2013 7th International Conference on Software Security and Reliability

CRAXweb: Automatic Web Application Testing and Attack Generation

Shih-Kun Huang*[†],Han-Lin Lu[†], Wai-Meng Leong[†],Huan Liu[†]
*Information Technology Service Center,[†]Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan
{skhuang,luhl,wmliang,hliu}@cs.nctu.edu.tw

***Abstract*—This paper proposes to test web applications and generate the feasible exploits automatically, including cross-site scripting and SQL injection attacks. We test the web applications with initial random inputs by detecting symbolic queries to SQL servers or symbolic responses to HTTP servers. After symbolic outputs detected, we are able to generate attack strings and reproduce the results, emulating the manual attack behavior. In contrast with other traditional detection and prevention methods, we can determine the presence of vulnerabilities and prove the feasibility of attacks. This automatic generation process is based on a dynamic software testing method-symbolic execution by S^2E. We have applied this automatic process to several known vulnerabilities on large-scale open source web applications, and generated the attack strings successfully. Our method is web platform independent, covering PHP, JSP, Rails, and Django due to the supports of the whole system environment of S^2E.**

***Index Terms*—Web security; Symbolic execution; Automatic exploit generation.**

I. INTRODUCTION

Web applications are the primary services in the Internet. However, they also bring about various security issues. Most of the issues are caused by the input from web pages, such as user data from the HTML forms and cookies. Practically, some inputs are validated or sanitized inadequately by developers. When a user sends an improper input, it results in bugs or wrong responses. However, malicious users will attempt to figure out an attack input over the inadequate development. Those attack input data, i.e. "exploit", often cause unexpected loss and damage. Our work is to build a web exploit generator for automatically figuring out the exploit in order to fix them in time.

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[5, 18] 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. 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 have 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. Regardless of white-box or black-box testing used, manual exploit generation is a high cost process[10]. Therefore, it is necessary for an automatic exploit generator to perform the overall process in order to reduce the cost.

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, and symbolic execution[16, 20]. 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.

A. Overview

This paper is organized as follows. Section II and III explain our method and implementation, respectively. Experimental results are reported in Section IV. Section V describes and compares related work. Finally, Section VI concludes our paper, with future work.

II. METHOD

Our method is based on symbolic execution to automate the web exploit generation process. Symbolic socket[6],first implemented in KLEE, 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 of path explosion on large-scale web applications. However, this option has its own restriction. All details are described in this section.

A. 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 sockets. For XSS attack, a malicious HTTP response is returned from a HTTP server. For SQL injection attacks, a malicious query

978-0-7695-5021-3/13 $26.00 © 2013 IEEE
DOI 10.1109/SERE.2013.26

impacts the database through a DB service port. 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 1. Symbolic data is prepared and injected into an HTTP request. If symbolic data can be propagated to HTTP response or database query during symbolic execution through socket, it will indicate 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, sent to and received from a 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.

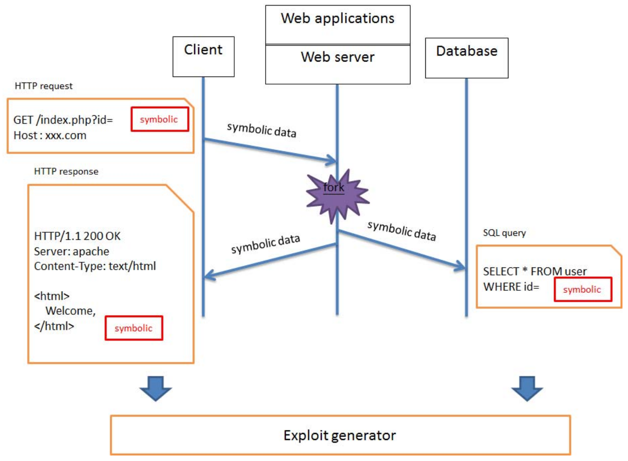


Fig. 1. Symbolic data propagates from HTTP request to HTTP response and database query

