Dynamic Taint Analysis for PHP in Quercus

Xavier NOUMBISSI NOUNDOU

xavier.noumbis@gmail.com

Abstract

Software vulnerabilities are security threats that exist in an application and may enable users to exercise unauthorized control of the program through supplied input. Unauthorized control of an application may be gained using techniques such as *buffer overflow*, *SQL injection*, *cross site scripting, etc*. Checking for software vulnerabilities in the context of web applications has become extremely important as businesses observe an increasing reliance on web applications. This paper presents a dynamic taint analysis for PHP. The analysis is implemented in the *Quercus Interpreter*, which is an interpreter for PHP 5 completely written in the Java programming language.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms Software Vulnerabilities, Web Applications, Taint Analysis, Dynamic Analysis

Keywords taint analysis, PHP, Java, dynamic analysis

1. Introduction

Human activities increasingly rely on software systems. Moreover, the ease of web applications deployment has rendered web browsers to a very popular software interface. However, the ubiquitousness of web applications poses new challenges for software security.

Web applications are an easy target for malevolent users because they are available to the public at almost all time, and are thus exposed to a wide range of security attacks. SQL injection and cross site scripting attacks are among the most commonly reported web application security threats. Both attacks are caused by insufficient *data validation* and *data sanitization*. Data validation is the process of checking that input data has the expected form. For instance, checking that a string input has the format of an email address. Data sanitization makes sure that validated input data is safe in a particular context. For instance, escaping string input before using them for an SQL query to the database.

A function that sanitizes an application's external input is called a *sanitizer*. Unsanitized external input is said to be *tainted* in the context of taint analysis. An input becomes untainted only when it has been sanitized[1].

This paper presents a dynamic taint analysis implemented in the `Quercus Interpreter for PHP`[7]. A taint analysis can be used to prevent the use of unvalidated and unsanitized input by an application. An application input may come from the network, the file system, the operating system environment, etc. Our analysis checks for tainted strings originating from users input via a PHP page, and emits a warning whenever tainted values are used by the application without validation and sanitization. This may help developers in removing unsafe code from their PHP programs, and thus improve the robustness of interpreted web applications.

`Quercus`[2] is completely written in the Java programming language, and thus enables a tighter integration between Java and PHP. For instance, developers are able to use available Java libraries and technologies from their PHP code. Sample technologies are *Java Message Service*, *Hibernate*, and *Spring* among others.

This paper is organized as follows: Section 2 gives a short introduction to `Quercus`. We describe our analysis in Section 3 and our testing methodology in Section 4. Section 5 discusses related work and we conclude in Section 6. Finally Appendix A gives additional information for potential `Quercus` developers and on how to configure our analysis.

2. The Quercus PHP Interpreter

The increasing popularity of the Java programming language has make it tempting to implement PHP interpreters in Java. Implementing PHP interpreters in Java should enable PHP code to reuse the huge quantity of existing Java technologies and libraries.

The company `Caucho Technology` [3] tried the experiment and developed `Quercus`[7], which is an interpreter for PHP 5 that is completely written in Java. `Quercus` was primarily developed to run within *Resin*, which is a Java application server developed by `Caucho Technology`. Since `Quercus` is developed as a servlet, it can virtually run in any Java application server. For instance, it is also known to run with *Apache Tomcat*[8] and *Websphere*[1].

We developed our taint analysis using `Quercus` as a module of `Resin 4.0`. We downloaded `Resin 4.0` from the subversion repository `svn://svn.caucho.com/home/svn/svnroot/resin/trunk` on May 9th of 2013.

[1] In the remainder of this paper, we will assume that data has been validated whenever it is sanitized.

[2] we will use `Quercus` for short to mean the `Quercus Interpreter for PHP` in the remainder of this paper

[3] `http://www.caucho.com/`

The `Quercus` module is located in the source tree at `RESIN_SOURCE/modules/quercus` and has around $140,000$ lines of Java code. Appendix A gives additional information for developing with `Quercus`.

3. Taint Analysis

This section describes how the main parts of a taint analysis (sources, sinks, and taint propagation) are reflected in our implementation.

Taint Sources Our analysis handles values from PHP built-in array variables `$_GET`, `$_POST`, and `$_COOKIE` as sources.

Taint Sinks Analysis users define sink functions by registering their name in the configuration file located at `RESIN_HOME/webapps/ROOT/WEB-INF/ta-sink.cfg`.

