Secure Programming with Static Analysis

Jacob West
jacob@fortify.com
Software Systems that are

• Ubiquitous
• Connected
• Dependable

Complexity

Unforeseen Consequences
The line between secure/insecure is often subtle
Many seemingly non-security decisions affect security
Small problems can hurt a lot
Smart people make dumb mistakes
As a group, programmers tend to make the same security mistakes over and over
We need non-experts to get security right
Success is foreseeing failure.

– Henry Petroski
Non-functional Security Failures

Generic Mistakes
- Input validation
- Memory safety (buffer overflow)
- Handling errors and exceptions
- Maintaining privacy

Common Software Varieties
- Web applications
- Network services / SOA
- Privileged programs
MSDN sample code for function DirSpec:

```c
int main(int argc, char *argv[]) {
    ...
    char DirSpec[MAX_PATH + 1];
    printf("Target dir is %s.\n", argv[1]);
    strncpy(DirSpec, argv[1], strlen(argv[1])+1);
    ...
}
```
Cross-Site Scripting

We never intended the code that's in there to actually be production-ready code”
- Ryan Asleson

```xml
<c:if test="${param.sayHello}">
  Hello ${param.name}!
</c:if>
```
Reliving Past Mistakes

- Cross-site scripting looks more and more like buffer overflow

**Buffer Overflow**
- Allows arbitrary code execution
- **Exploit is hard to write**
- Easy mistake to make in C/C++
- Well known problem for decades

**Cross-site Scripting**
- Allows arbitrary code execution
- **Exploit is easy to write**
- Easy mistake to make
- Well known problem for a decade
<table>
<thead>
<tr>
<th>Wrong Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Try Harder</strong></td>
</tr>
<tr>
<td>• Our people are smart and work hard.</td>
</tr>
<tr>
<td>• Just tell them to stop making mistakes.</td>
</tr>
<tr>
<td>• Not everyone is going to be a security expert.</td>
</tr>
<tr>
<td>• Getting security right requires feedback.</td>
</tr>
<tr>
<td><strong>Fix It Later</strong></td>
</tr>
<tr>
<td>• Code as usual.</td>
</tr>
<tr>
<td>• Build a better firewall (app firewall, intrusion detection, etc.)</td>
</tr>
<tr>
<td>• More walls don’t help when the software is meant to communicate.</td>
</tr>
<tr>
<td>• Security team can’t keep up.</td>
</tr>
<tr>
<td><strong>Test Your Way Out</strong></td>
</tr>
<tr>
<td>• Do a penetration test on the final version.</td>
</tr>
<tr>
<td>• Scramble to patch findings.</td>
</tr>
<tr>
<td>• Pen testing is good for demonstrating the problem.</td>
</tr>
<tr>
<td>• Doesn’t work for the same reason you can’t test quality in.</td>
</tr>
</tbody>
</table>
Security in the Development Lifecycle

- Plan
- Build
- Test
  - Firewalls
  - Intrusion Detection
  - Penetration Testing
- Field
Security in the Development Lifecycle

- Plan
  - Risk Assessment
  - Code Review
  - Security Testing

- Build

- Test

- Field

Effective security from non-experts
Overview

- Introduction
- Static Analysis: The Big Picture
- Inside a Static Analysis Tool
- Static Analysis in Practice
- What Next?
- Parting Thoughts
Static Analysis: The Big Picture
Static Analysis Defined

- Analyze code without executing it
- Able to contemplate many more possibilities than you could execute with conventional testing
- Doesn’t know what your code is supposed to do
- Must be told what to look for
The Many Faces of Static Analysis

- Type checking
- Style checking
- Program understanding
- Program verification / Property checking
- Bug finding
- Security review
Why Static Analysis is Good for Security

- Fast compared to manual code review
- Fast compared to testing
- Complete, consistent coverage
- Brings security knowledge with it
- Makes review process easier for non-experts
Prehistoric Static Analysis Tools

RATS

Flawfinder

ITS4
Prehistoric Static Analysis Tools

Glorified grep

(+) Good
- Help security experts audit code
- A place to collect info about bad coding practices

