Systematically Enhancing Black-Box Web Vulnerability Scanners

Thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Science (by Research)

in

Computer Science

by

Sai Sathyanarayan Venkatraman

National University of Singapore

Singapore

August 2012

National University of Singapore

Singapore

CERTIFICATE

It is certified that the work contained in this thesis, titled “Systematically Enhancing Black-Box Web Vulnerability Scanners” by Sai Sathyanarayan, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date	Advisor: Dr. Zhenkai Liang

Abstract

Black-box web vulnerability scanners are a class of tools that can be used in finding security vulnerabilities in web applications automatically regardless of server-side language implementation. These tools access a web application in the same way users do. Unfortunately, black-box tools both commercial and open-source suffer from a number of limitations. In particular, advanced SQL Injection (SQLI) vulnerabilities and authentication protocol implementation flaws are not currently detected by any of these tools. In this thesis, we propose two approaches to handle the above limitations - SQLR(SQLi Revisited) and WeakAuthScan.

The SQL injection attack is one of the major threats to web applications. Through malicious inputs, attackers can cause data leakage and damage, and even remote code execution on the victim servers. Since SQL injection vulnerabilities are caused by malicious inputs, a common solution is to use input sanitizers to filter out inputs that can result in SQL injection attacks. To validate the correctness of SQL injection sanitizers, recent solutions model web application's sanitizers, and check the model with SQL injection attack patterns. However, the attack patterns used by existing solutions only detect simple SQL injection attacks, which significantly limits the power of their solutions. In this thesis, we propose a novel solution, SQLR, to validate SQL sanitizers by systematically generating SQL injection attack patterns. Our approach uses the SQL grammar to guide the enumeration of malicious SQL queries efficiently, and summarizes the queries into patterns that can be used by existing solutions. In our evaluation, SQLR identified new attack patterns and weaknesses in sanitizers used in several real-world web applications. Using our approach, we show that current web scanners are not effective in detecting SQLi since they rely on generic attack patterns.

In practice, checking authentication protocol implementation is difficult due to lack of complete implementation (such as missing source code of protocol participants). Using black-box scanners, it is also difficult because the web applications require a user to create an account to access the authentication system which these scanners fail to do it automatically. In this thesis, we present a framework *WeakAuthScan* to automatically extract the authentication protocol logic. *WeakAuthScan* assumes no knowledge of the protocol being checked and does not require access to the source code of the implementation. We propose a blackbox analysis by analyzing messages between the authentication server and the web user. We evaluated our approach on two popular websites which have millions of users sharing deeply personal information and found security flaws in their implementation of authentication protocol.

Contents

1 Introduction **1**

1.1 Research Overview . . . 2

1.1.1 SQLR - SQLi Revisited . . . 2

1.1.2 WeakAuthScan . . . 4

2 Web Application Vulnerability and Black Box Scanners **6**

2.1 Web Vulnerability Black Box Scanners . . . 6

2.1.1 Black-Box Penetration testing . . . 7

2.1.2 Commercial Tools . . . 7

2.1.3 Free/Open Source Tools . . . 8

2.2 SQL Injection Attack and Input Sanitizers . . . 9

2.2.1 SQL Injection . . . 9

2.2.2 Input Sanitizer . . . 9

2.2.3 Current Solutions to Validate Sanitizers . . . 9

2.3 Weak Authentication Systems . . . 10

2.4 Related Works . . . 11

2.4.1 SQLi attack detection. . . . 11

2.4.2 Web application sanitizer generation. . . . 12

2.4.3 Evaluating Web Vulnerability Scanners . . . 12

3 Grammar-guided Validation of SQL Injection Sanitizers **13**

3.1 Overview of Our Solution . . . 14

3.2 Grammar Guided Generation of SQLi Attack Patterns . . . 15

3.2.1 Challenges . . . 15

3.2.2 Optimizing SQL Parsing Graph . . . 18

3.2.3 Enumerating Queries through Symbolic Analysis of the Final Graph . . 19
3.2.4 Generation of SQLi Attack Patterns 21
3.2.5 Discussion of Path Sensitivity and Over Approximation 22
3.3 Implementation 23
3.3.1 Symbolic Analyzer of Web Applications 23
3.3.2 Multiple Query Generation 24
3.3.3 Sanitizer Validator 24
3.3.4 Proactive Pruning of Blocked Queries 25
3.4 Evaluation 26
3.4.1 Generating New Attack Patterns 26
3.4.2 Effectiveness in Validating Sanitizers 27
3.4.3 Performance 29
3.5 Case Study with Web Scanners 29

4 Weak Authentication Systems (WAS) 31
4.1 Problems with existing Scanners 31
4.2 Overview of our approach 32
4.3 Evaluation 33
4.3.1 Online dating sites 33
4.3.2 Online Matrimony Site 34

5 Conclusion and Future Work 35

List of Figures

2.1 Example code with an SQLi vulnerability. 9

3.1 Overview of SQLR. 14

3.2 A simple SQL-like grammar 16

3.3 Shift-Reduce Graph and Final Graph 17

3.4 Context-free grammar for code in Figure 2.1 23

3.5 Attack patterns generated by SQLR. 27

3.6 Analysis on open-source web scanners 30

4.1 Traditional Blackbox Scanner 32

4.2 Overview of WeakAutoScan. 33

List of Tables

3.1 Parsing table of the sample grammar. 16
3.2 Evaluation Results. 27
3.3 Performance Evaluation with Threshold 4 29
3.4 Comparison of our attack patterns with the attack patterns of existing Web Scanner tools using `Webchess 0.9`. ✓means SQLi detected, and ✗means SQLi undetected. 30

Chapter 1

Introduction

Web applications have evolved considerably in terms of technology and functionality since the advent of CGI-based technology in 1993. These web applications can dynamically construct a web page in reply to each users request. Modern web applications provide rich, interactive user experience and have already become prevalent around the world with the success of a wide range of web services, such as on-line stores, e-commerce, social network services, etc.

As web technologies evolve, web applications have also been threatened by new security attacks. For example, web applications designed to interact with back-end databases are threatened by SQL injection [1]. Now-a-days, SQL injection attacks are becoming significantly more popular amongst hackers, according to recent data between Q1 2012 and Q2 2012, there has been an estimated 69 percent increase of this attack type [2].

Automated blackbox web vulnerability both commercial and open-source scanners have a very important role on helping the developers to detect vulnerability in their applications. Over the last few years, the web vulnerability scanner market has become a very active commercial space, with, for example, more than 50 products approved for PCI compliance [3]. These scanners test the security of the web applications by performing an attack, without malicious payload (i.e. they will not delete parts of the web application or the database it uses), against the web application that should be tested.

Web application scanners do have limitations. As most testing tools, they provide no guarantee of soundness. In the last few years, several studies have shown that state-of-the-art web application scanners fail to detect a significant number of vulnerabilities in test applications [4–6]. The main reason for these scanners to fail is because of limited attack vectors. These tools rely on attack vectors which are constructed with known input patterns used in past web vulnerability attacks.

Apart from this limitation, blackbox vulnerability scanners fail to detect vulnerabilities in authentication protocol implementations. One important feature that is common to most web sites is an authentication mechanism. Authentication protocols have historically been hard to design correctly and implementations have been found susceptible to logical flaws [7] . Web authentication protocols are no exception - several of these implementations have been found insecure in post deployment analysis [8–10]. Many web sites implement their own authentication protocols and their authentication logic are hidden from the user. Because this is so prevalent, for scanners to test if the authentication logic is implemented correctly they must properly handle authentication, possibly by creating accounts, logging in with valid credentials and, then check for implementation flaws in the authentication protocols. We say that the protocol is “weak”, because of poorly implemented authentication logic at the server side which makes easy for an attacker to bypass the authentication and we term such protocols as weak-authenticated (weak-auth) protocols. Current blackbox vulnerability scanners fail to handle such authentication mechanism [5], and therefore make them impossible to detect flaws in weak-auth protocols.

1.1 Research Overview

Our research is divided into two broad parts. First, we present SQLR (SQLi Revisited) to systematically validate input sanitizers of SQLi attacks.Second, we present a framework called *WeakAuthScan*, a semi-automated method to validate flaws in the authentication protocol implementations.

1.1.1 SQLR - SQLi Revisited

The root cause of SQLi vulnerabilities is the weaknesses in input validation, where the web application fails to detect malicious user inputs that can result in unexpected SQL queries. Therefore, a common solution is to check whether a user input used to generate SQL queries contains pieces of SQL commands. This type of checking code is often referred to as *input sanitizers* (in short, sanitizers). The main challenge faced by sanitizers is to detect all SQLi attack patterns, especially the ones not commonly used in attacks.

In order to validate sanitizers and detect injection vulnerabilities in web applications, a naive solution is to test the application with known input patterns used in past SQLi attacks using black box vulnerability scanners. However, such testing-based solutions are not effective in dealing with complex web applications, and thus often miss vulnerabilities. To make a more comprehensive analysis of the web applications, researchers have proposed solutions based on program analysis and symbolic execution [11–13]. The main idea is to generate a model of the web application's input sanitizer, and validate the model against SQLi attack patterns. Although such solutions have been shown to be effective in validating sanitizers and generating new SQLi attacks, they are significantly limited by their simple SQLi attack patterns. For example, the approach proposed by [13] only checks whether the input contains "`'or 1=1`". A more comprehensive set of attack patterns will help such approaches to detect more vulnerabilities.

Note that SQL injection attacks are caused by misinterpretation of inputs, i.e., user inputs intended for data are interpreted as SQL commands. A malicious input will result in a different parsing tree than benign inputs. This criteria have successfully been used as an effective detector in various solutions to SQL attacks [14–17]. Consequently, focusing on the parsing module of the database server is sufficient for generating SQLi patterns. Since the parsing module of the SQL language is based on a context-free grammar, we will use the grammar to guide the SQLi pattern generation. In essence, our SQLi attack pattern generation is based on systematically enumerating SQL queries that can be accepted by the grammar.

