Extraction of Web Applications Vulnerabilities

Fatma A. El-Licy
Department of Computer and Information Sciences, Institute of Statistical Studies and Researches, Cairo University, Giza, Egypt.
why2fatma@yahoo.com

Abstract

Web Security issues plays an important role in the development of real life web systems. Malicious attacks of web based systems, usually, inflect damages and losses in finance capitals and may, even, compromise the reputation of those institutes under attack. Therefore, the security of web applications is an essential issue to be addressed to and be understood by the application developers. Web applications vulnerability to intrusion and malicious attacks can be exposed by the application of software testing techniques. The early discovery of an application's vulnerabilities, would, normally, assist in rectifying the application software as well as adjusting the design and implementation for better practice to avoid such vulnerability.

The objective of this paper is to present an approach to extract vulnerabilities in web applications code, including both server side (Cookie Poisoning, SQL Injection, Cross-Site Scripting, CGI Parameters); and client side (Buffer Overflow, Bypass Restrictions on Input Choices and Hidden Field.)

The presented approach adopted white box code analysis to expose different types of vulnerabilities to ensure security. A general framework for the methodology of utilizing static analysis and code slicing verification technique is described. A prototype for the system has been designed and implemented to evaluate the presented approach. The method, not only, expose taint code in the web application, but it also, eliminates the false positive results incurred in most of static analysis-based scanners.

The system applied a proactive approach to provide advices and remedies to fix potential code vulnerabilities, and to avoid consequence, possible, attacks. The presented system can, easily, be adapted for any Web developing language; however, it was designed with a front end compiler for PHP based code.

Keywords: *Web applications security, Static analysis, SQL Injection, Cross-Site Scripting, Cookie Poisoning.*

1. Introduction

In the context of increased interconnection among information systems and networks, the application of successful, malicious attacks, usually, inflect negative consequences. Even, ordinary unskilled individuals may cause various types of harmful attacks by initiating malicious scripts [1, 2]. The results of malicious attacks include financial and reputation loss, drop in the value of a company's stock and many other legal issues [3].

SASN, in its 2015 survey, [4] found that 79% of Security Risk Management-Aligned with Development- are focused on applying security resources to public-facing Web applications, where security risks are the greatest. This trend in research has been even more

intensive on methods and algorithms for automatically detecting information-flow violations in Web application. Information-flow violations may lead to potential leakage of information and/or integrity breach such as cross-site scripting (XSS), SQL injection (SQLi), and others.

Research solutions, typically, focus on two approaches, type systems and program slicing. Both suffer from a high rate of false findings, which limits the usability of analysis tools based on these techniques. Attempts to reduce the number of false findings have resulted in analyses that are, either (i) unsound, suffering from the dual problem of false negatives, or (ii) too expensive due to their high precision, thereby failing to scale to real-world applications [5, 6, 7, 8]. However, the presented approach overcomes this by addressing the potential paths onto a sensitive computation that are influenced by untrusted input or tainted code.

One of the recent researches [9] focused on analyzing the existing practices in developing web applications and synthesizing security vulnerabilities evidence, based on, the empirical studies reported to address solutions for vulnerable web application.

The Web application security testing, adopted in this work, may be accomplished by adopting variety of verification techniques [10, 11, 12, 13]. They are either concerned with exercising the activities of the web applications in order to realize its vulnerabilities [14, 15], or examining the Web application vulnerabilities, to eliminate common security exploits and to secure the emerging classes in web applications [16, 17, 18] through vulnerabilities detection or prevention [19, 20, 21, 22, 23]. Some are concerned with the automatic generation of test cases for specific types of vulnerabilities [13, 14, 15, 16, 17, 18, 21, 22]; whereas others are applying different techniques to emulate the web pages themselves [23]. Tools and packages are available both, commercially and open sources, to detect some types of vulnerabilities [24, 25, 26, 27, 28, 29].

The basic idea of the approach is to isolate the vulnerable (tainted) code in a given application. The code became tainted whenever it is vulnerable and uses a tainted value/variable/parameter (defined by either untrusted input/source or tainted code). Therefore, our goal, here, is to trace the propagation of the tainted variables over the control paths of the application. However, precisely identifying all the paths in a given application is equivalent to the halting problem. Yet, a presented solution was to associate each tainted variable with its life scope and trace them, only, through their life paths onto a sensitive/vulnerable computation code.

The main objective of this paper is to expose the security vulnerabilities embedded in the web application code or transferred through the client side applications. Security verification should provide coverage for code related web security issues.

The activity of a given application are managed by the data flow and controlled by the data influences over the application variables. Some of these data are communicated to the application either by web application's user or externally through a linked database.

Those communicated data are one of the main instruments used by the attackers to intrude web applications. Therefore, this paper utilizes a data-flow analysis technique, namely: static slicing, to isolate the portion of the application code that is vulnerable to potential security breaches (tainted code).

Some type of attacks can be exposed by examining the client side application code, whereas, most of the vulnerabilities are found to be embedded in the server side code of the

application. This paper is concerned with the types of attacks encouraged by the code of client side (Buffer Overflow, Bypass Restrictions on Input Choices and Hidden Field) and that of the server side vulnerabilities (Cookie Poisoning, SQL Injection, Cross-Site Scripting and CGI Parameters).

SQL Injection and Cross Site Scripting (XSS) attacks are widespread forms of attacks in which the attacker crafts the data communicated to the application to access or modify user data and execute malicious code. In the most serious attacks (called second-order, or persistent, XSS), an attacker can corrupt a database so as to cause subsequent attacks that execute malicious code. This paper is organized as follows:

Section 2 reviews the related work, Section 3, illustrates the system theory, Section 4 introduces the framework for establishing the theory into a working scanning tool. Section 5 discusses the prototype for the system implementation for PHP and HTML web Languages. Section 6 discusses the results and evaluation Section 7 is the conclusion and discussion.

2. Related Work

This paper is concerned with information flow influence in the application code, demonstrated with static analysis of code slice dependencies of the security sensitive computation (taint analysis). Chang and Newsome [30, 31] introduced a survey for dynamic taint-analysis techniques. A detailed overview of works on program slicing is given in [32] and references therein.

