Journal of Web Engineering, Vol.9, No.1 (2010) 1-24
© Rinton Press

PRACTICAL ELIMINATION OF EXTERNAL INTERACTION VULNERABILITIES IN WEB APPLICATIONS

JAMES MILLER TOAN HUYNH
University of Alberta, Canada
jm@ece.ualberta.ca huynh@ece.ualberta.ca

Received January 8, 2009
Revised September 3, 2009

External Interaction Vulnerabilities (EIVs) are currently the most common vulnerability for web applications. These vulnerabilities allow attackers to use vulnerable web applications as a vessel to transmit malicious code to external systems that interact with the web applications. The malicious code will modify the semantic content of the information sent to the external application. Current vulnerability detection approaches are black-box oriented and do not take advantage of the data flow information which is available in the source code. In this paper, we introduce a white-box approach called EIV analysis to eliminate web applications' vulnerabilities. This strategy allows investigators to accurately identify all inputs entering the web application and model the input as it reaches external systems acting as data sinks. The strategy is partially automated resulting in substantial effort savings when compared with common industrial approaches; while also providing superior performance in terms vulnerability detection. A case study using a commercial, currently deployed, mission-critical web application is presented to demonstrate the validity of these claims.

Key words: Security analysis, web applications, web security, case study
Communicated by: D. Lowe & O. Pastor

1 Introduction

Researchers have noticed a significant increase in attacks on web applications. In fact, web application vulnerabilities make up 69% of all Internet vulnerabilities [60]. A recent survey of web applications [29] reveals that 73% of the vulnerabilities found are External Interaction Vulnerabilities (EIVs). Many approaches designed to address EIVs have been proposed – these approaches are discussed in Section 6 – further confirming that EIVs are an extremely important class of vulnerabilities for web applications. Current approaches are either: application security [43] oriented, static analysis methods or black-box techniques. White-box approaches to detecting all EIVs are not common in the research literature; nor in industrial settings. In this paper, we introduce a practical white-box software development process that can help detect and eliminate web applications' EIVs. The approach builds a model based on the data flow of the application. The approach is significantly enhanced by computer-support software which automates the much of the "straightforward" components in the approach, allowing the security team to concentrate of the "creative" components in vulnerability detection. This partial automation strategy also makes the approach highly effective in terms of effort and maximizing the quantity of vulnerabilities discovered. The partial automation strategy utilizes two pre-existing tools (a crawler; and a capture replay tool) and two purpose-built proof-of-concept (by the authors) tools, Web Application Input Collection (WAIC) and Web Application Graph Generation (WAGG), to automate portions of the process for the web application in the case study. The strategy can be combined with previous approaches to further harden web applications against EIV related attacks.

The remaining sections of this paper are organized as follows: Section 2 defines the terms used in the paper. Section 3 provides an overview of the research problem. Section 4 introduces EIV analysis. Section 5 presents an industrial case study for the presented strategy. Section 6 provides an overview of current approaches aimed at addressing EIVs. Finally, Section 7 concludes the paper.

2 Definitions

Before the approach can be introduced, several terms require to be defined:

EIV – External Interaction Vulnerabilities, these vulnerabilities allow attackers to use vulnerable web applications as a vessel to transmit malicious code to an external system that can interact with the web application. The malicious code will modify the semantic content of the information sent to the external application. In other words, EIVs allow attackers to send systems commands that interact with a web application, rather than the actual web application itself. Currently, four interaction categories are defined:

- DBMS interaction – this is the interaction between the web application and external DBMS. An example of a DBMS interaction would be a web application calling a "query" function to send a SQL statement to a DBMS.
- Browser interaction – interactions in this category are between web applications and clients (typically a browser). An example of a browser interaction would be a web application sending a HTML encoded webpage to a web browser.
- OS/Filesystem interaction – this is the interaction between the web application and the filesytem or operating system. An example of this interaction type would be a web application reading a configuration file from the hard drive.
- Interpreter interaction – interactions in this category are between the web application and a programming language interpreter (usually the same language as the web application). An example of an interpreter interaction would be a web application calling "eval" to execute a programming statement.

Popular EIVs include SQL injections and cross site scripting vulnerabilities. A vulnerability is classified as an EIV if it has the following properties:

- A malicious input is required to initiate the attack.
- The malicious input is transmitted from the web application to an external system.
- The malicious input does not exploit the web application directly. For example, a buffer overflow vulnerability cannot be classified as an EIV because it attacks the application's input buffer directly without interacting with an external system.

External Systems – These are systems that the web application depends upon for its operation. For example, a shopping cart web application retrieves its product information from a DBMS, the external system.

Sitemap – The sitemap is a structure describing the model of accessible web pages from a web application. The definition and creation of the map is discussed in Section 4.A.

Web page – A web page is a document that can be accessed by a client (browser). Web pages can be static or dynamic. Web pages are identified by unique URLs because they correspond to a unique resource on the web server. Note that dynamic web pages can accept parameters; these parameters often appear after the webpage. For example, *http://www.example.com/index.asp?id=1*, *http://www.example .com/index.asp?id=2*, and *http://www.example.com/index.asp?id=2&type=1* are one web page, *http://www.example.com/index.asp*, which accepts two optional parameters, "id" and "type". While the parameters usually appear after the webpage's name, web servers can be configured to accept parameters as subdomains. For example, *http://name. example.com/* can be interpreted as

http://www.example.com/index.asp?username=name. Hence, to determine which part of the URL is the parameter, the web server's configuration must be examined.

CG – Contamination graphs are specialized DEF/USE graphs that trace input sources to their exit point. Definition of these graphs and the algorithm to create them is given in Section 4.C.

An input source's exit point – This is the location in the web application system where the input's value is passed to an external system.

3 Research Problem

As with many research problems, a precise specification of the problem of interest is difficult to comprehensively frame, and is only likely to be available after the problem has been completely solved. However, as stated in Section 1, our research does not seek to address all aspects of vulnerabilities; rather we are dealing with a specific problem which is framed with the following constraints or objectives:

- The work is only interested in web applications and EIVs. However, any solution should be applicable to all types of web applications and seek to eliminate all types of EIVs. As web applications become increasingly reliant on other external systems, such as other web services [12][3] or NXDs [10], new types of EIVs will emerge. For example, XPATH[a] is becoming increasingly popular technique for querying XML documents. A web application that uses XPATH can be vulnerable to XPATH injection, which is a type of EIV. Although the number of exploits based on XPATH injection vulnerabilities is currently small compared to XSS and SQL injection, this number will only increase as more and more web applications take advantage of XPATH as a method of retrieving data from XML documents. A solution that cannot support future or, currently obscure, EIV types will quickly become obsolete. Web application technology moves at an incredible pace. Within a few years, web applications have evolved from simple guestbooks and web counters which relied on flat-text files for data support to fully interactive office productivity suites that interact with enterprise third party systems such as Oracle DB. A solution that cannot keep pace with the evolving web applications would not be practical for industrial use.

- Any solution must support a wide range, including multiple versions, of external systems. This can be viewed as a large configuration problem, see Eaton and Memon [16] for work in this area. Web applications can interact with many different external systems. For example, one application may interact with Internet Explorer 6.5 and MySQL 3.23; another application may interact with Internet Explorer 5.5, Mozilla FireFox 1.5, SQLite 3.4.2 and Google Maps API 2.1. While similar, different versions of external systems will have different interfaces. These differences often cause highly vulnerable situations as systems commonly fail to correctly adapt to these evolving interfaces. For example, only Internet Explorer 6 SP1 and later support HTTP-Only cookies[b]. This is an extension to the Set-Cookie header that mitigates XSS attacks targeting information stored within cookies. However, not all IE versions support this extension, and hence, some IE versions have a much higher risk of being vulnerable to XSS attacks targeting cookies than other IE versions. Furthermore, Mozilla FireFox 2.0.0.4 and lower only support HTTP-Only through an extension. Hence, the same version of FireFox (for example, 2.0.0.4) can have different risk levels, with regard to XSS attacks targeting cookies, depending on whether the HTTP-only cookie extension is enabled.

- Any solution must be language-independent. This is important as web applications utilize a wide range of scripting/programming languages (Java, Visual Basic, PHP, Perl, C#,

[a] http://www.w3.org/TR/xpath
[b] http://msdn2.microsoft.com/en-us/library/ms533046.aspx

Python, JavaScript, Ruby, Cold Fusion, etc.), which support a variety of different programming paradigms and styles. Furthermore, many web applications, such as AJAX enabled applications, utilize more than one scripting/programming language. Hence, any solution that can only support a single scripting/programming language would not be usable against these multi-language applications. This objective becomes more important as AJAX enabled web applications, such as Google Docs & Spreadsheets, and Mashups[c], which combines multiple web APIs in one hybrid web application, become more popular.

