Defeating SQL Injection

Lwin Khin Shar and Hee Beng Kuan Tan
Block S2, School of Electrical and Electronic Engineering,
Nanyang Technological University, Nanyang Avenue, Singapore 639798
{shar0035, ibktan}@ntu.edu.sg

ABSTRACT

Despite the abundance of techniques available for preventing SQL injection issues, recently, vulnerability databases reported new major SQL injection threats in many Web applications. This article reviews the techniques established for addressing this security threat. Our review shows that it can actually be well defended from a combination of some of these techniques. The current problem lies in the integration of these techniques in a practical environment and the developers' familiarities with SQL injection and the use of these techniques.

KEYWORDS

Web application security; SQL injection; exploits and defenses; survey

INTRODUCTION

Recently, security experts from SANS reported a major SQL injection threat that affected approximately 160,000 Web sites using Microsoft IIS, ASP and MSSQL frameworks (isc.sans.org/diary/SQL+Injection+Attack+happening +ATM/12127). The cause was due to the lack of validation on some input parameters in ASP pages. In 2011, National Vulnerability Database (nvd.nist.gov) reported a total of 289 SQL injection issues (7% of all reported vulnerabilities) found in Web sites including IBM, HP, Cisco, WordPress, and Joomla. Open Web Application Security Project (OWASP) ranked SQL injection as the most widespread security issue among Web applications (www.owasp.org/index.php/Top_10).

SQL injection is a class of code injection attacks in which SQL characters or keywords are inserted into an SQL statement via unrestricted user input parameters in order to change the logic of the intended query [1]. This threat exists in any Web application that accesses database via SQL statements constructed with external input data because an attacker could modify these statements by manipulating the data. Therefore, SQL injection is caused by inadequate validation and sanitization of user inputs. An SQL injection attack (SQLIA) could compromise the database and issue arbitrary SQL commands.

Researchers have proposed various techniques ranging from simple static analysis to complex dynamic analysis frameworks to address this problem. In 2006, Halfond, Viegas, and Orso [2] evaluated the then-available techniques and concluded that researchers should continue finding new solutions to improve precision and effectiveness. This review studies these "new" solutions and finds that developers' knowledge and application of current state-of-the-art techniques would finally defeat this serious security issue.

UNDERSTANDING SQL INJECTION

SQL is the standard language for accessing most database servers such as MySQL, Oracle, MSSQL, etc. [1]. Most Web programming languages such as Java, ASP, .NET, and PHP provide a number of different methods to construct and execute SQL statements. Due to circumstances such as lack of development time and training, and lack of experience and knowledge of potential security issues, developers often misuse these methods resulting in SQL injection vulnerabilities (SQLIVs). With some practical examples, the following discusses the insecure programming practices commonly adopted by developers and the injection methods that may be used by attackers to exploit developers' mistakes.

Insecure Coding Practices

Dynamic query building with string concatenation is commonly used by developers to construct SQL statements. Queries are formed with inputs directly received from external sources during runtime. This method is useful to developers as different queries can be built according to different conditions set by users. However, as this method is often the cause of many SQL injection issues, some other developers opt to use parameterized queries or stored procedures which are more secure methods. But, inappropriate use of these methods may still result in vulnerable code. In the following PHP code

Digital Object Indentifier 10.1109/MC.2012.283 0018-9162/$26.00 2012 IEEE

examples, `name` and `pwd` are the "varchar" type columns and `id` is the "integer" type column of `user` database table.

Absence of check: Developers often use inputs in SQL statements without any checks. This is the most common and serious programming mistake. For example, the following PHP code represents such a dynamic SQL statement.

```
$query = "SELECT info FROM user
                WHERE name = '$_GET["name"]' AND pwd = '$_GET["pwd"]'";
```

Attackers could use *tautologies* to exploit this insecure practice. By supplying the value `x' OR '1'='1` (known as *tautology-based attack*) to the input parameter `name`, an attacker could access user information without having a valid account because the WHERE-clause condition becomes:

```
WHERE name = 'x' OR '1'='1' AND …";
```

which shall be evaluated to be true.

