Static Code Analysis

Lecture 9
Oct 8, 2014

Source:
“Secure Programming with Static Analysis”
Static Analysis

- Analyzing code before executing it
  - Analogy: Spell checker
- Suited to problem identification because
  - Checks thoroughly and consistently
  - Can point to the root cause of the problem
    - E.g., presence of buffer overflow; helps to focus on what to fix
  - Help find errors/bugs early in the development
    - Helps reduce cost
  - New information can be easily incorporated to recheck a given program
Usefulness

- Better than manual code review
- Faster and more concrete than testing
- Consistency in coverage
- Embody the existing security knowledge and gets extended
- Great for use by non-experts
Key Issues

- Can give a lot of noise!
- False Positives & False Negative
  - Which is worse? Need to balance the FP and FN
- Defects must be visible to the tool
- Different types of Static analysis:
  - Type checking; Style checking
  - Program understanding; Program verification
  - Property checking; Bug finding
  - Security Review

*It is Computationally undecidable problem*
Type Checking

Example 2.1 A type-checking false positive: These Java statements do not meet type safety rules even though they are logically correct.

```java
10  short s = 0;
11  int i = s;    /* the type checker allows this */
12  short r = i;  /* false positive: this will cause a type checking error at compile time */
```

Example 2.2 Output from the Java compiler demonstrating the type-checking false positive.

```
$ javac bar.java
bar.java:12: possible loss of precision
  found : int
  required: short
    short r = i;    /* false positive: this will cause a type checking error at compile time */

1 error
```

Example 2.3 These Java statements meet type-checking rules but will fail at runtime.

```java
Object[] objs = new String[1];
objs[0] = new Object();
```

```java
! x
```
Style Checking

- Superficial set of rules
- Focused on rules related to
  - Whitespace, naming, deprecated functions, commenting, program structure
  - Affect: readability and maintainability rather than coding error
  - `-Wall in gcc`
    - Detect when a switch statement does not account for all possible values
  - For large project many people with their own style may be involved
  - Examples: lint, PMD
Program Understanding

● Helps make sense of a large Codebase

● Examples
  ● Tool example: Fujaba
    ▪ UML and Java Code – can help back and forth
      ▪ “Finding all uses of a method”
      ▪ “Finding declaration of a global variable”

● Helpful to work on code one has not written
  ● some reverse engineer the design – “big picture”

● IDEs typically include some
Program verification and Property checking

- Accepts a specification and associated Code
  - Aims to prove that the code is faithful implementation
  - “equivalence checking” to check the two match
- Complete specific consuming!
  - So “Partial” verification”
  - Try to find a “counterexample”
- Sound wrt the spec
  - It will always return a problem if one exists!
    - (false negative? False positive?)
    - Soundness may be very difficult to establish

Memory leak

```c
1 inBuf = (char*) malloc(bufSz);
2 if (inBuf == NULL)
3   return -1;
4 outBuf = (char*) malloc(bufSz);
5 if (outBuf == NULL)
6   return -1;
```

Violation of property "allocated memory should always be freed":
- line 2: inBuf != NULL
- line 5: outBuf == NULL
- line 6: function returns (-1) without freeing inBuf

Counter example
Bug Finding

- Points out places where the program will behave in a way that the coder did not intend
  - Use patterns that indicate bugs
  - Example: FindBug (Java), Coverity (C, C++)
- Early tools: ITS4, RATS, Flawfinder
  - Little more than glorified “grep”
  - Closer to style checkers
- Modern tools
  - Typically hybrid of property checkers and bug finders
Factors for utility of SA

- Ability of the tool to make sense of the program
- Trade-offs it makes between precision and scalability
- Errors that it can check/detect
- How easily usable by programmers/users
Some examples

