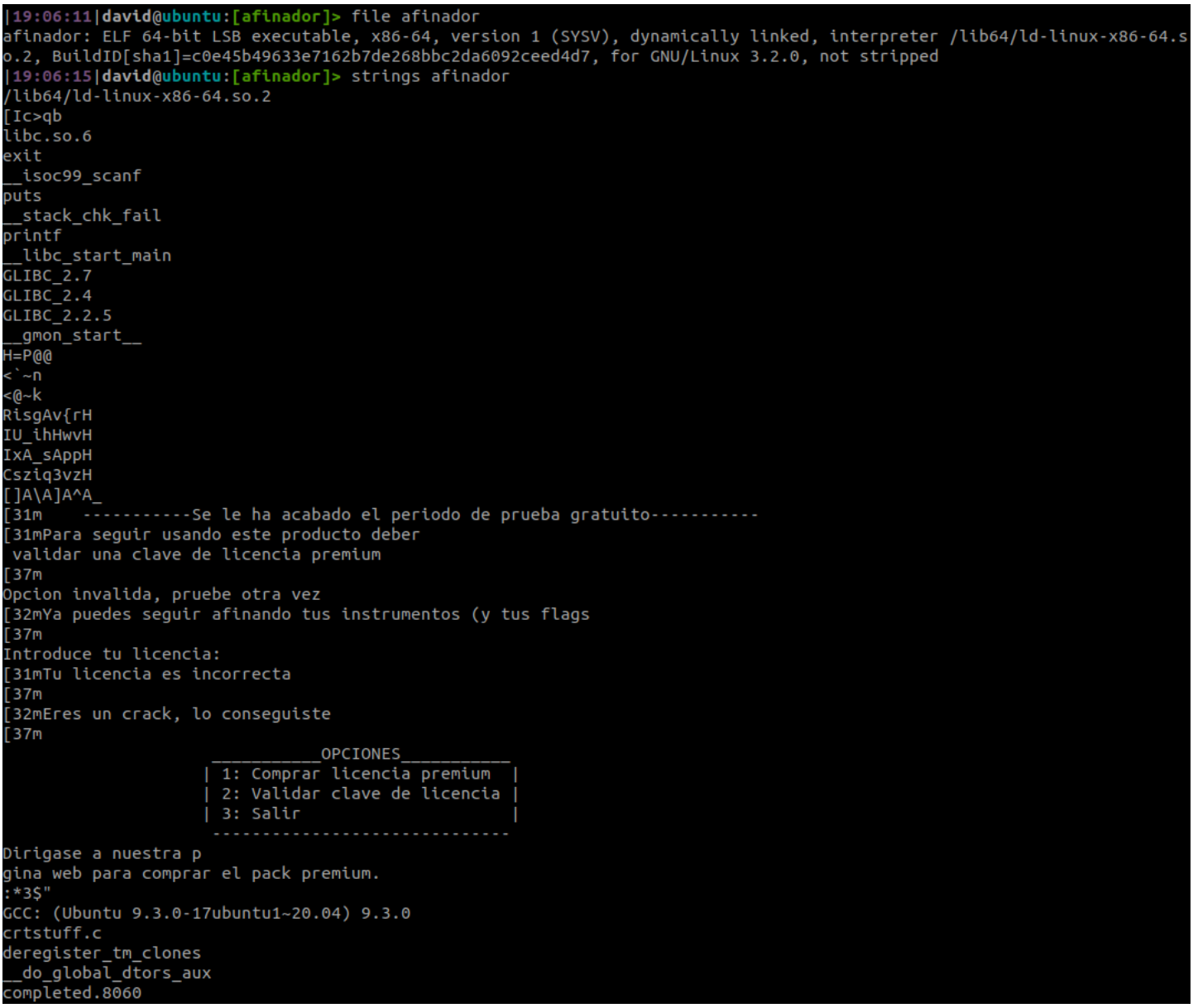
Afinando claves

Flag

```
HackOn{hAS_afFinAdO_cOrrEctam3ntE}
```

Writeup

- Sacamos información básica del binario. Vemos que es un elf de 64 bits no estripeado.



