Chapter XVII
Agile Development of Secure Web-Based Applications

A. F. Tappenden
University of Alberta, Canada

T. Huynh
University of Alberta, Canada

J. Miller
University of Alberta, Canada

A. Geras
University of Calgary, Canada

M. Smith
University of Calgary, Canada

ABSTRACT

This article outlines a four-point strategy for the development of secure Web-based applications within an agile development framework and introduces strategies to mitigate security risks that are commonly present in Web-based applications. The proposed strategy includes the representation of security requirements as test cases supported by the open source tool FIT, the deployment of a highly testable architecture allowing for security testing of the application at all levels, the outlining of an extensive security testing strategy supported by the open source unit-testing framework HTTPUnit, and the introduction of the novel technique of security refactoring that transforms insecure working code into a functionally-equivalent secure code. Today, many Web-based applications are not secure, and limited literature exists concerning the use of agile methods within this domain. It is the intention of this article to further discussions and research regarding the use of an agile methodology for the development of secure Web-based applications.
INTRODUCTION

E-commerce and Web-based applications have quickly become a staple for many businesses, and in some cases represents an entire business itself. Success or failure in the online marketplace can, in fact, determine a company’s fate. For example, in 2003, Dell Incorporated’s U.S. home and home office divisions generated $2.8 billion or nearly 50% of its revenues through its online storefront (Dell Inc., 2003). Web-based applications are typically “always on,” and although this allows customers to access products and services at all times, it also leaves the applications open to continuous access from malicious attackers. Security is therefore a major concern for Web-based applications, and a breach in security can lead to a significant loss in profit or the exposure of valuable information, such as trade secrets or confidential client information. Furthermore, security and privacy are listed as major concerns for customers utilizing e-commerce systems (Udo, 2001), and security is listed as one of the three most important quality criterion for Web-based application success (Offutt, 2002). According to a recent study, 75% of online security breaches occur at the application layer, not the transportation layer (Grossman, 2004a). Common exploits in modern Web-based applications are occurring due to security flaws within the Web-based application itself, regardless of protection from firewalls and Secure Socket Layer (SSL) communication channels. Malicious attackers appear to be concentrating on what they believe to be the weakest link—the application itself.

SQL injection, buffer overflows, cross-site scripting, file inclusion, URL injection, and remote code injection vulnerabilities have historically plagued Web-based applications developed by both the open-source and commercial communities. These vulnerabilities have been found in Web-based applications employed by organizations such as the FBI, CNN, Time Magazine, Ebay, Yahoo, Apple Computer, and Microsoft (Cgisecurity.com, 2002). These vulnerabilities are not only extremely common in many Web-based applications, but can also be extremely costly. For example, an SQL injection vulnerability in the PetCo.com Website resulted in 500,000 customers’ credit-card numbers and information being made vulnerable to anyone who could carefully construct a SQL query (Grossman, 2004b). Every month, 10 to 25 cross-site scripting security flaws are found within commercial Web-based applications (Cgisecurity.com, 2002). It is clear that the methodologies currently employed for the development of Web-based applications are not adequately meeting security needs.

The development of Web-based systems is often distinctly different from the development of traditional Information Technology (IT) systems as Web-based applications are typically much smaller than traditional IT applications, and consequently their production period is much shorter (Cusumano & Yoffie, 1999; Iansiti & MacCormack, 1997; Kirda, Jazayeri, Kerer, & Schranz, 2001; Ramesh, Pries-Heje, & Baskerville, 2002). Furthermore, these applications evolve at a much faster pace, and the development cycle is highly iterative in nature. Due to these factors, agile development methodologies are increasingly being used for the development of Web-based applications (Barnett & Narsu, 2005; Ramesh, Pries-Heje, & Baskerville, 2002). Until recently, agile development methodologies and traditional security engineering have been seen as orthogonal (Beznosov & Kruchten, 2004); however, recent discussions are beginning to harmonize the two seemingly different viewpoints (Beznosov, 2003; Beznosov & Kruchten, 2004). Discussions of agile security assurance to date have been at a very high level and although the works acknowledge the possibility of secure application development within an agile framework, they do not provide the necessary support infrastructure needed to develop secure applications for a specific domain. This paper will propose a strategy for the development of secure Web-based applications within an
Agile Development of Secure Web-Based Applications

Agile framework by adapting currently used and accepted agile development tools and techniques. The proposed strategy provides an appropriate level of technical detail to be translatable to agile software development, meanwhile remaining high-level enough to be extended and adapted to future Web-based application development.

Agile development methodologies are a group of software development methodologies that all ascribe to the Agile Manifesto (Beck et al., 2001). The Agile Manifesto, created in 2001 by 17 prominent figures in the then-lightweight software development field, is widely regarded as the canonical definition of an agile method. Today agile methods include extreme programming, scrum, and adaptive software development. Agile methodologies are generally characterized by close customer-developer collaboration and short, iterative development cycles. Compared to more traditional software development methodologies, agile methods rely on open, face-to-face communication between customer and developer, enabling the development team to react quickly to ever-changing customer requirements and expectations. The development process is broken down into short cycles typically lasting one to six weeks, allowing the development team to meet customer requirements iteratively. At the end of each cycle, the customer is provided with working code based on customer-prioritized functionality, which further facilitates the input of customer feedback into the software development process.

