First Things First

This lesson outlines several basic yet very core tasks to perform after completing the installation:

Apply Latest Trusted Patches

Patch the operating system and any installed software: Software security alerts are released more frequently, and it's critical that you take the steps to ensure that your system is fully patched. With explicit instructions and tools readily available on the Internet, even a novice malicious user can take advantage of an unpatched server. Several automated scanning tools increase the change your unpatched server will be exposed and compromised.

Even on managed servers, it is best to be proactive with the service provider to perform the necessary upgrades; Monitor and coordinate support updates to ensure the servers are in a secure state.

Cleanup restaged boxes

Often existing servers are restaged for hosting databases. To completely ensure that nothing is overlooked during such an audit, consider reformatting all attached drives and reinstalling the operating system.

Audit OS User Accounts

Ensure that all nonprivileged users are disabled or removed. Although operating system users have little bearing on MySQL users directly, the OS users have varying access to the server resources, increasing the chance of damage to the database server and its contents, intentional or otherwise.

Disable Unused System Services

Always take care to eliminate all unnecessary potential attack routes before you place the server on the network. Several attack vectors often piggyback on insecure system services such as HTTP servers, often running undetected. If you're not going to use a service, disable it.

Network

The network is another point of entry to your database, and it should be a part of your security audit on your database servers. As suggested earlier, if you don't need network access on your MySQL server, turn it off. There's no reason to leave it up and listening if it's not being used. It poses an unnecessary security threat.

Here are a few other ideas for securing your network:

Add Firewall to Shutdown Access

Adding a firewall to your machine not only blocks unwanted MySQL traffic, but it is also good practice for keeping your operating system secure.

Besides turning off unused system services, you can add a second layer of security by closing all unused access ports. For a dedicated database server, consider closing all ports below 1024 except the designated SSH port, 3306 (MySQL), and a handful of "utility" ports, such as 123 (NTP). In addition to making such adjustments on a dedicated firewall appliance or router, also consider activating firewalls that are part of the operating systems.

Restrict host connections

Use the HOST field restrictively in your GRANT statements, making sure that you are granting access only to connection requests from a set of specific DNS or IP addresses.

Use IP addresses if your DNS is unreliable

If you don't have a trusted DNS source, use IP addresses for connection control. An untrusted DNS means that you can't be sure that someone won't change the entry for trustedhost.promysql.com to untrusted.badhacker.net. In this case, your database, thinking connections are allowed from trustedhost.promysql.com, would let the connection through, even though the request was really coming from an untrusted machine.

Disable Networking

By default, MySQL will open up network port 3306 to allow remote requests. If you do not want this port open, append "--skip-networking" when running safe_mysqld to start the daemon:

/usr/bin/safe_mysqld --skip-networking > /www.null 2>&1 &

Now whenever running /etc/init.d/mysql start, it will not open up port 3306

Securing the mysqld Daemon

Here is a summary of several security options that you can use when you start the mysqld daemon:

- --chroot: Places the server in a restricted environment, altering the operating system's root directory as seen by the MySQL server. This greatly restricts unintended consequences should the server be compromised by way of the MySQL database.
- --skip-networking: Prevents the use of TCP/IP sockets when connecting to MySQL, meaning that remote connections aren't accepted regardless of the credentials provided. Disabling networking is a more drastic measure but makes sense if all your application clients and the MySQL installation are colocated on the same server
- --skip-name-resolve: Disables MySQL from resolving hostnames from DNS. A rogue DNS may resolve good names to bad hosts. This means that all Hos t column values in the grant tables consist either of an IP address or localhost.

- --skip-show-database: Prevents any user without the explicit SHOW DATABASES privilege from using the command entirely.

- --local-infile: Disabling this option by setting it to 0 disables use of the command LOAD DATA LOCAL INFILE , or prevent loading a file from a different machine.

- --safe-user-create: Prevents any user from creating new users via the GRANT command if they do not also possess the explicit INSERT privilege for the *mysql.user* table.

Files, Directories, and Processes

The files, directories, and processes on your server also need to be protected to secure your database. A user with read access to the database directory and files can easily copy them to another server and have their local MySQL load them. With access to the MySQL binary logs, a user can view any insert, update, or delete statements in the database. Having this information exposed via files is almost as unsecure as having the data files available to such users.

