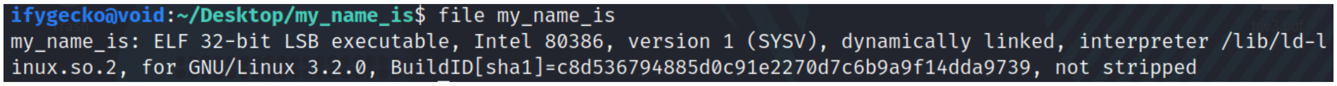
Challenge: Hi! My name is (what?)

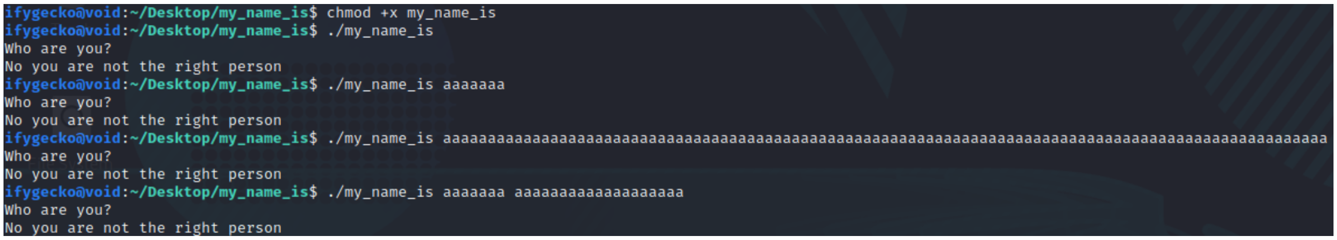
Solver(s): ifyGecko

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



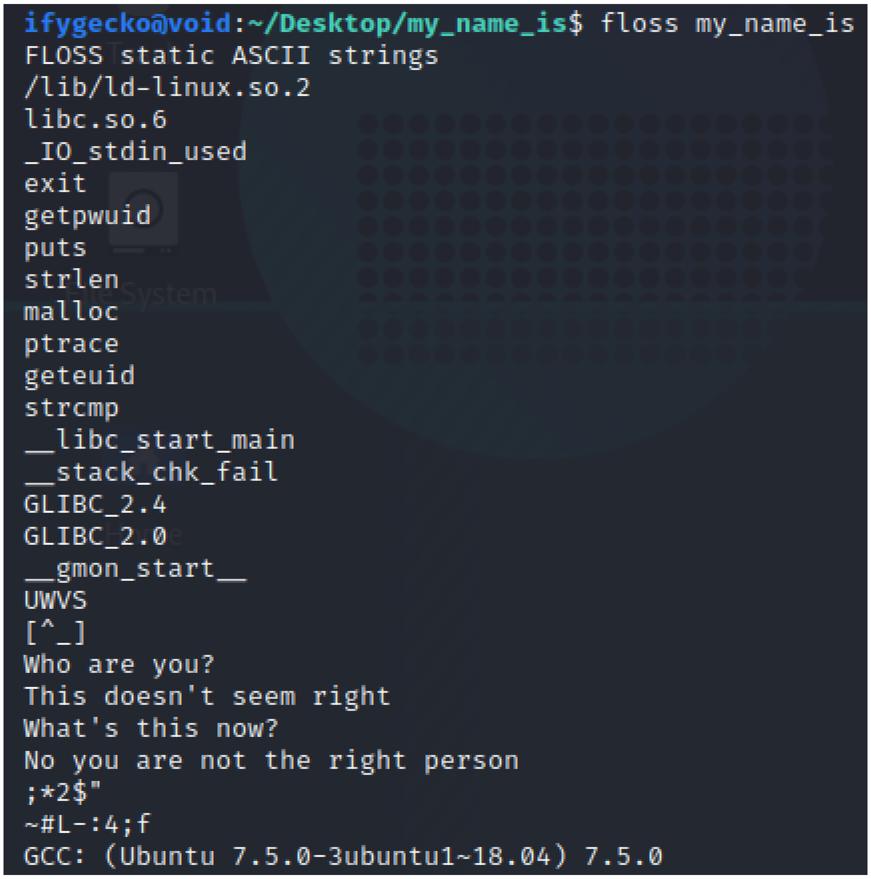
```
ifygecko@void:~/Desktop/my_name_is$ file my_name_is
my_name_is: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-l
inux.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=c8d536794885d0c91e2270d7c6b9a9f14dda9739, not stripped
```

