BITS

FINANCIAL SERVICES
ROUNDTABLE

SOFTWARE ASSURANCE FRAMEWORK

January 2012

BITS
A DIVISION OF THE FINANCIAL SERVICES ROUNDTABLE
1001 PENNSYLVANIA AVENUE, NW
SUITE 500 SOUTH
WASHINGTON, DC 20004
202-289-4322
WWW.BITS.ORG

Executive Summary

Software is the critical underpinning to virtually all of the products and services offered by financial institutions. It processes information that is sensitive both to financial institutions' customers and to the institutions themselves. Developing that software is a complicated process involving many steps including idea conception, design, coding and testing. Implementing software is also often complicated given that virtually all developed software operates within a broader ecosystem. These complexities can give rise to improperly designed or insecure software that exposes an organization to a variety of risks. The risks associated with insecure software include:

- Increased likelihood of attack on financial institution software,
- Theft of customer information such as account and credit card data,
- Theft of corporate information and intellectual property,
- Increased maintenance costs and lost opportunity costs, and
- Reduced system availability and negative client reputational impact.

These risks can lead to both direct costs as well as indirect costs (such as those associated with reputational risk losses). Developing software in a secure fashion can do more than avoid costs though – it can also have a positive return on the investment associated with software. A January 2011 study commissioned by Microsoft and conducted by Forrester Consulting found that those organizations that utilize a secure development process experience more positive ROIs in various areas including time to market, post-development maintenance, and the amount of time devoted to testing.[1] Generally speaking, it is far less expensive to build secure software than to fix security issues once the software is deployed. In addition, if a breach occurs, the costs can increase dramatically based on the type of compromise and remediation efforts involved. Secure coding practices are critical for developing secure code.

Today, there are numerous methodologies that address various aspects of software development. While most are good for their purpose, they tend to be very detailed, focus on specific components of the software development process, and are often very focused on the development of software using specific techniques or software languages. These methodologies have their place in developing secure software, but generally fail to outline an overall process, or framework, of practices. In addition, they generally do not focus specifically on the needs of the financial services industry.

The BITS Software Assurance Framework provides a higher level outline of the various components of a mature, strategic program for secure software development for software used within the financial services industry. The Framework was developed by subject matter experts in software development primarily in the financial services industry. It can be applied to software developed directly by financial institutions or developed by third parties for the benefit of financial institutions.

[1] Forrester Consulting Thought Leadership Commissioned by Microsoft, State of Application Security, Immature Practices Fuel Inefficiencies, But Positive ROI Is Attainable, (Microsoft, http://go.microsoft.com/?linkid=9762340, 2011).

BITS Software Assurance Framework

Table of Contents

BITS Software Assurance Framework

Introduction

The BITS Software Assurance Framework represents a set of common practices within financial services firms to improve software security for customer- and employee-facing applications. This framework leverages the maturity of software assurance controls that were adopted by financial services firms in recent years and is intended to provide guidance and serve as a reference tool for financial services firms interested in improving software security controls and practices.

The framework is based on the following principles:

1. Software security is an essential dimension of software quality and should be part of an overall effort to consistently measure and improve the quality of the software development and integration practices for financial service firms.
2. IT risk controls are most effective when they are imbedded within core business processes during the software development process as opposed to being subsequently added or "bolted on" during risk assessment processes outside of software development activities.
3. Security vulnerabilities in software should be treated as defects (bugs) that require remediation to the application architecture, design, or application code within the software development lifecycle.
4. The use of techniques, practices, and tools that identify security vulnerabilities should be applied as early in the lifecycle as feasible to improve software quality and reduce the increased cost associated with subsequent fixing of defects.
5. Building and integrating software from third parties is a common and efficient practice for financial service firms. The practices discussed in this Framework are appropriate whether software is developed internally, developed by a third party, or purchased through a third party.
6. Investment in the development of resilient software components that are shared across applications within an enterprise represents an example of an *application architectural framework* that both improves quality and is a vital source of preventative software security controls.

These principles have emerged over time from the practical experience of many software security program implementation efforts in major financial services organizations. They represent the collective experiences from each implementation effort. This Framework identifies practices and essential controls for a successful software security program implementation by a financial services organization. The above six principles should be considered design principles when developing or enhancing an existing software security assurance program.

The Framework addresses the following key component areas:

- Education & Training
- Security Software Assurance Development Standard
- Threat Modeling
- Coding Practices
- Security Testing
- Pre-Implementation Practices
- Software Assurance Documentation Archive Best Practices
- Post-Implementation Phase Controls

Chapter 1: Education & Training

The BITS Software Assurance Framework identifies the essential components and principles of a mature software security program. A primary criterion for a successful program rests with the level of influence and change in behavior of application developers and systems integrators. There are many complex technical challenges in deploying vulnerability detection and management capabilities within a software security program for medium and large organizations, but all successful programs share a common commitment to education and training. An effective measurement of program maturity is to poll application developers to determine if they know the difference between a software vulnerability and a software defect. If the majority of developers answer that vulnerabilities and defects are one in the same, then the education program can be considered successful. In fact, this is the single moment of truth for developers embracing secure development practices.

Software development, whether commercial or not, involves the constant removal of defects from application code through iterative testing and modifications until the software reaches acceptable quality for the investment made. Developers know how to deal with defects in a development process. Developers that defer or delegate security vulnerabilities need to understand that the vulnerabilities are nothing less than defects that need to be prioritized and fixed. This sounds straight-forward but it actually represents a significant challenge. The education and training program in a mature software security program represents the "lubricant" to behavior change in developers and as a result, is an essential ingredient in the change process.

In addition, a mature software security program also helps developers understand that while minimizing the impacts of security vulnerability defects is a key component of secure development, utilizing effective architectural features, such as memory protection techniques, can serve to minimize the impact of unknown vulnerabilities.

For example, in early 2008, Dan Kaminsky discovered a critical vulnerability in the domain name service (DNS) system. The DNS system translates domain names that people understand, like www.blah.com, to IP addresses that computers understand, like 204.11.200.1. This creates an entire family of vulnerabilities—for example, the DNS system on PCs can be fooled into thinking that the IP address for www.badsite.com is really the IP address for www.goodsite.com, and there is no way for users to tell the difference. This flaw allows cyber criminals at www.badsite.com to trick a user into doing all sorts of things when their browser takes them to their site, like giving up bank account details. Kaminsky had discovered a particularly nasty variant of this DNS cache-poisoning attack. This vulnerability was unknown in the original design and testing of the DNS system.

What is important in this example, however, is how *good design decisions* can make software secure and resilient from vulnerabilities and attacks yet to come. Years ago, cryptographer Daniel J. Bernstein looked at DNS security and decided that source port randomization was a smart design choice. Bernstein did not know about the specific vulnerability or of Kaminsky's eventual attack design, but he saw a general class of attacks and realized that this enhancement could protect against them. Consequently, the DNS program he wrote in 2000, "djbdns", did not need to be patched—it was already immune to Kaminsky's attack. It is a very good example of how teaching techniques of good

design and their subsequent use can result in software secure not just against known attacks, but also against unknown attacks and attacks yet to come[2].

Two fundamental constructs of a software security education and training program have emerged in recent years in recognition of the evolution of educational content and the significance of education as an essential ingredient in a successful implementation:

1. While many consider application developers to be the primary, critical constituency for education, that single constituency is not enough to reinforce appropriate behaviors within an organization. Education is essential for all of the key roles within the software development and integration process including:

   - DBAs
   - Project Leads
   - Project Managers
   - Architects
   - Quality Assurance Testers
   - Information Risk Professionals
   - Business and IT Management
   - Business Systems Analysts (BSAs)

   In addition, distinct application developer curricula, depending on the development platforms, are necessary. For example, here are several different groups of developers who require a separate curriculum:

   - Java developers
   - .NET developers
   - COBOL developers
   - Mobile application developers
   - SQL developers

2. Educational content for software security today is available from many different vendors and can be tailored to the specific groups of stakeholders that need it. The content can be delivered via e-learning, through workshops, and through traditional classroom settings. In addition, integrating this education into corporate Learning Management Systems (LMSs) is common and practical given recent adoption of standards. Some customization of the material specific to an organization's Software Development Life Cycle (SDLC) may be required along with specific workshops, but the majority of the educational content is already available from 3rd parties today.

---

[2] Merkow, M. S., & Raghavan, L., Secure and resilient software, requirements, test cases, and testing methods. (Auerbach Publishers, 2011).

One common approach includes defining the various stakeholder groups up front (Java developers, architects, project managers, Q/A staff, etc.) and assigning owners of each stakeholder group the responsibility for defining the mandatory and optional curriculum for their respective group. The owners evaluate and adjust the education and training content as the needs evolve. A formal review can be conducted annually to assure that the content remains current and relevant. In addition, the curriculum can be separated into awareness sessions and hands-on training for more in-depth content. Specific examples of awareness education for three stakeholder groups from an existing program are provided in Appendix A.

A sample description of stakeholder groups and specific curriculum is also available in Appendix A. By no means is this list exhaustive, but is illustrative of key items one should consider a baseline when evaluating an education program.

Integrating IT risk controls within the software development process as a part of a software security program requires on-going education for key stakeholders since the techniques and approaches are continually evolving. For example, this is clearly evident with the recent growth in developing applications for deployment on mobile platforms. Another model gaining traction is the development of an architectural construct, designed to take Java applications and port them to different mobile platforms so that developers can focus on Java development and this component will adapt the code to the targeted mobile platform. This allows the development teams to integrate Web development tools (e.g., static analysis) in the development process and leverage dynamic scanning capabilities (e.g., application penetration testing) in the Quality Assurance phase prior to configuring the applications for the target mobile platform. Dealing with the differences and complexities in security capabilities across mobile devices is indeed a challenge. A few organizations have taken steps to separate the responsibilities of development teams from software release teams that approve the distribution of mobile applications for both internal and external use. Final testing then includes manual penetration testing and/or using automated testing tools to test security and quality attributes of the program.

As development processes continue to evolve, on-going education for all stakeholders involved in the application development process is a must.

Chapter 2: Security Software Assurance Development Standard

A holistic, process-based approach to implementing secure application development techniques has been proven to lead to substantially better outcomes in terms of designing security into the application and creating mitigations for potential future vulnerabilities. One of the crucial early steps for this process is to conduct a risk analysis and to define a *minimum standard of security and privacy* attribute for the particular application. This process of considering these requirements early on sets the stage for decisions later in the process to classify issues and to make judgments about the quality of the security and privacy process. Creating this standard helps improve the efficiency of the entire development process - not just the security aspects. Noting the work done by Forrester[3] and Aberdeen Research[4], implementing a holistic secure development program with this type of requirement setting and early security design work can result in a significant return on investment in terms of resource time and schedule.

