

AUTOMATED PENETRATION TESTING FOR PHP WEB APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

By

Zixiang Zhu

Georgia Institute of Technology

November 2016

Copyright © Zixiang Zhu 2016

Tests are stories we tell the next generation of programmers on a project.

Roy Oshero, The Art of Unit Testing: With Examples in .NET

TABLE OF CONTENTS

List of Tables	3
List of Figures	4
Chapter 1: Introduction	1
Chapter 2: Technical Approach	3
2.1 Background	3
2.1.1 Data-flow Analysis	3
2.1.2 HipHop Bytecode	6
2.2 Inter-procedural CFG Construction	6
2.3 Testcase Generation Algorithm	6
2.3.1 Precise String Analysis for Input Generation	6
2.3.2 Array Content Propagation	15
2.4 Interface Discovery Algorithm	15
Chapter 3: Empirical Evaluation	16
3.1 Experiment Setup	16
3.1.1 Experimental Subjects	16
3.1.2 Tools Used in Evaluation	16

3.2	Results	16
3.2.1	Email Sender	16
3.2.2	Fancy Hotel (CS4400 Class Project)	17
3.2.3	Paycheck Calculator	19
Chapter 4:	Discussion	20
4.1	Interface Discovery	20
Chapter 5:	Conclusion	21
Appendix A:	Experiment Equipment	23
Appendix B:	Algorithm	24
References	25

LIST OF TABLES

2.1	Supported PHP Core operations	14
2.2	Supported PHP string operations	15

LIST OF FIGURES

2.1	Control Flow Graph for Function t3f1	4
-----	--	---

ABSTRACT

Penetration Testing emerged in the mid-1960s as an approach to exploit vulnerabilities of possible attacks of a software application by nefarious users. Traditional penetration testing is done manually, which is not only inefficient but also unstable in terms of reliability. In the recent decade, multiple automated penetration testing approaches have been proposed, including automatically test inputs generation based on genetic algorithms and neural networks learning. However, these black-box testing methods only have limited accuracy, and usually require a large number of data to train the agents before they can be used to do actual tests. To address this issue, we present a novel approach in which program static analysis is exploited. The proposed penetration testing system is able to not only estimate HTTP request data more precisely, but also discover dynamic interfaces exposed by the web applications. This research is focused on PHP web applications only.

CHAPTER 1

INTRODUCTION

As the scale of enterprise web applications grows rapidly, finding an effective way for testing site reliability is becoming increasingly important. As a widely-used testing technique, penetration testing is able to exploit the vulnerabilities of a web application backend by simulating sending HTTP requests from the client side. Since penetration testing is used to test as many parts of an application as possible, comprehensiveness is the most important factor in performing penetration testing. An ideal penetration test is the one that generates HTTP requests which cover 100% of the server-side code.

Traditionally, penetration tests are performed as black-box testing, which views the program to be tested as a "black box" whose implementation detail is unknown to the tester. Black-box testing allows the tester to focus on only the input and the output generated by the program; however, since the inner functionality of the program is unknown, it is difficult for the tester to generate a comprehensive test suite that guarantees 100% backend code coverage. In traditional manual penetration testing, a tester could only apply "educated guesses" when creating test requests. Some recent research proposed automated penetration testing using AI techniques such as genetic algorithm [1] and neural network models [2]. Although these automated approaches showed promising results in some scenarios, by nature they are still black-box testing, only improving the accuracy of "educated guess" by inferring from the statistical analysis results generated from the differences between expected outputs and actual outputs.

The lack of comprehensiveness and efficiency in black-box penetration testing prompted us to propose a new approach that performs white-box testing, during which tests are performed by looking at the implementation details in the source code of the program itself. In our implementation, both the interfaces that an application exposes and all the request

data that is processed in the backend can be inferred from static analysis results. The test suite goes through the program and checks at what positions each HTTP request variable is used. If some other variables are involved at any of the positions, then the program will use the result of data flow analysis to construct the exact value for such variables, which are also the possible values that the HTTP request variable could take. The test suite uses a library developed by Christensen, Møller, and Schwartzbach [3] to construct an automaton that represents possible string values during string variable construction.

