Nos dan un archivo llamado MigasDePan.

```
┌──(kali㉿kali)-[~/Documents/HackOn]
└─$ file MigasDePan
MigasDePan: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
```

Se trata de un archivo ELF. Lo ejecutamos:

```
┌──(kali㉿kali)-[~/Documents/HackOn]
└─$ ./MigasDePan

Introduce la letra correcta:
```

Nos pide introducir una letra. Con total seguridad nos encontraremos dos opciones, que sea correcta o incorrecta. Buscamos diferentes mensajes y comprobamos si estuviera alguna pista en texto claro:

```
┌──(kali㉿kali)-[~/Documents/HackOn]
└─$ strings MigasDePan
/lib64/ld-linux-x86-64.so.2
fflush
exit
__isoc99_scanf
printf
getchar
stdout
__cxa_finalize
__libc_start_main
libc.so.6
GLIBC_2.7
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
u/UH
[]A\A]A^A_
MdfnJk
jYx}
gWmfk
mlvpc
neU++w
Introduce la letra correcta:
Ya tienes la flag!!
Incorrecta
;*3$"
GCC: (Debian 10.2.1-6) 10.2.1 20210110
Scrt1.o
__abi_tag
crtstuff.c
```

Vemos que el programa nos devolverá esos mensajes según la letra que metamos. Además, podemos suponer que la parte superior a estos mensajes (recuadro verde) también es algo útil, quizá parte de la flag. Continuamos con la ejecución del programa:

```
┌──(kali㉿kali)-[~/Documents/HackOn]
└─$ ./MigasDePan

Introduce la letra correcta:    F
Incorrecta

┌──(kali㉿kali)-[~/Documents/HackOn]
└─$ ./MigasDePan

Introduce la letra correcta:    H

H

Introduce la letra correcta:
```

Conociendo que las flag empiezan con HackOn{, vamos a probarlo hasta ahí.

```
┌──(kali㉿kali)-[~/Documents/HackOn]
└─$ ./MigasDePan

Introduce la letra correcta:    H

H

Introduce la letra correcta:    a

a

Introduce la letra correcta:    c

c

Introduce la letra correcta:    k

k

Introduce la letra correcta:    O

O

Introduce la letra correcta:    n

n

Introduce la letra correcta:    {

{

Introduce la letra correcta:

Incorrecta

┌──(kali㉿kali)-[~/Documents/HackOn]
└─$
```

Efectivamente, vamos bien. Sin embargo, no hay forma aparente de continuar por esta vía.
Vamos a analizar el programa con Radare2:

1º Método:

```
┌──(kali㉿kali)-[~/Documents/HackOn]
└─$ r2 -d MigasDePan
Process with PID 143352 started...
= attach 143352 143352
bin.baddr 0x558a83723000
Using 0x558a83723000
asm.bits 64
[0x7f4a685c2050]> 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] Check for vtables
[TOFIX: aaft can't run in debugger mode.ions (aaft)
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x7f4a685c2050]> afl
0x558a83724090    1 42           entry0
0x558a83726fe0    4 4124 -> 4126 reloc.__libc_start_main
0x558a837240c0    4 41   -> 34   sym.deregister_tm_clones
0x558a837240f0    4 57   -> 51   sym.register_tm_clones
0x558a83724130    5 57   -> 50   sym.__do_global_dtors_aux
0x558a83724080    1 6            sym.imp.__cxa_finalize
0x558a83724170    1 5            entry.init0
0x558a83724000    3 23           map._home_kali_Documents_HackOn_MigasDePan.r_x
0x558a837243f0    1 1            sym.__libc_csu_fini
0x558a837243f4    1 9            sym._fini
0x558a83724390    4 93           sym.__libc_csu_init
0x558a83724352    4 55           sym.comprobacion
0x558a83724175   17 477          main
0x558a83724030    1 6            sym.imp.printf
0x558a83723000   20 792  -> 804  loc.imp._ITM_deregisterTMCloneTable
0x558a83724040    1 6            sym.imp.getchar
0x558a83724050    1 6            sym.imp.fflush
0x558a83724060    1 6            sym.imp.__isoc99_scanf
0x558a83724070    1 6            sym.imp.exit
[0x7f4a685c2050]>
```

