DNS Chess (FlareOn CTF)

Challenge Description:

Some suspicious network traffic led us to this unauthorized chess program running on an Ubuntu desktop. This appears to be the work of cyberspace computer hackers. You'll need to make the right moves to solve this one. Good luck!

Understanding the Challenge:

After Downloading the challenge files we have

```
root@kali:~/Desktop/4 - Dnschess# ls
capture.pcap  ChessAI.so  ChessUI  Message.txt
```

Message.txt content is the same as challenge description.

Looking at the other two files provides us below information

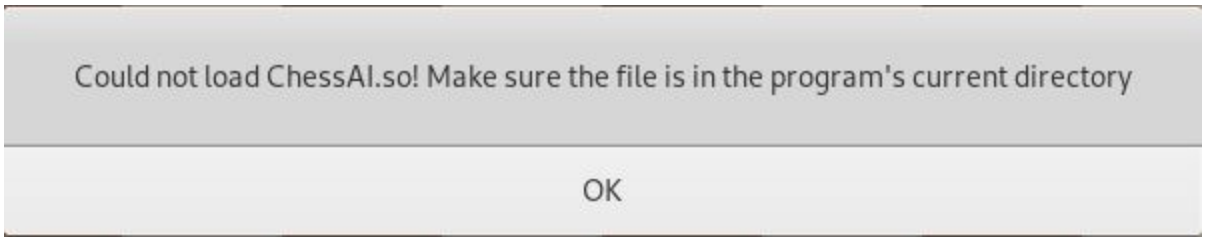
```
root@kali:~/Desktop/4 - Dnschess# file ChessUI

ChessUI: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=c30ec8b70e255aec7c93eb80321e4eab7bd52b3f, for GNU/Linux 3.2.0,
stripped

root@kali:~/Desktop/4 - Dnschess# file ChessAI.so

ChessAI.so: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically
linked, BuildID[sha1]=ed3bd3fae8d4a8e27e4565f31c9af58231319190, stripped
```

ChessUI is main binary having shared object file **ChessAI.so** which means whenever we run **ChessUI** binary it loads **ChessAI.so**

If that shared object file is not present in the current directory we get below error message.

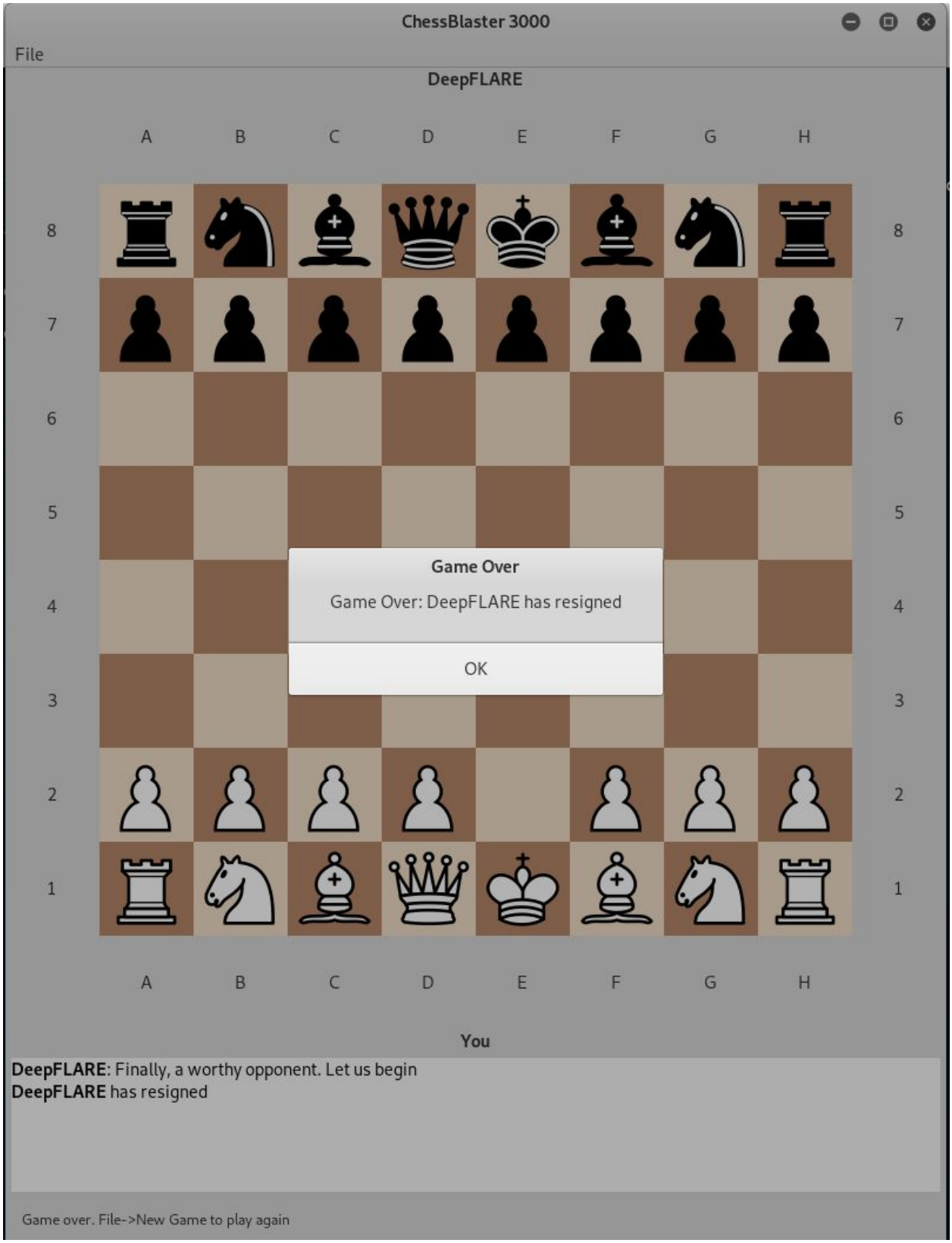
```
root@kali:~/Desktop/4 - Dnschess# ./ChessUI
dlopen: ./ChessAI.so: cannot open shared object file: No such file or directory
```

Also from UI it throws an exception and quits the game.



