9-2012

Predicting common web application vulnerabilities from input validation and sanitization code patterns

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

Hee Beng Kuan TAN
Nanyang Technological University

Predicting Common Web Application Vulnerabilities from Input Validation and Sanitization Code Patterns

Lwin Khin Shar and Hee Beng Kuan Tan
School of Electrical and Electronic Engineering
Nanyang Technological University
Singapore 639798
+65 97423763
{shar0035, ibktan}@ntu.edu.sg

ABSTRACT
Software defect prediction studies have shown that defect predictors built from static code attributes are useful and effective. On the other hand, to mitigate the threats posed by common web application vulnerabilities, many vulnerability detection approaches have been proposed. However, finding alternative solutions to address these risks remains an important research problem. As web applications generally adopt input validation and sanitization routines to prevent web security risks, in this paper, we propose a set of static code attributes that represent the characteristics of these routines for predicting the two most common web application vulnerabilities—SQL injection and cross site scripting. In our experiments, vulnerability predictors built from the proposed attributes detected more than 80% of the vulnerabilities in the test subjects at low false alarm rates.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification – *Statistical methods*. D.2.8 [**Software Engineering**]: Metrics – *performance measures*, *product metrics.*

General Terms
Measurement, Experimentation, Security.

Keywords
Defect prediction; static code attributes; web application vulnerabilities; input validation and sanitization; empirical study

1. INTRODUCTION
Recent research in software defect prediction shows that static code attributes such as cyclomatic complexity can be used to effectively predict defective software modules. Following the data mining techniques used in this defect prediction study, in our earlier work [10], we showed that vulnerability predictors, built from static code attributes that characterize input sanitization code patterns, provide an *alternative and cheaper* way of addressing common web application vulnerabilities.

ASE'12, September 3–7, 2012, Essen, Germany

However, web applications also adopt input validation as complementary or alternative to input sanitization. Conditional branching is a common input validation method for preventing web security vulnerabilities and thus, any technique that ignores control flow is prone to errors [14]. Data dependence graphs were used for data collection in our previous work. As data dependence graphs do not include predicates, we could not consider input validation code patterns previously. As such, our initial work is incomplete.

Hence, in this paper, we propose additional static code attributes that characterize input validation code patterns. Static code attributes are collected from backward static program slices of sensitive program points to mine both input sanitization code patterns and input validation code patterns. From such static code attributes and vulnerability information of existing web applications, we then train *vulnerability prediction models* for predicting SQL injection (SQLI) and cross site scripting (XSS) vulnerabilities, the two most common and critical security issues found in web applications [9].

2. CLASSIFICATION
SQLI vulnerability occurs when an SQL statement access database via a query built with unrestricted user inputs because an attack could include SQL characters or keywords in the inputs to alter the intended query syntax. Similarly, XSS vulnerability occurs when an HTML output statement references unrestricted user inputs because an attacker could insert malicious scripts into the inputs to change the intended HTML response output. As such, both SQLI and XSS vulnerabilities are caused by absence, inadequate, or insufficient implementation of input validation and sanitization methods. Since inputs accessed by web application programs are typically strings, input validation checks and sanitization operations if performed in programs are mainly based on string operations.

Therefore, the concept of our approach is to classify the string operations applied on the inputs according to their potential effects on the vulnerability of sensitive program statement k which reference those inputs.

Intuitively, such validation checks and sanitization operations can be found in *backward static slice* of the given web program with respect to k and set of variables referenced in k. As given by Weiser [12], backward static slice S_k with respect to *slicing criterion* $<k, V>$ contains all nodes (including predicate nodes) in the CFG which may affect the values of V at k where V is the set of variables used in k.

Following our initial work [10], we classify a node k in the CFG of a web program as *SQL* sink if the execution of k may lead to

SQLI attacks and as *HTML* sink if the execution of *k* may lead to XSS attacks. For example, Figure 1a shows the sample PHP code extracted from a vulnerable application called *PhpNuke*, and in the code, program statement 5 is an *HTML* sink.

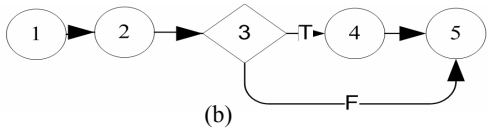
Likewise, we follow the classifications of input types and input sanitization methods. The attributes that represent these classification schemes are listed in the appendix (Table 2).

