An Early Testing and Defense Web Application Framework for Malicious Input Attacks

Michael Gegick[1], Eric Isakson[2], Laurie Williams[1]

[1]Department of Computer Science, North Carolina State University
[2]Independent
Raleigh, NC 27695
1-919-513-4151
{mcgegick, lawilli3}@ncsu.edu
eisakson@nc.rr.com

ABSTRACT

Input validation vulnerabilities, one of the largest problems in software security today, are readily identified by software assurance (SA) tools. Software development organizations are increasingly adopting SA tools to quickly identify vulnerabilities in their software systems. These tools, usually applied when implementation is complete, have a comprehensive and extendable rule set to detect known and new vulnerabilities. A *simple* and *effective* framework is needed to provide developers with a strategic approach to securing against malicious input attacks early in software development. We introduce a Java **W**eb **A**pplication **R**eliability and **D**efense (WARD) framework, a two-part security solution composed of a vulnerability detection component, SecureUnit, and a vulnerability protection component, SecureFilter. SecureUnit enables developers to write automated, reusable, and customizable JUnit penetration tests that launch attacks on their systems to reveal security vulnerabilities. SecureFilter is a customizable server-side choke point containing a regular expression-based filter to match legal input according to system requirements. WARD provides an attack-then-defend approach for developers to build security into a software system early in the software process. We integrated WARD v1.0 with WebGoat, an open-source web application security test bed, and successfully “warded off” 38 of 43 (88%) injected cross-site scripting exploits. WARD v2.0 will address the encoded (e.g. with HTML entities, hex characters) exploits that were not stopped from entering WebGoat in WARD v1.0.

Categories and Subject Descriptors

D.2.0 **[Software Engineering]:** General - *Protection mechanisms, standards.*

General Terms

Security, Verification

Keywords

reference validation mechanism, principles, security, security testing, penetration testing, cross-site scripting, application firewall

1. INTRODUCTION

Input validation vulnerabilities, one of the largest problems in software security today, are readily identified by software assurance (SA) tools. Input validation problems are weaknesses in a software system where developers trust that input is benign [8]. According to the Open Web Application Security Project (OWASP), four of the top ten[1] vulnerabilities for web applications are input validation problems. Software development organizations are increasingly adopting SA tools to quickly identify these vulnerabilities and others in their software systems. These tools, usually applied when implementation is complete, have a comprehensive and extendible rule set to detect known and new vulnerabilities [13]. However, addressing problems late in the software process is more costly and less effective than a proactive approach where developers can build security into their software [5]. A

[1] http://www.owasp.org/documentation/topten.html

simple and effective framework is needed to provide developers with a means to secure against malicious input attacks early in software development. *Our research objective is to provide developers with the combined ability of attacking (via testing) and defending their web applications from invalid input early in the software process*. Approaching software security[2] with a two-sided effort is captured in the idea that:

…[there are] two sides of software security – attack and defense, exploiting and designing, breaking and building--into a coherent whole. Like the yin and the yang, software security requires a careful balance. [13]

Preventing malicious input from entering a software system can be achieved with filtering against a white list[3]. Inputs to software products are likely to be unique due to the differing requirements specifications and thus a predefined/universal security mechanism is not an accurate solution for validating input.

To provide developers with a lightweight approach to white listing and penetration testing with a black list[4], we introduce the Java **W**eb **A**pplication **R**eliability and **D**efense (WARD) framework. WARD[5] is a two-part security solution composed of a vulnerability detection component, SecureUnit, and a vulnerability protection component, SecureFilter (see Figure 1). SecureUnit enables developers to write automated, reusable, and customizable JUnit penetration tests that attack their systems to reveal security vulnerabilities early in the development process. SecureUnit utilizes the JUnit[6] and HttpUnit[7] frameworks to perform unit-level[8] penetration testing via inserting exploits into a test case and injecting the exploits into the GUI of a Web application. SecureFilter is a customizable server-side choke point containing a regular expression-based filter that developers incorporate and modify to match legal input as defined by their requirements. A choke point is small, easily-defined interface through which data (e.g. user input) flow into and out of a system [20]. Security testing should start at the unit-level and proceed to the system-level once the system is integrated [13]. The WARD framework provides a means to unit test a Web application early in the development process for a software security approach to securing a software system.

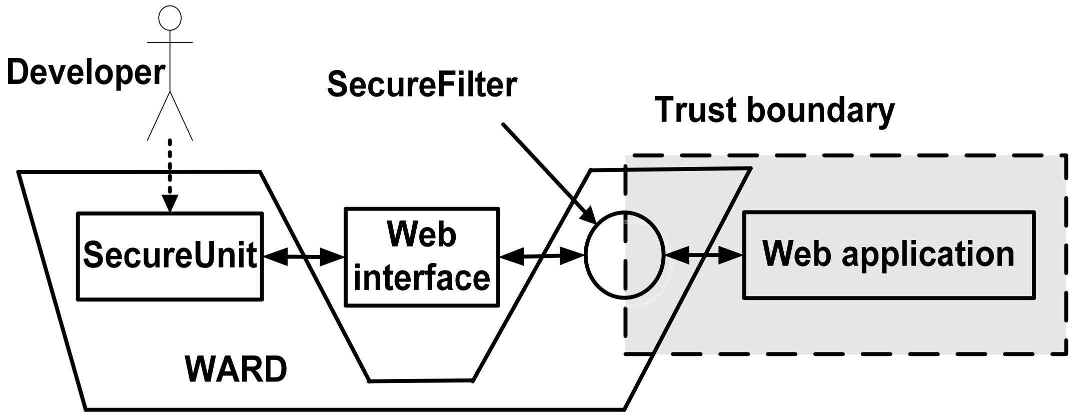


Figure 1: WARD is a Java Web application security framework consisting of SecureUnit, a penetration testing component, and SecureFilter, a filter-based component for guarding against security attacks.

WARD employs configuration files containing regular expressions to represent both white lists and black lists. WARD also contains an *exploit library* (EL), a configuration file, that represents specific instances of malicious input as defined by the black list. Developers have two main responsibilities to initialize WARD: (1) modify or

[2] The practice of building security into the software [8]
[3] A white list defines (e.g. with a regular expression) valid input as specified by the software system requirements. [8]
[4] A black list defines (e.g. with a regular expression) any input not defined as valid by the software system requirements. [8]
[5] WARD will be released soon as a free and open source software project.
[6] http://junit.org/index.htm
[7] http://httpunit.sourceforge.net/
[8] Logically separable parts of a computer program [12]

override the configuration files according to their specific forms of valid/invalid input (2) create a SecureUnit test to inject exploits from the EL into their Web application interface. After initialization, developers can test their Web applications to determine if their configuration files correctly filter input in SecureFilter. SecureFilter is not invasive to the developer's code, which is a result of the goal to provide a facilitated approach for matching valid/invalid inputs.