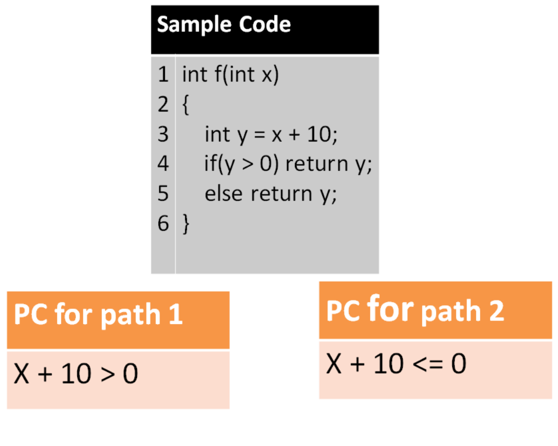
B. Exploit Generation

If the symbolic data is discovered in HTTP responses or database queries that are received from HTTP or DB service ports, it will be an opportunity for exploit generator to generate the exploit, as depicted in Figure 1.

Whenever the symbolic data is initially discovered in the 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 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 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 remaining part of the script.

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 2, 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*.

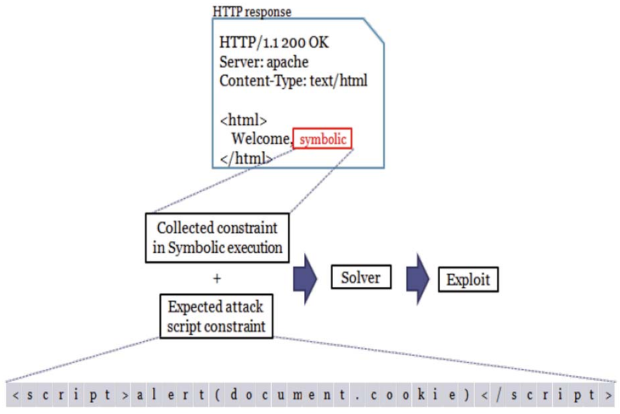


```
int f(int x)
{
    int y = x + 10;
    if(y > 0) return y;
    else return y;
}
```

Fig. 2. 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 3, 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 3. 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.

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 con-



HTTP response
HTTP/1.1 200 OK
Server: apache
Content-Type: text/html
<html>
Welcome, symbolic
</html>
Collected constraint in Symbolic execution
+
Expected attack script constraint
Solver
Exploit
<script>alert(document.cookie)</script>

Fig. 3. Exploit generation by constraint solving

straint solving can assist exploit generation of the potential vulnerability that is under some arithmetic operations, encoding operations or simple mutations. Those vulnerabilities are hard to be discovered in the past, but can be detected by our system.

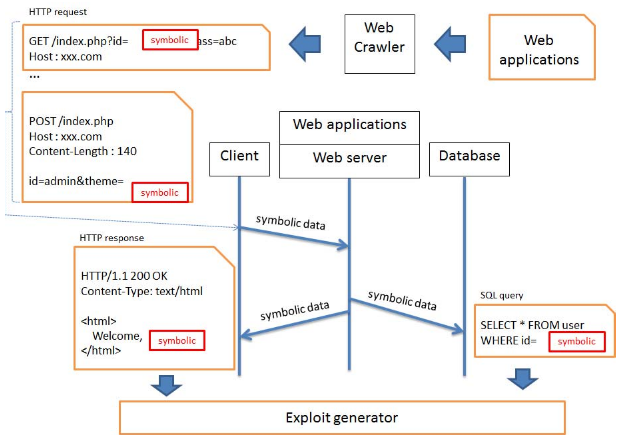


Fig. 4. Flow diagram of our automatic process

2) Cooperation with Web Crawler: Web crawler[1] 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 with symbolic data and process symbolic execution through socket. This situation is explained in Section II-A. To cooperate with exploit generator, a fully automatic process can be constructed to generate web exploit. The flow diagram is shown in Figure 4.

C. Single Path Concolic Mode

The weakness of symbolic execution is the path explosion problem during execution. This leads to the challenge in this paper 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. Exploring a particular single path is more effective than 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 inputs. 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 references the given concrete input instead of the original concrete value. The execution does not fork for another new path.

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 because the concrete input is given and fixed. Therefore, branch conditions can be abandoned and just be collected and kept in the later backup for the exploit generation process. This backup mechanism can also optimize the speed of overall execution.

