6-2012

Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities

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

Hee Beng Kuan TAN
Nanyang Technological University

Citation

SHAR, Lwin Khin and TAN, Hee Beng Kuan. Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities. (2012). *2012 34th International Conference on Software Engineering (ICSE): Zurich, June 2-9: Proceedings*. 1293-1296.
Available at: https://ink.library.smu.edu.sg/sis_research/4679

Mining Input Sanitization Patterns for Predicting SQL Injection and Cross Site Scripting Vulnerabilities

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

***Abstract*—Static code attributes such as lines of code and cyclomatic complexity have been shown to be useful indicators of defects in software modules. As web applications adopt input sanitization routines to prevent web security risks, static code attributes that represent the characteristics of these routines may be useful for predicting web application vulnerabilities. In this paper, we classify various input sanitization methods into different types and propose a set of static code attributes that represent these types. Then we use data mining methods to predict SQL injection and cross site scripting vulnerabilities in web applications. Preliminary experiments show that our proposed attributes are important indicators of such vulnerabilities.**

***Keywords*-defect prediction; data mining; static code attributes; web security vulnerabilities; input sanitization**

I. Introduction

In software defect prediction study, researchers correlate software code attributes with defects. They built defect prediction models by using classifiers that are trained using a set of attributes measured from software modules with known defect information [1]. Static code attributes such as lines of code and McCabe's code complexity attributes [2] are widely used because they can be easily collected and are proved to be capable of predicting defective software modules with high probabilities and low false alarms [3, 4, 5]. However, one drawback of defect prediction approaches is that there is no universal set of code attributes that works on any application domain.

On the other hand, to address the growing risks of security vulnerabilities in web applications, vulnerability detection approaches based on static and dynamic analysis techniques have been proposed. Static analysis approaches [6, 7, 8] are relatively simple to implement, but are known to produce too many false positives. Dynamic analysis approaches [9, 10] provide more accuracy but require potentially complex dynamic environments.

Web applications in general implement a variety of input sanitization schemes to prevent security vulnerabilities such as SQL injection (SQLI), cross site scripting (XSS), and path traversal [11]. An application is vulnerable if the implementation of input sanitization is inadequate or there is no such method implemented. Consequently, the characteristics of input sanitization implemented in a program could be useful for predicting the program's vulnerability.

Hence, in this study, we classify various input sanitization methods into different types and propose a set of attributes that represent these types. By mining such attribute data and vulnerability information from existing web applications, we could train and build vulnerability prediction models for newly developed web applications. Though these prediction models may not identify the vulnerabilities with the same accuracy as concolic execution methods, such static code attributes can be easily collected by using simple static analysis tools. With the availability of data mining tools such as *WEKA* [12], our models are practical. Therefore, they might provide an effective yet cheaper way of finding vulnerabilities in web applications.

To validate this claim, we implemented a proof-of-concept tool called *PhpMinerI* to extract the data of our proposed attributes from PHP programs. We trained two vulnerability prediction models, one for SQLI vulnerabilities and another for XSS vulnerabilities, using the extracted data and known vulnerability information. In our preliminary studies, these models predicted over 85% of the vulnerabilities present in different web applications.

II. Classification

The classification schemes are based on the control flow graph (CFG) of a web application program. As our prototype tool is targeted at PHP programs, we shall provide the examples using PHP language. The sample PHP code in Fig. 1 is extracted from one of our test subjects.

A. Input and Sink Classification

Web application vulnerabilities, such as SQLI and XSS, are mainly caused by the applications' weakness in handling user inputs properly. Typically, a web application program accesses user inputs and propagates them via its program variables for further processing of the application's logics. These processes may often include sensitive program operations such as database updates, HTML outputs, and file accesses. If the program variables propagating the inputs tainted by attackers are not cleansed before being used in those operations, security violations may occur. Therefore, in security, it is important to first identify the sources from which user inputs may be accessed.

