VilgarOs development

Flag

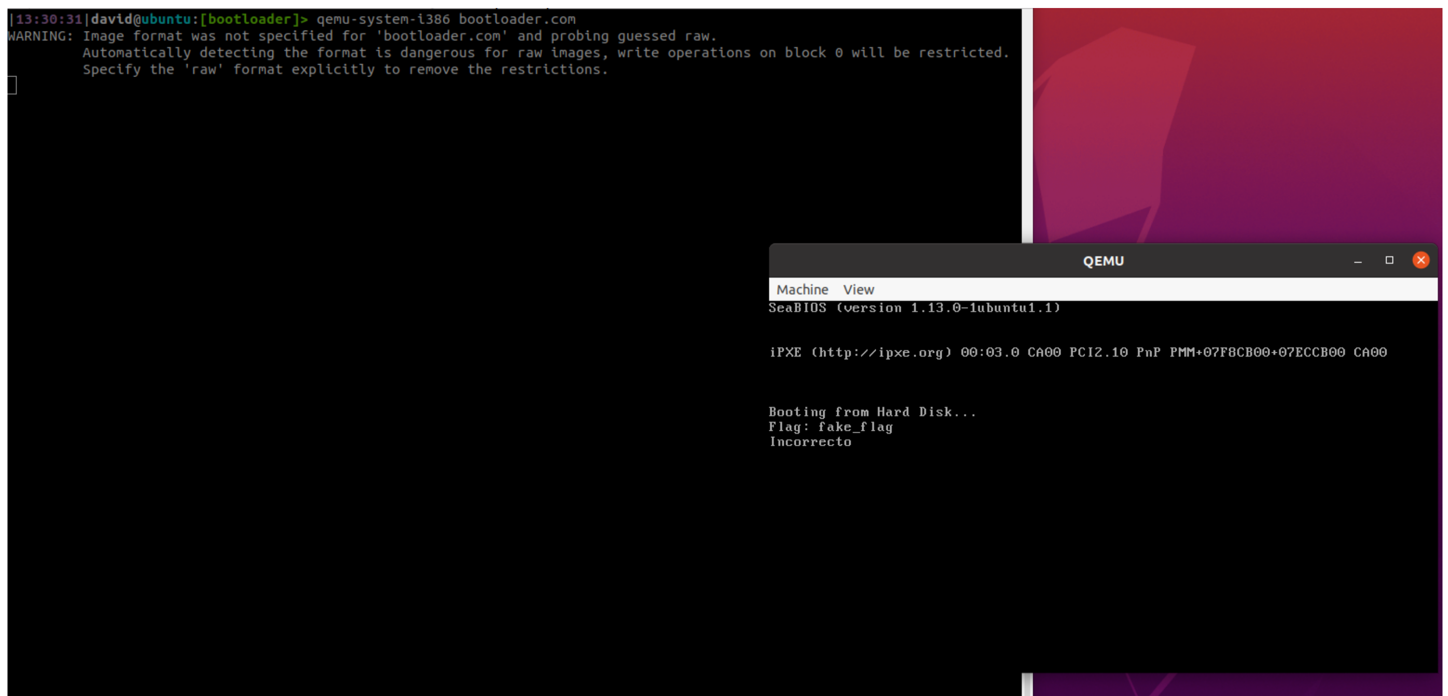
```
HackOn{l37_b00t_pLz}
```

Writeup

- Sacamos información básica del archivo. Vemos que es un bootloader de MS-DOS, por ello ocupa exactamente 512 bytes.

```
|13:33:27|david@ubuntu:[bootloader]> ll bootloader.com
-rw-rw-r-- 1 david david 512 dic  3 21:34 bootloader.com
|13:33:30|david@ubuntu:[bootloader]> file bootloader.com
bootloader.com: DOS/MBR boot sector
|13:33:33|david@ubuntu:[bootloader]> strings bootloader.com
Flag:
Incorrecto
vilgarOS booting...
J7?SR;;`SijV
|13:33:35|david@ubuntu:[bootloader]>
```

