

Web Application Security Assessment by Fault Injection and Behavior Monitoring

Yao-Wen Huang, Shih-Kun Huang,
and Tsung-Po Lin

Institute of Information Science, Academia Sinica
Nankang 115 Taipei, Taiwan
{ywhuang,skhuang,lancelot}
@iis.sinica.edu.tw

Chung-Hung Tsai

Department of Computer Science and Information
Engineering,
National Chiao Tung University
300 Hsinchu, Taiwan
chotsai@csie.ntu.edu.tw

ABSTRACT

As a large and complex application platform, the World Wide Web is capable of delivering a broad range of sophisticated applications. However, many Web applications go through rapid development phases with extremely short turnaround time, making it difficult to eliminate vulnerabilities. Here we analyze the design of Web application security assessment mechanisms in order to identify poor coding practices that render Web applications vulnerable to attacks such as SQL injection and cross-site scripting. We describe the use of a number of software-testing techniques (including dynamic analysis, black-box testing, fault injection, and behavior monitoring), and suggest mechanisms for applying these techniques to Web applications. Real-world situations are used to test a tool we named the Web Application Vulnerability and Error Scanner (WAVES, an open-source project available at <http://waves.sourceforge.net>) and to compare it with other tools. Our results show that WAVES is a feasible platform for assessing Web application security.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques – *Modules and interfaces*; D.2.5 [Software Engineering]: Testing and Debugging – *Code inspections and walk-throughs*, and *Testing tools*; H.3.1 [Information Storage and Retrieval]: Content Analysis and Indexing – *Dictionaries and Indexing Method*; K.6.5 [Management of Computing and Information Systems]: Security and Protection – *Invasive software and Unauthorized access*

General Terms

Security, Design.

Keywords

Web Application Testing, Security Assessment, Fault Injection, Black-Box Testing, Complete Crawling.

1. INTRODUCTION

Web masters and system administrators all over the world are witnessing a rapid increase in the number of attacks on Web applications. Since vendors are becoming more adept at writing secure code and developing and distributing patches to counter

traditional forms of attack (e.g., buffer overflows), hackers are increasingly targeting Web applications. Most networks are currently guarded by firewalls, and port 80 of Web servers is being viewed as the only open door. Furthermore, many Web applications (which tend to have rapid development cycles) are written by corporate MIS engineers, most of whom have less training and experience in secure application development compared to engineers at Sun, Microsoft, and other large software firms.

Web application security can be enhanced through the increased enforcement of secure development practices. Yet despite numerous efforts [42] and volumes of literature [20] [59] promoting such procedures, vulnerabilities are constantly being discovered and exploited. A growing number of researchers are developing solutions to address this problem. For instance, Scott and Sharp [54] have proposed a high-level input validation mechanism that blocks malicious input to Web applications. Such an approach offers protection through the enforcement of strictly defined policies, but fails to assess the code itself or to identify the actual weaknesses.

Our goal in this paper is to adopt software-engineering techniques to design a security assessment tool for Web applications. A variety of traditional software engineering tools and techniques have already been successfully used in assuring security for legacy software. In some studies (e.g., MOPS [18] and SPLint [24]), static analysis techniques have been used to identify vulnerabilities in UNIX programs; static analysis can also be used to analyze Web application code, for instance, ASP or PHP scripts. However, this technique fails to adequately consider the runtime behavior of Web applications. It is generally agreed that the massive number of runtime interactions that connect various components is what makes Web application security such a challenging task [30] [54].

In contrast, the primary difficulty in applying dynamic analysis to Web applications lies in providing efficient interface mechanisms. Since Web applications interact with users behind browsers and act according to user input, such interfaces must have the ability to mimic both the browser and the user. In other words, the interface must process content that is meant to be rendered by browsers and later interpreted by humans. Our interface takes the form of a crawler, which allows for a black-box, dynamic analysis of Web applications. Using a “complete crawling” mechanism, a reverse engineering of a Web application is performed to identify all data entry points. Then, with the help of a self-learning injection knowledge base, fault injection techniques are applied to detect SQL injection vulnerabilities.

Using our proposed Topic Model, the knowledge base selects the best injection patterns according to experiences learned through previous injection feedback, and then expands the knowledge base as more pages are crawled. Both previous experiences and knowledge expansion contribute to the generation of better injection patterns. We also propose a novel reply analysis algorithm in order to help the crawler interpret injection replies. By improving the observability [66] of the Web application being tested, the algorithm helps facilitate a “deep injection” mechanism that eliminates false negatives.

To imitate real-world interactions with Web applications, our crawler is equipped with the same capabilities as a full-fledged browser, thus making it vulnerable to malicious scripts that may have been inserted into a Web application via cross-site scripting. Since a malicious script that is capable of attacking an interacting browser is also capable of attacking the crawler, a secure execution environment (SEE) that enforces an anomaly detection model was built around the crawler. During the reverse engineering phase, all pages of a Web application are loaded into the crawler and executed. Input stimuli (i.e., simulated user events) are generated by the crawler to test the behavior of the page’s dynamic components. Any abnormal behavior will cause the SEE to immediately halt the crawler and audit the information. Thus, while offering self-protection, this layer also detects malicious scripts hidden inside Web applications.

This paper is organized as follows: SQL injection and cross-site scripting, along with our proposed detection models, are described in Section 2. The overall system architecture is introduced in Section 3. Comparisons with similar projects are made in Section 4, and experimental results are presented in Section 5. Conclusions are offered in Section 6.

2. DETECTING WEB APPLICATION VULNERABILITIES

We chose SQL injection and cross-site scripting vulnerabilities as our primary detection targets for two reasons: a) they exist in many Web applications, and b) their avoidance and detection are still considered difficult. Here we will give a brief description of each vulnerability, followed by our proposed detection models.

2.1 SQL Injection

Web applications often use data read from a client to construct database queries. If the data is not properly processed prior to SQL query construction, malicious patterns that result in the execution of arbitrary SQL or even system commands can be injected [6] [16] [25] [36] [60].

Consider the following scenario: a Web site includes a form with two edit boxes in its login.html to ask for a username and password. The form declares that the values of the two input fields should be submitted with the variables strUserName and strPassword to login.cgi, which includes the following code:

```
SQLQuery = "SELECT * FROM Users WHERE (UserName=" +  
strUserName + ") AND (Password=" + strPassword + ");"  
If GetQueryResult(SQLQuery) = 0 Then bAuthenticated = false;  
Else bAuthenticated = true;
```

If a user submits the username “Wayne” and the password “0308Wayne,” the SQLQuery variable is interpreted as:

```
"SELECT * FROM Users WHERE (strUserName= 'Wayne') AND  
(Password='0308Wayne');
```

GetQueryResult() is used to execute SQLQuery and retrieve the number of matched records. Note that user inputs (stored in the strUserName and strPassword variables) are used directly in SQL command construction without preprocessing, thus making the code vulnerable to SQL injection attacks. If a malicious user enters the following string for both the UserName and Password fields:

```
X' OR 'A' = 'A'
```

then the SQLQuery variable will be interpreted as:

```
"SELECT * FROM Users WHERE (strUserName='X' OR 'A' = 'A')  
AND (Password='X' OR 'A' = 'A');
```

Since the expression ‘A’ = ‘A’ will always be evaluated as TRUE, the WHERE clause will have no actual effect, and the SQL command will always be the equivalent of “SELECT * FROM Users”. Therefore, GetQueryResult() will always succeed, thus allowing the Web application’s authentication mechanism to be bypassed.

2.2 SQL Injection Detection

Our approach to SQL injection detection entails fault injection—a dynamic analysis process used for software verification and software security assessment. For the latter task, specially crafted malicious input patterns are deliberately used as input data, allowing developers to observe the behavior of the software under attack. Our detection model works in a similar manner—that is, we identify vulnerabilities in Web applications by observing the output resulting from the specially prepared SQL injection patterns.

Similar to other research on Web site testing and analysis [9] [48] [52], we adopted a black-box approach in order to analyze Web applications externally without the aid of source code. Compared with a white-box approach (which requires source code), a black-box approach to security assessment holds many benefits in real-world applications. Consider a government entity that wishes to ensure that all Web sites within a specific network are protected against SQL injection attacks. A black-box security analysis tool can perform an assessment very quickly and produce a useful report identifying vulnerable sites. To assure high security standards, white-box testing can be used as a complement [54].

