Non-Detrimental Web Application Security Scanning*

Yao-Wen Huang[1,2], Chung-Hung Tsai[1], D. T. Lee[2,3], Sy-Yen Kuo[1]
{ywhuang, chtsai}@openwaves.net; dtlee@iis.sinica.edu.tw; sykuo@cc.ee.ntu.edu.tw

[1]*Department of Electrical Engineering, National Taiwan University, Taipei 106, Taiwan.*

[2]*Institute of Information Science, Academia Sinica, Taipei 115, Taiwan*

[3]*Department of Computer Science and Information Engineering, National Taiwan University, Taipei 106, Taiwan*

Abstract

The World Wide Web has become a sophisticated platform capable of delivering a broad range of applications. However, its rapid growth has resulted in numerous security problems that current technologies cannot address. Researchers from both academic and private sector are devoting a considerable amount of resources to the development of Web application security scanners (i.e., automated software testing platforms for Web application security auditing) with some success. However, little is known about their potential side effects. It is possible for an auditing process to induce permanent changes in an application's state. Due to this potential, we have so far avoided large-scale empirical evaluations of our Web Application Vulnerability and Error Scanner (WAVES). In this paper we introduce a testing methodology that allows for harmless auditing, define three testing modes—heavy, relaxed, and safe modes, and report our results from two experiments. In the first, we compared the coverage and side effects of the three scanning modes using 5 real-world Web applications chosen from the 38 found vulnerable in a previous static verification effort. In the second, we used the relaxed mode to conduct a 48-hour test involving 1120 random websites, of which 55 were found to be vulnerable.

1. Introduction

On Feb 2, 2000, CERT issued an advisory [8] on "cross-site scripting" (XSS) attacks on Web applications. This hard-to-eliminate threat soon drew the attention and spawned active discussions among security researchers [32]. Despite the efforts of researchers in the private sector and academia to promote developer awareness and to develop tools to eliminate XSS attacks, hackers are still using them to exploit Web applications. A study by Ohmaki (2002) [33] found that almost 80 percent of all e-commerce sites in Japan were still vulnerable to XSS. A search on Google News (http://news.google.com) for XSS advisories on newly discovered XSS vulnerabilities within the month of March 2004 alone yielded 24 reports, among these were confirmed vulnerabilities in Microsoft Hotmail [54] and Yahoo! Mail [21], both of which are popular web-based email services.

As World Wide Web usage expands in such areas as B2B (business-to-business), B2C (business-to-client), P2P (peer-to-peer), social, healthcare, and e-government services, Web applications reliability and security are becoming increasingly important concerns. In a Symantec analysis of network-based attacks, known vulnerabilities, and malicious code recorded throughout 2003, eight of the top ten attacks were associated with Web applications; the report also stated that port 80 was the most frequently attacked TCP port. In addition to holding Web applications responsible for the sharp increase in moderately severe vulnerabilities found in 2003, the report's authors also suggested that Web application vulnerabilities were by far the easiest to exploit.

Researchers have proposed a broad range of defense strategies against XSS attacks. Park and Sandhu's [36] cookie-securing mechanism can be adopted to eliminate XSS, but it requires explicit modifications to existing Web applications. Scott and Sharp [50] have proposed using gateways for filtering malicious input at the application level. In addition to preventing XSS, the gateway also prevents SQL injection—another widespread Web application vulnerability. Sanctum's *AppShield* [45], Kavado's *InterDo* [20], and a number of other commercial products are based on similar strategies. Most of the leading firewalls now also incorporate deep packet inspection technologies [12] for filtering application-level traffic. A primary advantage of these types of deployment-phase mechanisms is their ability to provide immediate assurance; their main drawback is that they blindly protect against unpredicted behavior without investigating the actual defects that compromise quality. A second drawback is their need for careful configuration by experienced administrators [6].

As part of our *Web application Security by Static Analysis and Runtime Inspection (WebSSARI)* project [16]

* *This research was supported in part by the National Science Council, Taiwan under grants NSC-93-2213-E-001-013, NSC-93-2422-H-001-0001, NSC-93-2752-E-002-005-PAE, NSC 93-2219-E-010.*

[17], we applied static analysis to real-world Web application code and found numerous popular open source projects vulnerable to XSS and SQL injection vulnerabilities. However, static analysis requires source code, which in many situations is not available. Organizations conducting risk analysis would benefit from a remote, black-box auditing tool that produces vulnerable site identification reports. Our contribution to this goal is a security assessment framework we have named the *Web Application Vulnerability and Error Scanner*, or *WAVES*.

In [15], we acknowledged two serious deficiencies in our original WAVES design. First, its capabilities were restricted to detecting XSS scripts that had already been injected into Web applications. It was incapable of assessing whether a Web application is vulnerable to XSS. Second, its auditing process had a potential side effect of causing permanent modifications (or even damage) to the state of the targeted application. This potential prevented us from performing large-scale empirical evaluations of WAVES. It should be noted that *AppScan* [46], *InterDo* [19], *WebInspect* [51], and similar commercial and open-source projects have the same drawbacks. These drawbacks are the focus of this paper.

This paper is organized as follows: after defining XSS vulnerability as a secure information flow problem, we describe a formal model for Web application entry points and discuss HTTP interactions in terms of program function calls. We also describe how WAVES performs a reverse engineering process to extract all function call (entry point) definitions of a target application, then uses the semantics of each function to determine which ones can be tested for insecure information flow without inducing side effects. Next, we describe how information flow testing is performed in order to identify XSS vulnerabilities, and then report our results from two experiments. In experiment 1, we installed and scanned five Web applications that were found vulnerable by WebSSARI, and a) compared testing coverage with WebSSARI (static verification), and b) studied the results of our proposed non-detrimental scanning techniques and their impact on testing coverage. In experiment 2, we conducted a 48-hour testing involving 1120 random websites.

2. The Cross-Site Scripting Vulnerability

In this section we will provide a brief description of XSS and SQL injection vulnerabilities; readers are referred to Scott and Sharp [50], Curphey et al. [10], and Meier et al. [26] for more details.

2.1. Cross-Site Scripting

Cross-site scripting (XSS) is perhaps the most common Web application vulnerability. Figure 1 gives an example of an XSS bug we identified (using WebSSARI [16]) in *SquirrelMail*, a popular Web-based e-mail application.

```
$month=$_GET['month']; $year=$_GET['year'];
$day=$_GET['day'];
echo "<a href=\"day.php?year=$year&amp;"
echo "month=$month&amp;day=$day\">";
```

Figure 1. An XSS vulnerability found in *SquirrelMail*.

