|  |  |
| --- | --- |
| Logo  Description automatically generated | Challenge Writeup  Jedi Archives  by Kouretas Panagiotis |

# Main Concept

Venture into 'The Jedi Archives', where secrets of the Force are guarded by layers of history, and the droids have been busy constructing the digital walls. Only those as wise as Jedi Masters can navigate the archives' hidden paths to enlightenment.

# Exploitation

First of all, we deploy the challenge from CTFLib.

Then we are redirected at <http://localhost:4242/index> which is the home page of the challenge as seen in Figure 7.1.

A screenshot of a video game

Description automatically generated

Figure 7 1 - Challenge's Home Page / Jedi Archives website.

With a quick look around at the page we can see that there is a “/secret” page available.

After we make sure that there is nothing helpful in the source code of the “/index” page, we head over to <http://localhost:4242/secret>.

As depicted in Figure 7.2, we receive a "403 Forbidden" error while attempting to access the "/secret" page. This implies that access to some challenge pages may be restricted.

A screenshot of a computer

Description automatically generated

Figure 7 2 – Error message while trying to access “/secret” page.

We can see that the challenge's concept is based on "Star Wars" from both the description and the main website. A closer examination of the challenge description reveals that the droids are in charge of creating the page, as seen by the statement, "and the droids have been busy constructing the digital walls."   
  
Essentially, “Droids” are robots. Therefore, it is plausible this web application to contain a "/robots.txt" page. The next figure shows the contents of the "/robots.txt" page that appears when we navigate to <http://localhost:4242/robots.txt>.

A screen shot of a computer

Description automatically generated

Figure 7 3 - Contents of "/robots.txt" page.

A robots.txt file tells search engine crawlers which URLs they can access on your site. This is used mainly to avoid overloading your site with requests; it is not a mechanism for keeping a web page out of Google.

Figure 7.3 illustrates that there is something unusual about this "/robots.txt" page. It appears to be a string that has been base64 encoded or something similar.

This string is indeed base64 encoded, as we can see by using a base64 decoder. Personally, for the decoding of the string I used “hackvertor” which is an encoding/decoding extension on Burp Suite Proxy.

Once the string is decoded, we obtain the following message: "Our Droids have created a /config-backup route for extra safety." Remember to disable the route after saving the backup file.

A screenshot of a computer

Description automatically generated

Figure 7 4 - Decoding base64 string using Burp's Hackvertor.

This indicates that a "/config-backup" file might be accessible. This page is visible when we go to <http://localhost:4242/config-backup> and the content of it is shown in the Figure 7.5.

A screenshot of a computer

Description automatically generated

Figure 7 5 - Contents of "config-backup" file.

Upon initial observation, it appears to be a proxy server configuration that blocks access to the "/secret" page by establishing an access control list. Additionally, the top of the backup file has a reference indicating the precise proxy server version being utilized, which is "HAProxy Version: 2.2.16."

So, let's hop on some research about "HAProxy Version: 2.2.16.” and its possible vulnerabilities.

A screenshot of a computer

Description automatically generated

Figure 7 6 - Searching about HAProxy CVEs.

After further investigation, we have determined that there are a few CVEs related to HAProxy. Based on the information provided, I have concluded that “CVE-2021-40346” is the most likely to be applicable in our situation. This is because it describes how to conduct an "HTTP REQUEST SMUGGLING” attack to bypass HAProxy ACLs and possibly other ACLs. Also, our HAProxy version (2.2.16) is between 2.0 and 2.5.

A screenshot of a computer

Description automatically generated

Figure 7 7 - CVE-2021-40346 vulnerability details.

So, let’s do some more extensive googling while searching for the specific code of the “CVE-2021-40346” as seen in Figure 7.8.

A screenshot of a computer

Description automatically generated

Figure 7 8 - Searching about CVE-2021-40346.

Perfect, I found this GitHub repository that looks like it is demonstrating how to bypass a “HAProxy” ACL by performing an HTTP REQUEST SMUGGLING that is caused by an integer overflow.

A screenshot of a computer error

Description automatically generated

Figure 7 9 - CVE-2021-40346 Proof of Concept.

The payload's execution is depicted in Figure 7.10.

A screenshot of a computer

Description automatically generated

Figure 7 10 - Executing the payload.

Now let's try to perform the HTTP REQUEST SMUGGLING in our case so we bypass the ACL that is blocking access to the “/secret” page.

To intercept the request, I used Burp Suite Proxy as demonstrated in Figure 7.11.

A screenshot of a computer

Description automatically generated

Figure 7 11 - Intercepting websites traffic using Burp Suite Proxy.

