Secure Programming Lecture 14: Static Analysis II

David Aspinall

8th March 2016

Recap

We're looking at

- **principles and tools**

for ensuring software security.

This lecture looks at:

- further **example uses** of static analysis
- some hints about **how static analysis works**

Advanced static analysis jobs

Static analysis is used for a range of tasks that are useful for ensuring secure code.

Basic tasks include **type checking** and **style checking**, described last lecture.

More advanced tasks are:

- **Program understanding**: inferring meaning
- **Property checking**: ensuring no bad behaviour
- **Program verification**: ensuring correct behaviour
- **Bug finding**: detecting likely errors

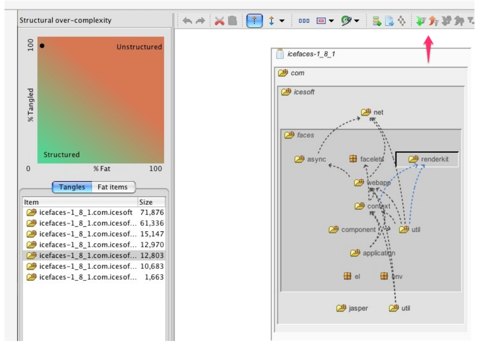
Program understanding tools

Help developers understand and manipulate large codebases.

- Navigation swiftly inside the code
  - finding definition of a constant
  - finding call graph for a method
- Support *refactoring* operations
  - re-naming functions or constants
  - move functions from one module to another
  - needs internal model of whole code base
- Inferring *design* from *code*
  - Reverse engineer or check informal design

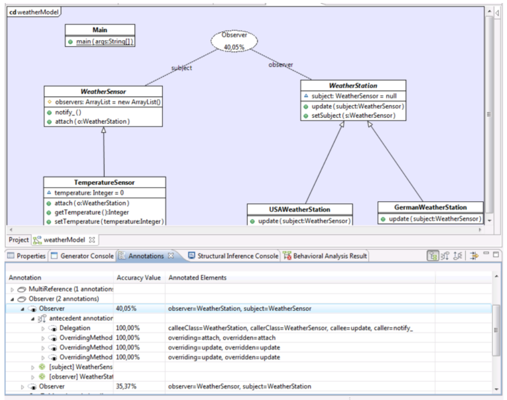
Outlook: may become increasingly used for security review, with dedicated tools. Close relation to tools used for malware analysis (reverse engineering).

Commercial example: Structure101



Item	Size
icefaces-1_8_1.com.icesoft	71,876
icefaces-1_8_1.com.icesof...	61,336
icefaces-1_8_1.com.icesof...	15,147
icefaces-1_8_1.com.icesof...	12,970
icefaces-1_8_1.com.icesof...	12,803
icefaces-1_8_1.com.icesof...	10,683
icefaces-1_8_1.com.icesof...	1,663

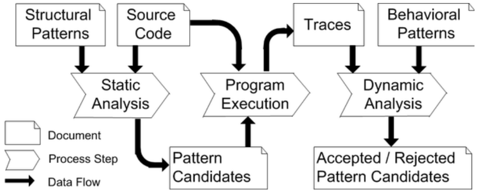
Research example: Fujaba and Reclipse



cd weatherModel
Main
main (args:String[])
Observer
40,05%
subject
observer
WeatherSensor
observers: ArrayList = new ArrayList()
notify_ ()
attach (o:WeatherStation)
WeatherStation
subject: WeatherSensor = null
update (subject:WeatherSensor)
setSubject (s:WeatherSensor)
TemperatureSensor
temperature: Integer = 0
attach (o:WeatherStation)
getTemperature ( ):Integer
setTemperature (temperature:Integer)
USAWeatherStation
update (subject:WeatherSensor)
GermanWeatherStation
update (subject:WeatherSensor)
Project
weatherModel
Properties
Generator Console
Annotations
Structural Inference Console
Behavioral Analysis Result