- Ejecutamos para ver la funcionalidad. El programa imprime un menú de opciones. Probando las opciones vemos que la importante es segunda, que valida la licencia premium (nuestra flag).

```
|19:10:00|david@ubuntu:[afinador]> ./afinador

    -----------Se le ha acabado el periodo de prueba gratuito-----------

Para seguir usando este producto deberá validar una clave de licencia premium


                    __________OPCIONES__________
                   | 1: Comprar licencia premium  |
                   | 2: Validar clave de licencia |
                   | 3: Salir                     |
                    ------------------------------
> 1

Dirigase a nuestra página web para comprar el pack premium.

                    __________OPCIONES__________
                   | 1: Comprar licencia premium  |
                   | 2: Validar clave de licencia |
                   | 3: Salir                     |
                    ------------------------------
> 2

Introduce tu licencia: licencia_fake

Tu licencia es incorrecta

                    __________OPCIONES__________
                   | 1: Comprar licencia premium  |
                   | 2: Validar clave de licencia |
                   | 3: Salir                     |
                    ------------------------------
> 3
|19:11:11|david@ubuntu:[afinador]>
```

- Vamos a abrir el programa con ghidra para analizarlo más en profundidad. Main tiene una funcionalidad que se ajusta a lo que habíamos probado.

```
undefined8 main(void)

{
  long in_FS_OFFSET;
  int local_18;
  int local_14;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_14 = 0;
  puts("\n\x1b[31m    ----------Se le ha acabado el periodo de prueba gratuito----------\n");
  puts(&DAT_00402058);
  do {
    banner();
    __isoc99_scanf(&DAT_004020b3,&local_18);
    if (local_18 == 3) {
                    /* WARNING: Subroutine does not return */
      exit(0);
    }
    if (local_18 < 4) {
      if (local_18 == 1) {
        comprar();
      }
      else {
        if (local_18 != 2) goto LAB_00401257;
        local_14 = validar();
      }
    }
    else {
LAB_00401257:
      puts("Opcion invalida, pruebe otra vez");
    }
    if (local_14 != 0) {
      puts(&DAT_004020e0);
      if (local_10 == *(long *)(in_FS_OFFSET + 0x28)) {
        return 0;
      }
                    /* WARNING: Subroutine does not return */
      __stack_chk_fail();
    }
  } while( true );
}
```

- Antes nos habiamos dado cuenta de que la importante era la segunda opción, la cual llama a la función validar, veamos que hace.

```
undefined8 validar(void)

{
  undefined8 uVar1;
  long in_FS_OFFSET;
  int local_6c;
  char local_68 [48];
  undefined8 local_38;
  undefined8 local_30;
  undefined8 local_28;
  undefined8 local_20;
  undefined2 local_18;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_38 = 0x727b764167736952;
  local_30 = 0x76774868695f5549;
  local_28 = 0x707041735f417849;
  local_20 = 0x7a763371697a7343;
  local_18 = 0x7d43;
  printf("\nIntroduce tu licencia: ");
  __isoc99_scanf(&DAT_00402142,local_68);
  encode(local_68);
  local_6c = 0;
  do {
    if (0x21 < local_6c) {
      puts("\n\x1b[32mEres un crack, lo conseguiste\x1b[37m");
      uVar1 = 1;
LAB_004014b8:
      if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
        __stack_chk_fail();
      }
      return uVar1;
    }
    if (local_68[local_6c] != *(char *)((long)&local_38 + (long)local_6c)) {
      puts("\n\x1b[31mTu licencia es incorrecta\x1b[37m\n");
      uVar1 = 0;
      goto LAB_004014b8;
    }
    local_6c = local_6c + 1;
  } while( true );
}
```

- **validar** declara varios valores en hexadecimal, pide la licencia, la escanea con scanf, la pasa como argumento a `encode()` y luego compara el input tras haberlo pasado por `encode()` con los valores hardcodeados al principio.

```
undefined8 validar(void)

{
  undefined8 uVar1;
  long in_FS_OFFSET;
  int local_6c;
  char local_68 [48];
  undefined8 local_38;
  undefined8 local_30;
  undefined8 local_28;
  undefined8 local_20;
  undefined2 local_18;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_38 = 0x727b764167736952;
  local_30 = 0x76774868695f5549;
  local_28 = 0x707041735f417849;
  local_20 = 0x7a763371697a7343;
  local_18 = 0x7d43;
  printf("\nIntroduce tu licencia: ");
  __isoc99_scanf(&DAT_00402142,local_68);
  encode(local_68);
  local_6c = 0;
  do {
    if (0x21 < local_6c) {
      puts("\n\x1b[32mEres un crack, lo conseguiste\x1b[37m");
      uVar1 = 1;
LAB_004014b8:
      if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
        __stack_chk_fail();
      }
      return uVar1;
    }
    if (local_68[local_6c] != *(char *)((long)&local_38 + (long)local_6c)) {
      puts("\n\x1b[31mTu licencia es incorrecta\x1b[37m\n");
      uVar1 = 0;
      goto LAB_004014b8;
    }
    local_6c = local_6c + 1;
  } while( true );
}
```