- Ejecutamos con Qemu para ver su funcionalidad. Vemos que pide una flag y si metemos cualquier cadena de caractéres nos devuelve el output `Incorrecto`



- Analizamos estaticamente con objdump (Hay que especificar la arquitectura i8086 para que desensamble para 16 bits y el tipo, que es binary porque es un binario puro, sin headers ni nada) y xxd.

```
|13:40:17|david@ubuntu:[bootloader]> objdump -D -m i8086 bootloader.com -b binary

bootloader.com:     formato del fichero binary


Desensamblado de la sección .data:

00000000 <.data>:
   0:   bb 88 7c                mov    $0x7c88,%bx
   3:   e8 15 00                call   0x1b
   6:   be ae 7c                mov    $0x7cae,%si
   9:   bf bb 7c                mov    $0x7cbb,%di
   c:   e8 2c 00                call   0x3b
   f:   bb ff ff                mov    $0xffff,%bx
  12:   e8 19 00                call   0x2e
  15:   e8 3e 00                call   0x56
  18:   e9 b9 00                jmp    0xd4
  1b:   80 fb ff                cmp    $0xff,%bl
  1e:   74 0d                   je     0x2d
  20:   b4 0e                   mov    $0xe,%ah
  22:   8a 07                   mov    (%bx),%al
  24:   84 c0                   test   %al,%al
  26:   74 05                   je     0x2d
  28:   cd 10                   int    $0x10
  2a:   43                      inc    %bx
  2b:   eb f5                   jmp    0x22
  2d:   c3                      ret
  2e:   e8 ea ff                call   0x1b
  31:   b8 0a 0e                mov    $0xe0a,%ax
  34:   cd 10                   int    $0x10
  36:   b0 0d                   mov    $0xd,%al
  38:   cd 10                   int    $0x10
  3a:   c3                      ret
  3b:   bb 0c 00                mov    $0xc,%bx
  3e:   b4 00                   mov    $0x0,%ah
  40:   cd 16                   int    $0x16
  42:   3c 0d                   cmp    $0xd,%al
  44:   74 0f                   je     0x55
  46:   88 04                   mov    %al,(%si)
  48:   88 25                   mov    %ah,(%di)
  4a:   b4 0e                   mov    $0xe,%ah
  4c:   cd 10                   int    $0x10
  4e:   46                      inc    %si
  4f:   47                      inc    %di
  50:   4b                      dec    %bx
  51:   85 db                   test   %bx,%bx
  53:   75 e9                   jne    0x3e
  55:   c3                      ret
  56:   b9 00 00                mov    $0x0,%cx
  59:   be ae 7c                mov    $0x7cae,%si
  5c:   01 ce                   add    %cx,%si
  5e:   8a 04                   mov    (%si),%al
  60:   bb c8 7c                mov    $0x7cc8,%bx
  63:   01 cb                   add    %cx,%bx
```

```
|13:40:22|david@ubuntu:[bootloader]> xxd bootloader.com
00000000: bb88 7ce8 1500 beae 7cbf bb7c e82c 00bb  ..|.....|..|.,..
00000010: ffff e819 00e8 3e00 e9b9 0080 fbff 740d  ......>.......t.
00000020: b40e 8a07 84c0 7405 cd10 43eb f5c3 e8ea  ......t...C.....
00000030: ffb8 0a0e cd10 b00d cd10 c3bb 0c00 b400  ................
00000040: cd16 3c0d 740f 8804 8825 b40e cd10 4647  ..<.t....%....FG
```

```
00000040: cd10 3c0d 7401 8804 8825 b40e cd10 4647  ..<.t....%....FG
00000050: 4b85 db75 e9c3 b900 00be ae7c 01ce 8a04  K..u.......|....
00000060: bbc8 7c01 cb8a 1f30 c3be bb7c 01ce 8a04  ..|....0...|....
00000070: 38d8 750d 4183 f90c 7cdf bb9a 7ce8 9bff  8.u.A...|...|...
00000080: c3bb 8f7c e894 ffc3 466c 6167 3a20 0049  ...|....Flag: .I
00000090: 6e63 6f72 7265 6374 6f00 7669 6c67 6172  ncorrecto.vilgar
000000a0: 4f53 2062 6f6f 7469 6e67 2e2e 2e00 0000  OS booting......
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000c0: 0000 0000 0000 0000 4a37 3f53 523b 3b60  ........J7?SR;;`
000000d0: 5369 6a56 ebfe 0000 0000 0000 0000 0000  SijV............
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000100: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000110: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000120: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000130: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000150: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000160: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000170: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000180: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000190: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001f0: 0000 0000 0000 0000 0000 0000 0000 55aa  ..............U.
|13:43:12|david@ubuntu:[bootloader]>
```

