Secure Coding.
Practical steps to defend your web apps.

Copyright SANS Institute
Author Retains Full Rights

This paper is from the SANS Software Security site. Reposting is not permitted without express written permission.

Interested in learning more?
Check out the list of upcoming events offering "Defending Web Applications Security Essentials (DEV522)"
at http://software-security.sans.orghttp://software-security.sans.org/events/
Introduction

The financial services industry (FSI) maintains high-value assets and typically operates in a very complex environment. Applications of all types—web applications, mobile applications, internal web services and so forth—are being developed quickly in response to market pressures by developers with limited security training and with relatively immature processes to support secure application development. This combination presents a juicy target for attackers, and data shows that the FSI continues to be a top target. Attempts to introduce security into the application life cycle frequently face challenges such as a lack of available application security expertise, concerns about costs for tooling, and a fear among product owners that security processes might impede the development cycle and slow their response to market conditions. This paper explores why the applications are being targeted, what is motivating the attackers and what some inhibitors of application security are. Most important, this paper specifies some best practices for developing a secure development life cycle to safeguard applications in the FSI.
Organizations across all industries are spending big dollars to protect the perimeter of their networks. At the same time, they frequently make their applications available over the web, punching holes in their perimeter to allow their applications to be served worldwide. The danger is that the resources dedicated to securing applications don’t come close to matching those focused on securing the perimeter. Attackers have noticed this. Software applications of all varieties (web, mobile, web services, etc.) continue to be a popular target for attackers. Among the most common targets for these attackers are organizations in the FSI. There are many reasons for this.

Financial Motivation

The FSI encompasses multiple subsectors, including banking, asset management, insurance, venture capital and private equity. What depository institutions, investment advisers, credit and financing organizations, fund managers and insurance companies have in common is, simply, money. The finance and insurance components of the FSI alone contributed almost $1.3 trillion to the 2015 U.S. gross domestic product, or about 7.2% of the total gross domestic product (GDP) (see Figure 1). And money is a great motivator for all types of people, including cyber attackers.

---


4 Bureau of Economic Analysis, U.S. Department of Commerce, www.bea.gov/Table/Table.cfm?ReqID=51 - reqid=51&step=2&isuri=1 [Registration required.]
Other Motivations

Financial gain is not the goal of all attackers, but the prominence of the FSI in the U.S. economy makes it a highly visible target for many of them. Nation-state attackers and cyber terrorists target the FSI because it provides the mechanisms of commerce for all industry. For “script kiddies,” attacking the FSI is a ticket to notoriety.

Regulations and Compliance

Besides wanting to protect their assets, FSI organizations are driven toward application security by the need to comply with regulations.

Given the FSI’s central role in the economy and people’s lives, the sector is highly regulated, with the risk from cyber attack gaining greater regulatory attention. The Gramm-Leach-Bliley Act, the Dodd-Frank Wall Street Reform and Consumer Protection Act, and the Sarbanes-Oxley Act all contain security requirements on how FSI applications must be designed, developed and tested. Nongovernmental regulation also plays a role; the Payment Card Industry Data Security Standard provides requirements for organizations that process payment cards.

New regulations continue to arise. The Federal Deposit Insurance Corporation recently announced proposed requirements that would cover five categories of cybersecurity: cyber risk governance, cyber risk management, internal dependency management, external dependency management, and incident response, cyber resilience and situational awareness. While the new Congress and president are showing hostility toward regulations, it may be expected that the FSI will continue to be one of the more heavily regulated industries (see Figure 2).

Although managing compliance can be time consuming, the process does force organizations to dedicate funds to support security initiatives. They should be careful, however, not to equate satisfying regulations with being secure.

While some forces drive the FSI toward taking greater security measures, others complicate the situation.

**Complexity**

FSI organizations have highly complex environments. Their applications frequently need to interface with multiple geographically distributed systems written in different languages and communicating over a range of protocols. The applications manage large volumes of complex data across multiple organizations, and speed and integrity are critical. And the applications themselves are complex, written in multiple languages and using multiple libraries, as well as using free and open source software (FOSS) served across multiple platforms. Unfortunately, complexity is the enemy of security, and complex applications are prone to vulnerabilities (see Figure 3).

---


---

**Figure 3. FSI Organizations and the Applications They Use Are Complex**
Market Pressures and Development Velocity

The market pressures that are driving software development life cycle (SDLC) methodologies in turn affect how the FSI performs its security testing. When software applications are being developed very rapidly, it can be difficult to insert sufficient security processes without affecting the development velocity adversely.