Hence, according to different natures of input sources, we classify the inputs into the following types:

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

```
1  $MAX=999;
2  if ($MAX > 0) {
3    $sz_orig = getimagesize('photos/id.jpg');
4    $ratio=$sz_orig[1]/$sz_orig[0];
5    if ($sz_orig[0]> $MAX) {
6      if ($ratio>1) {
7        $height=$MAX;
8        $width=(int)($MAX/$ratio);
         }
         else {
9        $width=$MAX;
10       $height=(int) ($MAX*$ratio);
         }
11     $img_size=“style=‘width:$width;height:$height’”;
     }
   }
12 echo "<div $img_size>";
```

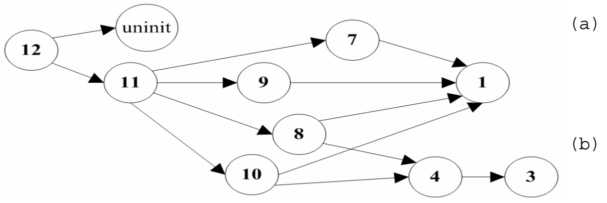

Figure 1. (a) Sample code from Yapig (simplified). (b) Data dependence graph of sensitive sink node 12.

1) *Client*: Data submitted via HTML forms and URLs (e.g., `$_GET`, `$_POST`).
2) *File*: Data accessed from external files such as cookies and XML files (e.g., `$_COOKIE`, `fgets()`).
3) *Database*: Data retrieved from database (e.g., `mysql_result()`).
4) *Persistent*: Data accessed from persistent data objects (e.g., `$_SESSION`).
5) *Uninit*: Variables which may not have been initialized (e.g., `$img_size` at line 12 in Fig. 1a).

We call a node k in the CFG of a web program a *sensitive sink* if the execution of k may lead to security attacks. For this study, we use two types of sensitive sinks:

1) *SQL*: Database operations that are susceptible to SQLI attacks (e.g., `mysql_query()`).
2) *HTML*: HTML output operations that are susceptible to XSS attacks (e.g., `print()` and `echo`).

B. Input Sanitization Classification

By default, inputs to web application programs are strings. As such, input sanitization operations performed in a program are mainly based on string operations. Therefore, our main objective is to classify the string operations that are applied on inputs according to their potential effects on the tainted-ness of the input values propagated.

For each sensitive sink k in a CFG of a web program, we extract its data dependence graph DDG_k from the CFG. This graph contains the data flow information of all the variables used in k. Fig. 1b shows the data dependence graph of the sensitive sink node at line 12 in Fig. 1a.

A variety of *preventive measures against security flaws* may be found in the nodes of DDG_k. Different preventive measures may serve different purposes and may have different effects on the tainted-ness of an input. Therefore, they should be categorized so that a type of preventive measure can be represented with an attribute for data mining purposes. Thereby, we classify input sanitization methods into the following types:

1) *Sanitization*: functions designed to prevent specific security issues (e.g., `mysql_real_escape_string()`, `htmlentities()`). For this study, we use *SQLI sanitization* functions and *XSS sanitization* functions.
2) *Encoding*: functions that encode arguments according to specific encoding formats (e.g., `convert_uuencode()`).
3) *Encryption*: encryption or hashing functions designed to ensure secure data transfer (e.g., `crypt()`, `sha1()`).
4) *Replacement*: string-based substring replacement functions (e.g., `str_replace()`).
5) *Regex-replacement*: regular expression-based substring replacement functions (e.g., `preg_replace()`).
6) *Numeric-conversion*: functions that process arguments and return numeric values (e.g., `intval()`) or numeric type casting operations (e.g., `$a = (int) $b/$c`).

For each node n in DDG_k, if n invokes a language-built-in function or performs an assignment operation, we simply check the function name or the operators used in n and classify it into zero or more of the above types (*PhpMinerI* handles over 300 PHP built-in functions for classification).