We tailored our configuration files for cross-site scripting (XSS) attacks in WARD 1.0 because it is an attack in which input filtering is the primary defense [9, 10]. WARD can be used to filter HTTP input and thus can defend against other malicious input attacks such as directory traversals and SQL injection attacks.

To examine the effectiveness of WARD at preventing attacks, we integrated WARD v1.0 with WebGoat[9], an open-source web application security test bed. We inserted the explicit XSS examples documented in [9] into our EL and performed penetration testing to determine the efficacy of filtering input. The remainder of this paper is as follows: in Section 2 we provide background information, in Section 3 we introduce WARD, in Section 4 we show the results from our tests, in Section 5 we indicate our limitations, and in Section 6 we describe future work, and in Section 7 we summarize and conclude our work.

2. BACKGROUND

We now discuss the fundamental principles of software security. A discussion of cross-site scripting attacks is given in Section 2.1, followed by an introduction to the principles of software security in Section 2.2, a brief survey of SA tools follows in Section 2.3, and in introduction to TDD is provided in Section 2.4.

2.1 Cross-site Scripting (XSS)

A XSS vulnerability is a flaw that occurs when untrusted user input is read by a victim's web browser [9]. The malicious input is usually in the form of a script (e.g. JavaScript, Jscript, VBscript, ActiveX) or embedded object (e.g. <APPLET>) [9, 10] An example of an XSS exploit is

```
    <script>baseForm.cookie.value=document.cookie;baseForm.submit();</script>
```

where an attacker can obtain a victim's session information or cookies. When a victim visits the web page, the script is executed on the victim's machine (see Figure 2). In this way, an attacker can gather confidential information or steal user's credentials.

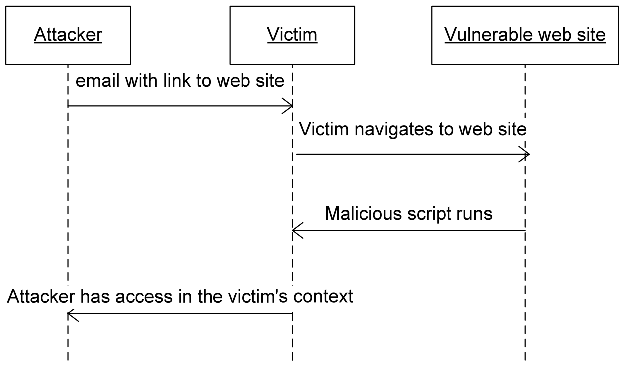

Attacker
Victim
Vulnerable web site
email with link to web site
Victim navigates to web site
Malicious script runs
Attacker has access in the victim's context

Figure 2: Sequence diagram of a XSS attack

XSS exploits can contain characters such as "(", ")", ";", which can also exist on a system's white list. Therefore, a predefined white list such as alphaNum[10] may not suffice for filtering XSS exploits.

[9] http://suif.stanford.edu/~livshits/securibench/
[10] All alphabetical and numerical characters

Malicious input can come from objects such as HTML forms, HTML headers, database queries, cookies, and certain HTML tags (e.g. anchor tag, image tag) [9, 10]. Input with a percent sign in it can be masking a script by utilizing hexadecimal encoding (or other encoding methods) for malicious characters to make it less suspicious. Any web browser supporting scripting is potentially vulnerable to XSS exploits [9]. The attacker only needs the name of a web server (including those inside a firewall) that does not validate data input via a web page. XSS attacks are resistant to security mechanisms such as SSL and TSL [9] thus defending against XSS must be accomplished with software security. Furthermore, developers cannot approach XSS by carefully setting access rights because the XSS exploits execute on a client's machine, which can be outside the domain of a developer's control.

2.2 Saltzer/Schroeder and the Twelve Fundamental Principles of Software Security

Principles[11] exist in software security to provide software engineers with the fundamental knowledge needed to build security into a software system. The first established principles in security were applied to protection mechanisms. A protection mechanism is responsible for controlling access to information in executing programs [17] and should return an error when an illegal attempt to access confidential information occurs [4]. Originally, security practitioners only knew how to create their protection mechanisms based on experiences of others that were spread by word of mouth. In fact, some of these experiences were conceived in contexts other than security. In 1975, Jerome Saltzer and Michael Schroeder wrote "The Protection of Information Systems" [17] which focuses on architectural structures (hardware or software) that are needed to protect information in computer systems. In their work, they formalize the experiences of developing protection mechanisms into the following eight design principles: *economy of mechanism*, *fail-safe defaults*, *complete mediation*, *open design*, *separation of privilege*, *least privilege*, *least common mechanism*, and *psychological acceptability* [17]. These principles are intended to be used to guide the design and implementation of protection mechanisms regardless of the programming language of the software system.

The Build Security In web portal[12] has documented 12 principles of software security that originate from seven of the eight design principles published by Saltzer and Schroeder. The additional five principles are well-recognized in security literature [4, 8, 15, 18, 20] and are thus incorporated in the portal: *securing the weakest link, defense in depth, reluctance to trust, never assuming that your secrets are safe, and promoting privacy*. Below is a brief description of each principle.

Principle of Complete Mediation – each and every access to any object should be checked for permission.

Principle of Defense in Depth – multiple fortifications should be used to defend an asset.

Principle of Economy of Mechanism – designs should be as simple and as small as possible.

Principle of Failing Securely – a system's fail state should be secure from attack.

Principle of Least Common Mechanism – users/objects should not share mechanisms that grant access to resources.

Principle of Least Privilege – only the absolute least privileges necessary to perform an operation should be granted to a user/object.

Principle of Never Assuming that your Secrets are Safe – anticipate that your secrets will be found out.

Principle of Promoting Privacy – information about users, systems, and objects should be kept private.

Principle of Psychological Acceptability – security mechanisms should be as transparent to developers and users as possible to prevent complexity and maintain ease of use of the system.

Principle of Reluctance to Trust – in the context of security, it's always good too be paranoid when it comes to trusting people, operations, and technology.

[11] A principle is a comprehensive and fundamental law, doctrine, or assumption (http://www.m-w.com/dictionary/principle).
[12] https://buildsecurityin.us-cert.gov/portal/article/knowledge/Principles

Principle of Securing the Weakest Link – attackers often attack the weakest aspect of a system. Secure the low hanging fruit of a software system.