In order to perform a black-box fault injection into a Web application, a reverse engineering process must first take place to discover all data entry points. To perform this task, we designed a crawler to crawl the Web application—an approach adopted in many Web site analysis [9] [48] [52] and reverse engineering [49] [50] [51] studies. We designed our crawler to discover all pages in a Web site that contain HTML forms, since forms are the primary data entry points in most Web applications. From our initial tests, we learned that ordinary crawling mechanisms normally used for indexing purposes [12] [19] [35] [38] [56] [62] are unsatisfactory in terms of thoroughness. Many pages within Web applications currently contain such dynamic content as Javascripts and DHTML. Other applications emphasize session management, and require the use of cookies to assist navigation mechanisms. Still others require user input prior to navigation. Our tests show that all traditional crawlers (which use static parsing and lack script interpretation abilities) tend to skip pages in Web sites that have these features. In both security assessment and fault injection,

completeness is an important issue—that is, all data entry points must be correctly identified. No attempt was made to exhaust input space, but we did emphasize the importance of comprehensively identifying all data entry points, since a single unidentified link would nullify the tests conducted for all other links. A description of our Web crawler is offered in Section 3.

During the reverse engineering process, HTML pages are parsed with a Document Object Model (DOM) [5] parser, and HTML forms are parsed and stored in XML format. These forms contain such data as submission URLs, available input fields and corresponding variables, input limits, default values, and available options.

Upon completion of the reverse engineering process, an attempt was made to inject malicious SQL patterns into the server-side program that processes the form's input. We referenced the existing literature on SQL injection techniques to create a set of SQL injection patterns [6] [16] [25] [36] [60]. The first step is to determine the variable on which to place these patterns. A typical example of HTML form source code is presented in Figure 1:

```
<Form action="submit.cgi" method="GET">
strUserName: <Input type="text" name="strUserName" size="20"
maxlength="20"><br>
strPassword: <Input type="password" name="strPassword"
size="20" maxlength="20"><br>
... (skipped)
```

Figure 1. A typical example of HTML form source code.

An example of a URL generated by submitting the form in Figure 1 is:

<http://waves.net/~lancelot/submit.cgi?strUserName=Wayne&strPasword=0308Wayne>

Note that two variables were submitted. If the SQL injection pattern is placed on strUserName, then the submitted URL will appear as:

Depending on the SQL injection technique being used, patterns are placed on either the first or last variable. If the server-side program does not incorporate a pre-processing function to filter malicious input, and if it does not validate the correctness of input variables before constructing and executing the SQL command, the injection will be successful.

However, if the server-side program detects and filters malicious patterns, or if the filtering mechanism is provided on a global scale (e.g., Scott and Sharp [54]), then injection will fail, and a false negative report will be generated. Many server-side programs execute validation procedures prior to performing database access. An example of a typical validation procedure added to the code presented in Figure 1 is shown below:

```
If Length(strUserName < 3) OR Length(strUserName > 20) Then
    OutputError("Invalid User Name") Else
If Length(strPassword <6) OR Length(strPassword) > 11) Then
    OutputError("Invalid Password") Else Begin
    SQLQuery = "SELECT * FROM Users WHERE UserName=" +
    strUserName + "AND Password=" + strPassword + ";"
    If GetQueryResult(SQLQuery) = 0 Then bAuthenticated = false;
    Else bAuthenticated = true;
End;
```

For the above code, our injection URL will fail because it lacks a password. The code requires that the variable strPassword carry text containing between 6 and 11 characters; if a random 6-11 character text is assigned to strPassword, injection will still succeed. We propose the use of a “deep injection” mechanism to eliminate these types of false negatives.

To bypass the validation procedure, the Injection Knowledge Manager (IKM) must decide not only on which variable to place the injection pattern, but also how to fill other variables with potentially valid data. Here we looked at related research in the area of automated form completion—that is, the automatic filling-out of HTML forms. A body of information approximately 500 times larger than the current indexed Internet is believed to be hidden behind query interfaces [10]—for example, patent information contained in the United States Patent and Trademark Office's Web site [64]. Since only query (and not browsing) interfaces are provided, these types of document repositories cannot be indexed by current crawling technologies. To accomplish this task, a crawler must be able to perform automatic form completion and to send queries to Web applications. These crawlers are referred to as “deep crawlers” [10] or “hidden crawlers” [29] [34] [46]. Here we adopted an approach similar to [46], but with a topic model that enhances submission correctness and provides a self-learning knowledge expansion model.

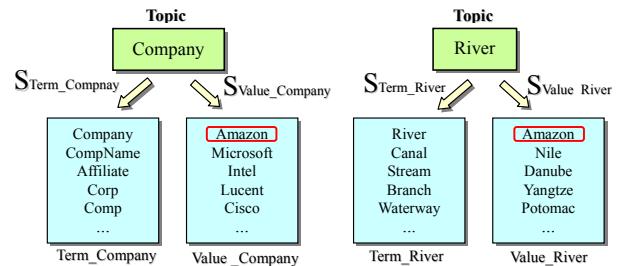


Figure 2. The Topic Model.

Automated form completion requires the selection of syntactically or even semantically correct data for each required variable. For example, the variable names “strUserName” and “strPassword” shown in Figure 1 reveal both syntactic and semantic information. The “str” prefix indicates text as the required data type, and the “UserName” suggests the text semantics (i.e., a person's name). Supplying the IKM with knowledge to provide a valid input for each variable requires the design and implementation of a self-learning knowledge base. Input values are grouped into “topics”—for example, the words or phrases “name,” “nick,” “nickname,” “first name,” “last name,” and “surname” are all indicators of human names, and are therefore grouped under the “Human Name” topic. Any value within this topic (e.g., Wayne or Ricky) can serve as semantically correct input data. These phrases form the value set elements of the topic.

Figure 2 is an illustration of the Topic Model. At its center is the “topic”—e.g., “Human Names,” “Company Names,” “River Names,” “Star Names,” and “Addresses.” Each topic is associated with a set (labeled $S_{Term_TopicName}$) that includes all possible word and phrases that describe the topic. Accordingly, $S_{Term_Company}$ might include “Company,” “Firm,” and “Affiliation,” and S_{Term_Sex} might include “Sex” and “Gender.” Each topic is associated with a second $S_{Value_TopicName}$ set containing a list of possible values. For instance, $S_{Value_Company}$ might include “IBM”,

“Sun”, and “Microsoft”. For any pair of topics (i.e., TopicA and TopicB), $|S_{Value_TopicA} \cap S_{Value_TopicB}| \geq 0$ (in other words, values can belong to more than one $S_{Value_TopicName}$). Thus, the value “Amazon” may appear in both $S_{Value_Company}$ and S_{Value_Rivers} , and “Sun” may appear in both $S_{Value_Company}$ and S_{Value_Stars} .

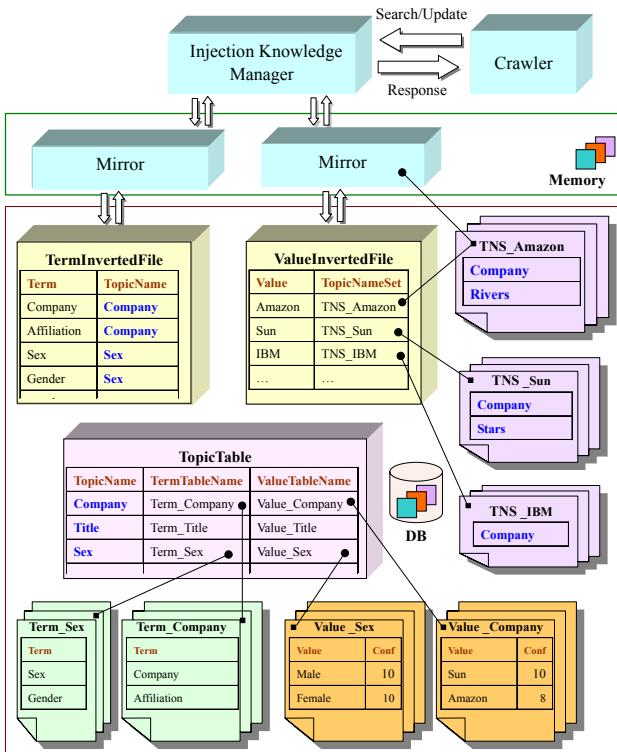


Figure 3. The Self-Learning Knowledge Base Model.

