A Hybrid Runtime Approach to Combating High-Level Semantic Attacks

A Proposal Defense

Presented in Partial Fulfillment of the Requirements for the

Degree of Doctor in Philosophy

with a

Major in Computer Science

in the

College of Graduate Studies at

The University of Idaho

by

Stu Steiner

Sept 2013

Major Professor: **Jim Alves-Foss**, Ph.D.

Abstract

Over the past few years, there has been a rapid increase in interactive web sites that offer a wide range of services including shopping carts, e-commerce transactions, social media interactions and much more. A majority of these websites contain a database backend where user information, and product information is stored and then retrieved when needed.

The Open Web Application Security Project (OWASP) maintains a list of the top ten vulnerabilities. This top ten list coincides with OWASP's mission to make software security visible to all, so individuals and organizations can make intelligent decisions about software security.

One challenge inherent with software security is how to combat injection vulnerabilities, such as SQL injection. Injection vulnerabilities occur when untrusted data is passed to the database interpreter without proper sanitization. Passing untrusted data to the interpreter can allow for exploits in the web application, such as revealing sensitive user information, data corruption, denial of access or complete host takeover. Code that properly sanitizes the untrusted data can easily combat injection vulnerabilities; however, an inexperienced developer may not understand or have knowledge of this concept or how to implement security measures.

Current techniques to combat injection vulnerabilities are typically classified into one of three categories, static analysis, dynamic analysis or a combination of both static and dynamic analysis. Static analysis involves verifying that the untrusted data is safe before it is passed to the interpreter. This is usually accomplished with input validation via string tokenization, control-flow graphs and many other techniques. Dynamic analysis, such as dynamic information flow tracking (DIFT), involves executing the interpreter with the untrusted data in a controlled environment and then examining the results of the execution. Static analysis is often ineffective because attackers are continually discovering new attack vectors that are able to avoid the checks static analysis provides. Dynamic analysis is also often ineffective because the rigorous overhead can ultimately slow the system in unacceptable ways. There is limited research on combining both dynamic and static analysis techniques. This dissertation proposes

the development of a hybrid approach that combines the best aspects of static and dynamic analysis while avoiding their individual shortcomings. This is accomplished by summarizing the static and dynamic analysis schemes proposed in recent years; introducing a hybrid approach, combining static analysis and dynamic analysis in a unique way limiting interaction from the developer; and, proposing new research to implement, evaluate and improve the detection and avoidance of injection vulnerabilities.

A goal of this project is to formalize the definition of injection vulnerabilities. In addition this dissertation discusses the formal model of injection vulnerabilities and how these vulnerabilities are handled by the interpreter and the hardware, including a policy-based architectural mapping framework. Finally, the remaining work of the dissertation is proposed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In 2011 the total sales for e-commerce transactions in the United States was 194 billion US dollars [32], with an estimate of at least a 16% growth in e-commerce transactions in future years. With such a high dollar volume occurring from e-commerce transactions, the security of those web transactions is a major concern to any Internet user. To complicate matters, the introduction of smart mobile devices and tablet computers is also having an impact on e-commerce and web transactions. In recent years there has been extensive research conducted related to e-commerce and web security; however, the problem of securing web transactions still exists. The problems with e-commerce and web security are often related to Structured Query Language (SQL) injection, cross-site scripting (XSS), cookie manipulation, and Uniform Resource Locator (URL) redirection [20, 32].

The term "web security" has many different meanings depending on the context. In October 2000, Bruce Schneier wrote a newsletter article for "Crypto-Gram Newsletter" where he provided commentary on different levels of network/security attacks.

In this article he wrote, "The first wave of attacks was physical: attacks against the computers, wires, and electronics. Over the past several decades, computer security has focused around syntactic attacks: attacks against the operating logic of computers and networks. This second wave of attacks targets vulnerabilities in software products, problems with cryptographic algorithms and protocols, and denial-of-service vulnerabilities – pretty much every security alert from the past decade. The third wave of network attacks is semantic attacks:

attacks that target the way we, as humans, assign meaning to content." [25] The levels of attack that Schneier described were paraphrased from the work of Martin C. Libicki [12].

In this article Schneier simply stated, "People are already taking advantage of others' naivete. ... Computer networks make it easier to start attacks and speed their dissemination, or for one anonymous individual to reach vast numbers of people at virtually no cost. ... semantic attacks will be more serious than physical or even syntactic attacks, because semantic attacks directly target the human/computer interface, the most insecure interface on the Internet [25]."

Typically, an individual (user) accesses an Internet web page to conduct a transaction. During this transaction, the user will be required to provide sensitive information, such as a credit card number. The website where the user provides this information should be a secure web server, which is nothing more than a computer that uses encryption to safely communicate and store sensitive information. Since the merchant cannot access credit card information directly, the merchant will most likely use a payment gateway. This payment gateway has the responsibility of determining the credit card company, and then contacting it via the credit card company's secure server. The credit card company will examine the transaction against stored account information, and the credit card company will either authorize or deny the transaction. The credit card company will pass this information back through the payment gateway. The payment gateway will then provide an acceptance or a denial back to the merchant's secure web server. This whole web transaction is considered secure if the information within the system cannot be accessed and/or modified by unauthorized users [6].

Web application vulnerabilities are tracked by The Open Web Application Security Project (OWASP) [20]. OWASP's mission is to make security visible, and to educate individuals about vulnerabilities and defenses. OWASP maintains a top ten list of vulnerabilities, where a vulnerability is a weakness or flaw in the application that allows an attacker to cause harm. According to OWASP, the top ten vulnerabilites [20] are:

1. Injection - Injection occurs when untrusted data is sent to the target interpreter and that untrusted data allows for an exploit in the interpreter. This exploit can result in data loss,

data corruption or even complete system takeover. Injection flaws are very prevalent, particularly in legacy code, such as code found in SQL queries, operating system (OS) commands, and many other systems.

2. Cross-Site Scripting (XSS) - XSS is currently the most prevalent web application security flaw. XSS flaws occur when an application sends untrusted content or non-sanitized content to a web browser. XSS flaws allow the attacker to execute selected code on the victim's browser. There are three known types of XSS flaws: 1) Stored, 2) Reflected, and 3) Document Object Model (DOM) based XSS.
    - Stored attacks are those where injected code is permanently stored on target servers; such as in a database. The victim retrieves the malicious code from the server when it requests stored information.
    - Reflected attacks occur when code is injected into the web server through means such as an error message. Reflected attacks are typically delivered to a victim via another route, such as in an e-mail message, where the user is tricked into clicking on a malicious link. The injected code travels to the vulnerable web server, which reflects the attack back to the victim's browser. The browser executes what appears to be trusted code; however, the code being executed is malicious.
    - DOM XSS occur when the attack is executed as a result of modifying the DOM in the user's browser. This will cause the client side code to run in an unexpected manner. The page the user sees does not change, but the client side code contained in the page executes differently due to the attack.
3. Broken Authentication and Session Management - User authentication is common for many web applications. Broken Authentication and Session Management attacks occur when authentication and session management schemes do not adequately protect user credentials for all aspects of the site. The flaws in these schemes frequently cause problems with logout, password management, timeouts, and account updates.
4. Insecure Direct Object References - Applications frequently use the actual name of an

object when generating web pages. Applications that don't verify the user, could allow an insecure direct object reference flaw.

