Agile Security Testing of Web-Based Systems via HTTPUnit

A. Tappenden, P. Beatty, J. Miller
University of Alberta
Edmonton, Alberta, Canada T6G 2E1
{andrewt, beatty, jm}@ece.ualberta.ca

A. Geras, M. Smith
University of Calgary
Calgary, Alberta, Canada T2N 1N4
{ageras, smithmr}@ucalgary.ca

Abstract

The technological advancements of web-based systems and the shift to iterative and evolutionary development processes have given rise to the idea of agile security testing, where the principles and practices of agile testing are applied to the domain of security testing. This paper explores common vulnerabilities for web applications and proposes two synergistic approaches for mitigating them. The first approach is to employ a highly testable architecture in the development of web-based systems, and the second is to support the security testing process using the open source unit testing framework HTTPUnit. The overall testing strategy mingles well with agile development efforts and gives the development team an opportunity to produce applications that have the “right” functionality and the “right” level of security.

1 Introduction

An increasingly connected lifestyle and the growing power and capabilities of ubiquitous computing devices have combined to generate an increasing need for security testing in many application development projects. At the same time, many projects are utilizing agile methods such as eXtreme Programming and Scrum to assist them in delivering high-quality applications in an iterative and evolutionary manner. Agile methods are typically viewed as orthogonal to traditional security engineering methods [1] and it is only in recent discussions that these two worlds have been combined [1, 2].

Security is major concern for many web-based software systems in development, as typically these systems are under constant exposure to elements of the outside world, both the good and the bad. Web-based systems are becoming increasingly important in many business models, and for some businesses the web-interface represents the sole point of contact between the business and the consumer. For these systems, security is a major concern as it can represent the possibility for a direct loss of profit, in the case of an attacker manipulating prices to their advantage, or an indirect loss of profit through a denial of service attack. In 2003 Dell’s U.S. home & home office division generated $2.8 billion or approximately 50% of its revenues through its online storefront [3]. The security of web-based systems is critical for Dell operations, as a breach in the security could literally lead to billions of dollars in lost profits.

Security concerns are also of increased importance to any software system that risks leaking private information about individuals to the public or a malicious attacker. These types of security threats may involve both large amounts of money, such as when a bank or other financial institution is involved, and potential litigation, in the case of the exposure of private information. It is clear that the security of any web-based system should be a major priority for any company that is seriously considering entering the e-commerce marketplace.

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 before the target system exists; 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 risks, with the customer ultimately deciding which ones would be of the greatest concern to them. This negotiation of the "good enough security" solution is outside the scope of this paper. Instead, we will focus on the test-driven characteristic of agile security testing and on its application to security testing of web-based applications.

We perceive a growing need for agile security testing based on the combination of the increasing power and complexity of web-based applications [4] and the increasing use of agile software engineering techniques to build them [5]. Development tool vendors are currently
advertising the benefits of smart client architectures (connected, semi-connected, and disconnected), which means that the front-end employing the web-based back-end may not even be a traditional web browser. This raises additional security concerns since these devices may employ HTTP only as a transport layer. The message payload between client and server is therefore not necessarily a one-way request, and it certainly may not only contain HTML. In short, the complexity of modern connected applications places new stresses on web application services. It's reasonable to assume that these new stresses should be represented in the application testing strategy.

In this paper, three aspects of developing secure web-based systems using agile methodologies are described: modeling of security requirements, employing a highly testable architecture, and describing and running automated security tests using HTTPUnit. HTTPUnit is an open source utility for automating tests that target the web application server. The paper will discuss the current strategies and the proposed solution primarily in terms of the graphical user interfaces to web applications.

