Identifying Memory Corruption Bugs with Compiler Instrumentations

이병영 (조지아공과대학교)
blee@gatech.edu

@POC2014

How to find bugs

- Source code auditing
- Fuzzing

Source Code Auditing

- Focusing on specific vulnerability patterns
  - integer overflow:)

- Focusing on newly introduced code bases
  - Keep track of commit logs

- Deeply understand complicated logics
  - Complex ⇒ More mistakes!

Fuzzing

- Why Fuzzing?
  - Simple. Just need to know the input format.
  - Understanding code logics in major OSS is too difficult.
  - Many modern (C++) bugs are too complicated.
    - Use-after-free
    - Bad-casting

Lessons from Futex

- Linux kernel futex local privilege escalation (CVE-2014-3153)
  - Found by Pinkie Pie
  - Android TowelRoot by GeoHot

Don’t know whether Pinkie Pie found this bug by fuzzing, but Trinity already triggered the issue.

Old days: Fuzzing with Debuggers

- A debugger’s role in fuzzing
  - Catch the (crashing) exception, and report!
- Number of debuggers
  - WinDbg, GDB, PyDbg, ...
- What was the problem?
  - a crash != a security bug, but too many crashes!

Crashes != Security Bugs

- **How the bug manifests itself in debuggers?**
  - Stack overflow, Integer overflow, Heap overflow, double-free
  - Use-after-free, Use-after-return, Uninitialized Memory Read, Bad-Casting
  - …

⇒ Memory Access Violation (Windows)
or Segmentation Fault (Linux)

Still old days: !exploitable

- A Windows debugging extension
  - For automated crash analysis and security risk assessment.
- Full of heuristic analyses
  - Whether return addresses or heap meta-data are controllable.

New direction: Compiler Instrumentations

- Collect execution contexts at runtime
- Monitor whether the program violates any of guarantees/assumptions

Tools and Coverage

Address Sanitizer	Stack/Heap overflows, Double-free, Use-after-free
Memory Sanitizer	Uninitialized Memory Read
Thread Sanitizer	Data races
UndefinedBehavior Sanitizer	Most of undefined behaviors (Signed overflows, Bad-castings, etc)
Dangle Nullifier	Use-after-free

{Address|Memory|UndefinedBehavior} Sanitizer

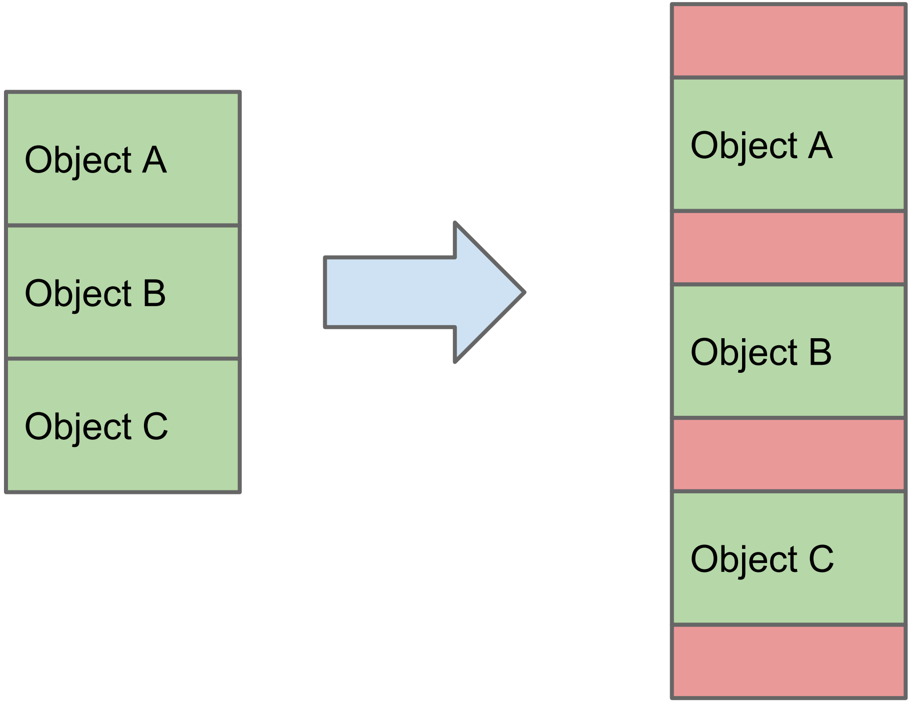
- Available from LLVM or GCC
  - e.g., -fsanitize=address for Address Sanitizer