Principle of Separation of Privilege – multiple conditions should be met for an object to be granted permission to a resource.

The principle of *open design* provided by Saltzer and Schroeder primarily applies to cryptographic solutions and is thus left out of the principles of software security. Contrary to open design, the principle of *never assuming that your secrets are safe* suggests that secrecy should be kept, but also points out that are secrets are easy to find out.

These principles do not necessarily guarantee security for there is no such thing as a 100% secured system [20]. Also, some of these principles contradict each other. The principle of *defense in depth* indicates that there should be multiple security checks, but the increase in complexity is discouraged by idea of *economy of mechanism*. Security engineers need to balance the principles according to their system's requirements [2].

2.3 Security Software Assurance Tools

Many SA tools exist to detect security vulnerabilities in software systems for developers. The essential factor that determines which tool a development team should use is the degree to which a tool "knows" about specific vulnerabilities [13] . Some SA tools also provide a feature or framework that aid in fortification efforts once a vulnerability is detected. We give a brief survey of SA tools that are capable of detecting malicious input attacks such as XSS and SQL injection.

Freely-available tools such as Nessus[13] and Nikto[14] can detect many of the vulnerabilities documented in the Common Vulnerabilities and Exposures[15] (CVE) list. The Nessus tool contains a plug-in for each vulnerability and can identify over 10,000 vulnerabilities. Nikto specializes its analysis to web server scans and can detect over 3,200 potentially dangerous files or CGIs. The Web Application Vulnerability and Scanner (WAVES) [11] is another tools for detecting vulnerabilities such as SQL injection and XSS attacks. WAVES contains a "crawler" that reverse engineers the web application to find all entry points. WAVES then injects faults into those entry points to determine if a SQL injection attack is possible. Unfortunately, developers tend to use these tools late in the software process when it may be too late to adequately secure their systems.

Tools also exist to help with defensive efforts. DevInspect[16] is a static/dynamic analyzer for .NET applications that detects vulnerabilities and indicates where in the code a vulnerability exists and automatically generates a test case to show how an attack can occur. DevInspect also aids in defensive efforts by automatically inserting a regular expression (serving as a white list) by the vulnerable code to filter malicious or malformed requests. Developers also have the option of making custom fortifications to vulnerabilities in their code that can be further tested. Stinger[17] is an HTTP request validation framework that developers modify to filter malicious or malformed input within their web JSPs[18] (or servlets). With Stinger, developers can create regular expression-based white lists that reside in XML files and apply those white lists to their code.

By limiting input to that of which is specified by a white list, Stinger and DevInspect conform to the principle of least privilege [8]. Stinger and DevInspect also adhere to psychological acceptability because they provide easy-to-use means to securing a software system. The author of Stinger suggests that having security code in the same place as the functional code is advantageous because if the functional code changes, then the developer can easily change the security code due to the proximity to the code. He also suggests that the Stinger framework can be adapted for use with J2EE filters, but to our knowledge a filter implementation to support Stinger does not exist. SecureFilter is an implementation of the J2EE Filter interface that provides developers with means to write and utilize their white lists.

[13] http://www.nessus.org/
[14] http://www.cirt.net/code/nikto.shtml
[15] A list of standardized names of vulnerabilities and security exposures (http://cve.mitre.org/).
[16] http://www.spidynamics.com/
[17] http://www.owasp.org/software/validation/stinger.html
[18] Java Server Page – a program that runs on a Web server and builds a Web page (http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/)

A filtering mechanism called the requestValidation attribute is also available in the .NET framework [9]. If a XSS exploit is entered into the system, then the .NET framework refuses the exploit by not allowing it into the application. The ASP[19] manual pages encourage the use of the requestValidation, but also note that it is not a panacea because the Microsoft developers cannot determine which data are malicious and which are innocuous for a given application. We will show that WARD provides a means of testing for security vulnerabilities and then an effective means of fortifying those vulnerabilities with SecureFilter, based on the fundamental principles of software security. WARD achieves these results without being invasive to a developer's code.

2.4 Test-Driven Development

Test-driven development (TDD) [3] is an testing technique that provides a strategy for unit testing. Software engineers begin writing tests for their code before they code the system. Next, they write the implementation code to pass their tests and then run the tests to determine if their code passes the tests. This paradigm continues in small iterations until software development is complete. The advantage of TDD over testing after a system is complete is that software engineers are given immediate notice of what defects are present in their system during the coding phase. We employ the TDD philosophy because we concentrate on building security into a software system, which requires software engineering strategies. Application security, the practice of securing an application after development is complete, is disadvantageous because of the increased cost to fix a vulnerability late in the software process [8]. WARD provides a framework for developers to adopt TDD in an attack-then-defend paradigm to secure against malicious input attacks.

3. WEB APPLICATION RELIABILITY AND DEFENSE (WARD) FRAMEWORK

WARD evolves the *test-then-code* paradigm from test-driven development (TDD) [3] to an *attack-then-defend* approach to build security into a software system early in the software process. SecureUnit supports the identification of vulnerabilities in a Web application via black-box, unit-level, penetration testing. Unit testing is a low-level testing approach usually performed while the code is implemented [16]. Penetration testing is a technique where developers/testers write tests that attempt to cause a software system to misbehave or violate a security policy [13]. Drawing from both types of testing approaches, SecureUnit tests inputs to Web applications to determine if the white lists correspond to the system requirements. SecureFilter complements SecureUnit by providing developers with a means to defend against an attack. The WARD framework is intended to provide software engineers with a simple and effective in-process means to fortify software systems from malicious input attacks.

At the low-level, developers use the WARD framework to place and utilize white lists defined by their requirements. White lists guard the digital assets in a software system from malicious data injected into input fields in the Web interface. Developers can test the robustness of their white lists with exploits in a default EL or with their own EL (see Figure 3).

