L-WMxD: Lexical based Webmail *XSS* Discoverer

Zhushou Tang*†, Haojin Zhu*, Zhenfu Cao*, Shuai Zhao*†
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
†Key Laboratory of Information Network Security, Ministry of Public Security, People's Republic of China
pll@sjtu.edu.cn, zhu-hj@cs.sjtu.edu.cn, zfcao@cs.sjtu.edu.cn, zhaosspirit@sjtu.edu.cn

***Abstract*—*XSS* (Cross-Site Scripting) is a major security threat for web applications. Due to lack of source code of web application, fuzz technique has become a popular approach to discover *XSS* in web application except Webmail.**

This paper proposes a Webmail *XSS* fuzzer called *L-WMxD* (Lexical based Webmail *XSS* Discoverer). *L-WMxD* , which works on a lexical based mutation engine, is an active defense system to discover *XSS* before the Webmail application is online for service. The engine is initialized by normal JavaScript code called seed. Then, rules are applied to the sensitive strings in the seed which are picked out through a lexical parser. After that, the mutation engine issues multiple test cases. Newly-generated test cases are used for *XSS* test. Two prototype tools are realized by us to send the newly-generated test cases to various Webmail servers to discover *XSS* vulnerability. Experimental results of *L-WMxD* are quite encouraging. We have run *L-WMxD* over 26 real-world Webmail applications and found vulnerabilities in 21 Webmail services, including some of the most widely used Yahoo!Mail, Mirapoint Webmail and ORACLE' Collaboration Suite Mail.

Keywords*-Webmail, *XSS*, fuzzer, *L-WMxD

I. Introduction

XSS is one of the most popular vulnerabilities in web application. There are three types of known *XSS* flaws [1]–[3]: 1)Stored *XSS*, 2)Reflected *XSS*, 3)*DOM* based *XSS*, among which Stored *XSS* vulnerability always allows the most powerful attacks. During this type of attack, attack vectors are submitted to web server and stored on the server (in a database, file system or other locations). When other users request this data and attack vectors are displayed on their browser without sufficient checks, an attack may happen. Hackers use *XSS* vulnerability to gain access to victims' browser for identification, e-mail or password. Because the malicious script running under the context of current user, firewall, encryption method or *IDS* (intrusion detection system) are ineffective in preventing this type of attack. As described in OWASP TOP10 security risks (both 2007 [4] and 2010 [5]), *XSS* is one of the most critical risks.

In order to find vulnerability of web application, two approaches are mainly used: "White Box" and "Black Box".

The "White Box" approach needs source code of the web application. This approach can be realized manually or by using automatic code analysis tools, such as Fortify SCA [6], Pixy [7]. Since manual code review is time-consuming, error-prone, costly and hard to evaluate, static analysis tools are more preferred. Most static analysis tools use the source code of web application as input, and then traversal all the possible paths through the manipulation of the *XSS* attack vectors and finally the detection results are presented.

The "Black Box" test is now commonly used by many commercial systems, including AppScan [8], Acunetix [9] and Nessus [10]. These tools inject attack vectors to the web application's *HTML* form elements or *URL* parameters and get feedback immediately. If input validation module of a web application can't sufficiently filter attack vectors, these tools can discover vulnerability of the test target. This kind of test procedure is fit for testing Reflected *XSS*, but not Stored *XSS*. For most Stored *XSS*, the malicious input is stored in the database and later included in another but not the immediately response. This is an obstacle for current existing tools to make the analysis procedure automatically, so, they principally focus on *URL*-based *XSS* like Reflected *XSS* detection. Furthermore, "Black Box" test can't take the advantage of the logic of the program, it is not efficient, and can not guarantee the coverage. Fonseca's [11] study shows that the percentage of false positives of current widely used tools is very high, ranging from 20% to 77%. However, the advantage of "Black Box" test is that it is not sensitive to the implementation programming language of the web application, which makes it ideal for wide deployment.

Both of the "White Box" and "Black Box" methods need to generate test cases to verify the ability of the *XSS* discoverer. Model checking and symbolic execution are the two methods now used to generate test cases, but these methodologies are also based on source code of web application.

"White Box" test is hard to be performed, because Webmail service venders are reluctant to release their Webmail source code. Therefore, "Black Box" test approach is finally adopted by us to discover Webmail's vulnerability. Through comprehensive consideration of different Webmail's filter mechanisms, we make corresponding rules for mutation to generate test cases. The results show that our method is quite effective in finding Webmail *XSS* Vulnerability.

