Pentesterlab Serialize Badge/API to Shell

Saikat Karmakar | Sept 8 : 2021

Introduction

This course details the exploitation of two vulnerabilities:

- A weakness in a signature check due to PHP type confusion.
- A call to PHP `unserialize`.

Using the first bug, you will be able to retrieve the source code of the application. Then, using this code, you will be able to find a call to `unserialize` and exploit it.

The application

The application allows you to download and upload files. A signature protects the download functionality. Only the server can issue a correct signature. You can retrieve a valid signature for your files by using the `list` API call.

To get started, register an account, upload a file, list all your files and retrieve the file you just uploaded.

When you are retrieving a file, you can tamper with the signature `sig` and see that any modification to this value will prevent you from accessing this file.

Weakness in signature check

Signature

Often an application needs to protect access to a resource. To do so, the application can sign part of the request and provide the signature. Since the application is the only one with access to the `secret` used to sign the request, no one can guess the signature and access the file. For example, this mechanism is used in AWS S3 to secure the access to resources.

When the application receives a request to access a protected resource, it will compute the signature based on the `secret` and the part of the HTTP request (the `uuid` parameter in this exercise). Then the application will compare the signature computed with the one provided by the client. If they match, the client will get the resource. If they don't match and error will be returned.

PHP comparisons

PHP provides two ways to compare two variables:

- Loose comparison using ==/!=. Both variables have "the same value".
- Strict comparison using ===/!==. Both variables have "the same type and the same value".

Other functions also allow you to use these two types of comparison. For example, the function `in_array` allows you to force a strict comparison by setting the variable `$strict` to `TRUE` (`FALSE` by default):

bool in_array (mixed $needle , array $haystack [, bool $strict = FALSE])

We can test the behavior of these comparisons using PHP interactive shell. Here we can see a simple example of the comparison, the string `"1"` and the integer `1` are not equals using a strict comparison (`bool(false)`) but are equals using a loose comparison (`bool(true)`):

```
% php -a
Interactive shell

php > var_dump("1" === 1);
bool(false)
php > var_dump("1" == 1);
bool(true)
```

This behavior helps PHP being a language with a very low barrier to entry for developers coming from other languages as they will be able to compare a string (for example an HTTP parameter) with an integer.

However, we can see that this behavior can be surprising:

```
php > var_dump("a" == 0);
bool(true)
php > var_dump("1' or 1=1" == 1);
bool(true)
php > var_dump("<script>alert(1)</script>" == 0);
bool(true)
```

Loose comparison and signature

This issue got hyped in 2015 under the name "Magic hash" (http://blog.astrumfutura.com/2015/05/phps-magic-hash-vulnerability-or-beware-of-type-juggling/). However, a lot of people have been researching how this comparison may affect the security of an application (an especially signature comparison) before that (since at least 2009).

Regarding signature comparison, most of the research was done using HTTP parameters with URLs like `bug.php?value=123&sign=12b...12ca`.

If the hash computed starts with "0e" (or "0..0e") only followed by numbers and the parameter provided is `0`, the two values will be equal if the application uses a loose comparison. However, it's very unlikely to get a hash that will follow this pattern. More details can be found in this write-up of a vulnerability in Simple Machine Forum: http://turbochaos.blogspot.com.au/2013/08/exploiting-exotic-bugs-php-type-juggling.html. This issue was exploitable (using brute-force) in Simple Machine Forum, as Simple Machine Forum was only using a subset of the hash.

Here, our application is a bit different as it's using JSON.

Impact of using JSON

As you know, the application uses JSON (JavaScript Object Notation). One of the key thing with JSON is that the person calling the API can decide on the type of data:

```
{ "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "height_cm": 167.6,
  "children": []
}
```

Here we can see that multiple types are used: `Array`, `Hash`, `Integer`, `Float`, `String`. Since the application is using JSON, we can decide on the type of the signature `sig`.

Using the fact that we can force the type of the signature, we can decide what the type of the signature is, we can force a comparison between a string and an integer and hope that the application is using a loose comparison.

The following example illustrate how the computed hash will match a signature using a loose comparison:

- If the computed hash starts with `0` (followed by a letter) or starts by a letter: `a` to `f`; it will be equal to `0`.

- If the computed hash starts with `19ee` it will be equals to `19`.
- If the computed hash starts with `3fee` it will be equals to `3`.
- …

You can solve this problem using two methods:

- You can tamper with the `uuid` parameter and compare it to the signature `0`.
- You can tamper with the `uuid` and play with the signature `sig` by increasing the value of `sig` (starting from `0`).

Using the most efficient method (first method), you can play with the `uuid` value and set the signature to `0`:

- `"uuid": "../../../etc/passwd"; "sig": 0;`
- `"uuid": "../../.././etc/passwd"; "sig": 0;`
- `"uuid": "../../.././././etc/passwd"; "sig": 0;`
- `"uuid": "../../.././././/./etc/passwd"; "sig": 0;`

Until you find a value for which you can use a signature of `0` and get the file. Adding `./` will still give you the same file but the application will generate a different signature that may be loosely equal to `0`.

Once you managed to retrieve the file `/etc/passwd`, you should be able to retrieve all of the application's source code. Then you can start reviewing the source code to find more vulnerabilities.

Unserialize exploitation

In PHP, if the application unserializes data under your control, you can potentially trigger unexpected behavior. To do so, you will need to find an object with a call to one of these "trampoline" functions:

- `__wakeup()` when an object is unserialized.
- `__destruct()` when an object is deleted.
- `__toString()` when an object is converted to a string.

By reviewing the source code of the application, you probably discovered that the `token` used for authentication was a serialized `User` object signed using a HMAC. However, the `User` class does not use any of the function listed above. To exploit this issue, we will need to use another class. The class `File` seems to contain everything we need:

```
1  <?php
2
3  class File {
4    public $owner,$uuid, $content;
```

```
5    public $logfile = "/var/www/logs/application.log";
6
7    function __destruct() {
8      // Loogging access
9      $fd = fopen($this->logfile, 'a');
10     fwrite($fd, $_GET['action'].":".$this->uuid.' by '.$this->owner."\n
           ");
11     fclose($fd);
12   }
13 [...]
```

First, we need to piece together all we need to generate a token:

- a serialized `File` object with our malicious payload.
- a valid signature for this object using the code and the `KEY` available in the application's source code.

Now, that we can generate a token, we are going to exploit this issue in two steps:

- Use the call do `__destruct()` to create a PHP file in the web root of the application.
- Access this PHP file.

The final trick for this exercise is to find the right API action to call, you need to remember that the application expects a `User` object and you are providing a `File` object. If the application calls a method of the `User` class on the `File` class, it will crash. To complete this exercise, you will need to find an action that does not call any method on the object provided.

Conclusion

This exercise showed you how to bypass a signature by forcing the type of the signature. Using this vulnerability, you were able to get access to the source code to find new issues. Here the application was unserializing a signed value, however, as soon as the key gets leaked (using the first vulnerability), you were able to forge your own data and exploit the call to `unserialize`. I hope you enjoyed learning with PentesterLab.