PHPGate: A Practical White-Delimiter-Tracking Protection against SQL-Injection for PHP

Lihua Zhang[1] Yu Ding[1] Chao Zhang[2] Lei Duan[1] Zhaofeng Chen[1] Tao Wei[1] Xinhui Han[1]
[1]*Peking University* [2]*UC Berkeley*

SQL injection has a long history as a dangerous threat, grabbing lots of security researchers' attentions. However, it is still ranked the first of the ten most dangerous web application threats by OWASP since 2008, indicating a big gap between researches and industry practice.

Most SQL injection vulnerabilities are introduced by ignorance or negligence, so a good protection solution should be "fool-proof". To be used in day-to-day deployment, it should also have little performance overhead.

In this paper, we propose a **white-delimiter-tracking** defense solution against SQL injections: unlike typical taint-tracking solutions that track user inputs or all SQL related strings hard-coded in web applications, we only track the SQL delimiters in these hard-coded strings, and only allow these delimiters to be used in the final SQL statements. Furthermore, instead of allocating extra shadow memory for taint tracking, we encode the tracking information directly into the same byte as the delimiters by utilizing the characteristics of UTF-8 encoding.

This solution tracks much less data than existing solutions and keeps the accuracy of byte-level taint tracking. A prototype called `PHPGate` is implemented on the PHP Zend engine. `PHPGate` can protect UTF-8 encoding web applications from real world SQL injection attacks and only introduces a performance overhead less than **1.5%**.

Architecture Figure 1 shows the framework and the workflow of `PHPGate`. The Zend engine's compiler will invoke `PHPGate`'s *taint introduction* module to mark hard-coded delimiters as white and encode them (step 5); its executor will cooperate with `PHPGate`'s *taint propagation* module to propagate the taint information (step 7); and the executor will invoke the `PHPGate`'s *security validating* module to validate SQL query strings (step 8) before sending them to the backend database (step 9).

White-Delimiter-Tracking `PHPGate` uses taint analysis to differentiate user inputs from legitimate strings in applications. Unlike traditional dynamic taint analysis which marks user input as tainted, `PHPGate` marks characters hard-coded in applications which are parts of a SQL query string as tainted. Moreover, only delimiter characters (e.g., quotation marks) that affect the structure of SQL query strings are marked as tainted and are tracked. These delimiter characters hard-coded in applications are called **white-delimiters** in this paper.

Taint Tracking with UTF-8 Encoding Some codes (e.g., those of the form `10xx xxxx`) are not used by UTF-8 encoding. So, neither user inputs nor hard-coded strings will use these codes. `PHPGate` then encodes each white-delimiter to a unique code reserved by UTF-8, for further taint tracking and security validating.

In this way, `PHPGate` only needs to track and encode a few fixed characters and keeps original strings' lengths unchanged. It thus introduces a much less performance overhead. The taint tags (i.e., encoded delimiters) will remain intact along with the applications' execution until reaching the boundary point where the SQL query will be sent to backend database or to other modules. We also modify the regular expression modules to adapt to this white-delimiter encoding scheme.

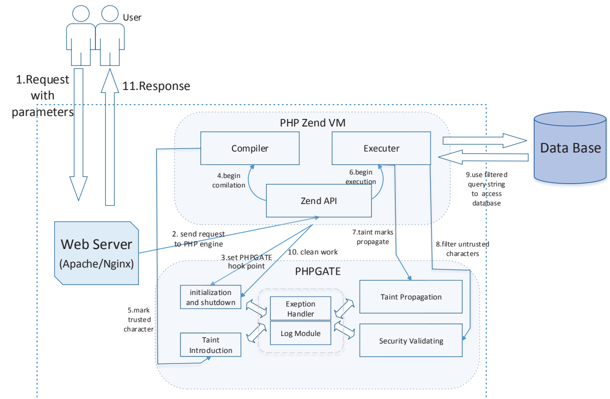

Figure 1: Architecture of `PHPGate`

Security Validating At the boundary point, a security validator is deployed. It will check untrusted user input in SQL queries to find whether there are non-encoded delimiters used in SQL query strings. And if so, the validator will then replace them with SQL escape sequences. The encoded white-delimiter characters in SQL query strings are also changed back to corresponding original characters. As a result, only legitimate SQL queries will be sent to the backend database, leaving the structure of the SQL query string unchanged.

Preliminary Results We have built a prototype of `PHPGate` as a PHP plugin in a LAMP-based environment, and tested several popular PHP applications in our framework. The performance overhead of `PHPGate` is only 1.28%. Other PHP solutions introduce much higher overhead, e.g., PHP Aspis's overhead is more than 220%[2] and Ding Xiang's solution introduces 28%-38% overhead[1]. Also, we tested SQL injection attacks found in exploit-db, including exploits which targets OSVDB-103365/103126, CVE-2013-6936/2007-4653/2003-1244 etc. Results show that `PHPGate` can successfully block all these attacks.

References

[1] X. Ding, Y. Qiu, and T. Zheng. A method to defend sql injection based on php. *Computer Engineering*, 37(11):152–154,157, June 2011.

[2] I. Papagiannis, M. Migliavacca, and P. Pietzuch. Php aspis: Using partial taint tracking to protect against injection attacks. In *Proc of the 2nd USENIX Conference on Web Application Development*, Jun 2011.