- Podemos pasar esos valores de hexadecimal a caracteres, cambiando el endianness, o podemos usar gdb para verlos directamente en hexadecimal. Ponemos un breakpoint tras la declaración de los valores, ejecutamos y miramos en `$rbp-0x30` que es donde empiezan los valores.

```
   0x00000000004013f4 <+25>:    xor    eax,eax
   0x00000000004013f6 <+27>:    movabs rax,0x727b764167736952
   0x0000000000401400 <+37>:    movabs rdx,0x76774868695f5549
   0x000000000040140a <+47>:    mov    QWORD PTR [rbp-0x30],rax
   0x000000000040140e <+51>:    mov    QWORD PTR [rbp-0x28],rdx
   0x0000000000401412 <+55>:    movabs rax,0x707041735f417849
   0x000000000040141c <+65>:    movabs rdx,0x7a763371697a7343
   0x0000000000401426 <+75>:    mov    QWORD PTR [rbp-0x20],rax
   0x000000000040142a <+79>:    mov    QWORD PTR [rbp-0x18],rdx
   0x000000000040142e <+83>:    mov    WORD PTR [rbp-0x10],0x7d43
   0x0000000000401434 <+89>:    lea    rdi,[rip+0xcee]        # 0x402129
   0x000000000040143b <+96>:    mov    eax,0x0
   0x0000000000401440 <+101>:   call   0x4010a0 <printf@plt>
   0x0000000000401445 <+106>:   lea    rax,[rbp-0x60]
   0x0000000000401449 <+110>:   mov    rsi,rax
   0x000000000040144c <+113>:   lea    rdi,[rip+0xcef]        # 0x402142
   0x0000000000401453 <+120>:   mov    eax,0x0
   0x0000000000401458 <+125>:   call   0x4010b0 <__isoc99_scanf@plt>
   0x000000000040145d <+130>:   lea    rax,[rbp-0x60]
   0x0000000000401461 <+134>:   mov    rdi,rax
   0x0000000000401464 <+137>:   call   0x401291 <encode>
   0x0000000000401469 <+142>:   mov    DWORD PTR [rbp-0x64],0x0
   0x0000000000401470 <+149>:   jmp    0x4014a1 <validar+198>
   0x0000000000401472 <+151>:   mov    eax,DWORD PTR [rbp-0x64]
   0x0000000000401475 <+154>:   cdqe
   0x0000000000401477 <+156>:   movzx  edx,BYTE PTR [rbp+rax*1-0x60]
   0x000000000040147c <+161>:   mov    eax,DWORD PTR [rbp-0x64]
   0x000000000040147f <+164>:   cdqe
   0x0000000000401481 <+166>:   movzx  eax,BYTE PTR [rbp+rax*1-0x30]
   0x0000000000401486 <+171>:   cmp    dl,al
   0x0000000000401488 <+173>:   je     0x40149d <validar+194>
   0x000000000040148a <+175>:   lea    rdi,[rip+0xcb7]        # 0x402148
   0x0000000000401491 <+182>:   call   0x401080 <puts@plt>
   0x0000000000401496 <+187>:   mov    eax,0x0
   0x000000000040149b <+192>:   jmp    0x4014b8 <validar+221>
   0x000000000040149d <+194>:   add    DWORD PTR [rbp-0x64],0x1
   0x00000000004014a1 <+198>:   cmp    DWORD PTR [rbp-0x64],0x21
   0x00000000004014a5 <+202>:   jle    0x401472 <validar+151>
   0x00000000004014a7 <+204>:   lea    rdi,[rip+0xcc2]        # 0x402170
   0x00000000004014ae <+211>:   call   0x401080 <puts@plt>
   0x00000000004014b3 <+216>:   mov    eax,0x1
   0x00000000004014b8 <+221>:   mov    rcx,QWORD PTR [rbp-0x8]
   0x00000000004014bc <+225>:   xor    rcx,QWORD PTR fs:0x28
   0x00000000004014c5 <+234>:   je     0x4014cc <validar+241>
   0x00000000004014c7 <+236>:   call   0x401090 <__stack_chk_fail@plt>
   0x00000000004014cc <+241>:   leave
   0x00000000004014cd <+242>:   ret
End of assembler dump.
pwndbg> b *0x0000000000401434
Punto de interrupción 1 at 0x401434
pwndbg> r
```

```
pwndbg> x/s $rbp-0x30
0x7fffffffde90: "RisgAv{rIU_ihHwvIxA_sAppCsziq3vzC}@"
pwndbg>
```