5. Cross-Site Request Forgery (CSRF) - CSRF is an attack that forces an authenticated user to execute unwanted actions on a web application. When browsers send credentials, such as session cookies, attackers create malicious web pages which generate malicious requests that are indistinguishable from legitimate ones. A successful CSRF attack can compromise the user's data, such as changing the victim's password or email address.
6. Security Misconfiguration - Security misconfiguration occurs when the attacker gains unauthorized access because the system was not configured properly. Attacks can occur via access to default accounts, or by exploiting unpatched vulnerabilities. Security misconfiguration can happen at all levels of the application process, including the web server, application server, and code framework.
7. Insecure Cryptographic Storage - Insecure Cryptographic Storage occurs when data is not encrypted when it should be. If encryption is used, and the key generation is unsafe, then it is easy to determine the encryption key. Insecure Cryptographic Storage failure frequently compromises data that should have been encrypted.
8. Failure to Restrict URL Access - Failure to Restrict URL Access occurs when web pages do not properly protect page requests; commonly caused by incorrect page configurations. An attacker can gain access to the system by changing the URL to a privileged page URL. This could allow the attacker to access privileged account information.
9. Insufficient Transport Layer Protection - Insufficient Transport Layer Protection occurs when web applications do not protect network traffic, by not correctly requiring authentication. Without authentication, data and session variables can be exposed.
10. Un-validated Redirects and Forwards - Unvalidated Redirects and Forwards occur when an application redirects users to other pages, without validating the target page. This type of flaw can allow the attacker to force the victim's browser to open a specified, possibly malicious, web page.

In order to remove the chance of vulnerabilities, it is up to the developer to understand them and then properly code to protect against them. Unfortunately, expecting the developer to code against vulnerabilities is also problematic. The developer may not know that vulnerabilities even exist; or worse yet, the developer knows there are vulnerabilities, but chooses to ignore them. In either case some other mechanism needs to be pursued to secure the web application from vulnerabilities. One such mechanism is runtime enforcement which is explained in the following section.

1.1 Runtime Enforcement

The concept of runtime enforcement deals with modeling runtime mechanisms via automata (self operating machine). Most runtime mechanisms involve truncation automaton which only halt the monitored program, upon software violation [15]. In terms of web security, attempting to halt the transaction sometimes often reveals more information instead of less. This is common in many web pages where there has been a security misconfiguration. This security misconfiguration will often display information about the server and the database, instead of displaying a simple error message. Figure 1.1 illustrates an asp.net error page. The page shows the stack trace, the database name and other information that could be used in a semantic attack. This is an example of a security misconfiguration vulnerability. The information displayed on the web page could have been hidden from the user by changing the configuration file [18]. In this case, a Security Misconfiguration vulnerability on the asp.net page shows the stack trace and database name, revealing more information that could be used to compromise this website.

Instead of halting, it is better to return a result, and have that result evaluated before the next action is processed. This is the essence of edit automata [13]. Edit automata use runtime monitors to continuously monitor the running system checking for correct execution. By using edit automata runtime monitoring there is a possibility to increase the assurance of correct

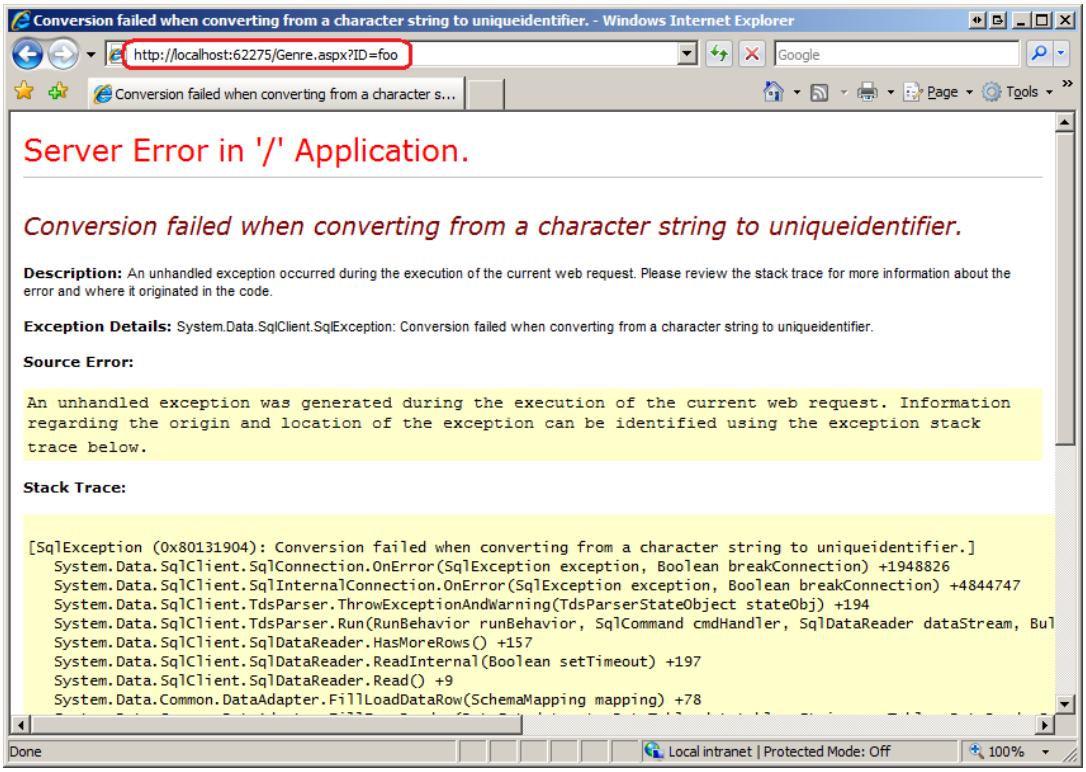

Figure 1.1: An example of a truncation automaton that halted execution.

execution. During monitoring one of three possibilities can occur:

- The software is well written and correct execution occurs. The edit automata will allow the software to run normally, because there are no errors or no attacks.
- The monitor observes software behavior that is not correct, and the monitor stops the software from executing incorrectly, as with the transaction automata. In the case of the asp.net page, the edit automata determined an error existed and halted.
- The monitor observes and changes the requested action. This occurs by the monitor inserting actions into the code or by halting. The problem with monitoring is that if the action being monitored is nondeterministic then the monitor will fail [14, 15, 24].

In order for runtime monitoring to be successful, there must be defined security mechanisms. This can be very difficult for web applications, since there are a large number of different vulnerabilities that can be exploited and those vulnerabilities can be nondeterministic.

There has been a significant number of defined security mechanism attempts to prevent

web application vulnerabilities. Those mechanisms are typically broken into two categories, static or dynamic. Static mechanisms typically involve scanning code for known vulnerability patterns. Static analysis is often conducted during software development [34, 36]. Dynamic analysis involves tracking the flow of tainted commands and values as the program is executing. If a tainted command attempts to access an area (memory, functions, etc) where access for that tainted value should not be granted then an exception is often raised [8, 9]. How that exception is handled is left to the program. In either instance there are significant drawbacks, such as large runtime overhead, or a large number of false positives. Since web application vulnerabilities are ever changing with new vulnerabilities constantly being discovered, it is very difficult to implement a solution.