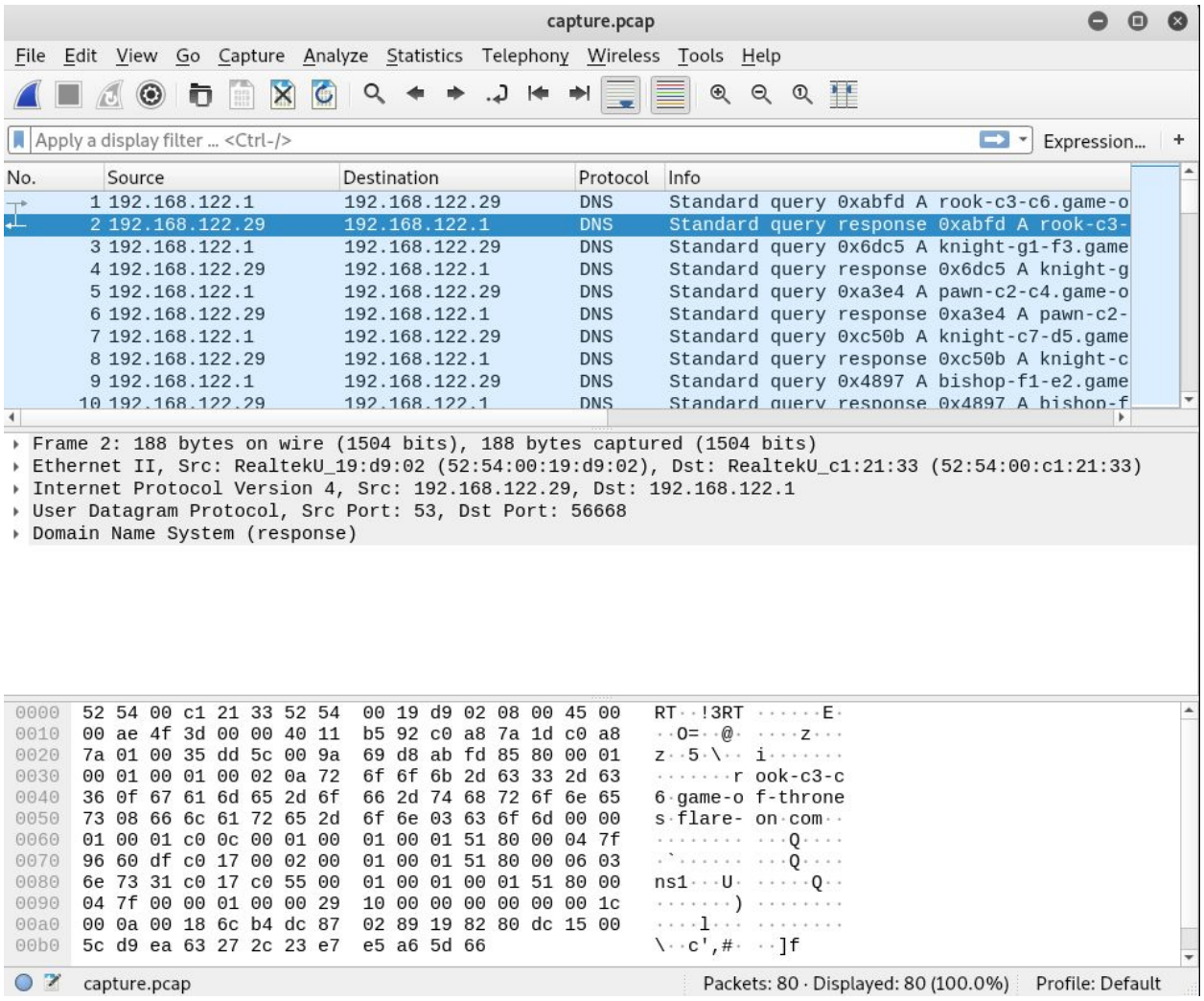
From this we can understand the Main Binary is importing certain functions from the Shared Library file (ChessAI.so)

Let’s open the main binary and start playing with DeepFLARE(Our Opponent)


ChessBlaster 3000
File
DeepFLARE
A B C D E F G H
8 7 6 5 4 3 2 1
Game Over
Game Over: DeepFLARE has resigned
OK
You
DeepFLARE: Finally, a worthy opponent. Let us begin
DeepFLARE has resigned
Game over. File->New Game to play again

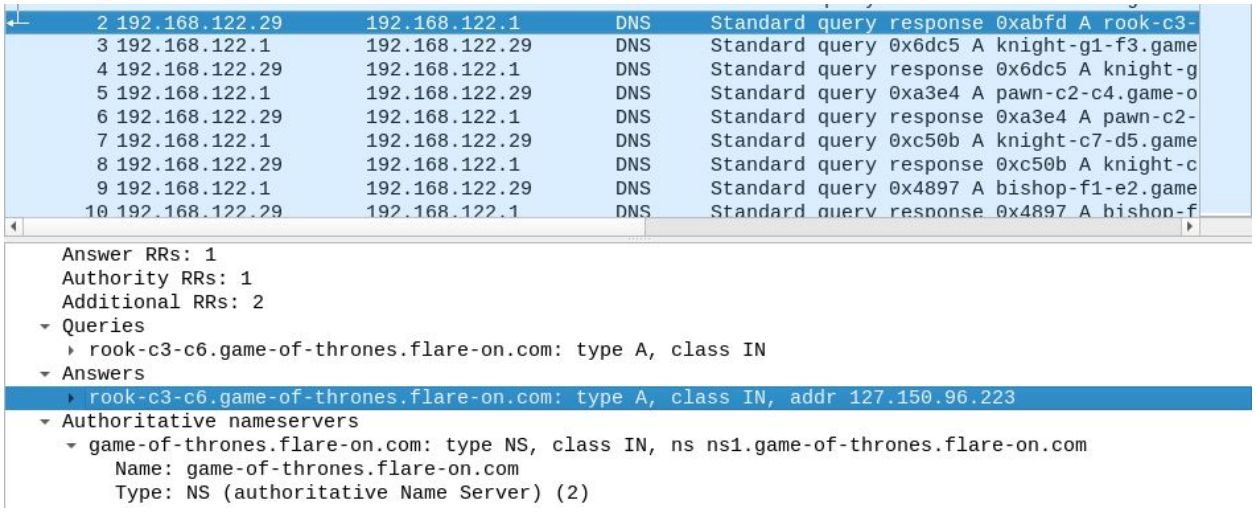
It seems it is checking for right move. How to know it then ?

Well. Looking at given pcap files reveals lot of useful information.



