The ART of SQL Injection Vulnerabilities Discovery *

Chenghong Wang
Harbin Engineering University
Harbin, China
wangchenghong@hrbeu.edu.cn

Donghong Zhang
Harbin Engineering University
Harbin, China
zhangdonghong@hrbeu.edu.cn

Jing Zhao†
Harbin Engineering University
Harbin, China
zhaoj@hrbeu.edu.cn

Jingyu Li
Harbin Engineering University
Harbin, China
lijingyu@hrbeu.edu.cn

Hualin Lu
Harbin Engineering University
Harbin, China
luhualin@hrbeu.edu.cn

Long Zhang
Institute of Software
Chinese Academy of Sciences
Beijing 100190
zlong@ios.ac.cn

Zhenyu Zhang
Institute of Software
Chinese Academy of Sciences
Beijing 100190
zhangzy@ios.ac.cn

Jian Zhang
Institute of Software
Chinese Academy of Sciences
Beijing 100190
zj@ios.ac.cn

ABSTRACT

SQL injection(SQLi) is one of the top risky vulnerabilities for web applications which may lead to significant security consequences such as authentication bypassing, privacy leakage, etc.. Therefore, discovering SQLi vulnerabilities is an important step for ensuring its quality. Effective testing techniques can ensure that applications are free from SQLi vulnerabilities before its been released and reduce related cost of performing manual analysis, monitoring or post deployment of other defensive mechanisms. However, due to the variety of SQLi attack types the generated test cases for discovering SQLi are scable and complex. Moreover, successful test cases which could discovery SQLi vulnerabilities consist only a small part of the whole test case space. Therefore, testing the SQLi vulnerabilities is a time consuming and inefficient task. Towards this problem we propose an effective SQLi testing approch based on Adaptive Random Testing(ART) method. We define the concept of test case distance and then present the computing method for test case distance by using TF-IDF algorithm. Then we apply the Fixed Size Candidate Set(FSCS) algorithm in SQLi test case selection with the help of test case distance metrics. We also conduct 6 experimental studies on our testing algorithm, where we tested the SQLi vulnerabilities in 4 open source benchmarks and 2 CMS applications. The experimental results indicate that our testing algorithm can reduce the redundant test cases by more than 50% and significantly increases the testing effectiveness of SQLi vulnerabilities discovering techniques.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Security, Algorithms

Keywords

SQL injection; adaptive random test; test case distance metrics; selection algorithm

1. INTRODUCTION

Nowadays, database-driven web applications have been rapidly adopted in a wide range of areas including online stores, e-commerce, social network services, etc. However such popularity makes them more attractive to attackers. The number of reported web attacks is growing sharply [10]. A recent web application attack report show us that, over a period of nine months, from August 1, 2013 to April 30, 2014 reveals an average increase of around 17% in different types of web attacks. In addition, the report also found that web attacks have became more sophisticated and gotten dramatically longer in length.(44% longer than they were in previous reports). Web applications are suffering more than 26 attacks in one minute [9]. Other security report indicate that at least 8% of the web services which belong to some technology companies like Microsoft and Google contains different types of security vulnerabilities [26].

Within the Web based vulnerabilities, SQLi vulnerabilities has been labeled as top risky vulnerabilities by Open

*This research was supported by the Open Fund of the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences (Grant No. SYSKF1405).

†All correspondence should be forwarded to Dr. Jing Zhao

WOODSTOCK '97 El Paso, Texas USA

Web Application Security[1]. SQLi refers to a class of code-injection attacks in which data provided by the user is included in a SQL query in such a way that part of the user's input is treated as SQL code [13]. This type of vulnerability is ultimately caused by insufficient validation of user-input [12–14]. By exploiting such vulnerabilities, attackers can submit malicious requests which contains sophisticated SQL queries to remote database-driven web applications. As a result, attackers will capture the privilege of accessing the underlying database arbitrarily. Attackers can even take control of and corrupt the system that hosts the Web applications by elaborately designing the injected SQL codes. During the past few years, SQLi vulnerabilities have already became the top 1 threat for database-driven web applications, and SQL injection attacks have also become the most popular web attack.

