Application Security Principles

throughout the Software Development Lifecycle

Stephen Evans
Secure Application Services APAC
Gathering basic functionality requirements

Considering use cases, charting DFD

Implementation phase – from concept to reality

Validation phase – functional testing

Deploy (client sites, Intranet, Web), maintain
Software Development Lifecycle

Define Requirements

Defining Security requirements

Negative Use Cases, Threat Models

Secure Design

Design

Code Review, Pair Programming

Secure Design

Develop

Security Validation – Security Testing

Secure Deployment

Test

Patches, Maintenance, Incident Handling

Deploy
Software Development Lifecycle

Conceptual phase:
• Bugs and faults captured in this stage – never existed

Natal phase:
• Bugs and faults captured in this stage – Incur implementation costs, possibly roll back design

[Proactive Security]

Existential Phase:  [Reactive Security]
• Bugs and faults captured in this stage – Incident response, patches, patch management
• Security Risk Profiling
• Security Requirements
Risk = Threat x Vulnerability x Cost

- Threat – frequency of potentially adverse events
- Vulnerability – likelihood of success of a particular threat against an organization
- Cost – total cost of the impact of a particular threat experienced by a vulnerable target
• An exercise to determine the risk rating associated with an application and its development
• Takes place in the beginning of the SDLC
• Output geared toward both Project Managers & Security Personnel
• Output specifies security tasks to be carried out during the SDLC
• “How risky is this application?” questionnaire/discussion
Creating a risk measurement rating

• Each application is evaluated and receives a risk rating
• The risk measurement is based on several questions (10-15) tailored to each organization
• Each question will receive a risk score based on a scale (such as, low/med/high or 1-5)
• The total risk score gives a general risk rating of the application
• Compare risk ratings across an organization to effectively allocate security resources
• **Application**: Estimated calendar time for project completion? Size of the total project team? How many different physical locations will the application be deployed?

• **Development**: How many developers will be used? Update of an existing application or a brand new application? Application architecture design created internally or by a partner?

• **Access & User**: Will this application use credential access? Who will have access to this application externally? Will this application be available externally on the Internet?

• **Application Processing**: What information is processed by the application? What is the highest data classification of the data used by the application? Is data converted by the application? If so, what is the difficulty?

• **Reputational Risk**: Regulatory - What is the project’s visability to regulators? Media - What is the project’s visibility to the media? Public Relations - What is the project’s visibility to the customers?
Risk Profile Analysis: Categorize Risk

• 75% and above: High Risk
  – High likelihood of application/data compromise and reputational damage
  – First applications addressed in corporate security budget

• 50% - 75%: Moderate Risk
  – Good possibility of application/data compromise and reputational damage

• Below 50%: Lower Risk
  – Low/medium chance of application/data compromise and reputational damage
Using the application’s risk rating, the Project Manager can plan the appropriate security reviews during the SDLC.

<table>
<thead>
<tr>
<th></th>
<th>High Risk</th>
<th>Med. Risk</th>
<th>Low Risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Risk Assess.</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Security Requirements Review</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Threat Modeling</td>
<td>X</td>
<td>(optional)</td>
<td></td>
</tr>
<tr>
<td>Security Design Review</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Security Code Review</td>
<td>X</td>
<td>(optional)</td>
<td></td>
</tr>
<tr>
<td>Security Testing</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Functional Requirements

• Determine application functionality
• Based on traditional use cases
• Define logical constraints
• .. Are modeled after a “lawful” user

Security Requirements

• Constrain application functionality
• Are modeled after ABuse cases
• Focus on the “else” rather than the “if”
• .. Are modeled after a “chaotic” user
Given the classic ATM use cases:

- Customer inserts card
- Customer enters PIN
- Customer asks for money
- Customer gets money
- Customer leaves
ABUse Cases

Customer inserts card
• Inserts a piece of plastic; inserts a forged card; attempts to bypass card

Customer enters PIN
• Does not enter PIN; enters longer PIN; brute forces PIN

Customer asks for money
• Asks for $10000; tries to overdraft; tries other operation