The self-learning knowledge base that was constructed according to this model is shown in Figure 3. The IKM provides the following functions to the crawler:

```
Get_Topic(String InputTerm [in], String Topic[out]);
Get_Value(String InputTerm [in], String CandidateValue[out]);
Expand_Values(String InputTerm [in], String Array PredefinedValues[out]);
Feedback(String Term [in], String Candidate [in], Boolean Succeeded [in]);
```

When confronted with a text box that carries no default value, the crawler calls IKM’s Get_Value(InputTerm, CandidateValue) to retrieve the best possible guess, where InputTerm is the variable name associated with the text box. Internally, Get_Value() utilizes Get_Topic(InputTerm, Topic), which checks whether a topic can be associated with InputTerm. The definitions of Get_Topic() and Get_Value() are given as follows:

```
Get_Topic(String InputTerm [in], String Topic [out]);
InputTerm - The newly encountered variable name or descriptive keyword. Denoted Terminput.
Topic - Name of the topic containing InputTerm, if any. Denoted Topicterm.
(1) Using TermInvertedFile, find Termmatch that is a term most similar to Terminput.
a.  $\forall$  Termi  $\in$  TermInvertedFile,
   Similarity(Termi) = 1/NearestEditDist(Terminput, Termi).
b. max = Max(Similarity(Term1), ..., Similarity(Termn)),
```

if max > p then

\exists Term_{match} \in TermInvertedFile,
Similarity(Term_{match}) = max
else Topic_{term} = "", return

(2) Return the name of the topic containing Term_{match}:

a. Topic_{term} = TermInvertedFile.GetTopicName(Term_{match})

Get_Value(String InputTerm [in], String CandidateValue [out]):

InputTerm - The newly encountered variable name or descriptive keyword. Denoted Term_{input}.

CandidateValue - The candidate value having highest confidence. Denoted Value_{MatchedTerm}.

(1) Check whether Term_{input} can be associated with a topic:

a. Get_Topic(Term_{input}, Topic_{term})

b. if Topic_{term} equal to "" then
Value_{MatchedTerm} = "", return

(2) Retrieve the candidate having the highest confidence:

a. S_{Value_MatchedTerm} = TopicTable.GetValueSet(Topic_{term})

b. \forall Value_i \in S_{Value_MatchedTerm},

max = Max(Conf(Value₁), ..., Conf(Value_n)),

c. \exists Value_{MatchedTerm} \in S_{Value_MatchedTerm},

Conf(Value_{MatchedTerm}) = max

Get_Topic() uses a simple string similarity-matching algorithm to compute InputTerm’s nearest edit distances to every term from every topic contained in the knowledge base. This approach ensures that similar phrases (e.g., “User_Name” and “UserName”) are marked as having a short distance. To reduce computation complexity, matching is performed using the TermInvertedFile table stored in memory (Figure 3). A minimum threshold p is set so that Get_Topic() may fail and return an empty string.

After an associated topic is identified, Get_Value() uses the ValueTableName field in TermInvertedFileTable to locate the corresponding S_{Value_MatchedTerm}, from which the candidate value with the highest confidence (denoted Value_{MatchedTerm}) is selected. If two or more candidates with the same confidence are identified, one is randomly selected. Value_{MatchedTerm} is then returned to the crawler, which calls Get_Value() iteratively until it has enough values to construct a deep SQL injection URL. Following an injection, the crawler calls Feedback() to supply the IKM with feedback on the successfulness of the injection. Confidence is adjusted for each value involved in the injection session.

The key terms used in the process just described consist of variable names gathered from the HTML form’s source code. Though programmers with good practices are likely to follow proper naming conventions, doing so is not considered as mandatory, and poor-looking codes will not affect a form’s appearance or functionality. For this reason, it is not possible to rely solely on these variable names to provide descriptive (syntactic or semantic) information regarding input fields. Raghavan [46] has proposed an algorithm called LITE (Layout-based Information Extraction Technique) to help identify input field semantics. In LITE, the HTML is sent to an approximate DOM parser, which calculates the location of each DOM element rendered on the screen; text contained in the element nearest the input field is considered descriptive. We took a similar approach: our crawler is equipped with a fully functional DOM parser, and thus contains knowledge on the precise layout of every DOM component. While variable names are extracted, the crawler also calculates the square-distance between input fields and all other

DOM components. The text from the nearest component is extracted as a second descriptive text. Keywords are further extracted from the text by filtering stop words. The keywords are used to call Get_Value() if the first call using the variable name fails.

After the query and ranking mechanisms are in place and the IKM begins to feed high-confidence terms to the crawlers, the next issue involves populating and automatically expanding the knowledge base. The IKM primarily relies on option lists found in HTML forms for the expansion of S_{Value} . Such option lists are rendered as “Combo Boxes” on the screen; when clicked, they present drop-down menus containing available options (e.g., a list of countries to be included in a registration form). When requesting data that has a fixed set of possible values—such as the country name in an address field, an option list is a commonly chosen input method.

When the crawler confronts an input variable with an attached option list, it calls Expand_Values(InputTerm, PredefinedValues), where InputTerm is the variable name and PredefinedValues the associated options list. According to InputTerm and PredefinedValues, Expand_Values() expands the knowledge base. We define Expand_Values() as follows:

Expand_Values (String InputTerm [in], String Array PredefinedValues[out];

InputTerm - The newly encountered variable name or descriptive keyword. Denoted $Term_{Input}$.
 PredefinedValues - The value returned to the caller.
 Denoted S_{Pred} .

- (1) Check whether $Term_{Input}$ can be associated with a topic:
 - a. Get_Topic($Term_{Input}$, Topic_{Match});
 - b. if Topic_{Match} not equal to “” then goto step (3)
- (2) $Term_{Input}$ not found in knowledge base, try to add $Term_{Input}$
 - a. try to find some value set S_{Sim_i} that resembles S_{Pred} :
 \forall topic Topic_i,
 $S_{Sim_i} = \{Value_i \mid Value_i \in Value_{Pred}, Value_j \in ValueInvertedFile, EditDist(Value_i, Value_j) < \rho, ValueInvertedFile.GetTopic(Value_i) = Topic_i\}$
 $S_{Value_i} = TopicTable.GetValueTable(Topic_i)$
 $Score(Topic_i) = \frac{|S_{Sim_i}|}{|S_{Value_i}|}$
 - b. max = MAX (Score(Topic₀),...,Score(Topic_n))
 - d. If max < ρ then return
 - e. $\exists Topic_{Match} \in TopicTable, Score(Topic_{Match}) = max$
 $S_{TermMatch} = TopicTable.GetTermTable(Topic_{Match})$
 - f. $S_{TermMatch} = S_{TermMatch} \cup S_{Pred}$
S_{TermMatch} is thus expanded.
- (3) $Term_{Input}$ associated with or added to a topic. Expand S_{Value} of the topic containing $Term_{Input}$.
 - a. $S_{ValueMatch} = TopicTable.GetValueTable(Topic_{Match})$
 - c. If $|S_{pred} - S_{ValueMatch}| > 0$ then
 $S_{ValueMatch} = S_{ValueMatch} \cup (S_{pred} - S_{ValueMatch})$
S_{ValueMatch} is thus expanded.

