Combining static analyses for helping detection and exploitability vulnerabilities in binary code

Laurent Mounier, Josselin Feist, Marie-Laure Potet, Sanjay Rawat

VERIMAG
University of Grenoble

November 2013
Outline

1 Introduction
   - Vulnerability analysis
     - Our approach
     - Experimental platform
2 Static Taint analysis
   - Intra-procedural taint analysis
   - An adapted value analysis
3 LiSTT: A Lightweight Static Taint Tracer
   - The proposed approach
   - Inter-procedural analysis
   - Experimental results
4 Use-after-free detection and exploitability
   - UaF
   - Our approach
   - Detection
   - Exploitability
   - Prototype
5 Bibliographie
Practice in terms of vulnerability analysis

1. Identification of flaws
   - dangerous patterns, fuzzing and crashes identification . . .

2. Possibility of exploit (exploitability)
   - poc elaboration, taint analysis, crash analysis . . .

3. Building an real exploit
   - hijacking countermeasures (sandboxing, DEP, ASLR) using well-established techniques and forms of shellcodes
Practice in terms of dynamic/static analysis

- **Dynamic analysis**
  - combinatorial testing exploiting input malformation
  - trace analysis using debugger
  - dynamic instrumentation (memory, taint, ...)

- **Static analysis including (symbolic computation)**
  - identification of sensible parts of code
  - input generation from symbolic paths
  - generalization of traces

⇒ Two complementary approaches with pros/cons that can be combined
Disassembly tools: IDA, OllyDbg, Miasm, Metasm
+ dynamic instrumentation: VALGRIND (Linux), PIN (Intel)
+ static analysis: Tom Reps works (CodeSurfer), Bitblaze (Berkeley), BAP (Brumley/CMU), LLVM/Klee, S2E (EPFL)
Fuzzers: Fuzzgrind (Sogeti), DART, Sage (Microsoft), Dowser (VU Amsterdam)
Dynamic taint analysis: TEMU (Berkeley), Dytan (Georgia Tech), TaintScope (Pekin Univ.)
Exploitability: !Exploitable (an extension of Windbg), AEG (CMU), Mayem (CMU)
Challenges

- Engineering of vulnerability analysis
  - Automatize as much as possible the vulnerability detection step
  - Formalisation of skills in term of exploitability

- Scientific challenges
  - New vulnerabilities such as Use after Free
  - Static analysis at the binary level (scalability/accuracy)
  - Traces analysis leading to an exploit
  - Memory models adapted for exploitability and symbolic analyses

⇒ Tools helping detection and traces classification.
Outline

1 Introduction
   - Vulnerability analysis
   - Our approach
   - Experimental platform

2 Static Taint analysis
   - Intra-procedural taint analysis
   - An adapted value analysis

3 LiSTT: A Lightweight Static Taint Tracer
   - The proposed approach
   - Inter-procedural analysis
   - Experimental results

4 Use-after-free detection and exploitability
   - UaF
   - Our approach
   - Detection
   - Exploitability
   - Prototype

5 Bibliographie
Our approach/Objective

⇒ Identifying exploitable paths and building appropriate inputs (a testing approach)

- Using static analysis in order to slice interesting behaviours
  - structural patterns and static taint analysis (SERE11)
- Using static/dynamic analysis for exploitability condition
  - Symbolic exploitability conditions and memory model (CSTVA12)
- Using mutation algorithms and SMT solvers to produce inputs
  - fitness function and mutations (ECND10, SECTEST11)

⇒ Content of the presentation: Static taint analysis, vulnerable path detection, Use after free detection, Ongoing works
⇒ Prototype for limitation/relevance/scalability evaluation
Outline

1 Introduction
   - Vulnerability analysis
   - Our approach
   - Experimental platform

2 Static Taint analysis
   - Intra-procedural taint analysis
   - An adapted value analysis

3 LiSTT: A Lightweight Static Taint Tracer
   - The proposed approach
   - Inter-procedural analysis
   - Experimental results

4 Use-after-free detection and exploitability
   - UaF
   - Our approach
   - Detection
   - Exploitability
   - Prototype

5 Bibliographie
Ida Pro

- Commercial disassembler and debugger
- Supports 50+ processors (intel, ARM, .NET, PowerPC, MIPS, ...)
- Recognizes library functions- FLIRT (C/C++ only)
- Builds call graphs and CFGs
- Provides scripting environment (C like (IDC), IDAPython)

Static memory address recognition

Syntactic identification of memory accesses:

- offsets with respect to frame or stack pointer
  
  \([\text{ebp} +/\delta\text{delta}], [\text{esp} +/-\delta\text{delta}]\)

  → local variables and arguments

- absolute addressing ([190098])

  ← global variables

⇒ provides a set of initial abstract locations
BinNavi

- Reverse engineering tool by Zynamics (Google)
- Works on the top of IDApro analyzed IDB files
- Has its own GUI for interaction and provides scripting environment (Python, Jython, Ruby, Java)

