Date of acceptance Grade

Instructor

Java vs PHP: A Security Approach

Anttijuhani Lantto

Helsinki March 4, 2011

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Anttijuhani Lantto

Työn nimi — Arbetets titel — Title

Java vs PHP: A Security Approach

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	March 4, 2011	9 pages + 9 appendices

Tiivistelmä — Referat — Abstract

One of the biggest questions in the beginning of developing web application is the choice of programming language. It affects available tools, libraries, frameworks etc. But does it affect security? This article analyzes four common web application vulnerabilities and studies 25 different web applications security problems. Several indicators are developed to find out if Java is more secure than PHP or vice versa.

ACM Computing Classification System (CCS):

Avainsanat — Nyckelord — Keywords

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

In this paper we examine security differences of two popular programming languages: PHP and Java. Web applications developed with either of these languages have some similar problems and we examine some of those Vulnerabilities. Java has better reputation than PHP but is there actual reason for that?

Although web applications have much more security vulnerabilities than, the four examined in this paper, they still give good overview on the security of applications.

2 Languages

In this paper we examine two different languages PHP and Java. Both of them are really popular programming languages but they have big differences in both how they got started and what kind of languages they are.

2.1 Java

The first version of Java was created around 1993 by James Gosling for device called Star7 in the Green Project while working at Sun Microsystems [Gre04]. Star7 looked like a big PDA and was designed demonstration platform and the software running on it was designed to be deployed on different platforms like televisions and so on. James Gosling created processor independent language that made the demo work. In the beginning language was called Oak after a large oak tree outside Goslings office window. Later the languages name was changed to Java and it was officially announced in May 1995.

2.2 PHP

PHP was originally designed by Rasmus Lerdorf in 1994 [Cho08] and the core scripting engine was rewritten in 1997 by Zeev Suraski and Andi Gutmans. PHP has had many significant changes during its history eg. object model was completely rewritten between PHP 4 and PHP 5.

One of the well know security problems with earlier PHP versions was registered globals. Registered globals populated global scope with all the variables from super globals (ENV, GET, POST, COOKIE and SERVER). For example if $_POST['action']

was set then global variable $action was also set. Although this has clear security problems registered_globals has been off by default since PHP 4.2.0, released in 2002.

```
<?php
  if (authenticated_user()) {
    $authorized = true;
  }

  if ($authorized) {
    include "/highly/sensitive/data.php";
  }
?>
```

In above example user could see "/highly/sensitive/data.php" simply by sending $_GET['authorized']=1 to application [php11]. This could be avoided by initializing $authorized to false but it's still clear how registered_globals is insecure.

Many of the problems of PHP come from the fact that fundamentally PHP has no solid or consistent initial design[Cho08]. One example of this are different search functions. Many string related search functions (strrchr, strstr, strrpos, substr_count) take haystack as first parameter and needle as second argument while array related functions (in_array, array_search) take needle as first argument and haystack as second[php11]. Other problem with core PHP functions is inconsistent use of underscores [Cho08], for example base64_encode, urlencode, strip_tags and stripslashes. PHP has similar naming conventions than eg. Java. Class names start with capital letter etc. The problem with PHP is that variable names are case sensitive but function names are not. With this absurd system variables $i and $I are different variables, but functions urlencode, urlEncode and UrLEnCOde are all same function.

PHP is a popular language and there are many reasons for PHPs popularity [Cho08]. PHP is easy to learn and use, it works in virtually any kind of web server, there is tons of free resources for it and many hosting platforms offer PHP by default. 87.5% of the top web hosting providers supported PHP [WDLM10].

2.3 Comparison

Java is an object-oriented language and it forces all runnable code to be in a class [Gos95]. This makes classes an important, logical and easy way to organize code. Newer versions of PHP support classes and object-oriented design but functions is still the most common way to organize code in PHP applications [WDLM10].

Java is statically typed language and PHP is dynamically typed language (although it is possible to specify argument types for functions in PHP). This is on of the reasons why PHP programs are on average shorter than Java programs [WDLM10].

PHP is often blamed to have inconsistent and weird API but Java too has some strange stuff in its libraries. Eg. Javas standard implementation of stack that has method to get any item in the stack [jav11].

3 Vulnerabilities

Web applications are often developed by programmers without security sophistication [LE07]. Many developers get pressured by their managers for extra functionality and "lesser" concerns like performance and security don't get enough attention. One part of this is developer education and bad tutorials (big part of the problem is that writing insecure code is extremely easy.

```
$sql="INSERT INTO Persons (FirstName, LastName, Age)
VALUES
('$_POST[firstname]','$_POST[lastname]','$_POST[age]')";

if (!mysql_query($sql,$con)) {
  die('Error: ' . mysql_error());
}
```

Of the four vulnerabilities in this chapter this code has two. It has both SQL injection and cross-site script vulnerability. This type of code in common it tutorials since it's easy and natural way to achieve the task of saving something to database. The problem with this is clear: the default is unsafe. To do same task in secure way developer must sanitize input and think all of the ways to pass Javascript or SQL to the application.

