Choosing the Right Software Assurance Tools

Paul E. Black
paul.black@nist.gov

http://samate.nist.gov/
Outline

- Types of Software Assurance (SA) Tools
- Considerations and Variants
- Adding SA Tools to Your Process
- Using the CWE-121 Effectiveness Set
- More Resources from the SAMATE Reference Dataset (SRD)
What Goes Into Assurance?

- Mature Process
- Study, Review, and Test the Software
- Assurance in IT Systems
  \[ A = f(p, s, e) \]
- Resilient Execution Environment
Two Kinds of Analysis: Static and Dynamic

**Static Analysis**
- Code review
- Binary, byte, or source code scanners
- Model checkers & property proofs
- Assurance case

**Dynamic Analysis**
- Execute code
- Simulate design
- Fuzzing, coverage, MC/DC, use cases
- Penetration testing
- Field tests
## Static and Dynamic Analysis Complement Each Other

### Static Analysis
- Handles unfinished code
- Higher level artifacts
- Can find backdoors, e.g., full access for user name “JoshuaCaleb”
- Potentially complete

### Dynamic Analysis
- Code not needed, e.g., embedded systems
- Has few(er) assumptions
- Covers end-to-end or system tests
- Assess as-installed
Different Static Analyzers Exist For Different Purposes

- To check intellectual property violation
- For developers to decide what needs to be fixed (and learn better practices)
- For auditors or reviewer to decide if it is good enough for use
Outline

- Types of Software Assurance (SA) Tools
- Considerations and Variants
- Adding SA Tools to Your Process
- Using the CWE-121 Effectiveness Set
- More Resources from the SAMATE Reference Dataset (SRD)
Consideration: Rates

- False alarm rate
- Miss rate (recall)
- Precision
- Discrimination
Tools don’t report the same flaws

Overlap in Not-False Buffer Errors
Consideration: Subject

- What level?
  - Design, Requirements, Source code, Byte code, or Binary

- Language(s) handled

- Compiler extensions

- Platform

- Speed, scalability, max program size
Consideration: Properties

- Analysis can look for anything from general or universal properties:
  - don’t crash
  - don’t overflow buffers
- to application-specific properties:
  - log the date and source of every message
  - cleartext transmission
  - user cannot execute administrator functions
- Can I write my own “rules”?
Consideration: Level of Rigor

- **Syntactic**
  - flag every use of `strcpy()`

- **Heuristic**
  - every `open()` has a `close()`, every `lock()` has an `unlock()`

- **Analytic**
  - data flow, control flow, constraint propagation

- **Fully formal**
  - theorem proving
Consideration: Human Involvement

- analyst aides and tools
  - call graphs
  - property prover
- human-aided analysis
  - annotations
- completely automatic
  - scanners
Consideration: Output Format (1)

    char sys[512] = "/usr/bin/cat ";
    25    gets(buff);
    strcat(sys, buff);
    30    system(sys);

foo.c:30:Critical:Unvalidated string 'sys' is received from an external function through a call to 'gets' at line 25. This can be run as command line through call to 'system' at line 30. User input can be used to cause arbitrary command execution on the host system. Check strings for length and content when used for command execution.
<table>
<thead>
<tr>
<th>Problem</th>
<th>Line</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter rawlog_save</td>
<td>/u1/paul/SATE/2010/c/rssil/rssl-0.8.14/src/core/rawlog.c</td>
<td></td>
</tr>
<tr>
<td>void rawlog_save(RAWLOG_REC *rawlog, const char *fname)</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td>char *path;</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>int f;</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>path = convert_home(fname);</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>f = open(path, O_WRONLY</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>O_APPEND</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>O_CREAT, log_file_create_mode);</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>g_free(path);</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>rawlog_dump(rawlog, f);</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>Enter rawlog_save / rawlog_dump</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>static void rawlog_dump(RAWLOG_REC *rawlog, int f)</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>GSList *tmp;</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>for (tmp = rawlog-&gt;lines; tmp != NULL; tmp = tmp-&gt;next) { /* Null Pointer Dereference */</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>f &lt;= -1</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>write(f, tmp-&gt;data, strlen((char <em>) tmp-&gt;data)); /</em> Negative file descriptor */</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>Exit rawlog_save / rawlog_dump</td>
<td>109</td>
<td></td>
</tr>
</tbody>
</table>
Consideration: Output Format (3)