BinNavi MonoReil

- A framework for forward/backward analysis on cfg
- Provides API to define lattice, lattice element, transfer function, fixed point solution
BinNavi Architecture

IDA pro
Binnavi plugin
IDB
file
BinNavi
GUI
Database
BinNavi
APIs
Own
Algo
code
- Reverse Engineering Intermediate Lang
- Simpler than x86, only 17 opcodes
- 3-address format: inst op1,op2,op3
- Support ARM, MIPS, PowerPC and x86 code

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Operations</th>
<th>Address</th>
<th>Instruction</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>40100A00</td>
<td>str 5, , eax</td>
<td></td>
<td>40401A00A</td>
<td>mov eax, 5</td>
<td></td>
</tr>
<tr>
<td>40100F00</td>
<td>add 0xFFFFFFFFF8, ebp, qword t0</td>
<td></td>
<td>404010F00F</td>
<td>mov ss: [ebp + var_8], eax</td>
<td></td>
</tr>
<tr>
<td>40100F01</td>
<td>and qword t0, 0xFFFFFFFF, t1</td>
<td></td>
<td>404010F017</td>
<td>mov ss: [ebp + var_C], eax</td>
<td></td>
</tr>
<tr>
<td>40100F02</td>
<td>stm eax, , t1</td>
<td></td>
<td>40101200</td>
<td>str 3, , eax</td>
<td></td>
</tr>
<tr>
<td>40101200</td>
<td>str 3, , eax</td>
<td></td>
<td>40101700</td>
<td>add 0xFFFFFFFFF4, ebp, qword t0</td>
<td></td>
</tr>
<tr>
<td>40101701</td>
<td>and qword t0, 0xFFFFFFFF, t1</td>
<td></td>
<td>40101701</td>
<td>and qword t0, 0xFFFFFFFF, t1</td>
<td></td>
</tr>
<tr>
<td>40101702</td>
<td>stm eax, , t1</td>
<td></td>
<td>40101A00</td>
<td>str 5, , eax</td>
<td></td>
</tr>
<tr>
<td>40101A00</td>
<td>str 5, , eax</td>
<td></td>
<td>40101F00</td>
<td>add 0xFFFFFFFFF0, ebp, qword t0</td>
<td></td>
</tr>
<tr>
<td>40101F01</td>
<td>and qword t0, 0xFFFFFFFF, t1</td>
<td></td>
<td>40101F01</td>
<td>and qword t0, 0xFFFFFFFF, t1</td>
<td></td>
</tr>
<tr>
<td>40101F02</td>
<td>stm eax, , t1</td>
<td></td>
<td>40102200</td>
<td>add 0xFFFFFFFFF8, ebp, qword t0</td>
<td></td>
</tr>
<tr>
<td>40102201</td>
<td>and qword t0, 0xFFFFFFFF, t1</td>
<td></td>
<td>40102202</td>
<td>ldm t1, , t2</td>
<td></td>
</tr>
<tr>
<td>40102203</td>
<td>str t2, , eax</td>
<td></td>
<td>40102202</td>
<td>ldm t1, , t2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40102202</td>
<td>str t2, , eax</td>
<td></td>
</tr>
</tbody>
</table>
Outline

1. Introduction
   - Vulnerability analysis
   - Our approach
   - Experimental platform

2. Static Taint analysis
   - Intra-procedural taint analysis
   - An adapted value analysis

3. LiSTTT: A Lightweight Static Taint Tracer
   - The proposed approach
   - Inter-procedural analysis
   - Experimental results

4. Use-after-free detection and exploitability
   - UaF
   - Our approach
   - Detection
   - Exploitability
   - Prototype

5. Bibliographie
Taint Analysis

- Identify **input dependent** variables at each program location
- Two kinds of dependencies:

  **Data dependencies**

  // x is tainted
  
  \[
  y = x \ ; \ z = y + 1 \ ; \ y = 3 \ ;
  \]

  // z is tainted

  **Control dependencies**

  // x is tainted

  \[
  \text{if} \ (x > 0) \ y = 3 \ \text{else} \ y = 4 \ ;
  \]

  // y is tainted

  ⇒ the STAC Frama-C plug-in [RCMP13]
  ⇒ Binary level: restricted on **data dependencies** (classical exploits)
Static taint data-dependency analysis

data-flow analysis problem:

- input functions return tainted values
- constants are untainted
- forward computation of a set of pairs \((v, T)\) at each program location:
  - \(v\) is a register or a memory
  - \(T \in \{\text{Tainted}, \text{Untainted}\}\) is a taint value
- fix-point computation (backward dependencies inside loops)

\[\Rightarrow\] More complex than source-level taintness

Need some VSA

- to track register and memory transfers (and then addresses and size)