The presented system (*WAVE*), employed static slicing to extract the vulnerable (taint) code from server side, and employed *Microsoft .Net Framework* Regular Expressions for checking client side code. *WAVE* system expose most types of code security vulnerabilities including Cookie Poisoning, SQL Injection, Cross-Site Scripting, CGI Parameters, Buffer Overflow, Bypass Restrictions on Input Choices and Hidden Field. The prototype of the system targeted PHP web application language, yet the presented system with a support of a front end compiler is applicable for most web application languages. Agosta et al, [33], presented a methodology and tool for vulnerability identification based on symbolic code execution exploiting Static Taint Analysis. Their tool target PHP web applications for identifying, only, cross-site scripting and SQL injection vulnerabilities. Omer Tripp et al, [34] introduced a scanning tool, which refrains from building global program representations. Their tool provides a demand driven analysis, which enables lazy computation of vulnerable information flows. It supports applications written in Java, .NET and JavaScript. Volpano et al. [4] showed a type-based algorithm that verifies implicit and explicit flows and also guarantees noninterference. Given a program, the principle of noninterference stated that low-security behavior of the program is not influenced by any high-security data, unless that high-security data has been previously downgraded [35]. Shankar et al. [36] presented a taint analysis for C using a constraint-based type-inference engine based on cqual. Similarly to the flow graph built by *WAVE*, a constraint graph is constructed for a cqual program, and paths from tainted nodes to untainted nodes are flagged. Myers' Java Information Flow (JIF) [37] utilized type-based static analysis to track information flow. Based on the Decentralized Label Model [38], JIF considered all memory as a channel of information, which requires that every variable, field, and parameter used in the program be statically labeled. Labels can either be declared or inferred, Similar to defined-used notation in the theory applied in the *WAVE* system. Ashcraft and Engler [39], also, applied taint analysis to detect software attacks due to tainted variables. Their approach provides user-defined sanity checks to untaint potentially

tainted variables. Pistoia et al, [40] presented a static analysis to detect tainted variables in privilege-asserting code in access-control systems based on stack inspection. Snelting et al, [41] made the observation that Program Dependence Graphs (PDGs) and noninterference are related, it employed backward slicing to map each statement to its static backwards slice. Based on this observation, Hammer et al, [42] presented an algorithm for verifying noninterference. Though promising, this approach has not been shown to scale. Unlike *WAVE* that employs forward analysis to expose potential vulnerabilities due to interference caused by code dependencies and data flow influence. Livshits and Lam [43] analyzed Java EE applications by tracking taint through heap allocated objects. Their solution required prior computation of Whaley and Lam's flow insensitive, based on Binary Decision Diagrams (BDDs) [44], which limits the scalability of the analysis [45, 46]. Guarnieri et al, [47] presented a taint analysis for JavaScript. Their work relies on Andersen's whole program analysis [48].

Wassermann and Su [49] extended Minamide's string-analysis algorithm [22] to, syntactically, isolate tainted substrings from untainted substrings in PHP applications. They labeled non-terminals in a context-free grammar with annotations reflecting taintedness and untaintedness. Their expensive yet elegant mechanism was applied to detect both SQLi and XSS vulnerabilities. Subsequent work by Tateishi et al, [50] enhanced taint-analysis precision through a string analysis that automatically detects and classifies downgraders in the application scope. The front end of the *WAVE* system prototype, however, engineered a predictive grammar from the context-free grammar of the PHP language, with terminals defined as regular expressions. This formalism facilitates the recognition of the vulnerable statement, therefore, the tainted statements (whenever influenced by a tainted code), without further analysis of the code string.

McCamant and Ernst [51] took a quantitative approach to information flow: instead of using taint analysis, they cast information-flow security to a network-flow-capacity problem, and describe a dynamic technique for measuring the amount of secret data that leaks to public observers.

Parameshwaran et al [52] proposed a technique to mitigate the DOM-based XSS injection vulnerability caused by the unsafe dynamic code generation of JavaScript applications. They generated secure patches to replace the unsafe string interpolation with safer client side code. Whereas, our approach, verifies the client side code (PHP) and HTML interpolation code against pre-specified Regular Expressions.

3. System Theory

Data-flow analysis technique, [53] was adopted to study the influence of the input data over the variables included in the Web application's statements. A portion/slice of the application software that is, potentially, influenced by such input data, (taint code) is to be isolated. This is accomplished by utilizing static slicing verification technique working as an end-to-end scanner.

3.1 Illustrations

The following is a typical example of SQL vulnerability, with a tainted code:

```
1. $name = $_GET['name'];
2. $q = "select * from users where name = '" . $name . "';";   3. $result = mysql_query($q);
```

The parameter \$name is defined by the user in statement 1, set as an argument that is used in the SQL query created on statement 2, and issued on statement 3, in the variable \$result.

Statement 1 defines the parameter variable \$name, therefore it is a *DEFINITION* of \$name; i.e., {1} = ***D***(\$name). Statement 2 defines the argument variable \$q, and uses the parameter variable \$name, so it is a definition of \$q ({2}= D(\$q))and a *USE* of \$name; i.e., {2} = ***U***(\$name). Statement 3 defines the variable \$result. The define-use chains (DU) for the variables in this code are:

$$\boldsymbol{DU}(\$name) = \{1\ 2\},\quad \boldsymbol{DU}(\$q) = \{2\ 3\},\quad \boldsymbol{DU}(\$result) = \{3\ -\}$$

The code here, was tainted by the variable \$name, that is the attacker control window to cause privacy breaches. The idea of the static analysis is to first identify those spoiling variables that are defined by the user or external input. Then apply data flow analysis to follow the propagation of those definitions throughout the application code. In the illustrative example, the definition (statement 1) of \$name propagated through the code to be used in statement 3. Being a vulnerable statement, it is tainted by propagated variable. The criteria of selecting the taint code, therefore, is to isolate the set of all sensitive U-Statements for any untrusted data, whether directly or through their propagation effects.

3.2 The Theory

The slicing criteria $\boldsymbol{C_v}$ constitutes the set of vulnerable statements in the application code, which could be a window for breaching the code:

$\boldsymbol{C_{v=}\{Vul, m_l\}}$, where:

Vul : the set of variables influenced by externally-defined parameters,

m: the serial number of last statement '*E*' in the code.

The code is analyzed as a program flow graph ***PFG*** with statements as graph nodes $\boldsymbol{\{N_1, N_2,...N_m\}}$ and the program paths as arcs $\boldsymbol{A = \{A_{ij}, i,j \in (1, 2,\ldots,m\} \wedge i \neq j}$. The program graph ***PFG*** for the application code is defined as $\boldsymbol{PFG = \{\{N\}, \{A\}, B, E\}}$, where $\boldsymbol{A_{ij}}$ *is the arc between the two nodes* $\boldsymbol{N_i}$ *and* $\boldsymbol{N_j}$, ***B*** is the start of the program, and ***E*** is the last statement.

Given the graph ***PFG***, that has ***m*** nodes and a set of ***J*** program variables, $\boldsymbol{Var = \{v_1, v_2, \ldots, v_J\}}$, that are manipulated through those ***m*** nodes.

- Any node $N_i \in \{\boldsymbol{N}\}$ that defines a variable $v \in \boldsymbol{Var}$ is in the set of definitions of v denoted by $\boldsymbol{D(v)}$, *i.e.*, $\boldsymbol{N_i \subseteq D(v)}$. The definition of v at N_i is therefore denoted by v^i.
- Each definition $\boldsymbol{N_i}$, for a given variable $\boldsymbol{v \in Var}$, is life only in the scope of its definition, as ***v*** could be redefined in some other node N_k , it follows that

  $\boldsymbol{D(v) = \{N_i, N_k, \ldots\}}, \qquad \boldsymbol{i \neq k}$
- Definition Life scope of a variable *LS(v)*: given that N_0 is the definition of *v*, then its life scope is the path between N_0 and the node N_k , where it is redefined, or the last node *m*, otherwise.

