IntFlow: Integer Error Handling With Information Flow Tracking

Marios Pomonis    Theofilos Petsios
Kangkook Jee     Michalis Polychronakis
Angelos D. Keromytis

December 7, 2014

Columbia University
Example

1. `img_t *table_ptr;`
2. `unsigned int num_imgs = get_num_imgs();`
3. `unsigned int alloc_size = sizeof(img_t) * num_imgs;`
4. `table_ptr = (img_t *) malloc(alloc_size);`
5. `for (i = 0; i < num_imgs; i++)`
6. `table_ptr[i] = read_img(i);`
Integer Errors

- Mathematical representation vs machine representation
- Instances:
  - Integer overflow/underflow
  - Precision loss
  - Signedness change
Characteristics

- **Mainly C/C++ specific:**
  - Signed integers only (Java, Python)
  - Overflow protection (Python)
- **Undefined:**
  - Negative → unsigned
  - INT_MAX + 1
  - Optimizations
  - Expected behavior
Importance

- Can lead to buffer overflows, memory leaks etc...
  - Integral part of exploits
  - Erroneous memory allocation
- Integer overflow in top 25 most dangerous software errors
- > 50 vulnerability reports (CVE) in 2014
  - QuickTime → Signedness change
  - launchd (iOS) → Integer overflow
  - Wireshark → Signedness change
  - Google Chrome → Integer overflow

mpomonis@cs.columbia.edu  IntFlow  Columbia University  6 / 29
Clang AST

Dangerous operation
- Static: operation → safe function
- Dynamic: detect errors
- Report and (optionally) abort

Clang trunk v3.3

/* a = b + c */
bool error = false;
a = safe_add(b, c, error);
if (error)
    report();
Dynamic detection mechanism
Offline use
Input set from user
IOC Issue

- Overly comprehensive
- Lack of severity level
- Error $\neq$ vulnerability
Developer Intended Violations

- Idioms → errors
- Controlled
  - Expected behavior
  - Not affected by attacker
- IOC → report all
  - Large list
  - Manually distill critical errors

Examples

```c
umax = (unsigned) -1;
neg = (char) INT_MAX;
smax = 1 << (WIDTH - 1) - 1;
smax++;```

mpomonis@cs.columbia.edu  IntFlow  Columbia University
Goals:

1. Eliminate reports of developer intended violations
2. Retain and highlight critical error reports
Challenges:

1. Can we identify potential vulnerabilities?
2. Can we identify potentially exploitable vulnerabilities?
3. Can we do it accurately?
Critical Arithmetic Errors

An error is potentially **critical** if:

1. **Untrusted source** $\rightarrow$ arithmetic error
   e.g. `read()`, `getenv()`...

   OR

2. Arithmetic error $\rightarrow$ **sensitive sink**
   e.g. `*alloc()`, `strcpy()`...
Static Information Flow Tracking

- Set of techniques analyzing data-flow
- Common compiler methodology
- Distinguishes flows to/from integer operations

**Pros**
- ✓ No runtime overhead
- ✓ Coverage

**Cons**
- ✗ Accuracy
- ✗ Scalability

mpomonis@cs.columbia.edu  IntFlow  Columbia University  15 / 29
c = read();

b = gettimeofday();

c = -5;

b = 1;

a = safe_add(b, c, error);

if (error)
    report();
Forward Slicing: Source $\rightarrow$ Operation

```
b = gettimeofday();

send(b);

return b;

a = safe_add(b, c, error);
if (error)
    report();
```
c = read();

return c;

write(c);

a = safe_add(b, c, error);
if (error)
    report();
Sources Examination

If sources = trusted → result = developer intended

```c
a = safe_add(b, c, error);
if (error)
    report();

b = gettimeofday();
c = -5;
b = 1;
c = read();
```
Remove IOC Check

```c
b = 1;
c = -5;
a = safe_add(b, c, error);
if (error)
    report();

b = gettimeofday();
c = read();
```
IntFlow: Architecture

C/C++ → Clang AST → LLVM-IR → Arith. Operations Instrumented → IFT Integration

IFT Integration: Trusted/Untrusted Input Tracking, Sensitive Operation Tracking

Compile, Link → Intflow binary → Runtime Execution → Integer error reports

Input Set
Sensitive Operations

- Dynamic detection
- Operations → sensitive functions
- Operation → bit
- Check before a sensitive function
- Report if any bit set

<table>
<thead>
<tr>
<th>Id</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
</tbody>
</table>

... 
1: x--; 
...
2: x = y * z;
...
3: a++; 
\[ x *= a; \]
...
4: x++; 
...
check_flags(); 
g = malloc(x);
Modes Of Operation

- Blacklisting mode
  - **Untrusted sources** → operation
- Whitelisting mode
  - **Trusted sources** → operation
- Sensitive mode
  - Operation → **sensitive sinks**
- Combination of modes
  - Blacklisting/Whitelisting + Sensitive
  - ↑ Confidence - ↓ Completeness
Evaluation

- Whitelisting mode
  - Flexible
  - Context agnostic
    - ✓ Untrusted sources
    - ✓ Error propagation
  - Upper bound on report number
Number of Reported Arithmetic Errors

- gzip
- vpr
- gcc
- crafty
- parser
- perlbench
- gap
- vortex

IOC Intended
IOC Critical
IntFlow Intended
IntFlow Critical
Real-world Applications

- Detected vulnerabilities:

<table>
<thead>
<tr>
<th>CVE Number</th>
<th>Application</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2009-3481</td>
<td>Dillo</td>
<td>Integer Overflow</td>
</tr>
<tr>
<td>CVE-2012-3481</td>
<td>GIMP</td>
<td>Integer Overflow</td>
</tr>
<tr>
<td>CVE-2010-1516</td>
<td>Swftools</td>
<td>Integer Overflow</td>
</tr>
<tr>
<td>CVE-2013-6489</td>
<td>Pidgin</td>
<td>Signedness Change</td>
</tr>
</tbody>
</table>

- Produced reports

<table>
<thead>
<tr>
<th></th>
<th>Overall</th>
<th>Dillo</th>
<th>GIMP</th>
<th>Swftools</th>
<th>Pidgin</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOC</td>
<td>330</td>
<td>31</td>
<td>231</td>
<td>68</td>
<td>0</td>
</tr>
<tr>
<td>IntFlow</td>
<td>82</td>
<td>26</td>
<td>13</td>
<td>43</td>
<td>0</td>
</tr>
</tbody>
</table>
Runtime Overhead

- Offline use
- CPU-bound (e.g. grep): 50-80%
- IO-bound (e.g. nginx): 20%
Summary

- Coupled IFT with IOC
- Identified critical errors
- Focused on potentially exploitable vulnerabilities
- Code:
Backup Slides
Additional Evaluation Results

- Independent stress test (red team)
  - Artificial vulnerabilities in popular applications
  - IO Inputs
    - Good: no exploit $\rightarrow$ normal execution
    - Bad: exploit $\rightarrow$ detect and abort
  - Aggregate result ($\frac{TP+TN}{Total}$): 79.30%