

國立交通大學

網路工程研究所

碩士論文

自動化網頁測試與攻擊產生

Automatic Web Testing and Attack Generation

研究生：梁偉明

指導教授：黃世昆 教授

中華民國一百零一年六月

自動化網頁測試與攻擊產生

Automatic Web Testing and Attack Generation

研究生：梁偉明

Student : Wai-Meng Leong

指導教授：黃世昆

Advisor : Shin-Kun Huang

國立交通大學

網路工程研究所

碩士論文

A Thesis

Submitted to Institute of Network Engineering

Department of Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2012

Hsinchu, Taiwan, Republic of China

中華民國一百零一年六月

自動化網頁測試與攻擊產生

學生：梁偉明

指導教授：黃世昆 教授

國立交通大學網路工程研究所碩士班

摘要

在資訊科技發達的年代，人們透過網頁方便的瀏覽或取得豐富的網路資源，但在急促的開發腳步下，開發者在開發過程中往往容易忽略安全的考量，導致駭客們能透過開發者的粗心，非法地存取或破壞資源。為了減少與彌補這類的安全問題，在網頁安全的領域上，已有各種不同的方法嘗試去防止或找出這類問題。本論文嘗試扮演攻擊者的角色，以自動產生攻擊字串為目標，達到駭客手動攻擊的相同效果。相較於其他傳統的檢測方法，更能確定漏洞的存在與證明攻擊的可行性。這樣的自動產生過程主要是基於一種動態的軟體測試方法－符號執行(symbolic execution)。最後以此自動化過程，測試幾個開源的大型網頁應用程式，針對已知的漏洞進行實驗，能成功產生相對應的攻擊字串。

關鍵字：網頁安全、符號執行、自動化攻擊碼產生

Automatic Web Testing and Attack Generation

Student : Wai-Meng Leong

Advisor : Dr. Shih-Kun Huang

Institute of Network Engineering
National Chiao Tung University

Abstract

In the well-developed information age, people are easy to get the rich internet resource through web pages. However, in the rapid development process, developers often tend to ignore the security concern carelessly. This leads to access or destroy the resource illegally by hackers. In order to reduce and fix these types of security issues, various methods have been proposed and attempted to locate or prevent them in the field of web security. This thesis attempts to act as an attacker and exploit web applications directly. Our target is to automatically generate the attack string and reproduce the results, emulating the manual attack behavior. In contrast with other traditional detection and prevention methods, this thesis can certainly determine the presence of vulnerabilities and prove the feasibility of attacks. This automatic generation process is mainly based on a dynamic software testing method – symbolic execution. Finally, we have applied this automatic process to several known vulnerabilities on large-scale open source web applications, and generated the attack strings successfully.

Keyword : Web security 、 Symbolic execution 、 Automatic exploit generation

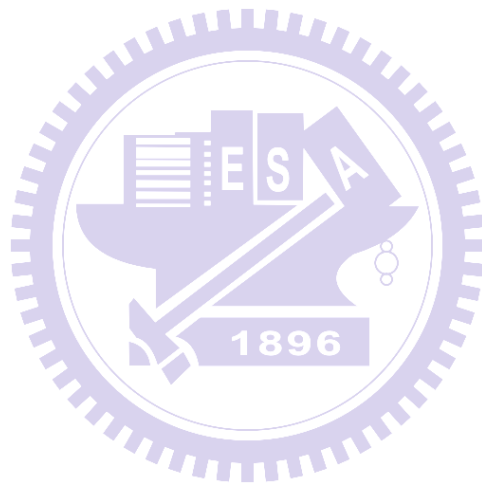
誌謝

從小到大的求學路上，遠方老家的父母與家人們總是會默默的支持著我，使得我可以無憂無慮地去求學問與知識，感謝他們多年來的養育之恩，體諒我任性遠離家鄉求學與工作的決定，一切盡在不言中。

學海無涯，在茫茫學海中，指導老師是我研究方向的一盞明燈，感謝黃世昆老師在這兩年的指導，給了我不少學習的機會與很好的研究環境，帶領我體驗到以前不敢想像的學術層次，過程中碰壁不少，成長也更多，在平日也是一個不時關心學生生活上問題的心靈導師，這一切成爲我生命中的榜樣。

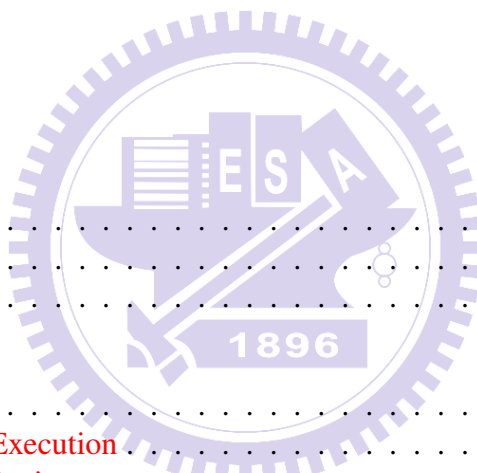
在這兩年的實驗室生活中，感謝肇鈞、銘祥、世欣、博彥、孟緯學長們，他們成爲我學習的目標，感謝翰霖、俊維、韋翔、基傑、奕任，在實驗室裡每天的討論與傾談，激發出不少的靈感與趣味，很高興可以與你們一起研究與畢業，還有正宇、伯謙、鍾翔、劉歡、俊諺學弟妹們，你們給了我研究所生活中重要的回憶，還有系計中工作伙伴們、室友們與其他一路上有幫助過我的人。

最後也感謝口試當天的孔崇旭老師與宋定懿老師，他們百忙之中抽空到來，對這論文的寶貴指導與建議，更令這論文盡善盡美。美好總是短暫的，兩年的研究所光陰，一瞬即逝，大恩不言謝，謹以此論文，獻給你們。



Contents

摘要	i
Abstract	ii
誌謝	iii
Contents	v
List of Figures	vii
List of Tables	viii
List of Algorithms	ix
1 Introduction	1
1.1 Motivation	1
1.2 Objective	2
1.3 Overview	2
2 Background	3
2.1 Software Testing	3
2.1.1 Symbolic Execution	3
2.1.2 Concolic Testing	4
2.1.3 Dynamic Taint Analysis	5
2.2 Web Security Issues	6
2.2.1 XSS	6
2.2.2 SQL Injection	7
2.2.3 RFI and LFI	7
2.2.4 Directory Traversal	8
2.2.5 Command Injection	8
3 Related Work	9
4 Method	11
4.1 Symbolic Socket	11
4.2 Exploit Generation	12
4.2.1 Constraint solving	13
4.2.2 Cooperation with Web Crawler	14
4.3 Single Path Concolic Mode	15
4.3.1 Restriction	16



5	Implementation	18
5.1	Symbolic Socket	18
5.1.1	Symbolic Environment on S ² E	18
5.1.2	Architecture	20
5.1.3	Symbolic Response and Query Handler	20
5.2	Exploit Generation	21
5.2.1	Simple HTML and SQL parser	23
5.3	Single Path Concolic Mode	24
5.3.1	Optimization	25
6	Evaluation	26
6.1	Experimental Environment	26
6.2	Evaluation for Different Platforms	26
6.3	Evaluation for Exploit Generation	28
7	Conclusion and Future Work	30
7.1	Conclusion	30
7.2	Future Work	31
7.2.1	Other Web Security Issues	31
7.2.2	Symbolic execution with Browser	32
	Reference	32



List of Figures

1	Sample code for software testing	4
2	Symbolic execution for Figure 1	4
3	Concolic testing procedure for Figure 1	5
4	Dynamic taint analysis procedure with sample code	6
5	A PHP sample code with web security issues	7
6	Symbolic data propagates from HTTP request to HTTP response and database query	12
7	Constraint solving for a sample function	13
8	Exploit generation by constraint solving	14
9	Flow diagram of our automatic process	15
10	Single path concolic mode	16
11	Different path with the given concrete input and the exploit	17
12	A simple architecture of S ² E	19
13	Sample code for symbolic socket between client and server	19
14	Overall architecture for automatic exploit generator	20
15	From web crawler to symbolic request	21
16	From symbolic response or query to exploit generator	21
17	Completing the syntax of the expected attack script	23
18	Concrete input constraints	24
19	Exploit for other web security issues	31
20	Symbolic execution with Browser	32

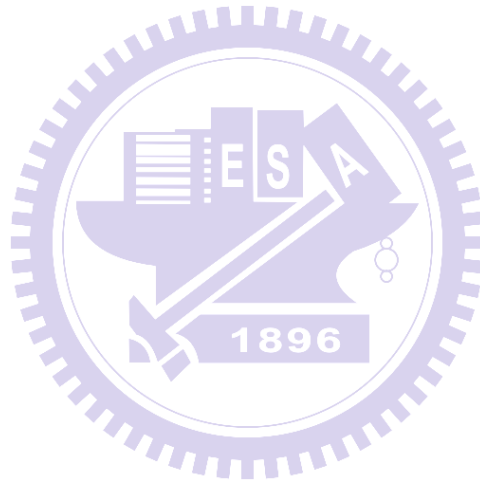
List of Tables

1	Related work for symbolic execution on web applications	9
2	Evaluation for different platforms	27
3	Evaluation for exploit generation with test cases from Ardilla	28
4	Evaluation for exploit generation	29



List of Algorithms

1	Searching for contiguous symbolic data	22
2	Solving the exploit constraints	23



Chapter 1

Introduction

The World Wide Web is a portal for people who want to experience the internet and web pages bring a lot of resources from the internet. This convenience also brings about various security issues at the same time. Most of the issues are caused by the input from web pages, such as user data from HTML form and HTTP cookie. Practically, some inputs are validated or sanitized inadequately by developers. When a user sends an improper input, it results in bug or wrong response. On the other hand, attackers attempt to figure out a malicious input over the inadequate development. Those malicious input data, i.e. “exploit”, often cause unexpected loss and damage. The work in this thesis is to build an exploit generator for automatically figuring out the exploit in order to fix them in time.