It is the traffic generated by a client **192.168.122.1** and it sends a DNS query for every move forming a pattern of domain name. If you look closely at the second packet it is a DNS reply from other party **192.168.122.29.**

The response packet also resolves the domain name into an ip



The IP address is **127.150.96.223**

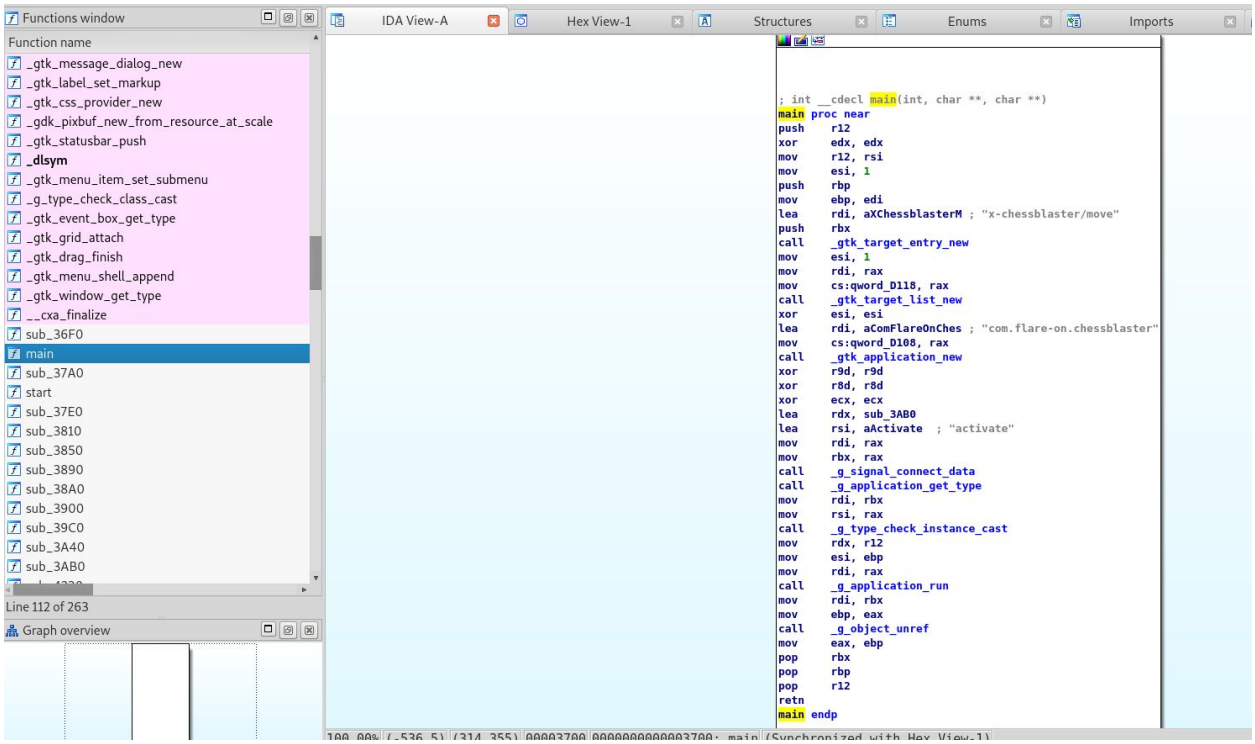
Total number of packets in the pcap file is 80 and if we ignore the DNS replies, count would be 40 packets. Identifying the right move out of these 40 moves is very tedious task (i feel almost impossible)

So somehow we have to identify the right moves from the main binary file and win the game. I feel then it prints the flag out.

Digging Deeper

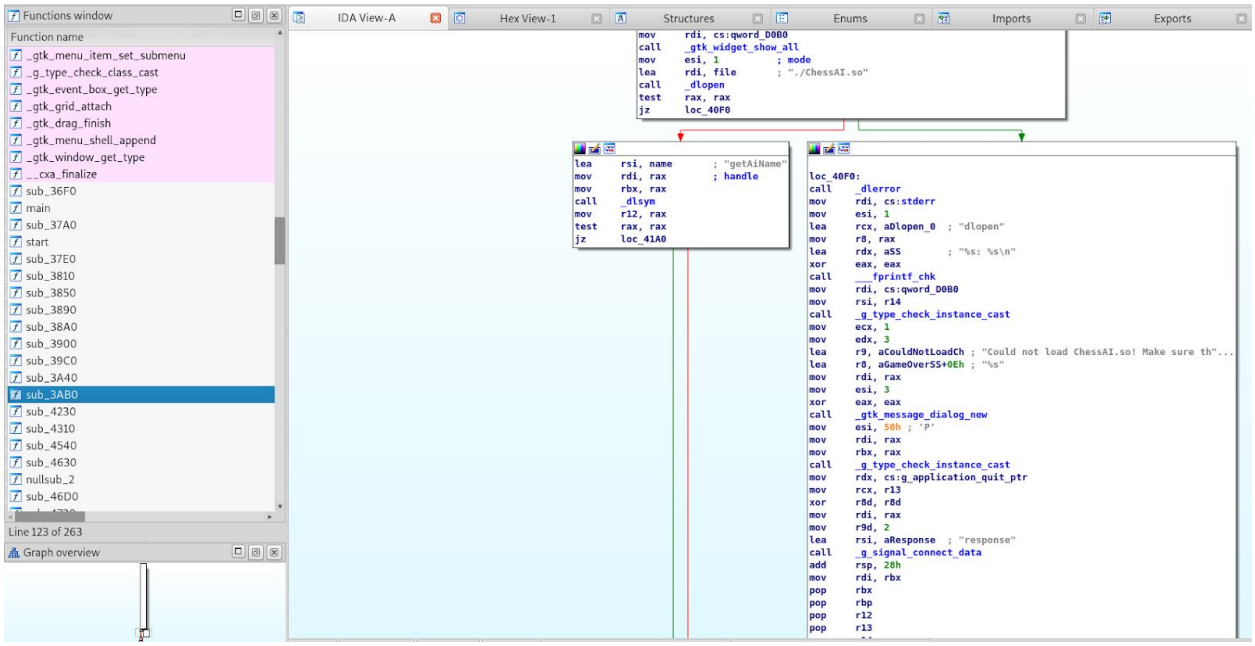
I’m going to use IDA (Interactive DisAssembler) to know the flow of the binary.

As usual for any binary challenge i first look at main function.