- Heavy users
  - Fuzzing framework for major browser vendors
    - Chromium, Firefox, …
  - Debugging/Fuzzing for server side implementations
    - Google search engines, Youtube back-ends
  - Individual fuzzer developers

Address Sanitizer

- Shadow Memory
  - Maintain truly addressable regions
  - Hooking all memory allocation functions
    - Stack variable allocations
    - Global variable allocations
    - Heap allocations
      - (e.g., malloc()/free(), new/delete operator, etc)
  - Map real 8 bytes into 1 shadow byte

Address Sanitizer



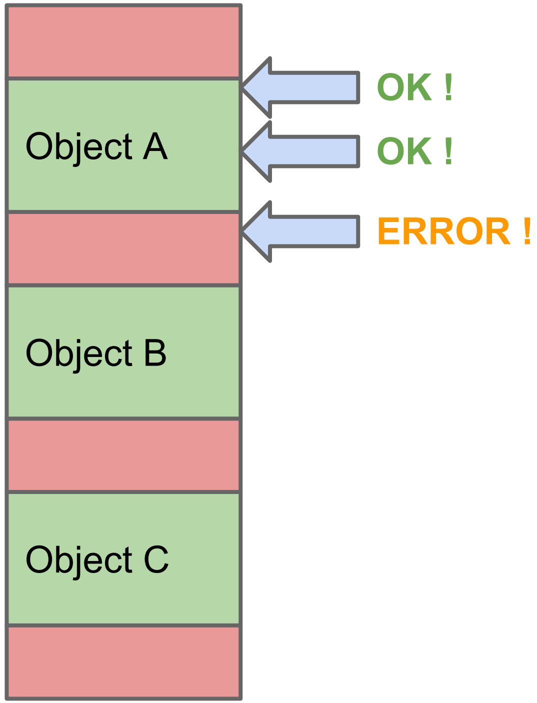
Shadow memory knows only green blocks are addressable.

Address Sanitizer

- All the memory read/write instructions are instrumented.
  - Always check with shadow memory if it is addressable.

Address Sanitizer

- Stack / Heap overflows
    - Immediately identifying the bug once it hits the red-zone.



Address Sanitizer: Use-after-free

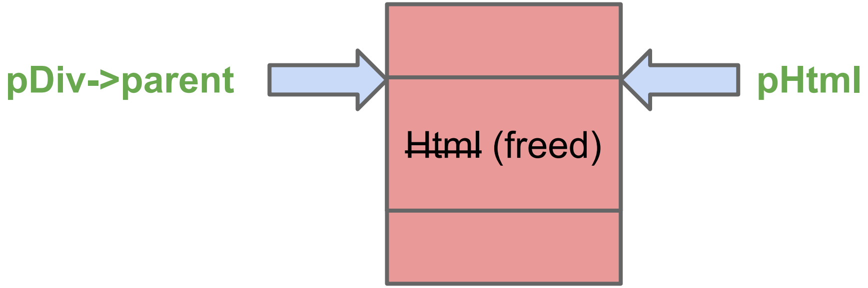
```
Div *pDiv = new Div;
Html *pHtml = new Html;
pDiv->parent = pHtml;

...

delete pHtml; // free

...

pDiv->parent->... // Use-after-free !
```



