CLS- BASIC CRACKME KEYGENING

Manuel Blanco

CRACKSLATINOS

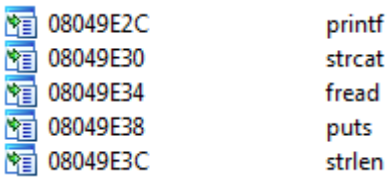
Tras descargar el ejecutable podemos ver que el formato del mismo es ELF y como consecuencia necesitaremos un sistema GNU/Linux para poder llevar a cabo el proceso dinámico de ingeniería inversa que aplicaremos.

Comencemos probando a poner un usuario y contraseña aleatorios:

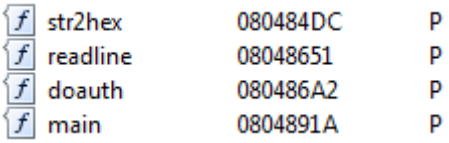
```
root@kali:~/Desktop# ./crackme
EZ Basic CrackMe 1
By ca0s
[+] User: manuelb
[+] Pass: crackslatinos
manuelb:CrACkslAtinos - Nope
root@kali:~/Desktop#
```

Nos saca por consola un mensaje donde apreciamos el nombre y la contraseña alterada en algunos caracteres seguida de un mensaje indicándonos el fracaso.

Antes de comenzar con el análisis dinámico se hará uso de IDA para analizar estáticamente el desensamblado del ejecutable y así ir atando cabos. Tras cargar el mismo en IDA observamos las funciones que importa e implementa:



Vemos varias interesantes, pero aún no podemos sacar nada en conclusión, seguramente hará uso de ellas en el algoritmo de validación interno pero ya lo veremos después. Acto seguido miramos los nombres de las funciones que alberga:



Vemos varias funciones públicas muy interesantes, sobre todo la que más nos puede llamar la atención sería doauth (do authentication).

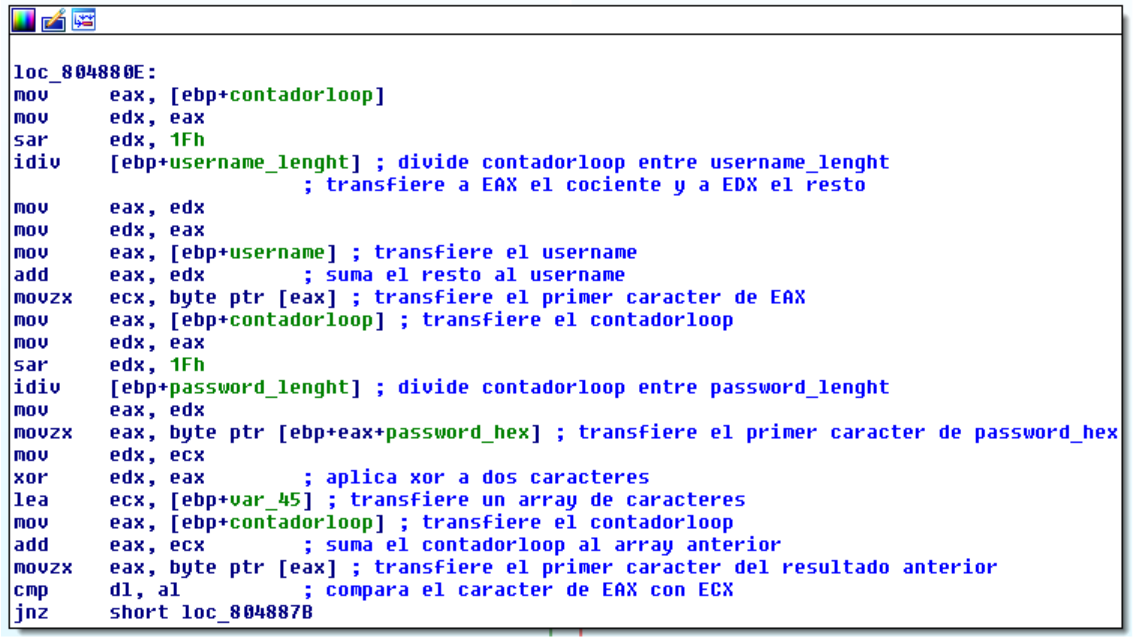
Lo primero que vamos a hacer va ser indagar en los bloques básicos de la función principal, después de una breve lectura superficial diferenciamos el bloque más importante:

```
loc_8048A17:            ; "EZ Basic CrackMe 1\nBy ca0s"
                mov     dword ptr [esp], offset s
                call    _puts
                mov     dword ptr [esp], offset format ; "[+] User: "
                call    _printf
                mov     dword ptr [esp+4], 32 ; int
                lea     eax, [esp+223]
                mov     [esp], eax ; ptr
                call    readline
                mov     dword ptr [esp], offset aPass ; "[+] Pass: "
                call    _printf
                mov     dword ptr [esp+4], 64 ; int
                lea     eax, [esp+158]
                mov     [esp], eax ; ptr
                call    readline
                lea     eax, [esp+30]
                mov     [esp+8], eax ; resultado
                lea     eax, [esp+158]
                mov     [esp+4], eax ; password
                lea     eax, [esp+223]
                mov     [esp], eax ; username
                call    doauth
                lea     eax, [esp+30]
                mov     [esp], eax ; s
                call    _puts
                mov     eax, 0
                lea     esp, [ebp-8]
                pop     ebx
                pop     edi
                pop     ebp
                retn
main            endp
```

