GRAY-BOX ANALYSIS OF WEB APPLICATIONS USING WAM ALGORITHM AND BURP SPIDER

SUMMARY

For the dynamic testing of websites, information gathering by crawling is a common first step. With the growing complexity of web applications, existing crawling techniques are increasingly ineffective. The dynamic content generation logic in modern web applications often hides several execution paths from typical crawlers that rely on parsing of plain HTML content in web pages to determine input points and values. To solve this problem, this project creates a gray-box tool which performs static analysis on the source code of a web page to determine information about its input interfaces. This information is used by an automated crawler to achieve better coverage of a website. This technique results in a more comprehensive site map of the target site, and therefore a more accurate picture of the attack surface. It thus increases the reliability of the vulnerability scan by reducing false negatives.

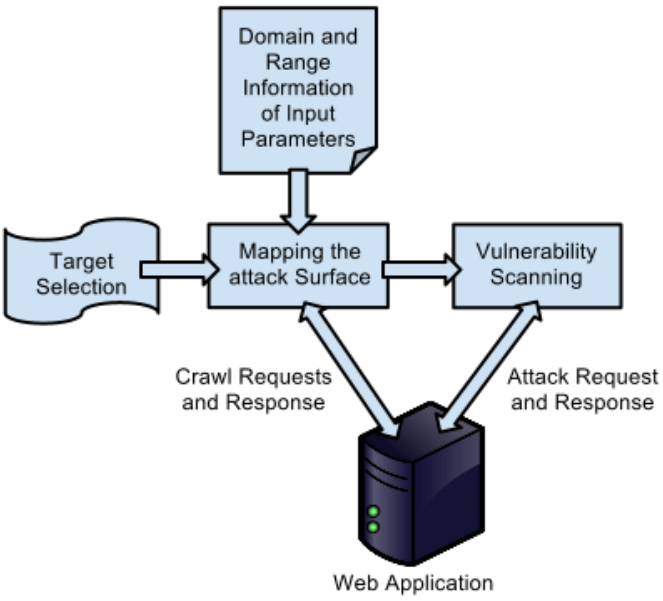
INTRODUCTION

Web applications are becoming increasingly ubiquitous and even more integrated with our day to day lives. A recent report by FireHost (provider of secure cloud hosting for web applications) revealed that its servers in the US and Europe blocked over 15 million cyber-attacks during Q3 2012 [1]. Web applications today carry more sensitive data and are thus lucrative targets for attackers. Also, a large number of IT professionals are continually churning out web applications meant to deliver products and services and thus ensure the success and profitability of a business. They are finding it hard to keep pace with the volume of code being produced and are increasingly short-staffed to ensure the quality and security of applications [2]. Thus automated testing of applications is not only appropriate, but necessary.

The first step in the automated black-box testing of a web application is “Information Gathering”. Most commonly, this step uses a web crawler to obtain information about the content of an application and its input vectors. A web crawler recursively visits all linked pages within a defined scope and analyzes the HTML content to identify input vectors. However, this technique is woefully inadequate to map a dynamic site, where HTML content is selectively exposed depending on the input provided. An automated crawler just does not have the information to provide the required input to trigger all paths within the application and thus obtain a comprehensive coverage. This can leave large parts of the application untested and vulnerable.

A suitable solution for this can be found in Gray-box testing methodologies. By definition, gray-box testing is a method where the internal structures and algorithms are partially known and are used to generate test cases which are then used in a black-box style functional test [3]. In our case, the specific problem of making the crawler aware of the various input paths to the application

can be solved by providing said information from static analysis of the application. The following figure illustrates the proposed technique.



This project aims to enhance the effectiveness of automated crawling of PHP based websites by providing additional information to the crawler by using a novel static analysis technique called WAM. WAM analyses the source code of a web application to identify interface information. It uses control flow graphs, data dependency analysis and string analysis to determine domain, and range information about interface parameters. This information can then be fed to the crawler which can now conduct a more thorough traversal of the control paths within the application. This enables better discovery of the attack surface of the website, a more thorough scan, and therefore lower false negatives. A plug-in for the BURP Spider is implemented as part of this solution that can read the output of this analysis to improve its crawl coverage of PHP based websites.