Although enumerating strings from the language of a context-free grammar is straightforward, our solution faces a unique challenge: because the SQL queries in a web application are generated by combining user inputs and command stubs, certain parts of the queries must be fixed to given values, while other parts are derived from user inputs. We call this constraint the *taint*[1] *constraint* in the generated SQL commands. Our technique must be able to systematically enumerate SQL queries under the taint constraint.

In this thesis, we present a novel approach, SQLR [2] discussed in Chapter 3, to systematically validate input sanitizers of SQLi attacks. The key technique of our approach is the grammar-guided generation of SQLi patterns. Using the SQL grammar as a guide, SQLR efficiently enumerates malicious SQL statements satisfying the taint constraint, which are then summarized into attack patterns. Once these attack patterns are combined with symbolic models of SQLi sanitizers, our approach validates sanitizers through generating malicious inputs that can bypass the sanitizers. We prototyped our approach and evaluated it using a number of real-world web applications. Our evaluation results demonstrated that our approach is effective in validating SQLi sanitizers and generating attack inputs.

Guided by the SQL grammar, SQLR discovered several new SQLi attack input patterns.

[1]Data derived from user inputs are often called "tainted".
[2]SQLR stands for SQLi Revisited.

Some of the patterns are even not meaningful to human. In our experiment, we verified that three open source web scanners do not include the SQLi attack patterns generated by our approach, and thus giving false negatives during vulnerability scanning.

In summary, we made the following contributions:

- We developed a complete solution to systematically validate SQLi attack sanitizers and generate SQLi attack inputs. To the best of our knowledge, SQLR is the first approach that uses database grammar to guide sanitizer validation and SQLi vulnerability detection.
- We proposed a novel technique to use a context-free grammar to guide the efficient enumeration of strings of the grammar under a taint constraint.
- Using our technique, we generated several new SQLi attack patterns. Our attack patterns can be used by existing solutions for better validating SQLi sanitizers.
- We show that both open source and commercial current black-box web scanners attack pattern is not complete.

1.1.2 WeakAuthScan

In practice, checking authentication protocols implementation is difficult due to lack of complete implementation (such as missing source code of protocol participants). As discussed in the introduction, blackbox scanners does not scale well for checking flaws in authentication protocols. The key challenges in ensuring that applications authenticate and federate user identities securely is checking the implementations of authentication logic. Using blackbox scanning, it is difficult due to lack of complete information (missing source code of authentication logic) and also the web applications require a user to create an account to access the authentication system which these scanners fails to do it automatically [5].

In this thesis, we present a framework *WeakAuthScan* to automatically extract the authentication protocols logic. *WeakAuthScan* assumes no knowledge of the protocol being checked and does not require access to the source code of the implementation. We propose a blackbox analysis by exchanging messages between the authentication server and the web user.

We apply *WeakAuthScan* to study real-world web sites. We tested on two popular websites which implement their authentication logic and have millions of users sharing deeply personal information. *WeakAuthScan* successfully recovers the authentication logic and reports security flaws in these implementation without their knowledge.

In summary, we made the following main contributions:

- First, we propose automatic technique to extract the authentication protocols from the messages exchanged between the server and the user. Our approach works with little user inputs and without requiring any knowledge of the protocol.
- Second, we apply our approach to two real-world web sites and we were successfully able to find security flaws in the implemented authenticated protocols.

Chapter 2

Web Application Vulnerability and Black Box Scanners

In this chapter, we begin by describing the software architecture of the black-box web vulnerability scanners. We then discuss the vulnerability categories mainly, SQL Injection and Weak Authentication Systems, which they fail to detect.

2.1 Web Vulnerability Black Box Scanners

Black-box web vulnerability scanners are a class of tools that can be used to identify security vulnerabilities in web applications. These tools evaluate the security of web applications automatically with little or no human support. These tools access a web application in the same way users do, and, therefore, have the advantage of being independent of the particular technology used to implement the web application.

Black-box testing consists of analysis of the program execution from an external point-of-view without actually looking into the source code. In short, it consists of exercising the software and comparing the execution outcome with the expected result. Testing is probably the most widely used technique for verification and validation of software. There are several levels for applying black-box testing, ranging from unit testing to integration testing and system testing. In this thesis, penetration testing that are black-box testing refers to a methodology where an ethical hacker has no knowledge of the system being tested. The goal of a black-box penetration test is to simulate an external hacking or cyber warfare attack and report if there is any vulnerabilities in the web application.

2.1.1 Black-Box Penetration testing

Black-Box Penetration testing, consists of the analysis of the program execution in the presence of malicious inputs, searching for potential vulnerabilities. In this approach the scanners does not know the internal working of the web application and it uses fuzzing techniques over the web HTTP requests. The scanner needs no knowledge of the implementation details and tests the inputs of the application from the user point of view. The number of tests can reach hundreds or even thousands for each vulnerability type. These penetration tools provide an automatic way to search for vulnerabilities avoiding the repetitive and tedious task of doing hundreds or even thousands of tests by hand for each vulnerability type. Despite the use of automated tools, in many situations it is not possible to test all possible input streams, as that would take too much time. So, as soon as software specifications are complete, test cases can be designed to have the biggest coverage and representativeness possible. This test approach may leave program paths untested and can lead to unnecessary repetition of tests between developers and testers. The most common automated security testing tools used in web applications are generally referred to as web security scanners (or web vulnerability scanners). Web security scanners are often regarded as an easy way to test applications against vulnerabilities. These scanners have a predefined set of tests cases that are adapted to the application to be tested, saving the user from define all the tests to be done. In practice, the user only needs to configure the scanner and let it test the application. Once the test is completed the scanner reports existing vulnerabilities (if any detected). Most of these scanners are commercial tools, but there are also some free application scanners often with limited use, since they lack most of the functionalities of their commercial counterparts.

Three very popular commercial security scanners and also the leader in the market which support web services testing are Acunetix Web Vulnerability Scanners [18], HP WebInspect [19] and IBM Rational Appscan [20].

2.1.2 Commercial Tools

HP WebInspect is a tool that performs web application security testing and assessment for today's complex web applications, built on emerging Web 2.0 technologies. "HP WebInspect delivers fast scanning capabilities, broad security assessment coverage and accurate web application security scanning results [19]. This tool includes pioneering assessment technology, including simultaneous crawl and audit (SCA) and concurrent application scanning. It is a broad application that can be applied for penetration testing in web-based applications.

IBM Rational AppScan “is a leading suite of automated Web application security and compliance assessment tools that scan for common application vulnerabilities” [20]. This tool is suitable for users ranging from non-security experts to advanced users that can develop extensions for customized scanning environments. IBM Rational AppScan can be used for penetration testing in web applications, including web services.

Acunetix Web Vulnerability Scanner “is an automated web application security testing tool that audits a web applications by checking for exploitable hacking vulnerabilities” [18] . Acunetix WVS can be used to execute penetration testing in web applications or web services and is quite simple to use and configure. The tool includes numerous innovative features, for instance the AcuSensor Technology.

Many other black-box tools were proposed in the past. Although those works target web applications, and not web services, we introduce some here due to the relevant innovations they introduced.

2.1.3 Free/Open Source Tools

w3af

w3af is the abbreviation of the Web Application Attack and Audit Framework [21]. It is an open-source program, written in Python. It uses plug-ins to perform the attacks on web applications. A description of the vulnerabilities that these plug-ins claim to detect can be found on the tool’s website . It uses a menu-driven text-based structure, but it also has a GUI.

wapiti

wapiti is another open-source program written in Python [22]. It works from the command-line completely automatically. However, command-line options can be used to customize scanning.

sqlmap

sqlmap is open-source penetration testing tool [23] that automates the process of detecting and exploiting SQL injection flaws.

```
1 <?
2 $name = $_POST['username'];
3 $sql = "SELECT * FROM user_table WHERE username='".$username."'";
4 if (!preg_match("/ OR /",$username)){
5   $result = $db->sql_query($sql);
6 }
7 ?>
```

Figure 2.1: Example code with an SQLi vulnerability.

2.2 SQL Injection Attack and Input Sanitizers

2.2.1 SQL Injection

Web applications are commonly implemented by a multi-tiered architecture. For example, in a three-tiered web application, the application server receives requests from users and generates SQL statements to query the database server. An SQL injection (SQLi) attack occurs when an attacker changes the intended effect of an SQL query by inserting SQL keywords or operators into the query, which gives attackers unauthorized access to data stored in the database.

Consider the code fragment shown in Figure 2.1. This code retrieves details of a user account. We will use this code throughout the paper to illustrate how SQLR works. Initially, at line 2, the variable `$name` is assigned to values from the POST parameter, which contains user inputs. Line 3 prepare the SQL query to the database. At line 4, we check if the user input `$username` contains the keyword “`OR`”. If not, it sends the query to the database server (Line 5); otherwise, this query will not be executed. This is a simple check of SQLi attempts. However, if the check cannot cover all possible SQLi cases, this program is still vulnerable.

2.2.2 Input Sanitizer

To prevent malicious SQLi inputs, the program has to ensure that the query executed in the database server is intended by the program. Because inputs from malicious users may contain arbitrary values, a common solution is to add code, such as line 4 in Figure 2.1, to check user inputs and detect SQLi attempts, a process often referred to as *input sanitizing*. The code for input sanitizing is called *input sanitizers*, or *sanitizers* in short. Many scripting languages provide built-in sanitizers, such as `trim`, `mysql_real_escape_string` in PHP. Alternatively, programmers can build their own sanitizers.

2.2.3 Current Solutions to Validate Sanitizers

The safety of a web application depends on the quality of input sanitizers. Recently, a few techniques [11, 12, 24, 25] have been proposed to check the correctness of sanitizers and detect

SQLi attacks. They analyze web application source code, and check whether a web application fails to sanitize inputs using patterns describing known attack patterns.