[19] Active Server Page – a Web page that contains server-side scripts and HTML (http://www.asptutorial.info/learn/Introduction.asp)

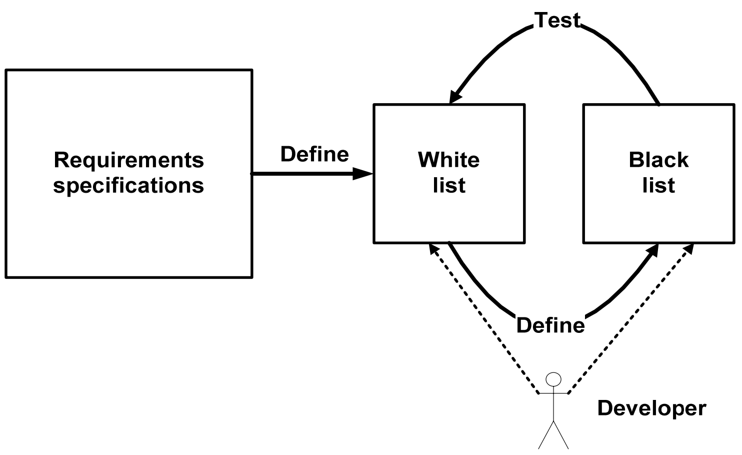


Figure 3: Developers have the responsibility of creating a white list and testing it with an specific instances of exploits defined by a black list

In Section 3.1 we introduce SecureUnit, and in 3.2 we introduce SecureFilter. In Section 3.3 we show how SecureFilter follows the principles of software security.

3.1 Attacking with SecureUnit

SecureUnit is a unit-level black-box penetration testing tool that identifies input validation vulnerabilities in a Web application. To support the black-box testing aspect the web application environment in which WARD resides, we employ HttpUnit, a Java-based black box testing framework. HttpUnit emulates browser functionality such as form submissions, cookies, basic authentication, and JavaScript. JUnit uses HttpUnit as a test fixture[20] to handle test that are submitted to a Web application and the response sent back by the server. Each test contains an exploit that is automatically read from either a default EL in WARD or modified/new EL created by the developer. The exploit library contains instances of strings defined by the developer's black list. Since a black list is potentially infinite, the EL is only a subset of the black list. Thus, passing SecureUnit tests only represent that the tests are passed and not necessarily if the application is secured [7]. Because SecureUnit is based on the JUnit framework, developers can perform automated regression tests when application code is modified (see Figure 4).

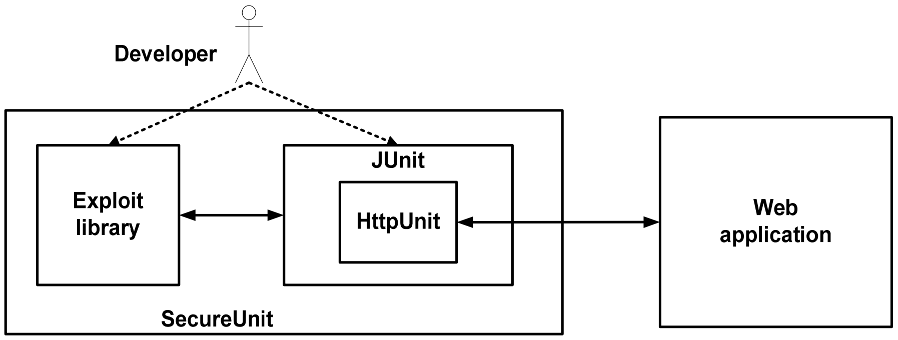


Figure 4: SecureUnit uses HttpUnit, JUnit, and an exploit library to launch attacks at a Web application.

The WARD framework returns a 403 SC_FORBIDDEN[21] response to the developer/attacker if malicious input is detected (see Section 3.2). Developers may also provide a custom error page that describes why the input was

[20] The set of objects that a test is run against (http://junit.sourceforge.net/doc/cookbook/cookbook.htm)
[21] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

rejected. This status code indicates that the server understood the request, but refused to fulfill it. When attempting to input malicious input, SecureUnit indicates a passed test case if a 403 SC_FORBIDDEN status code is returned (indicating the input was stopped), otherwise the test fails. HttpUnit provides SecureUnit with the ability to inject exploits in headers, cookies, and parameters for HTTP GET and POST requests. For example, using HttpUnit we can inject a XSS exploit into a Web form and submit the form to a Web server that contains a XSS vulnerability to test for the presence of a XSS vulnerability. The following code snippet (see Figure 5) is an example of how a XSS exploit can be sent to a Web server using HttpUnit.

```
String exploit = “<a src=javascript:alert('sploit')'>Ha gotcha</a>”;
String url = "http://127.0.0.1:8080/ward-demo";
request = new GetMethodWebRequest(url);
request.setParameter("aField", exploit);
response = conversation.sendRequest(request);
```

Figure 5: An HttpUnit fixture that sends an XSS exploit to a web application via a form field

The practice of injecting malformed, malicious, or random data to uncover faults is commonly called fuzzing [13, 14]. In WARD v1.0 we provide a default EL containing 43 XSS exploits. Using our penetration tests with the malicious data defined in the EL is a version of fuzzing. In WARD v1.0, developers must write their own HttpUnit fixtures that are adapted to their Web interfaces and corresponding JUnit tests. Also, they must indicate which configuration file containing the EL that SecureUnit reads to launch attacks at the Web application. We provide a snippet of a SecureUnit test that is initialized with an HttpUnit fixture in Figure 6. The assert methods pass or fail a penetration test based on the server response from the HTTP request. If the SecureUnit test fails, the developer is warned that a vulnerability potentially exists in their system.

```
isForbidden(){
  WebGoatBrowserEmulator emulator = getFixture(exploit);
  boolean isForbidden = false;
  try{
    emulator.attack();}
  catch(HttpException e){
    if(e.getResponseCode() ==  403){
      isForbidden = true;
    }
    else
      throw e;
  }
  return isForbidden;
}
public void testAllowed() throws Exception {
        assertFalse("application denied valid input 'foobar'", isForbidden("foobar"));}

private static void assertForbidden(String exploit) throws Exception {
  assertTrue("application is vulnerable and did not defend against exploit /"+exploit+"/",
    isForbidden(exploit));}
```

Figure 6: A SecureUnit test that injects an exploit into an HttpUnit fixture and tests the response.

3.2 Defending with SecureFilter

SecureFilter is an implementation of the J2EE `Filter` Interface, an API that filters HTTP requests and/or responses to/from a Java servlet container (e.g. Apache Tomcat). SecureFilter motivates the idea that matching input against a white list is more feasible than if done against a potentially infinite black list [8, 9]. Incorporating SecureFilter into a Web application requires only that a developer add the SecureFilter JAR file to the WEB-

INF/lib directory and modify their web.xml file so the servlet container routes requests to SecureFilter. The developer's main responsibility with SecureFilter is to strive to maintain a well-formed white list that strictly adheres to the requirements; a black list provides a second check to the input, but is not intended to be a surrogate for the white list. The filtering is provided by SecureFilter and occurs outside of the developer's code.