Agile security engineering involves employing the same values that drive agile software engineering to the traditional practice of mitigating security risks in software. This includes using a highly iterative process for delivering the security solution and translating security objectives into automated security test cases. The security test cases should be described prior to completion of the target system; that is, they should be used to elaborate the security objectives and to characterize the secured behavior that would be classified as “good enough security” from the customer’s perspective. The role of a security engineer in this regard would be to advise and coach the customer on the foreseeable risks, with the customer ultimately deciding which ones would be of the greatest concern to them. This paper will outline a four-point strategy for the development of secure Web-based applications within an agile development framework. This strategy includes representing agile security requirements as test-cases, deploying a highly testable architecture, employing automatable security testing, and implementing security refactoring. This strategy will provide a basis for the negotiation of “good enough security” in the agile development of secure Web-based applications.

AGILE SECURITY REQUIREMENTS

One of the principle challenges in developing a secure Web-based application is that the target itself, security, is really an objective—a relatively abstract requirement as opposed to a functional requirement. The first step in a security testing protocol is to set the context for the tests; that is, to identify the security objectives. In more traditional development methodologies this may have been done as part of the requirements workflow of the project, but for many Web-based applications, functional requirements are often of the highest priority due to time-to-market pressures, and comprehensive security requirements are often left out of preliminary discussions. For many projects, security requirements become urgent only close to deployment time, and are often added as an afterthought and tested in an ad hoc fashion. From a traditional security engineering viewpoint, security requirements must be strictly defined from the beginning of the project and monitored and reviewed throughout the development process; however, the highly iterative and evolutionary nature of Web-based applications suggests that these ideas are simply not realistic.
for Web-based systems. Web-based applications are continually in transition; often one version of a Web-based application is deployed to the public while the next release is being finalized and tested on a private server. Techniques for harvesting and representing security requirements during iterative development cycles are necessary for the agile development of secure Web-based applications.

Within the agile development framework there are a number of methods for modeling requirements. Scott Ambler (2004), in his book “The Object Primer 3rd Edition: Agile Model Driven Development with UML 2”, defines a number of requirement artifacts, one of which is a technical requirement. According to Ambler (2005), “a technical requirement pertains to a non-functional aspect of your system, such as a performance-related issue, a reliability issue, or technical environmental issue”. Security requirements, although not explicitly mentioned in the definition of a technical requirement, can be modeled as technical requirements since they represent a non-functional aspect of the software system. Like other requirement artifacts such as features, user stories, and business rules, technical requirements are understood by the agile development community, and the modeling of security requirements as technical requirements can be comfortably integrated into current agile development practices.

A major force behind many agile development methodologies is test driven development (TDD) (Astels, 2003; Beck, 2002). TDD utilizes test cases to characterize customer requirements and to specify designs. Within an agile framework, security requirements captured as technical requirements must ultimately be expressed as test cases. Security test cases can be quite difficult to accentuate; however, a number of preliminary security test strategies specifically tailored to Web-based applications have been articulated. The Institute for Security and Open Methodologies has released the Open Source Security Testing Methodology Manual (OSSTMM) version 2.1 (ISECOM, 2003). The OSSTMM provides a taxonomy of test facets for Web-based applications, but it does not provide any test implementation details. The taxonomy is very helpful in a discussion of what to test, and can be used by security engineers as a type of checklist in the negotiation of “good enough security” with customers. It does not however, help relate security requirements to specific test cases.

Modern agile development tools such as FIT (Cunningham, 2002) can be used to articulate tests of the application’s ability to satisfy security requirements. FIT is a tool that enhances communication and collaboration between the customer and the developer by allowing customers to create test cases using tools such as Microsoft Office. The test cases are typically written in tabular form using Microsoft Word, then stored as HTML files. The programmer-created FIT-fixture can then interpret, test, and provide feedback to the customer and developer through the modification of the HTML files, alerting them to the results of each test case. FIT is especially suited to Web-based application security testing for two primary reasons. First, FIT is designed to interact with the application by bypassing complex GUI interactions; this is a perfect match for Web-based applications as a major component in their security testing is bypassing the client-side GUI. Secondly, utilizing the same tool for the security requirements that is used to elaborate and test the functional requirements makes the security requirements more approachable, and it is thus more likely that the security requirements will be built-in with the functional requirements. FIT can be used to capture both security and functional requirements, ensuring that both requirement sets are met from the start of the project or build cycle.

As an example, consider a database-backed Web-based application that has generic search capabilities. The application will typically have a search form that will permit the user to enter
their search criteria and then submit their search. An appropriate server-side component would then generate the SQL query to perform the search against the database and execute it as required. This search page would be an ideal candidate on which to perform a test for SQL injection faults, since the database query is ultimately generated from user input. If the user input is only validated on the client-side, then there is the likelihood of an SQL injection fault occurring. The security objective to be targeted is therefore to ensure that the underlying database is not accessible by exploiting SQL injection faults. To test this with FIT, a format similar to Table 1 may be used. This example is based on an enterprise phone directory that is accessible from the public Internet, and assumes that there is a base set of test data already in place.

In the example above, specially-placed single quotes and semi-colons may cause a susceptible application to execute the ‘drop table’ statement, clearly something that the target application should not permit. If the application developers only used client-side input validation, then this test would expose the SQL injection flaw. If the application was built so that the server-side components also did input validation, then presumably the validation would clean the SQL query before executing it and the search would simply return 0 results. This type of test is possible because FIT bypasses the client-side functionality of the application and instead relies on fixtures that only emulate enough of the front-end to set up the appropriate tests.

A second example of security testing would be to run functional scenario tests but to annotate them with additional security information. These type of tests could target, for example, the typical role-based security that Web-based applications implement. In the example laid out in Table 2, an entry is being added to the enterprise phone directory, an application feature that only PhoneDirectoryAdministrators should be permitted to do. In this case, the FIT-fixture has been designed to return the data as inserted into the database with each field separated by a pipe (||) and a space. Note that this test again assumes that certain test data exists. Having an established base set of data is required for most automated testing situations in order to minimize the test setup/teardown code.