- Any solution must be applicable to current industrial strength web applications. Apart from the constraints stated above, this constraint is not too demanding. Current web applications are relatively small in scale (a recent survey on open source web applications show that a large number of these systems, range in size from 4 to 40 KLOC [29]); and hence, many of the restrictions placed by ultra-large scale systems are not of great concern here.
- Any solution must be "practical" in an industrial sense. Industrialists often express their frustration that many exciting pieces of software research are not applicable in their context. Research which requires large-scale retraining or complete redefinitions of their life-cycles are often considered by industrialists as "impractical". Hence, we seek solutions which can be viewed as an incremental development of most life-cycles; and solutions which can be utilized by many practitioners with minimal additional training or with on the job training. This latter approach is used in our case study.

In summary, the research problem can be viewed as a two-level problem. The lower-level problem is to find all EIVs that exist within a web application. The higher-level component, which generalizes the lower-level problem to cover all web applications, can be viewed as a large configuration space (CS): L × ET × NE, where L is the set of scripting/programming languages used to build web applications, ET is the set of EIV Types, and NE is the set of EIVs. Furthermore, NE is defined as ES × VES where ES is the set of different external systems and VES is the set of versions of these external systems.

Ideally, a solution should be fully automated. However, given the objectives and the large CS, an automated solution to solving this research problem is difficult to obtain. Although many automated approaches exist, all of them do not meet all objectives presented. For example, automated vulnerability scanners such as Secubat [33] cannot parse AJAX enabled web applications. Automated EIV detection systems often only detect the (second) most popular type of EIV – SQL injection [29]. Furthermore, these systems often require modification to the runtime engine of the supported scripting language; while other systems require additional hardware infrastructure to operate. This implies that these solutions will not be flexible enough to adapt with evolving web technologies.

4 External Interaction Vulnerability Analysis

The proposed strategy can be thought of as a white-box technique [47]; EIV analysis is performed using the following steps; these steps are further discussed in sections 4.A – 4.D:

1. Create a sitemap for the web application.
2. Identify all input sources.
3. Create contamination graphs.
4. Test the contamination graphs until a coverage criterion is met.

Although the CG generation step of EIV analysis is similar to static analysis, the two approaches are not the same. CG generation is just one of the four steps required for EIV analysis. With EIV analysis, CGs are a resource that security practitioners can utilize to uncover EIVs,

[c] http://www.ibm.com/developerworks/xml/library/x-mashups.html

whereas static analysis would simply presents the CGs without any further instructions on how these results should be handled. Section 4.C. discusses the difference between CGs and DEF/USE approaches used in traditional data flow analysis approaches. Hence, EIV analysis is most appropriately classified as specialized data flow testing that concentrates on tainted data flows. However, this approach is not a dynamic taint analysis approach as proposed by other researchers [22][53][71]. It does not contain a runtime component that monitors the application's memory for tainted values. In fact, dynamic taint analysis approaches often require modifications to the runtime platform or extra software which can complicate configuration and reduce performance.

4.1 Creating the Sitemap

The sitemap is a critical part of EIV analysis because it allows the security practitioner to identify path executions for EIV analysis. A sitemap is a set of directed graphs that represents a model of all accessible web pages from a web application. Each of these graphs contains a set of edges and nodes. More formally the sitemap is defined as S = {G} where:

- G = <N, E> where
    - N is a set of nodes representing the web pages. Each $n \in N$ represents a web page accessible by the client.
    - E is a set of directed edges. Each $e \in E$ between node n_1 and n_2 shows that n_2 is reachable from n_1.

Although the sitemap can be generated manually, the process is labour intensive and not very practical; hence a web crawler [11][24][45] is used to speed up the process. Specialized crawlers that can handle dynamic contents [54], or site specific pages [44], exist. A practitioner can use a crawler that can handle dynamic content to help create the site map for the web application under investigation; however, because crawlers can only follow web pages through links or forms, the practitioner's intervention is required in order to generate a complete sitemap. However, crawlers can only access web pages that are referenced from other pages; hence, if a web page is "hidden", no other web pages link or refer to it, then it cannot be accessed by the crawler. In order to create complete sitemaps, the number of web pages crawled needs to equal the total number of web pages for the web application. For example, consider a web application that has eight web pages: index, $normal_2$…$normal_5$, indexadmin, $admin_2$ and $admin_3$. These eight web pages comprise two distinct sections:

- One section is accessible to normal users. This section contains five web pages called index, $normal_2$…$normal_5$.
- Administrators can only access another section. This section contains 3 web pages called indexadmin, $admin_2$, and $admin_3$.

These two sections are separated and do not cross-reference each other; hence, the crawler needs to be executed twice.

The crawler must be configured to exclude pages not belonging to the web application, usually by restricting the crawling operation to a single domain or a directory of a web site. Crawling operation should not be limited to a single IP if the web application is hosted on multiple servers because the IP addresses for these servers are different. Although the IP addresses for these servers are different, the domain remains the same. For example, Amazon.com uses several servers to power its e-commerce application, but all the servers are under the Amazon.com domain.

4.2 Inputs

Inputs for web applications come from many sources [64] including clients (browsers). Black-box techniques for web applications primarily concentrate on this source of input ([50][61]); however, investigating this source alone is insufficient. Many web applications communicate with external applications to perform required tasks. Inputs from these external applications cannot be trusted and need to be examined to reveal all possible security faults. For example, Fig. 4.B.1 shows a

sequence diagram of a simplified interaction between a client and a search engine. The client sends a request to the search engine web application; the search engine then parses this request, creates an SQL statement and sends it to a DBMS. Once the results are retrieved from the DBMS, the search engine builds a HTML page and returns this page to the client. This interaction sequence has two input sources, one from the client (the search query that the user sends) and the other from the DBMS (the results that the DBMS returns). Code segments using these inputs have potential vulnerabilities associated with them. If the search engine does not parse the input from the DBMS then an attacker may compromise the DBMS and insert a JavaScript payload, which the search engine will return to the client after a search request. In this scenario, the search engine is vulnerable to a stored XSS [52] attack. Hence, if the security practitioner only examines the input from the client, the stored XSS vulnerability from the second input source will not be revealed until the product is released.

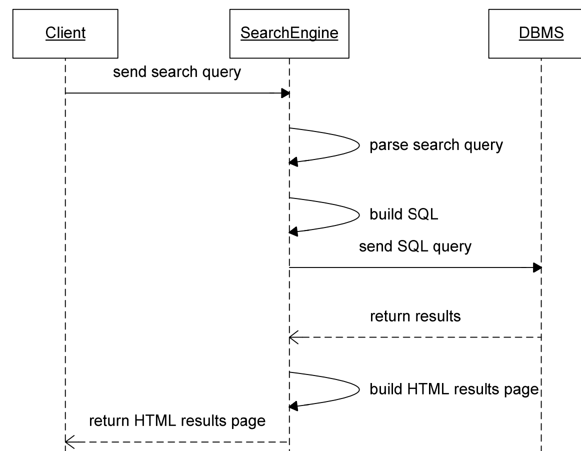

Fig. 4.B.1. A search sequence

4.2.1 Input Classification

Ideally, all inputs should be examined; however, software development companies have time and budget constraints limiting the amount of testing. To aid with the selection of inputs to be investigated, inputs are classified into two types:

- Inter-organization inputs – these are input values from unknown sources.
- Intra-organization inputs – These are input values entered by known and believed to be trusted sources (administrators, webmasters, employees, etc.). For example, a news article entered into a CMS (Content Management System) by an editor is considered as an Intra-organization input; whereas a comment posted by an anonymous user to a news item is considered an inter-organization input.

Inter-organization inputs should have a higher priority because, on average, they represent greater risks to the system. Intra-organizational inputs should still be examined because attacks can still happen under specific circumstances. For example, a spiteful employee can intentionally attack the system, or an attacker can access an employee's username and password through phishing [14][51] or other social engineering techniques [20], and use the account as a mechanism to inject payloads.

Multiple inputs from the same source do not imply that they are of the same type. For example, consider a simple e-commerce system that can display a product's name, price, and user reviews. All three data are retrieved from three columns within a DBMS. However, an employee enters the product's name and price, while web visitors, who claim to have used the product, enter the user

reviews. In this scenario, although the inputs are from the same source (DBMS), the two columns containing the values entered by the employee (product's name and price) are considered as intra-organization inputs while the other column containing the user reviews are considered as inter-organization inputs. Under constraints, a security practitioner can prioritize and examine the e-commerce system's ability to verify and validate the inter-organization inputs (reviews submitted by users) first, before investigating the intra-organization inputs (product's name and price).

4.2.2 Input Identification

To identify all inputs, the security practitioner will need to have access to the source code. Each input that enters the system can be stored in multiple variables; these variables are the starting nodes for the contamination graph. To allow automation, a formal model for the inputs is created.

Each source code file can have zero or more inputs. An input unit (IU) = (S, T, N) is used to describe inputs where:

1. S = The source of the input. This specifies which external system supplies the input value.
2. T = The type of the input. $T \in$ {Inter-organization, Intra-organization}.
3. N = an ordered pair (v, l) where v is a variable that stores the input value and l is the location where the variable is defined. In other words, *(v,l) = N iff (DEF(v) := input value* $\wedge$ *LOC(v) = l)* where DEF(v) is the statement that defines v and LOC(v) is the location where v can be located which is the line number and the filename.