Values for the variables $month, $day, and $year come from HTTP requests and are used to construct HTML output sent to the user. An example of an attacking URL would be:

```
http://www.target.com/event_delete.php?year=><script>m
alicious_script();</script>
```

Attackers must find ways to make victims open this URL. One strategy is to send an e-mail containing javascript that secretly launches a hidden browser window to open this URL (see Steps 1 and 2 in Figure 6). Another is to embed the same javascript inside a Web page; when victims open the page, the script executes and secretly opens the URL. Once the PHP code shown in Figure 1 receives an HTTP request for the URL, it generates the compromised HTML output shown in Figure 2.

```
<a href= "day.php?year=><script>malicious_script();</script>
```

Figure 2. Compromised HTML output.

In this strategy, the compromised output contains malicious script prepared by an attacker and delivered on behalf of a Web server (Step 3 in Figure 6). HTML output integrity is hence broken and the Javascript Same Origin Policy [27] [31] is violated. Since the malicious script is delivered on behalf of the Web server, it is granted the same trust level as the Web server, which at a minimum allows the script to read user cookies set by that server (Steps 4 in Figure 5). This often reveals passwords or allows for session hijacking (Step 5 and 6 in Figure 6); if the Web server is registered in the Trusted Domain of the victim's browser, other rights (e.g., local file system access) may be granted as well.

Another severe type of XSS involves the uploading of data by a user, which is then stored for later delivery by a Web application without performing any type of sanitization. Consider the following example: a commercial online auction site hosts a ticket service for users to get support, report bugs, and submit feature requests. Messages posted by users are submitted to a server-side script that inserts them into a backend database. Support tickets and bug reports can only be read by the website's support personnel, but feature requests

can be read by all users. When viewing tickets, a request is sent to a server-side script that retrieves corresponding data from the backend database and constructs HTML output. If a user submits a bug report (or a feature request) that contains a piece of malicious script, the script will be delivered to the support personnel (or other users) on behalf of the Web server. This grants rights that the script normally would not receive. Figure 3 presents a simplified version of a vulnerability that our WebSSARI [16] discovered in *PHP Support Tickets*.

```
$query="INSERT INTO tickets_tickets(tickets_id, ".
"tickets_username,tickets_subject, tickets_question) ".
"VALUES('$_SESSION['username']', '$_POST['ticketsubject'],'".
"'$_POST['message']')";
$result = @mysql_query($query);
```

Figure 3. An XSS vulnerability found in *PHP Support Tickets* code for ticket submission.

Note that user input values "ticketsubject" and "message" have been inserted into the database without sanitization. An example of code from *PHP Support Tickets* that uses the backend database to generate HTML output for displaying tickets is shown in Figure 4. Since the value "tickets_subject" (containing untrusted data submitted by the user) is used without sanitization to construct HTML output, the code is vulnerable to XSS.

```
$query="SELECT tickets_id, tickets_username,
tickets_subject FROM tickets_tickets";
$result = @mysql_query($query);
WHILE ($row = @mysql_fetch_array($result)) {
  extract($row);
  echo"$tickets_username<BR>$tickets_subject<BR><BR>"
}
```

Figure 4. Simplified code for displaying the tickets.

2.2. SQL Injection

Considered more severe than XSS, SQL injection vulnerabilities occur when untrusted values are used to construct SQL commands, resulting in the execution of arbitrary SQL commands given by an attacker. The example below is based on a vulnerability we discovered in *ILIAS Open Source*, a popular Web-based learning management system.

```
$sql="INSERT INTO track_temp VALUES('$HTTP_REFERER');"
mysql_query($sql);
```

Figure 5. A simplified SQL injection vulnerability found in *ILIAS Open Source*.

In Figure 5, $HTTP_REFERER (a global variable set by the Web server to indicate the referrer field of an HTTP request) is used to construct an SQL command. The referrer field of an HTTP request is an untrusted value given by the HTTP client; an attacker can set the field to:

```
');DROP TABLE ('users
```

This will cause the code in Figure 5 to construct the $sql variable as:

```
INSERT INTO track_temp VALUES('');
DROP TABLE ('users');
```

Table "users" will be dropped when this SQL command is executed. This technique, which allows for the arbitrary manipulation of backend database, is more serious than XSS and often results in theft of critical information such as credit card numbers.

3. The Testing Model

The primary objectives of information security systems are to protect confidentiality, integrity, and availability [47]. From our examples, it is obvious that for Web applications, compromises in integrity are the main causes of compromises in confidentiality and availability. The relationship is illustrated in Figure 6. When untrusted data is used to construct trusted output without sanitization, violations in data integrity occur, leading to escalations in access rights that result in availability and confidentiality compromises.

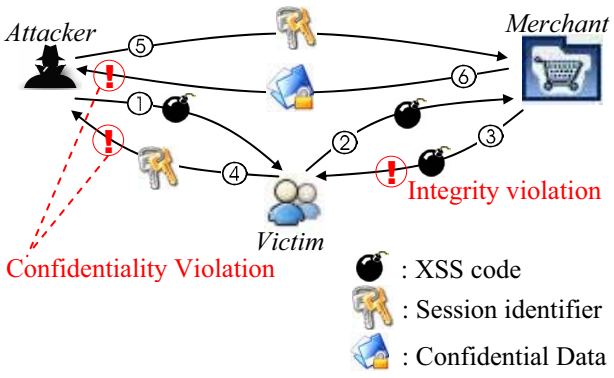

Figure 6. Web application vulnerabilities result from insecure information flow, as illustrated using XSS.

A mechanism is required in order to test for illegal information flow—specifically, to identify violations of Web application *noninterference* [14] policies. We made the following assumptions when designing our approach to detecting XSS and SQL injection vulnerabilities:

Assumption 1: All data sent by Web clients in the form of HTTP requests should be considered untrustworthy.
Assumption 2: All data local to a Web application are secure.
Assumption 3: Tainted data can be made secure (against known attacks) with appropriate processing.

In addition, the following policies were defined:

Policy 1: Tainted data must not be used in HTTP response construction.
Policy 2: Tainted data must not be written into local Web application storage.
Policy 3: Tainted data must not be used in system command construction.