Insufficient escaping: If special characters meaningful to a SQL parser are escaped, they shall not be interpreted as SQL commands. For example, the above tautology-based attack could be prevented by escaping the single-quote "'" character (to avoid being interpreted as a string delimiter) from the inputs used. However, many developers are either not aware of the full list of characters that have special meanings to the SQL parser or not familiar with the proper usage patterns. For example, in the following PHP code, `mysql_real_escape_string` (a MySQL-provided-escaping function) is used to escape MySQL special characters.

```
$name = mysql_real_escape_string($_GET["name"]);
$query = "SELECT info FROM user WHERE pwd LIKE '%$pwd%'";
```

The function `mysql_real_escape_string` would protect SQL statements which do not use pattern matching database operators such as LIKE, GRANT, and REVOKE. But, in this case, attackers could include additional wildcard characters "%" and "_" in the password field to match more password characters than the beginning and end characters because `mysql_real_escape_string` does not escape wildcard characters.

Absence of data type check: Another programming mistake made by developers is that data types are rarely checked before SQL statements are constructed. They rather apply programming language or database provided sanitization functions such as `addslashes` and `mysql_real_escape_string` to the input parameters before they are used in SQL statements. But when the query is to access the database columns of non-text-based data types such as numeric, the attack needs not consist of the escaped/sanitized characters. For example, the following PHP code shows a SQL statement for which a tautology–based attack could be conducted by supplying the value `1 OR 1=1` to the parameter `id`.

```
$id = mysql_real_escape_string($_GET["id"]);
$query = "SELECT info FROM user WHERE id = $id";
```

For such queries, instead of escaping, data type check (e.g., `if(is_numeric($id))`) should be used to prevent SQLIAs.

Absence or misuse of delimiters in query string: When a query string is constructed with inputs, proper delimiters have to be used to indicate the data type of the input used. If the delimiters are missed or mis-used, SQL injection could be carried out even in the presence of thorough input validation, escaping, and type checking. For example, in the following PHP code, string delimiters are not used to indicate the input string used in the SQL statement.

```
$name = mysql_real_escape_string($_GET["name"]);
$query = "SELECT info FROM user WHERE name = $name";
```

In this case, when the database server has the automatic type conversion function enabled, SQLIAs could be created using *alternate encoding method* that would circumvent input sanitization routines. For instance, if an attacker supplies an encoded HEX string `0x270x780x270x200x4f0x520x200x310x3d0x31` to the parameter `name`, the database parser may covert it to the "varchar" value resulting in the tautology string "`'x' OR 1=1`". Because the conversion happens in the database, the escaping function used in the server program would not detect any special characters encoded in the HEX string.

Digital Object Indentifier 10.1109/MC.2012.283 0018-9162/$26.00 2012 IEEE

Improper construction of parameterized queries or stored procedures: Most developers believe that SQL injection is impossible when parameterized queries or stored procedures are used to run SQL statements. Although this is true in general, some developers are not aware that SQL injection is still possible if parameterized query strings or stored procedures accept non-parameterized inputs. For example, in the following PHP code, although SQL injection cannot be conducted through the parameter `name`, it is possible through the parameter `order` which is not parameterized. An attacker may inject *piggy-backed query attacks* (malicious queries attached to the original query) such as "`ASC; DROP TABLE user; --`" into the parameter `order`.

```
$query = "SELECT info FROM user WHERE name = ?"."ORDER BY '$_GET["order"]'";
$stmt = $dbo->prepare($query);
$stmt->bindParam(1, $_GET["name"]);
$stmt->execute();
```

SQL INJECTION DEFENSES

SQL injection defense methods can be broadly classified into three types— (1) defensive coding; (2) SQLIV detection; and (3) runtime prevention of SQLIAs. Despite the shortcomings that each individual approach may have, developers could overcome them by adopting a combination of different defense schemes as depicted in Figure 1. Table 1 compares the strengths and weaknesses of various approaches from each category of defense methods.