For many years, the FSI followed the traditional waterfall approach to software development. This frequently led to late deliveries of software that did not meet market needs, often because the requirements were poorly understood and communicated or had evolved during the long course of development. In this slow process, application security was injected only at the end of the life cycle, if it was included at all.

In response to market pressures to speed up development, many organizations have migrated toward Agile processes and continuous deployment. They are finding that these methodologies are a poor fit with traditional security assessments that come at the end of the life cycle. The traditional heavyweight testing that was done at the end of a waterfall life cycle didn't work in the agile, lighter-weight development iterations.

As a result, security assessments are increasingly being incorporated within the development life cycle itself, but they must be adapted to integrate with fast development life cycles. One way to do this is to use automated scanning tools, although they can be difficult to implement in such environments. That leaves many FSI organizations asking themselves how best to ensure the security of their applications in a timely manner.
While complexity and market pressures can impede efforts to develop secure applications, a properly developed secure software development life cycle (SSDLC) can overcome these obstacles. The core fundamentals of an SSDLC—training, risk assessments, secure design and testing—remain, but some adaptations need to be made, especially when deployed in rapid development environments.

**Personnel and Training**

At the heart of the SSDLC are properly trained personnel. As the Research and Development Committee of the Financial Services Sector Coordinating Council (FSSCC, part of the U.S. Department of Homeland Security) advised: “Educate software developers on secure development techniques because better coding practices are needed to reduce the number of software flaws in the long term.”

But training shouldn’t be restricted to software developers; all participants in the development process should be trained on security matters. At a minimum, software developers should be trained on common vulnerabilities such as the SANS Top 25 or the Open Web Application Security Project (OWASP) Top 10 and on core security concepts such as least privilege, attack surface, confidentiality and integrity. Members of the quality assurance (QA) team should receive similar training, with a few key differences. QA personnel need not know how to fix a vulnerability, but they should be able to conduct basic security tests. For example, basic parameter tampering should be within the range of testing capabilities of QA engineers. Managers, project managers and other leaders should also be versed in security concepts so that they can help positively drive security into the life cycle.

Providing security training across the different roles of personnel associated with the SSDLC provides a number of benefits. Perhaps most important, it helps cultivate a security-conscious environment for developing the applications. When all are aware of security issues to some degree, achieving buy-in to address security issues becomes less burdensome. Once trained, security-aware personnel in different roles can look at the security of developed applications from different perspectives and be more capable of asking security questions that may illuminate issues. Finally, well-trained team members will be able to independently handle many simpler security issues, freeing the security team to focus on more complex or systemic concerns.

---

Software Security Requirements and Security Architecture

At the outset of the SSDLC of each application, key elements are developing security requirements and a security architecture. A risk analysis performed early in the life cycle, combined with the security governance of the organization, will drive the security requirements for the application. Security experts and software architects must work closely together in developing the software security architecture, ensuring that it is centralized and simplified. When all security decisions run through a small, centralized kernel, there is less likelihood that a security decision (such as an authorization) will be omitted, and the development team can build a secure app without giving much thought to the security mechanisms—they are already in place, and the code is secure by default.

Incorporated in the development life cycle early on, security requirements and architecture will be able to address, at a minimum, authentication, authorization (both vertical and horizontal), data storage and data transfer, and they can even expand, depending on the intended functional requirements of the application. These early life-cycle steps are especially critical when developing in a fast-paced development environment. When the end-of-life-cycle testing time is limited in these fast-paced environments, the importance of the early life cycle becomes all the more important.

Threat Modeling

With the SSDLC, security is also enhanced by using threat modeling as part of continuous risk assessment. Threat modeling is a structured approach to identify and address the security risks associated with an application. This is done in a few high-level steps. First, the application needs to be analyzed and broken into its components to ensure a full understanding of how the application works. Then various potential threat actors are identified, along with threat categories for the application, such as authentication, authorization, data storage and so forth. Threats against these categories are then identified from varying attacker perspectives. Finally, countermeasures to thwart the threats are identified (see Figure 4).

At the outset of the SSDLC of each application, key elements are developing security requirements and a security architecture.
This threat-modeling process can help identify the attack surfaces, vulnerabilities and potential attack paths so they can be mitigated or eliminated well before deployment. The goal of this process is to identify and minimize your applications’ attack surfaces and ensure that mitigating controls are in place where attack surfaces exist. Threat modeling and continuous risk assessment keep security awareness high within the development team by reinforcing the idea that the application will be constantly under attack.