These solutions are limited by the available attack patterns. For example, some of the solutions only check the existence of a few SQL keywords "`OR`", "`UNION SELECT`", "`HAVING`", and "`ORDER BY`" [12, 13, 25]. Sanitizers passing the check using such patterns will not prevent attacks using other patterns, especially when an attack pattern is previously unknown.

In Figure 2.1, line 4 is the input sanitizer. Whether an approach can successfully detect the weakness of this sanitizer depends on the attack patterns available to the approach. In this paper, we present a technique to systematically generate more attack patterns, which can be used by existing solutions to validate the sanitizers. Recent approaches [13,24] generate sanitizers from known attack patterns. Our attack patterns can be used by these approaches to generate more complete sanitizers.

2.3 Weak Authentication Systems

Web authentication mechanisms are fast evolving. Many web sites implement their own authentication protocols and manage their authentication logic either from third party mechanism or by itself. Most of the web sites implement AutoAuth (Automatic Authentication) mechanism [26] which allows users to automatically log in to your server. For example "you might use it if you have another software on your website which clients already log into, and once they have logged into that you don't want them to have to re-authenticate again separately to access the server". The way it works is by constructing a special URL to redirect the user to the server, which verifies and if valid, activates the users login session in the server automatically. One advantage using the mechanism it that it skips the need to know the users password to access the users account.

The security comes from having a key/nonce that is shared only between you and the third party code/server you're making the request to, and only knowing that this key allows an autoauth request to be constructed for your server. Generating the key/nonce is the important component involved in the security of AutoAuth. If attacker is able to guess this key then the entire authentication protocol is compromised and we call such systems as weak authentication systems.

2.4 Related Works

2.4.1 SQLi attack detection.

Several solutions have been developed to detect injection attacks [14–17]. They detect an attack by checking whether user inputs make the generated command's parsing tree different from that of the intended command. This idea is also used by our approach as the basis of malicious query generation. However, instead of detecting attacks when they happen, our approach aims to generate attack patterns and validate sanitizers.

SQLi vulnerability detection and sanitizer validation.

A few approaches in this category is based on static analysis of web application code. The basic technique is either to use static analysis to identify program locations that generate queries from user inputs [27–29], or to perform string analysis to model the queries outputted by the web application [30, 31].

Using such basic techniques, researchers proposed several techniques to discover SQLi vulnerabilities. Wassermann and Su [32] present a fully automated approach to detect subtle SQLi flaws. Adrilla [12] uses the input-generation component from Apollo [33] to generate SQLi attacks automatically. The Apollo input generator is based on systematic dynamic test-input generation that combines concrete and symbolic execution [34]. Saner [11] combines static and dynamic analysis to find cross-site scripting (XSS) and SQLi vulnerabilities by validating the sanitizer functions in web applications. Saner focuses on the sanitizing process and creates attack vectors by looking into the static dependency graph. Wassermann et al. developed a tool [35] that executes a PHP applications on a concrete input and collects symbolic constraints. They use dynamic test input generation to find attacks on full PHP web applications. Their tool performs source-code instrumentation and backward-slice computation by re-executing and instrumenting additional code. Upon reaching an SQL statement, their tool attempts to create an input that exposes SQL injection vulnerability, by using a string analysis [30].

Solutions in this category analyzes the web application code to detect SQLi vulnerabilities. In contrast, our approach analyzes the database using the grammar as a guide to systematically generate attack patterns. Our approach has complementary advantage to solutions in this category, generating new attack patterns to enhance such solutions.

2.4.2 Web application sanitizer generation.

BEK [24] is a language for modeling string transformations. It is used for writing sanitizers that enable systematic reasoning about their correctness. Another solution [36] develops an automatic system that, given a template and a library of sanitizers, automatically sanitizes each untrusted input with a sanitizer that matches the context in which it is rendered.

For adding sanitizers in web applications, the places where untrusted data appear in the code are hard to find. Even when they are found, it is not immediately clear which sanitizer should be used. To answer such questions, Weinberger et al. [37] did a formal study of common XSS sanitizing mechanisms, where they present a model of the web browser's parsing internals to explain the subtleties in XSS sanitizing.

2.4.3 Evaluating Web Vulnerability Scanners

Many researchers have assessed web vulnerability scanners, tested their performances, analyzed their behavior and gave details about the limitations identified [4–6]. Now-a-days they are growing body of literature on the evaluation of web vulnerability scanners. [38] implemented automated black-box web vulnerability scanners which generates more effective test cases targeting SQLI and XSS vulnerabilities. But these test cases are limited to known attack patterns and fails to detect SQLi attack if new attack pattern exists. They developed a new web vulnerability scanner and tested it on about 25,000 live web pages. Since no ground truth is available for these sites, the authors cannot discuss false negative rate or failures of their tool. [39] tested four web scanners on 300 web services but they report high rates of false positives and false negatives. [4–6] evaluated commercial web scanner and reports that none of the scanners report all the vulnerabilities present in the web application.

[40] compared three scanners against three different applications. To measure the effectiveness of each scanner they used code coverage and other metrics. In the follow-up study [41], the same authors assessed seven scanners and compared their detection capabilities and the time taken by each scanner to report the vulnerabilies in the web application. [42] assessed five unnamed scanners against a custom benchmark and in their survey they reported that black-box scanners perform poorly.

Chapter 3

Grammar-guided Validation of SQL Injection Sanitizers

SQL injection (SQLi) attacks are an important class of attacks on web applications. Commonly implemented by a three-tiered architecture, a web application executes its core business logic in the application server, and stores its data in the back-end database server. The application server receives requests from the user and interacts with the database server by generating SQL commands. In an SQLi attack, attackers inject malicious database queries through an input, causing data leakage and damage in the database. For the convenience to generate such SQL commands from user inputs and command stubs, weak-typed scripting languages, such as Perl and PHP, are widely used in building web applications. However, the convenience often results in SQLi vulnerabilities.

As discussed in Chapter 1, the root cause of SQLi vulnerabilities is the weaknesses in input validation. Therefore, a common solution is to use input sanitizers (refer Chapter 2). One of the main challenges faced by sanitizer is to detect all SQLi attack patterns, especially the ones not commonly used in attacks.

To test the effectiveness of these sanitizers often developers execute penetration tests on them. As numerous developers are not specialized on security, the common way to search for security vulnerabilities in the sanitizers is through blackbox web vulnerability scanners.

In this chapter, we present SQLR, a tool to systematically validate the input sanitizers of SQLi attacks. Section 3.1 gives an overview of our solution. Section 3.2 presents the key technique, grammar-guided SQLi attack pattern generation. Section 3.3 describes details of the implemen-

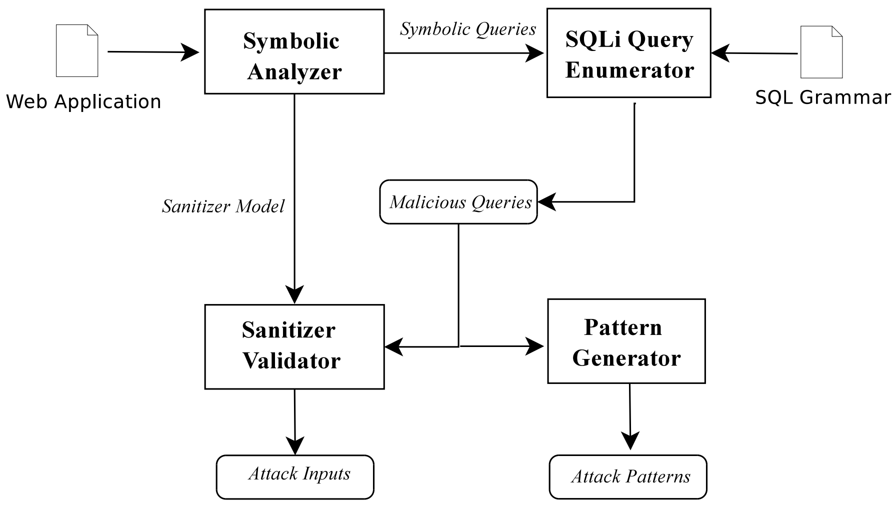


Figure 3.1: Overview of SQLR.

tation of our approach. We show the evaluation result in Section 3.4.

3.1 Overview of Our Solution

Figure 3.1 illustrates the overview of our approach, which consists of four main components: symbolic analyzer, SQLi query enumerator, sanitizer validator, and pattern generator.

The symbolic analyzer component analyzes the web application to identify the queries generated by the web application as symbolic strings, for example, "`SELECT * FROM user_table WHERE username=`$SymbolicPart$", where $SymbolicPart$ denotes the part derived from user inputs. It also outputs a model of the web application's input sanitizer. The SQLi query enumerator component uses the symbolic queries and the SQL grammar to enumerate all possible malicious SQL queries that can be generated by the application. The malicious SQL queries and the model of sanitizer are used by the sanitizer validator, which verifies whether the sanitizer can successfully detect all malicious SQL queries. Finally, to benefit existing solutions in sanitizer validation, the pattern generator summarizes the generated malicious queries into regular expression patterns that can be used by those solutions.

One of the key component of SQLR is the SQLi query enumerator. It uses the SQL grammar to guide the enumeration of malicious SQL queries. We will describe this component in the following section.

3.2 Grammar Guided Generation of SQLi Attack Patterns

The key technique of our approach, SQLR, is to use the SQL grammar to guide the generation of SQLi attack patterns. Because the SQLi attack queries are generated by the web application, they must be instances of a symbolic string, such as "`SELECT * FROM user_table WHERE username=`$SymbolicPart$" for the code in Figure 2.1. Note that the SQL grammar is a context-free grammar, so we define the problem as follows:

Problem definition: *Given a context-free grammar G, a symbolic string S, our goal is to efficiently enumerate concrete instances of S from G's language, where each enumerated string has a unique parsing tree.*

In the context of SQL queries generated by web applications, being an instance of S (the symbolic query) ensures that the enumerated strings are possible queries of the web application. Requiring each query to have a unique parsing tree ensures that the enumerated strings are malicious SQL queries.

