IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution

Tielei Wang\textsuperscript{1}, Tao Wei\textsuperscript{1}, Zhiqiang Lin\textsuperscript{2}, Wei Zou\textsuperscript{1}
\textsuperscript{1}Peking University, China
\textsuperscript{2}Purdue University
Outline

- Motivation
- Case Study
- Modeling
- Challenges & Approaches
- Implementation & Evaluation
- Related Work
- Conclusion
What is Integer Overflow?

- An integer overflow occurs when an operation results in a value greater than the maximum one of the integral data type.

```c
unsigned int a = 0xffffffff;
unsigned int b = 0x1;
a = a + b; //now, a is 0!
```

- Integer overflow vulnerability is an underestimated threat.
The # of integer overflow vulnerabilities grows rapidly
Integer Overflow Vulnerabilities affected various kinds of software

- **OS Kernel**
  - CVE-2008-4036 (Windows XP, Server 2003, Vista)
  - CVE-2008-3276 (Linux)
  - CVE-2008-4220 (Mac OS)
  - CVE-2008-1391 (NetBSD)
  - ...

- **Libraries**
  - CVE-2008-2316 (Python)
  - CVE-2008-5352 (JAVA)
  - ...

- **Applications**
  - CVE-2008-0726 (Adobe Reader)
  - CVE-2008-4061 (Firefox)
  - CVE-2008-2947 (IE7)
  - CVE-2008-0120 (PowerPoint)
  - CVE-2008-1722(CUPS)
  - CVE-2008-2430(VLC)
  - CVE-2008-5238(Xine)
  - ...
Most of Integer Overflow Vulnerabilities are dangerous

- According to Common Vulnerability Scoring System (CVSS), more than 60% of Integer Overflow vulnerabilities have the highest severity score.

Data Source: Common Vulnerability Scoring System
Outline

- Motivation
- **Case Study**
- Modeling
- Challenges & Approaches
- Implementation & Evaluation
- Related Work
- Conclusion
What are the common features of integer overflow vulnerabilities?

```c
unsigned int x = read_int();
if ( x > 0x7fffffff )
    abort();
unsigned int s = x*sizeof(int);
char* p = malloc(s);
read_int_into_buf(p, x);
```

- **an untrusted source**
- **an incomplete check**
- **an integer overflow**
- **a sensitive operation**
- **a heap overflow followed**
CVE-2008-5238(Xine)

......

if (version == 4) {
    const uint16_t sps = _X_BE_16 (this->header+44) ? : 1;
    this->w = _X_BE_16 (this->header+42);
    this->h = _X_BE_16 (this->header+40);
    this->cfs = _X_BE_32 (this->header+24);
    this->frame_len = this->w * this->h;
    this->frame_size = this->frame_len * sps;
    this->frame_buffer = calloc(this->frame_size, 1);
......

an untrusted source

an integer overflow

a sensitive operation
CVE-2008-1722(CUPS)

```
png_get_IHDR(pp, info, &width, &height, &bit_depth, 
    &interlace_type, &compression_type, &filter);
{
    ...
    if (width == 0 || width > CUPS_IMAGE_MAX_WIDTH ||
        height == 0 || height > CUPS_IMAGE_MAX_HEIGHT)
    {//error
        return (1);
    }
    img->xsize = width;
    img->ysize = height;
    ...
    if (color_type == PNG_COLOR_TYPE_GRAY || color_type == 
        PNG_COLOR_TYPE_GRAY_ALPHA)
        in = malloc(img->xsize * img->ysize);
    else
        in = malloc(img->xsize * img->ysize * 3);
    ...
}
```
if( ChunkFind( p_demux, "fmt ", &i_size ) )
{
    msg_Err( p_demux, "cannot find 'fmt ' chunk" );
    goto error;
}
if( i_size < sizeof( WAVEFORMATEX ) - 2 )
{
    msg_Err( p_demux, "invalid 'fmt ' chunk" );
    goto error;
}
stream_Read( p_demux->s, NULL, 8 ); /* Cannot fail */
/* load waveformatex */
p_wf_ext = malloc( __EVEN( i_size ) + 2 );

......
What’s the essential feature of integer overflow vulnerabilities?