Customer gets money
• No communication to mainframe; hatch could be jammed

Customer leaves
• Attempts repeated transactions; customer attempts to break ATM; customer attempts to take ATM home
• Gather up all valid use cases
• For each use case consider:
  – What am I assuming? Implicitly or explicitly?
  – What constraints have I placed on the user?
  – What could possibly go wrong?
• Allow yourself to go off on tangents
• Don’t get too specific as to attacks
  – Threat Modeling, next, will give you time for that
• Consider all possibilities
  – Most may be discarded, but consider all
Secure Design

• Threat Modeling

• Security Design Review
• Threat modeling analyzes theoretical risks and “attack vectors”

• Attack vectors define:
  – Direction: avenue of attack
  – Quantity: severity of attack

• Direction scoped by the STRIDE methodology
• Popular Methodology used by Microsoft
• Defines common attack vector classes

<table>
<thead>
<tr>
<th>Vector Class</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spoofing</td>
<td>Session Hijacking, MiM attacks</td>
</tr>
<tr>
<td>Tampering</td>
<td>Input malformation, cookie poisoning</td>
</tr>
<tr>
<td>Repudiation</td>
<td>Rogue clients, Transaction disavowal</td>
</tr>
<tr>
<td>Info. disclosure</td>
<td>Privacy leaks, overly descr. errors</td>
</tr>
<tr>
<td>Denial of Service</td>
<td>Broken exception handling</td>
</tr>
<tr>
<td>Esc. of Privilege</td>
<td>Broken access control</td>
</tr>
</tbody>
</table>

• Not all classes are necessarily applicable in your app
Another Popular Methodology used by Microsoft

Defines a metric to assign values to vectors

<table>
<thead>
<tr>
<th>Vector Class</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>D</strong>amage potential</td>
<td>Impact of successful exploitation</td>
</tr>
<tr>
<td><strong>R</strong>eproducibility</td>
<td>Special settings, or mitigating circumstances</td>
</tr>
<tr>
<td><strong>E</strong>xploitability</td>
<td>Likelihood of successful exploitation</td>
</tr>
<tr>
<td><strong>A</strong>ffected users</td>
<td>%-age and class of users affected</td>
</tr>
<tr>
<td><strong>D</strong>iscoverability</td>
<td>Likelihood of uncovering vulny</td>
</tr>
</tbody>
</table>

Metric may be used to prioritize attacks
Defining Attack Vectors: Attack Trees

Goal:
Obtain username/password to the system

Attack:
Brute Force Username/Password combinations

Countermeasure:
strong password policy

Countermeasure:
Account Lockout

Countermeasure:
Use additional Auth. (e.g. Client Certs)

Attack:
Sniff session traffic

Countermeasure:
Encrypt channel
• Areas of Analysis contain many topics to be reviewed and analyzed
• Topics follow company standard framework for secure design
• Follow security industry best practices if no company standards or defined security policies/guidelines
• Results in list of recommendations to current design
Areas of Analysis

- Authentication
- Authorization & Access Control
- Data integrity
- Error and exception handling
- Monitoring and logging
- Cryptography and encryption
- Database security
- Privacy, confidentiality and segmentation
- Web security
- Product Security
### Business Impact Matrix — Summary of Findings

<table>
<thead>
<tr>
<th>LEVEL OF EFFORT TO REMEDIATE</th>
<th>Business Impact:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LOW</strong></td>
<td><strong>HIGH</strong></td>
</tr>
<tr>
<td><strong>Red flag: fix immediately</strong></td>
<td></td>
</tr>
<tr>
<td>Data validation errors</td>
<td></td>
</tr>
<tr>
<td>Sensitive info manipulation and discovery</td>
<td></td>
</tr>
<tr>
<td>Training and awareness</td>
<td></td>
</tr>
<tr>
<td>Cross-site scripting</td>
<td></td>
</tr>
<tr>
<td><strong>Red flag: plan to remediate</strong></td>
<td></td>
</tr>
<tr>
<td>Security metrics</td>
<td></td>
</tr>
<tr>
<td>User and admin app colocation</td>
<td></td>
</tr>
<tr>
<td>DB/App controls and segmentation</td>
<td></td>
</tr>
<tr>
<td><strong>Fix at client's discretion</strong></td>
<td></td>
</tr>
<tr>
<td>Poor password handling</td>
<td></td>
</tr>
<tr>
<td>Bad error messages</td>
<td></td>
</tr>
<tr>
<td><strong>Bear risk</strong>, or fix at client's discretion</td>
<td></td>
</tr>
<tr>
<td>Dead code</td>
<td></td>
</tr>
<tr>
<td>Authentication errors</td>
<td></td>
</tr>
<tr>
<td>Authorization errors</td>
<td></td>
</tr>
</tbody>
</table>

Each finding’s x-y position in the Business Impact Matrix indicates the relative risk and likelihood of exploit (vertical axis) and the effort required to remediate (horizontal axis). The circle diameter signifies the overall impact on your business and brand value.
• Approaches

• Tools
What is Code Review?