A call to a sink function with a tainted value as argument emits a warning message in the `Resin` server log file: `RESIN_HOME/log/jvm-app-0.log`. Figure 1 illustrates the content of a sample sink configuration file `ta-sink.cfg`:

```
user@server:$RESIN_WEB_INF> cat ta-sink.cfg
printf
mysql_db_query
mysql_query
mysql_select_db
mysqli_query
```

Figure 1. Sample Sink Function Configuration File

Our analysis also handles standard PHP print functions (`echo`, and `print`) as sinks. Developers do not need to specify them in `ta-sink.cfg`.

Taint Propagation Taint propagation determines how taint information flows in the program. We implemented taint propagation for the following:

- Assignment operator (`=`).
- Arithmetic operators: addition (`+`), subtraction (`-`), division (`/`), multiplication (`*`), increment (`++`), decrement(`--`).
- Arithmetic bit operations: And, Or, Xor.
- Type promotion and conversion of integer and floating point numbers to PHP strings.
- String concatenation.
- Appending a character to a string.
- Retrieving of a character from a string.
- Retrieving of a substring from a string.

Table 1 in Appendix A summarizes which expressions and statements propagate taint information and additionally shows in which Java classes of `Quercus` we performed the corresponding modifications.

Our analysis also tracks the use of developer registered sanitizer functions and modifies tainted values accordingly: when a tainted data is sanitized at program point p_s, it is marked as untainted from p_s onwards, unless it gets tainted in some subsequent program points p_q $(q > s)$. In PHP code, sanitization of tainted data may happen with the use of functions such as `mysql_real_escape_string`, `htmlentities`, etc.

Figure 2 shows the content of a sample sanitizer configuration file `ta-sanitizer.cfg`.

```
user@server:$RESIN_WEB_INF> cat ta-sanitizer.cfg
mysql_real_escape_string
filter_var
htmlentities
```

Figure 2. Sample Sanitizer Function Configuration File

Our implementation assumes that the value to be sanitized is always the first parameter of the function. In cases some sanitizer functions use different parameter positions, further modification of the code to specify relevant parameter numbers in the configuration file `ta-sanitizer.cfg` should be trivial.

Currently, the analysis does not relate sanitizer functions with their context. For instance, Before a string is used in a database query, it should be escaped using the function `mysql_real_escape_string` (e.g. `select * from students where age =mysql_real_escape_string($_GET[age])`).

It follows that some sanitizers are expected to be used in particular scenarios. Our implementation does not take this into account, and therefore always mark as untainted the resulting value of a sanitizer function call.

Missing Taint Analysis Features Our taint analysis does not:

- Track external input from sources other than the built-in arrays `$_GET`, `$_POST`, and `$_COOKIE`. This means that values from the network (e.g. HTTP Headers), environment variables, stored input (e.g. uploaded files), etc. are not tracked by the analysis.

  Implementing taint propagation for taint sources we currently do not support would require developers to identify the location of their implementation in the code. This identification will enable to mark the first values from these sources as tainted. However, taint tracking for the operations listed in Section 3 are not to be implemented, as our implementation already takes care of this.

- Provide support for implicit taint propagation (i.e. control flow taint propagation). That is, our analysis do not propagate taint information in branches whose conditional expression uses a tainted variable or value.

  The implementation of this feature should not require a considerable amount of effort. In the class `com.caucho.quercus.statement.IfStatement`, the developer would need to check during the evaluation of the conditional expression whether any of its parts contains tainted values or variables[4]. If this is the case, all variables in the block[5] to be executed next have to marked tainted.

4. Evaluation

This section presents the goals and methodology we use to evaluate the effectiveness of our analysis implementation. We use the

[4] The evaluation of the conditional expression (`IfStatement._test`) occurs in the method `IfStatement.execute(Env)`).

[5] `IfStatement.__trueBlock` or `IfStatement.__falseBlock` depending on the evaluation of `IfStatement._test`.

HttpUnit[6] framework for automating our tests. The main purposes of our evaluation are to make sure that:

- Taint sources and sinks as given in Section 3 are properly identified.
- The operations listed in Section 3 correctly propagate taint information.

```
1  <?php
2  $req1 = $_GET["req1"];
3  $result = 'a string from GET ' . $req1;
4  printf ("Result: %s\n", $result);
5  ?>
```

Listing 1. PHP Script to Evaluate Taint Propagation for String Concatenation

The PHP code in Listing 1 stores a value from array `$_GET` in variable `$req1` at line 2. Variable `$req1` is then concatenated with the string constant `"a string from GET"` and the result stored in variable `$result`. Since the concatenation of a tainted and an untainted string shall produce a tainted value, we expect variable `$result` to be tainted from line 3 onwards. Furthermore, the call to the sink function `printf` in line 4 should warn the developer that the tainted variable `$result` is used without prior sanitization as argument of a sink function.