Table 1. SQL injection fault testing with FIT

<table>
<thead>
<tr>
<th>FixtureNamespace.FixtureName</th>
<th>Remark</th>
<th>FirstName</th>
<th>LastName</th>
<th>Dept</th>
<th>Find()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Known single result</td>
<td>Michael</td>
<td>Stipe</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Known multiple results</td>
<td>*</td>
<td>Smith</td>
<td>*</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Try to inject some SQL</td>
<td>*</td>
<td>Smith'; drop table employees;'</td>
<td>*</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2. Role-based security testing with FIT

<table>
<thead>
<tr>
<th>FixtureNamespace.FixtureName</th>
<th>Remark</th>
<th>FirstName</th>
<th>LastName</th>
<th>Dept</th>
<th>User</th>
<th>Pass.</th>
<th>Add()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid user/pwd</td>
<td>Annabelle</td>
<td>Laxative</td>
<td>Hosiery</td>
<td>jib</td>
<td>job</td>
<td>Annabelle</td>
<td>Laxative</td>
</tr>
<tr>
<td>Valid user/invalid pwd</td>
<td>Frank</td>
<td>Lin</td>
<td>Home</td>
<td>jib</td>
<td>ERROR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invalid user</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>jit</td>
<td>jot</td>
<td>ERROR</td>
</tr>
<tr>
<td>Valid user not in group</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>jat</td>
<td>jot</td>
<td>ERROR</td>
</tr>
</tbody>
</table>
Note that in the previous example, the error message could be even more precisely worded so that it could also be included as part of the test.

It is clear that security requirements can be captured and understood by the agile development community through the extension of the non-technical requirements artifact. These requirements can be gathered and translated into test cases through the use of emerging Web-based security testing strategies, and can be put on par with the functional requirements through the use of freely available tools such as FIT. Capturing security requirements in this fashion allows for security requirements to be built-in with functional requirements in an iterative, evolutionary fashion. Furthermore, the ability for security requirements to be added to the project at the beginning of every successive build cycle allows for the project to remain flexible and meet the changing customer requirements throughout the project. Through the use of tools such as FIT, both the customer and developers can view the status of the functional and security requirements concisely, providing up-to-date feedback as to the project's progress from both a functional and security viewpoint. Ultimately, this type of security requirements definition enables the communication of security requirements in the form of test cases, which can be used by an agile development team to explore the design space and produce a final product that satisfies the functional and security requirements.

HIGHLY TESTABLE ARCHITECTURE

Many Web-based applications today have increasingly complex functional requirements. This increase in complexity has a direct effect on the test effort required, and it is becoming more difficult to test these applications. Due to the increased complexity, manual testing, even with low-cost off-shore resources, is unlikely to be cost-effective. The only cost-effective solution is to automate testing at all levels, not just at the unit testing level. Automated testing suites such as FIT, mentioned in the previous section, JUnit, PHPUnit, and HTTPUnit can be used to automate testing activities for Web-based applications (Cunningham, 2002; Gold, 2004b; Open Source Technology Group, 2005a, 2005b). It has also been indicated that both the customer and the developer may benefit from the inclusion of higher levels of testing in the test-first activities (Pancur, Ciglaric, Trampus, & Vidmar, 2003). In other words, organizations that build customer tests to characterize requirements, both functional and non-functional, and then subsequently use developer tests to specify design decisions, are in a better position than companies that only do developer testing. The net effect is that there is a need to develop highly testable software, and there are perceived security benefits to building that software using a test-driven process. The next section will outline the evolution of the traditional layered application architecture into a test-oriented application architecture, allowing for security testing of the Web-based application.

The Current State — Layered Architecture

Many Web-based applications are constructed using a variant of a layered architecture. Using this approach, the Web-based application's overall responsibilities are split amongst three or more layers: the data services layer, the business services layer, and the presentation layer (see Figure 1) (Microsoft Corporation, 2004).

The data services layer is responsible for data storage and retrieval, the business services layer is responsible for business workflow and logic, and the presentation layer is responsible for supporting interactions with the end-user through the Internet and ultimately through the Web browser. This layered architecture has a number of advantages, particularly given the heterogeneous nature
of the Internet. Similarly, there are advantages in using this architecture when more than one type of user interface might be needed. In this case, the same business logic and database, for example, could be accessed through a Web page, a desktop (thick client) application, touch-tone phone (IVR system), or a personal digital assistant (PDA). Given the increasing ubiquity and capabilities of cell phones, it also makes sense to consider that accessing the business logic from cell phones may be required in some business domains.

The Class Type Architectures

Scott Ambler (2002) extended the classical layered architecture in describing the class-type architecture, as shown in Figure 2. In the class-type architecture, the classical business services layer is further divided into two layers, one that contains only business entities and another that contains business controllers. A business controller class is responsible for the control and sequencing of one or more business scenarios, while business entities are the domain objects that are created or manipulated in the business scenarios (Ambler, 2002).