Here are a few suggestions to secure access to files, directories, and processes:

Run MySQL as a non-root user

Do not use the root Unix account to run the database. Typically, you will create a *mysql* user account and use it for running the database. Then, if the MySQL server has a security vulnerability, the damage to the rest of the operating system is limited to what the mysql user is allowed to do. You can run mysql as any existing user, provided that the user is the owner of the data directories. For example, suppose you want to run the daemon using the user mysql
:

```
root> ./bin/mysqld_safe --user=mysql&
```

You may have MySQL users with FILE privilege. If MySQL server is running with *root* OS user, database users with FILE privilege may be using files with OS-level
root privilege
, allowing them access to load normally protected OS files from the operating system into a database.

Protect the socket file

Make sure your mysql.sock file, typically stored in the /tmp directory (/tmp/mysql.sock) and used for database interactions on the local machine, is protected. On some operating systems, any user can remove files from the /tmp directory. Removing mysql.sock will prevent any interaction with the database.

To protect against this, you can either set permissions on /tmp files or move mysql.sock to another location using the socket=/path/to/socket option in your database and client

configuration. Refer to http://dev.mysql.com/doc/mysql/en/problems-with-mysql-sock.html for more information.

Set Data File Permissions

Make sure your data files are available only to the mysql user or to mysql group as needed. Even users with no MySQL accounts may have the ability to read the data and log files on the server's filesystem, and they can make copies of these files.

Secure Log Files As Well

If you are logging database activity, include your log files in the list of files to secure. Give log files read and write permissions to only the mysql user (and perhaps the mysql group).

Use the mysql group

Create a mysql group, and use that group to control access to the log files. Depending on your level of logging, having exposed log files in MySQL can be just as bad as having exposed data files.

Consider an Encrypted File System

If protecting your data files is super-critical, evaluate the option of an encrypted file system or storing data on the disks in an encrypted format. An encrypted file system ensures your data is still protected even if your disk is maliciously removed from your server. Please remember to guard your encryption keys. Encrypted file system adds overhead in reading and writing data. Encrypted files also affects backing up and restoring of data.

Avoid Symbolic Links

If you don't need to use symbolic links, disable this feature by specifying the skip-symbolic-links
option. This will prevent users from creating symbolic links within MySQL to files they don't have permission to view but can get to through the server (running MySQL as root makes this problem even more serious). However, in some cases, using a symbolic link for tables is necessary to spread the data across several different disks for space and performance.

Securing Accounts

Just as when you're protecting access to data and database management functions from within MySQL, it is important to look at all the user accounts on your server and determine how to make sure the data and log files are available to only those who have authorization to access the files. Some pieces of this will be implemented with the security plan; others are just a part of being diligent about locking down your system.

Set a good MySQL root user password

By default, the MySQL root account password may be left blank during install. You must add a password for *root* (administrator) immediately.

Don't use the root account to manage your databases. Set up other admin accounts for that purpose. You should remove all the anonymous user accounts. For more information on securing initial MySQL accounts, see
http://dev.mysql.com/doc/mysql/en/default-privileges.html
.

User Access

User access is where most of the action happens when securing the data in your database. This involves creating accounts, granting access to databases and tables, restricting users to connections from certain hosts, and so on. Another lesson explains how to set up and modify user accounts.

Other areas related to user access include passwords, account privileges, and connections, as discussed in the following sections.

Requiring Passwords

You should require good hard-to-decipher passwords. MySQL doesn't enforce periodic password changes; in fact, it doesn't enforce having passwords at all. When you create a user account, make sure the user is assigned a password or creates a good password. You may choose to manually force a password change periodically. Users can reset their passwords by using this command:

Code Sample: SecureInstall/Demos/Set-Password.sql

```
SET PASSWORD FOR 'filmreader'@'localhost' = PASSWORD('reader');
```

Code Explanation

Set a password for filmreader user.

Controlling Account Privileges

The following are a few suggestions for controlling account privileges:

- Grant the minimal amount of access necessary for users to do their work. Avoid generalized permissions in most cases.
- Don't grant access at all, if it can be avoided. The user who runs an infrequent query or report doesn't need access if another user can run the query and send out results. Lesser accounts mean reduced security headaches.
- Never grant permission to view or change the MySQL privilege tables to general users. Users with INSERT, UPDATE, or DELETE privileges on the mysql.user can cause unforeseen issues. Instead, use the GRANT command to manage all accounts and permissions.
- Cautiously grant the PROCESS and SUPER privileges only to DBAs. With these permissions, a user can issue a SHOW PROCESSLIST command, which allows the user to see all queries issued against the database, including password resets.
- Grant the FILE privilege only to administrative users. This privilege allows a user to create files on the file system as the user running MySQL (Als avoid running mysqld as the root user).