1.1 Motivation

In the web security research, various methods have been proposed and attempted to solve web security issues. In contrast with traditional prevention and detection methods, exploit generation[1, 2] is a more precise way and provides a better result because it does not generate any false positive and inaccuracy results. The generated exploit is a strong evidence to identify the presence of vulnerabilities. On the other hand, the purpose of generated exploit is not only a harmful input for web applications, but also a practical sample for developers easier to recognize the vulnerability. It can also help developers prioritize the bug fixing process. If a bug is exploitable, it must be the highest priority to fix.

For the manual exploit generation, researchers require a strong security background and

knowledge to analyze vulnerabilities. Moreover, the cost of time is also an important consideration. Whether white-box testing or black-box testing is used, manual exploit generation is a high cost process[3]. Therefore, an automatic exploit generator is necessary to finish the overall process in order to save the cost of time and knowledge.

1.2 Objective

Our objective is to automatically generate the exploit for common web security issues on real-world web applications and reproduce the results, emulating the manual attack behavior. Moreover, this automatic process is based on a popular dynamic analysis technique in the field of software testing, symbolic execution[4, 5]. Many related works are also based on it. However, the overhead of symbolic execution on large-scale application is too expensive. Our challenge is to automate the exploit generation process on large-scale web applications.

1.3 Overview

This thesis is organized as follows. Chapter 2 describes the background of software testing and related web security issues. Chapter 3 describes and compares related works. Chapter 4 and 5 explain our method and implementation, respectively. Experimental results are reported in Chapter 6. Finally, Chapter 7 concludes our thesis, with future work.

Chapter 2

Background

In the field of software testing, many approaches and researches have applied to web security. They have their own advantage and disadvantage. For example, static analysis[6] is a testing of an application by examining the code without executing the application. It is good at handling large-scale applications, but false positive may happen. On the other hand, dynamic analysis[6] is a testing of an application during runtime. It is more accurate than static analysis, but its analysis overhead is too expensive. The first section in this chapter introduces advantage and disadvantage of three popular dynamic analysis techniques, which relate to this thesis.

Nowadays, web security issues have various kinds[7]. The second section describes five common web security issues including their exploit process and prevention. All of them are usually caused by inputs without proper validation and sanitization. The exploit generation for the first vulnerability, i.e. Cross Site Scripting, has been evaluated by automatically generating the exploits for vulnerable web applications.

2.1 Software Testing

2.1.1 Symbolic Execution

Symbolic execution[4, 5] is a popular testing approach for software verification and validation proving. Its objective is to explore as many paths in a program as possible. The main idea of symbolic execution is to replace the concrete value of particular variables with symbolic values during execution. Before executing, a path constraint is initialized as *true*. Whenever the program execution encounters an assignment statement that associates with symbolic variable, the

symbolic variable will taint other concrete variables and update the concrete value as symbolic. Whenever the program execution encounters a branch that associates with symbolic variable, the branch condition will be collected into the path constraint. Moreover, symbolic execution will fork a new execution for another path and negate the branch condition, which will be added to another new path constraint. The collection of branch condition in each execution and path constraint goes on until all forked executions finish. Each path constraint is accumulated by its own path execution. All path constraints may be finally solved for a test case by a constraint solver and the generated test cases can reproduce the same execution and explore the same path.

By considering symbolic execution on the example in Figure 1, symbolic variable *input* is specified a symbolic value X at line 1 and a path constraint is initialized as *true*. For the assignment at line 4, the symbolic variable *input* taints the concrete variable *key* and the symbolic value of *key* becomes $X + 10$. For the branch at line 5, the execution encounters the symbolic variable *key* and forks a new execution for another new path. The branch condition, $X + 10 = 54813$, and its negation, $X + 10 \neq 54813$, is added to their path constraint respectively. Whenever two forked executions finish, their path constraints can be solved by constraint solver for their own test cases, 123 and 54803. The procedure shows in Figure 2.

Sample Code	
1	void check(int input)
2	{
3	int key = 0;
4	key = input + 10;
5	if(key == 54813)
6	printf("True");
7	else
8	printf("False");
9	}

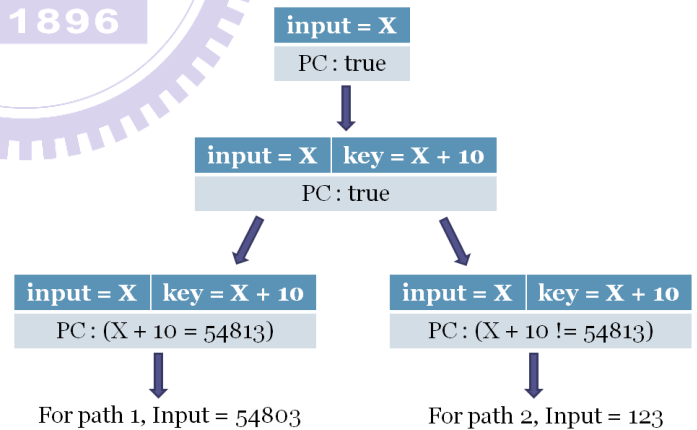


Figure 1: Sample code for software testing

Figure 2: Symbolic execution for Figure 1

2.1.2 Concolic Testing

However, the weakness of symbolic execution is the exponential growth for paths on large-scale program or infinite loop. Concolic testing[8] is another solution for software verification. It is a hybrid software verification technique that combines the concrete and symbolic execution. Its

main idea is to explore only one path at a time and prevents from path explosion simultaneously. From the beginning, a random concrete input is solved by constraint solver for concrete execution on the first path of program under test. For symbolic execution, branch conditions are collected into a path constraint during execution. Until the path finishes, concolic testing negates the last condition in the path constraint and solves a new test case for concrete execution on the next new path. The exploration of paths and the generation of test cases go on until all paths are explored as much as possible. The path traversal is based on depth-first search (DFS) algorithm.

By considering the same example in Figure 1, a random concrete input is assumed as 321. For the assignment at line 4, concrete variable *key* equals to 331. For the branch at line 5, it takes *false* and adds the branch condition $X + 10 \neq 54813$ into the path constraint. After the termination of first path, concolic testing negates the last condition in the path constraint and becomes another new path constraint $X + 10 = 54813$. A new test case 54803 is solved by constraint solver with the new path constraint and is used to explore for another new path. The procedure shows in Figure 3.

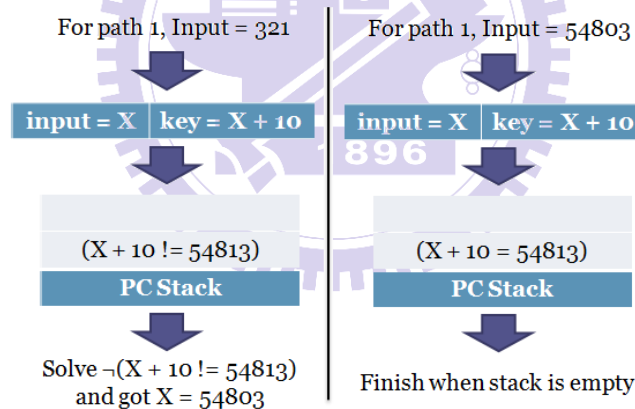


Figure 3: Concolic testing procedure for Figure 1

2.1.3 Dynamic Taint Analysis

Dynamic taint analysis[9, 5] is a technique for tracking the information flow in software applications and attempts to locate the data flow of possible vulnerabilities. It relies on the dependency of variables during the analysis, but not symbolic values, conditions or path constraints. If data are from un-trusted source, e.g. user inputs, HTTP parameters, file inputs, dynamic taint analysis will mark the state of the data as “tainted”. During the analysis, tainted variables propagate

its state to other non-tainted variables through the data dependency. If a tainted variable reaches the sink, e.g. database operations, command executions, HTML outputs, it can affirm that the data flow is dangerous and data from input source are vulnerable. However, taint analysis cannot generate any test case or exploit without symbolic data. And false positive may happen without considering the control flow problem in branches. This issue can be reduced by constructing a control flow diagram[10].

By considering an example in Figure 4, the state of un-trusted argument *input* is marked as tainted. For the assignment at line 4, tainted variable *input* propagates another variable *key* and marks as tainted. For the database query at line 8, it is considered as a sink and is tainted by *key*. Thus, the example is vulnerable. However, if the branch takes *false* at line 5, *key* may not taint the sink and false positive may happen. It is also the weakness of dynamic taint analysis.