But it just nothing here. I went checking every sub_xxxx function to see if i can find useful information.

sub_3AB0 seems the one i was looking for



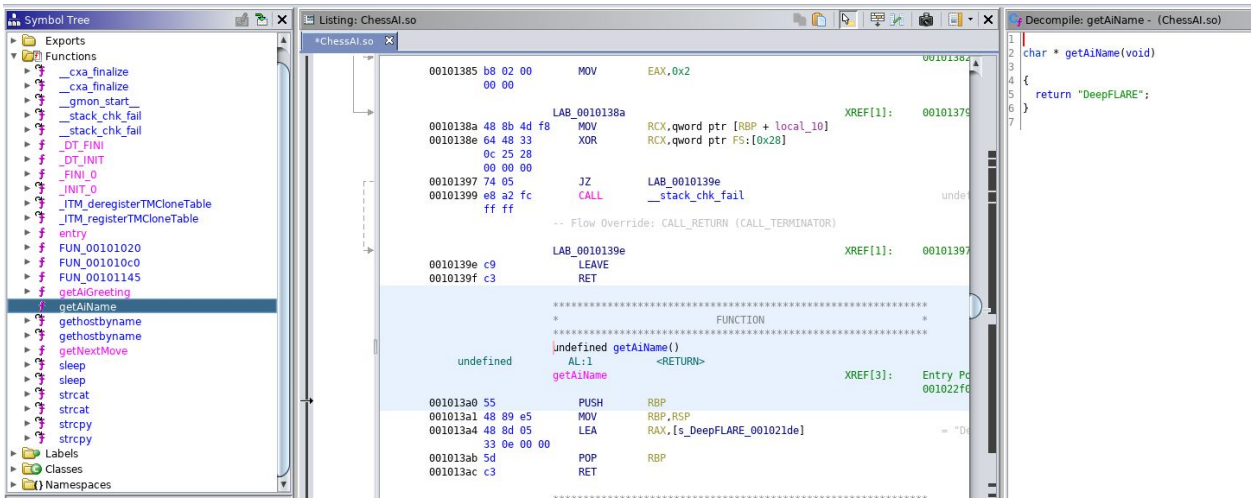
The reason is when we checked initial behavior of the binary it first loads **ChessAI.so** file so we could see that behavior here. If condition satisfied it jumps to below instruction set.

```
lea     rsi, name       ; "getAiName"
mov     rdi, rax        ; handle
mov     rbx, rax
call    _dlsym
mov     r12, rax
test    rax, rax
jz      loc_41A0
```

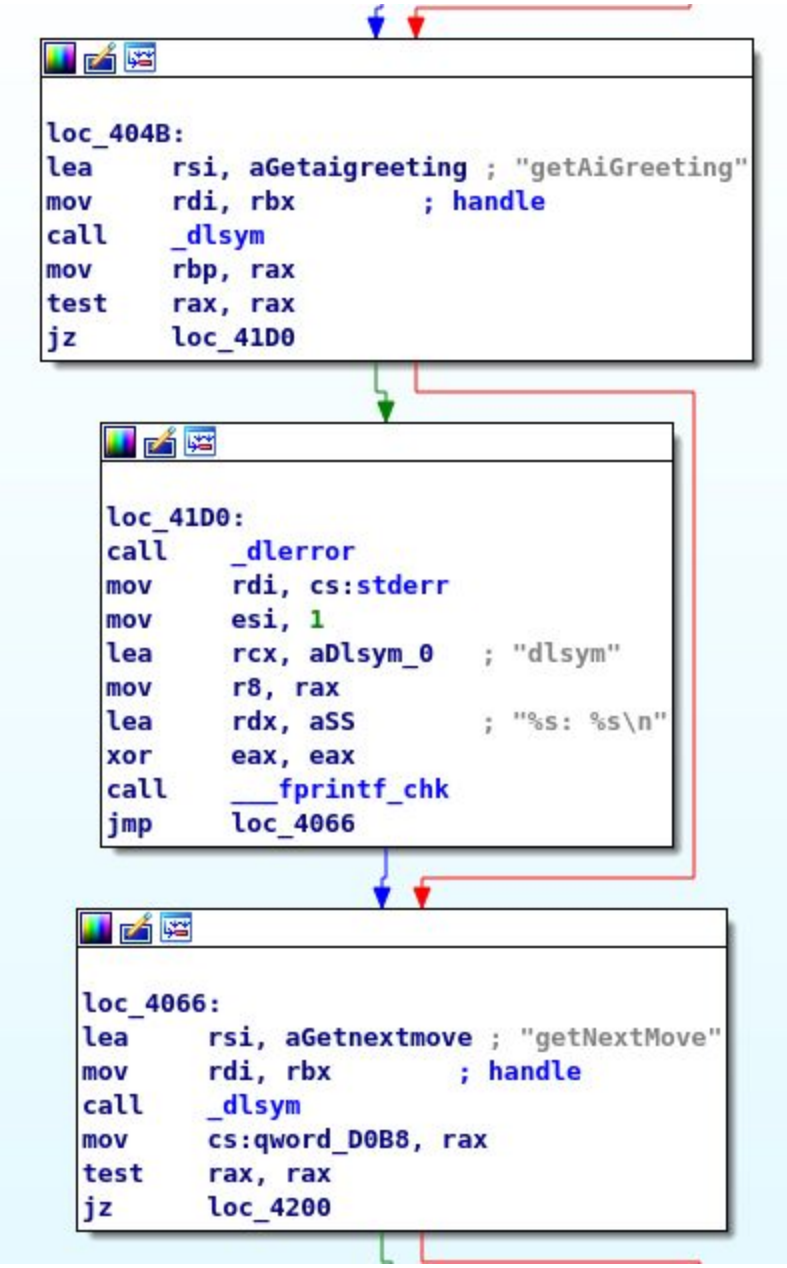
It just loads effective address of **getAiName** function from shared object. **dlsym()** simply returns the address of the function which loaded by **dlopen().**

I love having **Ghidra** open in parallel to understand instructions well as it decompiles the binary functions to source code.

From following image we could see **getAiName** is a function from shared object file (ChessAI.so) and just returns **DeepFLARE** string.

