

CRYPTO 400

EL presente reto se compone de las piezas de otros dos retos. Reversing 400 y pwning 300, donde se obtiene la ruta para descargar la definición de la API y la llave para interactuar con la misma.



http://critical_infrastructure_voting_enforcement_system.hack-defender.mx:3166/ff9cc879ccdbb7fa6b8ea728a5a86015/getAPI

```
Felicidades! Flag: hackdef{F1b0n4cc1_3n_3l3cc10nes_t4mbi3n}
```

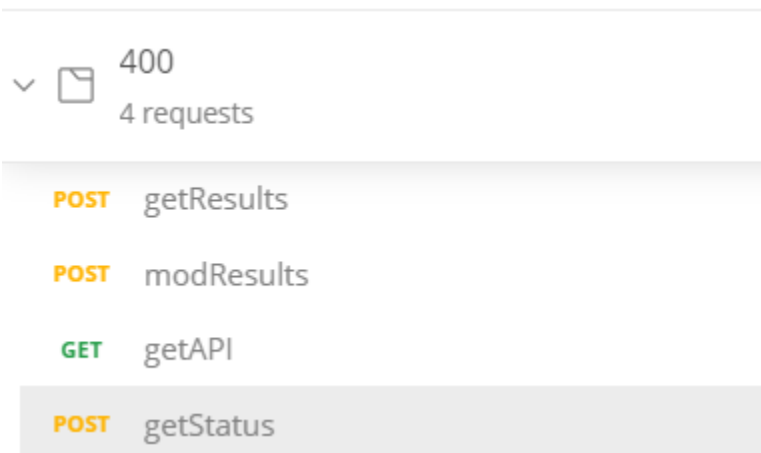
Descarga la definicion y codigo de la API aqui:

http://critical_infrastructure_voting_enforcement_system.hack-defender.mx:3166/afa06d12/definicion.zip

Una vez descargada la API, se obtiene una colección de postman, que muestra la documentación para interactuar con el servicio. Además del código fuente del servidor hecho con Flask.

	400.postman_col...	4,645	1,181	JSON File	10/13/2020 2:2...	F513FFBC
	server.py	2,824	958	PY File	10/14/2020 3:1...	126E3454

La definición de la API al abrirse con Postman muestra lo siguiente:



Cada petición que se realiza al servidor, contiene una descripción de lo que hace, los parámetros necesarios, además de la estructura del cuerpo de la petición. Por ejemplo:

getAPI

Descripcion

- Metodo: GET
- Descripcion: Obtiene el codigo fuente de la API asi como esta definicion

Parametros requeridos

- Ninguno

GET

http://critical_infrastructure_voting_enforcement_system.hack-defender.mx:3166/ff9cc879ccdbb7fa6b8ea728a5a86015/getAPI

Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

response

getResults

Descripcion

- Tipo: POST
- Descripcion breve: Obtiene el resultado de quienes han ganado la eleccion (Resultado el reto), junto a la firma del comando ejecutado en formato JSON

Parametros:

- JSON: {"access_code":"","hex_identifier":""}
- access_code:Codigo de acceso a la API
- hex_identifier: Valor hexadecimal, funciona como padding

POST

http://critical_infrastructure_voting_enforcement_system.hack-defender.mx:3166/getResults

Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

1 [{"access_code":"","hex_identifier":""}]

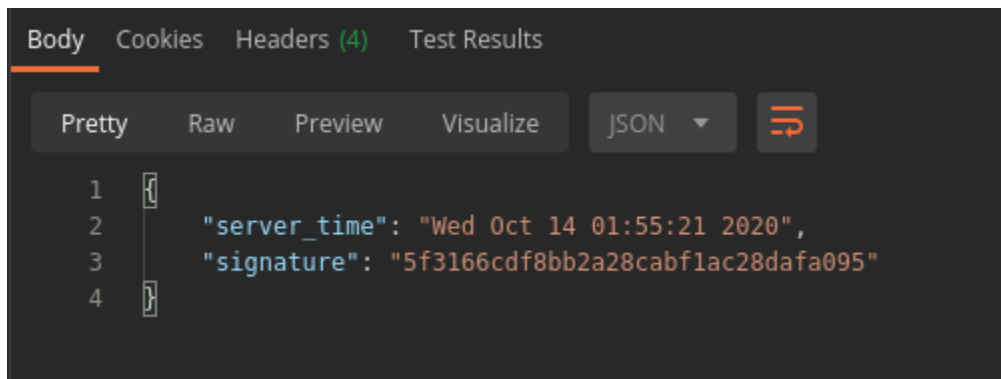
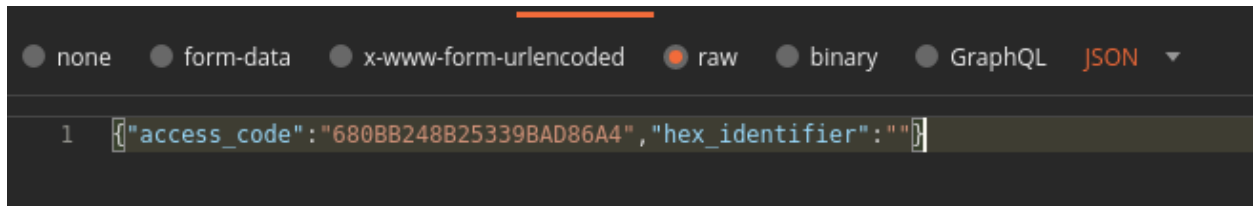
Revisando el código fuente de la aplicación podemos ver que los parámetros `access_code` y `hex_identifier` son necesarios para las peticiones por post. Access code esta dado por uno de los retos anteriores, mientras que `hex_identifier` puede estar vacío.

```
@app.route("/getStatus", methods=['POST'])
def getStatus():
    content = request.json
    access = content['access_code']
    if access == access_code:
        return jsonify( server_time = time.asctime(), signature = get_firma())
    else:
        return jsonify( error = "Codigo de acceso incorrecto")
```