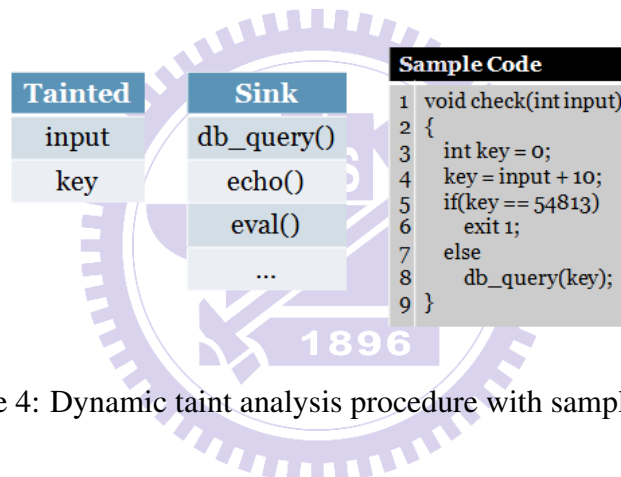


Figure 4: Dynamic taint analysis procedure with sample code

2.2 Web Security Issues

2.2.1 XSS

Cross Site Scripting (XSS) is the most frequent web security issue. Two common classifications of XSS are known as reflected and stored. An attacker crafts a link with a malicious payload. Whenever a victim is lured to click on the crafted link, the malicious payload will be executed forcibly in the browser of the victim. This is an example of reflected XSS. If an attacker injects a malicious payload into websites or databases and becomes persistent, victims will forcibly execute the malicious payload by visiting related pages. This is an example of stored XSS.

In the example of Figure 5, a reflected XSS exists at line 3. An attacker can craft a link, e.g. `index.php?id=<script>alert(document.cookie)</script>`, and lures a victim to click on it. The HTTP cookie of victim is forcibly displayed on a pop-up message box. The prevention of XSS is to validate or sanitize inputs on output functions of web pages, such as `echo()`, `printf()`. `htmlspecialchars()` is one of the suggested routines for sanitizing XSS in PHP.

```
Sample Code
1  <?php
2  include($_GET['theme'].'.php');
3  echo "Welcome, ".$_GET['id']."<br>";
4
5  $result = mysql_query("SELECT * FROM user WHERE
id=".$_GET['id']." AND pass=".$_GET['pass'], $conn);
6  if(mysql_num_rows($result))
7  {
8      echo "Login successful, listing your directory<br>";
9      system("ls -al /home/".$_GET['id']."/".$_GET['dir']);
10 }
11 ?>
```

Figure 5: A PHP sample code with web security issues

2.2.2 SQL Injection

SQL injection is also a common technique to attack databases through websites. By injecting a malicious SQL command into portion of SQL statement, an attacker can hijack the database query and request some illegal operations, e.g. dump password, modify database content or bypass authentication.

In the example of Figure 5, SQL injection exists at line 5. An attacker can craft a HTTP request, e.g. `index.php?id=admin&pass=admin or 1=1`, and the password of `admin` is always correct with the Boolean expression, `or 1=1`. Thus, the attacker can bypass the authentication and login as an administrator. The prevention of SQL injection is to validate or sanitize inputs on the parameters of database query. `mysql_real_escape_string()` is one of the suggested routines for sanitizing SQL injection in PHP.

2.2.3 RFI and LFI

Remote File Inclusion (RFI) and Local File Inclusion (LFI) allow an attacker to include a remote or local file, which is usually a script. This can lead to the arbitrary code execution on

web servers.

In the example of Figure 5, RFI and LFI also exist at line 2. An attacker can send a HTTP request, e.g. `index.php?theme=http://evil.com/webshell.txt?`, to inject a remotely hosted file containing a malicious code. Another HTTP request, e.g. `index.php?theme=/etc/passwd%00`, allows an attacker to read the content of the local password file on web server. A better prevention for RFI and LFI is to prepare a white-list of the allowed file in dangerous functions, such as `include()` and `require()` in PHP.

2.2.4 Directory Traversal

Directory traversal attack allows attackers to access restricted directories and sensitive files placed on a web server by stepping out of the root directory using dot dot slash.

In the example of Figure 5, directory traversal attack exists at line 8. Although the restricted directory, `/home/`, is defined, an attacker can also craft a HTTP request, e.g. `index.php?id=user&pass=123&dir=../`, to list the root directory on the system. The restriction fails finally. A better prevention of directory traversal attack is to configure the restriction in the web server configuration file.

2.2.5 Command Injection

Command injection is similar to SQL injection. A malicious command is injected into functions, which are used for shell command-line execution. An attacker can execute any command like a pseudo shell.

In the example of Figure 5, command injection exists at line 8. An attacker can craft a HTTP request, e.g. `index.php?id=user&pass=123&dir=|uname`, to execute a shell command, `uname`. The prevention of command injection is to validate or sanitize the argument of the shell command. `escapeshellarg()` is one of the suggested routines for sanitizing in PHP.

Chapter 3

Related Work

	SAFELI	Apollo	Ardilla	Kudzu	Rubyx	CRAX
Year	2008	2008	2009	2010	2010	2012
Symbolic executor	JavaSye	Zend Interpreter	Apollo	FLAX	Yices	S2E
Solver	SUSHI	HAMPI	HAMPI	STP+ HAMPI	Drails	STP
Platform	Java	PHP	PHP	JavaScript	Rails	All
Focus on	SQLI	Execution failures, malformed HTML	SQLI, XSS ₁ , XSS ₂	DOM-based XSS	XSS, CSRF, etc	XSS, SQLI
Detect/Exploit	Exploit	Exploit	Exploit	Exploit	Detect	Exploit
Mutation	No	No	No	Yes	No	Yes
Detection method	Syntax trees matching	HTML validator + Oracle	Taint propagate	Conjoin of attack grammars	Assertion + assumption	Symbolic response and query

Table 1: Related work for symbolic execution on web applications

Symbolic execution is a popular software testing technique. Some related works have applied this technique in the field of web security. Table 1 shows a comparison between them and our work, CRAX. SAFELI[11] was a SQL injection scanner based on Java web applications in 2008. It provided a concept for applying symbolic execution to web security early. Apollo[12] was a project from MIT in 2008. It modified the Zend interpreter in PHP to support concolic

execution for searching bugs in PHP web applications. A year later, MIT proposed an improved work called Ardilla[13] in 2009. It combined concolic testing and dynamic taint analysis to perform as an exploit generator. Its objective is similar to ours and we have experimental result for comparisons in later chapters. Kudzu[14] was a symbolic execution framework for JavaScript in 2010. It used attack grammars to solve the exploit and finally found out two unknown vulnerabilities. Rubyx[15] was a symbolic executor for Rails[16] in 2011. It was a recent work for symbolic execution on web security.

As mention in later Section 4.1, platform-independent is one of the contributions in our work than other related systems. The feature of mutation in Table 1 means the ability of constraint solving that explains in Section 4.2.1, which is the other contribution in this thesis.



Chapter 4

Method

Our method is mainly based on symbolic execution to automate the web exploit generation process. Symbolic socket[17] is used to propagate symbolic execution through socket between applications. Exploit generation uses the ability of constraint solving in symbolic execution to solve the constraints of the objective exploit. Single path concolic mode is an option to reduce the overhead on path explosion on large-scale web applications. However, this option has its own restriction. All details are described in this chapter.

4.1 Symbolic Socket

In a real-world scenario, attackers usually craft a malicious input over vulnerable entry points in web pages, such as GET parameters in URL, POST data from HTML form and HTTP cookie, and send a malicious HTTP request through socket. For XSS attack, a malicious HTTP response is returned through port 80 for Apache by default. For SQL injection attack, a malicious query impacts the database through port 3306 for MySQL by default. Symbolic socket can then be applied in this scenario to assist symbolic execution over web servers and applications together. This situation is shown in Figure 6. A symbolic data is prepared and injected into HTTP request. If symbolic data can be propagated to HTTP response or database query during symbolic execution through socket, it will represent that the response or query is vulnerable and can be controlled by the original symbolic data. Therefore, it is possible to identify vulnerabilities through symbolic socket.

The focus of symbolic socket is just on symbolic data, which sends to and receives from

socket. It is unnecessary to concern about what web server uses or how the source code of applications is. They are tested like a black-box. Therefore, it leads to platform-independent and source-independent testing on web server and applications during symbolic execution.

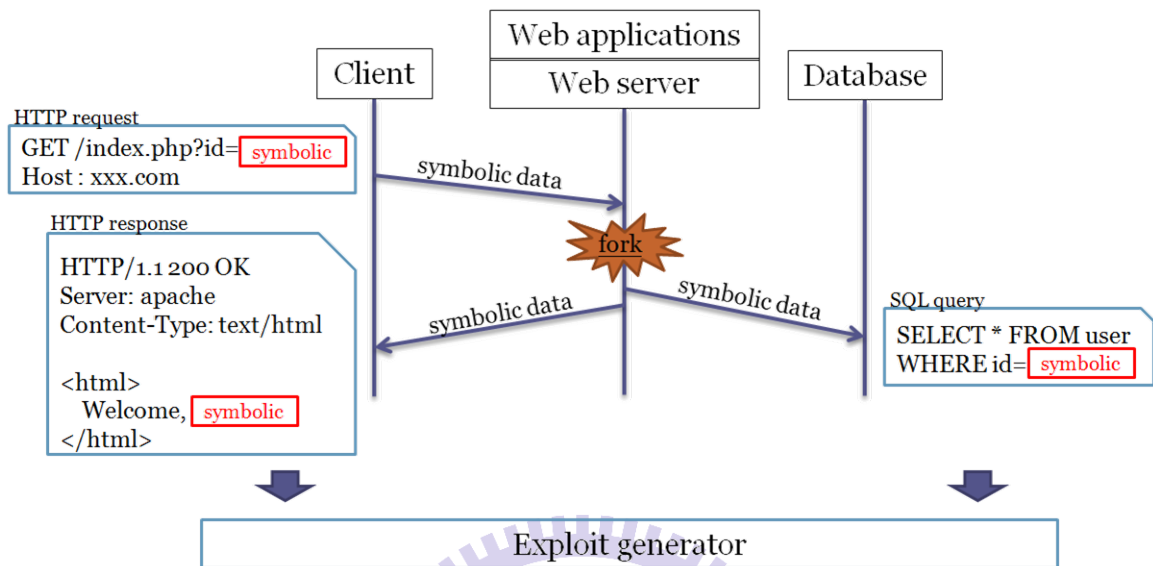


Figure 6: Symbolic data propagates from HTTP request to HTTP response and database query

4.2 Exploit Generation

If a symbolic data is discovered in HTTP response or database query that is received from port 80 or 3306 respectively, it will be an opportunity for exploit generator to generate the exploit in Figure 6.