Figure 5 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.

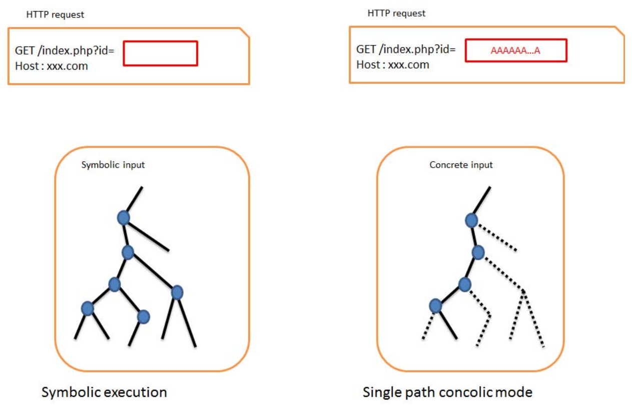


Fig. 5. Single path concolic mode

- 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 by symbolic execution on the given concrete input. This is a tradeoff 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 different paths between the given concrete input and the exploit. It usually occurs at the branch of validating, sanitizing or exception checking on the input string. Our evaluation results reveal that single path concolic mode is able to figure our most of the sanitizing problems. Figure 6 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*.

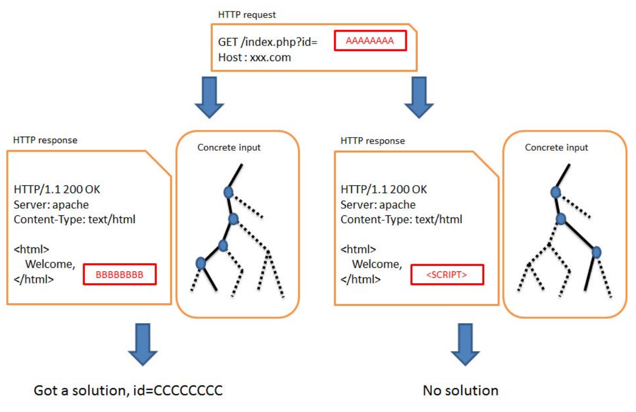


Fig. 6. Different path with the given concrete input and the exploit

III. Implementation

In this section, we explain in detail how our method is implemented on S^{2}E[8], which is a symbolic execution platform. The symbolic environment on S^{2}E assists symbolic propagation through sockets and a handler is built to receive symbolic data through sockets. After receiving the symbolic data, it triggers the exploit generator, which is wrapped as a plugin of S^{2}E. Moreover, single path concolic mode and other optimizations are also implemented to speed up the overall process inside S^{2}E. The symbolic environment on S^{2}E is shown in Figure 7.

A. Symbolic Socket

1) Symbolic Environment on S^{2}E: S^{2}E has an ability to perform symbolic execution on the whole operating system rather than applications. This ability comes from the combination of QEMU[4] and KLEE[6]. KLEE is a symbolic execution engine built on top of the LLVM compiler infrastructure[17]. 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, S^{2}E 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[13].

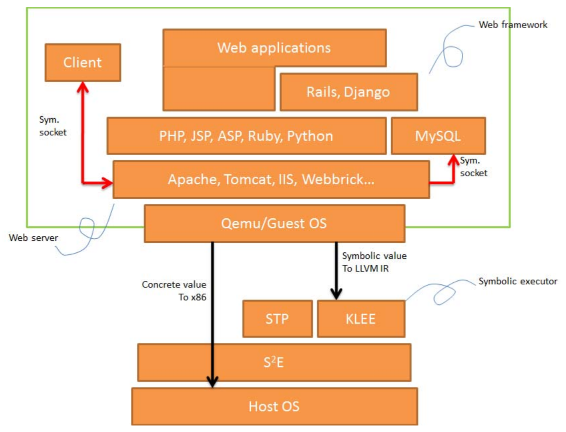


Fig. 7. The symbolic environment on S^{2}E

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 executions on the whole system in S^{2}E, all symbolic environments are already supported including symbolic socket. A sample code in Figure 8 demonstrates the use of 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 affected by the symbolic variable, *buf*, through symbolic socket.