Assumption 1 says that all data sent by Web clients (in the form of HTTP requests) should be considered untrustworthy. A majority of Web application security flaws result when this assumption is ignored. The Web uses a sessionless protocol in which each URL retrieval is considered an independent TCP session, established when the HTTP request is sent and terminated after a response is retrieved. Many transaction types (e.g., those that support user logins) clearly require session support. In order to keep track of sessions, Web applications require a client to include a session identifier within an HTTP request. An HTTP request consists of three major parts—the requested URL, form variables (parameters), and cookies. In practice, all three are used in different ways to store session information. Cookies are the most frequently used, followed by hidden form variables and URL requests

To manage sessions, Web applications are written so that browsers include all session information following initial requests that mark the start of a session; processing HTTP requests entails the retrieval of that information. Even though such information is transferred to the client by the Web application, it should not be considered trustworthy information when it is read back from an HTTP request. The reason is that such information is usually stored without any form of integrity protection (e.g., digital signatures), and is therefore subject to tampering. Using such information to construct HTML output without prior sanitization is considered a Policy 1 violation—the most frequent cause of XSS.

Assumption 2 states that all data local to a Web application should be considered secure. This includes all files read from the file system and data retrieved from the database. According to this assumption, all locally retrieved data are considered trusted, which results in Policy 2, which states that system integrity is considered broken whenever untrustworthy data is written to local storage. Since most applications use client-supplied data to construct output, our model would be too strict without Assumption 3, which states that untrustworthy data can be made trustworthy (e.g., malicious content can be sanitized and problematic characters can be escaped).

XSS vulnerabilities result in Policy 1 or Policy 2 violations. Script injection vulnerabilities such as SQL injection are generally associated with Policy 3 violations. To perform tests on Web applications for the purpose of detecting policy violations, a testing model must be established.

3.1. Modeling Web Application Data Flow

Program *functions* and/or *procedures* serve as the basic testing unit in many software testing processes. Pieces of software are often viewed as collections of interacting functions. In C, every function has a prototype definition that specifies its syntax, including its name, return type, and arguments and their types. Type qualifiers can be further added to define function behavior, (e.g., defining calling conventions). Comments within code or supplementary documentation are used to describe function semantics—what it does and how it can be used. Finally, there is the function's source code implementation.

More available information on a program's functions results in the possibility to develop more thorough software testing processes for that program. Before clarifying the kind of information to which a *Web application security scanner (WSS)* has access, it is necessary to define what a WSS is. Here we will define WSS as a testing platform posed as an outsider (i.e., as a public user) to the target application. Therefore, WSSs operate according to three constraints:
1. Neither documentation nor source code will be available for the target Web application.
2. Interactions with the target Web applications and observations of their behaviors will be done through their public interfaces, since system-level execution monitoring (e.g., software wrapping, process monitoring, and local files access) is not possible.
3. The testing process must be automated and testing a new target system should not require extensive human participation in test case generation.

In [15] we described our approach to detecting Policy 3 violations, but we were unable to detect Policy 1 and 2 violations. Here we describe an approach for testing against Policy 1 violations—the major cause of XSS. In white-box testing, source code analysis provides critical information for effective test case generation [38]. In black-box testing (in other words, when source code is unavailable), an information-gathering approach is to reverse-engineer executable code. We took a similar approach to identifying server-side scripts (scripts that read user input and generate output) within Web applications. These scripts constitute a Web application's *data entry points (DEPs)*. Web application interfaces that reveal DEP information include HTML forms and URLs within HTML that point to server-side scripts. In short, WAVES uses a crawler to crawl Web applications in order to gather DEP definitions—an approach described in many studies involving Web site analysis (VeriWeb [4], Ricca and Tonella [39] [40]) and reverse engineering

(Ricca et al. [41] [42] [43]). HTML contents such as anchors and forms *reveal* DEP definitions, and DEP definitions of a site can be enumerated by exhaustively collecting all HTML content. See Section 4 for a list of all ways that HTML pages reveal the existence of DEPs.

If each DEP is defined as a program function, then each revelation is the equivalent of a function call site. We define each revelation R of a DEP as a tuple: $R = \{URL, T, Sa\}$, where *URL* stands for the DEP's URL, T the type of the DEP, and $Sa = \{A_1, A_2, \ldots, A_n\}$ a set of arguments (parameters) accepted by the DEP. The type of a DEP specifies its functionality; the possible types include searching (tS), authentication (tA), account registration (tR), message posting (tM), and unknown (tU). By combining information on a DEP's URL with the names of its associated HTML forms, the names of its parameters, the names of form entities associated with those parameters, and the adjacent HTML text, WAVES can make a determination of DEP type [15]. Note that form variables are not the only sources of a DEP's input—cookies are also sources of readable input values. Therefore, the set of R's arguments $Sa = S_R \cup S_C$, where $S_R = \{P_1, P_2, \ldots, P_n\}$ is the set of parameters revealed by R, and $S_C = \{C_1, C_2, \ldots, C_n\}$ is the set of cookies contained within the page containing R.

Just as there can be multiple call sites to a program function, there may be multiple revelations of a DEP. In Google, both simple and advanced search forms are submitted to the same server-side script, with the latter submitting more parameters. We defined a DEP D as $\{dURL, dT, dSa\}$. For a set $S_D = \{R_1, R_2, \ldots, R_n\}$ of all collected revelations of the same DEP D, $dURL = R_1.URL = R_2.URL = \ldots = R_n.URL$. D's type dT = Judge_T($R_1.T$, $R_2.T$, …, $R_n.T$), where Judge_T is a judgment function [15] that determines a DEP's type, taking into account the types of all its revelations. D's arguments $dSa = R_1.Sa \cup R_2.Sa \cup \ldots \cup R_n.Sa$.

3.2. Test Case Generation

Given such a definition, a DEP can be viewed as a program function, with *dURL* being the function name, *dT* the function type, and *dSa* its arguments. The function output is the generated HTTP response (i.e., HTTP header, cookies, and HTML text). In this respect, testing a DEP is the same as testing a function—test cases are generated according to the function's definitions, functions are called using the test cases, and outputs are collected and analyzed.

Testing for Policy 1 violations involved using our DEP definition to generate test cases containing attack patterns, submitting them to the DEP, and studying the output for signs of the attack pattern. The appearance of an attack pattern in DEP output means that the DEP is using tainted (non-sanitized) data to construct output (details of this testing methodology are described in [15]). The two questions guiding our test case generation were a) What is an appropriate test case size that allows for thorough testing within an acceptable amount of time? and b) What types of test cases will/will not cause side effects?