As another major part of penetration testing, interface discovering was traditionally done by doing web crawling. However, since modern web frontend is becoming increasingly dynamic, only extracting information from the client-side HTML pages does not guarantee enough interfaces are discovered. In 2007, Halfond and Orso [4] proposed a novel approach for discovering web application interfaces using static analysis. Such technique showed promising results for finding and grouping dynamic interfaces that are not exposed directly by the frontend. In this project, the same algorithms proposed by Halfond et al. is applied for doing interfaces discovery, but is targeted for PHP web applications.

CHAPTER 2

TECHNICAL APPROACH

2.1 Background

2.1.1 Data-flow Analysis

In compiler theory, data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. The basis for performing data-flow analysis is control flow graph (CFG), which is used to determine those parts of the program to which a particular value assigned to a variable might propagate. In our particular case, we used reaching-definition for each instruction to represent the data flow information. Two techniques applied for performing reaching-definition analysis are *Liveness Analysis* and *Reaching-Def Analysis*.

Control Flow Graph (CFG) is a graph representation of all the paths that a computer program might be traversed through during its execution. In a control flow graph, each node represents a basic block - a sequential piece of a program that does not include jumps or jump targets, while the directed edges represent jumps in the program. Figure 2.1 is the complete control flow graph constructed for an example function t3f1. Each circle represents a basic block (first number in the circle denotes its line number), and the black edges represent the possible jumps that the program can take during execution. It is worth noting that the while loop in t3f1 includes a condition check at line 4, which can be reached from either line 4 (before loop starts) or from line 13 (after last instruction in the loop is finished). Therefore in the CFG there are two nodes representing line 5, with one coming from line 4 and another coming from line 13.