If Expand_Values() is able to associate a topic with InputTerm, it appends to the topic’s value set all possible values extracted from the newly encountered option list. This enabled the expansion of the value sets as pages are crawled. To expand the term sets, Expand_Values() search the ValueInvertedFile and

identify the existing value set S_{Value} that is most similar to the input set PredefinedValues. If one is identified, InputTerm is added to the term set of the topic of the matched S_{Value} . In the following example, assume that for the topic Company, $S_{Term_Company} = \{\text{Company}, \text{Firm}\}$ and $S_{Value_Company} = \{\text{IBM}, \text{HP}, \text{Sun}, \text{Lucent}, \text{Cisco}\}$. Then assume that a crawler encounters an input variable “Affiliation” that is associated with $S_{Value_Input} = \{\text{HP}, \text{Lucent}, \text{Cisco}, \text{Dell}\}$. The crawler calls Expand_Values() with “Affiliation” and S_{Value_Input} . After failing to find a nearest term for “Affiliation,” the Knowledge Manager notes that $S_{Value_Company}$ is very close to S_{Value_Input} , and inserts the term “Affiliation” into $S_{Term_Company}$ and the value $S_{Value_Input} - S_{Value_Company} = \{\text{Dell}\}$ into $S_{Value_Company}$. In this scenario, both $S_{Term_Company}$ and $S_{Value_Company}$ are expanded.

Here we will describe the mechanism for observing injection results. Injections take the form of HTTP requests that trigger responses from a Web application. Fault injection observability is defined as the probability that a failure will be noticeable in the output space [66]. The observability of a Web application’s response is extremely low for autonomous programs, which presents a significant challenge when building hidden crawlers [46]. After submitting a form, a crawler receives a reply to be interpreted by humans; it is difficult for a crawler to interpret whether a particular submission has succeeded or failed. Raghavan [46] [47] addresses the problem with a variation of the LITE algorithm: the crawler examines the top-center part of a screen for predefined keywords that indicate errors (e.g., “invalid,” “incorrect,” “missing,” and “wrong”). If one is found, the previous request is considered as having failed.

For successful injections, observability is considered high because the injection pattern causes a database to output certain error messages. By scanning for key phrases in the replied HTML (e.g. “ODBC Error”), a crawler can easily determine whether an injection has succeeded. However, if no such phrases are detected, the crawler is incapable of determining whether the failure is caused by an invalid input variable, or if the Web application filtered the injection and therefore should be considered invulnerable. To resolve this problem, we propose a simple yet effective algorithm called negative response extraction (NRE). If an initial injection fails, the returned page is saved as R_1 . The crawler then sends an intentionally invalid request to the targeted Web application—for instance, a random 50-character string for the UserName variable. The returned page is retrieved and saved as R_2 . Finally, the crawler sends to the Web application a request generated by the IKM with a high likelihood of validity, but without injection strings. The returned page is saved as R_3 . R_2 and R_3 are then compared using WinMerge [67], an open-source text similarity tool.

The return of similar R_2 and R_3 pages raises one of two possibilities: a) no validation algorithm was enforced by the Web application, therefore both requests succeeded; or b) validation was enforced and both requests failed. In the first situation, the failure of R_1 allows for the assumption that the Web application is not vulnerable to the injection pattern, even though it did not validate the input data. In the second situation, the crawler enters an R_3 regeneration and submission loop. If a request produces an R_3 that is not similar to R_2 , it is assumed to have bypassed the validation process; in such cases, a new SQL injection request is generated based on the parameter values used in the new, successful R_3 . If the crawler still receives the same reply after ten loops, it is assumed that either a) no validation is enforced but the

application is invulnerable, or b) a tight validation procedure is being enforced and automated completion has failed. Further assuming under this condition that the Web application is invulnerable induces a false negative (discussed in Section 5 as $P(F_L|V,D)$). If an injection succeeds, it serves as an example of the IKM learning from experience and eventually producing a valid set of values. Together with the self-learning knowledge base, NRE makes a deep injection possible. A list of all possible reply combinations and their interpretations are presented in Figure 4.

Combinations	Interpretation
$R_1 = R_2 = R_3$	1. All requests are filtered by validation procedure. Automated assessment is impossible. 2. Requests are not filtered, but Web application is not vulnerable.
$R_1 = R_2 \neq R_3$	R_3 bypassed validation. Regenerate R_1 with R_3 parameters and inject.
$R_2 = R_3 \neq R_1$	Malicious pattern recognized by filtering mechanism. Page not vulnerable.
$R_1 \neq R_2 \neq R_3$	R_1 is recognized as injection pattern, R_2 failed validation, R_3 succeeded. Regenerate R_1 with R_3 parameters and inject.

Figure 4. The NRE for deep injection.

2.3 Cross-Site Scripting

As with SQL injection, cross-site scripting [2] [15] [17] is also associated with undesired data flow. To illuminate the basic concept, we offer the following scenario.

A Web site for selling computer-related merchandise holds a public on-line forum for discussing the newest computer products. Messages posted by users are submitted to a CGI program that inserts them into the Web application's database. When a user sends a request to view posted messages, the CGI program retrieves them from the database, generates a response page, and sends the page to the browser. In this scenario, a hacker can post messages containing malicious scripts into the forum database. When other users view the posts, the malicious scripts are delivered on behalf of the Web application [15]. Browsers enforce a Same Origin Policy [37] [40] that limits scripts to accessing only those cookies that belong to the server from which the scripts were delivered. In this scenario, even though the executed script was written by a malicious hacker, it was delivered to the browser on behalf of the Web application. Such scripts can therefore be used to read the Web application's cookies and to break through its security mechanisms.

2.4 Cross-Site Scripting Detection

Indications of cross-site scripting are detected during the reverse engineering phase, when a crawler performs a complete scan of every page within a Web application. Equipping a crawler with the functions of a full browser results in the execution of dynamic content on every crawled page (e.g., Javascripts, ActiveX controls, Java Applets, and Flash scripts). Any malicious script that has been injected into a Web application via cross-site scripting will attack the crawler in the same manner that it attacks a browser, thus putting our WAVES-hosting system at risk. We used the Detours [28] package to create a SEE that intercepts

system calls made by a crawler. Calls with malicious parameters are rejected.

The SEE operates according to an anomaly detection model. During the initial run, it triggers a learning mode in WAVES as it crawls through predefined links that are the least likely to contain malicious code that induces abnormal behavior. Well-known and trusted pages that contain ActiveX controls, Java Applets, Flash scripts, and Javascripts are carefully chosen as crawl targets. As they are crawled, normal behavior is studied and recorded. Our results reveal that during startup, Microsoft Internet Explorer (IE)

1. locates temporary directories.
2. writes temporary data into registry.
3. loads favorite links and history lists.
4. loads the required DLL and font files.
5. creates named pipes for internal communication.

During page retrieval and rendering, IE

1. checks registry settings.
2. writes files to the user's local cache.
3. loads a cookie index if a page contains cookies.
4. loads corresponding plug-in executables if a page contains plug-in scripts.

The SEE uses the behavioral monitoring specification language (BMSL) [45] [55] to record these learned normal behaviors. This design allows users to easily modify the automatically generated specifications if necessary. Figure 5 presents an example of a SEE-generated BMSL description.

```
allowFile =
{"C:\WINDOWS\System32\mshtml.dll", ... (skipped)}
CreateFileW(filepath,access_mode,share_mode,SD,
           create_mode,attribute,temp_handle)
| (filepathnotinallowFile) → deny
```

Figure 5. A SEE-generated BMSL description.

The SEE pre-compiles BMSL descriptions into a hashed policy database. During page execution and behavior stimulation, parameters of intercepted system calls are compared with this policy database. If the parameters do not match the normal behavior policy (e.g., using "C:\autoexec.bat" as a parameter to call CreateFileEx), the call is considered malicious, since IE was not monitored to make any file access under the C:\ directory during the learning phase. Figure 6 illustrates the SEE mechanism.

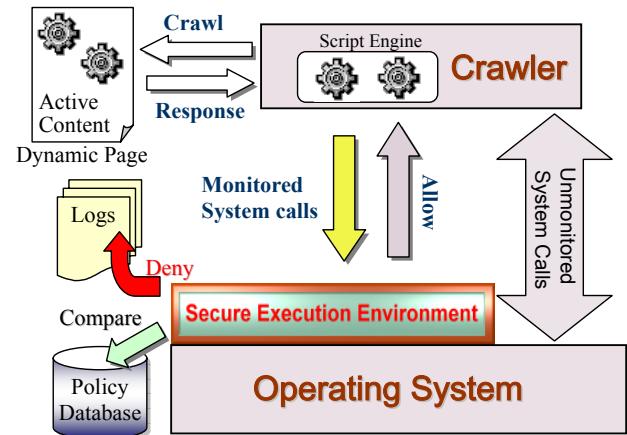


Figure 6. The Secure Execution Environment (SEE).