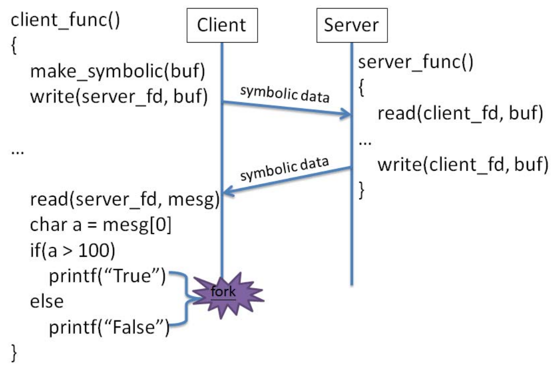

```
client_func()
{
  make_symbolic(buf)
  write(server_fd, buf)
…
  read(server_fd, mesg)
  char a = mesg[0]
  if(a > 100)
    printf("True")
  else
    printf("False")
}
```
```
server_func()
{
  read(client_fd, buf)
…
  write(client_fd, buf)
}
```

Fig. 8. Sample code for symbolic socket between client and server

2) Architecture: The overall architecture for our automatic exploit generation is based on S^{2}E and shows in Figure 9.

In the figure, the arrow represents the symbolic propagation through symbolic socket and solid block represents the main implementation part. *s2eget* and *s2e_ myop* are the S^{2}E instructions.

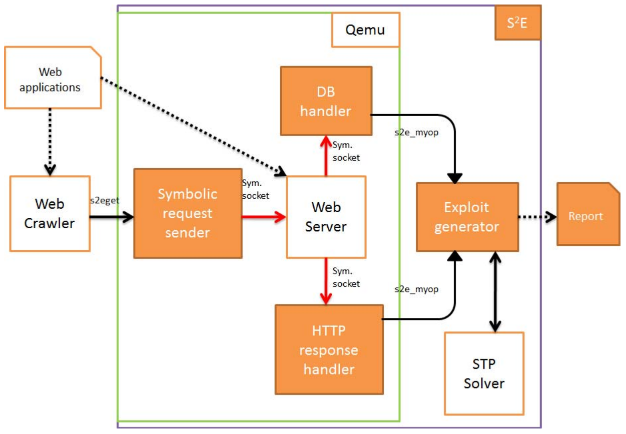
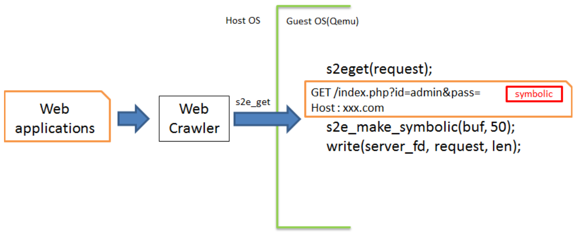


Fig. 9. Overall architecture for automatic exploit generator

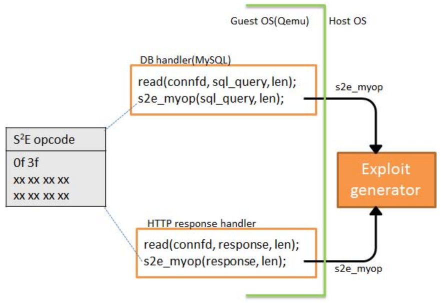
3) Symbolic Response and Query Handler: The concept of symbolic socket can be developed and applied to HTTP to perform symbolic execution on web applications and servers. To cooperate with the web crawler, all of the possible HTTP requests are crawled from web applications and send to the guest OS by a built-in S^{2}E instruction, *s2eget*. Each parameter in crawled HTTP requests is replaced with symbolic data. Then, a symbolic request is sent to the web server through the socket to perform symbolic execution on web applications and servers together. The flow diagram is shown in Figure 10. This is the first part from web crawler to symbolic request in Figure 9.



```
s2eget(request);
GET /index.php?id=admin&pass=
Host : xxx.com
s2e_make_symbolic(buf, 50);
write(server_fd, request, len);
```