It is clear that nodes in DDG_k may also include ordinary operations that may or may not serve any security purpose. They may either *propagate* or *un-taint* the input data. There may also be *other* types of preventive measures though we have not observed them for the security purposes in the context of SQLI and XSS. Furthermore, the function invoked at n may also be a user-written function as input sanitization is often *customized* to users' needs. Consequently, we classify the remaining nodes in DDG_k that are not classified as any of the above types into one or more of the following types:

1) *Propagate*: functions or operations that may convert arguments into different representations but return part or whole of the original arguments (e.g., `$a=$b`, `substr()`, `explode()`); functions that unquote or decode arguments (e.g., `html_entity_decode()`, `urldecode()`, `stripslashes()`).
2) *Un-taint*: functions or operations that return predefined information (e.g., `$a=‘text’`), information derived from configuration settings (e.g., `localeconv()`), or numeric information derived from program operations (e.g., `mysql_field_len()`).
3) *Custom*: user-written or library functions.
4) *Other*: functions or operations that are not classified as any of the above types.

As an illustration, in DDG_{12} shown in Fig. 1b, there is a node that can be classified as *Uninit* as the variable `$img_size` may not have been initialized at node 12. Node 7, 9, 11 can be classified as *Propagate* as they performs simple assignment of a variable or contains string concatenation operation. Nodes 4, 8, and 10 can be classified as *Numeric-conversion* as they perform arithmetic and numeric type casting operations. Node 3 can be classified as *Other* as it invokes a PHP built-in function not classified as any input sanitization type. Node 1 can be classified as *Un-taint* as a predefined literal value is assigned to a variable.

III. Preliminary Experiments

A. Data Collection

For data collection, we implemented a proof-of-concept tool called *PhpMinerI* based on an open source PHP code analysis tool called *Pixy* [6]. Since our method only requires traditional data flow analysis, any other program analysis tool could also be used. For each sensitive sink *k* in a given PHP program, *PhpMinerI* classifies the nodes in its data dependence graph DDG_k (generated by *Pixy*) according to their properties.

In total, there are 18 types, including sub-types, classified in Section II. Each classification type is represented with an attribute. From the nodes in DDG_k, *PhpMinerI* counts the number of nodes that correspond to each classification type and assigns the number to the attribute which represents that classification type. Therefore, each sensitive sink has one attribute vector consisting of the data of 18 attributes plus the target attribute—*Vulnerable?*. For example, the attribute vector for sensitive sink node 12 in Fig. 1b is (*1, 1, 3, 1, 1, 3, ..., vulnerable*) in terms of (*Uninit, HTML, Numeric-conversion, Propagate, Un-taint, Other, ..., Vulnerable?*).

We collected such attribute vectors from three open source PHP-based web applications from SourceForge (*sourceforge.net*) to evaluate the usefulness of our proposed prediction models. These benchmark applications have been used in evaluating some vulnerability detection approaches [6, 9]. Table I shows the information of the test subjects, the summary of the data set collected and their vulnerability information obtained from *Pixy* [6] and *Ardilla* [9].

B. Experimental Design

Different classification algorithms may produce different performances [3, 4]. Therefore, in this study, we use three different classifiers, C4.5/J48, Naïve Bayes (NB), and Multi-Layer Perceptron (MLP), to cross-check the robustness of the prediction models built with our proposed attributes.

C4.5/J48 is a decision tree-based classifier. Naïve Bayes is a simple statistical-based classifier. Multi-Layer Perceptron is an artificial neural network-based classifier. These classifiers assign a given software module to a class of the target attribute based on the training data. In our case, a module is a sensitive sink and the classes of the target attribute are '*vulnerable*' and '*not-vulnerable*'. The details of these classifiers are provided in data mining books such as Witten and Frank [12].