  $(N_0 \in D(v) \wedge N_k \in D(v) \Rightarrow LS(v) = \boldsymbol{A_{0k}}) \wedge$

  $(N_0 \subseteq D(v) \wedge \forall i \mid 0 < i < m \bullet N_i \not\subset D(v) \Rightarrow LS(v) = \boldsymbol{A_{0m}})$ **(1)**
- Any node $\boldsymbol{N_k \in \{N\}}$ that uses variable ***v*** is in set $\boldsymbol{U(v)}$, *i.e.*, $\boldsymbol{N_k \subseteq U(v)}$.
- A node $\boldsymbol{N_i \in D(v)}$, is the reaching definition of v^i *at node* N_k ,$(RD_k(v^i))$, iff

$N_k \in U(v^i)$ and there exist a feasible path, $i \rightarrow K$, ($\mathbf{A_{ik}}$), between N_i and N_k through which v^i *is* not redefined *(i.e., v^i is life at node k-or- node k is in the scope of v^i).*

$$[N_i = D(v^i) \wedge N_k \in U(v) \wedge (\forall r | i < r < k \bullet N_r \not\subset D(v))] \Rightarrow N_i = RD_k(v^i) \quad (2)$$

- A node N_k is influenced by node N_i iff the definition of variable v^i was not altered before it was used at N_k, namely: $RD_k(v^i)$ at node N_k.

$$N_k \in U(v) \wedge N_i = RD_k(v^i) \Rightarrow N_k \in infl(N_i) \quad (3)$$

Each node N_i that is influenced by any *of the* variable $v \in \{Vul\}$, is assumed vulnerable and is added to the slice $C_v(Vul, m)$.

$$N_i = RD_k(v^i) \wedge v^i \in \{Vul\} \Rightarrow N_k \in C_v\{Vul, m_l\} \quad (4)$$

- All nodes $N_k \in \{\mathbf{N}\}$ that are influenced by a vulnerable nodes $N_i \in C_v\{Vul,m\}$, are assumed candidates of potential vulnerability, therefore added to the slice.

$$N_k \in infl(N_i) \wedge N_i \in C_v\{Vul, m_l\} \Rightarrow N_k \in C_v\{V, m_l\} \quad (5)$$

- Each variable v^i defined at a vulnerable node N_i (i.e., $N_i \in C_v\{Vul,m_l\} \vee [N_k \in C_v\{Vul,m_l\} \wedge N_i \in infl(N_k)$) is assumed vulnerable, and is added to the set *{Vul}*. This is realized through an iterative application of rules (5) and (6).

$$N_i \in C_v\{Vul,m_l\} \wedge N_i \in D(v) \Rightarrow v^i \in \{Vul\} \quad (6)$$

- A node N_i is a decision node ($N_i \in DN$) if it has more than one arc,

  $\{A_{ij}, A_{ik}, \ldots, A_{in}) \subset \mathbf{A}$, onto several different nodes, $(N_j, N_k, \ldots, N_n)$.

- All nodes that are in the scope of a given branching node are influenced by it, at least as it is in its control scope.

$$N_k \in scope\ (N_i \in DN) \Rightarrow N_k \in infl(N_i) \quad (7)$$

While this rule is a verification rule, it may actually generate quite a good number of false positive, because the taint branching statement may only be controlling the execution path, without affecting the definitions of the statement in its scope. This case is eliminated by the application of rules (4) and (5).

- Defined-Used Chain of a given variable *v*, denoted by *DU(v)* is the set of couple $(N_{d1}\ N_{u1}, \ldots, N_{dq}\ N_{uq})$, such that $N_{di} \subseteq D(v)$ and $N_{ui} \subseteq U(RD_{ui}(v^{di}))$, $i = 1,2,\ldots,q$, where q is the product of the number of times *v* is defined and the number of times *v* is used [53].

The *DU-chain* of the parameters of the illustrative example (Section 3.1) are:

$DU(\$name) = \{1\ 2\}$, $DU(\$q) = \{2\ 3\}$, $DU(\$result) = \{3\ -\}$.

It is worth mentioning that a defined variable must be used before its redefinition to be called "life", otherwise it is a code anomaly. This is not included in the rule as it does not flag any security violation.

4. System Framework

The general framework of the web application vulnerability extractor system is shown in Figure 1. Its components are discussed in the following subsections.

4.1 Front End Compiler and Static Analyzer

The front end compiler extracts a parse tree from the given application code. It lexically and syntactically analyzes the code to generate the corresponding parse tree.

The static analyzer considers the parse tree to build a flow graph of the application code that is annotated with the node number (statement number). It computes the variable's dependencies and their influences. The data dependencies and influences of the nodes variables are realized through the extraction of variables' *DEFINITIONS*, *USES* and *DECISION CONSTRUCTS* and their *life scope*. It generates a set of tables that specify, for each variable, its set of *DEFINITION* nodes, *D(v)* (where the variable was defined), USE nodes, *U(v)* (where the variable was used) and *DECISION* nodes, *DN(v)* (where a variable was used in the conditional predicate of decision node, if any), and their associated *life scope*.

4.2 Defined-Use Chain Extractor

Defined-Used Chain of each variable *v, DU(v)* is constructed from the generated data dependencies and influences in the previous stage. The *defined-use chain* Extractor utilizes the sets D*efinition, Use* and *Decision nodes* in order to generate:

- The set of *Defined-Used, DU{v}* nodes for each variable *v* in the application code.
- The set of *Decision Nodes DN{v}* nodes for each variable *v* in the application code.
- The set of nodes in the scope of *the Decision Nodes, SDN{v}* for each branching (decision) node in the application code.

4.3 Static Slicing

Static slicing is a verification technique that analyzes the application code, statically, to isolate specific statements. For the purpose of this paper, static slicing process is serving to isolate the code statements which attain vulnerabilities directly or indirectly. It extracts a portion of the program (taint code) as a set of nodes that, directly or indirectly, are affected by any external data influence caused or delivered by the user of the application. The slicer manipulates the generated set of define-use nodes *DU{v},* decision nodes *DN{v}* and those in the scope of *the DECISION* nodes *SDN{v_j},* (controlled, therefore influenced by *DN{v}*), to extract the set of nodes that constitute the vulnerable slice or the potential taint code.

4.4 Slice Refinery

The isolated static slice is optimized through the refinery slicer to obtain the refined static slice (*RSTS*). *RSTS* is the precise set of taint statements that supports a potential program breach. Static refinery excludes, from the static slice, those nodes that are influenced by the external data, yet, are not security sensitive (sink) nodes, therefore, are not assumed to be vulnerability threat (i.e. it is false positive results). Depending on the application language, sink nodes are those that permit access to or modifications of the web page internal data (e.g. a database), control the HTML output of the web application, or any sensitive functions. Also, it excludes the nodes in the scope of an influenced Decision node (*SDN*) that is not an intrusion's vulnerability.