The class-type architecture maps particularly well to Web-based applications developed in Java and Microsoft .NET since in both cases a large set of classes are available for reuse. In Java, the presentation layer might be implemented as JavaServer Pages (JSP), and in the .NET environment the presentation layer would be implemented by Web Forms. The advantages of the class-type architecture over the classical layered architecture lie in splitting the business logic from the business entities. Entities tend to be constant in any given domain, yet the business processes in those same domains might be highly adaptive and change rapidly. Individuals shopping on the Internet, for example, still use a shopping cart to carry their goods to a checkout. Never mind that these objects are virtual objects—to the average shopper they are indistinguishable from the real thing. However, significant changes in retail business processes have been introduced because of the availability of the Web. Given that building Web-based applications based on the class-type architecture has proven to be successful in practice, and that there is an increased need for writing highly testable applications, it seems that there is an opportunity for merging these two trends. The intended result is an application architecture that retains the benefits of high testability and of the class-type architecture.
Proposed Architecture — Highly Testable Class-Type Architecture

This section will propose a revision of the representation of the class-type architecture shown in Figure 2 to include test layers (see Figure 3). This will allow for Web-based applications to utilize the benefits of a class-type architecture and also enable security testing at all levels. The test-cases and the testability of the target Web-based application are therefore put to the forefront of the design process. With the envisioned layers for both the Web-based application and the test-cases for that application in place, the design team has a foundation for iterative and evolutionary development. Security engineers can decide which of the test layers they propose to automate, what tools they propose to use for that automation, and how the test layers communicate with the product layers during the execution of tests, supporting the ability to test the Web-based application for security and functionality at all levels.

The addition of the test layers to the traditional class-type architecture highlights the testability hooks that application developers should build into their applications. Furthermore, building the test layers becomes a purposeful pursuit, as they are no longer the afterthought that typically occurs under traditional development. At the same time, introducing the testability hooks does not necessarily mean encouraging a big up-front design effort, since to varying degrees, each of these test layers can be built alongside their target layers in an iterative and evolutionary manner (test-first or test-last). The resulting architecture is useful for security testing since various security test patterns can be employed within any number of the test layers. Security testing can target an individual layer (probably in conjunction with the use of mock objects to isolate the target layer) or also perform integration testing, targeting an upper layer and the layers that it depends on. The introduction of a highly testable architecture will support the execution of security-orientated test-cases, allowing security testing to be performed upon the application without the development of an extensive ad hoc security testing support infrastructure at the end of the project, by which time security is typically deemed to be a significant risk.

AUTOMATABLE SECURITY TESTING

With the ability to express security requirements as test cases through the use of tools such as FIT, and the ability to test the application at all layers of implementation through the use of a highly testable architecture, agile methods can be extended to the development of secure Web-based applications. Detailed security testing strategies are needed to further support the development of secure Web-based applications. This section will focus on extending the current state of the art for Web-based security testing, and will explore ways to automate the testing strategy allowing for seamless integration into the agile development process.
Current Strategies

SQL injection and bypass testing are two security testing strategies specifically intended for Web-based applications. SQL injection is a security testing strategy that attempts to access the backend database of a system through carefully crafted inputs into HTML forms (Chris, 2002; Finnigan, 2002; Grossman, 2004b; SPI Dynamics, 2002). SQL injection is both a testing strategy and an attack strategy that has been used to bypass security protocols, gain administrative privileges, access privileged information, and to simply manipulate back-end databases. This testing strategy involves the input of SQL-specific reserved characters and keywords into HTML forms in an attempt to alter database queries. This type of testing and test-case is very specific to SQL databases and only addresses one type of security breach; however, it has provided a basis for further work in the area of security testing for Web-based applications.

The idea of SQL injection has been further generalized, and a testing strategy outlined in the paper “Bypass Testing of Web Applications” (Offutt, Wu, Du, & Huang, 2004). The paper introduces the idea of bypass testing. Bypass testing attempts to test the robustness of the Web-based application by circumventing the client-side GUI restrictions and attempting to alter the application’s behavior (Offutt, Wu, Du, & Huang, 2004). Bypass testing describes four categories of input-value testing. The first category, data type and value modification, calls for the input of invalid strings due to expected type restrictions. An example of this would be to provide the server with a real number instead of an integer. The second category, HTML built-in length violation, exploits client code that utilizes the built-in HTML form length restriction. This type of testing involves the input of longer-than-expected strings into the server. The third category, HTML built-in value violation, concentrates on modifying predefined input values that occur in check and radio boxes. The final category, special input value, focuses on the input of reserved characters into text fields. The reserved characters attempt to violate server interactions with backend databases and XML communications.

Although bypass testing and SQL injection provide a good basis for security testing of Web-based applications, they face several limitations. Testing a Web-based application for security using SQL injection exclusively is far too limited to detect a number of security vulnerabilities. SQL injection only deals with one very specific component of a Web-based application—interactions with a backend database—and does not uncover common vulnerabilities such as URL injection or cross-site scripting vulnerabilities.

The bypass testing strategy outlined by Offutt et al. (2004) is limited to fundamental textual information provided to a Web-based application through HTML forms and does not address a number of other types of inputs that modern Web-based applications frequently use. Security breaches can come from a wide variety of sources including, but not limited to, those addressed by the bypass testing strategy. Many Web-based applications exchange information with users through cookies, files, and base64 encodings. Each of these vehicles for interaction are also being used by malicious attackers to target Web-based applications, and must be tested to ensure integrity.

Expanding Security Testing

All generic inputs to a Web-based application should be presumed compromised regardless of client-side verification, and subject to verification by the server. Many, if not all of the common security risks to Web-based applications can be mitigated by thorough server-side input verification. Generic inputs into a Web-based application primarily come from two sources: GET and POST requests. The length of each input must be verified by the server to prevent buffer overflow vulner-
abilities. These inputs need to be checked against SQL and XML reserved characters, as outlined in
the SQL injection and bypass testing strategies, but additional testing is still required. Inputs need to
be verified and checked against UNIX, Windows, and JavaScript reserved characters, and invalid
character sets, ASCII or Unicode. Furthermore, type checking is required for all inputs. Integers,
naturals, and reals need to be verified of type, as well as being checked against the min/max and
zero values. In short, all inputs into the software system must be considered tainted and subject to
verification by the server.