3.2.1 Challenges

A straightforward solution is to use a top-down method on the grammar G to enumerate all strings of G, without considering the symbolic string S and the requirement for different parsing tree. Specifically, we begin with the start symbol of the grammar G, and recursively substitute non-terminals according to the productions of G. After a string is enumerated, we then check whether it is an instance of S and its parsing tree is different from previously enumerated strings. However, this solution is not efficient: it enumerates a large amount of strings that do not satisfy the two requirements (being instance of the symbolic string and having a unique parsing tree).

Since the generated strings are instances of the symbolic string S, they have common concrete parts, i.e., substrings with non-symbolic values. As an example, in the code shown in Figure 2.1, the generated SQL queries have the prefix "`SELECT * FROM user_table WHERE username = '`". We can leverage those concrete parts and the internal states of the grammar to make input enumeration more efficient. Intuitively, we can start with parsing the common prefix using the SQL grammar, and identify the grammar's "internal state" led to by the prefix. Continuing from that state, we enumerate SQL queries that result in a different parsing tree. In this way, we avoid generating lots of SQL queries that do not satisfy the two requirements.

We illustrate our approach using a simple grammar shown in Figure 3.2. This grammar is a simplified version of the SQL grammar. We apply LALR(1) parsing to generate a parsing table,

```
oneselect ::= SELECT STAR from where_opt SEMI (R0)
from      ::= FROM ID                         (R1)
where_opt ::= WHERE expr                      (R2)
expr      ::= expr EQ expr                    (R3)
expr      ::= expr AND expr                   (R4)
expr      ::= ID                              (R5)
```

Figure 3.2: A simple SQL-like grammar

State no	$	SELECT	STAR	FROM	ID	WHERE	SEMI	EQ	AND	oneselect	from	where_opt	expr
0		S10								Accept(A)			
1				S13							S2		
2						S3						S11	
3					S9								S6
4					S9								S7
5					S9								S8
6							R2	S4	S5				
7							R3	R3	R3				
8							R4	R4	R4				
9							R5	R5	R5				
10			S1										
11							S12						
12	R0												
13					S14								
14						R1							

Table 3.1: Parsing table of the sample grammar.

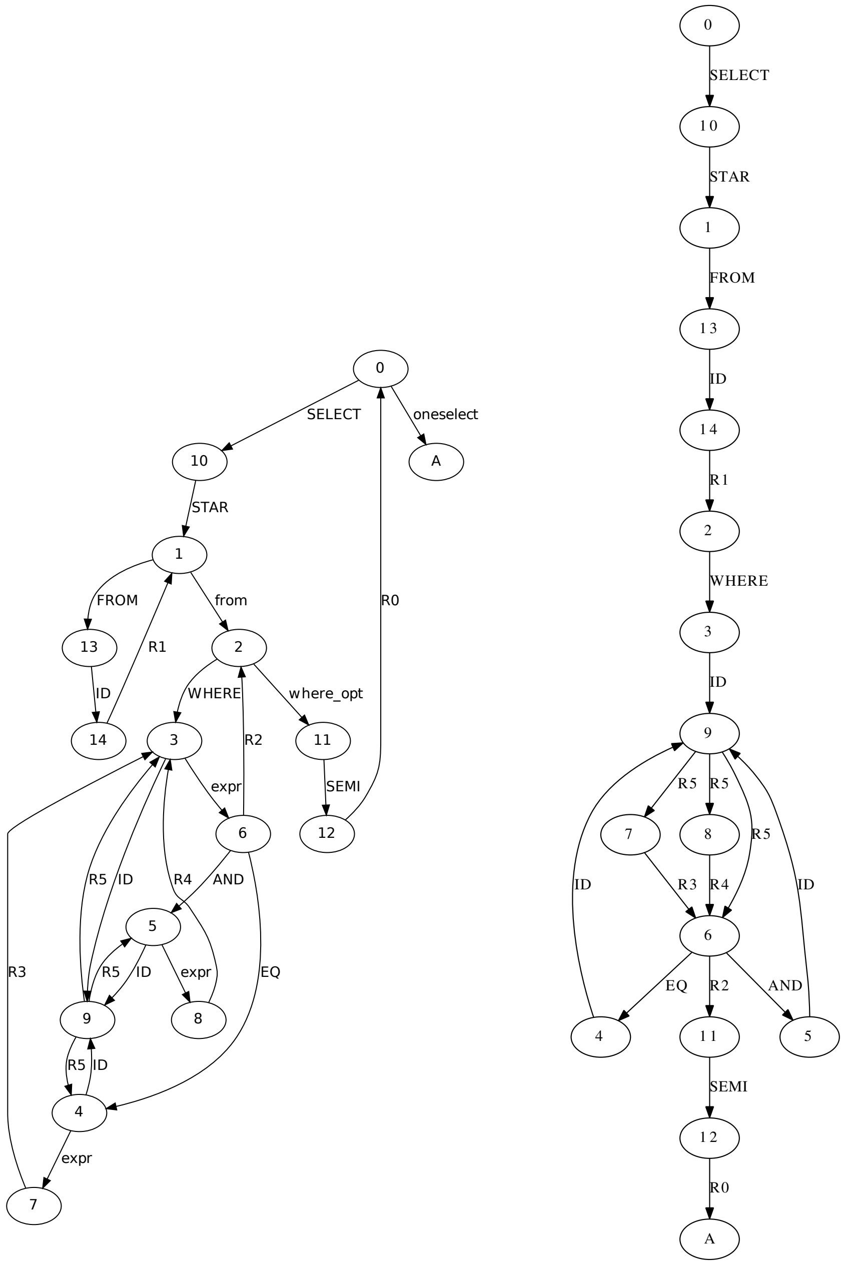
shown in Table 3.1.

In a query generated by the program shown in Figure 2.1, e.g., “`SELECT * FROM user_table WHERE username='alice'`”, its prefix “`SELECT * FROM user_table WHERE username='`” is lexically recognized as terminal nodes defined by the grammar: `SELECT STAR FROM ID WHERE ID EQ`. This prefix drives the LALR(1) parsing process to the `state 4` of its parsing table (Table 3.1). From this state, if we make the rest of parsing steps different from those of the original query, we get a query of different parsing tree, but with same prefix.

If we represent the relationship of the states in the LALR(1) parsing table into a state graph, whose nodes are the states and edges are shift operations, each parsing tree corresponds to a path in the graph. To get a different parsing tree, we need to find a different path in the graph. Using the previous example “`SELECT * FROM user_table WHERE username=`$SymbolicPart$”, the concrete prefix “`SELECT * FROM user_table WHERE username='`” will lead a path to `state 4` in the parsing tree. To find different path in the parsing graph, we look for different $shift$ operations leading out of `state 4`. We thus see a shift operation to `state 9` on ‘ID’. This requires the *SymbolicPart* begins with ’ID’. In this way, we will enumerate a string of the grammar, which leads down a unique path.

However, not all paths from the start state to the accept state in the state graph correspond to valid strings of the grammar G. The challenge is in minimizing such invalid paths during path selection. We achieve this by optimizing the state graph.

Without loss of generality, we focus on the simple SQL grammar in the rest of this section,



(a) Shift-Reduce Graph (b) Final Graph

Figure 3.3: Shift-Reduce Graph and Final Graph

but our technique can be applied to other context-free grammars as well.

3.2.2 Optimizing SQL Parsing Graph

For easier presentation and illustration, we represent the parsing table into an equivalent state graph. We then present our technique to optimize the state graph to reduce the number of paths during path enumeration.

First, we construct a shift-reduce graph from the parsing table. The shift-reduce graph is constructed by looking at all the shift and reduce operation in the parsing table. We create a state in the shift-reduce graph for each state in the parsing table. If there is a shift operation from state s to another state s', we add an edge from between state s and s' in the shift-reduce graph. If there is a reduce operation on state s we look for its corresponding rule from the grammar. From state s, we backtrack to number of symbols present on the right hand side of the rule to reach state s' and label its corresponding edge with the reduce operation. The complete shift-reduce graph for the sample grammar is shown in Figure 3.3(a).

To reduce the number of paths, we further optimize the shift-reduce graph, converting the shift-reduce graph into the final graph by removing the non-terminals. To do this, we need to handle reduce operations in the graph, since all the reduce operation depends on the non-terminal symbol. In our approach, when we see a reduce operation on state s, we look into the corresponding rule from the grammar. The grammar will tell us how many symbols it has on the right hand side, using this we can backtrack to that particular state. Once we backtrack, we go to a new state(s') with a corresponding outgoing edge that matches the right-hand rule of the grammar. This is how shift-reduce graph is created. Now, to construct the final graph we look into the state s' and see its outgoing edge which matches the left hand side of the grammar corresponding to the reduce operation to reach a new state s''. Finally, we add an edge from state s to state s'' and label its corresponding edge with the reduce operation. For example, in the shift-reduce graph on `state 7`, we see a reduce operation '$R3$' which goes to state 3. On `state 3`, when it sees a non-terminal '$expr$' it goes to `state 6`. This symbol matches with the reduce operation '$R3$'(left hand side of the grammar). Therefore, removing the edge from `state 3` to `state 6` and adding an edge from `state 7` to `state 6` will contain the same behavior as the reduce operation. In this way, all the non-terminal labeled edges are removed. The final graph representation of the parsing table is shown in Figure 3.3(b).