2 Current Strategies

Currently, many web-based systems utilize the client-side GUI or machine interface to perform input validation, and alarmingly, many rely exclusively on this method to enforce input rules [4]. Validation is done through built-in HTML functions, client-side scripting, or by a web services client application. HTML provides the ability to limit the length of input strings for text fields, and the ability to provide predefined input values through the use of drop-down menus and radio buttons. Client-side scripting languages such as JavaScript can be used to provide the user with an enriched GUI. These scripts can be used not only to enhance the user's experience, but also to provide initial input validation. Scripts can be written to check for bounds conditions, regular expressions, or valid input combinations. Although the built-in HTML validation and scripts are intended to validate inputs before they are sent to the server, they can be bypassed and invalid or malicious inputs may be propagated to the server.

It is clear that the GUI for any web-based system can be easily bypassed; furthermore, the source code for the GUI is exposed and provides implementation information to an attacker. All web-based systems are at risk due to this fact, and systems utilizing client-side input validation are at higher risk due to the increased amount of information available to attackers. Modern web-based systems must therefore be robust in order to ensure security; that is, a system must be able to properly handle unexpected inputs, and simply cannot assume that the information provided by a client is valid. Client-side input validation should only be used to enhance the user's experience and not for the validation of inputs to the server. For example, a webpage which asks the user to enter their billing address could be enhanced using client-side input validation. By asking the user for proper inputs based upon the country selected, such as requesting a postal code rather than a ZIP code for Canadian residents, the usability of the webpage could be improved. This type of input validation and subsequent changes in the GUI can easily be implemented by a scripting language such as JavaScript, and should be included in web-based systems to improve the user interface; however, input should not be assumed valid because of the client-side input validation.

Testing strategies are required that call the robustness of a web-based system into question. Current GUI black-box testing, involving inputs into a GUI and the monitoring of the output, is simply not sufficient for web-based systems due to the fact that the GUI itself can be circumvented. One testing strategy, which is also an attack strategy, is SQL injection [6-8]. SQL injection is the input of SQL-specific reserved characters to the server, in an attempt to alter the manner in which database queries are executed. This manner of testing seeks to evaluate database queries to true based upon information injected into the server either through HTML forms, or through circumventing the client GUI and passing inputs directly to the server. Although this testing strategy is not overly complex and is focused upon only one aspect of a web-based system, it does provide a basis for further work in this area.

Offutt et al. further generalize the idea of SQL injection and outline a testing strategy in the paper “Bypass Testing of Web Applications” [9]. The paper outlines three levels of bypass testing: value level, parameter level, and control flow level. Value level bypass testing involves the manipulation of inputs to a web-server to verify that a system correctly evaluates invalid inputs. Within value level bypass testing, Offutt et al. [9] describe four sub-categories of 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 as opposed to 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,
Parameter level bypass testing focuses on two concepts—the invalid pair of inputs and the required pair of inputs. Offutt et al. [9] provide pseudo code for an algorithm able to crawl through a website and identify all possible input patterns in the web-system. The test strategy proposes the testing of each input pattern to ensure that the system responds correctly to both required and invalid input pairs.

Control flow level bypass testing breaks the normal flow of control within a web-based system and verifies that the system properly responds to invalid system states. This level of testing may be divided into two categories: backward and forward control flow alteration and arbitrary control flow alteration [9]. Both categories subvert the normal flow of control. Backward and forward control flow alteration simply references invalid state transitions one state backward or two states forward and arbitrary control flow alteration references all other invalid state transitions. Control flow bypass testing basically consists of the identification of all invalid state transitions, and the subsequent testing of each. The algorithm devised by Offutt et al. [9] for determining all possible input patterns is also capable of identifying all invalid state transitions.

Although SQL injection and the bypass testing strategy proposed by Offutt et al. [9] provide a good basis for the security testing of web-based systems, they face several limitations in the context of current web-based systems and technologies. The primary limitation of SQL injection is that it only deals with one very specific aspect of a web-based system, the SQL database. SQL injection techniques do not uncover a majority of security flaws that are present within web-based systems, such as the inappropriate use of the hidden form fields or the passing of sensitive information via insecure communication channels or plain-text GET requests.