To introduce the algorithm, several variables and functions need to be defined:

- Let I be a set of IUs.
- Let F be a set of source code files.
- Readlines(f) is a function that returns a set of statements in file f.
- Source(v) is a function that returns the source of variable v. That is, it returns the external system that sends a value to the application under test.
- Type(v) is a function that returns the type of variable v.

The following algorithm derives all inputs for a web application. The algorithm requires the set of source code files to be known. The inputs generated from the algorithm are stored in I:

```
1.  I = {};
2.  ∀(f ∈ F) {
3.     ST := Readlines(f);
4.     ∀ (st ∈ ST) {
5.        if (st = (DEF(a) := input)) {
6.           I := I ∪ (Source(input),Type(input),(a, LOC(a)));
7.        }
8.     }
9.  }
```

The algorithm parses all source code files to search for statements where variables are initialized from an input.

4.3 Contamination Graphs

Contamination graphs (CGs) are a critical component of the EIV analysis process. These graphs will allow the tester to design test cases that can reveal potential EIVs for the web application under investigation. CGs are a variation on DEF/USE graphs used in data flow testing [19][23] [36][55]. Liu et. al. [39] has extended the technique for web applications; however, the approach concentrates on inter and intra-procedure data flows and not intersystem data flows which are critical to the EIV analysis process. Hence, this paper introduces an intersystem contamination graph (CG), which describes the path an input value travels upon entering the system under

investigation to reach various external systems. The graph is similar to the taint variable concept [28].

A CG differs from a traditional DEF/USE graph because it does not contain all statements within a program. Its nodes only contain DEF/USE statements related to the input that initializes the graph. The CG's purpose is to trace the path of an input value from its entry point to its various exit points (i.e. statements that send the input value to external systems). A CG graph is formally defined as $CG = < N, E, N_e >$ where:

- N is a set of nodes representing all statements containing either a DEF or USE instruction.
- E is a set of directed edges representing the data flow between statements. Each $e \in E$, between nodes n_1 and n_2, shows that the flow of the tainted data moves from n_1 to n_2.
- $N_e \subseteq N$ is a set of exit nodes where the input values exit the system and are transmitted to external systems.

The security practitioner needs to create a CG graph for each input identified in Section 4.B. Before the algorithm used to create the CG is introduced, several variables and functions need to be defined:

- V is a set of variable names.
- getLoc(x) returns the location (loc) of the variable defined in the input unit x. loc is comprised of a line number and a filename. Section 4.B discussed the model for the input.
- getVariable(x) returns the variable name (v) of the variable defined in the input x.
- getStatement(loc) returns the statement at location loc.
- getNextUse(V,loc) returns the location of the next statement that contains a USE instruction for one of the variables in the set V starting from the location loc. If a statement cannot be found before the end of the program is reached, then getNextUse(V,loc) returns EOP (End of Program) stating that no additional statements can be found.
- getPrevUse(V,loc) returns the location of the previous statement containing a USE instruction for one of the variables in the set V from the location loc. If the current location is the first USE within a branch, then it returns the last encountered USE instruction before the branch.
- EXITPOINT is the statement that sends the value of the variable stored in the input unit to an external system. For example, a system call to the print function is an exit point if the value of the variable stored in the input unit is passed into the print function.

The following algorithm can be used to produce a CG for each input unit:

```
1.  create_CG(inputUnit) {
2.     N := {};
3.     E := {};
4.     Ne := {};
5.     V := {}
6.     loc := getLoc(inputUnit);
7.     var := getVariable(inputUnit);
8.     V := var ∪ V;
9.     N := loc ∪ N;
10.    loc := getNextUse(V,loc);
11.    if (loc != EOP) {
12.       N := loc ∪ N;
13.       E := (getPrevUse(V,loc)→loc) ∪ E;
14.       st := getStatement(loc);
```

```
15.         if (st = DEF(w) {
16.            V := w ∪ V;
17.         }
18.         if (st = EXITPOINT) {
19.            N_e := loc ∪ N_e;
20.         }
21.      } else {
22.         Go to 6;
23.      }
24.      return <N,E,N_e>;
25.  }
```

The algorithm starts at the location where the input enters the system. It then searches for all statements utilizing the value and all variables assigned with the value. Finally, all exit points are then identified and flagged accordingly. Hence, using the above algorithm, a complete data flow path, from the entrance to the exit points, for the input is created.

The complexity analysis for the algorithm begins with the loop (lines 6 to 22). This loop has the order of O(n) because it goes through each statement in the program's source code. Inside the loop, there are two searches being performed (line 13 and line 14). These are searches that scan the source file to find a matching line of code. A conservative estimate of these searches is O(n). This is true because the search algorithm with the most expensive running time is the linear search algorithm, and this algorithm has a complexity of O(n). With the algorithm having O(n) for both the loop and the functions within the loop, the overall complexity of the algorithm is estimated to be $O(n^2)$.

The algorithm has a well-known limitation – it is unable to follow information through implicit flows [13] if constants are used to initialize the variable in the flow. For example, *if (INPUT == 1) then x:='a'; else x:='y'*. However, we are unaware of any approach which adequately resolves this limitation. Further, while implicit flows can exist in any program, their frequency of occurrence is not well known. In addition, a recent survey which explored the causes of vulnerabilities by tracing through the source code of the compromised systems, found that none of the paths through the compromised systems which lead to vulnerabilities contained implicit flows [26]. This survey covered 20 web applications implemented in five difference programming languages. The source code for the web applications ranged from 1.4 to 30 KLOC. The systems contained 138 vulnerabilities, and the survey classified 100 of the vulnerabilities as EIVs. Hence, while this theoretical limitation exists in our testing process, we find no compelling empirical argument that the limitation causes the process to be inadequate on a regular basis when applied to realistic systems.

4.3.1 AN EXAMPLE CG

In this section, an example program is used to demonstrate the creation of a CG. The following is a highly simplified search application:

```
1. var search_keyword = gets();
2. var sql_query = "SELECT text FROM contents WHERE text like
   '%"+search_keyword+"'";
3. var results = execute_query(sql_query);
4. if (results != EMPTY) {
5.    print "Your keyword: "+search_keyword+" returned the following results:";
6.    print results;
7. } else {
8.    print "Your keyword: "+search_keyword+" returned no results.";
9. }
```

This application contains two input sources, one from the user (line 1) and one from a DBMS (line 3); hence two CGs are required to be generated. Fig. 4.C.1 and 4.C.2 show the CGs created for these two inputs:

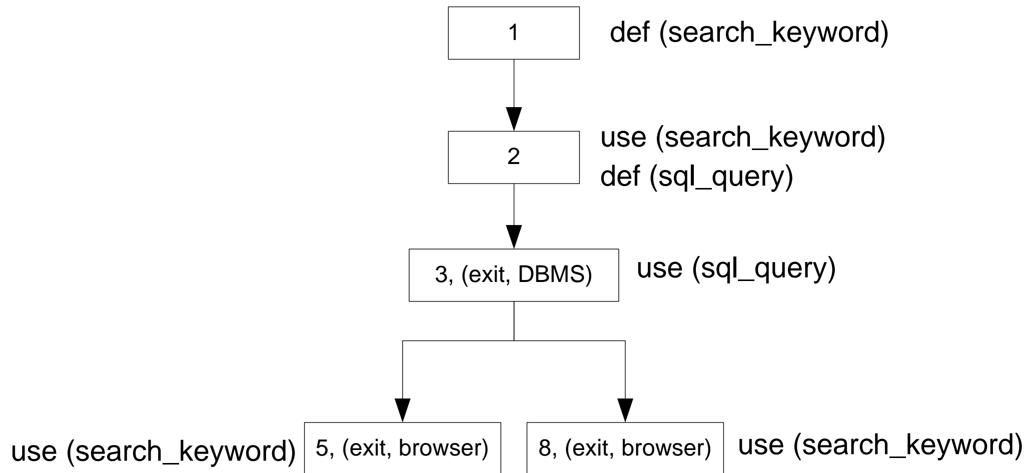

Fig. 4.C.1 A CG for search_keyword

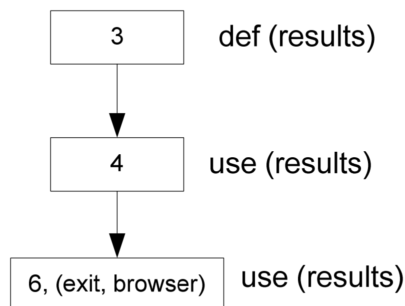

Fig. 4.C.2. A CG for results

These four exit points show that four possible EIVs exist in the system; however, the amount of testing needed to determine whether the EIVs exist is not known. In the next section, a coverage criterion will be defined. This criterion will help security practitioners determine how much testing on these graphs is considered sufficient.

4.4 Test Data Coverage, Selection, and Execution

4.4.1 Coverage Criterion

A number of detailed path coverage criteria for data flow testing have been proposed [26][36][49][69]. All proposed criteria aid the tester in selecting the most effective paths in a DEF/USE graph; however, because EIV analysis only concentrates on revealing one class of fault, a more specialized criterion is required. A path is a set of edges of the CG that demonstrates how node B can be reached from node A. Formally, the coverage criterion for EIV analysis is defined as:

- Let P be a set of paths to be tested for a CG.
- Let E and N_e be a set of edges and exit nodes for the CG respectively.
- P satisfies the coverage criterion for EIV analysis if
  - $\forall n \in N_e$, $\exists p \in P$ such that n is the last node in p.
  - $\forall e \in E$, $\exists p \in P$ such that e is included in p.