Besides GET and POST requests, cookies haveecome a third staple input used in Web-based
applications. Most Web-based applications utilize
cookies for a variety of reasons, of which session
management is one of the most common. Due
to the extensive use of cookies and the security
implications of a breached session management
protocol, the security testing of cookies must
be included in any testing strategy. Attackers
can utilize cookies in three primary ways: by
creation, modification, and deletion. Much like
generic inputs, the information in cookies can
be modified and should be subject to server-side
verification. Web-based applications should be
tested against cookie modification, spoofing, and
deletion. The same principles outlined for the
testing of generic inputs should be extended to
the input data accepted from cookies, with further
test cases generated to simulate the spontaneous
creation, modification, or deletion of cookies at
various states in the application control-flow.
Further consideration should be given to the
type of information stored in cookies, such as
whether the information is valuable, whether it
is encrypted, and whether an attacker can use the
information stored in the cookie to attack the ap-
application. These types of considerations can and
should be used in the generation of test cases to
examine cookies placed on a machine by a Web-
based application, as often an attacker, through a
single cross-site scripting vulnerability on another
insecure Website, can obtain these cookies.

Many Web-based applications interact with
users through the passing of files over the Inter-
et. Online e-mail services, for example, provide
the user with the ability to add attachments to
outgoing e-mails, while online-dating services
allow for the exchange of photographs and other
privileged information between customers; in
addition, defect tracking software often allows
for the uploading of software patches. Although
the ability to upload files is critical to the success
of many Web-based applications, files can pose
a serious threat to security. Images, documents,
and all other file types are continuously being
submitted to online services, and therefore need to
be validated. Web-based applications must detect
invalid or malicious images or documents before
they are utilized by the application. This issue has
been brought to the forefront by current computer
viruses that can infect a user’s system through the
simple act of viewing a JPEG image (Microsoft
Corporation, 2004). All told, files pose several
threats to Web-based applications. File size and
file name length must be verified by the system
to ensure that the file does not cause a buffer-
overflow, or utilize too many system resources or
too much of the connection bandwidth. File type
must also be validated, which requires more than
just a simple extension check, as an attacker can	en often spoof the file extension. In short, thorough
server verification of files needs to be performed
on any Web-based application that accepts the
uploading of user files.

Base64 encoding is a technique used to encode
binary files as a set of ASCII characters (Freed,
1996; Linn, 1993). This encoding is frequently
used on the Internet to transmit binary data over
any communication channel that is limited to
plaintext ASCII data, and is often used in conjunc-
tion with legacy e-mail applications. Web-based
applications utilizing this type of input need to
verify the incoming base64 encoding. This type of
verification should focus primarily on the base64
encoding itself. Length and size of the input must be checked to mitigate the risk of a buffer overflow, and then the encoding itself should be verified for correctness. The alphabet used in the encoding must also be examined, to ensure that the characters transmitted as part of the encoding are within the accepted standard alphabet. Furthermore, in a base64 encoding, binary data is transmitted by sets of three ASCII characters; thus it should be verified that the encoding contains a multiple of three characters, to ensure that character addition or subtraction has not taken place. Test cases should focus on the introduction of invalid characters and the addition and subtraction of valid characters. It is important to note that once the base64 encoding is validated, the binary file transmitted using the base64 encoding must also be validated as described above.

It is clear that all inputs into a Web-based application need to be validated by the server.

### Table 3. Summary of test inputs

<table>
<thead>
<tr>
<th>Valid Input</th>
<th>Test Input</th>
<th>SQL Injection</th>
<th>Offutt et al.</th>
<th>Proposed Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic Input</td>
<td>SQL reserved characters</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>XML reserved characters</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unix reserved characters</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Windows reserved characters</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Oversized string</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non-ASCII characters</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Non-Unicode characters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Null String</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Integer Input</td>
<td>Min/max values</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Zero</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Real number</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non-number</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Real Number Input</td>
<td>Min/max values</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Zero</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non-number</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Cookies</td>
<td>Create/Modify/Destroy cookie</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Invalid cookie of the same name</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Invalid data types within the cookie</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Generic, Real, Integer, etc)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Files</td>
<td>Different file type</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Invalid file of type</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Long file name</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Malicious file (script, image)</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Large file size</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Base64 Encoding</td>
<td>Invalid Base64 encoding</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Improper Base64 alphabet</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Invalid underlying file</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Inputs need to undergo checks for length, validity, character set (ASCII, Unicode, etc), reserved characters (SQL, XML, JavaScript, UNIX, Windows, etc), and type (string, integer, real, image, document, etc). These checks can eliminate many of the security vulnerabilities that affect Web-based applications. Table 3 below provides a summary of the input types and the test inputs used in the server input verification. The table also highlights the differences between the security testing strategy proposed in this paper and the strategies proposed by SQL injection (Chris, 2002; Finnigan, 2002; Grossman, 2004b; SPI Dynamics, 2002) and Offutt et al. (Offutt, Wu, Du, & Huang, 2004).

It is essential that any testing-based security strategy covers all of the aspects of Web-based security that are listed in Table 3. Without such rigorous efforts, a dynamic Web-based application cannot be assured of its integrity. Table 3 provides a comprehensive list of security testing issues that can be used in discussions with the customer to define “good enough security” and can be used by the developers to translate security requirements into specific test cases. These test cases can then be defined within the FIT framework and used to ensure that the Web-based application meets the agreed security requirements.