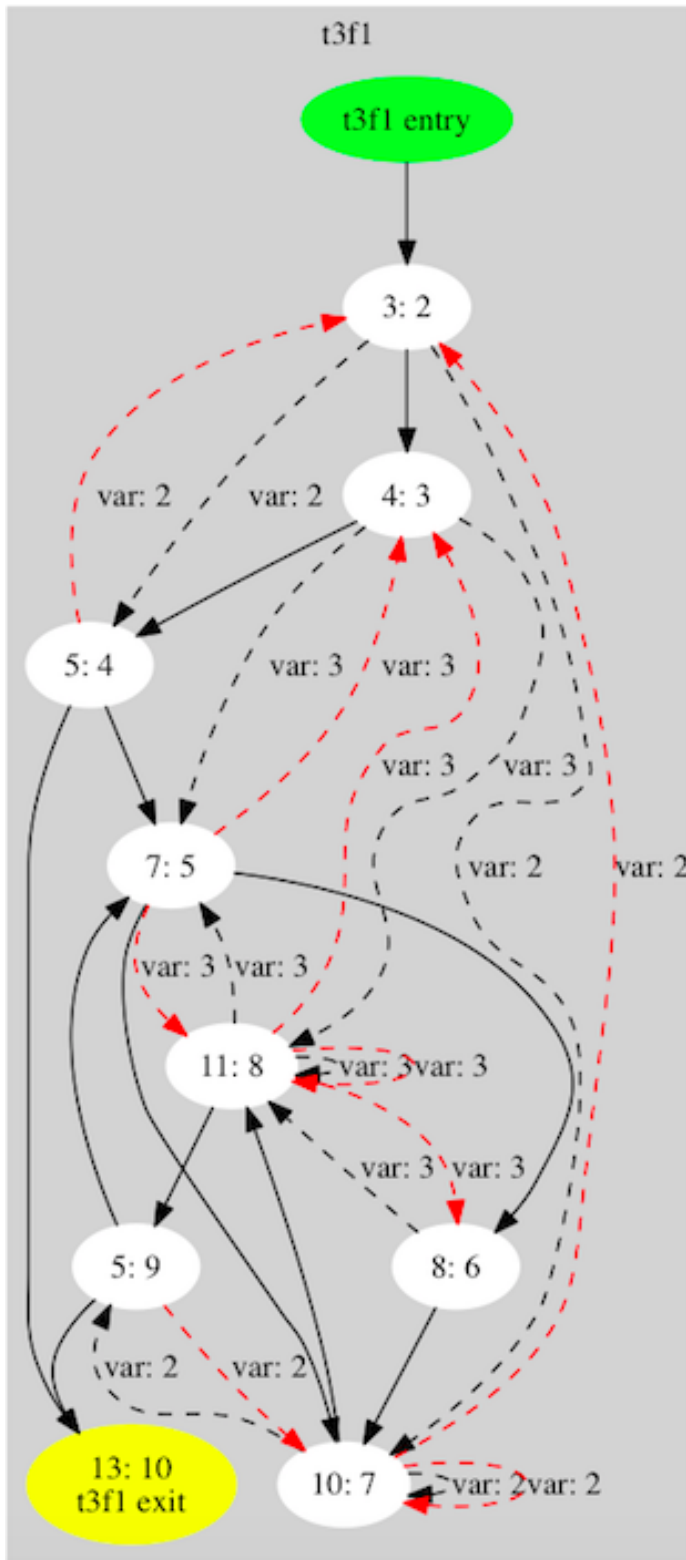


Figure 2.1: Control Flow Graph for Function `t3f1`

```

1 <?php
2 function t3f1($c, $d) {
3     $a = 0;
4     $b = 0;
5     while($a <= 5)
6     {
7         if ($b > 5) {
8             $b = 5;
9         }
10        $a = $a + 1;
11        $b = $b + 2;
12    }
13 }
14 ?>

```

As one of the two techniques used for performing reaching-definition analysis, liveness analysis is to calculate, at instruction level, the variables that may be potentially read before their next write, that is, the variables that are live at the exit from each program point. In the given example, variable *b* is first defined at line 4; it is also defined at line 8 and 11 in the main loop. Therefore, var *b* is said to be "alive" from line 4 to line 8, and from line 11 back to line 8. Given liveness analysis output, def-use chains can be constructed for each variable. The DU chain represents the exact places where a variable definition is later used in other instructions. The DU chains in *t3f1* are shown as black dotted edges in Figure 2.1.

To the contrary of liveness analysis, reaching-def analysis calculates the possible definition instructions for a given use variable. Reaching-def analysis helps generate use-def chains for each variable used in a program. The UD chains for variables in *t3f1* are shown as red dotted edges in Figure 2.1.

2.1.2 HipHop Bytecode

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.2 Inter-procedural CFG Construction

Usually during static analysis, an intra-procedural CFG is constructed for each individual function before all intra-procedural CFGs are combined into one inter-procedural CFG (ICFG) that represents the control flow for the entire program. ICFG is a combination of all individual functions and a function call graph (CG), which is a graph representation of function dependencies in a program, in which functions are represented as nodes and function invocations are represented as directed edges. Since one variable defined/used in any function can be used/defined in other functions, extra work needs to be done in order to expand a variable's DU and UD chains in ICFG. Algorithm 1 shows the construction of DU and UD chains in ICFG given individual CFGs and CG.

2.3 Testcase Generation Algorithm

2.3.1 Precise String Analysis for Input Generation

Typically, users interact with a web application through a user interface (e.g., web page) that allows them to enter input data (e.g., into a form) and submit such data to the web ap-

Algorithm 1 Dataflow Expansion in ICFG

Require:

- 1: CG : call graph of the application program;
- 2: $CFGs$: CFGs for all functions in the program

▷ main

```
3: procedure DATAFLOW EXPANSION
4:    $visited \leftarrow \emptyset$ 
5:   for Each  $CFG$  in  $CFGs$  do
6:     for Each instruction  $I$  in  $CFG$  do
7:       if  $I$  has KILL variable  $kv$  then
8:         COMPLETEDEFUSE( $CFG, I, kv, visited$ )
9:       end if
10:    end for
11:  end for
12:   $visited \leftarrow \emptyset$ 
13:  for Each  $CFG$  in  $CFGs$  do
14:    for Each instruction  $I$  in  $CFG$  do
15:      if  $I$  has GEN variable  $gv$  then
16:        COMPLETEUSEDEF( $CFG, I, gv, visited$ )
17:      end if
18:    end for
19:  end for
20: end procedure
```

```
21: function COMPLETEDEFUSE( $cfg, instr, var, visited$ )
22:   if  $instr$  in  $visited$  then
23:     return  $instr.duchain$ 
24:   end if
25:   for Each Use  $use$  in  $instr.duchain$  do
26:     if  $use$  is a function call site then
27:       Get the callee function  $targetfunction$ 
28:       Get the function parameter  $targetvar$  in  $targetfunction$  that corresponds
       to  $use.var$ 
29:       Get the instruction  $targetInstr$  that initiates  $targetvar$  in  $targetfunction$ 
30:        $newuses \leftarrow$  COMPLETEDEFUSE( $targetfunction.CFG, targetInstr,$ 
        $targetvar, visited$ )
31:        $instr.duchain \leftarrow (instr.duchain - use) \cup newuses$ 
32:     end if
33:   end for
34:    $visited \leftarrow visited \cup instr$ 
35:   return  $instr.duchain$ 
36: end function
```

```

37: function COMPLETEUSEDEF(cfg, instr, var, visited)
38:   if instr in visited then
39:     return instr.udchain
40:   end if
41:   for Each Def def in instr.udchain do
42:     if def is a function entry site then
43:       Get all the functions caller_funcs that calls current function
44:       for Each function caller_func in caller_funcs do
45:         Get the function parameter sourcevar in caller_func that corresponds
         to def.var
46:         Get the instruction sourceInstr in caller_func that passes sourcevar
         to callee stack
47:         newdefs  $\leftarrow$  COMPLETEUSEDEF(caller_func.CFG, sourceInstr,
         sourcevar, visited)
48:         instr.udchain  $\leftarrow$  (instr.udchain - def)  $\cup$  newdefs
49:       end for
50:     end if
51:   end for
52:   visited  $\leftarrow$  visited  $\cup$  instr
53:   return instr.duchain
54: end function