Combining static analyses for helping detection and exploitability vulnerabilities in binary code
Taint analysis at the assembly level

y at ebp-8, x at ebp-4 and z at ebp-12.

\[
y = 3 ;
\]

1: \( t3 := 3 \)
2: \( t4 := ebp-8 \)
3: \( \text{Mem}[t4] := t3 \)

\[\ldots\]

7: \( t5 := ebp-8 \)
8: \( t6 := \text{Mem}[t5] \)
9: \( t7 := ebp-12 \)
10: \( \text{Mem}[t7] := t6 \)

Needs to identify that:

- content of reg. \( t5 \) at line 7 = content of reg. \( t4 \) at line 2
- value written at line 8 \( \leftarrow \) mem. loc. written at line 3
Outline

1 Introduction
   - Vulnerability analysis
   - Our approach
   - Experimental platform

2 Static Taint analysis
   - Intra-procedural taint analysis
   - An adapted value analysis

3 LiSTT: A Lightweight Static Taint Tracer
   - The proposed approach
   - Inter-procedural analysis
   - Experimental results

4 Use-after-free detection and exploitability
   - UaF
   - Our approach
   - Detection
   - Exploitability
   - Prototype

5 Bibliographie
Value Set Analysis (VSA)

Memloc addresses:
- local variables and parameters $\rightsquigarrow$ offset w.r.t to ebp
- global variables $\rightsquigarrow$ fixed value
- dynamically allocated memory $\rightsquigarrow$ return values from malloc

\[
\begin{align*}
\text{Offset} &= \text{INT} \\
\text{Base} &= \{\text{Any, None, Global}\} \cup \{\text{Init}(r) \mid r \in \text{REG}\} \cup \text{HE} \\
\text{Value} &= \text{Base} \times P(\text{Offset}) \\
\text{Name}_{val} &= \text{REG} \cup (\text{Base} - \{\text{Any, None}\} \times \text{Offset}) \\
\text{AbsEnv} &= \text{Name}_{val} \rightarrow P(\text{Value})
\end{align*}
\]

\[
\begin{align*}
E_1 \sqcup E_2 &= E_2 \quad \text{if } E_1 = \emptyset \\
((b_1, x_1)) \sqcup E_1 \sqcup E_2 &= E_1 \sqcup ((b_1, x_1)) \cup E_2 \\
&\quad \quad \text{if } \forall (b, x) \in E_2 . \ b \neq b_1 \\
&= E_1 \sqcup ((b_1, x_1 \cup x)) \cup E_2 - \{(b_1, x)\} \quad \text{Otherwise}
\end{align*}
\]

Approximation: $\cup$ can produce INT and $\sqcup$ can produce Any.
Add $\text{op1}, \text{op2}, \text{op3}$

$$E_{\text{out}} := E_{\text{in}} \leftarrow \{ \text{op3} \mapsto \bigcup_{v_1 \in E_{\text{in}}(\text{op1}), v_2 \in E_{\text{in}}(\text{op2})} \text{Add}(v_1, v_2) \}$$

with $\text{Add}$ defined as follows:

$$\text{Add}((B_1, X_1), (B_2, X_2)) = \begin{cases} \{(B_1, \{x_1 + x_2 \mid x_1 \in X_1 \land x_2 \in X_2\})\} & \text{if } B_2 = \text{None} \\
\{(B_2, \{x_1 + x_2 \mid x_1 \in X_1 \land x_2 \in X_2\})\} & \text{if } B_1 = \text{None} \\
(\text{Any}, \emptyset) & \text{otherwise} \end{cases}$$

STM $\text{op1}, \text{op2}$ (meaning $\text{MEM}\{\text{op2}\} := \text{op1}$)

- if $E_{\text{in}}(\text{op2})$ denotes a single memory location, then this memory location is assigned with values associated to $\text{op1}$ (strong update);
- otherwise, for each element $a$ of $E_{\text{in}}(\text{op2})$, the current value of $\text{MEM}\{a\}$ is merged with the previous value of $\text{op1}$ (weak update).

Assuming $\text{Name}(E_{\text{in}}(\text{op2})) = E$:

$$E_{\text{out}} = \begin{cases} E_{\text{in}} \leftarrow \{n \mapsto E_{\text{in}}(\text{op1})\} & \text{if } E = \{n\} \\
E_{\text{in}} \leftarrow \{a \mapsto E_{\text{in}}(\text{op1}) \sqcup E_{\text{in}}(a) \mid a \in E\} & \text{if } |E| > 1 \end{cases}$$

With $\text{Name} : P(\text{Value}) \rightarrow P(\text{Name}_{\text{val}})$.

