**Academy walkthrough**

# **Index**

[Index 1](#_Toc202785321)

[List of pictures 1](#_Toc202785322)

[Disclaimer 2](#_Toc202785323)

[Reconnaissance 2](#_Toc202785324)

[Initial foothold 2](#_Toc202785325)

[User flag 3](#_Toc202785326)

[Privilege escalation 6](#_Toc202785327)

[Personal comments 7](#_Toc202785328)

[Appendix A – APP Key and CVE 2018-15133 7](#_Toc202785329)

[References 9](#_Toc202785330)

# **List of pictures**

[Figure 1 - nMap scan results 2](#_Toc202782349)

[Figure 2 - FFUF scan results 3](#_Toc202782350)

[Figure 3 - New admin registration 3](#_Toc202782351)

[Figure 4 - New not fixed subdomain found 4](#_Toc202782352)

[Figure 5 - Laravel technology used 4](#_Toc202782353)

[Figure 6 - Shell obtained 4](#_Toc202782354)

[Figure 7 - Credentials found 5](#_Toc202782355)

[Figure 8 - User flag 5](#_Toc202782356)

[Figure 9 - User group 6](#_Toc202782357)

[Figure 10 - New credentials found 6](#_Toc202782358)

[Figure 11 - mrb3n sudoers 6](#_Toc202782359)

[Figure 12 - Root shell and flag 7](#_Toc202782360)

# **Disclaimer**

I do this box to learn things and challenge myself. I’m not a kind of penetration tester guru who always knows where to look for the right answer. Use it as a guide or support. Remember that it is always better to try it by yourself. All data and information provided on my walkthrough are for informational and educational purpose only. The tutorial and demo provided here is only for those who are willing and curious to know and learn about Ethical Hacking, Security and Penetration Testing.

Just to say: I am not an English native person, so sorry if I did some grammatical and syntax mistakes.

# **Reconnaissance**

The results of an initial nMap scan are the following:

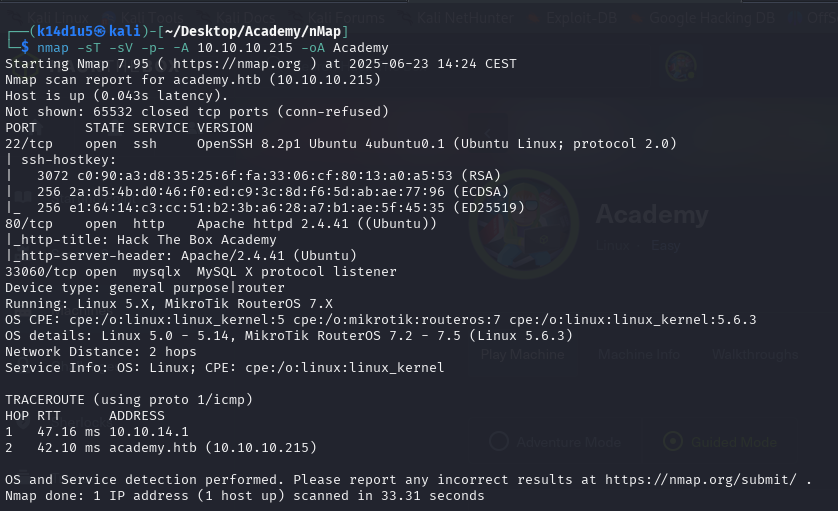


Figure 1 - nMap scan results

Open ports are 22, 80 and 33060. Therefore, enabled services are SSH (22), MySQL (33060) and there is a web application running on port 80. Also, nMap recognized Linux as operative system.