Address Sanitizer: Use-after-free in Practice

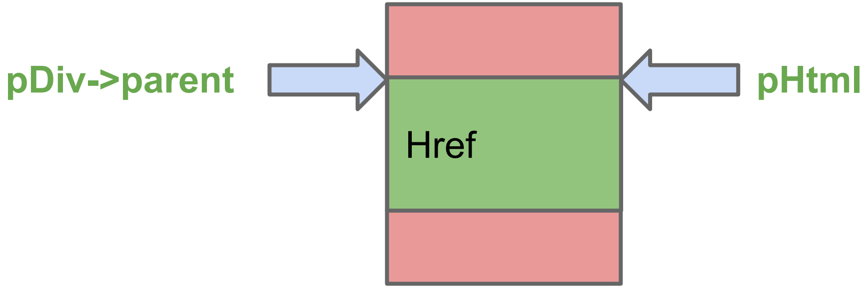
```
Div *pDiv = new Div;
Html *pHtml = new Html;
pDiv->parent = pHtml;

...

delete pHtml; // free

new Href; // x1000

pDiv->parent->... // Looks Valid !
```



This is why use-after-free is difficult to detect with typical debuggers!

Address Sanitizer:
How to handle use-after-free

- Quarantine zone
  - **Do not re-use** freed memory blocks if possible
  - Default size : 256MB

Address Sanitizer:
How to handle use-after-free

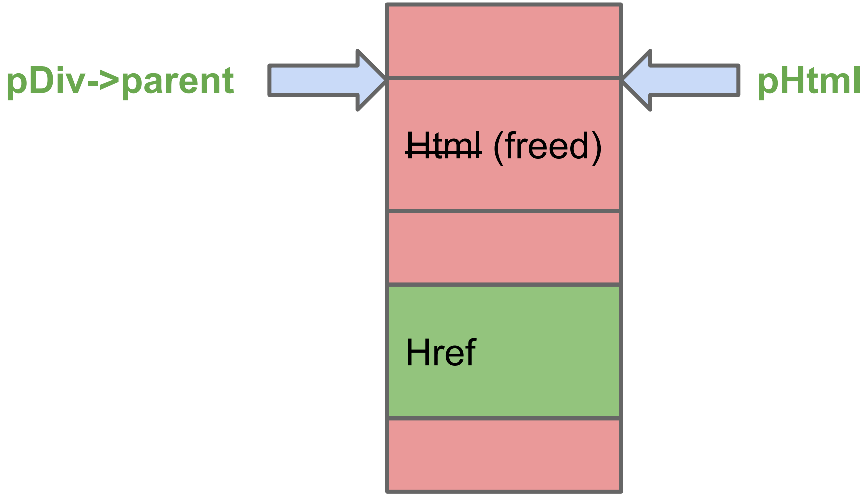
```
Div *pDiv = new Div;
Html *pHtml = new Html;
pDiv->parent = pHtml;

…

delete pHtml; // free

new Href; // say 1000 times!

pDiv->parent->... // Hitting red-zone!
```



Address Sanitizer

DEMO

Bad-casting (or Type Confusions)

- Downcasting
  - Casting a reference/pointer to one of its derived classes.
- Bad-casting
  - A destination object is an incomplete object of the target type
  - c.f., std::bad_cast

Bad-casting: simple examples

```
class S {
public:
  virtual ~S() {}
  int m_s;
  ...
};

class T: public S {
public:
  virtual ~T() {}
  int m_t;
  ...
};

S *ps = new S();
T *pt = static_cast<T*>(ps); // Bad-casting !
```

Bad-casting and Security

Memory region for T::m_t can be corrupted using T* pt as it was never allocated.

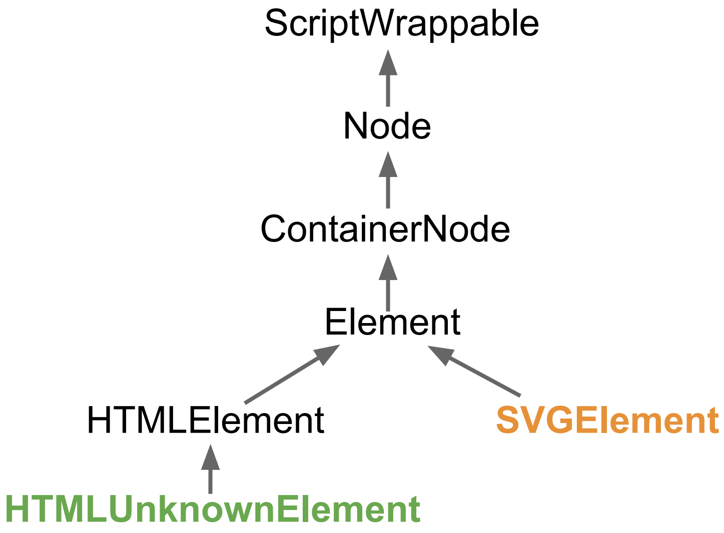
Bad-casting and Exploitability

- Overwrite meta-data in the class
  - virtual function table pointers
    - Multiple vtable ptrs in the class
    - Forge the vtable and then jump
  - Length variables
    - Vector length variables
    - Info-leak or trigger additional heap overflows