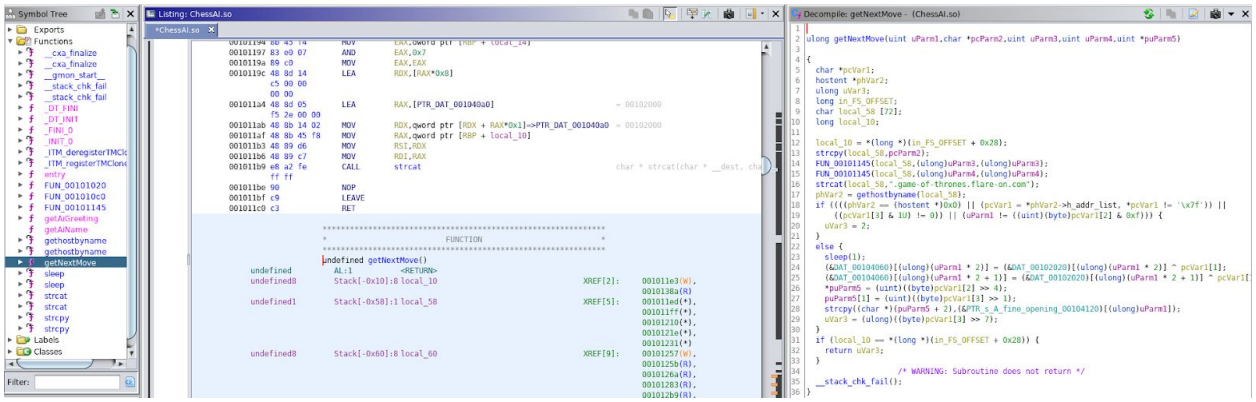


Awesome. So going down we could see other functions which displays getting loaded.



But **getNextMove** seems interesting to me as we are looking for the same.

I look at this function code from Ghidra to understand the logic behind getting right move.



So the code is as below.

```
ulong getNextMove(uint uParm1,char *pcParm2,uint uParm3,uint uParm4,uint
*puParm5)

{
  char *pcVar1;
  hostent *phVar2;
  ulong uVar3;
  long in_FS_OFFSET;
  char local_58 [72];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  strcpy(local_58,pcParm2);
  FUN_00101145(local_58,(ulong)uParm3,(ulong)uParm3);
  FUN_00101145(local_58,(ulong)uParm4,(ulong)uParm4);
  strcat(local_58,".game-of-thrones.flare-on.com");
  phVar2 = gethostbyname(local_58);
  if ((((phVar2 == (hostent *)0x0) || (pcVar1 = *phVar2->h_addr_list, *pcVar1 != '\x7f'))
||
     ((pcVar1[3] & 1U) != 0)) || (uParm1 != ((uint)(byte)pcVar1[2] & 0xf))) {
    uVar3 = 2;
  }
  else {
    sleep(1);
    (&DAT_00104060)[(ulong)(uParm1 * 2)] = (&DAT_00102020)[(ulong)(uParm1 * 2)]
^ pcVar1[1];
    (&DAT_00104060)[(ulong)(uParm1 * 2 + 1)] = (&DAT_00102020)[(ulong)(uParm1 *
2 + 1)] ^ pcVar1[1];
    *puParm5 = (uint)((byte)pcVar1[2] >> 4);
    puParm5[1] = (uint)((byte)pcVar1[3] >> 1);
```

```
    strcpy((char *)(puParm5 +
2),(&PTR_s_A_fine_opening_00104120)[(ulong)uParm1]);
    uVar3 = (ulong)((byte)pcVar1[3] >> 7);
  }
  if (local_10 == *(long *)(in_FS_OFFSET + 0x28)) {
    return uVar3;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

Looks complex ? ok let's tear it out a bit.

getNextMove function takes 5 arguments as input. Let's focus on below sample code

```
  strcpy(local_58,pcParm2);
  FUN_00101145(local_58,(ulong)uParm3,(ulong)uParm3);
  FUN_00101145(local_58,(ulong)uParm4,(ulong)uParm4);
  strcat(local_58,".game-of-thrones.flare-on.com");
  phVar2 = gethostbyname(local_58);
```

pcParm2 is being copied to **local_58** and it calls a function twice with **uParm3 & uParm4** then the output is being written to **local_58**

I believe **pcParm2** is holding name of object (pawn) we move in game and **uParm3** is from position (a2) and **uParm4** is to position (a3)

We can get clear picture by looking at **FUN_00101145**

```
void FUN_00101145(char *pcParm1,uint uParm2)
{
  strcat(pcParm1,"-");
  strcat(pcParm1,(&PTR_DAT_001040e0)[(ulong)(uParm2 >> 3 & 7)]);
  strcat(pcParm1,(&PTR_DAT_001040a0)[(ulong)(uParm2 & 7)]);
  return;
}
```

Nice as we expected it is forming the domain name. Then finally it appends remaining part of the domain name to **local_58**

So local_58 now looks like this **pawn-a2-a3.game-of-thrones.flare-on.com**
Then it tries to resolve this domain name calling **gethostbyname()** function.

From http://man7.org/linux/man-pages/man3/gethostbyname.3.html reference we could see that it returns a structure of type **hostent** for the given host name. It also specifies that it searches HOSTALIASES (/etc/hosts) first. Nice we can spoof dns replies with by just adding to hosts files.

Now we can look at the next code which performs a lot of comparisons to check the validation.

```
if ((((phVar2 == (hostent *)0x0) || (pcVar1 = *phVar2->h_addr_list, *pcVar1 != '\x7f')) ||
      ((pcVar1[3] & 1U) != 0)) || (uParm1 != ((uint)(byte)pcVar1[2] & 0xf))) {
    uVar3 = 2;
  }
else {
    sleep(1);
    (&DAT_00104060)[(ulong)(uParm1 * 2)] = (&DAT_00102020)[(ulong)(uParm1 * 2)]
^ pcVar1[1];
    (&DAT_00104060)[(ulong)(uParm1 * 2 + 1)] = (&DAT_00102020)[(ulong)(uParm1 *
2 + 1)] ^ pcVar1[1];
    *puParm5 = (uint)((byte)pcVar1[2] >> 4);
    puParm5[1] = (uint)((byte)pcVar1[3] >> 1);
    strcpy((char *)(puParm5 +
2),(&PTR_s_A_fine_opening_00104120)[(ulong)uParm1]);
    uVar3 = (ulong)((byte)pcVar1[3] >> 7);
  }
```

Before digging into the code i just checked what **DAT_00104060** and **DAT_00102020** are containing.

Interesting thing is first DAT chunk is pointing to flag formation

```
                    DAT_00104060                                    XREF[4]:     getNextMove:001012cf(*),
                                                                                 getNextMove:001012d6(W),
                                                                                 getNextMove:00101307(*),
                                                                                 00104190(*)
00104060 00             ??         00h