- Analizamos estaticamente en ida especificando 16 bits. Vemos que el programa empieza llamando a varias subrutinas y acabo saltando al final donde una instrucción se queda saltando sobre sí misma (una especie de hlt). Además se pueden observar un total de 4 subrutinas y varias variables como las que habiamos visto con `strings`. Al final del todo vemos el magic del bootsector (55h,AAh).

```
seg000:0000
seg000:0000 ; Segment type: Pure code
seg000:0000 seg000          segment byte public 'CODE' use16
seg000:0000                 assume cs:seg000
seg000:0000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:0000                 mov     bx, 7C88h
seg000:0003                 call    sub_1B
seg000:0006                 mov     si, 7CAEh
seg000:0009                 mov     di, 7CBBh
seg000:000C                 call    sub_3B
seg000:000F                 mov     bx, 0FFFFh
seg000:0012                 call    sub_2E
seg000:0015                 call    sub_56
seg000:0018                 jmp     loc_D4
seg000:001B
seg000:001B ; =============== S U B R O U T I N E =======================================
seg000:001B
seg000:001B
seg000:001B sub_1B          proc near               ; CODE XREF: seg000:0003↑p
seg000:001B                                         ; sub_2E↓p ...
seg000:001B                 cmp     bl, 0FFh
seg000:001E                 jz      short locret_2D
seg000:0020                 mov     ah, 0Eh
seg000:0022
seg000:0022 loc_22:                                 ; CODE XREF: sub_1B+10↓j
seg000:0022                 mov     al, [bx]
seg000:0024                 test    al, al
seg000:0026                 jz      short locret_2D
seg000:0028                 int     10h             ; - VIDEO - WRITE CHARACTER AND ADVANCE CURSOR (TTY WRITE)
seg000:0028                                         ; AL = character, BH = display page (alpha modes)
seg000:0028                                         ; BL = foreground color (graphics modes)
seg000:002A                 inc     bx
seg000:002B                 jmp     short loc_22
seg000:002D ; ---------------------------------------------------------------------------
seg000:002D
seg000:002D locret_2D:                              ; CODE XREF: sub_1B+3↑j
seg000:002D                                         ; sub_1B+B↑j
seg000:002D                 retn
seg000:002D sub_1B          endp
seg000:002D
seg000:002E
seg000:002E ; =============== S U B R O U T I N E =======================================
seg000:002E
seg000:002E
seg000:002E sub_2E          proc near               ; CODE XREF: seg000:0012↑p
seg000:002E                 call    sub_1B
seg000:0031                 mov     ax, 0E0Ah
```

```
seg000:0056 ; =============== S U B R O U T I N E =======================================
seg000:0056
seg000:0056
seg000:0056 sub_56          proc near               ; CODE XREF: seg000:0015↑p
seg000:0056                                         ; DATA XREF: sub_3B:loc_40↑r
seg000:0056                 mov     cx, 0
seg000:0059
seg000:0059 loc_59:                                 ; CODE XREF: sub_56+22↓j
seg000:0059                 mov     si, 7CAEh
seg000:005C                 add     si, cx
seg000:005E                 mov     al, [si]
seg000:0060                 mov     bx, 7CC8h
seg000:0063                 add     bx, cx
seg000:0065                 mov     bl, [bx]
seg000:0067                 xor     bl, al
seg000:0069                 mov     si, 7CBBh
seg000:006C                 add     si, cx
seg000:006E                 mov     al, [si]
seg000:0070                 cmp     al, bl
seg000:0072                 jnz     short loc_81
seg000:0074                 inc     cx
seg000:0075                 cmp     cx, 0Ch
seg000:0078                 jl      short loc_59
seg000:007A                 mov     bx, 7C9Ah
seg000:007D                 call    sub_1B
seg000:0080                 retn
seg000:0081 ; ---------------------------------------------------------------------------
seg000:0081
seg000:0081 loc_81:                                 ; CODE XREF: sub_56+1C↑j
seg000:0081                 mov     bx, 7C8Fh
seg000:0084                 call    sub_1B
seg000:0087                 retn
seg000:0087 sub_56          endp
seg000:0087
seg000:0087 ; ---------------------------------------------------------------------------
seg000:0088                 db  46h ; F
seg000:0089 aLag            db 'lag: ',0
seg000:008F                 db  49h ; I
seg000:0090 aNcorrecto      db 'ncorrecto',0
seg000:009A                 db  76h ; v
seg000:009B                 db  69h ; i
seg000:009C aLgarosBooting  db 'lgarOS booting...',0
seg000:00AE                 db 1Ah dup(0), 4Ah, 37h, 3Fh, 53h, 52h, 2 dup(3Bh), 60h
seg000:00AE                 db 53h, 69h, 6Ah, 56h
seg000:00D4 ; ---------------------------------------------------------------------------
seg000:00D4
seg000:00D4 loc_D4:                                 ; CODE XREF: seg000:0018↑j
seg000:00D4                                         ; seg000:loc_D4↓j
seg000:00D4                 jmp     short loc_D4
seg000:00D4 ; ---------------------------------------------------------------------------
seg000:00D6                 db 128h dup(0), 55h, 0AAh
seg000:00D6 seg000          ends
```