Whenever the symbolic data is initially discovered in HTTP response or database query, an expected attack script is then constructed, such as `<script>alert(document.cookie)</script>` for XSS or `' or 1=1 -` for SQL injection respectively. It is used to execute the attack on HTML page or database query and be triggered by submitting the exploit. However, the syntax correctness of the expected attack script is one of the concerns. It can be finished by parsing the received HTTP response or database query with a simple HTML or SQL parser respectively so that the syntax of the expected attack script is correct and available.

The format of the received symbolic data is another concern. The symbolic data must be contiguous and the length must be longer than or equal to the expected attack script. If the

symbolic data is longer than the expected attack script, blank spaces will be used to fill the lack of script.

4.2.1 Constraint solving

Constraint solving is an ability of solver in symbolic execution to generate test cases and also the exploit. By considering a sample function in Figure 7, what is the value of x if $f(x)$ has to return 100? The answer can be solved with constraint solving. After the termination of symbolic execution on $f(x)$, two PCs, $X + 10 > 0$ and $X + 10 \leq 0$, are collected for the first path and the second path respectively. To restrict the return value of $f(x)$, a constraint $y = 100$, i.e. $X + 10 = 100$, is added into each PC and attempts to solve each PC by solver to obtain the feasible value of x . Finally, x is solved and equals to 90.

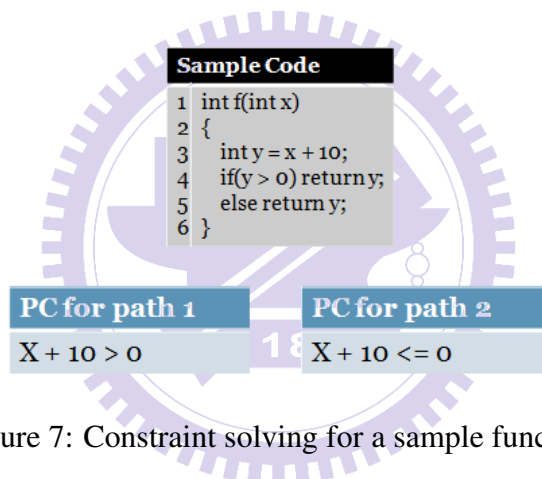


Figure 7: Constraint solving for a sample function

The same concept can apply on exploit generation for assuming x as exploit and $f(x)$ as an expected attack script. By considering an example in Figure 8, what is the exploit if the symbolic data in HTTP response equals to the expected attack script? Whenever symbolic data is discovered in the HTTP response, an expected attack script is constructed as `<script>alert(document.cookie)</script>` by a simple HTML parser. So the length of the contiguous symbolic data must be longer than 41. The later process shows in Figure 8. The expected attack script is used to construct additional constraints character by character and added into collected constraints. The solver in symbolic execution then attempts to solve this set of constraints to obtain the possible exploit. A similar approach can also be applied for the exploit generation of SQL injection.

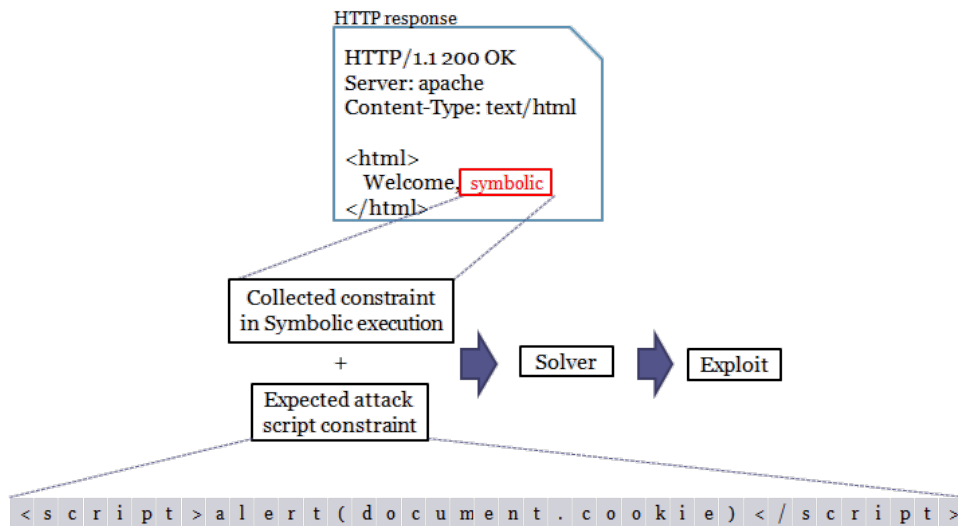


Figure 8: Exploit generation by constraint solving

On most of the traditional web vulnerability, the exploit is directly reflected onto HTTP response or database query without any arithmetic operation or simple mutation. These vulnerabilities can be easily discovered. The ability of constraint solving can assist exploit generation of the potential vulnerability that is under some arithmetic operations or simple mutations. Those vulnerabilities are hard to be discovered in the past and this is one of the contributions in our work.

4.2.2 Cooperation with Web Crawler

Web crawler[18] is a front-end in our web exploit generator. It can figure out all the possible HTTP requests including GET and POST parameters in a web application. Those parameters can be replaced by symbolic data and process symbolic execution through socket. This situation is explained in Section 4.1. To cooperate with exploit generator, a fully automatic process can be constructed to generate web exploit. The flow diagram shows in Figure 9.

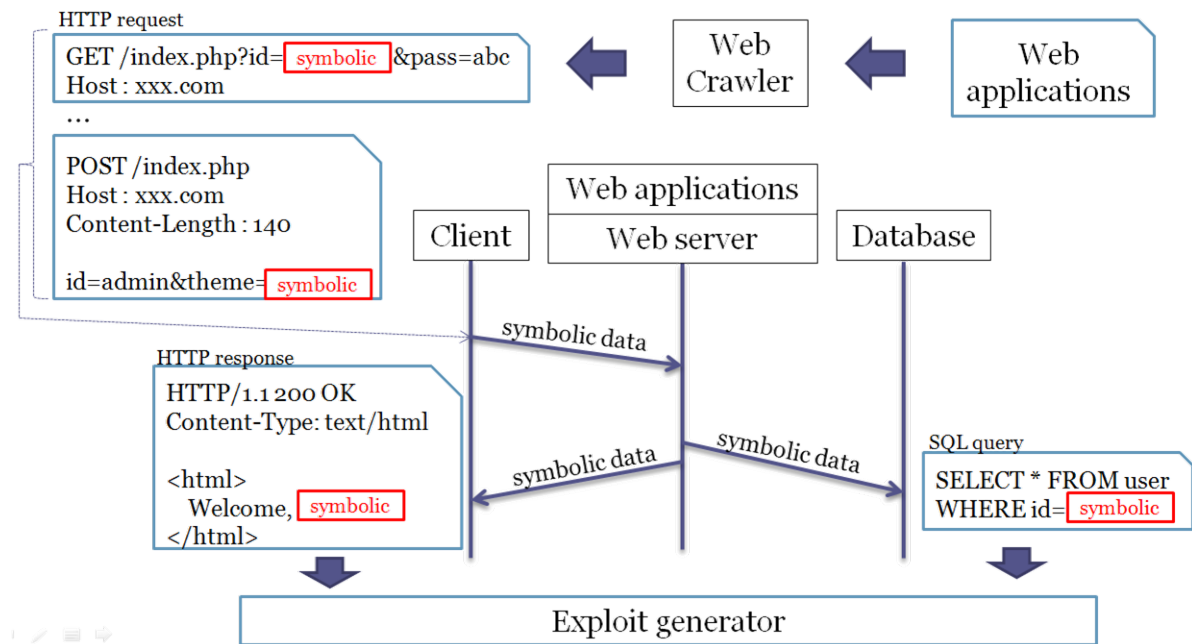


Figure 9: Flow diagram of our automatic process

4.3 Single Path Concolic Mode

The weakness of symbolic execution is the path explosion during execution. This leads to the challenge in this thesis for the exploit generation on large-scale web applications. To reduce the overhead in symbolic execution, we utilize the advantage in concolic testing that explores one path at a time. Regarding a particular single path is more effective than the exploration for all paths.

In concolic testing, concrete values are originally responsible for helping symbolic execution to determine the direction in branches and paths. All paths are explored with their own concrete input. In single path concolic mode, only one given concrete input is fed for fixing the exploration on a particular single path. Whenever symbolic execution encounters branches that associate with symbolic variables, the selection of branches reference the given concrete input instead of the original concrete value. The execution does not fork for another new path anymore.

Moreover, branch conditions are originally added into path constraints for solving a concrete value of the next new path. In single path concolic mode, branch conditions are not used anymore because the concrete input is given and fixed. So branch conditions can be abandoned

and just be collected and kept in the later backup on the exploit generation. This backup mechanism can also optimize the speed of overall execution.

Figure 10 shows that the difference between symbolic execution and single path concolic mode. The single path concolic mode explores only one path with a given concrete input rather than all paths. The overhead on path explosion is reduced. Symbolic data can still propagate and be discovered at HTTP response and database query.