SecureFilter takes as input HTTP headers, cookies and parameters, decodes the data, then passes the data through a white list and a black list, as shown in Figure 7. Input is permitted into the trust boundary when the data are matched by the white list and do not match the exploit library, otherwise a 403 SC_FORBIDDEN response is returned.

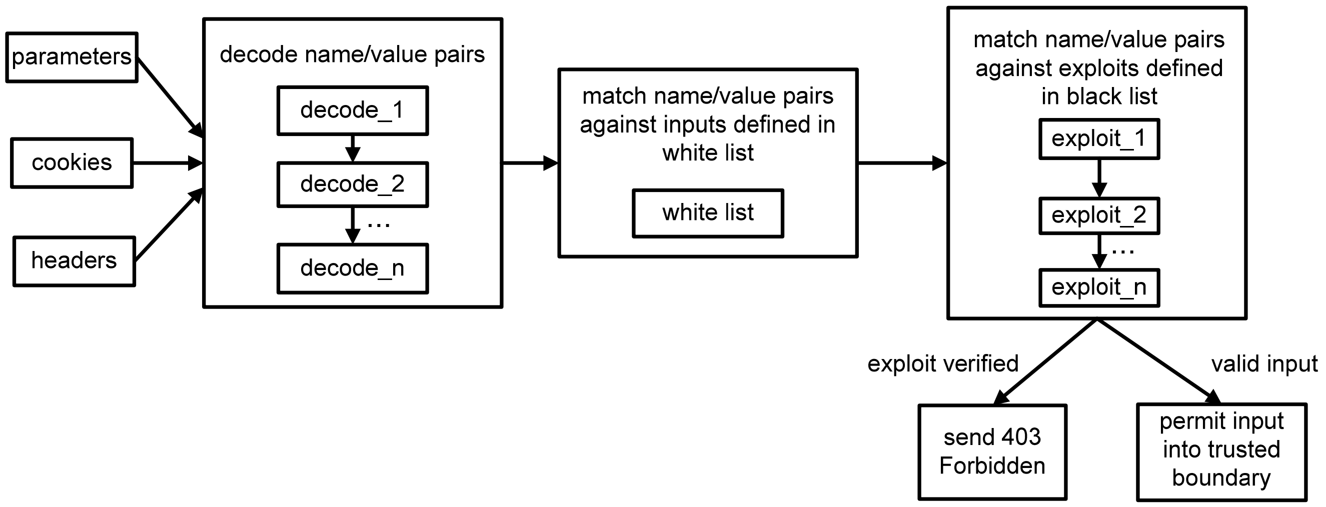


Figure 7: HTTP input flows through decoders, and is then checked against a white and black list before permitted or rejected by the Web application.

Encoded data must be decoded before matched against a white list or black list. SecureFilter provides an API for developers to decode their data. For example, if the value of a cookie is encoded with Base64, then the developer can set the decoders to decode from Base64 before the data are filtered. Developers can create a regular expression-based white list based on their requirements specifications and place them in a configuration file read by SecureFilter. If data do not match against the white list, then a 403 SC_FORBIDDEN status code is immediately returned and input is not allowed to enter the trust boundary. The default white list provided with WARD v1.0 is a grammar that defines all possible input (see Figure 8) and is intended to be a place holder for a developer's white list. As mentioned in Section 1, the requirements of every software product are likely to be unique and so we do not attempt to guess how their valid input is defined.

```
<inputSecurityDefinition inputType="header"
     namePattern="(?s)^.*$" validInputPattern="(?s)^.*$"/>
<inputSecurityDefinition inputType="cookie"
     namePattern="(?s)^.*$" validInputPattern="(?s)^.*$"/>
<inputSecurityDefinition inputType="parameter"
     namePattern="(?s)^.*$" validInputPattern="(?s)^.*$"/>
```

Figure 8: The default white list for SecureFilter provides a placeholder for developers to create their own white list.

Regular expressions are a powerful way to match against strings. Regular expressions can become intricate when describing complex languages. Developers are likely to have mistakes in the regular expressions for their white lists due to this complexity; therefore, input is subsequently checked by a black list to reinforce the robustness of the white list. We considered expressing grammars using Bison[22] or Antlr[23] but chose regular expressions to keep WARD use as simple as possible for developers

[22] http://dinosaur.compilertools.net/bison/bison_6.html
[23] http://www.antlr.org/grammar/list

Below are four testing scenarios that may occur during development with WARD.

Test Scenario 1: SecureUnit sends valid data to the target application. The white list matches the data as valid. The data are passed to the black list where an exploit is matched. A 403 SC_FORBIDDEN page is returned to SecureUnit and the test fails. WARD v2.0 will have the implementation that provides developers to allow input matched by their black list into their system.

Test Scenario 2: SecureUnit sends a malicious request to the target application. The data do not match against the white list. The data are passed to the black list where an exploit is matched. A 403 SC_FORBIDDEN page is returned to SecureUnit and the test fails.

Test Scenario 3: SecureUnit sends an innocuous request to the target application. The data match against the white list and do not match any exploits in the black list. The data are permitted to proceed into the trust boundary.

Test Scenario 4: SecureUnit sends an innocuous request to the target application. The data do not match against the white list. A 403 SC_FORBIDDEN page is returned to SecureUnit and the test fails.

Funneling all input through one choke point reduces the "surface area" of a trusted system with the objects it interacts with. Choke points can serve as a strategic location to mediate all input into a trusted boundary. The concept of a mechanism to decide if data are permitted to enter is known as a reference monitor, and the implementation of the deciding mechanism is called a reference validation mechanism (RVM) [4]. SecureFilter is an example of an RVM. A RVM must be tamperproof, must always be invoked (and can never be bypassed), and must be small enough to be subject to analysis and testing, the completeness of which can be assured [4]. Attackers will generally not have access to J2EE filters unless improper access permissions are assigned, making the filter tamperproof. The servlet container can be configured according to the J2EE specification to guarantee that all HTTP input is filtered. SecureFilter is also testable with SecureUnit, but the domain of web application exploitation is not finite therefore completeness cannot be assured.

3.3 Adhering to the Fundamental Principles of Software Security

The design of SecureFilter is based on 11 of the 12 fundamental principles of software security that originate from Jerome Saltzer and Michael Schroeder's [17] design principles published in 1975 to provide for a strategic approach of stopping attacks. Following these principles do not guarantee perfect security from malicious input attacks, but do potentially provide for a more effective solution than one without a guiding philosophy.

Principle of Promoting Privacy – attacks can threaten the privacy of a user and if stopped can mitigate the potential of revealing their information.

