Lightweight Static Taint Analysis for Binary Executables Vulnerability Testing

Sanjay Rawat, Laurent Mounier and Marie-Laure Potet

VERIMAG
University of Grenoble
Grenoble, France

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.

A test-based vulnerability detection technique

A static phase: Identifying "vulnerable execution paths"

- a notion of **vulnerable statement/function** VF
- a vulnerable path =
  contains a VF **that can be triggered** by a **program input**
- computed by **static analysis** of the program code

A dynamic phase: Fuzzing

- exercise at runtime the vulnerable paths
- expected results = assertion violations, **program crash**, etc.
- input selection: concolic execution, **genetic algorithms**, etc.

Vulnerable paths ($\sim$ test objectives)

Vulnerable functions VF

- unsafe library functions (`strcpy`, `memcpy`, etc.) or code patterns (unchecked buffer copies, memory de-allocations)
- critical parts of the code (credential checkings)
- etc.

Vulnerable paths = execution paths allowing to

- read external inputs (keyboard, network, files, etc.) on a memory location M_i
- call a vulnerable function VF with parameter values depending on M_i

Vulnerable paths detection based on **taint analysis**

Input:

- a set of input sources (IS) = **tainted data**
- a set of vulnerable functions (VF)

Output:

- a set of tainted paths =
  tainted data-dependency paths from IS to VF

$$\mathrm{x{=}IS()} \cdots \longrightarrow \cdots \ \mathrm{y} := \mathrm{x} \cdots \longrightarrow \cdots \ \mathrm{VF(y)}$$

Objectives of this work

→ Staticaly compute vulnerable execution paths:

- on large applications (several thousands of functions)
  scalability issues ⇒ lightweight analysis
- from **binary executable** code

→ Links with more general (test related) problems:

- interprocedural information flow analysis
- **program chopping** [T. Reps], **impact analysis**

Outline

Hypothesis on information flows

Inside procedures:

assigments: `x := y + z`

From caller to callee:

arguments: `foo (x, y+12)`

From callee to caller:

return value and pointer to arguments: `z = foo (x, &y)`

$\Rightarrow$ compute **procedure summaries** to express these dependencies.

Example

```
int main() {
    char dest[512], char *src, *tmp;
    src = read_data();                     // IS, taints src
    tmp = src;                              // alias
    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) ; // VF1    }
```

A summary-based inter-procedural data-flow analysis

intra-procedural level: summary computation

→ express side-effects wrt taintedness and aliases

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

Summary: x is tainted ⇒ z is tainted, z and *y are aliases

inter-procedural level: apply summaries to effective parameters

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

A summary-based inter-procedural data-flow analysis

intra-procedural level: summary computation

→ express side-effects wrt taintedness and aliases

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

Summary: x is tainted ⇒ z is tainted, z and *y are aliases

inter-procedural level: apply summaries to effective parameters

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

Scalability issues

Fine-grained data-flow analysis $\rightarrow$ not applicable on large programs

$\Rightarrow$ needs some "aggressive" approximations:

- some deliberate **over-approximations** (global variables, complex data structures, etc.)
- consider only **data-flow** propagation
- operate at fine-grained level only on a **program slice** (parts of the code outside the slice either **irrelevant** or **approximated**)

Slicing: basic idea

Inter-procedural information flow from IS to VF
How to reduce the set of procedures to be analysed ?

→ Split this set into 3 parts:

1. procedure that are not relevant
2. procedure those side-effect can be (implicitely) over-approximated
3. procedure requiring a more detailed analysis

→ A slice computation performed at the **call graph** level (inspired from *dynamic impact analysis*)

Call Graph and execution paths

Call Graph CG $= (E, \rightarrow_{CG})$, where:

E is the set of procedures, $p \rightarrow_{CG} q$ iff p calls q

Information can flow from IS to VF iff $\exists$ an execution path s.t.:

$$begin\_P \rightarrow \cdots \rightarrow end\_IS \rightarrow \cdots \rightarrow begin\_VF \rightarrow \cdots \rightarrow end\_P$$

