A Generic Web Application Testing and Attack Data Generation Method

Hsiao-Yu Shih[1], Han-Lin Lu[1], Chao-Chun Yeh[1,3], Hsu-Chun Hsiao[4], and Shih-Kun Huang[1,2(✉)]

[1] Department of Computer Science, National Chiao Tung University, Hsinchu 300, Taiwan
ncnoa221@gmail.com, littleflyer2015@gmail.com, skhuang@cs.nctu.edu.tw

[2] Information Technology Service Center, National Chiao Tung University, Hsinchu 300, Taiwan

[3] Computational Intelligence Technology Center, Industrial Technology Research Institute, Hsinchu 300, Taiwan
avainyeh@itri.org.tw

[4] Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan
hchsiao@csie.ntu.edu.tw

Abstract. With the advances of diversified online services, there is an increasing demand for web applications. However, most web applications contain critical bugs affecting their security, allowing unauthorized access and remote code execution. It is challenging for programmers to identify potential vulnerabilities in their applications before releasing the service due to the lack of resources and security knowledge, and thus such hidden defects may remain unnoticed for a long time until being reported by users or third-party risk exposure. In this paper, we develop an automated detection method to support timely and flexible discovery of a wide variety of vulnerability types in web applications. The key insight of our work is adding a lightweight *detecting sensor* that differentiates attack types before performing *symbolic execution*. Based on the technique of symbolic execution, our work generates testing and attack data by tracking the address of program instruction and checking the arguments of dangerous functions. Compared to prior analysis tools that also use symbolic execution, our work flexibly supports the detection of more types of web attacks and improve system flexibility for users thanks to the detecting sensor. We have evaluated our solution by applying this detecting process to several known vulnerabilities on open-source web applications and CTF (Capture The Flag) problems, and detected various types of web attacks successfully.

Keywords: Web application testing · Symbolic execution · Capture The Flag
Software vulnerability

1 Introduction

Web applications have become a significant part of the Web because of the attractive features such as easy installation, customization and high accessibility. However, they are often deployed with critical software bugs that can be maliciously exploited. Once

S.-L. Peng et al. (Eds.): SICBS 2017, AISC 733, pp. 232–247, 2018.
https://doi.org/10.1007/978-3-319-76451-1_22

a weakness is found, it can be exploited to take control of the system. Hence, there is a need for an analysis tool which can automatically detect vulnerabilities and defend against threats.

1.1 Motivation and Objective

Web applications are usually built with multiple utilities for customers in various programming languages. Our previous work, CRAXWeb [1], achieves the goal of detecting XSS and SQL injection attack and generating the corresponding exploit [2–4]. However, many types of attack remain unsupported. We propose to add an attack type differentiator called detecting sensor in a designated call site of the Web service. The complex step of inserting a detecting sensor into the web service components also decreases the flexibility in the testing process. Therefore, we improve CRAXWeb to support the detection of more attack types and speed up the procedure to deploy detecting sensors.

The aim of this work is to extend an existing exploit generator, CRAXWeb, for XSS and SQL injection attack on web applications to implement generic web attack generation. This work is based on a popular dynamic analysis technique in the field of software testing called symbolic execution [5, 6]. Many related works are also based on it. However, most of the works focus on only XSS and SQL injection attack. Our challenge is to protect web applications against multiple types of threats.

1.2 Overview

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

2 Background

2.1 Symbolic Execution

Symbolic execution is a testing approach that executes programs with symbolic rather than concrete inputs. Its objective is to explore as many paths in a program as possible. 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 as symbolic. When symbolic execution encounters a branch condition, it forks the execution state, following both branch directions and updating the corresponding path constraints on the symbolic input. If the program exits, or terminates unexpectedly, the current path constraint will be solved to compute a concrete test case that drives the program to this execution point.

By considering symbolic execution on the example in Fig. 1, `num` is assigned a symbol X at line 4 since it is user provided data. For the assignment at line 6, the symbolic variable `num` taints the concrete variable `key` and the symbolic value of `key` becomes

X - 100. For the branch at line 7, the execution encounters the symbolic variable `key` and forks a new execution for another new path. One takes the true path with an additional constraint *X - 100 == 0*. The other takes the false path, with an additional constraint *X - 100* $\neq$ *0*. Whenever two forked executions finish, their path constraints can be solved by the constraint solver for generating new test cases. One case is *num* = 100 and another case is *num* = 101(not equal to 100). The process is shown in Figs. 1 and 2

```
1  void vul()
2  {
3      int num, key = 0;
4      scanf("%d", &num);
5
6      key = num - 100;
7      if(key == 0)
8          printf("unsafe");
9      else
10         printf("safe");
11 }
```