3.2.3 Enumerating Queries through Symbolic Analysis of the Final Graph

Algorithm 1 Symbolic analysis on the final graph

Input: Starting node: N, Starting stack: S, Accepting state A, Graph G, Threshold $threshold$
Output: Query set Q

```
1:  $L = \emptyset$
2:  $lookahead = \top$
3:  $q = null$
4:  let $M$ be an empty stack
5:  $E = get\_out\_going\_edges(N)$
6:  for all $e \in E$ do
7:      push $\langle e, S, null, 0, \top \rangle$ into $M$
8:  end for
9:  while $|M| > 0$ do
10:     $\langle e, S, q, n, l \rangle = pop(M)$
11:     $lookahead_e == get\_lookahead(e)$
12:     if $lookahead_e \cap l == \emptyset$ then
13:         continue;
14:     end if
15:     if $e$ is an shift edge then
16:         $l' = \top$
17:         $S' = shift\_update\_stack(e, S)$
18:     else $\{e$ is a reduce edge$\}$
19:         if $S$ is not consistent with the reduce then
20:             continue;
21:         end if
22:         $l' = lookahead_e \cap l$
23:         $S' = reduce\_update\_stack(e, S)$
24:     end if
25:     $dst = get\_dst(e)$
26:     $q' = q@get\_terminal(e)$
27:     if $dst == A$ then
28:         $Q = Q \cup \{q'\}$
29:     end if
30:     if $n < threshold$ then
31:         $E = get\_out\_going\_edges(dst)$
32:         for all $e' \in E$ do
33:             push $\langle e', S', q', n+1, l' \rangle$ into $M$
34:         end for
35:     end if
36: end while
```

Once the final graph representation of the SQL grammar is ready, we will carry out symbolic analysis on the final graph to enumerate inputs.

The algorithm for symbolic analysis using the final graph is shown in Algorithm 1. The inputs of the algorithm include the final graph G, a starting node N, a starting stack S, an accepting state A and a threshold $threshold$. Recall that our goal is to enumerate all valid SQL queries matching a symbolic query. Suppose the constant prefix of the symbolic query is pre, after parsing pre, we record the state N and the parsing stack S. The output Q of Algorithm 1 is a set of partial queries. When concatenated with the constant prefix pre, each partial query represents one valid SQL query.

In Algorithm 1, we use @ to denote string concatenation. The destination node of a directed edge e can be obtained using $get\_dst(e)$. We use $get\_terminal(e)$ and $get\_lookahead(e)$ to get the terminal symbol and lookahead symbol set associated with an edge e correspondingly.

Algorithm 1 traverses graph G in a depth-first fashion to find all valid queries. It uses stack M to maintain untraversed edges. Each stack element in M consists of an shift/reduce edge e, an parsing stack status S, a partial query q, number of traversed edges n and a lookahead symbol set l. The edge e is used for depth-first traversal, other information maintained in stack are explained in the following. In addition to standard depth-first traversal, we do several additional checks to avoid generating redundant queries and also to guarantee generating only valid queries. To rule out invalid queries, we maintain the parsing stack when traversing the graph G. The initial state of the parsing stack is given as the input S of Algorithm 1. Whenever we traverse an edge, the stack is maintained correspondingly (line 17,23). When we traverse a reduce edge, we check whether the reduce operation is consistent with the current stack status. A reduce edge inconsistent with current stack status is never traversed (line 19-21).

We also keep track of the assumptions on lookahead symbols (l in each stack element) to avoid redundant graph traversal. Our graph is built based on the parsing table of LALR(1) grammar. LALR(1) parsing uses one lookahead symbol to determine the next parsing action. An action can only be taken if the next to-be-parsed symbol matches the lookahead terminal of that action. Mapping to our graph, an edge can only be traversed if the next to-be-parsed symbol matches the lookahead symbol of that edge. However, the next to-be-parsed terminal is *unknown* to us since we are enumerating all queries instead of parsing one concrete query. Therefore, when we take an edge, we are essentially making the assumption that the next symbol (lookahead symbol) is consistent with the edge. When we traverse a path consisting of several edges, the assumptions for all edges have to be consistent. The check for consistency on lookahead terminals are in line 11-14. For example, when we take the edge from `state 8` to `state 6` in Figure 3.3(b), we make the assumption that the next coming symbol is in $\{$SEMI,EQ,END$\}$. When we later take the shift edge from `state 6` to `state 4`, we check whether the symbol being shifted in falls into the set $\{$SEMI,EQ,END$\}$. Since the symbol is

EQ, which is in the set, the edge from `state 6` to `state 4` can be traversed.

The number n in each stack element of M is used to control the length of the generated query. We only continue exploration if n is less than $threshold$. When we reach the accept state A, we output one partial query q. Because we only consider the concrete prefix of the symbolic query in the previous steps, we need to check whether the generated query matches other parts of the symbolic query.

Handling multiple symbolic parts. If there multiple symbolic parts in the symbolic query, all the symbolic parts should be used together to guide the enumeration in the ideal solution. We can use our approach to explore one symbolic part at a time. It works when there is no dependency among the symbolic parts, which is typical for generated SQL queries. Moreover, a property of SQLi attacks makes the problem easier to handle. In SQLi attacks, attackers typical inject SQL pieces in the first symbolic part of the SQL query and comment out the rest of the query. Here is an example:

Symbolic query:

```
SELECT * FROM user_table WHERE username=<symbolic_part>
AND password=<symbolic_part>;
```

After SQL injection:

```
SELECT * FROM user_table WHERE username='' OR 1 = 1; --
AND password=<symbolic_part>;
```

In the above query, we see that attackers have to inject SQL conditions in the first symbolic part and comment out the second symbolic part, which is necessary to produce a tautology. Therefore, we need to mainly focus on the first symbolic part of the symbolic query. If we fail to generate a malicious query for the first symbolic part we then focus on the second or the next symbolic part in the query.

3.2.4 Generation of SQLi Attack Patterns

Once we have enumerated queries for a given symbolic query, the next step is to create an attack pattern out of it. These patterns will be used by existing solutions to validate sanitizers.

The attack patterns with similar behavior or structure is grouped together. Based on the longest common substrings, the enumerated queries are clustered into families. For each cluster obtained, we learn the regular expressions from that family. In previous approaches [12, 13], the attack patterns are based on tautology, which makes one or more conditional statements in the query always true. Therefore, only one attack pattern is used by those approaches:

```
SELECT ID FROM ID WHERE ID EQ STRING OR INTEGER EQ INTEGER
```

In our case, since we generate more number of attack patterns, we use longest common substring to group them into different clusters as shown below:

```
SELECT ID FROM ID WHERE ID EQ EXISTS ..
SELECT ID FROM ID WHERE ID EQ MINUS PLUS EXISTS  ..
SELECT ID FROM ID WHERE ID EQ STRING  ..
SELECT ID FROM ID WHERE ID EQ STRING LE ID HAVING ..
SELECT ID FROM ID WHERE ID EQ STRING PLUS MINUS EXISTS ..
```

Each cluster contains a set of malicious queries that can be used to exploit the program. In Section 3.4, we discuss the attack patterns in detail.

3.2.5 Discussion of Path Sensitivity and Over Approximation

As mentioned earlier, SQL parsing tree provides a good oracle for checking SQL injection attacks. It is worth noting that this checking is specific to a path in the web application. Let v be an variable whose value is SQL query sent to the database. Suppose when executing with input t, v gets a benign query q_1. Let t' be another input that follows the same path as the path of t. When executing the application with input t', v gets the value q_2. If q_1 and q_2 has different parsing tree, then q_2 must be malicious. This assertion is based on the fact that the two queries q_1 and q_2 are derived from the same program path in the application. Therefore, any difference in parsing tree is unexpected. However, if q_1 and q_2 are derived from different program paths in the application, the above assertion does not hold anymore. When derived from different program paths, the different parsing tree of q_1 and q_2 could be due to different ways of computing v along the two different paths instead of malicious query injection.

Given a variable v, in our implementation we compute the over-approximation of all possible values of v along all program paths using Algorithm 1. Different program paths are not distinguished. Hence, when we find an SQL query that has different parsing tree from the parsing tree of the reference benign query, it might be a false positive. To remove those false positives, we validate whether a query is indeed malicious after it is generated.

```
cfg temp0 := |temp2 ;
cfg temp1 := temp5 \079;
cfg temp2 := |temp6 \044| temp2 \044| temp5 \044 |temp1 \044;
cfg temp3 := \001|\002|.. ..|\251|\252;
cfg temp4 := \001|\002|.. ..|\251|\252;
cfg temp5 := temp2  ;
cfg temp6 := temp6 temp8 |temp2 temp10 |temp5 temp4 |temp1 temp3 ;
cfg temp7 := "SELECT * FROM user_table WHERE username="\044;
cfg temp8 := \001|\002|.. ..|\251|\252;
cfg temp9 := temp7 temp6 \044|temp7 temp5 \044|temp7 temp1 \044|temp7 temp0 \044;
cfg temp10 := \001|\002|.. ..|\251|\252;
```

Figure 3.4: Context-free grammar for code in Figure 2.1

3.3 Implementation

In this section, we discuss the implementation details of our approach, SQLR as well as techniques we used for optimizing our approach.

3.3.1 Symbolic Analyzer of Web Applications

In this component, we model the sanitizing procedure using techniques from static taint analysis with string analysis. Existing solutions model sanitizers using various methods, such as DFA with transducers or symbolic execution. In our approach, given a PHP web application, we use PHPSA [30], a string analyzer for PHP programs, to represent the sanitizing procedure S into a context free grammar G_S, whose language is exactly the set of strings that can pass S. PHPSA takes two inputs: the PHP web application and an input specification that describes the set of possible inputs to the program (POST parameters).

To illustrate the string analysis, let us consider the example code in Figure 2.1, which contains a regular expression based sanitizing code. The input specification defined for the example is

```
$_POST['username'] = [/.*/]
```

which says, the input expects any arbitrary value. Using the two inputs to PHPSA, we generate the grammar shown in Figure 3.4.

The string analyzer supports only limited functions from PHP standard library, that is, it does not support all features for PHP language. Currently we manually approximate unsupported lines of PHP code and verify that the changes do not affect the sanitizing process. We will add support for these features using the approach discussed in [32].

3.3.2 Multiple Query Generation

Some of the SQLi attacks contain a combination of two or more queries to make the attack effective. Such kind of attacks are called *piggy-backed queries*. Among SQLi attacks, this type of attack is one of the most dangerous and harmful attacks [25]. The intention of such kind of attack is to extract data, to add or modify data, to perform denial of service, and to execute remote commands.