<table>
<thead>
<tr>
<th>Type of Tool/Vendors</th>
<th>Web Site</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Style Checking</strong></td>
<td></td>
</tr>
<tr>
<td>PMD</td>
<td><a href="http://pmd.sourceforge.net">http://pmd.sourceforge.net</a></td>
</tr>
<tr>
<td>Parasoft</td>
<td><a href="http://www.parasoft.com">http://www.parasoft.com</a></td>
</tr>
<tr>
<td><strong>Program Understanding</strong></td>
<td></td>
</tr>
<tr>
<td>Fujaba</td>
<td><a href="http://wwwcs.uni-paderborn.de/cs/fujaba/">http://wwwcs.uni-paderborn.de/cs/fujaba/</a></td>
</tr>
<tr>
<td>CAST</td>
<td><a href="http://www.castsoftware.com">http://www.castsoftware.com</a></td>
</tr>
<tr>
<td><strong>Program Verification</strong></td>
<td></td>
</tr>
<tr>
<td>Praxis High Integrity Systems</td>
<td><a href="http://www.praxis-his.com">http://www.praxis-his.com</a></td>
</tr>
<tr>
<td>Escher Technologies</td>
<td><a href="http://www.eschertech.com">http://www.eschertech.com</a></td>
</tr>
<tr>
<td><strong>Property Checking</strong></td>
<td></td>
</tr>
<tr>
<td>Polyspace</td>
<td><a href="http://www.polyspace.com">http://www.polyspace.com</a></td>
</tr>
<tr>
<td>Grammatech</td>
<td><a href="http://www.grammatech.com">http://www.grammatech.com</a></td>
</tr>
<tr>
<td><strong>Bug Finding</strong></td>
<td></td>
</tr>
<tr>
<td>FindBugs</td>
<td><a href="http://www.findbugs.org">http://www.findbugs.org</a></td>
</tr>
<tr>
<td>Coverity</td>
<td><a href="http://www.coverity.com">http://www.coverity.com</a></td>
</tr>
<tr>
<td>Visual Studio 2005 \analyze</td>
<td><a href="http://msdn.microsoft.com/vstudio/">http://msdn.microsoft.com/vstudio/</a></td>
</tr>
<tr>
<td>Klocwork</td>
<td><a href="http://www.klocwork.com">http://www.klocwork.com</a></td>
</tr>
<tr>
<td><strong>Security Review</strong></td>
<td></td>
</tr>
<tr>
<td>Fortify Software</td>
<td><a href="http://www.fortify.com">http://www.fortify.com</a></td>
</tr>
<tr>
<td>Ounce Labs</td>
<td><a href="http://www.ouncelabs.com">http://www.ouncelabs.com</a></td>
</tr>
</tbody>
</table>
Analyzing Source vs Compiled

- Static analysis can examine a program
  - As a compiler sees it (Source code) OR
  - As a run-time env sees it (in some cases – bytecode or executable)

- Advantages of compile code analysis
  - No need to guess how compiler will interpret
  - Source code may be not available

- Disadvantages
  - Making sense is more difficult (e.g., may lack type info)
SA in Code Review
SA Metrics

- Metrics helps
  - Prioritizing remedial efforts
  - Estimating risk associated with code (tricky!)
    - False positive/negative – manual inspection needed
    - No way to sum/aggregate risks from flaws

- Some metrics for tactical focus
  - Measuring vulnerability density
    - #results/LOC – maybe deceptive
  - Comparing projects by severity
  - Breaking down results by category
  - Monitoring trends
SA Metrics

- Comparing modules based on severity
- Breaking down by categories
SA Internals

- A Generic SA Tool
Building a model

- Create a program model from code
  - A set of data structures
  - Depends on the type of analysis that a tool performs

- SA - Closer to compiler
  - Lexical analysis – e.g., regular expression for tokens
  - Parsing – uses a context free grammars
    - Set of production rules
    - Parse tree: Lex and Yacc
Parsing

- Can have nonterminal symbols
  - Syntactic sugar!
- Can perform analysis on Parse Tree – can be inconvenient
  - Directly from grammar

```plaintext
stmt := if_stmt | assign_stmt
if_stmt := IF LPAREN expr RPAREN stmt
expr := lval
assign_stmt := lval EQUAL expr SEMI
lval := ID | arr_access
arr_access := ID arr_index+
arr_idx := LBRACKET expr RBRACKET
```
Abstract Syntax Tree

- Does away with the details of grammar and syntactic sugar
  - Create a standard version of program
  - Lowering (e.g., loops may be converted to while loop)
Semantic Analysis & Control Flow

- Semantic analysis based on: AST + Symbol table
  - Type checking can be done
  - Semantic analysis – symbol resolution and type checking
  - Optimization or *intermediate* forms may be created

- Tracking Control Flow
  - Different execution paths need to be explored
  - Build a control flow graph on top of AST
Control Flow Graph

- **Trace**: sequence of blocks that define a path
  - E.g., bb0, bb1, bb3
Call graph

- Call graph – control flow between functions

```c
int larry(int fish) {
    if (fish) {
        moe(1);
    } else {
        curly();
    }
}

int moe(int scissors) {
    if (scissors) {
        curly();
        moe(0);
    } else {
        curly();
    }
}

int curly() {
    /* empty */
}
```

