
Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Lots of Vulnerability Types

• We have discussed several types of vulnerabilities
  ‣ Virus/worm
  ‣ Trojan horse
  ‣ Drive-by-downloads
  ‣ Buffer overflow (overwrite and overread)
  ‣ Integer overflow
  ‣ Resource retrieval
  ‣ …

• Are there other types of vulnerabilities to consider?
New/Other Vulnerabilities

- Type Casting
  - Bad casting and type confusion

- Flaws in roots of trust
  - Forge integrity measurements

- Abuse of system management
  - Windows Management Instrumentation

- What can we do to find vulnerabilities before adversaries?
Type Casting

- What is the purpose of type casting?
What Could Go Wrong?

- **Type casting**
  - Enables memory reference of one type to be processed using the features of another type
  - **Fields**
  - **Methods**
  - **Hierarchy**
  - Types may not be the same size
    - Such type casts would be invalid
      - Check statically or dynamically
  - What happens if we cast a reference to a type of a larger size?
Invalid Type Cast

- Why is this type cast invalid?

![Class Inheritance Diagram]

**Diagram Explanation:**
- **ContainerNode**: Represents a container node in the document object model (DOM).
- **Element**: Base class for all DOM elements.
- **HTMLUnknownElement**: Represents unknown elements.
- **HTMLElemnt**: Represents elements in the HTML namespace.
- **SVGElement**: Represents elements in the SVG namespace.
- **OwnerDocument**: Represents the document in which the node is placed.
- **(Allocated)**: Indicates that the class is allocated.
- **(56 siblings)**: Indicates the number of siblings in the DOM.
- **(size: 96 bytes)**: Indicates the size of the class in bytes.
- **static_cast**: Indicates a type cast operation.

**Figure 1:** Inheritance hierarchy of classes involved in the CVE-2013-0912 vulnerability.

**Figure 2:** A snippet of the actual report of the CVE-2013-0912 vulnerability.

**Figure 3:** Shows a detailed view of the allocation and casting workflow.
Type Cast Checking

- Static or dynamic (explicit in C++)
  - Upcast to parent
  - Downcast to originally allocated class

  Why are these safe?

- Static type checking
  - Check against valid casting rules at compile time

- Dynamic type checking
  - Check at runtime as origin may be unpredictable

```
class SVGElement: public Element { ... }; pDom (allocated)
Element *pDom = new Element();
SVGElement *pCanvas = new SVGElement();

// (1) valid upcast from pCanvas to pEle
Element *pEle = static_cast<SVGElement*>(pCanvas);

// (2) valid downcast from pEle to pCanvasAgain (== pCanvas)
SVGElement *pCanvasAgain = static_cast<SVGElement*>(pEle);

// (3) invalid downcast (-> undefined behavior)
SVGElement *p = static_cast<SVGElement*>(pDom);

// (4) leads to memory corruption
p->m_className = "my-canvas";

// (5) invalid downcast with dynamic_cast, but no corruption
SVGElement *p = dynamic_cast<SVGElement*>(pDom);
if (p) {
    p->m_className = "my-canvas";
}
```
Type Cast Checking

- Dynamic is reliable, but expensive
  - 90X slower than static cast
  - Banned in Chrome development
- Sound familiar?
  - So what can we do?
CAVER

- Dynamic is reliable, but expensive
  - 90X slower than static cast
  - Banned in Chrome development

- Sound familiar?
  - So what can we do?
  - Detect vulnerability
    - CAVER – focus on flaw of casting to larger data type
    - Ignores adversary accessibility and exploitation
      - Could they help?
CAVER Requirements

• **THTable**
  ‣ Basic idea: Given a pointer to an object allocated as type $T$, the THTable contains the set of all possible types to which $T$ can be casted.