Fig. 10. From web crawler to symbolic request

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 symbolic responses and symbolic queries respectively. The database handler is a modified version of MySQL. A new S^{2}E 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 11. This is the second part from a symbolic response or query to exploit generator in Figure 9.



```
read(connfd, sql_query, len);
s2e_myop(sql_query, len);

0f 3f
xx xx xx xx
xx xx xx xx

read(connfd, response, len);
s2e_myop(response, len);
```

Fig. 11. From symbolic response or query to exploit generator

B. Exploit Generation

Whenever the exploit generator is triggered by our customized S^{2}E instruction, the received data, which is an HTTP response or database query, is analyzed. To search a contiguous symbolic data, the process is shown in Algorithm 1. The received data should be judged if it is symbolic or concrete sequentially. Until a contiguous symbolic data is located, it has to ensure that the length is long enough to be involved in 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 II-B must be satisfied.

Algorithm 1: Searching for contiguous symbolic data

```
Input: data : received HTTP response or database query
1  symbolic_len ← 0
2  for $i \leftarrow 0$ to $length(data)$ do
3      if $isByteSymbolic(data + i)$ then
4          symbolic_len ← symbolic_len + 1
5      else
6          if $symbolic\_len \neq 0$ then
7              expect_attack ← constructAttack(data, i)
8              if $symbolic\_len \geq length(expect\_attack)$ then
9                  symbolic_data ← data + i - symbolic_len
10                 solveExploit(symbolic_data,
                       expect_attack)
11             symbolic_len ← 0
12         else
13             continue
14 expect_attack ← constructAttack(data)
15 if $symbolic\_len \geq length(expect\_attack)$ then
16     symbolic_data ← data + i - symbolic_len
17     solveExploit(symbolic_data, expect_attack)
```

Then, a process 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 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 obtained, it will report as maybe vulnerable instead of exploitable. Reason for unsolvable may be the restriction that mentions in Section II-C or the limitation of constraint solver [3].

Algorithm 2: Solving the exploit constraints

```
Input: symblic_data : symbolic data, expect_attack :
       target attack string
Output: exploit : the solved exploit
1 backupConstraints()
2 for $i \leftarrow 0$ to length(expect_attack) do
3     tmp ← readMemory(symbolic_data + i)
4     constraint ← constructConstraint(tmp, expect_attack
      + i)
5     addConstraint(constraint)
6 exploit ← getSymbolicSolution()
```

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 the correct syntax.

By considering an example in Figure 12, a stack is used to maintain the status of HTML syntax, such as <, >, " and '. Whenever symbolic data 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.

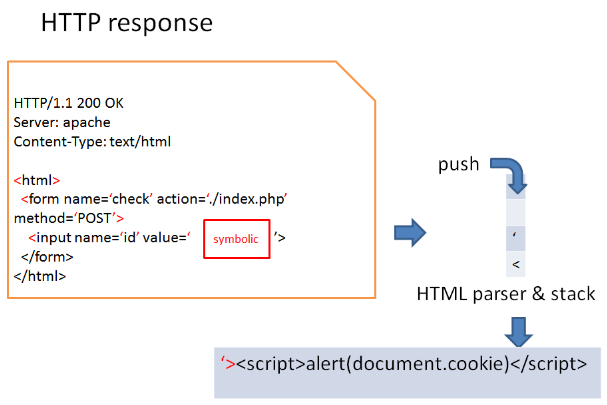
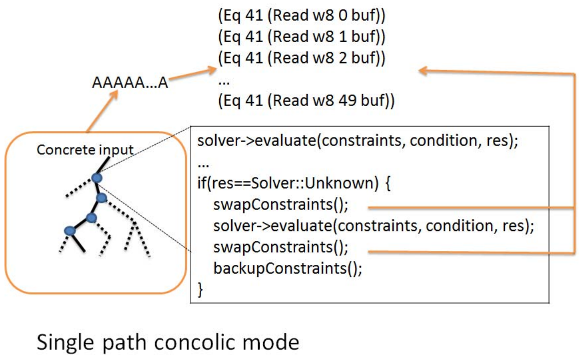


Fig. 12. Completing the syntax of the expected attack script

C. 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 13 for a concrete input, *AAAAAAAA*. A vector container, *concreteConstraints*, is used to store these constraints.



```
solver->evaluate(constraints, condition, res);
...
if(res==Solver::Unknown) {
   swapConstraints();
   solver->evaluate(constraints, condition, res);
   swapConstraints();
   backupConstraints();
}
```

Fig. 13. Concrete input constraints

Whenever a branch is encountered and its branch condition is evaluated by that 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 and is dependent on the given concrete input. So the branch is fixed and S^{2}E will not fork for two feasible paths. 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 II-C. An option is implemented for switching between single path concolic mode and the original symbolic execution at any time conveniently.