Como podemos ver básicamente le pasa tres punteros de caracteres a la función doauth en el orden: username, password, resultado.

También vemos que el número de bytes varía de 32 a 64 a la hora de leer las líneas, podemos deducir que el número de caracteres de la contraseña será superior al del usuario con creces.

Lo siguiente que debemos hacer es al igual que con la función main indagar dentro de la función doauth, tras una lectura detenida encontraremos dentro de un ciclo iterativo del tipo for un bloque básico muy interesante:



```
loc_804880E:
mov     eax, [ebp+contadorloop]
mov     edx, eax
sar     edx, 1Fh
idiv    [ebp+username_lenght] ; divide contadorloop entre username_lenght
                              ; transfiere a EAX el cociente y a EDX el resto
mov     eax, edx
mov     edx, eax
mov     eax, [ebp+username] ; transfiere el username
add     eax, edx            ; suma el resto al username
movzx   ecx, byte ptr [eax] ; transfiere el primer caracter de EAX
mov     eax, [ebp+contadorloop] ; transfiere el contadorloop
mov     edx, eax
sar     edx, 1Fh
idiv    [ebp+password_lenght] ; divide contadorloop entre password_lenght
mov     eax, edx
movzx   eax, byte ptr [ebp+eax+password_hex] ; transfiere el primer caracter de password_hex
mov     edx, ecx
xor     edx, eax            ; aplica xor a dos caracteres
lea     ecx, [ebp+var_45] ; transfiere un array de caracteres
mov     eax, [ebp+contadorloop] ; transfiere el contadorloop
add     eax, ecx            ; suma el contadorloop al array anterior
movzx   eax, byte ptr [eax] ; transfiere el primer caracter del resultado anterior
cmp     dl, al              ; compara el caracter de EAX con ECX
jnz     short loc_804887B
```

Como podrán apreciar en la ilustración, comenté el funcionamiento interno del desensamblado y aquí es donde se encuentra el algoritmo que nos dará la victoria sobre el crackme.

Es importante que renombréis las variables internas como os mostré para entender mejor el funcionamiento del algoritmo, si no, puede ser un proceso más tedioso aún. Y no solo con este crackme, si no, a la hora de analizar cualquier desensamblado pues identificar los elementos internos es una fase muy importante.

Ahora que ya tenemos una idea de cómo funciona el algoritmo hay que pasar a contrastar la hipótesis de manera dinámica con un depurador, en esta ocasión utilizaremos GDB. Comenzaremos poniendo un breakpoint en el inicio del bloque básico:

b *doauth+364

Tras poner este breakpoint solo quedaría ir verificando lo que anteriormente mencionamos, para eso lo ejecutamos y vamos trazando con steps in.

Vemos que tras cargar el carácter de sumar el resto al username aplica un XOR a dicho carácter junto al primer carácter de la clave en hexadecimal y luego compara si el resultado de dicho xor es igual al carácter hexadecimal del array de bytes que carga internamente. Si la condición se cumple no toma un salto y va cargando progresivamente el mensaje de chico bueno (si se sigue cumpliendo la condición con los siguientes caracteres obviamente).