Although the cause for SQLi is quite simple and the mechanisms of SQLi are well-understood, the SQLi vulnerabilities still increase. They persist because of a lack of effective techniques for discovering and preventing them. Previous approaches to detecting SQLi vulnerabilities and preventing exploits include static analysis [11, 19, 23], user input filtering [17], defensive programming [7], runtime monitoring [3,15,20,24], combining static analysis and runtime monitoring [13], source code fixing [25], application level intrusion detecting [18], SQL key words randomization [2], testing techniques [1, 16, 22], etc.. Among these previous researches, it is well accepted that security testing method is the most effective way towards discovering SQLi vulnerabilities. Security testing against SQLi vulnerabilities could significantly reduce the related costs of static analysis, monitoring or deploy application level defensive mechanisms, which are acknowledged time consuming and inefficient tasks. On the other hand, security test could guarantee the web applications free from SQLi vulnerabilities before they go into services. It is much easier to repair the applications before the applications' service lifecycle than during such period. However, very few works address the issues of testing techniques towards SQLi vulnerabilities, as SQLi testing suffers from three well-known problems. The problems for SQLi testing could be conclude as follows:

a) *Scalability of Test Case Space*

The difficulty for researchers to address the full scope of the SQLi vulnerabilities is that there are many types of SQLI attacks and countless variations on these basic types. As a result, the test case space should be designed as large as possible. Otherwise some effective attack vectors which could reveal the potential SQLi vulnerabilities of testing applications may be missed.(i.e. Functions and keywords filtering prevents web applications from being attacked by using a functions and keywords black list. If an attackers submits an injection code containing a keyword or SQL function in the black list, the injection will be unsuccessful. Although the technique is very strict and perform in real world, applications with such security mechanism may also vulnerable to SQLi attacks. Attacks can evade inspection by using case changing, character encoding or inline comment methods.) Therefore, it is important to design enough test cases for testing. Equally the scale of test case space increase to a great level.

[1]http://www.programmableweb.com,

b) *Sparse Distribution of Effective Test Cases*

Though the test case space is very large, the effective test cases which could flag SQLi vulnerabilities distributed sparsely. In our previous works, we found that the effective test cases for SQLi discovery consist only 2% of the total test case pool. Such small proportion makes the searching task for effective test cases more difficult than normal testing tasks. And the performing of rest 98% can be regard as redundant works which may limit the testing effectiveness.

c) *Large Expense for Executing SQLi Test Cases*

The performing costs for SQLi vulnerabilities discovery are also very high. The judgement of test cases execution results could be a complex work. Some researchers have proposed some automatic mechanism for testing result checking and judging [1, 8, 21]. However, in some cases, it also requires some professional testers to complete the checking and judging tasks manually. i.e. In some web applications the injectable parameters appear in some particular sections like HTTP Head Referer (CVE-2011-3340[2], CVE-2008-0850[3], CVE-2007-1061[4]) section. In this cases, the most accurate and effective way to check or judge the testing is to check by testing engineers. Testing engineer have to check every test cases results during the testing period.

According to all the aforementioned problems of SQLi testing techniques, we know that SQLi testing is indeed a timing consuming and complex task compared to normal software testing. Moreover the high level of testing redundancy for SQLi vulnerabilities discovering even amplify the testing ineffectiveness and complexity. How to reduce the verbosity of test cases have became an important problem for improving SQLi testing effectiveness. To address this problem, we propose a test case selection algorithm for SQLi vulnerabilities discovering in this paper based on Adaptive Random Testing(ART) method. Through various empirical observations showing that many successful test cases are very similar between each others (i.e. Some are encoded by the same encoding method or attached with similar prefix.). More specifically, the successful test cases which could discovery SQLi have a high degree of similarity. As a result successful test cases distributed together to form clusters in total test cases domain.