Fig. 1. Sample code

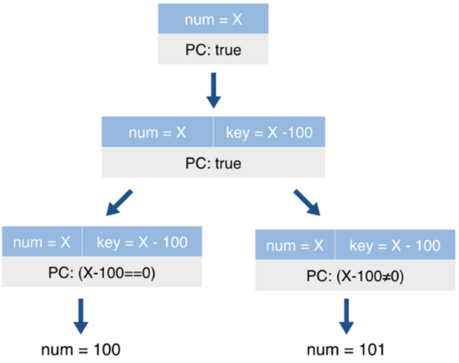

Fig. 2. Symbolic execution for sample code

2.2 Web Security Issues

Cross Site Scripting (XSS)
Cross Site Scripting (XSS) is a common attack vector that injects code into website to complete a range of actions from stealing logs to installing malicious software on user's computer. Two primary types of XSS are reflected XSS and stored XSS. In reflected XSS, an attacker crafts a URL containing a malicious string and sends it to the victim. Whoever clicks the link is going to have the script execute in the browser. In stored XSS, also known as persistent XSS, an attacker injects the malicious payload into databases or visitor logs and has it be available to all visiting users, the payload will run in each of the victims' browsers.

Cross-Site Request Forgery (CSRF)
Cross-site request forgery (CSRF) is a type of website exploit caused by transmitting unauthorized commands from a user whom the web application trusts. As opposed to XSS, which exploits the user's trust for a website, CSRF exploits a website's trust for a particular user's browser. CSRF attack forces a logged-on victim's browser to send a forged HTTP request, which allows the attacker to launch any desired requests against the website, without the website being able to distinguish whether the requests are legitimate or not.

SQL Injection
An SQL injection vulnerability occurs when the data from an untrusted source are used to dynamically construct a SQL query. Attackers trick the system by executing malicious

SQL commands to manipulate the backend database [7]. An SQL injection attack may result in data theft, loss, modification or corruption, or even complete takeover of the server.

Command Injection
Command injection occurs when an application passes unsafe user supplied data to a system shell. An attacker can execute operating system commands with the privileges of the vulnerable application. There are many ways to exploit a command injection, such as by injecting the command inside backticks (for example `` `ls` ``) and running another command if the first one succeeds (for example `&&ls`).

File Inclusion
A file inclusion vulnerability occurs when a user-controlled parameter is used as part of a file name in a call to an including function (`require()`, or `include()` in PHP for example). Depending on whether the file is remote or local, file inclusion can be categorized into Remote File Inclusion (RFI) and Local File Inclusion (LFI).

Remote File Inclusion allows an attack to include and execute a remotely hosted file. Since the included file is controllable, the attack can run arbitrary code either on the client side or on the server. In the scenario of Local File Inclusion, the attacker can access unauthorized files or utilize directory traversal characters to retrieve sensitive files available in other directories.

3 Related Work

This section presents a comprehensive survey of previous work undertaken in the field of testing and vulnerability analysis for web scripting languages. There are three main topics: Symbolic Execution based Test Generation, Static/Dynamic Analysis based Attack Detection, and Symbolic Execution based Attack Detection. Symbolic Execution based Test Generation.

Apollo [8] is a tool that uses concrete and symbolic (concolic) execution to generate failure-inducing inputs for PHP web applications. *Jalangi* [9] is a dynamic analysis framework for JavaScript that applies concolic execution to generate function arguments. *SymJS* [10] contains a symbolic execution engine for JavaScript and an automatic event explorer. *Jalangi* works for pure JavaScript programs, while *SymJS* works for general web applications. *Derailer* [11] is a security bugs finding tool for Ruby on Rails web applications. *Chef* [12] is a symbolic execution engines relying on the S^2E [13] for Python and Lua analysis. *MultiSE* [14] extends *Jalangi* and introduces a new technique for merging states to improve its effectiveness.

3.1 Static/Dynamic Analysis Based Attack Detection

Pixy [15] performs interprocedural flow-sensitive data flow analysis to detect XSS vulnerabilities in PHP scripts. *XSS-GUARD* [16] dynamically learns the set of scripts that a web application intends to create for any HTML request. *PIUIVT* [17] enhances the efficiency of invalid test inputs generation depending on feedback of analysis.