4.5 Vulnerability Remedies

The vulnerabilities embedded in the taint code realized in the ***RSTS*** set are classified according to its type –as a server side vulnerability- to include: Buffer Overflow, Cookie Poisoning, SQL Injection and Cross Site Scripting. Some of the web application vulnerabilities, however, are embedded in the client-side portion of the application. Those

types include: Bypass Restrictions on Input Choices, CGI Parameters and Hidden Fields. These types of client side code vulnerabilities are checked in the HTML code using the corresponding *Microsoft .Net Framework* Regular Expressions [54] as shown in Table 1. The next stage thereafter is to generate a report specifying the taint statements and the proper remedy for each, to guarantee a better secured code.

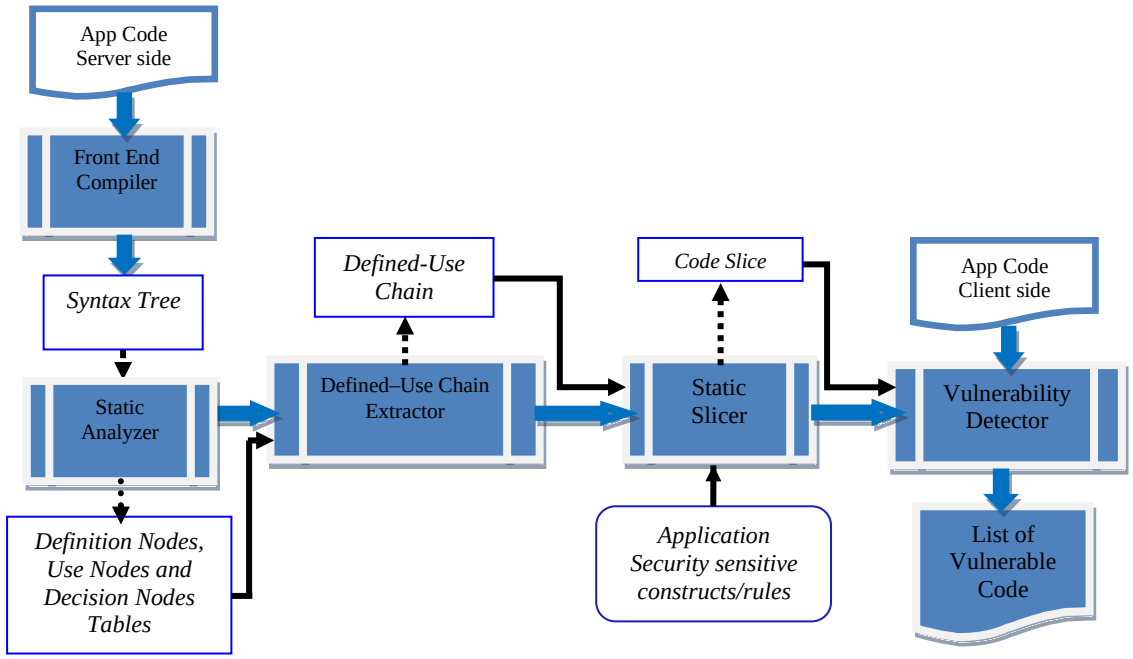

Figure 1: System Framework

5. System Architecture

This paper focuses on the vulnerabilities embedded in the code of Web application taint code). One of the most common web programming languages (**PHP**) was adopted to build a prototype of the presented framework. *PHP* is a server-side scripting language. Within an ***HTML*** page, one can embed ***PHP*** code that will be executed each time the page is visited/loaded. The ***PHP*** code is interpreted at the Web server to generate ***HTML*** or other client side web language.

The architecture of the proposed Web Application Vulnerability Extractor (*WAVE*) system is shown in Figure 2. Its components are discussed in the following subsections.

5.1 Static Analyzer

The front end compiler is merged with the static analyzer to generate the lists of *DEFINITION*, *USE* and *DECISION* nodes. Program code is lexically and syntactically analyzed and parsed through the *static analyzer*. It is composed of a lexical analyzer and a special syntax analyzer that generates the parse tree from code statements.

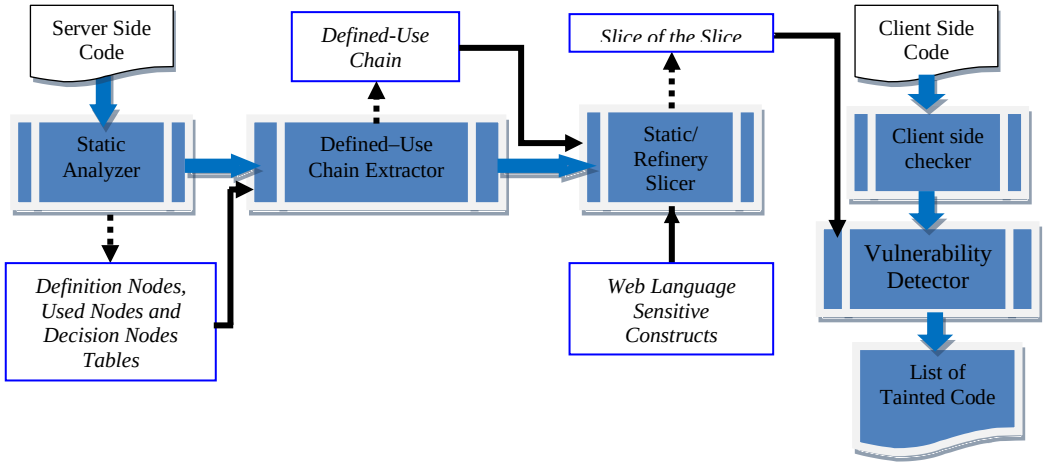

Figure 2: System Architecture

5.2 Defined-Use Chain Extractor

The *defined-use chain* Extractor (***DU-Extractor***) manipulates the D*EFINITION*, *USE* and *DECISION* nodes tables in order to generate the *list of DU{v}* nodes for every variable $v \in \{Var\}$. This ***DU*** list facilitates the computation of the variable's reaching definitions *RD(v)* rules (1, 2) in subsection 3.2. *DU*-Extractor reconstructs the *definition* and *use* tables into one Linked list indexed by the variable's name. Each entry of the linked list consists of the variable name and its list of *DU* pair of nodes associated with its life scope, using rule (1).