• **Type relationships**
  ‣ **IS-A** – inheritance
    • Class and subclass that extends class
  ‣ **HAS-A** – composition
    • Class and its member variables of different types

• Also, type size and type name
CAVER Requirements

• Secure vs. allowable casts

• Phantom classes
  ‣ We say a class P is a phantom class of a class Q if two conditions are met: (1) Q is directly or indirectly derived from P; and (2) compared to P, Q does not have additional member variables or different virtual functions.
  ‣ I.e., Q is same size as P
  ‣ I.e., Cannot cast of memory variable to another type

• So, casting from P to Q cannot be used to access data beyond the allocated type boundary
CAVER Use

- Instrument and Evaluate
  - A reference to the THTable and the base address of the object

```c
// Heap objects (dynamically allocated)
void func_heap_ex() {
    C *p_heap_var = new C;
    C *p_heap_array = new [num_heap_array];
    + trace_heap(&THTable(C, p_heap_var, i));
    + trace_heap(&THTable(C, p_heap_array, num_heap_array);
    ...
}

// Stack objects
void func_stack_ex() {
    C stack_var;
    + trace_stack_begin(&THTable(C, &stack_var, i));
    ...
    + trace_stack_end(&stack_var);
}

// Global objects
C global_var;

// .ctors: (invoked at the program’s initialization)
void trace_global_helper_1() {
    + trace_global(&THTable(C, &global_var, i));
}

// Verifying the correctness of a static casting
void func_verify_ex() {
    B *afterAddr = static_cast<A*>(beforeAddr);
    + verify_cast(beforeAddr, afterAddr, type_hash(A));
}
```

**Example 2** An example of how CAVER instruments a program. Lines marked with + represent code introduced by CAVER, and &THTable(T) denotes the reference to the THTable of class T. In this example, we assume that the THTable of each allocated class has already been generated by CAVER.

---

<table>
<thead>
<tr>
<th>Name</th>
<th># of tables</th>
<th># of verified cast</th>
</tr>
</thead>
<tbody>
<tr>
<td>THTable</td>
<td>192</td>
<td>192</td>
</tr>
</tbody>
</table>

**Table 4**: Comparisons of protection coverage between UBSAN and CAVER. In the # of tables column, THTable shows the number of virtual function tables and THTable shows the number of type hierarchy tables, each of which is generated to build the program. # of verified cast shows the number of static_cast instrumented in UBSAN and CAVER, respectively. Overall, CAVER covers 241% and 199% more classes and their castings, respectively, compared to UBSAN.

<table>
<thead>
<tr>
<th>Name</th>
<th>File Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>483.xalancbmk</td>
<td>6,111</td>
</tr>
<tr>
<td>450.oxplex</td>
<td>466</td>
</tr>
<tr>
<td>Chromium</td>
<td>249,790</td>
</tr>
<tr>
<td>Firefox</td>
<td>242,704</td>
</tr>
</tbody>
</table>

**Table 5**: The file size increase of instrumented binaries: CAVER incurs 64% and 49% less storage overheads in Chromium and Firefox browsers, compared to UBSAN.

**Figure 4**: Browser benchmark results for the Chromium browser. On average, while CAVER-naive incurs 30.7% overhead, CAVER showed 7.6% runtime overhead after the optimization. UBSAN exhibits 16.9% overhead on average.
Attacks on Roots of Trust

- **Static Root of Trust Measurement**
  - In BIOS
  - Can it be compromised without detection?
  - Can the adversary prevent reflashing from removing malware?
Attacks on Roots of Trust

- Static Root of Trust Measurement (SRTM)
  - In BIOS
  - Can it be compromised without detection?
  - Can the adversary prevent reflashing from removing malware?
SRTM Measurement

• Recall authenticated boot and IMA
  ‣ SRTM measured into PCR0
    • Do we measure all?
    • Not necessarily – only measure first 64 bytes – assume will be sufficient to detect any changes within the compressed module
    • Plus measure 0 for other cases
  ‣ By a function invoked via a function pointer
    • Can this be overwritten?
    • Yep.
Attack Vectors

- Kernel modules
  - BIOS flash chip can be directly overwritten by a kernel module unless provisions are implemented by the BIOS manufacturer to prevent this from occurring.