**Automated Security Testing**

Automation is needed for any agile unit-level testing suite. Security requirements, just like functional requirements, must be translatable to not only test-cases, but to automatable test-cases. Although this goal is inherently supported by the highly testable architecture described in Section 3, further integration with unit-level testing activities is needed for security requirements. This integration can be accomplished through a novel use of HTTPUnit to test for security. HTTPUnit is a unit-testing framework that provides the ability to bypass complex client-side GUI interactions and provide automated unit-level testing for Web-based applications (Gold, 2004b). Currently, HTTPUnit emulates HTML form submission, JavaScript, basic HTTP authentication, cookies, and automatic page redirection. This unit-test framework supports the analysis of returned HTML documents as plain text, an XML DOM, or containers of forms, tables, or links (Gold, 2004b). HTTPUnit is implemented in Java and is based on the JUnit framework, so it can be seamlessly integrated for use with other commonly used xUnit testing frameworks.

Hightower and Lesiecki (2002), in the book “Java Tools for Extreme Programming: Mastering Open Source Tools Including Ant, JUnit, and Cactus,” provide an initial basis for the use of HTTPUnit within an agile development framework. The book proposes that HTTPUnit is a tool that can be used to perform functional and unit-level testing upon a Web-based application; however, this reflects a very traditional view of the relationship between the server and the client—a view which many attackers exploit when targeting Web-based applications. The remainder of this section will focus on the novel use of HTTPUnit to bypass not only the client-side GUI, but also the client-side input restrictions, allowing for the automated security testing of a Web-based application.

Although HTTPUnit was specifically designed to bypass complex GUI interactions, it was not...
originally intended to bypass the client-side input restrictions. HTTPUnit was designed to exactly mimic Web-browser behavior, including client-side input restrictions, to allow for traditional, functional, blackbox testing of Web-based applications. The first step in automating security testing then is to disable the client-side input verification. In HTTPUnit, this is done by creating a WebRequest object that contains an unvalidated request as shown in below in Figure 4.

The newly created WebRequest object will allow HTTPUnit to provide the server with input data that has not been restricted by the client, effectively bypassing not only the cumbersome GUI interactions, but also the client-side input restrictions. This type of testing will allow testers to provide the system with unexpected and invalid inputs that will test the robustness of the application under unexpected input values.

The next two examples provide a basic outline for the creation of test-cases to check for cross-site scripting vulnerabilities and invalid cookie manipulation.

Cross-site scripting vulnerabilities are very common within Web-based applications that allow users to post messages to a trusted site. This example involves a typical Internet auction site. The site allows users to post or browse products for sale on the Website. The following HTML form, shown in Figure 5, enables a user to post a product to the Website.

This HTML form requires the user to specify and describe the product to be auctioned off by the application. This is a common situation in which a cross-site scripting vulnerability may surface. An attacker could place a malicious script within

Figure 5. Example HTML Form For Posting Product Information

```html
<form action="./" method="POST">
  Product: <input type="text" name="product" size="20" maxlength="20"/>
  <p>
  Description: <TEXTAREA NAME="description" ROWS=6 COLS=40>
    A Brief Description
  </TEXTAREA>
  </p>
  Contact E-mail Address: <input type="text" name="email" size="20" maxlength="20"/>
  <input name="Submit" value="submit" type="submit">
</form>
```

Figure 6. An Example HTTPUnit Cross-Site Scripting Test Case

```java
WebConversation wc = new WebConversation();
WebRequest request = new GetMethodWebRequest("http://usedcars.com/");
WebResponse response = wc.getResponse(request);
WebRequest form = response.getForms()[0].newUnvalidatedRequest();
form.setParameter("product","Laptop Computer");
form.setParameter("description","<script>alert(FAILED TEST)</script>");
form.setParameter("email","foo@foobar.com");
response = wc.getResource(form);
assertTrue(response.getText().matches("(?s).*<script>(?s).*")));
```
the description or title of the product. This script could redirect the browser to a dummy page asking for credit card information, or strip the user’s machine of valuable cookies containing privileged information such as a username and password combination. The HTTPUnit test case, shown in Figure 6 will test the description field for a cross-site scripting vulnerability.

This test case attempts to insert a script into the description field. If the server does not check the description field for HTML reserved characters, namely the \texttt{<script>} meta tag, then this input will cause the Web-based application to produce a pop-up window for every visitor who accesses the description of the product. Although a pop-up window is harmless, the payload of this script could be much worse. Inputs into a Web-based system that will be posted for the public to read must be checked and stripped of HTML meta tags and JavaScript. This type of server-side verification directly combats cross-site scripting and other common vulnerabilities.

A second example demonstrates the use of HTTPUnit to manipulate cookies for security testing. HTTPUnit provides the ability to manipulate cookies in two primary ways: by creation and deletion (Gold, 2004a). Creation of new cookies in order to overwrite existing ones is also a technique which may be used to effectively modify existing cookies. Creation and modification of cookies within the HTTPUnit framework is very straightforward. The HTTPUnit code segment provided in Figure 7 simulates an attacker at-

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure7.png}
\caption{An Example HTTPUnit Test Case Modifying A Cookie’s Value}
\end{figure}

\begin{verbatim}
WebConversation wc = new WebConversation();
WebRequest request = new GetMethodWebRequest("http://shop.com/cart.jsp");
wc.putCookie("ID","001");
WebResponse response = wc.getResponse(request);
wc.putCookie("ID","002");
WebLink link = response.getLinkWith("checkout");
link.click();
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure8.png}
\caption{A Function To Remove A Specific Cookie}
\end{figure}

\begin{verbatim}
public void remove_cookie(String name, WebConversation wc) {
    HashMap cookieJar = new HashMap();
    String[] cookies = wc.getCookieNames();
    for(int i=0;i<cookies.length;i++)
        cookieJar.put(cookies[i], wc.getCookieValue(cookies[i]));
    wc.clearContents();
    for(int i=0;i<cookies.length;i++)
        if(!cookies[i].equals(name))
            wc.addCookie(cookies[i], (String)cookieJar.get(cookies[i]));
}
\end{verbatim}
tempting to modify the value stored in a cookie in the middle of a checkout transaction.