# **Initial foothold**

My first point of attention is the web application. I was able to log in using credentials and I was able to create a new account too. In the meanwhile, I run ffuf to search some hidden web content:

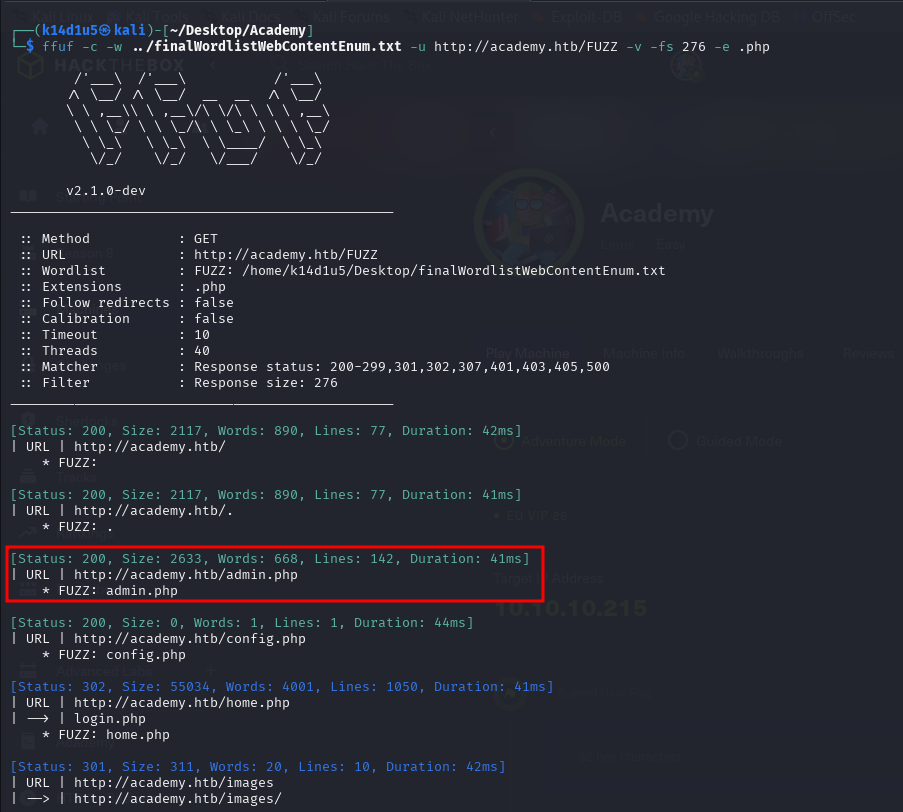


Figure 2 - FFUF scan results

# **User flag**

Since I found an interesting path, I navigate to it. On the path, I found a new login. However, I can’t log in here and I can’t register a new user from here. Therefore, I use the index page’s log in to register a new user. This time, I intercepted the request using Burp and I found out I can modify the role. In this way, I was able to register a new admin, as shown in the following figure:



Figure 3 - New admin registration

Using the new account I just created, I was able to log in on the admin page. In this page, I found out that a specific subdomain wasn’t fixed yet:

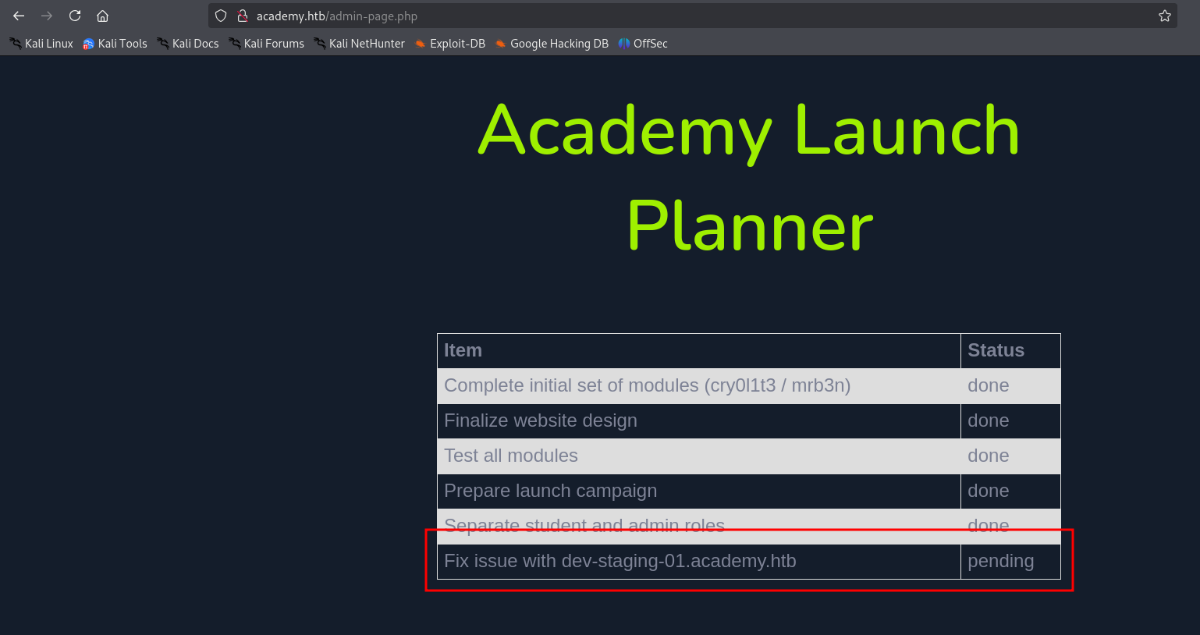


Figure 4 - New not fixed subdomain found

Therefore, I add the subdomain to my file and I browsed to it. Here, I found a lot of information. For example, I found an APP Key and that the site was developed in Laravel:

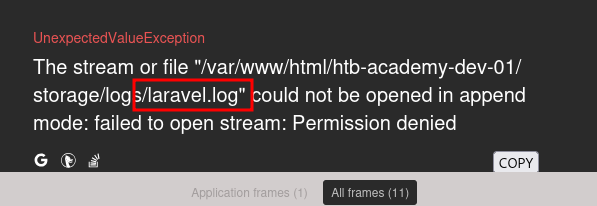


Figure 5 - Laravel technology used

At this point I looked for some interesting Laravel exploit and I found one on Metasploit. I set up the module using even the APP Key I found before, and I obtain a shell as :

