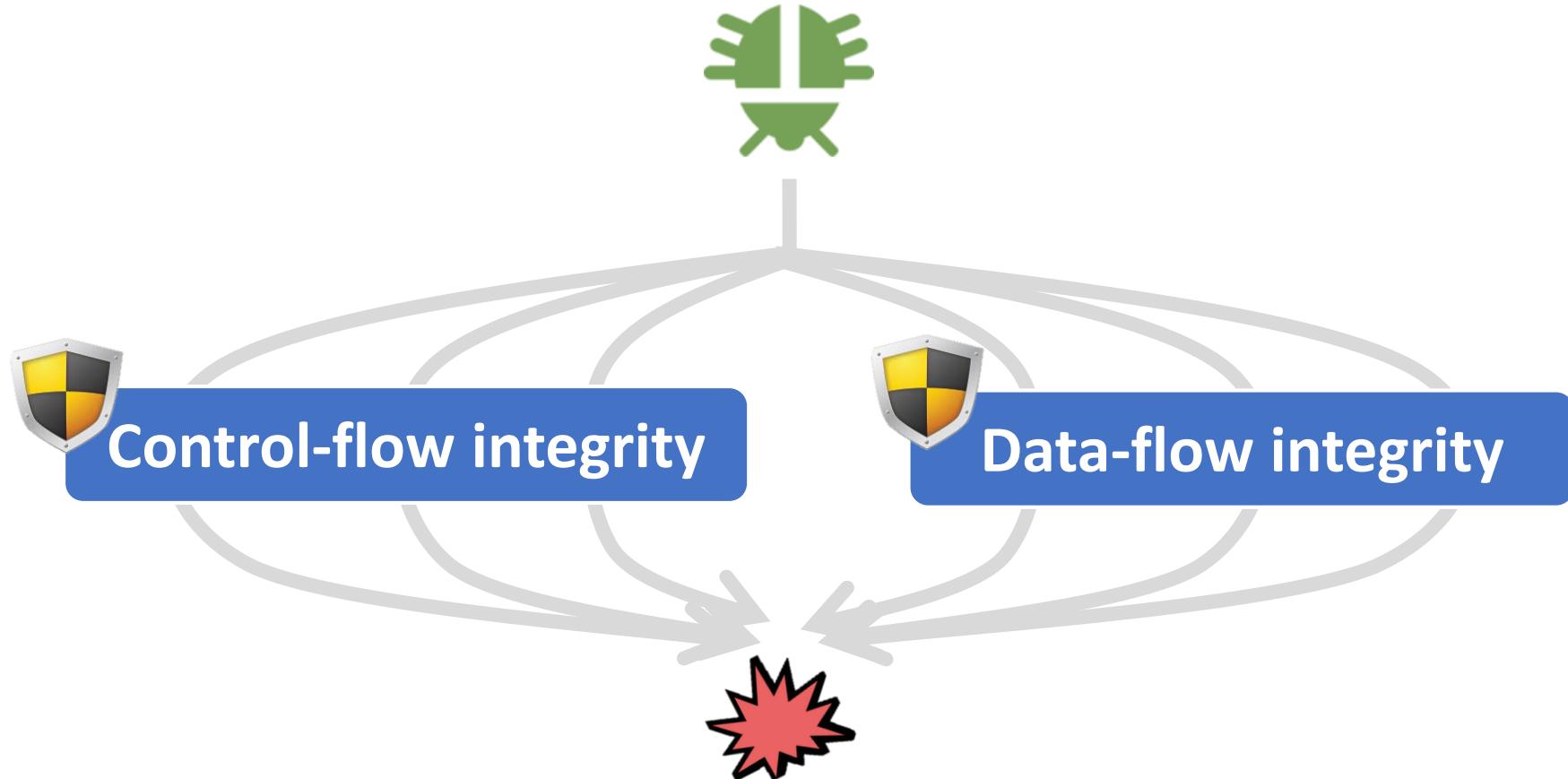
# Hacking: the art of exploitation



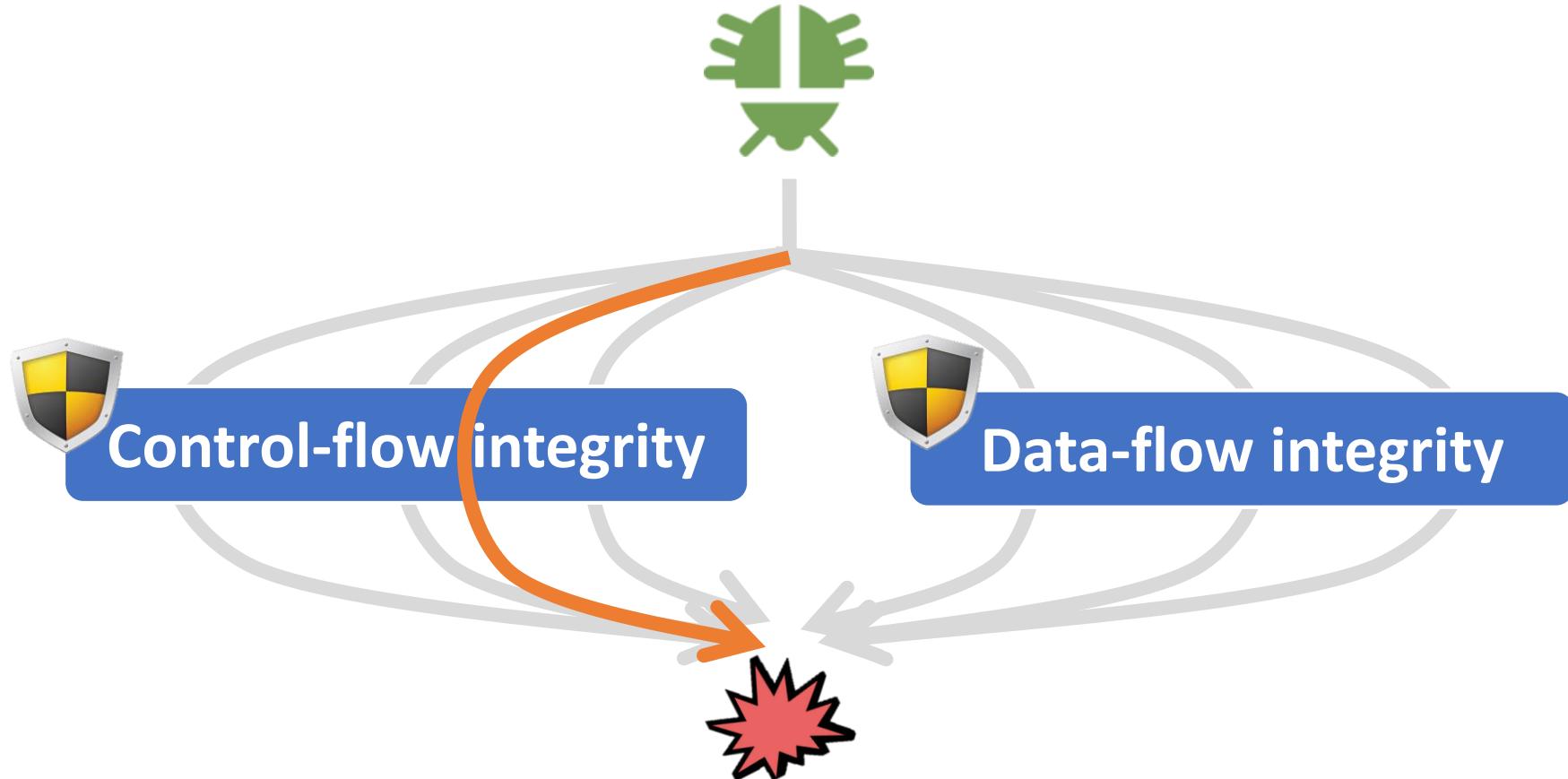
# Difficult to prevent all bad things



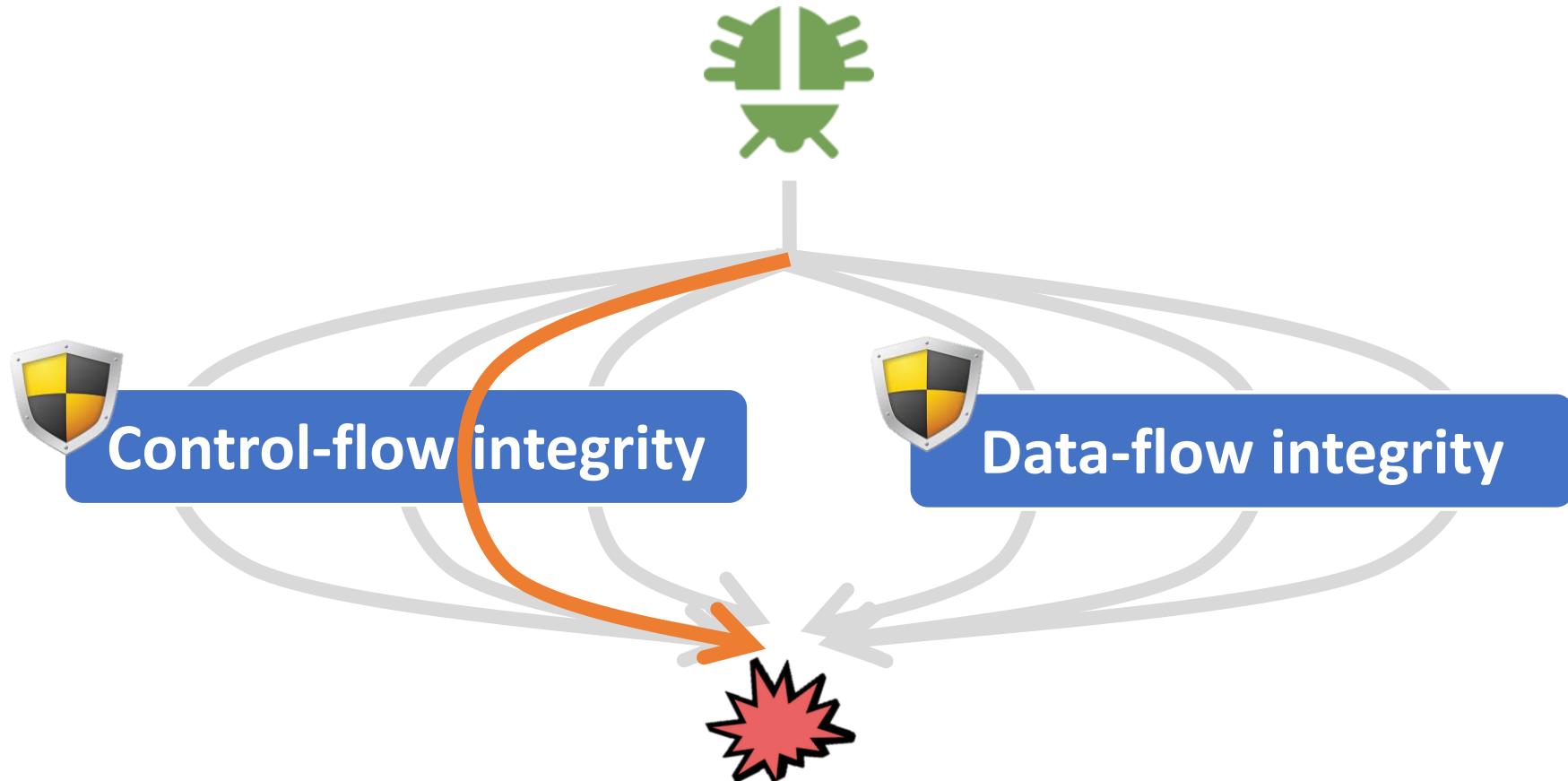
# Difficult to prevent all bad things



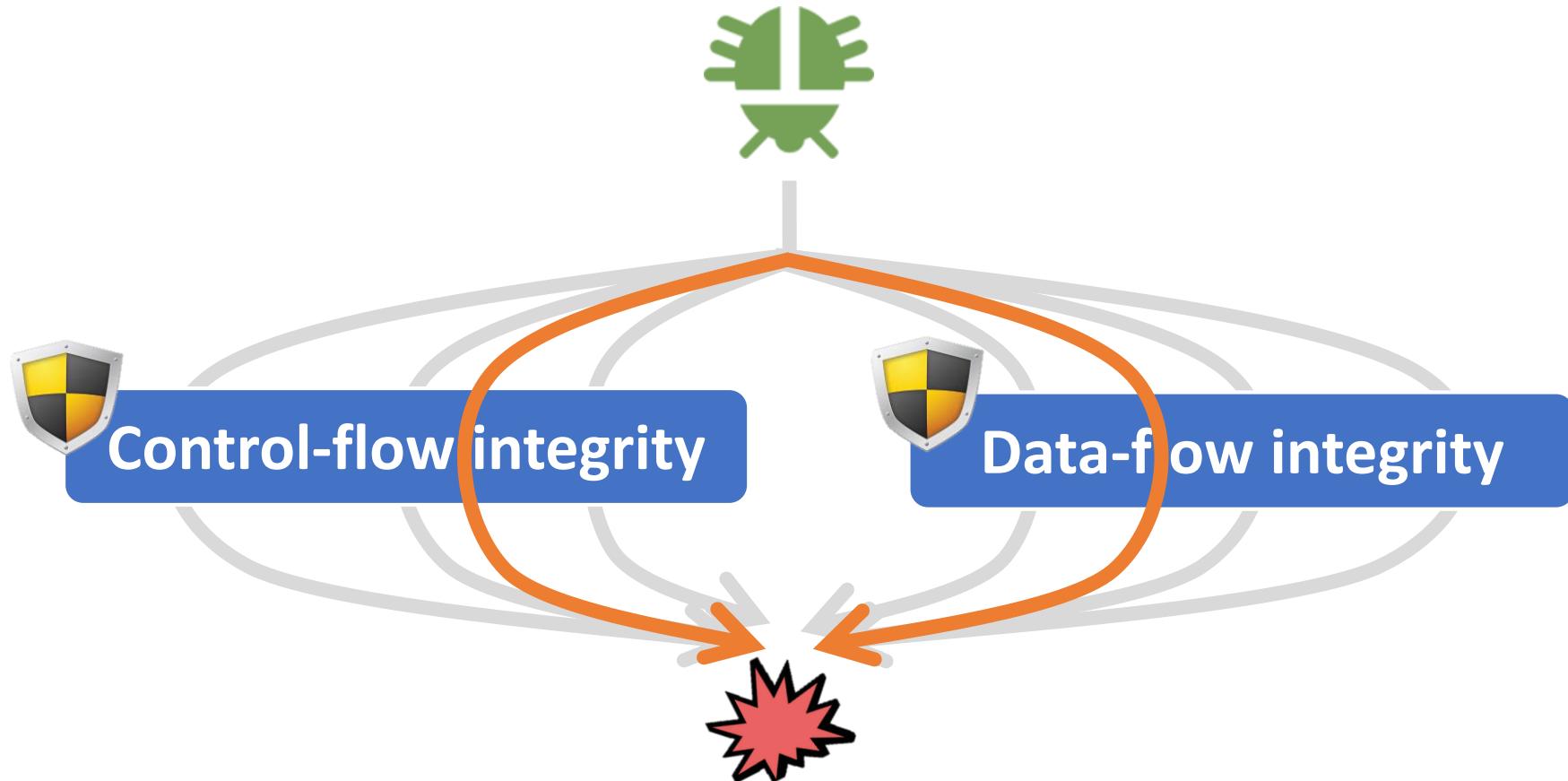
# Difficult to prevent all bad things



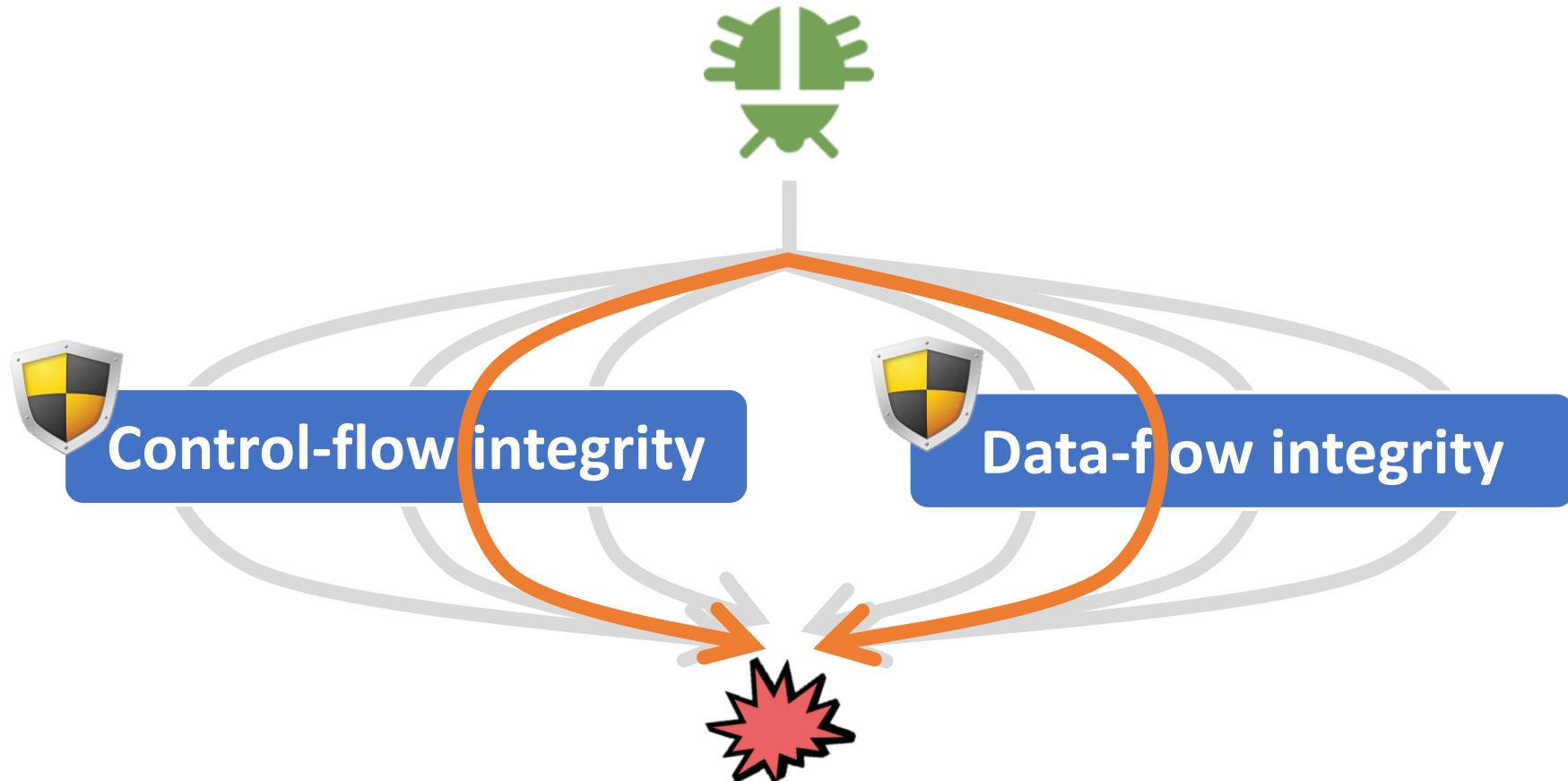
# Difficult to prevent all bad things



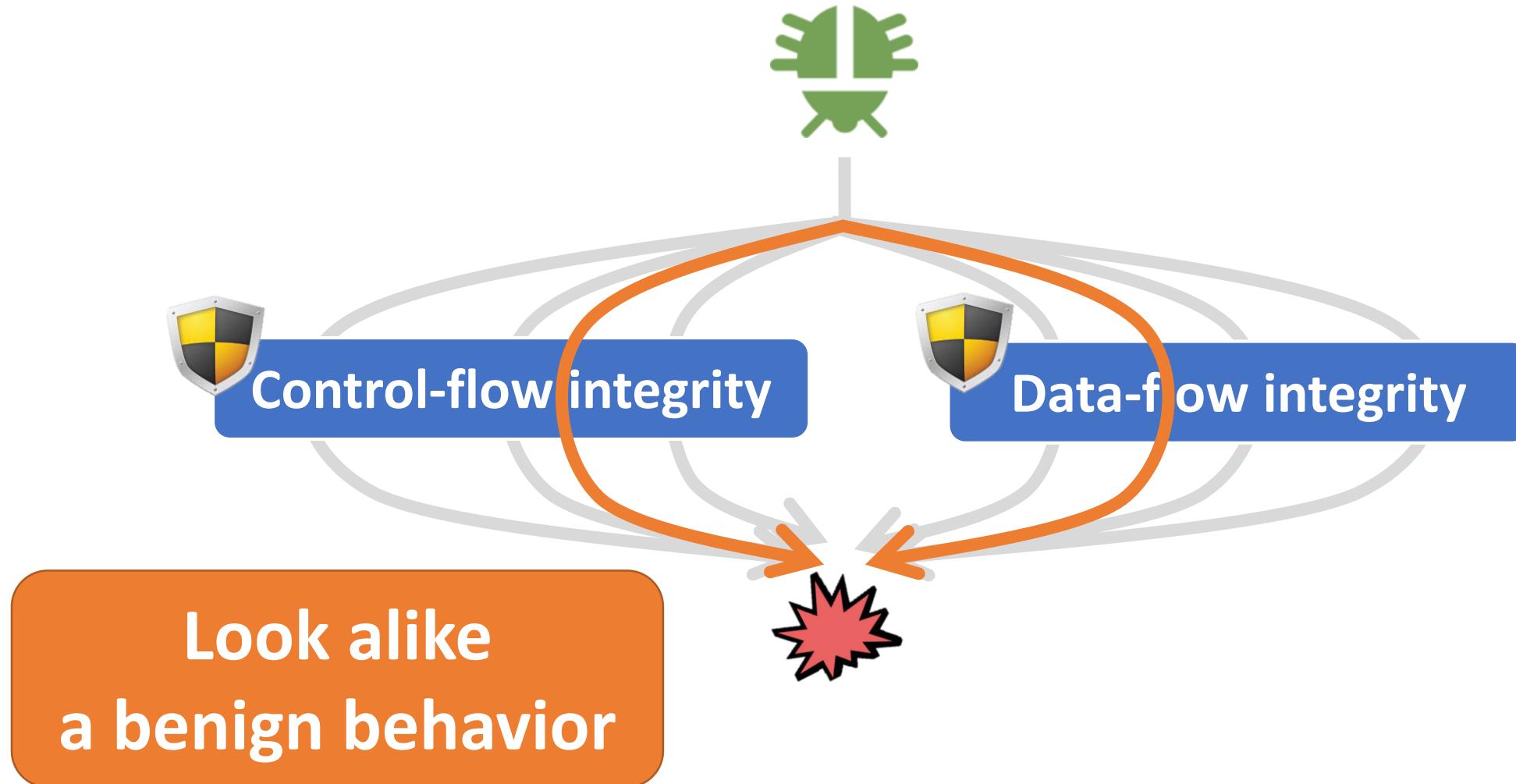
# Difficult to prevent all bad things



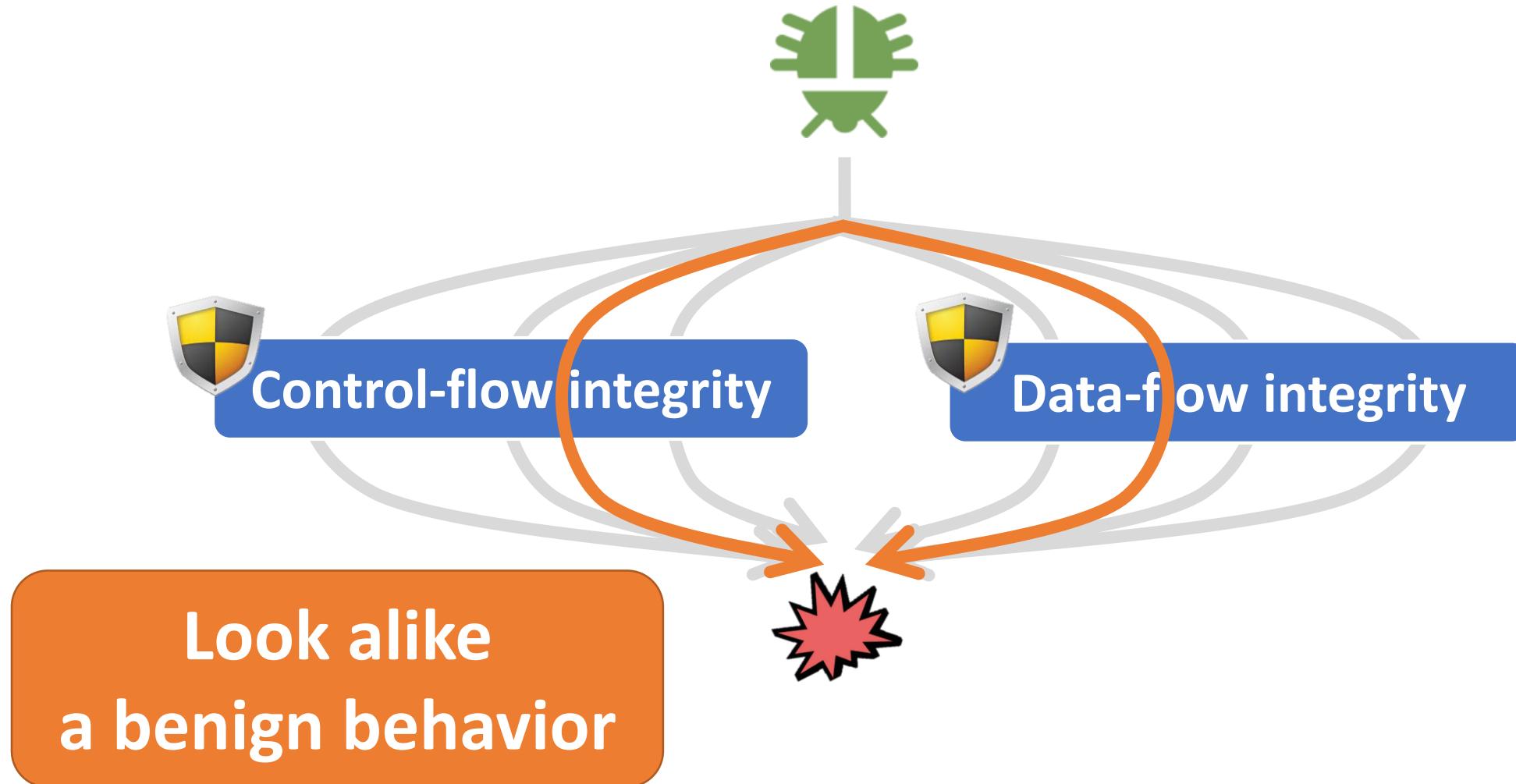
# Difficult to prevent all bad things



# Difficult to prevent all bad things



# Difficult to prevent all bad things



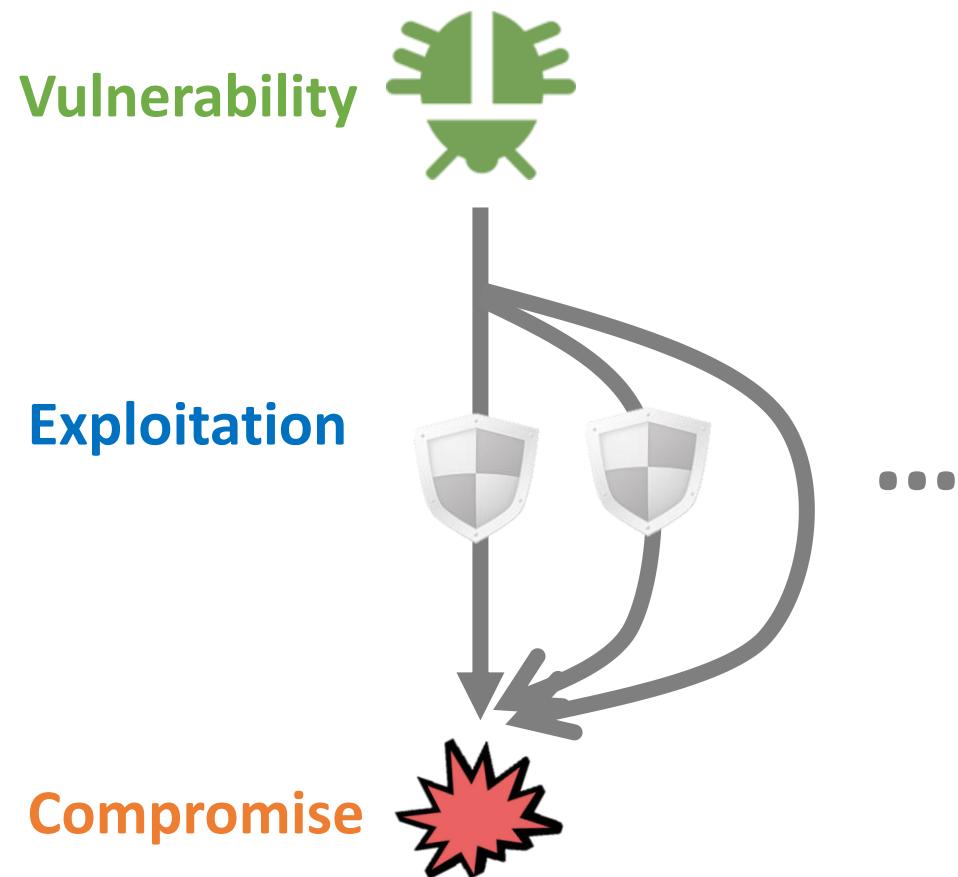
# Difficult to prevent all bad things

Difficult to know all legitimate control- and data-flows 😞

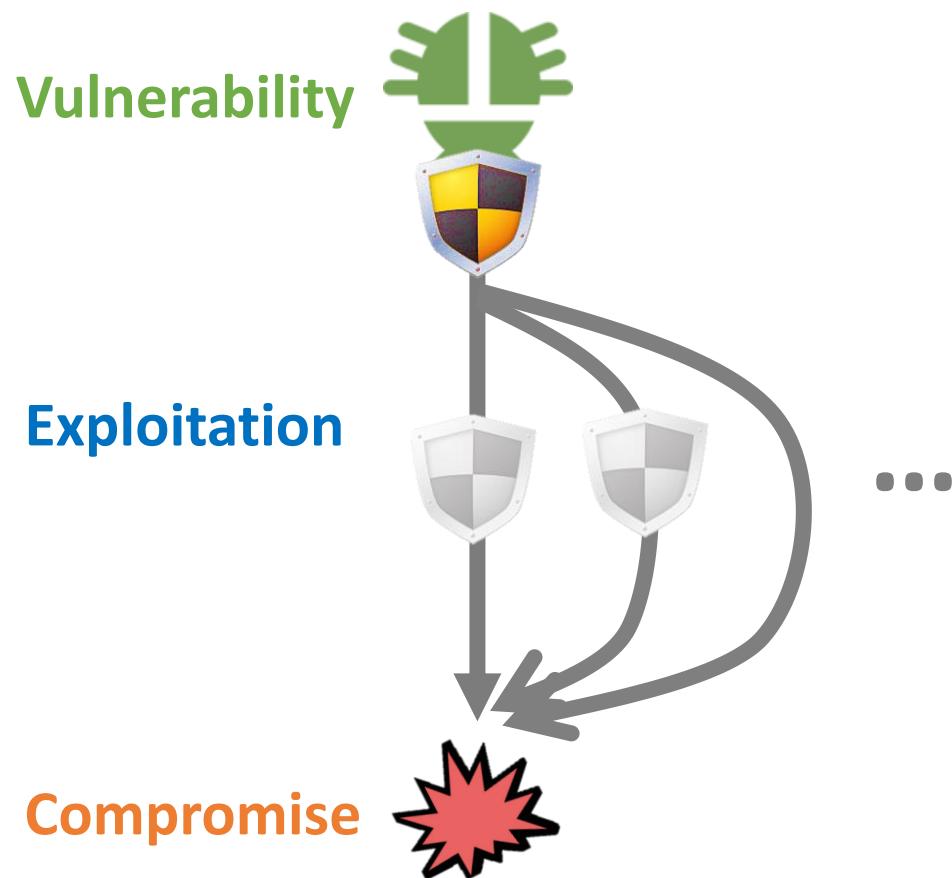


Look alike  
a benign behavior

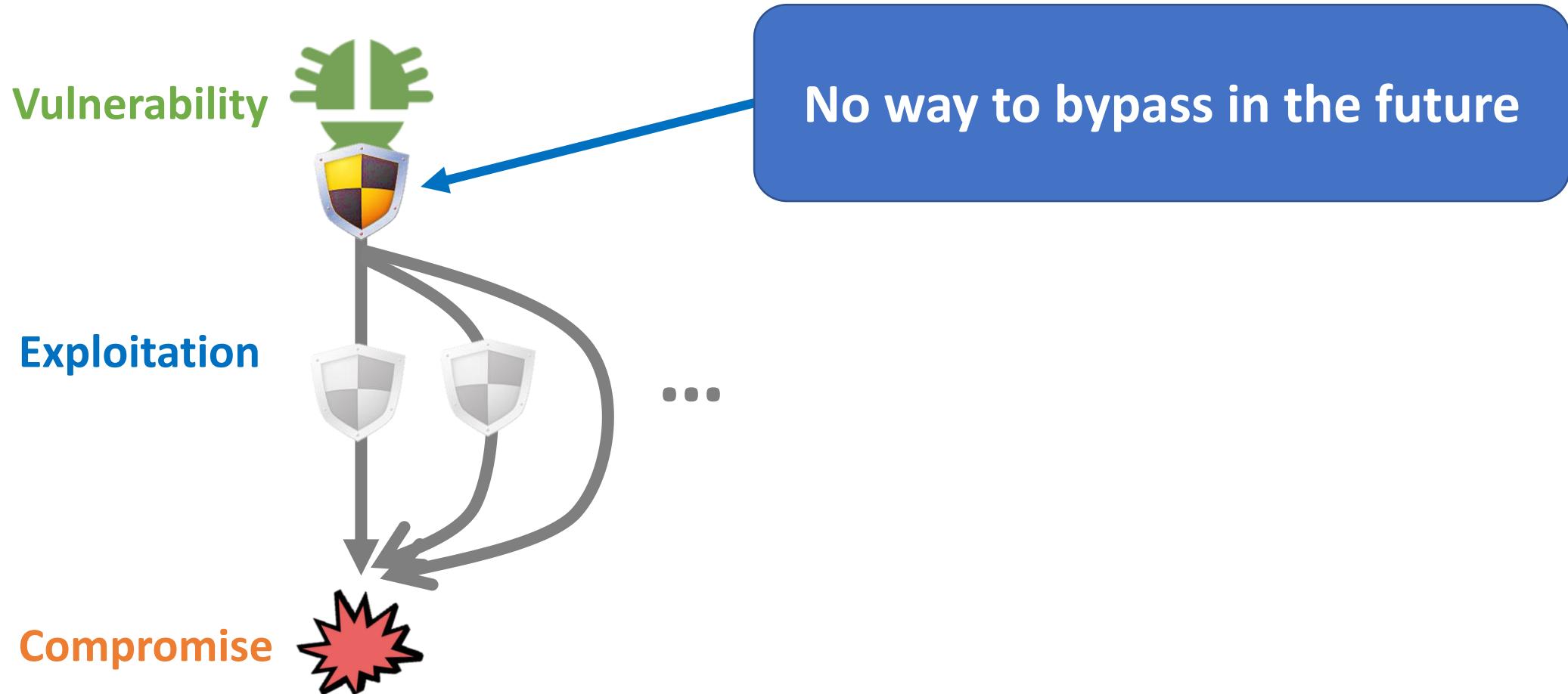
# Eliminating vulnerabilities



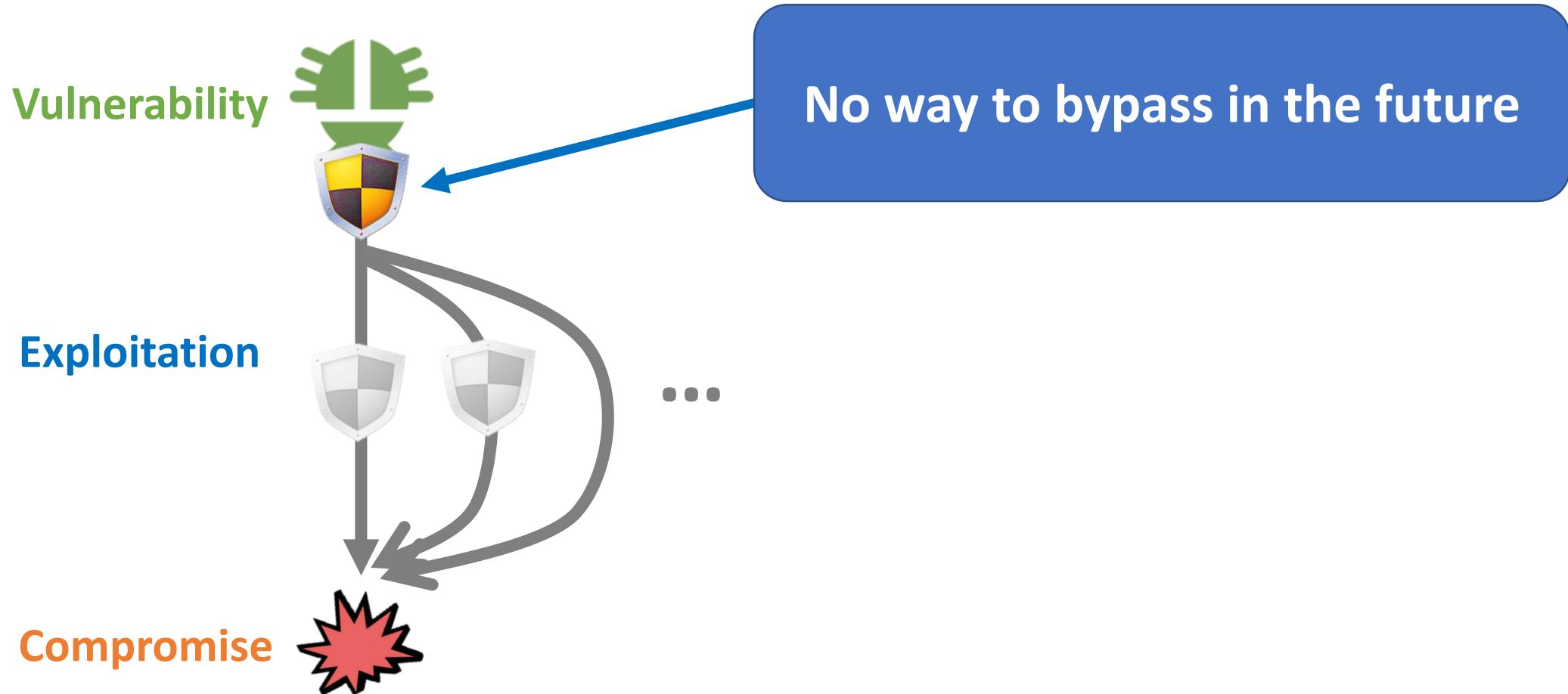
# Eliminating vulnerabilities



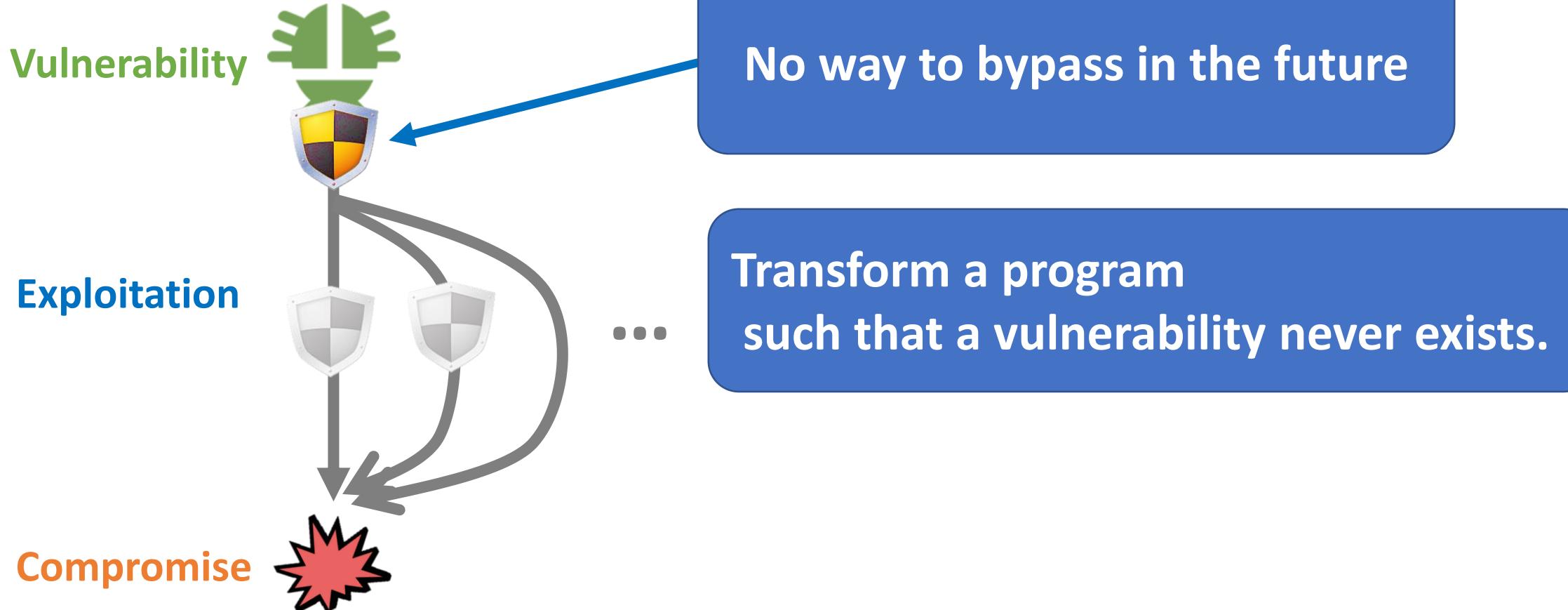
# Eliminating vulnerabilities



# Eliminating vulnerabilities



# Eliminating vulnerabilities



## 1. Eliminating vulnerabilities

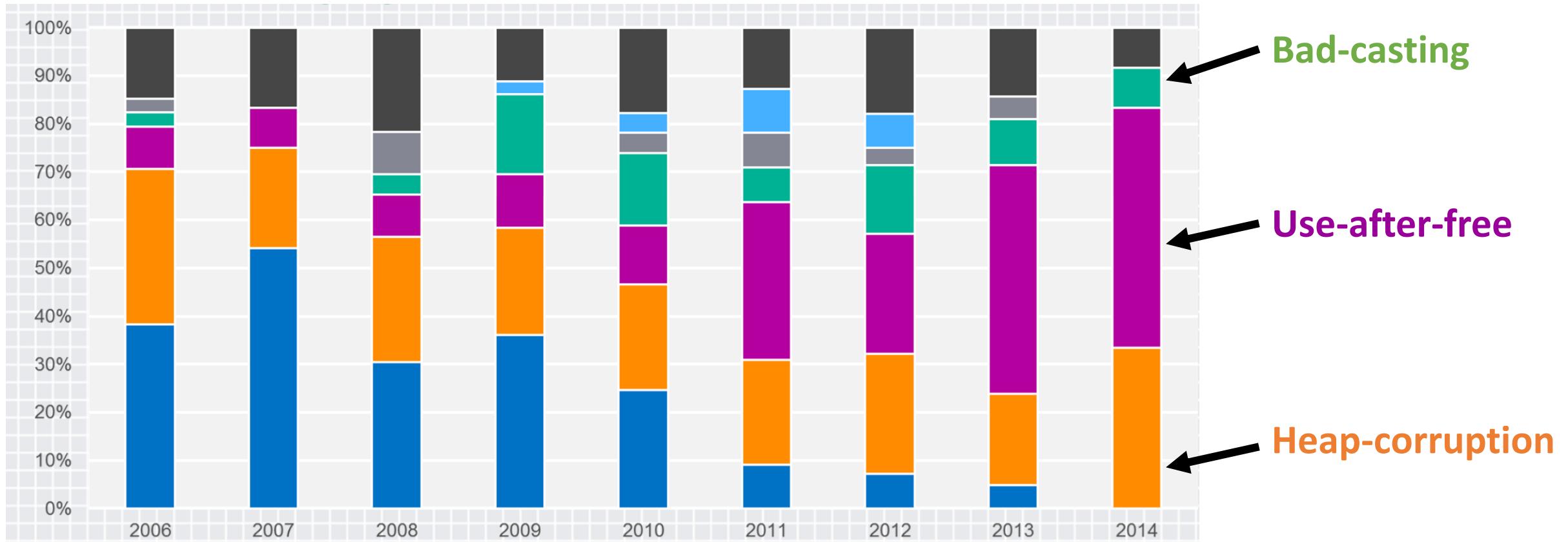
**DangNull [NDSS 15]: Eliminating use-after-free vulnerabilities**

**CaVer [Security 15]: Eliminating bad-casting vulnerabilities**

## 2. Analyzing vulnerabilities

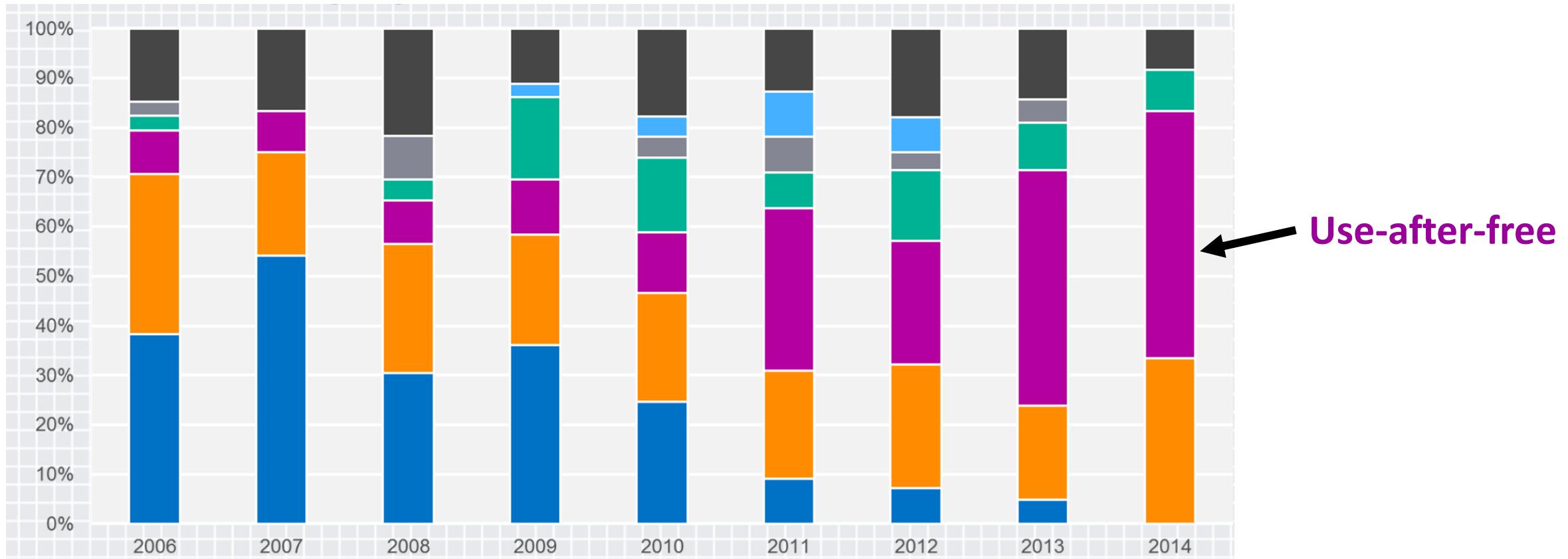
**SideFinder: Analyzing timing-channel vulnerabilities**

# Vulnerabilities in Microsoft products



Exploitation Trends: From Potential Risk to Actual Risk, Microsoft

# Vulnerabilities in Microsoft products



Exploitation Trends: From Potential Risk to Actual Risk, Microsoft

