

impossible_password

analysis

We are instructed to retrieve the flag?

After downloading the file we first use the file command to see what type of file we are messing with

```
~/Desktop/HTB/file
```

```
impossible_password.bin
```

```
impossible_password.bin: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),  
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,  
BuildID[sha1]=ba116ba1912a8c3779ddeb579404e2fdf34b1568, stripped
```

We know that its a 64-bit Linux ELF and it's stripped.

That means that it does not contain any debug information.

Next we use string to list the contents of the file.

we do this becuase often flags are hidden as strings in a .bin but are hard to interpret.

We also do this to see if we can find any other clues that will lead us to the flag.

```
libc.so.6
```

```
exit
```

```
srand
```

```
__isoc99_scanf
```

```
time
```

```
putchar
```

```
printf
```

```
malloc
```

```
strcmp
```

```
__libc_start_main
```

```
__gmon_start__
```

```
GLIBC_2.7
```

GLIBC_2.2.5

UH-x

UH-x

=1

[]A\A]A^A_

SuperSeKretKey

%20s

[%s]

;*3\$"

GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)

.shstrtab

.interp

.note.ABI-tag

.note.gnu.build-id

.gnu.hash

.dynsym

.dynstr

.gnu.version

.gnu.version_r

.rela.dyn

.rela.plt

.init

.text

.fini

.rodata

.eh_frame_hdr

.eh_frame

.init_array

.fini_array

.jcr

.dynamic

.got

.got.plt

.data

.bss

.comment

and it appears that we found a clue **SuperSeKretKey**
this is obviously not the flag but it's a start.

running the file

Next we run the binary file to see what happens

```
./
impossible_password.bin
* blah
[blah]
```

We see are prompted to enter a password but luckily we have the SuperSeKretKey.

```
./
impossible_password.bin
* SuperSeKretKey
[SuperSeKretKey]
**
```

It appears that after we enter the SuperSeKretKey it prompts us again for another password.

So meaning we are dealing with multilevel password protection.

Now that we found the first password we need to find the second.

It accepts that string and print that in a new line closed by brackets, also it printed a new line with ** and asking for input.

Inputting anything doesn't output anything

Using **ltrace**, a diagnostic tool, debugging, instructional userspace, utility for linux

```

__libc_start_main(0x40085d, 1, 0x7fffee85f6218, 0x4009e0 <unfinished ...>
printf("* ") = 2
__isoc99_scanf(0x400a82, 0x7fffee85f6100, 0, 0* SuperSeKretKey
) = 1
printf("[%s]\n", "SuperSeKretKey"[SuperSeKretKey]
) = 17
strcmp("SuperSeKretKey", "SuperSeKretKey") = 0
printf("*** ") = 3
__isoc99_scanf(0x400a82, 0x7fffee85f6100, 0, 0** givemeflag
) = 1
time(0) = 1605491924
srand(0x7b62e7a1, 10, 0x79e54090, 0) = 0
malloc(21) = 0xf0bac0
rand(0xf0bac0, 21, 33, 0xf0bad0) = 0x20a1dd80
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bac0, 94) = 0x2cf0ac66
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bac1, 94) = 0x745157a5
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bac2, 94) = 0x7f76c58d
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bac3, 94) = 0x60686316
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bac4, 94) = 0x2accd466
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bac5, 94) = 0x20759beb
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bac6, 94) = 0x6a707127
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bac7, 94) = 0x44caa3e6
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bac8, 94) = 0x48316a03
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bac9, 94) = 0x77c381b4
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0baca, 94) = 0x65bc3115
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bacb, 94) = 0x47a3304e
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bacc, 94) = 0x3b97af1e
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bacd, 94) = 0x3a497bd5
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bace, 94) = 0x3dc27782
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bacf, 94) = 0x10b4e92c
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bad0, 94) = 0x42651256
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bad1, 94) = 0x5aba2fb
rand(0x7ffb46c31740, 0x7fffee85f6064, 0xf0bad2, 94) = 0x706e6d9e
strcmp("givemeflag", "gG`>mGdrO\\+n;)Lwoad}") = 34
+++ exited (status 34) +++

```

We see that `__isoc99_scanf` is being used to read input.

Since After enter the SuperSeKretKey at the first prompt we see it again on the second

prompt. `printf("[%s]\n", "SuperSeKretKey"[SuperSeKretKey]`

We also see it comparing SuperSeKretKey for the right password.