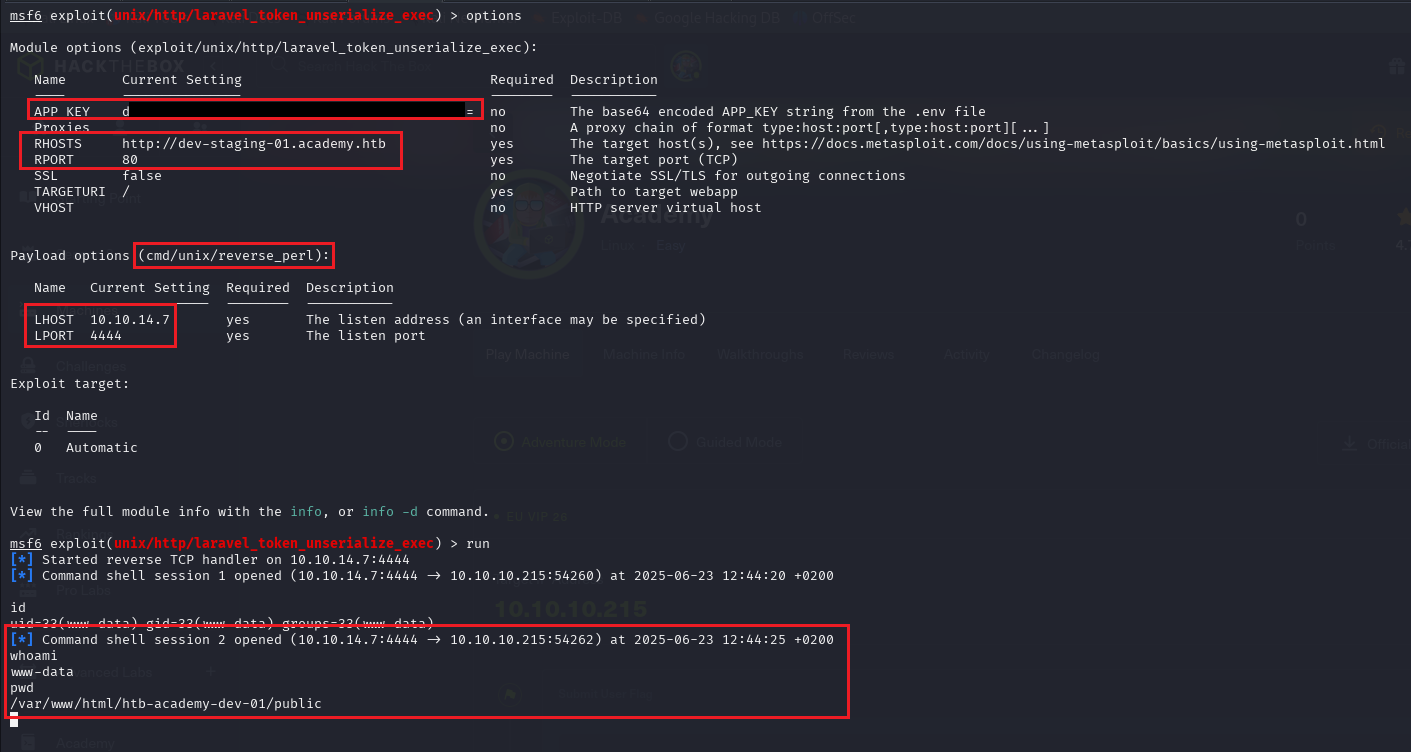


Figure 6 - Shell obtained

Therefore, I need to find some information to obtain a user shell. Browsing the file system, I found an interesting file in the web application folder. In this file, I found some credentials:

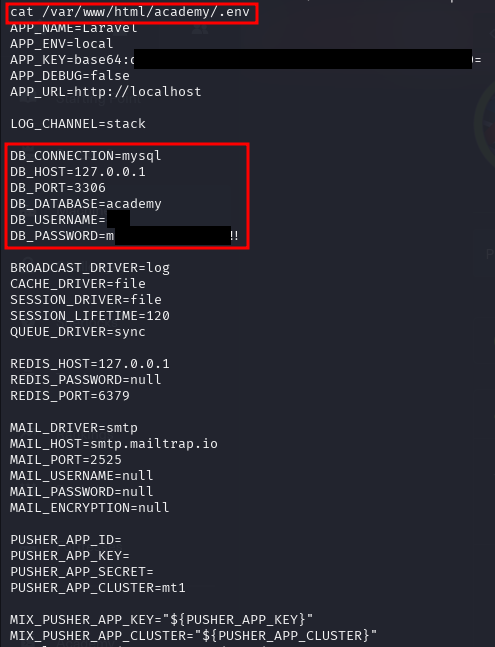


Figure 7 - Credentials found

Even if they are database credentials, I was not able to access to the database. Also, I read the file and I tried to log in as a user using the password I just found. Luckily, I found valid credentials for user:

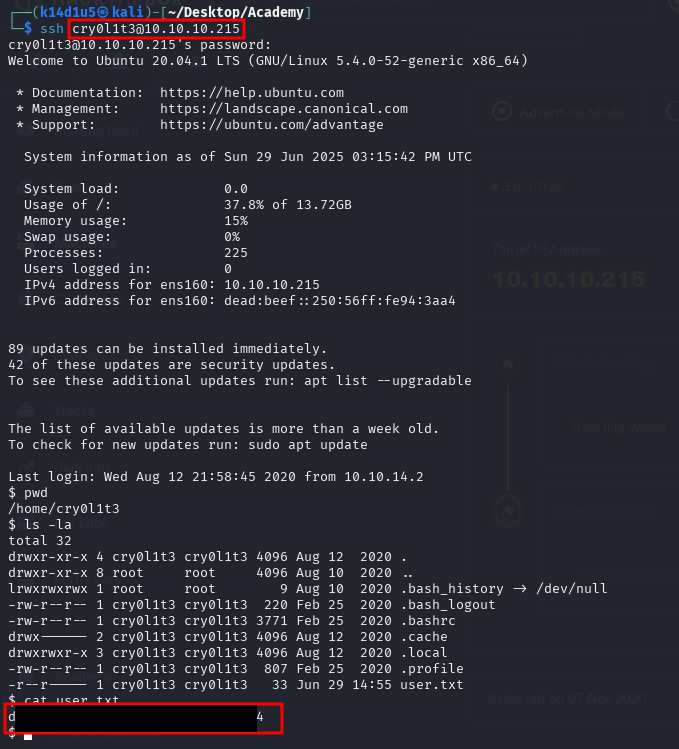


Figure 8 - User flag

# **Privilege escalation**

At this point, I needed to escalate my privileges. One of the first task I performed was checking in which group the user is in. Luckily, it is in a very interesting group:



Figure 9 - User group

Since the user is in the group, it is able to read system logs. Of course, there are so many logs to check. Therefore, after a while I found something interesting in the audit logs. I was able to read them using the command:

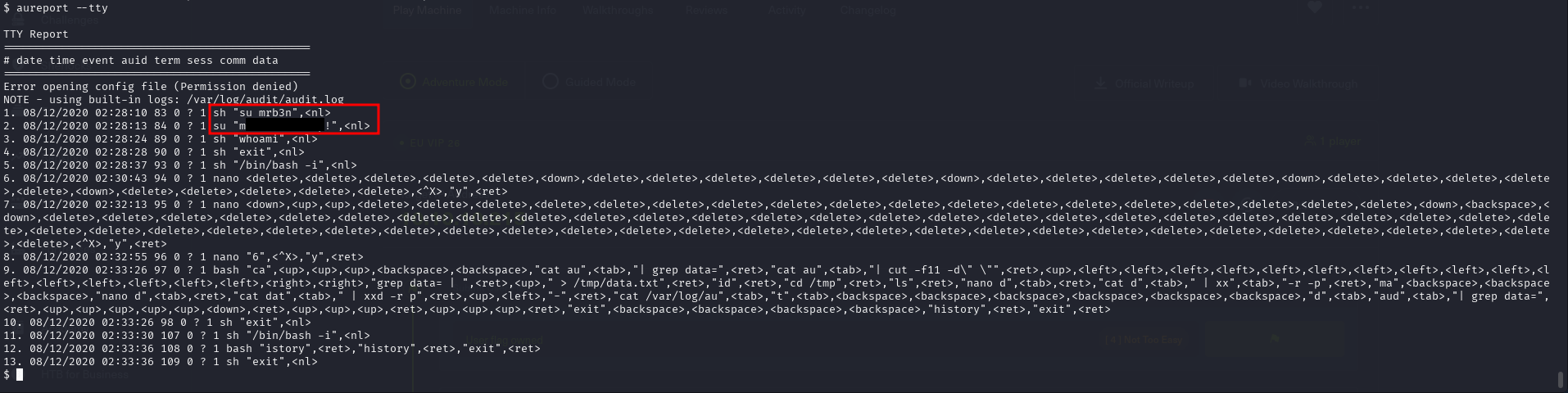


Figure 10 - New credentials found

Using these credentials, I can log in as user. At this point, I started again to look for a way to escalate my privileges. Luckily, this new user can run a specific program as :