Testing Methodology We now describe our testing methodology using the code in Listing 1 as illustrating example.

- We create a PHP page for each feature to test. Listing 1 is for instance the code for checking that string concatenation propagates taint information.
- Then we create a text file which specifies the test expectation in terms of tainted variables and line of code.

  For instance, we would write `"4:$result:tainted"`in the text file of expected results for Listing 1. The given string denotes that variable `$result` must be tainted at line 4[7].
- We use `HttpUnit` API to create a web interaction between `JUnit` and `Quercus`. Our web session consists in accessing the PHP code under test and pass it a value for the `$req1` parameter. The interpreter processes our request and writes a response page with taint information (we have modified the `Quercus` interpreter to produce specification strings as the one described in the previous item).
- Our test code then reads the text file with the expected output and compares it with the output from the response page generated by the `Quercus` interpreter. The test is successful if and only if both string outputs match.

Interactive Help for Developers To help developers, we use the tools *Firebug*[6] and *FirePHP*[3]. `Firebug` is a development plugin for the `Firefox` web browser and allows `JavaScript` code to log events into a debugging console. `FirePHP` extends `Firebug` and enables developers to write logs to `Firebug`'s JavaScript console from PHP code.

Our analysis instruments PHP pages generated by `Quercus` with `FirePHP` API calls to write logs to `Firebug`'s JavaScript console. We write a log for the following events:

- The use of tainted PHP variables in sink functions.
- The sanitization (or untainting) of PHP variables.

Figure 4 in Appendix A shows a screenshot with `Firebug` warning (yellow entry) and info (blue entry) logs generated by our taint analysis.

5. Related Work

To the best of our knowledge, no taint analysis has been developed for the `Quercus PHP interpreter` in the past. In the following, we present closely related taint analyses.

Taint Support for PHP [9] is a tool that checks PHP code for tainted variables and helps developers finds and remove HTML script injection, SQL or shell code injection, and PHP control hijacking. Similarly to our analysis, the tool performs a dynamic analysis within a PHP interpreter. The tool is implemented in the C programming language.

Pixy [4] statically scans for cross site scripting vulnerabilities in PHP scripts. `Pixy` and our project have similar goals. However `Pixy` performs a static analysis while our analysis is dynamic and runs during program execution. `Pixy` implements a flow-sensitive, interprocedural and context-sensitive dataflow analysis, which is based on an alias and literal analysis. `Pixy` has a false positives rate of 50%.

Ardilla [5] is a tool for automatic creation of inputs that expose SQL injection and cross site scripting vulnerabilities. `Ardilla` performs a dynamic taint analysis, generates input for the web application and mutates the input to create concrete exploits. `Ardilla` stores taint information using a novel combination of concrete and symbolic database.

Dytan [2] is a generic framework for implementing dynamic taint analyses. `Dytan` generates dynamic analyses that perform on x86 binaries. The source code of the analyzed application does not need to be available. Developers specify an analysis by giving taint sources, taint sinks, and a propagation policy. Taint sources can be variable names, function-return values, and data read from I/O stream such as a file or a network connection. Taint sinks are specified by memory/code location or based on the usage scenario. A usage scenario specification takes into account whether the developer wants check a taint information before the execution instructions of a given type (e.g., a jump instruction).

6. Conclusion

We have presented a dynamic taint analysis for PHP. Our analysis is implemented in the PHP interpreter `Quercus`. `Quercus` is written as a Java servlet. We used the `Resin` application server of `Caucho Technology` for our development.

Our analysis identifies input values from PHP built-in array variables `$_GET`, `$_POST`, and `$_COOKIE` as taint sources. Taint sinks are specified in a configuration file.

The analysis supports explicit taint propagation and does not support control flow taint propagation. We have implemented taint propagation for the following:

- Assignment operator (=).

[6] `http://httpunit.sourceforge.net/`

[7] a string `"7:$result:untainted"` would mean: at line 7, variable `$result` gets untainted.

```
user@server:$RESIN_HOME> bin/resin.sh start \
"-Xdebug -Xnoagent -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=77000"
```

Figure 3. Start of Resin with the debugging option

- Arithmetic operators: addition (+), subtraction (-), division (/), multiplication (*), increment (++), decrement(--).
- Type promotion and conversion of integer and floating point numbers to PHP strings.
- String operations: concatenation, appending of a character, retrieving of a character, getting a substring.