Al analizar las funciones, la primera que nos llama la atención es sym.comprobación. Es clara candidata a ser una función implementada en el código original para realizar comprobaciones, tal y como hace el programa cuando introducimos una letra. Vamos a investigar:

```
[0x7f4a685c2050]> pdf@sym.comprobacion
            ; CALL XREFS from main @ 0x558a837241f5, 0x558a83724231, 0x558a8372426d, 0x558a837242a6, 0x558a837242d9
┌ 55: sym.comprobacion (int64_t arg1, int64_t arg2, int64_t arg3);
│           ; var int64_t var_1ch @ rbp-0x1c
│           ; var int64_t var_18h @ rbp-0x18
│           ; var int64_t var_14h @ rbp-0x14
│           ; var int64_t var_4h @ rbp-0x4
│           ; arg int64_t arg1 @ rdi
│           ; arg int64_t arg2 @ rsi
│           ; arg int64_t arg3 @ rdx
│           0x558a83724352      55             push rbp
│           0x558a83724353      4889e5         mov rbp, rsp
│           0x558a83724356      89f9           mov ecx, edi            ; arg1
│           0x558a83724358      89f0           mov eax, esi            ; arg2
│           0x558a8372435a      8955e4         mov dword [var_1ch], edx ; arg3
│           0x558a8372435d      89ca           mov edx, ecx
│           0x558a8372435f      8855ec         mov byte [var_14h], dl
│           0x558a83724362      8845e8         mov byte [var_18h], al
│           0x558a83724365      c745fc000000.  mov dword [var_4h], 0
│           0x558a8372436c      0fbe55ec       movsx edx, byte [var_14h]
│           0x558a83724370      0fbe45e8       movsx eax, byte [var_18h]
│           0x558a83724374      3345e4         xor eax, dword [var_1ch]
│           0x558a83724377      39c2           cmp edx, eax
│       ┌─< 0x558a83724379      7507           jne 0x558a83724382
│       │   0x558a8372437b      b801000000     mov eax, 1
│      ┌──< 0x558a83724380      eb05           jmp 0x558a83724387
│      │└─> 0x558a83724382      b800000000     mov eax, 0
│      │    ; CODE XREF from sym.comprobacion @ 0x558a83724380
│      └──> 0x558a83724387      5d             pop rbp
└           0x558a83724388      c3             ret
[0x7f4a685c2050]>
```

Observamos que claramente hace una comprobación entre dos registros: cmp edx, eax. Vamos a poner un punto de ruptura justo ahí, y ver que valores almacenan:

```
[0x7fcc994bd050]> db 0x55fbea41f377
[0x7fcc994bd050]> dc

Introduce la letra correcta:    I
hit breakpoint at: 0x55fbea41f377
[0x55fbea41f377]> dr
rax = 0x00000048
rbx = 0x55fbea41f390
rcx = 0x00000049
rdx = 0x00000049
r8 = 0x0000000a
r9 = 0x00000000
r10 = 0xfffffffffffffb83
r11 = 0x7fcc9935a780
r12 = 0x55fbea41f090
r13 = 0x00000000
r14 = 0x00000000
r15 = 0x00000000
rsi = 0x0000004d
rdi = 0x00000049
rsp = 0x7fff915c9770
rbp = 0x7fff915c9770
rip = 0x55fbea41f377
rflags = 0x00000206
orax = 0xffffffffffffffff
[0x55fbea41f377]>
```

Como vemos, al introducir una letra I (0x00000049 en hexadecimal), nuestro registro rdx almacena este carácter y a su vez, el registro rax almacena el carácter correcto con el que lo compara, en este caso 0x00000048 o "H". Sabiendo que podemos obtener así la flag, continuamos ejecutándolo hasta ver el primer carácter después de "HackOn{"

```
Introduce la letra correcta:    F
hit breakpoint at: 0×561bbeb42377
[0×561bbeb42377]> dr
rax = 0×00000048
rbx = 0×561bbeb42390
rcx = 0×00000046
rdx = 0×00000046
r8  = 0×0000000a
r9  = 0×00000000
r10 = 0×561bbeb43047
r11 = 0×00000246
r12 = 0×561bbeb42090
r13 = 0×00000000
r14 = 0×00000000
r15 = 0×00000000
rsi = 0×00000059
rdi = 0×00000046
rsp = 0×7ffc5616a020
rbp = 0×7ffc5616a020
rip = 0×561bbeb42377
rflags = 0×00000206
orax = 0×ffffffffffffffff
[0×561bbeb42377]>
```

Como aparece, rax almacena la letra “H”, así que continuamos hasta el final “HackOn{Hilo_encontrado_!!}”

2º Método:

Analizamos el programa igual con r2. Sin embargo, en vez de centrarnos en la función comprobación, vamos a centrarnos en el main:

```
[0×563f20c2d358]> pdf@main
            ; DATA XREF from entry0 @ 0×563f20c2d0ad
┌ 477: int main (int argc, char **argv, char **envp);
│           ; var int64_t var_9h @ rbp-0×9
│           ; var int64_t var_8h @ rbp-0×8
│           ; var int64_t var_1h @ rbp-0×1
│           0×563f20c2d175      55             push rbp
│           0×563f20c2d176      4889e5         mov rbp, rsp
│           0×563f20c2d179      4883ec10       sub rsp, 0×10
│           0×563f20c2d17d      c645ff01       mov byte [var_1h], 1
│           0×563f20c2d181      c745f8000000.  mov dword [var_8h], 0
│       ┌─< 0×563f20c2d188      e9b4010000     jmp 0×563f20c2d341
│      ┌──> 0×563f20c2d18d      488b05e42e00.  mov rax, qword [reloc.stdout] ; [0×563f20c30078:8]=0×7f6e068a76c0
│      ││   0×563f20c2d194      4889c7         mov rdi, rax
│      ││   0×563f20c2d197      e8b4feffff     call sym.imp.fflush     ; int fflush(FILE *stream)
│      ││   0×563f20c2d19c      488d3d850e00.  lea rdi, str._nIntroduce_la_letra_correcta:_t ; 0×563f20c2e028 ; "\nIntroduce la letra correcta:\t"
│      ││   0×563f20c2d1a3      b800000000     mov eax, 0
│      ││   0×563f20c2d1a8      e883feffff     call sym.imp.printf     ; int printf(const char *format)
│      ││   0×563f20c2d1ad      488d45f7       lea rax, [var_9h]
│      ││   0×563f20c2d1b1      4889c6         mov rsi, rax
│      ││   0×563f20c2d1b4      488d3d8c0e00.  lea rdi, [r10]          ; 0×563f20c2e047 ; "%c"
│      ││   0×563f20c2d1bb      b800000000     mov eax, 0
│      ││   0×563f20c2d1c0      e89bfeffff     call sym.imp.__isoc99_scanf ; int scanf(const char *format)
│      ││   0×563f20c2d1c5      e876feffff     call sym.imp.getchar    ; int getchar(void)
│      ││   0×563f20c2d1ca      837df805       cmp dword [var_8h], 5
│     ┌───< 0×563f20c2d1ce      7f32           jg 0×563f20c2d202
│     │││   0×563f20c2d1d0      488b15792e00.  mov rdx, qword [obj.a]  ; [0×563f20c30050:8]=0×563f20c2e008 str.MdfnJk
│     │││   0×563f20c2d1d7      8b45f8         mov eax, dword [var_8h]
│     │││   0×563f20c2d1da      4898           cdqe
│     │││   0×563f20c2d1dc      4801d0         add rax, rdx
│     │││   0×563f20c2d1df      0fb600         movzx eax, byte [rax]
│     │││   0×563f20c2d1e2      0fbec8         movsx ecx, al
│     │││   0×563f20c2d1e5      0fb645f7       movzx eax, byte [var_9h]
│     │││   0×563f20c2d1e9      0fbec0         movsx eax, al
│     │││   0×563f20c2d1ec      ba05000000     mov edx, 5
│     │││   0×563f20c2d1f1      89ce           mov esi, ecx
│     │││   0×563f20c2d1f3      89c7           mov edi, eax
│     │││   0×563f20c2d1f5      e858010000     call sym.comprobacion
│     │││   0×563f20c2d1fa      8845ff         mov byte [var_1h], al
│    ┌────< 0×563f20c2d1fd      e9df000000     jmp 0×563f20c2d2e1
│    │└───> 0×563f20c2d202      837df809       cmp dword [var_8h], 9
│    │┌───< 0×563f20c2d206      7f36           jg 0×563f20c2d23e
│    ││││   0×563f20c2d208      488b15492e00.  mov rdx, qword [obj.b]  ; [0×563f20c30058:8]=0×563f20c2e00f str.jYx
│    ││││   0×563f20c2d20f      8b45f8         mov eax, dword [var_8h]
```

Observamos la secuencia del programa y vemos un bucle que nos pide al inicio introducir la letra correcta, lo imprime y posteriormente lee el carácter que introducimos. Justo después vemos que se esta cargando una cadena de texto: str.MdfnJk, que es justo una cadena que