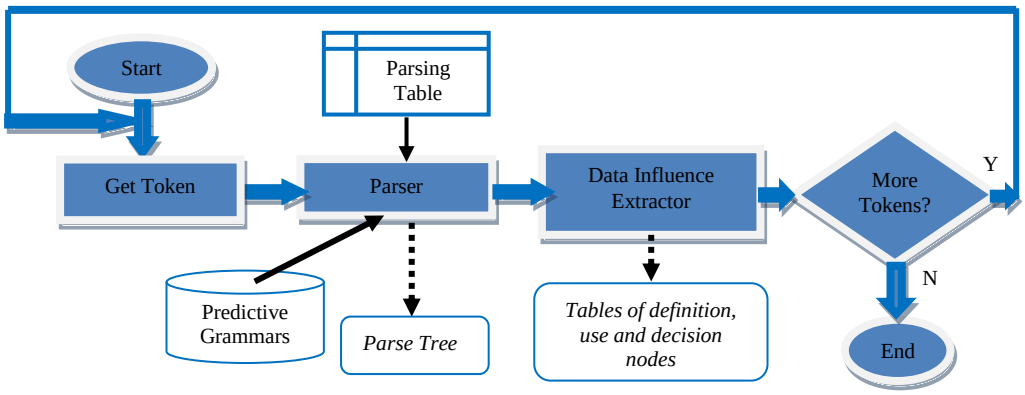

Figure 3: Syntax and data influence analyzer

5.3 Static Slicing and Refinery

The slicer manipulates the generated set of nodes *DU{v}*, decision nodes *DN{v}* and those in the scope of *the* decision nodes *SDN{v} applying rule (7)*, (controlled, therefore influenced by *DN{v}*), to extract the set of nodes that constitute the vulnerable slice.

All user-input constructs are assumed to be initial set of vulnerable code statement. Therefore, the set of vulnerable variables, *{Vul}* is the set of variables whose values are supplied externally by the user of the application, therefore could be tainted. An application of rules (4, 5) extracts the vulnerable nodes that *use* the reaching definition of any of vulnerable variables in the set *{Vul}*.

Those nodes are the initial slice $C_v(Vul, m_l)$of potential vulnerable code. Then a set of vulnerable variables is extracted from this slice using rule (6). Iteratively, rules (4, 5) then (6) are repeated till saturation. At which point $C_v(Vul, m_l)$ constitute the slice of potential taint code.

This slice of vulnerable code is optimized through the refinery slicer to obtain the refined static slice (RSTS). RSTS is the precise set of tainted statements that supports potential program breach. Static refinery module excludes, from the static slice, those nodes that are influenced, yet, does not indicate security threat. For example, the nodes in the scope of an influenced selection node (*SDN*) may be interpreted or not depending on the logic and control flow that could affects the behavior of the program, yet it might not be an intrusion's vulnerability.

Another example is the case of an influenced node that is not security sensitive; therefore, it does not constitute a threat (not a sink node). RSTS is the set of statements (nodes) that constitutes all vulnerable/tainted code that should be rectified through remedy and recommendations.

5.4 Clients-Side Vulnerability Check

The vulnerability types detected by the system so far are those in the server-side application code, including: Buffer Overflow, Cookie Poisoning, SQL Injection and Cross Site Scripting. Some of the web application vulnerabilities, however, are embedded in the client-side portion of the application. Those types include: Bypass Restrictions on Input Choices, CGI Parameters and Hidden Fields, which could be detected by checking client-side code using the corresponding *Microsoft .Net Framework* Regular Expressions [54] as shown in Table 1.

Table 1: Regular Expressions for Detecting Client-Side code Vulnerabilities

Attack Type	Regular Expressions
By-Pass Restrictions on Input Choices	(?i)<input( )+.*type( )*=( )*\"radio\"
	(?i)<input( )+.*type( )*=( )*\'radio\'
	(?i)<input( )+.*type( )*=( )*\"checkbox\"
	(?i)<input( )+.*type( )*=( )*\'checkbox\'
	(?i)<select
CGI Parameters	(?i)<form( )+.*method( )*=( )*\"get\"
	(?i)<form( )+.*method( )*=( )*\'get\'
Hidden Fields	(?i)<input( )+.*type( )*=( )*\"hidden\"
	(?i)<input( )+.*type( )*=( )*\'hidden\'

5.5 Vulnerability Remedies

The vulnerabilities embedded in the refined static slice (RSTS) are classified according to its type in order to provide the appropriate recommendations and/or remedies. At this stage, the system generates a report specifying the vulnerable statements and the proper remedy for each, to guarantee better secured code.

6. Results and Evaluation

For the purpose of evaluating the proposed *WAVE* system, a test suite was prepared. It was made up of fifty PHP codes in three categories: Codes collected from research journal papers (10 PHP codes), code examples from websites (28 PHP codes) and number of artificially generated synthetic programs (12 PHP codes).

The test suite were chosen to stress any given analysis tool, presenting it with a number of semantically complex (yet not uncommon) situations and challenging it to assess its capability to recognize and extract their intrinsic vulnerabilities. The test suite was utilized to exercise the *WAVE* system, as well as the free tools *Yasca* [26] and *RIPS* [27] for the purpose of comparison and evaluation of the presented *WAVE* system.

6.1 Results of the *WAVE* System

The *WAVE* system was exercised by the prepared test suite. The result of executing the presented system prototype under the test suite is shown in Figure 4. It extracted a total of ***87*** vulnerabilities from 50 PHP codes, which implies the existence of multiple types of vulnerability in a number of test codes. Figure 4, plots a detailed bar chart indicating the category of the test cases and the types of the extracted attack's vulnerability versus the count of each.

6.2 Comparison and Evaluation

The *RIPS* and *Yasca* tools were chosen for evaluating the *WAVE* system because they are free and easy to be configured. Both tools supported two of the main types of vulnerabilities, the SQL injection and cross site scripting. A subset of the test suite containing *26* codes was applied to compare between the *WAVE* system capabilities and those of *RIPS* and *Yasca* tools. Figure 5, illustrates two dimensional plot for the two types of attacks vulnerabilities, SQL Injection and Cross site Scripting with their test case categories (papers, web sites and synthetics), versus the count of the detected vulnerabilities for the tools *Yasca*, *RIPS* and the system under evaluation (*WAVE*).

6.3 Discussion

Some of the interested test cases are discussed to investigate the evaluation results. The PHP code depicted in test case 1, [55] was successful test case as recognized by Yasca, *RIPS* and *WAVE* system. It is an example of *cross-site scripting* vulnerability that was detected by all three.

Test case 1: [55]

```
1.  <?php echo "You searched for: " .$_GET["query"]; ?>
```

Test case 2, [56], however was a success for *RIPS* and *WAVE* but failed for *Yasca*. It is another example that endures cross-site scripting vulnerability. It seems that *Yasca* tool does not consider the indirect influence of the external data.

