Static Taint-Analysis on Binary Executables

Sanjay Rawat, Laurent Mounier, Marie-Laure Potet

VERIMAG
University of Grenoble

October 2011

Outline

1. Introduction
2. Intra-procedural taint analysis
3. Inter-procedural taint analysis
4. Tool plateform and experimental results
5. Conclusion

Context: vulnerability analysis

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

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

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

Work objective

$\rightarrow$ Staticaly compute *vulnerable execution paths*:

- on large applications (several thousands of functions)
  scalability issues $\Rightarrow$ lightweight analysis
- from **binary executable** code
- Evaluation on existing vulnerable code
  (Firefox, Acroread, MediaPlayer, ...)

$\rightarrow$ Links with more general (test related) problems:

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

Outline

1. Introduction
2. Intra-procedural taint analysis
3. Inter-procedural taint analysis
4. Tool plateform and experimental results
5. Conclusion

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
   if (x > 0) y = 3 else y = 4 ;
// y is tainted
```

$\Rightarrow$ we focus on **data dependencies** ...

Static taint data-dependency analysis

(classical) 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 variable
  - $T \in \{Tainted, Untainted\}$ is a *taint value*
- fix-point computation (backward dependencies inside loops)

Main difficulties:

- memory aliases (pointers)
- non scalar variables (e.g., arrays)

```
read (T[i]) ; ... ; x = T[j]
```

A source-level Example

```
int x, y, z, t ;
...
read(y) ; read(t) ;
y = 3 ;
x = t ;
z = y ;
// x and t are now tainted
```

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:  Mem[t4] := t3
                         ...
...                      7:  t5 := ebp-8
z = y ;                  8:  t6 := Mem[t5]
                         9:  t7 := ebp-12
                         10: Mem[t7] := t6
```

Needs to identify that:

- value written at `ebp-8` $\leftarrow$ mem. loc. written at line 3
- content of reg. `t4` at line 2 $=$ content of reg. `t5` at line 7

$\Rightarrow$ compute **possible values** of **each** registers and mem. locations ...

Value Set Analysis (VSA)

Compute the sets of mem. addresses defined at each prog. loc.

Difficult because:

- addresses and other values are not distinguishable
- both direct and indirect memory adresssing
- address arithmetic is pervasive

Compute at each prog. loc. an over-approximation of:

- the set of (abstract) adresses that are defined
- the value contained in each register and each abstract address

$\Rightarrow$ Can be expressed as a forward data-flow analysis ...

Memory model

- Memory = (unbounded) set of fix-sized memory cells
- Memloc = (consecutive) memory cells accessed during load/store ops.

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`

However:

- the exact value of `ebp` is unknown
- the value returned by a `malloc()` is unknown
- arithmetic computations to access non-scalar variables
- set of memory locations accessed is unknown statically

Abstracting Adresses and Values

Abstract address/value =

- set of **offsets** w.r.t. register content **at a given instruction**
- expressed as a pair $< B, X >$ s.t.:

- B is an (abstract) "base value", which can be either
  - a pair (instruction, register)
  - an element of $\{\text{Empty}, \text{None}, \text{Any}\}$
- X is a finite set of integers ($X \subseteq \mathbb{Z}$).

Concrete values represented by $< B, X >$ =

$$\begin{cases} \emptyset \text{ if } B = \text{Empty} & \text{(empty value)} \\ \mathbb{Z} \text{ if } B = \text{Any} & \text{(any value)} \\ X \text{ if } B = \text{None} & \text{(constant value)} \\ \{v + x \mid x \in X \wedge v \in \text{ concrete val. of } t \text{ at } i\} \text{ if } B = (i, t) \end{cases}$$

Example

1. `t0 = ebp + 8`

$$t0 =< (1, ebp), \{8\} >$$

2. `t1 = Mem[t0]` $\{$the content of `Mem[t0]` is **unknown** ... $\}$

$$t1 =< (2, t1), \{0\} >$$

3. `t2 = t1 + 4`

$$t2 =< (2, t1), \{4\} >$$

4. `Mem[t2] = 50`

$$< (2, t1), 4 >=< \text{None}, \{50\} >$$

Intraprocedural VSA as a data-flow analysis

Mapping

$$\{ \text{ Register } \times \text{ Abstract addresses } \} \rightarrow \text{ Abstract values}$$

associated to each CFG node

Forward least-fix-point computation:

- lattice of abstract address/values (more precise $\leq$ less precise)
- widening operator (set of offsets is bounded)
- merge operator: least upper bound

$$< B1, X1 > \sqcup < B2, X2 >= \begin{cases} < \text{Any}, \emptyset > & \text{if } B1 \neq B2 \\ < B1, X1 \cup X2 > & \text{if } B1 = B2 \end{cases}$$

- transfer function: abstracts the instruction semantics ...