- Overwrite other objects’ meta-data

Bad-casting and Exploitability

- **CVE-2013-0912**
  - Bad-casting from **HTMLUnknownElement** to **SVGElement**
  - Used to exploit Chrome's renderer process in Pwn2Own 2013



sizeof(**HTMLUnknownElement**) ⇒ 96
sizeof(**SVGElement**) ⇒ 168

Extra (168-96) bytes were writable via bad-casting

Bad-casting and Exploitability

```
SVGElement* SVGViewSpec::viewTarget() const {
  if (!m_contextElement)
    return 0;

  return static_cast<SVGElement*>(
         m_contextElement->treeScope()->getElementById(m_viewTargetString));
}
```

What's the runtime type where the expression points to?

How to avoid bad-casting

- Naive guideline to avoid bad-casting
  - static_cast for upcasting
  - always dynamic_cast otherwise

- Issues
  - dynamic_cast is slow
  - dynamic_cast is not allowed in many large scale software
    - -fno-rtti

How to catch bad-casting

- Identity predicates (in Blink)
  - Implement identity virtual functions for each type
  - assert() with the predicate
  - Effective, but difficult to scale

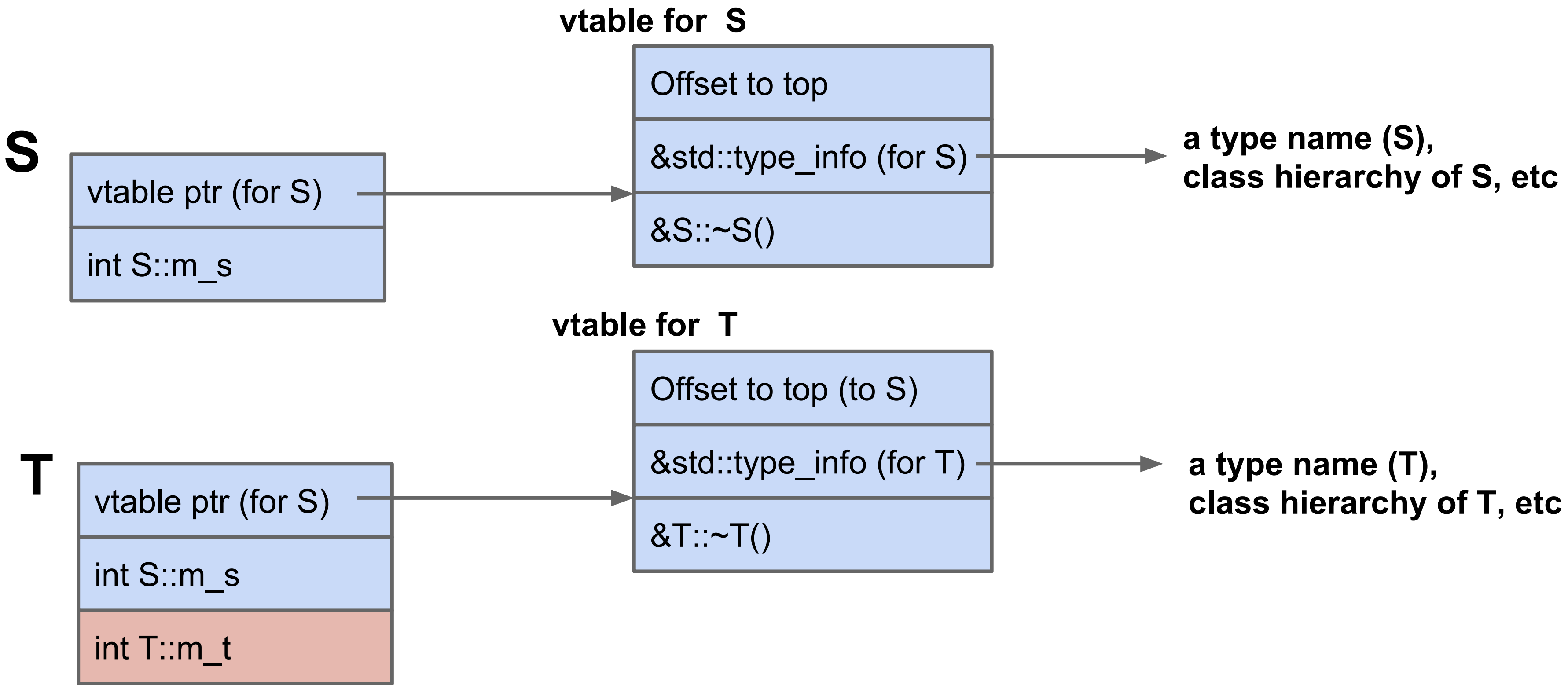
- dynamic_cast in debug builds (in ProtoBuf)
  - assert(p==NULL||dynamic_cast<T>(p)!=NULL)
  - Slow, and RTTI is required.