- 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 when requiring the current path constraints inside S^{2}E, 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 to a complex symbolic value.

IV. Results

Two types of experiments have been conducted to show the feasibility of web testing through the whole system symbolic executions. The first one demonstrates that platform-independent web testing with our method. The second one is the experimental results of our automatic exploit generator. Part of the test cases are from *Ardilla*. We also have test cases from real-world web applications.

A. Experimental Environment

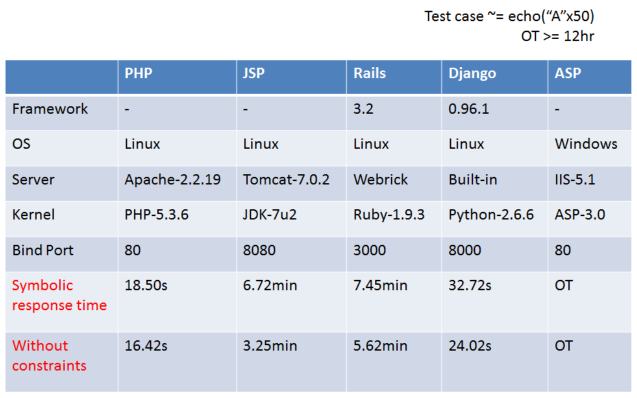
All experiments were performed on a host hardware including a 2.4Ghz CPU with 8 cores, 8GB physical memory and host OS with Ubuntu 10.10 64-bit desktop edition. The guest environment that is emulated by Qemu includes 2.83GHz CPU with a single core, 128MB physical memory and guest OS with Debian 5.0.7 32-bit for Linux platform and Windows XP sp2 for windows platform. The software environment is based on S^2E 1.0 and the database handler is based on MySQL 5.5.15.

B. 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[14] and Django[11].

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 I. 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 the link of symbolic data into the script is broken unexpectedly during the symbolic propagation inside JVM. Thus, the exploit cannot be solved because of the insufficient symbolic information. The experiment on ASP cannot be finished in 12 hours due to the ASP architecture or the complexity of program structure inside their kernel. This issue may be due to the ASP platform, influenced by the symbolic data.

We can also test in alternative way by 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 possibly 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 I.



Test case ~= echo("A"x50)
OT >= 12hr

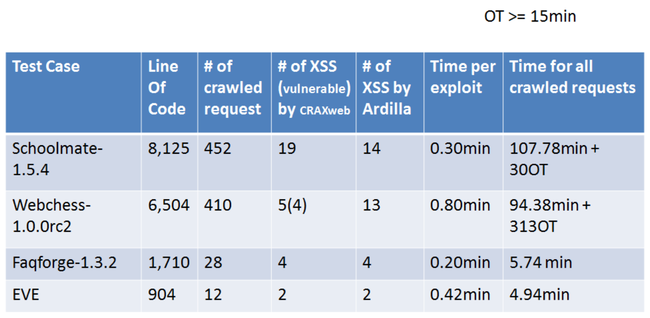
	PHP	JSP	Rails	Django	ASP
Framework	-	-	3.2	0.96.1	-
OS	Linux	Linux	Linux	Linux	Windows
Server	Apache-2.2.19	Tomcat-7.0.2	Webrick	Built-in	IIS-5.1
Kernel	PHP-5.3.6	JDK-7u2	Ruby-1.9.3	Python-2.6.6	ASP-3.0
Bind Port	80	8080	3000	8000	80
Symbolic response time	18.50s	6.72min	7.45min	32.72s	OT
Without constraints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

TABLE I
EVALUATION FOR DIFFERENT PLATFORMS

C. 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 II and III are the same test cases from *Ardilla*. The criteria in *Ardilla* for discovering new exploit are the different vulnerable line of code in PHP. Our criteria for discovering new exploit are with different paths between exploits generated. Thus, the comparisons 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*.



