Finding Buffer Overflow Inducing Loops in Binary Executables

Sanjay Rawat, Laurent Mounier

VERIMAG
University of Grenoble, France

SERE 2012
Outline

1. Motivation: software vulnerability analysis
2. Identifying Buffer-overflow Inducing Loops
3. Implementation and experimental results
4. Conclusion and perspectives
“A software flaw that may become a security threat . . . ”

**Examples:**
- memory safety violation (buffer overflow, dangling pointer, etc.)
- arithmetic overflow
- unchecked input data
- race condition, etc.

**Possible consequences:**
- denial of service (program crash)
- code injection
- privilege escalation, etc.
Software vulnerability detection

**Hand based analysis:**
(with some limited tool assistance, e.g. disassemblers, debuggers)
- conducted by security experts
- based on known security holes and/or security patches

**Static analysis:** abstract interpretation, symbolic execution
- ex: memory safety violations (buf. overflows)
- operate mostly at the source level
- over-approximations $\Rightarrow$ large number of *false positives* . . .

**Runtime analysis:** security testing, fuzzing
- execute the program with specific inputs
  (random mutations, bad string formats, etc.)
- may cover only a small part of the application code . . .
Software vulnerability detection

Hand based analysis:
(with some limited tool assistance, e.g. disassemblers, debuggers)
- conducted by security experts
- based on known security holes and/or security patches

Static analysis: abstract interpretation, symbolic execution
- ex: memory safety violations (buf. overflows)
- operate mostly at the source level
- over-approximations $\mapsto$ large number of false positives . . .

Runtime analysis: security testing, fuzzing
- execute the program with specific inputs
  (random mutations, bad string formats, etc.)
- may cover only a small part of the application code . . .
Software vulnerability detection

Hand based analysis:
(with some limited tool assistance, e.g. disassemblers, debuggers)
- conducted by security experts
- based on known security holes and/or security patches

Static analysis: abstract interpretation, symbolic execution
- ex: memory safety violations (buf. overflows)
- operate mostly at the source level
- over-approximations \(\leadsto\) large number of false positives ...

Runtime analysis: security testing, fuzzing
- execute the program with specific inputs
  (random mutations, bad string formats, etc.)
- may cover only a small part of the application code ...
A current trend: smart fuzzing

- A combination between static and runtime analysis
- Several approaches, e.g.:

Concolic (aka dynamic/symbolic) execution:
- symbolic execution of concrete paths
- coverage-oriented (explore uncovered execution paths)

Statically directed runtime analysis
- use static analysis techniques to identify “vulnerable” execution paths
- use test based techniques to explore them at runtime
How to find a needle in a haystack?

A common starting point to all analysis techniques:
→ identify a (small) subset of “potentially vulnerable” functions . . .

⇒ vulnerability patterns

Existing solutions:
- unsafe library functions (strcpy, memcpy, printf, etc)
- previously known vulnerable functions
- (smart) code coverage techniques (e.g. Sage)
→ Define and identify **semantic** vulnerability patterns ...

- **should be easy to compute**
  only lightweight analysis are affordable at the whole pgm level
- **should be discriminating enough ...**
  to give a precise pgm slice *and then* to conduct deeper analysis
- **... but not too much !**
  to avoid false negatives

We focus here on **Buffer Overflow** vulnerabilities

(-ranked 3 in last “Top 25 Most Dangerous Software Errors¹”)

¹http://cwe.mitre.org/top25/
Concrete examples

Three recent stack-based buffer overflow vulnerabilities:

- FreeType font library used by Mozilla products (CVE-2012-1144 and CVE-2012-1141)
- Adobe Flash Player: (CVE-2012-2035, under review)

Caused by dedicated buffer copy functions (≠ strcpy, etc.) ...

→ There may exist many similar functions (sleeping bombs!)

⇒ Requires a specific behavioral vulnerability pattern:

Buffer-Overflow Inducing Loops (BOILs)
Outline

1 Motivation: software vulnerability analysis

2 Identifying Buffer-overflow Inducing Loops

3 Implementation and experimental results

4 Conclusion and perspectives
Motivating example

```
char * bufCopy(char *destination, char *source)
{
    char *p = destination;
    while (*source != '\0')
    {
        *p++ = *source ++;
    }
    *p = '\0';
    return destination;
}
```

Listing 1: Example of a function that is similar to strcpy

- There is a **loop**, iterating over source and destination.
- Memory which is being read/written is **changing within the loop**.
BOILs and BOPs

**BOIL**

A loop is a BOIL if

1. there is a Memory Write within the loop
2. the written address is changing within the loop (→ write into a destination buffer)
3. the written value depends on a function argument (→ write a possibly tainted value)

Additionally

- the number of iterations should not be fixed,
- and not depending on the destination buffer . . .