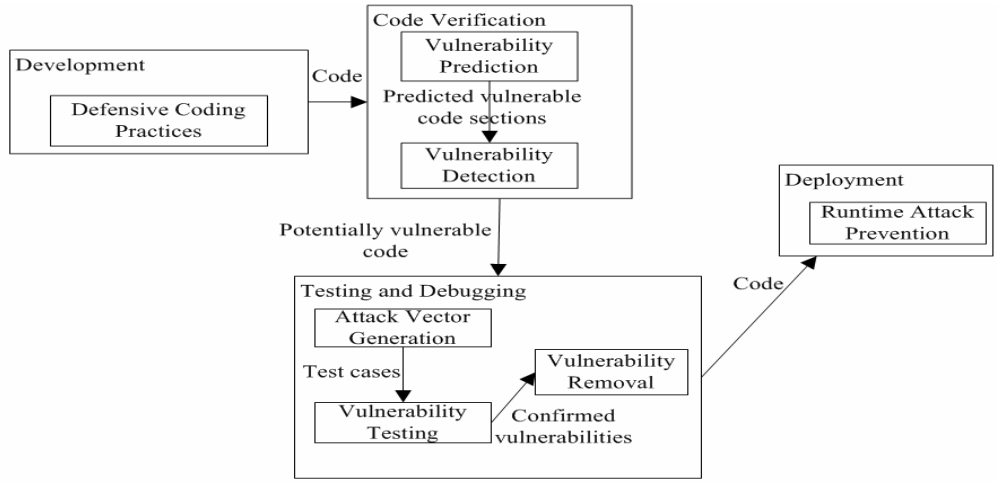

Figure 1. Overview of an ideal use of SQL injection defense schemes during a software development process.

Defensive coding

The use of proper coding practices is a straightforward solution as SQLIV is the direct consequence of insecure coding practices adopted by developers. Many security reports, such as OWASP's SQL injection prevention cheat sheet (owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet) and Chris Anley's white paper [1], have provided adequate coding practices. The following summarizes them.

Parameterized queries or stored procedures: If all developers use or replace their dynamic queries with properly-coded parameterized queries or stored procedures, SQL injection shall be non-existent these days. Proper coding style of parameterized queries or stored procedures enforces developers to define the structure of SQL code first before including parameters to the query. Because parameters are bound to the defined SQL structure, no injection of additional SQL code is possible.

Escaping: If dynamic queries could not be avoided, escaping all user-supplied parameters is the best option. But, as insufficient or improper escaping practices are common, the recommendation for developers is (1) identify all input sources

Digital Object Indentifier 10.1109/MC.2012.283 0018-9162/$26.00 2012 IEEE

to realize the parameters that need escaping; (2) follow database specific escaping procedures; and (3) use standard escaping libraries instead of custom escaping methods.

Data type validation: In addition to escaping, data type validation must be used. Validating whether an input is string or numeric could easily reject type-mismatched inputs. This could also simplify the escaping process because validated numeric inputs need no further cleansing action and could be safely used in queries.

Whitelist filtering: Developers often use *blacklist filtering* to reject known bad special characters such as "'" and ";" from the parameters to avoid SQL injection. However, the safer and recommended filtering approach is to accept only inputs that are known to be legitimate. This approach is suitable for well structured data such as (email) addresses, dates, zip codes, social security numbers, etc. Developers could keep a list of legitimate data patterns and accept only the input data which match them.

Although these recommended practices are the best way to defeat SQL injection, their application is, being manual, labor-intensive, error-prone, and difficult to be rigorous and complete. To alleviate these problems, McClure and Krüger 2005, cited in [3], developed *SQL DOM* – a set of classes that provide automated data type validation and escaping. Developers are required to provide *SQL DOM* with database schema and construct SQL statements using its APIs. This framework is especially useful when a developer needs to use dynamic query construction method (instead of parameterized queries) for greater flexibility.

Thomas, Williams, and Xie [4] proposed an automated *vulnerability removal* approach that generates parameterized queries. They find potentially vulnerable SQL statements (i.e., statements built dynamically) in the programs and replace them with parameterized SQL statements. For example, given the following PHP code:

```
$rs = mysql_query("SELECT info FROM user WHERE id = '$id');
```

Their approach shall replace it with the following code:

```
$dbh=new PDO("mysql:host=xxx;dbname=xxx;","root","pwd");
$PSinput00[] = Array();
$PSquery00 = "SELECT info FROM user WHERE id = ?";
$PSInput00[] = $id;
$stmt = $dbh->prepare($query);
$i = 1;
foreach($PSinput00 as $input){
        $stmt->bindParam($i++, $input);
}
$rs = $stmt->execute();
```

Although these automated approaches are certainly useful, there are some drawbacks. *SQL DOM* could only be used with new software projects and developers are required to learn new query-development process. Thomas *et al.*'s current tool uses pattern matching to deduce SQL structures. So it only works on SQL structures built with explicit strings. Incorporation of program analysis techniques is required to deduce SQL structures built with data objects or through function calls.