```

plication via HTTP request. The data that users enter are mostly in string literal forms (e.g., name, address, email, etc). Therefore, generating appropriate string inputs automatically during penetration testing is of our particular interest.

The most direct and effective approach for generating test string literal inputs is to check what values the input variable is compared with during input validation in the application backend. Usually after the application receives a request from the client side, it will compare each input field against some specific values or regular expressions, in order to check if the value is valid, or to process data differently according to the given value. For string literal fields, the checker strings/regex are particularly useful, because they provide a good reference for generating our test inputs (the testcase generator only needs to make the input value either match the given value or differ from it). Therefore, in order to generate our test inputs, the fundamental step is to construct the string literals or regular expressions that the inputs are compared with in the application.

Conventional static analysis does not try to get the information of what the value of a

variable exactly is. However, in order for our testcase generation tool to construct more plausible input string literals, it is important to estimate the exact value of string literals used to compare with inputs passed into the application. Fortunately, the structure of HipHop Bytecode makes it possible to perform such action. The compiled HHBC for a PHP program models the flow of program execution by using a stack of frames referred to as the "call stack". Each call stack maintains an "evaluation stack", on which data is pushed or popped based on type of HHBC instruction. By using a stack data structure to model this evaluation stack, along with the data flow analysis results that the earlier phase generates, we can partially simulate the actual program's behavior during execution, which is sufficient for us to construct exact string literal values.

Listing 2.1 shows a simple PHP program that uses a *create_name* function to build full names from first and last names. In line 3, first name is first concatenated with a space and then concatenated with last name. Listing 2.2 is the HHBC generated for line 3 after compilation, which shows how the act of concatenation in line 3 is performed by HHVM. HHVM starts the series of actions by declaring a String (a space) (line 152), which is pushed to the evaluation stack; next, HHVM looks for the variable with ID 0 (CGetL2 0 in line 157), and pushes it onto the stack; then, when it comes to the "Concat" instruction, HHVM takes the top two elements from the stack, append the topmost value (which is the space) to the back of the second topmost value (the variable with ID 0), and pushes the result onto the stack, thus completing the first concatenation. The rest of the HHVM actions are similar to what have happend: HHVM pushes the variable with ID 1 to the stack, pops the top two elements, concatenates them, and finally pushes the result to the stack.

Listing 2.1: Sample PHP Code and Compiled HHBC Snippet

```

1 <?php
2 function create_name($a, $b) {
3     return $a."_".$b;
4 }
5 $firstname1 = "John";
6 $lastname1 = "Doe";
7 $firstname2 = "George";
8 $lastname2 = "Burdell";
9 $name1 = ceate_name($firstname1, $lastname1);
10 $name2 = create_name($firstname2, $lastname2);
11 $n = $_POST["name"];
12 if($n == $name1) {
13     echo "You_are_a_UGA_student";
14 } else if($n == $name2) {
15     echo "You_are_a_Georgia_Tech_student";
16 } else {
17     echo "Failed";
18 }
19 ?>