Minimum Development Standards

All application software must have a documented risk analysis performed before design begins with clearly identified areas where security practices, such as threat modeling, design reviews and scope of dynamic testing are defined. This risk analysis should also highlight specific security requirements relative to the intended use scenarios and environment. Outcomes from this process include quality gates that define the required security and privacy standards. The quality gates also provide a mechanism to assess progress and to provide a sense of overall risk mitigation maturity of the application. Bug bars are one type of quality gate that set thresholds for classifying security and privacy vulnerabilities. As developed by Microsoft for use in its Security Development Lifecycle, bug bars provide a tool to complete the critical steps of both classifying vulnerabilities and the severity of vulnerabilities that are permissible.[5]

Examples of Security Bug Bars

Severity	**Example Issues**
Critical	Remote elevation of privilege
Important	Denial of Service
Low	Spoofing (As part of bigger attack scenario)

[3] Forrester Consulting Thought Leadership Commissioned by Microsoft, State of Application Security, Immature Practices Fuel Inefficiencies, But Positive ROI Is Attainable (http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=2629, 2011).

[4] Aberdeen Group; Securing Your Applications: Three Ways To Play, Part Three (http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=6968, 2010).

[5] Bryan Sullivan, Add a Security Bug Bar to Microsoft Team Foundation Server 2010 (MSDN Magazine, 2010).

Examples of Privacy Bug Bars

Severity	Example Issues
Critical	Lack of notice & consent
Important	Lack of data protection
Moderate	Data minimization

Components of a Secure Application Development Standard

The following is a high level list of the components needed for an effective secure development standard:

1. The development standard must be consistent with and reflect the institution's regulatory obligations and internal policies.
2. The development standard must address the key elements of privacy and data collection minimization. These issues need to be addressed early in the design process.
3. Applications must be mapped to the standard risk classification criteria so that all stakeholders understand what controls are required in all subsequent development phases.
4. The standard must be unambiguous and well documented to enable meaningful decisions during design, coding, testing and final security review.
5. The standard must explicitly map out the expected quality gates with exit criteria that provide a clear and unambiguous indication of the results for each quality gate, thereby creating a body of evidence that can be used in the final security review prior to application deployment.
6. Privacy and data governance requirements must be incorporated into the standard to assure these important issues are considered along with software security.
7. A tracking mechanism is also needed to enable tracking of all vulnerabilities, work items and quality gate completions.
8. The standard must produce, on a project-by-project basis, consistent artifacts that validate compliance with the standard. These artifacts must be retained for use by third-parties (e.g., auditors, regulators, software purchasers) who may wish to validate the process.

A critical consideration of all of the listed requirements is to assure that the development standard is well-defined and unambiguous so that clear quality decisions can be reached during the development and testing phases.

Quality of the Development Standard

The practical applicability of any development standard comes down to the quality and clarity of its construction. The key to that, is assuring that all of the components enumerated in the previous section are covered. The standard must also be practical relative to development and test phases being able to meet the risk mitigation criteria. The standard must be clear and implementable and must provide the ability to make clear choices about the severity and required remediation of bugs that are identified by the process. The usefulness of a development standard is derived from the ability for everyone to use it, consistency in the standard and its application. An example is that a developer when running a static analysis tool discovers a bug. Based on the criteria identified in the secure application development standard, the developer should be able to rate the severity of the issue and take the appropriate steps to mitigate without outside advice or assistance. Only when all

of the issues that do not meet the development standard or bug bar have been met can the gate be passed and the process can move to the next step. This allows an organization to see great efficiencies since the best, easiest and least expensive place to fix issues is at development time rather than fixing them later based on input from subsequent stages in the process. Testing can then be focused on other, more complex issues since many of the bugs would have already been identified and remediated. Before any application can be moved to production, it needs to demonstrate that it has met the application development standard at **every** quality gate and that there are no remaining issues.

It is also helpful to consider engaging internal or external resources who are security and privacy experts to help at this stage. Involving this type of expertise early can help identify issues earlier and help set the appropriate context for the subsequent development and testing. This can result in significant savings compared to having these issues arise later or even worse, missing key issues.

As the development standard, and more broadly the Framework, is applied, the following key questions should be considered to help identify where deep assessments need to be undertaken to identify the security and privacy functional aspects of the applications:

1. (Security) Which portions of the project will require threat models before release?
2. (Security) Which portions of the project will require security design reviews before release?
3. (Security) Which portions of the project (if any) will require penetration testing by a mutually agreed upon group that is external to the project team?
4. (Security) Are there any additional testing or analysis requirements that critical outside constituencies (e.g., auditors, regulators, third party assessors) deems necessary to mitigate security risks?
5. (Security) What is the specific scope of the fuzz testing requirements?
6. (Privacy) What is the Privacy Impact Rating? The answer to this question could be based on the following example guidelines:
    - P1 High Privacy Risk. The feature, product, or service stores or transfers PII, changes settings or file type associations, or installs software.
    - P2 Moderate Privacy Risk. The sole behavior that affects privacy in the feature, product, or service is a one-time, user-initiated, anonymous data transfer (for example, the user clicks on a link and the software goes out to a Web site).
    - P3 Low Privacy Risk. No behaviors exist within the feature, product, or service that affect privacy. No anonymous or personal data is transferred, no PII is stored on the machine, no settings are changed on the user's behalf, and no software is installed.

Improving the Secure Application Development Standard

As organizations exercise the Secure Application Development Standard through their application development projects, they have an opportunity to learn lessons from those activities. How that learning occurs is primarily dependent on both the size of the organization and the number of development projects it undertakes.

In smaller organizations, or those with fewer development projects, it may be possible to perform a post-project lessons learned exercise after every application development project. In larger

organizations, or those with many development projects, individual post-project exercises may not be feasible. These organizations might consider post-project exercises for select projects (e.g., those of higher criticality or investment). Larger organizations often use other options, which are also useful tools for smaller organizations, such as measuring defect results across projects at various phases and monitoring production environments for defects that they track back to specific projects. Regardless of approach, there needs to be a robust process for measuring the effectiveness of the overall secure development lifecycle that serves to identify the need for improvements.

In addition to measuring effectiveness, institutions should consider other methods to improve their process. For example, as new use cases emerge, institutions should use them to expand both the risk profile for the particular application for which it emerged and the applicability to other applications' risk profiles. Likewise, organizations should monitor changes to regulatory or compliance requirements and integrate those changes into their risk profiles. The key though is that, for a given risk profile, the development standard should stay relatively consistent within an organization.

Chapter 3: Threat Modeling

Threat modeling is essential for resilient software design and development and is defined as:

> *Threat is anything that can act against an asset resulting in a potential loss. Risk is the magnitude of a loss and the frequency or probability of that loss. Threats act on or exploit a vulnerability in an asset resulting in a risk of loss. Threat modeling is the process of introducing threats into the application design phase of a software development lifecycle with the explicit purpose of influencing the design to address the most likely threats to the application's security and resiliency. Threat Modeling is often one part of a security requirements phase or architecture review phase that is part of application design.*

Threat modeling is one of the most important components for addressing risk in the development of software yet it remains one of the more elusive controls to implement consistently within an organization. There are several reasons for this elusiveness in consistently applying threat modeling in the design of software including:

- Confusion on what constitutes a threat vs. a vulnerability vs. a risk
- Lack of guidance on methods to identify assets
- Requiring participants with requisite expertise and training in threat analysis, a strong understanding of application design and a well-structured process
- Security experts often learn from different risk profiles and use different techniques for modeling
- Teaching threat modeling requires an apprentice-based approach that involves an appropriate curricula, adequate investment in effective education tools and a process for educating appropriate constituencies
- Different types of applications have very different risk profiles meaning the threats will vary depending factors such as the application architecture

The basic components of a threat model typically include:

- **Threat Actor**- the "who" that wishes to attack an application
- **Threat Vector (or Attack Surface)** - applies to the approach and opportunity a threat actor has to execute a specific attack. A large attack surface means there are many opportunities for a threat actor to find and exploit vulnerabilities. A small attack surfaces means there are limited opportunities for the threat actor to succeed.
- **Use Case** – A specific, often important, intended action taken by the system or the end users of the system, such as logging into the application or querying a database.
- **Abuse Case**- this is similar to a use but identifies a specific attack approach and how the application may be compromised. These are sometimes referred to as misuse cases.
- **Trust Zone**- This refers to aspects of the application where imbedded controls (e.g., authentication) protect against malicious attacks or threats and those with access to the trust zone can be trusted at a higher level than users of public sites.

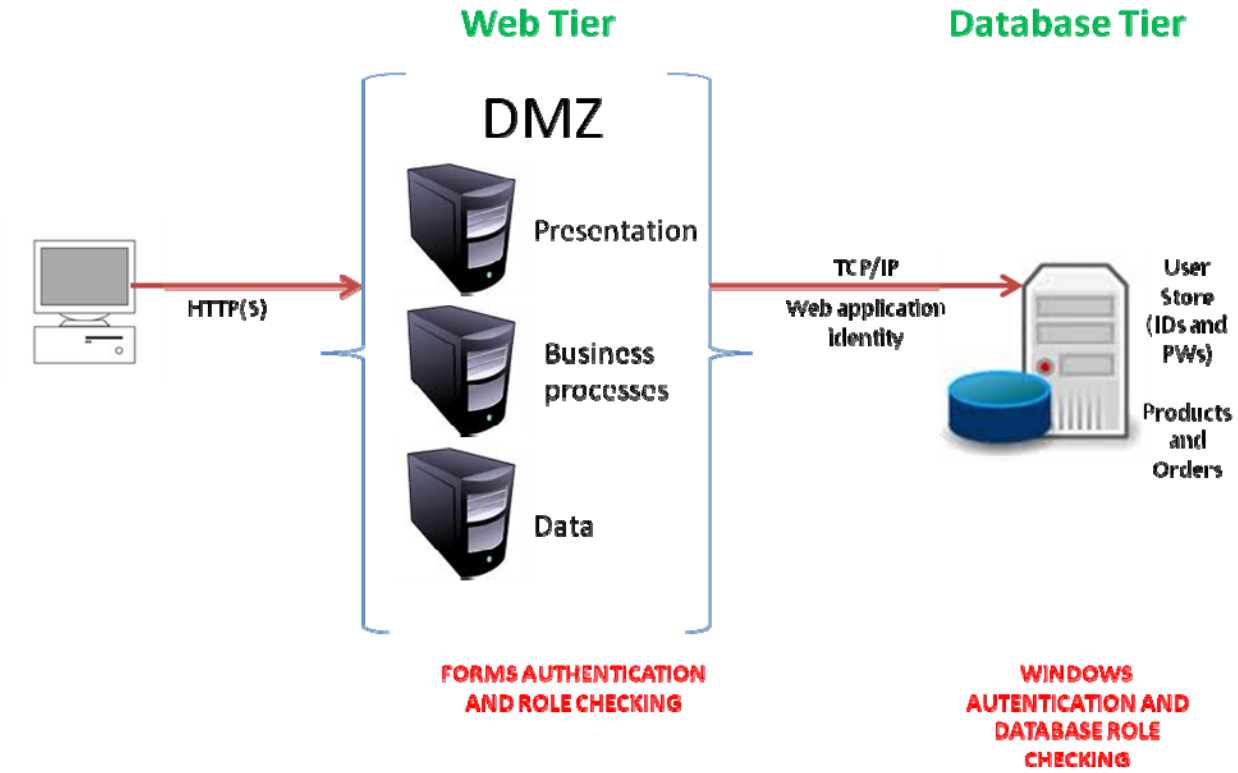
Threat modeling is most effective as part of an architecture or design review process where technical considerations are addressed along with the threats and risks of a specific design option. The most significant aspect of the introduction of threat modeling into an architecture review process is the recognition that making design changes to accommodate improved risk management is far less expensive versus changing application code that has already been designed, developed and implemented, and can reduce remediation costs by a factor of 100[6].