# Use-after-free

- Root cause: a dangling pointer
  - A pointer points to a freed memory region
- Using a dangling pointer leads to undefined program states
  - Easy to achieve arbitrary code executions
  - so called use-after-free

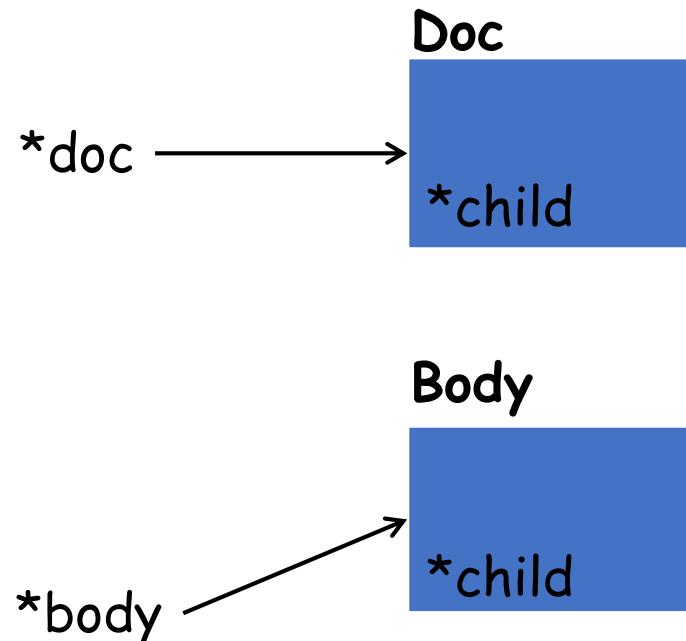
# Understanding use-after-free

A simplified use-after-free example from Chromium

```
class Doc : public Element {  
    // ...  
    Element *child;  
};  
  
class Body : public Element {  
    // ...  
    Element *child;  
};
```

```
Doc *doc = new Doc();  
Body *body = new Body();  
  
doc->child = body;  
  
delete body;  
  
doc->child->getAlign();
```

# Understanding use-after-free (in detail)



Allocate objects

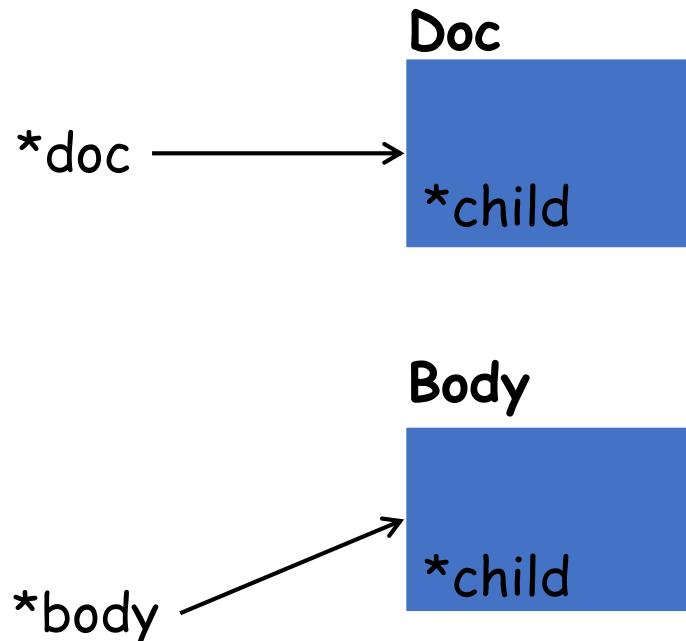
```
Doc *doc = new Doc();  
Body *body = new Body();
```

```
doc->child = body;
```

```
delete body;
```

```
doc->child->getAlign();
```

# Understanding use-after-free (in detail)



Allocate objects

```
Doc *doc = new Doc();
Body *body = new Body();
```

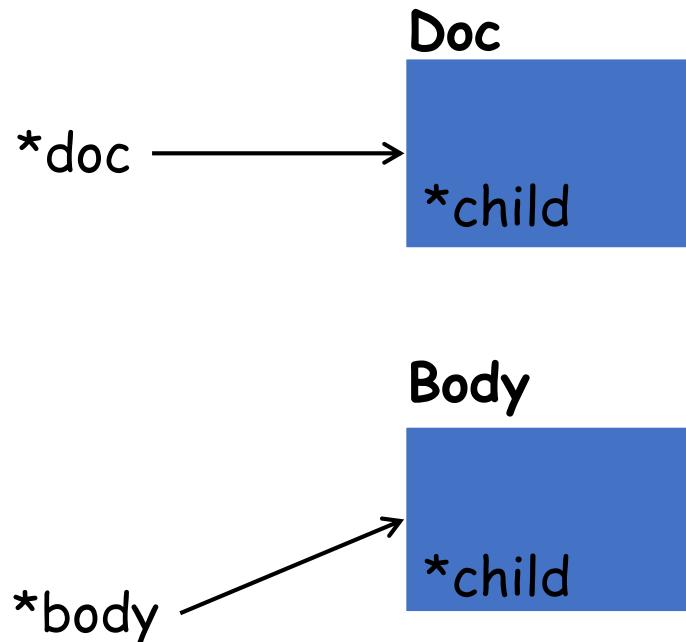
Propagate pointers

```
doc->child = body;
```

```
delete body;
```

```
doc->child->getAlign();
```

# Understanding use-after-free (in detail)



Allocate objects

```
Doc *doc = new Doc();  
Body *body = new Body();
```

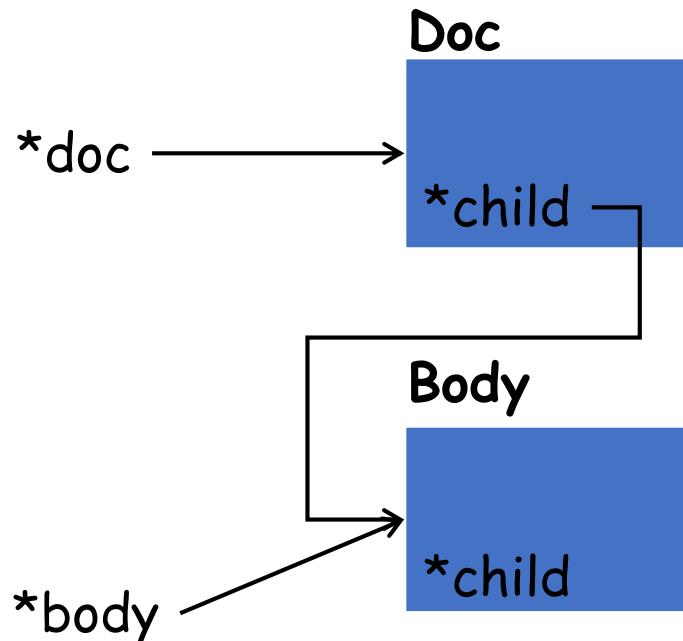
Propagate pointers

```
doc->child = body;
```

```
delete body;
```

```
doc->child->getAlign();
```

# Understanding use-after-free (in detail)



Allocate objects

```
Doc *doc = new Doc();
Body *body = new Body();
```

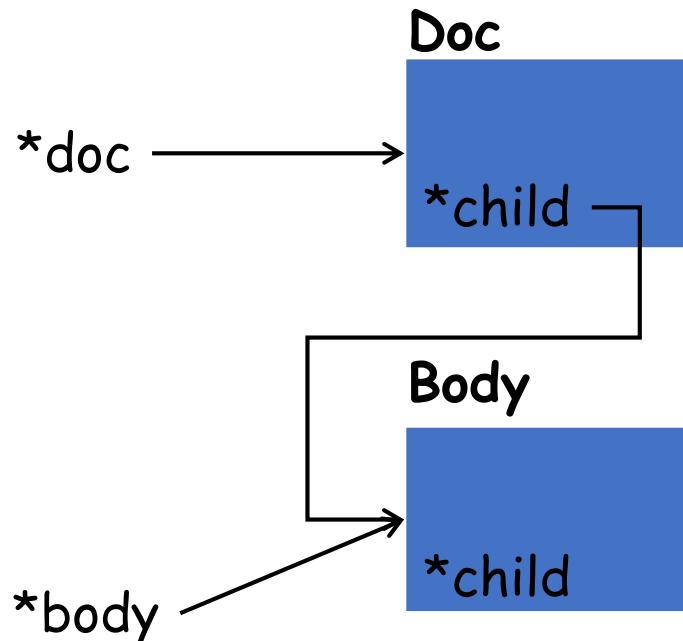
Propagate pointers

```
doc->child = body;
```

```
delete body;
```

```
doc->child->getAlign();
```