In response to the first question, given a DEP D of $dSa = \{A_1, A_2, \ldots, A_n\}$, a naïve approach would be to generate n test cases, each with a malicious pattern placed in a different argument. For each test case, arguments other than the one containing malicious data would be given arbitrary values. This appears to be a reasonable approach on the surface, but it is subject to a high rate of false negatives because DEPs often execute validation procedures prior to performing their primary tasks. For example, D may use A_1 to construct output without prior sanitization, but at the beginning of its execution it will check A_2 to see if it contains a "@" character, since A_2 represents an email address. In such situations, none of our n test cases would find an error, since they would not cause D to reach its output construction phase. Instead, they would cause D to terminate early and create an error message describing A_2 as an invalid email. However, D would indeed be vulnerable. A human attacker wanting to exploit D could supply a valid email address and learn that D uses A_1 to construct output without sanitizing it first.

To eliminate this kind of false negative, we added a *deep injection* [15] mechanism to WAVES. Using a *negative response extraction (NRE)* technique [15], the mechanism determines whether or not D uses a validation procedure. The naïve approach is used in the absence of validation. Otherwise, WAVES attempts to use its injection knowledge base to assign valid values to all arguments. Using a trial-and-error strategy, test cases are repeatedly generated and tested in an attempt to identify valid values for all arguments. If successful, then for each of the n test cases, valid values are used for arguments that do not contain malicious data. Otherwise, WAVES degrades to using the naïve approach and generates a message indicating that its test may be subject to a high false negative rate.

3.3. Side Effects Elimination

The second question is based on the tendency of some DEPs to cause permanent state changes in Web applications. For example, for every submission, a DEP D for user registration may add a new user to a database. If D accepts ten arguments and allows us to generate ten test cases, it is possible that ten new user records will be created as a side effect. Note that this procedure implies testing with a single malicious pattern; in practice, numerous patterns must be tested in order to provide good coverage. Testing for ten malicious patterns means that one hundred meaningless database records must be created.

We therefore added three testing modes to WAVES—heavy, relaxed, and safe modes. The heavy mode was our original mode; side effects were simply ignored in the interest of discovering all vulnerabilities. For the two new modes, DEPs were classified according to their types into three disjoint sets S_{safe}, S_{unsafe}, and $S_{unknown}$. $\forall D \in S_{safe}$, $D.T \in \{tS, tA\}$; $\forall D \in S_{unsafe}$, $D.T \in \{tR, tM\}$; $\forall D \in S_{unknown}$, $D.T$=tU. In both the relaxed and safe modes, DEPs belonging to S_{unsafe} are not tested, and S_{safe} DEPs are tested using the heavy mode. In the relaxed mode, $S_{unknown}$ DEPs are tested using the malicious pattern that is most likely to reveal errors. In safe mode, these are not tested.

3.4. Output Observation

After submitting a test case to a DEP, its output (HTTP response) is analyzed to detect Policy 1 violations [15]. To avoid XSS vulnerabilities, client-submitted data containing <script> HTML tags must be processed prior to being used for output construction. Proper processing entails a) outputting errors that indicate the detection of an attack, and b) removing the tag while still processing the request, and c) encoding the <script> tag so that it is *displayed* rather than *interpreted* by the browser. To help users observe whether such sanitization steps are being taken by a DEP, we designed test patterns so that the absence of a sanitization routine triggers the execution of a special Javascript by the browser when it renders the DEP output. An example test pattern is shown in Figure 7.

```
<script>alert("WAVES_TEST_1");</script>
```

Figure 7. An example of our test pattern for XSS.

As described in Section 4, Microsoft's Internet Explorer (IE) was added as a core WAVES component. Accordingly, after submitting the test pattern to a DEP and retrieving its output, it is possible to monitor embedded IE behavior. If IE makes an attempt to display a "WAVES_TEST_1" message box after the response is retrieved, we know that a) the DEP is using one of its arguments to construct output, and b) it did not perform proper sanitization prior to output construction. Such DEPs are considered vulnerable to XSS.

3.5. Test Case Reduction

For any DEP accepting n arguments, the naïve approach requires $n \times m$ test cases for testing against m malicious patterns. To reduce the number of test cases, we modified the test patterns according to the arguments in which the patterns were placed. For example, if placed in the first argument of a DEP, the test pattern shown in Figure 7 will change to:

```
<script>alert("WAVES_TEST_1_ARG_1");</script>
```

This allows for the use of IE behavior to identify vulnerable arguments. Using this strategy, we placed modified versions of the same malicious pattern into all arguments of a targeted DEP. This approach requires only $1 \times m = m$ cases to be tested against m malicious patterns. When two or more malicious patterns appear in the output, the message box events are captured sequentially and vulnerable arguments are identified.

4. System Implementation

WAVES' system architecture is shown in Figure 8. The crawlers act as interfaces between Web applications and software testing mechanisms. Without them we would not be able to apply our testing techniques to Web applications. To make the crawlers exhibit the same behaviors as browsers, they were equipped with IE's Document Object Model (DOM) parser and scripting engine. We chose IE's engines over others (e.g. Gecko [30] from Mozilla) because IE is the target of most attacks. User interactions with Javascript-created dialog boxes, script error pop-ups, security zone transfer warnings, cookie privacy violation warnings, dialog boxes (e.g. "Save As" and "Open With"), and authentication warnings were all logged but suppressed to ensure continuous crawler execution. Note that a subset of the above events is triggered by our test cases or by Web application errors. An error example is a Javascript error event produced by a scripting engine during a runtime interpretation of Javascript code. The crawler suppresses the dialog box that is triggered by the event and performs appropriate processing. When an event indicates an error, it logs the event and prepares corresponding entries to generate an assessment report.

When designing the crawler, we looked at ways that HTML pages reveal the existence of DEPs or other pages, and came up with the following list:

1. Traditional HTML anchors.
   Ex: Google
2. Framesets.
   Ex: <frame src = "http://www.google.com/top_frame.htm">
3. Meta refresh redirections.
   Ex: <meta http-equiv="refresh"
   Ex: content="0; URL=http://www.google.com">
4. Client-side image maps.
   Ex: <area shape="rect" href ="http://www.google.com">
5. Javascript variable anchors.
   Ex: document.write("\" + LangDir + "\index.htm");
6. Javascript new windows and redirections.
   Ex: window.open("\" + LangDir + "\index.htm");
   Ex: window.href = "\" + LangDir + "\index.htm";
7. Javascript event-generated executions.
   Ex: HierMenus (http://www.webreference.com)
8. Form submissions.