$\Rightarrow \exists$ a ("root") procedure R s.t.:

$$begin\_P \rightarrow \cdots \rightarrow \ldots \mathbf{begin\_R} \cdots \rightarrow end\_IS \ldots$$
$$\rightarrow begin\_VF \rightarrow \cdots \rightarrow \mathbf{end\_R} \rightarrow end\_P$$

$\rightarrow \exists$ Two relevant set of paths in the Call Graph:

- paths leading from R to IS
- paths leading from R to VF

Call Graph and execution paths

Call Graph CG $= (E, \rightarrow_{CG})$, where:

E is the set of procedures, $p \rightarrow_{CG} q$ iff p calls q

Information can flow from IS to VF iff $\exists$ an execution path s.t.:

$$begin\_P \rightarrow \cdots \rightarrow end\_IS \rightarrow \cdots \rightarrow begin\_VF \rightarrow \cdots \rightarrow end\_P$$

$\Rightarrow$ $\exists$ a ("root") procedure R s.t.:

$$begin\_P \rightarrow \cdots \rightarrow \ldots \mathbf{begin\_R} \cdots \rightarrow end\_IS \ldots$$
$$\rightarrow begin\_VF \rightarrow \cdots \rightarrow \mathbf{end\_R} \rightarrow end\_P$$

$\rightarrow$ $\exists$ Two relevant set of paths in the Call Graph:

- paths leading from R to IS
- paths leading from R to VF

Call Graph and execution paths

Call Graph CG $= (E, \rightarrow_{CG})$, where:

E is the set of procedures, $p \rightarrow_{CG} q$ iff p calls q

Information can flow from IS to VF iff $\exists$ an execution path s.t.:

$$begin\_P \rightarrow \cdots \rightarrow end\_IS \rightarrow \cdots \rightarrow begin\_VF \rightarrow \cdots \rightarrow end\_P$$

$\Rightarrow$ $\exists$ a ("root") procedure R s.t.:

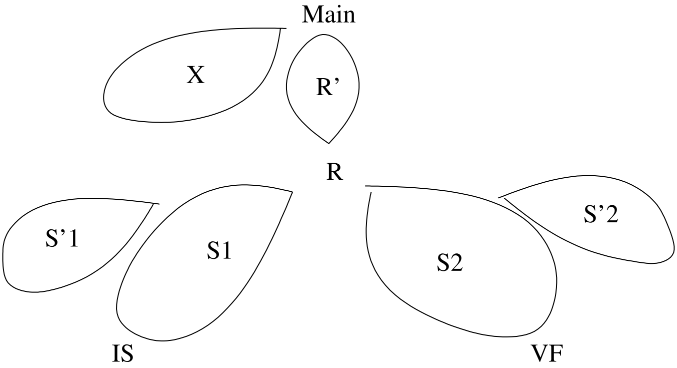
$$begin\_P \rightarrow \cdots \rightarrow \ldots \mathbf{begin\_R} \cdots \rightarrow end\_IS \ldots$$
$$\rightarrow begin\_VF \rightarrow \cdots \rightarrow \mathbf{end\_R} \rightarrow end\_P$$

$\rightarrow$ $\exists$ Two relevant set of paths in the Call Graph:

- paths leading from R to IS
- paths leading from R to VF

Spliting the Call Graph into regions

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



Procedure Summaries associated to Call Graph regions

Region X:

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

Regions S'1 and S'2: consider a **default summary**

- propagate tainted inputs
- create aliases between return values and pointer to arguments

```
z = foo (x, &y)
```

`x` tainted or `y` tainted ⇒ `z` and `y` tainted

`z` and `y` are may-aliases

Regions S1 and S2:

fine-grained summary computations ...

Binary-level intra-procedural data-flow analysis

2 objectives:

- taint propagation
- alias detection

Address a set of abstract memory locations MemLoc:

- registers
- (pointers to) local variables and arguments = offset w.r.t a base register (ebp)
- global variables = fixed addresses

Based on memory transfer operations:

load, store, arithmetic operations, etc.