- SMM
  - The OS uses SMM to modify the BIOS, then continue without rebooting the system.
  - Attacks from SMM will be difficult to prevent as its use is ad hoc.
Attack Types

• **Tick**
  ‣ to be a piece of parasitic stealth malware that attaches itself to the BIOS to persist, while hiding its presence by forging the PCR0 hash.
  ‣ E.g., small patch can record known good hash

• **Flea**
  ‣ We define a flea as parasitic stealth malware that, like a tick, forges PCR0, but is additionally capable of transferring itself (“hopping”) into a new BIOS image when an update is being performed.
  ‣ E.g., control BIOS update to avoid overwriting
Attacks on System Mgmt

• System management is useful
  ‣ Local and remote query and configuration of critical host software
  ‣ E.g., antivirus

• System management is also useful for attackers
  ‣ Perform system reconnaissance, AV and VM detection, code execution, lateral movement, persistence, and data theft
  ‣ Even without adding a file to the disk

• For both local and remote access
Windows Mgmt Instrumentation

- **System Architecture**

The structure/schema of the vast majority of WMI object is described in Managed Object Format (MOF) files. MOF files use a C++ like syntax and provide the schema in which the generated data is formatted. From a defender’s perspective, it is worth noting that WMI object definitions can be created without a MOF file. Rather, they can be inserted directly into the CIM repository using some basic .NET code.

Transmitting Data

Microsoft provides two protocols for transmitting WMI data remotely: DCOM and Windows Remote Management (WinRM).

Performing Actions

Some WMI objects include methods that can be executed. For example, a common method executed by attackers for performing lateral movement is the static Create method in the Win32_Process class. WMI also provides an eventing system whereby users can register event handlers upon the creation, modification, or deletion of any WMI object instance.

Figure 1: A high-level overview of the WMI architecture
Attack Surface

• Examples

• Reconnaissance
  ‣ Installed AV products will typically register themselves in WMI

• Arbitrary code execution
  ‣ The Win32_Process class contains a static method - Create - that can spawn a process locally or remotely.

• Data storage
  ‣ Creation of WMI classes dynamically and storing arbitrary data
Attack Scenario

• Use “WMI events” to launch attack
  ‣ Instance creation event
  ‣ Also, via Start Menu or Registry
• Use “WMI shell” for initiating Command & Control
• Store attacker data in “WMI class”
• Load malicious “WMI provider” to hide attack
• Install malicious “WMI Service”
Prevent Vulnerabilities

- Can we devise a semi-systematic method to prevent adversaries from exploiting vulnerabilities?
  - Assuming we cannot remove all program flaws…
Prevent Vulnerabilities

- Attack surface
  - Identify where an adversary can provide input
    - Adversary accessibility
  - Remote WMI requests, BIOS updates, references

- Think like an adversary
  - Whenever an adversary can perform an operation, consider adversarial action
    - Do reconnaissance, Add malicious event, etc.

- Detect program response to adversarial action
• We actively change the namespace whenever an adversary can write to a binding used in resolution
  ‣ Fundamental problem: adversaries may be able to write directories used in name resolution

• Use adversary model to identify program adversaries and vulnerable directories [ASIACCS 2012]
Launch Phase

1. Find bindings
2. Find adversary access
3. Launch attack (modify namespace)
4. Continue system call

```
fd = open("/var/mail/root", O_APPEND)
```

User-space

Kernel

```
delete("/var/mail/root");
symlink("/etc/passwd", "/var/mail/root")
```
Detect Phase

1. Victim accepts resource
2. Record vulnerability
3. Rollback namespace
4. Restart system call

write(fd)

User-space

Kernel

Victim (user root)

/etc
/var
/mail
/passwd
/root (symbolic link)
Prevent Vulnerabilities

- **Attack surface**
  - Adversary control bindings in name resolution
  - Adversary control references being cast (other inputs)
  - Adversary control BIOS updates
  - Adversary controls WMI queries

- **Think like an adversary**
  - What would enable privilege escalation?

- **Detect program response to adversarial action**
  - Detect escalation (CAVER?)
Take Away

• Several types of vulnerabilities
  ‣ But here are some new-ish ones to consider

• Other program operations that enable unexpected execution
  ‣ Type casting
  ‣ STRM (BIOS updates impact)
  ‣ WMI

• For each program operation, what do defenders need to consider to prevent exploitation?