This paper makes the following contribution:

- *Firstly introduce a semi-automatic XSS detector to Webmail*. Until now, there is no publicly released Webmail *XSS* detector. Due to the attribute of Webmail, most of the Webmail *XSS* discoverer job is now done by labor. We firstly introduce a semi-automatic fuzz technique to Webmail *XSS* discoverer and achieve a good success rate.
- *Deliberately selected seeds and rules*. Seeds we used can widely cover most possible usage of JavaScript in *HTML*. Based on well known documents and experiment, rules are selected deliberately and made by sufficient observation of various kinds of Webmail applications.

978-1-4577-0248-8/11/$26.00 ©2011 IEEE

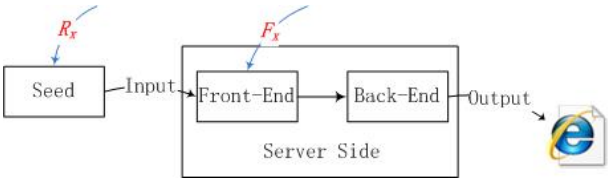

Fig. 1. Relationship between F_x and R_x

They are set on the assumption of the possible defects of a Webmail filter module, which should be responsible for finding scripting commands or meta-characters in untrusted inputs and filtering any such contents before these inputs are processed by the web applications and returned to users' browser.

- *A lexical-based test case generating engine*. Unlike other *XSS* fuzz methods which use pre-defined test cases to discover the vulnerability of Web application, we make different rules corresponding to different filter's strategies of Webmail server. Our Rules work on a lexical-based parser to translate the sensitive areas of the input seeds into different test cases.

The rest of the paper is organized as follows. Section II describes the target we want to defeat and the seeds, rules we used for generating *XSS* attack vectors. In section III, prototype tools for testing are described. We present experimental results in section IV. In this section, a brief analysis will be followed after the list of vulnerabilities our tools find. Section V presents related work on discovering and detecting *XSS* attacks. Finally, in section VI, we present our conclusion and future work.

II. Fundamental Design Principles

In this section, first we give a brief overview of seeds and discussion about rules which are used for generating test cases. Then we will discuss the lexical parser used, through which rules can be employed. The relationship of the Webmail's filter and rules is presented by Figure 1. On the server side, the filter module is divided into Front-End (F_x) and Back-End. The Front-End is designed to transform original user input data to a unified expression, then the Back-End of filter module can apply *XSS* attacks detection using fix signature. Therefore, if our mutated data can defeat the Front-End of the filter module, it is likely to bypass the Back-End. We design rules (R_x), which are applied to the seeds to generate new test cases to bypass the Front-End of a Webmail's filter module. The filter's strategies for transforming the original input data and the rules we used will be discussed in detail in the following subsections.

A. Seeds selected

After investigating more than 100 *XSS* samples, which are collected from previous *XSS* in the wild like Yahoo!Mail and Hotmail *XSS*, we select a serial of seeds deliberately. Each seed is a standard type of *XSS* attack vector, such as pure JavaScript, *HTML* tags, *HTML* attributes, events, Flash and ActiveX. S_x represents the seed we used and some of the seeds are listed in Table I.

Seed	Description
S_1	<script>alert('xss')</script>
S_2	<img src="javascript:alert('xss');">
S_3	<img style="xss:expression(alert('xss'))">
S_4	<body onload="alert('xss')">
...	...

TABLE I
Seeds used by *L-WMxD*

Filter	Description
F_{11}	The mail server does nothing on the mail.
F_{12}	The mail server employs a weak filter, it transforms *HEX* value with semicolons to *ASCII* string and checks sensitive string in it.
F_{13}	The mail server employs a weak filter, it transforms *DEC* value without semicolons to *ASCII* string and checks sensitive string in it.

TABLE II
Different strategies of transforming *HEX/DEC* to *ASCII* taken by a Webmail server

B. Rules for mutation

Considering most Webmail systems have their own mechanisms for sensitive string identification, like *"JavaScript"*, *"expression"*. We mainly apply rules to sensitive strings in order to make mutation escape capturing by the Webmail filter. Because sensitive strings always have some common features, we collect them in advance and build a database to hold them. The rules discussed in the following subsection can be applied to the sensitive strings, then we check whether they can bypass the Webmail filter mechanism.