OT >= 15min

Test Case	Line Of Code	# of crawled request	# of XSS (vulnerable) by CRAXweb	# of XSS by Ardilla	Time per exploit	Time for all crawled requests
Schoolmate-1.5.4	8,125	452	19	14	0.30min	107.78min + 30OT
Webchess-1.0.0rc2	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 II
EVALUATION FOR EXPLOIT GENERATION WITH TEST CASES FROM ARDILLA

Web applications in Table IV 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 our case study. *DedeCMS* is a famous content management system (CMS). The results of eleven generated exploits for *DedeCMS* came from a zero-day vulnerability that was found half a year

Test Case	Line Of Code	# of crawled request	# of SQLi by CRAXweb	# of SQLi by Ardilla	Time per exploit	Time for all crawled requests
Schoolmate-1.5.4	8,125	269	12	6	0.55min	148.58 min
Webchess-1.0.0rc2	6,504	65	6	12	0.39min	25.15 min
Faqforge-1.3.2	1,710	7	3	1	0.27min	1.88 min
EVE	904	9	3	2	0.24min	2.12 min

TABLE III
EVALUATION FOR SQL INJECTIONS

ago. The built-in admin interface from old version *Django* are also vulnerable and the exploit of CVE-2008-2302[9] 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.

OT >= 15min

Test Case	Line Of Code	Platform	# of crawled request	# of XSS (vulner-able)	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 IV
EVALUATION FOR EXPLOIT GENERATION

V. RELATED WORK

Symbolic execution is a popular software testing technique. Some related works have applied this technique in the field of web security. Table V shows a comparison between them and our work, CRAXweb. SAFELI[12] 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[2] 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. MIT later proposed an improved work called Ardilla[15] 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 the former section. Kudzu[19] 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[7] was a symbolic executor for Rails[14] in 2011. It was a recent work for symbolic execution on web security.

As mention in later Section II-A, platform-independent is one of the contributions in our work than other related systems. The feature of mutation in Table V means the ability of constraint solving that explains in Section II-B1, which is the other contribution in this paper.

	SAFELI	Apollo	Ardilla	Kudzu	Rubyx	CRAXweb
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, XSS1, XSS2	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 V
RELATED WORK FOR SYMBOLIC EXECUTION ON WEB APPLICATIONS

VI. CONCLUSION AND FUTURE WORK

In this section, we summarize the contributions and conclude our work. Some future work is proposed to explore more web security issues with similar methods in this paper.

A. Conclusion

This paper implemented an automatic exploit generator for web security issues on real-world web applications. Symbolic socket is used as the input for symbolic executions on the web platforms. In contrast with other related systems, applying symbolic socket on HTTP is a comprehensive solution and provides the capability of platform-independent web testing.

Whenever the symbolic information is received from sockets and propagates into security-concerned operations, such as SQL queries, a potential exploit can be solved by constraint solving. This constraint resolving ability leads to more feasible exploits for potential vulnerability, under some arithmetic, encoding operations, or complicated mutations.

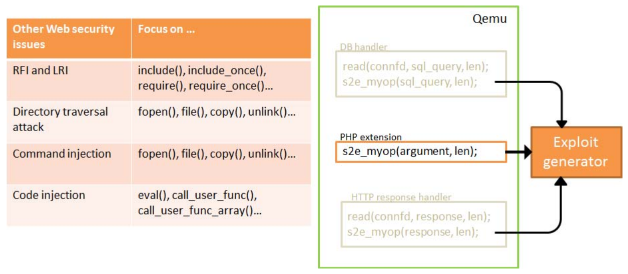
In order to apply our work on real-world web applications, single path concolic mode and some optimizations are proposed and implemented to overcome the challenge on large-scale applications. The objective of single path concolic mode is to force the exploration on symbolic execution in only one path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

In our evaluation, nine web applications are evaluated, include the benchmarks from *Ardilla* and real-world web applications for different platforms, such as PHP and Django. All applications were tested and exploits were generated by our system. The experimental results reveal the feasibility of our implementation.