The SEE provides a) a self-protection mechanism to guard against malicious code, and b) a method to detect malicious code inserted into Web applications. One deficiency is that the mechanism only detects code that has already been inserted, and not the weaknesses of Web applications that make them vulnerable to attack. Detecting such vulnerabilities requires an off-line static analysis of Javascripts retrieved during the reverse engineering phase. We are still in the initial phase of designing and experimenting with this analytical procedure.

3. SYSTEM ARCHITECTURE AND IMPLEMENTATION DETAILS

Figure 7 depicts the entire WAVES system architecture, which we will briefly describe in this section.

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 them exhibit the same behaviors as browsers, they were equipped with IE's DOM parser and scripting engine. We chose IE's engines over others (e.g. Gecko [39] 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. Please note that a subset of the above events is triggered by Web application errors. An obvious 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, but more importantly, it logs the event and prepares corresponding entries generating an assessment report.

When designing the crawler, we looked at ways that HTML pages reveal the existence of 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" 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 [21].
8. Form submissions.

We established a sample site to test several commercial and academic crawlers, including Teleport [62], WebSphinx [38], Harvest [12], Larbin [56], Web-Glimpse [35], 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. As described in Section 2.4, 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 IKM's 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.

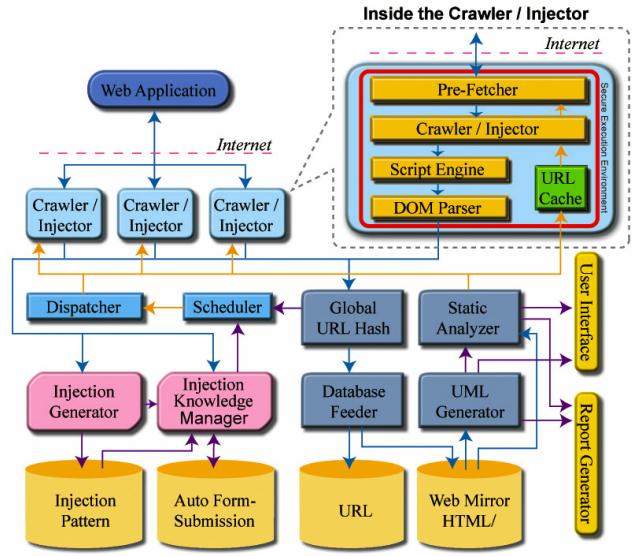


Figure 7. System architecture of WAVES.

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 [19] 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. Work is still being performed on the static analyzer and UML generator.

4. RELATED WORK

Offutt [41] surveyed Web managers and developers on quality process drivers and found that while time-to-market is still considered the most important quality criteria for traditional software, security is now very high on the list of concerns for Web application development. Though not specifically aimed at improving security attributes, there has been a recent burst of activity in developing methods and tools for Web application testing [9] [27] [48], analysis [48] [52], and reverse engineering [22] [23] [49] [50] [51] [63]. Many of these studies took black-box approaches to Web application analysis and reverse engineering. WAVES uses a similar process for identifying data

entry points, but also uses what we call a “complete crawling” mechanism to attempt more complete crawls. This is accomplished by three strategies—browser emulation, user event generation, and automated form completion. Similar efforts were made for the VeriWeb [9] project, which addresses the automated testing of dynamic Web applications. VeriWeb embeds Gecko [39] 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 [10] [29] [34] [46]. To automatically generate valid input data, VeriWeb uses Smart Profiles, which represents sets of user-specified attribute-value pairs. In contrast, WAVES incorporates a self-learning knowledge base.

Scott and Sharp [54] take a different approach to protecting against SQL injection and cross-site scripting attacks: a global input validation mechanism. They argue that Web application vulnerabilities are essentially unavoidable, meaning that security assurance needs to be “abstracted” to a higher level. However, to adapt this mechanism to a legacy Web application requires that rules be defined for every single data entry point—perhaps a difficult task for Web applications that have been developed over a long time period, since they often contain complicated structures with little documentation. It would be unusual for a Web manager to be familiar with all of the data entry points for a site with thousands of pages. Another protection approach, the <bigwig> project [14], also provides Web application input validation mechanisms. The mechanism is designed to automatically generate server- and client-side validation routines. However, it only works with Web applications developed with the <bigwig> language. In contrast, WAVES provides security assurance without requiring modifications to existing Web application architectures.

The authors of MOPS [18] and SPLint [24] have adopted a software-engineering approach to security assessment; however, they targeted traditional applications rather than Web applications. The Open Web Application Security Project (OWASP) [42] has launched a WebScarab [42] project aimed at developing a security assessment tool very similar to WAVES. Sanctum has recently incorporated routines to detect SQL injection vulnerabilities in Web applications into its *AppScan* [53]. Two other available commercial scanners include SPI Dynamics’ *WebInspect* [61] and Kavado’s *ScanDo* [32]. Reviews of these tools can be found in [4]. At the time of this writing, WebScarab has yet to be released, and no demo versions exist for the other scanners, therefore we were unable to compare their features with those in WAVES.

To expedite the reverse engineering and fault injection processes, the multi-threaded WAVES crawler performs parallel crawls. We adopted many of the ideas and strategies reviewed in [19] and [58] to construct fast, parallel crawlers. For the automated form completion task, we followed suggestions offered by Bergman [10] and Raghavan [46], but incorporated a more complex self-learning knowledge base.

Behavior monitoring has attracted research attention due to its potential to protect against unknown or polymorphic viruses [7] [8] [11] [13]. In addition to self-protection, we used behavior monitoring to detect malicious content before it reaches users. Furthermore, WAVES performs behavior stimulation to induce malicious behavior in the monitored components. In other words, it uses behavior monitoring for both reactive and proactive purposes.

We employed sandboxing technology to construct a self-contained SEE. Our SEE implementation is based on descriptions in [28] [31] [33]. In [31], a generic model is proposed for sandboxing downloaded components. Regarding the actual implementation, we had a choice between two open-source toolkits—Detours [28] and GSWTK [33]. We selected Detours because of its lighter weight. For a standard description of normal behaviors, we used BMSL [45] [55]. We compared our SEE with other commercial sandboxes, including Finjan’s *SurfinShield* [26], Aladin’s *ESafe* [1], and Pelican’s *SafTnet* [43] [44]. Surveys of commercially available sandboxes can be found in [3] and [65].

5. EXPERIMENTAL RESULTS

A snapshot of WAVES performing SQL injection is presented in Figure 8. We tested for thoroughness by comparing the number of pages retrieved by various crawlers. Teleport [62] proved to be the most thorough of a group of crawlers that included WebSphinx [38], Larbin [56], and Web-Glimpse [35]. This may be explained by Teleport’s incorporation of both HTML tag parsing and regular expression-matching mechanisms, as well as its ability to statically parse Javascripts and to generate simple form submission patterns for URL discovery.

On average, WAVES retrieved 28 percent more pages than Teleport when tested with a total of 14 sites (Figure 9). We attribute the discovery of the extra pages to WAVES’ script interpretation and automated form completion capabilities.

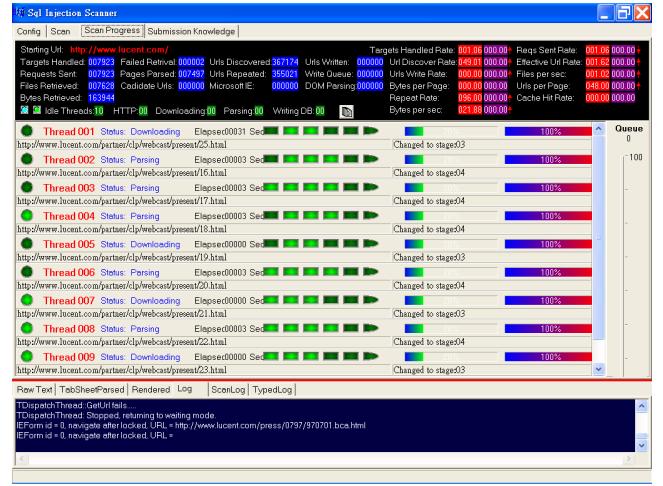


Figure 8. A snapshot of WAVES at work.