One of the key benefits of threat modeling is that the models should be developed in collaboration with the architects, developers and the QA/Test personnel. In doing threat modeling collaboratively, everyone can agree on the relative risk of different areas of an application so that when a developer has an issue he knows what severity a particular bug might be assigned and can fix it at the most efficient point - when it is being written. The QA/Test personnel benefit from threat modeling by virtue of the fact that they now have insights into how an attacker might target this application so that will be taken into consideration when exercising the threat models as part of the testing cycle. Architects can benefit since they can see where additional features or design changes could have a significant impact on the relative security of an application.

Threat modeling outputs should include identification of all relevant assets (e.g. databases, servers, applications, Web services, etc.), architectural diagrams, dataflow diagrams and trust boundaries (identified either in the architectural diagram or dataflow diagram). A mature threat modeling process will include a library of threats and attack patterns that can be applied to various scenarios. This ensures that all threats are considered and taken into account during the modeling phase. Any new threats/attack patterns identified as part of threat modeling are added to the library for future use. Some organizations choose to develop specific threat models for different architecture types or archetypes (e.g., ATM threat model, consumer on-line banking threat model, business-to-business institutional trading threat model, etc.).

Threat modeling should include creating a diagram for the application designed. A simple example of such a diagram follows.

[6] Boehm, Barry W., and Richard Turner, Balancing Agility and Discipline: a Guide for the Perplexed (Boston, MA: Addison-Wesley, 2006)



Threat modeling should also include a data flow diagram (DFD) since data should be clearly classified and managed accordingly. Data flow diagrams are used to graphically represent the application using a standard set of elements including: data flows, data stores, processes and interactors. Microsoft recommends including trust boundaries in the data flow diagram. DFDs can be drawn using these symbols according to Shawn Herman, Scott Lambert and Tomasz Ostwald and Adam Shostack from Microsoft.[7]

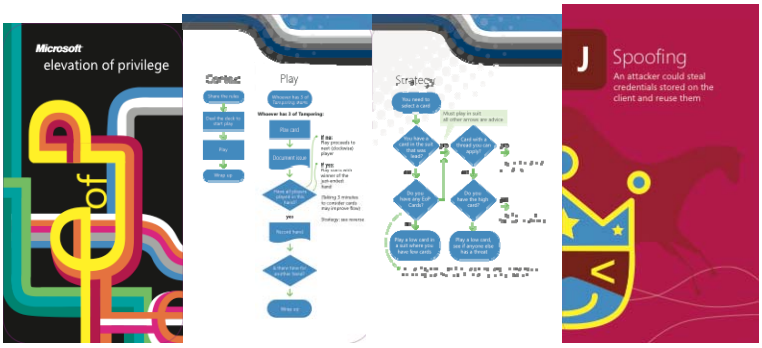
Item	Symbol
Data flow	One way arrow
Data store	Two parallel horizontal lines
Process	Circle
Multi-process	Two concentric circles
Interactors	Rectangle
Trust Boundary	Dotted line

[7] Shawn Hernan, Scott Lambert, Tomasz Ostwald and Adam Shostack, Uncover Security Design Flaws Using The STRIDE Approach (MSDN Magazine http://msdn.microsoft.com/en-us/magazine/cc163519.aspx, 2006).

This group recommends the use of the STRIDE approach:

Threat	Security Property
Spoofing	Authentication
Tampering	Integrity
Repudiation	Non-repudiation
Information disclosure	Confidentiality
Denial of service	Availability
Elevation of privilege	Authorization

Adam Shostack of Microsoft produced a highly creative way to teach software development teams using a card game called Elevation of Privilege[8], which is fun to play and makes it easy to learn threat modeling. A sample of a card is shown below:



One of the more challenging aspects of threat modeling is recognizing that an Internet-facing money movement banking application for retail consumers carries a very different risk profile from a business to business money movement application behind a VPN with two factor authentication. There are indeed risks for both applications but they differ and determining the actual risk for any given application architecture is essential to leverage threat modeling effectively. An Internet-facing application that requires detailed customer personal information will have an inherently higher risk than an application that offers marketing collateral to prospective customers without a registration process.

Many organizations will classify the application risk of a given project to develop a new application during the design or requirements phase using a simple High, Medium, or Low risk designation. The advantage of classifying applications early in the lifecycle is that it yields the opportunity to apply controls to the development process that are commensurate with the level of risk. Applications that process sensitive customer information gathered over the Internet will have a higher risk classification than Websites with public information. The corresponding risk classification informs which appropriate controls must be applied for the remainder of the SDLC. As an example, an application classified as high risk may require approval from a security architect

[8] Microsoft® Security Development Life Cycle, Announcing Elevation of Privilege: The Threat Modeling Game (Microsoft, http://blogs.msdn.com/b/sdl/archive/2010/03/02/announcing-elevation-of-privilege-the-threat-modeling-game.aspx, 2010)

before IT Management approves funding for the development of the application as a way to assure appropriate understanding and managing of risk in the application design phase.

Many organizations will use threat modeling to apply a risk score to a given application along with the classification. There are several advantages to using a risk score in addition to the risk classification of high, medium and low including:

- Makes it easier to determine the relative risk of one development project from another across application portfolios or business units
- Implements standards that require specific controls based on either the classification and/or the risk score
- Measures the risk at multiple points in the development process to better define residual risk throughout the lifecycle leading to production of the application
- Measures risk trends over a period of time across an application portfolio

The most effective use of threat modeling in financial services organizations integrates a security requirement or architecture review with a governance process. An example of this is when funding approval requires a specific architecture review that includes a risk assessment based on threat modeling in the design phase. In most organizations, application enhancements or development of new applications to support business needs represents a core business process that is essential to the success of the financial services organization. Therefore, the integration of IT risk controls within the software development process as part of IT Governance represents a high level of maturity for an organization.

Chapter 4: Coding Practices

Mature coding practices help to assure that the code being developed protects the confidentiality, integrity and availability of the data, business processes and informational assets necessary for successful operations. Often, developers do not account for security validation of the code as part of the code review or development process. There are several reasons for the absence of security as part of software coding practices including:

- Developers lack of understanding of security vulnerabilities
- Lack of secure coding process
- Lack of developer training and education on secure coding practices
- Insufficient architectural and design knowledge to identify security controls provided by the environment/frameworks as opposed to those that have to be coded
- Absence of a central repository of approved security software libraries

It is far less expensive to build secure software than to fix security issues once the software is deployed. In addition, if a breach occurs, the costs can increase dramatically based on the type of compromise and remediation efforts involved. Secure coding practices are **critical** for developing secure code.

Several secure coding practices processes are available in the marketplace (e.g., OWASP Secure Coding Practices Guide, Department of Homeland Security Build Security In Portal, CERT Secure Coding, MSDN Security Developer Center). Common recommended practices include:

- General Principles:
  - Principle of least privilege, late privilege (grant elevated privileges as late as possible in processing), and minimum duration privilege (drop elevated privileges as soon as possible)
  - Reducing the attack surface, i.e., expose the least number of services/functions necessary for satisfying requirements
  - Confirm the integrity of all third-party code using crypto (e.g. hash sums, digital signatures)
  - Consider timing and race conditions
  - Initialize all data stores and variables
  - Ensure application does not use functionality that breaks compatibility with any operating system security vulnerability protection features (e.g. ASLR & DEP])[9]
  - Use of Data Flow Diagrams to fully understand all entry and exit points
  - Complier switches that flag unsafe practices should always be used (e.g., /GS)
  - Developers should run static code analysis tools before checking in code

[9] Microsoft® Security Research and Defense, On the effectiveness of DEP and ASLR (Microsoft, http://blogs.technet.com/b/srd/archive/2010/12/08/on-the-effectiveness-of-dep-and-aslr.aspx8, 2010).

- Authentication and Access Control
  - Fail-safe and/or fail-secure, as appropriate, application authentication and access controls
  - Use trusted processes/objects (do not trust client-side controls) for access control decisions
  - Validate all requests made by authenticated users, not just the initial request
  - Ensure only authenticated and authorized users can access application resources
  - Do authentication and access controls checks **only** on the server
  - Centralize authentication and access control functionality using common approved libraries
  - Protect against brute force attacks by enforcing account lockout
  - Review design of password reset functionality and ensure it is secure
  - Ensure high entropy of authentication credentials and security questions. Enforce authentication credentials history and expiration policies
  - Use out-of-band communication and multi-factor authentication for critical applications
  - Do not store, transmit or log authentication credentials in the clear. Use strong cryptographically, one-way salted hashes to store authentication credentials. Ensure restricted access to credential store
  - Avoid information disclosure by ensuring failed authentication attempts do not divulge implementation details
  - Encrypt and securely store authentication credentials used by the application to access other services, applications, and databases

- Entitlements
  - Do entitlement checks **only** by the server
  - Avoid proliferation of roles and ensure secure design by abstracting privileges into appropriate roles
  - Centralize all entitlement checks and strive to externalize entitlement logic
  - Keep the role hierarchy simple. Complex hierarchical roles and groupings lead to misconfigurations resulting in loss of least privilege

- Session Management
  - Only implement custom session management as a last resort. Strive to use session management functionality provided by underlying frameworks, Web/application servers
  - Do not mix unauthenticated and authenticated session identifiers
  - Ensure high entropy for session identifiers
  - Protect session identifiers from accidental disclosure
  - Enforce session timeouts, deactivation and force-outs