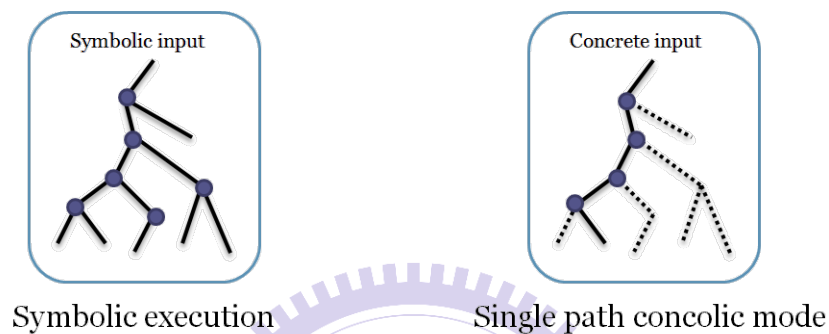


Figure 10: Single path concolic mode

4.3.1 Restriction

Actually, single path concolic mode not only reduces the overhead, but also brings a restricted condition in exploit generation. If an exploit exists in a vulnerable web page to trigger XSS and SQL injection, the path of symbolic execution from exploit to HTTP response or database query must be the same as the path for our given concrete input in single path concolic mode. Otherwise, the exploit cannot be solved by constraint solver with collected constraints that is collected by symbolic execution on the given concrete input. This is a cost of reducing the overhead, or restores this restriction by exploring all paths in traditional symbolic execution or concolic testing.

According to our experimental results, only a part of exploit cannot be solved in some vulnerable cases because of the different path between the given concrete input and the exploit. It usually occurs at the branch of validating, sanitizing or exception checking on the input string. Thus, we consider that the option of single path concolic mode is worth to do. Figure 11 shows that the validation of special characters leads to the different path and different collected

constraints for *BBBBBBBB* and *<SCRIPT>*. The expected attack script, *<SCRIPT>*, cannot be solved finally by a given concrete input, *AAAAAAAA*. But another input string, *CCCCCCCC*, can be solved to reproduce the output string, *BBBBBBBB*.

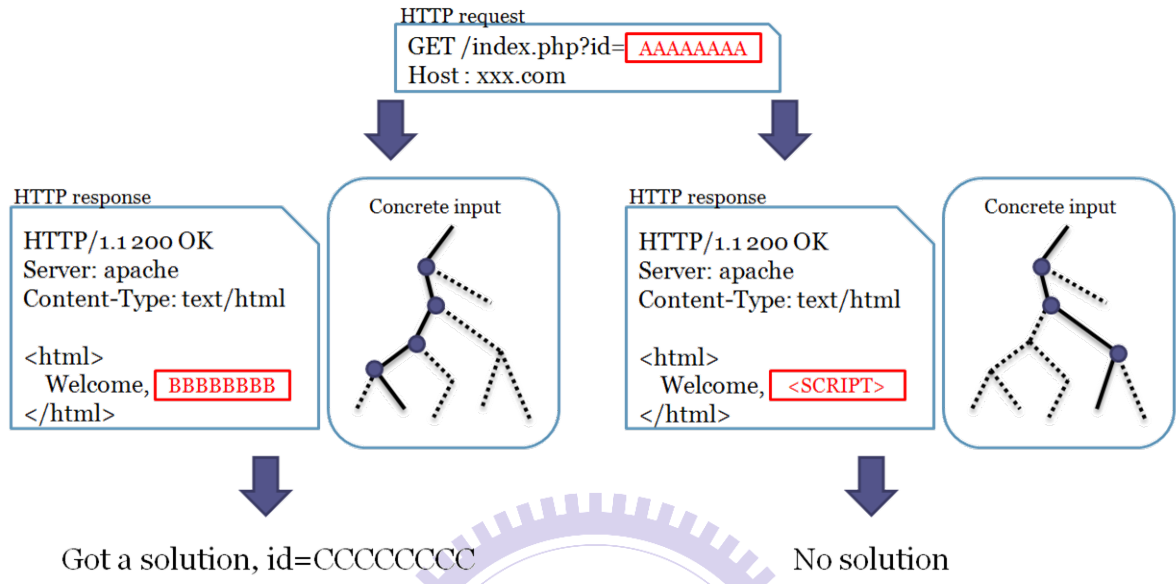
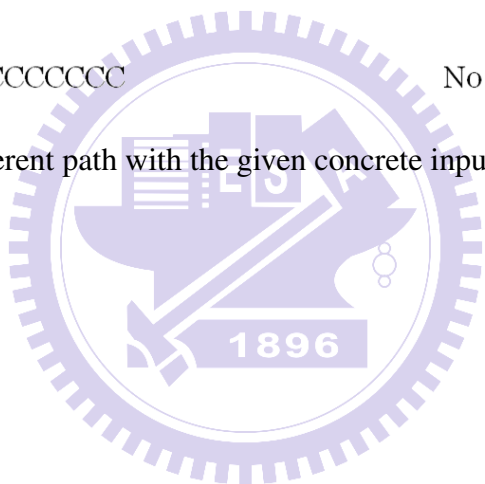


Figure 11: Different path with the given concrete input and the exploit



Chapter 5

Implementation

In this chapter, we explain in detail how our method is implemented on S²E[19], which is a symbolic execution platform. The symbolic environment on S²E assists symbolic propagation through sockets and a handler is built to receive symbolic data under sockets. After receiving the symbolic data, it triggers the exploit generator, which is wrapped as a plugin of S²E. Moreover, single path concolic mode and other optimizations are also implemented to speed up the overall process inside S²E.

5.1 Symbolic Socket

5.1.1 Symbolic Environment on S²E

S²E has an ability to perform symbolic execution on the whole operating system rather than applications. This ability comes from the combination of QEMU[20] and KLEE[17]. KLEE is a symbolic execution engine built on top of the LLVM compiler infrastructure[21]. It implements symbolic execution by interpreting LLVM bitcode. QEMU is a processor emulator that relies on dynamic binary translation to translate instructions between two different host CPU architectures. Whenever a program under test inside QEMU encounters symbolic data, S²E triggers a new LLVM back-end to translate instructions into LLVM bitcode and feeds to KLEE to perform symbolic execution on the whole system. The constraint solver of KLEE is STP[22]. A simple architecture of S²E shows in Figure 12.

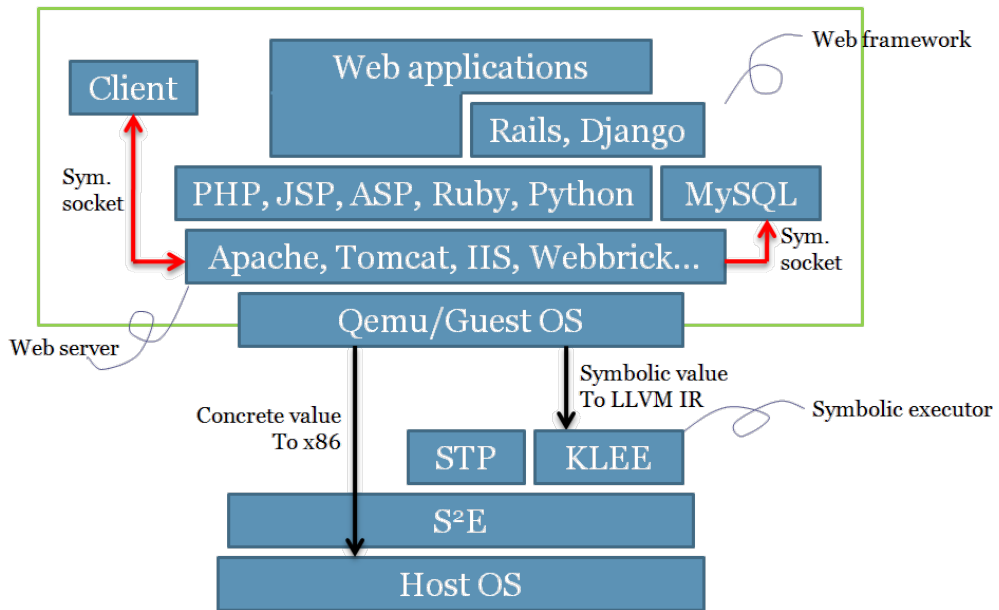


Figure 12: A simple architecture of S²E

Symbolic environment represents the existence and propagation of symbolic data in different environments, such as socket, file, argument, register and standard I/O. Due to symbolic execution on the whole system in S²E, a part of symbolic environment is already supported including symbolic socket. A sample code in Figure 13 demonstrates symbolic socket between client and server. The branch after reading *mesg* in client forks a new execution state because of the symbolic variable, *a*, which is tainted by the symbolic variable, *buf*, through symbolic socket.

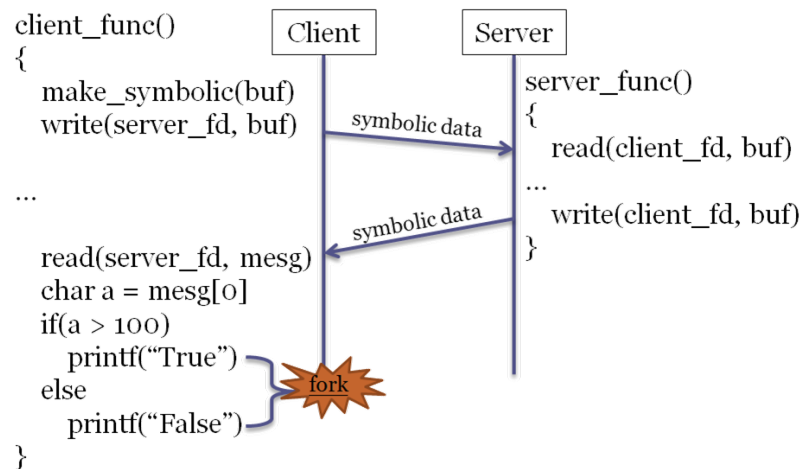


Figure 13: Sample code for symbolic socket between client and server

5.1.2 Architecture

The overall architecture for our automatic exploit generation is based on S²E and shows in Figure 14. In the figure, the red arrow represents the symbolic propagation through symbolic socket and solid blue block represents the main implementation part. *s2eget* and *s2e_myop* are the S²E instruction. The overall architecture is divided into three parts and explained detailedly in the following section.