Although HTTP [17] is a stateless protocol, web applications are not usually stateless; they can be stateful by using session management mechanisms such as cookies [35]. Attempting to access a web page while not in the right state commonly results in an error. Therefore, to reach the first node of a path to begin testing, the security practitioner needs to examine the sitemap (created in Section 4.A) and determine the path to reach the web page that allows the first node to be accessed.

4.4.2 Test Data Selection

Test data have to be carefully selected to cater to each specific exit point because each external system interprets the information differently. Input data, when passed to external systems, are categorized into two types by these external systems:

- Reserved words/characters – These are words and characters that have special meanings; they are interpreted and executed by the external systems.
- Data – The data can be classified into various data types such as String, Integer, etc.

Only reserved words/characters can modify the semantic of information passed from a web application to an external system. Hence, the security practitioner needs to select data that can satisfy one requirement:

- The data has to cause the external system to interpret the data as reserved words/characters rather than data.

Therefore the security practitioner has to examine the external system's documentations to determine how to force data to become reserved words/characters. For example, let's assume that the DBMS in Fig. 4.C.1 is MySQL [63]. Upon reviewing the MySQL's documentation, the practitioner may decide that if the data is not enclosed in single quotes (such as 'data') and it matches one of the reserved words/characters then MySQL will treat the data as reserved words/characters. Hence, the practitioner can simply use three test cases to test for the path leading to the MySQL exit point:

1. The data is not enclosed in single quotes, it can simply be any reserved word/character such as *SELECT*.
2. Escape the enclosure before injecting a reserved word or character, is *' SELECT*. The single quote before *SELECT* forces the data, enclosed in single quotes, to become *'' SELECT '*which means that *SELECT* is now treated as a reserved word/character.
3. To specify special characters in the data, the MySQL manual [66] states that MySQL recognizes several escape sequences; these sequences start with the backslash character \. Table 4.D.1 displays these escape characters. The table shows that a single quote character can be escaped using \'; if the web application inserts the escape character before the single quote character then the character loses its special meaning. Hence, the third test case needs to escape the escape character (Table 4.D.1).

The above data is only applicable for paths that do not contain any nodes within a branch, or if there are nodes within a branch, then the branching condition is not dependent upon the input under test. If the path contains nodes that are within a branch and the branching condition is dependent upon the test input, the practitioner needs to modify the test data for this input to satisfy the coverage criteria.

Table 4.D.1. Escape Sequences for MySQL

Escape Sequence	Character Interpreted
\0	An ASCII 0 (NUL) character
\’	A single quote (‘) character.
\”	A double quote (“) character.
\b	A backspace character.
\n	A newline (linefeed) character.
\r	A carriage return character.
\t	A tab character.
\Z	ASCII 26 (Control-Z).
\\	A backslash (\) character.
\%	A percent (%) character.
_	An underscore (_) character.

5 Case Study

A case study on a web application was performed in order to determine the fault detection capability and efficiency of the proposed approach. The application used for this case study is a commercial application, which was initially released on April 4th, 2004. The application is a powerful search engine that allows users to search for the latest product specification data from thousands of international standards. The web application has many users around the world; the users come from a wide variety of organizations from defense departments to automobile manufacturers. The application is a typical 3-tier web application, specifically using Internet Explorer, Apache+PHP and MySQL [68] on each tier. The web application contains ~25 KLOC. It has received eight revisions; these revisions added new features and corrected many bugs and vulnerabilities. The first six revisions were corrective maintenance [30] and were released in the application’s first 18 months of service. Revision six involved a detailed security review [25][38]; the security review involved the following steps:

1. All web pages of the web application were visited and parsed for inputs.
2. These inputs were then used in a penetration test.
3. The source code containing vulnerable inputs were reviewed and guards were either added or modified.
4. Steps 2 and 3 were repeated until the inputs were considered to be safe from EIV attacks.

The organization revealed that the security review took 24 person-hours to complete. The bug-tracking database used by the development team [15] reveals 68 EIVs were found and corrected for revision six. The remaining two revisions were adaptive and perfective maintenance with minor corrective maintenance (with no new EIVs were discovered) which suggests that the application is now stable; this status was confirmed by the developers of the application. For the case study, the source code for revision five was retrieved and investigated using the testing approach proposed. In order to provide a clear reference, all EIVs reported in the bug-tracking database were verified against revision five. Any EIVs that could not be replicated for revision five were discarded because they were introduced in later revisions with the addition of new features. Because no new EIVs have been discovered after the sixth revision, the total of confirmed EIVs for revision five is 68. One security practitioner was selected to perform the case study using the following steps described above.

5.1 Drawing the Application's Sitemap

To create the sitemap, the practitioner first examined the source files. Then a crawler (REL Link Checker)[d] was used to identify the majority of the web pages. The crawler used was not able to identify web pages linked using JavaScript [18] code; hence, the practitioner manually generated sections of the sitemap that were inaccessible to the crawler. The completed sitemap for the application took 1 hour to create, and is shown in Fig. 5.A.1.

Because of the proprietary nature of the application, all names of the web pages are removed. Web pages 6-17 are pop up pages, and required manual mapping by the practitioner. Because they are pop up pages, they don't have a navigation method to return to the previous page. This application implements a state check algorithm, so any user attempting to access a web page directly using its URL will receive an error message. Hence, to access a web page the user needs to follow one of the paths outlined in Fig. 5.A.1.

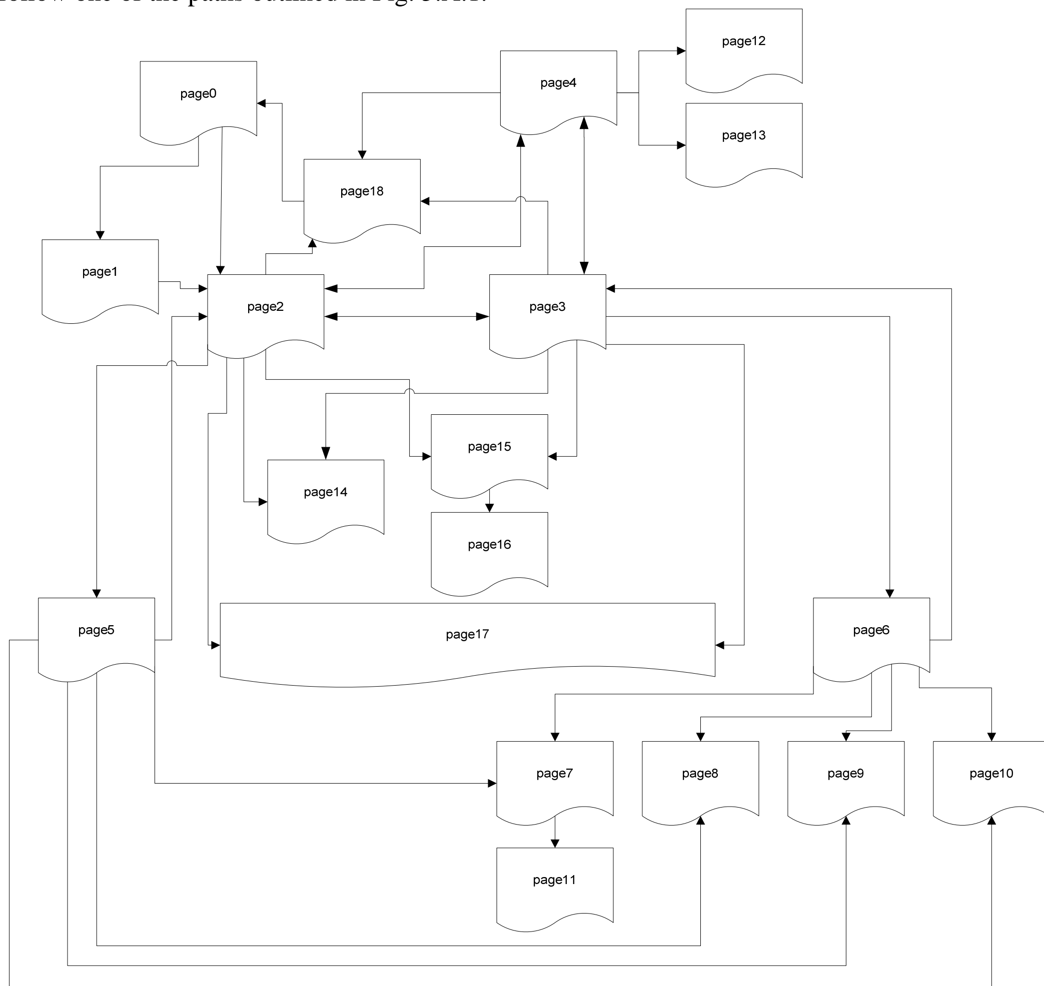

Fig. 5.A.1. The sitemap of the application

5.2 Identifying the Application's Inputs

Inputs for the web application originate from two sources: the client (browser) and the MySQL

[d] http://www.relsoftware.com/rlc/

database. The application is configured with *register_globals = off* [58]; hence, inputs originating from the client can be detected through the usage of super global arrays [1] within programming statements. To identify inputs from the MySQL database, the practitioner hosted several discussions with the developers of the web application. These discussions revealed that data from the MySQL database was retrieved using two function calls: *mysql_fetch_array* or *mysql_insert_id*.