Due to this distribution of successful test cases, ART is supposed to be the best choice to improve the effectiveness of test case selection tasks. In our work, the approach started from test cases distance metrics quantification, in this part we apply the TF-IDF algorithm to compute the test cases distance due to the previous observation consequences of the successful SQLi test cases distribution. Then we apply a well performed Adaptive Random Testing method called Fixed Size Candidate Set(FSCS) method to select test cases with the help of test cases distance metrics. Through FSCS algorithm, the test case selected to execute should be the most distinct (the farthest) one compared to the previous

[2]http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3340

[3]http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0850

[4]http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1061

one. It is because that regions of the unsuccessful test cases domain will also be contiguous. Thus, given a set of previously executed test cases that have not revealed any failures, new test cases located away from these old ones are more likely to reveal failures.

We have also evaluated our test cases selection algorithm by comparing the discovering effectiveness of some reported vulnerabilities(reported on National Vulnerability Database) between testing approach with and without FSCS-ART algorithm. The statistic used to compare the methods was the average number of tests required to detect the first successful test case, which is commonly known as the F-measure [6]. In most cases, the F-measure of FSCS-ART was 30-50% lower than that of testing approach without ART selection algorithm. Experimental result indicate that our selection can significantly reduce the SQLi testing redundancy and increase the SQLi vulnerabilities discovering effectiveness.

The main contributions of this work are:

- The presentation of a new technique to test SQLi vulnerabilities that based on adaptive random method.
- The observation and analysis of effective test case distribution for dectecting SQLi vulnerabilities which shows that successful test case tend to be clustered together.
- The presentation of test case distance metrics based on string similarity metrics.
- An experimental evaluation of the technique that shows the effectiveness and the efficiency of this technique.

The remainder of this paper is organised as follows: Section 2 and Section 3 provides a background on SQLi vulnerabilities and reviews related work. Section 4 is the introduction of Adaptive Random Testing and discussion of successful SQLi test case distribution. Section 5 presents our proposed security testing approach. Section 6 presents the evaluation together with a discussion of results and threats to validity. Finally, Section 7 concludes the work.

2. BACKGROUND OF SQLI

2.1 Injection Mechanisms

2.2 Example of SQL injections

3. RELATED WORKS

3.1 SQLi Prevention

3.2 Adaptive Random Testing

Adaptive Random Testing (ART) is an effective improvement of Random Testing (RT). It is based on the observation effective test cases for testing target program tend to be clustered together. ART, therefore, proposes to have randomly selected test cases being more evenly spread throughout the test case domain by employing the location information of the successful test cases. [4]

4. TEST CASE DISTRIBUTIONS

Essentially, the testing process can be viewed as taking samples from the set of all SQLi attack payloads(known as the test case space) to the web application under test, executing the samples one by one, and determining whether the malicious inouts been filtered by input validation mechanisms of testing application. If the inputs get through, a payload bypassing is revealed. The presence of a bypassing implies the existence of a SQLi vulnerability. A tester seeks to select test data with a view to minimising the cost for detecting SQLi vulnerabilities. To help the tester in this task, it is natural to firstly consider how effective test cases distribute in total test cases space.

Table 1: Successful Test Cases for CVE2010-0415

No.	Sucessful Test Case
1	; or 1 = 1 waitfor delay '0: 0: TIME –
2	); or 1 = 1 waitfor delay '0: 0: TIME –
3	'; or 1 = 1 waitfor delay '0: 0: TIME –
4	"; or 1 = 1 waitfor delay '0: 0: TIME –
5	'); or 1 = 1 waitfor delay '0: 0: TIME –
6	"); or 1 = 1 waitfor delay '0: 0: TIME –
7	)); or 1 = 1 waitfor delay '0: 0: TIME –
8	')); or 1 = 1 waitfor delay '0: 0: TIME –
...	...