Currently, there are six rules mainly used for making mutation, we present them from R_1 to R_6.

1) R_{1x}, ASCII → HEX/DEC transform: This rule is used to transform *ASCII* encoded string to *HEX/DEC* encoded string. The *HEX/DEC* encoding formats are *HEX* value with semicolons and *DEC* value without semicolons. Table II lists the different strategies the filter may take when dealing with *HEX/DEC* encoded string. The strategies are made by assumption. Table III is the corresponding rules we take.

The relation of F_{1x}, R_{1x}, and the result is:

$$XSS = (F_{11} \bigwedge R_{11}) \bigvee (F_{11} \bigwedge R_{12}) \bigvee (F_{11} \bigwedge R_{13}) \bigvee (F_{12} \bigwedge R_{12}) \bigvee (F_{12} \bigwedge R_{13}) \bigvee (F_{13} \bigwedge R_{11}) \bigvee (F_{13} \bigwedge R_{13})$$

If server can transform mixed type of encoding value from *HEX* and *DEC* to *ASCII*, our R_{1x} rule will not defeat the filter mechanism.

Rule	Description
R_{11}	*HEX* value with semicolons encoding. Using this rule, we can bypass F_{11} and F_{13}.
R_{12}	*DEC* value without semicolons encoding. Using this rule, we can bypass F_{11} and F_{12}.
R_{13}	Mixed encoding with *HEX* and *DEC* values. Using this rule, we can bypass F_{11}, F_{12} and F_{13}.

TABLE III
Rules we make to defeat filter strategies of F_{1x}

Filter	Description
F_{21}	The mail server does not filter annotation for the following procedure.
F_{22}	The mail server employs a week filter. The week filter takes the longest match principle, so that it wrongly deals with annotation embedded in user input. (The true algorithm *IE* use for identifying annotation is showed as Figure 2.).

TABLE IV
Different strategies of processing annotation a Webmail server takes

Rule	Description
R_{21}	Normal annotation like *"/*xss*/"*. This rule can help us to bypass F_{21}.
R_{22}	Complicated annotation like *"expre/*xss*/ssi/*xss*/on"*. After a well written filter, string "expression" should be left. If the filter mechanism is not designed well, the frond-end part will emit string "expreon". Certainly, this will bypass later check. We also provide other annotation for insertion, like "/*/*/", "/*/*xss*/".

TABLE V
Rules we make to defeat filter strategies of F_{2x}

2) R_{2x}, Annotation insertion: If the mail sever simply filters sensitive words like "*expression*" using a regular expression, the inserted annotation "*/*xss*/*" may help the injected script to bypass the filter. This type of grammar can work well under *IE* (Internet Explorer). Different strategies the filter takes when dealing with annotation are given in Table IV, and Table V gives the corresponding rules.

Relation between F_{2x}, R_{2x}, and the result is:

$$XSS = (F_{21} \bigwedge R_{21}) \bigvee (F_{21} \bigwedge R_{22}) \bigvee (F_{22} \bigwedge R_{22})$$

3) R_3, Enter insertion: Both *HTML* labels and JavaScript statements can be fragmented to multi-lines, for example, the maximal multi-line transaction of

```
''<script>alert('xss')</script>"
```

is:

```
''<script
>
alert
(
'xss'
)
</script
>"
```

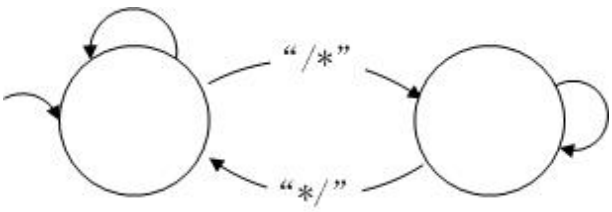

Fig. 2. *IE*'s lexical parser (Makes by observtion but undocumented)

The transformed string can work well under Internet Explorer. If the mail server's filter can not deal with the inserted new line in seed, a new *ENTER* may help us to escape capturing by the filter. The places where a new line can be inserted have some features in common. For example: *ENTER* can not be inserted into the place between the starting of tags or attributes (characterized by '<' symbol) and the first letter, but can exist between the last letter and the closure of tags or attributes (characterized by '>' symbol). There should be no *ENTER* in a string, which is presented by closure of double quotation marks. To identify a string, we need the assistance of a lexical parser which will be discussed later.