# Understanding use-after-free (in detail)



Allocate objects

```
Doc *doc = new Doc();
Body *body = new Body();
```

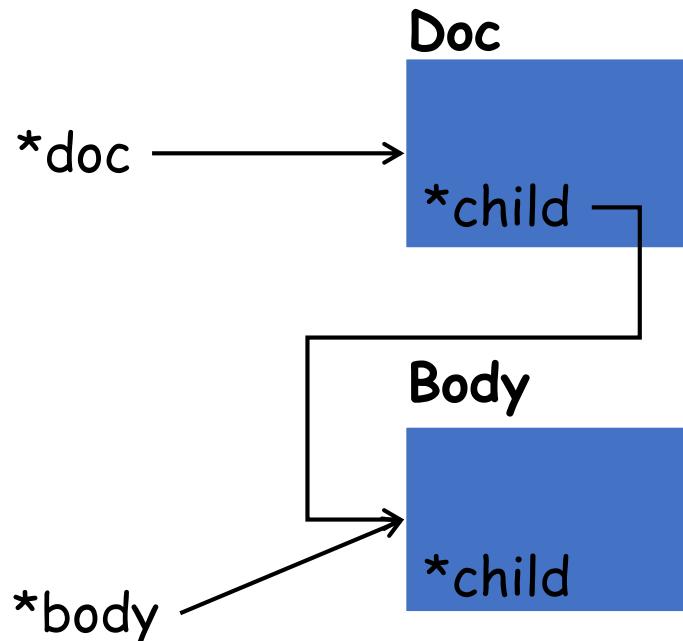
Propagate pointers

```
doc->child = body;
```

```
delete body;
```

```
doc->child->getAlign();
```

# Understanding use-after-free (in detail)



Allocate objects

```
Doc *doc = new Doc();
Body *body = new Body();
```

Propagate pointers

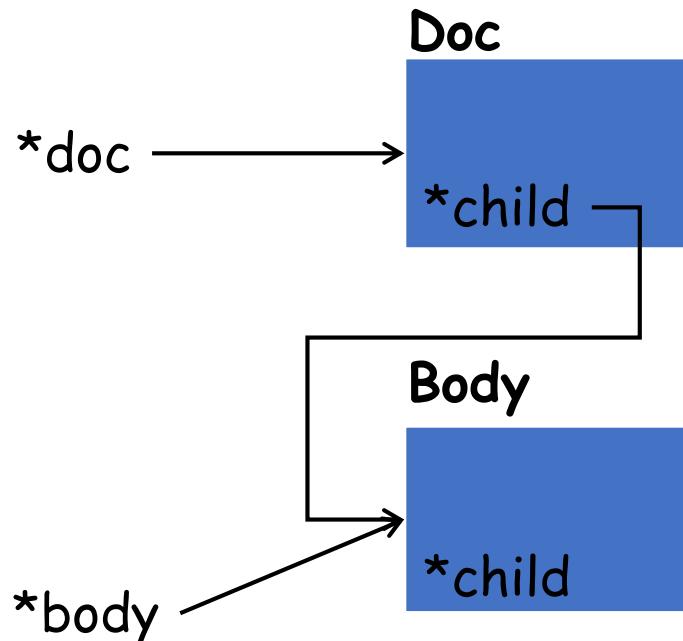
```
doc->child = body;
```

Free an object

```
delete body;
```

```
doc->child->getAlign();
```

# Understanding use-after-free (in detail)



Allocate objects

```
Doc *doc = new Doc();
Body *body = new Body();
```

Propagate pointers

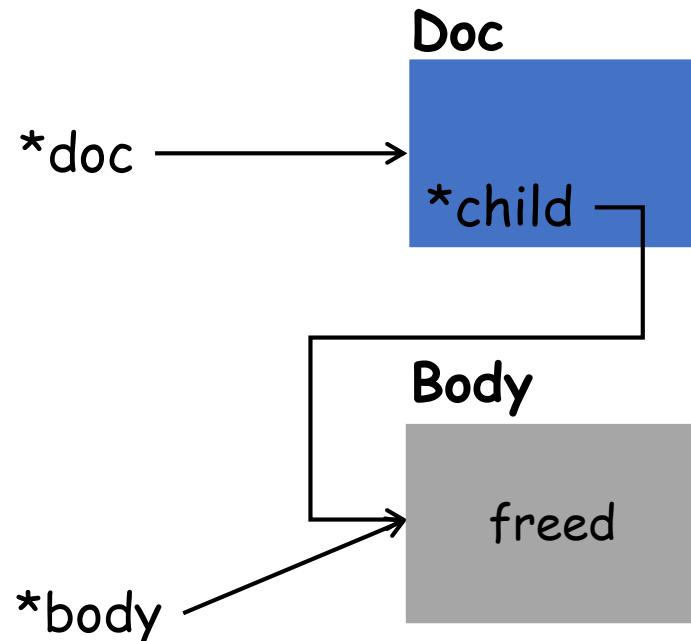
```
doc->child = body;
```

Free an object

```
delete body;
```

```
doc->child->getAlign();
```

# Understanding use-after-free (in detail)



Allocate objects

```
Doc *doc = new Doc();  
Body *body = new Body();
```

Propagate pointers

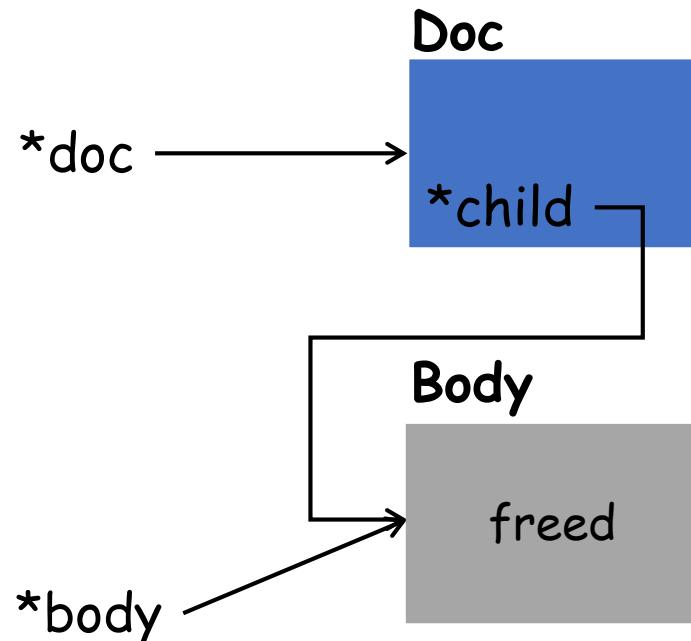
```
doc->child = body;
```

Free an object

```
delete body;
```

```
doc->child->getAlign();
```

# Understanding use-after-free (in detail)



Allocate objects

```
Doc *doc = new Doc();  
Body *body = new Body();
```

Propagate pointers

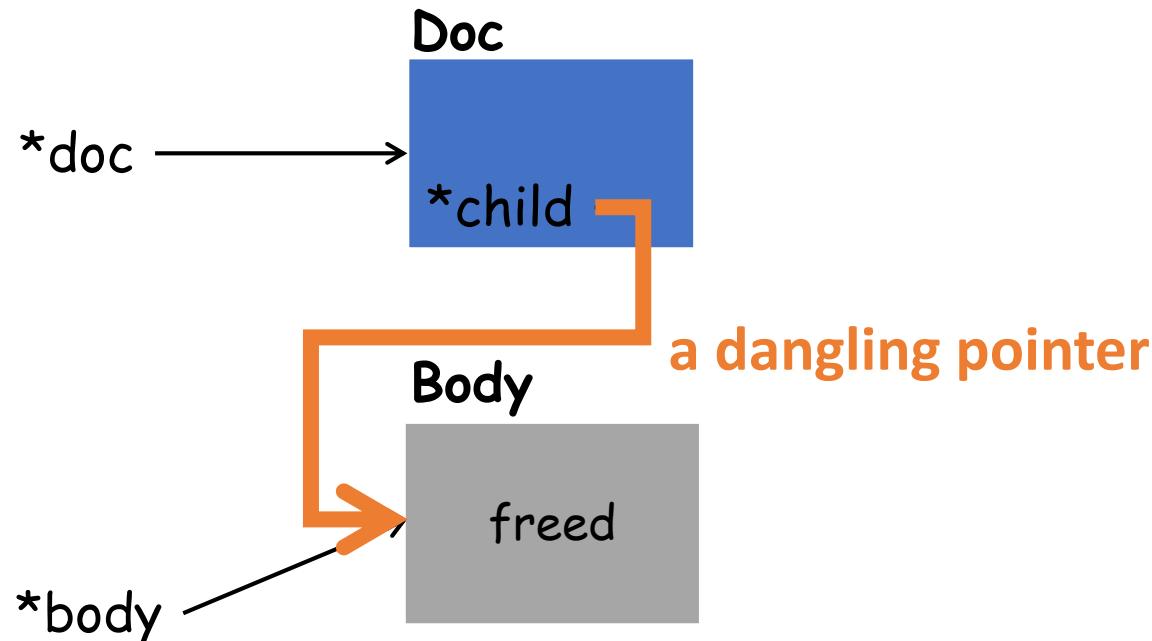
```
doc->child = body;
```

Free an object

```
delete body;
```

```
doc->child->getAlign();
```

# Understanding use-after-free (in detail)



Allocate objects

```
Doc *doc = new Doc();  
Body *body = new Body();
```

Propagate pointers

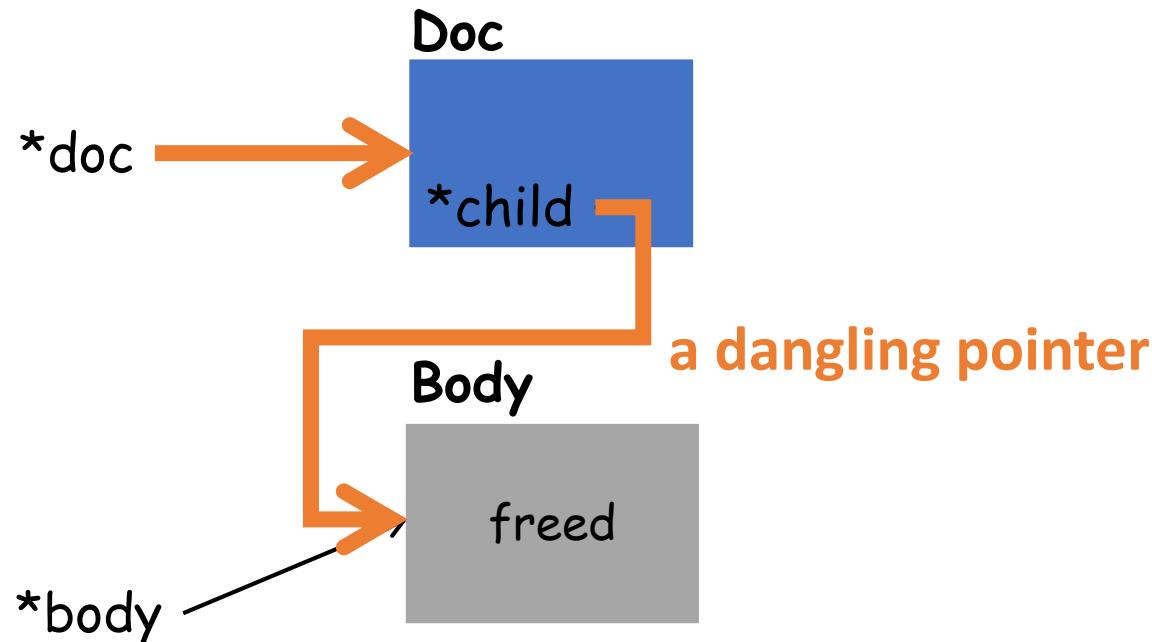
```
doc->child = body;
```

Free an object

```
delete body;
```

```
doc->child->getAlign();
```

# Understanding use-after-free (in detail)



Allocate objects

```
Doc *doc = new Doc();  
Body *body = new Body();
```

Propagate pointers

```
doc->child = body;
```

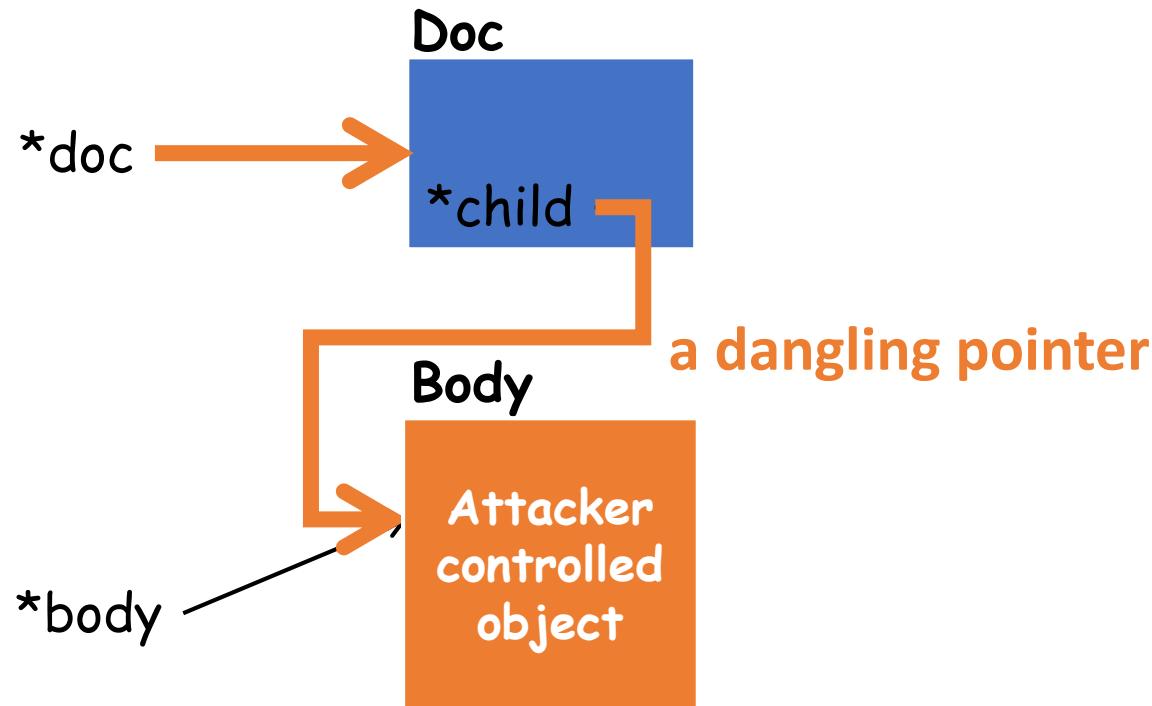
Free an object

```
delete body;
```

Use a dangling pointer

```
doc->child->getAlign();
```

# Understanding use-after-free (in detail)



Allocate objects

```
Doc *doc = new Doc();  
Body *body = new Body();
```

Propagate pointers

```
doc->child = body;
```

Free an object

```
delete body;
```

Use a dangling pointer

```
doc->child->getAlign();
```

# Challenges in identifying dangling pointers

```
Doc *doc = new Doc();
Body *body = new Body();

doc->child = body;

delete body;

doc->child->getAlign();
```

# Challenges in identifying dangling pointers

```
Doc *doc = new Doc();
Body *body = new Body();

doc->child = body;

delete body;

doc->child->getAlign();
```

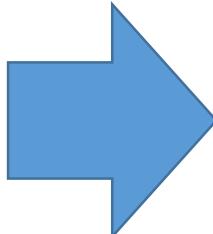
# Challenges in identifying dangling pointers

```
Doc *doc = new Doc();
Body *body = new Body();

doc->child = body;

delete body;

doc->child->getAlign();
```



```
Doc *doc = new Doc();
```

```
Body *body = new Body();
```

```
doc->child = body;
```

```
delete body;
```

```
doc->child->getAlign();
```

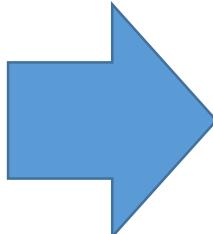
# Challenges in identifying dangling pointers

```
Doc *doc = new Doc();
Body *body = new Body();

doc->child = body;

delete body;

doc->child->getAlign();
```



```
Doc *doc = new Doc();
```

```
Body *body = new Body();
```

```
doc->child = body;
```

```
delete body;
```

```
doc->child->getAlign();
```

# Challenges in identifying dangling pointers

Static analysis: inter-procedural and points-to analysis

Dynamic analysis: precise pointer semantic tracking

doc->child->getAlign();

doc->child->getAlign();

# Challenges in identifying dangling pointers

Static analysis: inter-procedural and points-to analysis

Dynamic analysis: precise pointer semantic tracking

doc->child->getAlign();

doc->child->getAlign();

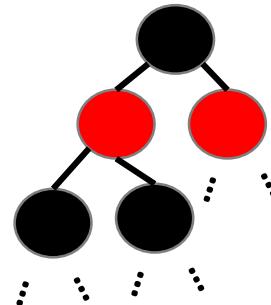
Difficult to scale for complex systems

# DangNull

- DangNull: Eliminating the root cause of use-after-free
- Design
  - Tracking Object Relationships
  - Nullifying dangling pointers

# Tracking object relationships

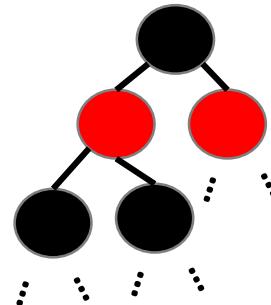
- Intercept allocations/deallocations in runtime
  - Maintain Shadow Object Tree
    - Red-Black tree to efficiently keep object layout information
    - Node: (base address, size) pair



# Tracking object relationships

- Intercept allocations/deallocations in runtime
  - Maintain Shadow Object Tree
    - Red-Black tree to efficiently keep object layout information
    - Node: (base address, size) pair

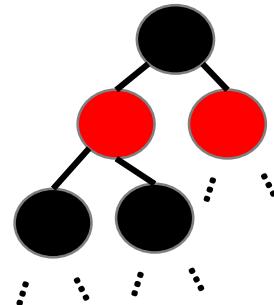
```
Doc *doc = new Doc();
```



# Tracking object relationships

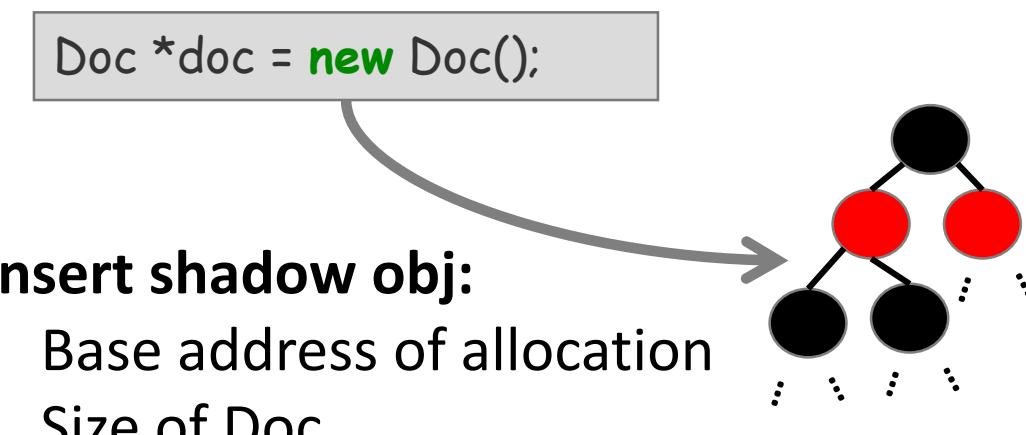
- Intercept allocations/deallocations in runtime
  - Maintain Shadow Object Tree
    - Red-Black tree to efficiently keep object layout information
    - Node: (base address, size) pair

```
Doc *doc = new Doc();
```



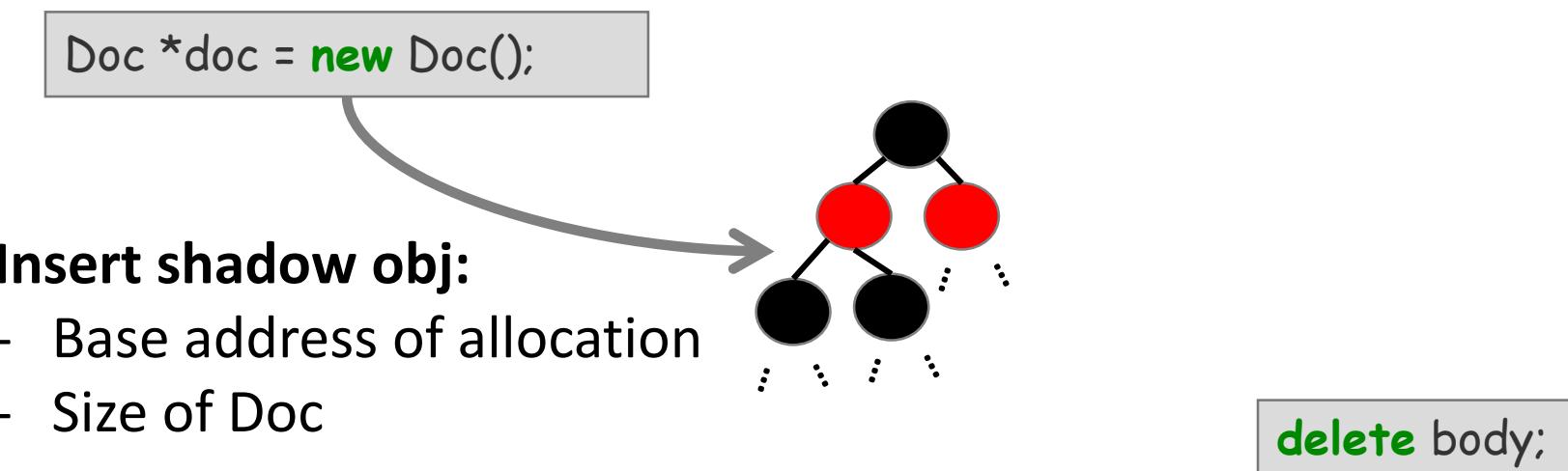
# Tracking object relationships

- Intercept allocations/deallocations in runtime
  - Maintain Shadow Object Tree
    - Red-Black tree to efficiently keep object layout information
    - Node: (base address, size) pair



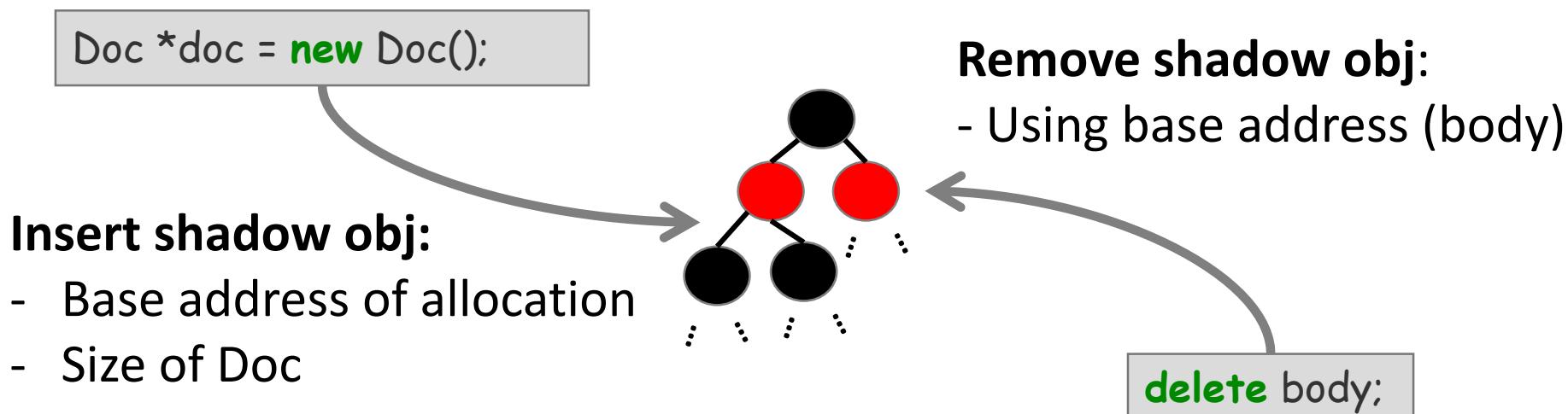
# Tracking object relationships

- Intercept allocations/deallocations in runtime
  - Maintain Shadow Object Tree
    - Red-Black tree to efficiently keep object layout information
    - Node: (base address, size) pair



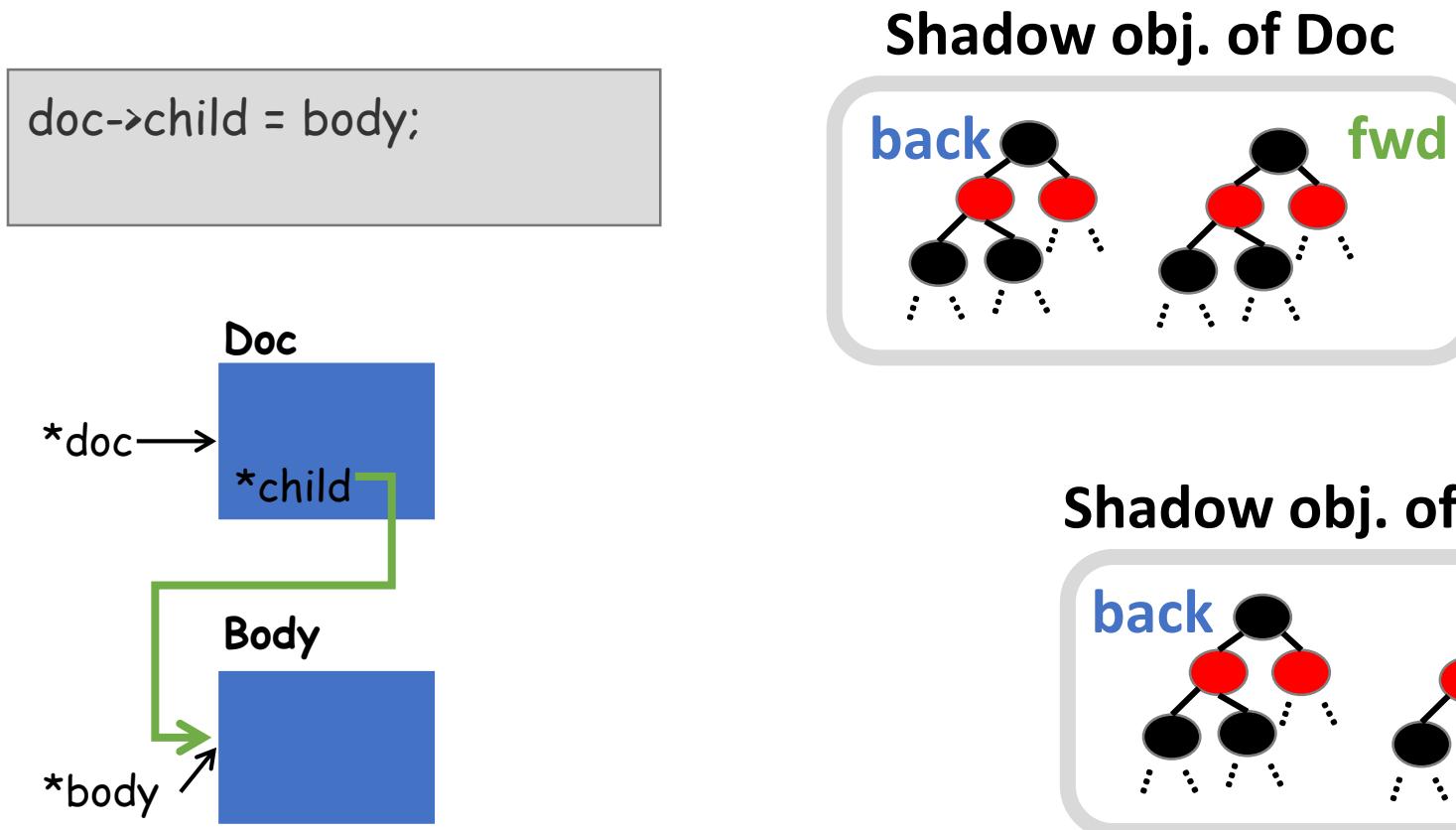
# Tracking object relationships

- Intercept allocations/deallocations in runtime
  - Maintain Shadow Object Tree
    - Red-Black tree to efficiently keep object layout information
    - Node: (base address, size) pair



# Tracking object relationships

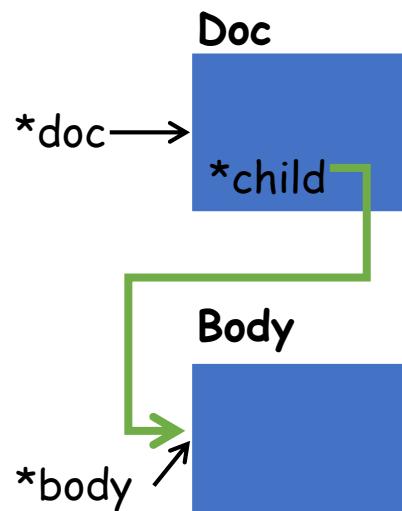
- Instrument pointer propagations
  - Maintain backward/forward pointer trees for a shadow obj