The remaining sections of this report are as follows. Section 2 talks about the background and motivation for this work. Sections 3 covers details of the Static Analyzer for PHP, and the details of the BURP plug-in. Section 4 describes the usage of the tool via a test example. Section 5 discusses the advantages of this tool and section 6 outlines the scope for future work.

BACKGROUND

PAST WORK

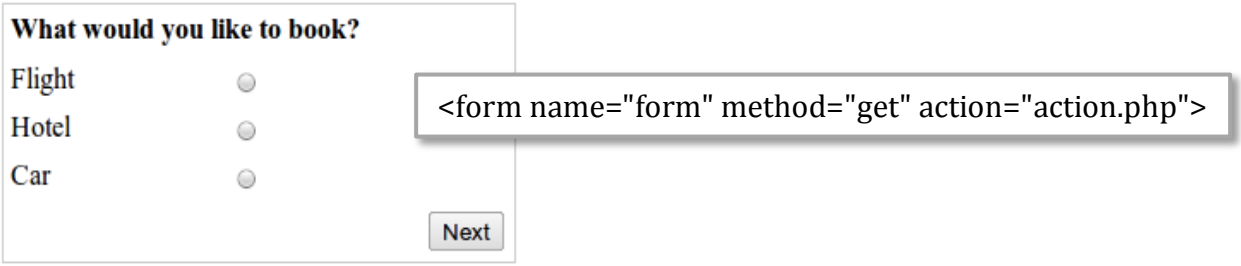
The static analysis technique used in this project to obtain design information about the application is WAM analysis. It was proposed by W. G. J Halfond and A. Orso in [4] as a technique for automatic discovery of web application interfaces. Input interfaces of an application are those points at which the application reads external data. This data can come from user input (e.g. form fields) or by fetching some other state information (e.g. cookies). In dynamically generated sites, these inputs are used to expose different content to different users. Thus, these values directly affect the flow of control in the program. In the same paper, a prototype tool to analyze Java servlet

based websites using the WAM algorithm reported 30% increase in block coverage, and 48% increase in branch coverage.

SDAPT [5] is another tool that incorporates the WAM technique to analyze Java based web applications. It uses the information gathered about the input interfaces in the application to generate better attacks for SQLIA and XSS. It also uses WASP [6], [7], a dynamic technique for runtime monitoring, to perform better response analysis of the attacks generated.

MOTIVATING EXAMPLE

Before describing the details of the tool, a simple example is presented below to illustrate the problem and its solution. Shown below are a simple form and the corresponding HTML code.



The submit button of this form sends the following request.

```
GET /action.php?actionselected=<paramvalue>&Next=Next
```

In the above URL, we can see that "actionselected" and "Next" are the parameters that make up the interface through which the application gets external data. Thus, these parameters are part of the ***"input interface"*** of the website. The code for *action.php* is given below.

```
1. <?php
2.      $action=$_GET['actionselected'];
3.      if($action == "Flight"){
4.              //some action
5.      }
6.      elseif($action == "Hotel"){
7.              //some other action
8.      }
9.      elseif($action=="Car"){
10.             //some other action
11.     }
12.     else{
13.             echo "We don't have that choice- We can only book Hotels, Flights, or Cars";
14.     }
15. ?>
```

