Nicola Mores, 15/03/2024

Challenge: Auction

Background

This challenge is about Injections and in particular SQL Blind Injection. This type of web attack consists in injecting arbitrary SQL code through user input in order to retrieve secrets, modify data or bypass access control measures [1]. In this challenge the application uses input provided by the user in order to store in a database data like user authentication credentials and the bids for the various products available. In order to use this information, the server needs to create various SQL queries using the provided input, with some code like `“SELECT x FROM y WHERE z = ‘“ + userInput +”’”`. Without proper sanitization or without the usage of specialized functions, like `prepare()` in PHP, code like this could be exploited leading to a vulnerability in the system, even when the app doesn’t directly return data to the user. When this is the case we are working with Blind Injections [2] and we can indirectly retrieve information by injecting some code that affects, under particular conditions, the behavior of the app, like adding a delay of some seconds using the function `sleep()`.

Vulnerability

In this case the vulnerability arises from the (partially) unsanitized input used during the bid process, where we can insert, besides the offer, SQL code that allows us to make a Blind SQL Injection. By doing so we can reconstruct the database schema and also hidden data like the password of the users.

Solution

As always, when we are working on challenges like this one, we have to understand the purpose of the application and how it manages to achieve it. In multiple pages of the app, users are allowed to insert their inputs, which we can imagine are used communicating with the database. Noticing this, we can try to find out if there is a vulnerability by sending malicious code, following some common patterns, like adding a `sleep(5)` after the main request body and checking if this affects the behavior of the app. By doing so, we can encounter a change in the response time in the first request made by the products’ bidding page. Since we have discovered the vulnerability, we now have to successfully exploit it to retrieve the admin’s password, as suggested by the challenge. First, we need to get the schema of the database in use, so that we know the tables’ and columns’ names, fundamental information in such scenarios. We can get these data by adding something like `AND (SELECT (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE '%') AND sleep(5)` [3]. By doing so and changing the `LIKE` operator we can find out the names of the tables in the database and notice a table “user”, that might be our target table. Similarly, we can work on the column `column_name` of `information_schema.columns` specifying the `table_name` “user” in the `WHERE` clause, in order to retrieve also the columns’ names: “username” and “password”. Once again we can apply the same reasoning in order to get the admin’s password, using queries like `AND (SELECT (SELECT password FROM user WHERE username = 'admin') like BINARY '%') AND sleep(2)` [4]. Notice the `BINARY` in order to have a case sensitive evaluation. This concludes the challenge, providing us with the credentials needed in order to get the flag.

References

[1] OWASP SQL Injection: https://owasp.org/www-community/attacks/SQL_Injection
[2] OWASP Blind SQL Injection:
https://owasp.org/www-community/attacks/Blind_SQL_Injection
[3] Table names finder Code Example:
https://gitfront.io/r/NikoMrs/PrSpp9XjTGu2/EthicalHacking2024/blob/Assignment2/tables.py
[4] Password finder Code Example:
https://gitfront.io/r/NikoMrs/PrSpp9XjTGu2/EthicalHacking2024/blob/Assignment2/password.py