In the example of Figure 2.1, if the attacker inputs "`'; DROP TABLE user_table;`" into the user-name field, the application generates the query:

```
SELECT * FROM user_table WHERE username='';
DROP TABLE user_table --';
```

After completing the first intended query, the database recognizes the query delimiter (";") and executes the injected second query, which drops the table user_table. The same attack technique can also be used to insert new users into the database or cause denial of service attack by using 'SHUTDOWN' command.

To generate multiple queries or piggy-backed queries, we need to add a rule to the database grammar that parses the graph again from the starting node by adding an edge from the accept node back to the start node. In the simple SQL grammar the following production rule is added:

```
Multiple_Query := oneselect;
      | oneselect;Multiple_Query
```

3.3.3 Sanitizer Validator

Once we generate the attack patterns to describe SQLi attempts, as well as the symbolic models of the sanitizer, our next step is to validate the sanitizing routine. We use Hampi [43], a solver for constraints over fixed-size string variables. Hampi constraints may contain a fixed-size string variable, context-free language definitions, regular language definitions and operations, and language-membership predicates. Hampi can be used in finding a string from the intersection of a bounded context-free language (CFL) and a regular language (RL). In our case, the CFL contains all possible string that can pass the sanitizer and RL represent a set of valid queries generated from the enumeration process. If Hampi finds a string, it means that the string is present in both the CFL and RL.

In our approach, the query generated using the grammar is represented into regular language(RL) as shown below

```
reg Idchar := or(['0'-'9'], ['a'-'z'],['A'-'Z']);
reg Id := concat(Idchar, star(Idchar));
reg Query := concat("SELECT ",\042," FROM user_table WHERE username='",Id,"' LE ",Id,"
     HAVING ",Id," EQ ",Id);
```

Given a set of regular languages, in our case the queries generated from the grammar and the context free language of the sanitizer, Hampi outputs a string if the intersection between the CFL and RL is not empty, otherwise reports that the intersection is empty. The string which Hampi outputs is an instance of attacks to bypass that sanitizer. For example, solving the CFL defined in Figure 3.4 and the regular language above, Hampi outputs the following string,

```
aa' LE 2 HAVING 1 EQ 1
```

The above string is the attack pattern for the sanitizing function defined in Figure 2.1. Finally, the malicious input generated for that program is

```
aa' < 2 HAVING 1 = 1
```

Therefore, we say that Figure 2.1 is vulnerable to SQLi attack.

3.3.4 Proactive Pruning of Blocked Queries

When a non-trivial sanitizer is applied in the web application, most of the generated queries could not pass the sanitizer and would be pruned away in the last step. To reduce the time cost by generating these queries, we proactively prune queries that cannot pass the sanitizer functions.

Given a program variable v whose value is SQL query, we over-approximate all the queries that v can hold using the approach mentioned in Section 3.3.1. Let L_v be the set of all queries that v can hold. Note that L_v already takes sanitizer into consideration. In earlier sections, we have described how to systematically enumerate all possible queries accepted by the database. The generated queries are in the form of token sequences. For example, `SELECT STAR FROM ID WHERE ID EQ ID SEMI`. Each token is a regular expression defined by the specification of SQL grammar. Therefore, each sequence of tokens represents a set of concrete SQL queries instead of only one query. Suppose by enumerating the SQL grammar, we generate several sets of queries in the form of token sequences $\{q_1, q_2, \ldots, q_n\}$. We use $L(q_i)$ to represent the set of concrete queries that q_i contains. In our earlier approach, we only check $L_v \cap L(q_i) \neq \emptyset$ when q_i leads to a complete query. However, we observe that even a prefix of q_i is sufficient to determine that any query with that prefix is not possible to pass the sanitizer. Suppose p is a prefix of q_i. If $L_v \cap L(p.*) == \emptyset$ ($p.*$ denotes p concatenated with any string), then it is clear that $L_v \cap L(q_i) == \emptyset$. In the process of enumerating SQL queries from SQL grammar, we indeed maintain the prefix of SQL queries. Therefore, instead of checking whether an SQL query passes the sanitizer after it is generated, we proactively check whether there is any query that has the same prefix can pass the sanitizer. If all the queries with the same prefix cannot pass the sanitizer, we do not generate any queries with that prefix. In other words, whenever we can be certain that a query is not able to pass the sanitizer, we stop putting any effort to generate it in our query generation process.

As an example, suppose the program contains a sanitizer which blocks the keyword "OR". Instead of generating many queries in such form

```
SELECT STAR FROM ID WHERE ID EQ ID OR ID SEMI
SELECT STAR FROM ID WHERE ID EQ ID OR ID AND ID SEMI
SELECT STAR FROM ID WHERE ID EQ ID OR ID AND ID EQ ID SEMI
...
```

We could quickly conclude that any query with prefix `SELECT STAR FROM ID WHERE ID EQ ID OR` will not pass the sanitizer. Hence, none of the above queries would be generated.

3.4 Evaluation

We have implemented SQLR on the Linux system. We evaluated SQLR on several vulnerable PHP applications that contain custom sanitizing routines. In our evaluation, we used the SQL grammar specification from the SQLlite [44] implementation, which has 141 terminals, 112 non-terminals, 330 rules, and 630 states.

3.4.1 Generating New Attack Patterns

Since our approach uses the SQL grammar as a guide, it can systematically enumerate malicious SQL queries. In our evaluation, it generated lots of surprising attack patterns. Some of the patterns are even not semantically meaningful to a human. However, they resulted in malicious behaviors in the database when we actually tested them in a database server.

For the purpose of generating new attack patterns, we start with two simple symbolic queries, i.e., "`SELECT * FROM user_table WHERE username=`$SymbolicPart$" and "`UPDATE user_table SET password=`$SymbolicPart$", where $SymbolicPart$ denotes the symbolic part of the query. We run our implementation on the symbolic queries to enumerate queries with different parsing trees. A threshold is defined to limit the length of the inputs to be generated. For example, if we want to generate queries with no more than 5 SQL keywords from the

```
SELECT * FROM user_table WHERE username ='' < 2;
SELECT * FROM user_table WHERE username = '' = '';
SELECT * FROM user_table WHERE username =  EXISTS (SELECT distinct * FROM user_table
    WHERE - ~ null);
SELECT * FROM user_table WHERE username = EXISTS (SELECT distinct * FROM user_table
    WHERE not not username);
SELECT * FROM user_table WHERE username= + - EXISTS (SELECT distinct * FROM user_table
     WHERE username);
SELECT * FROM user_table WHERE username='' + -  EXISTS (SELECT distinct username +
    password FROM user_table WHERE user_name + NULL);
SELECT * FROM user_table WHERE username='' < 1 HAVING username < 1;
UPDATE user_table set password='' WHERE username='' & 1;
UPDATE user_table set password='' WHERE username='' / 1;
UPDATE user_table set password='' WHERE username='' < 2;
```

Figure 3.5: Attack patterns generated by SQLR.

Name (version)	Sanitizer Size§	Grammar Size*	Vulnerable Sinks	Vulnerable Sanitizer Detected
Schoolmate (1.5.4)	17	27	5	Yes
PhpFusion (6.00.110)	11	85	2	Yes
PhpBB (2.0.10)	5	306	2	Yes
Webchess (1.0)	4	7	12	Yes
Extcal (2.0)	3	11	7	Yes
Phpsqlitecms (1.0)	3	6	10	Yes
GeCCBBlite (1.0)	4	7	2	Yes

§ Sanitizer size is measured by the number of lines. *Grammar size is measured by the number of production rules

Table 3.2: Evaluation Results.

symbolic part of the symbolic query, we set the threshold to 5.

In Figure 3.5, we show some of the attack queries[1] generated using our approach with a threshold of 5. SQLR generated previously unreported patterns of SQL injection attacks. When the queries are sent to the database server, the conditions in the WHERE clause are always evaluated to true.

3.4.2 Effectiveness in Validating Sanitizers

We evaluated the effectiveness of SQLR in validating sanitizers. We selected 7 real-world open-source web applications from Sourceforge: Schoolmate v1.5.4, PhpFusion v6.00.110, PhpBB v2.0.10, Webchess v1.0, Extcal v2, Phpsqlitecms v1.0, and GeCCBBlite v1.0. Some of these programs have been used for evaluation in existing solutions [12]. In each of these programs, there is custom sanitizing code that is used to detect SQLi inputs. To validate these sanitizers, we used SQLR on the web applications and generated attack SQL queries. We sent the generated malicious queries to the database and manually verified that they did result in malicious activities, i.e., corruption or unintended disclosure of data.

The result of our evaluation is shown in Table 3.2. For each program, the table shows the number of lines in the sanitizer code, the size of the sanitizer's context free grammar generated by PHPSA, the number of sinks that use this sanitizer, and the attacks detected. A sink is

[1]The attack patterns was successfully executed on database running MYSQL.

vulnerable if there is path from input source to the vulnerable point (SQL Query execution) through vulnerable sanitizing code. For example, in `Schoolmate` we have 5 vulnerable sinks because it has 5 different vulnerable points in the program and all the sinks are reached from sanitizing code. Now, we discuss the evaluation of each program in more detail.

`Schoolmate` is a PHP/MySQL solution for high schools to manage assignments and grades of the students. It contains a function `coverttodb` in *DBfunction.php*, which is used to convert dates into database format. This function does not check whether the user-supplied input contains characters other than numbers. This function is used in the files *Manageterms.php, ManagerGrades.php*, which are vulnerable to SQLi attacks. The sanitized input is used in the UPDATE statement which modifies the value in the database. Therefore, we enumerate all possible queries for the UPDATE statement and generate attack patterns.

`PhpFusion` is a light-weight content management system which contains a sanitizing function in file *maincore.php* and *setuser.php*, which checks if the input string contains the keywords "OR" and "=" in user inputs. SQLR generated SQL injection attacks that do not require "OR" and "=" keywords, similar to those in Figure 3.5.