Also, considering the application from an attacker’s perspective helps the development team think of how their application may be abused and shows them the importance of being more defensive in their development. Although any single threat model is bound to miss potential threats, combining threat models with continuous risk assessment increases the chance that downstream threat models will raise potential issues that were missed in prior assessments. The happy result is self-correction and more thorough threat assessments.

Automated Security Testing

In the past, application security testing, when it was done at all, was likely to consist of penetration tests conducted at the end of the life cycle, which guaranteed the late discovery of security vulnerabilities that could be remediated only at high cost. Early discovery of security issues is critical to keep costs down, and automated security testing is essential for this so that testing can be more easily introduced earlier in the development life cycle. This is especially true for fast-paced development environments. End-of-life-cycle testing prior to production deployment is still desirable, but when it is combined with automated security testing tools used for continuous assessment, the chances of not finding an expensive glitch until the last minute are greatly reduced.

Many automated testing tools, falling in various categories, are available, and each has its own relative strengths and weaknesses (see Figure 5).

**Dynamic Application Security Testing (DAST)**

**PROS**
- Can more readily find certain cross-site scripting (XSS) and authorization issues
- Easily integrated for testing
- Does not require language expertise to assess results

**CONS**
- Can’t readily identify certain classes of vulnerabilities such as password storage, key management and others
- Can be more difficult to ensure full coverage of application
- Mostly restricted to end-of-life-cycle testing
- Output of untrusted input not always clear, potentially leading to missed vulnerabilities
- Does not easily test all application types (e.g., client/server)

**Static Application Security Testing (SAST)**

**PROS**
- Able to find hidden back doors
- Can detect more complex vulnerabilities
- Can readily trace untrusted data
- Easily integrated early in life cycle
- Can readily assess all supplied code
- Identifies precise location of flaw
- Can test all application types, including client/server over nonstandard protocols

**CONS**
- Can frequently generate many false positives due to missed validation
- May not be aware of incomplete code submission if it does not perform build
- Does not easily find certain classes of vulnerabilities, such as authorization issues

**Interactive Application Security Testing (IAST)**

**PROS**
- Can help correlate vulnerabilities to problem source code
- Leverages pros of both SAST and DAST
- Produces quick tests

**CONS**
- Requires integration into tested applications

**Figure 5. Pros and Cons of Various Automated Testing Approaches**
DAST—DAST tools identify vulnerabilities in running applications (or components of applications). Also known as “black-box testing,” DAST is fast and effective in finding many types of application vulnerabilities, including authentication and authorization issues. DAST tools can be used by a wide range of personnel because they do not require knowledge of the coding languages. Although the effectiveness of DAST is highly dependent on the code coverage that the test is able to execute, the tools have the advantage of testing in the as-deployed environment, with web application firewalls and other defenses in place between the running application and the testing tool.

SAST—SAST tools may be used when those assessing the application have access to the source, byte code or binaries of the application. Also known as “white-box testing,” most SAST tools will, minimally, leverage data flow analysis to identify potential vulnerabilities within the application. Data flow analysis is a powerful feature that is used to identify where potentially untrusted data is being used and treated as trusted (without any form of validation, encoding, etc.).

SAST tools can be very useful in finding vulnerabilities that may not be readily discovered through black-box testing of a running application. For example, password (or other sensitive data storage) issues and cryptographic issues (such as weak algorithms, weak protocols or key storage) are more discoverable with access to the source/byte code/binaries that SAST tools have. Also, SAST tools are more likely to identify points where inadequate data validation is taking place. For example, consider a data flow where untrusted data undergoes partial but ineffective validation prior to being used. The data flow will quickly reveal where the ineffective validation is occurring, and manual review can pinpoint the weakness. By contrast, DAST tools frequently are unable to test the almost innumerable ways that a validation may be ineffective and won’t identify where the faulty validation is occurring. Additionally, because of their access to the source/byte code/binaries, SAST tools typically can identify more complex vulnerabilities than other testing platforms.

IAST—IAST tools combine the relative strengths of SAST and DAST tools. They instrument the running application, either on the application server or within the application itself. Running on the inside, they can assess included libraries and tune dynamic tests based on observations of application behavior. IAST tools often have fewer false positives than standalone DAST and SAST tools, since they can correlate across the different testing approaches.

---

The Negatives of Automated Testing

One drawback of automated testing is that it can produce many false positives. When automated testing is inserted into the standard development life cycle, false positives can be a drag on the development team, especially if they lack the training needed to appropriately sort out false positives.