Digital Object Indentifier 10.1109/MC.2012.283 0018-9162/$26.00 2012 IEEE

Table 1. Comparison of SQL injection defenses

Type	Approach	User involve-ment	Vulner-ability locating	Verification assistance	Code modi-fication	Generate test suite	Usage stage	Infrastructure
Defensive coding	Defensive coding practice [1]	Very High	No	No	Manual	No	Development	Developer training
	SQL DOM [3]	High						
	Parameter-ized query insertion [4]	Medium	No	No	Automated		Testing and debugging	Tool for code replacement
SQLIV detection	String analyzer [10]	Medium	Automated	Static data flow traces	No		Code verification	Static string analysis tool
	PhpMinerI [11]	Low	Automated	Statistics of sanitization methods implemented	No			Static analysis and data mining tool
	SQLUnitGen [5]	Medium	Automated	Unit test reports	No	Yes	Testing and debugging	Static analysis tool
	MUSIC [5]	Very High	Manual inspection	Test inputs that expose the weaknesses of implemented defense mechanisms	Manual			Manual tests
	Vulnerability and attack injection [6]	Low	Manual inspection		Automated			Injection tool
	SUSHI [7]	Low	Automated	Path conditions that lead to SQLIVs	Automated			Symbolic execution engine
	ARDILLA [8]	Low	Automated	Concrete attacks	Automated			Concolic execution engine
Runtime SQLIA prevention	SQLrand [12]	High	No	No	Manual	No	Deployment	Runtime checker
	AMNESIA [2]	Low	No	Static data flow traces	Automated			Static analysis tool and runtime checker
	SQL-Check [12]	Low	No	No	No			Runtime checker
	WASP [4]	Low	No	No	Automated			Instrumentation tool and runtime checker
	SQLProb [12]	High	No	No	No			Runtime checker
	CANDID [3]	Low	No	No	Automated			Instrumentation tool and runtime checker

Digital Object Indentifier 10.1109/MC.2012.283 0018-9162/$26.00 2012 IEEE

Detection of SQL Injection Vulnerabilities

Code based vulnerability testing: *SQLUnitGen* (Shin, Williams, and Xie 2006 cited in [5]), uses static analysis to track user inputs to database access points and generate unit test cases for these points. Test cases contain SQLIA patterns. *MUSIC* [5] generates mutated queries using nine mutation operators to replace the original queries in the Web programs. Only test cases that contain adequate SQLIAs could kill these mutants. Fonseca, Vieira, and Madeira [6] developed a tool that automatically injects SQLIVs into the Web programs and generates SQLIAs. *MUSIC* and Fonseca *et al.*'s approaches are fault-based techniques. These two approaches are able to assess the effectiveness of the security mechanisms implemented in the applications under test based on the injected vulnerabilities/mutants that are breached.

In general, this type of approaches aims to generate adequate test suite for testing SQLIVs. However, they do not explicitly find vulnerable program points and manual inspection is required to locate them.

Concrete attack generation: This type of approaches uses state-of-the-art symbolic execution techniques for automatic generation of test inputs that actually expose SQLIVs in the programs.

Symbolic execution is a well known technique for automatic generation of test inputs that could exercise various program points. It generates test inputs by solving the constraints imposed on the inputs along the path to be exercised. Traditionally, symbolic execution based approaches use constraint solvers which only handle numeric operations. Because inputs to Web applications are, by default, strings, if a constraint solver is able to solve a myriad of string operations applied to inputs, symbolic execution could be used to both detect the vulnerability of SQL statements which use inputs and generate concrete inputs that attack them. *SUSHI* [7] is one such approach which provides a powerful hybrid constraint solver (numeric and string). The solver is used to solve path conditions that lead to SQL statements, and extract test inputs containing SQLIAs from the solution pool. If one such test input is generated, the corresponding SQL statement is vulnerable. Figure 2 and side bar 1 illustrate this approach.

Although effective and useful, symbolic execution alone is generally not scalable to large programs (due to path explosion). Therefore, researchers proposed various solutions to improve code coverage. *ARDILLA* [8] incorporates concrete execution into symbolic execution. Randomized concrete test inputs are used to exercise program paths of which path conditions could not be symbolically solved by constraint solvers. *SWAT* [9] uses a search based algorithm in which test input adequacy criteria is formulated as fitness functions. Test inputs pooled from the input space are compared using these fitness functions and the best (fittest) test inputs are then used to explore further program paths. To the best of our knowledge, search based and other artificial intelligence algorithms such as genetic algorithms have not been used for the purpose of detecting vulnerabilities. But it is possible to incorporate these recent techniques into symbolic execution based techniques for effective detection of SQLIVs (e.g., eliminate false negatives resulting from code uncovered by symbolic execution).

```
1 $id = addslashes($_COOKIE["id"]);
2 $query = "SELECT info FROM user WHERE ";
3 if($id!=null) {
4    $query .= "id = $id";        //path 1
  } else {
5    $name = addslashes($_GET["name"]);
6    $order = addslashes($_GET["order"]);
7    $query .= "name = '$name' ORDER BY
                  '$order'";      //path 2
  }
8 $rs = mysql_query($query);
```