- Input Validation
  - Validate all input on the server
  - Centralize input validation functionality
  - Do input validation only on properly canonicalized data
  - Ensure all client data is validated prior to processing

  - Scan any uploaded files for malware

- Encoding/Sanitization of Output
  For systems where data and application controls are co-mingled (e.g. interpreters, SQL, etc.)
  - Encode (e.g., interpreters) or sanitize (e.g., SQL, XML) all output
  - Centralize and do encoding/sanitization on the server. Factor in requests to not only external systems but also internal systems

- Error Handling, Logging and Auditing
  - Avoid sensitive information disclosure in error messages, alerts, logs, and other application messages
  - Implement standard generic error messages to avoid unauthorized information disclosure
  - Centralize all error handling, logging and auditing functionality on the server
  - Ensure secure design of error reporting, logging, and auditing functionality. All sensitive operations should be logged. Type of information reported and frequency of logs should also be considered
  - Attempt to use industry standard routines
  - Restrict access to errors, logging and auditing information to authorized individuals only

- Data and Database Protection
  - Enforce encryption of sensitive data in transit, at rest and in memory
  - Ensure all entities that deal with data at rest or in transit are secure and protected against data loss (e.g., caching servers, external services/applications)
  - Limit duration of time that data exists on external systems
  - Ensure that only authorized individuals/systems can access data
  - Use parameterized queries, stored procedures and principle of least-privilege on the database

- Operational
  - Secure updating
    - Use crypto signatures for code updates
    - Secure transmission channels of code updates
  - Ensure all default accounts, passwords, services, methods, ports and processes on all systems supporting the application are reviewed, modified as appropriate and enabled only if necessary
  - Ensure only code that is required for the application to function resides on the production servers. Remove all test code, backup code, unneeded extensions, etc.
  - Ensure all authentication credentials used to access the database are encrypted and encryption keys are protected
  - Ensure all supporting systems (servers, databases, etc.) are using latest approved version and are patched. Where appropriate ensure that systems are hardened

- Memory Management
  - Ensure secure design of memory management and allocation. Avoid buffer overflow issues by validating requested memory and destination buffer sizes prior to use
  - Unallocate unused memory, close resources and implement garbage collection
  - Avoid use of vulnerable functions

A mature secure coding practice has a common approved library of security functions that developers can programmatically invoke. This approach has a very high return on investment (ROI) since it reduces developer training, coding, testing, documentation and increases software development velocity. In addition, it ensures consistency of implementation, higher degree of security and provides an ideal insertion point for dealing with new threats and attacks. The security libraries can be modified internally to deal with the new types of threats and attacks and if the APIs remain unchanged, the impact to developers is minimal.

A common failure is for firms to develop libraries only for a certain technology or language (e.g., Java or .NET). Firms should do an inventory of their enterprise and strive to ensure there are security libraries for all of their supported platforms.

Chapter 5: Security Testing

Introduction

The ability to effectively validate that secure design principles, decisions, and requirements have been properly developed into a software solution is critical to a secure SDLC process. Without effective testing practices implemented into the development lifecycle, well-conceived sets of declared security requirements, security architectural designs, and/or threat models may not be adequately implemented or mitigated, resulting in a system design more prone to compromise. The Testing Practices section of this Framework describes a set of principles financial services institutions should consider when integrating security into their overall development and testing lifecycles.

Security Testing – Program Goals

It is important to consider the intended goals or purpose that an effective security testing program intends to achieve. At a high-level, the primary goal of a security testing program should be to assure that the security requirements of a software development effort have been properly implemented. Regardless if the system is being developed by internal resources or by a third-party supplier, financial services institutions should consider the following key goals for a successful testing program:

Comprehensive: A Security Testing Program must include validation of security requirements across the entire application infrastructure - not just the application itself - to include all system design elements and components, including:

- *Databases and Database Servers*, such as Oracle, MS SQL, and DB2, which store application or sensitive client data.
- *Server/Mainframe Operating Systems*, such as MS Windows, Linux, and z/OS, which host the application tier.
- *Application Servers*, such as IBM Websphere and J2EE which host the application platform.
- *Web Servers*, such as Microsoft IIS, Apache, and other that serve content with which the intended end user interacts.
- *Enterprise Architecture components*, such as Extract, Transform, and Load (ETL) environments, or Service-Oriented Architectures (SOA).

Other components, such as network routers and firewalls, may also be part of the testing lifecycle depending on the nature of the development effort, but are not specifically within the scope of this document.

Creative: The practitioners responsible for performing security testing should be creative. Threats and exposed security vulnerabilities, by their very nature, involve some element of the unknown, where security testing teams are in fact attempting to test what *should not* happen versus, say, a traditional functional or non-functional requirement with defined test

criteria. For example, an Internet-facing application designed to allow private wealth clients the ability to view the contents of their investment portfolio may allow the client access to portfolio statements where the full path to the statement is somehow addressable via the URL. Without adequate authorization controls in place, an attacker or malicious user could modify or craft the URL to access statements for other clients. A successful security testing program should promote innovation during the testing process through adequate security training, effective threat modeling, and good fuzz testing practices (see page 23). Each of these practices, if they involve the testing team, can encourage the testing practitioners to essentially think more like an attacker, thereby making them more successful security testers and further adding value to the security testing process.

Flexible: Business and functional requirements inevitably change during the SDLC process. A successful testing program should be able to adapt to such modifications in project scope and adjust accordingly. For example, threat models should be assessed or re-assessed for validity, and as these are updated, testing requirements against those threat models should also be updated so that they are thoroughly vetted in the testing cycle.

Integrated: Security testing should not be performed independent of the overall testing process, but rather, should be effectively *integrated* into the financial services institution's testing lifecycle. Like any successful SDLC, financial services institutions should involve the testing organization earlier in the SDLC to develop appropriate acceptance criteria and measures for success. For example, a financial services institution may consider involving testers in threat modeling exercises to help the testing team to think more like attackers and carry that mindset into the actual testing lifecycle. As noted earlier, security testing should be integrated to include all components of an application platform instead of just the application itself, and should be enabled to include specific threats identified during the threat modeling phase, all exercised by the dynamic and manual (static) testing process. Finally, as an additional measure, financial services firms should ensure, to the degree applicable, that application vulnerability scans incorporate the latest vulnerability research available and are executed against the built code base prior to the system being introduced into production, and on an on-going basis as an operational function.

Measurable: Any successful security testing program must provide mechanisms that convey the effectiveness of the security testing. Financial services institutions should consider developing metrics that meaningfully describe the testing results of previously identified security requirements and threat models. This can be done both on a system-by-system basis and on the health or success of the enterprise security testing program as a whole. These metrics can offer a balanced risk view that accurately conveys the impact, or potential impact, of any defects found. Valuable measurements on the success of the testing effort will not only provide firms accurate information related to the results of the security testing, but will also provide demonstrable, repeatable artifacts that can be reproduced to other parties, such as internal/external auditors and regulators.

Security Testing – Program Components

Creating and maintaining a security testing program that is comprehensive, creative, flexible, integrated, and measurable for any financial services institution – especially as the industry continues

to witness ever-increasing market changes and regulatory pressures – is essential for any software security framework. The following are key components we recommend firms evaluate for introduction into their own SDLC process to provide better assurance around security testing. Each of these components is organized by the SDLC phase where they would ordinarily occur during an iterative software testing lifecycle.

Security Testing Planning

Before any developed software code is handed to the testing team, firms should take steps to assure that the testing effort is adequately scoped based on the nature of the application and the inherent risks identified by the project team and/or the information risk management organization. Financial services firms should evaluate the following with respect to planning and program management:

- **Develop a Risk-Based Framework to Security Testing**: It is important to clearly define standards for security testing, based on the agreed-upon risk profile of the software development effort. For example, a firm may choose to allow an internal-only application that does not handle Non-Public Personal Information (NPPI) to have one or more "High risk" or "Severity Level 1" defects move into production, but that same firm may not allow a client-facing Web application to have even a single high risk defect in the released product.

    - Software applications under development should undergo an information risk assessment to determine the levels of risk, and by extension minimum security testing acceptance criteria.
    - Deviations in minimum acceptance criteria across different applications with different levels of risk should be documented via bug bars or other documented artifacts.
    - Alignment should be obtained with the development teams to triage and remediate identified security defects. Security defects should be addressed and remediated during the development cycle's quality toll gates, and, as discovered subsequently, during the application's "warranty" period or during overall lifecycle of the application post-implementation.

- **Involve Testers in the Design Phase**: Involving testers during a project's design phase – particularly during threat modeling exercises - gives testers valuable insight into how a system can conceptually be compromised. This added perspective can prove valuable to the testing team during the various testing phases.

- **Security Defects are "Defects," not just "Security Defects"**: As deviations from existing security requirements are identified or as security vulnerabilities are discovered during the testing process, these defects should be classified and grouped along with defects found on other system functional and system non-functional requirements. Severity levels previously defined for defects should be included in defect logs. Test summary reports should be escalated to management along with other defects to ensure they receive the appropriate level of visibility across key stakeholders. For example, a

defect within a client-facing, mortgage lending application that allows for elevation of privilege (which will likely be assigned a Severity 1), should be reported as a high risk to management along with defects that are found in the application's inability to properly calculate a potential borrower's loan-to-value (LTV) for a subject property.

- **Utilize Privatized or Sanitized Test Data**: Financial services firms should always avoid using live production data, particularly data uniquely associated with actual customers, for any testing purposes. Data used in all phases of the test cycle should either be completely fabricated, or appropriately obfuscated. Using live production data increases the exposure of a confidentiality breach, and could also introduce regulatory impacts to applicable laws, such as the Gramm-Leach-Bliley Act (GLBA), the standards such as the Payment Card Industry Data Security Standards (PCI-DSS), and the disclosure guidance of the Securities and Exchange Commission (SEC).

Decisions regarding masking of sensitive data that is moved from a production environment to a non-production environment (test, quality assurance, user acceptance test, development, etc.) for the explicit purposes of testing the functionality of the application are often determined based on either the classification and/or the risk score. Most organizations ensure their production environment controls include more rigor than non-production environments so that the movement of data outside of a production environment requires a de-identification (or de-personalization) process. Data de-identification can include:

- Data masking – identifying sensitive data like a Social Security Number and changing the values
- Data tokenization – changing the characters using a method that enables changing them back to the original characters using a key structure
- Obfuscation – changing the format and number of characters randomly

Data de-identification is often used in the testing phase of development but it can also be incorporated into the application itself to prevent sensitive information from being shared when it is not necessary. A common example of this technique applies to customer statements that only include the last 4 digits of an account or credit card.