MySQLInjectior [18] is a web scanning tool to detect SQL injection vulnerabilities based on the identified styles. *NKSI Scan* [19] is a model based penetration test method for the SQL injection vulnerabilities. It can generate test case covering different types and patterns of SQL injection attack input. *Zheng et al.* [20] propose a path- and context-sensitive interprocedural analysis to detect remote code execution vulnerabilities on PHP applications. *XSSDM* [21] proposes a context-sensitive approach based on static taint analysis and pattern matching techniques to detect XSS vulnerabilities. *Joza* [22] is a hybrid approach which combines advantages of negative [23] and positive inference [24] for SQL injection detection. *DEKANT* [25] uses static analysis with the ability to learn to characterize vulnerabilities based on annotated source code slices.

3.2 Symbolic Execution Based Attack Detection

SAFELI [26] inspects Java to automatically generate SQL injection attack scenarios. *Adrilla* [27] is an exploit generator which stems from Apollo. It combines concolic testing and dynamic taint analysis to generate concrete attack vectors for PHP web applications. *Kudzu* [28] is the first symbolic execution based framework for JavaScript code analysis. It uses attack grammars to solve the exploit and finally finds out two unknown vulnerabilities. *Rubyx* [29] is a symbolic executor for Ruby, with builtin support for specification and verification of scripts. It proves complex security and correctness properties of Ruby-on-Rails web applications. *Huang et al.* [30] proposed a hybrid vulnerability analyzer for Java that applies symbolic execution to generate path constraints. *Codeminer* [31] combines static analysis and symbolic code execution to identify XSS and SQL injection vulnerabilities on PHP web applications. Compared to these works, our framework can detect XSS and SQL injection attack for the web applications written in any language. Moreover, for PHP web applications, our framework detects more types of attacks such as command injection, code injection, and file inclusion.

4 Method

Our work is based on the Selective Symbolic Execution (S^2E) [13] framework, which supports application emulation using QEMU. Figure 3 is the model of our method, which is divided into four main parts: Symbolic Environment, Dangerous Function Analysis, Symbolic Argument Checking, and Host Management.

The Guest OS comprises a client and a server which runs one or more back-end web applications and a database, working like the real-world web service environment. The only difference is the way client communicates with server. Client sends symbolic data to server along with HTTP requests. The Host OS keeps track of the program counter during the request processing. When the addresses of dangerous functions are reached, it will check whether the arguments of the functions are symbolic. The Host Management contains a list of functions we are interested in and a configuration file used to control the symbolic execution in the Host OS.

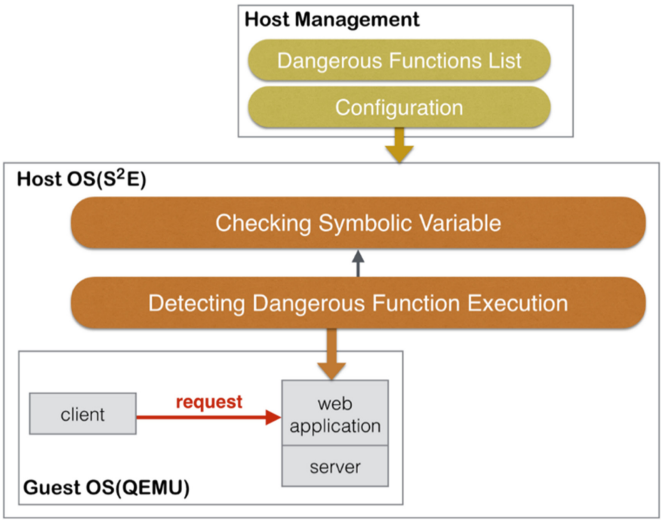

Fig. 3. Symbolic environment, dangerous function analysis, symbolic argument checking, and host management

4.1 Symbolic Environment

Symbolic Socket

To attack web applications, an attacker inject unexpected inputs to invoke abnormal program execution and reaction, such as system crash and sensitive data leakage. These malicious inputs are propagated within HTTP request, in the form of GET parameters in URL or POST data in message body. For SQL injection, the single quotation mark and the UNION operator are commonly leveraged to craft a malicious query, which is joined into the original query intended to be run by the web application. For command injection, the crafted command which is used as the arguments of dangerous functions will be processed on the operating system.

In order to explore all possible paths through the whole web executing procedure from request to response that correspond to all possible inputs, we have to make these inputs symbolic. Hence, we adopt symbolic socket, which is composed of HTTP request and symbolic data, to act as the communicator between the server and the client. The symbolic data is injected into HTTP request to replace the value of original inputs from users and passed to the web server along with the HTTP request. If the symbolic data can reach the functions we are interested in, it implies that the arguments of these functions can be controlled by the original symbolic data. Figure 4 shows the propagation of symbolic data.