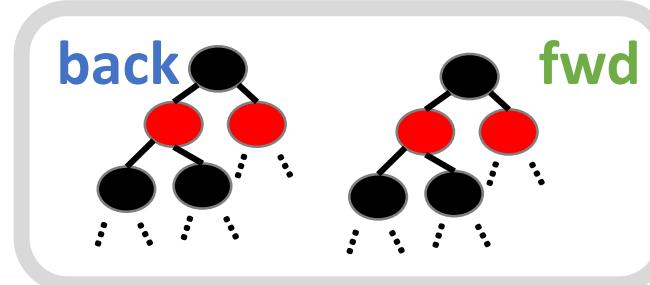
# Tracking object relationships

- Instrument pointer propagations
  - Maintain backward/forward pointer trees for a shadow obj

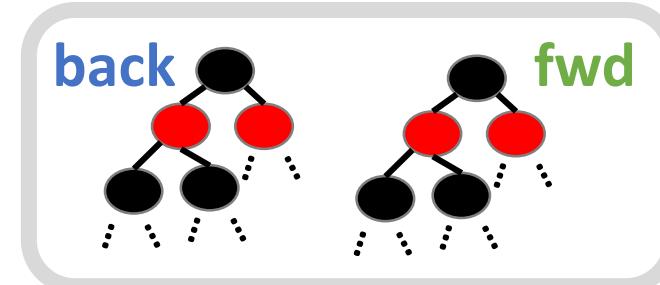
```
doc->child = body;  
trace(&doc->child, body);
```



Shadow obj. of Doc



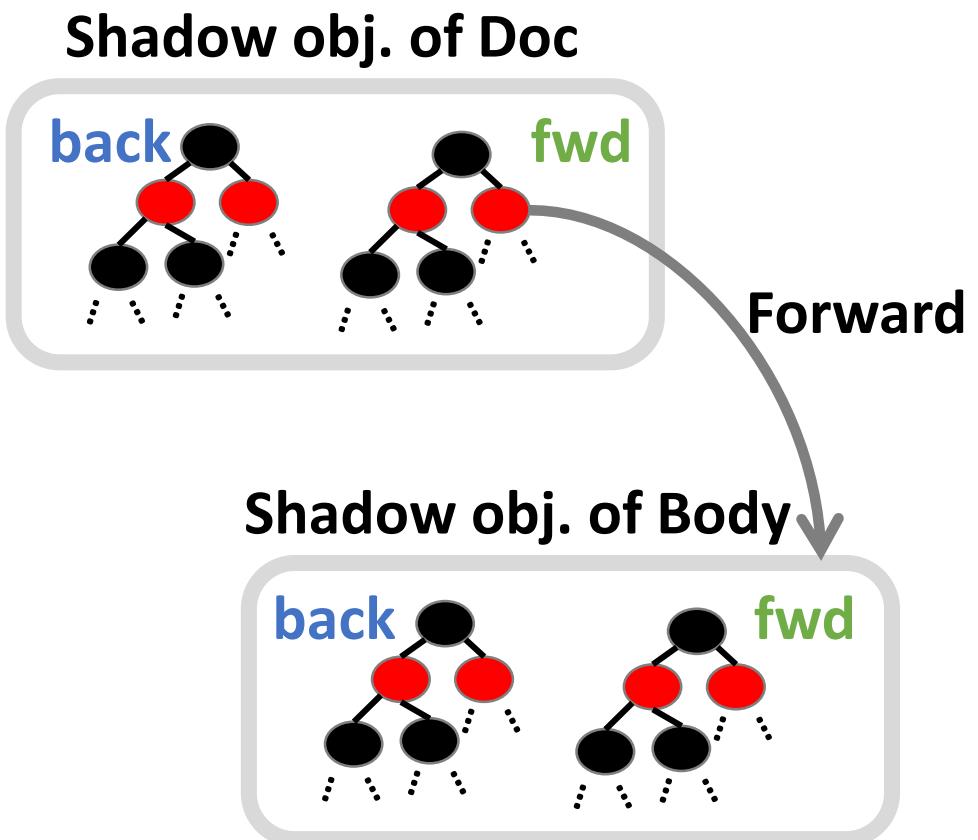
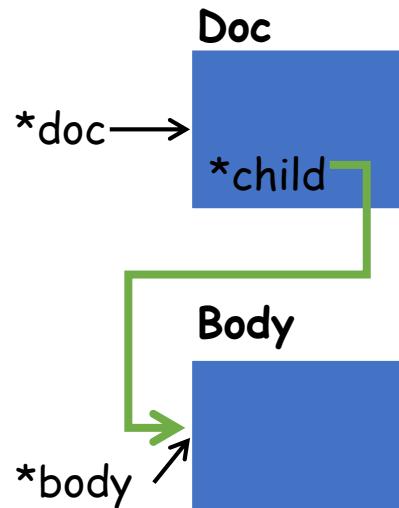
Shadow obj. of Body



# Tracking object relationships

- Instrument pointer propagations
  - Maintain backward/forward pointer trees for a shadow obj

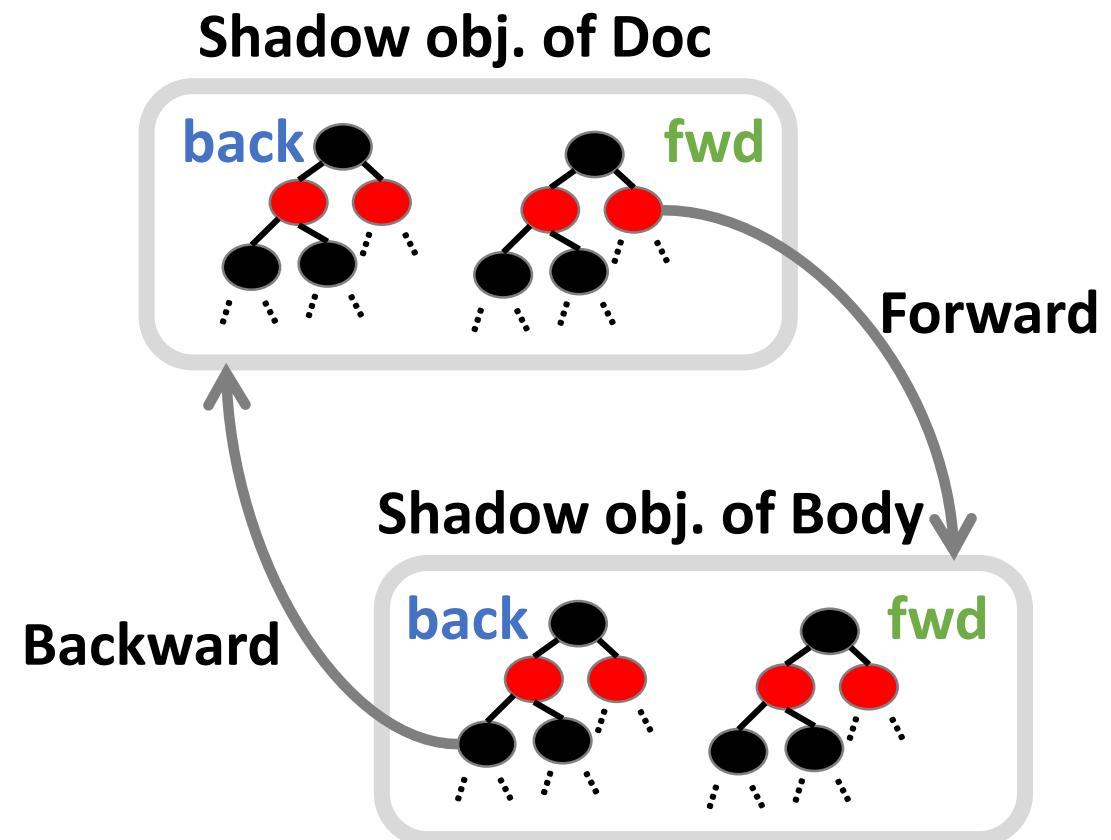
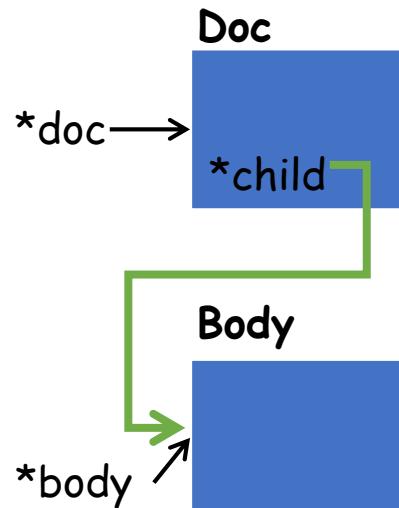
```
doc->child = body;  
trace(&doc->child, body);
```



# Tracking object relationships

- Instrument pointer propagations
  - Maintain backward/forward pointer trees for a shadow obj

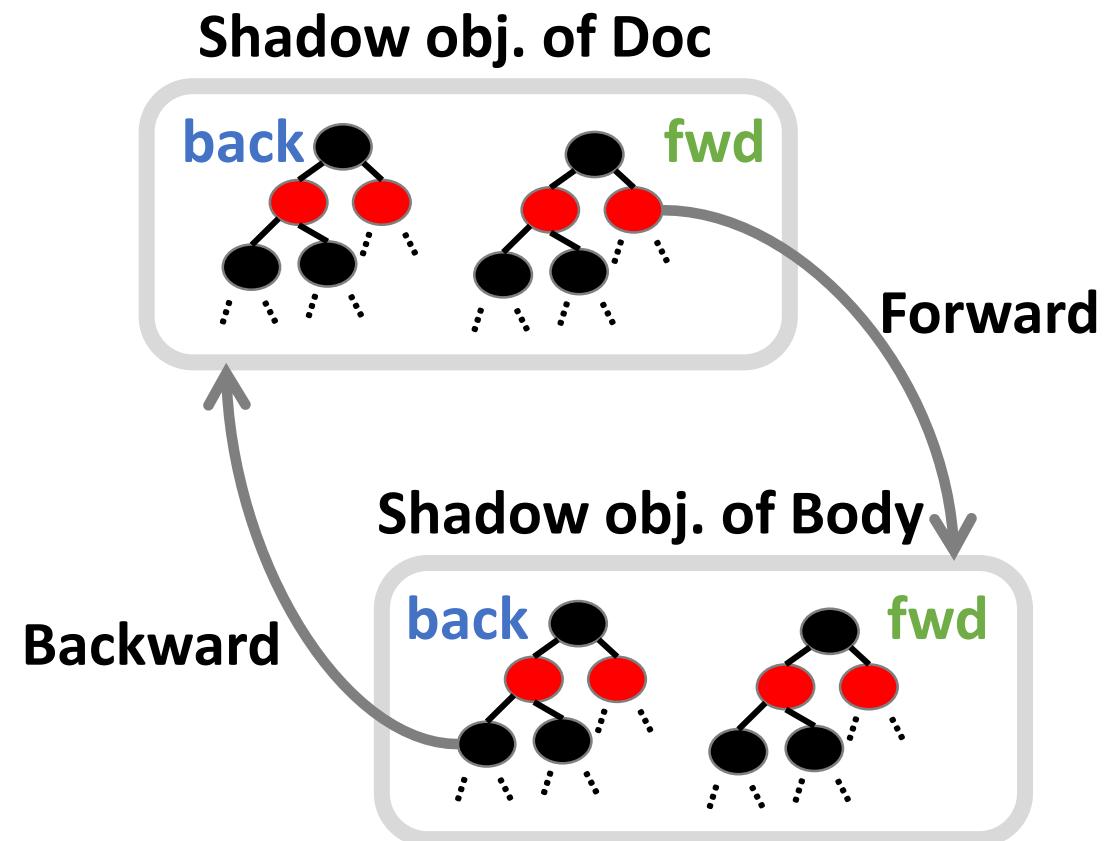
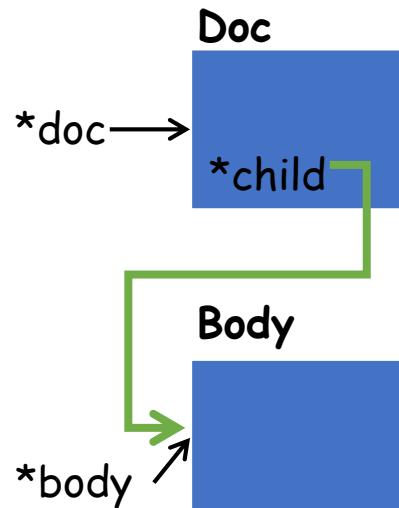
```
doc->child = body;  
trace(&doc->child, body);
```



# Tracking object relationships

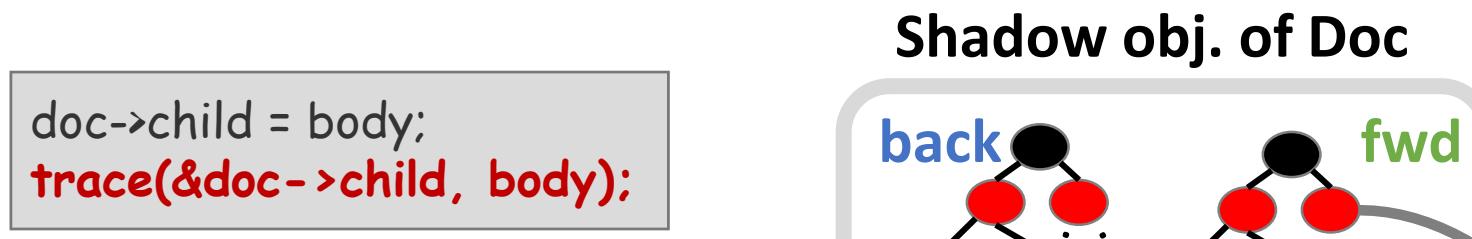
- Instrument pointer propagations
  - Maintain backward/forward pointer trees for a shadow obj

```
doc->child = body;  
trace(&doc->child, body);
```



# Tracking object relationships

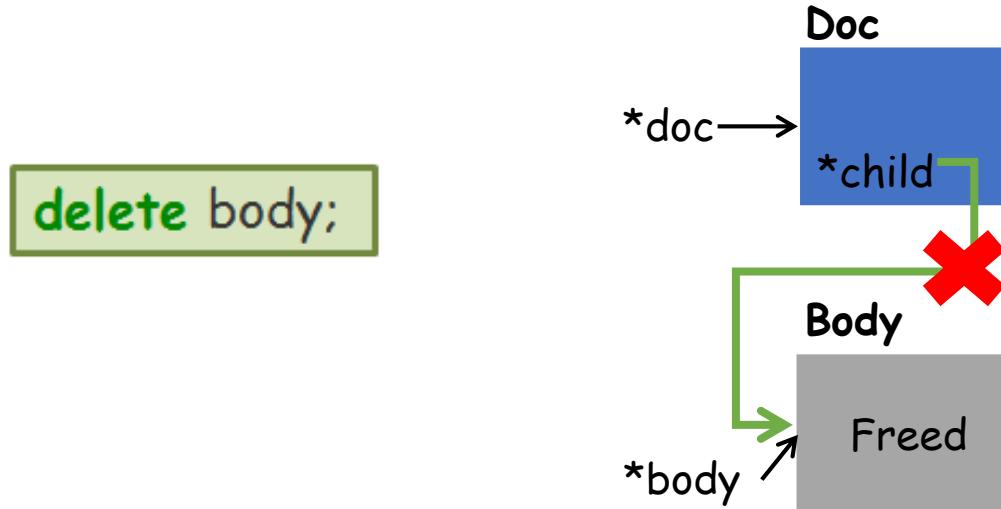
- Instrument pointer propagations
  - Maintain backward/forward pointer trees for a shadow obj



This is heavily abstracted pointer semantic tracking,  
but it is enough to identify all dangling pointers

# Nullifying dangling pointers

- Nullify all backward pointers once the target object is freed
  - All backward pointers are dangling pointers
  - Dangling pointers have no semantics



`delete body;`

# Implementation

- Prototype of DangNull
  - Instrumentation: LLVM pass, +389 LoC
  - Runtime: compiler-rt, +3,955 LoC
- Target applications
  - SPEC CPU 2006: one extra compiler and linker flag
  - Chromium: +27 LoC to .gyp build configuration file

# Evaluation on Chromium

- Runtime overheads
  - 4.8% and 53.1% overheads in JavaScript and rendering benchmarks, respectively
  - 7% increased page loading time for Alexa top 100 websites
- Safely prevented 7 real-world use-after-free exploits in Chrome

## 1. Eliminating vulnerabilities

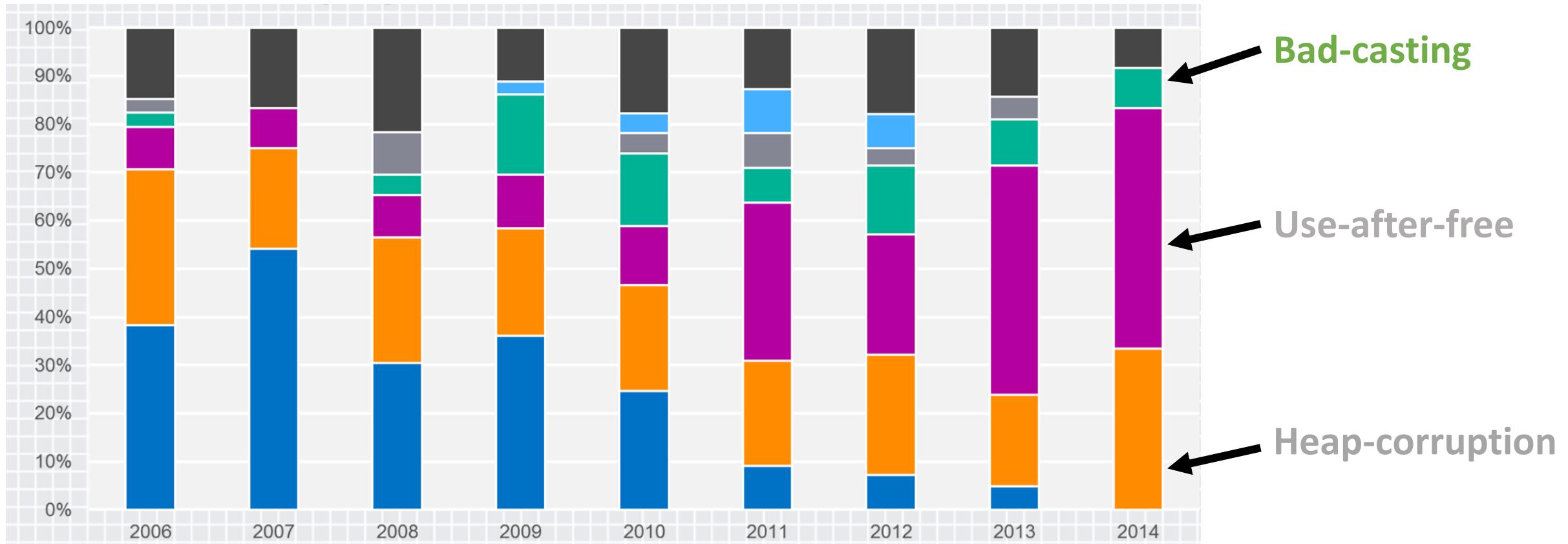
**DangNull [NDSS 15]: Eliminating use-after-free vulnerabilities**

**CaVer [Security 15]: Eliminating bad-casting vulnerabilities**

## 2. Analyzing vulnerabilities

**SideFinder: Analyzing timing-channel vulnerabilities**

# Vulnerabilities in Microsoft products



Exploitation Trends: From Potential Risk to Actual Risk, Microsoft

# Type conversions in C++

- `static_cast`
  - Compile-time conversions
  - Fast: no extra type verification in run-time
- `dynamic_cast`
  - Run-time conversions
  - Requires Runtime Type Information (RTTI)
  - Slow: Extra verification by parsing RTTI
  - Typically prohibited in performance critical applications

# Upcasting and Downcasting

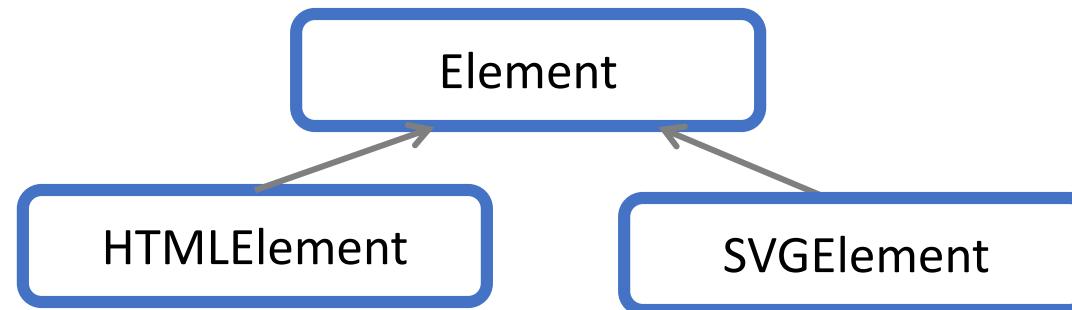
- Upcasting
  - From a derived class to its parent class
- Downcasting
  - From a parent class to one of its derived classes

# Upcasting and Downcasting

- Upcasting
  - From a derived class to its parent class
- Downcasting
  - From a parent class to one of its derived classes

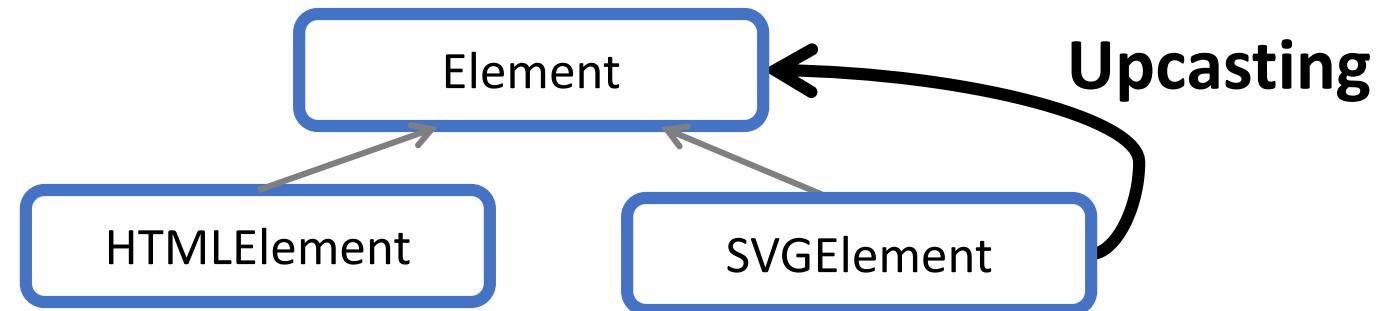
# Upcasting and Downcasting

- Upcasting
  - From a derived class to its parent class
- Downcasting
  - From a parent class to one of its derived classes



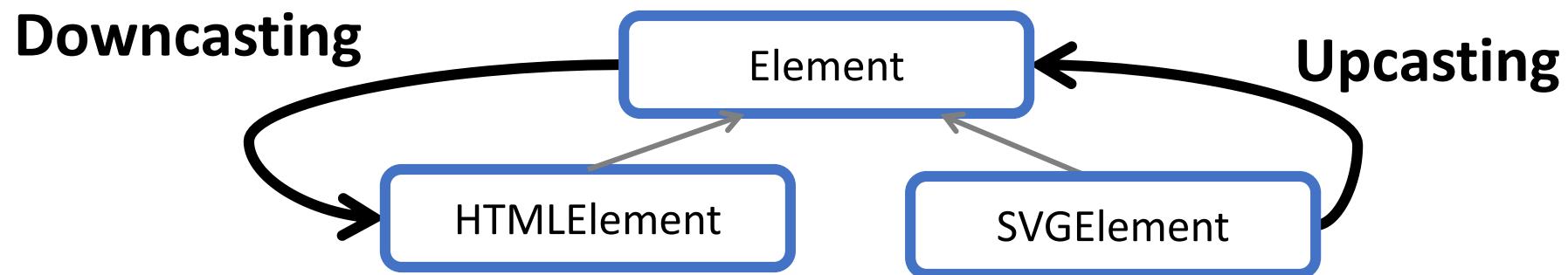
# Upcasting and Downcasting

- Upcasting
  - From a derived class to its parent class
- Downcasting
  - From a parent class to one of its derived classes



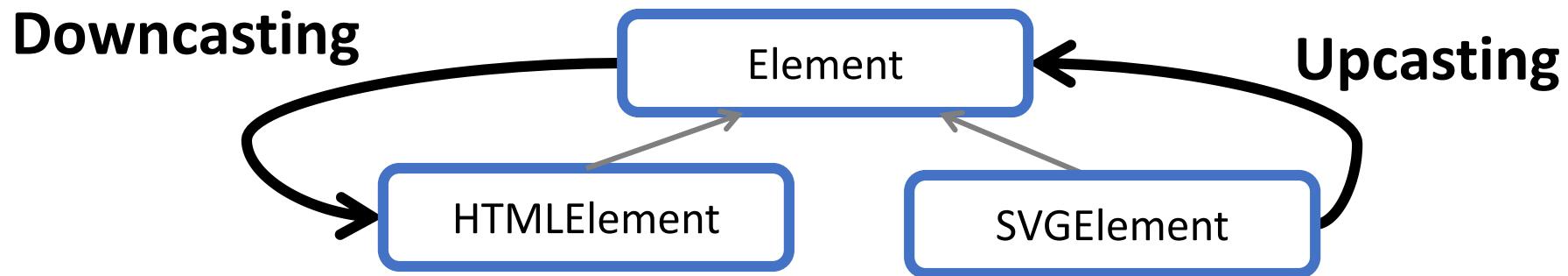
# Upcasting and Downcasting

- Upcasting
  - From a derived class to its parent class
- Downcasting
  - From a parent class to one of its derived classes



# Upcasting and Downcasting

- Upcasting
  - From a derived class to its parent class
- Downcasting
  - From a parent class to one of its derived classes



**Upcasting is always safe,  
but downcasting is not!**

# Downcasting is not always safe!

```
class P {  
    virtual ~P() {}  
    int m_P;  
};
```

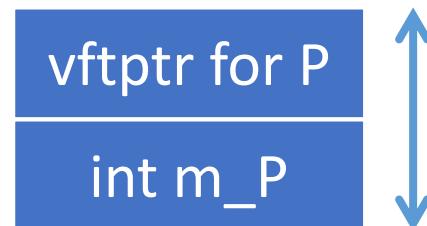
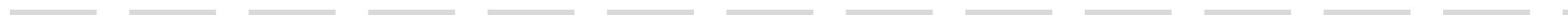
```
class D: public P {  
    virtual ~D() {}  
    int m_D;  
};
```



# Downcasting is not always safe!

```
class P {  
    virtual ~P() {}  
    int m_P;  
};
```

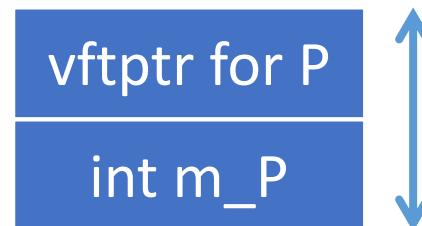
```
class D: public P {  
    virtual ~D() {}  
    int m_D;  
};
```



