KIT304 Server Administration and Security Assurance

Tutorial 11

Goal

In this tutorial you will focus on database administration tasks and learn about a simple database attack.

Introduction

A system administrator can fill a wide range of different roles. So far in this unit you have been exposed to a number of common tasks that a system administrator would undertake, including creating users and groups, modifying filesystem object permissions, and more recently, installing and configuring applications.

A database administrator carries out similar tasks, and in today's tutorial you will build on your earlier experience by creating, restoring and backing up a database.

Database administrators are also involved in other aspects of the database's operation, including its design, and changes to that design over time as the system scales or is migrated to new infrastructure. Database security extends to not just the design of the database, but how it is accessed, and the quality of the code that is passed to it for execution. You will see a classic example of this in today's tutorial when you examine SQL Injection attacks. This is your first step into thinking about security in a practical sense in the lab, and you will be doing more of this in the remainder of this module, and it will also be the focus of the fourth module.

In this tutorial you will use a MariaDB database. Except for syntactic differences and the specific commands used to interact with the database itself, the concepts we are covering are true across all databases, whether open source (like MySQL and PostgreSQL), or proprietary (like MS SQL and Oracle).

Activity

1. Launch the **CentOS** VM and give it an IP address in the **`192.168.1.x`** subnet.

2. Complete a basic secure setup of the Apache web server and MariaDB database engine to the point that you can load **`phpMyAdmin`** in FireFox. Here follows a brief summary of the commands you'll use, but for detailed steps refer back to tutorial 9:

```
firewall-cmd --permanent --zone=public --add-service=http
firewall-cmd --permanent --zone=public --add-service=https
firewall-cmd --reload

service daemon start
mysql_secure_installation
```

Modify the first **Directory** section of **/etc/httpd/conf.d/phpMyAdmin.conf** to match the following:

```
<Directory /usr/share/phpMyAdmin/>
   AddDefaultCharset UTF-8
   <IfModule mod_authz_core.c>
         # Apache 2.4
         <RequireAny>
               Require all granted
         </RequireAny>
   </IfModule>
   <IfModule !mod_authz_core.c>
         # Apache 2.2
         AllowOverride None
         Options None
         Allow from All
         Require all granted
   </IfModule>
</Directory>
```

```
service daemon reload
```

3. In the **Downloads** directory of the **student** account there is a subdirectory called **Tutorial11**. Navigate to that subdirectory and examine its contents – it includes an `.sql` file and several PHP files. Take a look specifically at the contents of the `.sql` file. How would you describe what is in this file?

____

4. Using **PHPMyAdmin**, or `mysql` from the command line, create a database user (with a password), and then create a database called **KIT304**, ensuring that the user you created has all privileges on that database. Once you've done this, quit the MariaDB command line interface. Again, refer back to the tutorial 9 worksheet if you need to double-check the steps.

5. Now that you have a database, you can restore the **.sql** database backup that is in the student account's **Downloads** folder. To do that, you'll enter a command based on the following pattern:

   ```
   mysql -u userName -ppassword databaseToRestore < /path/to/backupfile.sql
   ```

   Note two things here: there's no space between the "**-p**" and the *password* when passing the database user's password on the command line, and that by using the **<** Unix redirector in this way, you're feeding all of the commands in the **.sql** file into the **mysql** (MariaDB) command line, as though you were typing them yourself in the terminal window.

   Note also that with this command, you are entering the user's database password directly onto the command line, where it may be logged, or viewed by someone looking over your shoulder. If you omit the password from the command line (but still enter the **-p** option), you'll be prompted to enter the password before the restore happens.

   For your database and your user, what command did you enter to restore the database?