Combining static analyses for helping detection and exploitability vulnerabilities in binary code
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>AbsEnv</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( t3 := 3 )</td>
<td>( t3 \mapsto {(\text{None}, {3})}, \ldots )</td>
</tr>
<tr>
<td>2: ( t4 := ebp-8 )</td>
<td>( t4 \mapsto {(\text{Init}(EBP), {-8})}, \ldots )</td>
</tr>
<tr>
<td>3: ( \text{Mem}[t4] := t3 )</td>
<td>( \text{(Init}(EBP), -8) \mapsto {(\text{None}, {3})}, \ldots )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>7: ( t5 := ebp-8 )</td>
<td>( t5 \mapsto {(\text{Init}(EBP), {-8})}, \ldots )</td>
</tr>
<tr>
<td>8: ( t6 := \text{Mem}[t5] )</td>
<td>( t6 \mapsto {(\text{None}, {3})}, \ldots )</td>
</tr>
<tr>
<td>9: ( t7 := ebp-12 )</td>
<td>( t7 \mapsto {(\text{Init}(EBP), {-12})}, \ldots )</td>
</tr>
<tr>
<td>10: ( \text{Mem}[t7] := t6 )</td>
<td>( \text{(Init}(EBP), -12) \mapsto {(\text{None}, {3})}, \ldots )</td>
</tr>
</tbody>
</table>
Implementation and Possible improvements

- **Implementation**
  - Taint and VSA intra-procedural analyses using BinNavi
  - Large approximations

- **Tested improvements:**
  - Restrict VSA to registers and memory locations involved in address computations: backward analysis tracking registers used as memory destination
  - An inter-procedural taint based on a lightweight value analysis

- **ToDo**
  - Take into account the size of memory transfers
  - Strided intervals in particular for a fine-grained array treatment
Reps & Balakrishnan VSA

Objective = **program verification**

4 distinct memory regions
- global, local, heap and registers

Value representation

Strided intervals on a $k$-bits memory:

$$s[l, u] = \{i \in [-2^{k-1}, 2^{k-1} - 1] | l \leq i \leq u, i \equiv l \mod s\}$$

↔ good representation for array/field access operations

Iterative VSA

- iterate VSA and “structure identification” techniques to refine memory addresses recognition
- widening and affine relation analysis to retrieve index-based array iterations
- Recency-abstraction for heap-allocation (strong/weak update)
Debray’s et al. Alias Analysis

Value-analysis to identify **alias between memory references**: 

\[ \sim \] register sets containing **same values** at **same locations**

- computes only register values (not memory contents)
  \[ \hookrightarrow \] coarse abstractions \ldots
- use statement-based symbolic names for “unknown” values

\[ (m \text{ at } i, x) \sim \{ v + x \mid v \in \text{ concrete val. of } m \text{ at } i \} \]

**Example:**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>242</td>
<td>t0 = some unknown val.</td>
<td>( (t0 \text{ at } 242, 0) )</td>
</tr>
<tr>
<td>243</td>
<td>t1 = t0+4</td>
<td>( (t0 \text{ at } 242, 4) )</td>
</tr>
</tbody>
</table>

- may improve the accuracy of the analysis
- default solution for results of `malloc` calls
- needs some extra checks on the CFG to compare addresses
Outline

1 Introduction
   - Vulnerability analysis
   - Our approach
   - Experimental platform

2 Static Taint analysis
   - Intra-procedural taint analysis
   - An adapted value analysis

3 LiSTT: A Lightweight Static Taint Tracer
   - The proposed approach
   - Inter-procedural analysis
   - Experimental results

4 Use-after-free detection and exploitability
   - UaF
   - Our approach
   - Detection
   - Exploitation
   - Prototype

5 Bibliographie
Vulnerable path characterization

Objective: tracking data-dependencies in large applications (several thousands of functions)

**Inputs**
- **input sources** IS (→ tainted data)
- **vulnerable statement/function** VF (→ sensitive params)
- a vulnerable path = contains a VF that can be triggered by a tainted data
  
  \[
  x = \text{IS()} \quad \cdots \quad y := x \quad \cdots \quad \text{VF}(y)
  \]

**Output**
- a set of **vulnerable paths** in terms of call-graph slices
Scalability issues

Fine-grained dataflow analysis not applicable on large programs:
- consider only some forms of data-flow propagation
- operate at fine-grained level only on a program slice
- some parts of the code are considered as irrelevant or approximated (program chopping, dynamic impact analysis)

Information flows taken into account

Inside procedures:
- assignments:  $x := y + z$

From caller to callee:
- arguments: $\text{foo}(x, y+12)$

From callee to caller:
- return value and pointer to arguments: $z = \text{foo}(x, \&y)$

Combining static analyses for helping detection and exploitability vulnerabilities in binary code
```c
int main() {
    char dest[512], char *src, *tmp;
    src = read_data(); // IS, taints src
    tmp = src; // propagation
    process_data(dest, tmp); // calls VF1
    strcpy (dest, "processing OK"); // VF2
    return 0;
}

char *read_data() {
    char *buf;
    ReadFile(buf); // IS
    return buf;
}

void process_data(char *b1, char *b2)
{ strcpy(b1, b2); // b2 sensitive
}
```
Outline