We established a sample site to test several commercial and academic crawlers, including Teleport [53], WebSphinx [29], Harvest [7], Larbin [49], Web-Glimpse [25], and Google. None were able to crawl beyond the fourth level of revelation–about one-half of the capability of the WAVES crawler. Revelations 5 and 6 were made possible by WAVES' ability to interpret Javascripts. Revelation 7 also refers to link-revealing Javascripts, but only following an onClick, onMouseOver, or similar user-generated event. WAVES performs an event-generation process to stimulate the behavior of active content. This allows WAVES to detect malicious components and assists in the URL discovery process. During stimulation, Javascripts located within the assigned event handlers of dynamic components are executed, possibly revealing new links. Many current Web sites incorporate DHTML menu systems to aid user navigation. These and similar Web applications contain many links that can only be identified by crawlers capable of handling level-7 revelations. Also note that even though the injection knowledge manager's (IKM) main goal is to produce variable candidates so as to bypass validation procedures, the same knowledge can also be used during the crawl process. When a crawler encounters a form, it queries the IKM; the data produced by the IKM is submitted by the crawler to the Web application for deep page discovery.

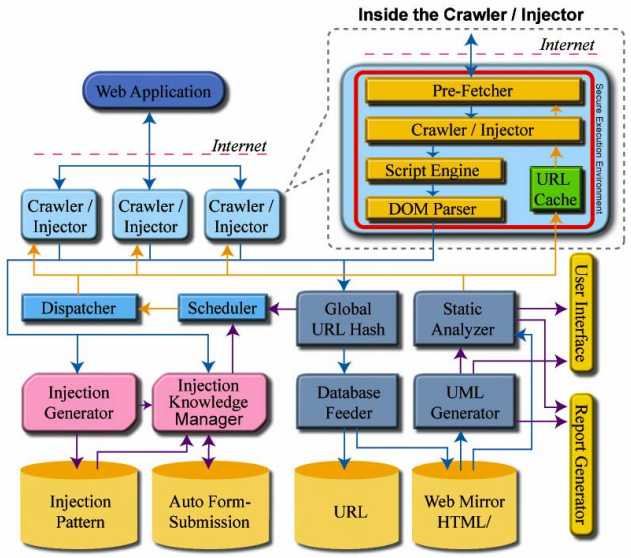

Figure 8. System architecture of WAVES.

In the interest of speed, we implemented a URL hash (in memory) in order to completely eliminate disk access during the crawl process. A separate 100-record cache helps to reduce global bottlenecks at the URL hash. See Cho [9] for a description of a similar implementation strategy. The database feeder does not insert retrieved information into the underlying database until the crawl is complete. The scheduler is responsible for managing a breadth-first crawl of targeted URLs; special care has been taken to prevent crawls from inducing harmful impacts on the Web application being tested. The dispatcher directs selected target URLs to the crawlers and controls crawler activity. Results from crawls and injections are organized in HTML format by the report generator.

5. Related Work

Offutt [34] surveyed Web managers and developers on quality process drivers and found that while time-to-market is still considered the most important quality criterion for traditional software, security is now very high on the list of concerns for Web application development. In [15], we described our preliminary results in developing WAVES—a testing platform for remote, black-box testing of Web application security. In the WebSSARI project, we formalized Web application vulnerability as a secure information flow problem, and experimented with static analysis techniques. We described our algorithm based on Strom and Yemini's [52] typestate in [16] and our method based on bounded model checking in [17]. Static analysis and runtime enforcement of secure information flow has been a classic problem in security research. The first widely accepted model for secure information flow was given by Bell and La Padula [2]. For compile-time detection of insecure information flow, Denning [11] first established a lattice model for analyzing imperative programming languages based on a program abstraction derived from an *instrumented semantics* of a language. Since then, much research efforts have been devoted to compile-time detection of information flow; some of which proposed the use of runtime validation techniques as compliments. For a comprehensive survey, the reader is referred to Sabelfeld and Myers [44].

We presented in this paper a Web application security scanner—an automated software testing platform for the remote, black-box testing of Web applications. In Bertolino's [3] words, software testing involves the "dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified behavior." Unit, regression, conformance, and other types of tests make use of carefully chosen or created test cases to identify faults. In white-box testing, a typical solution is to make test case selection based on source code—the strategy adopted by Rapps and Weyuker [38] in their data flow testing model based on program paths (which they refer to as du-paths) that connect variable definitions and

uses. Many errors in variable definition remain hidden until the variable is referenced at a program point far away from its definition. Data flow criteria are used to determine the extent to which du-paths are tested. Liu et al. [23] [24] were among the first to propose a model for applying data flow testing to Web applications. Treating individual Web application components as objects, they created a Web Application Test Model (WATM) to capture structural test artifacts. According to du-paths of interest, test cases are derived from flow graphs generated via an analysis of a Web application's source code.

Andrews, Offutt and Alexander [1] proposed a system-level testing technique based on finite state machines (FSMs) that model subsystems of Web applications. Test requirements are generated as FSM state subsequences, which are then combined to form complete executable tests. In general, all the above mentioned Web application testing approaches are "white-box" approaches that first build system models from source code and then use the models to identify test requirements. Generating test cases for the requirements require extensive user participation. In contrast, WAVES is a WSS that performs remote, black-box testing with automated test case generation (see WSS definition in Section 3.1).

AppScan [46], *ScanDo* [19], *WebInspect* [51] are commercial Web application security scanners similar to WAVES; as discussed in Section 1, they also share the same drawbacks as WAVES [15]. This paper attempted to address these drawbacks. VeriWeb [51] is a research project that addresses the automated testing of Web applications. VeriWeb embeds Gecko [30] for browser emulation, while WAVES embeds IE. IE was our first choice because most browser attacks are aimed at IE instead of Netscape Navigator. Both VeriWeb and WAVES perform automated form submissions, a reflection of studies on searching the hidden Web [4] [18] [22] [37]. To automatically generate valid input data, VeriWeb uses Smart Profiles, which represents sets of user-specified attribute-value pairs. In contrast, WAVES followed suggestions offered by Bergman [5] and Raghavan [37] and incorporated a more complex self-learning knowledge base.

