Engineering Secure Software

WHITE BOX TESTING & INSPECTIONS

The Power of Source Code

- White box testing
  - Testers have intimate knowledge of the specifications, design,
  - Often done by the original developers
  - Security consultants often get source code access too
- Code Inspections
  - aka “Technical Reviews”, “Code Reviews”
  - Stepping through code with security in mind
- Test & inspect the threats
  - Enumerated by abuse cases, threat models, architectural risks
  - Defensive coding concerns
- Testing → Failure-finding technique
  Inspection → Fault-finding technique

What to test for?

- Test & inspect the security mitigations
  - Bug in your mitigation → vulnerability
  - Bug in your security feature → vulnerability
- Test for both types of vulnerabilities
  - Low-level coding mistakes
  - High-level design flaws
- Test at every scope
  - Unit, integration, system
  - Try to keep an equal effort emphasis on each
  - Unit tests → bugs in mitigations & features
  - Integration → interaction vulnerabilities

Who's at the code inspection?

- Author
  - Made significant contributions to the code recently
  - Can answer any specific questions, or reveal blind spots
- People with *readability*, but *objectivity*
  - e.g. close colleagues
  - e.g. developer working on a similar feature on the same project, but different team
  - e.g. system architect
- People experienced with security
  - Consultants, if any
  - Developers on previous vulnerabilities in this system

Make an Inspection Checklist

- What will you talk about?
  - Keep a running checklist for the meeting
  - Adapt the checklist for future inspection meetings
- At the meeting, briefly identify the following that are pertinent to this code
  - Assets from risk analysis
  - Threats from your threat models
  - Malicious actors from your requirements
  - Abuse and misuse cases from your requirements
- Walk through the functionality of the code
  - Look for *missing* code more than wrong code
  - “If they missed this, then they probably missed that”

More for the checklist

- Look for too much complexity
  - Both structural and cognitive complexity
  - Too much responsibility in one place
- Look for common defensive coding mistakes
- Look for opportunities to build security into the design
  - e.g. repeated input validation? Make input validation the default
  - e.g. file canonicalization is done all in one place
  - e.g. using a third-party library

The Prioritization Problem

- What should we inspect?
  - Can’t inspect everything
  - Reacting to inspections can take some time
  - Can be too repetitive
- Inspect what probably has vulnerabilities
- Three approaches:
  - Code coverage – what have we not tested?
  - Static analysis – what tools say is vulnerable
  - Prediction – what history says is vulnerable

Code Coverage

- What has been executed as a result of our tests?
  - e.g. have exceptions been tested?
  - e.g. have we tested this input?
- Use a tool to record what code has been executed
  - Levels: package, class, line, branch
  - 80% is a common threshold for line coverage
- Benefits
  - Reveals what testers forgot
  - Relatively simple to deploy and execute
- Disadvantages
  - Unit test coverage != well-tested
    (add system tests to your coverage!)
  - Test coverage != security test coverage

eclEMMA

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
CollabCloud	62.9 %	7747	4568	12315
src/main/java	61.6 %	2971	1855	4826
org.chaoticbits.collabcloud	100.0 %	158	0	158
org.chaoticbits.collabcloud.visualizer.colc	100.0 %	31	0	31
org.chaoticbits.collabcloud.visualizer.spir	100.0 %	243	0	243
org.chaoticbits.collabcloud.visualizer.avat	96.1 %	222	9	231
org.chaoticbits.collabcloud.codeprocessc	72.4 %	71	27	98
org.chaoticbits.collabcloud.visualizer.font	76.7 %	224	68	292
org.chaoticbits.collabcloud.vc.svn	40.3 %	116	172	288
org.chaoticbits.collabcloud.vc	66.0 %	380	196	576

```
109                     while (scanner.hasNextLine()) { // scan until the diff part
110                         String line = scanner.nextLine();
111                         if (diffParser.isFile(line)) {
112                             currentSummarizable = diffParser.makeSummarizable(line
113                         }
114                         if (currentSummarizable != null)
115                             diffParser.processTextLine(line, weights, contribution
116                     }
117                 } catch (IOException e) {
118                     System.err.println("IO Exception on commit " + commit.getId().
119                     e.printStackTrace();
120                 }
121             }
122         }
123         loaded = true;
```

Automated Static Analysis

- Static analysis
  - Analyzing code without executing it
  - Manual static analysis == code inspection
  - Think: sophisticated compiler warnings
- Automated Static Analysis tools
  - Provide warnings of common coding mistakes
  - Use a variety of methods
    - Fancy grep searches
    - Symbolic execution & model checking
    - Data flow analysis
- Tools
  - Non-security: FindBugs, PMD
  - Security: Fortify, Coverity, JTest

ASA Benefits & Drawbacks

- Benefits
  - Quick and easy
  - Knowledge transfer from experts behind the tool
  - Provides a specific context in the code to drive the discussion
- Drawbacks
  - Huge false positive rates: >90% are FP in many cases
  - Fault-finding → exploitable?
  - Biased to code-level vulnerabilities
  - Cannot possibly identify domain-specific risks
  - Better for inspections than tests

Prediction-Based Prioritization

- Vulnerabilities are rare

  Typically, about 1% to 5% of source code files will require a post-release patch for a vulnerability

- Prediction is possible
  - Good metrics
  - Trained machine-learning models

- Many metrics are correlated with vulnerabilities
  - Files with *previous vulnerabilities*
  - Files with high *code churn*
  - Files committed to by *many developers*

    e.g. 10+ developers coordinating on a single file? Improbable.
  - Large files (==high cyclomatic complexity)