A variety of *input condition checks* may be found in the nodes in S_k (program slice). For example, Figure 1b shows the CFG of the backward static program slice with the slicing criterion <*5*, {*$url*, *$url_title*}>. Node 3 in the slice uses regular expression matching to check if an input data conforms to a valid URL format. Different condition checks may serve different purposes and may have different effects on the tainted-ness of an input. Thereby, we classify input validation methods carried out at a node into the following types:

1) *Null*: functions that check if the data is present (e.g., isset()).
2) *Size*: functions that return the size or length of an argument (e.g., strlen()).
3) *Containment*: functions that check if an argument contains any predefined characters (e.g., strops()).
4) *Match*: functions or operations that compare an argument with another argument (e.g., strnatcmp(), $a = = "abc").
5) *Regex-match*: regular-expression-based string matching functions (e.g., preg_match()).
6) *Type*: functions that check the type or format of an argument. We further classify this type into *Numeric-type* check (e.g., is_int()) and *Other-type* check (e.g., is_string(), is_file()) to reflect our experience that numeric data type checks usually un-taint the input values.

```
1  $url = htmlspecialchars($url);
2  $url_title = htmlspecialchars($url_title);
3  if (!eregi('(^http[s]*:[/]+)(.*)', $url))
4     $url = "http://" . $url;
5  echo "<b>RELATEDLINK:</b><a href='$url'>
          $url_title</a><br>";
```

(a)



(b)

Figure 1. (a) Sample vulnerable web application program. (b) CFG of the backward static program slice of statement 5.

3. DATA

Combining the 21 attributes listed in the appendix (Table 2) and the 7 attributes (including the sub-types) that we propose in Section 2, each sensitive sink in the program is represented with a 28-dimensional attribute vector that characterizes the input validation and sanitization implemented for the sink.

As an illustration, for the program in Figure 1a, the attribute vector for *HTML* sink node 5 can be collected from its backward program slice (shown in Figure 1b) as (1, 1, 2, 1,…, Vulnerable) in terms of (*HTML, Regex-match, XSS-sanitization, Propagate,..., Vulnerable?*).

We use a data collection tool called *PhpMinerII* to extract the attributes from seven open source PHP-based web applications downloaded from sourceforge.net. Table 1 lists these test subjects and shows the statistics of the 11 data sets collected for this study. The test subjects' vulnerability information can be found in BugTraq (http://securityfocus.com).

PhpMinerII is based on an open source PHP code analysis tool called *Pixy* [3]. Then, we classify the nodes contained in the slices by simply checking the functions invoked and the operators used. *PhpMinerII* handles over 300 PHP built-in functions and 30 PHP operators for classification. The detailed information of the tool and the collected data sets are provided in the authors' web site (http://sharlwinkhin.com/phpminer.html).

Table 1. Data sets

Test Subject	#HTML sinks	%Vuln. to XSS	#SQL sinks	%Vuln. to SQLI
geccbblite 0.1	20	50	9	44
schoolmate 1.5.4	172	80	189	80
faqforge 1.3.2	115	46	42	41
webchess 0.9.0	73	30	53	45
utopia news pro 1.1.4	74	23	-	-
yapig 0.95b	21	29	-	-
phpmyadmin 2.6.0-pl2	58	28	-	-

4. EXPERIMENT

4.1 Experimental Design

***Classifiers*:** We used *three different classifiers*, Naïve Bayes (NB), C4.5, and Multi-Layer Perceptron (MLP), for predicting SQLI and XSS vulnerabilities. Thereby, we could cross-check the general effectiveness and usefulness of the predictors built from our proposed attributes. Detailed information of these classification algorithms can be found in Witten and Frank [13].

***Training and testing method*:** We used 5-fold cross validation method to train and test the three classifiers. The data set is divided into 5 partitions. Each classifier is trained on 4 partitions and then tested on the remaining partition; this process is repeated five times where each partition serves as the test set once, randomizing the order of the test set each time.

***Performance measures*:** Learned predictors were evaluated based on 4 performance measures—probability of detection (*pd*), probability of false alarm (*pf*), precision (*pr*), and accuracy (*acc*). *Pd* measures the effectiveness of predictor at finding real vulnerabilities. *Pr* measures the efficiency, that is, the number of correctly predicted vulnerabilities over the total number of sinks predicted as vulnerable. *Pf* measures the cost of the predictor. *Acc* measures the number of correct predictions over the total number of sinks.

***Attribute ranking*:** We also used *gain ratio* method [13] to identify the most informative (best) attributes and the least informative ones. The objective is to check if using only the best attributes could improve performances.