As explained in Section 3.2, to improve test coverage, a WSS must supply valid arguments to DEPs. AppScan [46], WebInspect [51] and VeriWeb [51] rely on user-supplied values for effective test case generation, while ScanDo [19] incorporates an automated mechanism similar to WAVES' [15]. Though both WAVES [15] and ScanDo [19] addressed automated test case generation, neither discussed test case effectiveness. In this paper, we further defined three scanning modes and discussed test case generation strategies appropriate for each mode. Real-world experimental results for the three different strategies were also reported. In hidden Web research, Liddle et al. [22] researched strategies to generate effective test cases that cause a DEP to exhibit all possible behavior. We plan to enhance WAVES by incorporating similar strategies to further improve test case effectiveness. Elbaum, Sarre, and Rothermel [13] showed that user session data can be used to improve test data generation for Web applications. Compared to automated form filling techniques [4] [18] [22] [37] [15] that originated from hidden Web research, their approach provides a novel and effective way of collecting valid values for DEP arguments. However, according to definition given in Section 3.1, a WSS is posed as an outsider to the target Web application, and thus session data collection would be difficult.

Recently, Offutt et al. [35] proposed *bypass testing* of Web applications. Similar to our efforts here, they take a remote, black-box approach to testing Web applications. More precisely, their approach is a mixture of both black-box and white-box strategies; server-side source code is assumed unknown, so a mechanism is proposed for remote DEP discovery. On the other hand, client-side source code (Javascripts) are collected by crawling and then analyzed. Information gathered is combined to generate bypass tests to assess Web application quality and security. Currently, WAVES does not analyze Javascripts to improve test case generation.

6. Experimental Results

To study the effects of our proposed mechanisms, we conducted two experiments. In WebSSARI [16], we statically verified 230 open-source Web applications. Among the 69 projects found vulnerable, 38 acknowledged our finding and stated plans to provide patches. In experiment 1, we chose 5 from the 38 projects that a) allowed a public user to modify application state (database), and b) appeared to be easy to install—*phpPgAdmin*, *iNuke*, *WebMovieDB*, *Tiki CMSGroupware*, and *PHP Support Tickets*. We then setup and tested these applications using the original heavy and the new relaxed and safe modes of WAVES. We wanted to observe what modes induced side effects and to compare the coverage of the different modes to WebSSARI's static verification results.

Results are presented in Figure 9. None of the modes detected new vulnerabilities that had not been previously discovered by WebSSARI, and none were able to identify all WebSSARI-detected vulnerabilities. Some possible reasons for our testing failures include:

1. Undetected DEPs. The detection of some DEPs might require more complex procedures (e.g., form submissions) than those contained in WAVES.

2. Failure to observe HTML output. For example, WAVES is not capable of detecting SQL injection vulnerabilities when a Web application deliberately suppresses database error messages.
3. Detection pattern sets failed to reveal one or more vulnerabilities.
4. WAVES failed to provide valid arguments for certain DEPs that enforce validation procedures.

The widest coverage was provided by the heavy mode, which detected 80 percent of all vulnerabilities, with all failures attributing to reason #4. However, the heavy mode induced side effects in three of the five applications. In contrast, the relaxed mode did not induce any side effects, but it detected 58.4 percent of the vulnerabilities found by WebSSARI. The safe mode also failed to induce side effects, but it achieved coverage of only 20 percent.

Project \ Algorithm	Safe	Relaxed	Heavy	WebSSARI [16]
phpPgAdmin	0	2	3	3
INuke	0	2	2	3
WebMovieDB	3	4	7	7
Tiki CMSGroupware	2	6	8	12
PHP Support Tickets	8	24	32	40
Total	**13**	**38**	**52**	**65**

Figure 9. Results from experiment 1—the number of vulnerabilities found by WebSSARI [16] and the three scan modes.

In experiment 2, we tested random websites with WAVES for two consecutive days. To avoid side effects and potential legal issues, we limited this test to a search for XSS vulnerabilities using the relaxed mode. We defined a *potentially* vulnerable website as one whose cookies could be stolen by an XSS. A potentially vulnerable website whose users' interests could be exploited by stolen cookies was defined as being *strictly* vulnerable. An example is an online shop that a) uses cookies to support user authentication and b) stores credit card information in database to save users the time required to re-enter the information. For such a site, although stolen cookies do not reveal credit information, they can be used to make purchases on the victim's behalf.

Among the 1120 tested sites, 86 were potentially vulnerable and 55 were strictly vulnerable. Figure 10 lists the 55 strictly vulnerable sites; "SH" denotes sites with online shopping features; "COM" denotes commercial sites that do not incorporate online shopping, but is vulnerable to other types of privilege escalation. For example, www.no-ip.com allows for changing of customer domain name service (DNS) records by an attacker through XSS.

We observed that one of the 55 strictly vulnerable sites used a commercial shopping cart system called *ShopSite* that is sold and used world-wide. We informed the company of a vulnerability that would allow an outside attacker to hijack a session and make purchases using a victim's identification. The company that sells *ShopSite* acknowledged the vulnerability and stated that it had started to develop a patch that would soon be released. NetIQ, a Nasdaq company that sells many Web- and security-related products, had a similar vulnerability in their online store.

Site URL	Class	Site	Class
www.netiq.com	COM	www.Langreiter.com	COM
www.Xerox.com	SH	searchcgi.Apple.com	SH
www.Corel.com	SH	www.Shift4.com	COM
busca.Adobe.com	SH	searchnetworking.Techtarget.com	COM
www3.Addall.com	SH	us.Yoshioris.com	COM
www.China-airlines.com	SH	www.Britannica.com	COM
www.Shopsite.com	SH	intranetsolutions.Westlaw.com	COM
www.Dataviz.com	SH	www.No-ip.com	COM
www.Maguma.com	SH	nbci.Msnbc.com	COM
www2.HomeTime.com	SH	www.Atnet1000.com	COM
www.Kitchenshop.com	SH	www.Taihoo.com	COM
www.Shopnw.net	SH	usa.Asus.com	COM
www.Ronlee.com	SH	www.Nexus-pt.com	COM
www.Marquel.com	SH	internet.Flyer.co.uk	COM
thevault.Basiclink.com	SH	www.dotauthority.com	COM
www.Iste.org	SH	rrcrossing.com	COM
www.Movieflix.com	SH	www.quotablequotes.net	COM
www.Ticketmaster.co.uk	SH	www.Archive.org	ORG
www.Windance.com	SH	www.Ccidnet.com	PTL
www.Okwap.com	SH	www.Sohu.com	PTL
customer.3721.com	SH	www.163.com mail.21cn.com	PTL
www.Strollerdepot.com	SH	mail.21cn.com	PTL
www.Handmark.com	SH	Sina.com.tw	PTL
amos.Catalogcity.com	SH	Pchome.com.tw	PTL
www.aclens.com	SH	Searchtheweb.com	PTL
rosetta.Upenn.edu	ACM	Newsgroup.com.hk	PTL
stanfordwho.Stanford.edu	ACM	www.Yehey.com	PTL
www.Pse.cz	COM		

SH: Shopping site; COM: Commercial site;
PTL: Portal site; ACM: Academic site
Total: 55

Figure 10. The list of strictly vulnerable sites found during experiment 2.