- an untrusted source
- an incomplete check
- an integer overflow
- a sensitive operation

```c
unsigned int x = read_int();
if ( x > 0x7fffffff )
    abort();
unsigned int s = x*sizeof(int);
char* p = malloc(s);
read_int_into_buf(p, x);
```
What’s the essential feature of integer overflow vulnerabilities?

Typical view

The essential feature is the actual overflow itself

```c
unsigned int x = read_int();
if ( x > 0x7fffffff )
    abort();
unsigned int s = x*sizeof(int);
char* p = malloc(s);
read_int_into_buf(p, x);
```
Integer Overflow != Integer Overflow Vulnerability

- Case 1: The overflowed value is NOT used in any sensitive operation
  - e.g. TCP sequence number rolls back per 4GB

- Case 2: The overflowed value is NOT tainted
  - Most untainted integer overflows are on purpose, i.e., benign overflows, e.g. computing random seeds

- So Integer overflow itself is not the essential part of the vulnerability
What’s the essential feature of integer overflow vulnerabilities?

The essential feature is those sensitive operations which use some tainted overflowed data.

```c
unsigned int x = read_int();
if ( x > 0x7fffffff )
    abort();
unsigned int s = x*sizeof(int);
char* p = malloc(s);
read_int_into_buf(p, x);
```
Outline

- Motivation
- Case Study
- **Modeling**
- Challenges & Approaches
- Implementation & Evaluation
- Related work
- Conclusion
Modeling Integer Overflow Vulnerability

- An instance of taint-based problem

Diagram showing the flow from a source to a sink, with intermediate nodes and arrows indicating the path.
Modeling Integer Overflow Vulnerability

- An instance of taint-based problem

Source

an untrusted source: fread(), fscanf(), ...

Sink
Modeling Integer Overflow Vulnerability

- An instance of taint-based problem

**Source**
- an untrusted source: fread(), fscanf(), ...

**Sink**
- a sink using tainted overflowed data: *alloc(), array index, pointer offset, some predicates...
Modeling Integer Overflow Vulnerability

- An instance of taint-based problem

Source

- an untrusted source: fread(), fscanf(), ...

Sink

- A feasible path connecting the source and the sink

- a sink using tainted overflowed data: *alloc(), array index, pointer offset, some predicates
Outline

- Motivation
- Case Study
- Modeling
- **Challenges & Approaches**
- Implementation & Evaluation
- Related Work
- Conclusion
Natural Approach

- Based on general static taint analysis
- Given a binary program
Natural Approach

- Decompile the binary program
  - Generate the intermediate representations, call graphs, CFGs, ...

Diagram:
- Main function flowchart with various intermediate steps and connections.
Natural Approach

- Decompile
- Traverse all paths from main() using symbolic execution
Natural Approach

- Decompile
- Traverse, prune infeasible paths, whose path constraints cannot be satisfied, during traversing
Natural Approach

- Decompile
- Traverse, Prune, Check possible integer overflows during traversing
Natural Approach

- Decompile
- Traverse, Prune, Check, **Tag sources and sinks during traversing**
Natural Approach

- Decompile
- Traverse, Prune, Check, Tag
- Output suspicious paths in which tainted overflowed data used in sinks
Does this natural approach work efficiently?

- **Major Challenges**
  - 1. Lack of type information
  - 2. Path explosion
  - ......
Challenge 1. Lack of type information

- During traversing, how can we determine there is an overflow or not?

```assembly
mov  eax, 0xffffffff ; eax = 0xffffffff or -1
add  eax, 2         ; eax = eax + 2
```

Overflow or not?
How to solve this?