4.2 Results

Data & Classifier \ Measure (%)		*Pd*	*Pf*	*Pr*	*Acc*
geccbblite-html	**NB**	70	50	58	60
	C4.5	60	40	60	60
	MLP	70	50	58	60
schoolmate-html	**NB**	89	53	87	81
	C4.5	96	18	96	93
	MLP	95	15	96	93
faqforge-html	**NB**	93	21	79	85
	C4.5	94	3	96	96
	MLP	94	2	98	97
webchess-html	**NB**	46	24	46	67
	C4.5	77	8	81	88
	MLP	73	8	80	86
utopia-html	**NB**	77	25	48	76
	C4.5	71	5	80	89
	MLP	94	2	94	97
yapig-html	**NB**	17	27	20	57
	C4.5	50	13	60	76
	MLP	67	27	50	71
myadmin-html	**NB**	75	19	60	79
	C4.5	94	0	100	98
	MLP	94	2	94	97
Average	**NB**	**67**	**31**	**57**	**72**
	C4.5	**78**	**13**	**82**	**86**
	MLP	**84**	**15**	**81**	**86**

(a)

Data & Classifier \ Measure (%)		*Pd*	*Pf*	*Pr*	*Acc*
geccbblite-sql	**NB**	75	20	75	78
	C4.5	75	20	75	78
	MLP	100	20	80	89
schoolmate-sql	**NB**	84	38	90	79
	C4.5	93	27	93	89
	MLP	95	27	94	91
faqforge-sql	**NB**	88	8	88	91
	C4.5	100	8	90	95
	MLP	94	8	89	93
webchess-sql	**NB**	88	21	78	83
	C4.5	100	14	86	93
	MLP	92	17	82	87
Average	**NB**	**82**	**22**	**84**	**83**
	C4.5	**90**	**18**	**86**	**88**
	MLP	**97**	**18**	**88**	**91**

(b)

Figure 2. (a) Results on XSS vulnerability prediction. (b) Results on SQLI vulnerability prediction.

We used the *WEKA* tool presented in Witten and Frank [13] to run the classifiers on the 11 data sets. As shown in Figure 2, on average, MLP and C4.5 classifiers produced excellent results. To provide a comparison point, we used *Wilcoxon signed-rank* test (as suggested by Demšar [2]) to compare the two classifiers' results with a benchmark result of *pd* = 71% and *pf* = 25% (achieved by Menzies et al. [8]). In terms of *pd*, the two classifiers performed statistically better (significant at 99%). But, in terms of *pf*, they were neither statistically better nor worse. This is mainly due to the high *pf* resulting from the tests on *geccbblite-html*.

On average, MLP and C4.5 classifiers achieved *acc* ≥ 85% and *pr* ≥ 80%. These results can be interpreted as at least 8 out of 10 cases are correctly predicted and at least 8 out of 10 vulnerable cases reported by our predictors are worth investigating for security audits.

NB classifier performed the worst among the three classifiers probably due to its naïve assumption that attributes are conditionally independent [13]. To address this problem, we used *WEKA*'s gain ratio attribute evaluator to re-build NB classifier with only the best 3, 5, or 8 attributes. The classifier did achieve more performance increases over the 11 data sets. But, the improvements were not statistically significant according to our Wilcoxon tests. Similarly, performance increases over the 11 data sets were also not statistically significant for MLP and C4.5 classifiers when built with best 3, 5, or 8 attributes.

In summary, our models are effective predictors of XSS and SQLI vulnerabilities. Although data reduction could be applied to improve the predictive power in some specific cases, statistics suggest the limited use of data reduction.

4.3 Threats to Validity

First, our data sets could be biased because all the test subjects contain multiple vulnerabilities. Second, the results may not be applicable for other common web application vulnerabilities. Third, the application of additional data preprocessing methods such as data filtering might alter our results. Fourth, different classification algorithms might result in different performances as well. Lastly, our static analysis-based classifications may not always be precise. The best solution to address these threats is to replicate our experiments.

5. RELATED WORK

Vulnerability detection approaches identify vulnerabilities through tracking the flow of taints (user inputs) to sensitive sinks [3, 14]. Static and dynamic analysis techniques are generally used for taint tracking. Static analysis-based techniques suffer from low precision as these techniques generally overestimate the tainted-ness of inputs. Dynamic analysis-based techniques such as model checking [6] and concolic execution [4] produce zero false positive in principal. But these techniques are generally complex and expensive. By contrast, our approach only requires light-weight static analysis and data mining methods to report vulnerabilities.