To test the injection algorithm, WAVES was configured to identify all forms of interest (i.e., those containing textboxes; see column 2 of Figure 9), to perform an NRE for each form, to fill in and submit the form, and to make a judgment on submission success based on the reply page and the previously retrieved NRE. WAVES creates detailed logs of the data used for each automated form completion, the resulting HTML page, and submission success judgments. In Figure 10 (produced from a manual inspection of the logs), $P(S)$ denotes the probability that the semantics of an input textbox have been successfully extracted; $P(C|S)$ denotes the conditional probability that a form completion is successful given that semantic extraction was successful; $P(C_L|S)$ denotes the same probability, but after a learning process in which the IKM expands its knowledge base; $P(N)$ denotes the probability of a successful NRE process; and $P(F|V,D)$ denotes the probability of false negatives given that a form is both

validated and defected (i.e., vulnerable). False negatives are induced when all of the following conditions are true: a) the form is defected (vulnerable); b) the form is validated; c) WAVES cannot correctly complete the form; and d) the NRE process fails, but WAVES is unable to recognize the failure. Therefore, a general definition of the probability of false negatives given that the form being tested enforces validation can be defined as $P(F|V,D) = (1-P(C|S) - P(C|X)) * (1-P(N))$, where $P(C|X)$ denotes the probability that form completion has succeeded given that semantic extraction failed. In our analysis, we used the pessimistic definition of $P(C|X) = 0$, meaning that we assumed zero probability of correctly filling a validated form whose semantics could not be extracted.

Site	Forms of Interest	Waves	Teleport	WAVES' Advantage
www.nai.com	52	14,589	11,562	21%
www.lucent.com	21	8,198	7,929	3%
www.trendmicro.com	70	5,781	2,939	49%
www.palm.com	43	4,459	4,531	-2%
www.verisign.org	9	4,389	3,069	30%
www.apache.org	5	3,598	3,062	15%
www.verisign.com	42	3,231	3,069	5%
www.uload.com	3	1,624	1,417	13%
www.cert.org	3	1,435	1,267	12%
www.maxtor.com	4	1,259	863	31%
www.mazda.com	1	1,030	356	65%
www.linuxjournal.com	7	871	167	81%
www.cadillac.com	2	673	598	11%
www.web500.com	3	564	237	58%

Figure 9. Crawling statistics for WAVES and Teleport.

As a part of our approach, both self-learning (to assist automated submissions) and NRE are used in order to decrease false negative rates when injecting a validated form. To evaluate these mechanisms, we define three probabilities derived from $P(F|V,D)$: $P(F_0|V,D)$, $P(F_L|V,D)$, and $P(F_{LN}|V,D)$. $P(F_0|V,D)$ denotes the probability of $P(F|V,D)$ when neither the self-learning nor the NRE algorithms are applied. $P(F_L|V,D)$ denotes the probability of $P(F|V,D)$ when the learning mode is enabled. $P(F_{LN}|V,D)$ denotes the probability when applying both learning and NRE. As Figure 10 shows, the $P(F|V,D)$ average decreased more than 5 percent (from the 18.76% of $P(F_0|V,D)$ to the 13.62% of $P(F_L|V,D)$)—in other words, during this experiment, the WAVES' learning mechanism decreased the rate of false negatives by 5 percent. An additional drop of 11 percent occurred between $P(F_L|V,D)$ and $P(F_{LN}|V,D)$ due to a contribution from the NRE algorithm. In total, WAVES' self-learning knowledge base and the NRE algorithm combined contributed to a 16 percent decrease in false negatives, to a final rate of 2.46 percent.

In order to use behavior monitoring for malicious script detection, the WAVES crawler was modified to accommodate IE version 5.5 instead of 6.0 because of the greater vulnerability of the older version. To incorporate the most recent version would mean that we could only detect new and unknown forms of attacks. Furthermore, the behavior monitoring process is also dependent upon the crawler's ability to simulate user-generated events as test cases, and IE versions older than 5.5 do not support event simulation functions.

Site	P(S)	P(C S)	P(C _L S)	P(N)	P($F_0 V,D$)	P($F_L V,D$)	P($F_{LN} V,D$)
NAI	18.69	80.32	81.93	70.58	19.68	18.07	05.31
Lucent	83.90	79.87	83.76	77.70	20.13	16.24	03.62
Trend Micro	90.72	78.52	84.04	98.60	21.48	15.96	00.22
Palm	43.56	88.63	92.20	100	11.37	07.80	0
Olympic	88.23	100	100	100	0	0	0
Apache	75.00	77.77	77.77	100	22.23	22.23	22.23
Verisign	89.93	86.06	95.27	93.02	13.94	04.73	00.33
Ulead	100	83.72	91.86	100	16.28	08.14	0
Cert	55.55	100	100	100	0	0	0
Maxtor	96.77	36.66	51.66	100	63.34	48.34	0
Mazda	100	100	100	100	0	0	0
Linux Journal	100	84.61	84.61	100	15.39	15.39	0
Cadillac	100	73.30	86.60	25.00	26.70	13.40	10.05
Web500	91.30	67.80	79.50	100	32.20	20.50	0
Average	80.93	81.13	86.06	90.99	18.76	13.62	02.46

Figure 10. Automated submission results.

SecurityGlobal.net classified the impacts of vulnerabilities discovered between April, 2001 and March, 2002 into 16 categories [57]. We believe the items on this list can be grouped into four general categories: 1) restricted resource access, 2) arbitrary command execution, 3) private information disclosure, and 4) denial of service (DoS). We gathered 26 working exploits that demonstrated impacts associated with the first three categories, and used them to create a site to test our behavior monitoring mechanism. For this test, WAVES was installed into an unpatched version of Windows 2000. Figure 11 lists the impact categories and observed detection ratios.

Class of Impact	Exploits	Detection Ratio
1) Restricted resource access	9	9/9
2) Arbitrary command execution	9	9/9
3) Private information disclosure	6	0/6
4) Denial of service (DOS)	2	0/2

Figure 11. Detection ratios for each class of impact.

WAVES successfully detected category 1 and 2 impacts. One reason for this high accuracy rate is that IE exhibited very regular behavior during the normal-behavior learning phase. The system calls that IE makes are fixed, as are the directories and files that it accesses; such clearly defined behavior makes it easier to detect malicious behavior. Our exploits that demonstrate category 3 impacts operate by taking advantage of certain design flaws of IE. By tricking IE into misinterpreting the origins of Javascripts, these exploits break the Same Origin Policy [37] [40] enforced by IE and steals user cookies. Since these design flaws leak application-specific data, they are more transparent to a SEE and are therefore more difficult to detect. This is reflected in our test using three commercial sandboxes—SurfinShield [26], ESafe [1], and SafNet [43] [44]. Similar to WAVES, none of the sandboxes was able to detect any of the six exploits of Category 3. As well as for impacts of Category 4, a more sophisticated mechanism must be implemented for detection, and is an area of our future research.

File Management	Process Management
CreateFile	CreateProcess
WriteFile	CreateProcessAsUser
CreateFileMapping	CreateProcessWithLogonW
Directory Management	OpenProcess
CreateDirectory	TerminateProcess
RemoveDirectory	Communication
SetCurrentDirectory	CreatePipe
Hook	CreateProcessWithLogonW
SetWindowsHookEx	Registry Access
System Information	RegSetValueEx
GetComputerName	RegOpenKeyEx
GetSystemDirectory	RegQueryValueEx
GetSystemInfo	User Profiles
GetSystemMetrics	GetAllUsersProfileDirectory
GetSystemWindowsDirectory	LoadUserProfile
GetUserName	GetProfilesDirectory
GetVersion	Windows Networking
GetWindowsDirectory	WNNetGetConnection
SetComputerName	Socket
SystemParametersInfo	Bind
	Listen

Figure 12. System calls intercepted by the SEE.