`PhpBB`,a bulletin board contains a vulnerability in the file *viewtopic.php*, which allows arbitrary PHP code to be executed on the server. The sanitizing code involves an improper combination of the `htmlspecialchars` and `urldecode` functions, along with dynamically evaluated PHP code embedded within a call to `str_replace`, which performs a search-and-replace on a string using regular expression matching.

`Extcal` is a multi-user web-based calendar application which contains a vulnerability in `register.php`, `cal_search.php` and 5 more files. The sanitizing function used in this program is the `addslashes` function. It also checks if the length of the input is greater than 3. We use the technique presented in [45] to bypass the `addslashes` function. Attack pattern longer than three can be generated easily using our technique.

`Phpsqlitecms`, a web content management system uses `sqlite_escape_string` function as the sanitizer in 10 files. This function replaces (')(single quote) with ('')(double quote). It can be bypassed using the following attack pattern generated by SQLR.

```
SELECT id, name, pw, type, editor_type FROM user_table where lower(name) = `abc\'' <
    2;--';
```

The trick here is to submit the user input as (abc \'< 2;–) and *sqlite_escape_string* function will change the input into (abc \''< 2;–).

Sanitizing Function	Grammar Size	Time	Number of Queries Generated
w*(')( )(O)(R)	32	251m20s	1028
w*(')( )(H)(A)(V)(I)(N)(G)	64	252m.41s	1033
w*(')( )(O)(R)(D)(E)(R)( )(B)(Y)	73	257m12s	1073
w*(')( )(U)(N)(I)(O)(N)( )(S)(E)(L)(E)(C)(T)	105	236m40s	901

Table 3.3: Performance Evaluation with Threshold 4

Both `Webchess`(a server for playing chess over the Internet) and `GeccBBlite`(a weblog/-portal) use the same sanitizing function that checks if the input string is `empty` or not. If the string is not empty it executes the query in the database. This type of simple sanitizer was bypassed using SQLR.

3.4.3 Performance

We evaluated the performance of SQLR using a computer with dual core 3.2 GHz Pentium processor and 4 GB of memory, running Linux Kernel 2.6.38. Our sample sanitizer checks for SQLi patterns including “UNION SELECT”, “ORDER BY” and, “HAVING”. We evaluated our system performance on validating these sanitizers.

The results are shown in Table 3.3. In the table, each sanitizing function is converted into a context-free grammar by PHPSA and total number of production rules in this grammar is shown in the table. We show the time taken to generate all queries of threshold 4 using the symbolic model of the sanitizer. We also present the number of queries generate using our approach. For example, we have a sanitizing function that checks for ‘HAVING’ keyword in the user input. PHPSA converted this sanitizer into a context-free grammar of 64 production rules. Using this sanitizer model and the threshold 4, we generate 1033 malicious queries in 252 minutes and 41 seconds.

We also measure the time for the evaluation in Table 3.3. After the attack patterns are generated, the time taken to detect vulnerability in the sanitizing ranges from 5-30(minutes) excluding the time for generating attack patterns.

3.5 Case Study with Web Scanners

In this section, we analyze black-box web application vulnerability scanners which are automated tools that probe web applications for security vulnerabilities. In order to assess the newly generated SQL attack patterns using SQLR, we obtained access to three open-source tools: `w3af (1.0)` (Web Application Attack and Audit Framework) is an open-source web application security scanner used to test vulnerabilities in Web applications; `wapiti (2.2.1)` allows to audit the security of web applications and checks for SQLi attacks and; `sqlmap (0.9)` is an

```
1  /* sanitization function using general attack patterns */
2  if((!pre_match("/' or /",$_POST['txtNick']) && !preg_match("/ union /", $_POST['txtNick'])) &&
3     (!preg_match("/ or  /",$_POST['pwdPassword']) && !preg_match("/ union /",$_POST['pwdPassword']))) {
4      /* vulnerable sink */
5      $tmpQuery = "SELECT * FROM players WHERE  nick = '"
                                 .$POST['txtNick'].'" AND password =  '".$_Post['pwdPassword']."'";
6  }
7  else{
8      die("SQL Injection Detected");
9  }
```

(a) Vulnerable sink that cannot be detected by Web scanners

```
1) <username field> = ' + - exists(select distinct nick + password from players where password + NULL);--
2) <username field> = ' < 1 HAVING nick < 1 ; --
3) ' < 2; --
4) ' = ' '; --
5) <username filed> = ' + - exists (select distinct * from players where - ~ null);--
```

(b) Attack patterns that can exploit the above vulnerable sink

Figure 3.6: Analysis on open-source web scanners

Vulnerable sinks	`AcunetixWeb Vulnerability Scanner`	`IBM Rational AppScan`	`HP WebInspect`	`Open Source Tools`	Our Attack Patterns
Without sanitizers	✓	✓	✓	✓	✓
With sanitizers	✗	✗	✗	✗	✓

Table 3.4: Comparison of our attack patterns with the attack patterns of existing Web Scanner tools using `Webchess 0.9`. ✓means SQLi detected, and ✗means SQLi undetected.

open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws. We also tested on three commercial tools: `AcunetixWeb Vulnerability Scanner`, `IBM Rational AppScan` and `HP WebInspect`. Using the above six blackbox penetration tools, we use `webchess (0.9)` PHP application (discussed in previous section) and provide a sanitization function using the general attack patterns. These sanitization functions are meant to provide protection from SQLi attacks. To test the effectiveness of these sanitization functions we use those three tools. In Table 3.4, we list out the how these tools perform with sanitization function using general patterns and also you can see that `webchess` is still vulnerable from SQLi attack with our attack patterns shown in Figure 3.5.

The sanitization function and the list of attack patterns used in this evaluation is described in Figure 3.6. In Figure 3.6(a), the vulnerability sink in Line 5 cannot be detected by the web scanners. However, using the attack patterns shown Figure 3.6(b), the vulnerability sink can be exploited. We also confirmed that the patterns that we are generating are not in leading commercial scanners.

Chapter 4

Weak Authentication Systems (WAS)

In this chapter, we present WeakAutoScan, a tool to systematically find flaws in the authentication protocol implementations. Section 4.1 details the problem with existing blackbox scanners; Section 4.2 gives an overview of our solution; Section 4.3 describes details of the implementation and evaluation of our approach.

In our approach we assume that the Security analyst can only observe the network traffic and code execution at the browser end; the server side logic of the authentication protocol is not available for analysis.

4.1 Problems with existing Scanners

Web application scanners typically consist of two main modules: a crawler module and an attacker module, as shown in Figure 4.1. The crawling component is allowed to fetch a set of URLs, retrieves the corresponding pages, and follows links and redirects to identify all the reachable pages in the application. The attacker module analyzes the URLs discovered by the crawler and the corresponding input points. Then, for each input and for each vulnerability type for which the web application vulnerability scanner tests, the attacker module generates values that are likely to trigger a vulnerability. For example, the attacker module would attempt to inject strings that have a special meaning in the SQL language, when testing for SQL injection vulnerabilities. Input values are usually defined in the attack vectors generated using SQL injection cheat-sheets [46].

The authentication mechanism is the common feature that is found in most web sites. Because this is so prevalent, scanners must properly handle authentication, possibly by creating accounts, logging in with valid credentials, and recognizing actions that log the crawler out.

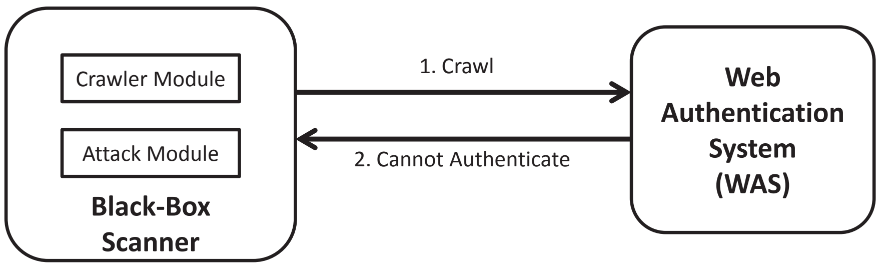


Figure 4.1: Traditional Blackbox Scanner

In Figure 4.1, we show that existing blackbox scanners fail to handle authentication systems. When the scanner tries to crawl the web server which has authentication pages, it fails to fetch because most web applications require a user to create an account in order to utilize the services offered. Hence, the scanners fails to analyze and report for any implementation flaw in the authentication system. In a recent study, it is shown that current blackbox scanners fail to handle authentication [4].

4.2 Overview of our approach

Figure 4.2 illustrates the overview of our approach which consists of three components WeakAuthScan, Web Mail Service (WMS), and Web Authentication System (WAS). WeakAuthScan is the main component which automatically extract the authentication protocols logic by analyzing the HTTP messages exchanged between the scanner and the web server. We use the traditional blackbox scanner approach with additional components to support authentication. WeakAuthScan consists of two modules - a crawler module and an attacker module. The crawler module is similar to those in the traditional blackbox scanner approach (discussed in previous section) and attacker module analyzes the URLs discovered by the crawler and reports if there is an implementation flaw in the authentication protocol.

The WeakAuthScan component first communicates with the WMS component to request for temporary email ID creation and WMS responds back with the “email-ID message. For our scanner to analyze the authentication implementation logic, it has to first create an account on WAS. Creating an account can be done using the “email-ID generated from WMS component. Once the scanner creates an account, WAS responds back with two messages - (a) a successful message to WeakAuthScan for account creation and (b) a mail addressed to the “email-ID (on WMS) and this mail contains a special URL that allows the user/scanner to connect the server and activates the users login session in the server automatically. One advantage using this mechanism is that it skips the need to know the users password to access the user account

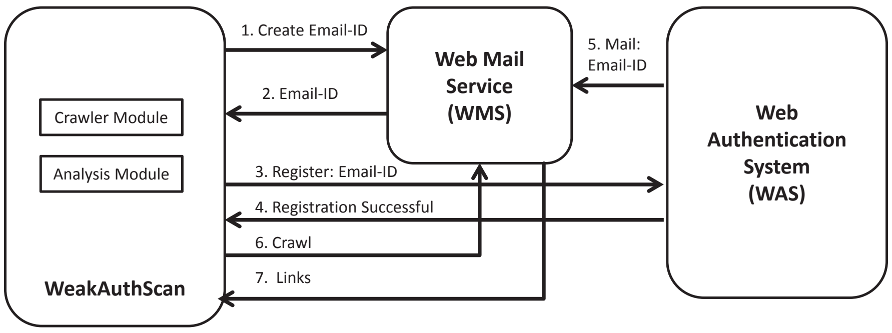


Figure 4.2: Overview of WeakAutoScan.