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



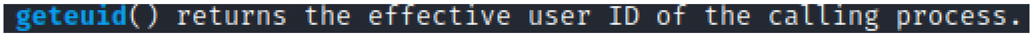
```
ifygecko@void:~/Desktop/my_name_is$ chmod +x my_name_is
ifygecko@void:~/Desktop/my_name_is$ ./my_name_is
Who are you?
No you are not the right person
ifygecko@void:~/Desktop/my_name_is$ ./my_name_is aaaaaaa
Who are you?
No you are not the right person
ifygecko@void:~/Desktop/my_name_is$ ./my_name_is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

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

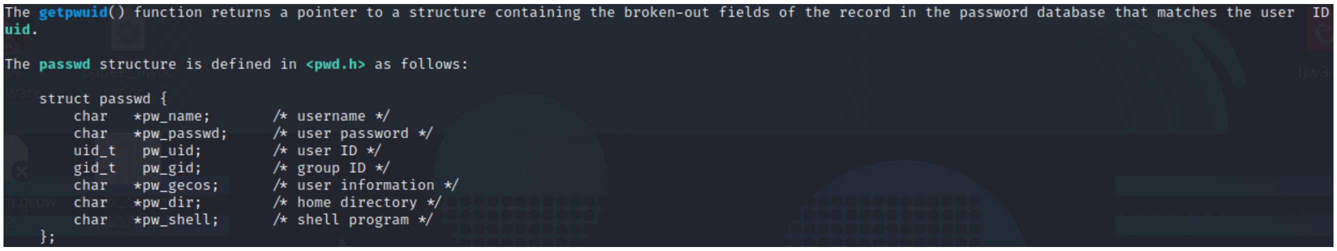


```
ifygecko@void:~/Desktop/my_name_is$ floss my_name_is
FLOSS static ASCII strings
/lib/ld-linux.so.2
libc.so.6
_IO_stdin_used
exit
getpwuid
puts
strlen
malloc
ptrace
geteuid
strcmp
__libc_start_main
__stack_chk_fail
GLIBC_2.4
GLIBC_2.0
__gmon_start__
UWVS
[^_]
Who are you?
This doesn't seem right
What's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

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



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



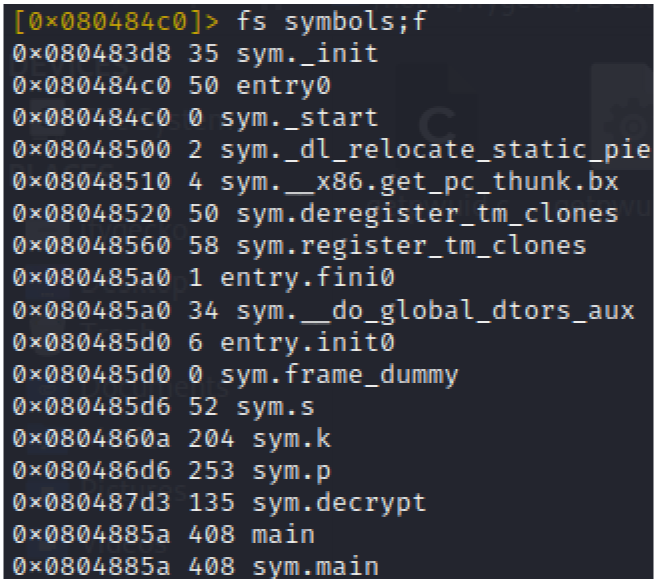
```
The getpwuid() function returns a pointer to a structure containing the broken-out fields of the record in the password database that matches the user  ID
uid.

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

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

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

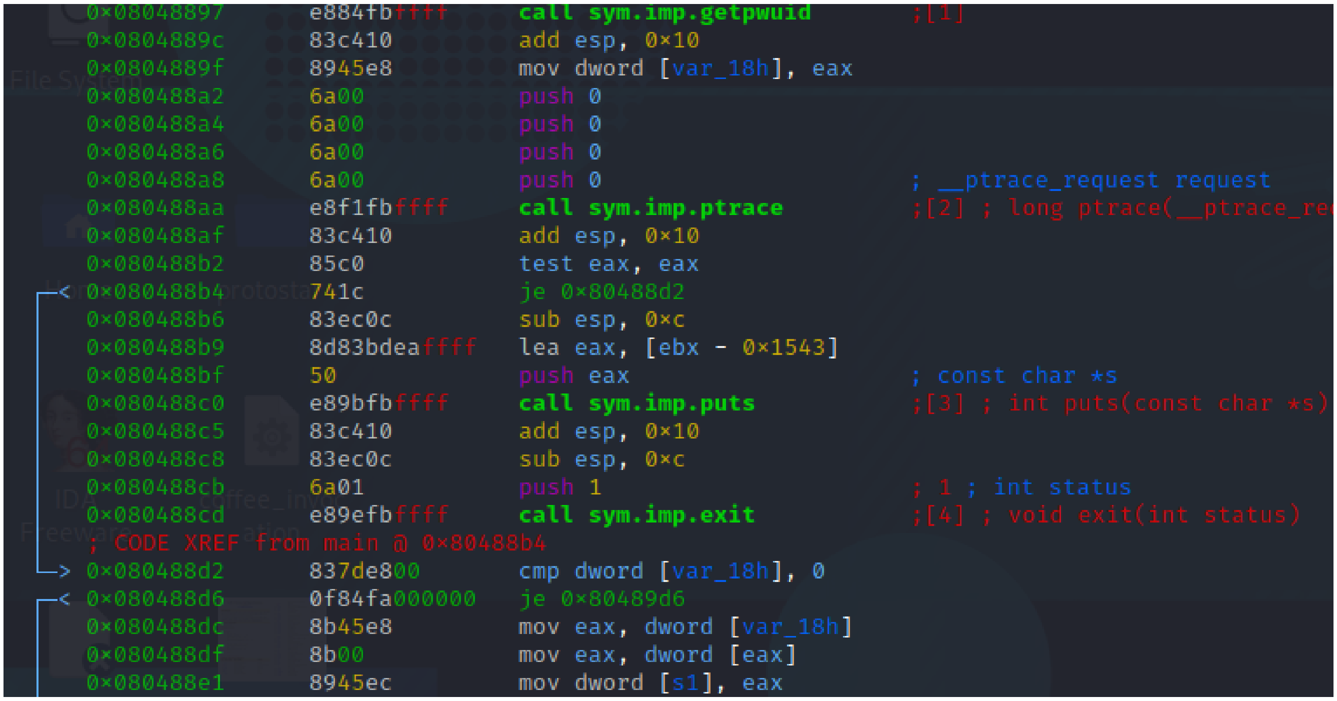
Knowing that this binary was not stripped I first wanted to see what symbols were available.