1.2 Policy Specification Language

A Policy Specification Language (PSL) is a language in which written policies can be expressed. In order to solve the problem of web application vulnerabilities, a set of dynamic security policies will need to be written. There have been significant numbers of PSLs proposed and developed, and some of these PSLs are written for tracking very specific policies for specific high level languages such as PHP or Java [10, 34]. Some of these PSLs are written in a more generalized manner to allow for application over various environments [13–15, 17, 22, 33, 38]. Very few of the languages are written in a way that allows for a combination of hardware and software to handle the security policies. Chapter 2 provides a review of PSLs as implemented in dynamic and static runtime monitoring.

Since web vulnerabilities are so dynamic and current runtime monitoring techniques have shown inconsistent results; there is a need to transform a high-level abstract PSL into a low-level concrete PSL. Once that transformation is complete, then security mechanisms will need to be implemented. The security mechanisms will allow enforcement of the transformed policies.

1.3 System Architecture

In order to apply security mechanisms, a decision has to be made concerning where the mechanisms should reside. In most cases, current research has applied the security mechanism at the software application level [8–10, 17, 33, 34, 36]. This work proposes creating a PSL that will define security mechanisms that use both static analysis and dynamic analysis in a hybrid manner. These multiple security mechanisms will be used to investigate multiple patterns of web vulnerabilities. Investigating the multiple patterns at different levels of the software hierarchy, allows for a mapping of those levels within the software architecture. Understanding the patterns and the mapping of the software architecture allows for the construction of the hybrid framework. The deployment of a policy-based hybrid mapping framework allows for decomposing, aggregating and handling of web vulnerabilities, including injection vulnerabilities such as SQL injection.

1.4 Problems

The policy-based hybrid mapping framework, which leverages both software support and hardware architectural support to handle web vulnerabilities, is very complex, and is not well supported by current security engineering techniques. Based on the OWASP classification, it is common knowledge that SQL injection and cross-site scripting are two of the web vulnerabilities that occur most frequently; there are limited non-formalized mechanisms to combat these vulnerabilities. Since there are limited mechanisms with limited results, it is important to solve the problem of creation and implementation of general security policies in both the software and the hardware architectural domains.

The research of this dissertation will focus on three problems:

1. How is a high-level semantic attack web vulnerability defined?
2. Can a technique be developed that will handle a high-level semantic attack within the

framework of this hybrid approach, using the security mechanisms of the PSL?

3. What does the policy-based hybrid mapping framework represent, and from that policy-based hybrid mapping what should be allowed and what should be denied?

Since policy-based hybrid mapping framework is complex and still in its infancy, this research will provide solutions to the problems previously discussed.

1.5 Motivation and Contributions

The goal of this dissertation is to provide a policy-based hybrid mapping framework for the enforcement of security policies at both the software and the architectural level. This dissertation will achieve the following specific objectives:

Objective 1: Provide a clear definition of web vulnerabilities including defining high-level semantic attacks. This definition will be the basis for defining the policy-based hybrid patterns.

Objective 2: Provide a set of policy-based hybrid patterns for decomposing, aggregating and handling of web vulnerabilities, including injection vulnerabilities such as SQL injection. These patterns will define a policy-based hybrid mapping framework. This framework, including the security policies, will determine which transactions will be allowed and which transactions will be denied.

Objective 3: Develop an algebraic framework for rule predicates and policies for the formal analysis of the policy-based hybrid mapping framework. This algebraic framework will provide a formal means for detecting policy conflicts.

Objective 4: Verify the proposed approach by applying the policies and security mechanisms to a customized hardware system executing a customized dynamic system, such as Real-Time Executive for Multiprocessor Systems (RTEMS), including a web service application system. The policy-based hybrid mapping framework is applied at different levels of the hardware and

custom RTEMS and web service. The RTEMS system is being developed by Cindy Song [29] and the hardware is being developed by a research group at Cornell University. Each step is verified and each security mechanism is analyzed to ensure the web vulnerability and the high-level semantic attacks are properly handled.

Objective 5: Verify the proposed approach by applying the policies and security mechanism to a customized interpreter engine, such as the PHP interpreter. The policy-based hybrid mapping framework is applied at different levels of the interpreter. Each step is verified and each security mechanism is analyzed to ensure the web vulnerability and the high-level semantic attacks are properly handled.

In summary, the work presented in this dissertation provides a practical security engineering framework that will benefit programmers and system architects.

1.6 Dissertation Proposal Overview

The rest of this proposal is organized as follows: Chapter 2 provides a review of the relevant research, including definition and analysis of web vulnerabilities, static runtime monitoring and dynamic (taint) runtime monitoring.

Chapter 3 provides a set of proposals and outcomes, including information concerning the policy-based hybrid mapping framework.

Chapter 2

Background

Web security has been, and continues to be an area of active research. Web security is extremely relevant as new technologies and hardware devices arrive on the scene, and there continues to be a shortage of qualified web programmers to keep up with the increasing demand. Although there are many protection mechanisms in place for combating web vulnerabilities, many of the textbooks lack an explanation of the web vulnerability or the protection mechanisms. With such a shortage of qualified programmers, and scarce coverage of the protection mechanisms, web vulnerabilities will exacerbate into the future. This dissertation focuses on protection mechanisms not only at the software level, but also at the computer hardware level. This will allow for some protection for the programmer, even if the programmer is unaware of the vulnerabilities or the protection mechanisms. This chapter sets the stage for the rest of the dissertation by providing definitions and discussions of relevant terms and concepts.

2.1 Security Mechanisms

In order for any web policy mechanism, either hardware or software related, to be effective it must meet some basic metrics. Those metrics include:

- **Correctness/Completeness**: The security mechanism needs to protect against vulnerabilities; yet it must provide that protection with a minimal number of false positives and/or false negatives. Security mechanisms such as control data (data that is loaded

into the program and is used to determine control-flow transfer), have been easy to circumvent or have inadvertently leaked information. The problem with program control is the large number of "false alarms" in information flow tracking [1].

- **Performance**: The security mechanism needs to provide protection with a minimal increase in execution time. In the Bond et al. [3] paper concerning Precise, Efficient, Context-sensitive, Anomaly detection (PECAN) the average performance overhead is 5%. There is no basis whether 5% is acceptable, it is just a factor that must be considered when implementing security mechanisms.
- **Adaptability**: Web attacks are extremely dynamic and constantly changing. It is important for the protection mechanism to have the ability to adapt with the evolving threats. In many cases, the security mechanism is written for a particular web language such as PHP or Java. This can be problematic in that web pages often use many different web languages and by applying a security mechanism to only one specific language leaves the other languages vulnerable.

2.2 Semantic Attack Definition/Code Injection Attacks Definition

The precise definition of a semantic attack varies greatly. In Scheiner's original article when he referred to semantic attacks he was referring to the "human element" and attacks such as modern day phishing or pump-and-dump schemes. This type of definition has become the definition of "semantic hacking." According to PC Magazine one definition of semantic attack is: "The use of incorrect information to damage the credibility of target resources or to cause direct or indirect harm. Examples include defamation (slander and libel), propaganda and stock manipulation schemes. Also known as 'semantic hacking.' " [16]

Other definitions of semantic attacks refer to some type of malicious code injection into the web browser or the running code. For this dissertation, a semantic attack will be defined as

an attack where malicious code is inserted into a running program. This definition of semantic attack classifies the semantic disconnect between the application programmer's mental model of the external environment and the actual reality of the operational environment. In most cases an injection attack will occur based on the application programmer not correctly validating user supplied input.