During the analysis, the usage of tainted values in sink functions is logged on the web server and to a Firebug JavaScript console. We also provide a set of Junit tests we used to check the accuracy of our implementation. Our implementation and the Junit tests can be found at https://ece.uwaterloo.ca/~xnoumbis.

Future work may include the identification of other program external input as taint sources (e.g. reading of files) and the implementation of control flow taint propagation. It would also be of great interest to analyze the time overhead caused by the analysis, and to test the analysis on other Java application servers (e.g. Apache Tomcat).

A. Additional Development Information

This section gives information on how to configure our taint analysis. It also provides useful information for doing development with the Quercus PHP interpreter in the Resin Java application server.

A.1 Taint Analysis Configuration

Quercus users may enable or disable our taint analysis by setting the value of the servlet initialization parameter *run-taint-analysis* to ON or OFF respectively. This happens in the configuration file RESIN_HOME/webapps/ROOT/WEB-INF/web.xml.

To run the HttpUnit tests we provide, users must set the servlet initialization parameter *taint-analysis-test-mode* to ON. This is done in the configuration file RESIN_HOME/webapps/ROOT/WEB-INF/web.xml. *taint-analysis-test-mode* has to be set to OFF when HttpUnit tests are not intended to run.

Sink functions are defined in the configuration file RESIN_HOME/webapps/ROOT/WEB-INF/ta-sink.cfg.

Sanitizer functions are defined in the configuration file RESIN_HOME/webapps/ROOT/WEB-INF/ta-sanitizer.cfg.

Developers may specify which php.ini file the Quercus interpreter shall use. We did this by adding a php.ini in the WEB-INF folder and by registering its location in the file RESIN_HOME/webapps/ROOT/WEB-INF/web.xml.

A.2 Firebug and FirePHP Usage

Firebug[8] and FirePHP[9] are to be installed as a Firefox plugin. We used version FirePHP 0.7.4 and Firebug 1.12.3 during our development. It is important to use compatible Firebug and FirePHP versions. Both tools should be used with a matching Firefox version, as advertised on their web pages.

Note that it is necessary to have both the *Console* and *Net views* of the Firebug console enabled in order to view the logs.

Developers must install the FirePHP library on their machine for Quercus to successfully execute PHP statements instrumented by the taint analysis to produce Firebug logs. Otherwise, logs would not appear in the debugging console. We used *Pear*[10] to install the FirePHP library on our Linux machine.

A.3 Debugging of Resin with Eclipse

We debugged Resin during the development of our analysis by performing the following steps:

- Create a project in Eclipse for Resin.
- Start Resin with the script RESIN_HOME/bin/resin.sh and pass it -Xdebug-Xnoagent-Xrunjdwp:transport=dt_socket, server=y,suspend=n,address=77000 as argument. Any free socket number can be assigned as value of address. Figure 3 illustrates a start of Resin with the debugging option.
- Create a remote debug session in Eclipse and use the same socket address as the one passed to the bash script resin.sh[11].
- After successfully preforming the previous steps and setting breakpoints in Eclipse, the access of PHP pages hosted in Resin triggers the Eclipse debugger at the chosen breakpoints.

[8] http://getfirebug.com/downloads

[9] https://addons.mozilla.org/en-US/firefox/addon/firephp/

[10] pear.php.net

[11] http://www.ibm.com/developerworks/library/os-eclipse-javadebug/

References

[1] Peter Bruns. Running cauchos quercus php java interpreter on websphere application server for z/os with db2 udb for z/os. `http://www.ibm.com/developerworks/websphere/library/techarticles/0809_bruns/0809_bruns.html`, called on September 26, 2013.

[2] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[3] Christoph Dorn. Firebug extension for ajax development. `http://www.firephp.org/`.

[4] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[5] Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE'09, Proceedings of the 31st International Conference on Software Engineering*, pages 199–209, Vancouver, BC, Canada, May 20–22, 2009.

[6] Mozilla. Firebug. `http://getfirebug.com/`.

[7] Caucho Technology. Quercus: Php in java. `http://quercus.caucho.com/`.

[8] Caucho Technology. Quercus: Tomcat - resin 3.0. `http://wiki3.caucho.com/Quercus:_Tomcat`, called on September 26, 2013.

[9] Wietse Venema. Taint support for php. `https://wiki.php.net/rfc/taint`.