# Downcasting is not always safe!

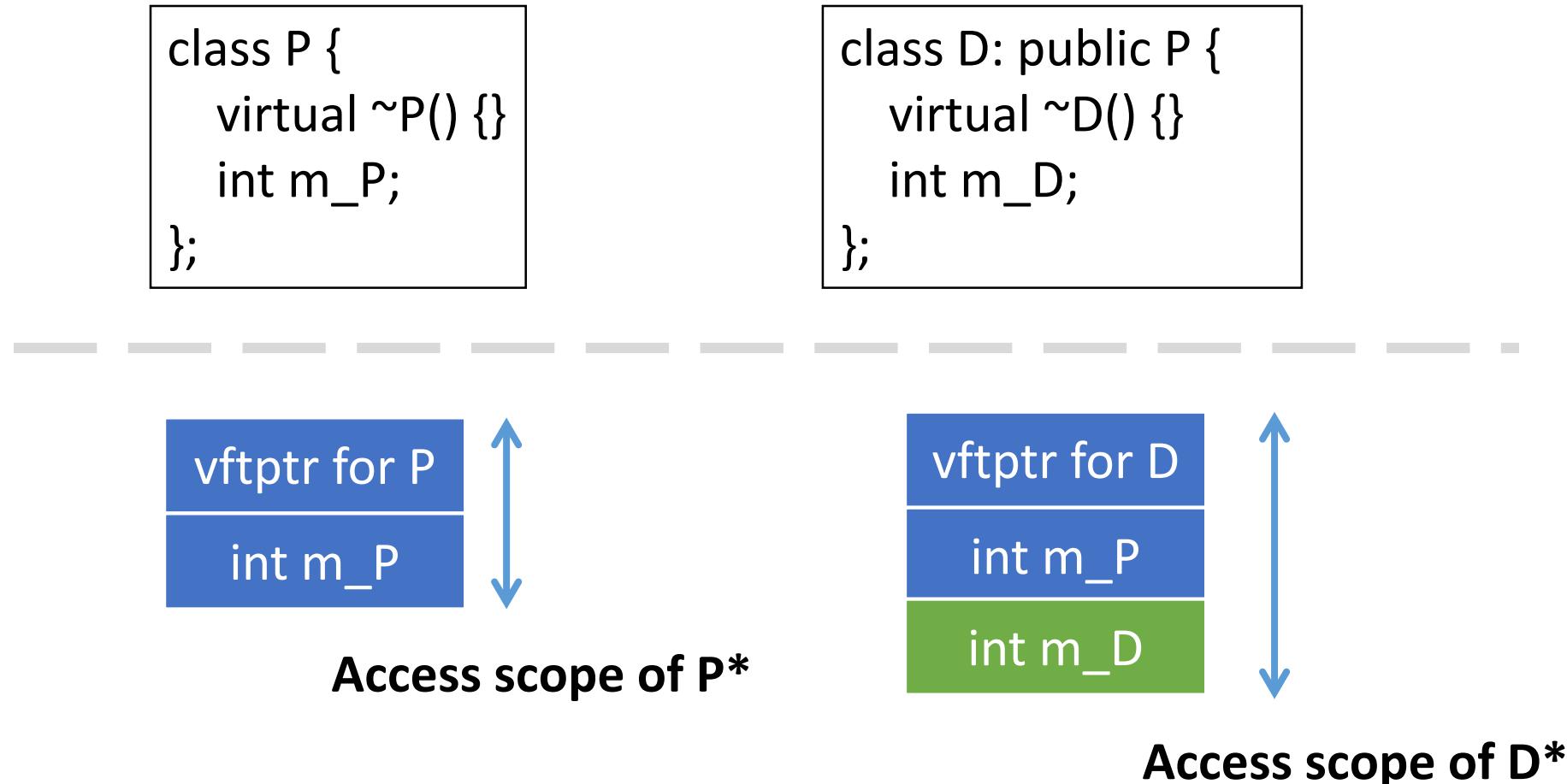
```
class P {  
    virtual ~P() {}  
    int m_P;  
};
```

```
class D: public P {  
    virtual ~D() {}  
    int m_D;  
};
```



**Access scope of P\***

# Downcasting is not always safe!



# Downcasting can be bad-casting

```
P *pS = new P();
D *pD = static_cast<D*>(pS);
pD->m_D;
```

vftptr for P

int m\_P

int m\_D

# Downcasting can be bad-casting

Bad-casting occurs: D is not a sub-object of P  
→ Undefined behavior

```
P *pS = new P();
D *pD = static_cast<D*>(pS);
pD->m_D;
```

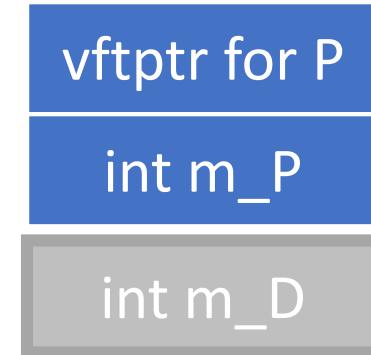
vftptr for P

int m\_P

int m\_D

# Downcasting can be bad-casting

```
P *pS = new P();
D *pD = static_cast<D*>(pS);
pD->m_D;
```



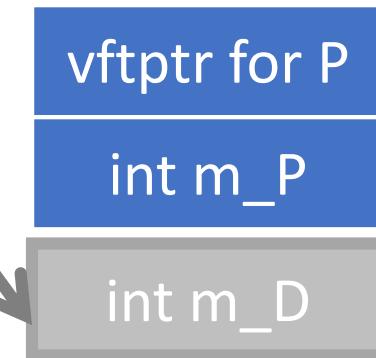
Memory corruptions

# Downcasting can be bad-casting

```
P *pS = new P();
D *pD = static_cast<D*>(pS);
pD->m_D;
```

Memory corruptions

$\&(pD \rightarrow m_D)$

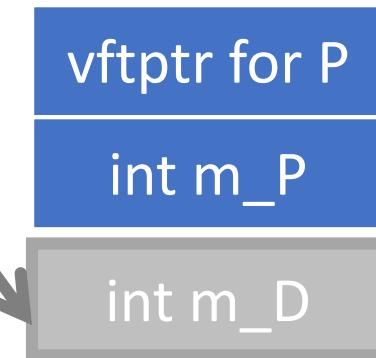


# Downcasting can be bad-casting

```
P *pS = new P();
D *pD = static_cast<D*>(pS);
pD->m_D;
```

Memory corruptions

$\&(pD \rightarrow m_D)$

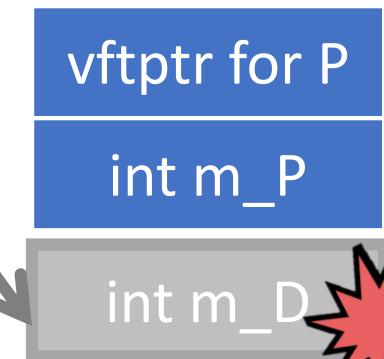


# Downcasting can be bad-casting

```
P *pS = new P();
D *pD = static_cast<D*>(pS);
pD->m_D;
```

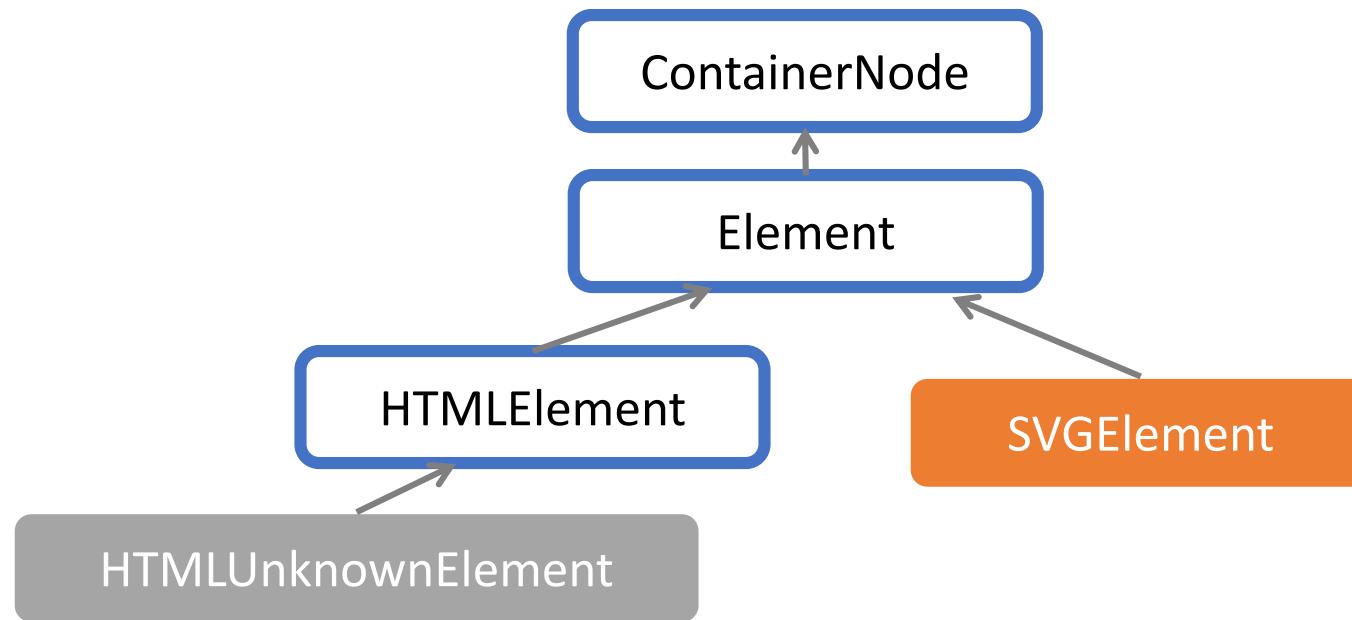
Memory corruptions

$\&(pD \rightarrow m_D)$



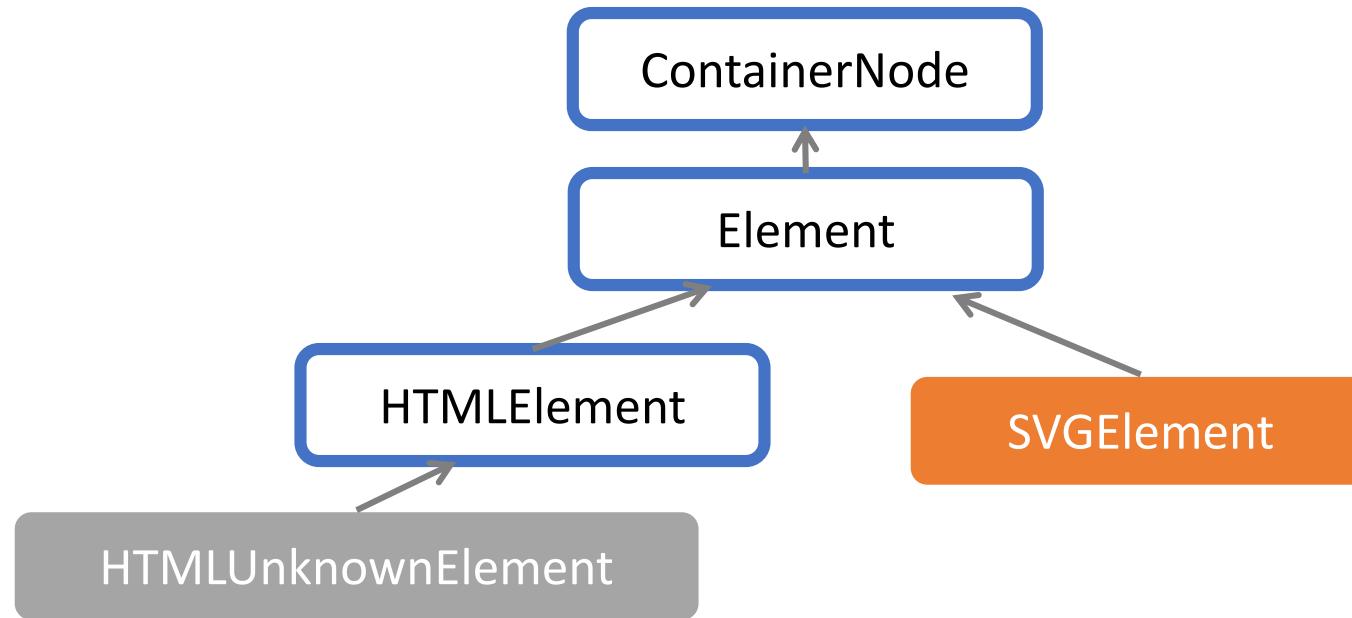
# Real-world exploits on bad-casting

- CVE-2013-0912
  - A bad-casting vulnerability in Chrome
  - Used in 2013 Pwn2Own



# Real-world exploits on bad-casting

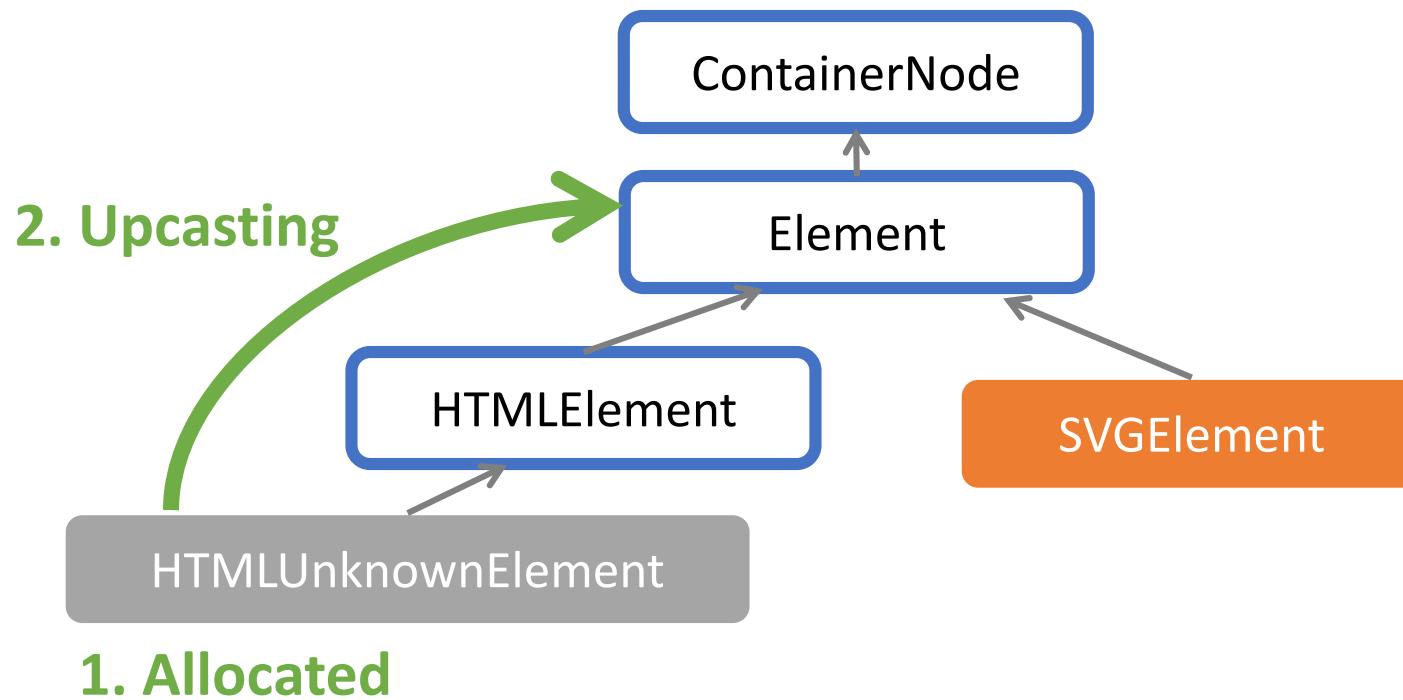
- CVE-2013-0912
  - A bad-casting vulnerability in Chrome
  - Used in 2013 Pwn2Own



**1. Allocated**

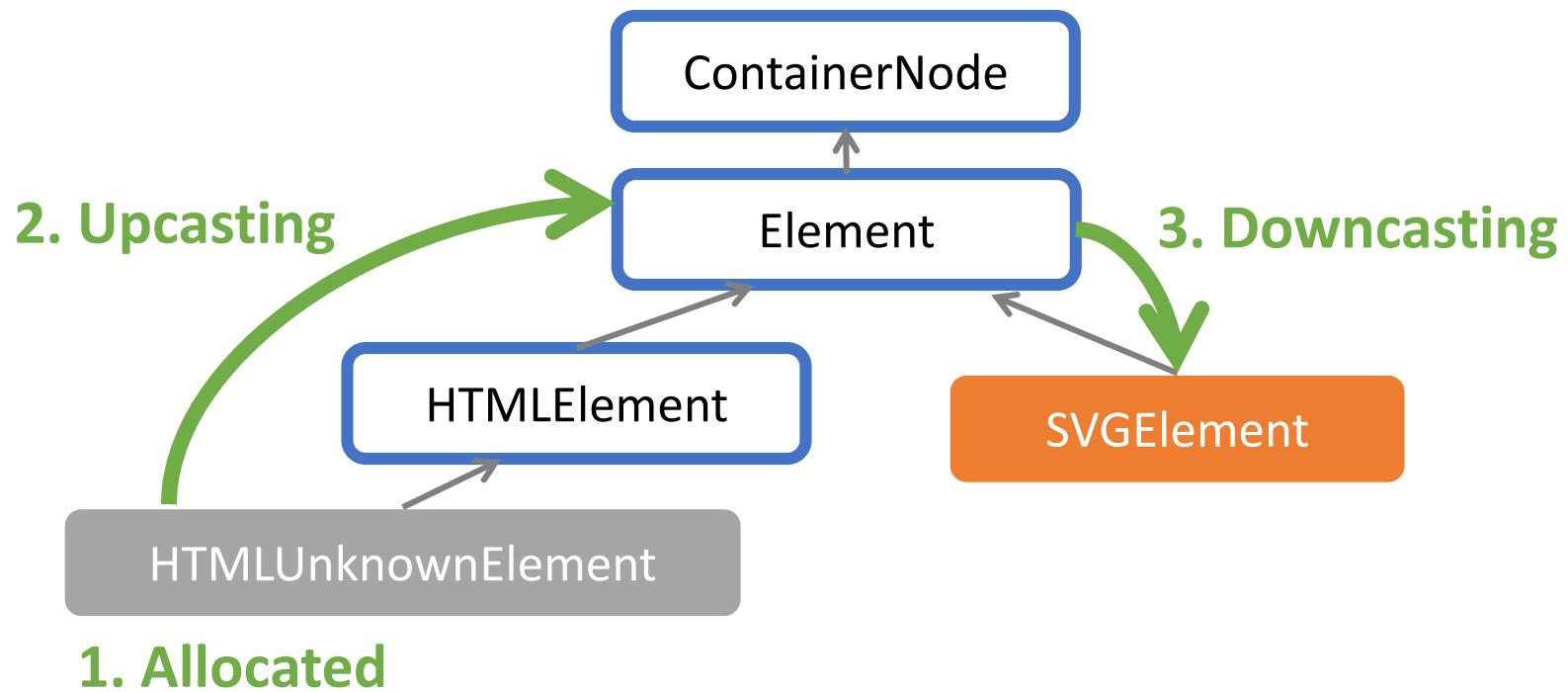
# Real-world exploits on bad-casting

- CVE-2013-0912
  - A bad-casting vulnerability in Chrome
  - Used in 2013 Pwn2Own



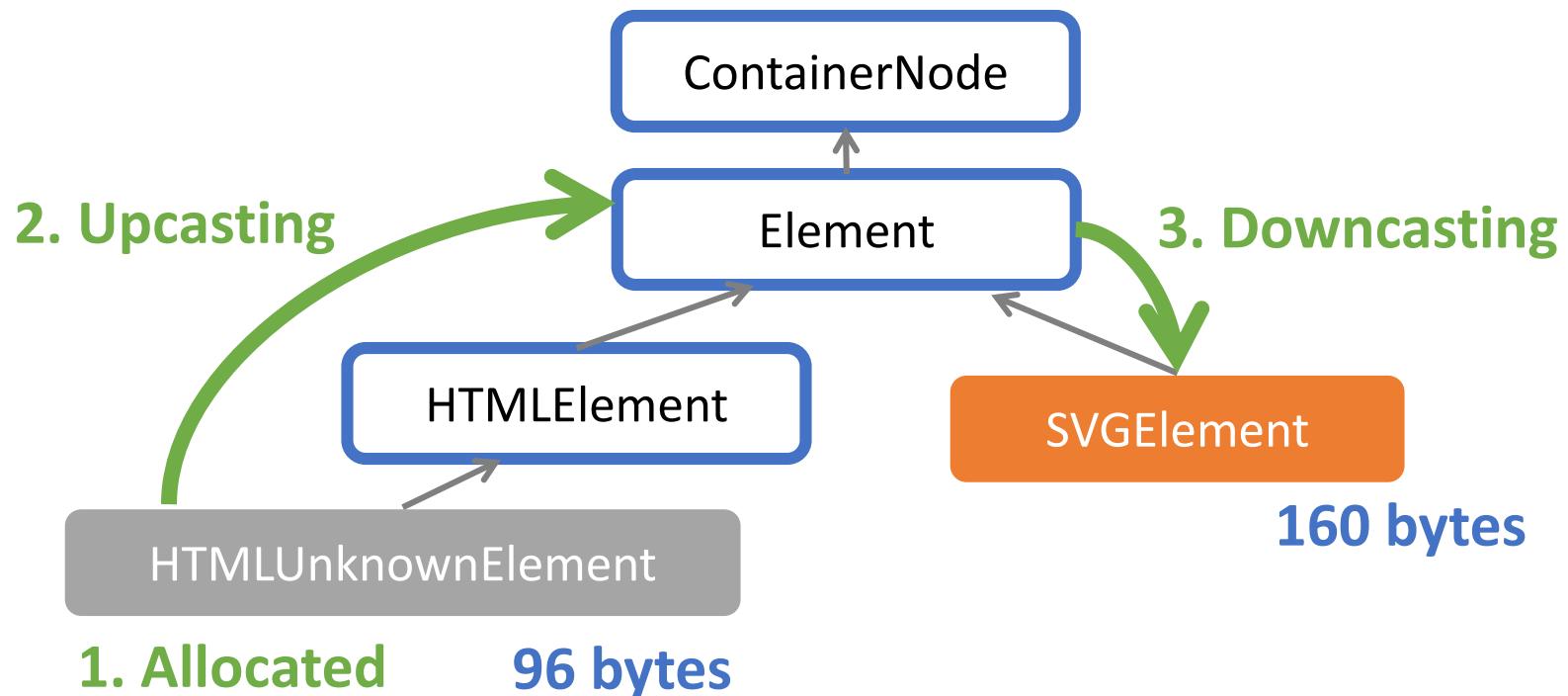
# Real-world exploits on bad-casting

- CVE-2013-0912
  - A bad-casting vulnerability in Chrome
  - Used in 2013 Pwn2Own



# Real-world exploits on bad-casting

- CVE-2013-0912
  - A bad-casting vulnerability in Chrome
  - Used in 2013 Pwn2Own



# CaVer

- CaVer: **CastVerifier**
  - A bad-casting elimination tool
- Design
  - Tracing runtime type information
  - Verify all casting operations

# Technical overview

```
P *ptr = new P;  
static_cast<D*>(ptr);
```

# Technical overview

```
P *ptr = new P;  
static_cast<D*>(ptr);
```

Q. Which class that **ptr** points to?  
→ **Runtime type tracing**

# Technical overview

```
P *ptr = new P;  
static_cast<D*>(ptr);
```

Q. Which class that **ptr** points to?  
→ **Runtime type tracing**

# Technical overview

```
P *ptr = new P;  
static_cast<D*>(ptr);
```

Q. Which class that **ptr** points to?

→ **Runtime type tracing**

Q. What are the class relationships b/w **P** and **D**?

→ **THTable**

# Type hierarchy table (THTable)

- A set of all legitimate classes to be converted
  - **Class names** are **hashed** for fast comparison
  - **Hierarchies** are **unrolled** to avoid recursive traversal

THTable (P)

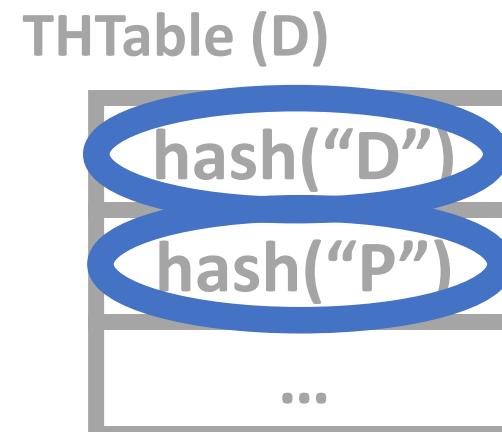
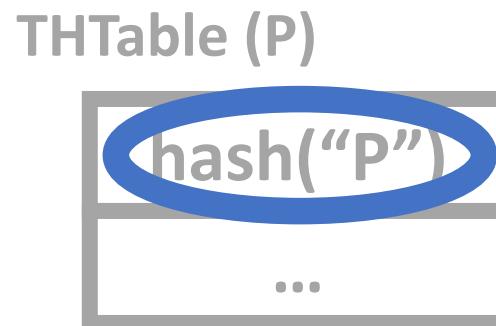


THTable (D)



# Type hierarchy table (THTable)

- A set of all legitimate classes to be converted
  - **Class names** are **hashed** for fast comparison
  - **Hierarchies** are **unrolled** to avoid recursive traversal



Hashed class names

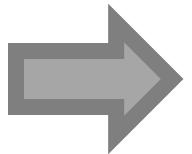
# Type hierarchy table (THTable)

- A set of all legitimate classes to be converted
  - **Class names** are **hashed** for fast comparison
  - **Hierarchies** are **unrolled** to avoid recursive traversal



# Runtime type tracing

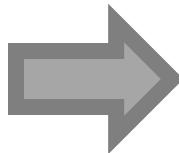
```
P *ptr = new P;
```



```
P *ptr = new P;  
trace(ptr, &THTable(P));
```

# Runtime type tracing

```
P *ptr = new P;
```



```
P *ptr = new P;  
trace(ptr, &THTable(P));
```

THTable (P)

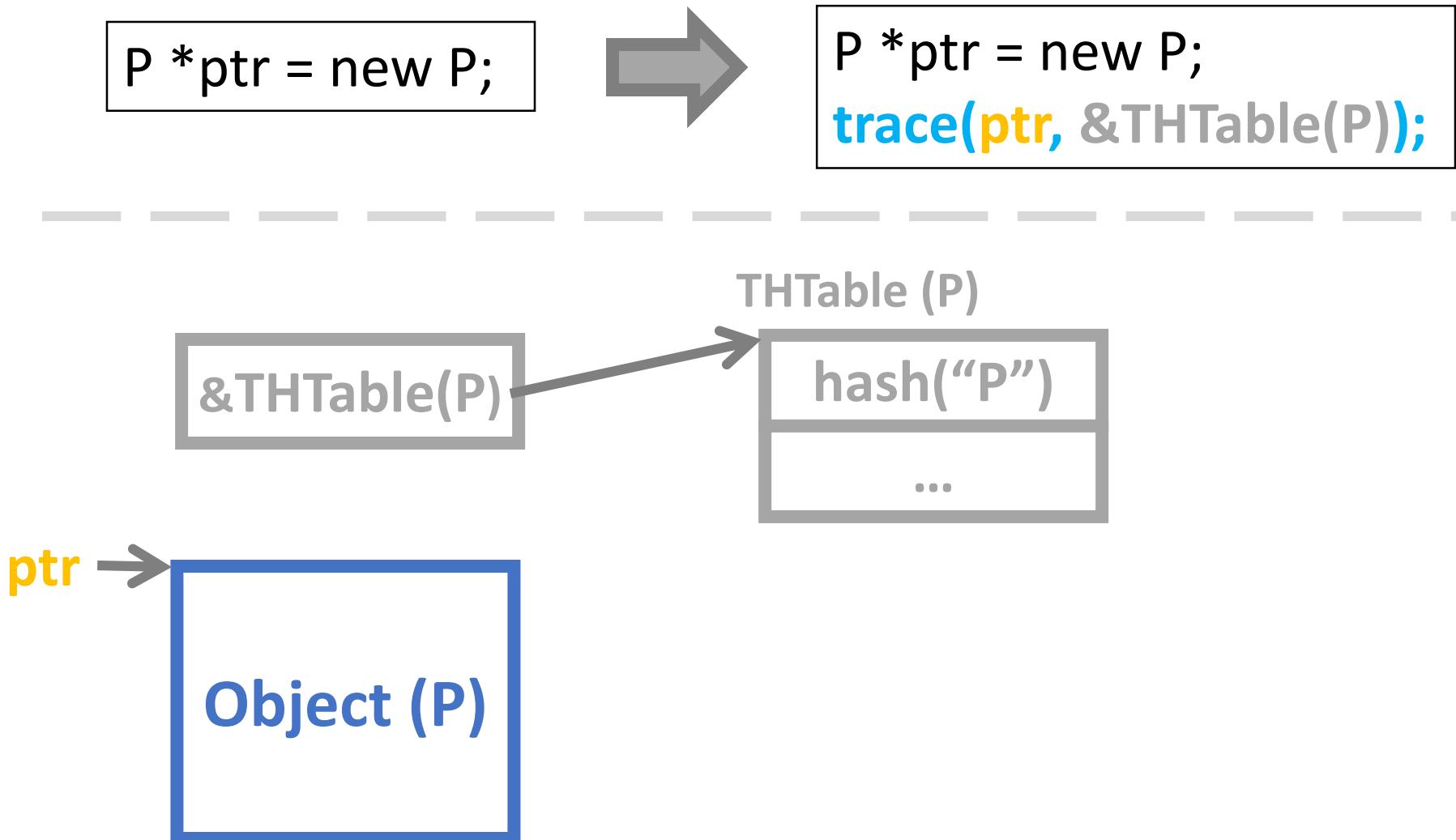
```
hash("P")
```

...