B. Future Work

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

1) Other Web Security Issues: Other types of web security issue are also possible to generate the exploit in the 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 14, 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.



Other Web security issues	Focus on ...
RFI and LRI	include(), include_once(), require(), require_once()...
Directory traversal attack	fopen(), file(), copy(), unlink()...
Command injection	fopen(), file(), copy(), unlink()...
Code injection	eval(), call_user_func(), call_user_func_array()...

Fig. 14. Exploit for other web security issues

2) Symbolic execution with Browser: To consider web security issues in Ajax or HTML5, our present method that mentions in Section III-A3 is not suitable for them. It is due to that 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 to Section VI-B1. Handler including the new S^2E instruction, *s2e_ myop*, should be built in browser and triggers the exploit generator later. The scenario is shown in Figure 15.

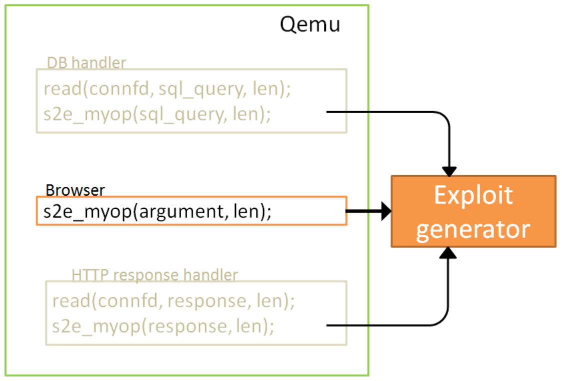


Fig. 15. Symbolic execution with Browser

VII. Acknowledgements

This work is supported in part by NCP, TWISC, National Science Council(NSC-101-2221-E-009-037-MY2, and NSC 100-2219-E-009-005) and Industrial Technology Research Institute of Taiwan (ITRI FY101 2Q-2).

References

[1] Acunetix, “Acunetix web crawler,” http://www.acunetix.com/.
[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst, “Finding bugs in dynamic web applications,” in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 261–272.
[3] C. Barrett, L. De Moura, and A. Stump, “SMT-COMP: Satisfiability modulo theories competition,” in *Computer Aided Verification*. Springer, 2005, pp. 503–516.
[4] F. Bellard, “QEMU, a fast and portable dynamic translator.” USENIX, 2005.
[5] P. Bisht, T. Hinrichs, N. Skrupsky, and V. Venkatakrishnan, “WAPTEC: whitebox analysis of web applications for parameter tampering exploit construction,” in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 575–586.
[6] 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*. USENIX Association, 2008, pp. 209–224.
[7] A. Chaudhuri and J. Foster, “Symbolic security analysis of ruby-on-rails web applications,” in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 585–594.
[8] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, “Selective symbolic execution,” in *Workshop on Hot Topics in Dependable Systems*, 2009.
[9] T. M. Corporation, “Common vulnerabilities and exposures,” http://cve.mitre.org/cve/.
[10] J. DeMott, R. Enbody, and W. Punch, “Towards an automatic exploit pipeline,” in *International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE, 2011, pp. 323–329.
[11] J. Forcier, P. Bissex, and W. Chun, *Python web development with Django*. Addison-Wesley Professional, 2008.
[12] 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*. ACM, 2008, pp. 34–39.
[13] V. Ganesh and D. Dill, “A decision procedure for bit-vectors and arrays,” in *Computer Aided Verification*. Springer, 2007, pp. 519–531.
[14] D. Hansson *et al.*, “Ruby on rails,” *Website. Projektseite: http://www. rubyonrails. org*, 2009.
[15] A. Kieyzun, P. Guo, K. Jayaraman, and M. Ernst, “Automatic creation of sql injection and cross-site scripting attacks,” in

IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009. Ieee, 2009, pp. 199–209.

[16] J. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[17] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 2004, pp. 75–86.

[18] M. Martin and M. Lam, “Automatic generation of xss and sql injection attacks with goal-directed model checking,” in *Proceedings of the 17th conference on Security symposium*. USENIX Association, 2008, pp. 31–43.

[19] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A symbolic execution framework for javascript,” in *IEEE Symposium on Security and Privacy (S&P)*.

[20] E. 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 *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2010, pp. 317–331.