4) R_4, lower→upper case transform: HTML language does not require lower or upper case strictly, but JavaScript language is case sensitive. Therefore, the mixed *HTML*, JavaScript and their different requirement for case encoding will make Webmail filter puzzled. For example, "expression" and "EXPRESSION" will all be accepted by *IE*, but if Webmail's filter do not sensitive to this change, our transformed attack vectors will successfully bypass the Webmail's filter. So R_4 rule mainly makes case change on sensitive string of *HTML* language.

5) R_5, ANSI encoding→Other encoding transform: There are many types of encoding, like *ANSI*, *UNICODE*, *UNICODE* of big endian and so on. *IE* accepts all these kinds of encoding formats. But if the server filter can not deal with these transformation, our attack vectors will reach victim's browser successfully.

6) R_6, DBC case→SBC case transform: Lower version of *IE* is likely to accept *SBC* case character, so we transform *DBC* case character to *SBC* case. If Webmail server does not deal with *SBC* case character, our attack vectors will execute on user's browser eventually. This means, even sensitive string "JavaScript" is filtered, "JavaScript" may not be captured.

7) Mix multiple rules: Multiple rules can be applied on the same seed, newly generated test cases M_i can be used for rotate mutation. Following is the mutation algorithm:

$$M_{i+1} = R_i(M_i)(i = 1 \sim 6, R \in \{R_1, R_2, R_3, R_4, R_5, R_6\}, M_0 = S_j)$$

To avoid the cases that the Front-End of a Webmail filter does not apply decoding procedure but simply dropping unknown encoded string, the algorithm we design do abundant job. For example, if we apply R_{11} rule to the sensitive string *"expression"* to get string M_{11}: "&#x65;$xpression$", and the Webmail filter drops the transformed string for the unrecognized string format, then the next string "&#x65;&#120$pression$" based on M_{11} and R_{12} is also dropped by the server. So, we adjust the sequence of test cases generation to make the test cases take effect. The test case generation procedure can be presented as following:

```
1  SeedGeneration(seed){
2      curRule = 1;
3      if(curRule == 1)
4      {
5          TestCase = Rule1(seed);
```

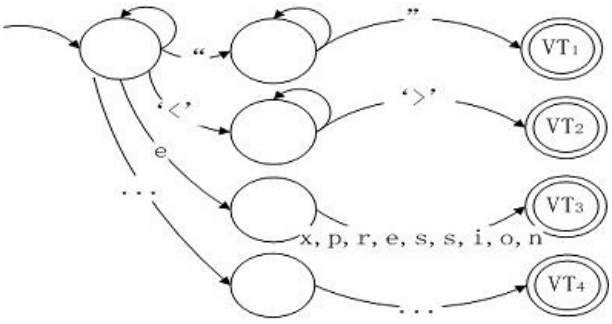

Fig. 3. Lexical parser for identifying sensitive string (For the sake of brevity, the figure does not depict other edges.)

```
6           SendTestCaseToWebmailServer(TestCase);
7           curRule ++;
8           SeedGeneration(TestCase);
9       }
10      if(curRule == 2)
11      {
12          TestCase = Rule2(seed);
13          SendTestCaseToWebmailServer(TestCase);
14          curRule ++;
15          SeedGeneration(TestCase);
16      }
17      ...
18  }
```

C. lexical parser

As described above, rules can not be applied to certain part of seed, so a lexical parser is needed for identifying sensitive strings or tags of *HTML* or JavaScript language. The parser we used is described in Figure 3.

At end point VT_1, a string is identified, almost all rules can be applied to the string; A tag of *HTML* is found at end point VT_2, R_3 rule can be applied between the last character of a string and the "$>$" symbol. Considering certain mail areas do not accept multi-lines input, test cases generated by R_3 normally should not be applied to these areas, like "Subject" area.

III. Implementation

We have implemented two prototype tools to send new test cases generated by *L-WMxD* to Webmail server.