Function pointer & Virtual functions complicate things

Dynamically loaded modules make it further challenging

Data flow & data type analysis may be needed

Call graph may be incomplete
Dataflow

- Analyzes how data move through the program.
  - Helps compilers optimize!
- Traverse functions control flow graph
  - Where data values are generated & where used
  - Convert a function to *static single assignment* form (SSA)
    - SSA: allows assigning a value to a variable only once
      - New variables may need to be added
    - SSA variable can have a constant (use that to replace future variable places)
    - SSA variable may have different values along different control paths – need to be reconciled
      - Merge point: \( \varphi \)-function
SSA Examples

Regular source code form:

```
sum = sum + delta;
sum = sum & top;
y = y + (z<<4)+k[0] ^ z+sum ^ (z>>5)+k[1];
y = y & top;
z = z + (y<<4)+k[2] ^ y+sum ^ (y>>5)+k[3];
z = z & top;
```

SSA form:

```
sum_2 = sum_1 + delta_1;
sum_3 = sum_2 & top_1;
y_2 = y_1 + (z_1<<4)+k[0]_1 ^ z_1+sum_3 ^ (z_1>>5)+k[1]_1;
y_3 = y_2 & top_1;
z_2 = z_1 + (y_3<<4)+k[2]_1 ^ y_3+sum_3 ^ (y_3>>5)+k[3]_1;
z_3 = z_2 & top_1;
```

Regular source form:

```
if (bytesRead < 8) {
    tail = (byte) bytesRead;
}
```

SSA form:

```
if (bytesRead_1 < 8) {
    tail_2 = (byte) bytesRead_1;
}
tail_3 = \phi(tail_1, tail_2);
```
Taint Propagation

● It is important
  ● to identify which values in a program an attacker could potentially control/target
    ● Need to know where values enter and how they move
      ▪ E.g., Buffer overflow vulnerability
  ● Taint propagation algorithm
    ● Key to identifying many input validation and representation defects
    ● Static as well as dynamic taint propagation analysis
Pointer Aliasing

- Several pointers may refer to the same memory
  - \( *p1 = 1 \)
  - \( *p2 = 2 \)

Can \( p1 \) and \( p2 \) refer to the same location?
Can these be reordered?

For the following, compiler should understand that input data flows to process input:

\[
p1 = p2;
*p1 = getUserInput();
processInput(*p2);
\]
SA Algorithms

- Local component and global component
  - Improve context sensitivity

![Diagram: Analysis Algorithm]

- Intraprocedural analysis component for analyzing an individual function
- Interprocedural analysis component for analyzing an individual function
Assertions

- Many properties can be specified as assertions – which need to be true

  Example: Buffer Overflow prevention check
  ```c
  strcpy(dest, src);
  ```

  Add assertion
  ```c
  assert(alloc_size(dest) > strlen(src));
  ```

- If there are conditions under which an assertion can fail – report potential overflow
Assertions

- Typically three varieties of assertions
  - Taint propagation problems
    - When programmers trust input when they should not – so SA should check data values moving
    - data is tainted (controlled by an attacker) or not
  - Range Analysis
    - To Identify buffer overflow – need to know the size of the buffer and the data value
      - Understand the range of values data or size may have
  - Type state: concern about the state of an object as execution proceeds
Naïve Local Analysis (informal)

Consider:

\[
\begin{align*}
&x = 1; \\
y = 1; \\
&\text{assert}(x < y);
\end{align*}
\]

- Maintain facts before each statement is executed:

\[
\begin{align*}
&x = 1; \quad \{ \text{no facts} \} \\
y = 1; \quad \{ x = 1 \} \\
&\text{assert}(x < y); \quad \{ x = 1, y = 1 \}
\end{align*}
\]

- Always false!! SA should report a problem.