Annotation	Accuracy Value	Annotated Elements
MultiReference (1 annotation:		
Observer (2 annotations)		
Observer	40,05%	observer=WeatherStation, subject=WeatherSensor
antecedent annotatior		
Delegation	100,00%	calleeClass=WeatherStation, callerClass=WeatherSensor, callee=update, caller=notify_
OverridingMethod	100,00%	overriding=attach, overridden=attach
OverridingMethod	100,00%	overriding=update, overridden=update
OverridingMethod	100,00%	overriding=update, overridden=update
[subject] WeatherSens		
[observer] WeatherStat		
Observer	35,37%	observer=WeatherSensor, subject=WeatherStation

How Reclipse works



We'll explain some of these processes later.

See Fujaba project at University of Paderborn

Program verification

- The gold standard, ultimate guarantee
- Uses **formal methods** techniques, e.g.,
  - theorem proving
  - model checking
- Drawback: needs precise **formal specification** to verify against
- Very expensive to industry
  - time consuming
  - needs experts (logic/maths)
- Currently only used in safety critical domains
  - e.g., railway, nuclear, aeronautics
  - emerging: automobile, *security*

Example tools: SPARK, Event-B. See also general purpose **interactive theorem provers**. Many other research-quality and/or unmaintained tools.

Property checking

***Lightweight* formal methods**

- Make specifications be *standard* and *generic*
- this program cannot raise NullPointerException
- all database connections are closed after use

Static *checking* (not verification)

- Prevent many violations of specification, not all
- May produce *counterexamples* to explain violations
- Chain pre-conditions (`requires`) and post-conditions (`ensures`)
  - allows *inter-procedural* analysis

Examples: Code Contracts, Splint, JML, Grammatech CodeSonar, PolySpace, ThreadSafe, PRQA, Facebook Infer.

Assertion checking

Many languages have support for *assertions*.

These are dynamic (runtime) checks that can be used to test properties the programmer expects to be true.

`assert(exp)`

- fails if exp evaluates to false
- assertion tests **usually disabled**
  - treated as comments
  - may be enabled for testing during development
  - or when running unit tests

Question. What is the risk with running tests only with assertions enabled?

Assertions in Java

```
private static int addHeights(int ah, int bh) {
    assert ah > 0 && bh > 0 : "parameters should be positive";
    return ah+bh;
}
```

pause

Notice above method is private.

- API (public) functions should *always* test constraints
  - throw exceptions if not met
  - eliminate clients (or attackers) who break API contract
- Internal functions may rely on local properties
  - if maintained in same class, easier to check/ensure

Assertions for security

We could use assertions as safety checks for functions that are at risk of being used in a buggy way.

```
assert(alloc_size(dest) > strlen(src));
strcpy(dest, src);
```

[alloc_size() is not a standard C function, but GCC, for example, has support for trying to track the size of allocated functions with function attributes]

From dynamic to static

With static analysis, we *may* be able to automatically determine whether assertions (if enabled) will:

1. always succeed
2. may sometimes fail (unknown)
3. will always fail

Easy cases:

1. `assert(true);`
2. `x=readint(); assert(x>0);`
3. `assert(false);`

The perfect case would be showing that assertions in a program can only succeed: thus they do not need to be checked dynamically.

Question. what troubles can you see with case 2?

Reasoning with assertions

How does a static analyser reason?

Computations about assertions can be chained through the program, using a *program logic* inside the tool.