To present our work clearly, let us first define a few terms. Suppose $T = \{t_1, t_2, t_3, \ldots, t_n\}$ is a SQLi test case space for some specific web application program (denoted by P) with n test cases. For each test case t_i in T is a type of specific string which could be injeted into web applications through input functions and flag SQLi attacks, we also call these strings SQLi attack payloads. Within test case space T we define $S = \{s_1, s_2, s_3, \ldots, s_k\}$ to be the effective test cases set with total k test cases, where every test case s_i in S can successfully flag SQLi vulnerabilities in application P. On the other hand $F = \{f_1, f_2, f_3, \ldots, f_d\}$ are defined as ineffective test cases set or redundant test cases set such that $F, S \subseteq T$ and $F \cap S = \varnothing$. The size number of total test case space, effective test case space and redundant test case space are denoted by n, k, d respectively, where $n = k + d$.

In our recent works, we have tested 29 reported SQLi vulnerabilities (Reported on NVD) by using the test cases (SQLi attack payloads) from real world. After that we have two observations from our testing. First, the test cases space collected for dectecting SQLi vulnerabilities is very huge, in other word the n is very large. However, the effective test cases consist only a small part of the test cases space, the successful ratio k/n is always below 2 percent. Second, effective SQLi test cases which could detect some specific SQLi vulnerability are always have the similar string format or constructure.

Let's take the CVE-2010-0425 (a reported SQLi vulnerability) as an example. Table I is the effective test cases we collected in our testing, it shows that the effective test cases for detecting CVE-2010-0425 are all similar to each others. All successful test cases are tautology attack payloads and the injection type are all the time based blind SQL injection mode (they contain the same terms like waitfor, delay, which are the key words for time based blind SQLi). To improve that the effective test cases share high similarity between each others, we select one effective test case s_j and then compute the string similarity between s_j and other test cases used for dectecting CVE-2010-0425.

From the details we observed, we can conclude that, effec-

tive SQLi test cases for testing target tend to be clustered together within whole test case domain, in other word if s_j is an effective test cases twords some specific SQLi vulnerability, the test cases close to s_j (similar with s_j in string format) will more likely to reveal such vulnerability.

5. THE ART OF SQLI DISCOVERY

Clustering distribution of effective test cases for detecting SQLi are indeed common. Therefore, given a set of previously executed test cases that have not revealed SQLi vulnerabilities, new test cases located away from(string format distinct from) these old ones are more likely to reveal vulnerabilities-in other words, test cases should be more evenly spread throughout the total test case space. According to this intuition, Adaptive Random Testing (ART) method was applied to improve the effectiveness of SQLi discovering technique.

5.1 Test Case Similarity Metrics

Adaptive random testing require that in each round of testing when we generate a new test case, we need to make sure that the new test case should not be too close to any of the previously generated ones. Therefore, it is necessary to define the distance metrics first. Because we observe such clustering phenomenon from the perspective of string structure. Thus, we define test case distance based on string similarity, in this paper we apply TFC method and Cosine Similarity in defining test case distance metrics:

Definition 1 [Test Case Term Space] Each SQLi test case is a type of string, the elements consist of such string are called test cases terms. Suppose T is the test case space for application P. Then, $TS = \{tm_1, tm_2, ..., tm_t\}$ is the test case term space of T, denote the set of all distinct terms appear in whole test case space. The element terms in SQLi test cases area not the same terms in documents, they are more complicated. Table 2 provides a summary of the terms in SQLi test cases.

Definition 2 [Test Case Feature Vector] Suppose $T = \{t_1, t_2, t_3, ..., t_n\}$ is the test case space of testing application P, and $TS = \{tm_1, tm_2, tm_3, ..., tm_k\}$ is the term space, denote k distinct terms in total test case space. And we use following method to vectorize each test case t_j:

The vector of t_j is denoted as $\mathbf{t_j} = [w_{1,j}, w_{2,j}, w_{3,j}..., w_{k,j}]$. Each dimension is called the (term) weight corresponds to a separate term in TS, where

$$w_{i,j} = \frac{\log(TF_{i,j} + 1.0) \times \log \frac{|T|}{|\{t' : tm_i \in t'\}|}}{\sqrt{\sum_{i=1}^{k} [\log(TF_{i,j} + 1.0) \times \log \frac{|T|}{|\{t' : tm_i \in t'\}|}]^2}} \quad (1)$$

- $TF_{i,j}$ is the term frequency of term tm_i in test case t_j.
- $|T|$ is total number of test cases , and $|\{t^{'} : tm_i \in t^{'}\}|$, is number of test cases containing term tm_i.