• Security bugs may stem from many reasons:
  – Improper use of language API calls (for example, Strcpy)
  – Incorrect framework/class utilization (as in, Java/.Net)
  – Design bugs or use-cases that were not considered

• Project source is carefully scrutinized, looking for:
  – Coding errors
  – Dangerous API calls
  – Implementation faults
  – Design-Level and Logic security problems

• Challenges:
  – Optimally, code review will obtain 100% coverage
  – Practically, this is almost never achieved
  – Methodologies have been devised to max efficiency
Approaches: Low Hanging Fruit

- Scan code for potential buffer overflows
  - Insecure copy operations: strcpy/strcat
  - Improper formatting: sprintf
  - Insecure input methods: scanf, read, recv

- Find the “easy” bugs
  - Variable format strings: *printf(var); /* C/C++ */
  - malloc()/free() pairings /* C/C++ */
  - Improperly escaped input /* SQL injection, Null Bytes */
  - Insecure system calls /* all languages */
  - Command injection /* all languages, system/exec */
  - Directory Traversals /* all languages, file input */
Approaches: API Handling

• Focus on specific API calls:
  – Object Creation (Win32 CreateXXX, fopen…)
  – System escapes (exec, CreateProcess, system())
  – Dangerous APIs (str* functions, JNI, Unmanaged code)
  – Third Party/other component API calls

• Validate all API return codes
  – Make sure API calls are assigned as an lvalue
• Extra care is given to sensitive segments:
  – Authentication logic
  – Authorization logic
  – Cryptography-oriented code
  – Integer Arithmetic
  – Input handling
  – Exception Handling
  – Multi-Threaded code

• At the expense of static segments:
  – Functions with no input
  – Constant code paths (with no flow control)
• A programmer is likely to overlook his own faults

• Reviewer is a different person, validating:
  – Design was properly implemented
  – Patterns were followed
  – Code was correctly annotated and commented
  – Assumptions made in code documented and validated
Text processing utilities are especially useful:

- grep: clever regular expressions to find:
  - Dangerous APIs (e.g. “egrep ‘str(cpy|cat)””)
  - Format string Bugs (e.g. “grep ‘printf’ | grep –v \” “)
- find: quickly find header files, or external resources
- Use IDE “find in files”
  - find variable/function definitions
Automatic tools may often be used

- Secure Software’s (now Fortify) RATS
  - Rough Auditing Tool for Security scans C, C++, Perl, PHP and Python source code
- FortifySoftware’s Fortify Source Code Analysis Suite
- Microsoft’s FxCop
  - Analyzes .NET managed assemblies
- Can jumpstart code review with penetration test results
Security Testing

• Quality Assurance vs. Security Testing
• Code Review and Security Testing
• Penetration Testing
• Tools
Security Testing vs. QA

• QA testing tests Functional Requirements
  – Making sure functionality is “as documented”
  – Bugs are defined as intended functionality that differs from the actual functionality
  – Program does not do LESS than it is supposed to

• Security Testing tests Security Requirements
  – Making sure the program does not exceed its design
  – Faults are defined as actual functionality that differs from the intended functionality
  – Program does not do MORE than it is supposed to
• Security Testing complements Code Review
  – Code Review is a WHITE BOX approach
    • Useful only when the source code is available
  – Code Review may be severely limited by size
    • Automated CR may yield false positives/negatives
    • Manual CR is extremely time consuming

• Security Testing is a BLACK or GRAY BOX approach
  • Always possible, even on closed source
  • Insider knowledge helps, but not a prerequisite

• Security Testing can often be automated
• Incorporate “abuse cases” conceived during design