Analysis 1: Alias computation

A forward analysis: $\sim$ copy propagations

computes a mapping MLoc $\rightarrow_a 2^{MLoc}$

$\rightarrow_a$ associates to each MLoc its set of possible (relevant) values

$$x \rightarrow_a \{v_1, \ldots v_n\} \text{ if } x = v_1 \text{ or } \ldots \text{ or } x = v_n$$

ex: (epb+12) $\rightarrow_a$ {(ebp−8), *(epb+16)}

produces a may-alias summary:

ex: return value (eax) and *arg2 are may-aliases

Analysis 2 : taint propagation

A forward analysis: $\sim$ reaching definitions

computes a mapping MLoc $\rightarrow_t \{T, U, \perp\} \cup 2^{MLoc}$

$\rightarrow_t$ associates a "taint value" to each MLoc

$(\text{epb} + 12) \rightarrow_t T, *(\text{epb} - 8) \rightarrow_t (\text{ebp} + 16)$

produces a taint summary

ex: taint of return value (eax) taint of *arg2

Outline

Tool Highlights

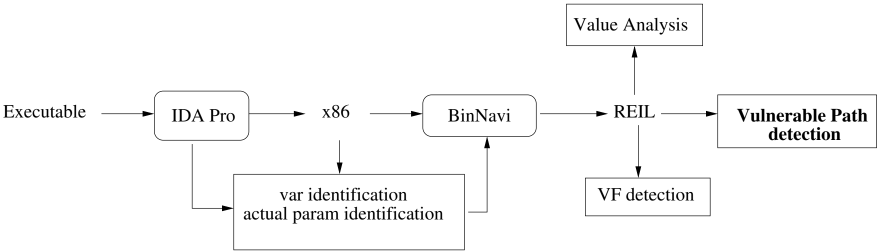
Based on two existing platforms:

- IDA Pro, a "general purpose" disassembler
- BinNavi:
  - translation to an intermediate representation (REIL)
  - a data-flow analysis engine (MonoREIL)

+ an additional set of Jython procedures

But still under construction/evaluation . . .

Tool Architecture



Some specific binary-level engineering problems

Arguments and Local Variable Recognition

- ebp or esp based access ...
- we rely on IDA Pro "semi-naive" variable recognition
- but a "lightweigth" value-analysis also available ...

Actuals to formal parameters mapping

- context sensitive analysis
  → maintain an actual-to-formal mapping at each call site.
- arguments may be PUSHed or passed through registers ...
- PUSHed instructions identified at the x86 level
- needs a specific data-flow analysis at the REIL level ...

Some specific binary-level engineering problems

Arguments and Local Variable Recognition

- ebp or esp based access ...
- we rely on IDA Pro "semi-naive" variable recognition
- but a "lightweigth" value-analysis also available ...

Actuals to formal parameters mapping

- context sensitive analysis
  $\rightarrow$ maintain an actual-to-formal mapping at each call site.
- arguments may be PUSHed or passed through registers ...
- PUSHed instructions identified at the x86 level
- needs a specific data-flow analysis at the REIL level ...

Experimental results

Name: Fox Player

Total functions: 1074

Total vulnerable functions: 48

Total slices found: 20

Smallest slice: 3 func

Largest slice: 40 func

Average func in slice: 18

$\Rightarrow$ **About 10 "vulnerable paths" discovered . . .**

Outline

Conclusions

- Identification of "vulnerable paths" between taint sources and vulnerable functions
- A scalable inter-procedural data-flow analysis
  - defined at the binary level
  - based on a Call Graph slicing
- Implementation based on IDA Pro and BinNavi
- Part of a more complete "Vulnerability Detection and Exploitability Analysis" tool chain

Future Work

- Finalize the tool and perform some realistic experiments (e.g., from existing vulnerability databases)
- Several possible improvements:
  - argument and local variable identifications
  - global variables
  - vulnerable path construction
- to be continued within the BinSec ANR project . . .