Dojo challenge #37 "Hacker forum" - writeup

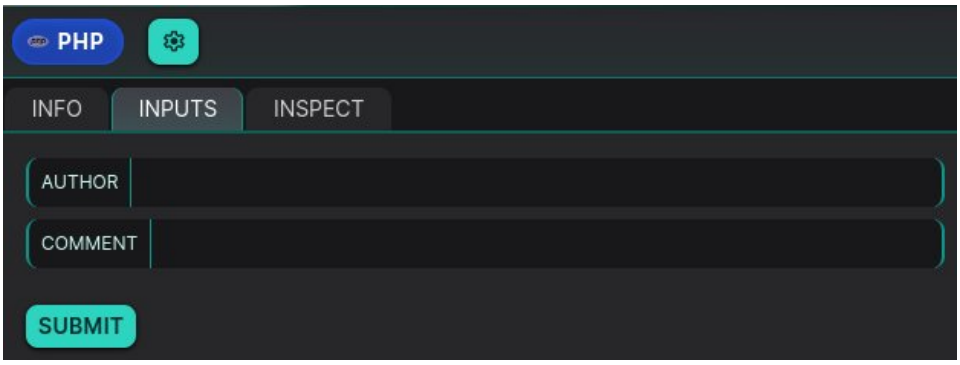
Part1: Challenge and code analysis

The challenge description indicates that the scope is a recent hacking forum that contains a zero-day vulnerability. The language used is PHP.

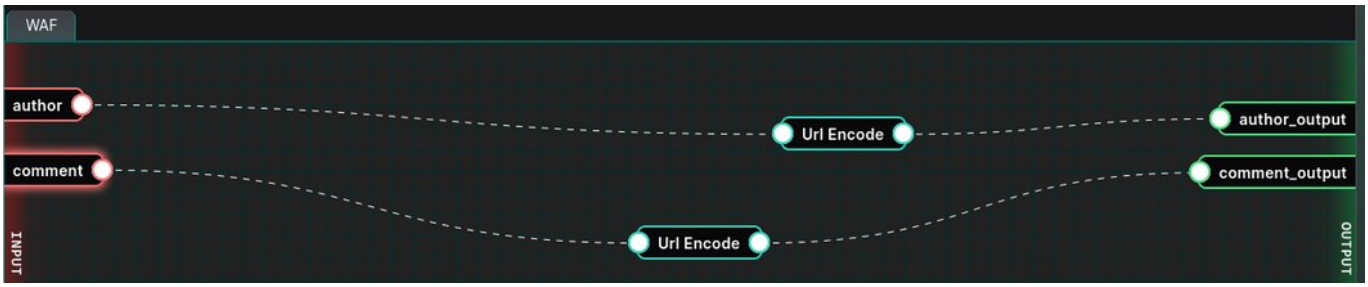
The task is to fetch the password of user `brumens`.

A quick analysis allows us to notice that the forum is comprised of a post and that we are allowed to comment on that post.

For this, we can provide two values in the corresponding input fields: `author` and `comment`.



Before reading the code, let's focus on the WAF part: the two input fields are processed by a "URL encode" filter.



When reading the code, we first notice that the two input fields pass through the `urldecode` function. Hence, we can ignore the WAF part: whatever string we put in the input fields, those strings will be URL-encoded and then URL-decoded, so there will be no restriction in what we can write there.

```
// User input
$input_author = urldecode("(author_output: ) ");
$input_comment = urldecode("(comment_output: ) ");
```

We see that the code uses an SQLite3 connection. We see a first SQL query, which looks totally safe: it is a prepared statement with two parameters (`$input_author` and `$input_comment`) passed with bindings. So the input values we provide will be used to create a line in the `comments` table, for `post_id` #1. Interesting, but there is nothing to exploit here.

```
$stmt = $db->prepare(
    "INSERT INTO comments (post_id, author, comment, image) VALUES (1, :author, :comment, 'https://static.vecteezy.com/system/resources/previews/
    035/672/488/non_2x/ai-generated-orange-cat-with-sunglasses-giving-thumbs-up-on-transparent-background-free-png.png')"
);
$stmt->bindValue(":author", $input_author, SQLITE3_TEXT);
$stmt->bindValue(":comment", $input_comment, SQLITE3_TEXT);
$stmt->execute();
```