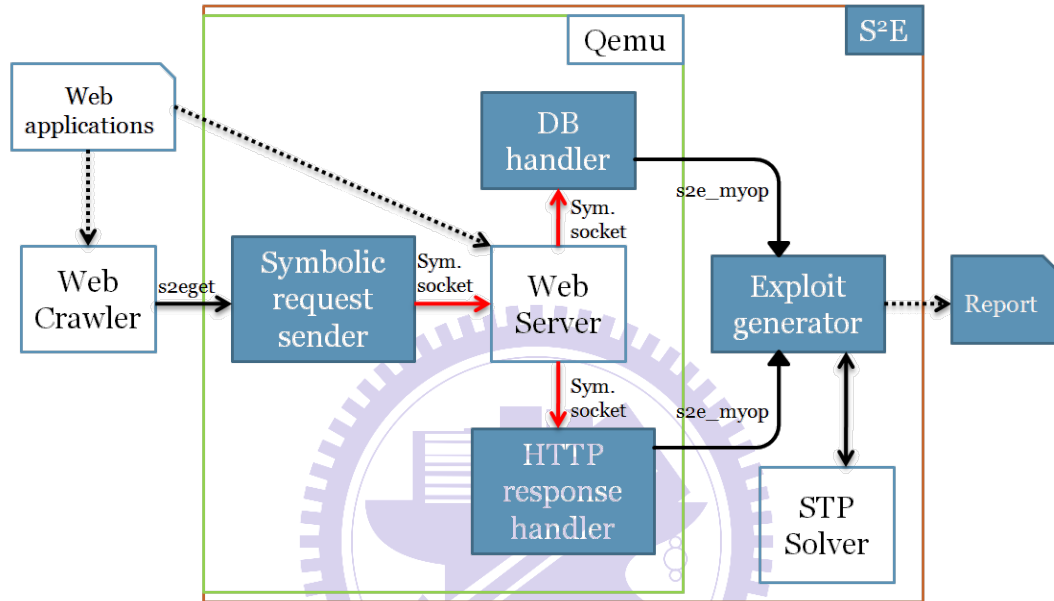


Figure 14: Overall architecture for automatic exploit generator

5.1.3 Symbolic Response and Query Handler

The concept of symbolic socket can be developed and applied to HTTP to perform symbolic execution on web applications and server. To cooperate with web crawler, all of the possible HTTP requests are crawled from web applications and send to guest OS by a built-in S²E instruction, *s2eget*. Each parameter in crawled HTTP requests is replaced by symbolic data. Then, a symbolic request is sent to web server through socket to perform symbolic execution on web applications and server together. The flow diagram shows in Figure 15. This is also the first part that is from web crawler to symbolic request in Figure 14.

Handlers are implemented and listened on port 80 and 3306 on LAMP (Linux, Apache, MySQL, PHP) architecture by default. During symbolic execution, handlers are ready to receive

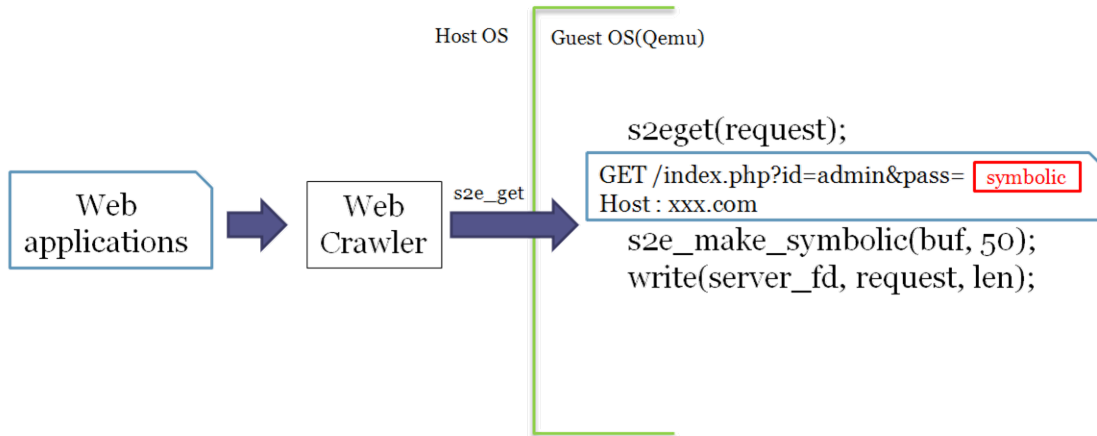


Figure 15: From web crawler to symbolic request

symbolic response and symbolic query respectively. The database handler is a modified version of MySQL. A new S²E instruction, *s2e_myop*, is created and built in handlers for transferring the received data directly from QEMU at guest OS to exploit generator at host OS. The received data are analyzed by exploit generator later. The flow diagram shows in Figure 16. This is the second part from a symbolic response or query to exploit generator in Figure 14. The final part is explained in the following section.

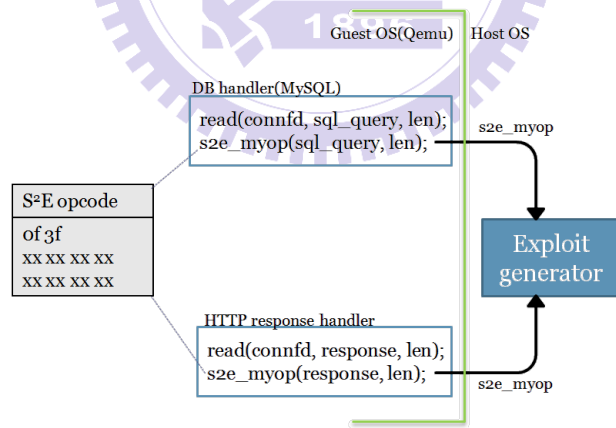


Figure 16: From symbolic response or query to exploit generator

5.2 Exploit Generation

Whenever the exploit generator is triggered by our customized S²E instruction, the received data, which is HTTP response or database query, is analyzed. To search a contiguous sym-

bolic data, an algorithm is shown in Algorithm 1. The received data are necessary to determine whether it is symbolic or concrete sequentially byte by byte. Until a contiguous symbolic data is located, it has to ensure that the length is long enough to involve the expected attack script, which is constructed by a simple HTML or SQL parser for the correct syntax. Thus, concerns that are mentioned in Section 4.2 is satisfied here.

Algorithm 1: Searching for contiguous symbolic data

Input: data : received HTTP response or database query

```

1 symbolic_len ← 0
2 for i ← 0 to length(data) do
3   if isByteSymbolic(data + i) then
4     symbolic_len ← symbolic_len + 1
5   else
6     if symbolic_len ≠ 0 then
7       expect_attack ← constructAttack(data, i)
8       if symbolic_len ≥ length(expect_attack) then
9         symbolic_data ← data + i - symbolic_len
10        solveExploit(symbolic_data, expect_attack)
11      end
12      symbolic_len ← 0
13    else
14      continue
15    end
16  end
17 end
18 expect_attack ← constructAttack(data)
19 if symbolic_len ≥ length(expect_attack) then
20   symbolic_data ← data + i - symbolic_len
21   solveExploit(symbolic_data, expect_attack)
22 end

```

Then, an algorithm in Algorithm 2 is used to solve the exploit. All constraints, which are collected during symbolic execution, are restored. Extra constraints are constructed and added byte-by-byte to restrict the result, emulating the expected attack script. Finally, an exploit may be solved as a solution that can reproduce the expected attack script. If constraints are unsolvable and no solution obtains, it will report as possible vulnerable instead of exploitable. Reason for unsolvable may be the restriction that mentions in Section 4.3.1 or the limitation of constraint solver itself[23].

Algorithm 2: Solving the exploit constraints

Input: `symbolic_data` : symbolic data, `expect_attack` : target attack string

Output: `exploit` : the solved exploit

```
1 backupConstraints()
2 for i ← 0 to length(expect_attack) do
3   tmp ← readMemory(symbolic_data + i)
4   constraint ← constructConstraint(tmp, expect_attack + i)
5   addConstraint(constraint)
6 end
7 exploit ← getSymbolicSolution()
```

5.2.1 Simple HTML and SQL parser

Common attack script, such as `<script>alert(document.cookie)</script>` for XSS or `' or 1=1--` for SQL injection, may not work for all cases of vulnerability due to the wrong syntax. To ensure the availability of the expected attack script, a simple HTML or SQL parser is necessary to construct the attack script in correct syntax.

By considering an example in Figure 17, a stack is used to maintain the status of HTML syntax, such as `<`, `>`, `"` and `'`, at anytime. Whenever symbolic data are discovered at HTTP response, `>` and `"` are already kept in stack at that time and popped to complete the expected attack script. So `"><script>alert(document.cookie)</script>` should be constructed for the expected attack script. The same concept can apply on the SQL parser.