```

Listing 2.2: HHBC Snippet for line 3

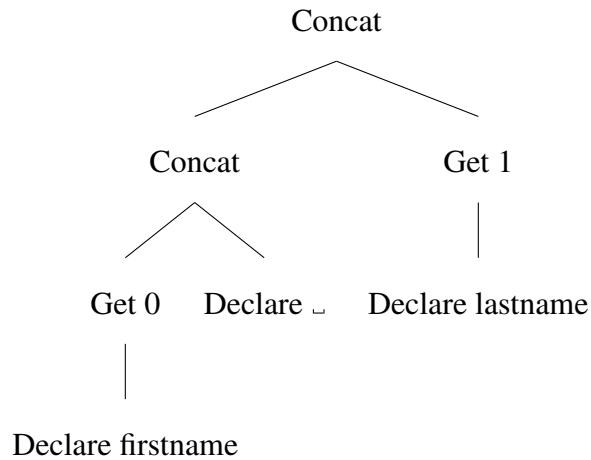
```

1 152: String "_"
2 157: CGetL2 0
3 159: Concat
4 160: CGetL 1
5 162: Concat
6 163: RetC

```

Given the fact that HHBC instructions also represent stack operations, we can take ad-

vantage of such property to build the string literal by mocking HHVM stack operations. In our implementation, a global stack data structure is maintained by the static analyzer. After each basic block is created, all the instructions in the basic block are simulated to work on the stack. The result is an ActionNode tree. ActionNode is a data structure that represents the specific HHVM "Action" performed by the given instruction. For example, "Concat" and "String" instructions represent "Concatenation" and "Declaration" actions respectively. Different type of ActionNode. Essentially each instruction should correspond to an action node and different instruction operators should be represented by distinct action nodes. However, given the large number of PHP string operations, we only implemented ActionNodes for certain commonly used operations (Table 2.2). ActionTree is a hierarchical representation of ActionNode actions upon each other. In the HHBC sample given by Listing 2.2, the ActionNode tree looks like the following:



As shown in the tree diagram above, Concat ActionNode always has two children, while Get ActionNode has one child, and Declare ActionNode has zero. Different ActionNodes have different number of children, depending on their specific actions.

In the last step, static analyzer uses the action trees and a modified Java String Analyzer (JSA) library to construct string literal values. Java String Analyzer (JSA) [4] is a tool that uses static analysis information to predict the possible values of string expressions in Java programs. JSA is consisted of several phases, each transforming the program into a different form. The phases are separated into two parts: The front-end and the back-end.

At the front-end, JSA performs Java language specific static analysis and generates a Flow Graph for each string expression; at the back-end, JSA takes Flow Graphs as its input and converts them in to finite-state automata.

Flow Graph is language-independent, and can be constructed directly from data flow analysis(liveness and reaching-def analysis). Therefore, in our implementation, the static analyzer uses Def-Use chains and Use-Def chains along with the action trees to build flow graph for each string expression; the flow graph is then fed into the backend of JSA library, which will take care the rest of string analysis and generate a finite-state automaton to represent the string expression.

Algorithm 2 Flow Graph Construction Algorithm

Require:

- 1: *CG*: Call graph of the application program
- 2: *ICFGS*: Inter-procedural CFG with complete Def-Use and Use-Def chains

▷ main

- 3: **procedure** FLOW GRAPH CONSTRUCTION
- 4: Group functions in *CG* into Strongly Connected Components *sccGraph*
- 5: Get the topological ordering of SCCs, *sccList* , in *sccGraph*
- 6: $continue \leftarrow true$
- 7: **until** *continue* is *false* **repeat**
- 8: **for** Each SCC *scc* in *sccList* **do**
- 9: **for** Each Function *function* in *scc* **do**
- 10: $cfg \leftarrow function.cfg$
- 11: **for** Each BasicBlock *block* in *cfg.getTopologicalSortedBlocks* **do**
- 12: $success \leftarrow \text{STRINGANALYSIS}(block, ICFG)$
- 13: **if** *success* is *true* **then**
- 14: $continue \leftarrow false$
- 15: **end if**
- 16: **end for**
- 17: **end for**
- 18: **end for**
- 19: **end until**
- 20: **end procedure**

```

21: function STRINGANALYSIS(basicblock, ICFG)
22:   Build ActionTrees trees from basicblock.instructions
23:   for Each ActionTree tree in trees do
24:     success  $\leftarrow$  BUILDFLOWGRAPH(tree.root, ICFG)
25:     if not success then
26:       return false
27:     end if
28:   end for
29:   return true
30: end function