Table 5.B.1. Number of inputs and their sources

Input source	Number of inputs
Client/Browser	96
MySQL	545
Total	641

Using the algorithm provided in Section 4.B, the Web Application Input Collection (WAIC) tool was created to aid security practitioners with this step. The tool automatically parses the source files of the web application and outputs all inter and intra organization inputs. WAIC's output contains the input type, the file, location and the input name of each identified input. The tool required 30 minutes to parse all the source files for inputs using an Athlon X2 3800 CPU with 2GB of RAM.

WAIC identified 641 inputs for the application. Table 5.B.1 displays the total number of inputs found and their sources. Personnel from the organization enter all of the database items; hence these items were initially considered as intra-organization type. To further verify this, each input was carefully examined using the available design and SRS documents. Table 5.B.2 shows the results from the examination.

Table 5.B.2 Input types

Input source	Input type	Number of inputs
Client/Browser	Inter-organization	96
	Intra-organization	0
MySQL	Inter-organization	1
	Intra-organization	544

The examination identified one inter-organization input within the database, a field that allows the customer to customize one of the display features.

5.3 Creating the CGs and Choosing Test Data

5.3.1 Creating the CGs

To create the CGs, the Web Application Graph Generation (WAGG) tool was created based on the algorithm provided above. WAGG accepts the output of WAIC as its inputs. WAGG allows security practitioners to automatically generate all CGs associated with each input identified by WAIC for the web application under test. The majority of the CGs (99%) are very simple and contain just one exit point per graph; the graphs also do not span across more than three source files. Each line represents a node and contains a node id, previous node id, source file, line number, any DEF/USE information, and the external system if it's an exit node.

Graphs for intra-organization inputs were extremely simple, involving only one source file; hence, only three hours were required to generate all of the graphs for the intra-organization inputs. Four hours were used to create the graphs for inter-organization inputs because they were slightly more "complex". Fig. 5.C.1 displays the most "complex" CG that WAGG produced. Each node is labeled with the source filename, followed by the line number in brackets. Once again, filenames are obscured to ensure confidentiality. Nodes containing exit points are labeled with the source filename, followed by the line number and the name of the external system.

This graph shows that the input value, stored in $login, was indirectly passed to an external

system eight times; the original variable used to store the input value was never passed to the external system. Instead, the original variable was used to define other variables, such as $username, $query and $_SESSION['username'], which are sent to the external system. If the practitioner simply examines the original variable at the exit points, he would not be able to perform a comprehensive test of the system.

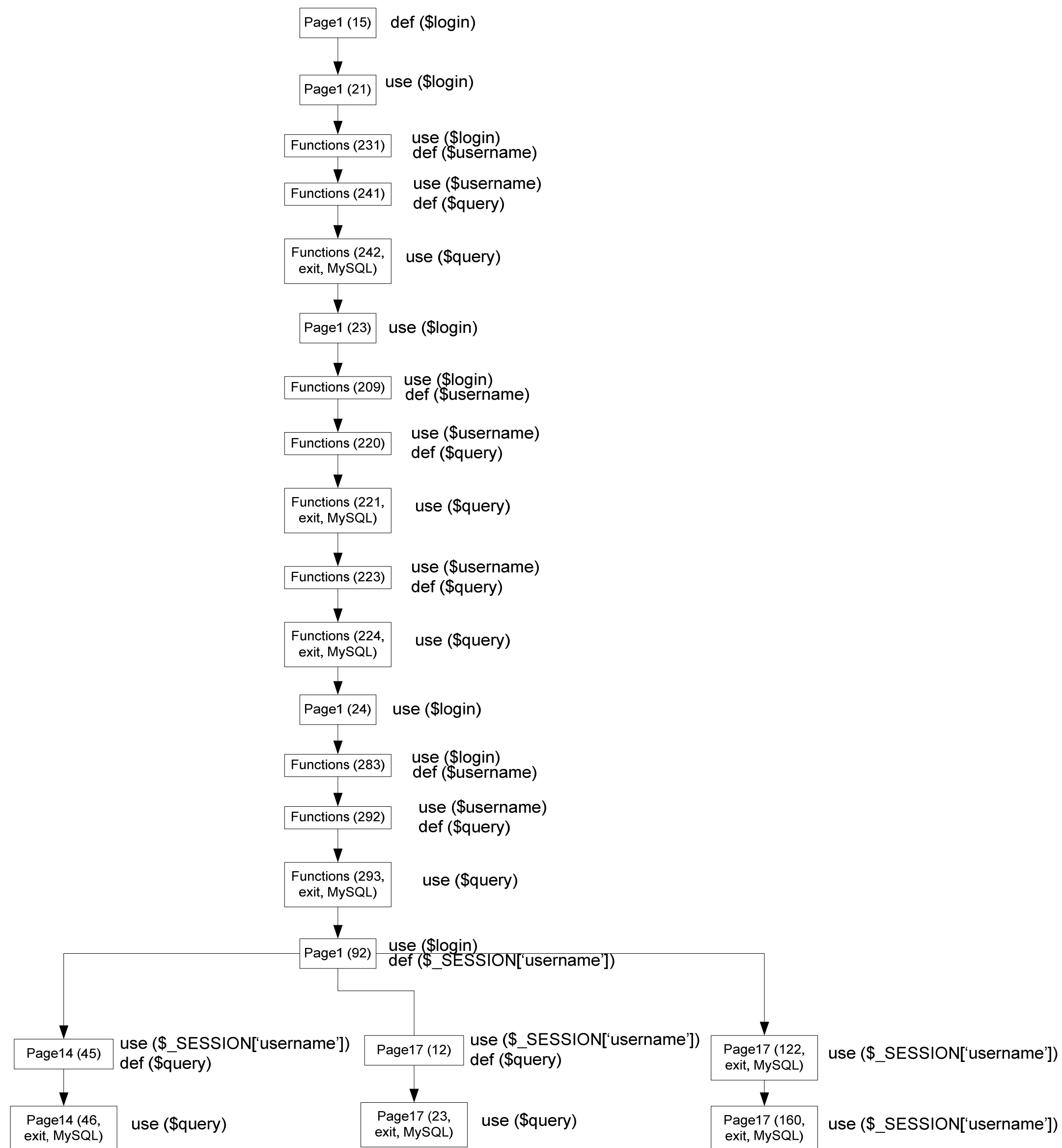

Fig. 5.C.1. A sample CG for the application under test

To satisfy the coverage criterion for this CG, the practitioner needs to cover all unique paths leading to the exit points. By examining the CG, eight paths need to be traversed:

{Page1(15)→Functions(242);Page1(15)→ Functions(221); Page1(15) → Functions(224); Page1(15)→Functions(293);Page1(15)→Page14(46);Page1(15)→Page17(23); Page1(15) → Page17(122); Page1(15) → Page17(160)}

For the remaining 640 graphs, the practitioner identified 640 exit points; i.e. only 1 exit point existed per graph. From these 648 exit points, the practitioner identified two external systems acting as information sinks, the Browser with 545 exits points and the MySQL database with 103 exit points. Each exit point requires all unique paths leading to it to be covered. For this application, there is a 1-to-1 mapping between the number of exit points and number of paths. However, other applications may have multiple unique paths leading to a single exit point; all these unique paths need to be tested in order for the coverage criterion to be met.

5.3.2 Selecting Test Data for the CGs

For paths leading to the MySQL exit point, the practitioner used the three test cases discussed in Section 4.D. In terms of web browsers, this application only supports Internet Explorer. Upon reviewing all available documentation for IE 6.5, which is the lowest version supported by the application under investigation, the practitioner selected the following test data to be used for IE exit points:

- <script>alert(‘hello’)</script> - This input value attempts to insert a payload directly without any obfuscation. If the browser pops up a message box when a path is executed with this value, then the web application has an EIV.
- <b onmouseover="alert('hello')">A - This input value will hide the JavaScript code in a harmless formatting tag. If the browser pops up a message box, after a path is executed with this value, when the mouse is moved over to the letter A, then the web application has an EIV.
- "> <script>alert(‘hello’)</script> - This input value attempts to escape the enclosure of a parameter within a tag, then closes the tag and inserts a malicious payload. For example, a benign tag like * Hello!* when expanded with the input value will become <script>alert(‘hello’)</script>Hello! which means that the JavaScript code is successfully embedded. If the browser pops up a message when a path is executed with this value, then the web application has an EIV.
- " style="background:url(javascript:alert('hello'))"> - This input value also attempts to escape the enclosure of a parameter within a tag; it also obfuscates the payload code by embedding it within a style parameter. This input value will only work with IE because IE accepts JavaScript code from many uncommon tags and parameters. If the browser pops up a message when a path is executed with this value, then the web application has an EIV.