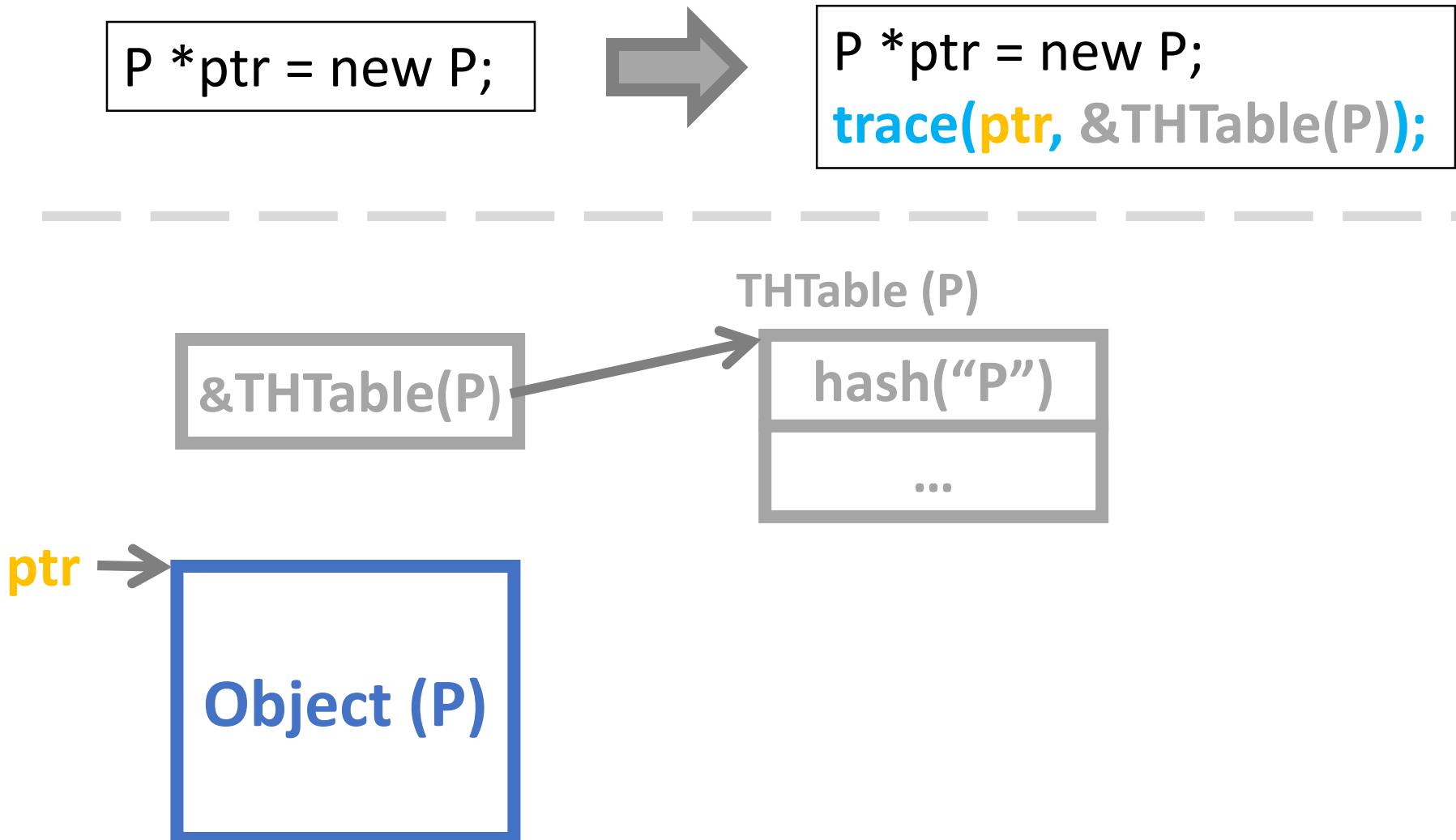
ptr →

Object (P)

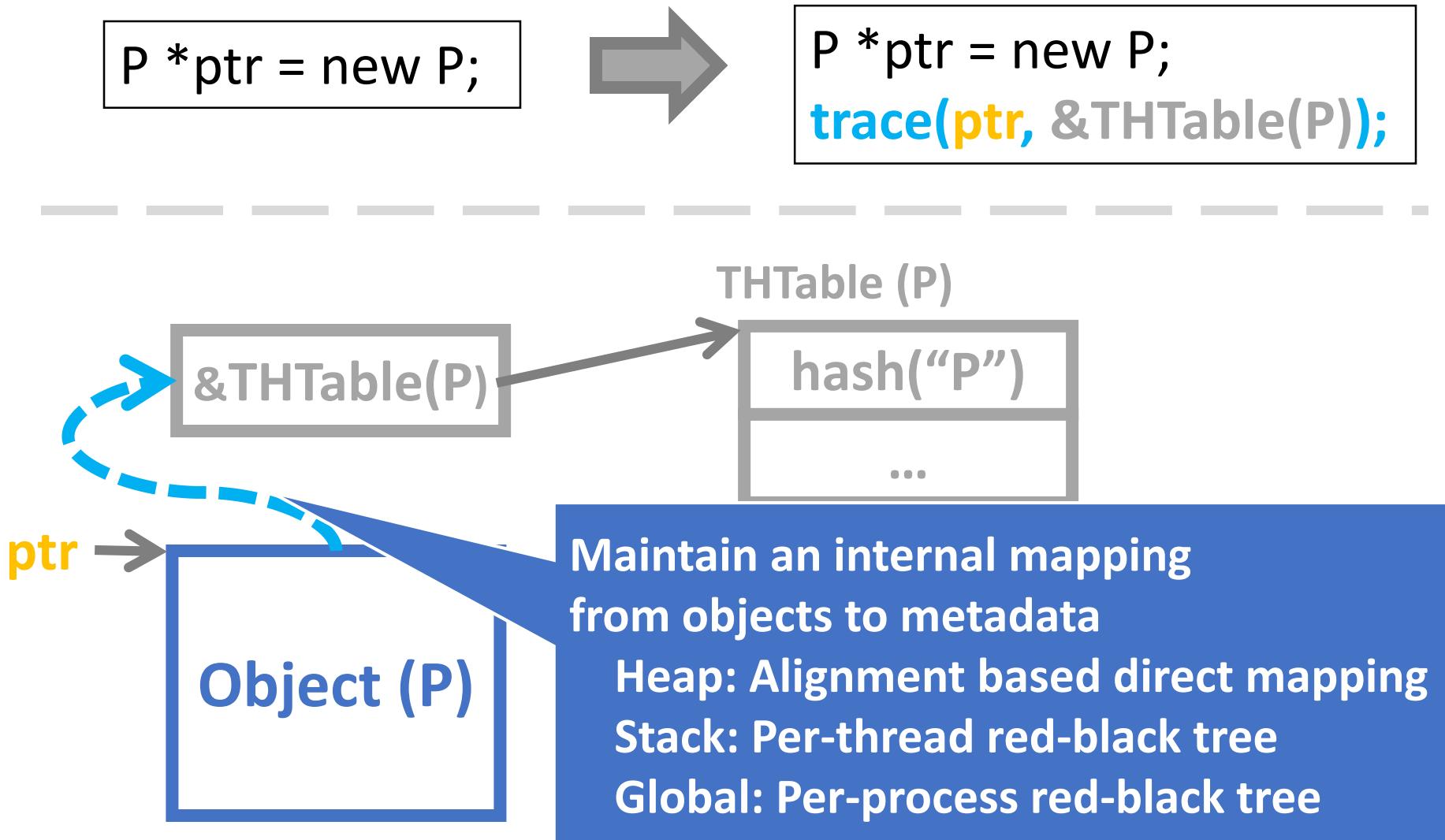
# Runtime type tracing



# Runtime type tracing



# Runtime type tracing



# Runtime type verification

```
static_cast<D*>(ptr);
```

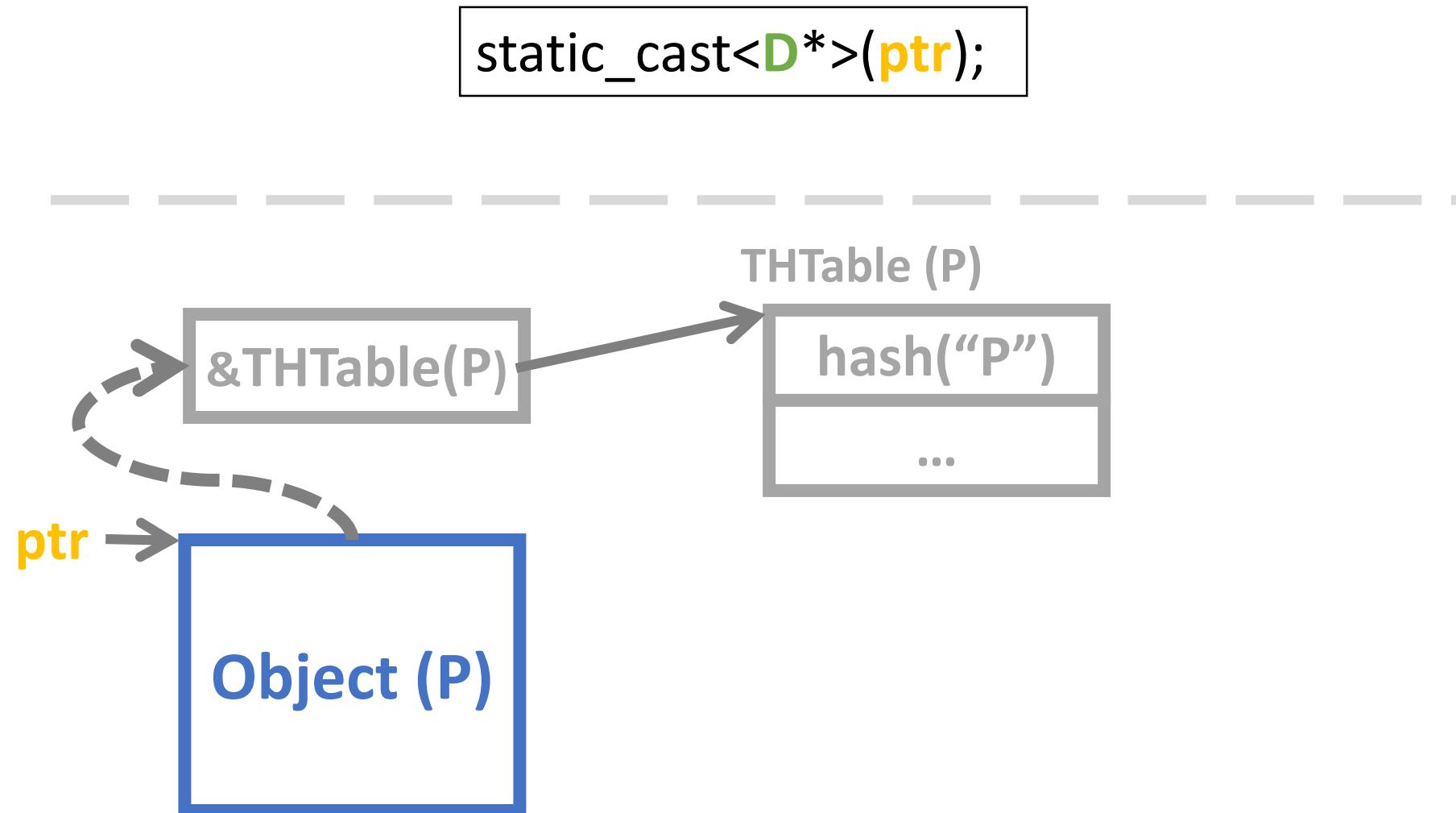


# Runtime type verification

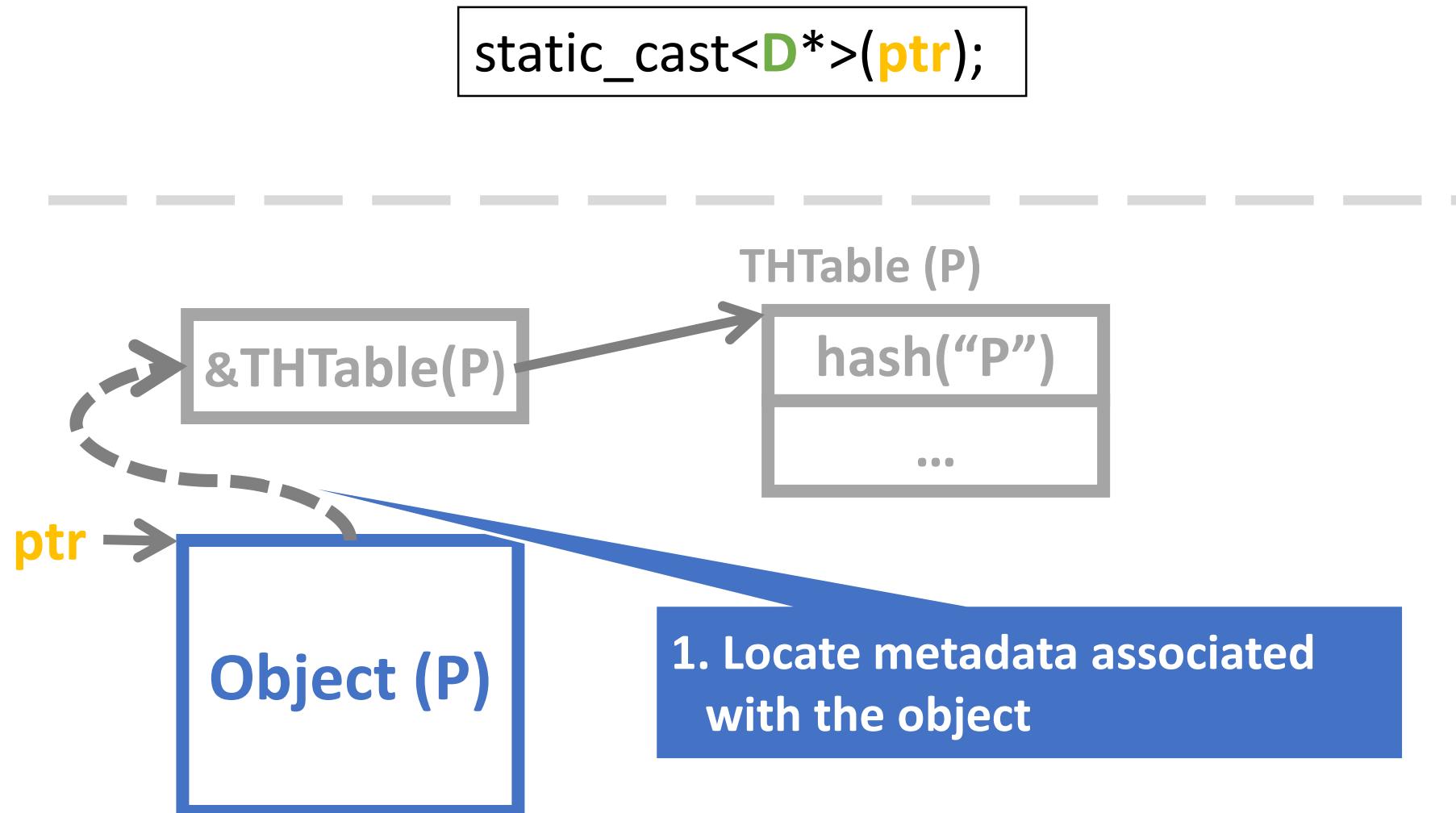
```
static_cast<D*>(ptr);
```



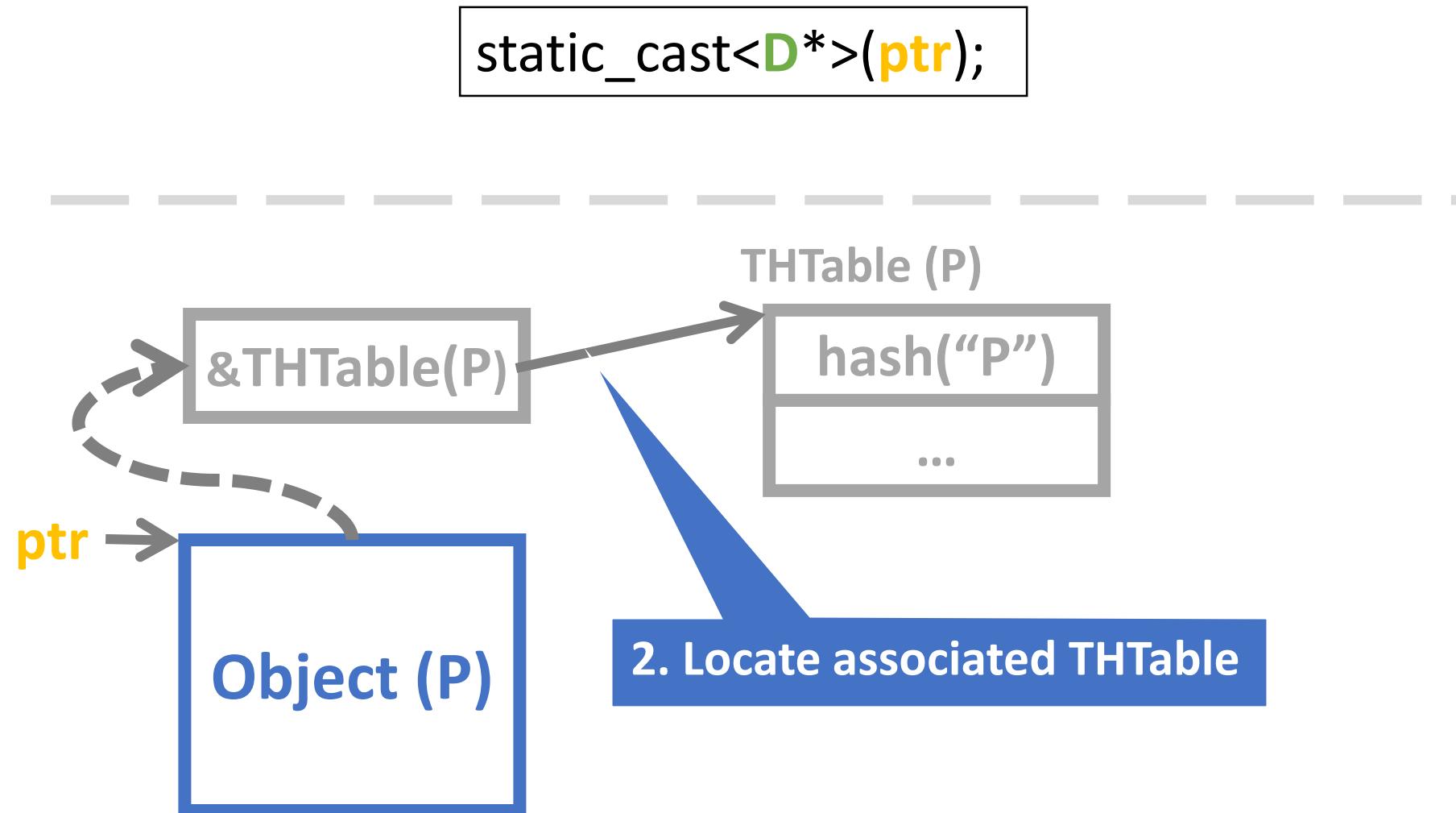
# Runtime type verification



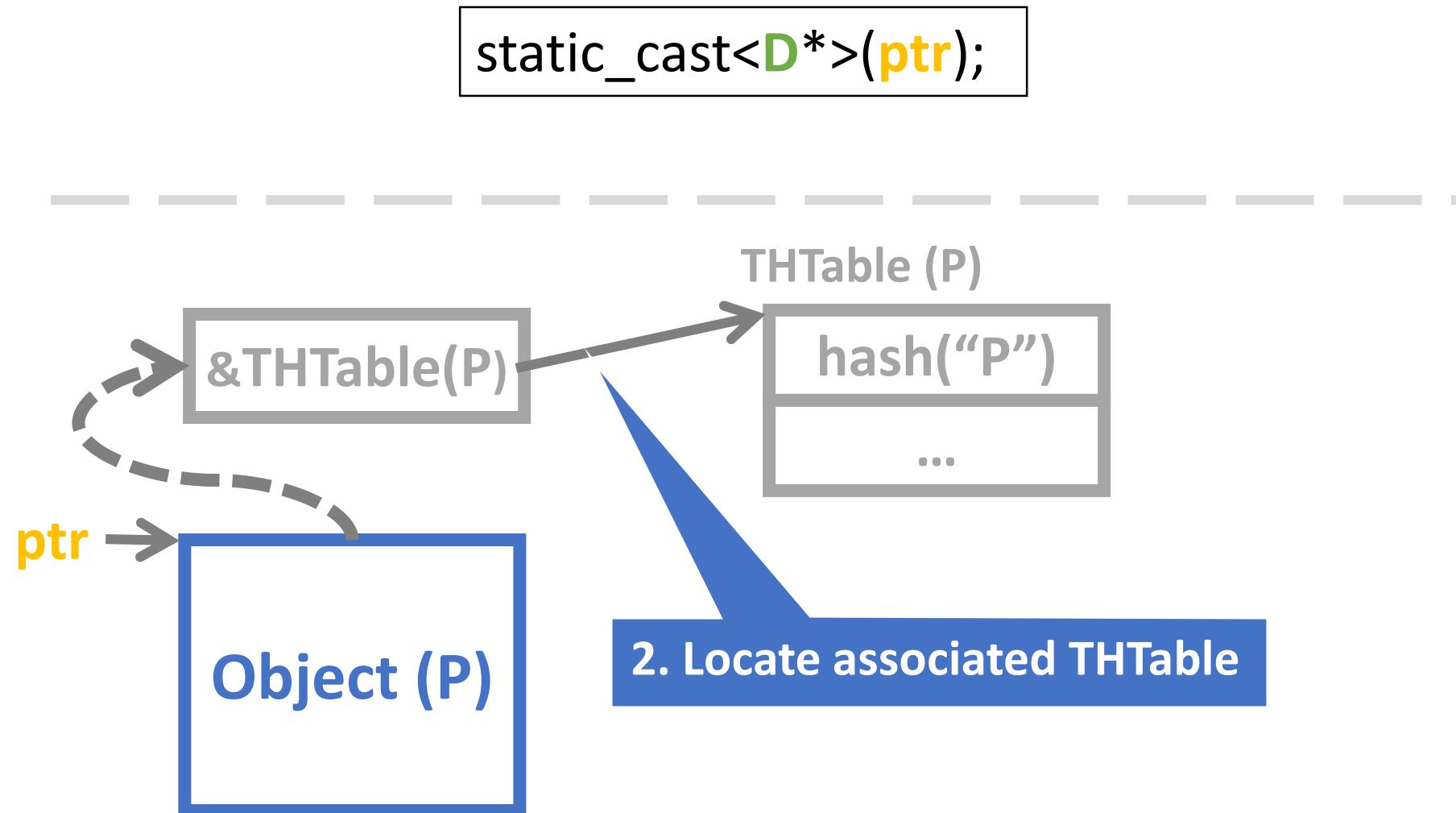
# Runtime type verification



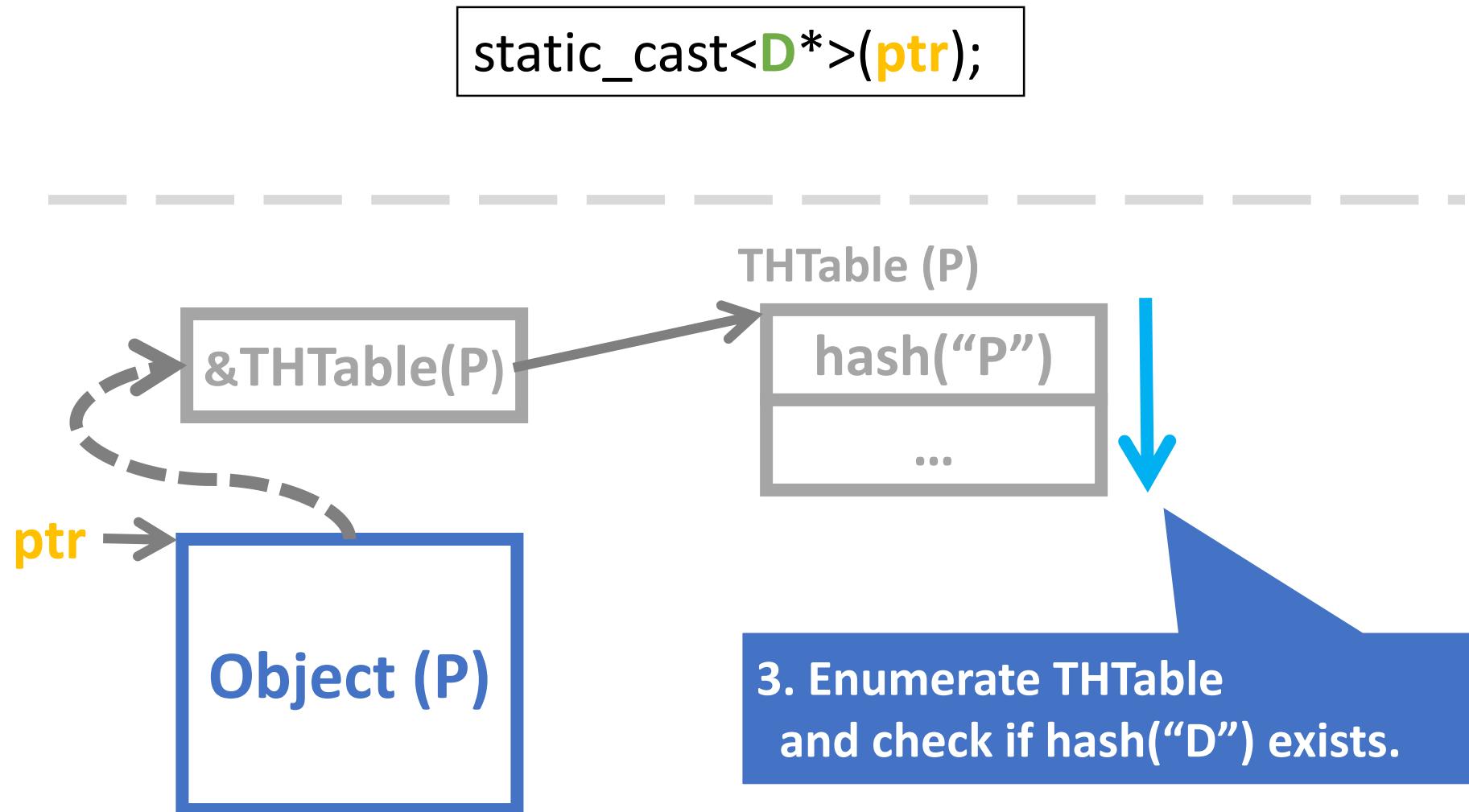
# Runtime type verification



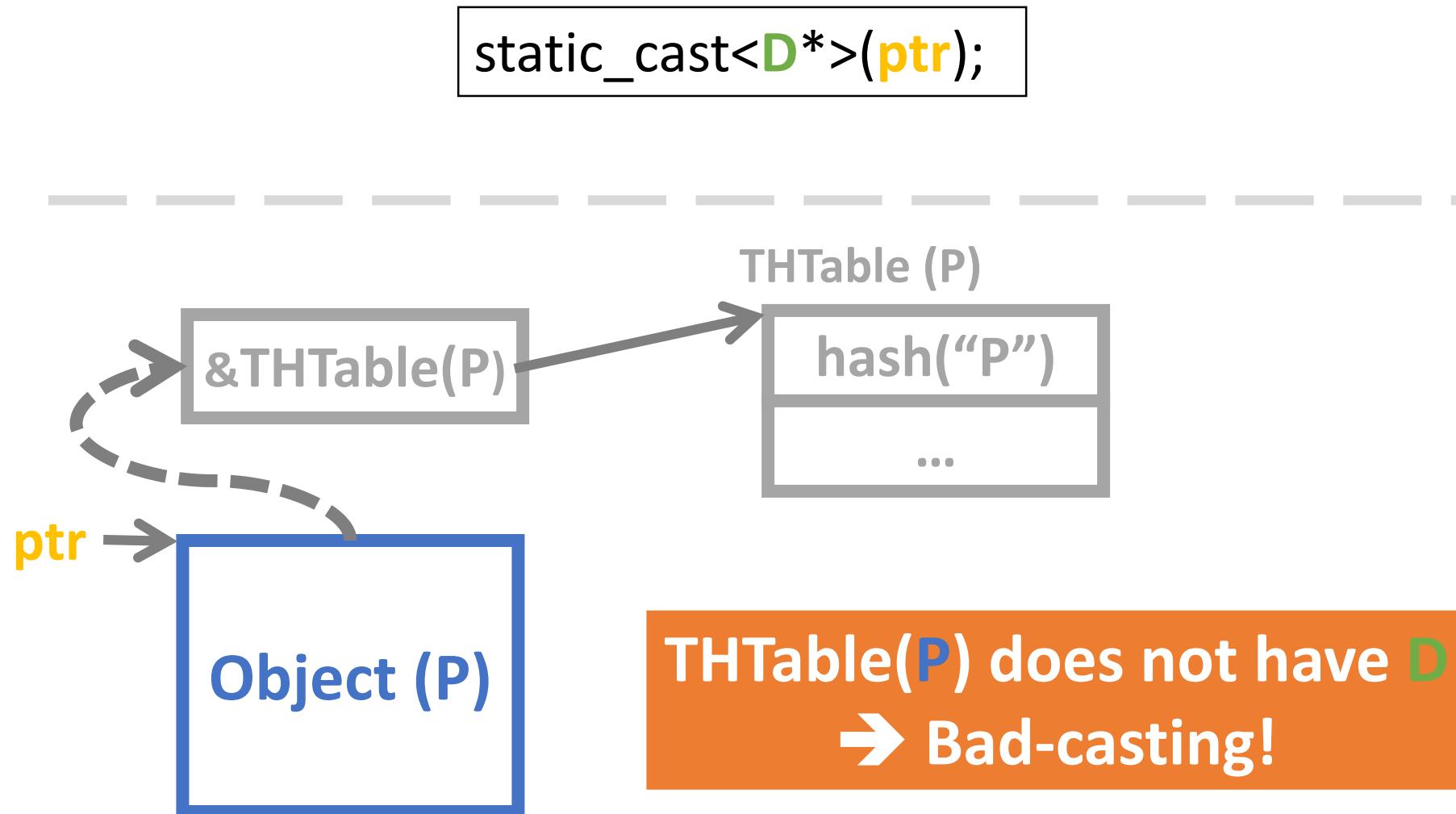
# Runtime type verification



# Runtime type verification



# Runtime type verification



# Implementation

- Prototype of CaVer
  - Added 3,540 lines of C++ code to LLVM compiler suites
- Target applications,
  - SPEC CPU 2006: one extra compiler and linker flag
  - Chromium: 21 line changes to build configurations
  - Firefox: 10 line changes to build configurations