The SEE does not intercept all system calls. Doing so may allow the SEE to gather more information, but will also induce unacceptable overhead. Therefore, the set of intercepted system calls was carefully chosen to contain calls that IE does not normally make, but that malicious components needs to make. A list of intercepted system calls is given in Figure 12. The Detours interception module has a maximum penalty of 77 clock cycles per intercepted system call. Even for a slow CPU such as the Pentium Pro, this only amounts to approximately 15 μ s. Since IE does not call most intercepted calls after initialization, the interception mechanism costs little in terms of overhead. Greater overhead potential lies in the policy matching process that determines whether a call is legal by looking at its parameters. The regular behavior exhibited by IE resulted in only 33 rules being generated by the learning process. Since the rules (expressed in BMSL) are pre-compiled and stored in memory using a simple hash, matching call parameters against these rules cost little in terms of overhead.

```
For i:=1 to TotalElements do Begin
  If Assigned(Elements[i].onmouseover) then do Begin
    Event = Doc.CreateEvent();
    Doc.FireEvent(Elements[i], "onmouseover", Event);End;
End;
```

Figure 13. Our event-generation routine.

In addition, the event generation process was inexpensive in terms of CPU cost. Our experimental scans show that the index page of <http://www.lucent.com/> contained 635 DOM elements, 126 of which carried the onMouseOver event handler. In other words, the 126 elements execute a block of pre-assigned code whenever the user moves a mouse over the elements. The routine used to generate the onMouseOver event for all 126 elements is shown in Figure 13. For a 2 GHz Pentium IV, this routine took approximately 300 milliseconds.

Thus, we conclude that while successfully intercepting malicious code of category 1 and 2, the behavior monitoring mechanism was cost-effective and feasible. However, as more sophisticated strategies are used to detect category 3 and 4

impacts, larger overheads may be induced. Note that the event-generation routine contributes not only to behavior monitoring, but also to a more complete URL discovery.

6. CONCLUSION

Our proposed mechanisms for assessing Web application security were constructed from a software engineering approach. We designed a crawler interface that incorporates and mimics Web browser functions in order to test Web applications using real-world scenarios. During the first assessment phase, the crawler attempts to perform a complete reverse engineering process to identify all data entry points—possible points of attack—of a Web application. These entry points then serve as targets for a fault injection process in which malicious patterns are used to determine the most vulnerable points. We also proposed the NRE algorithm to eliminate false negatives and to allow for “deep injection.” In “deep injection”, the IKM formulates an invalid input pattern to retrieve a negative response page, then uses an automated form completion algorithm to formulate the most likely injection patterns. After sending the injection, WAVES analyzes the resulting pages using the NRE algorithm, which is simpler, yet more accurate than the LITE approach [46]. A summary of our contributions is presented in Figure 14.

Mechanisms	Based on	Facilitates
Self-learning knowledge base	Topic Model	1. Complete crawling 2. Deep injection
Negative response extraction (NRE)	Page similarity	Deep injection
Intelligent form parser	DOM object locality	Deep injection
Complete crawling	1.Javascript engine 2.DOM parser 3.Javascript event generation	Web application testing interface
Behavior monitoring	1.Self-training, anomaly detection model 2.Event simulation (Test case generation) 3.Detours (Sandboxing)	1. Self-protection 2. Cross-site scripting detection 3. Unknown malicious script detection
Behavior stimulation	Event simulation (Test case generation)	1. Behavior monitoring 2. Complete crawling

Figure 14. A summary of our contributions.

One contribution is an automated form submission algorithm that is used by both the crawler and IKM. Here we propose two strategies to assist this algorithm. To extract the semantics of a form’s input fields, we designed an “intelligent form parser” (similar to the one used in LITE [46]) that uses DOM object locality information to assist in automated form completion. However, our implementation is enhanced by incorporating a fully-functional DOM parser, as opposed to an approximate DOM parser used in [46]. To automatically provide semantically correct values for a form field, we propose a self-learning knowledge base based on the Topics model.

Finally, we added a secure execution environment (SEE) to the crawler in order to detect malicious scripts by means of

behavior monitoring. The crawler simulates user-generated events as test cases to produce more comprehensive behavior observations—a process that also aids in terms of crawl thoroughness. While functioning as a self-protection mechanism, the SEE also allows for the detection of both known and unknown malicious scripts.

As a testing platform, WAVES provides the following functions, most of which are commonly required for Web application security tests:

1. Identifying data entry points.
2. Extracting the syntax and semantics of an input field.
3. Generating potentially valid data for an input field.
4. Injecting malicious patterns on a selected input field.
5. Formatting and sending HTTP requests.
6. Analyzing HTTP replies.
7. Monitoring a browser's behavior as it executes active content delivered by a Web application.

As an interface between testing techniques and Web applications, WAVES can be used to conduct a wide variety of vulnerability tests, including cookie poisoning, parameter tampering, hidden field manipulation, input buffer overflow, session hijacking, and server misconfiguration—all of which would otherwise be difficult and time-consuming tasks.

7. REFERENCES

- [1] Aladdin Knowledge Systems. “eSafe Proactive Content Security.” <http://www.ealaddin.com/>
- [2] Apache. “Cross Site Scripting Info.” <http://httpd.apache.org/info/css-security/>
- [3] Armstrong, I. “Mobile Code Stakes its Claim.” In: SC Magazine, Cover Story, Nov 2000.
- [4] Auronen, L. “Tool-Based Approach to Assessing Web Application Security.” Helsinki University of Technology, Nov 2002.
- [5] W3C. “Document Object Model (DOM).” <http://www.w3.org/DOM/>
- [6] Anley Chris. “Advanced SQL Injection In SQL Server Applications.” An NGSSoftware Insight Security Research (NISR) Publication, 2002.
- [7] Apap, F., Honig, A., Herschkop, S. Eskin E., Stolfo S., “Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses.” In: Fifth International Symposium on Recent Advances in Intrusion Detection (Zurich, Switzerland, Oct 2002).
- [8] Balzer, R., “Assuring the safety of opening email attachments.” In: DARPA Information Survivability Conference & Exposition II, 2, 257-262, 2001.
- [9] Benedikt M., Freire J., Godefroid P., “VeriWeb: Automatically Testing Dynamic Web Sites.” In: Proceedings of the 11th International Conference on the World Wide Web (Honolulu, Hawaii, May 2002).
- [10] Bergman, M. K. “The Deep Web: Surfacing Hidden Value.” Deep Content Whitepaper, 2001.
- [11] Bernaschi, M., Gabrielli, E., Mancini, L.V., “Operating system enhancements to prevent the misuse of system calls.” In: Proceedings of the 7th ACM conference on Computer and communications security (Athens, Greece, 2000).
- [12] Bowman, C. M., Danzig, P., Hardy, D., Manber, U., Schwartz, M., Wessels, D. “Harvest: A Scalable, Customizable Discovery and Access System.” In: Technical Report CU-CS-732-94., Department of Computer Science, University of Colorado, Boulder, 1995.
- [13] Bowen, T., Segal, M., and Sekar, R. “On preventing intrusions by process behavior monitoring.” In: Eighth USENIX Security Symposium (Washington, D.C., Aug 1999).
- [14] Brabrand, C., Møller, A., M. I. “The <bigwig> project.” ACM Transactions on Internet Technology, 2(2), 79-114, May 2002.
- [15] CERT. “CERT® Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests.” <http://www.cgisecurity.com/articles/xss-faq.shtml>
- [16] Cesar Cerrudo. “Manipulating Microsoft SQL Server Using SQL Injection.” Whitepaper, 2002.
- [17] CGI Security. “The Cross Site Scripting FAQ.”
- [18] Chen, H., Wagner, D. “MOPS: an Infrastructure for Examining Security Properties of Software.” In: ACM conference on computer and communication security (Washington, D.C., Nov 2002).
- [19] Cho, J., Garcia-Molina, H. “Parallel Crawlers.” In: Proceedings of the 11th International Conference on the World Wide Web (Honolulu, Hawaii, May 2002), 124-135.
- [20] Curphey et. al. Mark. “A Guide to Building Secure Web Applications.” The Open Web Application Security Project, Sep 2002.
- [21] DHTML Central. HierMenus. <http://www.webreference.com/dhtml/hiermenus/>
- [22] Di Lucca, G.A.; Di Penta, M.; Antoniol, G.; Casazza, G. “An approach for reverse engineering of web-based applications.” In: Proceedings of the Eighth Working Conference on Reverse Engineering (Stuttgart, Germany, Oct 2001), 231-240.
- [23] Di Lucca, G.A., Fasolino, A.R., Pace, F., Tramontana, P., De Carlini, U. “WARE: a tool for the reverse engineering of web applications.” In: Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (Budapest, Hungary, Mar 2002), 241- 250.
- [24] Evans D., Larochelle, D. “Improving Security Using Extensible Lightweight Static Analysis.” In: IEEE Software, Jan 2002.
- [25] Finnigan, P., “SQL Injection and Oracle.” SecurityFocus, 2002. <http://online.securityfocus.com/infocus/1644>
- [26] Finjan Software. “Your Window of Vulnerability - Why Anti-Virus Isn't Enough.” <http://www.finjan.com/mcrc/overview.cfm>
- [27] Gold, R. “HttpUnit.” <http://httpunit.sourceforge.net/>
- [28] Hunt, G., Brubacher, D. “Detours: Binary Interception of Win32 Functions.” In: USENIX Technical Program - Windows NT Symposium 99, 1999.
- [29] Ipeirotis P., Gravano L., “Distributed Search over the Hidden Web: Hierarchical Database Sampling and Selection.” In: The 28th International Conference on Very Large Databases (Hong Kong, China, Aug 2002), 394-405.