1 Introduction
   - Vulnerability analysis
   - Our approach
   - Experimental platform

2 Static Taint analysis
   - Intra-procedural taint analysis
   - An adapted value analysis

3 LiSTT: A Lightweight Static Taint Tracer
   - The proposed approach
   - Inter-procedural analysis
   - Experimental results

4 Use-after-free detection and exploitability
   - UaF
   - Our approach
   - Detection
   - Exploitability
   - Prototype

5 Bibliographie

Combining static analyses for helping detection and exploitability vulnerabilities in binary code
Summary computation

Computes data-dependencies between:

- function **inputs** (input parameters)
- and function **outputs** (return value and output parameters)

A forward analysis: computes a mapping \( MLoc \rightarrow_d 2^{MLoc} \)

\( \rightarrow_d \) associates to each MLoc its set of data-dependencies:

\[ x \rightarrow_d \{ y_1, \ldots y_n \} \text{ if } x \sim y_1 \text{ and } \ldots \text{ and } x \sim y_n \]

where \( x \sim y \) means “value of \( x \) can flow to \( y \)”

**ex:** \((epb+12) \rightarrow_d \{(eax), *(epb+16)\}\)

Taintness and sensitivity

- identify sources of taintness or sensitivity
- propagate both taintness and sensitivity based on data-dependencies computation
→ A bottom-up summaries computation

**intra-procedural level: summary computation**

```c
int foo(int x, int *y){
    int z;
    z = x+1 ; *y = z ;
    return z ; }
```

**Summary:** z and *y depend on x

x tainted ⇒ z and *y tainted, z or *y sensitive ⇒ x sensitive

**inter-procedural level: apply summaries to effective parameters**

```c
read(b) ; // taints b
a = foo (b+12, &c) ; // a and c are now tainted ...
vuln(a) ; // a sensitive (⇒ b sensitive)
```
Objective

Fine-grained data-flow analysis not applicable on large programs
⇒ Chopping based on the Call Graph structure

Information can flow from IS to VF iff ∃ an execution path s.t.:

\[ begin_P \rightarrow \cdots \rightarrow end_{IS} \rightarrow \cdots \rightarrow begin_{VF} \rightarrow \cdots \rightarrow end_P \]

⇒ ∃ a ("root") procedure R s.t.:

\[ begin_P \rightarrow \cdots \rightarrow \ldots begin_R \cdots \rightarrow end_{IS} \ldots \]
\[ \rightarrow begin_{VF} \rightarrow \cdots \rightarrow end_R \rightarrow end_P \]

→ ∃ Two relevant sets of paths in the Call Graph:

- paths leading from R to IS
- paths leading from R to VF
Splitting the Call Graph into regions

Regions defined w.r.t reachability of IS and VF from R

Combining static analyses for helping detection and exploitability vulnerabilities in binary code
### Region X:

- can be ignored, **no summary computations**

### Regions S’1 and S’2: consider a **default summary**

- does not produce taintness or sensitivity
- create dependencies between the function inputs and outputs

\[ z = \text{foo}(x, \&y) \]

- \( z \) and \( y \) depend on \( x \) and \( y \)

### Regions S1 and S2:

**explicit summary computations**
Outline

1 Introduction
   - Vulnerability analysis
   - Our approach
   - Experimental platform

2 Static Taint analysis
   - Intra-procedural taint analysis
   - An adapted value analysis

3 LiSTT: A Lightweight Static Taint Tracer
   - The proposed approach
   - Inter-procedural analysis
   - Experimental results

4 Use-after-free detection and exploitability
   - UaF
   - Our approach
   - Detection
   - Exploitability
   - Prototype

5 Bibliographie
Tool Architecture

Executable → IDA Pro → x86 → BinNavi → REIL → Vulnerable Path detection

- Value Analysis
- VF detection

Actuals to formal parameters mapping
- Context sensitive analysis based on an actual-to-formal mapping at each call site
- PUSHed instructions identified at the x86 level + a specific data-flow analysis at the REIL level

Combining static analyses for helping detection and exploitability vulnerabilities in binary code
Objective of the Experiments

Setup and Objectives

- Test on real applications, with user-provided input source functions and vulnerable functions detection using [RM12]
- Retrieve existing vulnerabilities? Size of resulting slices? Scalability of the analysis?