(-) Bad
- NOT BUG FINDERS
- Not helpful without security expertise
int main(int argc, char* argv[]) {
  char buf1[1024];
  char buf2[1024];
  char* shortString = "a short string";
  strcpy(buf1, shortString); /* eh. */
  strcpy(buf2, argv[0]);    /* !!! */
...
What You Won’t Find

- Architecture errors
  - Microscope vs. telescope
- Bugs you’re not looking for
  - Bug categories must be predefined
- System administration mistakes
- User mistakes
Security vs. Quality

- Bug finding tools focus on high confidence results
  - Bugs are cheap (plentiful)
  - Bug patterns, bug idioms
  - False alarms are killers
- Security tools focus on high risk results
  - More human input required
  - The bugs you miss are the killers
Inside a Static Analysis Tool
Critical Attributes

- Language support
  - Understands the relevant languages/dialects

- Analysis algorithms
  - Uses the right techniques to find and prioritize issues

- Capacity
  - Ability to gulp down millions of lines of code

- Rule set
  - Modeling rules, security properties

- Results management
  - Allow human to review results
  - Prioritization of issues
  - Control over what to report
Building a Model

- Front end looks a lot like a compiler
- Language support
  - One language/compiler is straightforward
  - Lots of combinations is harder
- Could analyze compiled code...
  - Everybody has the binary
  - No need to guess how the compiler works
  - No need for rules
- ...but
  - Decompilation can be difficult
  - Loss of context hurts. A lot.
  - Remediation requires mapping back to source anyway
Analysis Techniques

- **Taint propagation**
  - Trace potentially tainted data through the program
  - Report locations where an attacker could take advantage of a vulnerable function or construct

```plaintext
buff = getInPutFromNetwork();
copyBuffer(newBuff, buff);
exec(newBuff);  // (command injection)
```

- Many other approaches, no one right answer
Capacity: Scope vs. Performance

The diagram illustrates the relationship between execution time and analysis scope for various tools. The x-axis represents the analysis scope ranging from Line to Program, and the y-axis represents execution time ranging from Blink of an eye to Overnight. The graph shows the relative performance and scope of tools such as ITS4, Flawfinder, FindBugs, Klocwork, Ounce, Coverity, and Fortify.
Only Two Ways to Go Wrong

- False positives
  - Incomplete/inaccurate model
  - Conservative analysis

- False negatives
  - Incomplete/inaccurate model
  - Missing rules
  - “Forgiving” analysis

The tool that cried “wolf!”

Missing a detail can kill.
Rules: Dataflow

- Specify
  - Security properties
  - Behavior of library code

```java
buff = getInputFromNetwork();
copyBuffer(newBuff, buff);
exec(newBuff);
```

- Three rules to detect the command injection vulnerability
  1) `getInputFromNetwork()` postcondition:
     return value is tainted
  2) `copyBuffer(arg1, arg2)` postcondition:
     arg1 array values set to arg2 array values
  3) `exec(arg)` precondition:
     arg must not be tainted
Rules: Control Flow

- Look for dangerous sequences
- Example: Double-free vulnerability

```c
while ((node = *ref) != NULL) {
    *ref = node->next;
    free(node);
    if (!unchain(ref)) {
        break;
    }
}
if (node != 0) {
    free(node);
    return UNCHAIN_FAIL;
}
```
Rules: Control Flow

- Look for dangerous sequences
- Example: Double-free vulnerability

```c
while ((node = *ref) != NULL) {
    *ref = node->next;
    free(node);
    if (!unchain(ref)) {
        break;
    }
}
if (node != 0) {
    free(node);
    return UNCHAIN_FAIL;
}
```
Rules: Control Flow

- Look for dangerous sequences
- Example: Double-free vulnerability

```c
while ((node = *ref) != NULL) {
    *ref = node->next;
    free(node);
    if (!unchain(ref)) {
        break;
    }
}

if (node != 0) {
    free(node);
    return UNCHAIN_FAIL;
}
```
Displaying Results