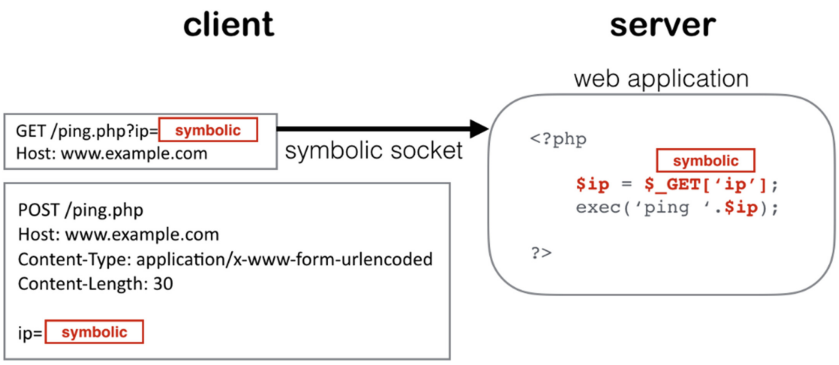

Fig. 4. Symbolic data propagates to the server along with HTTP request

4.2 Dangerous Function Analysis

Target Function Detection

To facilitate web security testing, our technique should be able to detect as many types of vulnerabilities as possible. Because most of the vulnerabilities occur when untrusted data is passed into and executed by functions in web application languages, what we care about in this detecting method is the address of such dangerous functions. S^2E is extensible to support different architectures and features by means of a plugin interface. We implement a customized plugin in the Host OS to monitor the address that the symbolic data flows through the programs in the Guest OS during the symbolic execution.

In order to fulfill the goal of detecting SQL injection attack in web applications which are written in different programming languages, it is essential to analyze the query processing in the database server. The `dispatch_command()` function in MySQL source code is where MySQL actually starts the analysis of commands, including queries, prepared statements and other command types. Figure 5 shows part of the code in `dispatch_command()`. As the name of the function implies, it is responsible for dispatching the query to the appropriate handler. Since the SQL query run by a web

```
bool dispatch_command(enum enum_server_command command, THD *thd,
              char* packet, uint packet_length)
{
    switch (command) {
        case COM_INIT_DB:
        // …
        case COM_QUERY:
        {
          if (alloc_query(thd, packet, packet_length))
            break;                    // fatal error is set

          // …

          mysql_parse(thd, thd->query(), thd->query_length(), &parse_state);

          // …
        }
```

Fig. 5. MySQL function dispatch_command()

application is a standard SQL query over the connection, we focus on the *COM_QUERY* block in the switch statement. If the block is reached during symbolic execution, we can continuingly check whether the query string (i.e. *thd-> query()*) is controllable.

For other types of attack such as command injection, code injection, file inclusion and more, we target at the functions in PHP since it is the most widely used web application programming language and has raised substantial number of security issues due to the improper use of functions. Take command injection for example, one of our target functions is `shell_exec()`, which allows users to execute an external program. Figure 6 shows the `shell_exec()` function in *ext/standard/exec.c* of PHP 5.5.38. The char pointer, *char *command*, is the command that will be executed as well as the argument that we have to check.

Symbolic Argument Checking

If the recorded functions are reached, we have to continuingly verify the controllability of the arguments. S^2E fetches blocks of guest code, translates them to the host's instruction set, and passes the resulting translation to the execution engine. It determines which code to fetch and translate by reading the state of the virtual CPU and the guest memory.

Function Argument Checking.

It is time-consuming to insert s2e opcode to PHP or MySQL source code and recompile the binaries.

To deal with this problem, we design a method to analyze the program state at the Host OS when S^2E notices that a dangerous function is executed instead of invoking the checking process from the Guest OS. This reduces the manipulation on web service components, leaving the web environment easy to deploy. The web server, database,

```
PHP_FUNCTION(shell_exec)
{
    FILE *in;
    char *command;
    size_t command_len;
    zend_string *ret;
    php_stream *stream;

    ZEND_PARSE_PARAMETERS_START(1, 1)
        Z_PARAM_STRING(command, command_len)
    ZEND_PARSE_PARAMETERS_END();

#ifdef PHP_WIN32
    if ((in=VCWD_POPEN(command, "rt"))==NULL) {
#else
    if ((in=VCWD_POPEN(command, "r"))==NULL) {
#endif
        php_error_docref(NULL, E_WARNING, "Unable to execute '%s'", command);
        RETURN_FALSE;
    }
    //…
}
/* VCWD_POPEN will finally call popen() in C standard library */
```

Fig. 6. PHP function shell_exec()

and programming language can be built through a package manager like pip or apt-get without hardcoding other functions needed by symbolic execution.