E.g., build up a set of facts known before each statement:

```
                              //  { }   (nothing known)
x = 1;                        //  { x = 1 }
y = 1;                        //  { x = 1, y = 1 }
assert (x < y);               //  FAIL
```

Symbolic evaluation

This can work also with variables, whose value is not known statically:

```
                              //  { }   (nothing known)
x = z;                        //  { x = z }
y = z+1;                      //  { x = z, y = z+1 }
assert (x < y);               //  SUCCEED  (provided no z<MAXINT)
```

Conditionals and loops

These make static analysis *much* harder, of course.

```
                              // {}    (nothing known)
x = v;                        // {x=v}
if (x < y)                    //
   y = v;                     // {x=v, x<y}
assert (x < y)                // Either: {x=v,y=v}: FAIL
                              // Or: {x=v,¬(x<y)}: FAIL
```

For conditionals, we need to either

- explore every path
- merge information at *join-points*

For loops, we need to either

- unroll for a finite number of iterations
- capture variation using logical *invariants*

Security assertions

Using logical (or other) reasoning techniques, there are various different types of assertions that are useful for security checking, for example:

- **Bounds and range analysis**
- **Tainted data analysis**
- **Type state** and **Resource** tracking

Exercise. What kinds of security issues can these assertions help with? What kinds of security issues would need other assertions?

Bound/range Analysis

alloc_size(dest)>strlen(src)

array_size(a)>n before a[n] access

- Check integers are in required ranges

Taintedness

tainted(mypageinput)

untainted(newkey)

- Tracks whether data can be affected by adversary.
- Tainted input shouldn't be used for security sensitive choices
- and should be sanitized before being output
- Taint analysis approximates information flow
  - information may be leaked *indirectly* as well as directly

Type State (Resource) Tracking

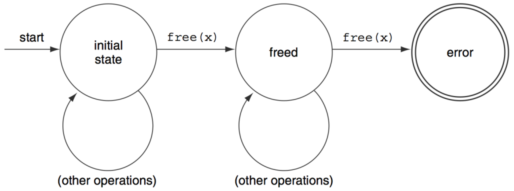
isnull(ptr), **nonull**(ptr)

isopen_for_read(handle), **isclosed**(handle)

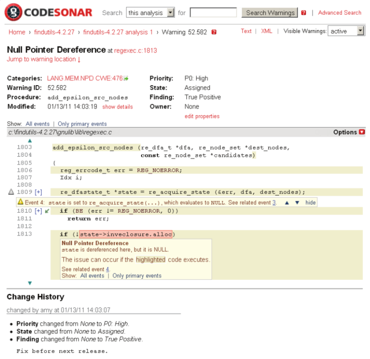
uninitialized(buffer), **terminatedstring**(buffer)

- Tracks status of data value held by a variable
- Helps enforce API usage contracts to avoid errors
  - e.g., DoS
- Usage/lifecycle may be expressed with automaton

Example: avoiding double-free errors



Null Pointers in CodeSonar



CODESONAR
Search
this analysis
for
Search Warnings
Advanced Search
Home › findutils-4.2.27 › findutils-4.2.27 analysis 1 › Warning 52.582
Text | XML | Visible Warnings: active

Null Pointer Dereference at regexec.c:1813

Jump to warning location ↓

Categories:	LANG.MEM.NPD CWE:476	**Priority:**	P0: High
Warning ID:	52.582	**State:**	Assigned
Procedure:	add_epsilon_src_nodes	**Finding:**	True Positive
Modified:	01/13/11 14:03:19 show details	**Owner:**	None
			edit properties

Show: All events | Only primary events

c:\findutils-4.2.27\gnulib\lib\regexec.c Options

```
1803     add_epsilon_src_nodes (re_dfa_t *dfa, re_node_set *dest_nodes,
1804                             const re_node_set *candidates)
1805     {
1806       reg_errcode_t err = REG_NOERROR;
1807       Idx i;
1808
1809 [+]   re_dfastate_t *state = re_acquire_state (&err, dfa, dest_nodes);
```

Event 4: state is set to re_acquire_state(...), which evaluates to NULL. See related event 3. ▲ ▼ hide

```
1810 [+]   if (BE (err != REG_NOERROR, 0))
1811         return err;
1812
1813       if (!state->inveclosure.alloc)
```

Null Pointer Dereference
state is dereferenced here, but it is NULL.

