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

Applications

Operating system

Hardware
Commodity computers are complex

- Applications
- Operating system
- Hardware
Commodity computers are complex
Commodity computers are complex

- **Applications**
  - Firefox
  - Chrome
  - Safari

- **Operating system**
  - Windows
  - Linux
  - Android

- **Hardware**
  - Processor
  - RAM
Commodity computers are complex

Applications

Operating system

Hardware

9 million lines of code
Commodity computers are complex

Applications

Operating system

Hardware

9 million lines of code

9 million lines of code
Many vulnerabilities

<table>
<thead>
<tr>
<th>Year</th>
<th># of vulnerabilities (CVE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012</td>
<td>5,297</td>
</tr>
<tr>
<td>2013</td>
<td>5,191</td>
</tr>
<tr>
<td>2014</td>
<td>7,946</td>
</tr>
<tr>
<td>2015</td>
<td>6,412</td>
</tr>
</tbody>
</table>

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

- Vulnerability
- Exploitation
- Compromise

Offer unexpected actions
Attacker’s view on vulnerabilities

Vulnerability

Exploitation

Compromise

Offer unexpected actions

Abuse vulnerabilities
Attacker’s view on vulnerabilities

Vulnerability → Offer unexpected actions

Exploitation

Compromise → Run malicious actions

Abuse vulnerabilities
Thesis topics

Vulnerability

Exploitation

Compromise
Thesis topics

1. Eliminating vulnerabilities
   - DangNull [NDSS 15]: Use-after-free
   - CaVer [Security 15]: Bad-casting
Thesis topics

1. Eliminating vulnerabilities
   - DangNull [NDSS 15]: Use-after-free
   - CaVer [Security 15]: Bad-casting
Thesis topics

1. Eliminating vulnerabilities
   - DangNull [NDSS 15]: Use-after-free
   - CaVer [Security 15]: Bad-casting

2. Analyzing vulnerability
   - SideFinder: Timing-channels in hash tables
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

Vulnerability

Exploitation

Compromise
Hacking: the art of exploitation

Vulnerability → Exploitation → Compromise
Hacking: the art of exploitation

Vulnerability

Exploitation

Compromise
Hacking: the art of exploitation

Vulnerability

Exploitation

Compromise
Hacking: the art of exploitation

Vulnerability → Exploitation → Compromise
Hacking: the art of exploitation
Hacking: the art of exploitation

Vulnerability

Exploitation

Compromise
Hacking: the art of exploitation

Vulnerability

Exploitation

Compromise
Hacking: the art of exploitation

Vulnerability

Exploitation

Compromise
Hacking: the art of exploitation

Vulnerability

Exploitation

Compromise
Difficult to prevent all bad things
Difficult to prevent all bad things

Control-flow integrity  Data-flow integrity
Difficult to prevent all bad things
Difficult to prevent all bad things

Control-flow integrity

Data-flow integrity
Difficult to prevent all bad things
Difficult to prevent all bad things
Difficult to prevent all bad things

Look alike a benign behavior
Difficult to prevent all bad things

Control-flow integrity

Data-flow integrity

Look alike a benign behavior
Difficult to prevent all bad things

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

Control-flow integrity

Data-flow integrity

Look alike a benign behavior
Eliminating vulnerabilities

Vulnerability

Exploitation

Compromise
Eliminating vulnerabilities
Eliminating vulnerabilities

Vulnerability

Exploitation

Compromise

No way to bypass in the future
Eliminating vulnerabilities

Vulnerability → Exploitation → Compromise

No way to bypass in the future
Eliminating vulnerabilities

Vulnerability

Exploitation

Compromise

No way to bypass in the future

Transform a program such that a vulnerability never exists.
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

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

doc->child = body;

delete body;

doc->child->getAlign();

*doc

```cpp
Doc
```

*body

```cpp
Body
```

*doc

```cpp
*child
```

*body

```cpp
*child
```
Understanding use-after-free (in detail)

Allocate objects

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

Propagate pointers

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

```cpp
delete body;
```

```cpp
doc->child->getAlign();
```
Understanding use-after-free (in detail)

Allocate objects

```cpp
Doc *doc = new Doc;
Body *body = new Body;
```

Propagate pointers

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

```cpp
delete body;
```

```cpp
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

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

doc->child = body;

delete body;

doc->child->getAlign();
```
Challenges in identifying dangling pointers

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

```cpp
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

Static analysis: inter-procedural and points-to analysis

Dynamic analysis: precise pointer semantic tracking

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

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

```cpp
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

```cpp
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();
```

**Insert shadow obj:**
- Base address of allocation
- Size of 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

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

**Insert shadow obj:**
- Base address of allocation
- Size of Doc

```cpp
delete body;
```
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

Insert shadow obj:
- Base address of allocation
- Size of Doc

Doc *doc = new Doc();

Remove shadow obj:
- Using base address (body)

delete body;
Tracking object relationships

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

```c
doc->child = body;
```
Tracking object relationships

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

```
*doc = &child;
trace(*doc, *child);
```
Tracking object relationships

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

```c
*doc
  *child

*body
```

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

![Diagram](image-url)
Tracking object relationships

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

```c
*doc *
*child

*body

Doc

Body

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

![Diagram showing object relationships and pointer propagations](image)
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

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

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
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!

```cpp
class P {
    virtual ~P() {}
    int m_P;
};
class D: public P {
    virtual ~D() {}
    int m_D;
};
```
Downcasting is not always safe!

