Detecting Integer Overflow Vulnerabilities in Binaries

Tielei Wang
Institute of Computer Science and Technology, Peking University
About Me

- 4th Ph.D. student in Peking Univ.
- Interested in binary program analysis, reverse engineering and fuzzing.
- Detected many vulnerabilities in some popular applications:
  - CVE-2009-2989, CVE-2009-2995
  - CVE-2009-1882, CVE-2009-2660
  - CVE-2009-2347, CVE-2009-2688
  - ....
Introduction

Nearly All Binary Searches and Mergesorts are Broken.

--From Bloch’s blog (2006)

- Chief Java Architect at Google
- Top 40 Software People in the World, 2004
- Lead the design and implementation of numerous Java platform features
- Effective Java, Java Puzzlers author

--From Wikipedia
An error in schoolbook

1: public static int binarySearch(int[] a, int key)
   {
2:     int low = 0;
3:     int high = a.length - 1;
4:     
5:     while (low <= high) {
6:         int mid = (low + high) / 2;
7:         int midVal = a[mid];
8: 
9:         if (midVal < key)
10:             low = mid + 1;
11:         else if (midVal > key)
12:             high = mid - 1;
13:         else
14:             return mid; // key found
15:     }
16:     return -(low + 1); // key not found.
17:   }

high = 0x7fffffff
low = 0x40000000
mid = 0xdfffffff
    = -536870913

Array access out of bound
The number of integer overflow (IO) vulns is growing rapidly.
IO vulns are heavily represented in OS vendor advisories, rising to number 2 in 2006
Underestimated Threat (3/3)

- IO vulns are the springboard to buffer overflows
  - >50% IO vulns can further cause buffer overflows
    - OS Kernel
      - CVE-2008-4036 (Windows XP, Server 2003, Vista)
      - CVE-2008-3276 (Linux), CVE-2008-4220 (Mac OS), CVE-2008-1391 (NetBSD)
    - Applications
      - CVE-2008-0726 (Adobe Reader), CVE-2008-4061 (Firefox), CVE-2008-2947 (IE7), CVE-2008-0120 (PowerPoint)
Outline

- Introduction
- Case study
- System Design & Implementation
- Demo
- Conclusion
Typical View: All Integer overflows are evil

- Can we capture all integer overflows at runtime?
- GCC, RICH, SafeInt...

```c
__addvsi3 (SItype a, SItype b){
    const SItype w = a + b;
    if (b >= 0 ? w < a : w > a)
        abort ();
    return w;
}
```
Not All Integer Overflows are Vulns!

- **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
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);
}

......
CVE-2008-1722 (CUPS)

```
png_get_IHDR(pp, info, &width, &height, &bit_depth, &color_type,
    &interlace_type, &compression_type, &filter_type);

......
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);
```
CVE-2008-2430 (VLC)

......
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 ) +
......

an untrusted source
an incomplete check
an integer overflow
a sensitive operation
Basic Pattern of IO vulns

unsigned int x = read_int();
if (x > 0x7fffffff || x==0)
    abort();
unsigned int s = x*sizeof(int);
int* p = malloc(s);
memset(p, 0, x);

an untrusted source
an incomplete check
an integer overflow
a heap overflow followed
a sensitive operation
An instance of taint-based problem

an untrusted source: fread(), fscanf()

A feasible path connecting the source and the sink

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

- Introduction
- Case study
- System Design & Implementation
- Demo
- Conclusion
What We Can Use (1/3)

- Disassemble && Intermediate Representation
  - IDA Pro, objdump
  - CodeSurfer/x86, Halvar Flake [bh’04]
What We Can Use (2/3)

- Static taint analysis
  - Taint the untrusted data, and infer the possible propagation of such untrusted data
  - Similar to type systems (e.g., CQual)

\[
\Gamma \vdash Q_1 \subseteq Q_2 \quad \Gamma \vdash Q_1 \text{ int } \leq Q_2 \text{ int } \\
\vdash Q_1 \subseteq Q_2 \quad \vdash \rho_2 \leq \rho_1 \quad \vdash \rho_1' \leq \rho_2' \\
\vdash Q_1 (\rho_1 \rightarrow \rho_1') \leq Q_2 (\rho_2 \rightarrow \rho_2')
\]
What We Can Use (3/3)