The example initially accesses the Web-based application with the cookie ID set to the value 001. This could be reflective of a shopping cart based application. The example then modifies the value of the cookie from 001 to 002 in an attempt to assume the ID of another user. This example is over-simplified, as many Web-based applications utilize several cookies in conjunction rather than a single entity, but it demonstrates the ability to create and modify cookies within the HTTPUnit framework.

HTTPUnit also provides the ability to remove all the cookies from the test system. This is very useful in simulating test cases where a user accesses a Web-based application from a different machine, or where the cookies have been removed from the user machine due to the time-to-live mechanism. The framework does not, however, expressly provide the ability to remove a specific cookie from the test system, as this is not a typical user action. The following procedure, show in Figure 8, provides HTTPUnit with the ability to remove a specific cookie from the test system.

The function above can be added to test cases requiring the removal of specific cookies, or can be fitted into the HTTPUnit framework as a method within the WebConversation class.

HTTPUnit can also be outfitted to perform security testing for Web-based applications that utilize files and base64 encodings. The UploadFileSpec and Form classes provide the ability to upload files to a Web-based application under test and the Base64 class provides the ability to encode and decode base64 encodings. Not only can HTTPUnit be retrofitted to perform all of the types of testing described in this section, as summarized in Table 3, but it can also be used in conjunction with FIT to bridge the gap between requirements and testing, and can be used to automate security test-cases at the Presentation and Process/Control test layers outlined in Agile Security Requirements section.

**SECURITY REFACTORING**

Security flaws within any application ultimately derive from human error, be that in the specification, design, implementation, or maintenance phase of any project. Furthermore, Web-based applications are frequently updated with new content, features and improvements, and each new line of code introduced to an application has the potential to create a security flaw. Utilizing the strategies from the previous three sections, an agile development team can express and explicitly test a Web-based application for security. Armed with the ability to detect security flaws, the ability to quickly react and patch potential security flaws is required to further support the

**Figure 9. Escape output variables refactor**

<table>
<thead>
<tr>
<th>Your message:_authentication_process</th>
<th>Initial Insecure Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;?php</td>
<td></td>
</tr>
<tr>
<td>echo($_REQUEST['message']);</td>
<td></td>
</tr>
<tr>
<td>?&gt;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Your message:_authentication_process</th>
<th>Refactored Secure Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;?php</td>
<td></td>
</tr>
<tr>
<td>echo(htmlspecialchars($_REQUEST['message']));</td>
<td></td>
</tr>
<tr>
<td>?&gt;</td>
<td></td>
</tr>
</tbody>
</table>
agile development of secure Web-based applications. The introduction of the novel technique of security refactoring will meet the need for agile development teams to quickly react to security flaws. Security refactoring provides the ability to address security flaws by merging the disciplined approach of refactoring with the current state of the art secure coding practices. This section will introduce security refactoring and present two security refactors that can be used to combat common security flaws that plague Web-based applications.

As stated by Martin Fowler (2005), “refactoring is a technique to restructure an existing part of the source code, that is, to alter the code’s internal structure without changing its external behavior”. Each refactor (transformation) represents a small, behavior preserving change to the source code. A sequence of refactors can result in a significant structural change to the code. Researchers assert that refactoring can lead to clearer, cleaner, simpler, and more reusable code (Opdyke, 1992; Thompson & Reinke, 2001). Refactoring is a technique accepted and understood by the agile community, and several catalogues of refactoring techniques are currently available (Fowler, Beck, Brant, & Opdyke, 1999; Thompson & Reinke, 2001). Although the notion of refactoring an application is standard practice within the agile community, the notion of refactoring an application to enhance security is in its infancy and remains unexplored.

One proposed security refactor is the *Escape Output Variables* refactor. This refactor replaces special characters, such as `<`, `>`, or `\`, within strings outputted to the client with the benign equivalent. This constitutes a very simple refactor, but as mentioned earlier, many refactors are small transformations to source code. This refactor can directly combat cross-site scripting security vulnerabilities. Figure 9 shown below is an example of an insecure PHP code segment that displays a user-inputted message to the browser both before and after refactoring.

This code segment is very typical of the type of code developed for Web-based applications. Developers, often under intense time restrictions and pressure, simply do not escape output strings. Using the *Escape Output Variables* refactor, the vulnerable code in the above example is refactored to become secure. The change to the code is very minimal, the only difference is the addition of the `htmlspecialchars()` function to the print statement. This built-in PHP function simply converts any special HTML character into HTML entities which are subsequently displayed as text on the client browser. This function has equivalents in all major Web-development languages.
Agile Development of Secure Web-Based Applications

Agile Development of Secure Web-Based Applications currently in use, including PHP, JSP, .Net, and Rails. Despite the simplicity of this example, it is clear from the sheer number of cross-site scripting errors discovered in Web-based applications that this type of simple refactoring is required to support the development of secure Web-based applications.

The second proposed security refactoring is the Validate Variables refactoring. This refactoring attempts to verify variables used in dynamic database query generation before they are passed to the database. This refactoring will directly combat SQL injection and will ensure that an attacker does not compromise the integrity of the backend database. Figure 10 below is an example of an insecure dynamic database query in PHP, generated from a user-defined variable, both before and after refactoring.