- Standard findings interchange format, e.g., SAFES or TOIF
Consideration: Tool Integration

- Eclipse, Visual Studio, etc.
- Penetration testing
- Execution monitoring
- Bug tracking
Consideration: Non-Functional

- **Cost**
  - per seat, or per line of code

- **View issues by**
  - Category
  - File or Package
  - Priority

- **New issues since last scan**

- **Are issues increasing or decreasing?**

- **Which modules are hot spots?**
The report explains
- use of the Juliet test suite
- the 14 weakness classes covered
- automated run and scoring
- measures: precision, recall, discrimination, etc.
- graphs and tables to understand result

It does not evaluate specific tools.
Outline

- Types of Software Assurance (SA) Tools
- Considerations and Variants
- Adding SA Tools to Your Process
- Using the CWE-121 Effectiveness Set
- More Resources from the SAMATE Reference Dataset (SRD)
Set up “consulting” group

- They have time to learn the tool, customize it for a project’s need, run it, and interpret results.
- Gradually withdraw support, e.g. longer turn around, less face-to-face effort – natural as consultants help other projects.
Start with one class of flaw

- Choose the class that is most critical or is easiest to catch.

- Add other flaw classes as value is demonstrated.
Only Look at New Code

- Ignore warnings from existing code
  - it already runs, doesn’t it?

- Require that any brand new code needs to be “clean” – either code changed to avoid warnings or explicit justification.

- Then include code that is modified.
Increasingly Require Over Time

- At first, the only requirement is that every developer *had* a static analyzer.
- Then required that it be run.
- Then standardize on one or two that developers found beneficial.
- Then require that warnings be reported.
- Then require that warnings be addressed (fixed or dismissed).
Survivor effect in software

Mistakes that matter

Mistakes that don’t matter

Unit Test

System Integration

Field Reports

after Bill Pugh
SATE workshop
Nov 2009
Late automated analysis is hard

Mistakes that matter

Mistakes that don’t matter

Unit Test

System Integration

Field Reports

Automated Static Analysis

after Bill Pugh
SATE workshop
Nov 2009
Automated analysis best at start

Mistakes that matter

Automated Static Analysis

Unit Test

System Integration

Field Reports

Mistakes that don’t matter

after Bill Pugh
SATE workshop
Nov 2009
When is survivor effect weak?

- If testing or deployment isn’t good at detecting problems
  - True for many security and concurrency problems
- If faults don’t generate clear failures
  - Also true for many security problems
Outline

- Types of Software Assurance (SA) Tools
- Considerations and Variants
- Adding SA Tools to Your Process
- Using the CWE-121 Effectiveness Set
- More Resources from the SAMATE Reference Dataset (SRD)
MITRE’s CWE Compatibility and Effectiveness Program

- Phase 1 – Declare compatibility
- Phase 2 – Verify mapping to CWEs
- Phase 3 – Test cases show effectiveness
  - tool effectively locates CWEs
  - tool deals with code complexities

http://cwe.mitre.org/compatible/program.html
What is “Code Complexity”?

```c
char data;

data = 'C';
data = 'Z';
printHexCharLine(data);
```

```c
if (1) {
    data = 'C';
    printHexCharLine(data);
} else {
    data = 'C';
    printHexCharLine(data);
}

if (1) {
    data = 'Z';
    printHexCharLine(data);
} else {
    printHexCharLine(data);
}
```

CWE-563 Unused Variable, after SRD test cases 35455 and 35456
What is content like?