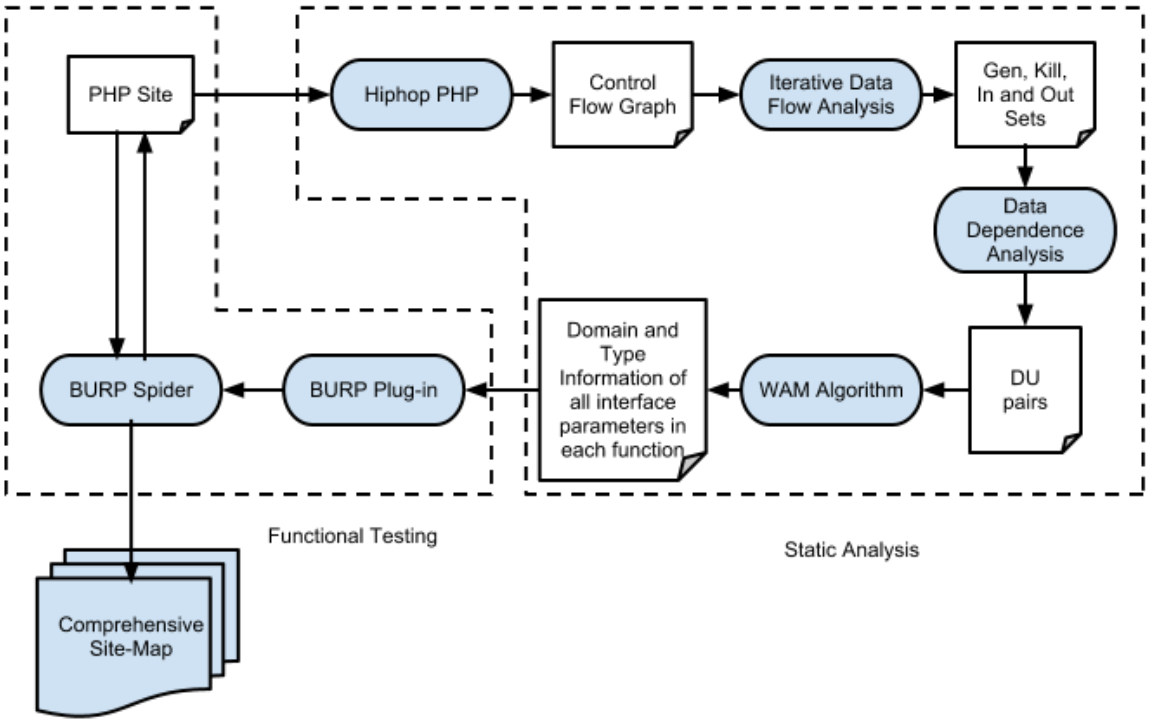
On line 2 of the above code, the page reads the parameter value by dereferencing the array $_GET. This corresponds to the ***"parameter function"*** call in WAM (e.g. *request.getParameter('actionselected'))*. In PHP, some other commonly used parameter functions

are $_POST, $_REQUEST and $_COOKIE. An unaware crawler sends random or predefined form inputs with this request whose value gets called using the parameter function. Hence the crawler is unable to trigger the first three paths of the if-else condition on this page. Only the last condition gets executed every time. Thus the unaware crawler is able to cover only 1/4th of the webpage.

Lines 3, 6, and 9 are ***"domain operations"***. They give us information about the significant or acceptable values for this parameter. The function of static analysis in this project is to collate all such domain and range information about each interface parameter across complicated control flow logic to arrive at the values (or type of values) that a parameter needs to take in order to trigger all paths of execution within the application.

APPROACH AND IMPLEMENTATION

This project creates a tool to achieve the above functionality. The design of the tool is illustrated by the diagram below.



PHASE 1: PHP STATIC ANALYSER

The static analyzer implemented for this tool can be broken down into the following modules.

GENERATION OF CONTROL FLOW GRAPH

A control flow graph (CFG) is a representation of all the paths that a program may take in its execution. It is typically implemented as a directed graph in which each node represents a basic block, and each edge represents the flow of control from one block to another. A basic block is

defined as a set of consecutive instructions in a program that has no jumps or jump targets. That is, it has only one point where the flow of control enters, and only one point where it exits without halting or branching in between.

To generate control flow graphs for PHP code, we used Hiphop for PHP [8]. This is a set of PHP execution engines developed by Facebook. The intended goal for Hiphop PHP was to increase the efficiency of Facebook's PHP based web site code by converting it into C++. An intermediary step in this direction is to generate control flow graphs for various functions. This project re-uses the control flow graphs generated by Hiphop and the associated functions to traverse the CFGs as a base to perform static analysis on PHP code.

ITERATIVE DATA FLOW ANALYSIS

This tool performs a 2-part preprocessing of the information contained in the control flow graph obtained from Hiphop-PHP. The first of these is to compute the "Reaching Definitions". A "definition" in this context is any statement in a program that results in the creation (or re-creation) of a variable. To compute "Reaching Definitions" at a point P is to compute all the definitions in the program that can traverse some path in the CFG without being "killed" and therefore are available at point P. To compute reaching definitions, we must compute four sets of definitions for each node in the CFG.