- **Evaluate the need for Penetration Testing**: Certain high-risk financial systems, such as mobile banking or mobile payments systems, and even automated teller machines (ATMs), may benefit from a targeted penetration test. Consider however, that any penetration testing effort is only as comprehensive as the amount of time or money devoted to it and the competency of the tester. Consequently, firms should consider the expected value or benefit from conducting such testing, perhaps limiting it to only its highest-risk systems.

Security Testing Execution

Once testing requirements have been identified, the appropriate testing resources allotted and the application correctly deployed in a Quality Assurance (QA) or other non-production environment, firms can begin validating that security requirements have been effectively coded into the developed

software. What follows is an overview of a variety of test execution methods for financial firms to consider, based on both the application's architecture and its risk profile.

As mentioned previously, firms should ensure that the testing process is tightly integrated with the existing enterprise software development and testing framework, and where possible, should leverage existing tools to track results and defects to provide consistent reporting to all stakeholders.

- **Automated Testing**: Similar to traditional functional, integration, and/or client acceptance testing of non-security related requirements, testing results for security requirements are usually performed most consistently when automated tools are used. For example, a routine requirement such as input validation can be evaluated using a variety of automated tools, such as Hewlett Packard's QuickTest Professional (QTP) or Microsoft's AppVerifier, with a greater degree of assurance than manually testing input validation on every field and in every form that requests user-supplied input. Web application vulnerability scanners can also be useful in this regard (see Web Application Vulnerability Scanners, as discussed later in this section).

- **Static Code Analysis**: Static code analysis involves evaluating pre-compiled source code to evaluate conformance to identified security requirements. Because the code is not yet compiled or built, static code analysis is usually performed by the development staff, before the compiled code is released for testing.

  Examples of where static code analysis could be helpful can include identifying instances where:
  - Insecure or unauthorized APIs or language functions are referenced
  - Actual SQL statements being directly called in stored procedures
  - Code could be exploited to create memory buffer overflows

- **Dynamic Code Analysis**: As opposed to static code analysis, dynamic testing should be performed on post-compiled source code that is actually running. For this reason, dynamic testing is only capable of identifying defects on code while it is actually executed, as opposed to the entire code base available to static testers. Financial services firms should rely on their testing organizations to perform the dynamic code analysis. Care should also be taken to ensure that dynamic testing efforts achieve appropriate coverage across all the application's components (i.e., Web servers, databases, etc.). Dynamic code analysis should be performed, as appropriate per a firm's pre-defined risk assessment, on completed and/or compiled code for both client/server, desktop or mobile applications, and web-based applications.

  - **Client/Server & Compiled Code**: There are a variety of commercial and open-source dynamic testing tools available to firms, including *Fortify, HP DevInspect*, and *Veracode*. Microsoft also makes available several tools which can be used to dynamically test Win32/Win64 code, such as *Driver Verifier* and *Device Path Exerciser*.

  - **Web Application Vulnerability Testing**: Web application vulnerability scanner tools provide another form of dynamic code analysis for web-based applications, and

the reports they generate can provide a financial services firm a quick and relatively low-cost, high-level summary of security vulnerabilities detected by the tools. Commercially-available and freeware tools and offerings from IBM, HP, Qualys, WhiteHat Security, Burp, and many, many others are available, which financial services firms can both evaluate and deploy to identify security vulnerabilities within systems under development as well as production systems. These tools work by "scanning" the application's infrastructure, and by submitting arbitrary input in an automated fashion to attempt to identify any software or system security vulnerabilities. These tools then provide the findings as a report to be viewed and assessed by system administrators and developers for further action.

Financial services firms, however, should be cautious in relying on these tools as the sole basis for evaluating an application's security for the following reasons:

- Vulnerability scanners detect vulnerabilities using an almost signature-based approach, whereby the scanner evaluates an application for vulnerabilities by inputting the same, almost predictable data. This "false negative" rate makes it possible for a skilled attacker to identify or exploit basic Web application vulnerabilities not identified by the tool.
- Conversely, vulnerability scanners often also report false positives, whereby the scanner mistakenly reports a vulnerability that does not actually exist.
- Vulnerability scanners do not take into account any threat models identified by the project team, nor do they consider emerging threats (i.e., mobile banking, remote deposit image capture, or mobile payments threats).
- Vulnerabilities identified during a vulnerability scan often require code changes to remediate, followed by another vulnerability scan to validate. Performing other security testing activities, such as static dynamic testing, may have identified and remediated these vulnerabilities at a lower cost point.

Vulnerability scanning, where performed, should be performed during the integration testing phase, ideally before client acceptance testing (CAT) or user acceptance testing (UAT) is performed.

- **Fuzz Testing**: Fuzz testing involves the practice of providing unexpected input to an application during the testing phase, and evaluating how the application reacts as a result. For example, a Web application used between mortgage companies to evaluate and underwrite flood insurance applications may provide a means by which the lender can securely upload the relevant borrower loan documents in either MS Word .docx or Adobe .pdf format. A valid fuzz test of that functionality may involve re-naming a very large, multi-gigabyte file with a .docx or .pdf extension and attempting to upload that file as a document to see how the Web application responds. The application might reject the upload, but it may also accept the upload. In doing so, does the application become unresponsive to other requests, does the application crash due to limited disk space, and if so, does it provide any revealing errors to the end user which could be useful as part of another attack?

Firms should strongly consider integrating fuzz testing into their software development lifecycle, particularly for publicly-available, Internet-facing applications. Fuzz testing can be an extremely valuable tool in validating identified threat models or design assumptions, and can be particularly useful in validating application code developed by third-party software providers or business partners.

There are many automated fuzz testing tools and frameworks available, many of which are built to focus on application development languages or platforms, such as Python or Java. Many fuzz testers are also available as open-source software and free of charge.

Third-Party Application Testing

Applications developed or hosted by third-party software providers present challenges to testing teams, primarily because the testing team usually does not have access to the source code. The following table lists the different testing methods previously described, and how these can be performed (where applicable) with third-party applications:

Testing Type	Comments
Automated Testing	**Available**. Requires firm to have automated testing tools in place.
Static Testing	**Emerging Availability**. Generally cannot be performed without the application's source code. Reports can be requested from the software vendor or software provider. (A limited number of vendors now offer products that can perform a binary scan without access to the source code.)
Dynamic Testing	**Available**. Requires firm to have possession of the compiled binaries and libraries. Web-based applications require a vulnerability scanner, and the consent of the third-party (if hosted externally).
Fuzz Testing	**Available**. Results are better when the testers are creative, have appropriate tools, understand the business process, and any identified threat models.

Financial services firms may come across resistance from third-party service providers regarding receiving authorization to perform a vulnerability scan, due to confidentiality, availability, or other concerns posed by the provider. Firms may consider accepting the reported results of a recent vulnerability scan performed by the third-party, or the reported results conducted by an independent party. Firms should also impose contractual controls as part of their overall supplier risk management and due diligence process to ensure the provider is appropriately obligated to build and maintain appropriate security controls. Firms should use this Framework document as a basis for the types of processes and practices for building secure software that they should require of their providers.

Other challenges exist with third-party developed systems when it comes to assessing specific security requirements as imposed by financial services firms, such as encryption in transit. Without a network analyzer installed between system components and a cryptanalyst to perform a packet analysis, it can be difficult to determine if network communications are encrypted with an authorized encryption algorithm and key length.

Firms may also have standard security requirements imposed on third-party applications related to other information security domains, such as credential management (i.e., User ID/password lengths), or logging requirements. Firms should consider developing standard acceptance criteria for these requirements so that a testing team can more easily build customized test cases to validate compliance with these requirements. For example:

Standard Security Requirement	Acceptance Criteria
[PASSWORD LENGTH] All passwords must be at least eight characters in length.	While logged into the system, attempt to modify the user's password to a value containing fewer than 8 characters and confirm a system error returns and that the password change is disallowed.

- **Validate Test Coverage vs. Code Complete (Regression Testing)**: Changes often occur in the software's scope, features, or functionality during the development and testing process. As these changes are vetted and approved, it is important for security testing teams to carefully assess the nature of such changes against the security tests already executed. Do the changes impact any validated threat models, and should they be re-evaluated? Can the changes invalidate other security testing performed previously? Financial services firms should re-assess completed security activities, and perform regression tests as appropriate to re-validate existing findings.

  For example, if an investment bank is building a new internal system to manage trust accounts for its client base, and security testing for the application is nearly complete when the business requests that the application integrate with a data feed from another application that was previously out of scope, how should that new requirement impact security testing? Should a threat model be created around that new integration? Should fuzz testing tests be performed against the integration itself? Should specific new controls integrated as a result of the new requirements (e.g., encryption in transit, if required) be validated for this new integration?

Security Testing Reporting

Reporting security testing activities is critically important in measuring the success of the security testing effort. Reporting can provide valuable insight on shortcomings not only within the specific, targeted applications, but also across entire technology portfolios or even the entire enterprise. Moreover, because of highly-regulated nature of the financial services industry, useful reporting and metrics can become instrumental in validating the effectiveness of a security testing program to other parties including auditors and regulators.

Below are certain components of security testing reporting that financial services firms should focus on as they further build or mature their security testing efforts.

- **Validate Test Coverage vs. Requirements/Development Standard**: Software security teams at financial services firms must be able to effectively discern gaps between identified security requirements and development standards to what will actually be tested (see Development Standards section). One method of achieving this can be through a documented Testing Plan, where all stakeholders approve the testing methodology for the specific project or product prior to the code being delivered to the testing environments. The software security team should be a stakeholder or approver of the document, which along with other requirements could serve as a prerequisite for testing, and thereby help to assure that the appropriate security requirements will be evaluated.

- **Tracking Bug/Defect Remediation**: Once security testing execution is complete, the testing team must be able to provide information regarding any bugs or defects identified during the testing, along with a mechanism for tracking that defect until it has been remediated or approved, based on the severity level of the defect.

For example, assume testing reveals a brokerage application used to track dividends paid to investors provides detailed server, IP address, and database information to the end user when a fault occurs, in violation of security standards. This defect would be assigned a Severity Level and sent back to Development team for remediation. The software security team, or at a minimum the testing team, should have visibility into the status of this effort until the defect has been corrected. Commercial testing platforms such as *HP Quality Center* provide tools to allow security stakeholders and testers insight into such activities.

- **Reporting Defects**: When the testing cycle ends, a major goal is that defects that can be resolved in the application's upcoming release are in fact resolved. There may, however, remain one or more defects that cannot be remediated. Often, these are defects whose remediation requires more significant updates to the application code, which may impact delivery timelines. Once all security testing has been completed, final Defect Reports and Test Summary Reports should be prepared and distributed to all stakeholders, including the software security team, for analysis and approval, and used as an artifact to determine whether the release can move into the production environment. The primary consideration firms should have at this stage is an understanding as to whether there are any high-risk security defects remaining in the software code. Where such defects exist, firms should incorporate a risk-based approach to determine if the release will be allowed to continue into production. Firms should also consider whether a remediation plan or point upgrade can be deployed shortly after implementation.