Such mechanisms are termed as 'AutoAuth' and now-a-days many websites use this method to authenticate user. Once WAS send a mail to the "email-ID we use the crawler module to crawl WMS and use our analysis module to extract the URLs sent from WAS to WMS. These URLs contains HTTP parameters which are used to authenticate a user. We analyze these parameter values either by human or we automatically check whether these parameter values are generated using a sequential algorithm listed in the Online Integer Sequences Encyclopedia [47] or in Wolfram Aplha [48]. Once we find out the logic used to generate the parameter values, WeakAuthScan create a new AutoAuth link and upon successful it produces a vulnerability report.

4.3 Evaluation

We tested on two popular websites, where users share deeply personal information, both of which have from hundreds of thousands to millions of users and utilize weak authorization mechanism. WeakAutoScan successfully uncovered the authentication protocol for both sites.

4.3.1 Online dating sites

Meeting Millionaire is one of the top 10 dating website in the world [49]. The vulnerability exists in one of the link which is sent by the server to unsubscribe from the mailing list. This link contains the information of username, password, contact details of a particular member. For example, consider the following unsubscribe link -

```
http://app.icontact.com/icp/
mmail-mprofile.pl?r=36958596&l=2601&
s=21DS&m=318326&c=752641
```

'$http://app.icontact.com'$ is an email marketing solution used by 'Meeting Millionaire' [49] for maintaining the contact information of its members. Here, in this link the parameters

’c’,’l’,’m’ is all constants for the Meeting Millionaire website. Icontact.com provides services for many such websites and these fields are different for each site. The parameter ’r’ is used to verify which user has sent this request and its value is incremented with arithmetic value. And finally the parameter ’s’ consists of 4 character from the set [A-Z 0-9]. If we want to generate a valid ’s’ field value for a different member then we need to generate total of 36^4 combination and test which one is valid. Using our approach, we are able to successfully generate legitimate URL’s and also capture the username/password for that particular member.

4.3.2 Online Matrimony Site

IyerMatrimony [50] website is a part of CommunityMatrimony which ranks 6096th in Alexa’s most popular websites in India and contains millions of user accounts. The vulnerability in this site can allow attacker to get into any user account without the password. This vulnerability can be exploited by understanding the communication between the server and user. When user registers for a new account using his email address, the server sends a link to the user’s email and on clicking the link will allow user to login into his account. This link contains 5 parameters and among these only 3 fields are required to authenticate the user. To understand how these parameters are generated, we use 10MinutesMail [51] to generate many email accounts and these accounts are used to create new accounts on IyerMatrimony. We write a Perl script to automatically create new accounts and extract those three important parameters associated for each user. We confirm that the parameters generated for consecutive user accounts follows a sequential algorithm like incrementing the parameters value by a constant number or algorithms listed in the Online Integer Sequences Encyclopedia [47].

Chapter 5

Conclusion and Future Work

We studied the vulnerabilities on SQL Injection and Weak Authentication Systems that current black-box scanners fail to detect. In this thesis, we first present SQLR(SQLi Revisited) to systematically validate input sanitizers of SQLi attacks. Second, we present a framework called *WeakAuthScan* a semi-automated method to validate flaws in the authentication protocol implementation.

SQLR is used to systematically validate SQL injection sanitizers in a web application. Our approach is unique in that it uses the SQL grammar to guide a comprehensive validation process. Specifically, our approach traverses the enhanced SQL parsing graph and enumerates malicious SQL queries that can be generated possibly by the web application. The malicious queries are then used by a model of the web application sanitizer to detect vulnerabilities in the sanitizer. Our approach also summaries the enumerated malicious queries into attack patterns that can be used to improve existing sanitizer validators. We evaluated our solution using real-word web applications and also on popular web vulnerability scanners. Our approach generated new attack patterns and successfully detected vulnerabilities in these sanitizers.

WeakAuthScan is an end-to-end scanner to semi-automatically recover authentication protocol logic from its implementation and verify if it contains security flaws. *WeakAuthScan* requires no knowledge of the protocol specifications. We successfully detected security vulnerability on two popular real-world applications.

As shown by our experimental result, black-box scanners are not complete and they fail to detect SQLi and cannot detect flaws in the authentication implementations. In future, we plan to enhance our *WeakAuthScan* method, using white-box testing (analyzing javascript) in the authentication process. We also use verification tools to verify the security properties on the model generated by both white-box and black-box analysis, which makes our approach completly automatic.

Bibliography

[1] "Sql injection attack." https://www.owasp.org/index.php/SQL_Injection, 2012.

[2] "Sql injection attacks up 69%." http://www.zdnet.com/sql-injection-attacks-up-69-7000001742/, 2012.

[3] "Approved scanning vendors. payment card industry security standards council." https://www.pcisecuritystandards.org/pdfs/asv_report.html, 2012.

[4] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Vulnerability Scanner," in *USENIX*, 2012.

[5] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: an analysis of black-box web vulnerability scanners," in *7th international conference on Detection of intrusions and malware, and vulnerability assessment*.

[6] J. Bau, E. Bursztein, D. Gupta, and J. C. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *IEEE S&P*, 2010.

[7] D. Wagner and B. Schneier, "Analysis of the ssl 3.0 protocol," in *2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce*.

[8] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra, "Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps," in *FMSE*, 2008.

[9] R. Wang, S. Chen, and X. Wang, "Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services," in *IEEE S&P*, 2012.

[10] S. Jurag, M. Andreas, S. Jorg, K.Marco, and J.Meiko, "On breaking saml: Be whoever you want to be," in *USENIX*, 2012.

[11] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applica-

tions," in *IEEE S&P*, 2008.

[12] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *ICSE*, 2009.

[13] F. Yu, M. Alkhalaf, and T. Bultan, "Patching vulnerabilities with sanitization synthesis," in *ICSE*, 2011.

[14] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *POPL*, 2006.

[15] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using parse tree validation to prevent sql injection attacks," in *SEM*, 2005.

[16] W. Halfond and A. Orso, "AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks," in *ASE*, 2005.

[17] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in *USENIX Security*, 2006.

[18] "Acunetix web security scanner." `http://www.acunetix.com`, 2012.

[19] "Hp webinspect." `https://download.hpsmartupdate.com/webinspect/`, 2012.

[20] "Ibm security appscan." `http://www.ibm.com/software/awdtools/appscan/`, 2012.

[21] "w3af - web application attack and audit framework." `http://w3af.sourceforge.net/`, 2012.

[22] "Wapiti - web application security auditor." `http:/wapiti.sourceforge.net/`, 2012.

[23] "sqlmap: Automatic sql injection and database takeover tool." `http://sqlmap.org`, 2012.

[24] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and precise sanitizer analysis with bek," in *USENIX Security*, 2011.

[25] W. G. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," in *ISSSE*, 2006.

[26] "Autoauth." `http://docs.whmcs.com/AutoAuth`, 2012.

[27] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities," in *PLAS*, 2006.

[28] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web appli-

cation code by static analysis and runtime protection," in *WWW*, 2004.

[29] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *USENIX Security*, 2006.

[30] Y. Minamide, "Static approximation of dynamically generated web pages," in *WWW*, 2005.

[31] N. Kobayashi, N. Tabuchi, and H. Unno, "Higher-order multi-parameter tree transducers and recursion schemes for program verification," in *POPL*, 2010.

[32] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *PLDI*, 2007.

[33] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst, "Finding bugs in dynamic web applications," in *ISSTA*, 2008.

[34] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, 2005.

[35] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *ISSTA*, 2008.

[36] M. Samuel, P. Saxena, and D. Song, "Context-sensitive autosanitization in web templating languages using type qualifiers," in *CCS*, 2011.

[37] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A systematic analysis of xss sanitization in web application frameworks," in *ESORICS*, 2011.

[38] S. Kals, E. Kirda, C. Krügel, and N. Jovanovic, "Secubat: a web vulnerability scanner," in *WWW*, 2006.

[39] M. Vieira, N. Antunes, and H. Madeira, "Using web security scanners to detect vulnerabilities in web services.," in *DSN*, 2009.

[40] L. Suto, "Analyzing the effectiveness and coverage of web application security scanners," in *Case-Study*, 2007.

[41] L. Suto, "Analyzing the accuracy and time costs of web application security scanners," in *Case-Study*, 2010.

[42] M. Curphey and R. Araujo, "Web application security assessment tools," *IEEE S&P*.

[43] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: A solver for string constraints," in *ISSTA*, 2009.

[44] "Sqlite database engine." `http://www.sqlite.org/`, 2011.

[45] C. Shiflett, "By-pass addslashes." `http://shiflett.org/blog/2006/jan/addslashes-versus-mysql-real-escape-string/`, 2011.

[46] “Sql injection cheat sheet.” http://ha.ckers.org/sqlinjection/, 2012.

[47] “The online encyclopedia of integer sequences.” http://www.oeis.com/, 2012.

[48] “Wolfram—alpha.” http://www.wolframalpha.com/, 2012.

[49] “Meeting millioanaires.” www.meetingmillionaires.com/, 2012.

[50] “Iyer matrimony.” www.iyermatrimony.com, 2012.

[51] “10 minute mail.” http://10minutemail.com/, 2012.