Usually, $\mathbf{t_j}$ is a column vector with high level dimensions, thus such feature vector will bring lot of costs in computing procedures. Here we apply Principal Component Analysis (PCA) method to reduce $\mathbf{t_j}$.

Suppose $\mathbf{T}$ is the vectorized representation of test case space. The test case space matrix $\mathbf{T}$ can be described as below:

$$\mathbf{T} = (\mathbf{t_1}, \mathbf{t_2}, \mathbf{t_3}, ..., \mathbf{t_n})$$

$$\mathbf{T} = \begin{pmatrix} w_{11} & w_{12} & w_{13} & \cdots & w_{1n} \\ w_{21} & w_{22} & w_{23} & \cdots & w_{2n} \\ w_{31} & w_{32} & w_{33} & \cdots & w_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & w_{m3} & ... & w_{mn} \end{pmatrix}$$

where each colum of matrix $\mathbf{T}$ corresponds to the vector of each test case Before we apply principle component analysis method, we need to firstly standardized the weight data in matrix T, here we use $z - score$[**] to standardize the term weight data. Suppose $\mathbf{T}'$ is the standardized matrix, $\mathbf{T}'$ can be represented as follow:

$$\mathbf{T}' = \begin{pmatrix} w'_{11} & w'_{12} & w'_{13} & \cdots & w'_{1n} \\ w'_{21} & w'_{22} & w'_{23} & \cdots & w'_{2n} \\ w'_{31} & w'_{32} & w'_{33} & \cdots & w'_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w'_{m1} & w'_{m2} & w'_{m3} & ... & w'_{mn} \end{pmatrix}$$

where the standardized weight w'_{ij} in matrix $\mathbf{T}'$ can be computed by the following equations:

$$w'_{ij} = \frac{w_{ij} - \bar{w}_i}{s_i} \quad (2)$$

$$\bar{w}_j = \frac{\sum_{j=1}^{n} w_{ij}}{n}, \; s_j^2 = \frac{\sum_{j=1}^{n} (w_{ij} - \bar{x}_i)^2}{n-1} \quad (3)$$

$$\frac{1}{m} \sum_{j=1}^{n} (\mathbf{t_j}^{\mathrm{T}} e_i)^2 = \frac{1}{m} \sum_{j=1}^{n} e_i^{\mathrm{T}} \mathbf{t_j} \mathbf{t_j}^{\mathrm{T}} e_i \quad (4)$$

$$= \frac{1}{m} e_i^{\mathrm{T}} \left(\sum_{j=1}^{n} \mathbf{t_j} \mathbf{t_j}^{\mathrm{T}} \right) e_i \quad (5)$$

$$= e_i^{\mathrm{T}} \Sigma e_i \quad (6)$$

Definition 3 [Test Case Distance Metrics] Suppose two test case t_i and t_q, their feature vectors are $\mathbf{t_j} = [w_{1,j}, w_{2,j}, w_{3,j}, ..., w_{k,j}]^{\mathrm{T}}$, $\mathbf{t_q} = [w_{1,q}, w_{2,q}, w_{3,q}, ..., w_{k,q}]^{\mathrm{T}}$. We define the distance between test case t_i and t_q by using Cosine Distance(or Cosine Similarity) measurement. Such distance is represented as follows:

$$cosine(t_j, t_q) = \frac{\mathbf{t_j} \cdot \mathbf{t_q}}{||\mathbf{t_j}|| \; ||\mathbf{t_q}||} = \frac{\sum_{i=1}^{k} w_{i,j} \cdot w_{i,q}}{\sqrt{\sum_{i=1}^{k} w_{i,j}^2} \cdot \sqrt{\sum_{i=1}^{k} w_{i,q}^2}} \quad (7)$$

$$dis(t_j, t_q) = \frac{1}{cosine(t_j, t_q)} \quad (8)$$

In definition 2 , it is obviously that $\forall \; t_i \; in \; T$, the (term) weight in feature vector $\mathbf{d_i}$ are all belongs to $\mathbb{R}$. Therefore, the outcome of Equation (2), test cases distance (similarity), is neatly bounded in $[\, 0, 1 \,]$. The resulting distance (similarity) ranges from 0 indicating orthogonality (decorrelation), to 1 meaning exactly the same, and in-between values indicating intermediate similarity.