Expressions / Statements	Modified Java Classes

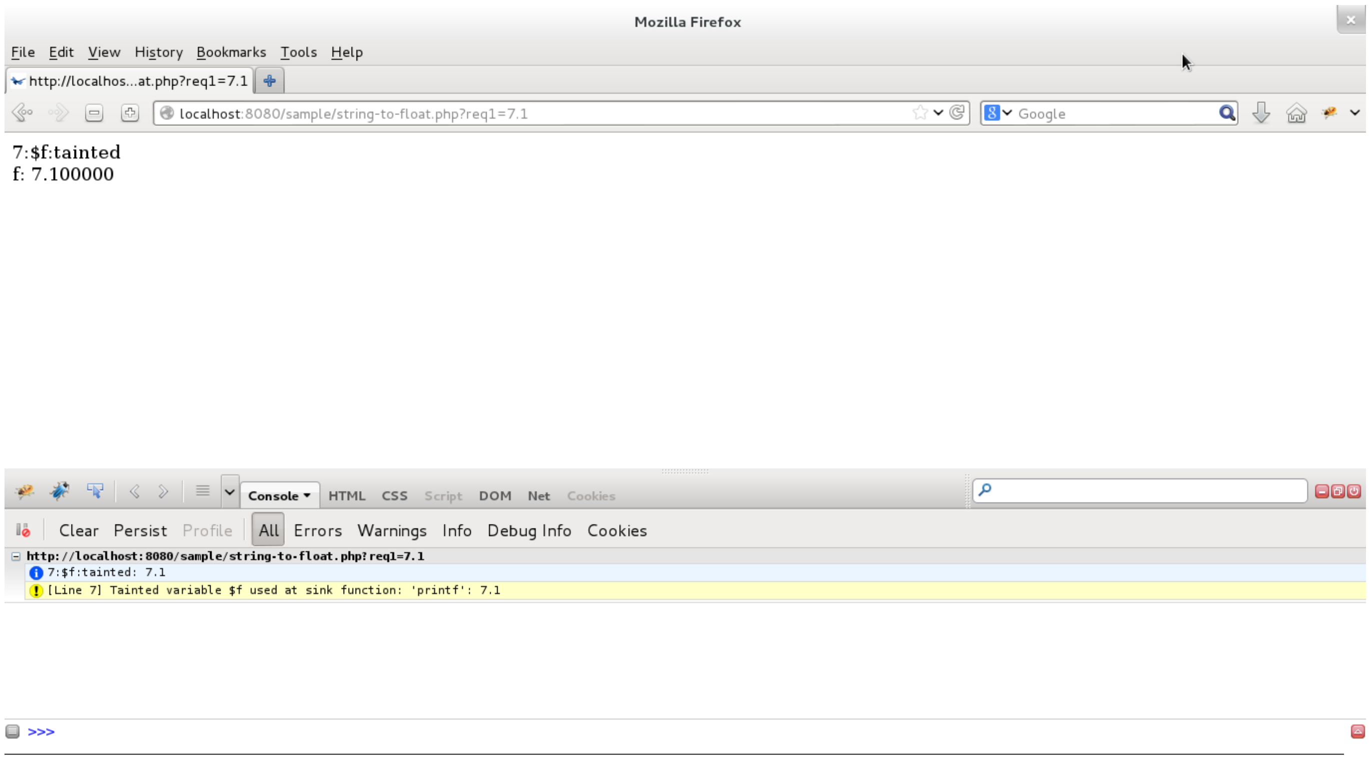
`Addition (+)`
`Subtraction (-)`
`Division (/)`
`Multiplication (*)`
`Increment (++)`
`Decrement (--)`
	`com.caucho.quercus.env.LongValue`
	`com.caucho.quercus.env.DoubleValue`
	`com.caucho.quercus.env.StringValue`

`Bitwise And (&)`
`Bitwise Or (|)`
`Bitwise Xor`
	`com.caucho.quercus.env.StringValue`

`Type promotion and conversion from integer, double, long, and float to PHP strings`	`com.caucho.quercus.env.LongValue`
	`com.caucho.quercus.env.DoubleValue`
	`com.caucho.quercus.env.StringBuilderValue`

`String concatenation`
`Appending a character to a string`
`Retrieving a character from a string`
`Retrieving a substring of a string`
	`com.caucho.quercus.env.StringValue`
	`com.caucho.quercus.env.StringBuilderValue`

Table 1. Expressions and Statements Propagating Taint Information


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhos...at.php?req1=7.1
localhost:8080/sample/string-to-float.php?req1=7.1
Google
Console HTML CSS Script DOM Net Cookies
Clear Persist Profile All Errors Warnings Info Debug Info Cookies
>>>

```
7:$f:tainted
f: 7.100000
```

```
http://localhost:8080/sample/string-to-float.php?req1=7.1
7:$f:tainted: 7.1
[Line 7] Tainted variable $f used at sink function: 'printf': 7.1
```

Figure 4. Firebug Logs