```c++
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!

```cpp
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!

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

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

Access scope of P*

Access scope of D*
Downcasting can be bad-casting

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

vtptptr 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

\[
P * pS = \text{new} \ P(); \quad D * pD = \text{static\_cast}<D*>(pS); \quad pD->m_D;
\]

Memory corruptions
Downcasting can be bad-casting

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

Memory corruptions
Downcasting can be bad-casting.

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

& (pD->m_D)
```

Memory corruptions
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
```

```
ContainerNode

Element

HTMLElem

SVGElem

HTMLUnknownElement
```
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

2. Upcasting

3. Downcasting
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

```cpp
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
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 \textit{ptr} points to?
\Rightarrow \textit{Runtime type tracing}

Q. What are the class relationships b/w \textit{P} and \textit{D}?
\Rightarrow \textit{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

```
  hash("P")
  ...
  hash("D")
  hash("P")
  ...
```
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("P")
hash("D")
hash("P")
...
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

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

THTable (P)
```
```
hash("D")
hash("P")
...

THTable (D)  Unrolled linearly
```
Runtime type tracing

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

P *ptr = new P;

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

Object (P)

&THTable(P)

hash(“P”)

...
Runtime type tracing

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

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

Diagram:
- `P *ptr = new P;`
- `trace(ptr, &THTable(P));`
- `&THTable(P)`
- `hash("P")`
- `...`
- `ptr`
Runtime type tracing

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

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

Maintain an internal mapping from objects to metadata

Heap: Alignment based direct mapping
Stack: Per-thread red-black tree
Global: Per-process red-black tree
Runtime type verification

\[ \text{static\_cast}<D*>(ptr); \]
Runtime type verification

static_cast<D*>(ptr);
Runtime type verification

\[
\text{static\_cast}\langle \text{D}\rangle\ast\langle\text{ptr}\rangle;
\]

![Diagram showing object (P) and type table verification](image)
Runtime type verification

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

1. Locate metadata associated with the object
Runtime type verification

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

2. Locate associated THTable

- &THTable(P)
- hash(“P”)
- THTable (P)
Runtime type verification

\[
\text{static\_cast}\langle\text{D}\rangle\text{\_}\ast\rangle\text{\_}(\text{ptr});
\]
Runtime type verification

\[
\text{static\_cast}\langle D^*\rangle(ptr);
\]

3. Enumerate THTable and check if hash(“D”) exists.
Runtime type verification

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

THTable(P) does not have D → Bad-casting!
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)

```
inode_hash(sb, ino1)
inode_hash(sb, ino2)

#0 bucket
#1 bucket
#2 bucket
#3 bucket

inode_hashtable

inode1 → inode2
```
Case study on inode cache

• **inode_hash_table**
  • 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_hash_table
  • **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)

![Diagram of inode_hash function and inode_hashtable]

- inode_hash(sb, ino₁)
- inode_hash(sb, ino₂)
- ... (up to inode_hash(sb, inoₖ))
- #0 bucket
  - #1 bucket
    - #2 bucket
      - #3 bucket
        - inode₁
        - inode₂
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)

```
inode_hash(sb, ino₁)
inode_hash(sb, ino₂)
...,
inode_hash(sb, inoₖ)
```

```
#0 bucket
#1 bucket
#2 bucket
#3 bucket
```

```
inode₁ → inode₂ → ... → inodeₖ
```

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
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

1. Backward slicing
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

inode_hash(sb, ino)

1. Backward slicing
2. Concolic execution
An example of workflow: inode cache

1. Backward slicing
2. Concolic execution
An example of workflow: inode cache

1. Backward slicing
2. Concolic execution
An example of workflow: inode cache

1. Backward slicing
2. Concolic execution
An example of workflow: inode cache

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

<table>
<thead>
<tr>
<th>FS</th>
<th>#S</th>
<th>True src. description</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>ext2</td>
<td>1</td>
<td>inode@ext2_inode_by_name()</td>
<td>✓</td>
</tr>
<tr>
<td>ext4</td>
<td>11</td>
<td>inode@ext4_lookup()</td>
<td>✓</td>
</tr>
<tr>
<td>reiserfs</td>
<td>5</td>
<td>k_objectid@reiserfs_iget()</td>
<td>✓</td>
</tr>
<tr>
<td>jfs</td>
<td>3</td>
<td>inum@jfs_lookup()</td>
<td>✓</td>
</tr>
<tr>
<td>btrfs</td>
<td>4</td>
<td>objectid@btrfs_iget_locked()</td>
<td>✓</td>
</tr>
<tr>
<td>xfs</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Evaluation on inode cache attacks
- Backward slicing

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

# 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)

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

N/A
Evaluation on inode cache attacks
- Backward slicing

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

SideFinder successfully identified all true sources
Evaluation on inode cache attacks
- Backward slicing

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

XFS implements its own hash table
Evaluation on inode cache attacks
- Concolic execution and synthesizing

Complexity: the number of clauses in a bucket expression

<table>
<thead>
<tr>
<th>FS</th>
<th>Bucket</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># OP</td>
<td></td>
</tr>
<tr>
<td>ext2</td>
<td>48</td>
<td>319</td>
</tr>
<tr>
<td>ext4</td>
<td>50</td>
<td>336</td>
</tr>
<tr>
<td>reiserfs</td>
<td>48</td>
<td>321</td>
</tr>
<tr>
<td>jfs</td>
<td>48</td>
<td>318</td>
</tr>
<tr>
<td>btrfs</td>
<td>50</td>
<td>324</td>
</tr>
<tr>
<td>xfs</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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)