- Each CWE has one or more tests
  - short (this is not about handling megacode)
  - code is vulnerable, i.e., exploitable
  - (usually) synthetic
  - fairly “clean”, but not necessarily pristine; meet SRD “accepted” standard
  - standard code; no language extensions
- Test cases have corresponding “fixed” cases, to prodive data on false positives
As a Proof-of-Concept

- We started with CWE-121 Stack-based Buffer Overflow (in C language)
  - CWE-121 is a frequent, serious problem.
  - It is well-defined and easily understood.
  - We have thousands of examples.
  - It is addressed by static analysis, compile-time techniques, or run-time detection.
Background Work

- Over the summer NIST researchers installed five static analyzers, then examined 7,338 in 9,962 files from
  - Juliet (split into 5,892 good & bad cases)
  - Kratkiewicz (1,139 cases)
  - KDMA TCG (249 cases)
  - 2005 Fortify (41 cases)
  - other SRD (17 cases)
Proposed CWE-121 Basic Set

- It consists of five cases.
- The most basic case is basic-00001-min.c
  ```c
  char buf[10];
  buf[10] = 'A';
  ```
  - This is so trivial it never occurs in real code.
- We added four more cases as simple variants.
Other Basic Cases

- basic-00034-min.c
  - access through a pointer
- basic-00045-min.c
  - use strcpy()
- basic-00182-min.c
  - fgets(): limited copy and external input
- stack_overflow_loop.c
  - loop initializes array, but bad bounds check
Next Step – Complexity Cases

- Cases related to SATE or Lippman
- Other fns: str(n)cpy/cat, memcpy/move, s(n)printf
- Separate files (caseA.c & caseB.c)
- Duplicate function names
- Dynamic allocation - alloca()
- Array indexing - see Kratkiewicz
- Data Types
- Buffer in struct
- Dead (infeasible) code
- Open coded or obfuscated str(n)cpy()
- Cases with a difference between I/J/M or min/med
What about tool “short cuts”?

- Tool makers may build to a public, static set.
  - A secret or dynamic set has other problems.
- Change comments and identifier names for every download?
- Add innocuous statements?
- Transform code, like unroll loops?

Proposal:
- If concerns arise, privately corroborate results.
Outline

- Types of Software Assurance (SA) Tools
- Considerations and Variants
- Adding SA Tools to Your Process
- Using the CWE-121 Effectiveness Set

- More Resources from the SAMATE Reference Dataset (SRD)
SAMATE Reference Dataset

- Public repository for software assurance test cases
- Over 60,000 cases in C, C++, Java, C#, and Python
- Search and select by language, weakness, etc.
- Contributions from CAS, Fortify, Defence R&D Canada, Klocwork, MIT Lincoln Laboratory, Praxis, Secure Software, etc.
Juliet 1.1 cases

- 23,957 cases in Java and 57,099 in C/C++ covering 181 weaknesses
- Each case is a page or two of code, sometimes crossing multiple files
- Most cases include similar unflawed code
- Organized by weakness, then variant, then complexity
- Described in IEEE Computer, Oct 2012

STONESOUP cases

- About 460 cases in Java and C, each a program typically 200-300 lines long
- Cover weaknesses in Number Handling (e.g. integer overflow), Tainted Data (e.g. input validation), Injection (e.g. command injection, Buffer Overflow, and Null Pointer
- Each case has inputs triggering the vulnerability, as well as “safe” inputs
- Available about November 2012
Kratkiewicz MIT cases

- 1164 cases in C for CWE-121 Stack-Based Buffer Overflow
- Created to investigate static analysis and dynamic detection methods
- Each case is one of four variants:
  - access within bounds (ok)
  - access just outside bound (min)
  - somewhat outside bound (med)
  - far outside bound (large)
- Code complexities: index, type, control, ...
Other SRD Content

- Zitser, Lippmann, & Leek MIT cases
  - 28 slices from BIND, Sendmail, WU-FTP, etc.
- Fortify benchmark 112 C and Java cases
- Klocwork benchmark 40 C cases
- 25 cases from Defence R&D Canada
- Robert Seacord, “Secure Coding in C and C++” 69 cases
- Comprehensive, Lightweight Application Security Process (CLASP) 25 cases
- 329 cases from our static analyzer suite