7. Conclusion

In their efforts to assess Web application security, researchers from both academic and private sector are devoting a considerable amount of resources to developing Web application security scanners (WSS). They are achieving some success, but little is known about potential side effects—e.g., how some auditing processes induce permanent changes in application states. Another drawback is that current WSSs (including our original WAVES) focus on SQL injection detection, but are deficient in XSS detection. In this paper, we addressed these problems by:

1. giving a formal definition of a WSS and a list of design challenges;
2. describing how Web application vulnerabilities emerge from insecure information flow and describing a data flow testing model to detect them;
3. listing test types that may induce side effects and describing a test case generation process capable of producing a non-detrimental set of test cases;
4. showing how a Web application can be observed from a remote location during testing;
5. defining three modes of remote security auditing, with a focus on potential side effects;
6. conducting an experiment using three different modes and five real-world Web applications to compare differences in their coverage and induced side effects; and
7. conducting an experiment using the relaxed mode to scan random websites.

In our experiments, the heavy mode revealed 80 percent of all errors found by static verification. This shows that our remote, black-box testing approach provides a useful alternative to static analysis when source code and local access to the target Web application is unavailable. The 58.4 percent coverage of the relaxed mode shows that effective non-detrimental testing is possible. The 55 strictly vulnerable sites identified during a 48-hour, relaxed mode scan shows that a) our proposed mechanism for testing insecure information flow can be successfully used to detect XSS, b) non-detrimental testing still yields effective results, and c) XSS still poses a significant threat to today's Web applications. Furthermore, since tools that are similar to WAVES in many respects are being developed and used by hackers, we note that vulnerable websites can be easily identified by performing controlled "attacks" similar to experiment 2 with more malicious motivations.

8. References

[1] Andrews, A., Offutt, J., Alexander, R. "Testing Web Applications by Modeling with FSMs." Submitted for publication, January 2004.

[2] Bell, D. E., La Padula, L. J. "Secure Computer System: Unified Exposition and Multics Interpretation." Tech Rep. ESD-TR-75-306, MITRE Corporation, 1976.

[3] Bertolino, A. "Knowledge Area Description of Software Testing," In Guide to the Software Engineering Body of Knowledge SWEBOK (v. 0.7), Chapter 5, Software Engineering Coordinated Committee (Joint IEEE Computer Society-ACM Committee), April, 2000. http://www.swebok.org

[4] Benedikt M., Freire J., Godefroid P., "VeriWeb: Automatically Testing Dynamic Web Sites." In *Proc. 11th Int'l Conf. World Wide Web,* Honolulu, Hawaii, May 2002.

[5] Bergman, M. K. "The Deep Web: Surfacing Hidden Value." Deep Content Whitepaper, 2001.

[6] Bobbitt, M. "Bulletproof Web Security." Network Security Magazine, TechTarget Storage Media, May 2002. http://infosecuritymag.techtarget.com/2002/may/bulletproof.shtml

[7] Bowman, C. M., Danzig, P., Hardy, D., Manber, U., Schwartz, M., Wessels, D. "Harvest: A Scalable, Customizable Discovery and Access System." Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, Boulder, 1995.

[8] CERT. "CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests." http://www.cgisecurity.com/articles/xss-faq.shtml

[9] Cho, J., Garcia-Molina, H. "Parallel Crawlers." In *Proc 11th Int'l Conf. World Wide Web*, p.124-135, Honolulu, Hawaii, May 2002

[10] Curphey, M., Endler, D., Hau, W., Taylor, S., Smith, T., Russell, A., McKenna, G., Parke, R., McLaughlin, K., Tranter, N., Klien, A., Groves, D., By-Gad, I., Huseby, S., Eizner, M., McNamara, R. "A Guide to Building Secure Web Applications." The Open Web Application Security Project, v.1.1.1, Sep 2002.

[11] Denning, D. E. "A Lattice Model of Secure Information Flow." Communications of the ACM, 19(5):236-243, 1976.

[12] Dharmapurikar, S., Krishnamurthy, P., Sproull, T., and Lockwood, J. "Deep Packet Inspection Using Parallel Bloom Filters." In *Proc. 11th Symp. High Performance Interconnects (HOTI'03)*, p.44-51, Stanford, California, 2003.

[13] Elbaum, S., Karre, S., Rothermel, G. "Improving Web Application Testing with User Session Data." In *Proc. 25th Int'l Conf. Software Engineering*, p. 49-59, Portland, Oregon, 2003.

[14] Goguen, J. A., Meseguer, J. "Security Policies and Security Models." In *Proc. IEEE Symp. Security and Privacy*, pages 11-20, Oakland, California, Apr 1982.

[15] Huang, Y. W., Huang, S. K., Lin, T. P., Tsai, C. H. "Web Application Security Assessment by Fault Injection and Behavior Monitoring." In *Proc. 12th Int'l World Wide Web Conference*, p.148-159, Budapest, Hungary, 2003.

[16] Huang, Y. W., Yu, F., Hang, C., Tsai, C. H., Lee, D.T., Kuo, S. Y. "Securing Web Application Code by Static Analysis and Runtime Inspection." In: *Proc. 13th Int'l World Wide Web Conference*, New York, 2004.

[17] Huang, Y. W., Yu, F., Hang, C., Tsai, C. H., Lee, D. T., Kuo, S. Y. "Verifying Web Applications Using Bounded Model Checking." In: *Proc. 2004 Int'l Conf.*

Dependable Systems and Networks (DSN2004), Florence, Italy, Jun 28-Jul 1, 2004.

[18] Ipeirotis P., Gravano L., "Distributed Search over the Hidden Web: Hierarchical Database Sampling and Selection." In The 28th Int'l Conf. Very Large Databases (VLDB2002), p.394-405, Hong Kong, China, Aug 2002.