<table>
<thead>
<tr>
<th>Application</th>
<th>#Func</th>
<th>#VF</th>
<th>#Slices (# Func)</th>
<th># Vuln. Paths (# Func)</th>
<th>Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>muPDF PDF viewer</td>
<td>7722</td>
<td>303</td>
<td>71 (7)</td>
<td>5 (4)</td>
<td>25 mn</td>
</tr>
<tr>
<td>FoxPlayer Audio player</td>
<td>1074</td>
<td>41</td>
<td>16 (8)</td>
<td>5 (6)</td>
<td>33 mn</td>
</tr>
<tr>
<td>Serenity Audio player</td>
<td>559</td>
<td>1</td>
<td>1 (3)</td>
<td>1 (3)</td>
<td>3 sec</td>
</tr>
<tr>
<td>htget</td>
<td>144</td>
<td>10</td>
<td>5 (3)</td>
<td>2 (3)</td>
<td>8 mn</td>
</tr>
</tbody>
</table>
- 7722 functions, 303 flagged as “vulnerable” (VF)
- Input Source: fz_open_document

→ 71 call-graph slices found, 5 of them with a tainted path

Example: CVE-2011-0341
LiSTT: a (pre-processing) tool for vulnerability detection

- based on interprocedural and scalable data-flow analysis
- application to static taint analysis for BoF detection
- promising results

Going further with LiSTT

- possibility to take into account global variables
- refine/improve the results produced
  - extract a set of vulnerable paths, at the CFG level
  - slices used as input by other analysis (fuzzing, symbolic execution, etc.)
- application to other kinds of vulnerabilities
1 Introduction
   - Vulnerability analysis
   - Our approach
   - Experimental platform

2 Static Taint analysis
   - Intra-procedural taint analysis
   - An adapted value analysis

3 LiSTT: A Lightweight Static Taint Tracer
   - The proposed approach
   - Inter-procedural analysis
   - Experimental results

4 Use-after-free detection and exploitability
   - UaF
   - Our approach
   - Detection
   - Exploitability
   - Prototype

5 Bibliographie
typedef struct {
    void (*f)(void);
} st;

void nothing()
{
    printf("Nothing\n");
}

int main(int argc, char * argv[])
{
    st *p1;
    char *p2;
    p1 = (st*) malloc(sizeof(st));
    p1->f = &nothing;
    free(p1); // p1 freed
    p2 = malloc(strlen(argv[1])); // possible re-allocation
    strcpy(p2, argv[1]);
    p1->f(); // Use
    return 0;
}
Motivations

- *Use-After-Free* more and more frequent
- Static approach for finding exploitable vulnerabilities
  → an adapted modelling of the heap

![Graph showing the number of CVEs related to UaF over years](https://web.nvd.nist.gov/view/vuln/search, 4 June 2013)
State of art

Specificity of UaF
- No easy "pattern" (like for buffer overflow / string format)
- Trigger of several dispatched events (alloc/free/use)
- Strongly depends on the allocation/liberation strategy

Binary code
On binary code, state of the art focused more on dynamic analysis
- Fuzzing + custom allocator (AddressSanitizer)
- Exploit studied after UaF found (Undangle)
But some static analysis on binary seems exist:
- tools based on BAP ? Bugwise (WIRE format/Deakin University) ?
Outline

1. Introduction
   - Vulnerability analysis
   - Our approach
   - Experimental platform

2. Static Taint analysis
   - Intra-procedural taint analysis
   - An adapted value analysis

3. LiSTT: A Lightweight Static Taint Tracer
   - The proposed approach
   - Inter-procedural analysis
   - Experimental results

4. Use-after-free detection and exploitability
   - UaF
   - Our approach
     - Detection
     - Exploitability
     - Prototype

5. Bibliographie
Proposed approach

Goal: extract subgraphs of CFG leading to exploitable Use-After-Free

Approach

2 steps:
1: Detection of Use-After-Free
   - Value analysis
   - Characterization of Use-After-Free
2: Exploitability of Use-After-Free
   - Determining possible re-allocations
   - Exploitability condition (ongoing work)

Semi-automatic: manually description of allocation strategies
### Modelling heap
- **$HE$** = all possible memory blocks in the heap
- Member of $HE$ represented $(heap_i, size_i)$ (simplified in $chunk_i$)
- $HA(pc)$ (resp. $HF(pc)$) member of $HE$ allocated (resp. freed)
- $HA: PC \rightarrow \mathcal{P}(HE)$
- $HF: PC \rightarrow \mathcal{P}(HE)$
- $HA(pc) \cap HF(pc) = \emptyset$

### VSA for detection
- Track allocation, free and heap accesses
- Size of allocation (for exploitability)
- One allocation = new $chunk$
Transfer functions for heap operations

1: function malloc(pc, size)
2:     id := id_max;
3:     id_max ++;
4:     HA := HA ← {pc ↦→ (HA(pc) ∪ {(base_id, size)})};
5:     point_alloc := point_alloc ← {(base_id, size) ↦ pc};
6:     return (base_id, size)
7: end function