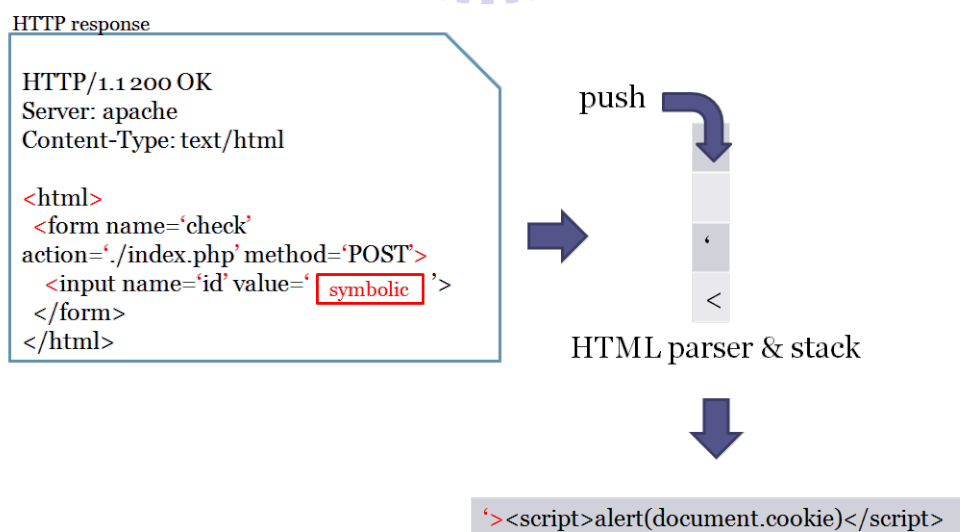


Figure 17: Completing the syntax of the expected attack script

5.3 Single Path Concolic Mode

The implementation of single path concolic mode has two parts. One is that the selection of branches and paths depend on a given concrete input. The other one is to keep branch conditions during symbolic execution and restore them at the later exploit generation. Before symbolic execution, the given concrete input is read and constructed as constraints. An example is constructed in Figure 18 for a concrete input, *AAAAAAAA*. A vector container, *concreteConstraints*, is used to store these constraints.

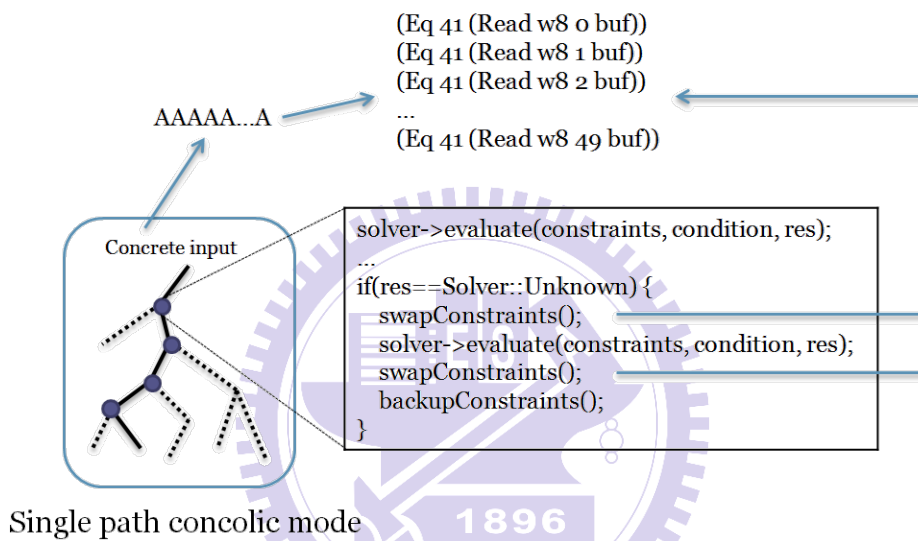


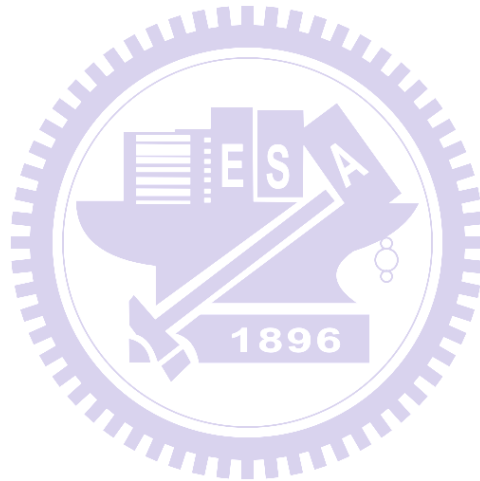
Figure 18: Concrete input constraints

Whenever a branch is encountered and its branch condition is evaluated by solver that is feasible for *true* and *false*, the current path constraints are swapped out and *concreteConstraints* are swapped in. Then, solver evaluates again with *concreteConstraints* to determine which direction of the branch should go dependent on the given concrete input. So the branch is fixed and S²E will not fork for two feasible paths anymore. Moreover, branch conditions here are also kept in a vector container, *backupConstraints*, which is restored at the later exploit generation process.

Single path concolic mode is an option to reduce the overhead on path explosion, but also with the restriction mentioned in Section 4.3.1. A option is implemented for switching between single path concolic mode and the original symbolic execution at any time conveniently.

5.3.1 Optimization

For single path concolic mode, the current path constraints are replaced by *concreteConstraints* to restrict and determine the selection of branches. In addition, *concreteConstraints* can also be used as somewhere that requires the current path constraints inside S²E, such as constraint solver evaluation on symbolic data. The overhead on solver can be reduced because the evaluation on solver prefers a simple concrete value rather than a complex symbolic value.



Chapter 6

Evaluation

In order to show our contribution and evaluate our work, two experiments have been reported in this chapter. The first one demonstrates and proves the feasibility of platform-independent web testing with our method. The second one is the experimental result of our automatic exploit generator. Parts of the test cases are the comparisons with *Ardilla*. The remaining test cases are the real-world web applications.

6.1 Experimental Environment

All experiments performed on a host hardware including a 2.4Ghz CPU with 8 cores, 8GB physical memory and host OS with Ubuntu 10.10 64-bit desktop edition. The guest environment that is emulated by Qemu includes 2.83GHz CPU with a single core, 128MB physical memory and guest OS with Debian 5.0.7 32-bit for Linux platform and Windows XP sp2 for windows platform. The software environment is based on S²E 1.0 and the database handler is based on MySQL 5.5.15.

6.2 Evaluation for Different Platforms

The first experiment evaluates a test case on different platforms to prove the feasibility of platform-independent web testing with our method. The test case is a simple web application that acquires a GET parameter from URL and prints it on a web page directly. Different platforms are based on five popular dynamic web programming languages including PHP, ASP, JSP, Ruby and Python. Ruby and Python may cooperate with their own framework together,

such as Rails[16] and Django[24].

The experiment attempts to perform symbolic execution in single path concolic mode with a given concrete input, which is a string with fifty *A*. Hypothetically, a symbolic response is detected and an exploit of XSS is generated by our automatic exploit generator. When the symbolic response is detected, the time spent during the overall execution is marked as *Symbolic response time* in Table 2. The result in PHP and Django reveals that it is feasible to generate the exploit in a short time. The experiment on Rails finished in minutes, but the exploit constraints cannot be solved because of the default security prevention mechanism. Moreover, the experiment on JSP finished, but a part of symbolic data is discontinued unexpectedly during the symbolic propagation inside JVM. Thus, the exploit cannot be solved because of the insufficient symbolic data. However, the experiment on ASP cannot complete in 12 hours because of the large-scale on program architecture or the complexity of program structure inside their kernel. This challenge may be optimized in the future.

By the way, we can test in another way of giving up all the collected constraints with single path concolic mode to speed up symbolic execution. Actually, the exploit cannot be generated finally without collecting constraints but it can still report as possible vulnerable for web applications after discovering the symbolic response. This strategy has the same effects as dynamic taint analysis. The time spent during the overall execution is marked as *Without constraints* in Table 2.

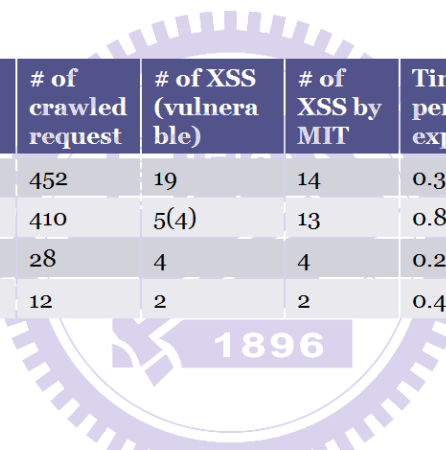
	PHP	JSP	Rails	Django	ASP
Framework	-	-	3.2	0.96.1	-
OS	Linux	Linux	Linux	Linux	Windows
Server	Apache-2.2.19	Tomcat-7.0.2	Webrick	Built-in	IIS-5.1
Kernel	PHP-5.3.6	JDK-7u2	Ruby-1.9.3	Python-2.6.6	ASP-3.0
Bind Port	80	8080	3000	8000	80
Symbolic response time	18.50s	6.72min	7.45min	32.72s	OT
Without constraints	16.42s	3.25min	5.62min	24.02s	OT

Table 2: Evaluation for different platforms

6.3 Evaluation for Exploit Generation

The second experiment reports the exploit generation on different web applications. All web applications are under single path concolic mode and a string with fifty *A* is fed as the given concrete input.

Web applications in Table 3 are the same test cases from *Ardilla*. The criterion in *Ardilla* for discovering new exploit is the different vulnerable line of code in PHP. And our criterion for discovering new exploit is the different path between each exploit that is generated by our exploit generator. Thus, the comparison in numbers of exploit between us and *Ardilla* may differ. *OT* is defined as over fifteen minutes during symbolic execution and exploit generation in *Time for all crawled request*.