   ____

6. With your database now restored from the **dbdump.sql** file, use either the command line or **PHPMyAdmin** to explore the database and the tables that it contains. For example, to explore the database from MariaDB, you could use commands as follows:

   - connect to MariaDB: `mysql -u kit304 -p`
   - select your database: `use kit304;`
   - show tables: `show tables;`
   - list content of a table: `select * from Users;`

   Exporting a dump of a database into a **.sql** file, like the file **dbdump.sql**, is a simple and common way to make a complete snapshot (or backup) of a database which are also very easy to restore. But while this technique is sufficient for relatively small databases, as the database grows the approach becomes less useful, and other backup techniques are used.

7. Ensuring that backups work (and are restorable) is a vital part of security planning and policy. For this step, you're going to explore different ways to create database dumps with MariaDB. Try each command from the shell prompt (i.e. not from within MariaDB) and then describe the outcome:

   ```
   mysqldump -u root -p KIT304 > backup1.sql
   ```

   ____

```
mysqldump -u root -p KIT304 Users > backup2.sql
```

____

```
mysqldump -u root -p --databases KIT304 mysql > backup3.sql
```

____

```
mysqldump -u root -p --all-databases > backup4.sql
```

____

```
mysqldump -u root -ppassword KIT304 | gzip -9 > backup5.sql.gz
```

____

The last example *pipes* the output of the **mysqldump** command through the **gzip** command so that the resulting database backup is significantly smaller than it would otherwise be. To decompress the archive you would use the **gunzip** command.

8. What entry would you add with **crontab** so that a **cron job** that would back up the KIT304 database every night at 2 am?

   ____

   Ask your tutor to check your work, and again – note that storing the database password in the crontab file in this way does reduce security.

9. For this next step, you need to create a new website with a domain name of your choosing. This will also require that you create a virtual host configuration file, and add the domain name to **/etc/hosts** so that you can navigate to the site in a browser. Here is a summary of the commands from tutorial 9 that you will need to create your website, but you'll need to figure out where to apply them (or refer back to that tutorial handout):

```
NameVirtualHost *:80
IncludeOptional sites-enabled/*.conf

<VirtualHost localhost:80>
     DocumentRoot /var/www/html
</VirtualHost>
<VirtualHost *:80>
     ServerName www.yourDomain
     DocumentRoot /var/www/yourWebDirectory/html
     ServerAlias yourDomain
     ErrorLog /var/www/yourWebDirectory/error_log
```

```
    CustomLog /var/www/yourWebDirectory/access_log combined
</VirtualHost>

cd /var/www/yourWebDirectory
touch error_log
touch access_log
chcon --reference /var/log/httpd/error_log error_log
chcon --reference /var/log/httpd/access_log access_log
```

Other commands you may need to complete the task: **`service`**, **`mkdir`**, **`chown`**, **`chmod`**.

10. Copy the **PHP** files that you located earlier in the **Tutorial11** directory into the new website that you have now set up

11. Open the newly copied **`dbConnection.php`** file in an editor. This file creates a connection to the database you set up earlier. Make the following changes to the file:

    - modify the **`$path`** variable to reflect the URL of your website.
    - update the password for the **`root`** account used on line 15 to what you used for the root user when you set up MariaDB installation
    - save your changes and quit the editor

12. Open a web browser, and try to load the page **`http://`*`yoursite`*`/awesome.php`**. If you've completed the earlier steps correctly, you should see a list of users and scores – pulled from the database you imported earlier, and now displayed by the PHP scripts. If you don't see this, and instead see a PHP database connection error, check the values you entered at step 11 and try again. Ask your tutor for assistance if you can't see the list of users and scores.

13. If you click on the **Login** link at the top of the page, you will be presented with a login form. Using **PHPMyAdmin**, or the **`mysql`** command line tool (or browse the database **`mysqldump`** file), determine the login credentials for a user, and log in and then log out to observe the site's normal behaviour. Be sure to also attempt to login with incorrect credentials so you can see the normal behaviour for a failed login.