Principle of Never Assuming that your Secrets are Safe – fortifying vulnerabilities can prevent attackers from obtaining digital assets such as passwords and session id's.

Principle of Defense in Depth – developers can use their own validation mechanism to validate the white lists/black lists.

Principle of Economy of Mechanism – the use of a choke point simplifies input to the system for filtering.

Principle of Failing Securely – when the system is attacked an error page is returned and the sensitive data are not returned to the client.

Principle of Psychological Acceptability – SecureFilter is not invasive to the developer's code. Developers need only modify their web.xml file to benefit from SecureFilter.

Principle of Reluctance to Trust – all input is assumed malicious and is filtered at the choke point.

Principle of Securing the Weakest Link – one of the largest problems with software systems today is that input is trusted [9, 20]. SecureFilter filters all HTTP input for potentially injected exploits.

Principle of Least Privilege – developers have the opportunity to check HTTP requests with their custom white list and our black list.

Principle of Complete Mediation – "all input is evil"[9] and thus all input of each web request is checked to for permission to enter the system.

Principle of Separation of Privilege – During development, a developer's white list can be checked for correctness using SecureUnit's black list as a test. Testing data through both lists represents two conditions that data are checked against.

In WARD, SecureFilter funnels all input through a single choke point, which adheres to the economy of mechanism. But, the simplicity of the choke point comes at the sacrifice of the principle of least common mechanism because all users share the same choke point to access the web application.

4. ILLUSTRATIVE EXAMPLE

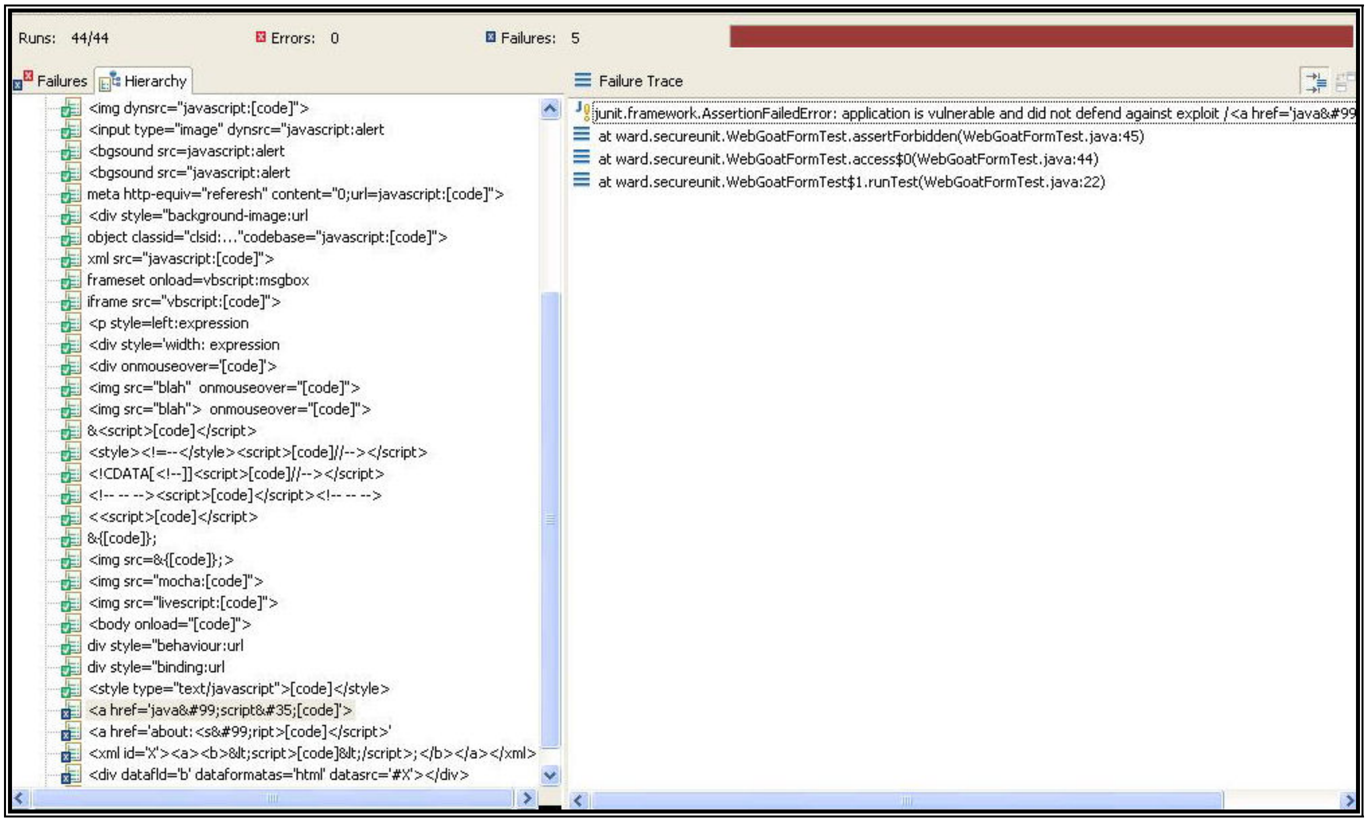
To test Ward, we required a test bed that could serve as a realistic web application. We chose Stanford SecuriBench .91a, an open source suite of software applications that are designed for running static and dynamic tests. These test beds are vulnerable to XSS, SQL injection, HTTP splitting and path traversal attacks. Specifically, we demonstrate the use of SecureUnit on WebGoat 3.7, a J2EE application implemented in part to demonstrate XSS attacks. WebGoat is an educational-based program in that there are security lessons in the program that teach one how to attack the system. We required only the binary release of WebGoat because the WARD framework is not invasive to the source code.

We configured WARD to use the default white list (see Section 3.2) to allow all data to enter WebGoat. A "wide open" or faulty white list demonstrates that WARD can still alert developers that a vulnerability may be present. Furthermore, having not implemented WebGoat nor having the requirements specifications, we could not determine what inputs are valid. We created a black list (see Figure 9) containing 12 regular expressions based on the 43 explicitly documented XSS exploits in [9] as shown below. We assumed that each exploit would be malicious to WebGoat if it was a live application released into the field. The EL used by SecureUnit contained each of the 43 exploits.

```
<exploit name="exploit1" pattern="(?si).*javascript:.*"/>
<exploit name="exploit2" pattern="(?si).*vbscript:.*"/>
<exploit name="exploit3" pattern="(?si).*expression.*"/>
<exploit name="exploit4" pattern="(?si).*onmouseover.*"/>
<exploit name="exploit5" pattern="(?si).*&lt;script&gt;.*&lt;/script&gt;.*"/>
<exploit name="exploit6" pattern="(?si).*\&amp;\{.*\}.*"/>
<exploit name="exploit7" pattern="(?si).*mocha:.*"/>
<exploit name="exploit8" pattern="(?si).*livescript:.*"/>
<exploit name="exploit9" pattern="(?si).*onload=.*"/>
<exploit name="exploit10" pattern="(?si).*behaviour:url.*"/>
<exploit name="exploit11" pattern="(?si).*binding:url.*"/>
<exploit name="exploit12" pattern="(?si).*&lt;style.*&gt;.*&lt;/style&gt;.*"/>
```

Figure 9: The default black list used by SecureFilter. This list also provides a source for developers to create their EL with SecureUnit.