- Ida proporciona información a la derecha de los interrupts que se van utilizando en el código. En este caso se utilizan dos: `int 10h` y `int 16h` , que sirven para escribir por pantalla y para escanear el teclado respectivamente (output e input). Con está información podemos entender de manera aproximada las subrutinas y por lo tanto renombrarlas para que sea más legible el código.

```
seg000:0000 ; Segment type: Pure code
seg000:0000 seg000          segment byte public 'CODE' use16
seg000:0000                 assume cs:seg000
seg000:0000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:0000                 mov     bx, 7C88h       ; MAIN
seg000:0003                 call    print
seg000:0006                 mov     si, 7CAEh
seg000:0009                 mov     di, 7CBBh
seg000:000C                 call    scan
seg000:000F                 mov     bx, 0FFFFh
seg000:0012                 call    println
seg000:0015                 call    sub_56
seg000:0018                 jmp     Fin
```

- La subrutina print imprime la string a la cual apunta un puntero guardado en bx. Hay que tener en cuenta que un bootloader se carga en la dirección 0x7c00 de memoria por lo que si hace referencia a una variable en 0x7c88, la variable estará a un offset de 0x88 del principio del

binario. Por ejemplo, el primer print imprime `Flag:` , ya que es la string que está a un offset de 0x88.

```
seg000:0088                  db  46h ; F
seg000:0089 aLag             db 'lag: ',0
seg000:008F                  db  49h ; I
seg000:0090 aNcorrecto       db 'ncorrecto',0
seg000:009A                  db  76h ; v
seg000:009B                  db  69h ; i
seg000:009C aLgarosBooting   db 'lgarOS booting...',0
seg000:00AE                  db 1Ah dup(0), 4Ah, 37h, 3Fh, 53h, 52h, 2 dup(3Bh), 60h
seg000:00AE                  db 53h, 69h, 6Ah, 56h
seg000:00D4 ; ---------------------------------------------------------------------------
seg000:00D4
seg000:00D4 Fin:                                     ; CODE XREF: seg000:0018↑j
seg000:00D4                                          ; seg000:Fin↓j
seg000:00D4                  jmp     short Fin
seg000:00D4 ; ---------------------------------------------------------------------------
seg000:00D6                  db 128h dup(0), 55h, 0AAh
seg000:00D6 seg000           ends
```