1: function Free(pc, (base_x, size))
2:     HA := HA ← {pc ↦→ (HA(pc) \ {(base_x, size)})};
3:     HF := HF ← {pc ↦→ (HF(pc) ∪ {(base_x, size)})};
4:     point_free := point_free ← {(base_x, size) ↦
5:     {point_free(base_x, size) ∪ pc}};
6: end function
1 Introduction
   • Vulnerability analysis
   • Our approach
   • Experimental platform
2 Static Taint analysis
   • Intra-procedural taint analysis
   • An adapted value analysis
3 LiSTT: A Lightweight Static Taint Tracer
   • The proposed approach
   • Inter-procedural analysis
   • Experimental results
4 Use-after-free detection and exploitability
   • UaF
   • Our approach
   • Detection
   • Exploitability
   • Prototype
5 Bibliographie
typedef struct {
    void (*f)(void);
} st;

int main(int argc, char * argv[])
{
    st *p1;
    char *p2;
    p1 = (st *) malloc(sizeof(st));
    free(p1);
    p2 = malloc(sizeof(int));
    strcpy(p2, argv[1]);
    p1->f();
    return 0;
}

<table>
<thead>
<tr>
<th>Code</th>
<th>AbsEnv</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 : p1=(st*)malloc(sizeof(st))</td>
<td>(Init(EBP), -4) $\mapsto$ {chunk}_0,...</td>
<td>HA = {chunk}_0 &lt;br&gt;HF = \emptyset</td>
</tr>
<tr>
<td>10 : free(p1)</td>
<td>(Init(EBP), -4) $\mapsto$ {chunk}_0,...</td>
<td>HA = \emptyset &lt;br&gt;HF = {chunk}_0</td>
</tr>
<tr>
<td>11 : p2=malloc(sizeof(int))</td>
<td>(Init(EBP), -4) $\mapsto$ {chunk}_0, &lt;br&gt;(Init(EBP), -8) $\mapsto$ {chunk}_1</td>
<td>HA = {chunk}_1,... &lt;br&gt;HF = {chunk}_0</td>
</tr>
</tbody>
</table>
Detection: characterization of Use-After-Free

**AccessHeap**

AccessHeap returns all elements of HE that are accessed at pc

Examples with REIL memory transfer instructions:

- \( \text{AccessHeap}(LDM \ ad, , \ reg) = \text{AbsEnv}(ad) \cap HE. \)
- \( \text{AccessHeap}(STM \ reg, , \ ad) = \text{AbsEnv}(ad) \cap HE \)

**Research the use of a freed element of the heap**

- \( \text{EnsUaf} = \{(pc, chunk) \mid \text{chunk} \in \text{AccessHeap}(pc) \cap \text{HF}(pc)\} \)
- Extraction of executions leading to each Use-After-Free: all reachable nodes including the following paths:
  - \( pc_{entry} \rightarrow pc_{alloc} \)
  - \( pc_{alloc} \rightarrow pc_{free} \)
  - \( pc_{free} \rightarrow pc_{uaf} \)
Example: *Use-After-Free* detection and extraction

Combining static analyses for helping detection and exploitability vulnerabilities in binary code
Outline

1 Introduction
   - Vulnerability analysis
   - Our approach
   - Experimental platform

2 Static Taint analysis
   - Intra-procedural taint analysis
   - An adapted value analysis

3 LiSTT: A Lightweight Static Taint Tracer
   - The proposed approach
   - Inter-procedural analysis
   - Experimental results

4 Use-after-free detection and exploitability
   - UaF
   - Our approach
   - Detection
   - Exploitability
   - Prototype

5 Bibliographie
We consider a Uaf as exploitable if another pointer points to the same memory zone (~ alias unwanted).

**Steps**

1. Determine paths where new allocations take place between the free and use locations
2. Determine if some allocations can reallocate the same memory area: based on a particular allocation strategy (worst case, all allocations are considered as dangerous)
3. Is the size of new allocations a tainted value? Is the content modified by a tainted value?
4. How is the AccessHeap used: a read, write or jump patterns?
1. Extracting paths with re-allocations

Replay allocations between free $\rightarrow$ use

- Allocation order is important for exploitability
- Find all "heap operations paths" (with loop summary)
2. Replay re-allocations

Reallocate of the same memory area

- Simulate an allocator on each "heap operation path" replaying VSA
- Allocator modelisation (with potentially a new heap model):
  - Define some general behaviour/property of allocator:
    - P1: Heap space is divided into blocks. Blocks are classified according to their size and state (allocated/freed)
    - P2: A new block can take place into a freed block
    - P3: A freed block can be split
    - P4: Two freed blocks can be consolidated
    - ...