- **Symbolic Execution**
  - Statically “run” the program with symbolic values instead of concrete ones
  
  \[
  \begin{align*}
  x &= x + y; \\
  y &= x - y; \\
  x &= x - y;
  \end{align*}
  \]

- **Solvers**
  - MiniSAT, Chaff, CVC3, STP, Z3, Yices

\[
\forall x \exists y \exists z ((x \lor y \lor z) \land (\neg x \lor \neg y \lor \neg z)).
\]

\[
\begin{align*}
3x + 2y - z &\geq 4 \\
x - y &\leq c
\end{align*}
\]
Natural Approach

- Disassemble the binary to IR
- From the function main, symbolically execute each path:
  - Collect path constraints, and check the feasibility of the path
  - Track the propagation of untrusted data
  - Check whether untrusted data causes integer overflows

Feasible or not???
Challenge 1: Lack of type information

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

\[
\begin{align*}
\text{mov eax,0xffffffff} \\
\text{add eax,1} \\
\text{eax = 0xffffffff} \\
\text{eax = eax+1} \\
\end{align*}
\]

Overflow or not?
Challenge 1: Lack of type information

- There is no type information in binaries

| eax = 0xffffffff
| eax = eax + 1 |

If eax is “signed int”

| eax = -1
| eax = eax + 1 |

If eax is “unsigned int”

| eax = 4294967295
| eax = eax + 1 |
Challenge 2: Benign Integer Overflows

- Programmers (even compilers) may make use of harmless integer overflows.

```c
int x = read_from_file();
if(x>= -2 && x<= 0x7fffffff)
    mov eax, x; // eax = x
    add eax, 2; // eax = eax +2
    js target
```

GCC -O2
Strategy: Lazy Check

- Lazy check: Only check tainted values used in sinks, instead of each arithmetic operation

```assembly
mov eax, x
shl eax, 2
add eax, 4
push eax
call malloc  // malloc(x*4+4)
```

Do not check whether the add instruction

Only check the symbolic expression “x*4+4” when meeting “malloc”
Challenge 3: Path explosion

- We need check each path, but the number of paths through software is huge.
Strategy: Extract security-sensitive component

- First, based on call graph (CG), identify the source and sink functions
  - Source: introduce untrusted data
  - Sink: malloc, alloc, etc
Strategy: Extract security-sensitive component

- Find the common ancestors of a taint source function node and a sink function node
Strategy: Compute a chop from source to sink

- Only consider paths between sources and possible sinks
- Ignore unrelated paths
- Significantly reduce the number of paths
Put it together

- **Workflow:**
  - Disassemble the binary to IR, construct the control flow graphs and call graph
  - Extract the security-sensitive component
    - Only select the path from a source to a sink
  - **symbolically execute each path in the component:**
    - Collect path constraints, and check the feasibility of the path
    - Track the propagation of untrusted data
    - Only check whether untrusted data causes integer overflows at sink points
System Implementation

- **3rd Party Modules**
  - Disassembler: IDA Pro
  - CAS: GiNaC
  - Constraint Solver: STP
Outline

- Introduction
- Case study
- System Design & Implementation
- Demo
- Conclusion
Experiment Results

- Several widely used applications
  - QEMU, Xen
  - Media players
    - Mplayer
    - Xine
    - VLC
    - FAAD2
    - MPD
  - Others
    - Cximage, Hamsterdb, Goom
Winamp Integer Overflow Vulnerability

- The problem has been published (SA35126, secunia)
- Affected: Winamp <= v5.552

**Winamp MP4 Processing Integer Overflow Vulnerability**

SA35126 // 2 credits // Exploit and/or PoC code included

An integer overflow error in Winamp can be exploited to compromise a user's system via a specially crafted MP4 file.
Winamp Integer Overflow Vulnerability

- The problem has been published (SA35126, secunia)
- Affected: Winamp <= v5.552

Winamp MP4 Processing Integer Overflow Vulnerability
SA35126 // 2 credits // Exploit and/or PoC code included
An integer overflow error in Winamp can be exploited to compromise a user's system via a specially crafted MP4 file.
Winamp Integer Overflow Vulnerability