After we enter the 2nd password we see calls `time()` and `rand()` to generate random data.

And we see our second input being compared.

`strcmp("givemeflag", "gG`>mGdrO\\+n;)Lwoad}")`

Here our input is being compared to another string "givemeflag" and "gG`>mGdrO\\+n;)Lwoad}"

If this is what our input is being compared to the this must be password right?

Sadly no, this is the wrong password after rerunning ltrace we get error.

After analyzing the output once more we get the following:

```
strcmp(">mGdrO\\\\\\+n;)Lwoad}", "8iS<wr(-;Ib}{O^S]-TW")'
```

This means that our password is dynamic but it changes on runtime.

Well this explains the time() and rand() calls we saw.

To summarize the password is generated on runtime using the timestamp and random data.

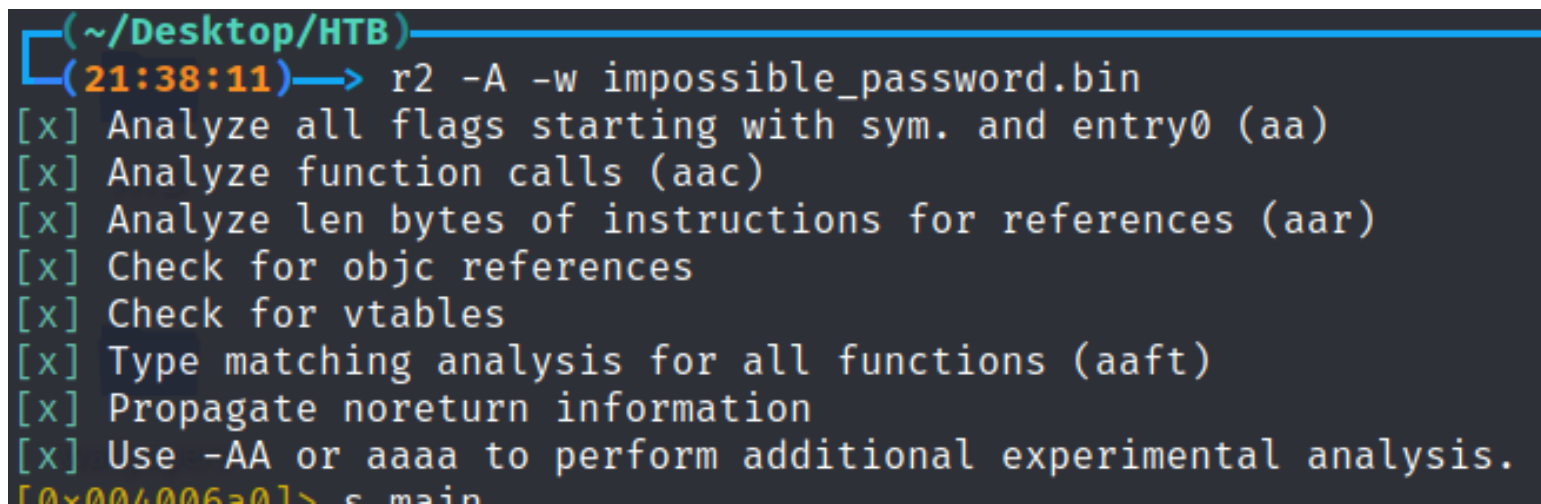
disassembly

Next is some reverse engineering.

I am going to be using radare2 but there are a lot of other options for this.

First let's open the file in Analysis and write mode using

```
r2 -A -w impossible_password.bin
```



```
(~/Desktop/HTB)
(21:38:11) → r2 -A -w impossible_password.bin
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00400620] > c main
```

