AUTOMATED PENETRATION TESTING FOR PHP WEB APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

By

Zixiang Zhu

Georgia Institute of Technology

November 2016

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

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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 AND BACKGROUND

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 gurantees 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 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.

PHP is a high-level language that abstracts complex actions into human-readable syntaxes. Although high-level abstraction makes it easy and fun for programmers to write code, it is not preferrable for doing static analysis, which requires each instruction should be as precise as possible, so that no ambiguity would exist in any line of the code.

In 2014 Facebook released the Hack language, an enhanced version of PHP powered by HipHop Virtual Machine (HHVM), a virtual environment that is able to execute HipHop byte code cross-platform. The Hip-Hop byte code can be compiled from either Hack or native PHP. The syntax of Hip-Hop byte code is similar to assembly: variables are fetched/updated by calling get/set instructions explicitly, flow of program is controlled by branch instructions, and a stack space is explicitly used for each procedure call. The explicitness of Hip-Hop byte code makes it ideal for our static analysis.

CHAPTER 2
TECHNICAL APPROACH

2.1 System Requirements

1. Ubuntu 14.04
2. HHVM 3.10.0
3. Java Runtime Environment 1.7

2.2 HHVM Bytecode Processing

HHVM compiles Hack and PHP into an intermediate bytecode. The intermediate bytecode is processed to build control flow graphs.

2.2.1 Intra-Procedural Control Flow Graph Construction

Control Flow Graph (CFG) is a graph representation of a program that represents all the paths that might be traversed through during its execution, which is the basis for performing variable liveness check and building def-use/use-def chains. The procedure for building a CFG has been extensively investigated and there is a standard algorithm for building CFG from intermediate representations (Hip-Hop byte code, in this case):

Basic Block Partitioning

Identify leader statements (i.e. the first statements of basic blocks) by using the following rules:

1. Identify leader statements
    (a) The first statement is a leader

(b) Any statement that is the target of a branch statement is a leader (for most intermediate languages these are statements with an associated label)

(c) Any statement that immediately follows a branch or return statement is a leader

2. The basic block corresponding to a leader consists of the leader, plus all statements up to but not including the next leader or up to the end of the program

Control Flow Graph Construction

The basic block corresponding to a leader consists of the leader, plus all statements up to but not including the next leader or up to the end of the program

1. There is a branch from the last statement of B1 to the first statement of B2, or

2. Control flow can fall through from B1 to B2 because

   (a) B2 immediately follows B1, and

   (b) B1 does not end with an unconditional branch

2.2.2 Data Flow Analysis

In order to keep track of how each variable is used in the program, it is critical to perform data flow analysis, which outputs def-use chains and use-def chains, both of which are critical to the later stages of the project.

Def-Use Chains and Liveness Analysis

Def-use chain is used to keep information about reaching definition of a variable. Specifically, for each use of a variable x, its def-use chain is a list of the definitions of x reaching that use. Liveness Analysis is used to get def-use chains.

- At each program point:

  - Which variables contain values computed earlier and needed later – they are said to be live

- For instruction I:
  - in[I]: live variables at program point before I
  - out[i]: live variables at program point after I
- For a basic block B:
  - in[B]: live variables at beginning of B
  - out[B]: live variables at end of B
- DU[I] = out[I] – in[I]
- Note:
  - in[I] = in[B] for first instruction of B
  - out[I] = out[B] for last instruction of B

Algorithm 1 Liveness Calculation in CFG

```
for Each instruction I do
    in[I] ← ∅
    out[I] ← ∅
end for
until no change in set repeat
    for Each instruction I do
        in[I] ← (out[I] - def[I]) ∪ use[I]
    end for
    for Each basic block B do
        out[B] ← ∪ in[S] (S ∈ Successor[B])
    end for
end until
```

Use-Def Chains and Reaching-definition Analysis

To the opposite of def-use chains, use-def chain is used to keep track of reachable definitions for a use of a variable. Reaching-definition analysis is the method to get use-def chains.

- For instruction I:
  - in[I]: reaching definitions going into I
  - in[I]: reaching definitions going into I
  - in[I]: reaching definitions going into I
  - kill[I]: definition at I
- For a basic block B:
  - in[B]: reaching definitions going into B
  - out[B]: reaching definitions coming out of B
- UD[I] = in[I] – out[I]

Algorithm 2 Reaching-def Calculation in CFG

for Each instruction I **do**
  in[I] $\leftarrow \emptyset$
  out[I] $\leftarrow \emptyset$
end for
until no change in set **repeat**
  for Each instruction I **do**
    out[I] $\leftarrow$ (in[I] - kill[I]) $\cup$ gen[I]
  end for
  for Each basic block B **do**
    in[B] $\leftarrow \cup$ out[P] (P $\in$ Predecessor[B])
  end for
end until

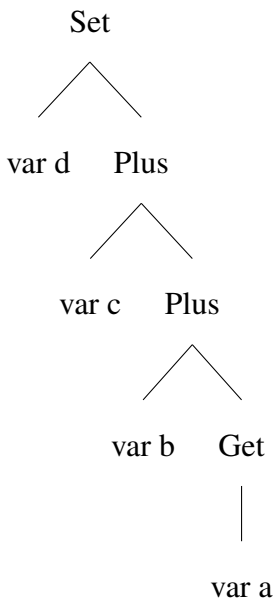
2.3 String Automaton Construction

The second step is to construct estimated input variables by using the result of data flow analysis. The Use-Def chain is particularly useful, because it allows a used variable to be traced back to where it is originally defined.

As string typed inputs are the most common interface inputs of PHP web applications, this section mainly focuses on string estimation. Christensen, Møller, and Schwartzbach [3] have proposed a promising algorithm for performing regular expression estimation for a string variable through data flow analysis. In this section a third-party package, dk.brics.automaton, is used. dk.brics.automaton is an open-source Java package implemented based on the research of Christensen, et al. It is language-independent, and supports most commonly used string operations. dk.brics.automaton is able to take flow graph composed of operators and operands (known characters), and generate a definite finite automaton (DFA), which is an estimated regular expression for the analyzed string.

In order to build a complete flow-graph for a string variable, a stack is used to mimic HHVM executions. During HHVM execution, variables and functions are pushed to top of a running stack. Each action (arithmetic operations, string operations, etc.) takes one or more elements popped from the stack, and pushes the new result to stack top. Sometimes when multiple operations are chained together, the stack actions will have a hierarchical form. For example, the stack action for d = (a + b + c) is:

Supported PHP String Operations
Concat
StrComp
Substr
ToUpperCase
ToLowerCase
Reverse
Trim
Split
Replace
StrToTime



2.4 Interface Discovery and Grouping

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 PHPCrawl, an open-sourced framework for crawling/spidering websites written in PHP. 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

Table 3.1: Interface Discovery Results

	# Identified Interfaces		
Project Name	PHPCrawl	Current	Improvement
CS4400	#	#	#
Email	3	3	0 (0%)
STP	12	16	4 (33%)
TodoList	#	#	#

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.

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 Engineerin 2007.