(a)

Path 1:

1 → $id: $X_{c \to \backslash c}$
Path Condition: true

2 → $id: $X_{c \to \backslash c}$
$query: "SELECT …"
Path Condition: true

3 → $id: $X_{c \to \backslash c}$
$query: "SELECT …"
Path Condition: $X_{c \to \backslash c}$!=null

4 → $id: $X_{c \to \backslash c}$
$query: "SELECT …"."id = $X_{c \to \backslash c}$"
Path Condition: $X_{c \to \backslash c}$!=null

8 → Database access point found,
constraint solver is invoked!

(b)

Figure 2. (a) Sample vulnerable code snippet. The function addslashes() escapes string delimiters. Therefore, SQLIA is not possible through path 2 because a single-quote string delimiter is required to cancel out the delimiters (`'$name'`) used in the query; but it is possible through path 1 because column "id" is an integer type and thus, string delimiter is not required to create an attack. (b) Symbolic execution traces that detected an SQLIV.

Digital Object Indentifier 10.1109/MC.2012.283 0018-9162/$26.00 2012 IEEE

(SideBar 1)
$X_{c \to \backslash c}$ is the symbolic expression of addslashes function (which escapes 4 SQL special characters ', ", \, \0) performed on a cookie parameter "id". When symbolic execution engine reaches the query execution statement at line 8, SUSHI [7] constructs and solves the following constraint which is a conjunction of two string equations, where + represents string concatenation and ≡ separates the left and right hands of the equations:

$$!(X_{c \to \backslash c} \equiv \text{null}) \ \wedge \ (\text{"id = "} + X_{c \to \backslash c} \equiv \text{id = [0-9]* OR 1=1 --})$$

The second equation asks: Is it possible to obtain a solution for X such that the string on the left hand side is matched by the regular expression on the right hand side? If yes, the query structure constructed with the string on the left hand side is vulnerable because the regular expression on the right hand side is a representation of a tautology attack (SUSHI maintains a set of regular expressions that represent different types of attack patterns). In this case, SUSHI clearly has a solution for this constraint, thereby, detecting an SQLIV.

Taint based vulnerability detection: Researchers formulated the problem of SQL injection as one of information flow integrity (Biba 1977 cited in [3]). As such, it can be avoided by restricting tainted data (i.e., user inputs) from affecting un-tainted data (e.g., programmer-defined SQL query structure). Static and dynamic analysis techniques are used to realize this requirement. The latter techniques are utilized by runtime prevention approaches which shall be discussed later.

Livshits and Lam 2005, and Xie and Aiken 2006, both cited in [10], apply prominent static analysis techniques, such as flow-(in)sensitive analysis, context-sensitive analysis, alias analysis, and interprocedural dependency analysis, to (1) identify input sources and data sinks (database access points) and (2) check whether or not every flow from a source to a sink is subject to any input validation and/or input sanitization routine. These approaches typically suffer from precision issues due to one or more of the following limitations: (1) semantics of input validation or sanitization routines are not precisely modeled; (2) input validation using predicates is not considered; (3) vulnerability patterns need to be specified; or (4) user intervention is often required to state the tainted-ness of external or library functions that inputs pass through. All these limitations could result in false negatives or false positives.

Hence, to provide more precision, Wassermann and Su [10] used context free grammars to model the effects of input validation and sanitization routines. Their approach checks whether the string values returned from those routines are syntactically confined in SQL queries. If yes, the routines used are (automatically) concluded as correctly implemented or vice visa. As Wassermann and Su's approach is sound, it would not miss any vulnerability. But, its major weakness is that some of the complex string operations are not precisely handled and thus, assumptions used, which tend to be conservative, may result in false positives.

Data mining based vulnerability prediction: *PhpMinerI* [11] mines static code attributes which represent the characteristics of input sanitization routines implemented in Web programs. The mined attributes and the associated vulnerability information of existing programs are then fed to light weight classification techniques for building vulnerability predictors. For example, if *PhpMinerI* is run on the program in Figure 2a, it shall produce the following vulnerability predictor in the form of a classification tree:

```
sql_sanit < 0 : Vulnerable
sql_sanit ≥ 0
|       dbattr_num < 0 : Not-Vulnerable
|       dbattr_num ≥ 0
|       |       num_check < 0 : Vulnerable
|       |       num_check ≥ 0 : Not-Vulnerable
```

The tree can be interpreted as follows: "a database access point is vulnerable if no database specific sanitization routine is implemented or if there is an access to a database table's numeric column without any numeric type check in the program".