The method to analysis function arguments is divided into two steps: reading the memory address from the register where the function argument stores and determining whether the value in the memory is symbolic or concrete. Figure 7 shows an overview of the workflow.

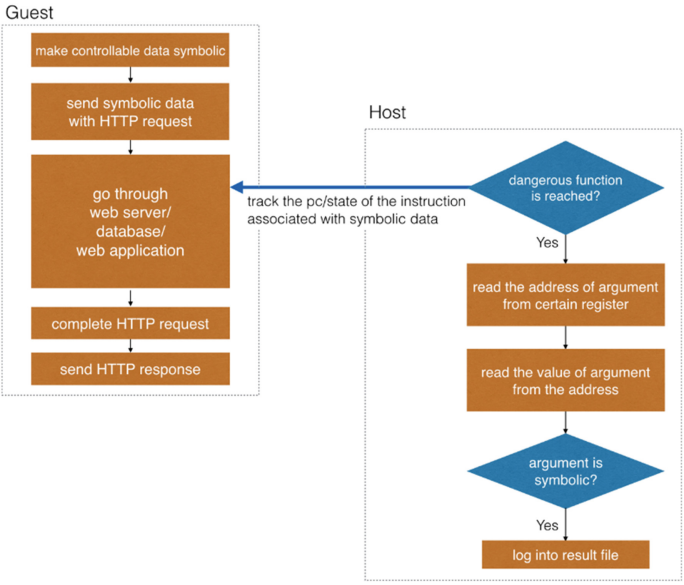

Fig. 7. The overview of the workflow

4.3 Host Management

Dangerous Function List

In order to reduce the complexity of processes for users, we build a list of functions and their corresponding attack types. Users can choose what kind of detection they want to implement on the web application by writing a python script to specify the attack types or functions that are of interest. There is no need for users to consider the addresses of the function and the argument. Figure 8 is the example of the python script. In this example, although the `file_get_contents()` function does not belong to any of the two attack types, it is also the target function.

```
1 attack_type = { "SQL injection", "Command injection" }
2 function = { "file_get_contents" }
```

Fig. 8. Python script example that is provided by user

5 Implementation

In this section, we explain in detail how our method is implemented on S^2E.The first part is Symbolic Environment including the whole system architecture of our framework and the propagation of symbolic data. The second part is Host OS Risk Detection; it relates to the plugin we design to detect function execution and identify symbolic arguments. The third part is Host Management; it works as the communicator between users and S^2E.

5.1 Symbolic Environment

System Architecture

The architecture of the system is summarized in Fig. 9: the web application is built in the Guest OS, which is running on a machine emulator called QEMU. The Host OS constructed by S^2E receives the information propagated from the Guest OS, and then our customized S^2E plugin can verify whether the dangerous functions are reached and whether the function arguments are symbolic. Users can specify the functions or attack types in a python script. The configuration writer creates the configuration file depending on the script and the dangerous function list to control the plugin.

Most web applications are based on the client-server architecture where the client submits data while the server stores and retrieves data. We make the testing application run on top of Debian 7 with Apache as the web server and MySQL as the database in light of flexibility and accessibility of use.

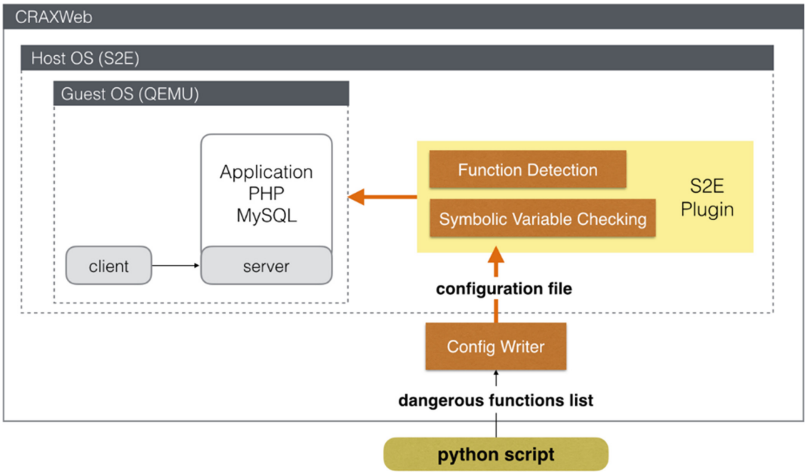

Fig. 9. System Architecture

Symbolic Socket

We deploy a program acting as the client in the Guest OS to generate symbolic socket, which is made up of a HTTP request with injected symbolic data. Then, the HTTP

request message with the symbolic string is written to the socket and sent to the web server, as shown in Fig. 10.