- Must convince programmer that there’s a bug in the code
- Different interfaces for different scenarios:
  - Security auditor parachutes in to 2M LOC
  - Programmer reviews own code
  - Programmers share code review responsibilities
- Interface is just as important as analysis
- Don’t show same bad result twice
- Try this at home: Java Open Review
  http://opensource.fortify.com
Static Analysis in Practice
Two Ways to Use the Tools

- Analyze completed programs
  - Fancy penetration test. Bleah.
  - Results can be overwhelming
  - Most people have to start here
  - Good motivator

- Analyze as you write code
  - Run as part of build
  - Nightly/weekly/milestone
  - Fix as you go
### Typical Objections and Their True Meanings

<table>
<thead>
<tr>
<th>Objection</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>“It takes too long to run.”</td>
<td>“I think security is optional, so I don’t want to do it.”</td>
</tr>
<tr>
<td>“It has too many false positives.”</td>
<td>“I think security is optional, so I don’t want to do it.”</td>
</tr>
<tr>
<td>“It doesn’t fit with the way I work.”</td>
<td>“I think security is optional, so I don’t want to do it.”</td>
</tr>
</tbody>
</table>
Adopting a Static Analysis Tool

1) Some culture change required
   - More than just another tool
   - Often carries the banner for software security program
   - Pitfall: the tool doesn’t solve the problem by itself

2) Define the playing field
   - Choose specific objectives
   - Build a gate

3) Do training up front
   - Software security training is paramount
   - Tool training is helpful too
Adopting a Static Analysis Tool

4) Start small
   - Do a pilot rollout to a friendly dev group
   - Build on your success

5) Go for the throat
   - Tools detect lots of stuff. Turn most of it off.
   - Focus on easy-to-understand, highly relevant problems.

6) Appoint a champion
   - Make sure there is a point person on the dev team
   - Choose a developer who knows a little about everything
7) Measure the outcome
   - Keep track of tool findings
   - Keep track of outcome (issues fixed)

8) Make it your own
   - Investigate customization
   - Map tool against internal security standards.
   - Best case scenario is cyclic:
     - The tool reinforces coding guidelines
     - Coding guidelines are written with automated checking in mind

9) The first time around is the worst
   - Budget 2x typical cycle cost
   - Typical numbers: 10% of time for security,
     20% for the first time
Metrics

- **Defect Density → Vulnerability Density**
- **NOT A GOOD RISK BAROMETER**
- Good for answering questions such as
  - Which bugs do we write most often?
  - How much remediation effort is required?
What Next?
Seven Pernicious Kingdoms

- Catalog, define, and categorize common mistakes
- http://www.fortify.com/vulncat

- Input validation and representation
- API abuse
- Security features
- Time and state
- Error handling
- Code quality
- Encapsulation
- Environment
Security Testing

- Most widely used security testing techniques are about controllability
  - Fuzzing (random input)
  - Shooting dirty data (input that often causes trouble)
- A different take: improve observability
  - Instrument code to observe runtime behavior: **Fortify Tracer**
- Benefits
  - Security-oriented code coverage
  - Vastly improved error reporting
  - Finds more bugs
- Uses rule set from static analysis tool!
Detecting Attacks at Runtime

- If you can find bugs, can you fix them?
- Instrument program, watch it run: **Fortify Defender**
- More context than external systems
- Flexible response: log, block, etc
- Low performance overhead is a must
- Potential to detect misuse in addition to bugs
Parting Thoughts
Design

Algorithms

Conventions

Data Structures

Protocols

Platform

Language

Libraries

<Your Code>
The Buck Stops With Your Code

- **Security problems everywhere you look**
  - Languages, libraries, frameworks, etc.

- **Right answer**
  - Better languages, libraries, frameworks, etc.

- **Realistic answer**
  - Build secure programs out of insecure pieces
Mistakes happen. Plan for them.

Security is now part of programming

For code auditors: tools make code review efficient

For programmers: tools bring security expertise

Critical components of a good tool:
  - Algorithm
  - Rules
  - Interface
  - Adoption Plan
Brian Chess
brian@fortify.com

Jacob West
jacob@fortify.com