```
[0x080484c0]> fs symbols;f
0x080483d8 35 sym._init
0x080484c0 50 entry0
0x080484c0 0 sym._start
0x08048500 2 sym._dl_relocate_static_pie
0x08048510 4 sym.__x86.get_pc_thunk.bx
0x08048520 50 sym.deregister_tm_clones
0x08048560 58 sym.register_tm_clones
0x080485a0 1 entry.fini0
0x080485a0 34 sym.__do_global_dtors_aux
0x080485d0 6 entry.init0
0x080485d0 0 sym.frame_dummy
0x080485d6 52 sym.s
0x0804860a 204 sym.k
0x080486d6 253 sym.p
0x080487d3 135 sym.decrypt
0x0804885a 408 main
0x0804885a 408 sym.main
```

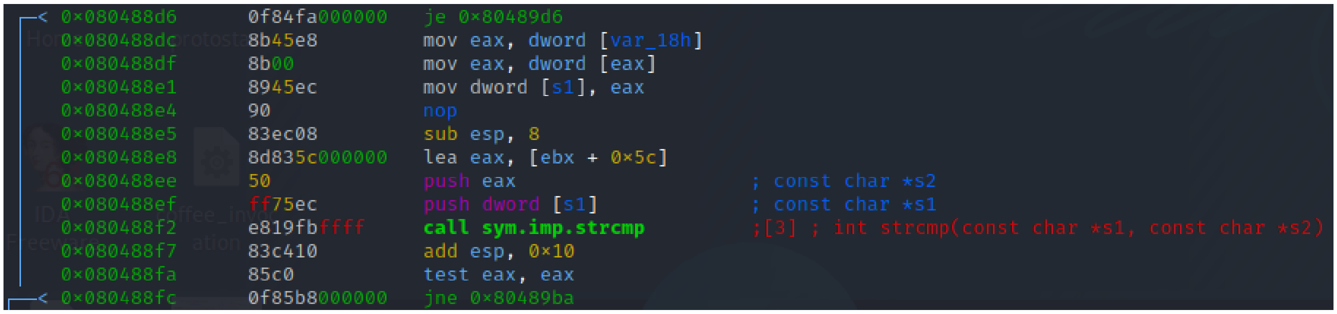
In the top of the listing of symbols I saw a decrypt symbol so I figured the binary would decrypt a string which would end up being the flag. However, knowing the potential use case of ‘getpwuid’ I thought it could be using my username/password as either a check to authenticate the decryption or maybe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

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



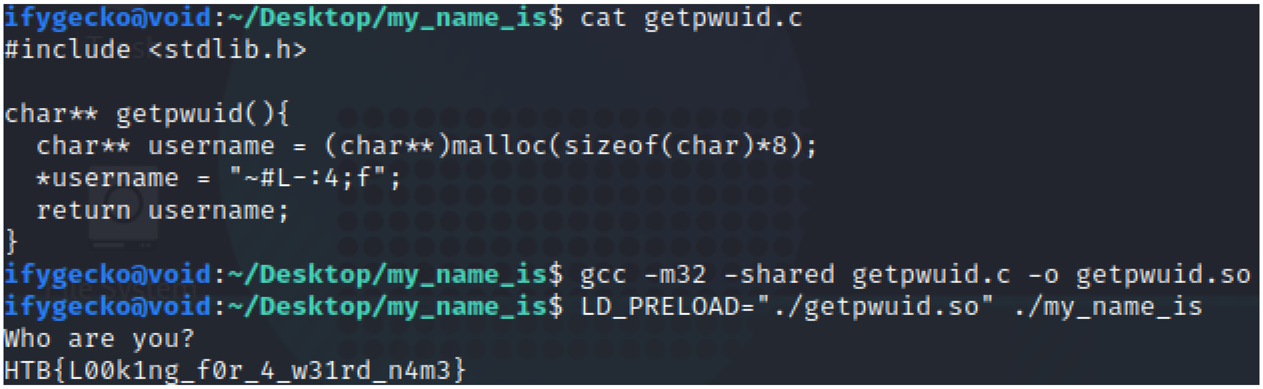
```
      0x08048897      e884fbffff     call sym.imp.getpwuid       ;[1]
      0x0804889c      83c410         add esp, 0x10
      0x0804889f      8945e8         mov dword [var_18h], eax
      0x080488a2      6a00           push 0
      0x080488a4      6a00           push 0
      0x080488a6      6a00           push 0
      0x080488a8      6a00           push 0                      ; __ptrace_request request
      0x080488aa      e8f1fbffff     call sym.imp.ptrace         ;[2] ; long ptrace(__ptrace_re
      0x080488af      83c410         add esp, 0x10
      0x080488b2      85c0           test eax, eax
  ┌─< 0x080488b4      741c           je 0x80488d2
  │   0x080488b6      83ec0c         sub esp, 0xc
  │   0x080488b9      8d83bdeaffff   lea eax, [ebx - 0x1543]
  │   0x080488bf      50             push eax                    ; const char *s