- Lazy check: only check integer overflows used in sinks

- Decompile
- Traverse, Prune, Check, Tag
- Output
Lazy check

- Only check integer overflows used in sinks

```
mov  eax, 0xffffffff
add  eax, 2
sub  eax, 4
jb   label1
```

- `eax` is 0xffffffff, not -1
- `eax` is 1 now, overflowed
- `unsinged cmp => eax` is unsigned
Benefit of Lazy check

- Useful **type information** hints
  - Signed/unsigned comparisons
    - signed: JG, JGE, JNL, JNGE, JLE, JNG, JE, JNE
    - unsigned: JA, JAE, JNB, JB, JNAE, JBE, JNA, JE, JNE
  - void *calloc(size_t nmemb, size_t size);
  - void *malloc(size_t size);
  - ...

- Much less checks, much more efficiency
Challenge 2. Path explosion

- We need path-sensitive analysis, but the number of paths through software is very large.
Exponential Traversing Time

- Only pruning during execution is not enough
Solution: Pre-pruning before traversing

- Only consider paths between sources and possible sinks
Pre-pruning

- Tag sources and possible sinks before traversing
Pre-pruning

- Tag
- Cut off those paths irrelevant to sources and sinks using some inter-function slicing algorithms
Put it all together

- Decompile
- Tag, Pre-prune
- Traverse, Prune, Lazy Check
- Output suspicious paths
Put it all together

- Given a binary program
Put it all together

- **Decompile the program**
  - Generate the IR, call graph, CFGs, and so on
Put it all together

- Decompile
- Tag possible sources and sinks
Put it all together

- Decompile
- Tag, **Pre-prune:** Cut off those paths irrelevant to sources and sinks
Put it all together

- Decompile
- Tag, Pre-prune
- Traverse paths left using symbolic execution
Put it all together

- Decompile
- Tag, Pre-prune
- Traverse, **Prune infeasible paths during traversing**
Put it all together

- Decompile
- Tag, Pre-prune
- Traverse, Prune, **Lazy Check**: check integer overflows used in sinks
Put it all together

- Decompile
- Tag, Pre-prune
- Traverse, Prune, Lazy Check
- Output suspicious paths
Outline

- Motivation
- Case Study
- Modeling
- Challenges & Approaches
- Implementation & Evaluation
- Related Work
- Conclusion
IntScope Architecture

- IntScope
  - Decompiler
    - BESTAR [SAS2007]
  - Cut off irrelevant paths
    - Pre-pruning Engine
  - Symbolic Execution
    - Environment
    - Engine
  - Pruning during traversing
    - Constraint Solver
  - Lazy Checker

- 3rd Party Modules
  - Disassembler: IDA Pro
  - CAS: GiNaC
  - Constraint Solver: STP
How to use IntScope

1. Binary
2. IntScope
3. Suspicious Paths
4. Confirm
5. Report
Evaluation

- Two Windows DLLs
  - GDI32.dll
  - comctl32.dll

- Several widely used applications
  - QEMU, Xen
  - Media players
    - Mplayer
    - Xine
    - VLC
    - FAAD2
    - MPD
  - Others
    - Cximage, Hamsterdb, Goom
Effectiveness

- Detected known integer overflow bugs in Windows DLLs

- Detected 20+ zero-day integer overflow vulnerabilities
  - Confirmed by developers or concrete test cases
  - Some projects have released patches

- We have reported vulnerabilities in QEMU and FAAD2 to French Security Incident Response Team (FrSIRT)
  - CVE-2008-4201
  - FrSIRT/ADV-2008-2919
  - ......
Effectiveness

- Among 26 integer overflow vulnerability points, 21 of them have been confirmed
Efficiency