Next we open the main function

```

[0x004006a0]> s main
[0x0040085d]> pdf
; DATA XREF from entry0 @ 0x4006bd
283: int main (int argc, char **argv);
; var int64_t var_50h @ rbp-0x50
; var int64_t var_44h @ rbp-0x44
; var int64_t var_40h @ rbp-0x40
; var int64_t var_3fh @ rbp-0x3f
; var int64_t var_3eh @ rbp-0x3e
; var int64_t var_3dh @ rbp-0x3d
; var int64_t var_3ch @ rbp-0x3c
; var int64_t var_3bh @ rbp-0x3b
; var int64_t var_3ah @ rbp-0x3a
; var int64_t var_39h @ rbp-0x39
; var int64_t var_38h @ rbp-0x38
; var int64_t var_37h @ rbp-0x37
; var int64_t var_36h @ rbp-0x36
; var int64_t var_35h @ rbp-0x35
; var int64_t var_34h @ rbp-0x34
; var int64_t var_33h @ rbp-0x33
; var int64_t var_32h @ rbp-0x32
; var int64_t var_31h @ rbp-0x31
; var int64_t var_30h @ rbp-0x30
; var int64_t var_2fh @ rbp-0x2f
; var int64_t var_2eh @ rbp-0x2e
; var int64_t var_2dh @ rbp-0x2d
; var int64_t var_20h @ rbp-0x20
; var int64_t var_ch @ rbp-0xc
; var int64_t var_8h @ rbp-0x8
; arg int argc @ rdi
; arg char **argv @ rsi
0x0040085d      55          push rbp
0x0040085e      4889e5      mov rbp, rsp
0x00400861      4883ec50    sub rsp, 0x50
0x00400865      897dbc      mov dword [var_44h], edi ; argc
0x00400868      488975b0    mov qword [var_50h], rsi ; argv
0x0040086c      48c745f870a. mov qword [var_8h], str.SuperSeKretKey ; 0x400a7
0 ; "SuperSeKretKey"
0x00400874      c645c041    mov byte [var_40h], 0x41 ; 'A' ; 65
0x00400878      c645c15d    mov byte [var_3fh], 0x5d ; ']' ; 93
0x0040087c      c645c24b    mov byte [var_3eh], 0x4b ; 'K' ; 75

```

While looking at the main function we notice those input statements that we saw in the ltrace.

```

    mov qword [var_50h], rsi ; argv
700a. mov qword [var_8h], str.SuperSeKretKey ; 0x400a70 ; "SuperSeKretKey"
    mov byte [var_40h], 0x41 ; 'A' ; 65
    mov byte [var_3fh], 0x5d ; ']' ; 93
    mov byte [var_3eh], 0x4b ; 'K' ; 75
    mov byte [var_3dh], 0x72 ; 'r' ; 114
    mov byte [var_3ch], 0x3d ; '=' ; 61
    mov byte [var_3bh], 0x39 ; '9' ; 57
    mov byte [var_3ah], 0x6b ; 'k' ; 107
    mov byte [var_39h], 0x30 ; '0' ; 48
    mov byte [var_38h], 0x3d ; '=' ; 61
    mov byte [var_37h], 0x30 ; '0' ; 48
    mov byte [var_36h], 0x6f ; 'o' ; 111
    mov byte [var_35h], 0x30 ; '0' ; 48
    mov byte [var_34h], 0x3b ; ';' ; 59
    mov byte [var_33h], 0x6b ; 'k' ; 107
    mov byte [var_32h], 0x31 ; '1' ; 49
    mov byte [var_31h], 0x3f ; '?' ; 63
    mov byte [var_30h], 0x6b ; 'k' ; 107
    mov byte [var_2fh], 0x38 ; '8' ; 56
    mov byte [var_2eh], 0x31 ; '1' ; 49
    mov byte [var_2dh], 0x74 ; 't' ; 116
0000 mov edi, 0x400a7f ; const char *format
0000 mov eax, 0
ffff call sym.imp.printf ; int printf(const char *format)
    lea rax, qword [var_20h]
    mov rsi, rax
0000 mov edi, str.20s ; 0x400a82 ; "%20s" ; const char *format
0000 mov eax, 0
ffff call sym.imp.__isoc99_scanf ; int scanf(const char *format)
    lea rax, qword [var_20h]
    mov rsi, rax
0000 mov edi, str.s ; 0x400a87 ; "[%s]\n" ; const char *format
0000 mov eax, 0
ffff call sym.imp.printf ; int printf(const char *format)
    mov rdx, qword [var_8h]
    lea rax, qword [var_20h]
    mov rsi, rdx ; const char *s2
    mov rdi, rax ; const char *s1
ffff call sym.imp.strcmp ; int strcmp(const char *s1, const char *s2)
    mov dword [var_ch], eax
    cmp dword [var_ch], 0
    je 0x400a25