• Focus on boundary conditions:
  – Large (or obviously invalid) input
  – Border-range integers
  – Metacharacters (anything non-alphanumeric)
  – Timeouts
  – Limited resource availability
  – High system load
Penetration Testing

• Vulnerability classification schemes
  – OWASP Top 10, WASC, and Fortify donation to OWASP

• Vulnerability scanning vs. penetration testing

• Black hat hacking vs. pentesting as white hat testing and application testing

• Non-intrusive vs. intrusive

• What pentesting can’t do:
  – Conclude that an application or system is safe

• What pentesting can do:
  – Find design & implementation security issues
  – Verify that configuration & hardening is done properly
• Virtualization Software
  – VMWare, Virtual PC

• Unit Testing software
  – JUnit, NUnit, C++Test, …

• HTTP Tools
  – Browser Plugins
  – Application Proxies (Paros, WebScarab, BurpSuite)

• Input “Fuzzers”
  – Automated Tools (SPI Dynamics, etc.)
  – Custom Scripts (Perl, Python, etc.)
• Low Level Tools
  – Ethereal/Wireshark
  – TCPDump/TCPReplay
  – Netcat, Nmap, SNORT

• Vulnerability Scanners
  – Nikto
  – Wikto
  – Nessus
  – Metasploit
Secure Deployment

• Configuration
• Deployment
• Maintenance
• Should be simple and easy to understand
• Configuration files should be properly protected
  – An application’s management program should have read and write access to the configuration files
  – The application should have read access to the configuration files
  – Other users and groups should be denied any access to the configuration files
• Initialization files, if any, should be protected by the file system and stored where only authorized administrators can access them
• Access privileges should be limited by default until configured otherwise
• It is critical to harden the operating system where the application will reside
• Strip out all unnecessary functionality
• Create standardized host builds
• Harden the overall operating environment as well such as:
  – Router configurations, firewalls, etc.
  – Physical security
Deployment: Security Documentation

- Document and map security requirements to installed features, modules, etc.
- Document all security features fully:
  - The security aspects and configurations of the application
  - All configuration settings that have security implications
  - The security ramifications of enabling any supported feature
  - Areas where privacy compliance is important
Deployment: For Administrators

- Make sure they have the tools to secure it
- Make sure that they have access to the information needed to make configuration decisions
- Split administrative tasks among different administrative roles
- Use admin or root privilege as little as possible, both on the system and within the application
- Enforce best security practices on the administrative account
• Log to a local log file or system log file
• Also log to a central logging server (hackers modify local log files to cover their tracks)
• Ensure synchronized time across log machines for log file correlation & for forensic readiness
• Logging Configuration:
  – Regulate the amount of information that is logged
  – Log critical information under normal operation
  – Reconfigure to log extensively when troubleshooting
• Logging Sources: The documentation should identify:
  – What software modules produce log messages
  – To what dirs and files they write these log messages

• Log Messages and their meaning: The vendor docs should provide a table with the following info accompanying the message ID:
  – The event statement
  – A brief explanation of the event
  – The severity level of the event
  – Recommended action, particularly for higher severity events
• Provide information that allows the admin to:
  – Detect suspicious activity
  – Corroborate and correlate suspicious activity
  – Demonstrate accountability for that activity

• Remote Management: If allowed, follow strict security practices:
  – Authenticate the client (account and machine) using multi-factor authentication controls
  – Allow access only to legitimate accounts/machines
  – Use an encrypted channel for all communication on the management interface
• Updates and patches delivered electronically should be transmitted and executed over secure channels and have technical controls to ensure their integrity (e.g., digital signatures).

• The procedures for procuring and applying software updates and patches should be clearly documented.

• A penetration test is recommended after each major software upgrade.
• Develop or recommend procedures for reporting known/suspected vulnerabilities and providing workarounds, recommendations and mitigating strategies

• CERT provides excellent resources, documentation, and training to establish a computer security incident response team, or CSIRT (http://www.cert.org/csirts/)
Summary: Conclusions

- Security is a property, not a feature. It’s very difficult to bolt it in on at the end of the SDLC.
- Security features add complexity, which increases cost and project duration.
- Secure application principles can be added bit-by-bit (i.e. CLASP).
- The goal is a baked-in security process from beginning to end that becomes repeatable and measurable.
• http://seclists.org

• OWASP (www.owasp.org)

• Books

• Blogs
Thank You