Being a statistic and probabilistic based approach, such an approach does not provide precise analysis of vulnerabilities. However, as static code attributes can be easily collected and powerful data mining tools such as *WEKA* (cs.waikato.ac.nz/ml/weka/) are readily available, such approaches are useful and practical. Developers could save much effort by focusing on the code sections predicted to be vulnerable. Furthermore, current vulnerability prediction approaches

Digital Object Indentifier 10.1109/MC.2012.283 0018-9162/$26.00 2012 IEEE

could still be enhanced with more data mining techniques. For example, graph mining on control flow and data dependency graphs can be incorporated to better discriminate vulnerability signatures and improve the precision in vulnerability localization.

Runtime Prevention of SQL Injection Attacks

Randomization: Boyd and Keromytis presented *SQLrand*, cited in [12], which enforces developers to construct queries using randomized SQL keywords instead of normal keywords. A proxy filter intercepts queries sent to the database and de-randomizes the keywords. An attacker could never inject SQL code without the secret key to randomization.

Learning based prevention: This type of approaches basically uses runtime monitoring systems deployed between application servers and database servers, to intercept all queries and check (SQL keywords) whether or not their syntactic structures are legitimate (i.e., programmer-intended) before they are sent to the database. Legitimate query structures are learnt based on (1) user specification; (2) static analysis; or (3) dynamic analysis.

Specification-based approaches require developers to specify legitimate query structures using formal language expressions such as Extended Backus Naur Form (Kemalis and Tzouramanis 2008 cited in [5] and Huang *et al.* 2004 cited in [12]). *AMNESIA* (Halfond and Orso 2005 cited in [2]) uses static analysis to deduce queries that may legally appear at each database access point in Web programs via isolation of tainted data and un-tainted data. Instead of targeting query statements in a server program like *AMNESIA*, Wei, Muthuprasanna, and Kothari (home.eng.iastate.edu/~muthu/papers/cnf05.pdf) target stored procedures in a database. They use a similar query learning approach as *AMNESIA*.

However, as pointed out by Bisht, Madhusudan, and Venkatakrishnan [3], statically inferred legitimate query structures may not be accurate and attackers could exploit this weakness to conduct SQLIAs. As such, dynamic-analysis-based approaches are later proposed to provide more accuracy. *SQL-Check* (Su and Wassermann 2006 cited in [12]) tracks tainted data at runtime through marking them with meta-characters. When a query is invoked, its legitimate structure is learnt by excluding marked data from the query. Conversely, *WASP* (Halfond, Orso, and Manolios 2008, cited in [4]) tracks un-tainted data motivated by the fact that it is often hard to identify all input sources in practice and some tainted data may be missed. These approaches require low user effort, but as meta-character marking changes the structure of original data, it may cause unpredictable errors on benign inputs.

SQLProb [12] executes the program with various valid inputs to collect all possible queries that may legally appear during runtime. It then uses a global pair-wise alignment algorithm to compare the issued user queries against those from collected legitimate query repository and extract user inputs. An extracted input is then validated (check whether it is part of the syntactic structure of the issued query from which it is extracted) using a SQL parser. Only if user input is syntactically confined, query is sent to the database. This approach requires users to exercise test inputs and assumes that test inputs used are sufficient to exercise all possible queries in the program.

CANDID [3] dynamically mines legitimate query structure at each program path by executing the program with valid and non-attacking inputs and thereafter, it compares the actual issued query with the legitimate query structure mined for the same program execution path. Sidebar 2 illustrates this approach.

Digital Object Indentifier 10.1109/MC.2012.283 0018-9162/$26.00 2012 IEEE

(SideBar 2)
For the program in Figure 1a, if the runtime user input is `id←"1 OR 1=1 --"`, it exercises path 1 and generates a query whose structure is:

```
SELECT ? FROM ? WHERE ?=? OR ?=? --
```

while its corresponding candidate input (a valid input that exercises the same path as the runtime input) `id←"1"` generates a different query structure:

```
SELECT ? FROM ? WHERE ?=?
```

Thereby, *CANDID* [3] detects an SQLIA and prevents the query from being executed. If the runtime user input is `id←null; name="x' OR '1'='1"; order←"ASC"`, it exercises path 2 and generates a query whose structure is:

```
SELECT ? FROM ? WHERE ?=? ORDER BY ?
```

In this case, due to the use of escaping in source code, the attempted SQLIA does not generate a query structure different from the structure generated by its corresponding candidate input `id←null; name="x"; order←"x"`. Therefore, *CANDID* shall not consider it as an attack.