This code segment represents a typical SQL injection vulnerability within a Web-based application. The variable \$_REQUEST["username"] represents a variable passed into the function from a GET request and can be modified to be any string that an attacker desires. This type of error is very common in many Web-based applications, and can often lead to stolen or modified information from a backend database. This type of error can be safeguarded against by simply validating input variables before they are used within a database query. This is the goal of the Validate Variables refactoring. Again, this change in the code is minimal; however, this refactoring can effectively safeguard a Web-based application against SQL injection attacks. The refactoring simply adds an if statement into the code to verify that the username is valid. In this example, the isValidUsername is not a pre-defined PHP function. This function requires further development and must at a minimum ensure that the input variable does not contain any SQL special characters. An example of the minimum required functionality of the isValidUsername function is provided in Figure 11.

This function would check the input for the presence of special SQL characters used for SQL injection attacks. If the username does not contain those characters, than the input variable would be considered a valid input into the function. It is important to note that this example represents the minimum checking required, and that further integrity checks could be added to the function.

Other security refactors are currently under development. Security refactoring will focus on three primary areas—inputs, outputs, and external interactions. Security refactors for inputs will focus on inputs from traditional GET and POST sources, as well as cookies, environmental variables, and files. Output security refactors will primarily focus on variables used in print (or other functionally equivalent methods). Finally, external interaction refactors will be concerned with dynamically generated SQL queries, calls to external libraries, and system calls. Security refactoring will enable agile developers to refactor existing working code to meet security requirements in future build cycles, allowing the implementation of secure Web-based application development within an iterative framework.

CONCLUSION AND FUTURE WORK

Agile development methodologies and test driven development (TDD) are becoming increasingly
Agile Development of Secure Web-Based Applications

popular as development methodologies for Web-based applications. This paper presents a novel strategy for the development of secure Web-based applications within an agile development framework. The paper presents four synergetic strategies that enable the development of secure Web-based applications within an agile framework. The four-point strategy outlined in this paper provides an effective way to capture, implement, and test security requirements for Web-based applications.

The first strategy extends the currently accepted requirements artifacts within agile development to included security requirements as technical requirements. The security requirements can then be expressed as test-cases and placed alongside functional requirements within the FIT framework, allowing the agile development team to record and implement security requirements as though they were functional requirements. This ability enables the team to add-in both functional and security requirements at the beginning of each successive build cycle, and ensures that the team remains agile and able to quickly react to changing customer requirements, including changing customer security requirements.

The second strategy, the utilization of a highly testable architecture, further supports the ability for the development team to extend the principles of agile development and TDD to the realm of security engineering. Through the use of a highly testable architecture, the Web-based application developed will be testable at every level of implementation. This architecture supports both the functional and security development through the use of TDD, and also supports the ability to extensively test the system after development is complete for both functional and security defects. This architecture becomes the backbone for all testing activities, at both the unit- and system-level. With this type of architectural support, test-cases that support the development of secure applications can be developed quickly, without the overhead of building testing infrastructure found within traditional software projects. The ability to quickly respond to customer security requirements in the form of test-case development is necessary for the creation of secure applications within an agile framework, and the presence of a highly testable architecture ensures that the ability to test the system for security at all levels will be present within the Web-based application.

The third strategy outlines an extensive testing strategy for Web-based applications. This strategy is paramount in the process of secure development within an agile framework. The importance of a security testing strategy is clearly reflected in the dependence of converting security requirements to test-cases within the first strategy. The third strategy outlines a number of test-cases that can be applied to a Web-based application to test for security. It also outlines an efficient open source solution that can be incorporated within current unit-level testing activities to provide unit-level testing of security requirements. This type of testing can be used in conjunction with the FIT framework outlined in the first strategy, and when utilized alongside a highly testable architecture, represents an incredibly powerful test suite for ensuring the functionality and security of a Web-based application.

The fourth and final strategy provides the ability for an agile development team to quickly respond to the security requirements added to a project at the start of any build cycle. Security requirements not being fulfilled (i.e. not passing specific security test-cases) can be quickly implemented using the novel technique of security refactoring. This technique, an extension of the commonly used agile technique of refactoring, defines a number of simple transformations that will enhance the security of existing code, without influencing the functionality of the code. The process of security refactoring is further supported by the ability to incorporate security requirements as test cases. The test cases recorded with the FIT and HTTPUnit frameworks not only constitute
system requirements, but also a set of automated regressions tests that can be used to ensure that any refactor does not modify the functionality of the system.

A major challenge within the development of secure applications is that attackers are always developing new ways of attacking secure systems. This challenge requires that the software development teams developing the Web-based applications stay up with the methods and types of attacks being used against Web-based applications. This paper is a snap-shot of the current situation, and simply cannot provide an extensive list of security measures for all time; however, this paper does outline a high-level strategy for the development of both current and future secure Web-based systems. With the development of new technologies and Web-based user interactions, the implementation of security within Web-based applications will change; however, the methods behind the development need not.

Future work in this area will focus on the further development of security testing strategies, security refactoring and empirical test studies. The security testing strategies outlined represent the current state of the art; however, as mentioned previously, technology and attack strategies are constantly changing, and future testing strategies must reflect these changes. There are a number of potential areas in which security refactoring can be explored. Initially there is a need for more security refactorors. A number of new security refactorors are currently under development. Further work is also being directed towards the development of a refactoring tool to aid programmers in security refactoring. Finally, there is a need for empirical results supporting this development strategy. Currently we are seeking industrial partners to provide case-studies and real world data to support the use of these techniques within an agile development framework. Initial reactions to the proposed development strategy are positive; however, empirical data is required to further validate the proposed strategy.

REFERENCES