Defect prediction approaches [1, 5, 8, 11] typically use static code attributes such as lines of code and code complexity attributes [7] to predict defective software modules. These approaches could correctly predict more than 70% of defects at false alarm rates of 25% or lower. By contrast, our work builds vulnerability predictors through extending the concepts used in defect prediction. We use a set of static code attributes that characterize input validation and sanitization code patterns.

Our previous work [10] ignored input validation code patterns and we observed some false positive cases as a result. Our new predictors could handle those cases. Although the previous results (*pd* > 95 and *pf* < 17) are better than our new results, previous evaluation was on only 3 test subjects and data analysis was

performed on all the subjects together. This may have over-fitted the data.

6. CONCLUSION

In this work, we build vulnerability predictors from the static code attributes that characterize input sanitization and validation code patterns. Static code attributes are measured from backward static program slices of sensitive sinks in the web program. In the experiments, our best predictors achieved $pd \geq 84\%$ and $pf \leq 15\%$ on predicting XSS and SQLI vulnerabilities. As such, the proposed models offer an alternative and cheap way of mitigating common web applications vulnerabilities. Our future work is to enhance our prediction models with more precise dynamic analysis-based classification methods.

7. REFERENCES

[1] Arisholma, E., Briand, L. C., and Johannessen, E. B. 2010. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83, 1, 2–17.

[2] Demšar, J. 2006. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7, 1-30.

[3] Jovanovic, N., Kruegel, C., and Kirda, E. 2006. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*. 258-263.

[4] Kieżun, A., Guo, P. J., Jayaraman, K., and Ernst, M. D. 2009. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering*. 199-209.

[5] Lessmann, S., Baesens, B., Mues, C., and Pietsch, S. 2008. Benchmarking classification models for software defect prediction: a proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34, 4, 485-496.

[6] Martin, M. and Lam, M. S. 2008. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *Proceedings of the 17th USENIX Security Symposium*. 31-43.

[7] McCabe, T. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, 2, 4, 308–320.

[8] Menzies, T., Greenwald, J., and Frank, A. 2007. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33, 1, 2–13.

[9] OWASP. Top Ten project 2010. *http://www.owasp.org*, accessed January 2012.

[10] Shar, L. K. and Tan, H. B. K. 2012. Mining input sanitization patterns for predicting SQLI and XSS vulnerabilities. In *Proceedings of the 34th International Conference on Software Engineering*. 1293-1296.

[11] Tosun, A. and Bener, A. 2010. Ai-based software defect predictors: applications and benefits in a case study. In *Proceedings of the 22nd Innovative Applications of Artificial Intelligence Conference*.

[12] Weiser, M. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*. 439-449.

[13] Witten, I. H. and Frank, E. 2005. *Data Mining*. 2nd ed. Morgan Kaufmann.

[14] Xie, Y. and Aiken, A. 2006. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th USENIX Security Symposium*. 179-192.

APPENDIX

Table 2. Static code attributes that characterize input and sink types and input sanitization methods [10]

Attribute	Description
Client	The number of nodes which access data from external users
File	The number of nodes which access data from files
Database	The number of nodes which access data from database
Text-database	Boolean value 'TRUE' if there is any text-based data accessed from database; 'FALSE' otherwise
Other-database	Boolean value 'TRUE' if there is any data except text-based data accessed from database; 'FALSE' otherwise
Session	The number of nodes which access data from persistent data objects
Uninit	The number of nodes which reference un-initialized program variable
SQL	The number of SQL sink nodes
HTML	The number of HTML sink nodes
SQLI-sanitization	The number of nodes that apply language-provided SQLI prevention functions
XSS-sanitization	The number of nodes that apply language-provided XSS prevention functions
Encoding	The number of nodes that encode data
Encryption	The number of nodes that encrypt data
Replacement	The number of nodes that perform string-based substring replacement
Regex-replacement	The number of nodes that perform regular-expression-based substring replacement
Numeric-conversion	The number of nodes that convert data into a numerical format
Propagate	The number of nodes that propagate the tainted-ness of an input string
Un-taint	The number of nodes that return information not influenced by external users
Custom	The number of user-defined functions
Other	The number of nodes that are not classified as any of the above types
Vulnerable?	Target attribute which indicates a class label—Vulnerable or Not-Vulnerable