If one examines the OWASP top ten vulnerabilities, it is evident that most of those vulnerabilities are considered semantic attacks; yet, those attacks are also classified as injection attacks. The reason for this duplicate classification can be attributed to the "human element." One of OWASP's top ten vulnerabilities is SQL semantic attacks. Based on the above definition SQL semantic attacks will be defined as SQL injection attacks. Furthermore, for the purpose of this dissertation, code injection attacks will be discussed with a focus on two of the top OWASP vulnerabilities. It is the goal of this dissertation that the security mechanisms developed will be general enough to be applied to the other OWASP vulnerabilities.

2.2.1 SQL Injection

OWASP classifies injection attacks as the number one attack for web applications, in particular SQL Injection. SQL injection occurs when an application sends an untrusted SQL query to the interpreter. A SQL query is untrusted if the query string is sent without correctly filtering the untrusted input.

For example, the normal query might appear as:

```
SELECT * FROM user_table WHERE email = 'User supplied from a web form';
```

Presuming the user entered in the web form: **name@address.com** the query string would be expanded to:

```
SELECT * FROM user_table WHERE email = 'name@address.com';
```

It is important to understand that the single quotes that surround the email address and the

ending semicolon are important aspects of the SQL query. The semicolon indicates the end of the query string. The single quotes indicate the beginning and end of the user input. Since the interpreter is building a string, it searches for the opening and closing single quotes. If there is not an equal number of quotes (too many or too few) then the interpreter will abort with a syntax error, and an error message will be displayed to the user. In the above example the single quote marks align. Therefore, as an unfiltered query this query would return the desired results.

The above query can easily be altered to inject additional SQL commands. Instead of entering the email address the user enters the following:

```
'name@address.com' or '1' = '1'; --
```

In SQL the double dash indicates a comment so everything after the double dashes is ignored. Expanding out the new query string the command would appear as

```
SELECT * FROM user_table WHERE email = 'name@address.com' or 1 = 1; --
```

Since or 1 = 1 is always true, then the interpreter would return every item in the database. In this very simplified example not much can be done with just email addresses; however, in a more complicated example the database would be queried to return all usernames and passwords, including the administrator username and password.

2.2.2 SQL Injection Security Mechanisms

A SQL injection is very difficult to detect because, to the database, the SQL injection appears as a normal SQL query. There are three main security mechanisms that protect against SQL injection.

- **Application Developer**

  Most security policies to combat SQL Injection are implemented on the application developer's end. It is the responsibility of the application developer to filter the user input.

This is typically accomplished through language specific filtering functions. For example, in the PHP language there is a function called mysql_real_escape_string. This function prepends backslashes to a set of characters including the single quote. The problem with leaving it to the application developer is, the developer may not know or even understand SQL injection.

- **Filtering and Monitoring Software**

  Filtering and monitoring tools at the Web application and database levels will help block attacks and detect attack behavior. At the application level organizations can possibly prevent SQL injection by implementing runtime security monitoring; furthermore web application firewalls can help organizations by creating behavior-based rule sets to block SQL injection attacks. Database activity monitoring can filter attacks on the back end, especially for known SQL injection attacks. There are generic filters that query for typical SQL injections such as uneven numbers of quotes. The drawback with filtering and monitoring software is two-fold. The software can be very expensive which often times a small business can't afford, and if the business is able to afford the software, the installation/setup is often beyond the database/business owner's skill set.

- **Database Patches**

  The risks associated with SQL injections are increased when the databases tied to the web applications are poorly maintained, including poor patching and configuration. Part of the configuration process requires better management on web application's associated accounts, especially with accounts that interact with the back-end databases. Many problems arise due to database administrators not understanding security, so the administrators give the web application accounts greater privileges than required. These super accounts are very vulnerable to attack and thus greatly broaden the risks to databases.

2.2.3 Cross-site Scripting

Cross-site Scripting flaws occur when an application includes user-supplied data in a page sent to the browser without properly validating or "escaping" that content. Cross-site Scripting vulnerabilities target scripts embedded in a page which are executed on the client-side, in the victim's web browser, rather than on the server-side, where the web page is held. Cross-site Scripting, in itself, is a threat which is brought about by the security weaknesses of client-side languages such as HTML, PHP and JavaScript. The concept of Cross-site Scripting is to manipulate client-side scripts of a web application to execute in the manner desired by the malicious user. Such a manipulation can embed a script in a page which can be executed every time the page is loaded, or whenever an associated event is performed. Cross-site Scripting vulnerabilities can have significant consequences such as tampering and sensitive data theft.

Essentially, Cross-site Scripting allows the attacker to execute selected code on the victim's browser. Once the page is loaded the victim's machine is compromised. A comprised machine could:

- Allow for the victim's cookies, which often contain username and password to be stolen
- Control the browser remotely
- Spread worms

A simple example:

The URL on your site **http://www.mysite.com/search?q=plants** returns HTML containing <**p**>**Your search for 'plants' returned the following results:** </**p**>

The value of the query parameter q is inserted into the page returned by the site. Since the data is not validated, filtered or escaped (sanitized) then a malicious attacker could put up a page that causes a malicious URL to be loaded in the browser.

```
http://www.mysite.com/search?q=flowers+%3Cscript%3EevilScript()%3C/script%3E
```

When a victim's browser loads the URL above. The document loaded will contain:

```
<p> Your search for 'plants <script> evilScript() </script> ' </p>
```

Loading this page will cause the browser to execute evilScript(), which might allow the browser to be controlled remotely.

2.2.4 Cross-site Scripting Security Mechanisms

Like SQL Injection, Cross-site Scripting is also very hard to detect. There are three main security mechanisms that protect against Cross-site Scripting.

- **Filtering and Monitoring Software**

  Similar to SQL Injection filtering and monitoring tools can be applied at the Web application level. As with SQL injection there are the two drawbacks of expense and insufficient skills.

- **Validate input**

  Validating input is very complex. A good programmer will validate the input received; however, a novice programmer has no idea about Cross-site Scripting and how to validate the input. Validating input quickly fails when the programmer has no knowledge of the problem.

- **Allow the client to disable client-side scripts**

  In order for Cross-site scripting to be effective the malicious script needs to run in the victim's browser. By disabling scripts the client browser cannot be compromised. This is an effective mechanism; however, it relies on web sites not requiring or needing scripts to display the web page. Without some kind of scripting, the web pages would not be dynamic. In our modern world, most users rely on dynamic web content to enhance their web experience.

2.2.5 Summary of Semantic Attack Security Mechanisms

The above sections outline the two most prevalent web vulnerabilities and the security mechanisms required to prevent them. In every case, the required security mechanism relies on the programmer to implement, then requiring the programmer to handle any security mechanism is problematic. First, it presumes the programmer fully understands what security mechanism is required. If the programmer knows the required security mechanism, then it presumes the programmer knows how to properly code that mechanism. Other defenses are also problematic. Filtering and monitoring are often prohibitive based on the cost or the system administrator knowledge.

