Software Security Process & Testing

a primer for quality professionals

Fred Pinkett
VP, Product Management
Security Innovation
About Security Innovation

**Application Security Experts**
- 10+ years research on vulnerabilities
- Security testing methodology adopted by SAP, Symantec, Microsoft, and McAfee
- Authors of 8 books

**Products & Services**
- **STANDARDS**: best practices adoption
- **TRAINING**: eLearning & instructor-led
- **ASSESSMENT**: software and SDLC

**Reducing Application Security Risk**
- Uncover critical vulnerabilities
- Roll out a secure, repeatable SDLC
- Build internal competency
Reducing Application Security Risk at the Source

3 Pillars of success for a secure SDLC

1. Standards & Policies
   Create security requirements for your team (insource or outsource)
   - Align development activities with policies, compliance mandates, requirements
   - Roll out secure development best practices with TeamMentor™

2. Education
   Give your teams the knowledge to succeed
   - Technical and awareness courses
   - OWASP, PCI, Mobile, Web 2.0, .NET, Java, C/C++, C#, PHP, Database

3. Assessment (Application/SDLC)
   Audit your team against standards and policies
   - For all problems identified, we provide remediation recommendations
   - Results drive policy, standards, education and tools usage improvements
Application Security Fault Model

Understanding Application Behavior

Intended Behavior (the “spec”)

Actual Behavior (as-implemented/as-spec’d)

Traditional Bugs (not to “spec”)

Most Security Bugs
Understanding the Roots of Software Vulnerabilities

• Developers have an implicit trust in the user
  – Often think of functionality (practical) rather than security
  – Who would ever sit on the keyboard and press enter?

• Applications let users be abusers
  – Extension of trust, lack of segregation

• Lots of misinformation out there
  – Mostly by non-practitioners and others that don’t understand functionality trade-offs, ship/release pressures, risk management, etc.

• Vulnerabilities are unintended functionality
  – How do you look for and prevent something that you don’t know exists?

• Developers are not incented on security

• Security is tough when you don’t know what you’re doing!
Common Application Security Process Mistakes

• Using Vulnerability Count
  – Need to consider Impact, Severity, Exploitability

• Early Tool adoption without people and process changes

• Looking at the application in isolation
  – Need to at as-deployed environment and black box test

• Using security testing as the catch-all
  – You cannot test security in any more than you can test quality in

• Neglecting vulnerability analysis
  – What and where are the common vulnerabilities and why?
Respondents by company size (point-in-time data)

- SAST in place
- Web Vuln Scan
- AppSec training
- 3rd Party Reviews
- Tool before Training

- SMB <500
- SME 500-2500
- >2500
Security Investments in the Wrong Place
## Secure Software Development Key Activities

<table>
<thead>
<tr>
<th>Core</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Planning</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Security Objectives</strong></td>
</tr>
<tr>
<td><strong>Requirements</strong></td>
<td></td>
</tr>
<tr>
<td>and Analysis</td>
<td><strong>Design Guidelines for Security</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Threat Modeling</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Arch and Design Review for Security</strong></td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td></td>
</tr>
<tr>
<td>and Design</td>
<td><strong>Design Guidelines</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Architecture and Design Review</strong></td>
</tr>
<tr>
<td><strong>Development</strong></td>
<td><strong>Code Review for Security</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Code Review for Security</strong></td>
</tr>
<tr>
<td><strong>Testing</strong></td>
<td><strong>Security Testing</strong></td>
</tr>
<tr>
<td><strong>Deployment</strong></td>
<td><strong>Deployment Review for Security</strong></td>
</tr>
<tr>
<td><strong>Maintenance</strong></td>
<td></td>
</tr>
</tbody>
</table>

*Image credit: Security Innovation*
Secure Development Best Practices

• **Integrate security into your lifecycle**
  – Upfront security design, secure coding practices, and testing for security must all be an integral part of your application development processes

• **Know your threats**
  – Analyze your application in a structured and systematic way to recognize its threats and vulnerabilities

• **Use an iterative approach**
  – Some activities should be performed multiple times during the development process in order to maximize application security

• **Do not rely too heavily on testing to improve security**
  – Should be no more than a backstop to check your progress based on all the other activities you’ve performed
Best Practices for Integrated Security

• **Adopt a secure development framework**
  – Helps integrate security into SDL, e.g., MS SDL
  – Helps integrate Risk Management framework with application activities and assets (or liabilities, depending on how you look at apps)