                    DAT_00104061                                    XREF[1]:     getNextMove:0010130e(W)
00104061 00             ??         00h
00104062 00             ??         00h
00104063 00             ??         00h
00104064 00             ??         00h
00104065 00             ??         00h
00104066 00             ??         00h
00104067 00             ??         00h
00104068 00             ??         00h
00104069 00             ??         00h
0010406a 00             ??         00h
0010406b 00             ??         00h
0010406c 00             ??         00h
0010406d 00             ??         00h
0010406e 00             ??         00h
0010406f 00             ??         00h
00104070 00             ??         00h
00104071 00             ??         00h
00104072 00             ??         00h
00104073 00             ??         00h
00104074 00             ??         00h
00104075 00             ??         00h
00104076 00             ??         00h
00104077 00             ??         00h
00104078 00             ??         00h
00104079 00             ??         00h
0010407a 00             ??         00h
0010407b 00             ??         00h
0010407c 00             ??         00h
0010407d 00             ??         00h
0010407e 40 66 6c       ds         "@flare-on.com"
         61 72 65
         2d 6f 6e ...
```

So we could see that flag is of 30 character length. So if we can make that if loop false eventually we can get into the flag part. Also i could understand it is decrypting two characters for every move so there must be 15 valid moves in the pcap that we have to identify.

Let's break the loop.

```
if ((((phVar2 == (hostent *)0x0) || (pcVar1 = *phVar2->h_addr_list, *pcVar1 != '\x7f')) ||
     ((pcVar1[3] & 1U) != 0)) || (uParm1 != ((uint)(byte)pcVar1[2] & 0xf))) {
   uVar3 = 2;
 }
```

Every condition has Logical OR which means if anyone of the above condition is true then the entire loop is true and we quit the game instead of moving ahead.

1. phVar2 == (hostent *)0x0) (Domain shouldn't be resolved)
2. (pcVar1 = *phVar2->h_addr_list, *pcVar1 != '\x7f' (First octet of first ip must be not equal to 127)
3. pcVar1[3] & 1U) != 0 (If we perform AND operation of last octet of ip with 1 we shouldn't get 0)
4. uParm1 != ((uint)(byte)pcVar1[2] & 0xf (AND of ip address third octet and 0xf mustn't be equal to uParm1 value)

We should make above conditions false. As we know we are going to use /etc/hosts to resolve the domains properly so the first condition goes as false. Also the most of ip's resolved in pcap file are beginning with 127 which also make second condition false.

For third condition we need list of ip's. I'll use scapy to parse the packets easily and grep for ip part.

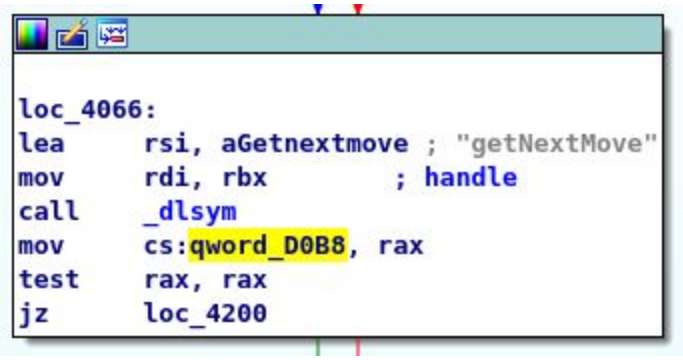
```
from scapy.all import *

packets = rdpcap('capture.pcap')

for packet in packets:
        if packet.haslayer(DNSRR):
                if isinstance(packet.an, DNSRR):
                        print(packet.an.rdata)
```

This gives me all ip addresses from dns replies in pcap. We can try to perform 3rd comparison operation using python.

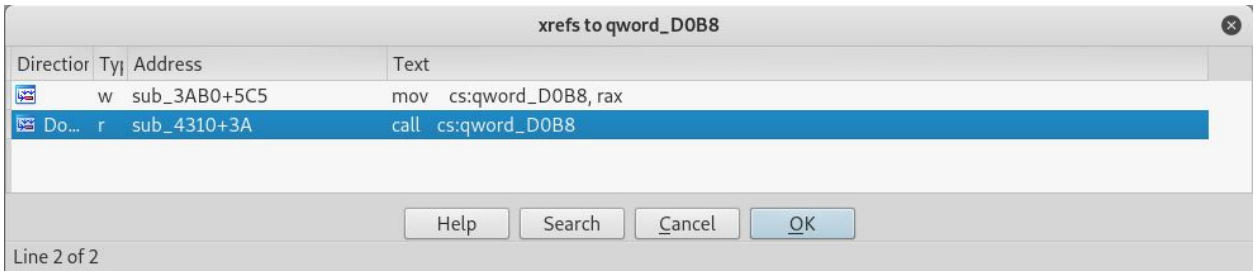
```
f = open('ips','r')
content = f.readlines()