The bypass testing strategy proposed by Offutt et al. [9] is limited to textual information entered into a web-based system via GET and POST requests. Although these types of inputs are fundamental to the online experience, malicious attackers are not limited to attacking via GET and POST requests. Attacks can come from a variety of different angles and through different types of inputs. Cookies are quickly becoming a third common mechanism of web-interaction along with GET and POST requests. The information stored in a cookie can be manipulated, like the information passed into a web-based system through a request, and can be used maliciously by an attacker. Attacks can also come from sources beyond textual inputs, such as the uploading of malicious files, improper Base64 encodings, and the input of reserved characters from systems other than SQL databases.

In the next Section we propose to elaborate on the secure programming practices and their testing as described above, with a three-part solution. The first part consists of a discussion of the agile capturing of security requirements. The second part is an architectural one, that is, a proposed high-level design that brings testing to the forefront and promotes the testability software quality characteristic. The third solution is an automated testing solution that works in conjunction with the proposed architecture to enable early and continuous feedback on the security of the system under development.

### 3 Proposed Testing Strategies

#### 3.1 Agile Security Requirements Gathering

The project characteristics for web-based systems differ from traditional Information Technology projects in a number of ways. We characterize web-based systems as any project which evolves and is deployed via the World Wide Web. This would include a variety of projects from the field of business and commerce to education and professional development. Projects may range in sophistication from the simple deployment of HTML to applet enhancements in Java and ActiveX. Some systems may have both a web-based and traditional I.T. project components for example a web deployed front end with a significant back end performing bulk calculations or interacting with a database.

Web-based systems are typically much smaller than many traditional projects, and consequently their production period is much shorter—projects lasting from several weeks to a few months in duration are not uncommon. They also tend to evolve at a much faster rate; the “shelf life” of web-based systems may be only a few months, and hence their production is highly iterative in nature. Starting from a largely descriptive overview of the required functionality, rather than potentially a complete and consistent statement of exact system requirements, they often move through a quick build and test cycle, followed immediately with subsequent short build and test cycles. In general, web-based system developers are regularly faced with a potentially endless array of short build and test cycles. Conversely, traditional projects tend to spend a large time in production, following traditional models, such as the waterfall model, followed by release, and the prospect of perfective, adaptive or corrective maintenance.
Traditional ideas for the development of secure software tend to assume the existence of a production model similar to the waterfall model, where a complete set of requirements for a system is produced initially, followed by a lengthy design, implementation and verification process [1]. Although current approaches to secure systems assume a traditional software development process, the realities of web-based systems require an iterative, evolutionary approach to development, not a formal rigorous methodology. For the reality of web-based systems, working code, not large-scale up-front analysis and design, is paramount, and agile development methods are perfectly suited to this domain. To presume that customers who require secure web-based systems are willing to wait for the production of a strict security requirements document and all of the accompanying documentation is simply not realistic.

Scott Ambler, in his book “The Object Primer 3rd Edition: Agile Model Driven Development with UML 2” [10], proposes a number of ways to model requirements within an agile framework. One requirement artifact mentioned by Ambler is a technical requirement. According to Ambler, “a technical requirement pertains to a non-functional aspect of your system, such as a performance-related issue, a reliability issue, or technical environment issue” [11]. These technical requirements could easily be extended to model security requirements, as they are a non-functional aspect of a software system. Technical requirements, much like other requirement artifacts such as features, user stories, and business rule definitions, are understood by the agile community so there should be little difficulty in the acceptance of security requirements modeled as technical requirements.

Agile methodologies are maturing, and agile security requirements capture is now coming to fruition [1, 2]. These security requirements can be expressed as technical requirements, which are accepted by the agile community [11], and hence can be incorporated into existing agile processes. To support the agile development of secure systems, an automatable security testing infrastructure is needed. Sections 3.2 and 3.3 provide a basis for security testing infrastructure that can be integrated into existing agile development processes.