• **Provide Awareness and Technical Training**
  – Development teams want to do the right thing
  – Development teams want to build secure applications
  – Development teams want to follow corporate policies: show them how!
  – Don’t forget senior management – biggest roadblock, or enabler

• **Understand how applications functions in the environment**
  – Business impact analysis (see earlier slide)
  – Technical/component diagrams to aid threat modeling
  – Mitigate or accept what you cannot control
Best Low Cost Process Improvements

• **Attack Surface Analysis and Reduction**
  – Real measure of risk in your application
  – Fewer “doors” (entry points) generally means lower risk

• **Threat Modeling**
  – Identifies the most critical threats and risks
  – Based on understanding an asset’s value and cost of exploitation
  – Used to make more informed decisions and resource allocation

Both activities are powerful components of reducing application risk for the user
Reducing your Application’s Attack Surface

*Software Risk Mitigation out of the gate*

- Helps with vulnerabilities you don’t know about and makes it harder for hackers to exploit

- **Software exposes assets via entry points**
  - user interfaces
  - web services
  - direct database access
  - network channels, pipes, files, APIs …

- **The entire collection of entry points in a product is known as it’s Attack Surface**
  - comprises of all the areas where an application may be exploited
  - not limited to local resources; channels used to communicate with remote resources are vectors for attack

- **Big Attack Surface = Big Security Work = Big Security Problems**
Attack Surface & Entry Points

• **To attack an application, attackers will**
  – Identify entry points
  – Exploit the entry point or the backend exposed behind it
  – Try to deny legitimate users access to these entry points

• **In our software, we have to**
  – Understand the attack surface
  – Reduce it as much as possible
  – Test against it for exposed vulnerabilities

• **Mapping our Attack Surface**
  – Architecture and design docs, code...
  – But how many entry points are undocumented
    • Temporary files? Pipes? Network communication? Registry keys? Embeddable code?
Reducing your Attack Surface – Best Practices

• Reduce the amount of running code
• Reduce access to entry points by untrusted users
  – restrict access to network endpoints used by your application to the local subnet or IP range
  – limit access to network endpoints using authentication.
• Reduce privilege to limit damage potential
• Raise authorization bar on anonymous code paths
• Watch out for those little protocols
  – Private Communications Transport (PCT), User Datagram Protocol (UDP), etc.
• Reduce Attack Surface Early
  – when designing your application, you should have a section in the design outlining what the attack surface will look like.
• Measure Attack Surface Often
To know your Enemy, you must become your Enemy.

-Sun Tzu
How Threat Modeling Saved My Life
What is Threat Modeling?

- Security analysis technique to identify threats and visualize risk to an application or system

- **How it works**
  - Define a set of attacks and/or negative scenarios
  - Assess probability, harm, priority, and business impact of each threat
  - Use the model to make better decisions throughout development

- **Perform at any stage during the development process**
  - Most commonly performed at design where it is most effective

- **Who’s it for?**
  - Technical Teams: Software Development & IT
  - Business Teams: C-level & Risk Managers (CISO, CRO, CIO, CSO)

"Experience shows that nearly 50% of security flaws will be discovered from Threat Modeling because it finds different threats than those found through code review"

-Michael Howard, author of "Writing Secure Code" and Security Program Manager, Microsoft
Threat Modeling – Activity Overview

1. Identify Security Objectives

2. Application Overview

3. Decompose Application

4. Identify Threats

5. Identify Vulnerabilities

You can add more detail as you move through your application development lifecycle and discover more about your application design.
Threat Modeling Best Practices

- **Enumerate assets of value**
  - What assets matter to you as a business?
  - What assets may matter to an attacker?

- **Consider threats**
  - What are the potential threats that could impact each asset?
  - Think about CIA on each asset: Confidentiality, Integrity, Availability

- **Brainstorm attacks**
  - For each threat, what attacks could realize the threat?

- **Identify success conditions**
  - Under what conditions would each attack succeed?
Optimizing Test Efforts by Threat Modeling

• Threat profile serves as basis for security test planning:
  – assets of value have been identified
  – threats that could compromise those assets have been determined
  – attacks that could realize the threats have been uncovered
  – key conditions that must be met for each attack to be successful have been discovered

• Test plan should focus on testing the key attack conditions in order to prove/disprove threats to your app
  – This ensures you are testing the areas where the difficulty of attack is least and the impact is highest

• Grab Microsoft’s Free Threat Modeling Tool
Software Security Testing

- **Functional test techniques cannot uncover security bugs**
  - You are trying to find what the application is NOT supposed to do

- **Specific attacks should be applied to uncover vulnerabilities**

- **Many flaws are caused by environment interaction**
  - what if a resource is not available?
  - will we page that to disk? If so, when?