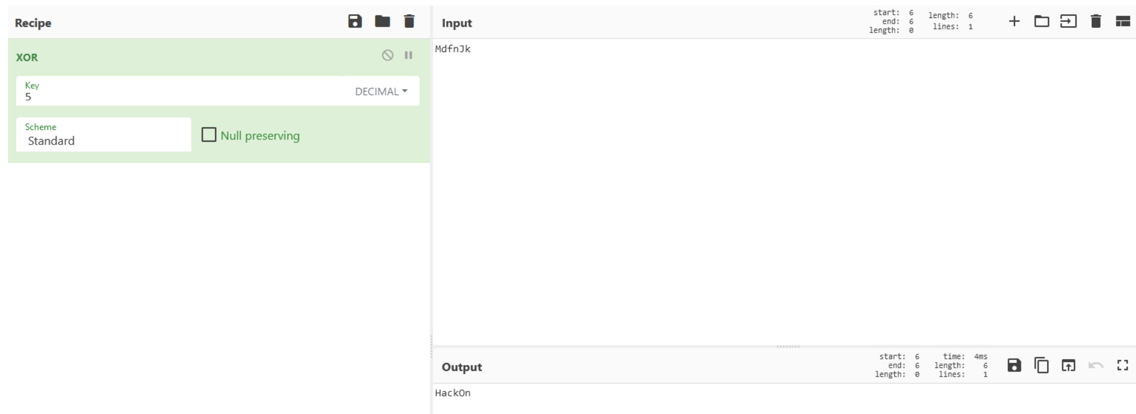
habíamos encontrado con strings. Sin embargo, comprobamos si tuviera algún ROT, pero no aparece nada. Lo siguiente que nos llama la atención es que justo antes de la primera llamada a comprobación, se cargan tres parámetros en registros: edx, ecx y eax.

Si nos dirigimos a la función sym.comprobación y buscamos estos parámetros:

```
[0x563f20c2d358]> pdf@sym.comprobacion
            ; CALL XREFS from main @ 0x563f20c2d1f5, 0x563f20c2d231, 0x563f20c2d26d, 0x563f20c2d2a6, 0x563f20c2d2d9
┌ 55: sym.comprobacion (int64_t arg1, int64_t arg2, int64_t arg3);
│           ; var int64_t var_1ch @ rbp-0x1c
│           ; var int64_t var_18h @ rbp-0x18
│           ; var int64_t var_14h @ rbp-0x14
│           ; var int64_t var_4h @ rbp-0x4
│           ; arg int64_t arg1 @ rdi
│           ; arg int64_t arg2 @ rsi
│           ; arg int64_t arg3 @ rdx
│           0x563f20c2d352      55             push rbp
│           0x563f20c2d353      4889e5         mov rbp, rsp
│           0x563f20c2d356      89f9           mov ecx, edi               ; arg1
│           ;-- rip:
│           0x563f20c2d358 b    89f0           mov eax, esi               ; arg2
│           0x563f20c2d35a      8955e4         mov dword [var_1ch], edx ; arg3
│           0x563f20c2d35d      89ca           mov edx, ecx
│           0x563f20c2d35f      8855ec         mov byte [var_14h], dl
│           0x563f20c2d362      8845e8         mov byte [var_18h], al
│           0x563f20c2d365      c745fc000000.  mov dword [var_4h], 0
│           0x563f20c2d36c      0fbe55ec       movsx edx, byte [var_14h]
│           0x563f20c2d370      0fbe45e8       movsx eax, byte [var_18h]
│           0x563f20c2d374      3345e4         xor eax, dword [var_1ch]
│           0x563f20c2d377      39c2           cmp edx, eax
│       ┌─< 0x563f20c2d379      7507           jne 0x563f20c2d382
│       │   0x563f20c2d37b      b801000000     mov eax, 1
│      ┌──< 0x563f20c2d380      eb05           jmp 0x563f20c2d387
│      │└─> 0x563f20c2d382      b800000000     mov eax, 0
│      │    ; CODE XREF from sym.comprobacion @ 0x563f20c2d380
│      └──> 0x563f20c2d387      5d             pop rbp
└           0x563f20c2d388      c3             ret
[0x563f20c2d358]>
```

Observamos que edx (que tenia un valor entero de 5), se carga sobre var_1ch, que posteriormente descubrimos que hace un XOR 5 con mi variable en eax, que antes era esi!!

Vamos a probar haciendo un XOR con la cadena de texto "raro" que hemos encontrado:



Efectivamente, el programa realiza una comparación de la letra que introducimos con la una cadena en XOR. Buscamos las siguientes cadenas, con sus respectivas claves del XOR y ya tenemos la flag!

"HackOn{Hilo_encontrado_!!}"

PD: Se podría realizar en GHidra siguiente este método 2.