El punto más importante para poder elaborar el keygen sería la instrucción que transfiere un array de caracteres, una vez llegados a dicho punto podemos observar que la string contiene 27 caracteres de los cuales extraeremos sus bytes en formato hexadecimal.

```
gdb-peda$ x/s 0xbffff313
0xbffff313:     "W2^2v\031w\022\\9A5z\024q&0#0\rh A3W2@"
gdb-peda$ x/27x 0xbffff313
0xbffff313:     0x57    0x32    0x5e    0x32    0x76    0x19    0x77    0x12
0xbffff31b:     0x5c    0x39    0x41    0x35    0x7a    0x14    0x71    0x26
0xbffff323:     0x4f    0x23    0x4f    0x0d    0x68    0x20    0x41    0x33
0xbffff32b:     0x57    0x32    0x40
gdb-peda$
```

Y entonces llegamos a formar la siguiente igualdad donde debemos despejar la incógnita que será los caracteres de nuestra contraseña:

PRIMERCARACTER **XOR** INCOGNITA = PRIMERCARACTERARRAY

Lo despejamos fácilmente:

PRIMERCARACTER **XOR** PRIMERCARACTERARRAY = INCOGNITA

Es importante destacar que el array de bytes lo indexa al revés, es decir empieza por el final y también algo crucial será el formato de la contraseña. Son 27 bytes lo que quiere decir que son 54 caracteres (27*2) y puede ser que el resultado del xor sea interpretado con un único carácter hexadecimal de ahí que hay forzar el formato.

También es importante a la hora de revertir de dos en dos los caracteres, podemos hacerlo simple para concatenar de manera que nos lo invierta en cada iteración.

Para hacer el keygen yo escogí python, obviamente podéis utilizar el lenguaje que queráis, a mí me parece un lenguaje muy adecuado a estos quehaceres.

Tras investigar el bucle for, vemos que va descendiendo de 1 en 1 hasta llegar a 0 y comienza en 26, pues solo queda este resultado:

```
#!/usr/bin/env python
#! coding: utf-8

import sys
username = sys.argv[1]

byte_list = ['57','32','5e','32','76','19','77','12','5c','39','41','35','7a','14','71','26','4f','23','4f','0d','68','20','41','33','57','32','40']
password = ""

for i in range(26, -1, -1):
    password = "%.2x" % (int((username[i % len(username)]).encode("hex"), 16) ^ int(byte_list[i], 16)) + password

print "Password:", password
```

https://github.com/manuelbp01/CrackMe/blob/master/basic.py

No hay mucho que añadir la verdad. El “PRIMERCARACTER” equivale al carácter de la cadena username que esté en la posición del módulo de la división del contadorloop entre el largo del usuario, a ese carácter se le aplica el operador lógico disyunción exclusiva (xor) y se almacena en una variable en el formato adecuado y de la manera adecuada para que cuando el algoritmo interno del crackme lo indexe esté en la posición correcta.

Una prueba:

```
root@kali:~/Desktop# python basic.py manuelb
Password: 3a5330471375157f3d57345016761c4721562a610a4d205d22572c
root@kali:~/Desktop# ./crackme
EZ Basic CrackMe 1
By ca0s
[+] User: manuelb
[+] Pass: 3a5330471375157f3d57345016761c4721562a610a4d205d22572c
manuelb:3A5330471375157F3D57345016761C4721562A610A4D205D22572C - WellDoneNextOneWillBeHarder
root@kali:~/Desktop#
```

Como podemos ver el resultado fue el esperado, hemos logrado generar claves para cualquier usuario así que damos por finalizado el desafío. Un agradecimiento muy grande a ca0s de ka0labs por el crackme.

Espero que os haya parecido amena la lectura y si alguien ha aprendido algo nuevo mejor aún!

Para finalizar quería agradecer a mi gran amigo **PastaCLS** su apoyo y ayuda incondicional, sin el nada de esto habría sido posible.

Si queréis poneros en contacto conmigo podéis hacerlo en la lista o por mi correo: Manuelbp01@gmail.com

Un saludo y hasta la próxima!