5.2 Candidate Test Case Selection Algorithm

In this paper, we use the first ART method proposed, Fixed Size Candidate Set (FSCS)[**] algorithm to select candidate test cases. In our process, the selection algorithm makes use of two sets of test cases, named executed set and candidate set which. Suppose $E = \{e_1, e_2, e_3, ..., e_f\}$ is the executed set, and $C = \{c_1, c_2, c_3, ..., c_\kappa\}$ be the candidate set such that $E \cap C = \emptyset$.

E is the set of distinct test cases that have already been selected and executed but without revealing any SQLi vulnerability while the C is a set of test cases that are randomly selected from T. The executed set is initially empty and the first test case is randomly chosen from the test case space. The executed set is then incrementally updated with the selected element c_h, ($c_h \in C$) until a SQLi vulnerability is revealed. The selected element c_h should satisfy the following condition: For all $j \in \{1, 2, 3, ..., \kappa\}$,

$$\min_{i=1}^{f} dis(c_h, e_i) \geq \min_{i=1}^{f} dis(c_j, e_i)$$

where dis is defined in Definition 3. This selection criterion of FSCS-ART is referred to as $S_{distance}$ in this paper. Specific algorithm of $S_{distance}$ is described in Algorithm 1.

Algorithm 1 $S_{distance}$ Selection Criterion

Input:
The Executed Test Case Set, E;
Candidate Test Case Set, C.
Output:
Candidate Test Case, *candidate*.

1: $BestDis = 0$
2: **for** each candidate c_j *in* C **do**
3: compute d_j
where $d_j = \min_{i=1}^{f} dis(c_j, e_i)$
4: **if** $d_j \geq D$ **then**
5: $BestDis = d_j$
6: $candidate = c_j$
7: **end if**
8: **end for**
9: **return** *candidate*;

Chen et al. [4] have observed that the effectiveness of FSCS-ART can be significantly improved by increasing κ when $\kappa \geq 10$.And he also suggested that $\kappa = 10$ is close to the optimal setting of FSCS-ART. Hence, they have used $\kappa = 10$ in their investigation. [5] In our $S_{distance}$ algorithm, we also use the suggested value of κ, which is 10.

Algorithm 2 Fixed Size Candidate Selection Algorithm

Input:
The set of test case space for application P, T;
The Size of Candidate Set, κ.

1: $E = \{\}$
2: randomly select a test case $t_{initial}$ from T
3: Add $t_{initial}$ in to E
4: Testing application P by using $t_{initial}$
5: **if** *Any SQLi vulnerability is revealed* **then**
6: STATUS = **true**
7: **else**
8: STATUS = **false**
9: **end if**
10: **while not** STATUS **do**
11: $C = \{\}$
12: Randomly select κ test cases $c_1, c_2, c_3, ..., c_\kappa$ from T
13: Add $c_1, c_2, c_3, ..., c_\kappa$ into C
14: Executing $S_{distance}$ to select candidate test case c_h.
15: Testing application P by using c_h
16: **if** *Any SQLi vulnerability is revealed* **then**
17: STATUS = **true**
18: **else**
19: $E = E + \{c_h\}$
20: **end if**
21: **end while**

6. EXPERIMENTS

6.1 Experiment Setup

6.2 Scenarios

6.3 Result & Analysis

7. CONLUSION

8. REFERENCES

[1] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan. Automated testing for sql injection vulnerabilities: an input mutation approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 259–269. ACM, 2014.

[2] S. W. Boyd and A. D. Keromytis. Sqlrand: Preventing sql injection attacks. In *Applied Cryptography and Network Security*, pages 292–302. Springer, 2004.