Test case 2: [56]

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

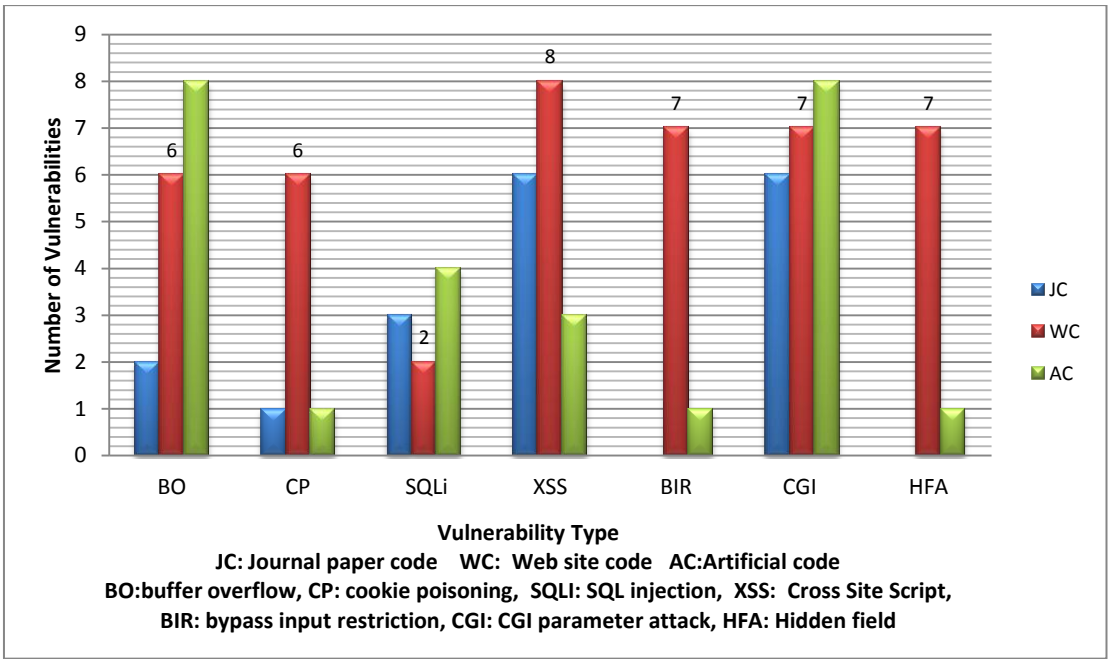

Figure 4: Vulnerabilities extracted by the WAVE System

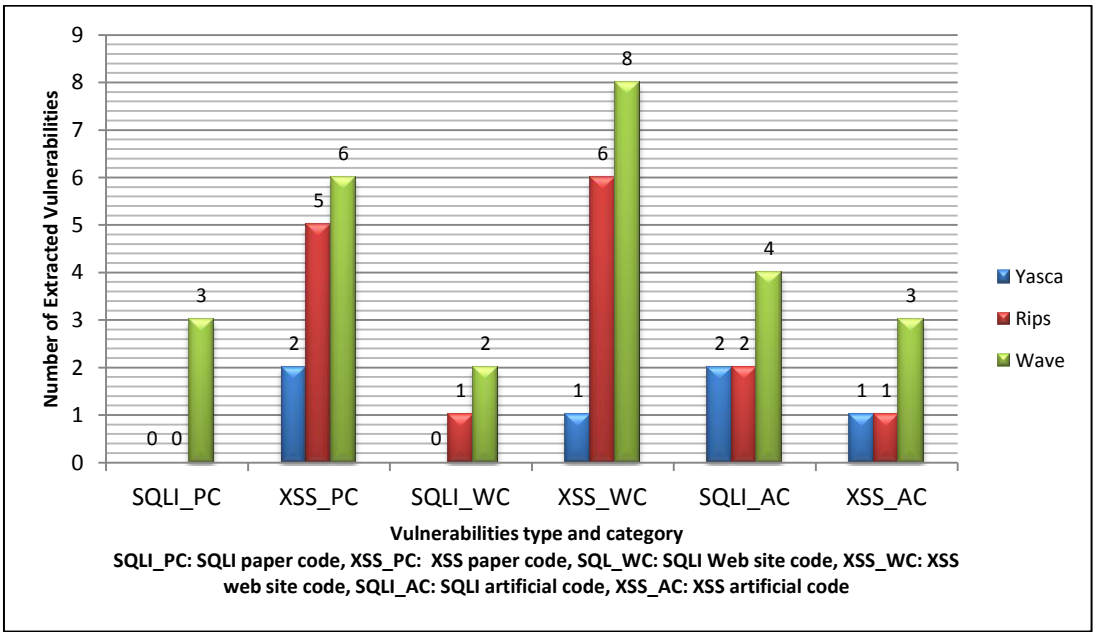

Figure 5: Tainted code detected by RIPS tool, Yasca tool and WAVE System

Yet, test case 3, [57] was a success for, only, *WAVE* system. It is another example with a Cross-site scripting attack vulnerability. In statement *8*, the user input *"user_comment"* is inserted in a SQL query (to be inserted into a database). It could be the case that both *Yasca* and *RIPS* tools consider the *cross site scripting* vulnerability, only, for a visible input in a direct output statement, both tools however, did not consider this vulnerability.

Test case 3: [57]

```
1.  <form method ="get"> <input name="user_comment" type="text">
2.  <input type="submit" name="Submit1"> </form>   3.   <?php
4.  $con = mysql_connect("localhost","root","");   5.   if (!$con)  { die('Could not connect: ' . mysql_error()); }
6    mysql_select_db("comments", $con);            7.   if ( isset($_REQUEST['Submit1'] ) )
8   { mysql_query("INSERT INTO comments  (comment) VALUES  (".$_REQUEST['user_comment'].")",$con);}
9   $result = mysql_query("SELECT * FROM comments",$con);   10.  echo  $row['comment'];   echo "<br />";
    mysql_close($con); ?>
```

Test case 4 depicts an example for *SQL injection* vulnerability for potential attacks, induced by the influenced variables *$m* and *$y*. *WAVE* system flagged statement *4* as a vulnerability because of the indirect influence of the input variables in the *SQL* command. *Yasca* tool could not detect this vulnerability, because the variables involved in the catenation (statement 4), are not direct inputs (*$m* and *$y* are defined by '$user' and $pass, respectively). Also, both *Yasca and RIPS* tools failed to detect this as vulnerability because the SQL statement is passed to the database server, from a different page.

Test case 4: Artificial code

```
1.  <?php   $user=$_POST['username'];      2.  $pass=$_POST['password'];  3. $m=$user; $y=$pass;
3.  $sql="SELECT count(*) FROM users WHERE  username= $m' AND password = '$y'";    ?>
```

The major vulnerability in the test cases 5-to- 8, was the indirect influence imposed by the external variable, supplied by the user of the application. It induces vulnerability for potential SQL injection attacks. This influence was detected by neither Yasca nor RIPS.