Although SQL injection, cross-site scripting, command injection and path manipulation were chosen because they were only security vulnerabilities found by both Java static-analysis tools and PHP static-analysis tools. They are still include common and important vulnerabilities. SQL injection was (with other injections) first on OWASPs Top 10 Application Security Risks -list in 2010 and second on the 2007 list. On the same list cross-site scripting was number one in 2007 and number two in 2010.

3.1 Cross-Site Scripting

Cross-site scripting (XSS) is a class of webapplication vulnerabilities where the attacker exploits hole in applications data validation to output HTML that execute malicious Javascript in victim's machine with the privileges of trusted host [KGJE09] [WS08].

Webapplications have XXS vulnerabilities because the validation (if there is validation at all) they perform is not enough to prevent browser's Javascript interpreter invocation and this validation is even harder if users are allowed to add some HTML mark-up [WS08].

There are clear reasons why XSS vulnerabilities are so common. The requirements are minimal: any web application that displays user input is potentially vulnerable and of those every application that has any faults in it's sanitation or validation is vulnerable [WS08]. Other reason for XSS vulnerabilities is that most programming languages provide an unsafe default to retrieve untrusted input data from user and to display untrusted data back to users. It's really easy to just take raw input data and add it to SQL query or print it back to user in the next request.

There are two types of XSS attacks. The first type of XSS attack is known as first-order, Type 1 or reflected XSS and the second is known as second-order, persistent, stored, or Type 2 XSS [KGJE09]. The third type of XSS attack is DOM-based XSS [WS08].

First-order XSS results form application inserting user's input to next page it renders [KGJE09]. The attacker must use social engineering to convince victim to click on malicious URL. The victims browser then displays HTML and executes malicious Javascript. Attacker can steal eg. browser or other sensitive user data. Users can avoid first-order by checking link anchors before clicking on them but to applications also needs reject or modify values that may contain script code.

A second-order XSS results form application storing attackers malicious input in database or other storage and displaying it later to multiple victims [KGJE09]. Second-order XSS is harder to prevent than first-order because application need to reject or sanitize inputs that may contain script code and are displayed in HTML output like against first-order attacks and also reject or sanitize input values that may contain SQL code.

Second-order XSS is more damaging than first-order XSS for two reasons. First reason is that there is no need for social engineering since malicious code can be injected to page itself, instead of URL, and user can't avoid attack. Second reason is that one XSS in database can get executed in multiple victim users database.

3.2 SQL Injection

In SQL Injection (SQLI) attacker executes malicious SQL statements by exploiting applications data validation [KGJE09]. This is possible by adding (partial) SQL statement in the input. SQLI can give attacker unauthorized access to data or damage the data in database [KGJE09]. Classic example is to send "1' OR '1' = '1" to application which can lead to statement like "SELECT * FROM foo where bar = '1' OR '1' = '1'". This SQL query selects all rows form foo instead of just those where bar is 1.

SQL injection can be blocked by sanitizing inputs that are used in SQL-statements or reject potentially dangerous inputs [KGJE09].

3.3 Path Manipulation

In Path manipulation vulnerability application uses users input to select target for filesystem operation and attacker can do otherwise unpermitted things to system [CW07]. One example is photo hosting service where users can remove their own photos. If attacker can add file system metacharacters (eg. '/' '\' or '.') in target path then attacker can either specify absolute path instead of relative path (../../tomcat/conf/server.xml) or go up in the directory tree to eg. delete servers config file.

Path manipulation is easier to prevent than XSS or SQLI by using a whitelist. One working solution is to deny use of slash and backslash, set maximum length to filepath and allow only on dot on the filename.

3.4 Command Injection

A Command Injection may occur when user input is allowed as part of system command that program executes [CW07]. In PHP this may happen via the 'system()' function. If input can include file system or shell metacharacters then attacker may speficy absolut path instead of relative path or add malicious second command after original command. Command injection is even more dangerous if command is executed as privileged user.

Command injection can be avoided by using APIs instead of system calls or by input validation.

4 Projects and Analysis

Walden et al. examined vulnerability density of 25 open source web applications [WDLM10]. Eleven projects where written in Java and fourteen in PHP. All applications had source code repository ranging from July 2006 to July 2008.

Java			PHP		
alfresco	contelligent	daisywiki	achievo	obm	roundcube
dspace	jackrabbit	jamwiki	dotproject	phpbb	smarty
lenya	ofbiz	velocity	gallery2	phpmyadmin	squirrelmail
vexi	xwiki		mantisbt	phpwebsite	wordpress
			mediawiki	po	

Size of codebase of the projects varied greatly. The smallest PHP project had under 6,000 lines of code and the largest had 388,00 lines of code. Other PHP projects ranged from 25,00 to 150,000 lines of code. The largest Java project was xwiki with over million lines of code and the rest of Java projects ranged from 30,00 to 500,000 lines of code.

Applications source code was analyzed by a static analysis tool. Static analysis tools try to find common security vulnerabilities by evaluating source code without executing it. Static code analysis can by used with just the code and without installation and configuration process. Dynamic analysis on the other hand requires installation and configuration. This means that results of dynamic analysis depend on the configuration of the application and also the configuration of the server.

