Securing PHP – Approaches to Web Application security

Stanislav Malyshev
stas@zend.com

1 Introduction

Security remains one of the biggest concerns for any web application, and one that is hardest to address. This paper gives a short summary of security approaches that were tried in the past and currently researched on the example of PHP language, which is used by up to 67% of the web developers [1].

The security of the web application bases on the security of the underlying layers, such as OS and application platform layers and the application itself. While the OS layer is beyond control of the PHP project, experience shows that the language is to assist developers in developing more secure code and running it in more secure manner. The majority of the problems in PHP applications is caused by the insecure application code [2], which may allow injecting untrusted data into the output (XSS[13]), database queries and other sensitive commands, running external code in the trusted context (remote include) or disclosing data that the user is not authorized to access. The challenge for the PHP language as a platform is both to provide tools for the developers to avoid such problems and for the site administrators to detect and prevent insecure code from doing harm. The following techniques were employed or researched in PHP, with varying success.

AUTOMATIC QUOTING

PHP allows the user to set configuration variables [7] so that PHP would automatically make any input data safe for using in the database query contexts by quoting special symbols – this approach is called "magic quotes". However, since this approach is applied to all variables, regardless of their usage in the application and since this option could be turned on or off by the site administrator, relying on this option proved to be very problematic in the applications, leading to complex code and configuration errors, and ultimately to security problems. Since different contexts required different quoting styles, the one-size-fits-all approach failed to provide adequate answer to the developers' security needs. Thus it was decided by the PHP group to discontinue the "magic quoting" in future PHP versions [10].

SAFE MODE

PHP has the concept of the "safe mode" [8], allowing the administrator to run PHP applications in the restricted mode which disallowed user from accessing files and scripts that did not belong to the user and prohibiting the application from doing certain kinds of dangerous operations, such as calling the OS shell. This feature gained some popularity among shared hosting ISPs, however due to the variety and multitude of functions supported by PHP, most of which rely on third-party C libraries not controlled by the PHP project, it proved unfeasible to restrict all untrusted operations in the correct manner and create truly secured environment on the PHP level. It was decided by the PHP group to discontinue safe mode in future PHP versions [10]. Currently it is recommended for ISPs needing secure shared hosting to use OS-level security [14].

INPUT FILTERING

PHP version 5 offers a set of input filters[9] that can be applied to input variables in order to provide standard ways to verify the data conforms to certain type or does not contain dangerous symbols. Using this approach alone, however, can not ensure that the data is properly filtered, as the task of the running the right filter for the right context is still on the developer, however having standard filters ensures the data is properly sanitized once correct filter is applied.

DATA TAINTING

Recently, a number of solutions were proposed for introducing data tainting concept to PHP, as available in languages like Perl [11] or Ruby [12]. This would allow the developer to ensure that no untrusted data is passed to the sensitive functions without prior filtering.
Static tainting, as proposed by [3], can protect applications before actually running them, so the problem could be eliminated before deploying the code into a sensitive environment. However, due to the complexity of PHP code, allowing widely used dynamic code generation and multiple levels of indirection in variable and function access, static analysis is unable to achieve comprehensive coverage of the application functionality. Even though, the results in [3] show that many existing problems could be eliminated by this approach.
Tainting at the runtime, as proposed by [5] and [6], while ensuring comprehensive coverage, could lead to significant performance reduction, since each variable access needs special care with regard to tainting, and requires significant modification of the PHP engine code in order to classify the functions by their relation to tainted data. Also, it might be hard to apply this approach to existing applications employing custom filtering techniques, since it is not possible to automatically decide if the filtering is done correctly.

CSSE

Different approach proposed by [4], is aimed at preventing injection attacks by tracking the source of the data on per-string basis and verifying the safety of the data item for each specific functions. This approach, as above, would require extensive modification of PHP functions and relies on the module to actually understand the semantics of the underlying data formats, such as SQL – which might prove to implement comprehensively for all data formats and contexts, facing challenges not unlike those that led to the failure of the safe mode.
Surprisingly, the results shown in [4] indicate the performance impact is not very significant.

RUNTIME VARIABLE TYPE DETECTION

Proposed approach employs runtime analysis of the application input data as to collect information about kinds of input parameters used by various scripts composing the application. As an example, if we take PHP bug database, the typical call would look like:
http://bugs.php.net/bug.php?id=41270
From a few of such calls it could be automatically derived that bug.php has one parameter named "id" and it is an integer. Thus, if this request is translated by the application to a query like:
SELECT * FROM bugs WHERE id=$_GET[id]
and an attacker attempts to exploit this code by passing string like "0 OR 1=1" it can be detected that the value passed to "id" is not of the correct type and thus the request could be refused. In the same spirit, it can be detected that certain portions of input data find their way into the output and thus data containing unsafe code (such as XSS exploits) could be rejected – without actually having to inspect application code.
This approach can not be recommended as a comprehensive security solution but combined with other approaches can reduce the risk of the application being exploited by restricting permitted inputs to the set that was expected and tested by the application developers.

2 Conclusion

Application security remains one of the biggest concerns among the PHP developers, and significant part of the application problems boil down to inadequate data filtering [2]. Employing the techniques for static and dynamic detection of unfiltered data may allow sites significantly improve their security while running existing applications, by detecting and preventing insecure code execution. It is yet to be found which of these approaches could be integrated into the PHP engine to provide comprehensive solution without significantly impacting performance.

3 References

[1] 2006 State of Web Development, SitePoint Pty Ltd. and Ektron, Inc., August 2006
[2] National Vulneravility Database, http://nvd.nist.gov/
[3] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T. Lee, Sy-Yen Kuo. "Securing Web Application Code by Static Analysis and Runtime Protection". Proceedings of the 13th international conference on World Wide Web (May 2004). http://www2004.org/proceedings/docs/1p40.pdf
[4] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. http://chris.vandenberghe.org/publications/csse_raid2005.pdf
[5] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. "Automatically hardening web applications using precise tainting." 20th IFIP International Information Security Conference, 2005. http://www.cs.virginia.edu/evans/pubs/infosec05.pdf
[6] Wietse Venema. PHP tainting proposal, http://news.php.net/php.internals/26979
[7] PHP Magic Quotes, http://www.php.net/manual/en/security.magicquotes.php
[8] PHP Safe mode, http://www.php.net/features.safe-mode
[9] PHP input filtering, http://www.php.net/filter
[10] PHP 6.0 Plans, http://oss.backendmedia.com/PhP60
[11] Perl variable tainting, http://perldoc.perl.org/perlsec.html
[12] Ruby variable tainting, http://www.rubycentral.com/book/taint.html
[13] http://en.wikipedia.org/wiki/XSS
[14] Running PHP on shared hosting, http://hostingfu.com/article/running-php-on-shared-hosting