Test case 5: [57]

```
1.   <?php        2.    $sql="select * from usermaster where username='{$_POST['username']}'";  ?>
```

Test case 6: [58]

```
1.  <?php        2.    $show=$_POST['username'];
3.  $query = "SELECT  usernames FROM users WHERE usernames LIKE '".$show."%'  ";  ?>
```

Test case 7: Artificial code

```
1.  <?php        2.    $id = $_POST['id'] ;
```

```
3.  $query = "SELECT id, title, content FROM news WHERE id = $id" ; ?>
```

Test case 8: [59]

```
1.  <?php  2.  $HTTP_REFERER =$_POST['value'];
3.  $sql="INSERT INTO tracking_temp VALUES('$HTTP_REFERER');"; mysql_query($sql,$con);  ?>
```

As shown in Figure 5, *Yasca*, tool performed poorly due to its incapacity of handling the vulnerabilities caused by indirect influence of externally provided data (inputs). *RIPS* tool, however, were capable of detecting the direct and indirect inputs for Cross Site Scripting attack vulnerability. Both tools were capable of detecting Cross Site Script whenever user inputs reaches output statement, but not when it was written into the database. According to the evaluation test, it might be the case that neither *Yasca*, nor *RIPS* tools were considering the indirect influences of the external data that could potentially induced SQL injection attacks.

7. Conclusion

Whether a software developer is maintaining a legacy web application or building a new one, security is a crucial aspect. The aim of this research paper was to expose the code dependent web vulnerabilities that cause security breaches. Most of the important web application code vulnerabilities were examined, specially, those concerned with communicated external data provided by a user of the application. An adaptation of one of the famous testing technique is established to isolate the vulnerabilities of web application code. The technique of the static slicing was applied to isolate program constructs that attain vulnerabilities directly or indirectly. A prototype for the presented framework provided a code oriented vulnerability Extractor with recommendations for mending the vulnerable/tainted code against the corresponding security breaches.

Web application vulnerabilities, including buffer overflow, cookie poisoning, SQL injection and cross-site scripting, were detected by analyzing the server-side code using static slicing analysis technique. The Bypass restrictions on input choices, CGI parameters and hidden field vulnerabilities, however, cannot be detected by analyzing the server-side code alone. Rather, they could be detected by checking the client-side code with the appropriate regular expressions.

Typically, there is a gap between developers, who build and maintain web application and the security personnel, who help them in making the application more secure. The proactive nature of the proposed system, however, will help to avoid this gap.

Applying the presented system, during Web application development cycle and/or maintenance of a legacy application, will assist the developers and provide them with agile understanding of the security breaches and its causes. The provided recommendations suggest better style and proper- security preserving- utilization of the programming constructs, through an empirical study of the constructed software.

The presented system is also applicable for integration testing, delivery testing and deployment testing (against the investigated types of attacks).

Generally speaking, web application code dependent vulnerabilities are neither intrinsic, nor amenable to the cleverness of the intruder. It could, always, be avoided by skilled and experienced application's developers aided by security policies and supporting tools.

References

[1]. M. Andrews, J.A. Whittaker, "How to Break Web Software," Addison-Wesley Professional, USA, 2006.

[2]. D. Melnichuk, "The Hacker's Underground Handbook," Create Space Independent, USA, 2010.

[3]. R. L. Krutz and R. D. Vines, "The Comprehensive Guide to Certified Ethical Hacking," Wiley, Indiana, 2007.

[4]. J. Bird, E. Johnson and F. Kim, "2015 State of Application Security: Closing the Gap," InfoSec Reading Room SANS Institute, May 2015.

[5]. D. Volpano, C. Irvine and G. Smith, "A Sound Type System for Secure Flow Analysis," JCS, vol. 4, pp. 167-187, 1996.

[6]. A. C. Myers. "JFlow: Practical Mostly-static Information Flow Control." In POPL, 1999.

[7]. U. Shankar, K. Talwar, J. S. Foster and D. Wagner, "Detecting Format String Vulnerabilities with Type Qualifiers." In USENIX Security Symposium, 2001.

[8]. P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz and V. N. Venkatakrishnan "NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications." In Proc. of the ACM Conf. on Computer and Communications Security, 2010.

[9]. S. Rafique, M. Humayun, B. Hamid, A. Abbas, M. Akhtar and K. Iqbal, "Web application security vulnerabilities detection approaches: A systematic mapping study." In Proc. of the 16th Int. Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), IEEE/ACIS, Takamatsu, Japan, 2015.

[10] A. Garg and S. Singh,"A Review on Web Application Security Vulnerabilities," Int. Journal of Advanced Research in Computer Science, and Software Engineering, vol. 3, pp. 222-226, 2013.

[11] S. Mairdula and D. Manivannan, "Security Vulnerabilities in Web Application – An Attack Perspective," Int. Journal of Engineering and Technology (IJET), vol. 5, pp. 1806-1811, 2013.

[12] A. Kiezun, P.J. Guo, K. Jayaraman, M.D. Ernst, "Automatic Creation of SQL Injection and Cross-Site Scripting Attacks", Computer Science and Artificial Intelligence Laboratory, Massachusetts institute of technology, Cambridge, MIT-CSAIL-TR-2008-054, USA, 2008.

[13] M. Martin, and M.S. Lam, "Automatic Generation of XSS and SQL Injection Attacks." In Proc. Of The Advanced Computing System Association Conf., Canada, 2007, pp. 31-43.

[14] F. Dysart and M. Sherriff, "Automated Fix Generator for SQL Injection Attacks." In Proc. 19th Int. Symposium on Software Reliability Engineering (ISSRE), Seattle, WA, 2008, pp. 311-312.

[15] A. Guha, S. Krishnamurthi and T. Jim, "Using Static Analysis for Ajax Intrusion Detection," In Proc. Int. World Wide Web Conf. : Web Security, Madrid, Spain. April, 2009, pp. 561-570.

[16] Y. Shin, L. Williams and T. Xie, "Test Case Generation for SQL Injection Detection," North Carolina State University Department of Computer Science, TR-2006-21, USA, 2006.

[17] N. Jovanovic, E. Kirda and C. Kruegel, "Preventing Cross Site Request Forgery Attacks." In Proc. Second Int. Conf. on Security and Privacy in Communication Networks, USA, 2006, pp. 1-10.

[18] K. Wei, M. Muthuprasanna and S. Kothari, "Preventing SQL Injection Attacks in Stored Procedures," In Proc Australian Software Engineering Conf. (ASWEC'06), Australia, 2006.

[19] J. Bau, E. Bursztein, D. Gupta and D. J. Mitchell,"Automated Black-Box Web Application Vulnerability Testing," Security and Privacy (SP), IEEE Symposium, USA, 2010, pp. 332 - 345.

[20] H. Shahriar and M. Zulkernine "Automatic Testing of Program Security Vulnerabilities." In Proc. 33rd Annual IEEE Int. Computer Software and Applications Conf., USA, 2009, pp. 550-555.

[21]. T. Sultan, S. Kholeif and M.A. Sultan, "Securing Web Application Against SQL Injection Attack," Egyptian Computer Science Journal, Egypt, 35(3), pp. 11-22, 2011.