- La subrutina println llama a print y luego imprime `\n\r` , es decir un cambio de línea. Y si le pasas como argumento -1 o 0xFFFF imprime solo un cambio de línea, como si no tuviese argumento la subrutina. Este es el caso de la única llamada a esta subrutina.
- La subrutina scan escanea el input con `int 16,0` guardando en la dirección de memoria apuntada por `si` el carácter en ascii escaneado y en la dirección de memoria apuntada por `di` el scan code de la tecla presionada. Esto lo hace en un bucle hasta haber escaneado un total de 12 caracteres. Un scan code es un código único por cada tecla, y por tanto varios carácteres pueden tener el mismo (La 'A' y la 'a' tienen el mismo scan code y diferentes asciis). En este caso se llama a scan con los argumentos 0x7cae y 0x7cbb, que si miramos tienen 13 bytes a 0 cada uno (12 escaneados y un null byte)(esto lo representa `1Ah dup(0)` en ida, 26*0x00).

```
seg000:003B
seg000:003B
seg000:003B scan        proc near               ; CODE XREF: seg000:000C↑p
seg000:003B             mov     bx, 0Ch
seg000:003E
seg000:003E scan_loop:                          ; CODE XREF: scan+18↓j
seg000:003E             mov     ah, 0
seg000:0040
seg000:0040 loc_40:                             ; DATA XREF: print+D↑r
seg000:0040                                     ; println+6↑r ...
seg000:0040             int     16h             ; KEYBOARD - READ CHAR FROM BUFFER, WAIT IF EMPTY
seg000:0040                                     ; Return: AH = scan code, AL = character
seg000:0042             cmp     al, 0Dh
seg000:0044             jz      short end_scan
seg000:0046             mov     [si], al
seg000:0048             mov     [di], ah
seg000:004A             mov     ah, 0Eh
seg000:004C             int     10h             ; - VIDEO - WRITE CHARACTER AND ADVANCE CURSOR (TTY WRITE)
seg000:004C                                     ; AL = character, BH = display page (alpha modes)
seg000:004C                                     ; BL = foreground color (graphics modes)
seg000:004E             inc     si
seg000:004F             inc     di
seg000:0050             dec     bx
seg000:0051             test    bx, bx
seg000:0053             jnz     short scan_loop
seg000:0055
seg000:0055 end_scan:                           ; CODE XREF: scan+9↑j
seg000:0055             retn
seg000:0055 scan        endp
seg000:0055
```

- Ahora que entendemos esas tres subrutinas vamos a tratar de entender la que queda, que es un poco más compleja. Esta hace un xor byte a byte entre el string guardado en 0x7cae (los ascii escaneados) y el string guardado en 0x7cc8 (unos valores hardcodeados) y luego lo compara con el string guardado en 0x7cbb (los scan codes). Si en algún momento del bucle la comparación da