3.2 Highly Testable Architecture

Most web-applications employ a layered architecture that consists of the presentation layer, the business services layer, and the data services layer. In a closed architecture, the presentation layer has access only to the business services layer and cannot interact directly with the data services layer. An open architecture would employ the same layers, but permit the presentation layer to interact with the data services layer to satisfy at least some of its requirements.

In a highly testable architecture (see Figure 1), the architect inserts a test layer on top of each of the base application layers. This means that there is a software layer dedicated to testing the data services layer, another layer dedicated to testing the business services layer, and a third dedicated to testing the presentation layer. In more elaborate variations on this architecture, the business services layer might be further split into two layers—the business entity layer and the process/control layer [12]. A highly testable architecture would again mirror these additional layers with test layers.

![Figure 1. A highly testable architecture](image)

The resulting architecture is useful for security testing since various security test patterns can be employed within any number of the test layers. Bypass testing as described above could target an individual layer (probably in conjunction with the use of mock objects to isolate the target layer) or be more of an integration test by targeting an upper layer and the layers that it depends on. Tools such as FIT employ bypass testing for testing the business functionality of the system under development [13] and they have been, anecdotally at least, effective in that role. We speculate that the benefits of using test-first, business-oriented integration tests extends to test-first, security-oriented tests as well, especially given that bypass testing is one of the key strategies for both.

3.3 Automated Security Testing via HTTPUnit

The ability to bypass client-side GUIs constitutes a doubled-edged sword. On one hand, there is the potential for a breach of security, but on the other hand, bypassing of client-side GUIs allows for automated testing of web-based systems. HTTPUnit [14] is a unit-test framework...
that provides the ability to bypass the client GUI by emulating relevant portions of the web browser. Currently, HTTPUnit is able to emulate form submission, JavaScript, basic HTTP authentication, cookies, and automatic page redirection. It also allows Java test code to examine returned pages either as text, a XML DOM, or containers of forms, tables, and links [14].

One of the most important features of HTTPUnit is its ability to analyze returned pages in a variety of ways. With any kind of automated testing, one of the most important aspects is the assessment of a pass or fail criterion to a response. HTTPUnit provides functionality to aid in the identification of responses; however, developers can also place a unique identifier in every web-page or server response. HTML provides the ability to add comments to documents—a feature which developers can take advantage of in order to simplify the assessment of the pass or fail criterion.

Hightower et al. provides an initial basis for the use of HTTPUnit in the agile development process in the book “Java Tools for Extreme Programming: Mastering Open Source Tools Including Ant, JUnit, and Cactus” [15]. Hightower proposes HTTPUnit as a tool that can facilitate both functional and unit testing of web-based systems. This reflects a very traditional view of the server-client relationship, one that attackers use to their advantage when attacking web-based systems. This Section will focus on the novel use of HTTPUnit to directly test the web-server and completely bypass all client-side validations and restrictions, allowing for the testing of the web-server directly.

HTTPUnit was designed to bypass cumbersome GUI interactions with the web-browser but it was not originally intended for use in testing web-based systems as described in this paper. HTTPUnit was designed to, by default, accept all client-side input constraints. This allows for a traditional, functional black-box testing approach to be performed on the system; however, this default will not allow for client bypass testing. Disabling client-side verification is therefore a necessary step in the process of bypass testing using HTTPUnit.