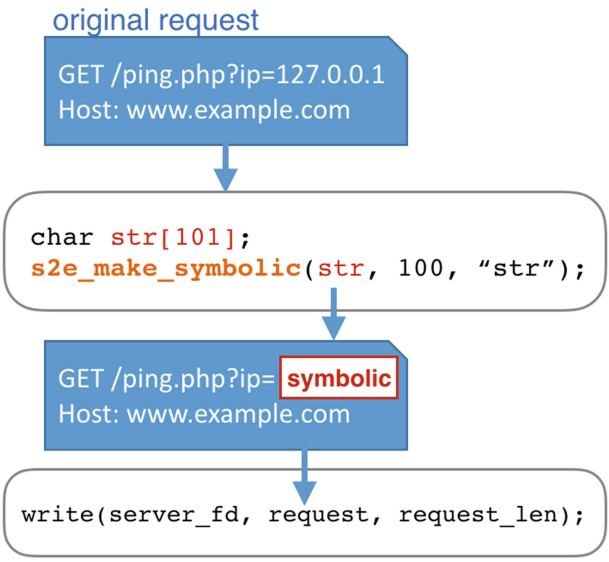

Fig. 10. Generate a symbolic socket

Whenever a branch referring to symbolic data is encountered, the entire states (i.e. memory, registers and PC) will be forked and each side of the branch will be explored by S^2E.

5.2 Dangerous Function Analysis

In order to detect multiple types of attacks, we focus on the execution of the functions that might be used by attackers with malicious intentions. In this section, we detail the analysis process of finding function address and the implementation of checking symbolic variable in our customized plugin.

Target Function Detection

- **MySQL Executable Analysis**

MySQL supplies different executables to serve different purposes for users. For example, *mysql* is a command-line client for executing SQL statement interactively and *mysqld* is the server executable. Since the symbolic data is sent to the web server along with HTTP request from the client-side, the target to analyze is *mysqld*, which is the server daemon in the Unix-like operating system.

By reversing *mysqld* with the IDA Pro Disassembler, we can find the address of `dispatch_command()` function, which is the entry point of MySQL query.

In addition to the address of the function, the value of the argument is also necessary for checking symbolic variable. S^2E keeps track of the instruction executed in the Guest OS with the corresponding state. We observe the disassembly code of `dispatch_command()` and find the register where the argument is stored.

- **PHP Module Analysis**

Using the PHP module to execute PHP scripts on Apache is the default mode set at the creating phase of most web frameworks. The PHP module acts as the PHP interpreter that is embedded in each Apache processes, which means that no external PHP process is required. As the interpreter is started along with Apache, it can cache certain information and need not to repeat the same tasks each time a script is executed, leading to the good performance on PHP heavy sites.

Apache loads numerous modules when the service starts, including the PHP module which is named *libphp5.so*. In order to get the base address of the PHP module inside the Apache process, we use *pmap* command in Linux which can report memory map of a process to list the information related to *libphp5.so*.

Then we reverse *libphp5.so* in the Apache process to get the starting address and length of functions such as zif_shell_exec() or shell_exec() in PHP.

Symbolic Argument Checking

S^2E provides macros to access the registers and memory of S^2E-specific state. To get the contents in registers, we use `readCpuRegisterConcrete()` macro to read concrete value from general purpose CPU register. There are two macros to read the content in memory according to the status of the target memory: `readMemoryConcrete()` gets concrete data, if the target memory is concrete status; `readMemory8()` gets symbolic data, if the target memory is symbolic status. However, `readMemoryConcrete()` fails if the value is symbolic. Since the argument has chance to be symbolic, we use `readMemory8()` to read the content from memory and then cast it into the `KLEE Expression`. S^2E uses the `KLEE Expression` class as the fundamental building block of all values in the emulated memory object. A concrete value is merely a `KLEE ConstantExpression` which is derived from `KLEE Expression`. We determine a value is symbolic if its cast expression is not a `KLEE ConstantExpression`.