    You may want to write down the credentials you used successfully to save looking them up repeatedly later:

    ____

    Note that you were able to retrieve a user's password from several locations – either the dump file, or the database directly. Being able to retrieve user passwords directly from the database shows an extremely poor security practice. Passwords should never be stored in plain text like this – they should be only ever stored as *hashes* using a one-way hashing algorithm like MD5 or

SHA-256. If they are stored as plain text, anyone who gains access to some, or all, of the database then has the passwords.

Hashing passwords is also not a perfect solution – ideally, they should also include a random *salt* value so that the hashed version is not predictable. Today, however, you will update each password in the database to a hash.

14. One of the PHP files that you should have moved to your web site's **`html`** folder is named **`updatePasswords.php`**. Open it and have a look at the code. When run, this script will iterate through the **`Users`** table, creating an MD5 hash of each user's password which is then written back to the database in place of the original password.

    Run the script ***once*** in a web browser (by loading the page **`updatePasswords.php`** on your site). Do not run it twice (ie., don't reload the page), as this will hash the previously hashed value.

    Once completed you will no longer be able to log in as you previously could (try it, to be sure), because the password you enter is being compared to the hashed version in the database (rather than the previous plain text version. To fix this, you need to add one line of code to the **`logon.php`** file to hash the password entered by the user *before* comparing it to the one stored in the database, using the PHP **`md5()`** function call. At least one line needs to be inserted in front of line 37 (of 54) of the file, what is it?

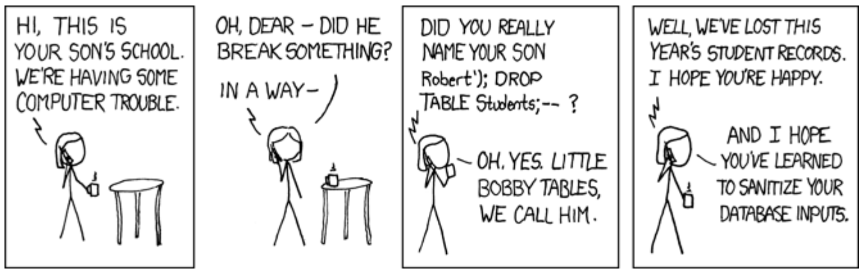
    ____

    After making this change, check again that you can now log in. Now, you're comparing the hashed version of the entered password with the hashed version of the stored password. This is much more secure than storing passwords in plain text.

    ➢ You will learn more about secure password storage in the next module.

15. Applications on systems (including web sites) often have to deal with input from users. In the context of database driven applications and web pages this information is often used directly in a database query. In a poorly-implemented system, if user-provided data contains the right sequence of characters, there is a chance that the database will recognise the input as a valid database command, rather than the subject of a query, and execute it. This is what we refer to as an *SQL injection* – where SQL syntax has been "injected" into the stream of data that is being processed, thus changing the intended outcome of the query.

    Carefully crafted SQL injections can produce changes in database content or structure, or result in information leakage. Such attacks are often simple to execute, and can have dramatic effects, as is illustrated in this xkcd comic (https://xkcd.com/327/):


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

16. On the first page of the website, **awesome.php**, you can see a list of names sorted by a value called **Awesome Score**. At the bottom of the page are three links that reload the page, each limiting the minimum score that should be shown. By clicking on those links you can change how many names are shown on the page.

    The number of names that are displayed on the page is a function of the variable **value** that is passed as a parameter in the URL. For example, clicking on the **Over 25** link loads the URL:

    ```
    http://networks.com/awesome.php?value=25
    ```

    If you change that value (in the address bar), you can directly specify your own minimum score rather than having to use one of the three preset values. This is a simple example of a parameter changing the database query that is executed on the server. There are plenty of examples in the wild where direct modification of a URL-encoded variable can cause information leakage.

17. Click the **Login** link at the top of the page, enter the following values, and write down what occurs:

    ```
    Username: admin' #
    Password: whoops
    ```

    ____

    Why do you think this occurred?