We wrote a HttpUnit test fixture to log into WebGoat (using basic authentication), browse to a Web page that contained a vulnerable HTML form which used a POST to submit data to a servlet container (Tomcat 5.5). Using SecureUnit, we injected 43 XSS exploits into the form and tested the response from the server (see Figure 10). Results indicate that 38 of 43 (88%) XSS vulnerabilities were "warded off" when matched against the black list. The remaining five XSS exploits in the EL are encoded (e.g. with HTML entities, hex characters) and thus pass through regular expressions in SecureFilter. In the next installation of WARD, exploits will be decoded with built-in decoders before input is filtered. We predict that once input is decoded, WARD will prevent 100% of the 43 exploits in the EL from entering into WebGoat.



```
Runs: 44/44    Errors: 0    Failures: 5
```

```
<img dynsrc="javascript:[code]">
<input type="image" dynsrc="javascript:alert
<bgsound src=javascript:alert
<bgsound src="javascript:alert
meta http-equiv="referesh" content="0;url=javascript:[code]">
<div style="background-image:url
object classid="clsid:…"codebase="javascript:[code]">
xml src="javascript:[code]">
frameset onload=vbscript:msgbox
iframe src="vbscript:[code]">
<p style=left:expression
<div style='width: expression
<div onmouseover='[code]'>
<img src="blah"  onmouseover="[code]">
<img src="blah">  onmouseover="[code]">
&<script>[code]</script>
<style><!=--</style><script>[code]//--></script>
<!CDATA[<!--]]<script>[code]//--></script>
<!-- -- --><script>[code]</script><!-- -- -->
<<script>[code]</script>
&{[code]};
<img src=&{[code]};>
<img src="mocha:[code]">
<img src="livescript:[code]">
<body onload="[code]">
div style="behaviour:url
div style="binding:url
<style type="text/javascript">[code]</style>
<a href='java&#99;script&#35;[code]'>
<a href='about:<s&#99;ript>[code]</script>'
<xml id='X'><a><b>&lt;script>[code]&lt;/script>;</b></a></xml>
<div datafld='b' dataformatas='html' datasrc='#X'></div>
```

```
junit.framework.AssertionFailedError: application is vulnerable and did not defend against exploit /<a href='java&#99
at ward.secureunit.WebGoatFormTest.assertForbidden(WebGoatFormTest.java:45)
at ward.secureunit.WebGoatFormTest.access$0(WebGoatFormTest.java:44)
at ward.secureunit.WebGoatFormTest$1.runTest(WebGoatFormTest.java:22)
```

Figure 10: A screen shot of SecureFilter after performing 43 penetration tests against WebGoat.

Thorough testing for XSS vulnerabilities should include penetrating a web application for all possible entry points of an XSS attack. In Ward 1.0, we also created a test that injects XSS exploits in the headers, nonexistent form fields, and cookies of a Web request even though the Web application is not expecting such input. Attackers play by their own rules and following the rules of the web interface is not one of them. Bypassing the web interface adheres to the attack pattern, "Make the Client Invisible"[8], that describes situations where attackers make their own client to submit requests to a server. Tools such as netcat[24] provide the utilities an attacker needs to perform such attacks. Also, some web pages can accept GET and POST requests without a user entering form data so it is not sufficient to simply inject exploits into the interface that a vendor supplies.

```
request.setParameter("nonExistentField", "<a href='javascript:alert('sploit')'>Ha gotcha</a>");
request.setHeaderField("headerExploit", "<script>alert('headerExploit');</script>");
conversation.putCookie("cookieExploit", "<script>alert('cookieExploit');</script>");
```

Figure 11: A snippet from a SecureUnit test that injects exploits into an HTTP request without using a web interface.

The results of injecting XSS exploits into a random parameter, header, and cookie are identical to our earlier test where we submitted XSS exploits into an existing HTML form field; 38 of 43 (88%) of the XSS exploits were prevented. We believe that an application should "just say no" to XSS exploits even if they are never allowed to execute on a user's system. Therefore, in v1.0, all malicious inputs (that are not encoded) are prevented from entering the Web application.

5. LIMITATIONS

Ward is based on regular expressions that match known exploits. If an attacker submits a new exploit, then the developer's white list and black list may not prevent the exploit from entering the system. If the software is vulnerable to the exploit, then developers must refactor their white lists and black lists and update their tests to accommodate the new exploit. Also, SecureFilter represents a single point of failure. If an attacker can disable

[24] http://netcat.sourceforge.net/

SecureFilter and a developer exclusively relied on SecureFilter to defend their application, then the software is subject to the attack. Furthermore, SecureFilter is a choke point that only filters HTTP traffic over a designated port (e.g. 8080). An attacker can route an exploit to another port via another application to attack the software. For example, an attacker can submit an XSS exploit to a system via a database connection using another port that is not mediated by SecureFilter.

According to the Seven Pernicious Kingdoms taxonomy there exists 26 different types of input validation and representation problems in software security[19]. Some of these malicious input attacks can be defended against via input filtering while others have better defenses. For example, if an attacker can submit a privileged filename to a software system, then they may be able to read that file if the system is executing with the proper access rights. Instead of filtering input for privileged filenames, the system should prevent access to the files [8]. In the case of SQL injection attacks, filtering for special SQL characters may not be adequate to stop an attack. Implementing prepared statements that bind variables is a better solution [1, 10] to SQL injection. Securing a software system by filtering for exploits in black lists is "fundamentally flawed" approach to software security [8]. Developers should use white lists to validate input for vulnerabilities where input filtering is the primary defense. XSS is an instance where input validation is the primary defense. Disabling JavaScript functionality in a Web browser is a good defense for a client, but a developer for a Web application cannot enforce that upon the client. Furthermore, the developer's Web application may require JavaScript for content to display correctly.