**BOP**

A function is Buffer-Overflow Prone (BOP) if it contains a BOIL.
Identifying BOILs?

A **lightweight** decision procedure operating on **binary code**:

1. Loop detection
   ↦ classical algorithms based on dominator tree computations
2. A Memory Write inside the loop
   ↦ “store” instruction recognition
3. Written address is changing within the loop
   ↦ ∃ a **self def-use dependency chain**
4. Written value depends on function argument
   ↦ ∃ a **def-use dependency chain**
Example 1: function strcpy

<table>
<thead>
<tr>
<th></th>
<th>1 004075F0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>004075F0</td>
<td>push edi</td>
</tr>
<tr>
<td>3</td>
<td>004075F1</td>
<td>mov edi, ss:[esp+Dest]</td>
</tr>
<tr>
<td>4</td>
<td>004075F5</td>
<td>jmp loc_407661</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>00407661</td>
<td>mov ecx, ss:[esp+Source]</td>
</tr>
<tr>
<td>7</td>
<td>00407665</td>
<td>test ecx, 3</td>
</tr>
<tr>
<td>8</td>
<td>0040766B</td>
<td>jz loc_407686</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0040766D</td>
<td>mov byte dl, byte ds:[ecx]</td>
</tr>
<tr>
<td>11</td>
<td>0040766F</td>
<td>inc ecx</td>
</tr>
<tr>
<td>12</td>
<td>00407670</td>
<td>test byte dl, byte dl</td>
</tr>
<tr>
<td>13</td>
<td>00407672</td>
<td>jz loc_4076D8</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>00407674</td>
<td>mov byte ds:[edi], byte dl</td>
</tr>
<tr>
<td>16</td>
<td>00407676</td>
<td>inc edi</td>
</tr>
<tr>
<td>17</td>
<td>00407677</td>
<td>test ecx, 3</td>
</tr>
<tr>
<td>18</td>
<td>0040767D</td>
<td>jnz loc_40766D -&gt; loop back to</td>
</tr>
</tbody>
</table>

Assembly code of strcpy

→ Within the loop, memory is accessed via registers
→ Dependency on argument & local variable not visible inside the loop

Memory is written once:

change of memory address = incrementing registers
Example 2: function bufCopy

Assembly code of bufCopy

→ Dependency on argument/local variable visible inside the loop

Memory is written 3 times:

- to change the stored address of the next character
- to store the character itself
Strided memory access pattern within a loop

**Pattern A (strcpy function, rather straightforward)**

1:  $\text{regd} \leftarrow \text{MEM}[\text{base+dest}]$ \hspace{1cm} \text{DEST adr.}
2:  $\text{regs} \leftarrow \text{MEM}[\text{base+src}]$ \hspace{1cm} \text{SRC adr.}
(loop)
3:  $\text{MEM}[\text{regd}] \leftarrow \text{MEM}[\text{regs}]$ \hspace{1cm} \text{copy SRC to DEST}
4:  $\text{regd} \leftarrow \text{regd}+\text{stride}$
5:  $\text{regs} \leftarrow \text{regs}+\text{stride}$
(endloop)
pattern B (bufCopy function, less straightforward):

\[
\begin{align*}
MEM[\text{base}+p] & \leftarrow MEM[\text{base}+\text{dest}] & \text{DEST adr.} \\
\text{loop} & & \\
1: & \text{reg1} \leftarrow MEM[\text{base}+p] & \\
2: & \text{regd} \leftarrow \text{reg1} & \\
3: & \text{reg1} \leftarrow \text{reg1}+\text{stride} & \\
4: & \text{MEM[base}+p] \leftarrow \text{reg1} & \text{next DEST adr.} \\
5: & \text{reg2} \leftarrow \text{MEM[base}+\text{src}] & \text{SRC adr} \\
6: & \text{regs} \leftarrow \text{reg2} & \\
7: & \text{reg2} \leftarrow \text{reg2}+\text{stride} & \\
8: & \text{MEM[base}+\text{src}] \leftarrow \text{reg2} & \text{next SRC adr.} \\
9: & \text{MEM[regd]} \leftarrow \text{MEM[regs]} & \text{copy SRC to DEST} \\
\text{endloop} & & \\
\end{align*}
\]
Self def-use dependency chains