    ____

18. Look at the code in the **logon.php** file, and find the line that starts with **$query = "SELECT *…** (originally line 37). This query selects a record from the database that contains the entered username and entered password. A simplified version of this code (with the PHP string manipulation removed) looks as follows:

    ```
    SELECT * FROM Users WHERE name='username' AND pass='password';
    ```

    What you entered at step 17 changed the query to this:

```
SELECT * FROM Users WHERE name='admin' #' AND pass='password';
```

In the SQL query language, the **#** character marks the start of a comment, so the remainder of the **`select`** statement (notably, the part that includes the password in the search) is not processed. Now, the database is just returning the user named **`admin`**, and since that user is returned, the remaining code assumes the user is authenticated.

19. The “always true” type of exploit, like the one you see above where you disable a large part of the query, is good for getting around poorly implemented authentication mechanisms. These techniques can also be used to extract information with something like the following (the bold part is injected):

    ```
    SELECT * FROM Users WHERE username ='x' or email like '%bob%'; # ' and pass...
    ```

    What do you think this would discover?

    ____

20. MySQL is reasonably robust when it comes to SQL injections. Unlike many other databases, by default it doesn’t allow multiple queries to be executed at the same time. That kind of query is what takes place in the **xkcd** comic, and in the following query. What does this one attempt to do?

    ```
    SELECT * FROM Users WHERE username ='x'; insert into Users ('email', pass,
    'name') values ('bob@bob.com', '464668D58274', 'bob'); # ' and pass...
    ```

    ____

21. To defend against this, you can learn from the comic. It states that you need to “sanitise your database inputs”. This is fundamentally about not trusting any input to the program, but instead, treating it as though it may be malicious, and sanitising all input values. All that the injection examples above required was the addition of a quote character to close the quoted sequence in the SQL select statement, after which could be added other, more malicious, SQL code.

    In PHP, the easiest way to mitigate this kind of attack is to remove the possibility of quotes and other special formatting characters being used unmodified in SQL statements. You can do that with code like this:

    ```
    $safe_variable = mysqli_real_escape_string($unsafe_variable);
    ```

    This built in function “escapes” special characters (such as single and double quotes, as well as newline characters) in an unsafe variable, producing a sanitised, or “safe” version that cannot be used to subvert the normal operation of an SQL query.

Other languages may use phrases such as "add slashes" for this type of sanitisation, as they add a back slash in front of every special character to escape that character. Later, if you want to output or display the sanitized string, you need to "strip the slashes" to return it to its original form.

22. Another effective way to defend against SQL injections is to use *parameter binding* when constructing database queries, as in the following code:

```
$query = $dbh->prepare (
   "INSERT INTO user (name, awesomeScore) VALUES (:name, :ascore)"
);
$query -> bindParam(':name', $_POST['username']);
$query -> bindParam(':ascore', $_POST[score]);
$query -> execute();
```

The resulting query string is similar to that produced by the sanitisation step described earlier. It clearly takes more effort to use parameter binding like this, which may be a factor in why such attacks are rather common. Lazy code isn't usually good code.

Final Note

In addition to the kinds of defences discussed in the latter part of this tutorial, there have in recent years been developments in the area of database firewalls. These are basically a security policy layer that sits between the database and the applications that use it. Queries have to pass a series of tests before they will be executed.

Although we've only covered SQL injection attacks in this tutorial, malicious code injection is fundamental to many attacks. For example, in the web context of this module, JavaScript Injection attacks are another common threat you may face as a web administrator. While this type of attack only runs inside the browser, because of the significant power that JavaScript has over the browser's execution environment, it has the capability to monitor user input, and modify it before sending it to a server. Similarly, it can parse and process data returned from the server without the user being aware of this – potentially passing that data on to other sites as part of an exfiltration attack.

Skills to Master

To successfully complete this tutorial, you must have mastered the following skills:

- dumping the content of a MySQL/MariaDB database to a `.sql` file
- importing data into a MySQL/ MariaDB database from a `.sql` file
- sanitizing PHP input that is passed to an SQL query

You could be required to demonstrate any of these skills in this module's practical exam.