A quick way to kill security team credibility is to insert a tool into the development process, run a full scan with all rules enabled and deliver a report to the development team that is overladen with false positives. A better approach is to develop a subset of high-confidence rule sets (with very few positives) to give to the development team to run at high frequencies. This allows the development team to detect and correct many security issues in the code early on, which is essential for fast-paced development environments.

This approach introduces a gap in vulnerability discovery where only a subset of all rules—the high-confidence rules with low false positives—is run early and often, whereas the complete rule set (with many false positives but with some true positives) is not being run. It must still be run, but the interval in which the rule set is run will be highly dependent on the overall development process. However, by having good security requirements, good security architecture and continuous threat modeling and risk assessments, these strong early-life-cycle security processes can help limit potential gaps that may emerge between when the high-confidence small rule set scans are run and the low-confidence full rule set scans are run.

Manual Security Testing

As useful as automated tools are, their limitations, such as an inability to find logic flaws and design flaws, make it essential that they be augmented with manual security testing.

As useful as automated tools are, their limitations, such as an inability to find logic flaws and design flaws, make it essential that they be augmented with manual security testing. Both manual code review and manual penetration testing should be conducted with a focus on the portions of the application identified as most risk-prone by the automated risk assessments.
Even strong security governance and a well-implemented SSDLC can leave the applications developed by FSI organizations with significant exposures. For example, many FSI organizations make extensive use of third-party applications and libraries, as well as FOSS, and the use of third-party code with known vulnerabilities is included by the OWASP in its list of the top 10 most common vulnerabilities. When using third-party applications or FOSS, the risks must be managed.

Mechanisms for managing such risk include the following:

- Incorporating a maturity model such as the Building Security in Maturity Model for Vendors (vBSIMM), which provides a mechanism to assess the maturity suppliers of third-party software.

- Performing security testing on the third-party code or FOSS. Ideally, if source code is available, both dynamic and static assessments of the third-party code should be performed.

- Ensuring that policies are in place for managing third-party code or FOSS and associated vulnerabilities. For example, all third-party code in an application should be documented, including version, and regularly monitored for identified vulnerabilities.

- Making sure that the procurement language in the contract for third-party code enforces strong security controls.

Because contract language is essential to protecting FSIs when security vulnerabilities are introduced by third-party providers, the American Bankers Association and the FSSCC have jointly developed suggested procurement language in the 2016 Cyber Insurance Buying Guide. This language includes a guarantee from the provider that it adheres to security standards, with specific guidance regarding to which areas those standards apply. The language also calls for independent third-party assessments to validate adherence to the security standards. In addition, the guide provides sample language to include in an FSI procurement contract to help ensure that the vendor is fully responsible for assuring the security of the software it provides.

---

10 BSIMM.com, www.bsimm.com/about/bsimm-for-vendors
FSI organizations play a critical role in the economy, making them highly visible and frequent targets of application attacks. They are similar to many other businesses in that they are under tremendous pressure to respond to the market quickly, but the FSI is highly regulated, and FSI organizations typically operate in complex environments.

All of these factors make ensuring application security difficult, and the adoption of rapid-deployment environments can contribute to the difficulty. Nonetheless, the fundamentals of maintaining a secure system development life cycle still apply: One of the most effective tactics is to inject security early in the software development life cycle.
Steve Kosten, an instructor for SANS’ Secure Coding in Java/JEE: Developing Defensible Applications course, holds the GSSP-JAVA, GWAPT, CISSP and CISM certifications. Experienced in secure code review, vulnerability assessment, penetration testing and risk management, he is a security consultant at Cypress Data Defense. Steve previously performed security work in the defense and financial sectors, and headed up the security department for a financial services firm. A frequent presenter at security-related conferences, he is currently leader of the Denver chapter of the Open Web Application Security Project (OWASP).
# Upcoming SANS App Sec Training

<table>
<thead>
<tr>
<th>Training</th>
<th>Location</th>
<th>Dates</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Univ. of California - Davis - DEV522: Defending Web Applications Security Essentials</td>
<td>Online Only, CA</td>
<td>May 11, 2020 - May 16, 2020</td>
<td>vLive</td>
</tr>
<tr>
<td>SANS Network Security 2020</td>
<td>Las Vegas, NV</td>
<td>Sep 20, 2020 - Sep 27, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS OnDemand</td>
<td>Online</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
<tr>
<td>SANS SelfStudy</td>
<td>Books &amp; MP3s Only</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
</tbody>
</table>