[22] K. Umar, A.B. Sultan, H. Zulzalil, N. Admodisastro and M.T. Abdullah, "On the Automation of Vulnerabilities Fixing for Web Application." In Proc. of The 9th Int. Conf. on Software Engineering Advances (ICSEA), 2014, pp. 221-226.

[23] Y. Minamide, "Static Approximation of Dynamically Generated Web Pages." In Proc. of Int. World Wide Web Conf. Committee (IW3C2), Japan, 2005, pp. 432 – 441.

[24] NOTAMPER Supplementary, http://sisl.rites.uic.edu/notamper, last retrieved on July, 2015.

[25] The Official Website of Backtrack http://www.backtrack-linux.org/, last retrieved on July, 2015.

[26] Yasca Tool. Available at http://www.scovetta.com/yasca.html, last retrieved on July, 2015.

[27] J. Dahse, "RIPS - A static source code analyzer for vulnerabilities in PHP Scripts," Horst Görtz Institute, Ruhr University Bochum, German, 2010.

[28] WAVSEP Web Application Scanner Benchmark 2014, available at: http://sectooladdict.blogspot.com/2014/02/wavsep-web-application-scanner.html

[29] Vulnerability Scanning Tools, available at: https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools

[30] W. Chang, B. Streiff and C. Lin, "Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis." In Proc. of the 15th ACM Conf. on Computer

and Communications Security (CCS'08), NY, USA, 2008, pp. 39-50.

[31] J. Newsome and D. Song. "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software." In Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS'05), Feb. 2005.

[32] B. Xu, J. Qian, X. Zhang, Z. Wu and L. Chen, "A brief survey of program slicing," ACM SIGSOFT Software Engineering Notes, NY, USA, vol. 30, pp. 1-36, 2005.

[33] G. Agosta, A. Barenghi, A. Parata and G. Pelosi, "Automated Security Analysis of Dynamic Web Applications through Symbolic Code Execution." In Proc. of the 9th Int. Conf. in Information Technology: New Generations (ITNG'12), April 2012, pp.189-194.

[34] O. Tripp, M. Pistoia, P. Cousot, R. Cousot and S. Guarnieri, "Andromeda: Accurate and scalable security analysis of web applications," Fundamental Approaches to Software Engineering, Ed. Vittorio Cortellessa and Dániel Varró, ch.15, pp. 210–225, 2013.

[35] J. A. Goguen and J. Meseguer. "Security Policies and Security Models." In Proc. of the IEEE Symposium on Security and Privacy, 1982, pp. 11-20.

[36] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. "Detecting Format String Vulnerabilities with Type Qualifiers." In Proc. of USENIX Security Symposium, 2001.

[37] A. C. Myers and B. Liskov, "A Decentralized Model for Information Flow Control." In Proc. of the IEEE Symposium on Security and Privacy(S&P), California, USA, 1998, pp. 1-12.

[38] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng and X. Zheng, "Secure web applications via automatic partitioning." In Proc. of 21st ACM SIGOPS Symposium on Operating systems principles (SOSP'07), NY, USA, 2007, pp. 31-44. Doi:10.1145/1294261.1294265

[39] K. Ashcraft and D. Engler, "Using Programmer-Written Compiler Extensions to Catch Security Holes." In Proc. of the IEEE Symposium on Security and Privacy (S&P), 2002, IEEE Computer Society Press.

[40] M. Pistoia, R. J. Flynn, L. Koved and V.C. Sreedhar, "Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection," ECOOP, LNCS 3586, Springer Verlage, Berlin, pp. 362–386, 2005.

[41] G. Snelting, T. Robschink and J. Krinke, "Efficient Path Conditions in Dependence Graphs for Software Safety Analysis," ACM Trans. on Software Engineering and Methodologies (TOSEM), vol. 15(4), pp. 410-457, 2006.

[42] C. Hammer, J. Krinke and G. Snelting. "Information Flow Control for Java Based on Path Conditions in Dependence Graphs." In the IEEE Symposium on Security and Privacy(S&P), 2006.

[43] V. B. Livshits and M. S. Lam. "Finding Security Vulnerabilities in Java Applications with Static Analysis." In Proc. of the 14th USENIX Security Symposium, MD. US, 2005.

[44] J. Whaley and M. S. Lam, "Cloning Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams." In proc. Programming Language Design and

Implementation (PLDI'04), DC, USA, 2004.

[45] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for Java." In Proc. of the 27th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'06), NY, USA, 2006, pp. 387-400.

[46] O. Lhot´ak and L. J. Hendren, "Context-Sensitive Points-to Analysis: Is It Worth It?" In Int. Conf. on Compiler Construction, 2006.

[47] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby and S. Teilhet, "Saving the World Wide Web from Vulnerable JavaScript." In Proc. Int. Symposium Software Testing and Analysis, ON, Canada, July 2011. Download from: http://sammyg.org/Work/Actarus/actarus.pdf

[48] L. O. Andersen. "Program Analysis and Specialization for the C Programming Language." PhD. Thesis, University of Copenhagen, Copenhagen, Denmark, May 1994.

[49] G. Wassermann and Z. Su. "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities." In Proc. Of the 28th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'07), NY, USA, 2007, pp 32-41.

[50] T. Tateishi, M. Pistoia, and O. Tripp. "Path- and Index-sensitive String Analysis Based on Monadic Second-order Logic," ACM Transactions on Software Engineering and Methodology, NY, USA, 22(4), no. 33, 2013.

[51] S. McCamant and M. D. Ernst, "Quantitative Information Flow as Network Flow Capacity " In Proc. of the 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'08), NY, USA, 2008, pp. 193-205.

[52] I. Parameshwaran, E. Budianto S. Shinde, H. Dang A. Sadhu and P. Saxena, "Auto-patching DOM-Based XSS at Scale." In Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15), Bergamo, Italy, 2015.

[53] B. Beizer, "Software Testing Techniques." Second Edition, New York. USA, 2003.

[54] MSDN.NET Framework Regular Expressions. Available at: https://msdn.microsoft.com/en-us/library/az24scfc%28v=vs.110%29.aspx

[55] Cross-Site Scripting Attacks. Available at: http://www.sitepoint.com/php-security-cross-site-scripting-attacks-xss/

[56] Y. W. Huang, F. YU, C. Hang, C. H. Tsai, D. T. Lee, S.Y. Kuo, "Securing Web Application Code by Static Analysis and Runtime Protection." In Proc. of the 13th Int. Conf. on World Wide Web, USA, 2005, pp. 40–52.

[57] How to add data into sql db using AJAX and PHP. Available at: http://stackoverflow.com /questions/13456101/how-to-add-data-into-sql-db-using-ajax-and-php

[58] S. P. Chandran and M. Angepat, "Comparison between ASP.NET and PHP Implementation of a Real Estate Web Application." Thesis, School of Innovation, Design and Engineering, Sweden, 2011.

[59] The Open-source PHP Compiler, http://www.phpcompiler.org/