All static tools produce false positive results. Walden et. al [WDWW09] estimated their false positive rate to be 18.1% with PHP applications. They did not do similar

estimation for Java applications.

Several indicator were computed from all of the projects:

- SLOC source lines of code. This excludes empty lines and comments.
- SAVC Static analysis vulnerability count. Sum of all vulnerabilities.
- SAVD Static analysis vulnerability density. Vulnerability count normalized by KSLOC (thousand source lines of code).
- CVM Common vulnerability metric. Sum of XSS, SQLI, Path Manipulation and Code Injection vulnerabilities.
- CVD Common vulnerability density. CVM normalized by KSLOC.

5 Results

Metric	PHP 2006	PHP 2008	Java 2006	Java 2008	
SLOC	870,00	1,400,000	5,000,000	11,000,000	
SAVC	7730	8459	29,473	42,581	
SAVD	8.86	6.02	5.87	3.85	Both Java and PHP
CVM	5425	3318	5801	7045	
CVD	6.25	2.36	1.15	0.63	
Min CVD	0.03	0.03	0.52	0.04	
Max CVD	121.4	60.41	14.39	5.96	

applications improved as sets from 2006 to 2008. PHP's lower CVD was mostly because of decreased vulnerability count while Java's improvement on CVD was mainly due increased code size. Large part of PHP's decreased CVD was from decline in SQL injections. One part of this change might be use of parameterized query interfaces.

The variation between projects was greater than between languages. The most insecure application of all applications was written in PHP but the most secure one was also in written in PHP. Based on this analysis the selected programming language is not an important part of developing secure web applications.

6 Security Resource Indicator

To measure importance of security to the project Walden et al.[WDLM10] also counted Security Resource Indicator (SRI) to all projects. Security resource indicator is based on four items: security documentation for application installation and configuration, a dedicated e-mail address to report security problems, a list or database of security vulnerabilities and documentation of secure development practises, such as coding standards or techniques to avoid common secure programming errors. SRI is the sum of these indicator items and it ranges from zero to four.

Six of the eleven Java projects scored one form SRI. They all had security documentation. The other five of the Java projects all scored zero.

PHP projects were totally different story. Even though only five PHP projects had security documentation, so the percentage was lower, six PHP projects had sercurty e-mail address, five had vulnerability database, and four had secure coding documentation. Three PHP projects had SRI value of zero but one project (squirrelmail) had SRI value of four.

Although the SRI value had no clear correlation with CVD it did had clear correlation with change of SAVD in PHP projects. SRI was not useful indicator with Java projects since it was practically just a boolean value.

7 Conclusion

Even though used programming language was not significant for applications security this analysis still leaves open questions. This analysis does not say anything about security of new applications or if language has matter with the first version of application.

SRI was significant for PHP applications. This is not surprise since ones first guess would be that projects with more focus and resources for security are more secure.

References

Cho08 Cholakov, N., On some drawbacks of the PHP platform. *CompSysTech*, Rachev, B. Smrikarov, A., . ACM, 2008, 12, URL `http://doi.acm.org/10.1145/1500879.1500894`.

CW07 Chess, B. West, J., *Secure programming with static analysis*. Addison-Wesley Professional, , 2007.

Gos95 Gosling, J., *Java: an Overview*. Sun Microsystems Computer Company, 1995.

Gre04 Greanier, T., *Java Foundations*. Wiley, 2004.

jav11 Java platform, standard edition 6 api specification, 2011. URL `http://download.oracle.com/javase/6/docs/api/`.

KGJE09 Kiezun, A., Guo, P. J., Jayaraman, K. Ernst, M. D., Automatic creation of SQL injection and cross-site scripting attacks. *ICSE*. IEEE, 2009, 199–209, URL `http://dx.doi.org/10.1109/ICSE.2009.5070521`.

LE07 Livshits, V. B. Erlingsson, Ú., Using web application construction frameworks to protect against code injection attacks. *PLAS*, Hicks, M. W., . ACM, 2007, 95–104, URL `http://doi.acm.org/10.1145/1255329.1255346`.

php11 Php manual, 2011. URL `php.net/manual/en`.

WDLM10 Walden, J., Doyle, M., Lenhof, R. Murray, J., Idea: Java vs. PHP: Security implications of language choice for web applications. *ESSoS*, Massacci, F., Wallach, D. S. Zannone, N., , 5965 *Lecture Notes in Computer Science*. Springer, 2010, 61–69, URL `http://dx.doi.org/10.1007/978-3-642-11747-3`.

WDWW09 Walden, J., Doyle, M., Welch, G. A. Whelan, M., Security of open source web applications. *ESEM*, 2009, 545–553, URL `http://doi.acm.org/10.1145/1671248.1671292`.

WS08 Wassermann, G. Su, Z., Static detection of cross-site scripting vulnerabilities. *ICSE*, Schäfer, W., Dwyer, M. B. Gruhn, V., . ACM, 2008, 171–180, URL `http://doi.acm.org/10.1145/1368088.1368112`.