- **Identifying Trends**: Effective security testing reporting can provide software security teams and managers valuable insight into the overall software security development lifecycle. Well thought out reporting and metrics in this area can be used to provide answers to many questions, including the following:
  - Are certain developers or product teams in need of additional security training?

- Is security training having a positive effect on the number of testing defects over time?
- Are certain types of defects more prevalent than others?
- How long does it take to remediate security defects by type?
- Are minimum requirements for regulatory compliance being met by product?
- How long does it take to perform security testing?

Chapter 6: Pre-Implementation Practices

Purpose

This section of the Framework provides a comprehensive list of requirements for software security assurance that are needed for the final production launch of applications.

Usage and Outcome

This chapter of the Framework applies during the "Release Phase" of a firm's Secure Software Development Lifecycle. Pre-Implementation Review (PIR) is a comprehensive examination of all security activities performed on software prior to its release. The PIR includes, but is not limited to, the review of threat models, bug or defect reports and reports from vulnerability assessment tools.

The PIR should result in one of the following three outcomes: Passed PIR, Passed PIR with exceptions and PIR with escalation.

- Passed PIR – Indicates that the review was conducted successfully with zero, or perhaps very minor, deviations from expected results as defined by the project team in collaboration with the security organization.
- Passed PIR with Exceptions – Indicates a conditional acceptance of the security results, and authorization to proceed into the implementation phase, under the condition that certain deviations or defects be mitigated or resolved, usually within a defined and agreed upon timeframe. For example, an internal .NET web-based application that does not store or process sensitive company or client data, whose database connection strings and passwords are stored in clear text in the web server's web.config file, may be allowed to proceed into production, so long as the connection string are encrypted in the file within two weeks of going into production.
- PIR with Escalation – The review identified significant deviations or security defects, which require escalation within the line of business to make a risk-based decision on whether or not to allow the system to proceed into production.

Any decision and relevant artifacts such as a checklist, provided in a PIR should be appropriately documented and maintained if required for independent review post-implementation.

The following checklist contains examples of considerations for use during the "Release Phase" of the Secure Software Development Lifecycle.

PIR Requirements Examples

#	**Requirement**
Security Requirements	
S1	All action items from the architecture and design review have been completed.
S2	All security defects reported from manual code review at the level of severity required have been fixed.
S3	All security defects reported from automated code review at the level of severity required have been fixed.
S4	All security defects reported from dynamic analysis at the level of severity required have been fixed.
S5	All security defects reported from penetration testing at the level of severity required have been fixed.
S6	Review threat models and ensure no new threats and vulnerabilities have arisen and all existing ones have b fully identified and mitigated.
S7	Does each unfixed security defect have an approved exception from management?
S8	Have all security settings turned on to the highest recommended level by default?
S9	Has all development data been purged from the system of record?
S10	Have all development and default service accounts been reissued and reassigned?
Privacy Requirements	
P1	All action items from the privacy review have been completed.
P2	Are all privacy related documentation/disclosures ready to launch along with the application?
Incident Response	
I1	Is a security incident response plan ready, tested and approved?
I2	Is a privacy incident response plan ready, tested and approved?
Launch Readiness	
L1	Necessary Change and Release Management formalities completed and approvals obtained?
L2	Network hardening completed in the production environment (as per approved network hardening standar
L3	Host hardening completed in the production environment (as per approved host hardening standard(s))?
L4	Is security monitoring implemented and tested in the production environment?
L5	Is a roll back plan ready, tested and approved?
L6	Is emergency patching process ready, tested and approved?
L7	Is the Disaster Recovery (DR) plan ready, tested and approved?

Chapter 7: Software Assurance Documentation Archive Best Practices

Documentation corroborates the process used to develop software securely, the outcomes and results of each phase in the lifecycle and the actions taken to correct identified defects. This documentation will be required by various constituencies including:

- Internal and external risk review individuals such as internal auditors, external auditors and regulators
- For third-party developed software, clients for whom the software was developed.

This section of the Framework describes the types of documentation for retention, as well as suggests processes for gathering and retaining the documentation.

Inventory

There must be an application inventory that includes technology and business risk associated with the application. This inventory should have the ability to assess multiple dimensional factors and determine the inherent risk of applications.

Ideally, the inventory tool should be able to support intrinsic information and metadata about the application whether internally developed, contractor developed, or developed by offshore resources. The tool should include information such as the following:

- Application architecture
- Transaction processing amounts (ranges)
- Application interdependencies
- Regulatory requirements/oversight
- Application components such as:
  - Code base
  - Code framework
  - Development methodology utilized
  - Source code control system
  - Control points for sign-off of the code/workflows for code delivery
  - Business process supporting documentation
  - Entitlements
  - Authentication models/tools utilized

If a single source tool does not meet these needs, then a federated model of systems of record should be used to obtain the information. Without the proper identification of the risk of the application, priority decisions on the software assurance lifecycle will not be aligned with business priorities and objectives.

For vendor purchased or hosted applications, the firm must still know the information about the application and the inherent risk of it. During the analysis phase of selecting software, the same items for an internal application must be obtained.

Lastly, the inventory must be maintained. A strong governance structure must be put in place to ensure the quality and accuracy of the data, including the key business stakeholders. Regular attestation cycles must be managed and governed to ensure data quality. Firms should leverage the information risk management of other Governance, Risk and Compliance (GRC) tools at their disposal to effectively assess and manage applications' risk in their enterprise. The implementation and use of these tools are outside the scope of this paper.

Documentation for Stages of Software Development

Throughout all of the stages of software development, artifacts should be maintained and available to the personnel who require access to the information. Hence, the process of documenting must be fully embedded within the software development lifecycle with checkpoints to ensure that the documentation is in place prior to proceeding to the next phase in the lifecycle.

As discussed below, there are key elements that should be maintained per application, preferably in the inventory or in a documentation management system. Security documentation must be included with all other documentation – it should not be maintained separately from the rest of the application. Institutions should, however, consider placing higher restrictions on access to security-related documentation. Documentation can be placed into traditional word processing or spreadsheet applications. However, when requirements are very complex, specialized requirements management tools may be warranted.

There are key document requirements that should be maintained to ensure proper quality and transparency. The process of documenting, including security documentation should be fully integrated into all phases of the development lifecycle and the documents should be side-by-side with all other application documentation.

Institutions should consider a requirement wherein they verify the documentation requirements from the previous stage have been met and verified in the application inventory before allowing movement to the next stage of the software development life cycle (e.g., gating at the appropriate stage).

Business Requirements

Business Requirements define the reasons that the application exists – they are the foundation for what will be implemented. Business analysts familiar with business needs should create the requirements and collaborate with the development teams to ensure they can be met. Without proper requirements documentation, software changes become more difficult—and therefore more error prone (decreased software quality) and time-consuming (expensive).

If the software is a first release that is later built upon, requirements documentation is very helpful when managing subsequent changes of the software and verifying that nothing has been broken in the software when it is modified.

If vendor-purchased code is sought, the business requirements should be part of a Request for Proposal.

Application Purpose and Expected Outcomes

The reason for the application's existence must be properly documented so that functionality requirements of the business are noted. It is important to clearly understand these requirements so that during testing phases the requirements are tested.

Security Property and Attribute Requirements

The expected security requirements of the application must be documented and integrated into the business requirements. Security documentation cannot be a stand-alone effort, but instead, must be integrated with the business requirements.

Entitlements

Expected functionality, including entitlements should be part of the business requirements. Periodic access control reviews typically rely on the accuracy of this information.

Functional Design Specification

This phase of the secure software development lifecycle utilizes the business requirements to develop a conceptual model of the application. Here, the business analysis and design teams work to design the functional specification for the application. This documentation should contain details regarding the workflow, architecture and overall topology as it relates to the underlying infrastructure.

Detailed Design/Architecture Specification

These documents do not describe how to program a specific routine, but indicate why the routine exists. They describe all inputs, outputs and possible data formats. Key elements that should be included are:

- Entitlement Models and Solutions Utilized (e.g., home grown or standard central corporate solution)
- Expected interfaces with other applications/utility applications
- Data Models/Relational database schema
    - Entity Sets and their attributes
    - Relationships and their attributes
    - Tables, attributes, and their properties
    - Views
    - Constraints such as primary keys and foreign keys

Threat Modeling

The finalized threat model must be maintained in the document archive facility. The threat model should document key elements of the threat analysis such as account the type of code, development framework, algorithms, interfaces, APIs and underlying infrastructure. The

application artifacts/documentation archive should be taken into account when developing the threat model.

Test Plans

The functionality and security test plans should be derived from the design specifications and threat model. Details of the test plans should be stored in a centralized repository accompanied with results and sign-off by the testers for all of the use cases.

Output of the test cases, which should include formal outcomes linked to defects in a defect tracking system tied to risk treatment, should also be part of the documentation.

Defect Tracking

Defect tracking is a key component of the documentation. Security defects are a subset of a functionality defect. Care should be taken to limit detailed defect knowledge to those who need to know, combined with a corporate policy that prohibits the distribution of this information, unless explicitly provided by an approved level of management.

In addition, defects can also be a powerful tool in the measurement of quality code from developers. Accurate information about the code defect is critical when this information is used in performance management activities for the developers.

A key element of defect tracking should also include the "cost to fix." If this is not measured, the ability to illustrate the cost savings of addressing vulnerabilities early in the development life cycle cannot be tied to actual data of the firm.

When dealing with suppliers, there should be formal documentation that attests to the quality of the software delivered. Any third party code defects should also be tracked in the same manner so that these can be utilized for overall supplier management, particularly during maintenance renegotiations and contract renewals.

Risk Mitigation and Acceptance

The documentation library should contain a repository to track and acknowledge the risks and mitigations or accepted risk levels. Any risk acceptance should be aligned with the acceptable risk tolerance of the firm. Personnel at the business or in centralized risk management functions who may sign off on the appropriateness of the level of residual risk should be pre-identified. Automated workflows should be used to assure approval by these key personnel. However, if the risk crosses multiple business lines and can have an enterprise impact, it must be agreed upon by a corporate governing body due to the potential impact to the firm as a whole.

Certificates of Testing

If independent firms are used for activities such as ethical hacking, they should provide written documents of their results – often in the form of certificates. There should be a central point of contact for certificates from the testing. In addition, a documented request

and release process should be maintained to limit distribution to those who need to know, including the appropriate level of detail to provide.