Example 1: conditional statement

```c
#include <stdio.h>
int main()
{
    int x, y=5, z ;
    if (y<4) {
        x=3; z=4;
    } else {
        x=4; z=3;
    } ;
    y=x+z;
    return z;
}
```

```
'ebp' = <'40105601', {0}>
'esp' = <'init', {4}>
'init-12' = <'noval', {6, 7, 8}>
'init-16' = <'noval', {3, 4}>
'init-8' = <'noval', {3, 4}>
```

Example 2: iterative statement

```c
#include <stdio.h>
int main()
{
    int x=0, i, y;
    for (i=0; i<4;i++) {
        y=6; x=x+i;
    }
    return 0;
}
```

```
'ebp' =  <'40105701', {0}>
'esp' = 'init', {4}>
'initESP-12' = <'anyval', {}>
'initESP-16' = <'noval', {6}>
'initESP-8' = <'anyval', {}>
```

Possible improvements

- use more sophisticated abstract domain ?
  (e.g., stridded intervals ?)
- restrict VSA to registers and memory locations **involved** in address computations
  Partially done ...
- take into account the **size** of memory transfers

Outline

1. Introduction
2. Intra-procedural taint analysis
3. Inter-procedural taint analysis
4. Tool plateform and experimental results
5. Conclusion

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)`

And **global variables** . . .

⇒ compute **procedure summaries** to express these dependencies.

A summary-based inter-procedural data-flow analysis

intra-procedural level: summary computation

$\rightarrow$ 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 $\Rightarrow$ 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 the Call Graph

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

→ A slice computation performed at the **call graph** level

→ Split this set into 3 parts:

1. procedure that are not relevant
2. procedure those side-effect can be (implicitely) over-approximated
   → use of default summaries ...
3. procedure requiring a more detailed analysis
   → summary computation through intra-procedural analysis

Outline

1. Introduction
2. Intra-procedural taint analysis
3. Inter-procedural taint analysis
4. Tool plateform and experimental results
5. Conclusion

Tool Highlights

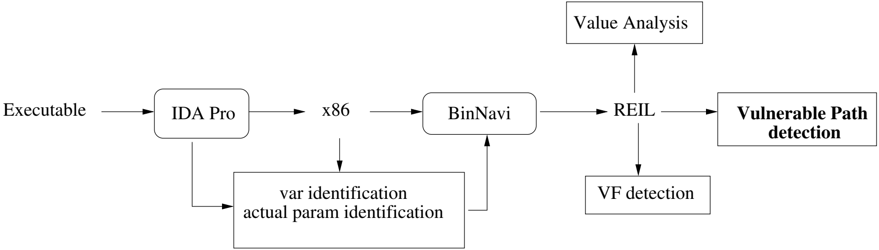
Based on two existing platforms:

- 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



Example of experimental result

Name: Fox Player

Total functions: 1074

Total vulnerable functions: 48

Total slices found: 16 (**5 with a tainted data flow**)

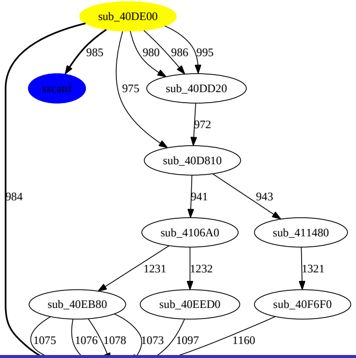
Smallest slice: 3 func

Largest slice: 40 func

Average func in slice: 18

⇒ **About 10 “vulnerable paths” discovered . . .**

Taintflow slice for Foxplayer Example



Conclusion

- Part of a more complete tool chain for "Vulnerability Detection and Exploitability Analysis"
- To be continued within the BinSec ANR project . . .