for ip in content:
        a,b,c,d = ip.split('.')
        out = int(d) & 1
        if out == 0:
                print ip.strip()
```

This gives me 15 ip addresses as output which is what we have figured out initially that each move decrypts 2 characters at a time.

To check 4th condition let's identify what is **uParm1** value initially.

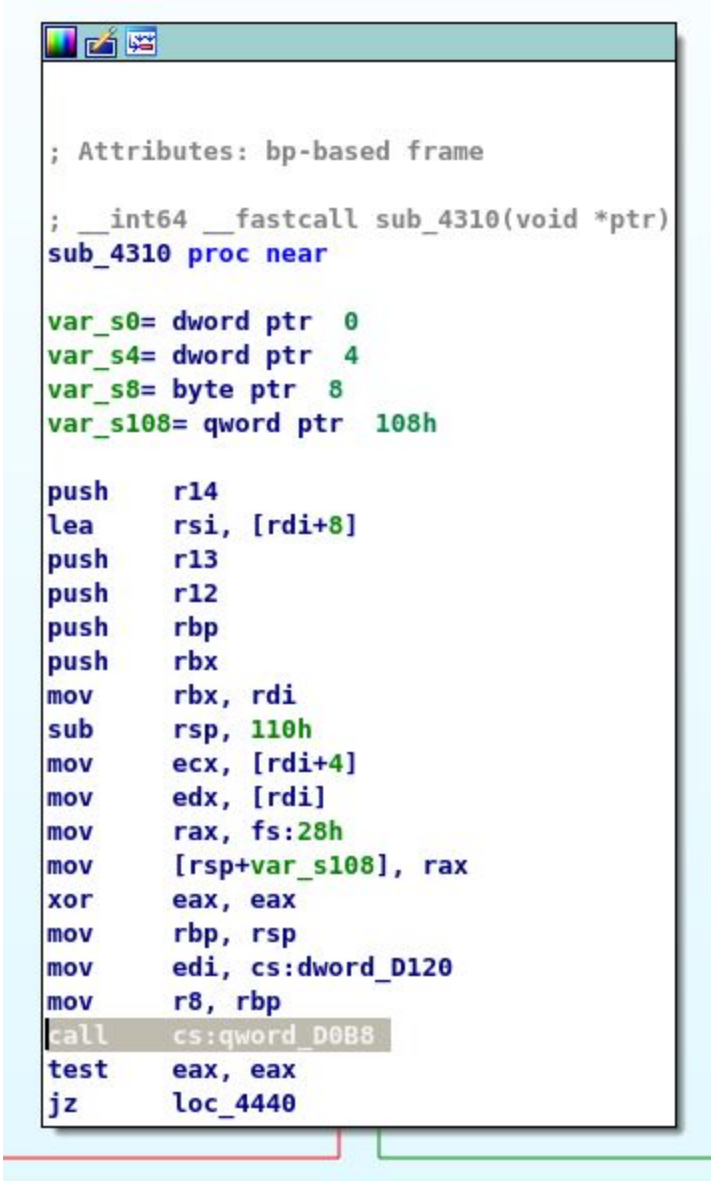


If we well aware about x64 calling conventions we can identify that **getNextMove** functions return address gets loaded into **rax** register and it has two xrefs



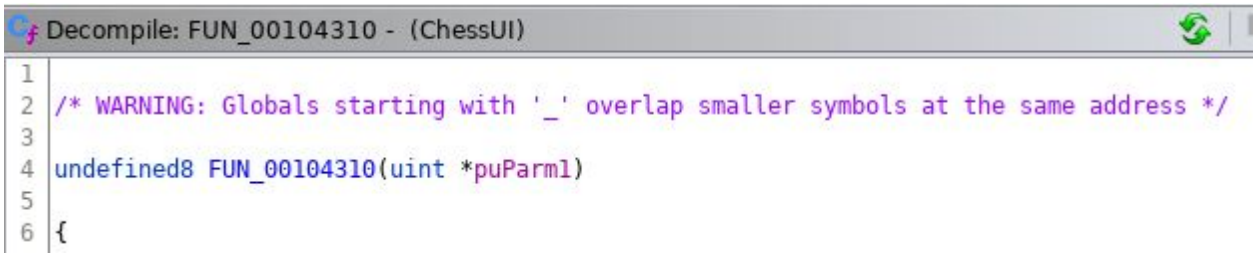
We can follow the call **cs:qword_D0B8**

It does jump to **sub_4310**



```
; Attributes: bp-based frame

; __int64 __fastcall sub_4310(void *ptr)
sub_4310 proc near

var_s0= dword ptr  0
var_s4= dword ptr  4
var_s8= byte ptr  8
var_s108= qword ptr  108h

push    r14
lea     rsi, [rdi+8]
push    r13
push    r12
push    rbp
push    rbx
mov     rbx, rdi
sub     rsp, 110h
mov     ecx, [rdi+4]
mov     edx, [rdi]
mov     rax, fs:28h
mov     [rsp+var_s108], rax
xor     eax, eax
mov     rbp, rsp
mov     edi, cs:dword_D120
mov     r8, rbp
call    cs:qword_D0B8
test    eax, eax
jz      loc_4440
```

If we check this function input params in ghidra it is same we are looking for **uParm1**



Decompile: FUN_00104310 - (ChessUI)

```
1
2  /* WARNING: Globals starting with '_' overlap smaller symbols at the same address */
3
4  undefined8 FUN_00104310(uint *puParm1)
5
6  {
```

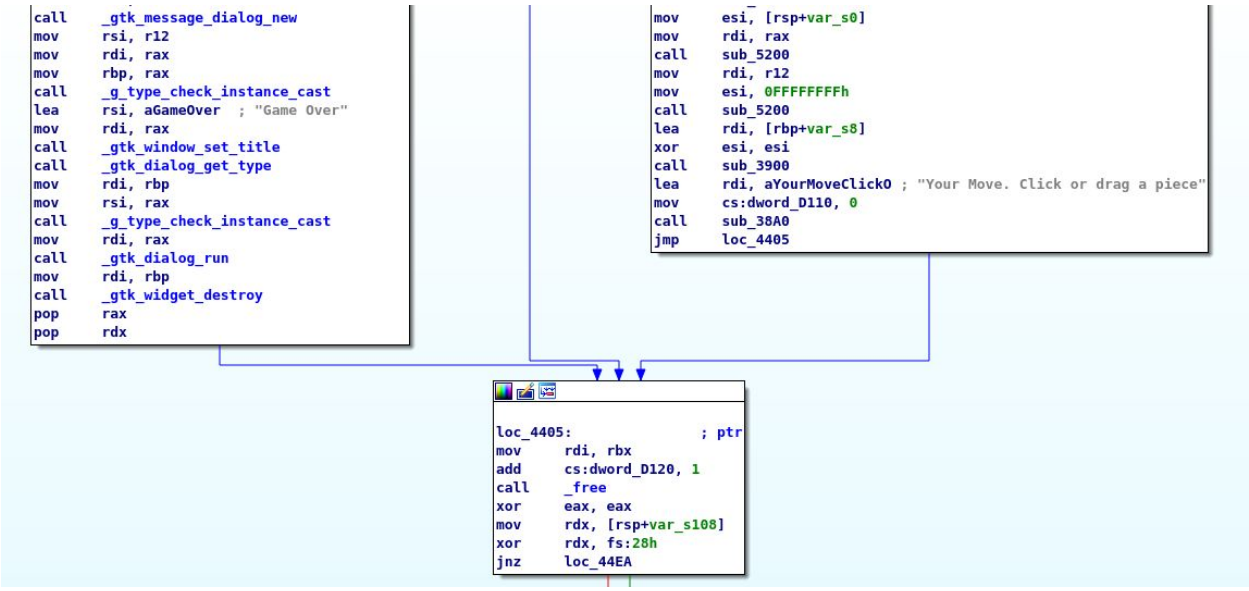
Ok now we can look at RDI register assignment to find what is its initial assigned value.