2.2.6 Static Analysis

A simple analysis of cross-site scripting or SQL injection indicates that security violations occur when the input data is sent to the server without input validation. Static analysis tries to prevent this security violation by first checking the input data. This input checking can be broken into one of three categories:

- **String Evaluation**

  String evaluation takes the input string provided by the user and breaks that string into tokens. The tokens are then evaluated to determine whether changes occurred in the query's intended result. According to Lambert, "In fact for all types of SQL injection, there is no way someone can perform injection without inserting a space, single quotes or double dashes in a query [11]." Different techniques for string evaluation include: query tokenization, context-sensitive string evaluation, filtering vulnerable characters and input string parse trees [11,21,27,30,37]. In all instances, relying on string evaluation for input strings is problematic. String evaluations as sole mechanisms cannot defend against sophisticated attacks such as alternate encodings and commands that construct dynamic strings.

Table 2.1: Comparison of Static Analysis Techniques

Defense Type	Advantages	Drawbacks
String Evaluation [11,21,27,30,37]	Easy to tokenize strings to determine vulnerabilities	Cannot handle sophisticated attacks
Static-Tainting [35]	Taints and tracks untrusted user strings	Typically browser specific and difficult to implement across platforms
Formal Analysis [2]	Easily detects attacks by mining intended query against actual query	Applied to a specific language

- **Static-Tainting**

  String evaluation typically generates a context-free grammar or a parse tree based on the input string. With static-tainting, a context-free grammar is created; however each set of strings within that grammar is tainted to represent that the source of the set of strings is from an untrusted source [35]. The issue with static-tainting is the exploits that are identified are specific to the Firefox browser and by the author's own admission in order to handle other browsers would still require user interaction.

- **Formal Analysis Techniques**

  Formal analysis techniques try to detect SQL injection via query structure analysis and formal analysis of certain SQL commands [2]. Query structure analysis verifies the intended structure of issued queries by dynamically mining the programmer-intended query structure on any input. The formal analysis detects the attack by comparing the mined query structure against the structure of the actual query that was issued. The deficiency with this technique is that the formal analysis has only been applied to Java applications, and in order for the application to be effective, the Java Virtual Machine was modified.

Static analysis is ineffective as the only basis to protect against SQL injection attacks. Static analysis techniques cannot handle complex sophisticated attacks with alternative encodings of dynamically constructed strings. Static analysis should not be discounted, it should be

another piece in the puzzle used to combat SQL injection attacks. Static analysis techniques are summarized in Table 2.1.

2.2.7 Dynamic Analysis via tainting (DIFT)

DIFT commonly tracks the flow of untrusted information through the system during runtime. This tracking is commonly implemented though a technique that tags the untrusted information. If the untrusted information is used in an unsafe manner then a security exception of some type is typically raised. Depending on the raised security exception different outcomes can occur. In order for DIFT to be effective, a set of policies must exist. This policy specifies how information is tagged, how that tag propagates through the system, and what exceptions will be raised on unsafe operations. In a DIFT environment, the original programs are unmodified. The tags and the propagation are handled by the runtime environment and the DIFT policy as specified. "By tracking and restricting the flow of untrusted information, DIFT policies can prevent user input from being used unsafely [4]."

Suh et al. specified five information flow dependencies that a DIFT system must track in order for that DIFT system to be considered complete [31]. These five dependencies include:

- **Computational Dependencies**

  Computational dependencies are defined as operations on the data, such as two registers being added together with the results being stored into a register.

- **Control Dependencies**

  Control dependencies are defined as operations that affect the flow of the program, such as a branch condition.

- **Copy Dependencies**

  Copy dependencies are defined as movements of data from register to register, register to memory, or memory to register.

- **Load-Address Dependencies**

  Load-address dependencies are defined as data being loaded from a memory address.

- **Store-Address Dependencies**

  Store-address dependencies are defined as data being stored into a memory address.

DIFT is a promising powerful security technique; however, DIFT has a few drawbacks. These drawbacks include:

- **False Positives**

  An incorrect DIFT policy will cause the system to produce false positives. False positives occur when a normal system command is marked as being a security exception.

- **False Negatives**

  Just as an incorrect DIFT policy will raise false positives it can also raise false negatives. False negatives occur when a security exception should have been raised; however the system considers the command as safe.

- **Runtime Performance Overhead**

  Since the original program is not modified in the runtime environment, then to check for vulnerabilities each program has to be run in a sand-boxed environment, potentially running each program twice. Running the program multiple times can cause performance issues where the program seems to be running much slower than if the original program would have been run just once.

- **Require Software Support**

  DIFT requires operating system or run-time system support for original tagging of the data, and for exception handling. If left to the programmer, these are new avenues for vulnerabilities.

There exists a set of DIFT policies to combat many different intrusion detections. More specifically policies exist to combat SQL injection, cross-site scripting and many other attacks and vulnerabilities [30]; however, there remains quite a few challenges. These challenges can be found in the work of others trying to solve the SQL injection problem using DIFT.

In Halfond's et al. [7] research, positive tainting is used to identify a SQL injection attack. According to the research, positive tainting allows for increased automation over the traditional

Table 2.2: Comparison of Dynamic Analysis Techniques

Defense Type	**Advantages**	**Drawbacks**
DIFT [30]	Known policies allow for tainting and the ability to combat many different intrusion detections	Tainting can lead to false positives, false negatives and high performance overhead
Positive Tainting [7]	Identifies and marks trusted data instead of untrusted data. This trusted data is tracked within the DIFT system.	Possible high rate of false positives, due to some strings not being considered part of the trusted data
Rashka [4]	Uses dynamic techniques to identify SQL injection with minimal false positives	Requires the user to be fully authenticated. Tracks strings at the word level, and by the author's own admission, the SQL validation routine needs work. The routine only scans for tainted characters in place of parsing the SQL grammar.

negative tainting use conducted via DIFT. Furthermore, positive tainting minimizes both false positives and false negatives. By identifying trusted strings performance overhead is also minimized. Although Halfond et al. tries to remove developer interaction, the developer has to be involved in a minimum of two instances. The first instance is checking for false positives. A false positive can occur when a trusted string is missed. The second instance of developer interaction is at deployment. In order for the solution to work a special library must be deployed at installation.

In Dalton's research concerning Raksha [4], SQL injection, cross-site scripting, buffer overflows and many other security attacks were handled via DIFT; however, the performance overhead was quite large, as well as new hardware had to be developed in order to support DIFT. The hardware, five years later, is still not commercially available.

Although there has been some advancement using DIFT, it is clear that DIFT by itself will not solve the problem of SQL injection or cross-site scripting. Simply tagging or tainting the user input will lead to false positives, false negatives, runtime performance overhead problems,

or a combination of all the drawbacks of DIFT. Dynamic analysis techniques are summarized in table 2.2.

2.2.8 Hybrid Analysis

Based on the limitations of input parsing of purely static analysis or the limitations of the runtime overhead of dynamic analysis a novel approach of hybrid analysis has been proposed. Hybrid analysis is meant to combine the static analysis techniques of tokenization to produce a set of statements that could be considered dangerous. These dangerous statements are then tagged and tracked through the runtime environment using a modified DIFT [19]. In both Monga et al. [19] and Razzaq et al. [23] research their static analysis created Control Flow Graphs (CFGs).