```

After first strcmp() call, it's comparing our input with that string.

```

0x00400907 4889d6 mov rsi, rdx ; const char *s2
0x0040090a 4889c7 mov rdi, rax ; const char *s1
0x0040090d e81efdffff call sym.imp.strcmp ; int strcmp(const char *s1, const char *s2)
0x00400912 8945f4 mov dword [var_ch], eax
0x00400915 837df400 cmp dword [var_ch], 0
0x00400919 740a jz 0x400925
0x0040091b bf01000000 mov edi, 1 ; int status
0x00400920 e85bfdffff call sym.imp.exit ; void exit(int status)
; CODE XREF from main @ 0x400919
0x00400925 bf8d0a4000 mov edi, 0x400a8d ; const char *format
0x0040092a b800000000 mov eax, 0
0x0040092f e8ccfcffff call sym.imp.printf ; int printf(const char *format)
0x00400934 488d45e0 lea rax, [var_20h]
0x00400938 4889c6 mov rsi, rax
0x0040093b bf820a4000 mov edi, str.20s ; 0x400a82 ; "%20s" ; const char *format
0x00400940 b800000000 mov eax, 0
0x00400945 e826fdffff call sym.imp.__isoc99_scanf ; int scanf(const char *format)
0x0040094a bf14000000 mov edi, 0x14 ; 20 ; size_t arg1
0x0040094f e839feffff call fcn.0040078d
0x00400954 4889c2 mov rdx, rax
0x00400957 488d45e0 lea rax, qword [var_20h]
0x0040095b 4889d6 mov rsi, rdx ; const char *s2
0x0040095e 4889c7 mov rdi, rax ; const char *s1

```

We see a void exit statement which is thrown when the wrong input is detected which stops the program.

Otherwise it will continue to the next function.

```

0x00400934 488d45e0 lea rax, qword [var_20h]
0x00400938 4889c6 mov rsi, rax
0x0040093b bf820a4000 mov edi, str.20s ; 0x400a82 ; "%20s" ; const char *format
0x00400940 b800000000 mov eax, 0
0x00400945 e826fdffff call sym.imp.__isoc99_scanf ; int scanf(const char *format)
0x0040094a bf14000000 mov edi, 0x14 ; 20 ; size_t arg1
0x0040094f e839feffff call fcn.0040078d
0x00400954 4889c2 mov rdx, rax
0x00400957 488d45e0 lea rax, qword [var_20h]
0x0040095b 4889d6 mov rsi, rdx ; const char *s2
0x0040095e 4889c7 mov rdi, rax ; const char *s1

```

We see a interesting function call fcn.0040078d when we seek to this function. It seems to be related to the malloc, srand, and time function.


```

0x00400795 897ddc mov dword [var_24h], edi ; arg1
0x00400798 48c745f00000. mov qword [var_10h], 0
0x004007a0 c745ec7e0000. mov dword [var_14h], 0x7e ; '~' ; 126
0x004007a7 c745e8210000. mov dword [var_18h], 0x21 ; '!' ; 33
0x004007ae bf00000000 mov edi, 0 ; time_t *timer
0x004007b3 e898feffff call sym.imp.time ; time_t time(time_t *timer)
0x004007b8 89c2 mov edx, eax
0x004007ba 8b45dc mov eax, dword [var_24h]
0x004007bd 0fafd0 imul edx, eax
0x004007c0 8b05ae082000 mov eax, dword [0x00601074] ; [0x601074:4]=0x17da710
0x004007c6 83c001 add eax, 1
0x004007c9 8905a5082000 mov dword [0x00601074], eax ; [0x601074:4]=0x17da710
0x004007cf 8b059f082000 mov eax, dword [0x00601074] ; [0x601074:4]=0x17da710
0x004007d5 01d0 add eax, edx
0x004007d7 89c7 mov edi, eax ; int seed
0x004007d9 e842feffff call sym.imp.srand ; void srand(int seed)
0x004007de 8b45dc mov eax, dword [var_24h]
0x004007e1 83c001 add eax, 1
0x004007e4 4898 cdqe
0x004007e6 4889c7 mov rdi, rax ; size_t size
0x004007e9 e872feffff call sym.imp.malloc ; void *malloc(size_t size)
0x004007ee 488945f0 mov qword [var_10h], rax
0x004007f2 48837df000 cmp qword [var_10h], 0
0x004007f7 7458 je 0x400851
0x004007f9 c745fc000000. mov dword [var_4h], 0
0x00400800 eb31 jmp 0x400833
; CODE XREF from fcn.0040078d @ 0x400839
0x00400802 e889feffff call sym.imp.rand ; int rand(void)
0x00400807 8b55ec mov edx, dword [var_14h]
0x0040080a 83c201 add edx, 1
0x0040080d 89d1 mov ecx, edx