- [30] Joshi, J., Aref, W., Ghafoor, A., Spafford, E. "Security Models for Web-Based Applications." *Communications of the ACM*, 44(2), 38-44, Feb 2001.
- [31] Kaiya, H., Kajiri, K. "Specifying runtime environments and functionalities of downloadable components under the sandbox model." In: *Proceedings of the International Symposium on Principles of Software Evolution* (Kanazawa, Japan, Nov 2000), 138-142.
- [32] KaVaDo. "Application-Layer Security: InterDo 2.1." KaVaDo Whitepaper, 2001.
- [33] Ko, C., Fraser, T., Badger, L., Kilpatrick, D. "Detecting and Countering System Intrusions Using Software Wrappers." In: *Proceedings of the 9th USENIX Security Symposium* (Denver, Colorado, Aug 2000).
- [34] Liddle, S., Embley, D., Scott, D., Yau, S.H., "Extracting Data Behind Web Forms." In: *Proceedings of the Workshop on Conceptual Modeling Approaches for e-Business* (Tampere, Finland, Oct 2002).
- [35] Manber, U., Smith, M., Gopal B., "WebGlimpse - Combining Browsing and Searching." In: *Proceedings of the USENIX 1997 Annual Technical Conference* (Anaheim, California, Jan, 1997).
- [36] Meer, H. "SQL Insertion," 2000.
- [37] Microsoft. "Scriptlet Security." Getting Started with Scriptlets, MSDN Library, 1997. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnindhtm/html/instantdhtmlscriptlets.asp>
- [38] Miller, R. C., Bharat, K. "SPHINX: A Framework for Creating Personal, Site-Specific Web Crawlers." In: *Proceedings of the 7th International World Wide Web Conference* (Brisbane, Australia, April 1998), 119-130.
- [39] Mozilla.org. "Mozilla Layout Engine." <http://www.mozilla.org/newlayout/>
- [40] Netscape. "JavaScript Security in Communicator 4.x." http://developer.netscape.com/docs/manuals/communicator/j_ssec/contents.htm#1023448
- [41] Offutt, J. "Quality Attributes of Web Software Applications." *IEEE Software*, 19(2), 25-32, Mar 2002.
- [42] OWASP. "WebScarab Project." <http://www.owasp.org/webscarab/>
- [43] Pelican Security Inc. "Active Content Security: Risks and Solutions." Pelican Security Whitepaper, 1999.
- [44] Privateer, P., "Making the Net Safe for eBusiness: Solving the Problem of Malicious Internet Mobile Code." In: *Proceedings of the eSolutions World 2000 Conference* (Philadelphia, Pennsylvania, Sep 2000).
- [45] Uppuluri, P., Sekar, R. "Experiences with Specification Based Intrusion Detection System." In: *Fourth International Symposium on Recent Advances in Intrusion Detection* (Davis, California, Oct. 2001).
- [46] Raghavan, S., Garcia-Molina, H. "Crawling the Hidden Web." In: *Proceedings of the 27th VLDB Conference* (Roma, Italy, Sep 2001), 129-138.
- [47] Raghavan, S., Garcia-Molina, H. "Crawling the Hidden Web." In: *Technical Report 2000-36*, Database Group, Computer Science Department, Stanford (Nov 2000).
- [48] Ricca, F., Tonella, P. "Analysis and Testing of Web Applications." In: *Proceedings of the 23rd IEEE International Conference on Software Engineering* (Toronto, Ontario, Canada, May 2001), 25 -34.
- [49] Ricca, F., Tonella, P., Baxter, I. D. "Restructuring Web Applications via Transformation Rules." *Information and Software Technology*, 44(13), 811-825, Oct 2002.
- [50] Ricca, F., Tonella, P. "Understanding and Restructuring Web Sites with ReWeb." *IEEE Multimedia*, 8(2), 40-51, Apr 2001.
- [51] Ricca, F., Tonella, P. "Web Application Slicing." In: *Proceedings of the IEEE International Conference on Software Maintenance* (Florence, Italy, Nov 2001), 148-157.
- [52] Ricca, F., Tonella, P. "Web Site Analysis: Structure and Evolution." In: *Proceedings of the IEEE International Conference on Software Maintenance* (San Jose, California, Oct 2000), 76-86.
- [53] Sanctum Inc. "Web Application Security Testing – AppScan 3.5." <http://www.sanctuminc.com>
- [54] Scott, D., Sharp, R. "Abstracting Application-Level Web Security." In: *The 11th International Conference on the World Wide Web* (Honolulu, Hawaii, May 2002), 396-407.
- [55] Sekar, R., Uppuluri, P., "Synthesizing Fast Intrusion Detection/Prevention Systems from High-Level Specifications." In: *USENIX Security Symposium*, 1999.
- [56] Sebastien@ailleret.com. "Larbin – A Multi-Purpose Web Crawler." <http://larbin.sourceforge.net/index-eng.html>
- [57] SecurityGlobal.net. Security Tracker Statistics. Apr 2002 – Mar 2002. <http://securitytracker.com/learn/statistics.html>
- [58] Shkapenyuk, V., Suel, T. "Design and Implementation of a High-Performance Distributed Web Crawler." In: *Proceedings of the 18th IEEE International Conference on Data Engineering* (San Jose, California, Feb 2002), 357-368.
- [59] SPI Dynamics. "Complete Web Application Security: Phase 1–Building Web Application Security into Your Development Process." SPI Dynamics Whitepaper, 2002.
- [60] SPI Dynamics. "SQL Injection: Are Your Web Applications Vulnerable." SPI Dynamics Whitepaper, 2002.
- [61] SPI Dynamics. "Web Application Security Assessment." SPI Dynamics Whitepaper, 2003.
- [62] Tennyson Maxwell Information Systems, Inc. "Teleport Webspiders." <http://www.tenmax.com/teleport/home.htm>
- [63] Tilley, S., Huang, S. "Evaluating the Reverse Engineering Capabilities of Web Tools for Understanding Site Content and Structure: A Case Study." In: *Proceedings of the 23rd IEEE International Conference on Software Engineering* (Toronto, Ontario, Canada, May 2001), 514-523.
- [64] United States Patent and Trademark Office. <http://www.uspto.gov/patft/>
- [65] Vibert, R., "AV Alternatives: Extending Scanner Range." In: *Information Security Magazine*, Feb 2001.
- [66] Voas, J., McGraw, G., "Software Fault Injection: Inoculating Programs against Errors." John Wiley & Sons, 47-48, New York, 1997.
- [67] WinMerge. "WinMerge: A visual text file differencing and merging tool for Win32 platforms." <http://winmerge.sourceforge.net>