In Razzaq's et al. [23] the CFGs were analyzed using a Validation Analysis. This validation analysis categorizes the input into one of three categories: non vulnerable, vulnerable, and potentially vulnerable. For non-vulnerable category, the system has determined that the CFG will not cause a secuity exception, indicating the command is safe to execute. The vulnerable category indicates the system has determined the command is not safe and a security exception will be raised; therefore, the command should not be executed by the system. Concerning the potentially vulnerable category, the system indicates that a potential security exception maybe raised; therefore, the developer is made aware and then must initiate an investigation.

In Monga et al. [19] a static model or CFG of the entire application is constructed. From this CFG the statements that are considered dangerous are constructed. These dangerous statements are then tracked within the run-time system to determine if a SQL injection has occurred.

In Monga et al. [19]and Razzaq et al. [23] hybrid analysis, serious limitations exist. In both instances a trade off must be made between identifying vulnerabilites and passing those vulnerabilities to the runtime environment. Too many vulnerabilities will decrease the runtime performance. Furthermore, the output produced from a sanitized routine is untainted. This untainted output could easily be generated from an incorrect process based on the CFG and the

Table 2.3: Hybrid Analysis Techniques

Defense Type	Advantages	Drawbacks
Razzaq et al. [23]	Uses both static and dynamic techniques to identify SQL injection. First constructs a CFG, and then passes portions of the CFG to the run-time system.	Trusted, unverified and untainted output could be very dangerous especially if that output is later used for input. Any potential security exceptions must be investigated by the developer.
Monga et al. [19]	Uses both static and dynamic techniques to identify SQL injection. Constructs a CFG of the entire system identify potentially dangerous statements. Passes the dangerous statements to the run-time system.	Trusted, unverified and untainted output could be very dangerous especially if that output is later used for input. the system has only been implemented in experimental prototypes, with extensive hooks written into the PHP interpreter.

original input. Trusted, unverified and untainted output could be very dangerous especially if that output is later used for input. Hybrid analysis techniques are summarized in table 2.3.

Of all the techniques, hybrid analysis seems to have the most promise despite its limitations.

2.2.9 Analysis Summary and Conclusion

Ultimately SQL injection occurs within a web application due to inadequate validation and sanitization of user inputs. There are three basic classifications used as defense mechanisms to combat these injection attacks. These classifications are defensive coding, SQL injection detection techniques and SQL injection runtime prevention techniques [26, 28]. This dissertation does not discuss the advantages or drawbacks of defensive coding. SQL injection detection techniques are discussed under the subsection static analysis above. Table 2.1 clearly illustrates the advantages and drawbacks of these static analysis tools. SQL injection runtime detection techniques are discussed under the subsection dyamic analysis and hybrid analysis above. Table 2.2 and reftable:hybridAnalysis clearly illustrates the advantages and drawbacks of these dynamic and hybrid analysis tools.

It is the proposal of this dissertation that a general solution cannot be implemented using only static analysis or using only dynamic information flow tracking; that a hybrid approach will provide the best implementation. Furthermore, the proposal of this dissertation is to implement a hybrid general solution for SQL injection that minimizes developer interaction, with the goal of trying to eliminate developer interaction. The policies for this solution will be outlined in the next chapter.

Chapter 3

Proposal

The research of this dissertation will focus on three problems:

1. How is a high-level semantic attack web vulnerability defined?
2. Can a technique be developed that will handle a high-level semantic attack within the framework of this hybrid approach, using the security mechanisms of the PSL?
3. What does the policy-based hybrid mapping framework represent, and from that policy-based hybrid mapping what should be allowed and what should be denied?

Since policy-based hybrid mapping framework is complex and still in its infancy, this research will provide solutions to the problems previously discussed.

3.1 Motivation and Contributions

The goal of this dissertation is to provide a policy-based hybrid mapping framework for the enforcement of security policies at both the software and the architectural level. This dissertation will achieve the following specific objectives:

Objective 1: Provide a clear definition of web vulnerabilities including defining high-level semantic attacks. This definition will be the basis for defining the policy-based hybrid patterns.

A web applications can be defined as an interface, accessed through a web browser, for any user to view and interact with static and dynamic content. Dynamic content is typically provided via a general-purpose programming language such as Java, PHP, etc. Dynamic con-

tent typically involves user interaction within the web browser. The user interacts with the general-programming language via user input in the web browser.

As a start to clearly defining web vulnerabilities, a definition by Su and Wasserman is presented and analyzed [30]. Su and Wasserman focus on defining a web application specifically to handle SQL Injection Command Attacks (SQLCIA). Su and Wasserman use an augmented grammar to filter the user input. That augmented grammar requires the following characteristics relevant to the defined SQLCIA:

1. It takes input strings, which it may modify
2. It generates a string by combining filtered inputs and constant strings
3. The query is generated without respect to the SQL grammar.
4. The generated query provides no information about the source of its characters/substrings

Based on the above characteristics, Su and Wasserman define a web application as:

Definition 3.1. Su and Wasserman Web Application Definition

We abstract a web application $P : \langle E^*, \ldots, E^* \rangle \to E^*$ as a mapping from user inputs (over alphabet E) to query string (over E). In particular P is given by $\{\langle f_1, \ldots, f_n \rangle, \langle s_1, \ldots, s_m \rangle\}$where

- $f_i : E^* \to E^*$is an input string;
- $s_i : E^*$is a constant string

The arguement to P is an n-tuple of input strings $\langle i_1, \ldots, i_n \rangle$, and P returns a query $q = q_1 + \cdots + q_l$where, for $1 \leq j \leq l$,

$$q_j = \begin{cases} s & \text{where } s \in \{s_1, \ldots, s_m\} \\ f(i) & \text{where } f \in \{f_1, \ldots, f_n\} \wedge i \in \{s_1, \ldots, s_n\} \end{cases}$$

That is, each $q_j$is either a static string or a filtered input.

The Su and Wasserman definition of a web application is specific to SQLCIA, and it does not handle the general case of user input not being passed to a SQL interpreter. To define a general web application the following must be present.

1. A web application takes user input as a string, which the web application may modify

2. The general-purpose programming language generates a string that is either the original string, or the original string combined with other string including constant strings, or the empty string converted to a constant string.

Definition 3.1 is not constructed for the general definition of a web application, since it is based on a very specific instance of SQLCIA. In order to construct a general definition for a web application, the definition of Su and Wasserman must be modified. The definition is very similar to Definition 3.1; however, in the Su et al. there are two possibilities for the returned query string. Definition 3.2, Proposed Initial General Definition of a Web Application, there are three possibilities for the returned generated string.

Definition 3.2. Proposed Initial General Definition of a Web Application

P represents a mapping from user inputs (over the alphabet E)
to the generated string E $P : \langle E^*, \ldots, E^* \rangle \rightarrow E^*$
P is given by $\{\langle u_1, \ldots, u_n \rangle, \langle s_1, \ldots, s_n \rangle\}$ where (over E)

- $u_i : E^* \rightarrow E^*$ is a user string;
- $s_i : E^*$ is a constant string

P returns a generated string $g = g_1 + \cdots + g_m$

$$g_k = \begin{cases} s & \text{where } s \in \{s_1, \ldots, s_n\} \\ u(i) & \text{where } u \in \{u_1, \ldots, u_n\} \wedge i \in \{s_1, \ldots, s_n\} \\ emptystring & \end{cases}$$

$g(k)$ is a generated string that is either the original user string or
the user string combined with a static string or the empty string
converted to a constant string

Based on Definition 3.2, a web vulnerability is defined as a weakness or flaw within the web application that allows an attacker to cause harm. This vulnerability is typically created when the user string is not filtered. Based on the work of Ray and Ligatti [22] the definition of a vulnerability will be defined and expanded to include specific definitions that will be processed by the static analysis of the hybrid framework.