```

```

31: function BUILDFLOWGRAPH(ActionNode, ICFG)
32:   Initialize FlowGraph flowgraph
33:   Initialize FlowGraphNode fgnode
34:   for Each child node childnode of ActionNode.children do
35:     success  $\leftarrow$  BUILDFLOWGRAPH(childnode, ICFG)
36:     if not success then
37:       return false
38:     end if
39:     flowgraph  $\leftarrow$  childnode.flowgraph
40:     Add all nodes in childnode.flowgraph to flowgraph
41:   end for
42:   switch ActionNode.type do
43:     case Declare
44:       v  $\leftarrow$  ActionNode.value
45:       at  $\leftarrow$  v.GETAUTOMATON
46:       fgnode  $\leftarrow$  flowgraph.ADDINITIALIZATIONNODE(at)
47:     case Get
48:       instr  $\leftarrow$  ActionNode.instr
49:       fgnode  $\leftarrow$  flowgraph.ADDASSIGNMENTNODE;
50:       for Each Definition def in instr.usedefchain do
51:         Get the ActionNode setNode corresponding to def
52:         setFlowGraph  $\leftarrow$  setNode.flowgraph
53:         Add all nodes in setFlowGraph to flowgraph
54:         Set Def-Use relationship between setNode.fgnode and fgnode
55:       end for
56:     case Set
57:       ActionNode toSet  $\leftarrow$  ActionNode.child
58:       fgnode  $\leftarrow$  flowgraph.ADDASSIGNMENTNODE(toSet)
59:       Set Def-Use relationship between toSet and fgnode
60:     case Load

```

```

61:      case Function
62:          fgnode  $\leftarrow$  flowgraph.ADDASSIGNMENTNODE
63:          Get called function name funcName
64:          retCFG  $\leftarrow$  ICFG.GETCFG(funcName)
65:          retBlocks  $\leftarrow$  retCFG.GETRETBLOCKS
66:          vars  $\leftarrow$  ActionNode.children
67:          varMap  $\leftarrow$  Associate each variable in vars with its formal parameter in
function pointed by funcName
68:          for Each BasicBlock retBlock in retBlocks do
69:              retActionNode  $\leftarrow$  retBlock.GETLASTACTIONNODE
70:              fg_copy  $\leftarrow$  COPYFLOWGRAPH(retActionNode.flowgraph, varMap)
71:              Add all nodes in fg_copy to flowgraph
72:              Set Def-Use relationship between the last node in fg_copy and fgnode
73:          end for
74:              ActionNode.type  $\leftarrow$  Default
75:              Clear ActionNode.children
76:      case Default
77:          do nothing
78:      end function

```

▷ change type to Default

▷ For string specific actions, refer to Appendix B

Supported PHP Core Operations

Function
Declare
ArrayDeclare
Set
Get
Load

Table 2.1: Supported PHP Core operations

Supported PHP String Operations

Concat
StrComp
Substr
ToUpperCase
ToLowerCase
Reverse
Trim
Split
Replace
StrToTime

Table 2.2: Supported PHP string operations

2.3.2 Array Content Propagation

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.4 Interface Discovery Algorithm

In this step, variables exposed by the application interface(input variables) are identified and grouped logically. This part of the program implements two interface discovery algorithms proposed by Halfond and Orso [4].

CHAPTER 3

EMPIRICAL EVALUATION

3.1 Experiment Setup

3.1.1 Experimental Subjects

The experimental subjects used in the study is consisted of three student-developed projects that use PHP as their backend language.

3.1.2 Tools Used in Evaluation

We evaluate the effectiveness of our interface discovery mechanism by comparing the number and quality of interfaces that it generates with the interfaces extracted by web crawlers. The web crawler tool that we used is OpenWebSpider, an open-sourced framework for crawling/spidering websites. For each interface output, we check if any of the HTTP requests in this output is processed in the backend program.

We evaluate the effectiveness of our test request generation mechanism by comparing it against another PHP test case generation tool, PHPUnit. For each HTTP request, a set of possible values is generated for all of the fields in this request, creating a set of test HTTP requests, which are then sent to the server. Then the final code coverage in the backend is recorded.

3.2 Results

3.2.1 Email Sender

A simple PHP program that processes a form which contains 4 input fields and sends an email according to the information supplied in the form.

Form Inputs

- first_name (required)
- last_name (required)
- email (required)
- telephone (optional)

All required fields are checked against a regular expression. The only optional field (telephone) is not checked against any regex or string variables.