- In some cases, it may be better to give a user the ability to connect to the database and execute a procedure with required statements, instead direct access to a table or set of tables.

- If the user only needs to view certain data fields, create a view that is not updatable to avoid unnecessary permissions and confusion.

If you Forget the Root Password!

What do you do if you have forgotten the *root* password for MySQL and there is no other MySQL user with sufficient administrative privileges and a known password to restore the forgotten password?

Here are some ways of addressing this:

Stop MySQL and then restart it with the option skip_grant_tables. The result is that the table

with access privileges is not loaded. You can now delete the encrypted password for *root*, terminate MySQL, and then restart without the given option. Now you can give the *root* user a new password.

Syntax
```
# changes in my.cnf and my.ini
...
[mysqld]
skip_grant_tables
...
```

You will need system administrator privileges on the operating system hosting MySQL.

Now you can use mysql to reset the *root* password for various hosts:

Syntax
```
root# mysql -u root
Welcome to MySQL monitor.
mysql> USE mysql;
Database changed.
mysql> UPDATE user SET password=PASSWORD('new password')
    > WHERE user='root' AND host='localhost';
Query OK, 1 row affected (0.00 sec)
mysql> UPDATE user SET password=PASSWORD('new password')
    > WHERE user='root' AND host='computername';
Query OK, 1 row affected (0.00 sec)
```

After these changes in the *mysql.user* table, stop the MySQL server. Delete skip_grant_tables option and restart the MySQL server.

Caution: The procedures described here can be used by an attacker as well.

To revoke all privileges issued to a user account, use the REVOKE ALL command:

Syntax
```
REVOKE ALL ON *.* FROM 'sakilauser'@'localhost';
```

This would remove all global privileges from sakilauser@localhost except for the GRANT OPTION privilege.

To include the GRANT OPTION privilege in the REVOKE command, issue the following version:

Syntax REVOKE ALL, GRANT OPTION ON *.* FROM 'sakilauser'@'localhost';

This syntax is available from MySQL 4.1.2. Prior to this version, two statements are necessary to remove all privileges for a user:

Syntax REVOKE ALL ON *.* FROM 'sakilauser'@'localhost';
REVOKE GRANT OPTION ON *.* FROM 'sakilauser'@'localhost';

Limiting User Resources

In addition to useful monitoring resource usage, it's possible to limit the consumption of MySQL resources on a per-user basis. These limitations are managed like any other privilege, via the user
privilege tables:

The request authorization stage determines whether the user has exceeded the maximum allowable connections per hour, and whether the user has exceeded the maximum simultaneous connections (MySQL 5.0.3 and greater). During this stage, the user table also determines whether SSL-based authorization is required; if it is, the
user
table checks the necessary credentials.

In addition to the user account's global privileges, the mysql.user grant table also houses a number of additional fields that can aid you as a database administrator in restricting the account's use of the database server. Starting with version 4.0.2, MySQL provides three fields— max_questions,
max_updates , and ma
x_connections
—which allow you to limit the interaction a particular account has with the server. Before 4.0.2, all you could do was set the
max_connections
configuration variable to limit the number of connections made by a single user account, meaning you couldn't vary the setting per user. Now, you have much more flexibility in how you handle resource usage.

You can update the following columns in user table to restrict user accounts:

- max_connections: Determines the maximum number of times the user can connect to the database per hour
- max_questions: Determines the maximum number of queries (using the SELECT command) that the user can execute per hour
- max_updates: Determines the maximum number of updates (using the INSERT and UPDATE commands) that the user can execute per hour
- max_user_connections: Determines the maximum number of simultaneous connections a given user can maintain

Code Sample: SecureInstall/Demos/Limit-User-Grant.sql

```
CREATE USER 'sakilauser2' IDENTIFIED BY 'sakila';

SELECT max_questions, max_updates, max_connections
FROM mysql.user
WHERE User ='sakilauser2';

GRANT INSERT, SELECT, UPDATE ON sakila.* TO 'sakilauser2'
WITH max_connections_per_hour 1200
 max_updates_per_hour 10000
 MAX_QUERIES_PER_HOUR 30;

SELECT max_questions, max_updates, max_connections
FROM mysql.user
WHERE User ='sakilauser2';
```

Code Explanation

Create a user sakilauser2.