WebConversation wc = new WebConversation();
WebRequest request = new GetWebRequest("http://localhost/");
WebResponse response = wc.getResponse(request);

The first example consists of HTML code which takes a number from the user and divides it by the predefined values of one, two, or three, depending on the user’s input. The example utilizes HTML input length constraints and predefined values to restrict the input:

```
<form action="./" method="POST">
  Input a dividend:
  <input type="text" name="dividend" size="6" maxlength="6"><p>
  Choose a divisor:
  <select name="divisor" size="1">
    <option value="1">one</option>
    <option value="2">two</option>
    <option value="3">three</option>
  </select><p>
  <input name="submit" value="Divide!" type="submit">
</form>
```

Although this is a very simple and straightforward example, a number of potential vulnerabilities do exist. The first vulnerability is the possibility for an attacker to cause a divide by zero condition. This is a notorious error that has crashed many computer systems. If the system assumes that the divisor will be one of the three possible values, one, two, or three, constrained by the client-side GUI, then bypassing the predefined selection criteria and inputting a zero will most likely cause an error. Another point of attack for this simple interface would be to bypass the built-in HTML length validation. With the length validation removed, an arbitrary length string could be passed into the system as a valid input. This is a common programming mistake made in every domain and can lead to a buffer-overflow.

The following test case addresses the divide by zero possibility:

```
WebConversation wc = new WebConversation();
WebRequest request = new GetMethodWebRequest("http://localhost/");
WebResponse response = wc.getResponse(request);
```
A second example of bypass testing using HTTPUnit involves a simple login page. The relevant code for the site is provided below:

```html
<form action="./" method="POST">
  Username :
  <input type="text" name="user" size="10"> <p>
  Password :
  <input type="text" name="password" size="10"> <p>
  <input type="hidden" name="state" value="login">
  <p><input name="submit" value="Login" type="submit">
</form>
```

Two vulnerabilities will be tested in the following test case. The first vulnerability is the existence of a hidden field which can be modified, potentially leading the system into an unexpected state. Unexpected state transition is a reality for web-based systems due to the user’s ability to simply move backwards and forwards using the web-browser; however, unexpected state transitions can pose a serious security threat to web-based systems. The hidden input in the above example may be used to put the system into the login state. If the name of the logged-in state is known by an attacker, he or she may be able to circumvent the user identification process. The second vulnerability that will be tested is the potential for SQL injection. It may be possible to place SQL reserved words into the form fields and gain access to the system. The following test case can be used to test for both hidden field manipulation and SQL injection:

```java
WebRequest form = response.getForms()[0].newUnvalidateRequest();

form.setParameter("dividend","10");
form.setParameter("divisor","0");
response = wc.getResource(form);

assertTrue(response.getText().trim().split("\n")[0].matches("ERROR"));
```

A second test case is required to ensure that the system properly constrains the input from the text field:

```java
WebRequest request = new GetMethodWebRequest("http://localhost/");
WebResponse response = wc.getResponse(request);
WebRequest form = response.getForms()[0].newUnvalidateRequest();

form.setParameter("dividend","1234567");
form.setParameter("divisor","1");
response = wc.getResource(form);

assertTrue(response.getText().trim().split("\n")[0].matches("ERROR"));
```

This test case ensures that the system has been tested against input strings longer than six characters. Server-side input validation directly combats potential buffer-overflow vulnerabilities.

The example test cases given above are very simple, but they do provide an introduction to the use of HTTPUnit for testing the robustness of a system. There are a huge number of test cases that could be created for almost any input interface. The previous examples focus on the system input constraints, but type checking, parameter relationships, and control-flow can also be tested using HTTPUnit. Once the initial framework is established for the navigation to a particular system state, the inputs to that state can be modified with ease, allowing for large numbers of test cases to be generated.
form.setParameter("state","loggedin");

response = wc.getResource(form);

assertTrue(response.getText().trim().split("\n") [0].matches("ERROR"));

This example test case shows how simple it is to test for SQL injection and hidden field manipulation within the HTTPUnit testing framework. The parameters can be manipulated in a variety of ways and each permutation can be used as a test case for unit or acceptance testing.

In the above examples, the pass-fail criterion for the test cases is evaluated by the assertTrue function. Invalid input is provided to the server so that the functions evaluate to an error condition. A simple check for the presence of the word “ERROR” is all that is required to ensure that the server correctly identifies the invalid inputs. These simple test cases illustrate the need for unique error-condition identifiers to be embedded into the source code, since without the identifiers, the process of verifying the validity of the output can become a very lengthy undertaking.

