MySQL Attacks Self-Detection

OSSEC, Security, Software,

I'm currently attending the Hashdays security conference in Lucerne (Switzerland). Yesterday I attended a first round of talks (the management session). Amongst all the interesting presentations, Alexander Kornbrust got my attention with his topic: "*Self-Defending Databases*". Alexander explained how databases can be configured to detect suspicious queries and prevent attacks. Great ideas but there was only one negative point for me: Only Oracle databases were covered. But it sounds logical; In 2008, Oracle was the first (70%) in terms of database deployments, followed by Microsoft SQLServer (68%) and MySQL (50%). I did not find more recent numbers but the top-3 should remain the same. Alexander gave me the idea to investigate how to do the same with MySQL.

IMHO, MySQL is very poor in logging features. There are three types of log files:

- The server log
- The general query log
- The slow query log

The server log contains all events related to the MySQL daemon: when the server started, when it was stopped and all critical errors that may occur during operations. The general query log is where are stored all events related to clients connections and (optionally) each SQL statements received from clienst. This log is very interesting to track the behaviour of clients but it has also two constraints: it can have a huge performance impact on heavy loaded servers (CPU, IO & storage) and the errors returned by bad queries are **NOT** logged. Finally, the slow query log is more relevant for developers and DBA's who can track queries which affect the server performace. More information about the MySQL logging can be found here. If you're implementing a log management solution, having a copy of all queries is of course interesting but errors are even more important. MySQL has no way to log errors returned by malformed queries. Why is this information relevant? In most cases, queries are performed automatically by applications and should not return errors (I mean, at syntax level). If malformed queries are received, it could be an attacker trying to guess databases, tables or rows names. It can be an attempt of MySQL injection. Errors can be detected at application level, by a WAF but, like we say in French: "*On n'est jamais mieux servi que par soi-même*". This can be translated to: "*If you want it done right, do it yourself*". The goal is to catch the error at server level. How?

MySQL has mechanisms to detect errors using "*handlers*". You can declare a handler like this:

DECLARE EXIT HANDLER FOR SQLSTATE <value> <statement>

Again, we are facing a lack of flexibility: handlers can be used only in stored procedures and you must know which errors to track (the SQLSTATE values). This is unmanageable to track all errors.

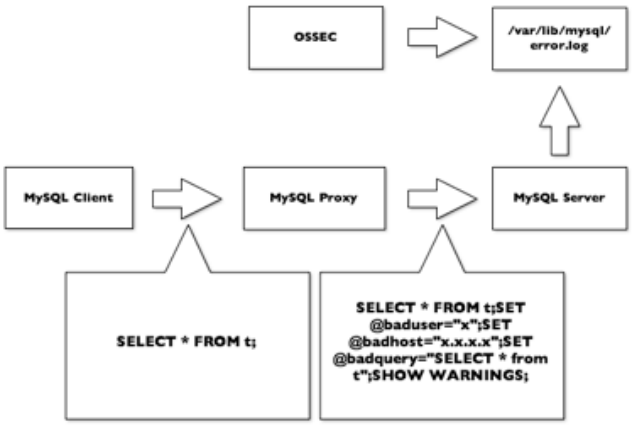
MySQL Attacks Self-Detection

The alternative is to use "mysql-proxy". This tool is not well-known but does very interesting stuff in our case. MySQL proxy is deployed in front of your MySQL server(s) to monitor, analyse and load-balance the requests. The killer feature is the LUA interface (nmap alike). With the help of LUA scripts, you can inspect, filter or modify the requests before forwarding them to the server. In our case, we won't load-balance the traffic, let's install and run the proxy on the MySQL server itself (Debien & Ubuntu packages exist).

The other component required is an UDF ("*User Defined Function*") to write the errors in a log file. I re-used the same as I did in a previous blog post: lib_mysqludf_log. This UDF allows you to output data in the regular server log:

mysql> drop function lib_mysqludf_log_info; mysql> drop function log_error; mysql> create function lib_mysqludf_log_info returns string so name 'lib_mysqludf_log.so'; mysql> create function log_error returns string so name 'lib_mysqludf_log.so'; mysql> select log_error(«This is a major failure!»);

The next step will be to inspect the packets and modify them on the fly to add some logging capabilities. In MySQL, this is achieved by appending the "SHOW WARNINGS" statement and some variables declared with "SET". The data flow is described in the schema below:



Let's use a LUA script to perform this. It will contain three functions:

- read_packet() will intercept MySQL requests and rewrite the SQL statement
- read_query_results() will intercept the SQL results (the answer sent back to the client) and will extract useful information like the error type, the code and the error
- write_log() will dump the information to the server log using the log_error() UDF

The script is available here.

How does it works? First, let's start the MySQL proxy. By default, it binds on port 4040:

mysql-proxy --log-level=debug --daemon \ --proxy-lua-script=/home/xavier/log_errors.lua

Then, let's connect to your MySQL server and create a database "*customers*" with a table "*details*":

mysql -u xavier -P 4040 -p Enter password: mysql> create database customers; mysql> use customers; mysql> create table details (id integer, -> firstname varchar(64), -> lastname varchar(64));

MySQL Attacks Self-Detection

Now, let's write a buggy SQL query (with a double "i" the row name):

mysql> select * from customers where iid=1; ERROR 1105 (07000): Unknown column 'iid' in 'where clause'

In your server log, you should see something like:

SQL-Error: 2012-11-01 15:10:11 xavier@127.0.0.1:56899 -> Error 1055: Unknown column 'iid' in 'where clause' -- select * from customers where iid=1

Now, you have all the required information in a flat file to generate alerts using your preferred log management tool. Are you using OSSEC? Just use the following decoder:

<decoder name=»mysql_sql_error»> <prematch>^SQL-Error:</prematch> </decoder> <decoder name=»mysql_sql_error_details»> <parent>mysql_sql_error</parent> <regex>(\S+)@(\d+.\d+.\d+.\d+):\d+ -> Error (\d+): (\.+)</regex> <order>user, srcip, id, extra_data</order> </decoder>

This decoder will extract and fulfill the following OSSEC variables:

- user: the MySQL user who send the SQL query
- scrip: the client IP address
- id: the MySQL error number
- extra_data will contain the error message concatenated with the faulty SQL query

Once working properly, you can change the TCP ports used by the mysqld and mysql-proxy. Change the MySQL default port to an alternative port like 33060 and bind the proxy to 3306. This will be transparent for all the clients.

A last important remark, mysql-proxy is still an Alpha version for a while! I don't know if a stable version will be release and when. Using a proxy may also have an impact on the performance. This setup is a kind of proof-of-concept to show how to extract valuable data directly from the database.

database, MySQL, OSSEC, Security, Software, SQL

Hack.lu 2012 Wrap-Up Day #3 Hashdays Wrap-up Day #1