1. An automatic mail sender built on *SMTP* protocol is developed to make a comprehensive test. This tool has the following characters to support the rules we discussed above:

- Test areas include not only mail body, subject, but also sender name, receiver name, *CC* name, etc.
- Multiple test case encoding types for selection.
- For most of Webmail server can not tackle multi-line text of subject, sender name, receiver name or CC name areas, these kinds of messages will be rejected at once when received. Therefore, it is programmed as optional for us to decide whether to apply R_3 to them or not.
- Most mail server set anti-spam mechanism and this will interrupt the procedure of testing, we choose gmail as our relay server. Since gmail server needs *SSL* connection, we add *SSL* support for this tool.

2. A Paros [12]-based tool is developed to make an interception. Using this tool, we can modify and resend the captured message. Paros is powerful to deal with *HTTPS* protocol. Using a fake certification, Paros can get plain text of the encrypted data over *HTTPS*. This tool can also help us to do interception when testing Microblog or web-based online chat service.

IV. Evaluation

Some special mail systems are hard to register (identification card number needed, etc). In order to test these Webmail systems, we use hacking tool *HScan* to guess week password. Through this method, we gain password of certain Webmail user account, then we can do the subsequent tests on it.

Another trick is adopted to deal with the case that weak password is not available. We call this trick "blind injection". For example, we transform the functions of test cases from invoking a message box to downloading an empty text file residing on our server. The test cases are sent to a Webmail server and then the Webmail user interacts with the new coming message, after that, a request may happen on our server. If there is a request for the empty text file, we think *XSS* attack may affect this kind of Webmail application.

L-WMxD has been applied to 26 real-world Webmail services using our tools and the discovered *XSS* is verified by the latest web browser: *Internet Explorer 8.0.7600.16385*. The *XSS* we found and *L-WMxD*'s performance will be discussed in the following.

A. Found vulnerabilities

Table VI shows *XSS* we have discovered by testing. As for Webmail, sending message and receiving the same message are asynchronous. This means that a sequence of interactions are needed when viewing a new coming message, and more labor work is needed when making the audit phase automatical. Till now, we simply check mails one by one after sending all test cases, and manually confirm whether there is a vulnerability (a message box or a desired request appeared through an unmodified Paros). If there is, we classify it as *find-xss*. As for "blind injection" test method, we are not sure if this request is made by a Webmail user or *XSS*, this type of request is classified as *unknown*. Others, test procedures which are terminated in advanced (because there are too much test cases generated) are grouped as *none-xss*. Test results show that we have found *XSS* vulnerability in 21 Webmail service and 1 Webmail service is likely to have defects.

B. Case Study

Although we have informed all the venders, most of the *XSS* we discovered keeps the state of un-patched. Till now, we are providing supports to the Webmail venders to fix their problems, ChinaUnionPay for example. Therefore, we will only give a brief description and the test cases will not be presented.

1) Test result using R_{1x} rule: Through R_{13} rule, we find *XSS* vulnerability in Mirapoint Webmail 3.10.6, Mirapoint Webmail 3.10.8, mail.126.com, mail.163.com, etc.

Webmail	*find-XSS*	*none-XSS*	unknown
mail.eastday.com	√	-	-
mail.hexun.com	√	-	-
mail.186sh.com	√	-	-
mail.133sh.com	√	-	-
mail.139.com	√	-	-
mail.wo.com.cn	√	-	-
mail.163.com.cn	√	-	-
mail.263.com	√	-	-
mail.yeah.net	√	-	-
mail.sohu.com	√	-	-
mail.vip.sohu.com.cn	-	√	-
mail.sina.com.cn	√	-	-
mail.vip.sina.com	√	-	-
mail.21cn.com	√	-	-
mail.tom.com	√	-	-
mail.vip.tom.com	√	-	-
mail.yahoo.com	√	-	-
gmail.com	-	√	-
KOAL secure mail	√	-	-
mail.ru	√	-	-
freemail.ru	-	√	-
ORACLE' Collaboration Suite Mail	√	-	-
freemail.eyou.com	√	-	-
Mirapoint 3.10.x	√	-	-
mail.qq.com	-	√	-
mail.aol.com	-	-	√
Total	21	4	1

TABLE VI
Webmail vulnerability that have been discovered by *L-WMxD*

2) Test result using R_{2x} rule: By using R_{21} on seed generation we discover *XSS* vulnerability in mail.186sh.com, mail.133sh.com, mail.139.com, etc.

3) Test result using R_3 rule: Test result shows that R_3 is very effective in discovering *XSS* vulnerability. More than 10 Webmail services are defective in filtering such kind of attacks, such as mail.wo.com.cn, mail.163.com. Some of them are patched without the public announcement on March 2010.