The issue can occur if the highlighted code executes.

See related event 4.
Show: All events | Only primary events

Change History

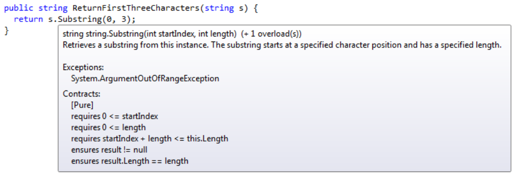
changed by amy at 01/13/11 14:03:07

- **Priority** changed from *None* to *P0: High*.
- **State** changed from *None* to *Assigned*.
- **Finding** changed from *None* to *True Positive*.

```
Fix before next release.
```

Not all null pointer analyses are equal! Some compilers spot only "obvious" null pointer risks, others perform deeper analysis like CodeSonar. IDE analysis may be in between.

Code Contracts in .NET



```
public string ReturnFirstThreeCharacters(string s) {
  return s.Substring(0, 3);
}
```

string string.Substring(int startIndex, int length) (+ 1 overload(s))
Retrieves a substring from this instance. The substring starts at a specified character position and has a specified length.

Exceptions:
System.ArgumentOutOfRangeException

Contracts:
[Pure]
requires 0 <= startIndex
requires 0 <= length
requires startIndex + length <= this.Length
ensures result != null
ensures result.Length == length

For Java, there is a language called JML which adds similar pre- and post-conditions (requires/ensures). Open source JML toolsets have been through several versions but have had trouble keeping up with Java, Eclipse changes.

Bug finding

Bug finding tools look for suspicious patterns in code.

FindBugs is an example:

- Finds possible Java bugs according to *rules*
  - rules are suspicious patterns in code
  - designed by experience of buggy programs
  - . . . collected from real world and student(!) code
- Warnings are categorized by
  - **severity**: how serious in general the problem is
  - **confidence**: tool's belief of true problem

Example bugs

Common accidents

An error found in Sun's JDK 1.6:

```
public String foundType() {
    return this.foundType();
}
```

Misunderstood APIs

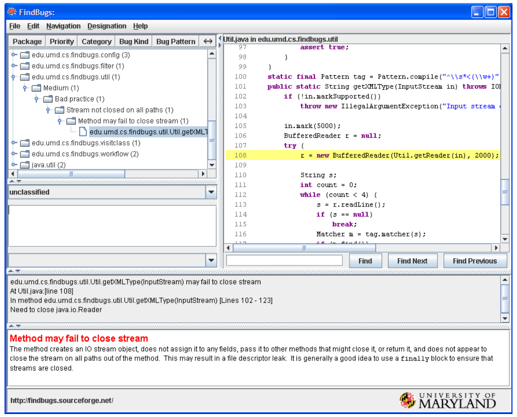
```
public String makeUserid(String s) {
   s.toLowerCase();
   return s;
}
```

Anti-idiom: double-checked locking in Java

```
if (this.fitz == null) {
    synchronized (mylock) {
       if (this.fitz == null) {
           this.fitz = new Fitzer();
       }
    }
}
```

```
[dice]da: findbugs Fitz.class
M M DC: Possible doublecheck on Fizz.fitz in Fitz.getFitz()
        At Fitz.java:[lines 1-3]
```

Findbugs GUI



FindBugs:
File Edit Navigation Designation Help
Package Priority Category Bug Kind Bug Pattern
Find
Find Next
Find Previous