  │   0x080488c0      e89bfbffff     call sym.imp.puts           ;[3] ; int puts(const char *s)
  │   0x080488c5      83c410         add esp, 0x10
  │   0x080488c8      83ec0c         sub esp, 0xc
  │   0x080488cb      6a01           push 1                      ; 1 ; int status
  │   0x080488cd      e89efbffff     call sym.imp.exit           ;[4] ; void exit(int status)
  │   ; CODE XREF from main @ 0x80488b4
  └─> 0x080488d2      837de800       cmp dword [var_18h], 0
  ┌─< 0x080488d6      0f84fa000000   je 0x80489d6
  │   0x080488dc      8b45e8         mov eax, dword [var_18h]
  │   0x080488df      8b00           mov eax, dword [eax]
  │   0x080488e1      8945ec         mov dword [s1], eax
```

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



```
 ┌─< 0x080488d6      0f84fa000000   je 0x80489d6
 │   0x080488dc      8b45e8         mov eax, dword [var_18h]
 │   0x080488df      8b00           mov eax, dword [eax]
 │   0x080488e1      8945ec         mov dword [s1], eax
 │   0x080488e4      90             nop
 │   0x080488e5      83ec08         sub esp, 8
 │   0x080488e8      8d835c000000   lea eax, [ebx + 0x5c]
 │   0x080488ee      50             push eax                    ; const char *s2
 │   0x080488ef      ff75ec         push dword [s1]             ; const char *s1
 │   0x080488f2      e819fbffff     call sym.imp.strcmp         ;[3] ; int strcmp(const char *s1, const char *s2)
 │   0x080488f7      83c410         add esp, 0x10
 │   0x080488fa      85c0           test eax, eax
┌──< 0x080488fc      0f85b8000000   jne 0x80489ba
```

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



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

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

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