5.4 Test Execution, Results and Analysis

To prioritize the test cases, the practitioner grouped the test cases according to the input type. Table 5.D.1 displays the test cases required for each input type.

Table 5.D.1. Number of paths and test cases

Input type	External system at exit point	Number of paths	Number of test cases
Inter-organization	Browser	1	4
	MySQL	103	309
Intra-organization	Browser	544	2176
	MySQL	0	0

This table shows that 648 paths should be tested; a maximum of 2489 test cases (3 test cases per path with MySQL as the exit point, and 4 test cases per path with the browser as the exit point) were executed in order for all the paths to be covered. To speed up the testing process, if one test case for a path fails (demonstrating that a EIV exists), then the practitioner simply ignored the rest of the test cases for the path.

Because the web application under investigation implements client side protection for inputs originating from the browser, a technique similar to bypass testing [50] was used to test inputs from the browser. To test for inputs from the DBMS, the practitioner used the MySQL command line client to insert the test values into the tables used to store data.

A capture and playback tool (AutoIt)[e] was used to aid the execution of the test cases. Only one test case per path was executed (to record the necessary key strokes and mouse clicks). The recorded scripts were then modified to change the test data to accommodate the remaining test cases. The execution process took 3 hours to complete for inter-organization test cases and 32 hours for intra-organization test cases. Table 5.D.2 displays the results of the tests.

Table 5.D.2. Test results

Data type	External System	Paths tested		Test cases	
		Passed	Failed	Passed	Failed
Inter-organization	Browser	0	1	0	1
	MySQL	29	74	87	102
Intra-organization	Browser	453	90	1816	90
	MySQL	0	0	0	0

This table reveals that all intra-organization inputs are trusted by the web application. It does not perform any input verification and validation for intra-organization inputs. As more and more web applications increase their reliance on intra-organization inputs from external systems, the number of EIVs will only increase unless developers begin to validate inputs from these external data sources.

The test results show that 165 EIVs exist (Table 5.D.2 shows that 165 paths failed) for revision five of the test application. However, the security review only identified 68 EIVs. The 165 potential EIVs were tested in revision six; (Table 5.D.3)

Table 5.D.3. EIVs found

Data type	Revision 5		Revision 6		Revision 7		Revision 8	
	EIV analysis	Review	EIV analysis	Review	EIV analysis	Review	EIV analysis	Review
Inter-organization	75	68	7	0	7	0	7	0
Intra-organization	90	0	90	0	90	0	90	0

Upon discussions with the developers, the application was found to be highly susceptible to intra-organization inputs because it assumes all intra-organization inputs are inherently safe. While intra-organization inputs were not examined during the security review process and hence they were not detected by the security review, the remaining 7 inter-organization EIVs should have been identified and addressed. When presented with the results, the organization revealed that the approach they used was not able to identify the 6 inter-organization inputs originating from JavaScript rather than the common form fields. The 7th EIV detected resulted in a stored XSS vulnerability. This vulnerable inter-organization input was code reviewed; however, because the input is transmitted to a MySQL server to be stored rather than being printed to the browser, the code review process examined the guard for the MySQL exit point rather than the guard for the browser exit point. Therefore, the EIV was not detected during the security review. The developers have confirmed that the additional EIVs discovered using this approach are valid; they have addressed all the EIVs found in a recently released revision of the web application. When the test cases were re-applied to this new revision, no EIVs were detected.

---

[e] http://www.autoitscript.com/autoit3/

Table 5.D.4. Effort

Technique	Security Review	EIV Analysis	
Input Space	Inter-organization inputs only	Inter-organization inputs	Intra-organization inputs
Time required	**24 hours**	**7.5 hours**	37.5 hours

Table 5.D.4 shows the effort required for the security review and EIV analysis. The total effort required for EIV analysis is not a sum of the intra and inter organization effort because the effort for the sitemap creation and input identification are shared. Although the security review took only 24 hours to complete, it did not consider intra-organization inputs. If EIV analysis did not examine intra-organization inputs, then the testing process would only require 7.5 hours to complete; and it identified 7 additional EIVs than the security review process. This means that EIV analysis can reduce the required time to perform a security review by 69%. Finally, the security review process was penetration testing with a patching component. Penetration testing uses a "librarian testing" approach which simply attempts to exploit known EIVs on a new application [62]. Unlike penetration testing, EIV analysis is a testing strategy designed to discover EIVs; it is a technique belonging in the "unanticipated user input" class of techniques [65].

6 Related Work

Many techniques and approaches to detect, or mitigate against, vulnerabilities have been proposed. In this section, these techniques are briefly presented and discussed.

Many techniques address an individual class of web application vulnerability. These techniques often concentrate on one popular vulnerability type: SQL injection. SQLrand [8], AMNESIA [21], SQL-Guard [9], SQLCheck [59], CSSE [53], WASP [22] are all approaches aimed at addressing SQL injection vulnerabilities. SQLRand inserts random tokens into SQL statements and uses a proxy server to translate these tokens. An incorrect query can be detected if the SQL query does not contain the correct tokens. This approach, while effective, can be defeated if the randomized tokens can be guessed; it is also complex to setup with the addition of the proxy server. AMNESIA, SQLGuard and SQLCheck are all model-based approaches. AMNESIA uses static analysis and runtime monitoring to detect for SQL injection vulnerabilities. Static analysis is used to build models of the SQL statements, while the runtime engine detects whether the query strings matches the models. This approach is prone to false negatives and positives if the static analysis used to build the model is not effective. SQLGuard requires the developers to call special functions to build a model of the SQL query to be used. SQLCheck uses a formal definition of a SQL injection vulnerability and identifies SQL injection attacks based on the formal definition. Both approaches require developers to learn and gain experience with complex models (in case of SQLCheck) or APIs (if SQLGuard is utilized). CSSE and WASP are dynamic approaches designed to address SQL injection vulnerabilities using taint analysis. These approaches attempt to mark negative tainting (CSSE) or positive tainting (WASP) to identify malicious query statements before they are passed onto the DBMS. Both approaches involve modification to either the runtime engine or usage of a specialized API; hence, deployment can be expensive or programmers need to learn yet another API respectively. While some of the approaches listed claim to support other types of EIVs (SQLCheck, CSSE), their supplied tool only concentrates on detecting one type of EIV (SQL injection) which leaves the system vulnerable to other types of EIVs. EIV analysis does not have this limitation because the strategy is designed to address all types of EIVs.

General approaches to applications' security have also been proposed which address all types of EIVs. Security Gateway proposed by Scott and Sharp [56] is an application firewall that filters out all malicious inputs before they reach the web application. The effectiveness of this approach is dependent on an administrator's ability to produce complex and effective rule sets. Nguyen-

Tuong et. al. [48] proposed a dynamic approach to detect EIVs through taint analysis. This approach requires the runtime engine to be modified which causes complex deployment and increased overhead.

All approaches discussed are application security techniques. That is, they protect the software after it has been built [43]. EIV analysis is a software security strategy; the approach increases the security of web applications during the development process and before they are deployed on live servers. Several software security approaches related to EIV analysis currently exist. They can be classified into two categories: static analysis approaches and black-box testing [6] techniques.

Static approaches have been used to detect vulnerabilities with some success. Shankar et. al. [57] proposed a static approach that can detect format-string vulnerabilities commonly found in C-based applications. The method defines two extended data types, tainted and untainted, which help reduce the amount of false positives generally associated with static analysis methods. Zhang et. al. [72] and Johnson and Wagner [31] further extend the approach by using it to assess security issues with the Linux Security Modules framework and user/kernel pointers successfully. These approaches are designed to detect vulnerabilities in C-based applications, and hence, their effectiveness with scripting languages such as PHP, Ruby, and Python remain unknown.

Although static analysis is a well known technique, approaches that specifically target web applications' EIVs are not common. Proposed approaches such as those by Livshits and Lam [40], Martin et. al. [41], Balzarotti et. al. [5], WebSSARI [27] and Pixy [32] have limitations. Techniques proposed Livshits and Lam [40], Martin et. al. [41] are designed specifically for SQL injection vulnerabilities, and hence, cannot be used to detect other EIVs. Balzarotti et. al. [4] presents a static analysis approach capable of detecting both workflow attacks and data-flow attacks. However, the approach cannot detect all EIVs. For example, many websites now have multiple web applications sharing the same database. An attacker can utilize a vulnerability in one web application (A) to inject a payload into the database which will then be used by the other web application (B). If the approach is used to analyze (B), this vulnerability would be undetected. WebSSARI [27] does not model conditional branches that result in many false positives. Furthermore, the WebSSARI tool is not available and hence, no comparison with it can be made. Pixy [32] is currently the most advanced static taint analysis tool available for PHP. However, attempts to use the tool for comparison with EIV analysis reveal several issues:

- Pixy cannot detect stored XSS, and other types of EIVs (OS/Filesystem and Interpreter interactions).
- 6 of the 7 XSS vulnerabilities it detected, when used on the case study's application, are false positives.
- It ignores path information and tainted data inside objects, and hence, its reports contain false positives and negatives.
- It requires a very large amount of memory to model SQL injections. In fact, on the test machine which has 2GB of RAM, it crashed repeatedly when used on the case study's application.