The black list we used in our demonstration with WebGoat is not an exhaustive list of known XSS exploits. To thoroughly test the strength of WARD and its ability to mitigate/secure input validation vulnerabilities, we would need to mine vulnerability databases and other Web resources that contain a larger list of XSS exploits. A more thorough list of exploits should be added to our default EL.

6. FUTURE WORK

Our vision for SecureUnit is to have a static analyzer detect vulnerabilities in a software system and dynamically generate penetration tests to run against the target application. The dynamic test would verifies that the static analyzer did not produce a false positive by providing a test case that demonstrates the vulnerability and educates the developer on how such an attack may occur [6]. Developers may also choose to use their own testing tools or extend an existing tool (e.g. Nessus) and test against the 403 status code. The advantage with SecureUnit is that it provides a framework for developers to test their code in an attack-then-defend paradigm to allow security to be built into the software system early in the development process.

SecureFilter can be operated in one of three modes: error, pass through, and sanitize. In WARD v1.0, we have implemented error mode, which is responsible for sending a 403 Forbidden page back to the client if an XSS exploit is found in a request. In the next version of WARD, we will implement pass through mode where developers can allow XSS exploits to enter the system; it will be the responsibility of the developer to handle the exploit safely. In sanitize mode, SecureFilter sanitizes XSS exploits to allow XSS exploits by encoding the special characters of XSS exploits (e.g. <, >, &, :) into their HTML entities. Also, we plan to finish implementing the decoders that will enable us to decode malicious input before it is filtered. Future installations of SecureFilter will include checks for exploits in database queries and hidden input fields.

We also plan to validate WARD against the weaknesses described in the Common Weakness Enumeration[25] (CWE), which will provide a more formal description of which XSS weaknesses WARD can defend against. The CWE is an aggregation of sources including Seven Pernicious Kingdoms, CLASP, PLOVER, ten from OWASP, the Web Security Threat Classification, 19 Deadly Sins, etc. that describes software weaknesses (to date ~500 of them) in a consistently named fashion and provides a taxonomy to describe the relationships between the weaknesses. Approximately 15-20 tool vendors, including Fortify Software[26] and Secure Software[27], contribute and map their content to the CWE. In conforming to the CWE, we can assess the efficacy of WARD with other tools to determine its advantages and disadvantages in a common way.

[25] http://cwe.mitre.org/
[26] http://www.fortifysoftware.com/
[27] http://www.securesoftware.com/

7. SUMMARY AND CONCLUSION

The goal of our research is to produce a simple and effective framework for developers to secure their software from input validation vulnerabilities in an attack-then-defend software security approach. We hypothesize that if developers adopt a security-oriented TDD process, then they will have a more effective and less costly means to practice due diligence. The WARD framework affords developers with the ability to attack their systems to identify malicious input vulnerabilities with SecureUnit. Testing for security vulnerabilities provides an objective means to produce quantifiable results about how a system can be attacked. The complementary component to the attack-then-defend approach is SecureFilter, which provides a framework for developers to create and use white lists and black lists. The filtering functionality is provided by our J2EE implementation of the `Filter` interface. Developers utilize the WARD framework by constructing SecureUnit tests that attempt to infiltrate their filters before exploiting Web application. We have demonstrated the effectiveness of WARD on WebGoat, an open-source Web application test bed. The results are positive and indicate that we should continue implementing WARD to provide the capability to decode encoded data and allow developers to selectively allow input into their system that may match against their black list.

Acknowledgements

We would like to thank the North Carolina State University Software Engineering Realsearch reading group for their helpful suggestions on this paper. This research was supported by the National Science Foundation. Special thanks to Lucas Layman for providing valuable criticisms in the paper.

8. REFERENCES

[1] Anley, C., "Advanced SQL Injection In SQL Server Applications," Next Generation Security Software Ltd, 2002.
[2] Barnum, S. and M. Gegick, "Design Principles," 2005.
[3] Beck, K., *Test-Driven Development -- by Example*. Boston: Addison Wesley, 2003.
[4] Bishop, M., *Computer Security: Art and Science*. Boston: Addison-Wesley, 2003.
[5] Boehm, B., "Industrial Metrics Top 10 List," *IEEE Software*, vol. 4, pp. 84-85, 1987.
[6] Csallner, C. and Y. Smaragdakis, "Check 'n' Crash: Combining static checking and testing," presented at In Proceedings of the 27th international conference on Software Engineering, 2005.
[7] Dijkstra, E., "Structured Programming," in *Software Engineering Techniques*, J. N. Buxton and B. Randall, Eds. Brussels, Belgium, 1970, pp. 84/88.
[8] Hoglund, G. and G. McGraw, *Exploiting Software*. Boston: Addison-Wesley, 2004.
[9] Howard, M. and D. LeBlanc, *Writing Secure Code*, 2nd ed. Redmond: Microsoft Corporation, 2003.
[10] Howard, M., D. LeBlanc, and J. Viega, *19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. Emeryville: McGraw-Hill/Osborne, 2005.
[11] Huang, Y., S. Huang, T. Lin, and C. Tsai, "Web Application Security Assessment by Fault Injection and Behavior Monitoring," presented at World Wide Web Conference 2003, Budapest, Hungary, 2003.
[12] IEEE, "ANSI/IEEE Standard Glossary of Software Engineering Terminology," 1990.
[13] McGraw, G., *Software Security: Building Security In*. Boston: Addison-Wesley, 2006.
[14] Miller, B. P., D. Koski, R. Murthy, C. Lee, A. Natarajan, V. Maganty, and J. Steidl, "Fuzz Revisited: A Re-Examination of the Reliability of Unix Utilities and Services, tech. report CS-TR-95-1268.," Department of Computer Science, Univ. Wisconsin, Madison April 1995.
[15] NIST, "Engineering Principles for Information Technology Security," in *Special Publication 800-27*, 2001.
[16] Pressman, R., *Software Engineering: A Practitioner's Approach*, 5th ed. New York: McGraw-Hill, 2001.
[17] Saltzer, J. and M. Schroeder, "The Protection of Information in Computer Systems," presented at IEEE, 1975.
[18] Schneier, B., "The Process of Security," in *Information Security Magazine*, 2000.

[19] Tsipenyui, K., B. Chess, and G. McGraw, "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors," presented at Automated Software Engineering, Long Beach, CA, 2005.
[20] Viega, J. and G. McGraw, *Building Secure Software How to Avoid Security Problems the Right Way*. Boston: Addison-Wesley, 2002.