- GEN set- This is the set of definitions created/ generated inside the node.
- KILL set- This is the set of definitions that are destroyed within a particular node (perhaps because of recreation) and are not accessible after it.
- IN set- This is the set of definitions that were created in predecessor nodes and are available at the beginning of a node.
- OUT set- This is the set of definitions that are available at the exit of each node (after accounting for the incoming, generated and the killed definitions) and therefore available to the successors of the node.

To compute these sets, the tool traverses the control flow graph to record all the initialization and assignment statements. They are then added to the GEN and KILL sets for the respective nodes. Next, the tool implements the "ReachingDefs" algorithm from [9] to compute the IN and OUT sets for each node. The algorithm is reproduced below for reference.

algorithm ReachingDefs

Input. A flow graph for which KILL[n] and GEN[n] have been computed for each node n.
Output. IN[n] and OUT[n] for each node n.
Method. Use an iterative approach, starting with IN[n] = {} for all n, and converging to the desired values of IN and OUT. Iterate until the sets do not change; variable change records whether a change has occurred on any pass.

```
1. for each node n do OUT[n] = GEN[n] endfor
2. change = true
3. while change do
4.         change = false
5.         for each node n do
6.                 IN[n] = ∪OUT[P ], where P is an immediate predecessor of n
7.                 OLDOUT = OUT[n]
8.                 OUT[n] =GEN[n]∪(IN[n]–KILL[n])
9.                 if OUT[n] = OLDOUT then change = true endif
10.        endfor
11.endwhile
```

DATA DEPENDENCE ANALYSIS

This is the second step in the preprocessing where we compute the Definition-Use pairs for each variable. A "Use" is any reference to a variable where it is fetched and is either directly used in computation or otherwise affects the flow of control in the program. A definition-use pair of a variable 'V' is an ordered pair (D,U) such that definition D of V can follow some path in the control flow graph to reach the point at U without being killed, and is in fact used there. The Def-Use pairs are computed using the algorithm from [9]. The algorithm is reproduced below for reference.

algorithm ComputeDefUsePairs

Input. A flow graph for which the IN sets for reaching definitions have been computed for each node n.
Output. DUPairs: a set of definition-use pairs.
Method. Visit each node in the control flow graph. For each node, use upwards exposed uses and reaching definitions to form definition-use pairs.

```
1. DUPairs = {}
2. for each node n do
3.         for each upwards exposed use U in n do
4.                 for each reaching definition D in IN[n] do
5.                         if D is a definition of v and U is a use of v then
6.                                 DUPairs = DUPairs ∪ (D,U) endif
7.                 endfor
8.         endfor
9. endfor
```

To be usable by the WAM algorithm, these DU pairs have been implemented as a chain using a Multimap in C++. This ensures that given two arguments V, N (where V is a variable and N is the node where it was defined), the function can return those set of nodes S that are reachable by the definition V and which use the variable V. The use of variable V itself can be in the definition of another variable, thus leading to another set of nodes reachable by this new definition. Such a linked set of DU pairs is called a DU chain.

ANALYSIS BASED ON WAM

Here, the tool traces each interface parameter and its use through the application to derive information about the possible data values it can take. In accordance with the WAM algorithm, this module has two main phases- Discover Domain Information and Computing Interfaces.

In the first phase, the tool computes annotations for each node that either directly calls a parameter function (PF) or calls a method that invokes a PF (directly or indirectly). It identifies the return variable that is defined by the return value of the PF call and traces it through the DU chain. The domain information is obtained by identifying instances in the DU chain where the variable (directly, or through another variable defined using it) is cast into a particular data type or compared to specific values to affect the flow of execution. For instance, if a variable "X" defined by the return value of a parameter function, is type cast using the int() in PHP, then we know that at least a subset of all legitimate values of X need to be numeric. Also, if X is compared to specific values during the flow of execution, then we know that those values are definitely present in the domain of values that X can take. Thus, various uses of X down the DU chain can give us information about its expected Domain and range. This information is recorded by a recursive function in the form of annotations at the node in which X was defined.