4 Extending Bypass Testing

SQL injection, buffer overflows, cross site scripting, file inclusion, URL injection, and remote code injection vulnerabilities are common ailments that plague modern web-based systems. These vulnerabilities are real and are continually exploited by attackers on systems susceptible to these types of attacks. The web-based application phpMyAdmin is used for online administration of a MySQL database over the Internet. Versions 2.6.1 and earlier of this product include several security issues such as cross-site scripting and file inclusion vulnerabilities [16] stemming from improper validation of inputs. The popular web-based application ProjectBB is a bulletin board application that is used mainly in France. This application has cross-site scripting and SQL injection vulnerabilities [17]. Finally, Microsoft Outlook Web Access suffers from a URL injection vulnerability that allows attackers to redirect users to a malicious counterfeit login page [18]. These software applications all contain security vulnerabilities due to the improper validation of inputs, and are not merely simple websites maintained by inexperienced developers, but rather by both the members of the open-source community and by employees of Microsoft, one of the largest software development companies in the world.

The SQL injection and bypass testing strategies discussed in Section 2 provide an initial basis for verifying the security of a web-based system from the perspective of malicious inputs. However, the input space currently available for a modern web-based system is so large that there are many input methods overlooked by these strategies. There are several areas in which the testing strategies could be extended, including cookie support, file support, base64 encodings, and consideration of non-Unix based web servers. Although these types of inputs are not valid for every web-based system, any system using these inputs needs to be tested for robustness and security with emphasis placed on the untrustworthy nature of the inputs.

Cookies, as previously mentioned, are a staple of modern web-based systems and thus need to be addressed as part of a bypass testing strategy. Cookies can be affected in three primary ways—by creation, modification, and deletion. Data stored in cookies can be modified in the same manner as the data entered to the system by bypassing the client GUI. Web-based systems should be tested against cookie modification, spoofing, and deletion. The same principles suggested for testing data inputs should be extended to cookie modification, with further test-cases consisting of the spontaneous deletion and creation of cookies at a variety of different states in the control-flow of a web-based system. Currently, HTTPUnit provides the ability to manipulate cookies in two primary ways—through creation and deletion [19]. Creation and modification of cookies is relatively straight forward and can both be accomplished through the creation of a cookie. The example below demonstrates how one could create a cookie, and then overwrite or modify the cookie's value for subsequent web transactions:

```java
WebConversation wc = new WebConversation();
WebRequest request = new GetMethodWebRequest("http://localhost/cart.php");
wc.putCookie("id","001");
WebResponse response = wc.getResponse(request);
wc.putCookie("id","002");
WebLink link = resp.getLinkWith("checkout");
link.click();
```

The above example initially accesses the web-based system with the cookie id set to the value 001. This could be reflective of a shopping cart based web-system. The example then modifies the value of the cookie, from
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]));
        }
    }
}

The procedure shown above acts as a stand-alone procedure that can be added to a test-script itself; however, it could also be fitted into the HTTPUnit code as a method inside of the WebConversation class.

Many modern web-based systems interact with the user through the use of files. Online e-mail services provide the ability to attach files to messages, while online dating services allow users to post their pictures. Images, documents, and other files are continually being submitted to online services and need to be validated. The server must be able to detect invalid images or documents before they are utilized by a web-based system. This issue is further complicated by the current computer viruses that can infect users’ systems through the presentation of simple JPEG images [20].

Files can pose several threats to a web-based system, and the file type needs to be validated by the system. File type validation is not as simple as merely checking the extension of the file, as an attacker can spoof these features. Other problems arise when filenames are too long, or the file size is too big for the system or web-server. Problems can also occur due to the uploading of malicious files, such as a JPEG image with a virus payload. It is clear that all input files must be inspected and validated by the web-server. HTTPUnit provides the ability to upload files as part of the automated testing process through the UploadFileSpec class as well as through built-in file functions in the Form class [19].