- Ahora vamos a ver que hace encode. Encode pasa por todos los caracteres de su argumento (el input) y los va cambiando.

```
void encode(long param_1)

{
  int iVar1;
  int local_c;

  local_c = 0;
  while (local_c < 0x22) {
    if ((*(char *)(param_1 + local_c) < 'a') || ('z' < *(char *)(param_1 + local_c))) {
      if ((*(char *)(param_1 + local_c) < 'A') || ('Z' < *(char *)(param_1 + local_c))) {
        *(undefined *)(param_1 + local_c) = *(undefined *)(param_1 + local_c);
      }
      else {
        iVar1 = (*(char *)(param_1 + local_c) + -0x41) * 5 + 8;
        *(char *)(param_1 + local_c) = (char)iVar1 + (char)(iVar1 / 0x1a) * -0x1a + 'A';
      }
    }
    else {
      iVar1 = (*(char *)(param_1 + local_c) + -0x61) * 5 + 8;
      *(char *)(param_1 + local_c) = (char)iVar1 + (char)(iVar1 / 0x1a) * -0x1a + 'a';
    }
    local_c = local_c + 1;
  }
  return;
}
```

- En el caso de que el carácter no sea una letra es decir que sea menor que 'a' o mayor que 'z' y menor que 'A' o mayor que 'Z', ese carácter no se cambia.
- En el caso de que sea una letra mayúscula se hacen las siguientes operaciones:

```
y=(x-0x41)*5+8
x=y+(y/26)*(-26)+'A'
```

> La segunda operación es equivalente a `x=y%26+'A'` , ya que la división que se utiliza es división entera, y 0x41 es 'A' en hexadecimal, por lo que haciendo esos cambios la operación sería la siguiente:

```
x=((x-'A')*5+8)%26+'A'
```

- En el caso de que sea una letra minúscula se hace algo parecido:

```
x=((x-'a')*5+8)%26+'a'
```

- Ahora que ya sabemos como se cifra, debemos sacar la operación inversa para descifrar. Por ello voy a simplificar aún más la operación de cifrado. Lo que hace al restar o sumar 'a' o 'A' es mapear una letra a un número de tal manera que la letra 'a' o 'A' sea 0 (ejemplo: C->2->0x43->C). Si quitamos este simple mapeo el cifrado quedaría así:

```
cifrado = input*5+8 mod 26
```

- Recordemos que en aritmética modular la operación inversa a la multiplicación por un número es la multiplicación por su inverso modular. El inverso modular de 5 es 21 en módulo 26 (se puede hallar con python haciendo `pow(5,-1,26)`). Aplicando esto:

```
input = (cifrado-8)*21 mod 26
```

- Con esto y sabiendo que el texto cifrado es `RisgAv{rIU_ihHwvIxA_sAppCsziq3vzC}` creamos un solver en python que nos saca la flag.

```
string = "RisgAv{rIU_ihHwvIxA_sAppCsziq3vzC}"
flag =  ""

'''
(5*x+8)%26=y -> (5*x)%26=(y-8)%26 -> x=((y-8)*5⁻¹)%26 -> x=((y-8)*21)%26
21 inverso modular de 5
'''

inv = pow(5,-1,26)  # inv=21
dec = lambda x,y: chr(((inv * (x-y-8)) % 26)+y) # la y es para mapear letras a numeros de 0 al

for char in string:
    if (char >= 'A' and char <= 'Z'):   # si es letra minuscula
        flag += dec(ord(char), ord('A'))
    elif (char >= 'a' and char <= 'z'): # si es letra mayuscula
        flag += dec(ord(char), ord('a'))
    else:
        flag+=char

print(flag)
```

- Ejecutamos el solver y tenemos la flag.

```
|19:37:48|david@ubuntu:[afinador]> python3 solucion.py
HackOn{hAS_afFinAdO_cOrrEctam3ntE}
|19:50:44|david@ubuntu:[afinador]>
```

> Nota: El cifrado utilizado es un cifrado real llamado affine cipher, si te das cuenta de esto hay herramientas online que lo descifran.

Probado por

- Dbd4

- DiegoAltF4

Autores

- David Billhardt
  - Twitter
  - Github
- Diego Palacios
  - Twitter
  - Github