In the last part of the code, it is quite obvious that two SQL queries might be injected: for those, the query string is modified using the `sprintf` function. This is inherently unsafe: a prepared statement (like previously) would be the safe approach.

```php
$posts = $db->query("SELECT * FROM posts");
while ($p = $posts->fetchArray(SQLITE3_ASSOC)) {
    $post = new Post();
    $post->makePost($p['author'], $p['banner'], $p['title'], $p['post']);

    $comments = $db->query(
        sprintf("SELECT author, comment, image FROM comments WHERE post_id = '%d'", $p["id"])
    );
    // Ban haters (haters gonna hate)
    while ($comment = $comments->fetchArray(SQLITE3_ASSOC)) {
        if ( preg_match("/(bad|terrible|worst|skid)/", $comment['comment']) ) {
            $db->exec(
                sprintf("UPDATE users SET banned = 1 WHERE username = '%s'", $comment['author'])
            );
            $post->addComment($comment["image"], $comment["author"], "*****", true);
        } else {
            $post->addComment($comment["image"], $comment["author"], $comment["comment"]);
        }
    }
    array_push($lst_posts, $post->get());
}
```

We also notice that for the first unsafe SQL query, the parameter used (`$p['id']`) is the ID of a post and we can't control it.

So, we will focus on the second unsafe SQL query and try to inject `$comment['author']`, which comes from the `author` field that we submit.

We note that this SQL query will only be performed if `$comment['comment']` matches the regex `/(bad|terrible|worst|skid)/`.

Part2: Exploitation

We want the vulnerable SQL query to be executed, so we input the word `bad` (or `terrible`, etc.) in the `comment` input field.

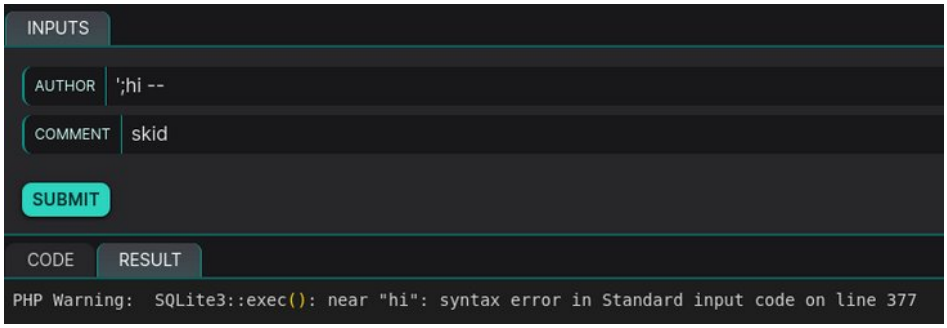
Out inputs will be used by the prepared statement to create a comment. And because our comment contains that `bad` word, the second unsafe query will be executed, using the content that we provide in the `author` input field.

Now, we want to write SQL instructions in the `author` input field.

Those instructions should contain a single quote that will close the one already opened. For the second already opened single quote, we can simply end our instructions with `--` so anything after will be treated as a comment.

Let's start with a simple payload: `';hi -`

In the result, we can see a SQLite3 error, which proves that the instructions we provide are interpreted.



Now, let's think of the right payload. Here are a few elements to consider:

- it looks like a blind SQL injection, because we inject an `UPDATE` statement: we can add a `SELECT` statement, but the result of it will not be displayed in the page
- we could treat this like a blind SQL injection or "error blind" SQL injection (see Hacktricks) and make tests to guess the letters of the password (one by one), but it would be very long or require automation that we cannot set up in the Dojo

So let's think a little more: the vulnerable SQL query is placed in a `while` loop that selects all the comments for the current post.

So if we add a comment for the current post, the next iteration of the `while` loop will load our inserted comment and display it in the page! We only have to write the flag to this comment and it will appear in the page.

Let's read the challenge settings carefully to avoid any mistakes in the column names or table names.

We notice that the flag is the password of the first user, which we can select using its username or `id=1` or simply select users without a `where` statement but with `LIMIT 1`.

So know how to select the flag: `SELECT password FROM users LIMIT 1`.

We also know how to insert a comment: `INSERT INTO comments (post_id, author, comment, image) VALUES (1, 'a', 'b', 'c')`

We combine the two statements. We can write the flag to any column that will be used to render the comment, for instance `author`.

Part3: PoC and success!

Now, the payload becomes quite obvious:

```
a';INSERT INTO comments (post_id, author, comment, image) VALUES (1, (SELECT password FROM users LIMIT 1), 'test', 'test') --
```

Bingo! We get a congratulation message and we read the flag in the HTML result:

Thanks for providing this challenge (my first!).

As for the CVSS calculation, I propose CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H because without authentication we can read/write/delete any data in the database. Some may argue that the availability do not concern the data but the platform itself (see https://www.first.org/cvss/v3.1/user-guide 3.2).