Set upper limits on the user capabilities.

Check the row in user table.

Using the USAGE Privilege to Change Global User Restriction Variables

One good use of the USAGE privilege is in changing these variables without affecting any

other privileges. Code demonstrates changing all three of these variables, as well as a direct query on mysql.user to show the change.

Alternately, you can use the following GRANT options to restrict user accounts:

- MAX_QUERIES_PER_HOUR: Limits the number of queries a user may issue against the server in one hour.
- MAX_UPDATES_PER_HOUR: The number of update requests the user may issue.
- MAX_CONNECTIONS_PER_HOUR: The number of times a user may log in to the database server in a single hour.
- MAX_USER_CONNECTIONS: This option is not time limited and refers to the total amount of connections simultaneously made by any user account, someone with many connections idle or sleeping.

The next elaborate exercise updates user table directly, test the parameters:

Exercise: Limit connections with direct update

Duration: 20 to 30 minutes.

In this exercise, you will limit and test user resources.

1. Create a user filmreader with password as reader.
2. Limit the user filmreader to only one simultaneous connection and 10 queries per hour.
3. Directly update mysql.user table.
4. Open two separate mysql sessions with user filmreader, both sessions go through.
5. What happened here? Why did the limits did not apply?
6. Updating user table directly needs to be followed up with updating grants in memory.
7. Clear up privileges in memory.
8. Try opening two sessions with filmreader user again. You will see an error: ERROR 1226 (42000): User 'filmreader' has exceeded the 'max_user_connections' resource (current value: 1)
9. Now create a stored procedure to query the film table *N* times.
10. Run this proc, passing 15 as the *N* parameter.
11. The execution of procedure should fail after 10 queries: ERROR 1226 (42000): User 'filmreader' has exceeded the 'max_questions' resource (current value: 10)

Using a combination of these resource limiters, you can achieve a fine level of control over the resource usage of user accounts. They are useful when you have a high-traffic, multi-user database server, as is typical in shared hosting environments, and you want to ensure that the database server shares its available resources fairly.

Controlling Connections

A major part of securing your database is controlling the connections to the database. Preventing connections is the first line of defense in keeping unauthorized users out of reach to your databases. Here are some ways that you can control connections:

- Revoke the ability for the root to connect from anywhere except from the localhost. This assumes you have a shell account on the database server. If you do not, you may want to change the host of the root to a specific domain name or IP address to ensure that there is only one place to connect as root.
- Restrict the number of connections that can be made by a user.
- Restrict the number of bad connection attempts that can come from a host before connections are blocked, using the max_connect_errors setting. By default, this is limited to 10, which usually works well.
- Use SSL connections from remote hosts from untrusted or public networks. This is done by adding REQUIRE SSL or REQUIRE X509 in the GRANT statement.

If network connectivity is required for your database server, use a private network or SSL encryption for the data, or set up a secure tunnel using SSH or an SSL tunnel.

You can verify whether MySQL is ready to handle secure connections by logging in to the MySQL server and executing:

```
mysql> SHOW VARIABLES LIKE 'have_openssl';
+---------------+----------+
| Variable_name | Value    |
+---------------+----------+
| have_openssl  | DISABLED |
+---------------+----------+
1 row in set (0.02 sec)
```

There are a number of grant options that determine the server's and the user's SSL requirements. Further SSL setup is out-of-scope in this lesson.

Securing user accounts is an ongoing process. Periodically, review your priviliege tables and system history to verify and enforce your security policies. Also, auditing of the user accounts in MySQL is will help locate and address potential security holes before an incident.

Application Access

Building a secure application is most important to protecting your database. The security of the data is closely dependent on an application's ability to protect the data from misuse.

For extremely sensitive data, you may choose to not create a general-use application account, and have authentication credentials presented via the application for data manipulation. This can be a headache to maintain, especially if you don't create scripts or procedures to manage the accounts.

However, having accounts specifically assigned to users means you control the ability to change data from the MySQL permissions structure and don't need to rely on the application to manage permissions. Also, when a change is made with the binary log enabled, you have a trail of changes that can be tied to specific user connections.

