# Writeup HTB
# [Impossible Password]

Carlos Caminero

## Desafío de *reversing*

*Enlace: [https://app.hackthebox.com/challenges/impossible-password](https://app.hackthebox.com/challenges/impossible-password)*

En esta prueba se nos proporciona un ejecutable denominado **impossible_password.bin**.

A priori se nos ha indicado que este ejecutable nos mostrará la bandera si conseguimos saltar la barrera que se nos impone (una contraseña imposible).

El primer paso que realizaremos será analizar el tipo de fichero. Se trata de un ELF por tanto puede ser ejecutado en cualquier distribución Linux (en este caso, estamos usando Kali Linux).

```
┌──(kali㉿kali)-[/tmp]
└─$ 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
```

Otorgamos permisos de ejecución y lanzamos el programa:

```
┌──(kali㉿kali)-[/tmp]
└─$ chmod +x impossible_password.bin

┌──(kali㉿kali)-[/tmp]
└─$ ./impossible_password.bin
*
```

Nos muestra un **prompt**, para que el usuario introduzca datos. Sin embargo, ante cualquier entrada aleatoria, el programa termina su ejecución erróneamente.

```
┌──(kali㉿kali)-[/tmp]
└─$ ./impossible_password.bin
* jfkdjf
[jfkdjf]

┌──(kali㉿kali)-[/tmp]
└─$ echo $?
1
```

Al no saber cómo utilizar el ejecutable para hallar la bandera, aplicaremos Ingeniería Inversa. La herramienta que utilizaremos será **Radare2** (instalada por defecto en Kali).

Para lanzar Radare ejecutaremos el siguiente comando:

```
└─$ r2 -AAA 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] Finding and parsing C++ vtables (avrr)
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information (aanr)
[x] Finding function preludes
[x] Enable constraint types analysis for variables
```

Con los flags -A, lanzaremos varios análisis al binario que nos facilitará la tarea de reversing.

Primero veremos qué cadenas de texto se encuentran en el segmento de datos del programa. Para ello, ejecutaremos el comando **iz** del shell de radare:

```
[0×004006a0]> iz
[Strings]
nth paddr      vaddr      len size section type  string
―――――――――――――――――――――――――――――――――――――――――――――――――
0   0×00000a70 0×00400a70 14  15   .rodata ascii Supe[redacted]
1   0×00000a82 0×00400a82 4   5    .rodata ascii %20s
2   0×00000a87 0×00400a87 5   6    .rodata ascii [%s]\n
```

Veremos una cadena de texto que según el nombre nos da a entender de que se trata de una clave. Si probamos esta string en el binario, comprobaremos que subimos de nivel y nos solicita una segunda contraseña distinta

```
┌──(kali㉿kali)-[/tmp]
└─$ ./impossible_password.bin
* Sup[redacted]
[Supe[redacted]
** jfkdjfkjdf
```

Como ya vimos las cadenas de texto del segmento de datos, la siguiente opción para hallar la segunda clave (totalmente desconocida) podría ser analizar el código en ensamblador y ver qué hace el ejecutable.

En radare ejecutamos **afl** para ver todas las funciones del programa y nos centraremos en la función **main**.

```
[0×004006a0]> afl
0×004006a0    1 42           entry0
0×00400610    1 6            sym.imp.__libc_start_main
0×004005f0    1 6            sym.imp.putchar
0×00400600    1 6            sym.imp.printf
0×00400620    1 6            sym.imp.srand
0×00400630    1 6            sym.imp.strcmp
0×00400650    1 6            sym.imp.time
0×00400660    1 6            sym.imp.malloc
0×00400670    1 6            sym.imp.__isoc99_scanf
0×00400680    1 6            sym.imp.exit
0×00400690    1 6            sym.imp.rand
0×0040085d    5 283          main
0×00400760    8 141  → 99     entry.init0
0×00400740    3 28           entry.fini0
0×004006d0    4 41           fcn.004006d0
0×00400640    1 6            loc.imp.__gmon_start__
0×0040078d    7 208          fcn.0040078d
0×00400978    5 96           fcn.00400978
0×004005c0    3 26           fcn.004005c0
```

Para visualizar la función main, ejecutamos **pdf@main.** Una vez visualicemos el código ensamblador nos centraremos en la sección donde se compara la segunda cadena de texto.

```
bf8d0a4000      mov edi, 0×400a8d
b800000000      mov eax, 0
e8ccfcffff      call sym.imp.printf

488d45e0        lea rax, [s1]
4889c6          mov rsi, rax
bf820a4000      mov edi, str._20s

b800000000      mov eax, 0
e826fdffff      call sym.imp.__isoc99_scanf

bf14000000      mov edi, 0×14
e839feffff      call fcn.0040078d
4889c2          mov rdx, rax
488d45e0        lea rax, [s1]
4889d6          mov rsi, rdx
4889c7          mov rdi, rax
e8cafcffff      call sym.imp.strcmp
```

Vemos que antes de realizar la comparación, se llama a una función etiquetada como *fcn.0040078d* que devuelve la dirección de una cadena de texto almacenado en el registro rax. Si analizamos la función con **pdf @fcn.0040078d**, veremos que genera una string a partir de números aleatorios, por lo que nos puede dificultar mucho la tarea de introducir una cadena que sea la misma que la que genera el programa.

```
e842feffff      call sym.imp.srand
8b45dc          mov eax, dword [size]
83c001          add eax, 1
4898            cdqe
4889c7          mov rdi, rax
e872feffff      call sym.imp.malloc
```

La solución más rápida será **parchear** el binario para saltarnos la segunda contraseña. Para ello, insertaremos una instrucción de salto justo antes de llamar a la segunda función *printf* que muestra el segundo prompt. El salto apuntará a la instrucción continúa a la comparación de cadenas de texto. Para realizar el parcheo, haremos una copia del ejecutable y lanzaremos radare en modo escritura (-w).

```
┌──(kali㉿kali)-[/tmp]
└─$ cp impossible_password.bin patched.bin

┌──(kali㉿kali)-[/tmp]
└─$ r2 -w patched.bin
[0×004006a0]> aaa
[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] Finding and parsing C++ vtables (avrr)
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information (aanr)
[x] Use -AA or aaaa to perform additional experimental analysis.
```

Abrimos el modo de visualización en el main (**V @main**), pulsamos **p** para mostrar el código ensamblador, y nos posicionamos sobre la instrucción que sustituiremos

```
[0×00400925 [xAdvc]0 0% 165 patched.bin]> pd $r @ main+200 # 0×400925
            ; CODE XREF from main @ 0×400919
            0×00400925      bf8d0a4000      mov edi, 0×400a8d           ; const char *f
            0×0040092a      b800000000      mov eax, 0
            0×0040092f      e8ccfcffff      call sym.imp.printf         ;[1] ; int prin
```

Pulsamos **A** para habilitar el modo de edición y escribimos **JMP** [dirección de salto]:

```
Write some x86-64 assembly ...

[VA:2]> JMP 0×0040096a
* eb43

            ; CODE XREF from main @ 0×400919
        ┌─< 0×00400925   1  eb43          jmp 0×40096a                ; const char *f
            0×00400927      0a4000          or al, byte [rax]
            0×0040092a      b800000000      mov eax, 0
```

Guardamos los cambios, salimos del entorno de radare y ejecutamos nuevamente el binario. Una vez que introduzcamos la primera contraseña, el programa nos devolverá la bandera:


**RETO SUPERADO!**