**Rails Security Features**

**Introduction**

The threats against web applications include user account hijacking, bypass of access control, reading or modifying sensitive data, or presenting fraudulent content. In order to provide security to your application and prevent attacks you have to fully understand the attack methods in order to find the correct solution. Following are some methods of attack.

#### **Session Hijacking**

#### Many web applications have an authentication system. A user provides a username and password, the application authenticates the user and stores user id in the session hash. The session id stored in the cookie identifies the session. Hence anyone who seizes a cookie from someone else, may use the web application with possibly severe consequences.

### For the web application developer, it is a best practice to provide a secure connection to the web server over SSL. In Rails 3.1 and later, this could be done by always forcing SSL connection in the application config file.

config.force\_ssl = true

#### 

#### **Session Fixation**

#### In session fixing, the attacker fixes a session id known to them. This attack focuses on fixing a user's session id and forcing the user's browser using this fixed id. Hence the attacker don’t need to steal the session id afterwards.

### The most effective solution for session fixing is to issue a new session identifier and declare the old one invalid after a successful login. The following code will protect you from session fixation.

reset\_session

#### **Cross-Site Request Forgery (CSRF)**

#### CSRF is a type of attack that occurs when a malicious program causes a user's web browser to perform an unwanted action on a trusted web application. If the session for that trusted web application has not timed out, an attacker may execute unauthorized commands.

#### Rails protect against CSRF attacks by default by including a token named authenticity\_token within HTML responses. Although it is enabled by default, developer must double check the protect\_from\_forgery method is within the ApplicationController.

CSRF protection doesn’t apply to GET requests. So the GET requests should not be used to change the application state or any content in the database.

#### **File Uploads**

#### Many applications allow users to upload files. File name should be filtered always, as the attacker can enter anything in file name. Let’s have an example of file name as "../../../etc/passwd". By uploading the file as this name will replace the file that contains password information of the server if your application is running with root privilege.

#### All the ‘/’, periods (.) numbers must be replaced with underscores ( \_ ) so that this type of vulnerability can be avoided.

#### **SQL Injection**

#### SQL injection is when a user is able to manipulate a value which is used unsafely inside a SQL query. The bug allows SQL injection through dynamic finder methods. This can lead to data leaks, data loss, elevation of privilege, and other unpleasant outcomes.

#### Let's consider following code snippets.

#### User.where("email = #{payload}").first

#### Since the attacker has full control over payload, they can insert whatever they would like in the where query. For example

#### payload = "someone@example.com; DROP TABLE users;"

#### The above payload with where query will execute as two SQL commands in the database like this

#### SELECT \* FORM users WHERE email='someone@example.com; DROP TABLE users;'

#### This will result the entire data loss from user table.

#### The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures, and avoiding dynamic queries. Following are some examples for safe queries.

#### User.where(email: email)

#### User.where("email = ?", email)

#### User.where("email = :email", email: email)

#### **HTML/JavaScript Injection**

#### In this type of attack the attackers inject HTML or javascript codes into your application. Escaping user input is essential before storing the data or manipulating with it.

#### Consider the following code injection

#### <script>window.location.href="soemeevilsite.com"</script>

#### Whenever a user tries to visit your application they will be redirected to the soemeevilsite.com given by the attackers. That means your application will become inaccessible**.**

#### **CSS Injection**

#### CSS Injection one kind of JavaScript injection, because some browsers (IE, some versions of Safari and others) allow JavaScript in CSS.

#### Consider the following code

#### <div style="background:url('javascript:eval(document.evil.script)">

#### Here the payload is in the style attribute. The JavaScript eval() function executes any string as code.

#### It is good practice to restrict custom CSS in your web application.

#### **Conclusion**

#### Rails does many things automatically when it comes to application security. While developing an application in Rails, it feels like Rails has your back. Although Rails provides many things for security, as a developer you should take care whether everything is in place and double check your application for any kind of security holes.