[3] G. Buehrer, B. W. Weide, and P. A. Sivilotti. Using parse tree validation to prevent sql injection attacks. In *Proceedings of the 5th international workshop on Software engineering and middleware*, pages 106–113. ACM, 2005.

[4] T. Y. Chen, D. H. Huang, and F.-C. Kuo. Adaptive random testing by balancing. pages 2–9, 2007.

[5] T. Y. Chen, F.-C. Kuo, and H. Liu. Adaptive random testing based on distribution metrics. *Journal of Systems and Software*, 82(9):1419–1433, 2009.

[6] T. Y. Chen, H. Leung, and I. Mak. Adaptive random testing. In *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*, pages 320–329. Springer, 2005.

[7] W. R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 97–106. IEEE, 2005.

[8] B. Damele and M. Stampar. Sqlmap. *Online at http://sqlmap. org*, 2012.

[9] T. B. et. al. Web application attack report. 2015.

[10] M. Fossi and E. Johnson. Symantec global internet security threat report. *White Paper, Symantec Enterprise Security*, 1, 2009.

[11] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting sql injection vulnerabilities. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 87–96. IEEE, 2007.

[12] W. Halfond, J. Viegas, and A. Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, pages 65–81. IEEE, 2006.

[13] W. G. Halfond and A. Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183. ACM, 2005.

[14] W. G. Halfond and A. Orso. Detection and prevention of sql injection attacks. In *Malware Detection*, pages 85–109. Springer, 2007.

[15] W. G. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185. ACM, 2006.

[16] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 199–209. IEEE, 2009.

[17] D. LeBlanc and M. Howard. *Writing secure code*. Pearson Education, 2002.

[18] J.-C. Lin and J.-M. Chen. The automatic defense mechanism for malicious injection attack. In *Computer and Information Technology, 2007. CIT 2007. 7th IEEE International Conference on*, pages 709–714. IEEE, 2007.

[19] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Usenix Security*, pages 18–18, 2005.

[20] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, pages 124–145. Springer, 2006.

[21] A. Riancho. w3af-web application attack and audit framework. *World Wide Web electronic publication*, page 21, 2011.

[22] H. Shahriar and M. Zulkernine. Music: Mutation-based sql injection vulnerability checking. In *Quality Software, 2008. QSIC'08. The Eighth International Conference on*, pages 77–86. IEEE, 2008.

[23] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM SIGPLAN Notices*, volume 41, pages 372–382. ACM, 2006.

[24] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM SIGPLAN Notices*, volume 41, pages 372–382. ACM, 2006.

[25] S. Thomas and L. Williams. Using automated fix generation to secure sql statements. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, page 9. IEEE Computer Society, 2007.

[26] M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *Dependable Systems and Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 566–571. IEEE, 2009.

APPENDIX

A. HEADINGS IN APPENDICES

The rules about hierarchical headings discussed above for the body of the article are different in the appendices. In the **appendix** environment, the command **section** is used to indicate the start of each Appendix, with alphabetic order designation (i.e. the first is A, the second B, etc.) and a title (if you include one). So, if you need hierarchical structure *within* an Appendix, start with **subsection** as the highest level. Here is an outline of the body of this document in Appendix-appropriate form:

A.1 Introduction

A.2 The Body of the Paper

A.2.1 Type Changes and Special Characters

A.2.2 Math Equations

Inline (In-text) Equations.

Display Equations.

A.2.3 Citations

A.2.4 Tables

A.2.5 Figures

A.2.6 Theorem-like Constructs

A Caveat for the TEX Expert

A.3 Conclusions

A.4 Acknowledgments

A.5 Additional Authors

This section is inserted by LaTeX; you do not insert it. You just add the names and information in the `\addition-alauthors` command at the start of the document.

A.6 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references) to create the .bbl file. Insert that .bbl file into the .tex source file and comment out the command `\thebibliography`.

B. MORE HELP FOR THE HARDY

The sig-alternate.cls file itself is chock-full of succinct and helpful comments. If you consider yourself a moderately experienced to expert user of LaTeX, you may find reading it useful but please remember not to change it.