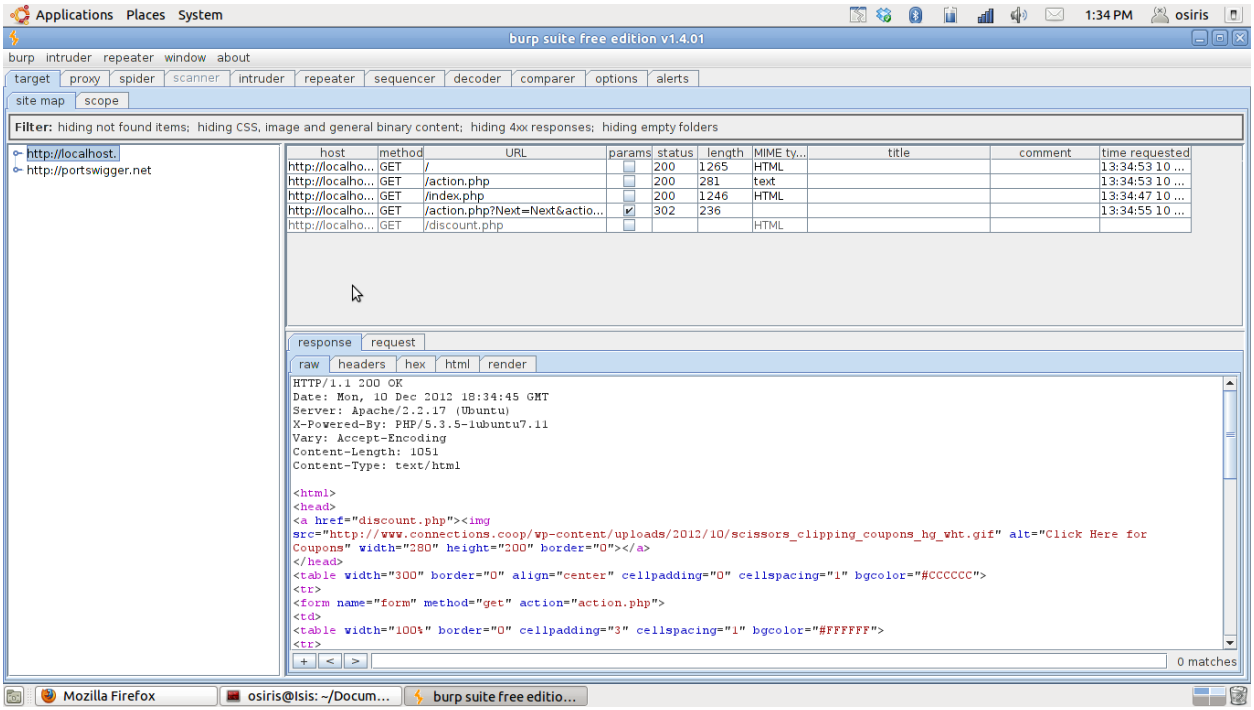
In the second phase, the tool associates the annotated information of the variables with the interface parameters (IPs). A second pass over the control flow graph is made and each annotation of a variable is associated with the interface parameter that was used to initially define the variable. This information is written out to a file in a predefined format to be read by the BURP plug-in.

PHASE 2: BURP PLUG-IN

BURP is a popular suite of tools that are used to test the security of web applications [10]. It has several tools, including a web proxy, a web spider and an intruder. It also provides BurpExtender [11], which is an interface to extend its functionality. This project builds an extension for BURP called Hiphop Crawler. This extension is coded in java and implements three out of the five interface methods provided by BURP. This plug-in reads the output of the above static analysis in a file from a designated folder. It can be accessed using a custom menu option in the right click menu inside burp. When used on a particular site, it starts the spider tool from that page. It intercepts and reads all the requests generated during the spidering process and stores the interface variable names along with the URL which used them. Once the initial spidering is complete, this plug-in matches all the interface variable names encountered in the crawl with those present in the file from the static analyzer. It reads the domain values associated with each parameter from this file and uses them to construct additional requests. These requests are then sent to the spider enabling it to traverse additional paths and thus provide a comprehensive coverage of the site.