5.3 Host Management

Dangerous Function List

The dangerous function list is written in JSON format. It contains numerous function names as the keys and each of them has the following components: the function address, the offset of the register which stores the function argument, and the attack type. S^2E uses the predefined offset value to access each register from QEMU. Figure 11 is the example of the function list.

```
"dispatch_command" : {
    "address" : 0x8228296,
    "argument" : 0x0,
    "type" : "SQL injection",
},

"shell_exec" : {
    "address" : 0xb5c672b4,
    "argument" : 0x0,
    "type" : "Command injection"
}
```

Fig. 11. The example of the function list

6 Evaluation

6.1 Evaluation of Vulnerable Applications

Experimental Environment
All experiments performed on a host hardware including a 2.4 Ghz CPU with 8 cores, 8 GB physical memory and host OS with Ubuntu 12.04 64-bit desktop edition. The guest environment that is emulated by QEMU includes 2.8 GHz CPU with a single core, 128 MB physical memory and guest OS with Debian 7 32-bit for Linux platform. The software environment is based on S^2E 1.0. The database handler is based on MySQL 5.5.49 and the PHP version is 5.5.38.

Experimental Results
The experiment reports the vulnerabilities detection on different platforms to prove the feasibility of platform-independent web testing with our method. Test 1 is a PHP web service that contains numbers of dangerous functions. Test 2 is a website with SQL injection vulnerability and it is built on a Python web framework called Flask. mfw is a challenge of CSAW online CTF in 2016. The forth test case is the web services of RCTF final attack-and-defense contests in 2015; it is built on Codeigniter and with various types of vulnerabilities. The fifth, sixth, and seventh test cases are both the plugin of Wordpress and have been recorded in the CVE list. Table 1 shows the experimental result of vulnerability detection.

Table 1. Evaluation of vulnerable applications

Test case	Attack types	Detected functions	# of lines	Testing time (sec)	Platform	CVE
Test 1	SQLi, Commandi, LFI	mysql_query, system, shell_exec, assert, fopen	55	102.06	PHP	
Test 2	SQLi	MySQL–dispatch_commend	36	31.66	Python (Flask)	
Test 3	Code injection	assert	62	6.25	PHP	
Test 4	SQLi, Code injection	create_function, unserialize	44553	34.59/41.7	PHP (Codeigniter)	
Test 5	Commandi	shell_exec	23086	33.53	PHP	2015-5227
Test 6	Code injection	call_user_func	5377	60.19	PHP	2014-1215
Test 7	Path traversal	file_get_contents	2264	48.58	PHP	2014-5368

*Test 3: mfw (CSAW CTF 2016 web 125), Test 4: RCTF Final 2015, Test 5: Landing Pages (WordPress plugin), Test 6: Download Manager (WordPress plugin), Test 7: wp-source-control (WordPress plugin)

7 Conclusion and Future Work

The aim of this work is to extend an existing dynamic analysis framework to implement automatic attack detection for web framework. By detecting the execution of dangerous functions, developers can figure out potential vulnerabilities before releasing the web service. This means that software flaws can be fixed early on, and that developers can complete quick security audits.

Our work fulfills the goal of multiple types of web attacks, and has implemented the testing procedure on web applications that are built on different framework and written in different programming languages. The experimental result proved the feasibility of our implementation. In addition, some of the test cases were announced as known vulnerabilities in the CVE database.

Our work can automatically detect SQL injection and XSS attack and generate corresponding exploit string. A SQL injection exploit payload possibly contains the string such as "*'or 1 = 1;–*", so we set the exploit generator to solve the constraints to generate an input that formed by the basic exploit string. Thanks to our generic construction, it is also possible to generate exploits for other types of web security issues with the same method. By considering the exploit generation on code injection, when the symbolic data reaches the `eval()` or `assert()` functions, exploit generator can continuingly generate the exploit string that formed by "`system('ls')`".

Acknowledgements. This work was supported by the Institute for Information Industry under the grant 106-EC-17-D-11-1502.