4) Test result using R_4 rule: R_4 rule has been used to discover vulnerability in mail.21cn.com.

5) Test result using R_5 rule: Using rule R_5, we discover *XSS* in mail.ru and "KOAL secure mail", it's likely that they can not deal with message encoded with *UNICODE*.

6) Test result using R_6 rule: Since R_6 need to be tested under elder version of *IE*, we don't do much test on it.

7) Others: Taking advantage of the tools we developed based on *L-WMxD*, we can perform a comprehensive test. Through the test procedure, we discover *XSS* in mail.sohu.com, freemail.eyou.com and so on. For example, when attack cases are applied to "title" area of a message, we discover vulnerability of mail.sohu.com even with pure seed (seed is send directly without transformation). We believe such area of a message is ignored by filter mechanism for years.

Distribution of *XSS* we have found when using corresponding rule can be presented in Figure 4.

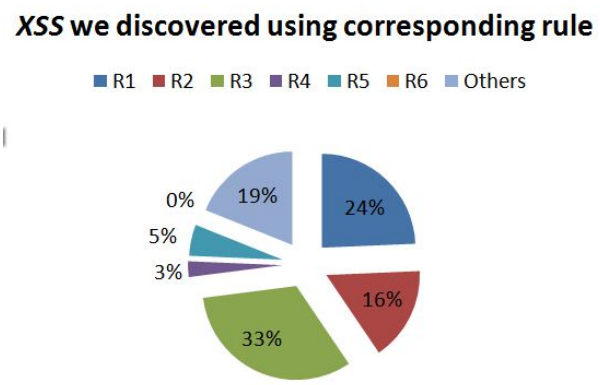

Fig. 4. There are overlaps of *XSS* in a same Webmail server (different rules take effect on the same Webmail server).

V. Related Work

Nowadays, most *XSS* detection tools rely on source code of web applications. Techniques employed by such tools include control-flow, data-flow or formal method. Considering the stage that the *XSS* detection method is employed, we classify current work into "Active Defense" and "Passive Defense". "Active Defense" means discovering flaws of software in advance before it is publicly released. "Passive Defense" means that discovering mechanism is applied to online software to detect *XSS* attack in real time. Michelle Ruse [13] has done a good job of collecting latest research about *XSS* attack detection.

A. Active Defense

MUTEC [3] applies mutation on *PHP* source code of a product to verify whether a test case can bypass a filter. Pixy [7] uses flow-sensitive, interprocedural and context-sensitive data flow analysis to discover vulnerable points in a *PHP* program. In addition, alias and literal analysis are employed to improve the correctness and precision of the results. QED [14] is a goal-directed model-checking system (based on Java PathFinder Model Checking system) that automatically generates attacks in standard Java web applications. It uses programmable technique to automatically generate attacks for large web-based applications.

B. Passive Defense

The "Passive Defense" approach needs intervention with original web applications, which will have an impact on robustness and stability of the application potentially. Further, it also specifies the language in which an application is written.

"Passive Defense" mainly resides on client side [15]–[18] and most work focuses on "Cross Site Request Forgery (CSRF). Using technique of dynamic data tainting and static analysis, this kind of defending technique can avoid *XSS* attacks effectively. Since it is hard to deploy such technique to every browser, it's better to find vulnerability in advance.

Most current methods have difficulty in verifying the true and false of the *XSS* found (because simulating the real world browser is very difficult), which leads to high false positives [11]. XSSDS [1], XSS-GUARD [19] use Firefox components like rbNarcissus [20] to precisely identify scripts in a web page. Wassermann's [21] work is based on tainted information flow with string analysis. He also checks whether un-trusted

parts of the document can invoke the JavaScript interpreter or not. Such a method is a good remedy for the shortcoming of current identification problem. However, for *IE*, it's not an open source software and is hard to decide whether certain JavaScript can run on it.

C. Test case generation

Test case generation is commonly considered as a criterion for code coverage. Well generated test cases can test more branches of a program. Dynamic execution is used for attack vectors generation [22]–[25]. For example, Kieżun et al [23] presents a dynamic tool, Ardilla, to create SQL and *XSS* attacks. This tool uses dynamic tainting, concolic execution for attack-candidate generation and validation. Moreover, Kudzu [25] introduces a string constraint solver for JavaScript to perform symbolic execution in-depth.