Offutt et. al. [50] proposed a black-box testing approach that requires a customized client to test web applications. The customized client allows the tester to bypass all client side protection mechanisms; and hence, if a web application is dependent on client side verification of inputs, it will fail the test cases. QED [42] and Ardilla [34] attempt to generate SQL Injection and XSS attacks automatically. However, QED cannot target second order XSS attacks and requires users to learn a custom specification language. Ardilla suffers from low code coverage and a 42% false positive rate. Secubat [33] and other commercial web scanners such as Acunetix Web Vulnerability Scanner [2] extend bypass testing by creating tools that provide automatic penetration testing for web applications without using the web applications' target clients. Commercial applications are proprietary and closed source; hence they cannot be examined in detail. Secubat currently has no plug in to detect all types of XSS; for example, stored XSS [52]. Lin and Chen [37] extend traditional black-box testing techniques with elements of static analysis

by including a tool to automatically inject guards at input points found through the crawling component. This approach does not guarantee correctness of the modified program and hence, the modified program may not meet the original requirements. All black-box testing approaches for web applications have a limitation that not all inputs can be detected through web page parsing [50]; hence, only an approximation of the inputs is possible.

While many black-box approaches to web application security testing have been proposed; no white-box strategies have been presented. EIV analysis is a white-box approach that utilizes data flow graphs to test for EIVs. Just as other software security approaches, EIV analysis allows a company to test its web applications before they are launched. EIV analysis can also coexist with all of the approaches presented. That is, an organization can use static analysis approaches to automatically identify some EIVs, then use EIV analysis and black-box approaches to locate additional vulnerabilities. Finally, application security approaches can be applied to monitor the web application when it is deployed.

Table 6.1 presents an overview comparison of the mentioned approaches and EIV analysis based on five features. These five features are chosen based on the available information contain in other approaches. Other benchmarks such as detection rate, false positive rate, false negative rate and overhead costs cannot be reliably compared due to insufficient information. The approaches are grouped into categories for comparison because of the numerous amounts of approaches within each category. A category with "Varies" means the approaches belong in that category may or may not have that feature. The categories are as follows.

- SQL Injection – this category covers all techniques primarily aimed at addressing SQL injection vulnerabilities.
- XSS – this category covers all techniques that address XSS vulnerabilities.
- Taint analysis – this category covers all techniques that use taint analysis as a main method of detecting vulnerabilities.
- Application Firewall – this category covers all tools that act as an application firewall for web applications.
- Static Analysis – this category covers all techniques that use static analysis to detect vulnerabilities.
- Black Box Testing – this category covers all black box testing techniques used to detect vulnerabilities

Table 6.1. An Overview Comparison of Various EIV Detection Approaches

Approach	Detection of all known EIVs	Inter-input detection	Intra-input detection	Require changes to run-time environment	Require knowledge of a new language
SQL Injection	No	Yes	No	Varies	Varies
XSS	No	Yes	No	Varies	Varies
Dynamic Taint Analysis	Yes	Yes	No	Yes	No
Application Firewall	Yes	Yes	No	Yes	No
Static Analysis	Yes	Yes	No	No	Varies
Black Box Testing	Yes	Yes	No	No	No
EIV Analysis	Yes	Yes	Yes	No	No

7 Conclusions

This paper introduces a novel technique aimed at detecting EIVs. The paper provides a systematic

approach that security partitioners can apply in order to reveal EIVs from web applications. EIV analysis contains four steps:

1. Sitemap generation
2. Input identification
3. Contamination Graph generation
4. Test case selection and execution

The steps are semi-automated using a web crawler, WAIC, WAGG and a capture playback tool. This divide and conquer approach allocates the repetitive and time consuming steps to various tools, and hence, reduces a significant effort required by security practitioners. Furthermore, security practitioners' expertise and experience remain an essential part which allows the approach to have a high detection rate without any false positives. This approach proposed satisfies the research problems identified in Section 3; because it is a software development process, it is applicable to all web applications, the large configuration space and language dependent. A case study was performed to determine the effectiveness of the approach; it demonstrates that the approach is practical and applicable to commercial strength applications.

EIV analysis has been found to have a very high detection rate for EIVs. The approach found all of the vulnerabilities found by the professionals during a security review; and in addition, found 7 new vulnerabilities missed by the review process. These vulnerabilities were missed because the review process did not correctly identify all of the inputs and hence, several tainted paths were missed. Furthermore, EIV analysis reduced the time required for a security review by 69%. Finally, because the security review process is an example of a penetration testing with a patching component, it suffers from the same limitations as all penetration testing techniques as described by Thompson [62] whereas EIV analysis does not possess similar limitations.

Our strategy has several advantages over existing proposed approaches. It is a software security technique which helps secure web applications before they are deployed on the Internet. It does not generate false positives which static analysis approaches are prone to. Furthermore, current proposed static approaches cannot detect all types of EIVs, have false negatives, or not practical due to performance issues. By using a white-box approach, all inputs can be identified which is not possible with black-box testing approaches. Detailed information flow from the data source to data sinks can be modeled which allows security practitioners to choose test cases aimed at uncovering EIVs within these tainted data flows. Because EIV analysis is a generic testing technique used to identify EIVs, it is applicable to any web application created using any technology and not limited to the PHP/MySQL platform used in the case study. Finally, for extremely sensitive web applications such as online banking systems, our approach can be used in addition to other existing approaches to further enhance the security of the application.

Although this paper only lists four external types of systems that the web application interacts with, EIV analysis is applicable to interactions between the web application and other external systems as well. For example, a web application can interact with other web services [12][3] or NXDs [10]. As companies increasingly rely on COTS [7] to power their web applications, EIV analysis becomes more valuable as a technique to ensure that web applications are secured from interactions with external third party applications.

References

[1] Achour, M., Betz, F., Dovgal, A., Lopes, N., Olson, P., Richter, G., Seguy, D., Vrana, J., PHP manual, http://www.php.net/manual/en/, 2007.
[2] Acunetix Ltd., Acunetix Web Vulnerability Scanner, http://www.acunetix.com/, Last accessed Feb. 7, 2006.
[3] Alonso, G., Casati, F., Kuno, H., and Machiraju, V., Web Services: Concepts, Architectures, and Applications. Springer Verlag, 2003.

[4] Bandhakavi, S., Bisht, P., Madhusudan, P., Venkatakrishman, V., CANDID: Preventing SQL Injection Attacks Using Dynamic Candidate Evaluations, CCS 2007.
[5] Balzarotti, D., Cova, M., Felmetsger, V.V., Vigna, G., Multi-Module Vulnerability Analysis of Web-based Applications. Proc. 14th ACM Conference on Computer and Communication Security, pp. 25-35, 2007.
[6] Beizer B. Black-Box Testing: Techniques for Functional Testing of Software and Systems. Wiley: New York, 1995.
[7] Boehm, B., and Abts, C., COTS Integration: Plug and Pray?, IEEE Computer, 32 (1): pp. 135-138, January 1999.
[8] Boyd, S. W., and Keromytis, A. D., SQLrand: Preventing SQL Injection Attacks, In Proc. of the 2nd Applied Cryptography and Network Security Conf. (ACNS '04), pages 292–302, Jun. 2004.
[9] Buehrer, G. T., Weide, B. W., and Sivilotti, P. A. G., Using Parse Tree Validation to Prevent SQL Injection Attacks, In Proc. of the 5th Intl. Workshop on Software Engineering and Middleware (SEM '05), pages 106–113, Sep. 2005.
[10] Chaudhri, A., Zicari, R., & Rashid, A., XML Data Management: Native XML and XML Enabled DataBase Systems, USA: Addison-Wesley, 2003.
[11] Cruwys, D., C Sharp/VB - Automated WebSpider/WebRobot, http://www.codeproject.com/csharp/DavWebSpider.asp, March 2004.
[12] Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S., Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI, IEEE Internet Computing, v.6 n.2, p.86-93, March 2002.
[13] Denning, D.E., Denning, P.J., Certification of programs for secure information flow. Comm. Of the ACM, 1977.
[14] Dhamija, R., Tygar, J. D., Hearst, M., Why phishing works, Proceedings of the SIGCHI conference on Human Factors in computing systems, Montréal, Québec, Canada, April 22-27, 2006.
[15] Doar, M.B., Practical Development Environments, O'Reilly Media, 2005.
[16] Eaton, C., Memon, A. M., An Empirical Approach to Testing Web Applications Across Diverse Client Platform Configurations, International Journal on Web Engineering and Technology (IJWET), Special Issue on Empirical Studies in Web Engineering, 2007.
[17] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., and Berners- Lee. T., Hypertext Transfer Protocol-HTTP/1.1, RFC 2068 (http://www.ietf.org/rfc/rfc2068), Jan 1997.
[18] Flanagan, D., JavaScript (2nd ed.): the definitive guide, O'Reilly & Associates, Inc., Sebastopol, CA, 1997.
[19] Frankl, P.G., Weyuker, E.J., "An applicable family of data flow testing criteria," IEEE Transactions on Software Engineering, vol.14, no.10pp.1483-1498, Oct 1988.
[20] Granger, S., Social Engineering Fundamentals, Part I: Hacker Tactics, Security Focus, http://www.securityfocus.com/infocus/1527, 2003.
[21] Halfond, W. G., and Orso, A,. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks, In Proceedings of 20th ACM International Conference on Automated Software Engineering (ASE), Nov 2005.
[22] Halfond, W. G., Orso, A., and Manolios, P., Using positive tainting and syntax-aware evaluation to counter SQL injection attacks, In Proceedings of the 14th ACM SIGSOFT international Symposium on Foundations of Software Engineering, pp. 175-185, 2006.
[23] Harrold, M., J., and Rothermel, G., Performing data flow testing on classes, In Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (New Orleans, Louisiana, United States, December 06 - 09, 1994), SIGSOFT '94, ACM Press, New York, NY, pp. 154-163, 1994.
[24] Heydon, A., Najork, M., Mercator: A scalable, extensible Web crawler. World Wide Web, 2(4):219–229, December 1999.
[25] Howard, M., and LeBlanc, D. Writing Secure Code, Second Edition, Microsoft Press, 2003.