```

It appears the this function is our random data is generated.
Due to the amount of rands being called.
Lets go back the main and see what we can do with this new information.

```

0x0040095b 4889d6 mov rsi, rdx ; const char *s2
0x0040095e 4889c7 mov rdi, rax ; const char *s1
0x00400961 e8cafcffff call sym.imp.strcmp ; int strcmp(const char *s1, const char *s2)
0x00400966 85c0 test eax, eax
0x00400968 750c jne 0x400976
0x0040096a 488d45c0 lea rax, qword [var_40h]
0x0040096e 4889c7 mov rdi, rax ; int64_t arg1
0x00400971 e802000000 call fcn.00400978
; CODE XREF from main @ 0x400968
0x00400976 c9 leave
0x00400977 c3 ret
0085d]>

```

On main, it's the second strcmp call we've seen using ltrace.
It will set it's return value to eax register.

```

0x0040095b 4889d6 mov rsi, rdx ; const char *s2
0x0040095e 4889c7 mov rdi, rax ; const char *s1
0x00400961 e8cafcffff call sym.imp.strcmp ; int strcmp(const char *s1, const char *s2)
0x00400966 85c0 test eax, eax
< 0x00400968 750c jne 0x400976
0x0040096a 488d45c0 lea rax, qword [var_40h]
0x0040096e 4889c7 mov rdi, rax ; int64_t arg1
0x00400971 e802000000 call fcn.00400978
; CODE XREF from main @ 0x400968
> 0x00400976 c9 leave

```

The next instruction is test, it performs a bitwise AND operation on two operands. This will set ZF(Zero_Flag) to 1 if eax is < 0.

```

0x00400961 e8cafcffff call sym.imp.strcmp ; int strcmp(const char *s1, const char *s2)
0x00400966 85c0 test eax, eax
< 0x00400968 750c jne 0x400976
0x0040096a 488d45c0 lea rax, qword [var_40h]
0x0040096e 4889c7 mov rdi, rax ; int64_t arg1
0x00400971 e802000000 call fcn.00400978
; CODE XREF from main @ 0x400968
> 0x00400976 c9 leave
0x00400977 c3 ret
[0x0040085d]> 

```

In next instruction jne 0x400976 means, if ZF is not 1 jump to instruction 0x400976. Meaning if the input is incorrect then it will not call fcn.00400978(). Lets seek to instruction pointer 0x00400966 and patch it.

```

[0x0040085d]> s 0x00400966
[0x00400966]> wa jmp 0x0040096a
Written 2 byte(s) (jmp 0x0040096a) = wx eb02
[0x00400966]> pdf
; DATA XREF from entry0 @ 0x4006bd

```

We write jmp 0x00400966 to make a jump on that instruction as if our password matched

```

0x0040095b 4889d6 mov rsi, rdx ; const char *s2
0x0040095e 4889c7 mov rdi, rax ; const char *s1
0x00400961 e8cafcffff call sym.imp.strcmp ; int strcmp(const char *s1, const char *s2)
0x00400966 eb02 jmp 0x40096a
< 0x00400968 750c jne 0x400976
> 0x0040096a 488d45c0 lea rax, qword [var_40h]
0x0040096e 4889c7 mov rdi, rax ; int64_t arg1
0x00400971 e802000000 call fcn.00400978
; CODE XREF from main @ 0x400968
> 0x00400976 c9 leave
0x00400977 c3 ret
[0x00400966]> 

```

Finally we run it binary file entering the SuperSeKretKey for both inputs.

```
[0x00400966]> exit
~/(Desktop/HTB)
(22:47:03)→ ./impossible_password.bin
* SuperSeKretKey
[SuperSeKretKey]
** SuperSeKretKey
HTB{40b949f92b86b18}
~/(Desktop/HTB)
```

And we receive our flag!