If your system is read-oriented, you may want to create an account with SELECT-only permissions. Create a more privileged user to perform data update tasks.

Data Storage and Encryption

For most applications, entry barriers into the database are sufficient. However, in some cases, the data in the database requires an extra layer of protection.

For instances where data is extremely sensitive and damaging to individuals if tampered with, you might want to consider using one-way encryption, two-way encryption, or no storage options. For details on encryption functions, refer to
http://dev.mysql.com/doc/mysql/en/encryption-functions.html
.

One-Way Encryption

Use one-way encryption for information that doesn't need to be reversed. This technique is used for passwords. The password is encrypted once with a nonreversible algorithm, and then on future attempts to verify the password, the data is run through the algorithm again and compared to the previously computed result.

Functions in MySQL for one-way encryption are password(), old_password(), md5(), encrypt(), and sha1(). These functions are easy to use, and like all functions, are embedded directly into your SQL statements, as in this example:

Signing Your Data

MySQL Functions for generating Signatures are md5() and sha1():

```
        SELECT sha1("Needs serious verification") AS SHA1_Sign;
SELECT md5("Needs serious verification") AS MD5_Sign;
```

Code Explanation

The sh1 generates a signature with SH1 algorithm.

The md5 then generates a signature with MD5 algorithm.

```
      +------------------------------------------+
| SHA1_Sign                                |
+------------------------------------------+
| dbf13ef0ef7d8b131ad82f7b31fe66718228a8f0 |
+------------------------------------------+
...
+----------------------------------+
| MD5_Sign                         |
+----------------------------------+
| 76fbd563dbbfb6e8a91fe489bc2e1837 |
+----------------------------------+
```

You don't have to come up with your own encryption algorithm. Several widely used cryptographic algorithms are already available in MySQL.

Two-Way Encryption

Two-way encryption functions allow you to encrypt data using a value and a key. With the encrypted value and the key, you can get back to the original value.

There are MySQL functions to support two-way encryption: aes_encrypt(), aes_decrypt(), encode(), decode(), des_encrypt(), and des_decrypt().

The Advanced Encryption Standard (AES) is currently regarded as the most sound encryption routines currently available in MySQL. The encoding is performed with a 128-bit key, which is significantly secure and still speedy.

Note: The Data Encryption Standard (DES) functions work only if MySQL has been compiled with SSL support, and uses DES key files from the file system.

As with other MySQL functions, you use the two-way encryption functions directly in your SQL statements, as in this example:

Code Sample: SecureInstall/Demos/AES-Crypto.sql

```
    SELECT aes_decrypt(
aes_encrypt("Must guard this with life!","guardkey"),
"guardkey") AS OrigData;        Code Explanation
```

The aes_encrypt first encrypts the data with given key.

The aes_decrypt then decrypts with the same key.

Because the AES functions generate binary information, an attempt to show the output from the function in text is futile.

Because the AES functions generate binary information, an attempt to show the output from the function in text is of not much use and will likely mess up your interactive session. In this example, we are encrypting a string with the key "SecRet", and then immediately decrypting the result of that encryption using the same key.This use wouldn't be that useful in your application, but it demonstrates using two-way encryption. Here is the output of the statement:

```
    +------------------+
| AES              |
+------------------+
| a horrible secret |
+------------------+
1 row in set (0.00 sec)
```

Attempting to use the incorrect key for decryption will result in a NULL response from the decryption function:

```
    mysql> SELECT aes_decrypt(aes_encrypt("a horrible secret","SecRet"),
"PubLic") AS AES;
+-----+
| AES |
+-----+
| NULL|
+-----+
1 row in set (0.00 sec)
```

Two-way encryption functions in MySQL provide a powerful tool for encrypting data so its value in the database is meaningless unless coupled with the key for decryption.

Don't Store Sensitive Data

A very useful option and the most reassuring way of keeping data secured in your system is to not have the data stored anywhere in your system.

If you accept bank account information in your application and send a request to a third party to charge the user account, you may not need to keep the bank information around after that transaction. The token returned by the payment-processing vendor for future reference may make more sense. Similarly, as you process various use cases without keeping sensitive information, you don't have to worry about guarding what you don't keep.

Securing MySQL Installation Conclusion

This lesson covered the aspects of securing a MySQL installation at operating system level.

To continue to learn MySQL go to the top of this page and click on the next lesson in this MySQL Tutorial's Table of Contents.