Contract Language

Contracts with third party suppliers of software should allow for the serviced organization to conduct reviews of the practices of the supplier. This Framework can provide the basis for conducting such reviews. Serviced organizations should assess the secure development processes of their suppliers against the recommended practices in each area of the framework.

Standard expectations for quality from third party suppliers of code must be maintained within a centralized group. Once created, employees must be educated on who to contact for the standard language and the legal areas to work with. If there are any areas where the third party firm is not willing to agree to terms, this should follow a standard risk treatment, aligned with the risk that this could pose to the firm.

Metrics

Meaningful metrics regarding the application secure software development lifecycle must be created and reported to various levels of management. Examples of metrics include:

- Top issues identified
- Issues by Development Manager
- Issues dimensioned to the business criticality of the application
- Cost to remediate, tied to the stage of the development life cycle
- Developer strengths/weaknesses determined by the software defects and training programs

Disposal

The desired retention period for all documentation should be established at the beginning of the application and adhered to. In addition, the disposal methodology for the documentation at application retirement must also be considered and noted in the user manual for the application.

Chapter 8: Post Implementation Phase Controls

An application, whether developed internally, via contract with a third party or acquired via purchase, eventually makes its way into an organization's internal or third party hosted production environment.

Once implemented, applications then enter the Post Implementation Phase of the Software Development Lifecycle (SDLC). There are significant post implementation activities required to address software security. Some of these are already in practice, but there are several new practices that are emerging as mainstream practice.

This section on Post Implementation addresses both mature (established) and emerging (new) practices that are not in widespread use, but represent a potential opportunity to enhance the software security ecosystem.

Established Practices

Defense in depth approaches to software security require multiple layers of controls within the SDLC. Recent technology innovation has enabled controls that are applicable to the early phases of the SDLC to improve effectiveness in risk management, increase efficiencies, and reduce operating costs. Multiple layers of controls also provide opportunities to "test" controls for effectiveness. For example, the use of dynamic scanning technology in Q/A following the use of static (source code) analysis in the development phase helps to determine the effectiveness of the static analysis control and the effectiveness of threat modeling in the design phase. Penetration testing beyond these other layers offers an additional opportunity to determine if the controls applied earlier in the SDLC are effective (both preventative and detective). Using a layered approach to detective controls in the development process achieves defense in depth and enables application developers to think of security vulnerabilities identified simply as *software defects* that require work to resolve. Encouraging those responsible for development to treat all security vulnerabilities found as software defects that need to be fixed according to priority (risk) is an indicator of maturity in software security. The fundamental difference between applying detective controls *prior* to production implementation and applying detective controls *within* production is in how the results should be treated or handled. Security vulnerabilities identified **during** the development and Q/A processes should be addressed simply as software defects. Software development teams, whether working at a vendor or within a company, already know how to deal with software defects. Developers may require some help prioritizing defects (based on risk), but they already know how to fix defects.

Security vulnerabilities identified in applications in production, whether within a firm or at a third party, should ***not*** be treated as software defects but as one part of the organization's security incident response process. By contrast then:

- Vulnerabilities discovered in development and Q/A are defects.
- Vulnerabilities detected in production should be treated as security incidents that require immediate attention for corrective action (remediation) and an opportunity to determine

if adjustments to the secure software development lifecycle are necessary to prevent future repetition of issues.

Applying this notion (defects in development, incidents in production) may represent a level of maturity that some organizations are not prepared to implement. If the organization is not ready for this approach then a target date for implementation should be established to work toward this level of maturity.

Emerging Practices

Technology advances and software security program maturity present an opportunity to significantly improve application security by treating the organization's Web infrastructure and IT infrastructure as an ecosystem filled with information that can be mined to determine specific threat trends that are unique to the ecosystem. Some organizations consider this practice area as *Software Security Intelligence*, one part of the traditional security intelligence function that exists to focus on the specific threats to specific Web applications within the organization's Web complex.

The technology advancements include software and devices designed to monitor, and in some cases prevent, security threats within the production environment. In addition, tools to detect the presence of malware on Web applications or spread through banner ads and pop-ups have emerged as useful production monitoring capabilities. Another category of tools support the detection of security vulnerabilities in Web applications running in production. The software includes commercial or custom software components designed to closely monitor Web application behavior within production and identify potential security threats and exploits. The hardware and software appears in the form of application firewalls designed to operate in a passive or protective mode, running rules that interpret application behavior in real time and either report on or block certain application behavior deemed to be malicious based on the application firewall rules.

Both of these very different technology components share one common characteristic – they offer real time monitoring of threats within a Web complex to observe potential exploits and learn how threat attack vectors evolve. The threat trend information from these sources is highly valuable and should influence the following:

1. Threat models used in application design
2. Rules used for static analysis tools
3. Types of dynamic scanning used
4. Techniques used in penetration testing

All of these controls today are based on either OWASP Top 10 or SANS Top 25 Application Programming Errors. Both sources are valuable, but both represent an earlier generation of software security intelligence. They represent commonly exploited vulnerabilities across the Web landscape for all organizations and they have had a profound effect on improving software security programs over the past 5-6 years.

The point is, when you want to know whether to bring your umbrella to work or not, do you look at a weather satellite image of North America from last year or do you look at the local weather forecast for your location? The latter helps you answer the question, the former is simply

interesting. Knowing the threat trends within a firm's ecosystem is significantly more valuable than knowing the most commonly exploited vulnerabilities across all organizations. Intelligence from a specific ecosystem will demonstrate, over time, that threat trends have some unique characteristics that inform administrators on how to tune and adjust the controls for that ecosystem's effectiveness. In actual use, some parts of a firm's ecosystem will indicate different threats from other parts. Consumer banking threat models, for example, are very different from Institutional banking threat models and new threats are better targeted and more serious.

After web applications have been deployed in production there are many ways to monitor the applications to determine the types of threats that occur. Threat trends may vary from one organization to another and across different types of web applications. Understanding the types of threats that are likely for specific applications, enables organizations to make adjustments in their controls imbedded within the SDLC (threat models, static rules, pen test techniques, test scripts, etc.) based on specific threats within an organization's ecosystem that can ultimately discourage hackers and make other targets more desirable.

In the future, software security intelligence may emerge as a foundational component of software security programs as the maturity of tools to measure threats and the need for threat intelligence improves and matures.

Acknowledgements

We thank the following members for their contributions to this Framework:

Jim Apple .....Bank of America
Lester Bain .....Capital One Corporation
Jonathan Beck.....PNC
Jamie Bowser .....Key Bank
Jill Brungardt.....Commerce Bank
Michele Cantley .....Regions Bank
Michael Fabrico.....NASDAQ/OMX
Even Gaustad .....US Bank
Diallo Gentry.....HSBC
Kostas Georgakopoulos ....NASDAQ/OMX
Dana Guild.....State Farm
James R. Hinsey .....Wells Fargo
James Killian .....Regions
Greg Kyrytschenko.....NASDAQ/OMX
Al Lopez.....LPL Financial
Kendall Mahan .....State Farm
Lisa E. Matthews.....Bank of America
Mike McCormick .....Wells Fargo
Tim Mullins.....BNY Mellon
Melissa Netram.....Intuit
Bryan Orme .....Capital One Corporation
John Rouse.....Huntington
Ken Schaeffler .....Comerica
David Robert Smith.....Fidelity
Robert Ward .....ING Direct USA

We especially want to acknowledge the following individuals who took a leadership role in developing various sections of the Framework:

Carlos Batista .....SunTrust Banks, Inc.
Doug Cavit.....Microsoft
Mahi Dontamsetti .....DTCC
Susan Koski .....BNY Mellon
Mark Merkow .....PayPal
Laksh Raghavan .....PayPal
Jim Routh .....JPMorgan Chase & Co.

About BITS

BITS addresses issues at the intersection of financial services, technology and public policy, where industry cooperation serves the public good, such as critical infrastructure protection, fraud prevention, and the safety of financial services. BITS is the technology policy division of The

Financial Services Roundtable, which represents 100 of the largest integrated financial services companies providing banking, insurance, and investment products and services to the American consumer. For more information, go to http://www.bits.org/.

Appendix A – Education & Training

Application Software Security
Developer Awareness Course Outline

This course is intended for designers, developers, and testing staff

1.0 Security Activities in the SDLC
- 1.1 Security begins from within
- 1.2 SDLC Phase 0 - Developer Training
- 1.3 Requirements gathering and analysis
- 1.4 Systems design
  - 1.4.1 Functional decomposition
  - 1.4.2 Categorizing threats
    - 1.4.2.1 Ranking threats
    - 1.4.2.2 Mitigation planning
- 1.5 Design reviews
- 1.6 Development phase
- 1.7 Testing phase
- 1.8 Deployment phase

2.0 Proven best practices for secure software development
- 2.1 What are software best practices?
- 2.2 Basic concepts
  - 2.2.1 Attack surface Security perimeter
- 2.3 Best practices for secure development
  - 2.3.1 Apply Defense in Depth
  - 2.3.2 Use a positive security model (Whitelisting)
  - 2.3.3 Fail securely
  - 2.3.4 Run with least privilege
  - 2.3.5 Avoid security by obscurity
  - 2.3.6 Detect intrusions
  - 2.3.7 Don't trust infrastructure
  - 2.3.8 Don't trust services
  - 2.3.9 Establish secure defaults

3.0 Design Phase Security Activities
- 3.1 Recommendations for secure software designs
  - 3.1.1 Misuse case modeling
  - 3.1.2 Design and Architecture reviews
  - 3.1.3 Threat and risk modeling
  - 3.1.4 Risk analysis
  - 3.1.5 Security requirements/test case generation
- 3.2 Design to meet security requirements
- 3.3 Design patterns

- 3.4 Architecting Web applications

4.0 Vulnerability Taxonomies
- 4.1 Overview of common taxonomies
- 4.2 Why are different views of threats, vulnerabilities, and errors needed?
  - 4.2.1 OWASP Top 10 – Vulnerability view
  - 4.2.2 CWE/SANS Top 25 Most Dangerous Programming Errors – Developer error view
  - 4.2.3 Web Application Security Consortium (WASC) Threat Classification – Threat view
- 4.3 Taxonomies compared