This type of approaches could prevent all SQLIAs because actual runtime values are checked against legitimate queries. Some drawbacks associated with runtime attack prevention approaches are: (1) runtime checks incur performance penalty; (2) some of these approaches require code instrumentation to enable runtime checking, which may introduce further complexity to the debugging of security vulnerabilities.

TOOL SUPPORT

To aid developers and security testers, some of the researchers have made their works or implementations available online.

OWASP provides proper defensive coding rules for avoiding SQLIVs to support developer training (owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet). For retrofitting SQL injection defense mechanisms into existing applications, OWASP also provides *ESAPI* that contains various security APIs (owasp.org/index.php/Category:OWASP_Enterprise_Security_API).

String analyzer (score.is.tsukuba.ac.jp/~minamide/phpsa/) used in Wassermann and Su's work [10] is available and it works on PHP. Fu and Li's symbolic execution based vulnerability detection tool [7] is available and it works on Java (people.hofstra.edu/Xiang_Fu/XiangFu/index.php). It contains two components—*JavaSye* (symbolic execution engine) and *SUSHI* (string constraint solver). As *SUSHI* is an independent solver, it can be used with different symbolic execution engines or used for different programming languages such as PHP. To improve coverage of SQLIVs, developers may also use an independent concolic execution engine called *JavaPathFinder* (babelfish.arc.nasa.gov/trac/jpf) in place of *JavaSye*. *PhpMinerI* [11] works on PHP (sharlwinkhin.com/phpminer.html). It uses static analysis to extract static code attributes from PHP programs. The extracted attributes are a measurement of input validation and sanitization implementations in the programs, which can be used to predict vulnerabilities. Static analysis based runtime protection tool *AMNESIA* (www-bcf.usc.edu/~halfond/amnesia.html) is available and it works on Java. Dynamic analysis based runtime protection tool *WASP* is in commercialization (www-bcf.usc.edu/~halfond/projectlist.html).

There are also useful off-the-shelf vulnerability scanners. These tools may not be perfect (they generally do not detect all the vulnerabilities) but are extremely useful for quick detection of the presence of SQL injection issues in Web sites. *Secubat* (secubat.codeplex.com/) is an open source black-box vulnerability scanner. It uses a Web spider to identify test targets (e.g., Web pages which accept user inputs). It then launches pre-defined attacks against these targets and concludes if an attack was successful by evaluating the server response against attack-specific response criteria (e.g., SQL exceptions raised and program crashes). There are many other open source scanners, such as *Nikto2* (cirt.net/nikto2) and *SQLMap* (sqlmap.sourceforge.net/), which work in a similar fashion as *Secubat*; but they generally require known vulnerability patterns or user intervention to conclude successful attacks. Vieira, Antunes, and Madeira also tested and reported the performance of three popular commercial vulnerability scanners—HP WebInspect (fortify.com/products/web_inspect.html),

Digital Object Indentifier 10.1109/MC.2012.283 0018-9162/$26.00 2012 IEEE

IBM Rational AppScan (ibm.com/software/awdtools/appscan), and Acunetix Web Vulnerability Scanner (acunetix.com/vulnerability-scanner). Their report can be found in (eden.dei.uc.pt/~mvieira).

CONCLUDING REMARKS

Defensive coding practices are labor-intensive but they offer non-vulnerable code. Vulnerability detection approaches could identify all vulnerabilities though they may generate many false alarms. Runtime prevention approaches require dynamic monitoring systems but they could prevent all attacks. An overlapped use of these approaches would certainly provide a complete defense mechanism against SQL injection. The security awareness and the application of current state-of-the-art approaches seem to be the remaining cause for continual occurrence of SQL injection problems. This missing link could be bridged from two different perspectives.

Developers should be provided with adequate training to raise security awareness and get familiar with existing state-of-the-art defense techniques and tools. More importantly, they should be given time and resources to implement security. Often, project managers pay less attention to security and rather focus on functional requirements.