```
mov     edi, cs:dword_D120
```

I've found this instruction right before calling the actual function so again we have to check xrefs to **cs:dword_D120**

```
mov     cs:dword_D120, 0
```

Its initial value is 0. And if we look after each move this value gets incremented by 1.



Do you see that **add cs:dword_D120, 1** ? Cool. Now I can identify the right order of the moves with the above check.

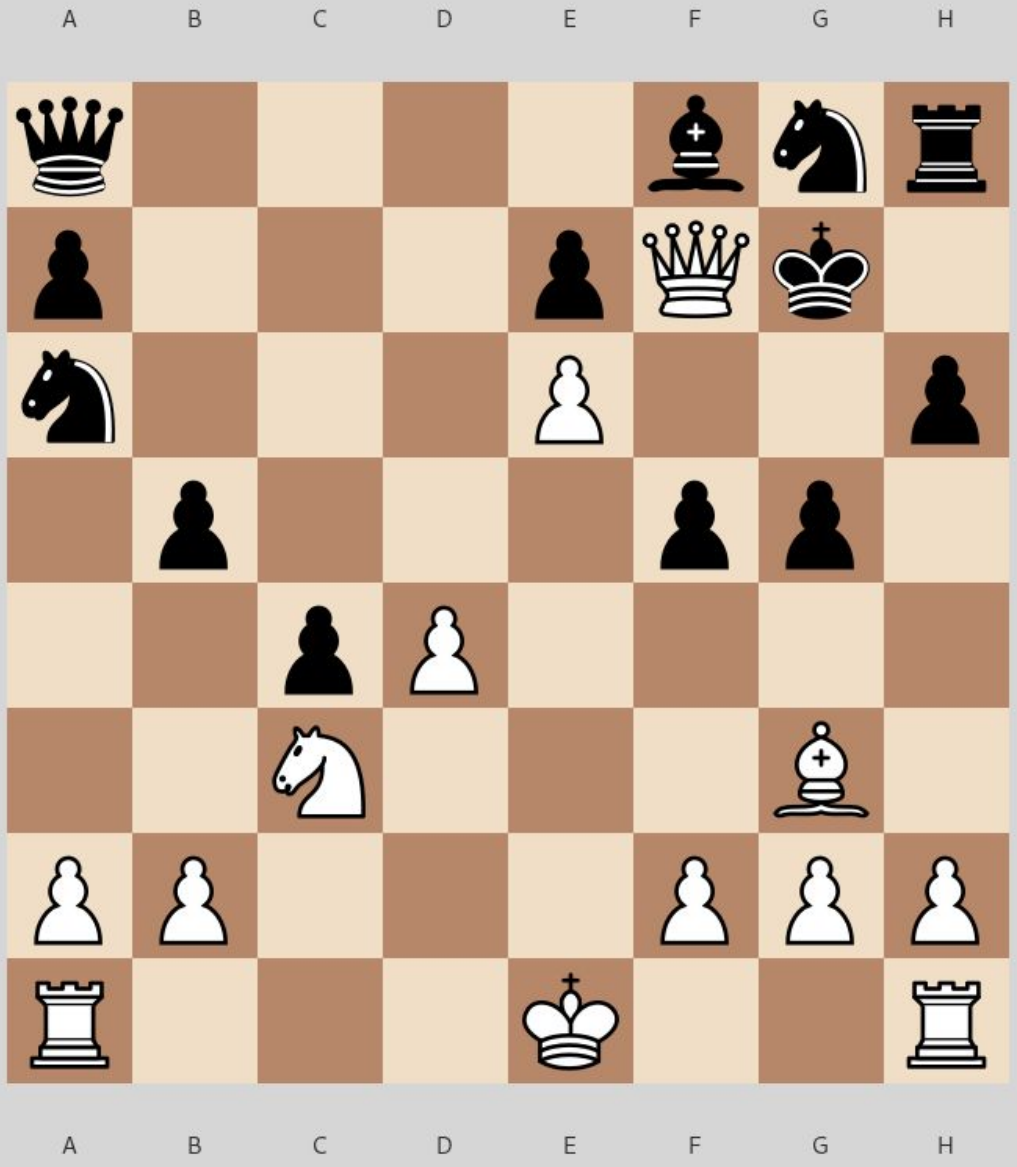
```
f = open('rightmoves','r')
content = f.readlines()

moves=[]
for move in content:
        ip,move = move.split(':')
        a,b,c,d = ip.split('.')
        out = int(c) & 0xf
        moves.append(str(out)+':'+ip+':'+move.strip())

moves = sorted(moves)
for a in moves:
        print a
```

This produces a list of valid moves in proper order.

```
0:127.53.176.56:pawn-d2-d4.game-of-thrones.flare-on.com
1:127.215.177.38:pawn-c2-c4.game-of-thrones.flare-on.com
2:127.159.162.42:knight-b1-c3.game-of-thrones.flare-on.com
3:127.182.147.24:pawn-e2-e4.game-of-thrones.flare-on.com
4:127.252.212.90:knight-g1-f3.game-of-thrones.flare-on.com
5:127.217.37.102:bishop-c1-f4.game-of-thrones.flare-on.com
6:127.89.38.84:bishop-f1-e2.game-of-thrones.flare-on.com
7:127.230.231.104:bishop-e2-f3.game-of-thrones.flare-on.com
8:127.108.24.10:bishop-f4-g3.game-of-thrones.flare-on.com
9:127.34.217.88:pawn-e4-e5.game-of-thrones.flare-on.com
10:127.25.74.92:bishop-f3-c6.game-of-thrones.flare-on.com
11:127.49.59.14:bishop-c6-a8.game-of-thrones.flare-on.com
12:127.200.76.108:pawn-e5-e6.game-of-thrones.flare-on.com
13:127.99.253.122:queen-d1-h5.game-of-thrones.flare-on.com
14:127.141.14.174:queen-h5-f7.game-of-thrones.flare-on.com
```

We can add them to hosts list and by playing the game according to order we can get the flag.

DeepFLARE



You

DeepFLARE: I have gained a tempo
DeepFLARE: You have weak squares around your king
DeepFLARE: With my next move I will seize control
DeepFLARE: An exchange of pieces is in order
DeepFLARE: A bold move
DeepFLARE: LooksLikeYouLockedUpTheLookupZ@flare-on.com

Game over. You win!!!!1