- edu.umd.cs.findbugs.config (3)
- edu.umd.cs.findbugs.filter (1)
- edu.umd.cs.findbugs.util (1)
  - Medium (1)
    - Bad practice (1)
      - Stream not closed on all paths (1)
        - Method may fail to close stream (1)
          - edu.umd.cs.findbugs.util.Util.getXMLT
- edu.umd.cs.findbugs.visitclass (1)
- edu.umd.cs.findbugs.workflow (2)
- java.util (2)

unclassified

Util.java in edu.umd.cs.findbugs.util

```
 97                 assert true;
 98             }
 99         }
100     static final Pattern tag = Pattern.compile("^\\s*<(\\w+)"
101     public static String getXMLType(InputStream in) throws IO
102         if (!in.markSupported())
103             throw new IllegalArgumentException("Input stream
104 
105         in.mark(5000);
106         BufferedReader r = null;
107         try {
108             r = new BufferedReader(Util.getReader(in), 2000);
109 
110             String s;
111             int count = 0;
112             while (count < 4) {
113                 s = r.readLine();
114                 if (s == null)
115                     break;
116                 Matcher m = tag.matcher(s);
```

```
edu.umd.cs.findbugs.util.Util.getXMLType(InputStream) may fail to close stream
At Util.java:[line 108]
In method edu.umd.cs.findbugs.util.Util.getXMLType(InputStream) [Lines 102 - 123]
Need to close java.io.Reader
```

Method may fail to close stream

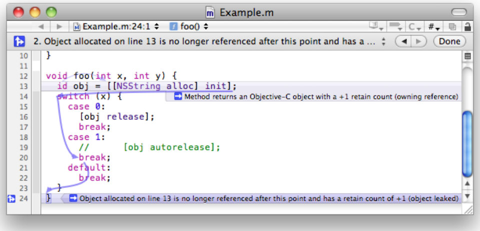
The method creates an IO stream object, does not assign it to any fields, pass it to other methods that might close it, or return it, and does not appear to close the stream on all paths out of the method. This may result in a file descriptor leak. It is generally a good idea to use a `finally` block to ensure that streams are closed.

http://findbugs.sourceforge.net/

UNIVERSITY OF MARYLAND

Clang Static Analyser

An open source tool for C, C++, Objective-C included in XCode.



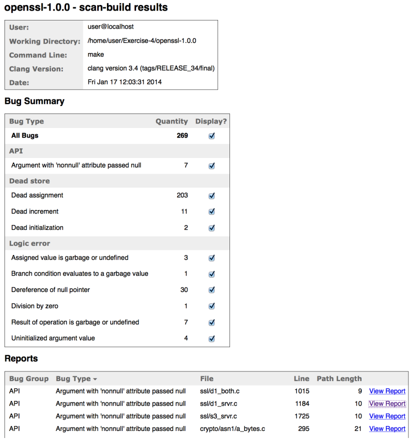
2. Object allocated on line 13 is no longer referenced after this point and has a ...

```
10  }
11 
12  void foo(int x, int y) {
13    id obj = [[NSString alloc] init];
14    switch (x) {
15      case 0:
16        [obj release];
17        break;
18      case 1:
19        //      [obj autorelease];
20        break;
21      default:
22        break;
23    }
24  }
```

Method returns an Objective-C object with a +1 retain count (owning reference)

Object allocated on line 13 is no longer referenced after this point and has a retain count of +1 (object leaked)

Clang Static Analyser HTML reports



openssl-1.0.0 - scan-build results

User:	user@localhost
Working Directory:	/home/user/Exercise-4/openssl-1.0.0
Command Line:	make
Clang Version:	clang version 3.4 (tags/RELEASE_34/final)
Date:	Fri Jan 17 12:03:31 2014

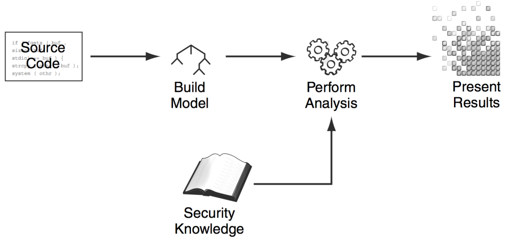
Bug Summary

Bug Type	Quantity	Display?
All Bugs	**269**	☑
API		
Argument with 'nonnull' attribute passed null	7	☑
Dead store		
Dead assignment	203	☑
Dead increment	11	☑
Dead initialization	2	☑
Logic error		
Assigned value is garbage or undefined	3	☑
Branch condition evaluates to a garbage value	1	☑
Dereference of null pointer	30	☑
Division by zero	1	☑
Result of operation is garbage or undefined	7	☑
Uninitialized argument value	4	☑