Researchers should implement their proposed approaches and provide their implementations (with comprehensive user manuals) either commercially or as open source. Furthermore, researchers should find simple ways to effectively combine existing defense methods to overcome the limitations of each individual method rather than finding new engineering methods to address SQL injection. Too often, researchers' state-of-the-art techniques are either not available for use or difficult to be adopted. The readily available tools and the easy adoption of combined defense methods would motivate developers to deal with security and finally defeat SQL injection.

Traditionally, SQL injection issue was limited to personal computing environments. But as smart phones, tablets, and other mobile devices are increasingly used, this issue has now extended to mobile and cloud computing environments where vulnerabilities could spread much faster and exploitations become much easier. Hence, beside the above issues, security researchers may also have to deal with additional problems arisen from greater flexibility and mobility offered by mobile and cloud computing platforms, and newer programming languages (such as HTML5).

Reference:

[1] C. Anley, "Advanced SQL injection in SQL server applications," *Next Generation Security Software Ltd.*, White Paper, 2002.

[2] W.G.J. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," In *Proceedings of the International Symposium on Secure Software Engineering* (ISSSE 06), 2006.

[3] Prithvi Bisht, P. Madhusudan, and V.N. Venkatakrishnan, "CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks," *ACM Transactions on Information and System Security*, 13 (2), February 2010, pp. 1-39.

[4] S. Thomas, L. Williams, and T. Xie, "On Automated Prepared Statement Generation to Remove SQL Injection Vulnerabilities," *Information and Software Technology*, 51 (3), March 2009, pp. 589-598.

[5] H. Shahriar, M. Zulkernine, "MUSIC: Mutation-Based SQL Injection Vulnerability Checking," In *Proceedings of the 8th International Conference on Quality Software* (QSIC 08), 2008, pp. 77-86.

[6] J. Fonseca, M. Vieira, and H. Madeira, "Vulnerability & Attack Injection for Web Applications," In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN 09), 2009, pp. 93-102.

[7] X. Fu and C.C. Li, "A String Constraint Solver for Detecting Web Application Vulnerability," In *Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering* (SEKE 10), 2010, pp. 535–542.

[8] A. Kieżun, P.J. Guo, K. Jayaraman, and M.D. Ernst, "Automatic Creation of SQL Injection and Cross-Site Scripting Attacks," In *Proceedings of the 31st International Conference on Software Engineering* (ICSE 09), 2009, pp. 199-209.

[9] N. Alshahwan and M. Harman, "Automated Web Application Testing Using Search Based Software Engineering," In *Proceedings of the 26th International Conference on Automated Software Engineering* (ASE 11), 2011, pp. 3-12.

[10] G. Wassermann and Z. Su, "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities," In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 07), 2007, pp. 32-41.

[11] L.K. Shar and H.B.K. Tan, "Mining Input Sanitization Patterns for Predicting SQL Injection and Cross Site Scripting Vulnerabilities," to appear in *34th International Conference on Software Engineering* (ICSE 12), 2012.

[12] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou, "SQLProb: A Proxy-Based Architecture towards Preventing SQL Injection Attacks," In *Proceedings of the 24th ACM Symposium on Applied Computing* (SAC 09), 2009, pp. 2054-2061.

Digital Object Indentifier 10.1109/MC.2012.283 0018-9162/$26.00 2012 IEEE

Biography

Lwin Khin Shar is a research student in the school of Electrical and Electronic Engineering, Nanyang Technological University. His research interests include software security and web security. Shar received a BE in electrical and electronic engineering from Nanyang Technological University. He is a member of IEEE. Contact him at shar0035@ntu.edu.sg.

Hee Beng Kuan Tan is an associate professor of information engineering in the School of Electrical and Electronic Engineering, Nanyang Technological University. His research focuses on software security, analysis, and testing. Tan received a PhD in computer science from the National University of Singapore. He is a senior member of IEEE and a member of ACM. Contact him at ibktan@ntu.edu.sg.

Contact Information

Lwin Khin Shar (Coordinating author)
INFINITUS@EEE, InfoComm Centre of Excellence
School of Electrical and Electronic Engineering,
Block S2, Nanyang Technological University, Nanyang Avenue,
Singapore 639798
Ph: +65 97423763
shar0035@ntu.edu.sg

Hee Beng Kuan Tan
School of Electrical and Electronic Engineering,
Block S2, Nanyang Technological University, Nanyang Avenue,
Singapore 639798
Ph: +65 67905631
ibktan@ntu.edu.sg

Digital Object Indentifier 10.1109/MC.2012.283 0018-9162/$26.00 2012 IEEE