falso, la ejecucción salta a un trozo de código en el que se imprime 0x7c8f (que contiene el string `Incorrecto`). En el caso de que todas las comparaciones sean satisfactorias se imprime `VilgarOs booting...` , que es lo que queremos conseguir.

```
seg000:0056 ; =============== S U B R O U T I N E =======================================
seg000:0056
seg000:0056
seg000:0056 comprobacion    proc near               ; CODE XREF: seg000:0015↑p
seg000:0056                                         ; DATA XREF: scan:loc_40↑r
seg000:0056                 mov     cx, 0
seg000:0059
seg000:0059 comprobacion_loop:                      ; CODE XREF: comprobacion+22↓j
seg000:0059                 mov     si, 7CAEh
seg000:005C                 add     si, cx
seg000:005E                 mov     al, [si]
seg000:0060                 mov     bx, 7CC8h
seg000:0063                 add     bx, cx
seg000:0065                 mov     bl, [bx]
seg000:0067                 xor     bl, al
seg000:0069                 mov     si, 7CBBh
seg000:006C                 add     si, cx
seg000:006E                 mov     al, [si]
seg000:0070                 cmp     al, bl
seg000:0072                 jnz     short incorrecto
seg000:0074                 inc     cx
seg000:0075                 cmp     cx, 0Ch
seg000:0078                 jl      short comprobacion_loop
seg000:007A                 mov     bx, 7C9Ah
seg000:007D                 call    print
seg000:0080                 retn
seg000:0081 ; ---------------------------------------------------------------------------
seg000:0081
seg000:0081 incorrecto:                             ; CODE XREF: comprobacion+1C↑j
seg000:0081                 mov     bx, 7C8Fh
seg000:0084                 call    print
seg000:0087                 retn
seg000:0087 comprobacion    endp
seg000:0087
seg000:0087 ; ---------------------------------------------------------------------------
seg000:0088                 db  46h ; F
seg000:0089 aLag            db 'lag: ',0
seg000:008F                 db  49h ; I
seg000:0090 aNcorrecto      db 'ncorrecto',0
seg000:009A                 db  76h ; v
seg000:009B                 db  69h ; i
seg000:009C aLgarosBooting  db 'lgarOS booting...',0
seg000:00AE                 db 1Ah dup(0), 4Ah, 37h, 3Fh, 53h, 52h, 2 dup(3Bh), 60h
seg000:00AE                 db 53h, 69h, 6Ah, 56h
seg000:00D4 ; ---------------------------------------------------------------------------
```

- En resumen:
  - Se imprime `Flag:`
  - Se escanea el input, tanto ascii como scan codes.
  - Se imprime un cambio de linea.
  - Se comprueba nuestro input haciendo un xor con valores hardcodeados y comparando con scan codes.
  - Si la comprobación es correcta se imprime `VilgarOs booting...` , en otro caso se imprime `Incorrecto` .
- Para sacar la flag tenemos que crear un mapa que relacione valores en ascii y sus scancodes. Como en el reto se especifica que los únicos caracteres que puede tener la flag son letras (tanto en mayúscula como en minúscula), números y el carácter '_', creamos un mapa solo con esos

caracteres. En mi caso lo he implementado con una matriz, en la que el primer array son los valores en hexadecimal de los scan codes y el segundo array los valores en ascii de cada carácter.

- Una vez creado el mapa recorremos los 12 valores hardcodeados y vamos probando con cada carácter posible comparando el valor de hacer un xor entre el carácter en ascii y su scan code con cada valor hardcodeado (hay que tener en cuenta que el inverso de la operación xor es la misma operación). En el caso de que esta comparación sea satisfactoria imprimimos el carácter.

```
#include<stdio.h>

int main(int argc, char const *argv[])
{
    int caracteres[2][63]={{0x1E,0x30,0x2E,0x20,0x12,0x21,0x22,0x23,0x17,0x24,0x25,0x26,0x32,0
    int flag[12]={0x4a,0x37,0x3f,0x53,0x52,0x3b,0x3b,0x60,0x53,0x69,0x6a,0x56};
    printf("HackOn{");
    for (int i = 0; i < 12; ++i)
    {
        int encontrado=0;
        for (int j = 0; j < 63&&!encontrado; ++j)
        {
            if((caracteres[0][j]^caracteres[1][j])==flag[i]){
                encontrado=1;
                printf("%c",caracteres[1][j]);
            }
        }
    }
    printf("}\n");
    return 0;
}
```

Al compilar y ejecutar este solver conseguimos la flag:

```
|15:42:21|david@ubuntu:[bootloader]> ./solver
HackOn{l37_b00t_pLz}
|15:50:23|david@ubuntu:[bootloader]>
```

> Nota: Se ha usado está página de documentación de scan codes para crear la matriz usada en el solver.

Probado por

- Dbd4

Autor

- David Billhardt

- Twitter
- Github