Base64 is an encoding used frequently on the internet for the encoding of binary data into a series of seemingly random ASCII characters [21, 22]. This encoding allows for the transfer of binary data over any communication channel that utilizes plain ASCII text only, such as e-mail services. Inputs of this type to web-based systems need to ensure the validity of the incoming base64 encoded data. This type of testing should focus primarily on the base64 encoding itself. Any base64 encoded data will contain a multiple of three ASCII characters that are valid within the base64 alphabet. Testing of these types of inputs should consist of invalid character introduction and valid character addition or deletion. Once a base64 input has been validated, it is important that the underlying binary file again be verified as discussed above. HTTPUnit provides the ability to encode and decode data in base64 encodings and can be used to automate the testing process through the Base64 class [19].

The Internet is a collection of heterogeneous computing units networked together, and any testing strategy for web-based systems must be capable of testing a wide range of potential systems. When testing against reserved input characters, testers must be careful to choose special characters relevant to the operating environment used on the server. Reserved characters in a Unix-based platform, for example, are different than those in a Windows-based platform. Also, one must be cautious of the database syntax used when selecting reserved characters, as not every database follows the SQL framework. Another set of reserved characters that developers need to be aware of is XML characters, as many of the inputs to web-based systems are translated into XML documents and distributed. Reserved words and database injection can be implemented in HTTPUnit with little or no adaptation as discussed in Section 3.3.

It is clear that all inputs into a web-based system need to be validated by the server. Inputs need to undergo checks for length, validity, character set (ASCII, Unicode,
etc), reserved characters, and type (String, Integer, Real, etc). These checks can eliminate many of the security vulnerabilities that affect web-based systems. Table 1 provides a summary of the input types and the test inputs used to verify the server verification of inputs. The table also highlights the differences between the security testing strategies proposed in the paper and the strategies proposed by SQL injection [6-8] and Offutt et al. [9].

5 Conclusion

Modern web-based systems represent the union of several complex heterogeneous units. These units work together to provide services to users over the internet. Web-based systems must be tested for robustness, as they are open to attack from users at all times. The testing of these systems must be automated, representative of possible interactions between clients and servers, and be iterative and evolutionary, just as the development effort is iterative and evolutionary. These requirements can be met by employing a highly testable architecture and using an automated testing framework that can bypass the presentation layers and interact directly with the underlying web application server via HTTPUnit. This gives the team the ability to perform security testing for critical vulnerabilities that are best mitigated by secure programming practices on the web application server. The recommended testing strategy is to use a highly testable architecture, to perform security testing using HTTPUnit, and to perform functional testing using a tool such as FIT.

<table>
<thead>
<tr>
<th>Valid Input</th>
<th>Test Input</th>
<th>SQL Injection</th>
<th>Offutt et al.</th>
<th>Our Approach</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>X</td>
</tr>
<tr>
<td></td>
<td>Unix reserved characters</td>
<td>X</td>
<td></td>
<td>X</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></td>
<td>X</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>X</td>
</tr>
<tr>
<td>Integer input</td>
<td>Min/max values</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Zero</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Real number</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Non-number</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Real Number input</td>
<td>Min/max values</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Zero</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Non-number</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Cookie</td>
<td>Create/Modify/Destroy cookie</td>
<td>X</td>
<td></td>
<td></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>X</td>
</tr>
<tr>
<td></td>
<td>(Generic, Real, Integer, etc)</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>File input</td>
<td>Different file type</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Invalid file of type</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Long file name</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Malicious file (Script, image)</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>File size</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Base64 encoding</td>
<td>Invalid Base64 encoding</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Improper Base64 alphabet</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Invalid underlying file</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It is essential that any testing-based security mechanism covers all of these aspects of web-based security. Without such rigorous efforts, a dynamic website can not be ensured of its integrity.
References