[19] Kavado. "ScanDo Web Application Scanner v.2.0." KaVaDo Whitepaper , 2004. http://www.kavado.com

[20] Kavado. "Application-Layer Security: InterDo 3.0." KaVaDo Whitepaper, 2003. http://www.kavado.com

[21] Krishnamurthy, A. "Hotmail, Yahoo in the run to rectify filter flaw." TechTree.com, March 24, 2004. http://www.techtree.com/techtree/jsp/showstory.jsp?storyid=5038

[22] Liddle, S., Embley, D., Scott, D., Yau, S. H. "Extracting Data Behind Web Forms." In *Proc. Workshop on Conceptual Modeling Approaches for e-Business*, Tampere, Finland, Oct 2002.

[23] Liu, C. H., Kung, D. C., Hsia, P., Hsu, C. T. "Structural Testing of Web Applications." In *Proc. 11th Int'l Symp. Software Reliability Engineering (ISSRE2000)*, p.84-96, Oct 8-11, 2000.

[24] Liu, C. H., Kung, D. C., Hsia, P., Hsu, C. T. "Object-Based Data Flow Testing of Web Applications." In Proc. 1st Asia-Pacific Conf. on Quality Software (APAQS'00), Hong Kong, China, Oct 30-31, 2000.

[25] Manber, U., Smith, M., Gopal B., "WebGlimpse—Combining Browsing and Searching." In *Proc. USENIX 1997 Annual Technical Conf., Anaheim*, California, Jan 1997.

[26] Meier, J.D., Mackman, A., Vasireddy, S. Dunner, M., Escamilla, R., Murukan, A. "Improving Web Application Security—Threats and Countermeasures." Microsoft Corporation, 2003.

[27] Microsoft. "Scriptlet Security." Getting Started with Scriptlets, MSDN Library, 1997.

[28] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnindhtm/html/instantdhtmlscriptlets.asp

[29] Miller, R. C., Bharat, K. "SPHINX: A Framework for Creating Personal, Site-Specific Web Crawlers." In *Proc. 7th Int'l World Wide Web Conf.*, p.119-130, Brisbane, Australia, April 1998.

[30] Mozilla.org. "Mozilla Layout Engine." http://www.mozilla.org/newlayout/

[31] Netscape. "JavaScript Security in Communicator 4.x." http://developer.netscape.com/docs/manuals/communicator/jssec/contents.htm#1023448

[32] Neumann, P. G. "Risks to the Public in Computers and Related Systems." *ACM SIGSOFT Software Engineering Notes*, 25(3), p.15-23, 2000.

[33] Ohmaki, K. "Open Source Software Research Activities in AIST towards Secure Open Systems." *In Proc. 7th IEEE Int'l Symp. High Assurance Systems Engineering (HASE'02)*, p.37, Tokyo, Japan, Oct 23-25, 2002.

[34] Offutt, J. "Quality Attributes of Web Software Applications." IEEE Software, 19(2), 25-32, Mar 2002.

[35] Offutt, J., Wu, Y., Du, X., Huang, H. "Web Application Bypass Testing." Submitted for publication, Mar 2004.

[36] Park, J.S., Sandhu, R. "Secure Cookies on the Web." *IEEE Internet Computing Archive*, 4(4), p.36-44, Jul 2000.

[37] Raghavan, S., Garcia-Molina, H. "Crawling the Hidden Web." In Proc. 27th Int'l Conf. Very Large Databases (VLDB2001), p.129-138, Roma, Italy, Sep 2001.

[38] Rapps, S., Weyuker, E. J. Selecting Software Test Data Using Data Flow Information. IEEE Transactions on Software Engineering, SE-11, p.367-375, 1985.

[39] Ricca, F., Tonella, P. "Analysis and Testing of Web Applications." In *Proc. 23rd IEEE Int'l Conf. Software Engineering*, p.25–34, Toronto, Ontario, Canada, May 2001.

[40] Ricca, F., Tonella, P. "Web Site Analysis: Structure and Evolution." In *Proc. IEEE Int'l Conf. Software Maintenance*, p.76-86, San Jose, California, Oct 2000.

[41] Ricca, F., Tonella, P., Baxter, I. D. "Restructuring Web Applications via Transformation Rules." Information and Software Technology, 44(13), 811-825, Oct 2002.

[42] Ricca, F., Tonella, P. "Understanding and Restructuring Web Sites with ReWeb." IEEE Multimedia, 8(2), 40-51, Apr 2001.

[43] Ricca, F., Tonella, P. "Web Application Slicing." In *Proc. IEEE Int'l Conf. Software Maintenance*, p.148-157, Florence, Italy, Nov 2001.

[44] Sabelfeld, A., Myers, A. C. "Language-Based Information-Flow Security." IEEE Journal on Selected Areas in Communications, 21(1):5-19, 2003.

[45] Sanctum Inc. "AppShield 4.5 Whitepaper." 2003. http://www.sanctuminc.com

[46] Sanctum Inc. "AppScan 4.5 QA & Audit Editions—White paper," 2004. http://www.sanctuminc.com

[47] Sandhu, R. S. "Lattice-Based Access Control Models." IEEE Computer, 26(11):9-19, 1993.

[48] Sanjit, A. S., Bryant, R. E., "Unbounded, Fully Symbolic Model Checking of Timed Automata using Boolean Methods." In *Proc. 15th Int'l Conf. Computer-Aided Verification*, p.154-166, volume LNCS 2725, Boulder, Colorado, 2003. Springer-Verlag.

[49] Sebastien@ailleret.com. "Larbin – A Multi-Purpose Web Crawler." http://larbin.sourceforge.net/index-eng.html

[50] Scott, D., Sharp, R. "Abstracting Application-Level Web Security." In *Proc. 11th Int'l World Wide Web Conference*, p.396-407, Honolulu, Hawaii, 2002.

[51] SPI Dynamics. "Layer 7—The Future of Vulnerabilities (Featuring SPI Dynamics WebInspect)." SPI Dynamics Whitepaper, Sep 2003.

[52] Strom, R. E., Yemini, S. A. "Typestate: A Programming Language Concept for Enhancing Software Reliability." IEEE Transactions on Software Engineering, 12(1):157-171, Jan 1986.

[53] Tennyson Maxwell Information Systems, Inc. "Teleport Webspiders." http://www.tenmax.com/teleport/home.htm

[54] Varghese, S. "Microsoft patches critical Hotmail hole." TheAge.com, March 24, 2004. http://www.theage.com.au/articles/2004/03/24/1079939690076.html