- **Many flaws are found after analyzing application environments**
  - discover sensitive information by sniffing the network
  - uncover temporary files

- **Use automated tooling for low-hanging fruit**

- **Use proven manual testing techniques for elusive bugs**
Software Security Testing Techniques

- **Application Penetration Testing**
  - Mainly a manual effort by experts

- **Scanning Tools – DAST & SAST**
  - Leverages automated tools to find common vulnerabilities

- **Model Based Testing**
  - Build a model of your application, test against it

- **Fault Injection**
  - Hit the error paths

- **Fuzzing**
  - Technique for generating malformed data streams to expose vulnerabilities
Pen Test and Code Review

• Will find the most elusive and dangerous vulnerabilities
  – Should be done in conjunction with tools for optimal effectiveness
  – Ideally performed iteratively, not just when code complete or application is compiled
  – Will find business logic and other flaws that tools can’t detect

• Uses humans to uncover vulnerabilities
  – Requires expert, well-trained minds
  – Expensive, time-consuming
  – Not easily scalable
Thinking Like an Attacker

• **Complete Knowledge of the System**
  – HTTP, SSL, IIS, Network Card, Apache, JavaScript, DOM, Browser, OS, Routers, VPN, TCP/IP,

• **Good Imagination**
  – What’s really going on there?
  – How would I build this?

• **Evil Streak**
  – Thinking like an attacker
  – Taking Vulnerabilities to their logical end
Conducting Attacks

Dependency Attacks
- block access to libraries
- manipulate registry values
- force application to use corrupt files (write protected, inaccessible, data errors...) and file names
- replace files that the application reads from, writes to, creates and executes
- force application to operate in stressed memory/disk space/network availability conditions

User Interface Attacks
- overflow input buffers
- examine all command line switches and input options
- explore escape characters, character sets and commands

Design Attacks
- try common default and test account names and passwords
- expose unprotected test APIs
- connect to all ports
- fake the source of data
- exploit loop conditions
- use alternate routes to accomplish the same task
- force the system to reset values

Implementation Attacks
- time of check and time of use
- create files with the same name as files protected with a higher classification
- force all error messages
- look for temporary files and screen their contents for sensitive information
Web Scanners

• **Pros – Automates testing; Finds what we already know exists**
  – Rely on a database of constantly growing *known vulnerabilities*
  – Identify common vulnerabilities faster than manual efforts
  – Can discover a large amount of information about a device
    • Misconfigurations, exposed usernames/passwords, cookie usage, vulnerable scripts
    • Directory/file structure, helper files, Java applets, Flash/ActiveX controls
    • Forms, query strings, hidden fields, input validation, header information

• **Cons – Scanners are just that – scanners**
  – Limited to *known* vulnerabilities, many serious vulnerabilities are missed
  – No prioritization of vulnerabilities
    • Vulnerabilities on critical assets ranked identically to low value assets
    • False positives are time consuming to validate
  – May provide false sense of security
Static Analysis Tools

- Look for patterns of bad code, simulate running conditions and find bugs

- Find a lot of the *common* coding errors, faster than humans
  - Can drastically reduce a number of bugs which may be difficult to find in black box testing
    - i.e. buffer overrun, encoding and dangerous function usage

- Do not predict exploitability of flaws

- Cannot detect certain types of vulnerabilities
  - E.g., Privilege escalation, cross-site scripting

- Beware of False Positives
  - Can result in wasted effort and drain on security resources
Best Practices for Using Tools

• Leverage to find low-hanging fruit

• Sequencing of tools introduction critical. When?” is just as important as “Which?”

• Adopt tool when you have baseline skills
  – Ability to interpret false positives
  – Ability to fix the problems you are finding
  – Ability to compliment with manual test efforts

• Compliment with sound process
  – What good are tools if not required and/or used at critical security gates?