[26] Howden, W. E., “Methodology for the generation of program test data,” IEEE Trans. Comput., vol. C-24, no. 5, pp. 554-559, May 1975.
[27] Huang, Y. W., Yu, F., Hang, C., Tsai, C. H., Lee, D. T., and Kuo, S. Y., Securing web application code by static analysis and runtime protection, in WWW '04: Proceedings of the 13th International Conference on World Wide Web. New York, NY, USA: ACM Press, pp. 40-52, 2004.
[28] Hurst, A., Analysis of Perl’s taint mode, http://hurstdog.org/papers/hurst04taint.pdf, 2004.
[29] Huynh, T., Miller, J., An empirical investigation into the causes of open source web applications vulnerabilities, Submitted for publication, 2008.
[30] IEEE, IEEE Standard for Software Maintenance (IEEE Std 1219–1998), Institute for Electrical and Electronic Engineers: New York NY, 1998.
[31] Johnson, R., and Wagner, D., Finding user/kernel pointer bugs with type inference. In Proceedings of the 2004 Usenix Security Conference, pages 119—134, 2004.
[32] Jovanovic, N., Kruegel, C., and Kirda, E., Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities, In 2006 IEEE Symposium on Security and Privacy, May 2006.
[33] Kals, S., Kirda, E., Kruegel, C., and Jovanovic, N., SecuBat: A Web Vulnerability Scanner, The 15th International World Wide Web Conference (WWW 2006), Edinburgh, Scotland, May 2006.
[34] Kiezun, A., Guo, P.J., Jayaraman, K., Ernst, M.D., Automatic Creation of SQL Injection and Cross-Site Scripting Attacks, MIT Technical Report, MIT-CSAIL-TR-2008-054, 2008.
[35] Kristol, D.M., and Montulli, L., HTTP State Management Mechanism, RFC 2965 (http://tools.ietf.org/html/rfc2965), October 2000.
[36] Laski, J. W., Korel, B., Data flow oriented program testing strategy, IEEE Transactions on Software Engineering. Vol. SE-9, no. 3, pp. 347-354. 1983.
[37] Lin, J.-C., Chen, J.-M., "An Automatic Revised Tool for Anti-Malicious Injection," Sixth IEEE International Conference on Computer and Information Technology (CIT'06), pp. 164-170, 2006.
[38] Lipner, S.B., "Security and Source Code Access: Issues and Realities," 2000 IEEE Symposium on Security and Privacy (S&P 2000), pp. 124-125, 2000.
[39] Liu, C. H., Kung, D., Hsia, P., and Hsu, C. T., Structure testing of web applications. In Proceedings of the 11th Annual International Symposium on Software Reliability Engineering, pages 84–96, San Jose CA, October 2000.
[40] Livshits V. B., and Lam., M. S., Finding Security Vulnerabilities in Java Applications with Static Analysis, In Proceedings of the 14th Usenix Security Symposium, Aug. 2005.
[41] Martin, M., Livshits, B., and Lam, M. S., Finding Application Errors and Security Flaws Using PQL: a Program Query Language. In OOPSLA ’05: Proc. of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, pages 365–383, Oct. 2005.
[42] Martin, M., Lam, M., Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking, Security 08.
[43] McGraw, G., “Software Security,” IEEE Security & Privacy, vol. 2, no. 2, pp. 80–83, 2004.
[44] Miller, R.C., and Bharat, K. SPHINX: A framework for creating personal, site-specific Web crawlers. In Proceedings of the Seventh International World Wide Web Conference, pages 119--130, April 1998.
[45] Moody, K., and Palomino, M., SharpSpider: Spidering the Web through Web Services, First Latin American Web Congress (LA-WEB 2003), 2003.
[46] Musciano, C., Kennedy, B., HTML and XHTML: The Definitive Guide, O'Reilly & Associates, Inc., Sebastopol, CA, 2002.
[47] Myers, G. J., The Art of Software Testing. Wiley, 1979.
[48] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D., Automatically Hardening Web Applications Using Precise Tainting, In Proceedings of the 20th IFIP International Information Security Conference, 2005.

[49] Ntafos, S. C., “Required element testing,” IEEE Trans. Software Eng.,vol. SE-10, no. 6, pp. 795-803, Nov. 1984.
[50] Offut, J., Wu, Y., Du, X., & Huang, H., Bypass testing of Web applications. In Proceedings of The Fifteenth IEEE International Symposium on Software Reliability Engineering, Saint-Malo, Bretagne, France, pp.187-197 2004.
[51] Ollman, G., The phishing guide - understanding and preventing phishing attacks, White Paper, Next Generation Security Software Ltd., 2004.
[52] OWASP, Cross site scripting, http://www.owasp.org/index.php/Cross_Site_Scripting, Accessed January 22, 2007.
[53] Pietraszek, T., and Berghe, C. V., Defending Against Injection Attacks through Context-Sensitive String Evaluation, In Proceedings of Recent Advances in Intrusion Detection (RAID2005), 2005.
[54] Raghavan, S. and Garcia-Molina, H., Crawling the hidden web, In Proc. of 27th Int. Conf. on Very Large Databases, Sept. 2001.
[55] Rapps, S., and Weyuker, E., J., Selecting software test data using data flow information. IEEE Trans. Softw. Eng. 11, 4 (Apr. 1985), 367-375. 1985.
[56] Scott, D., and Sharp, R., Abstracting Application-level Web Security, In Proc. of the 11th Intl. Conference on the World Wide Web (WWW 2002), pages 396-407, May 2002.
[57] Shankar, U., Talwar, K., Foster, J. S., and Wagner, D., Detecting format string vulnerabilities with type qualifiers. In 10th USENIX Security Symposium, D.C., pages 201-220, 2001.
[58] Shiflett, C., PHP Security, O’Reilly Open Source Convention, Portland, Oregon, USA, 26 Jul 2004.
[59] Su, Z., and Wassermann, G., The Essence of Command Injection Attacks in Web Applications, In The 33rd Annual Symposium on Principles of Programming Languages, pages 372–382, Jan. 2006.
[60] Symantec, Symantec Internet Security Threat Report, http://www.symantec.com/enterprise/threatreport/index.jsp, March 2006.
[61] Tappenden, A.; Beatty, P.; Miller, J.; Geras, A.; Smith, M., "Agile security testing of Web-based systems via HTTPUnit," Agile Conference, 2005. Proceedings, vol., no.pp. 29- 38, 24-29 July 2005.
[62] Thompson, H.H., “Why Security Testing Is Hard,” IEEE Security & Privacy, vol. 1, no. 4, pp. 83–86, 2003.
[63] Vaswani, V., MySQL: The Complete Reference, McGraw-Hill/Osborne, 2004.
[64] Wheeler, D., A., Secure Programming for Linux and Unix HOWTO, http://dwheeler.com/secure-programs, 2003.
[65] Whittaker, J., and Thompson, H., How to Break Software Security, Addison-Wesley, 2003.
[66] Widenius, M., Axmark, D., and MySQL AB, MySQL Reference Manual, Sebastopol, Calif.: O’Reilly, 2002.
[67] Wiegers, K., Software Requirements, Microsoft Press, Redmond, 1999.
[68] Williams, H. E., and Lane, D., Web Database Applications with PHP & MySql, O'Reilly, 2002.
[69] Woodward, M. R., Hedley., D., and Hennel, M. A., “Experience with path analysis and testing of programs,” IEEE Trans. Software Eng., vol. SE-6, no. 3, pp. 278-286, May 1980.
[70] Xie, Y., Aiken, A., Static Detection of Security Vulnerabilities in Scripting Languages, Security 2006.
[71] Xu, W., Bhatkar, S., and Sekar, R., Practical dynamic taint analysis for countering input validation attacks on web applications., Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook University, May 2005.
[72] Zhang, X., Edwards, A., and Jaeger, T., Using CQual for static analysis of authorization hook placement. In the Proceedings of the 11th USENIX Security Symposium, pp. 33-48, 2002.