# Evaluation on Chromium and Firefox

- Runtime overheads
  - 7.6% and 64.6% overheads in benchmarks
- Safely prevented five real-world bad-casting exploits
- Found 11 new vulnerabilities in Firefox and libstdc++

## 1. Eliminating vulnerabilities

**DangNull [NDSS 15]: Eliminating use-after-free vulnerabilities**

**CaVer [Security 15]: Eliminating bad-casting vulnerabilities**

## 2. Analyzing vulnerabilities

**SideFinder: Analyzing timing-channel vulnerabilities**

# Timing-channel vulnerabilities in hash tables

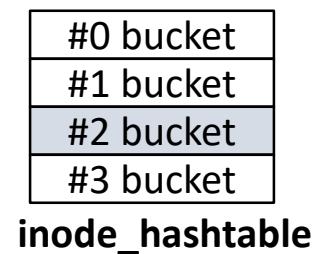
- Timing-channel
  - A time taken to perform a certain operation leaks some sensitive information.
- Hash tables
  - # of buckets are limited → collisions happen at some point
  - Collision resolution methods
    - Deterministic algorithm to decide the next bucket to be used

# Security sensitive data in hash tables

- Address information
  - ASLR protections can be bypassed.
  - Discovered examples
    - PrivateName in WebKit
    - Inode cache in the Linux kernel
- Filename information
  - Privacy can be breached.
  - Discovered examples
    - Dentry cache in the Linux kernel

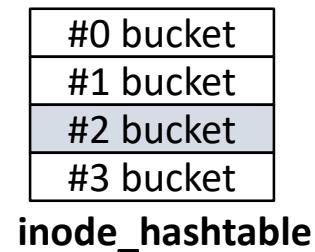
# Case study on inode cache

- **inode\_hashtable**
  - Mapping from an inode number to an inode object
  - Using a linked list to handle collisions
- **inode\_hash(sb, ino)**
  - A hash function computing a bucket index for inode\_hashtable
  - **sb**: an address of superblock (fixed, hidden security information)
  - **ino**: an inode number (controllable)



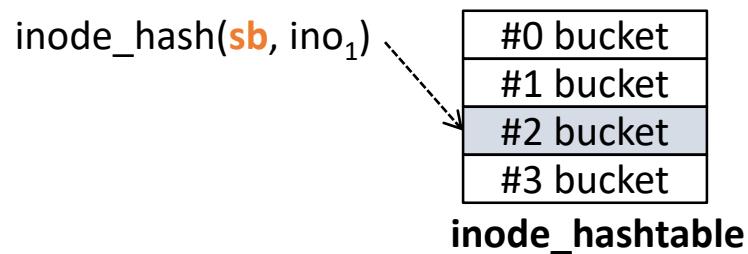
# Case study on inode cache

- **inode\_hashtable**
  - Mapping from an inode number to an inode object
  - Using a linked list to handle collisions
- **inode\_hash(sb, ino)**
  - A hash function computing a bucket index for inode\_hashtable
  - **sb**: an address of superblock (fixed, hidden security information)
  - ino: an inode number (controllable)



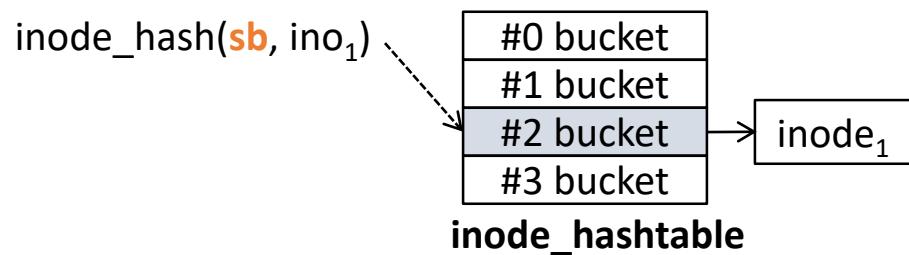
# Case study on inode cache

- **inode\_hashtable**
  - Mapping from an inode number to an inode object
  - Using a linked list to handle collisions
- **inode\_hash(*sb*, *ino*)**
  - A hash function computing a bucket index for `inode_hashtable`
  - **sb**: an address of superblock (fixed, hidden security information)
  - *ino*: an inode number (controllable)



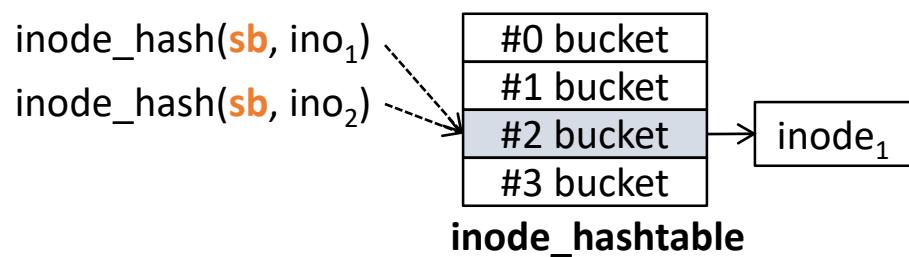
# Case study on inode cache

- **inode\_hashtable**
  - Mapping from an inode number to an inode object
  - Using a linked list to handle collisions
- **inode\_hash(*sb*, *ino*)**
  - A hash function computing a bucket index for `inode_hashtable`
  - **sb**: an address of superblock (fixed, hidden security information)
  - *ino*: an inode number (controllable)



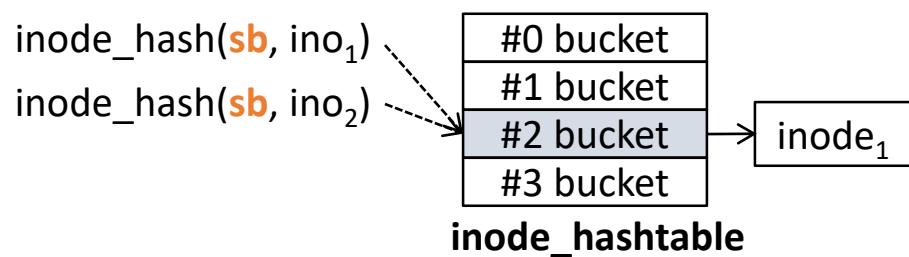
# Case study on inode cache

- **inode\_hashtable**
  - Mapping from an inode number to an inode object
  - Using a linked list to handle collisions
- **inode\_hash(*sb*, *ino*)**
  - A hash function computing a bucket index for `inode_hashtable`
  - *sb*: an address of superblock (fixed, hidden security information)
  - *ino*: an inode number (controllable)



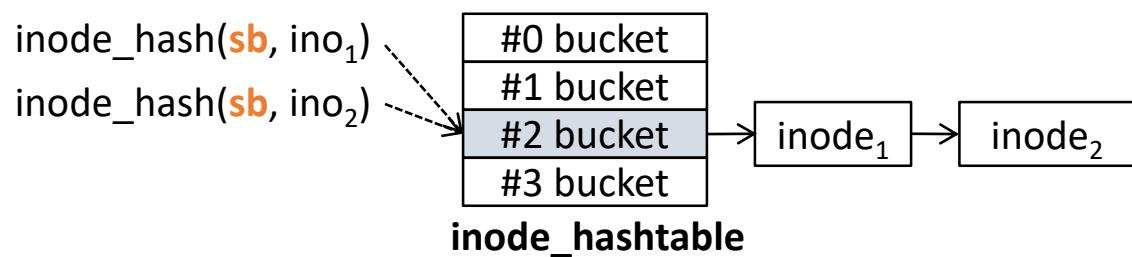
# Case study on inode cache

- **inode\_hashtable**
  - Mapping from an inode number to an inode object
  - Using a linked list to handle collisions
- **inode\_hash( $sb$ ,  $ino$ )**
  - A hash function computing a bucket index for `inode_hashtable`
  - $sb$ : an address of superblock (fixed, hidden security information)
  - $ino$ : an inode number (controllable)



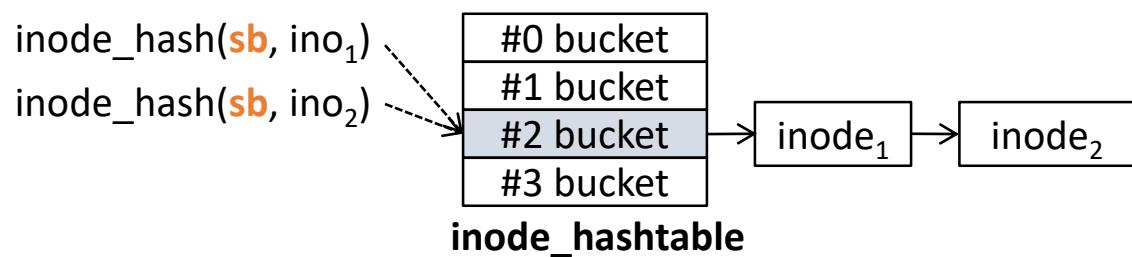
# Case study on inode cache

- **inode\_hashtable**
  - Mapping from an inode number to an inode object
  - Using a linked list to handle collisions
- **inode\_hash(*sb*, *ino*)**
  - A hash function computing a bucket index for `inode_hashtable`
  - **sb**: an address of superblock (fixed, hidden security information)
  - *ino*: an inode number (controllable)



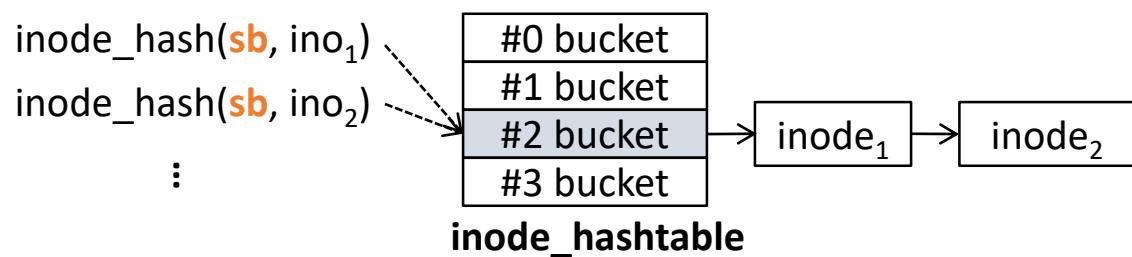
# Case study on inode cache

- **inode\_hashtable**
  - Mapping from an inode number to an inode object
  - Using a linked list to handle collisions
- **inode\_hash(*sb*, *ino*)**
  - A hash function computing a bucket index for `inode_hashtable`
  - **sb**: an address of superblock (fixed, hidden security information)
  - *ino*: an inode number (controllable)



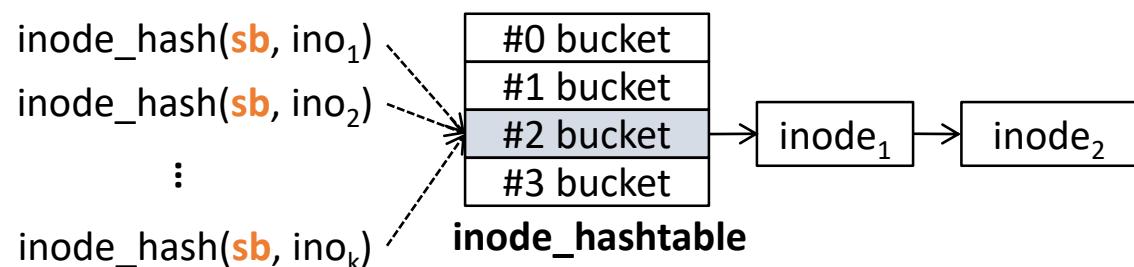
# Case study on inode cache

- **inode\_hashtable**
  - Mapping from an inode number to an inode object
  - Using a linked list to handle collisions
- **inode\_hash(*sb*, *ino*)**
  - A hash function computing a bucket index for `inode_hashtable`
  - **sb**: an address of superblock (fixed, hidden security information)
  - *ino*: an inode number (controllable)



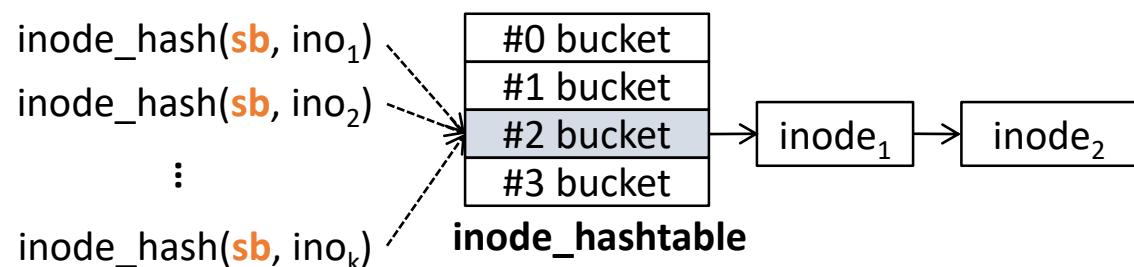
# Case study on inode cache

- **inode\_hashtable**
  - Mapping from an inode number to an inode object
  - Using a linked list to handle collisions
- **inode\_hash( $sb$ ,  $ino$ )**
  - A hash function computing a bucket index for `inode_hashtable`
  - $sb$ : an address of superblock (fixed, hidden security information)
  - $ino$ : an inode number (controllable)



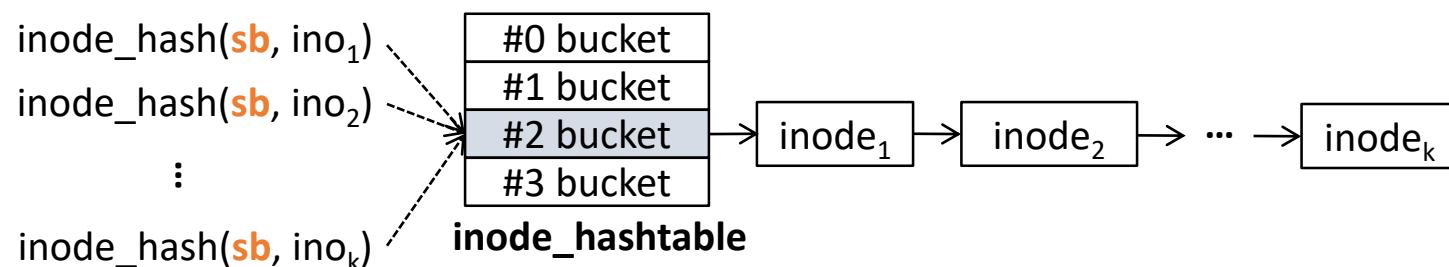
# Case study on inode cache

- **inode\_hashtable**
  - Mapping from an inode number to an inode object
  - Using a linked list to handle collisions
- **inode\_hash( $sb$ ,  $ino$ )**
  - A hash function computing a bucket index for `inode_hashtable`
  - $sb$ : an address of superblock (fixed, hidden security information)
  - $ino$ : an inode number (controllable)



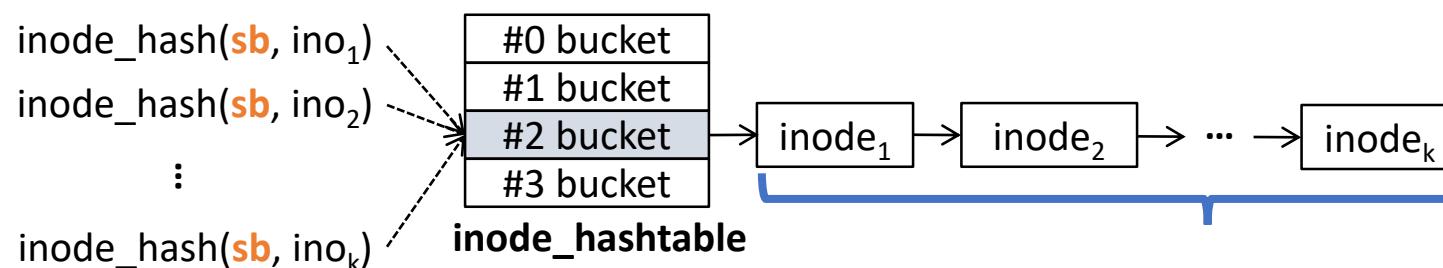
# Case study on inode cache

- **inode\_hashtable**
  - Mapping from an inode number to an inode object
  - Using a linked list to handle collisions
- **inode\_hash( $sb$ ,  $ino$ )**
  - A hash function computing a bucket index for `inode_hashtable`
  - $sb$ : an address of superblock (fixed, hidden security information)
  - $ino$ : an inode number (controllable)



# Case study on inode cache

- **inode\_hashtable**
  - Mapping from an inode number to an inode object
  - Using a linked list to handle collisions
- **inode\_hash( $sb$ ,  $ino$ )**
  - A hash function computing a bucket index for `inode_hashtable`
  - $sb$ : an address of superblock (fixed, hidden security information)
  - $ino$ : an inode number (controllable)



Execution time differences  
→ Inferring the value of  $sb$

# Motivations

- Confirming timing-channel vulnerability is difficult
  - No explicit data flows on leaked data
  - Involve multiple paths/runs to trigger an attack
  - Need to find a number of colliding inputs
- Q. How to confirm the existence of security sensitive timing-channels?  
→ SideFinder

# SideFinder

- Goal
  - Semi-automatically synthesize attacking inputs of side-channels in hash tables
- Design
  - Input
    - A signature of a target hash function
  - Workflow
    - 1. Backward slicing
      - Identifying possible execution paths
    - 2. Concolic execution
      - Driving symbolic execution while avoiding path explosions
    - 3. Synthesize
      - At the bucket index computation, querying the solver to obtain multiple colliding inputs

# An example of workflow: inode cache

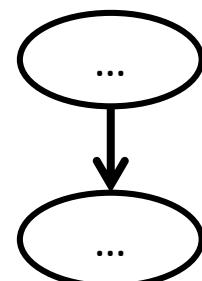
## 1. Backward slicing



inode\_hash(**sb**, **ino**)

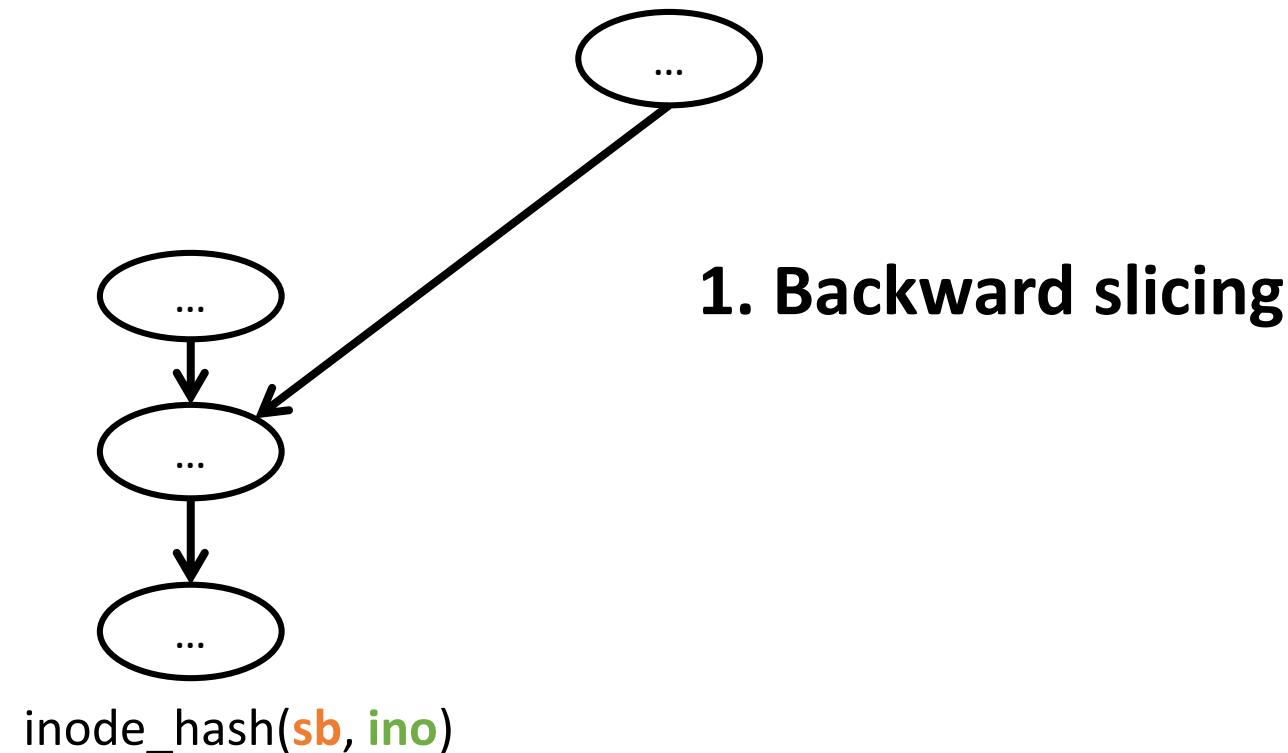
# An example of workflow: inode cache

## 1. Backward slicing

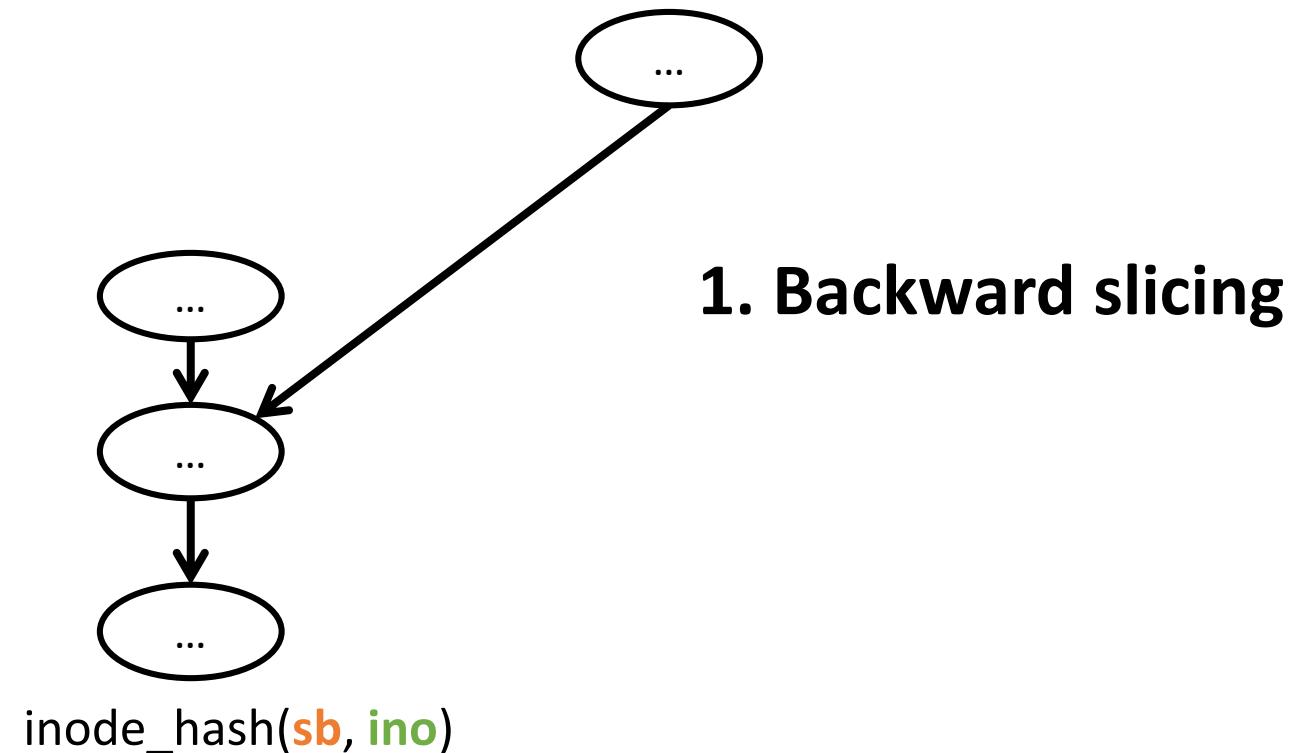


inode\_hash(**sb**, **ino**)