• Compliment with Training
  – Tools don’t make your organization more mature
  – Tools are more productive when you know what you are looking for and can use them to prevent problems down the road
Model Based Testing

- Solves the problem of state
  - E.g. I need to call MethodX before MethodY will work.
- Hard to model a complex system and get the right level of detail
- Can be effective but requires a significant investment
Fault Injection

- Simulate environmental error conditions to target error code paths
  - Low memory
  - Corrupt files
  - Constrained bandwidth
  - Missing libraries
  - API errors
  - Etc

- Can be used to augment existing test suites and increase coverage
- Historically performed on hardware and has migrated to software
- Specifically targeted at reducing application fragility in the real world
Fuzzing Defined

• What is Application Fuzzing?
  – Technique for generating malformed data streams to expose vulnerabilities
  – Most effective when automated

• Why do it?
  – Quickly assess an application’s ability to deal with large amounts of invalid data
  – Hackers will use fuzzing to flush out basic validation problems. You should too!
  – Very effective at discovering obscure weaknesses and spectacular bugs in applications.
    • Maybe Blaster could have been found with a fuzzer?
Fuzzing Conceptualized

Web Application

TCP/IP

HTTP

Input Validation

Framework
The Problem Fuzzing Addresses

- Applications accept input from many sources. We want them to:
  - Do the right thing when they get good input
  - Fail securely when they get bad input

- Attackers want applications to:
  - Fail insecurely when they get bad input

- Parsers are used to process complex application input
  - Parsers are notoriously buggy and difficult to write
  - Malicious, corrupt input can result in fireworks!
  - Show me an elevation of privilege bug and I bet it’s in a parser
    • Ok, maybe not all of them are in parsers …

- Automated tools find low-hanging fruit and common vulnerabilities. What do you do next?
  - You need to test with the parser logic in mind.
Fuzzing – What Will it Find?

- Helpful for finding bugs that cause erratic application behavior and server crashes
  - Buffer overflows
  - Denial of service
  - Parsing issues
  - Data handling issues

- Less helpful at finding:
  - Elevation of privilege (EoP)
  - Weak encryption
  - Repurposing attack
  - Cross-site scripting
  - Information disclosure
  - DTD attacks
  - XML validation
  - Script injection
  - Repudiation
  - Session replay
Getting Good Coverage is Hard

• Data is composed of content and format
  – 1,2,3,5,7,11,13,17,19,23

• Good coverage requires:
  – Knowledge of the input format (protocols, file format, delimiters, etc.)
  – Knowledge of input validation techniques
  – Knowledge of side-effects
  – Knowledge of error paths
Fuzz Testing

• **Approach**
  - Identify Target
  - Identify Inputs
  - Generate/Execute Fuzzed Data
  - Monitor for Exceptions
  - Determine Exploitability

• **Define lists of "known-to-be-dangerous values" (fuzz vectors) for each type**
  - For integers: zero, possibly negative or very big numbers
  - For chars: escaped, interpretable characters / instructions (ex: For SQL Requests, quotes / commands...)
  - For binary: random ones
Fuzz Testing Techniques

• Fuzzing Techniques
  – Manual Fuzzing
    • Intercept and modify the data by hand
  – Automated Fuzzing
    • Use a tool to generate fuzz test cases
    • Can be done in real-time or stored for later use

• Types of Fuzzing
  – Random Fuzzing
    • Easy to implement
    • Randomly modify valid input and data types
  – Format Aware Fuzzing
    • More difficult to implement
    • Generate targeted random data to pinpoint failure spots in the software
Random Fuzzing

Pros
- Very fast to get started
- Easy to set up
- Finds low hanging fruit

Cons
- Unfocused
- Can generate wild border cases that you don’t care about
- Can generate a large amount of data
Format Aware Fuzzing

• **Pros**
  – Less likely to get hung up on format validation
  – Will greatly improve code coverage and depth of your attacks
  – Can target specific functionality
    • Input that is echoed into HTML for XSS
    • Input that is echoed into SQL statements for SQLi
    • Buffer size indicators
    • End of buffer tokens

• **Cons**
  – Large setup costs
  – Assumptions about the file format may be different than actual implementation
SOAP Fuzzing

• Take the client requests for each service and isolate the element values to be manipulated
  – Change the entire value, without conforming to the value format
    • Should turn up gross errors or denial of service conditions
  – Then present the value in the correct format
    • Should find issues that would pass a validation gateway, but still cause problems when the data is consumed

• Typical payloads used:
  – Character multiples
  – Max unsigned and signed integer values
  – Variations on format strings using ‘%n’
  – Long strings
  – Empty strings and null values
  – Extended ASCII
  – Binary values
  – Base64 and HTML encoded values
    - SQL Injection
    - Common bad ASCII (‘ ‘ ” < >)
    - All numbers
    - All letters
    - All spaces
    - Invalid date formats
    - Dictionaries relevant to the application
Web Services Fuzzing

• To monitor the behavior of the web service during fuzzing
  – Attach a debugger on the server

• To determine denial of service conditions use Perfmon to observe the process’s CPU and memory usage

• In SOAP fuzzing insert a unique marker into each request, and also log each sent request
  – Allows you to later reproduce the condition that caused the error
  – i.e. place an incremented number in the User Agent value of the SOAP requests, which is readable in the IIS logs.