VI. Conclusion and future work

An effective test of *XSS* vulnerability helps to fix implementation early and decrease losses incurred for the end users. Experimental test shows that *XSS* vulnerability extensively exist in Webmail service and our *L-WMxD* is effective in discovering it. We will continue the test procedure using tool derived from Paros to discover *XSS* vulnerability in other web applications like online chat system which is used by Google, AOL, etc.

As noted above, it is hard to evaluate the coverage when using "Black Box" testing. We also notice some attack vectors from existing attack like hotmail *XSS* (patched):

```
<style>p{height:expression((window.rrr==914)?xxx=8:(
    eval(code.title)==88)||(rrr=914))},80,180);}</
    style><p><img id=``code" width=1 height=1title
    =``emailkey='emailID23456';var cc=alert('x');
    eval(cc);">
```

The above attack vector can bypass hotmail filter. When it reaches victim's browser, *IE* also accepts such encoding. This is an obstacle for most of *XSS* discover methods, because it's difficult for them to generate such a complex test case. In our opinion, the combination of symbolic execution on server side (based on source code of Webmail to bypass filter mechanism) and client side (based on *HTML* parser to reach the JavaScript interpreter) can generate such kind of attack vector automatically. We will apply symbolic execution to find Webmail *XSS* vulnerability for future work.

ACKNOWLEGEMENT

This research is supported by National Natural Science Foundation of China (Grant No. 61003218 and No. 61033014), Doctoral Fund of Ministry of Education of China (Grant No.20100073120065), Natural Science Foundation of Hohai University under Grant NO. 2009427811, Science and Technology Commission of Shanghai, (Grant No.10511501503), and Opening Project of Key Lab of Information Network Security of Ministry of Public Security (The Third Research Institute of Ministry of Public Security).

References

[1] M. Johns, B. Engelmann, and J. Posegga, "XSSDS: Server-Side Detection of Cross-Site Scripting Attacks," in *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*. IEEE, 2008, pp. 335–344.
[2] A. Klein, "DOM based cross site scripting or XSS of the third kind," *Web Application Security Consortium, Articles*, vol. 4, 2005.
[3] H. Shahriar and M. Zulkernine, "Mutec: Mutation-based testing of cross site scripting," *Software Engineering for Secure Systems, ICSE Workshop on*, vol. 0, pp. 47–53, 2009.
[4] A. Stock, J. Williams, and D. Wichers, "OWASP top 10," *OWASP Foundation, July*, 2007.
[5] J. Williams and D. Wichers, "OWASP top 10–2010," *OWASP Foundation, April*, 2010.
[6] B. Chess and J. West, "Secure programming with static analysis," 2007.
[7] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 6–263.
[8] AppScan, "http://www-01.ibm.com/software/awdtools/appscan/."
[9] Acunetix, "http://www.acunetix.com/."
[10] Nessus, "http://www.nessus.org/."
[11] J. Fonseca, M. Vieira, and H. Madeira, "Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks," in *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*. IEEE, 2008, pp. 365–372.
[12] Paros, "http://www.parosproxy.org/."
[13] "MichelleRuse: Paper surverys." [Online]. Available: http://www.cs.iastate.edu/~mruse/
[14] M. Martin and M. Lam, "Automatic generation of XSS and SQL injection attacks with goal-directed model checking," in *Proceedings of the 17th conference on Security symposium*. USENIX Association, 2008, pp. 31–43.
[15] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: a client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 2006, pp. 330–337.
[16] M. Johns and J. Winter, "RequestRodeo: client side protection against session riding," in *Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*, pp. 5–17.
[17] A. Barth, C. Jackson, and J. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 75–88.
[18] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, vol. 42. Citeseer, 2007.
[19] P. Bisht and V. Venkatakrishnan, "XSS-GUARD: precise dynamic prevention of cross-site scripting attacks," *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 23–43, 2008.
[20] rbnarcissus, "http://code.google.com/p/rbnarcissus/."
[21] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2009, pp. 171–180.
[22] A. Kieyzun, P. Guo, K. Jayaraman, and M. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 199–209.
[23] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst, "Finding bugs in dynamic web applications," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 261–272.
[24] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 249–260.
[25] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for JavaScript," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 513–528.