UndefinedBehavior Sanitizer (UBSan)

- UBSan catches various undefined behaviors
  - Signed integer overflow, out of array bound accesses, etc
  - Implemented in Clang/Compiler-rt

- UBSan vptr: -fsanitize=vptr
  - Detect any use of an object where vptr indicates the wrong dynamic type

How UBSan vptr works: Utilize C++ABI and RTTI



How UBSan vptr works

- At compile time (Clang)
  - For any operations on polymorphic class types, invoke a sanity check function, **type_check**()

- (Simplified) Instrumentation example

```
static_cast<T*>(ps);
```

→

```
type_check(typeinfo of T*, ps);
static_cast<T*>(ps);
```

How UBSan vptr works

- At runtime (Compiler-rt)
  - Parse RTTI information given a base pointer of an object
  - Check whether the operation is valid

How UBSan vptr works

- Caching to speed up
  - Store the checked results
    - Hash(type_name || vtable ptr)
  - Hash collisions?!

    ⇒ Minimize impacts with ASLR (especially for 64-bit targets!)
- Instrumentation example

```
static_cast<T*>(ps);
```

```
if Hash( “T” || vtable ptr of the object where ps points to) does not exist:
    type_check(typeinfo of T*, ps);
static_cast<T*>(ps);
```

UndefinedBehavior Sanitizer

DEMO

Limitations of Address Sanitizer

- Address Sanitizer may miss use-after-free
- Abusing quarantine zone
  - Keep allocating buffers to force the re-allocation.
- Cannot detect the bug if it accesses beyond red-zones.

Simplified Chrome exploits (CVE-2012-5137)

```
function onOpened() {
  buf = ms.addSourceBuffer(...);
  // disconnect the target obj
  vid.parentNode.removeChild(vid);
  vid = null;
  // free the target obj
  gc();
  var drainBuffer = new Uint32Array(1024*1024*512);
  drainBuffer = null;
  // drain the quarantine zone
  gc();

  for (var i = 0; i < 500; i ++) {
    // allocate/fill up the landing zone
    var landBuffer = new Uint32Array(44);
    for (var j = 0; j < 44; j ++)
      landBuffer[j] = 0x1234;
  }
  // trigger use-after-free
  buf.timestampOffset = 100000;
}

ms = new WebKitMediaSource();
ms.addEventListener('webkitsourceopen', onOpened);
vid = document.getElementById('vid');
vid.src = window.URL.createObjectURL(ms);
```

Force to re-allocate freed buffers. Just like heap-spraying!

Limitations of UBSan vptr

- Difficult to deploy
  - RTTI ⇒ Requires Blacklisting

- Cannot handle non-polymorphic class types.

Conclusions

- Compiler instrumentations tools are useful!
  - It is very easy to use

- Extremely useful for fuzzing
  - Easy bug identification!

감사합니다!