Afterwards with Burp’s Repeater, I modified the captured request and sent the new malformed.

A screenshot of a computer

Description automatically generated

Figure 7 12 - Modifying the captured request and sending the malformed that includes the payload using Burp’s Repeater.

The "/index" page appears in the response after sending the updated request for the first time, as seen in the figure.

However, when we send the modified request for the second time, the integer overflow occurs successfully, and we get the “/secret” page in the response as seen in Figure 7.13.

A screenshot of a computer

Description automatically generated

Figure 7 13 - Executing the payload successfully.

Next, we right-click on the response, and we choose “Show response in browser” to access the secret page through Burp’s browser.

A screenshot of a computer

Description automatically generated

Figure 7 14 - Choosing the "Show response in browser" from the response.

In doing so, we are able to access the "/secret" page and view its contents in Burp’s browser as depicted in Figure 7.15.

A screenshot of a computer

Description automatically generated

Figure 7 15 – Displaying "/secret" page contents on Burp’s browser.

After a brief glance at the contents of the page, we notice that we can contact the Jedi at the “/create\_message” page, therefore there may be another page to investigate.

Indeed, there is a new page at <http://localhost:4242/create_message> where we can send messages to the Jedi as seen in Figure 7.16.

A screenshot of a computer

Description automatically generated

Figure 7 16 - Contents of "/send\_message" page.

After sending a “test message” we notice that we are getting redirected to a “/message” page where the message we just sent to the Jedi is displayed. With a closer look on the URL, we observe that our displayed message has an “id” and a “token” parameter.

A screen shot of a computer

Description automatically generated

Figure 7 17 - “id” and “token” parameters.

Let's experiment a little bit so we can understand how these messages work. After sending a new message it seems like a new “id” is assigned in ascending order and a new “maybe” unique token. Pressing the "Create Another Message" button will allow us to return to the "/create\_message" page and send more messages.

If we try to change the “id” of the message from “id=2” to “id=1” so we can access the previous message we sent, we get the following error as depicted in the figure.

A screen shot of a computer

Description automatically generated

Figure 7 18 - Error message while trying to access previous messages by modisying the "id" parameter.

While searching at the "/create\_message" page source code I found a comment that may be useful for the continuation of the challenge. It indicates that we can find a configuration of how the messages work on the “/message\_conf” page as seen in Figure 7.19.

A screen shot of a computer code

Description automatically generated

Figure 7 19 - Useful comment in the page's source code.

At <http://localhost:4242/message_conf>, there is a configuration showcasing how the messaging system works. So, let's break it down and try to understand this configuration.   
I transferred the code into “PyCharm” which is a “Python IDE” to facilitate more detailed analysis and enhance visibility.

A screen shot of a computer

Description automatically generated

Figure 7 20 - Contents of "message\_conf" file in PyCharm IDE.

A screen shot of a computer

Description automatically generated

Figure 7 21 - Figure 7 20 - Contents of "message\_conf" file in PyCharm IDE.

The Python file provided is a Flask web application that allows users to create and view messages. It also includes a mechanism for generating and validating tokens to ensure that only users with the correct token can view a specific message.

The flag seems to be stored as the first message of a “messages” list.

The “generate\_token” function generates a secure token using HMAC with SHA-256, based on the secret key and a given ID. However, some key components appear to be vulnerable.

As seen in figure 7.20, the “SECRET\_KEY” variable is a string. Actually, it should be calling “str(uuid4())” to generate a “UUID” and we can also verify that from the “from uuid import uuid4” on imports that has grey color. This indicates that it's not used anywhere in the code.

So, we conclude that since the “SECRET\_KEY” is a static string, the token that is generated is only determined by the “id” of each message.

Because of this, I created my own Python script with a static SECRET\_KEY = "secret-{uuid4}" that allows me to generate the token for any message by simply selecting the appropriate "id."

We already know that the flag is the first message of a “messages” list. To regenerate this message’s token, I set the "message\_id=0".

A screenshot of a computer

Description automatically generated

Figure 7 22 – Python script for regenerating a message's token.

After running the “solve.py” script we get the following regenerated token: “6a33f9165c4f27ee039e44350f87ecbb1ddb2bddd1ef6a0cbef7684fa9cd8a7c”

A screenshot of a computer

Description automatically generated

Figure 7 23 - Retrieving the first message of the list / Successfully retrieved the hidden Flag.

We try to retrieve the first message and BOOM; we got the hidden FLAG:

CTFLIB{B3w4r3\_Th3\_Dr01d\_D3v3l0p3r\_Err0r$}