• In addition, the randomness of fuzzing can be seeded with a value which would allow for reproducibility
• Everything between the Value tags should be fuzzed as a single blob

• The string “org:division/category/DATA=DATA” will be replaced as a whole with the fuzz strings.

<table>
<thead>
<tr>
<th>Original</th>
<th>&lt;Value&gt;org:division/category/DATA=DATA&lt;/Value&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fuzzed</td>
<td>&lt;Value&gt;AAAAAAAAAAAAAAAAAAAAAAAAAAA&lt;/Value&gt;</td>
</tr>
</tbody>
</table>
Web Service Fuzzing

- Next: separate values into subcomponents

<table>
<thead>
<tr>
<th>Original</th>
<th>&lt;Value&gt;org:division/category/DATA&lt;/Value&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fuzzed #1</td>
<td>&lt;Value&gt;AAAAAAAAA:division/category/DATA=DATA&lt;/Value&gt;</td>
</tr>
<tr>
<td>Fuzzed #2</td>
<td>&lt;Value&gt;org:AAAAAAAAAAAAA/category/DATA=DATA&lt;/Value&gt;</td>
</tr>
<tr>
<td>Fuzzed #3</td>
<td>&lt;Value&gt;org:division/AAAAAAAAAAAAA/DATA=DATA&lt;/Value&gt;</td>
</tr>
<tr>
<td>Fuzzed #4</td>
<td>&lt;Value&gt;org:division/category/AAAAAAAAAAAAA=DATA&lt;/Value&gt;</td>
</tr>
<tr>
<td>Fuzzed #5</td>
<td>&lt;Value&gt;org:division/category/DATA=AAAAAAAAAAAA&lt;/Value&gt;</td>
</tr>
</tbody>
</table>

In addition to fuzzing strings with other strings, INT’s, byte arrays, and separator values (e.g. : and / and =) should also included.
The Future of Fuzzing

• Use application feedback to improve test performance
  – Similar to manual iterative approach

• Watch application and tweak input to improve code coverage
  – Really hard, similar to the halting problem

• Use fault injection + Fuzzing to improve code coverage
  – Inject faults to vary the code path
  – Use fuzzing to target input/data validation on error paths

• Multi-tier Fuzzing
  – Improve coverage by considering system as a whole, not just a single process on one machine
  – Coordinate fuzzing on specific interfaces
    • Client to app server
    • App server to DB server
    • Web services interface
Getting Started with Fuzzing

• **Using a Fuzzer**
  – Prepare a correct file to input to your program.
  – Replace some part of the file with random data
  – Open the file with the program
  – See what breaks.

• **Start off with network or file fuzzing**
  – If your application accepts file input, start here
  – If your application accepts network input, go there next

• **When to Fuzz**
  – As soon as your code is complete, test your input/data validation
  – You can add simple fuzzing to your unit tests
  – Fuzz extensively before release: over 10,000 files
Fuzz Testing Tools

- **Burp Suite**
  - Intercept and fuzz network packets. Try for free, pay to unlock features

- **Skipfish**
  - Use to fuzz HTTP parameters and directory names. Free, hosted by Google.

- **Appscan**
  - Fuzzing and other testing features. Expensive but comprehensive

- **Peach**
  - Can fuzz files, RPC, network packets, stored procedures, many others

- **Spike**
  - Widely used and very flexible but requires coding skills to use

- **What the Fuzz**
  - Open source, written by Security Innovation

- **Dozens of others**
  - [www.fuzzing.org](http://www.fuzzing.org) for a large listing
  - [http://www.owasp.org/index.php/Fuzzing](http://www.owasp.org/index.php/Fuzzing) for more information
How Security Innovation can Help

• Computer-Based Training
  – 35+ courses for secure design, coding and testing
  – OWASP Top Ten, PCI-DSS for Developers, Writing Secure Code, Threat Modeling, etc.

• Secure Development Knowledge Base
  – Over 3,500 knowledge assets
  – How-to’s, code snippets, checklists, etc

• Secure SDLC Services
  – SDLC Optimization
  – Threat Modeling
  – Code Review
  – Security Testing
Contact

Fred Pinkett
fpinkett@securityinnovation.com