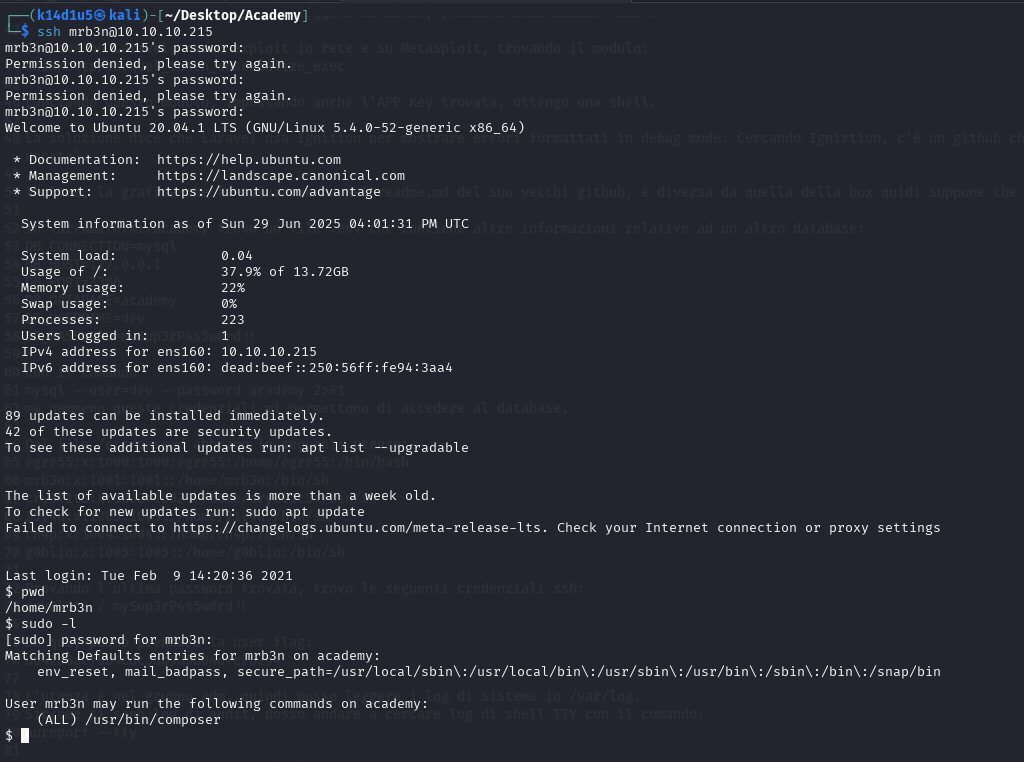


Figure 11 - mrb3n sudoers

Therefore, I looked for and found a plausible exploit on GTFObins. Following the procedure, I obtain the root shell and the root flag:

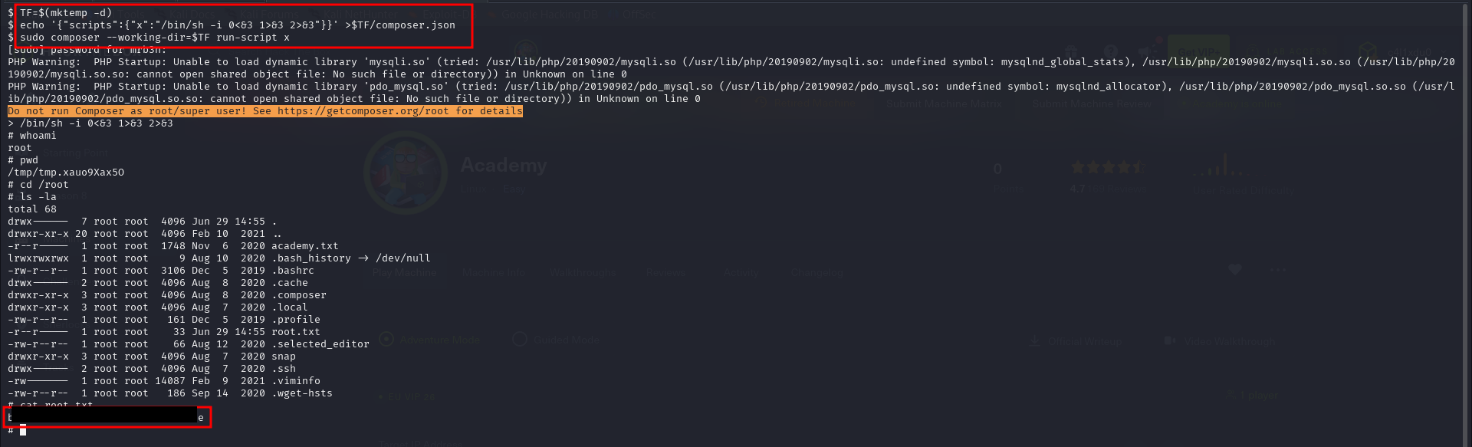


Figure 12 - Root shell and flag

# **Personal comments**

It could be considered as an Easy box, in fact the hardest part were found the APP Key (you must note it), analyze the system log and found the new credentials. However, you can’t have a version of Laravel, so you just have to try the exploit. Lastly, I found it very interesting and I learned something new.

# **Appendix A – APP Key and CVE 2018-15133**

Laravel expects its environment file within the root folder of the app. This file contains several Laravel configuration settings, including secrets such as the APP\_KEY, database credentials and general settings. Leaking of the file content (e.g. through an arbitrary file read vulnerability), has a severe impact, as the leaked information can be abused in multiple ways.

