Project Report

LDAP Injection

Software Security

MEIC-A

4/12/2015

Group 4

70171 João Miguel Soares de Matos Neves
72904 Luís Filipe Pookatham Ribeiro Gomes
75735 Beatriz Ventura Brêda Pinho dos Santos

Study of LDAP Injection vulnerabilities

LDAP injection vulnerabilities arise from the failure to correctly validate user input.

In web applications this input can come from several entry points that can be controlled by an attacker (see Table 1 - entry points).

Queries to the LDAP server are constructed with the help of filters that provide parameters for the LDAP server to match with entries in its directory tree.

Through the introduction of meta-characters an attacker can change the outcome of certain queries performed by the web application to achieve goals such as elevation of privilege, information disclosure and modification of data on the server.

Table 1 shows the various entry points for user input that can be used in php web applications, the functions that are vulnerable to LDAP injection and the function that should be used to sanitize input before using it in a query.

Entry points	**Sensitive sinks**	**Sanitization functions used for untainting**
$_GET $_POST $_COOKIE $_REQUEST HTTP_GET_VARS HTTP_POST_VARS HTTP_COOKIE_VARS HTTP_REQUEST_VARS	ldap_search ldap_add ldap_compare ldap_delete ldap_list ldap_mod_add ldap_mod_del ldap_mod_replace ldap_modify_batch ldap_modify ldap_read ldap_rename ldap_bind ldap_sasl_bind	ldap_escape

Table 1 - LDAP injection entry points, sensitive sinks and sanitization functions

Modification of WAP

To get the WAP tool to detect LDAP injection vulnerabilities in PHP code, the tool was modified by replicating the code that performed the detection of SQL injection vulnerabilities and changing the names of the functions used in SQL to the ones described in Table 1.

The original WAP tool is also capable of correcting vulnerable SQL code, however due to differences between SQL and LDAP, and the way corrections were being done, the modified WAP tool is only capable of correcting LDAP code with vulnerabilities in the definition of filters (AND and OR injection) but not in the usage of only distinguished names(DN).

Experimental Results

We wrote a simple php program that received input from some of the entry points in Table 1, and after connecting to an LDAP server used that input as arguments in calls to various ldap functions. We also sanitized the input with “ldap_escape” before some of those calls.

The tool correctly identified the simpler vulnerable code and ignored the calls with sanitized arguments. It also correctly gave code corrections for calls that used AND and OR filters.

However it was incapable of detecting vulnerabilities that involved the usage of php aliases and also vulnerabilities introduced by using tainted variables defined in “included” php files.

State of the art on PHP static analysis

The usage of security-typed languages that detect information flow vulnerabilities as suggested in [4] use label annotations and special compilers/interpreters. And even though the languages might be similar (based on) existing common languages, they still require extensive effort and awareness from the programmers when writing their applications and cannot simply be used on existing applications. Nevertheless security-typed languages allow for both static analysis and runtime validations and even dynamic security requirements while producing few false positives.

The usage of multitier architectures are frequently in the source of code injection attacks. When we write a web application (p.e. PHP) that dynamically generates programs in a target language (p.e. HTML, JavaScript) based on user input, it represents a very serious integrity violation if that input can be confused with the original target program code. To prevent this, a new technique is proposed in [1] which is using a unified syntax for server and client code (multitier language, in this case, HOP language). In opposite of [4], no programmer intervention nor plugins are needed to ensure security against code injection. This solution basically consists in obtaining the HOP AST from server-side run-time environment and generating the HTML document (with a HTML parser) to be sent to the client. Client-side, a DOM tree is generated based on the the HTML document. It the AST and DOM trees do not match, there has been a code injection.

Data flow analysis and alias analysis, as described in [5] and applied in [2], prevent taint-style vulnerabilities. Static analysis tool Pixy parses PHP input using JFlex and JCup and linearizes the generated parse tree into a forms, which, with file inclusions, will be preprocessed for literal analysis. Alias relationships between analysed variables are then made to perform static detection of vulnerabilities. However, Pixy doesn’t support neither PHP object oriented features, such as objects and arrays, nor complex file inclusions, making this tool unsound.

SQL Command Injection Vulnerabilities (SQLCIVs) are very common in web applications that deal with databases. The static analysis proposed in [3] models string values (acquired from String-Taint Analysis of SQL queries in PHP Files) as context free grammars and string operations as language transducers. Every SQL string is labeled either with “direct” (like GET) or “indirect”.

Application of research to LDAP injection problem

The technique proposed in [1] is meant to use in multitier compilers, so it is not applicable to WAP. Also, it is not possible to obtain a DOM tree from a LDAP query. However, tree comparison could detect some code injections. If we used a PHP parser to generate an AST for every LDAP query, before and after user input, if the trees mismatched, we could have a AND or OR injection. But this would detect all possible LDAP injections because some string analysis is needed to discover irregularities in LDAP queries.

The advantages of a security-typed language could also be applied to the php language and interpreter to prevent LDAP injection. However that approach would require the applications to be written purposefully in that language variant. It could however solve the problem of dynamic includes, and even alias analysis.

References

[1] Zhengqin Luo et. al. Automated Code Injection Prevention for Web Applications, TOSCA 2011.
[2] Nenad Jovanovic et. al. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities, PLAS 2006.
[3] Gary Wassermann and Zhendong Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities, PLDI 2007.
[4] Stephen Chong et. al. SIF: Enforcing Confidentiality and Integrity in Web Applications, Usenix 2007.
[5] Yao-Wen Huang et. al. Securing Web Application Code by Static Analysis and Runtime Protection, WWW 2004.