Our penetration testing tool successfully identified all the fields (required and optional). It generated test cases for all the required variables from the given regex expression. For the "telephone" field, which is optional, our testing tool did not infer any information from static analysis, therefore would be producing random strings for this field during testing.

3.2.2 Fancy Hotel (CS4400 Class Project)

A web application powered by PHP and MySQL that serves as an online room reservation system for a non-existing hotel.

User Login

Form Inputs

- username (string)
- password (string)

Our penetration testing tool successfully identified both username and password fields in the interface. Since the application does not compare username and password to specific patterns or values, our static analysis did not infer possible value information for these fields.

However, the analyzer identified session information created during login, which could potentially make subsequent penetration testing more thorough by exploiting user sessions.

In addition, the user login program can be redirected to three different URLs once login information is validated. Since these URLs are not exposed to the front-end HTML, they were not identified by webcrawler. However, by performing static analysis on the source code, our penetration testing tool was able to discover all three of them.

User Registration

Form Inputs

- username (string)
- password (string)
- confirmed_password (string)
- email (string)

Our penetration tool successfully identified all request (or interface) parameters and their names (usr, psw, con_pwd, email). It identified the regular expression that was used to check valid email and username in the program and used this information to construct test cases for the email and username fields.

Room Search

Form Inputs

- start_date (string)
- end_date (string)
- location (string)

Note

- start_date must be greater than 2015-08-01;

- start_date cannot be greater than end_date;
- start_date cannot be smaller than today's date (2015-08-31);
- end_date must be smaller than 2016-01-31;

Our penetration tester successfully identified all the required fields as well as the session information not exposed by the interface itself. Furthermore, static analysis has inferred key information about both start_date and end_date. For start_date, output test cases include "20150831" and "20150801"; for end_date, output test cases include "20160131".

3.2.3 Paycheck Calculator

A basic PHP web application that takes in one input (salary) and shows how much tax should be deduced.

Form Inputs

- salary (float)

Our penetration tool successfully identified the interface. Static analysis generated all possible levels of salary which covered 100% of application code.

CHAPTER 4

DISCUSSION

4.1 Interface Discovery

As Table 3.1 shows, the number of interfaces identified by our automated penetration testing tool is always greater than or equal to what are identified by traditional web crawling.

Such result is in accordance with what is reported by Halford et al. [4], which demonstrates the correctness of our implementation of the interface discovery algorithms. We manually inspected the interfaces exposed by each of the applications and analyzed the procedure that our tool took during interface discovery phase. We identified that there are some cases in which the the web application does not use all the information sent from the client side. PHPCrawler does not have any knowledge of what request is actually processed, so it could generate more request than necessary; however, our interface discovery mechanism is able to send the HTTP requests with only the fields that are known to be handled in the backend, which improves the overall efficiency of penetration testing by reducing the number of unnecessary requests.

CHAPTER 5

CONCLUSION

In this paper we presented a novel approach for performing automated penetration testing for PHP web applications. It uses static analysis to identify the application program's behavior on each variable and function, and then construct the possible interfaces exposed by the application as well as the possible values that each request field can take. We compared the effectiveness of this new test system with traditional penetration testing tools, in terms of interface discovery and test case generation. The result shows our proposed testing tool can not only identify more interfaces for dynamic web applications, but also generate fewer test cases in order to cover 100% of the backend code.

Appendices

APPENDIX A

EXPERIMENT EQUIPMENT

1. Ubuntu 14.04
2. Java Runtime Environment 1.7
3. PHP 5.5
4. HHVM 3.10.0
5. OpenWebSpider

APPENDIX B

ALGORITHM

REFERENCES

- [1] P. S. Srivastava and T. Kim, “Application of genetic algorithm in software testing,” International Journal of Software Engineering and Its Applications, vol. 3, no. 4, Nov. 2009.
- [2] R. Zhao and S. Lv, “Neural-network based test cases generation using genetic algorithm,” in Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on Dec. 2007, pp. 97–100.
- [3] A. S. Christensen, A. Møller, and M. I. Schwartzbach, “Precise analysis of string expressions,” in The 10th International Conference on Static Analysis, 2003.
- [4] W. G. J. Halfond and A. Orso, “Improving test case generation for web applications using automated interface discovery,” in The 6th Joint Meeting of the European Software Engineering 2007.