References

1. Huang, S.-K., Lu, H.-L., Leong, W.-M., Liu, H.: Craxweb: automatic web application testing and attack generation. In: 2013 IEEE 7th International Conference on Software Security and Reliability (SERE), pp. 208–217. IEEE (2013)
2. Bisht, P., Hinrichs, T., Skrupsky, N., Venkatakrishnan, V.: WAPTEC: whitebox analysis of web applications for parameter tampering exploit construction. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 575–586. ACM (2011)
3. Martin, M., Lam, M.S.: Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In: Proceedings of the 17th Conference on Security Symposium, pp. 31–43. USENIX Association (2008)
4. Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., Brumley, D.: Automatic exploit generation. Commun. ACM **57**(2), 74–84 (2014)
5. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
6. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 2010 IEEE Symposium on Security and Privacy (SP), pp. 317–331. IEEE (2010)
7. Halfond, W.G., Viegas, J., Orso, A.: A classification of SQL-injection attacks and countermeasures. In: Proceedings of the IEEE International Symposium on Secure Software Engineering, vol. 1, pp. 13–15. IEEE (2006)
8. Artzi, S., et al.: Finding bugs in dynamic web applications. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis, pp. 261–272. ACM (2008)
9. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 488–498. ACM (2013)
10. Li, G., Andreasen, E., Ghosh, I.: SymJS: automatic symbolic testing of JavaScript web applications. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 449–459. ACM (2014)
11. Near, J.P., Jackson, D.: Derailer: interactive security analysis for web applications. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 587–598. ACM (2014)
12. Bucur, S., Kinder, J., Candea, G.: Prototyping symbolic execution engines for interpreted languages. ACM SIGARCH Comput. Archit. News **42**(1), 239–254 (2014)
13. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: a platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Not. **46**(3), 265–278 (2011)
14. Sen, K., Necula, G., Gong, L., Choi, W.: MultiSE: multi-path symbolic execution using value summaries. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 842–853. ACM (2015)
15. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: a static analysis tool for detecting web application vulnerabilities. In: 2006 IEEE Symposium on Security and Privacy, pp. 258–263. IEEE (2006)
16. Bisht, P., Venkatakrishnan, V.: XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 23–43. Springer (2008)
17. Li, N., Xie, T., Jin, M., Liu, C.: Perturbation-based user-input-validation testing of web applications. J. Syst. Softw. **83**(11), 2263–2274 (2010)
18. Ali, A.B.M., Abdullah, M.S., Alostad, J.: SQL-injection vulnerability scanning tool for automatic creation of SQL-injection attacks. Procedia Comput. Sci. **3**, 453–458 (2011)

19. Tian, W., Yang, J.-F., Xu, J., Si, G.-N.: Attack model based penetration test for SQL injection vulnerability. In: 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops (COMPSACW), pp. 589–594. IEEE (2012)
20. Zheng, Y., Zhang, X.: Path sensitive static analysis of web applications for remote code execution vulnerability detection. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 652–661. IEEE Press (2013)
21. Gupta, M.K., Govil, M.C., Singh, G., Sharma, P., XSSDM: towards detection and mitigation of cross-site scripting vulnerabilities in web applications. In: 2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp. 2010–2015. IEEE (2015)
22. Naderi-Afooshteh, A., Nguyen-Tuong, A., Bagheri-Marzijarani, M., Hiser, J.D., Davidson, J.W.: Joza: hybrid taint inference for defeating web application SQL injection attacks. In: 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 172–183. IEEE (2015)
23. Sekar, R.: An efficient black-box technique for defeating web application attacks. In: NDSS (2009)
24. Nguyen-Tuong, A., et al.: To B or not to B: blessing OS commands with software DNA shotgun sequencing. In: 2014 Tenth European Dependable Computing Conference (EDCC), pp. 238–249. IEEE (2014)
25. Medeiros, I., Neves, N., Correia, M.: DEKANT: a static analysis tool that learns to detect web application vulnerabilities. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 1–11. ACM (2016)
26. Fu, X., Qian, K.: SAFELI: SQL injection scanner using symbolic execution. In: Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, pp. 34–39. ACM (2008)
27. Kieyzun, A., Guo, P.J., Jayaraman, K., Ernst, M.D.: Automatic creation of SQL injection and cross-site scripting attacks. In: IEEE 31st International Conference on Software Engineering, ICSE 2009, pp. 199–209. IEEE (2009)
28. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: 2010 IEEE Symposium on Security and Privacy (SP), pp. 513–528. IEEE (2010)
29. Chaudhuri, A., Foster, J.S.: Symbolic security analysis of ruby-on-rails web applications. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 585–594. ACM (2010)
30. Huang, Y.-Y., Chen, K., Chiang, S.-L.: Finding security vulnerabilities in Java Web applications with test generation and dynamic taint analysis. In: Proceedings of the 2011 2nd International Congress on Computer Applications and Computational Science, pp. 133–138. Springer (2012)
31. Agosta, G., Barenghi, A., Parata, A., Pelosi, G.: Automated security analysis of dynamic web applications through symbolic code execution. In: 2012 Ninth International Conference on Information Technology: New Generations (ITNG), pp. 189–194. IEEE (2012)