- AMD Opteron Server (2.6 GHz) with 8GB memory

<table>
<thead>
<tr>
<th>Name</th>
<th>Executable</th>
<th>File Size</th>
<th>Binary-to-IR time (seconds)</th>
<th>IR Size</th>
<th>Traversing Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDI32.dll</td>
<td>GDI32.dll</td>
<td>271KB</td>
<td>614</td>
<td>7.61 MB</td>
<td>574</td>
</tr>
<tr>
<td>comctl32.dll</td>
<td>comctl32.dll</td>
<td>597 KB</td>
<td>1131</td>
<td>13.7 MB</td>
<td>0.1</td>
</tr>
<tr>
<td>QEMU</td>
<td>Qemu-img</td>
<td>341 KB</td>
<td>124</td>
<td>12.8 MB</td>
<td>358</td>
</tr>
<tr>
<td>Xine</td>
<td>cdda_server</td>
<td>14.5 KB</td>
<td>4</td>
<td>116 KB</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>xine</td>
<td>966 KB</td>
<td>590</td>
<td>12.9 MB</td>
<td>327</td>
</tr>
<tr>
<td>Mplayer</td>
<td>avisubdump</td>
<td>14.2 KB</td>
<td>1</td>
<td>36.8 KB</td>
<td>0.3</td>
</tr>
<tr>
<td>MPD</td>
<td>mpd</td>
<td>243 KB</td>
<td>131</td>
<td>2.74 MB</td>
<td>667</td>
</tr>
<tr>
<td>GOOM</td>
<td>libgloom2.so</td>
<td>439KB</td>
<td>94</td>
<td>1.42 MB</td>
<td>445</td>
</tr>
<tr>
<td>faad2</td>
<td>faad</td>
<td>57.6 KB</td>
<td>29</td>
<td>693 KB</td>
<td>113</td>
</tr>
<tr>
<td>Hamsterdb</td>
<td>libhamsterdb.so</td>
<td>260 KB</td>
<td>164</td>
<td>3.46 MB</td>
<td>426</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>320.3KB</td>
<td>288.2</td>
<td>5.46MB</td>
<td>293.6</td>
</tr>
</tbody>
</table>

- Average time: about 5 min
- Longest time: < 12 min
Outline

- Motivation
- Case Study
- Modeling
- Intuition & Challenge
- Implementation & Evaluation
- Related Work
- Conclusion
Related Work

◆ w/ source code
  ➢ Run-time Protection
    ✓ Safe integer libraries
    ✓ RICH [NDSS’07]
    ✓ GCC
  ➢ Dynamic and/or Static analysis
    ✓ Range checker [S&P’02]
    ✓ CQual[PLDI02], EXE[CCS06], KLEE[OSDI08], DART[PLDI05], CUTE[FSE05]

◆ w/o source code
  ➢ Fuzzing
    ✓ SAGE [NDSS’08]
    ✓ Catchconv [Molnar and Wagner, Berkeley]
  ➢ Static analysis of integer overflows using sym exec <= IntScope
Outline

- Motivation
- Case Study
- Modeling
- Intuition & Challenge
- Implementation & Evaluation
- Related Work
- Conclusion
Conclusion

- **IntScope**
  - Modeling Integer Overflow Vulnerability as a taint-based problem
  - Lazy Check: only check integer overflows lazily at sinks
  - Pre-prune: prune paths irrelative to sources and possible sinks before traversing
  - Detect 20+ Zero-day integer overflow vulnerabilities
Questions?

weitao@icst.pku.edu.cn
Backup slides
Modeling Integer Overflow Vulnerability

- An instance of taint-based problem

A feasible path connecting the source and the sink

Source

an untrusted source: fread(), fscanf(), ...

Sink

a sink using tainted overflowed data: *alloc(), array index, pointer offset, some predicates
Suspicious Paths

- IntScope is a static analysis tool, so it may generate false positives.
  - Missing of the constraints between inputs.
  - Lack of global information
  - Imprecise symbolic execution

- For each vulnerability, if we cannot construct a concrete test case to trigger it, we leave it as a suspicious one.
False positives

- IntScope is a static analysis tool, so it may generate false positives.
  - Missing of the constraints between inputs.
  - Lack of global information
  - Imprecise symbolic execution
- It’s hard to prove an alert is a real vulnerability
  - we need to construct a concrete test case to trigger the vulnerability.
- If we can not construct such test cases, we take these alerts as suspicious ones.