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

Software Vulnerability

"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
- priviledge 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 $\rightsquigarrow$ 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 $\rightsquigarrow$ 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 $\rightsquigarrow$ 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

Vulnerability Patterns

How to find a needle in a haystack ?

A common starting point to all analysis techniques:
$\hookrightarrow$ identify a (small) subset of "potentially vulnerable" functions ...

$\Rightarrow$ **vulnerability patterns**

Existing solutions:

- unsafe library functions (`strcpy`, `memcpy`, `printf`, etc)
- previously known vulnerable functions
- (smart) code coverage techniques (e.g. Sage)

Work objective

$\rightarrow$ 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 Errors[1]")

[1] http://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 ($\neq$ `strcpy`, etc.) ...

$\hookrightarrow$ There may exist many similar functions (*sleeping bombs!*)

$\Rightarrow$ 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

```c
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
   ($\rightarrow$ write into a destination **buffer**)
3. the written value depends on a function argument
   ($\rightarrow$ 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
   $\hookrightarrow$ classical algorithms based on dominator tree computations
2. A Memory Write inside the loop
   $\hookrightarrow$ "store" instruction recognition
3. Written address is changing within the loop
   $\hookrightarrow$ $\exists$ a **self def-use dependency chain**
4. Written value depends on function argument
   $\hookrightarrow$ $\exists$ a **def-use dependency chain**

Example 1: function `strcpy`

```
004075F0    _strcpy
004075F0    push    edi
004075F1    mov     edi, ss:[esp+Dest]
004075F5    jmp     loc_407661

00407661    mov     ecx, ss:[esp+Source]
00407665    test    ecx, 3
0040766B    jz      loc_407686

0040766D    mov     byte dl, byte ds:[ecx]   <---
0040766F    inc     ecx
00407670    test    byte dl, byte dl
00407672    jz      loc_4076D8

00407674    mov     byte ds:[edi], byte dl
00407676    inc     edi
00407677    test    ecx, 3
0040767D    jnz     loc_40766D -> loop back to -
```

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`

```
00401000    bufCopy

00401010    mov      eax, ss:[ebp+arg_4]-->*source <-
00401013    movsx    ecx, byte ds:[eax]              |
00401016    cmp      ecx, 0x0                        |
00401019    jz       loc_40103C                      |
                                                     |
0040101F    mov      eax, ss:[ebp+var_4] --> *p      |
00401022    mov      ecx, eax                        |
00401024    add      eax, 0x1                        |
00401027    mov      ss:[ebp+var_4], eax             |
0040102A    mov      eax, ss:[ebp+arg_4]             |
0040102D    mov      edx, eax                        |
0040102F    add      eax, 0x1                        |
00401032    mov      ss:[ebp+arg_4], eax             |
00401035    movsx    eax, byte ds:[edx]              |
00401038    mov      byte ds:[ecx], byte al          |
0040103A    jmp      loc_401010 --> loop back to ----
```

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

Characterizing the code patterns (1)

Strided memory access pattern within a loop

Pattern A (strcpy function, rather straightforward)

```
1:  regd ← MEM[base+dest]              DEST adr.
2:  regs ← MEM[base+src]                SRC adr.
loop
3:  MEM[regd] ← MEM[regs]         copy SRC to DEST
4:  regd ← regd+stride
5:  regs ← regs+stride
endloop
```

Characterizing the code patterns (2)

pattern B (bufCopy function, less straightforward):

```
 MEM[base+p] ← MEM[base+dest]                    DEST adr.
loop
1:  reg1 ← MEM[base+p]
2:  regd ← reg1
3:  reg1 ← reg1+stride
4:  MEM[base+p] ← reg1                      next DEST adr.
5:  reg2 ← MEM[base+src]                           SRC adr
6:  regs ← reg2
7:  reg2 ← reg2+stride
8:  MEM[base+src] ← reg2                     next SRC adr.
9:  MEM[regd] ← MEM[regs]                 copy SRC to DEST
endloop
```

Self def-use dependency chains

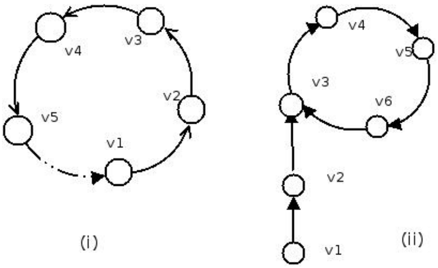
def-use dependency chain

Sequence of the type: $v_1 \to v_2 \to v_3 \to \ldots \to v_k$

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:

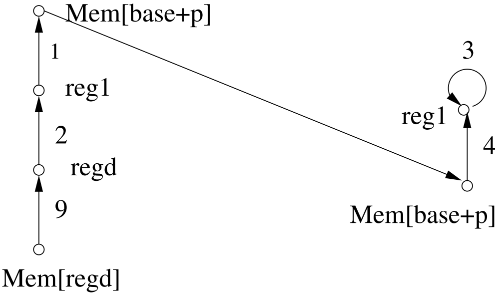


$\Rightarrow$ a simple data-flow analysis (reaching definitions) ...

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]
```



Extension

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

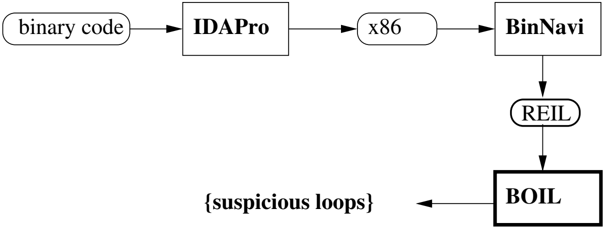
Remark:

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



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

$\rightarrow$ include known vulnerable applications/libraries in the benchmark

Module	# func	# loops	BOILs	BOP func
FreeFloat FTP	309	146	21 (14%)	12 (3%)
CoolPlayer	995	1036	156 (15%)	56 (5%)
GDI32.dll	1775	655	70 (10%)	51 (3%)
freeType	1910	2568	409 (15%)	249 (13%)
msvcr80.dll	2321	1154	188 (16%)	113 (4%)

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.

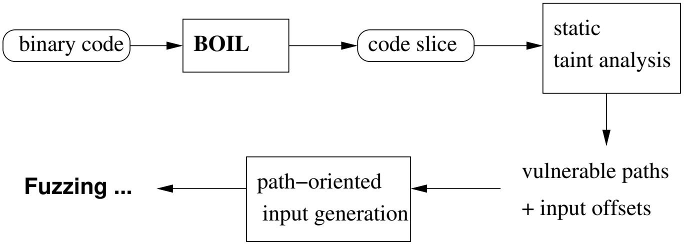
Outline

Conclusion

- Vulnerability detection methods driven by
  vulnerability patterns
- These patterns needs 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:



Similar approaches for other kinds of vulnerabilities:

e.g., **use-after-free**