Symbolic Simulation:

\[
\begin{align*}
&x = v; \\
y = v; \\
&\text{assert}(x < y); \\
\text{Same Result}
\end{align*}
\]
Conditional makes it complex!

\[
x = v;
\]
\[
\text{if } (x < y) \{ \\
  \quad y = v; \\
  \}
\]
\[
\quad \text{assert } (x < y);
\]

\[
x = v; \\
\text{if } (x < y) \{ \\
  \quad y = v; \\
  \}
\]
\[
\quad \text{assert } (x < y);
\]

\[
x = v; \\
\text{if } (x < y) \{ \\
  \quad y = v; \\
  \}
\]
\[
\quad \text{assert } (x < y);
\]

\[
\text{assert } (x < y); \\
\text{if } (x < y) \{ \\
  \quad y = v; \\
  \}
\]
\[
\quad \text{assert } (x < y);
\]

\[
ex < y \text{ is FALSE}
\]

\[
\text{Need to check the conjunction of assertion predicate and all the facts:}\]
\[
(x < y) \land (x = v) \land \neg(x < y)
\]
\[
\text{Again fails!}
\]
Conditional makes it complex! Loops add further ..

- The previous approach is problematic
- \#paths grows with the number of conditionals
  - Share info among common subpaths
  - *Program slicing* – to remove code cannot effect the outcome of the assert predicate
  - Also eliminate *false paths* – logically inconsistent paths that will never be executed
- Adding loops makes it even more complex!
Approaches to Local Analysis

- Abstract interpretation
  - Abstract away aspects of the program that are not relevant to properties of interest and perform and interpretation
- Loop problems – do flow-insensitive analysis
  - Tries to guarantee that all statement orderings are considered (not follow the program statement order)
    - No need for control flow analysis
    - But some useless execution order may be performed as well
- More practical tools – partially flow sensitive!
Predicate Transformers

- Use the weakest precondition
  - Fewest set of requirements on the callers of a program that are necessary to arrive at a desired final state or post condition

E.g.,

\[(x < 0 \land y > 0)\] is a strong requirement than \[(x < y)\];
Model Checking Approach

- Accepts properties as specifications, transforms the program to be check into an automaton (called the model)
  - Now compare the specification to the model
  - Example: “memory should be freed only once”

Model checking will look for a variable wrt which system will reach state error
Global Analysis

- **Context-sensitive analysis**
  - Takes into account the context of the calling function

- **Whole-program analysis**
  - Tries to analyze every function with a complete understanding of the context of its calling functions
  - One way is "inlining" (Recursion will be problem)
  - Time consuming and very ambitious

- **More flexible approach**
  - Local analysis generates the *function summaries*

```c
memcpy(dest, src, len) [  
  requires:  
    ( alloc_size(dest) >= len ) ∧ ( alloc_size(src) >= len )  
  ensures:  
    ∀ i ∈ 0 .. len-1: dest[i]' == src[i]  
]```

- **Example**
Rules

- Good SA tools externalize the rules they check
  - Added, removed, altered easily

RATS will report a violation of the rule whenever it sees a call to `system()` where the first argument is not constant.

In some cases rules are annotated within the program (in JML)
Rules for Taint Propagation

• Variety of rule types to accommodate different taint propagation problems
  • **Source rules** define program locations where tainted data enter the system.
    • Functions named `read()` often introduce taint in an obvious manner; others: `getenv()`, `getpass()`, `gets()`.
  • **Sink rules** define program locations that should not receive tainted data.
    • For SQL injection in Java, `Statement.executeQuery()` is a sink.
    • For buffer overflow in C, assigning to an array is a sink, as is the function `strcpy()`
Rules for Taint Propagation

- **Pass-through rules** define the way a function manipulates tainted data.
  - E.g., a pass-through rule for the `java.lang.String` method `trim()` might explain “if a String s is tainted, the return value from calling s.trim() is similarly tainted.”

- **Cleanse rule** is a form of pass-through rule that removes taint from a variable.
  - represents input validation functions.

- **Entry-point rules** (similar to source)-
  - they introduce taint into the program, entry-point functions are invoked by an attacker.
  - E.g., `main()` is an entry point (java, C)
Example: Command injection vulnerability

```c
if ( fgets ( buf , sizeof(buf), stdin) == buf ) {
    strcpy ( othr , buf );
    system ( othr );
}
```

1. A source rule for `fgets()` taints `buf` `othr`
2. Dataflow analysis connects uses of `buf`
3. A pass-through rule for `strcpy` taints
4. Dataflow analysis connects uses of `othr`
5. Because `othr` is tainted, a sink rule for `system()` reports a command injection vulnerability
Taints

- Essentially BINARY attribute
  - But can have taint flags to indicate variety of tainted data – can help prioritize!
    - FROM_NETWORK data from network
    - FROM_CONFIGURATION data from config file
  - Sing functions may be dangerous for a specific taint type

- Taint propagation rules include various elements
  - Method or function
  - Precondition
  - Postcondition
  - Severity