Objective 2: Provide a set of policy-based hybrid patterns for decomposing, aggregating and handling of web vulnerabilities, including injection vulnerabilities such as SQL injection. These patterns will define a policy-based hybrid mapping framework. This framework, including the security policies, will determine which transactions will be allowed and which transactions will be denied.

Based on the web application and web vulnerability definitions previously discussed and the work of Ray and Ligatti [22] the output from the web application will be broken into two parts. The first part will be passed to a static scanner. This static analysis will involve scanning the input string to create both a parse tree and a Control Flow Graph (CFG). Previous research usually either created a parse tree or a CFG, but not both; previous research hypothesized that a parse tree or a CFG could correctly identify SQL injection, and further static analysis was not required [19, 30]. Based on the user input, the parse tree will be created first. For the parse tree to correctly identify possible SQL injection attacks the following characteristics of the formal grammar are required:

1. A web application takes user input as a string, which the web application may modify.
2. The general-purpose programming language generates a string that is either the original string, or the original string combined with other string including constant strings, or the empty string converted to a constant string.
3. A query string is created by filtering a generated string. Filtering is defined as denoting the beginning and ending of the generated string.
4. A query string is generated without regard to the SQL grammar.

This parse tree will represent the concrete syntax of the formal grammar defined in other objectives, and it will be used to determine if the user input violates any syntactical structure of the grammar. If no parse tree violation exists, then the user input will be passed to the second part, constructing a CFG. The CFG will identify any paths that could be considered invalid.

If the parse tree or the CFG identify any invalid statements within the user input, then a security exception will be raised. The user will receive a simple notification, stating “Invalid

```
inst checkParse+ filter1
{
        subject         parse tree results
        target          CFG
        action          Results(RC, exception)
                        {in RC=1, in exception=1}
}
```

Figure 3.1: Ponder specifications for the rule checkParse

input", and the user input will not be passed to the SQL interpreter.

If the parse tree or the CFG does not identify any invalid statements, then the user input will be passed to a dynamic information flow tracking system (DIFT). This DIFT analysis is discussed within other objectives.

Deliverable This approach of formalizing the web vulnerabilities definition, including static checking using both the parse tree and CFG should lead to a publication.

Objective 3: Develop an framework for rule predicates and policies for the formal analysis of the policy-based hybrid mapping framework. This algebraic framework will provide a formal means for detecting policy conflicts.

Based on the the work of Zhou [38], Uszok et al. [33], and high level policy specification languages such as Ponder [5], the run-time enforcement policies will model component behavior defined in terms of inputs (i) and outputs (o). The user input (ui) is sent to the policy rule, and o is the output of the decision made by the rule. For example, in a policy rule checkParse-Tree, specified using the Ponder language 3.1 A majority of the policy rules rely on the Ponder Information filtering policy, Ponder obligation policy or Ponder refrain policies. The Information filtering policy transforms input or outputs into an interaction [5]. The Ponder obligation policy is an event-triggered policy that specifies the actions that must be performed by subjects (s). A subject can be a manager program that determines certain events once the event is triggered. The Ponder refrain policy determines the actions that the subjects must not perform. For example, the user submits input; the Ponder filtering policy takes the input and produces the

interaction representing an invalid parse tree. That interaction is checked by the Ponder refrain policy, which verifies the interaction should refrain from continuing. The Ponder refrain policy triggers the Ponder obligation policy where an invalid command is raised and delivered to the user.

A policy rule (r) is defined as a safety predicate (allowed action) (P) and a security predicate (Q) such that: $r = P \wedge Q$. The entire language to specify the outcomes from the parse tree and the CFG will be developed using the Ponder language specifications. In order to detect policy conflicts, A Computational Logic for Applicative Common Lisp (ACL2) will be used. ACL2 is a first-order logic and mechanical theorem prover, and since the Ponder language specification is also a first-order logic.

Deliverable This approach of defining the rules and predicates using the Ponder language specification and ACL2 should lead to a publication.

Objective 4: Verify the proposed approach by applying the policies and security mechanisms to a customized hardware system executing a customized dynamic system, such as Real-Time Executive for Multiprocessor Systems (RTEMS), including a web service application system. The policy-based hybrid mapping framework is applied at different levels of the hardware and custom RTEMS and web service. Each step is verified and each security mechanism is analyzed to ensure the web vulnerability and the high-level semantic attacks are properly handled.

If the SQL command is deemed safe from the static analysis, then the generated filtered string will be passed to the DIFT system; where, using the tainting process, developed by Song [29] the generated filtered strings will be tainted. During execution if a security exception is raised then a simple error message will be displayed to the user. This error message will be along the lines of "Invalid Command." The error messages from the static analysis portion of the hybrid system will be different than the dynamic analysis portion. This will allow easy recognition of which system generated the the error message.

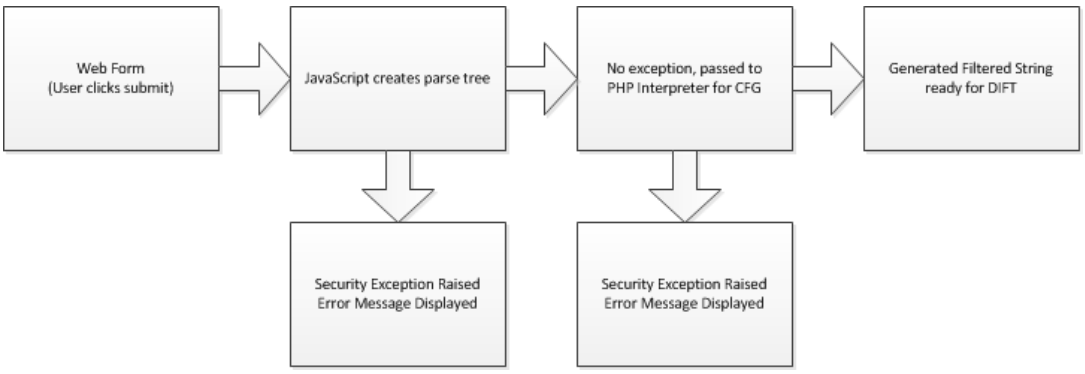

Figure 3.2: The required flow to test static analysis

Deliverable This approach of passing the filtered generated string to the DIFT system should lead to a joint publication with Song.

Objective 5: Verify the proposed approach by applying the policies and security mechanism to a customized interpreter engine, such as the PHP interpreter. The policy-based hybrid mapping framework is applied at different levels of the interpreter. Each step is verified and each security mechanism is analyzed to ensure the web vulnerability and the high-level semantic attacks are properly handled.

Portions of the PHP interpreter will need to be rewritten in order to add in the CFG. Before the command is sent to the new PHP interpreter, the parse tree will be generated using JavaScript. Figure 3.2 outlines the flow from after the user entered the string and pressed submit. In past research the parse tree was typically created within the general programming language engine. In the case of PHP, the parse tree and the CFG were generated after the interpreter was modified.