TEST EXAMPLE

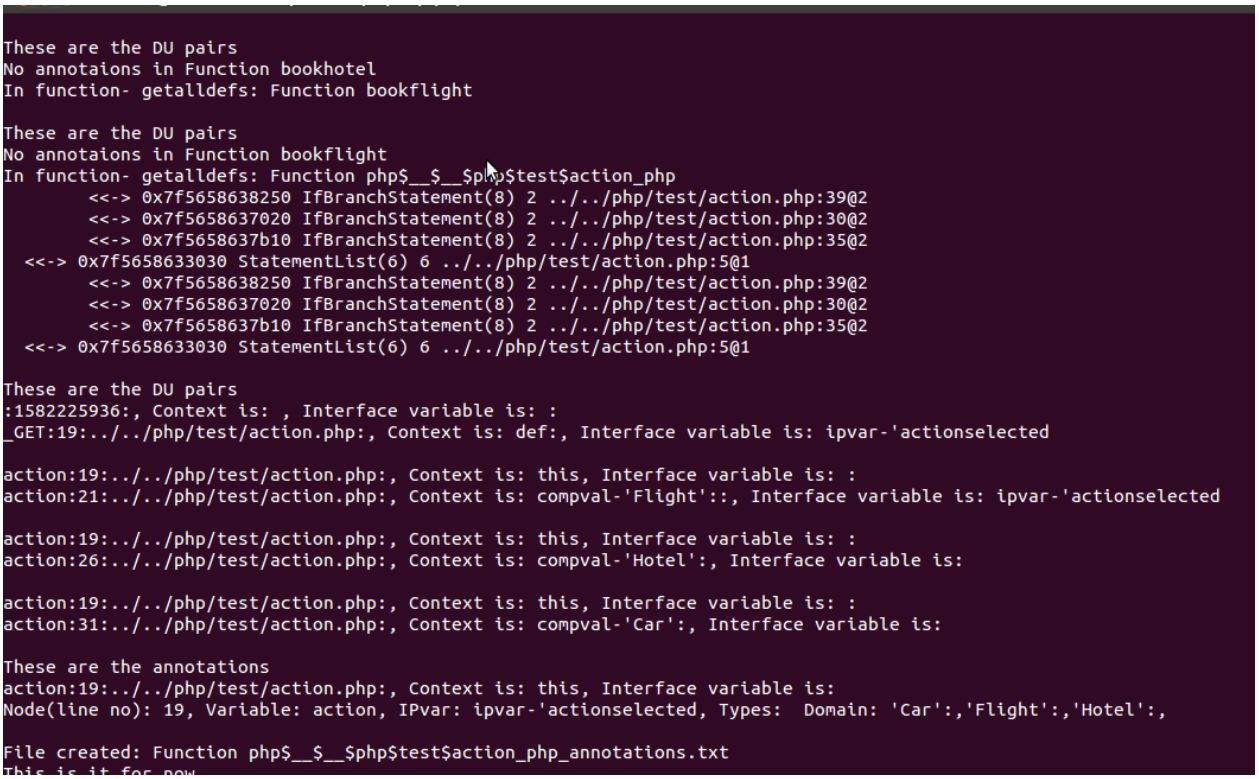
A dummy travel booking site was created for the testing of this tool. Below is a snapshot of a vanilla crawl done on it using BURP Spider tool.



The source code for the Static Analyzer has been uploaded to GitHub. Approximately 36 libraries are needed to compile this code. The full instructions for that can be found on the Readme page of the Git repository. Once compiled to a binary, the following command can be used to analyze a PHP file.

```
your/path/to/binary --nofork 1 --staticanalysis 1 -i your/path/to/PHP/file
```

To test the tool, we ran the static analyzer on our dummy site using the above command. The snapshot below shows the output.