The selected classifiers are implemented in an open source data mining tool called *WEKA* [12]. The tool allows us to simply supply the collected data set for training and testing the three classifiers. Similar to Menzies et al. [3], we used (M=10) * (N=10)-way cross validation on the training data. The data is divided into 10 buckets. The classifier is trained on 9 buckets and tested on the remaining bucket; this is iterated 10 times without testing the same bucket twice.

We used three measures—*probability of detection* (*pd*), *probability of false alarm* (*pf*), and *precision* (*pr*) to assess the performance of learned classifiers. These measures can be computed from the following contingency table:

		Actual-> Vulnerable	Not-Vulnerable
Predicted->	Vulnerable	True Positive (*tp*)	False Positive (*fp*)
	Not-Vulnerable	False Negative (*fn*)	True Negative (*tn*)

The *pd* (*tp/(tp+fn)*) measures how good our prediction model is in finding actual vulnerable sinks. The *pf* (*fp/(fp+tn)*) measures false alarm rate. In an ideal situation, *pd* should be close to 1 and *pf* should be close to 0, that is, the model neither misses actual vulnerabilities nor throws false alarms. The *pr* (*tp/(tp+fp)*) reports the probability that a predicted vulnerable case is actually vulnerable.

C. Results

We ran *WEKA* on a Pentium 3.4GHz 4GBRAM PC. Each classifier was run twice, one run was for the data set of SQL sinks and another run was for the data set of HTML sinks. Both C4.5/J48 and NB took less than a second to complete each run whereas MLP took nearly 2 minutes to complete each run. Results are shown in Table II.

In the experiments, we encountered a few cases that our prediction models could not appropriately handle. For example, see a case from the test subject *Yapig*:

```
1 if(!is_int((int)$phid))
2      die;
3 echo "<div>$phid</div>";
```

Since our approach does not consider input validations through predicates, the input condition check at line 1 will be missed for data mining purposes. We also encountered cases that check the validity of HTTP referrer before the rest of the program operations is executed. For such cases, the mined data may not be appropriate as the data dependence graph used for data mining do not include predicate nodes. The inclusion of control dependency analysis targeted as our future work might handle such cases.

However, in general, our vulnerability prediction models achieved promising results. The models achieved *pd* > 85 and *pf* < 22 which are better than *pd* > 70 and *pf* < 25 benchmarked by software defect prediction studies based on traditional size and complexity metrics [3, 4, 5, 13]. The result *pr* > 93 says that at least 9 out of 10 predicted vulnerable cases are worth investigating for security audits. Although we did not compare the results directly, these results seem to be better than the results reported by static analysis-based vulnerability detection approaches [6, 7], which tend to report many false positive cases. Furthermore, our manual inspections confirmed that our models predicted all the vulnerabilities detected by a dynamic analysis-based approach [9].

In summary, since the proposed attributes can also be easily collected, our models are practical and they offer an alternative and cheap way of detecting security vulnerabilities in web applications.

D. Threats to Validity

First, the data sets used might be small. Second, they might also be imbalanced as the test subjects are bench-marked as vulnerable applications. Third, this preliminary study only focused on SQLI and XSS vulnerabilities due to the limited information available for other types of web vulnerabilities. However, we believe that the proposed

TABLE I. DATA SETS

Data Set	Gecc-BBLite 0.1	SchoolMate 1.5.4	Yapig 0.95b (view.php)
Description	A simple bulletin board	A tool for school administration	Image gallery
LOC	338	8145	4748
#SQL sinks	9	189	0
%Vuln to SQLI	44.4%	80.4%	0
#HTML sinks	17	172	13
%Vuln to XSS	58.8%	80.2%	7.7%

TABLE II. RESULTS

Measure (%) Model		*pd*	*pf*	*pr*
SQLIV Prediction	**NB**	85.3	21.4	93.7
	C4.5/J48	98.7	16.7	95.7
	MLP	97.4	16.7	95.6
XSSV Prediction	**NB**	87.9	9.4	96.3
	C4.5/J48	98.7	11.3	96.1
	MLP	98.0	7.5	97.3

method can be applied to any input-related web vulnerabilities. The best way to prove or refute our results is to replicate and extend our experiments. Interested researchers may request the data sets and the tool used in this study through the authors' emails.