Reports

Bug Group	Bug Type ▾	File	Line	Path Length	
API	Argument with 'nonnull' attribute passed null	ssl/d1_both.c	1015	9	View Report
API	Argument with 'nonnull' attribute passed null	ssl/d1_srvr.c	1184	10	View Report
API	Argument with 'nonnull' attribute passed null	ssl/s3_srvr.c	1725	10	View Report
API	Argument with 'nonnull' attribute passed null	crypto/asn1/a_bytes.c	295	21	View Report

Basic overview



Building a program model

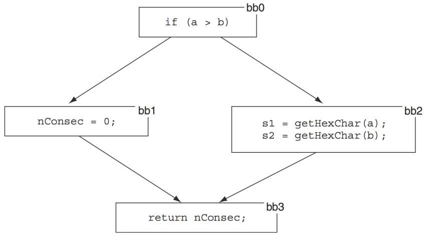
Starts off like a compiler, in stages. Simpler/older static analysis tools only use first stages.

1. **Lexical analysis**: tokenise input
2. **Parsing**: builds a *parse tree* from grammar
3. **Abstract Syntax Tree**: simplify parse tree
4. **Semantic analysis**
   - check program well-formedness
   - including **type-checking**
5. Produce an **Intermediate Representation** (IR)
   - higher level than for compiler
6. Produce **model** to capture control/data flows
   - *control-flow* and *call graphs*
   - variable-contains-data relationships
   - pointer analysis: aliasing, points-to

Control flow graphs

```
if (a > b) {
  nConsec = 0;
} else {
  s1 = getHexChar(1);
  s2 = getHexChar(2);
}
return nConsec;
```

The CFG consists of *basic blocks* and the paths between them.

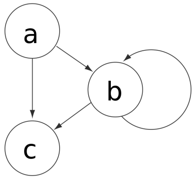


- A *trace* is a possible sequence of basic blocks.
- Above: [bb0,bb1,bb3] and [bb0,bb2,bb3].

Traces can be used to check against security constraints (e.g., as automata), to construct counterexamples. The CFG is also used to combine/chain assertions.

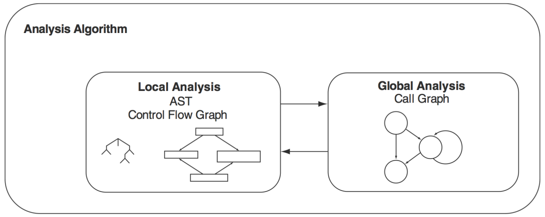
Call graphs

```
int a(int x) {
  if (x) {  b(1);  } else  { c(); }
}
int b(int y) {
  if (y) { c(); b(0); } else { c(); }
}
int c() { /* empty */ }
```



- Call graphs are used for *inter-procedural* analysis
- Check requires-ensures contracts connect together

Putting them together: local and global



Take away points

Static analysis tools can help find security flaws.

Massive benefits:

- examine millions of lines of code, repeatedly

Some tools are generic bug finding, built into IDE.

Others are specific to security, may include.

- risk analysis, including impact/likelihood
- issue/requirements tracking
- metrics

Expect these (gradually?) to become mainstream

- current frequency of security errors unacceptable
- incentives will eventually affect priorities

References and credits

Some of this lecture is based Chapters 2-4 of

- *Secure Programming With Static Analysis* by Brian Chess and Jacob West, Addison-Wesley 2007.

Recommended reading:

- Al Bessey et al. *A few billion lines of code later: using static analysis to find bugs in the real world*, CACM 53(2), 20101.