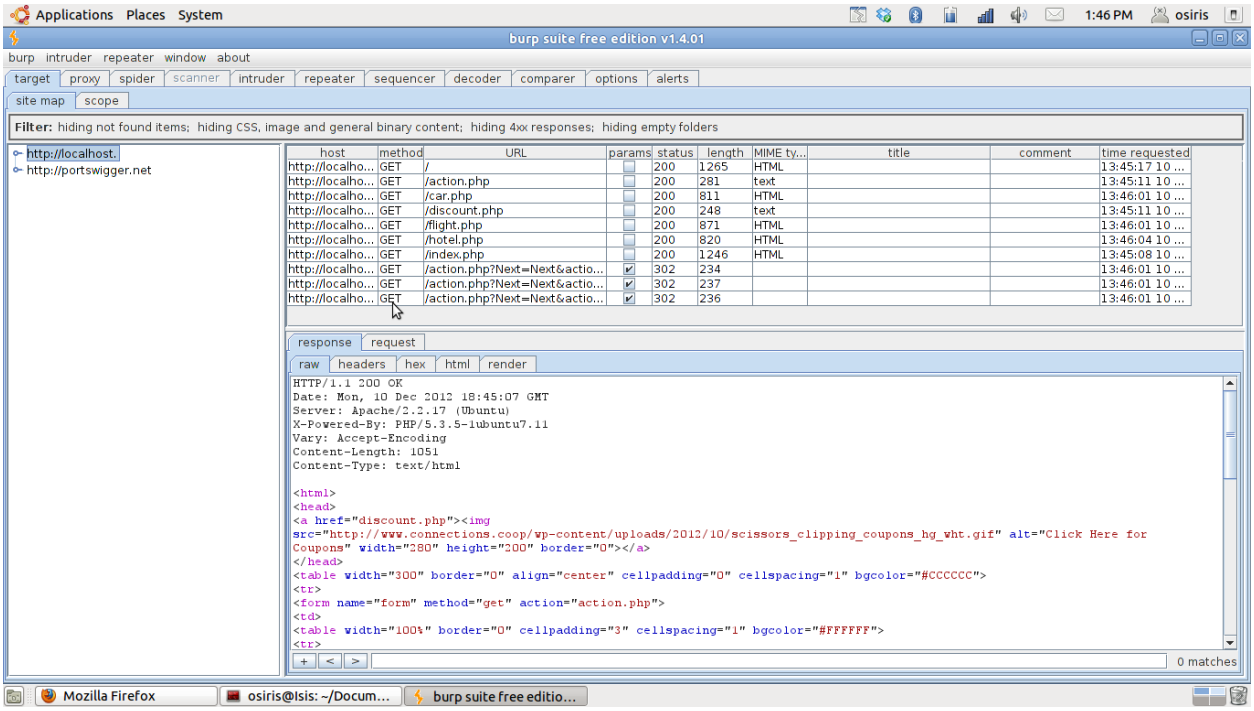
This will create a .csv format file for each function for which annotations were created. The files are named with the following convention.

Function <function_name>_annotations.txt

This analysis should be performed for the entire source code of the website to be tested. The output files will need to be manually copied to a location where they can be accessed by the BURP Spider. The source code for the BURP plugin and the compiled .jar file are available on GitHub. To run the second phase of the analysis, copy all the "_annotations.txt" files into the input folder that has been created along with Hiphop Crawler. In this test, the file named "Function php$_$_phptest$actio_php_annotation.txt" was created and fed into the plug-in. The BURP Spider can be run with the plug-in, using the following command.

java -jar hiphopcrawler.jar burp.startBurp

This should fire-up an instance of Burp with the Hiphop Crawler Plug-in. To use the second phase of the tool, configure your browser to route requests through the Burp Proxy. Navigate to the site to be crawled in the browser window. This site will get listed in the "site map" tab under the "target" tab in Burp. Here, right click on your website and select "Hiphop Crawler". The plugin will now read data from the "input" folder to send additional requests, resulting in a more comprehensive crawl of the website. Below we have a snapshot of the final crawl of the dummy site.



We can see that the Hiphop crawler has doubled the known site map by discovering additional pages in the dummy site.

ADVANTAGES

- Better coverage of code in dynamic websites- Pure black-box crawling of modern web applications puts several paths within the application out of the crawler's purview. This tool allows the crawler to see the various data flow paths within the application and make them available for the next stages of manual or automated testing.
- Fewer false negatives- In conventional test using a spider tool, due to lack of thoroughness in the initial information gathering, only a small number of test inputs can be generated. Thus, several aspects and parts of the application may remain untested, and therefore vulnerable. This will lead to higher false negative rates. By providing a more comprehensive site map of the target site, this tool enhances the effectiveness of existing testing techniques.
- Plug-in to popularly used crawler increases adoption and use- This tool is implemented as a plugin for BURP, the interface and functioning of which is familiar to a large subset of the target audience. This increases the usability of the tool.
- More reliability on test results- Based on the increase in percentage coverage and lower false negative rates, the reliability of this tool increases, giving higher security assurance.