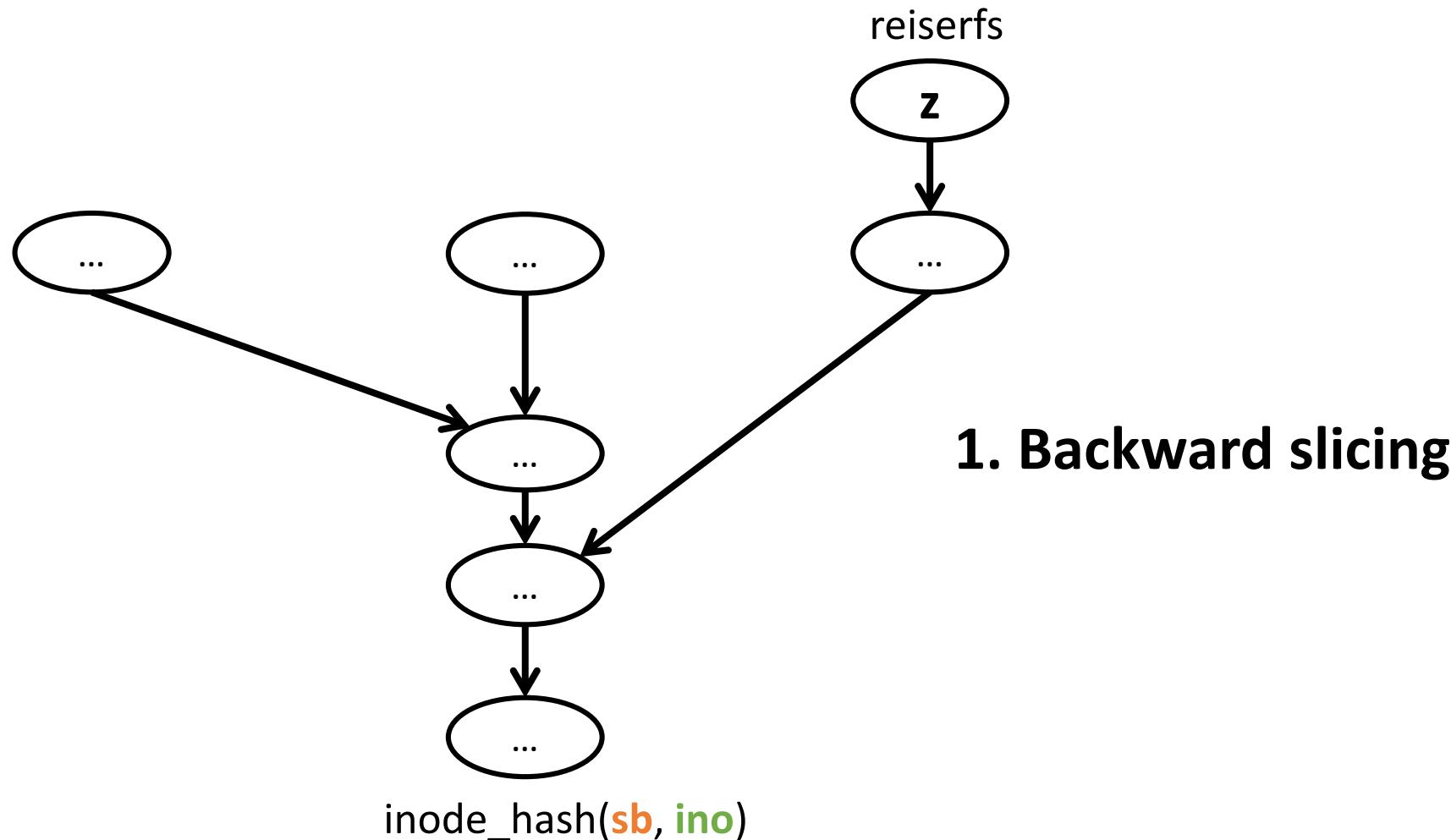
# An example of workflow: inode cache



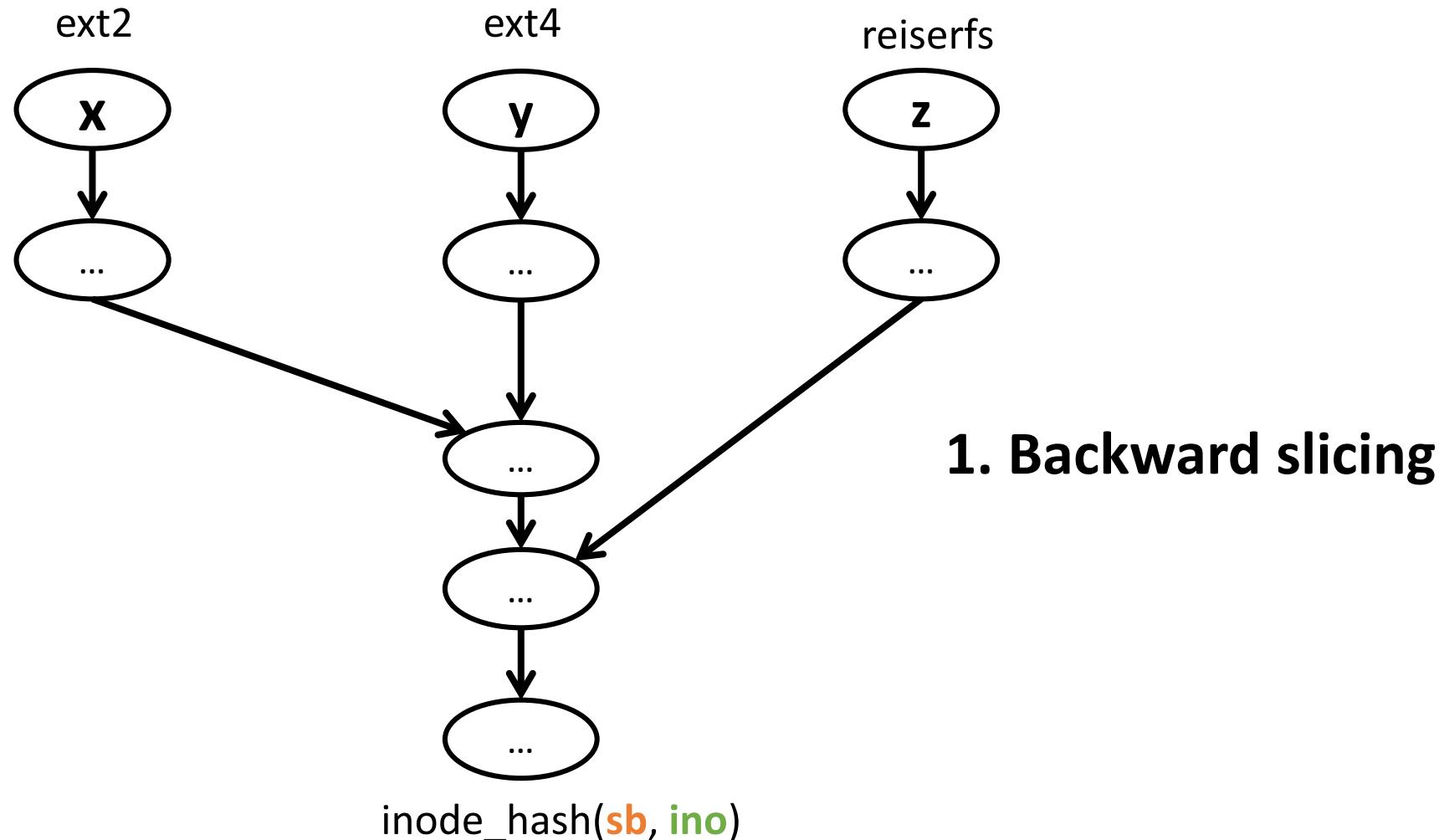
# An example of workflow: inode cache



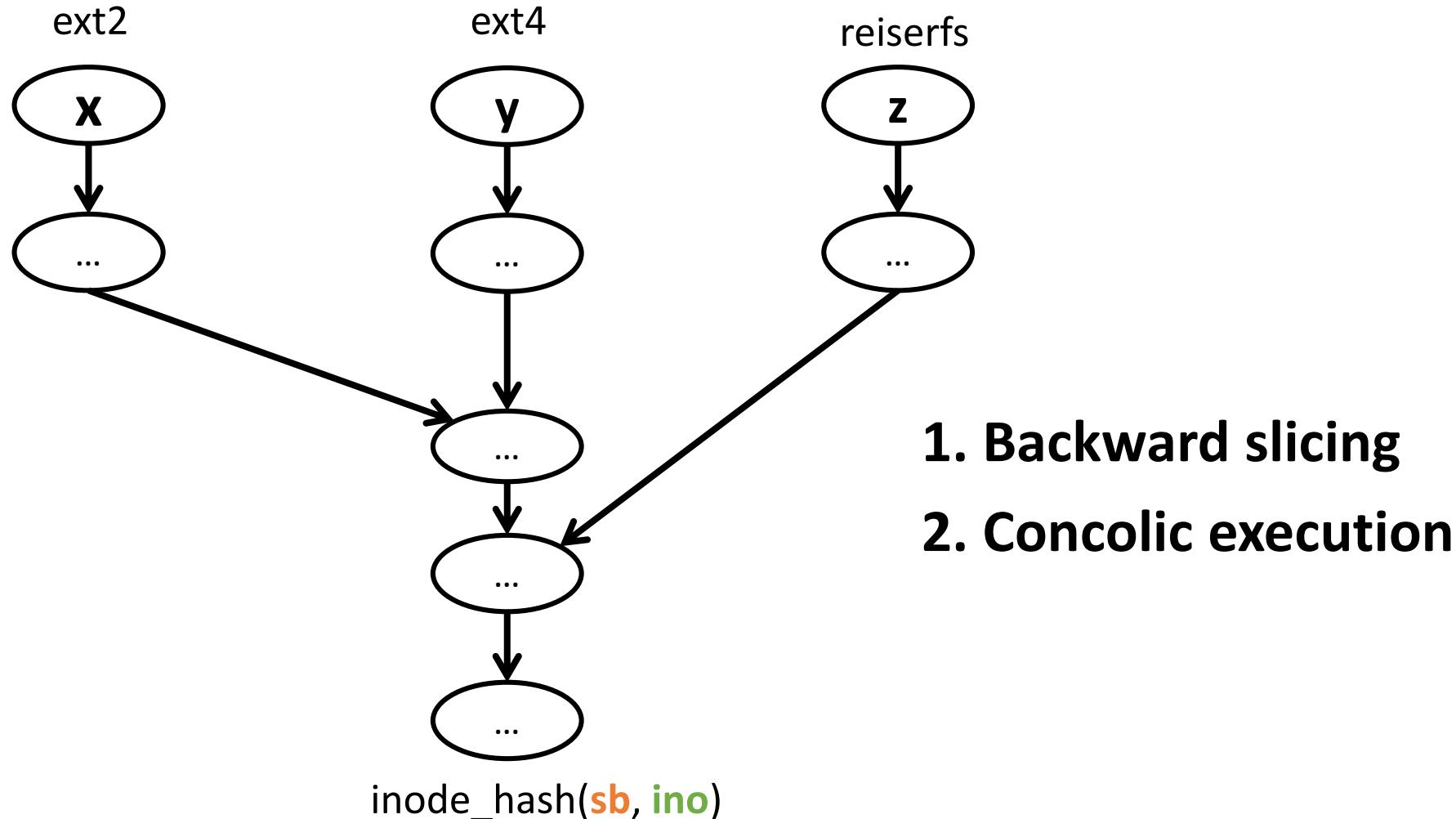
# An example of workflow: inode cache



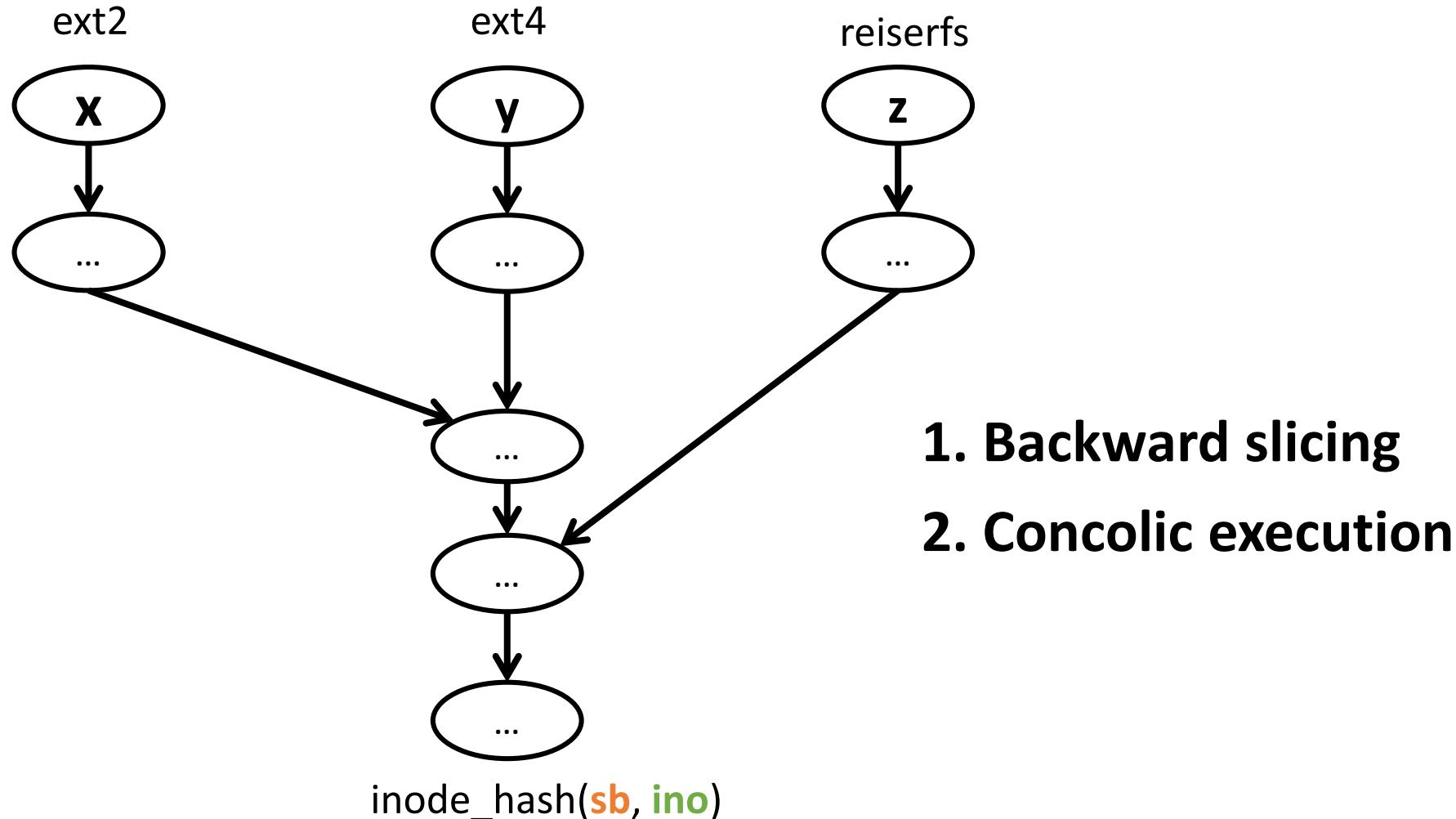
# An example of workflow: inode cache



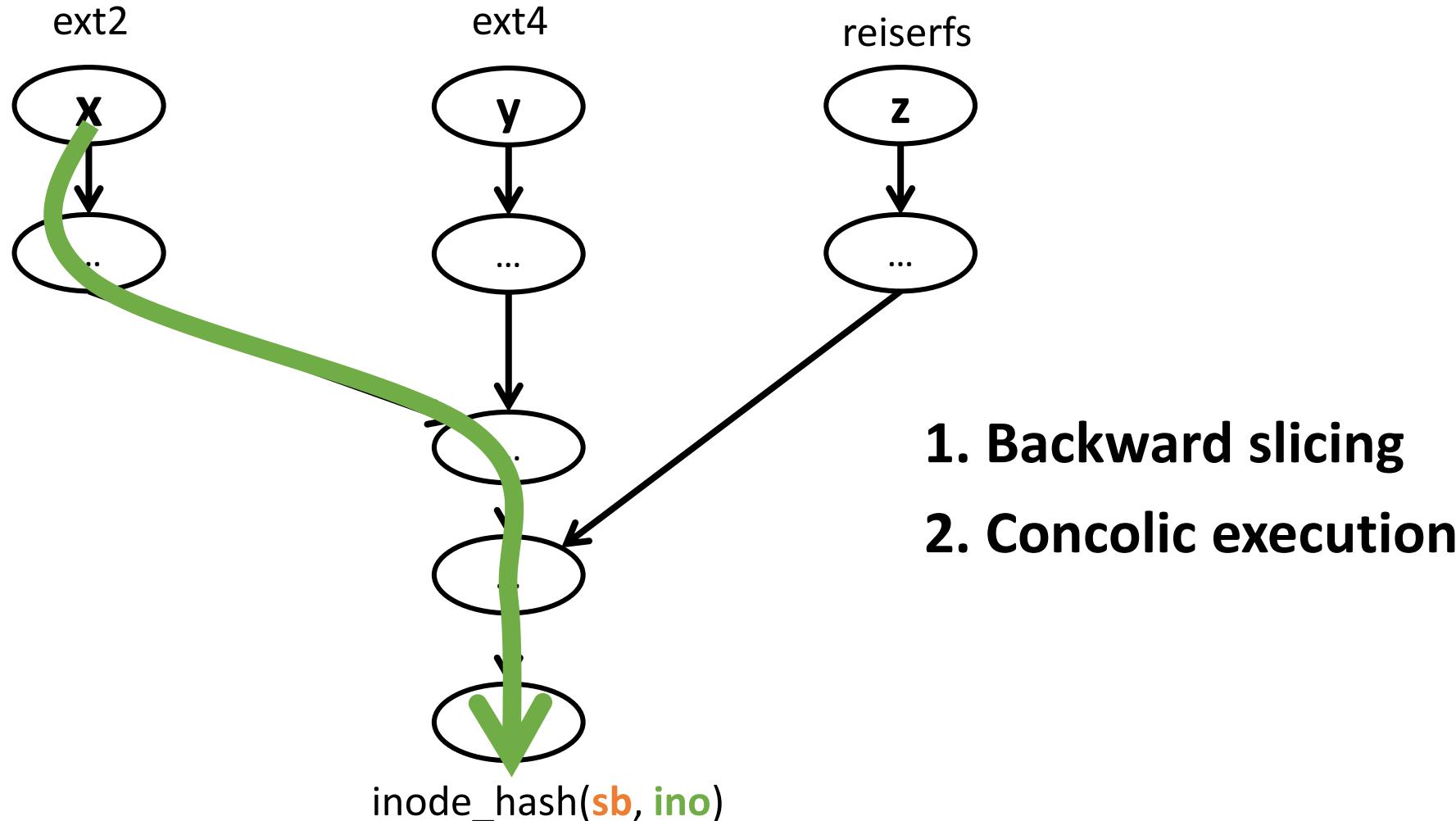
# An example of workflow: inode cache



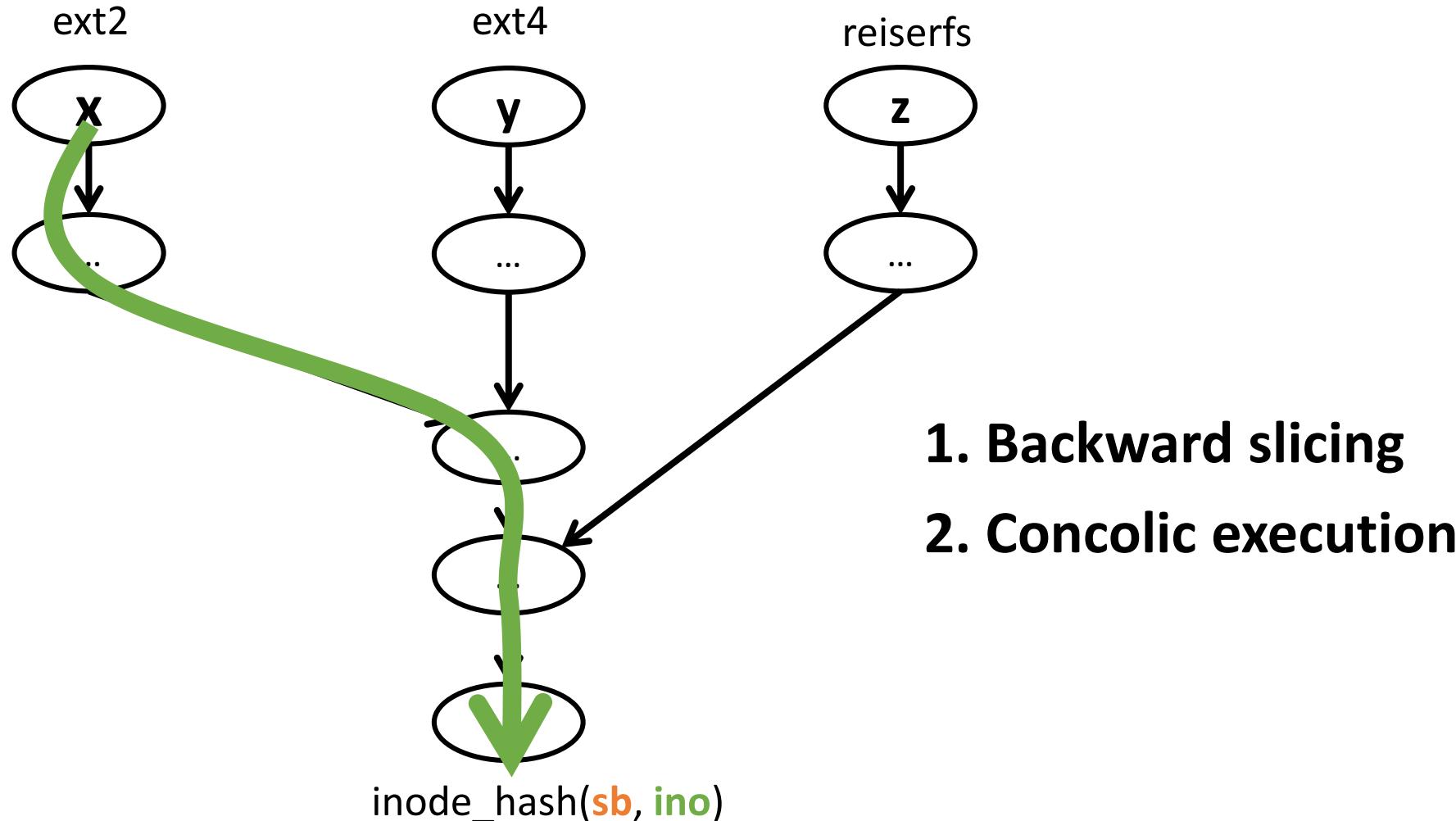
# An example of workflow: inode cache



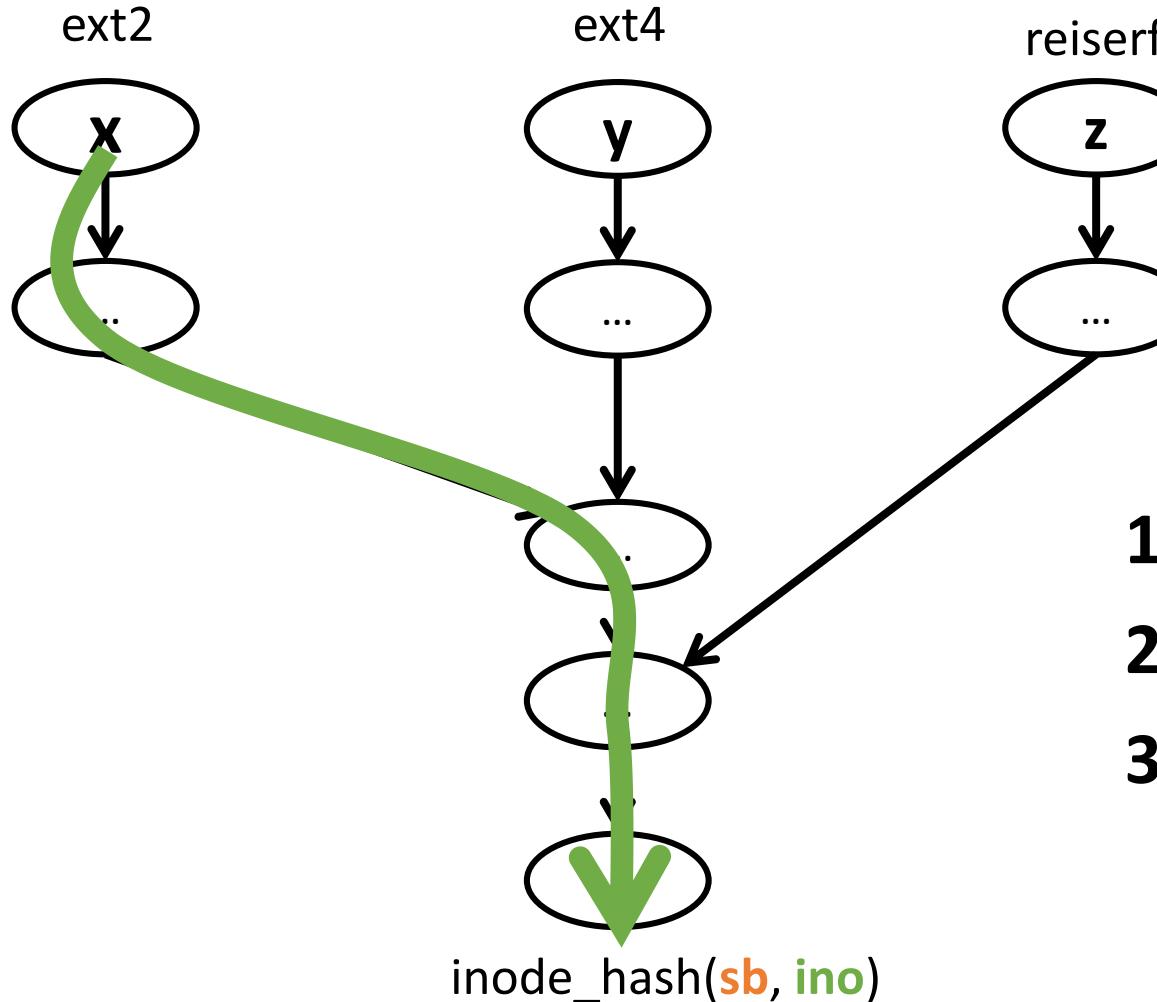
# An example of workflow: inode cache



# An example of workflow: inode cache



# An example of workflow: inode cache



1. Backward slicing
2. Concolic execution
3. Synthesize
  - find a set of X, which satisfies  $\text{inode\_hash}(\text{sb}, \text{ino}) == \text{bucket\_index}$

# Implementation

- Backward slicing
  - Based on LLVM
  - Used a dependency analysis
  - Flow-insensitive, context-sensitive, and field-sensitive
- Concolic execution
  - Based on S2E
  - Added a helper function for memory symbolization
  - Added a special op code to obtain multiple concrete values

# Evaluation on inode cache attacks

- Backward slicing

FS	#S	True src. description	I
ext2	1	inode@ext2_inode_by_name()	✓
ext4	11	inode@ext4_lookup()	✓
reiserfs	5	k_objectid@reiserfs_iget()	✓
jfs	3	inum@jfs_lookup()	✓
btrfs	4	objectid@btrfs_iget_locked()	✓
xfs	0	N/A	N/A

# Evaluation on inode cache attacks

- Backward slicing

FS	#S	True src. description	I
ext2	1	inode@ext2_inode_by_name()	✓
ext4	11	inode@ext4_lookup()	✓
reiserfs	5	k_objectid@reiserfs_iget()	✓
jfs	3	inum@jfs_lookup()	✓
btrfs	4	objectid@btrfs_iget_locked()	✓
xfs	0	N/A	N/A

# of taint sources SideFinder identified

# Evaluation on inode cache attacks

- Backward slicing

A true interface reaching to `inode_hash`  
(i.e., to control an inode number)

FS	#S	True src. description	I
ext2	1	<code>inode@ext2_inode_by_name()</code>	✓
ext4	11	<code>inode@ext4_lookup()</code>	✓
reiserfs	5	<code>k_objectid@reiserfs_iget()</code>	✓
jfs	3	<code>inum@jfs_lookup()</code>	✓
btrfs	4	<code>objectid@btrfs_iget_locked()</code>	✓
xfs	0	N/A	N/A

# Evaluation on inode cache attacks

- Backward slicing

FS	#S	True src. description	I
ext2	1	inode@ext2_inode_by_name()	✓
ext4	11	inode@ext4_lookup()	✓
reiserfs	5	k_objectid@reiserfs_iget()	✓
jfs	3	inum@jfs_lookup()	✓
btrfs	4	objectid@btrfs_iget_locked()	✓
xfs	0	N/A	N/A

SideFinder successfully identified all true sources

# Evaluation on inode cache attacks

- Backward slicing

FS	#S	True src. description	I
ext2	1	inode@ext2_inode_by_name()	✓
ext4	11	inode@ext4_lookup()	✓
reiserfs	5	k_objectid@reiserfs_iget()	✓
jfs	3	inum@jfs_lookup()	✓
btrfs	4	objectid@btrfs_iget_locked()	✓
xfs	0	N/A	N/A

XFS implements its own hash table

# Evaluation on inode cache attacks

- Concolic execution and synthesizing

Complexity: the number of clauses in a bucket expression

FS	Bucket	Time (s)
	# OP	
ext2	48	319
ext4	50	336
reiserfs	48	321
jfs	48	318
btrfs	50	324
xfs	-	-

Efficiency: time taken to synthesize 2,048 colliding inputs

# Thesis contributions

- **Analysis and formalization of emerging vulnerability classes**
  - Use-after-free, bad-casting, and timing-side channels
- **Designs and implementations of practical security tools**
  - Automated elimination: DangNull and CaVer
  - Analysis assistant: SideFinder
- **New security vulnerabilities**
  - 14 previously unknown vulnerabilities in Firefox, stdlibc++, and the Linux kernel

# Conclusion

- Protect a system by eliminating or analyzing vulnerabilities
  - Eliminating vulnerabilities
    - DangNull: use-after-free vulnerabilities
    - CaVer: bad-casting vulnerabilities
  - Analyzing vulnerabilities
    - SideFinder: timing-channel vulnerabilities in hash tables

# Future work

- Performance optimizations
  - Design efficient data structures for metadata
  - Utilize (or design new) hardware features
- Eliminating other vulnerability classes
  - Heap overflow: a boundless heap
- Survive from memory corruptions
  - Utilize existing error handling functions
  - Runtime exception transformation
    - DangNull transforms use-after-free to null-dereference

# Thank you!

## Research collaborators:

- Wenke Lee (advisor), Taesoo Kim (advisor), William Harris, Chengyu Song, Yeongjin Jang, Wei Meng, Kangjie Lu, Changwoo Min, Sanidhya Kashyap (Georgia Institute of Technology)
- Xinyu Xing (Pennsylvania State University)
- Long Lu (Stony Brook University)
- Billy Lau (Google)
- Tielei Wang (Pangu Team)