Deliverable This approach of using JavaScript to generate the parse tree is new and should lead to a publication. Furthermore, based on the algebraic framework discussed above, the edits to the PHP interpreter to construct the CFG should also lead to a publication.

In summary, the work presented in this dissertation provides a practical security engineering framework that will benefit programmers and system architects.

Bibliography

[1] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu, "Strict control dependence and its effect on dynamic information flow analyses," in *Proc. of the 19th ACM ISSTA'10*, New York, NY, USA, 2010, pp. 13–24.

[2] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 2, pp. 14:1–14:39, Mar. 2010.

[3] M. D. Bond, V. Srivastava, K. S. McKinley, and V. Shmatikov, "Efficient, context-sensitive detection of real-world semantic attacks," in *Proc. of the 5th ACM SIGPLAN*, ser. PLAS '10, New York, NY, USA, 2010, pp. 1:1–1:10.

[4] M. Dalton, "The Design and Implementation of Dynamic Information Flow Tracking Systems for Software Security," Ph.D. dissertation, Stanford University, 2009.

[5] N. Damianou, N. Dulay, E. Lupu, M. Sloman, and N. D. N. Dulay, "Ponder: An object-oriented language for specifying security and management policies," Imperial College, Tech. Rep., 2000.

[6] T. Grance, J. Hash, S. Peck, J. Smith, and K. Korow-Diks, "Security Guide for Interconnecting Information Technology Systems: Recommendations of the National Institute of Standards and Technology," *NIST Special Publication 800-47*, 2002.

[7] W. G. J. Halfond, A. Orso, and P. Manolios, "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation," vol. 34, no. 1, pp. 65–81, 2008.

[8] W. G. J. Halfond, S. Anand, and A. Orso, "Precise interface identification to improve testing and analysis of web applications," in *Proc. ACM ISSTA'09*, New York, NY, USA, 2009, pp. 285–296.

[9] W. G. J. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," in *Proc. ACM SIGSOFT'06*, New York, NY, USA, 2006, pp. 175–185.

[10] M. S. Lam, M. Martin, B. Livshits, and J. Whaley, "Securing web applications with static and dynamic information flow tracking," in *Proc. ACM SIGPLAN*, ser. PEPM '08, New York, NY, USA, 2008, pp. 3–12.

[11] N. Lambert and K. S. Lin, "Use of Query tokenization to detect and prevent SQL injection attacks," in *Proc. of the IEEE ICCSIT'10*, vol. 2, 2010, pp. 438–440.

[12] M. C. Libicki and N. D. University., *What is information warfare?* Washington, DC, USA: National Defense University, Institute for National Strategic Studies, 1995.

[13] J. Ligatti, L. Bauer, and D. Walker, "Edit Automata: Enforcement Mechanisms for Run-time Security Policies," Princeton University, Tech. Rep., 2003.

[14] ——, "Run-Time Enforcement of Nonsafety Policies," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 3, pp. 19:1–19:41, Jan. 2009.

[15] J. Ligatti and S. Reddy, "A theory of runtime enforcement, with results," in *Proc. ESORICS'10*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 87–100.

[16] P. C. Magazine, "Definition of: semantic attack." [Online]. Available: http://www.pcmag.com/encyclopedia_term/0,1237,t=semantic+attack&i=51087,00.asp

[17] J. Magazinius, A. Russo, and A. Sabelfeld, "On-the-fly inlining of dynamic security monitors," in *Proc. International Information Security Conference IFIP'10*, 2010.

[18] S. Mitchell, "Displaying a Custom Error Page (C#)." [Online]. Available: http://www.asp.net/web-forms/tutorials/deployment/deploying-web-site-projects/displaying-a-custom-error-page-cs

[19] M. Monga, R. Paleari, and E. Passerini, "A hybrid analysis framework for detecting web application vulnerabilities," in *Soft. Eng. for Secur. Syst.*, ser. SESS '09, 2009, pp. 25–32.

[20] OWASP, "Category:OWASP Top Ten Project." [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[21] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Proc. of the 8th Internat. Conf. on Recent Advances in Intrusion Detection*, ser. RAID'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 124–145.

[22] D. Ray and J. Ligatti, "Defining code-injection attacks," in *Proc. of the 39th annual ACM SIGPLAN-SIGACT*, ser. POPL '12, New York, NY, USA, 2012, pp. 179–190.

[23] A. Razzaq, A. Hur, N. Haider, and F. Ahmad, "Multi-Layered Defense against Web Application Attacks," in *Proc. of the 6th Internat. Conf. on Infor. Tech. New Generations*, ser. ITNG'09, 2009, pp. 492–497.

[24] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, Feb. 2000.

[25] B. Schneier, "Inside risks: semantic network attacks," *Commun. ACM*, vol. 43, no. 12, p. 168, Dec. 2000.

[26] T. Scholte, W. Robertson, D. Balzarotti, and E. Kirda, "An empirical analysis of input validation mechanisms in web applications and languages," in *Proc. of the 27th ACM SAC'12*, New York, NY, USA, 2012, pp. 1419–1426.

[27] S. V. Shanmughaneethi, S. C. E. Shyni, and S. Swamynathan, "SBSQLID: Securing Web Applications with Service Based SQL Injection Detection," in *Proc. of Advances in Computing, Control, Telecom. Tech.*, ser. ACT'09, 2009, pp. 702–704.

[28] L. K. Shar and H. B. K. Tan, "Defeating SQL Injection," vol. 46, no. 3, pp. 69–77, 2013.

[29] J. Song, "Development and Evaluation of a Security Tagging Scheme for a Real-Time Zero Operating System Kernel," Master's thesis, University of Idaho, 2012.

[30] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *Proc. of the 33rd ACM SIGPLAN-SIGACT*, ser. POPL'06, New York, NY, USA, 2006, pp. 372–382.

[31] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proc. of the 11th ACM ASPLOS'11*, New York, NY, USA, 2004, pp. 85–96.

[32] W. D. Thomas Ian and D. Weidenhamer, "Quarterly Retail E-Commerce Sales 1st Quarter 2012," U.S. Census Bureau, Washington, DC, USA, Newsletter CB12-78, May 2012.

[33] A. Uszok, J. M. Bradshaw, M. Johnson, R. Jeffers, A. Tate, J. Dalton, and S. Aitken, "Kaos policy management for semantic web services," vol. 19, no. 4, pp. 32–41, 2004.

[34] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proc. of the 2007 ACM SIGPLAN*, ser. PLDI '07, New York, NY, USA, 2007, pp. 32–41.

[35] ——, "Static detection of cross-site scripting vulnerabilities," in *Proc. of the 30th ACM ICSE'08*, New York, NY, USA, 2008, pp. 171–180.

[36] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Proc. of the 15th conference on USENIX-SS'06*, vol. 15, Berkeley, CA, USA, 2006.

[37] A. S. Yeole and B. B. Meshram, "Analysis of different technique for detection of SQL injection," in *Proc. of the ACM ICWET'11*, New York, NY, USA, 2011, pp. 963–966.

[38] J. Zhou, "Policy-based Architectual Refinement Techniques for the Design of Multi-level Secure Systems," Ph.D. dissertation, University of Idaho, 2008.