Winamp遇到问题需要关闭。我们对此引起的不便表示抱歉。

如果您正处于进程当中，信息有可能丢失。

请将此问题报告给 Microsoft。

我们已经创建了一个错误报告，您可以将它发送给我们。
此报告为保密和匿名的。

要查看这个错误报告包含的数据，请单击此处。

调试 (F)  |  发送错误报告 (F)

m: winamp.exe - 应用程序错误

“0x0559e741”指令引用的“0x00000025”内存。该内存未分配。

要终止程序，请单击“确定”。

要调试程序，请单击“取消”。

About Winamp

Winamp  |  Credits  |  Keyboard shortcuts  |  Version history

Copyright © 1997-2009 Nullsoft, Inc.
v5.552 (x86) - Apr 10 2009
Visit www.winamp.com for updates.
Vulnerability Analysis (1/3)

- An MP4 file consists of numerous “atoms”. An atom is in form of:

- The code in libmp4v2.dll looks like:

```c
uint32_t numEntries = Get_number_of_Entries();
...
buffer = realloc(buffer, numEntries * 4);
...
for (uint32_t i = 0; i < numEntries; i++) {
    ReadEntry(buffer, i, inFile);
}
```
Vulnerability Analysis (2/3)

- E.g., modify “ctts” atom’s numEntries to 0x40000001

- An error occurs: call dword ptr [eax+24h]

```
06E7E734  397D F8    CMP DWORD PTR SS:[EBP-8],EDI
06E7E737  76 11    JBE SHORT 06E7E74A
06E7E739  8B06    MOV EAX,DWORD PTR DS:[ESI]
06E7E73B  57    PUSH EDI
06E7E73C  FF75 08    PUSH DWORD PTR SS:[EBP+8]
06E7E73E  8BCE    MOV ECX,ESI
06E7E740  FF50 24    CALL DWORD PTR DS:[EAX+24]
```

 Registers (MMX) 

```
EAX  000005C9  
ECX  0233D3C8  
EDX  00000000  
EBX  0233D3DC  
ESP  06B6FBE4  
EBP  06E6FC00  
esi  0233D3C8  
edi  0000003F  
EIP  06E7E741  
```

```
libmp4v2.06E7E741
```
Vulnerability Analysis (3/3)

- In fact, 0x000005C9 is from the input file:

```
00000460h: C9 00 00 06 02 00 00 05 F4 00 00 05 C9 00 00 05
00000470h: ED 00 00 05 E8 00 00 05 84 00 00 05 F2 00 00 05
```

- Replace 0x000005C9 with 0xffffffff

<table>
<thead>
<tr>
<th>Offset</th>
<th>Opcode</th>
<th>Machine Instruction</th>
<th>Registers (MMX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>04B7E739</td>
<td>8B06</td>
<td>MOV EAX, DWORD PTR DS:[ESI]</td>
<td>EAX FFFFFFFF</td>
</tr>
<tr>
<td>04B7E73B</td>
<td>57</td>
<td>PUSH EDI</td>
<td>ECX 021BD998</td>
</tr>
<tr>
<td>04B7E73C</td>
<td>FF75 08</td>
<td>PUSH DWORD PTR SS:[EBP+8]</td>
<td>EDX 00000000</td>
</tr>
<tr>
<td>04B7E73E</td>
<td>8BCE</td>
<td>MOV ECX, ESI</td>
<td>EBX 021BD9AC</td>
</tr>
<tr>
<td>04B7E741</td>
<td>FF50 24</td>
<td>CALL DWORD PTR DS:[EAX+24]</td>
<td>ESP 04B6FBE4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>EBP 04B6FC00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ESI 021BD998</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>EDI 0000003F</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>EIP 04B7E741 libmp4v2</td>
</tr>
</tbody>
</table>
Outline

- Introduction
- Case study
- System Design & Implementation
- Demo

Conclusion
Conclusion

- Integer overflow vulnerability is an underestimated threat.

- IntScope
  - Modeling Integer Overflow Vulnerability as a taint-based problem
  - Symbolic execution + taint analysis + SMT
Thanks
Tielei.wang@gmail.com