Protecting your database

Database Intrusion Detection

Your database can be one of the most vulnerable elements in your organization. We share some tips for detecting and preventing attacks. *By Kurt Seifried*

When it comes to attacks against organizations, databases are generally the soft underbelly – easy to attack once you're past the front line – and they usually hold all the really interesting data. But, databases are hidden behind firewalls, and only a complete fool would allow direct remote access to them, right? Right. But for a database to be useful, you generally have to make it available to users so they can interact with it.

This setup is more often than not accomplished through web-based applications. So, as long as the web-based applications are secure, your database should be safe. Of course, this is rarely the case: The CVE database contains the phrase "allows remote attackers to execute arbitrary SQL commands" in 3,773 entries, and the phrase "SQL injection" occurs 5,717 times. And, the CVE database is by no means a complete list of all the SQL-related vulnerabilities in common software applications.

Why is SQL injection such a problem? The sad reality is that most applications, especially web-based applications, use a single account for everything, including applications such as WordPress, MediaWiki, and Bugzilla. Not only do they all use a single user by default for all database operations, I'm not actually sure that you can split up the database usage into different users without rewriting significant portions of the applications. Additionally, most of these applications don't offer much guidance regarding exactly which permissions are actually required.

MySQL Permissions

Accounts in MySQL can have a variety of privileges assigned to them. These permissions can be applied at the global level (e.g., against all databases or special properties such as shutting down the server), at the database level (which contains one or more tables), at the table level (which contains one or more columns), and within a table against specific columns. Unfortunately, MySQL does not currently support row-level (e.g., per record) permissions; however, this can be added at the application level. Of course, if you bypass the application-level security through an SQL injection, your row-level protection won't help. Because most applications don't provide much guidance on exactly which permissions are needed, how do you go about locking them down?

Calculating SQL Permissions

The good news is that Dan Cornell of the Denim Group has done some work in this area. His excellent presentation is available as a video [1], and he also provides slides [2]. The first step in calculating SQL permissions is to enable logging of SQL queries (`mysql_start_logging.sh`), which basically boils down to setting two global variables:

```
mysql -e "SET GLOBAL log_output = ⮐
  'TABLE'" --user=root mysql
mysql -e "SET GLOBAL general_log = ⮐
  'ON'" --user=root mysql
```

This setup will log queries to the `mysql.general_log` table. You'll then need to execute every possible operation within your application: adding users; removing users; modifying users; creating, updating, and deleting records through any operations, such as creating items or modifying them; running internal application updates; and so on.

KURT SEIFRIED

Kurt Seifried is an Information Security Consultant specializing in Linux and networks since 1996. He often wonders how it is that technology works on a large scale but often fails on a small scale.

A good way to do this is to install a recent version of the application and then run the upgrade process, logging everything that happens. The problem is that if you fail to run an operation (and thus don't generate a log entry for the queries it needs to run), you might break your application later when you lock it down.

Now comes the tricky part: Turning these logs into a set of permissions is far from trivial. You can do a lot of the heavy lifting using the sqlpermcalc program (and associated tools) [3]. Basically, you feed the exported `mysql.general_log` data into `sqlpermcalc.py`, but it only handles `SELECT`, `INSERT`, `DELETE`, and `UPDATE`.

You will need to handle permissions such as `FILE` manually; this allows importing and exporting of data from files and is generally not needed. Other permissions relating to the structure of databases, such as `CREATE`, `ALTER`, `INDEX`, `DROP`, and so on, will typically only be used during installation or upgrades of the system. Finally, the administration permissions like `GRANT` and `SUPER` are almost never needed. These are typically given only to administrative users.

Building Applications Properly

I thought it might be useful here to include some general information on how applications should be built. Ideally, you should examine the various roles needed by your applications components (e.g., a component that creates a user account, a component that is used to modify the user account, and the component that is used by administrators to manage users all need different access levels).

If you can split them up – for example, only giving the create user component the ability to `INSERT` values, giving the user account modification component the ability to `SELECT` and `UPDATE`, and granting the administrative component privileges, such as `SELECT`, `INSERT`, and `UPDATE` – you will significantly reduce the effect of any SQL injection vulnerability in the user-facing components. Your administrative components are locked down, right?

Faking Row-Level Permissions

Another consideration is adding row-level permissions into MySQL. The first question to ask is what kind of permissions are needed? Will each row have one user specified that can read or modify it, or do you need to support multiple users and groups for each row with varying levels of permissions? If one user owns a record, you can simply add a column to the table in question and list the user that is allowed access to that row.

If you need more complicated permissions, your best bet is probably to create a table of users, a table of groups, or both, along with their associated permissions, and then use a link table to connect rows in the table that you want to protect with the row in the permissions table that defines what is allowed. You will then need to instruct your application to support these permissions. Another option would be to add a database abstraction layer, which mediates access (so you pass the query, the username or group, and possibly an access token/password).

Building Tripwires

One classic way to detect attacks against a database is to build tripwires. One simple method of doing so in MySQL is to add columns that are not needed and restrict access to them. You would then enable the general query log and watch it for failed queries (the overhead here would be an issue). Thus, if an attacker issues a query such as `SELECT * FROM`, it will be detected. You also would have to ensure that your application only requests the columns it needs and does not use `SELECT * FROM` itself.

Another option that can work well in conjunction with an IDS such as Snort is to insert fake records into the database. Using special values that will never be encountered in reality (i.e., "kwiegdbvikwebvkjwb") and then configuring your Snort IDS to watch for those values is an effective way to raise the alarm if the attacker starts sucking down the entire database.

Additionally, if you flood your database with fake records, the attacker will need more time to download the data, which will give you more time to react. This setup can also affect performance (making your database 10 times larger will probably not be overly popular with the admin who has to tune the database). However, if you architect your tables correctly, for example, by splitting passwords into a separate table, you can easily add a lot of fake records without too much of an effect. Just make sure it is indexed.

A third option is to log database activity and build a statistical model of what is "normal" for your application. Assuming you have multiple front-end servers using the database back end, creating separate user accounts for each application server will let you more closely track what each server is doing. A five percent increase in overall queries is probably nothing much to worry about. However, if you have 10 servers splitting the load, a 50 percent increase load on one server is definitely cause for concern.

sqlmap

Finally, here is the pointy end of the stick. Traditional penetration testing involved a lot of hand-crafted tools and manual work. That approach is, of course, quite inefficient and tedious at times, so people have created software suites and frameworks for testing the security of applications and systems. For web-based SQL injection, the tool to use is sqlmap [4].

sqlmap supports virtually every database (MySQL, PostgreSQL, Oracle, MS-SQL, etc.). It also supports a wide variety of attacks, including time-based SQL injection (longer queries mean you're probably doing something you are not supposed to) and error-based SQL injection. Additionally, it can inject attacks into cookies, requests, HTTP-referer, and User-agent, just to name a few features. These tools, along with careful permissions setup, will help you detect database attacks and protect your organization. ■■■

INFO

[1] Dan Cornell video on SQL permissions: *https://www.youtube.com/watch?v=jzRShMGZe3U&hd=1*

[2] Dan Cornell blog entry on SQL permissions: *http://blog.denimgroup.com/denim_group/2012/04/source-boston-2012-follow-up-what-permissions-does-your-database-user-really-need.html*

[3] sqlpermcalc: *https://github.com/denimgroup/sqlpermcalc/*

[4] sqlmap: *http://sqlmap.org/*