SCOPE FOR FUTURE WORK

- Include Inter-procedural flow analysis- The WAM algorithm is capable of summarizing domain and range information about interface parameters across functions. For example, a

website may be constructed such that the parameter functions read the interface parameters in one function and use the return values to call another function which does the actual processing (and therefore contains all the domain/range information). The WAM algorithm uses a Call graph of the entire application to allow this information to be associated with the interface parameter regardless of function boundaries. Due to limitations in the structure of Hiphop PHP, this tool does not yet incorporate this functionality. However, by generating a call graph of the application outside the structure of Hiphop and using it to perform these steps would improve the advantage given by this tool.

- More automation- This tool is essentially two distinct parts- the static analyzer is implemented in C++ as an addendum to Hiphop, while the BURP extension uses Java. The output of the first phase needs to be manually fed to the second. These two can be packaged into a single application using a shell script. This would make the tool easier to use.
- Use parameters based on requests other than GET- Currently, the second phase of this tool sends additional requests using only those interface parameters that are accessed using GET requests. To use POST (and other) requests is more round-about since changing the parameters in a post request does not change the request URL, which is what is read by the Spider tool. In order to achieve better results by using these parameters, the plug-in would either need to write the new domain values to the automatic form input component of BURP, or would need to externally parse the response HTML and add it to the site tree.

ACKNOWLEDGEMENTS

This project was completed under the guidance of Prof. Alex Orso and Shauvik Roy Choudhary (College of Computing, Georgia Institute of Technology). The author would like to thank them for the invaluable support they provided.

REFERENCES

[1] "Q3 2012 FireHost Web Application Attack Report Shows Marked Increase In Cross-Site Attacks | FireHost." [Online]. Available: http://www.firehost.com/company/newsroom/web-application-attack-report-third-quarter-2012. [Accessed: 10-Dec-2012].

[2] "Half of companies surveyed report Web application security problems." [Online]. Available: http://www.networkworld.com/news/2012/091812-web-application-security-262520.html. [Accessed: 10-Dec-2012].

[3] "Gray Box Testing | Software Testing Fundamentals." [Online]. Available: http://softwaretestingfundamentals.com/gray-box-testing/. [Accessed: 10-Dec-2012].

[4] W. G. J. Halfond and A. Orso, "Improving test case generation for web applications using automated interface discovery," *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*, p. 145, 2007.

[5] W. G. J. Halfond, S. R. Choudhary, and A. Orso, "Improving penetration testing through static and dynamic analysis," no. April, pp. 195–214, 2011.

[6] W. G. J. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering - SIGSOFT '06/FSE-14*, p. 175, 2006.

[7] W. G. J. Halfond, A. Orso, and I. C. Society, "WASP : Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation," vol. 34, no. 1, pp. 65–81, 2008.

[8] "HipHop for PHP - Wikipedia, the free encyclopedia." [Online]. Available: http://en.wikipedia.org/wiki/HipHop_for_PHP. [Accessed: 10-Dec-2012].

[9] M. J. Harrold, "Representation and Analysis of Software," pp. 1–19.

[10] "Burp Suite." [Online]. Available: http://portswigger.net/burp/. [Accessed: 10-Dec-2012].

[11] "Burp Extender." [Online]. Available: http://portswigger.net/burp/extender/. [Accessed: 28-Sep-2012].

IMPORTANT LINKS

The source code for this project is available on GitHub using the following links.

Static Analyzer using Hiphop-PHP: https://github.com/gatech/php_analysis
BURP Plug-in (pre-compiled .jar file): https://github.com/bsoman3/hiphopcrawler
BURP Plug-in source code: https://github.com/bsoman3/hiphopccrawler_sourcecode
Dummy travel booking site for testing: https://github.com/bsoman3/Dummytravelsite