IV. RELATED WORK

Vulnerability detection approaches such as [6, 7] track the flow of tainted data and determine whether or not the tainted data is referenced in sensitive program operations. Such approaches are simple and relatively easy to be adopted; however they have low precision as they do not analyze the correctness of input sanitization operations. More advanced approaches [8-10] applies techniques such as string analysis and concolic execution to determine if the data has been properly sanitized before used in sensitive program operations. These techniques reduce false positives; however, they are computationally expensive.

By contrast, our work predicts vulnerabilities by using data miners learned from code attributes representing the characteristics of input sanitization code patterns. It requires simple static analysis of data flow to collect the data of our proposed attributes.

Defect prediction approaches [3, 4, 13, 14] investigated the predictive performances of classifiers built with static code attributes such as LOC counts, McCabe [2], and other miscellaneous attributes. Their works can be summarized as defect predictors which produce *probability of detection* > 70% and *probability of false alarm* < 25% are useful in practice and much software engineering effort could be saved by using defect predictors. However, Menzies et al. [14] observed that information contents available from size and code complexity attributes are limited. Motivated by this fact, Zimmermann and Nagappan [5] proposed a set of network dependency-based attributes for predicting defects in binaries. Their model performed better than models built from code complexity attributes.

The primary difference with current defect prediction studies is that our work focuses on web security vulnerabilities rather than general software defects. And our work is to predict whether or not a particular program statement is vulnerable whereas existing software defect prediction models in general predict whether or not a software module has defects.

V. CONCLUSION

In this paper, we first classified the types of inputs and sinks that may cause security attacks. Then, we classified the types of sanitization methods that are commonly applied to inputs to avoid security issues. For each sensitive sink in a web program, we collect the static code attributes that characterize these classification schemes. Vulnerability prediction models are then built using the collected data and the vulnerability information of each sink. In our preliminary studies, these models predicted over 85% of SQLI and XSS vulnerabilities in different web applications. Our future work is to conduct more comprehensive experiments on a larger set of systems to further validate these results.

REFERENCES

[1] N. F. Schneidewind, "Methodology for validating software metrics," IEEE Trans. Softw. Eng., vol. 18(5), 1992, pp. 410-422.

[2] T. McCabe, "A complexity measure," IEEE Trans. Softw. Eng., vol. 2(4), 1976, pp. 308–320.

[3] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," IEEE Trans. Softw. Eng., vol. 33(1), 2007, pp. 2–13.

[4] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: a proposed framework and novel findings," IEEE Trans. Softw. Eng., vol. 34(4), 2008, pp. 485-496.

[5] T. Zimmermann and N. Nagappan, "Predicting defect using network analysis on dependency graphs," In ICSE'08, 2008, pp. 531-540.

[6] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," In S&P'06, 2006, pp. 258-263.

[7] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," In USENIX Security'06, 2006, pp. 179-192.

[8] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," In ICSE'08, 2008, pp. 171-180.

[9] A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," In ICSE'09, 2009, pp. 199-209.

[10] M. Martin and M. S. Lam, "Automatic generation of XSS and SQL injection attacks with goal-directed model checking," In USENIX Security'08, 2008, pp. 31-43.

[11] OWASP Top 10, 2010, http://www.owasp.org/

[12] I. H. Witten and E. Frank, Data Mining, 2nd ed., Morgan Kaufmann, Los Altos, CA, 2005.

[13] T. Mende, "Replication of defect prediction studies: problems, pitfalls and recommendations," In PROMISE'10, 2010.

[14] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," Autom. Softw. Eng., vol. 17(4), 2010, pp. 375-407.