The most prominent secret is the APP\_KEY, which often can be abused to gain RCE. The most noteworthy functions that use the APP\_KEY are:

* Laravel’s default [“encrypt()” and “decrypt()”](https://laravel.com/docs/9.x/encryption) functions. If developers want to encrypt/decrypt any value or object, then they most likely will use these functions. They are also used to encrypt, and therefore tamper-protect, Laravel’s session cookies.
* Laravel also has an in-built feature to create tamper-resistant [signed URLs](https://laravel.com/docs/9.x/urls#signed-urls). Furthermore, most signing-functions use the applications APP\_KEY as secret.
* In a complex web application, you might see the need to queue tasks which are not time sensitive (e.g. sending a reminder mail). Such tasks can be handled through Laravel [queues](https://laravel.com/docs/9.x/queues). As queue objects might be stored externally, they are also signed (e.g. [here](https://github.com/laravel/serializable-closure/blob/master/src/SerializableClosure.php#L32-L35)) with the APP\_KEY.

If attackers have knowledge of the APP\_KEY they can exploit a vulnerable Laravel instance by:

1. Creating a malicious serialized PHP object
2. Encrypt the object with a leaked APP\_KEY
3. Send the payload to the vulnerable opcache handler: https://<vulnApp>/opcache-api/status?key=<encryptedPayload>
4. The application will try to insecurely decrypt the attacker-controlled object.

By design, [Laravel Queues](https://laravel.com/docs/9.x/queues) need to temporarily store tasks and objects within an (external) queue provider. To prevent tampering at rest, these objects are partially signed with the APP\_KEY.

Since that Laravel handles queue objects insecurely before the signature validation check. This allows any attacker with access to the configured queue (e.g. AWS SQS access) to gain remote code execution, even without knowledge of the APP\_KEY.

To understand the issue, we need to have a general overview of queue object structure. Usually, a queue object contains the following (for us important) elements:

* job - The class which queued the job
* data - The data object which holds our actual Queue command which will be executed
* data.commandName - The name of the command
* data.command - The actual command as a serialized object
* data.command.hash - A signature of data.command, to prevent tampering

The command object contains a hash which ensures that the serialized object was not tampered with. However, as the hash is part of the serialized PHP object, this check can only be performed **after** the object is unserialized.

Due to this the unserialize call on the command object is performed without any prior validation, resulting in an insecure deserialization vulnerability.

One exploit path is “Exploit due to arbitrary scopes for Queueing Closures”. Closures are “simple tasks that need to be executed outside of the current request cycle”, and they allow an attacker to execute arbitrary PHP code through a queue.

However; in this scenario an attacker needs access to the Queue and the APP\_KEY. If the Laravel environment file is disclosed, then these conditions are often met as the file contains all necessary information. Unlike the previous example, this approach does not rely on deserialization and will therefore also work if no working gadget chain is available.

An attacker can execute arbitrary PHP code, as long as the scope within the queueing closure job exists. A malicious closure can be sent through attacker malicious Laravel Artisan command. Please, note that the function used to interact with application scope needs to exist within the target application as well.

During the job dispatch the queueing closure is created and signed with the leaked APP\_KEY. Following we can see the serialized job stored within the queue. Our SerializableClosure can be found within the command object. As the attacker sent a closure to the queue, all required function definitions are included within the closure. Once the target application retrieves the queue object it tries to (insecurely) deserialize the command object. Once this is done, Laravel validates the hash with the APP\_KEY. After the hash is verified, Laravel will try to resolve the scope App\Http\Middleware\EncryptCookies. If this scope exists, then the attacker-controlled closure/code is executed. This can immediately be verified, as our attacker-controlled echo is called from within the context of the targets’ [queue worker](https://laravel.com/docs/9.x/queues#running-the-queue-worker).

Especially with closures in mind, it should be mentioned that Laravel has no expectations on what it will accept through queues. The only required setup for a developer is the configuration of a Laravel Queue. Once this queue is configured, Laravel does not check which Queue features it should process, instead Laravel will try to process everything it gets fed through a queue. The code does not differentiate between a queue for handling queueing closures, or a queue which (should) handle only a Newsletter class object.

# **References**

1. Laravel exploit explaination: <https://mogwailabs.de/en/blog/2022/08/exploiting-laravel-based-applications-with-leaked-app_keys-and-queues/>;
2. CVE 2018-15133: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15133>.