Test Case	Line Of Code	# of crawled request	# of XSS (vulnerable)	# of XSS by MIT	Time per exploit	Time for all crawled request
Schoolmate-1.5.4	8,125	452	19	14	0.30min	107.78min + 30OT
Webchess-1.0.orc2	6,504	410	5(4)	13	0.80min	94.38min + 313OT
Faqforge-1.3.2	1,710	28	4	4	0.20min	5.74 min
EVE	904	12	2	2	0.42min	4.94min

Table 3: Evaluation for exploit generation with test cases from *Ardilla*

Web applications in Table 4 are real-world web applications. *SimpGB* is a simple PHP guestbook web application with vulnerabilities such as XSS, SQL injection and malicious file execution (MFE). It is a good benchmark for case studying. *DedeCMS* is a famous content management system (CMS) in China. The result of eleven generated exploits for *DedeCMS* came from a zero-day vulnerability that was found half a year ago. The built-in admin interface from old version *Django* are also vulnerable and the exploit of CVE-2008-2302[25] is generated in our result. The last two cases are *Discuz!* and *Joomla!*. *Discuz!* is an internet forum software written in PHP. It is the most popular internet forum program in China. *Joomla!* is a free and open source content management framework for publishing content on internet. Finally, our automatic exploit generator did not generate or find any exploit or vulnerability for these two cases.

Test Case	Line Of Code	Platform	# of crawled request	# of XSS (vulnerable)	Time per exploit	Time for all crawled request
SimpGB-1.49.02	41,296	PHP	1,299	33(57)	0.91min	7.67hr + 334OT
DedeCms-5.6	84,544	PHP	1,111	11(13)	0.48min	8.32hr + 9OT
Django-admin-0.96.1	3,558	Python	5	1	5.29min	5.29min + 4OT
Discuz!-6.0	67,088	PHP	613	0(1)	0.85min	8.37hr + 12OT
Joomla-1.6	253,711	PHP	215	0(7)	2.17min	1.26hr + 117OT

Table 4: Evaluation for exploit generation

Chapter 7

Conclusion and Future Work

In this chapter, we summarize the contribution and conclude the superiority and inferiority in our work. Some future work is proposed to explore more web security issues with similar methods in this thesis.

7.1 Conclusion

This thesis implemented an automatic exploit generator for web security issues on real-world web applications. Symbolic socket is an evolution of symbolic execution and it is an important idea in our work. In contrast with other related works, applying symbolic socket on HTTP is a comprehensive solution and provides the capability of platform-independent web testing.

Whenever a symbolic data is received under sockets, an exploit has an opportunity to be solved by constraint solving, which is an ability of constraint solver in symbolic execution. In contrast with other traditional vulnerabilities, this ability leads to generate more feasible exploits for potential vulnerability, which is under some arithmetic operations or simple mutations. This is one of our contributions in our work.

In order to apply our work on real-world web applications, single path concolic mode and some optimizations are proposed and implemented to overcome the challenge on large-scale applications. The objective of single path concolic mode is to force the exploration on symbolic execution in only one path with a given concrete input for reducing the overhead on path exploration.

In our evaluation, nine web applications include the benchmarks from *Ardilla* and real-world web applications at different platforms, such as PHP and Django. All applications were tested and exploits were generated by our automatic exploit generator. The experimental result proved the feasibility of our implementation. In addition, some of the exploits were announced as known vulnerabilities in CVE database.

7.2 Future Work

To develop our automatic exploit generator and become a more comprehensive solution in web security, future work is suggested as follows.

7.2.1 Other Web Security Issues

As mentioned in Section 2.2, other types of web security issue are also possible to generate the exploit with same method. By considering the exploit generation on RFI and LFI, vulnerability happens at particular functions, such as *require()*, *include()* in PHP platform. All implementations are in the same way except the handler, which should be implemented as a PHP extension and hook *require()* or *include()* inside PHP kernel for triggering the exploit generator and detecting symbolic data. By hooking different vulnerable functions that mention in Figure 19, it is possible to generate exploits for directory traversal attack, command injection or code injection. However, platform-dependent ways exist because of the PHP extension and particular functions in PHP kernel.

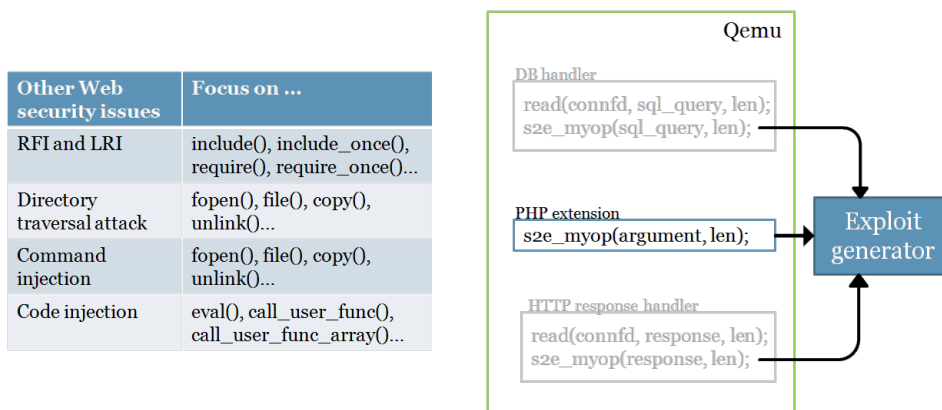


Figure 19: Exploit for other web security issues

7.2.2 Symbolic execution with Browser

To consider web security issues in Ajax or HTML5, our present method that mentions in Section 5.1.3 is not suitable for them. Because the issues happen at client-side rather than server-side and they should be determined at a browser rather than HTTP response. Thus, symbolic execution with browser is necessary to figure out those issues. The strategy is similar with Section 7.2.1. Handler including the new S²E instruction, *s2e_myop*, should be built in browser and triggers the exploit generator at later. The scenario shows in Figure 20.

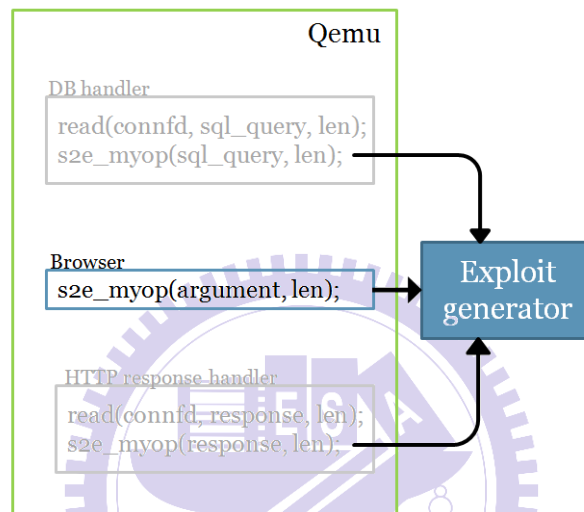


Figure 20: Symbolic execution with Browser

Reference

- [1] P. Bisht, T. Hinrichs, N. Skrupsky, and VN Venkatakrishnan. Waptec: whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 575–586. ACM, 2011.
- [2] M. Martin and M.S. Lam. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *Proceedings of the 17th conference on Security symposium*, pages 31–43. USENIX Association, 2008.
- [3] J.D. DeMott, R.J. Enbody, and W.F. Punch. Towards an automatic exploit pipeline. In *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for*, pages 323–329. IEEE, 2011.
- [4] J.C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [5] E.J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331. IEEE, 2010.
- [6] R.E. Fairley. Tutorial: Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, 1978.
- [7] OWASP. The ten most critical web application security risks. http://www.owasp.org/index.php/Top_10_2010-Main.
- [8] K. Sen, D. Marinov, and G. Agha. *CUTE: A concolic unit testing engine for C*, volume 30. ACM, 2005.

- [9] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *Computer Security Applications Conference, 21st Annual*, pages 9–pp. IEEE, 2005.
- [10] Zhe Chen, Xiao Juan Wang, and Xin Xin Zhang. Dynamic taint analysis with control flow graph for vulnerability analysis. *Instrumentation, Measurement, Computer, Communication and Control, International Conference on*, 0:228–231, 2011.
- [11] X. Fu and K. Qian. Safeli: Sql injection scanner using symbolic execution. In *Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, pages 34–39. ACM, 2008.
- [12] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M.D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 261–272. ACM, 2008.
- [13] A. Kiezun, 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.
- [14] 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*, pages 513–528. IEEE, 2010.
- [15] A. Chaudhuri and J.S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 585–594. ACM, 2010.
- [16] D.H. Hansson et al. Ruby on rails. *Website. Projektseite: <http://www.rubyonrails.org>*, 2009.
- [17] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224. USENIX Association, 2008.
- [18] Acunetix. Acunetix web crawler. <http://www.acunetix.com/>.

- [19] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.
- [20] F. Bellard. Qemu, a fast and portable dynamic translator. USENIX, 2005.
- [21] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.
- [22] V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, pages 519–531. Springer, 2007.
- [23] C. Barrett, L. De Moura, and A. Stump. Smt-comp: Satisfiability modulo theories competition. In *Computer Aided Verification*, pages 503–516. Springer, 2005.
- [24] J. Forcier, P. Bissex, and W. Chun. *Python web development with Django*. Addison-Wesley Professional, 2008.
- [25] The MITRE Corporation. Common vulnerabilities and exposures. <http://cve.mitre.org/cve/>.