5.0 Current Version of OWASP Top 10

6.0 Current Version of CWE/SANS Top 25 Most Dangerous Programming Errors
- 6.1 Module 6 Review

7.0 Current Version of WASC Threat Classification

8.0 Programming Best Practices
- 8.1 Input validation and handling
  - 8.1.1 Input sanitization
  - 8.1.2 Canonicalization
  - 8.1.3 Handling bad input
- 8.2 Avoiding Cross-Site Scripting attacks
  - 8.2.1 Same origin policy
  - 8.2.2 Prevention techniques
- 8.3 Preventing injection attacks
  - 8.3.1 SQL injection
  - 8.3.2 Identifying SQL injection attacks
  - 8.3.3 Preventing SQL injection attacks
- 8.4 Authentication and Session Management
  - 8.4.1 Attacking log-in functionality
  - 8.4.2 Attacking password reset functionality
- 8.5 Avoiding Cross-Site Request Forgery (CSRF)
  - 8.5.1 Mitigating CSRF attacks
- 8.6 Session Management
  - 8.6.1 Attacking log-out functionality
  - 8.6.2 Defending against log-out attacks
  - 8.6.3 Defending against cookie attacks
- 8.7 Access Control
  - 8.7.1 Access control issues
  - 8.7.2 Testing for broken access controls
  - 8.7.3 Defending against access control attacks
- 8.8 Cryptography
  - 8.8.1 Hashing and password security
  - 8.8.2 Attacking hashes

- 8.9 Error Handling
  - 8.9.1 User error messages
  - 8.9.2 Developer error messages
- 8.10 Module 8 review

9.0 Testing
- 9.1 Source code analysis
  - 9.1.1 Automated analysis
  - 9.1.2 Tools for automated analysis
- 9.2 Black Box Testing
  - 9.2.1 Automated analysis
  - 9.2.2 Tools for automated analysis
- 9.3 Code review process
  - 9.3.1 Manual source code reviews
  - 9.3.2 Peer reviews
- 9.4 Integration testing

10.0 Enterprise Security API (ESAPI)
- 10.1 Overview of ESAPI
- 10.2 Interface Encoder
  - 10.2.1 Overview
  - 10.2.2 Information flows
- 10.3 Interface User
  - 10.3.1 Overview
  - 10.3.2 Information flows
- 10.4 Interface Authenticator
  - 10.4.1 Overview
  - 10.4.2 Information flows
- 10.5 Interface Access Controller
  - 10.5.1 Overview
  - 10.5.2 Information flows
- 10.6 Interface Access Reference Map
  - 10.6.1 Overview
  - 10.6.2 Information flows
- 10.7 Interface Encryptor
  - 10.7.1 Overview
  - 10.7.2 Information flows
- 10.8 Interface HTTP Utilities
  - 10.8.1 Overview
  - 10.8.2 Information flows
- 10.9 Interface Logger
  - 10.9.1 Overview
  - 10.9.2 Information flows
- 10.10 Module 10 Review

BITS Software Assurance Framework
Appendix A

Application Software Security
Foundation Awareness Course Outline

This course is intended for everyone involved in the SDLC – it assumes no prior knowledge and establishes the foundation for further education in software security.

1.0 Enduring Truths About Software
- 1.1 Business user perspective
  - 1.1.1 What business users buy when they ask for software
  - 1.1.2 What business users get when they ask for software
  - 1.1.3 Ignoring security requirements and their consequences
- 1.2 Developer perspectives
  - 1.2.1 What developers hear when they're asked to develop software
  - 1.2.2 Get it out the door now, fix it later
  - 1.2.3 Consequences of ignoring security controls in software development
- 1.3 A delicate balance of time, cost, scope, and quality
  - 1.3.1 Good, cheap, or fast – pick two
  - 1.3.2 Casting off security requirements to meet time to market constraints
- 1.4 Industry, Federal, and State Regulations related to secure coding
  - 1.4.1 Industry Standard: PCI-DSS
    - 1.4.1.1 Overview
    - 1.4.1.2 Secure code requirements
  - 1.4.2 Federal Regulation: HIPAA
    - 1.4.2.1 Overview
    - 1.4.2.2 Secure code requirements
  - 1.4.3 Federal Regulation: Sarbanes-Oxley Act Sections 404 and 409
    - 1.4.3.1 Overview
    - 1.4.3.2 Secure code requirements
- 1.5 Introduction to Software Security Concerns
  - 1.5.1 Uncontrolled access
    - 1.5.1.1 Consequences of unleashing programs on public networks to anonymous users
    - 1.5.1.2 Questioning the trust of users
  - 1.5.2 Bad code
    - 1.5.2.1 Flawed assumptions about how users will use publically accessible software
    - 1.5.2.2 Failures in testing for unexpected conditions
    - 1.5.2.3 Failures that lead to flawed security control implementations
  - 1.5.3 Unaware and uneducated developers
    - 1.5.3.1 Developers only learn to make a language work
    - 1.5.3.2 Developers don't understand security or non-functional requirements
      - 1.5.3.2.1 Security requirements overview
      - 1.5.3.2.2 Non-functional requirements overview

2.0 Software Security Principles and Requirements
- 2.1 What are software security principles?
  - 2.1.1 The concept of principles

BITS Software Assurance Framework
Appendix A

Application Software Security
Management Awareness Course Outline

This course is intended for managers within any phase of the SDLC

1.0 Building Security In Application Software Development Processes
- 1.1.1. Review of Traditional SDLC Phases
- 1.1.2. Secure Software Development Lifecycle Overview
  - 1.1.2.1. Project Initiation
  - 1.1.2.2. Requirements Gathering
  - 1.1.2.3. Requirements Analysis
  - 1.1.2.4. Threat Modeling
  - 1.1.2.5. System Design
  - 1.1.2.6. System Development
  - 1.1.2.7. System Testing
  - 1.1.2.8. User Acceptance Testing
  - 1.1.2.9. Operations and Maintenance

2.0 Software Security Requirements
- 2.1 What are quality or nonfunctional requirements?
- 2.2 Security requirements families
  - 2.2.1 Identification Requirements
  - 2.2.2 Authentication Requirements
  - 2.2.3 Authorization Requirements
  - 2.2.4 Immunity Requirements
  - 2.2.5 Integrity Requirements
  - 2.2.6 Intrusion Detection Requirements
  - 2.2.7 Non-repudiation Requirements
  - 2.2.8 Privacy Requirements
  - 2.2.9 Security Auditing Requirements
  - 2.2.10 Survivability Requirements
  - 2.2.11 System Maintenance Security Requirements
- 2.3 Eliciting Requirements
- 2.4 Documenting Requirements

3.0 CLASP (Comprehensive, Lightweight Application Security Process)
- 3.1 What is CLASP?
- 3.2 Roles for Software Development
  - 3.2.1 Architect Role
  - 3.2.2 Designer Role
  - 3.2.3 Implementer Role
  - 3.2.4 Project Manager Role
  - 3.2.5 Requirements Specifier
  - 3.2.6 Security Auditor
  - 3.2.7 Testing Analyst
- 3.3 CLASP Best Practices for Software Security
  - 3.3.1 Institute awareness programs
  - 3.3.2 Perform application assessments

BITS Software Assurance Framework
Appendix A

Stakeholder	Curriculum	Delivery Type	Owner
Risk Managers	Foundations of Software Security	E-Learning	
	Risk-Based Security Testing Strategy	E-Learning	
	Database Security - Basics	E-Learning	
	Secure Baselines and Compliance	E-Learning	
	Monthly - Small Group meetings	In-Person	
	PDF Changes	E-Learning	
	PLC LOB Specific	E-Learning	
	Remediation Tracking	Workshop	
Application Security Champions/Mavens	Foundations of Software Security	E-Learning	
	Defensive Programming for JavaEE or Defensive Programming for C/C++ or Defensive Programming for C# for ASP.NET	E-Learning	
	Architecture Risk Analysis	E-Learning	
	Product Specific Training	E-Learning	
	Static Code Training	E-Learning	
	Enterprise Security Portal	Instructor Lead/Story board/Recorded	
	OWASP Top-10	E-Learning	
	Attack and Defense	E-Learning	
	Threat Modeling	E-Learning	
	Risk-Based Security Testing Strategy	E-Learning	
	Intranet Articles/Discussion boards	SharePoint	
	Monthly - Small Group meetings	On-line Forum	
	SDLC Changes	E-Learning	
Advanced Champions	GSSP - Java/C++/.NET	Instructor Lead/Recorded	
	CSSLP	Instructor Lead/Recorded	
	CISSP	Instructor Lead/Recorded	
	Intranet Articles/Discussion boards	Community portal	
Technical Security Officers	Foundations of Software Security	E-Learning	
	Defensive Programming for JavaEE or Defensive Programming for C/C++ or Defensive Programming for C# for ASP.NET	E-Learning	
	Threat Modeling	E-Learning	
	Risk-Based Security Testing Strategy	E-Learning	
	GSSP - Java/C++/.NET	Instructor Lead/Recorded	

BITS Software Assurance Framework
Appendix A

Stakeholder	Curriculum	Delivery Type	Owner
	CSSLP	Instructor Lead/Recorded	
	Architecture Risk Analysis	E-Learning	
Developers	Foundations of Software Security	E-Learning	
	Tools Training	E-Learning	
	Intranet Articles/Discussion boards	SharePoint	
	OWASP Top-10	E-Learning	
PMO	PDF Changes	Workshop	
	PLC LOB Specific	Workshop	
	Training required for Firm Wide changes	E-Learning	
	Intranet Articles/Discussion boards	SharePoint	
QA	Foundations of Software Security	E-Learning	
	Intranet Articles/Discussion boards	SharePoint	
	Risk-Based Security Testing Strategy		
Risk Comm.	Periodic Presentations- Guest Speakers	In-Person	
DBAs	Fundamentals of Database Security	E-Learning	
	Advanced Oracle Security	E-Learning	
Architecture	Foundations of Software Security	E-Learning	
	Defensive Programming for JavaEE or Defensive Programming for C/C++ or Defensive Programming for C# for ASP.NET		
	Architecture Risk Analysis		
	FAST/Security Tools Integration	Presentations	
Security Intelligence Staff		Presentations	
Developer Leads	Foundations of Software Security	E-Learning	
	Architecture Risk Analysis	E-Learning	
Mobile Developers	Mobile Application Security Overview	E-Learning	
	Secure BlackBerry Development	E-Learning	
	Secure iPhone iPad Development	E-Learning	
	Secure Windows Mobile Development	E-Learning	
	Using Static analysis for Java Apps on Androids	E-Learning	
	Threat Modeling for Mobile Applications	Workshop	

Additional Resources

- CERT Software Engineering Institute Software Assurance - http://www.cert.org/work/software_assurance.html
- Microsoft® Security Development Lifecycle - http://www.microsoft.com/security/sdl/default.aspx
- The Open Web Application Security Project (OWASP) – https://www.owasp.org/index.php/Main_Page
- The Build Security In Maturity Model (BISSM) - http://bsimm.com/
- *Building Security in Maturity Model* (version 2) May 2010. Gary McGraw, Ph.D., Brian Chess, Ph.D., & Sammy Migues
- *Secure and resilient software, requirements, test cases, and testing methods.* (2011). Merkow, M. S., & Raghavan, L., Auerbach Publishers