<table>
<thead>
<tr>
<th>Code</th>
<th>Heap</th>
</tr>
</thead>
</table>
| 9 : \text{p1=\textit{(st*) malloc(sizeof(st))}} | \text{HA} = \{(heap_0, 4)\}  \\
|            | \text{HF} = <> |
| 10 : \text{free(p1)} | \text{HA} = 0  \\
|            | \text{HF} = < (heap_0, 4) > |
| 11 : \text{p2=malloc(sizeof(int))} | \text{HA} = \{(heap_0, 4)\}  \\
|            | \text{HF} = <> |
```c
typedef struct {
    void (*f)(void);
} st;

void nothing() {
    printf("Nothing\n");
}

int main(int argc, char * argv[]) {
    st *p1;
    char *p2;
    p1 = (st*) malloc(sizeof(st));
    p1->f = &nothing;
    free(p1);
    p2 = malloc(strlen(argv[1])); // size is tainted
    strcpy(p2, argv[1]); // content of p2 is tainted
    p1->f(); // Access as a jump
    return 0;
}
```
Discussions on the approach

Separating detection / exploitability

- Triggering *Use-After-Free* independent of the allocation strategy
  - Programming error, always present
  - "Cause" of *Use-After-Free*
- Exploitability of *Use-After-Free* depending on the allocation strategy
  - What has happened between the free / use of the item?
  - "Consequence" of *Use-After-Free*
- Advantage of this approach:
  - Using "classic" technique for detecting
  - Study of exploitability on a subset of possible executions of the program
  - For an *Use-After-Free* detected opportunity to study several allocation strategies (or worst case)
Outline

1 Introduction
   - Vulnerability analysis
   - Our approach
   - Experimental platform

2 Static Taint analysis
   - Intra-procedural taint analysis
   - An adapted value analysis

3 LiSTT: A Lightweight Static Taint Tracer
   - The proposed approach
   - Inter-procedural analysis
   - Experimental results

4 Use-after-free detection and exploitability
   - UaF
   - Our approach
   - Detection
   - Exploitability
   - Prototype

5 Bibliographie
⇒ only *Use-After-Free* detection step

**Characteristic**
- *IDA Pro* + *BinNavi*
- *Jython* ≃ 3000 lines

**VSA**
- loops are unrolled 0 and 1 times
- *Naive* version of inter-procédural

**Validation**
- Validation of the approach on simple examples
- Further study of a CVE
Relevance of the approach

Real *Use-After-Free*

- ProFTPD : CVE 2011-4130, studied by Vupen
- Structures, function pointer, global variables...
- Assisted detection (subset of 10 functions).
- From 2200 nodes $\rightarrow$ 460, 30 min on i7-2670QM
Conclusion and Perspectives

- Use of subgraphs and VSA for smart fuzzing
- An adapted IR and flow graph construction and memory model ANR project (BinSec)
- More efficient implementation
- Exploitability steps (including impact of exploitability)
- Build traces using symbolic exploitability conditions (and allocation strategy)
- Detection of home-made allocators
- Complexity of *Use-After-Free* in navigators (several allocation locations including GC, heap spraying)
Outline

1 Introduction
   - Vulnerability analysis
   - Our approach
   - Experimental platform

2 Static Taint analysis
   - Intra-procedural taint analysis
   - An adapted value analysis

3 LiSTT: A Lightweight Static Taint Tracer
   - The proposed approach
   - Inter-procedural analysis
   - Experimental results

4 Use-after-free detection and exploitability
   - UaF
   - Our approach
   - Detection
   - Exploitability
   - Prototype

5 Bibliographie
Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier.
A taint based approach for smart fuzzing.

Josselin Feist, Laurent Mounier, and Marie-Laure Potet.
Statically detecting Use-after-Free on binary code.

Gustavo Grieco, Laurent Mounier, Marie-Laure Potet, and Sanjay Rawat.
A stack model for symbolic buffer overflow exploitability analysis.
In *Proceedings of CSTVA (ICST Workshop)*, pages 216–217, Luxembourg, March 2013. IEEE.

Sanjay Rawat, Dumitru Ceara, Laurent Mounier, and Marie-Laure Potet.
Combining static and dynamic analysis for vulnerability detection.

Sanjay Rawat and Laurent Mounier.
Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: Few preliminary results.

Sanjay Rawat and Laurent Mounier.
Finding buffer overflow inducing loops in binary executables.
In *Proceedings of Sixth International Conference on Software Security and Reliability (SERE)*, pages 177–186, Gaithersburg, Maryland, USA, 2012. IEEE.
GreHack’13: int. symposium in Grey-Hat Hacking Conference

- PC: The Grugq, Fermin J Serna, Manuel Egele, Eric Filiol, etc
- Papers: **28% (9/32)** acceptance rate
  - Invited Speakers: H. Bos, H. Flake, J. Caballero
  - Speakers: Ruo Ando (Japan), E. Leverett (IOActive US) . . .
- **220 attendees**
  - 50% security engineers
  - several pentest teams (italia, spain, france)
- 95% of last wave tickets sold within 1 morning!

Capture The Flag (CTF)

- 100 Competitors; international teams
- Prizes: HP Slate 7 Tablet PC
- pizza