**def-use dependency chain**

Sequence of the type: \( v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \ldots \rightarrow v_k \)

\( v_i \) = register, variable or argument

\( v_i \) is defined in terms of \( v_{i+1} \) (\( v_i := \ldots v_{i+1} \ldots \))

**self def-use dependency:**

\[ \Rightarrow \] a simple data-flow analysis (reaching definitions) \[ \ldots \]
Example on pattern B

code for pattern B:

1:  reg1 ← MEM[base+p]
2:  regd ← reg1
3:  reg1 ← reg1+stride
4:  MEM[base+p] ← reg1
...
9:  MEM[regd] ← MEM[regs]
Check if iteration condition depends on the destination buffer?

**A simple heuristic:**

1. find the loop controlling variables  
   (look for comparison inst. before cond. jumps)
2. compute its def-use dependency chain  
   \(\rightarrow\) should reach a variable or argument
3. check if this argument is the dest buffers  
   \(\rightarrow\) if yes, assume it is not a vulnerable loop

**Remark:**

May be **too strict**, possible **false negatives** ...
Outline

1 Motivation: software vulnerability analysis

2 Identifying Buffer-overflow Inducing Loops

3 Implementation and experimental results

4 Conclusion and perspectives
**Tool Chain**

- **binary code** → **IDAPro** → **x86** → **BinNavi**
  - **REIL**
    - **BOIL**
      - {suspicious loops}

**BinNavi and REIL intermediate language**
- only 17 instructions, very simple addressing mode;
- powerful jython API;
- CFG construction and analysis;
- MonoREIL: execution engine for data-flow analysis
Experimentations

Objectives
Evaluate the relevance of the BOIL criterion:
- percentage of “vulnerable functions” detected?
- do they contain real vulnerabilities?
- scalability of the analysis?

Methodology
→ include known vulnerable applications/libraries in the benchmark
## Experimental Results

<table>
<thead>
<tr>
<th>Module</th>
<th># func</th>
<th># loops</th>
<th>BOILs</th>
<th>BOP func</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeFloat FTP</td>
<td>309</td>
<td>146</td>
<td>21</td>
<td>12</td>
</tr>
<tr>
<td>CoolPlayer</td>
<td>995</td>
<td>1036</td>
<td>156</td>
<td>56</td>
</tr>
<tr>
<td>GDI32.dll</td>
<td>1775</td>
<td>655</td>
<td>70</td>
<td>51</td>
</tr>
<tr>
<td>freeType</td>
<td>1910</td>
<td>2568</td>
<td>409</td>
<td>249</td>
</tr>
<tr>
<td>msvcr80.dll</td>
<td>2321</td>
<td>1154</td>
<td>188</td>
<td>113</td>
</tr>
</tbody>
</table>

Execution times = a few minutes . . .

### Remarks

- **freeType**: recognized `t42_parse_sfnts` function (array index error) CVE-2010-2806, CVE-2012-1144 and CVE-2012-1141 (Mozilla)
- **GDI32.dll**: recognized strcpy-like functions (StringCchCopy)
- **FreeFloat FTP/CoolPlayer**: recognized strcpy, wcscpy functions responsible for BoF. OSVDB: 69621.
1 Motivation: software vulnerability analysis

2 Identifying Buffer-overflow Inducing Loops

3 Implementation and experimental results

4 Conclusion and perspectives
Vulnerability detection methods driven by **vulnerability patterns**

These patterns need to be:
- easy to compute (scalability)
- discriminating enough (reduced slice)

We proposed a BOIL criterion for BoF vulnerabilities

Experimental results are good:
- flags $\sim 10\%$ of loops as “suspicious”
- allows to retrieve existing vulnerabilities
Future Work

Integration within a complete vulnerability analysis framework:

- Binary code → BOIL → Code slice → Static taint analysis
  - Path-oriented input generation → Vulnerable paths + input offsets
  - Fuzzing...

Similar approaches for other kinds of vulnerabilities:

- E.g., use-after-free