Se valida únicamente el código de acceso, y lo que retorna el servidor es un archivo JSON con la hora del servidor y la firma del comando ejecutado.

http://critical_infrastructure_voting_enforcement_system.hack-defender.mx:3166/getStatus

```
{"access_code":"680BB248B25339BAD86A4","hex_identifier":""}
```



Pero como es generada la firma?

```
assert len(secret) == 6 and len(access_code) == 21

def get_firma(comments = b'', command = b''):
    s = bytes(secret, "utf-8")
    a = bytes(access_code, "utf-8")
    return hashlib.new("md5", b"".join([s, a, comments, command])).hexdigest()
```

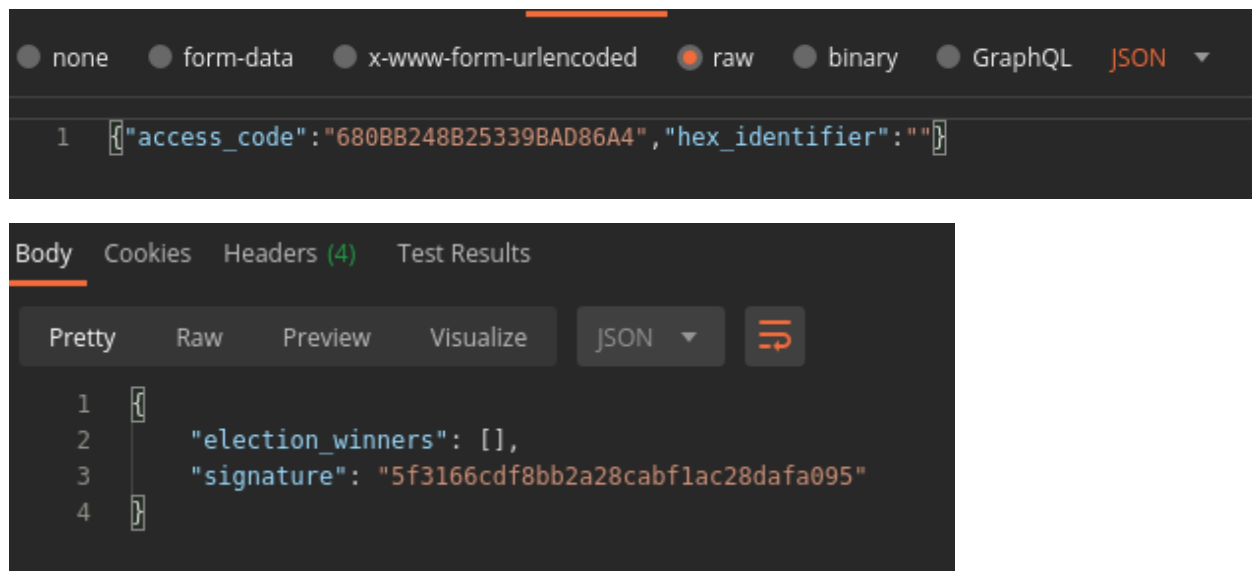
No es mas que la concatenación de una llave secreta, el código de acceso, comments (que hace referencia a hex_identifier, valor que servirá como padding), y command, que es el comando a validar.

Como atacantes, tenemos conocimiento de todos los elementos con excepción de la llave secreta.

La llamada getResult de la API no devuelve el contenido del archivo resultados.db, donde escriben las personas que sean capaces de resolver el reto

```
@app.route("/getResult", methods=['POST'])
def getResult():
    content = request.json
    access = content['access_code']
    comments = content['hex_identifier']
    if access == access_code:
        with open("./resultados.db", "r") as r:
            resultados = [x for x in r.readlines()]
            r.close()
            return jsonify(election_winners = resultados, signature = get_firma(codecs.decode(comments, "hex")))
    else:
        return jsonify(error = "Codigo de acceso incorrecto")
```

http://critical_infrastructure_voting_enforcement_system.hack-defender.mx:3166/getResults



El valor signature para las peticiones anteriores es la suma MD5 generada a partir de la concatenación de los elementos previamente mencionados con la llave secreta. Este valor es sumamente relevante para generar una firma valida, ya que nos permite saber como se genera una firma con tres de los cinco elementos necesarios.

La opción modResults en la API nos permite agregar un valor a la base de datos de ganadores de las elecciones, o borrar dicho archivo, sin embargo esta opción necesita una firma, command_signature, para validar que quien envia la petición es un origen confiable, por ejemplo una aplicación móvil.

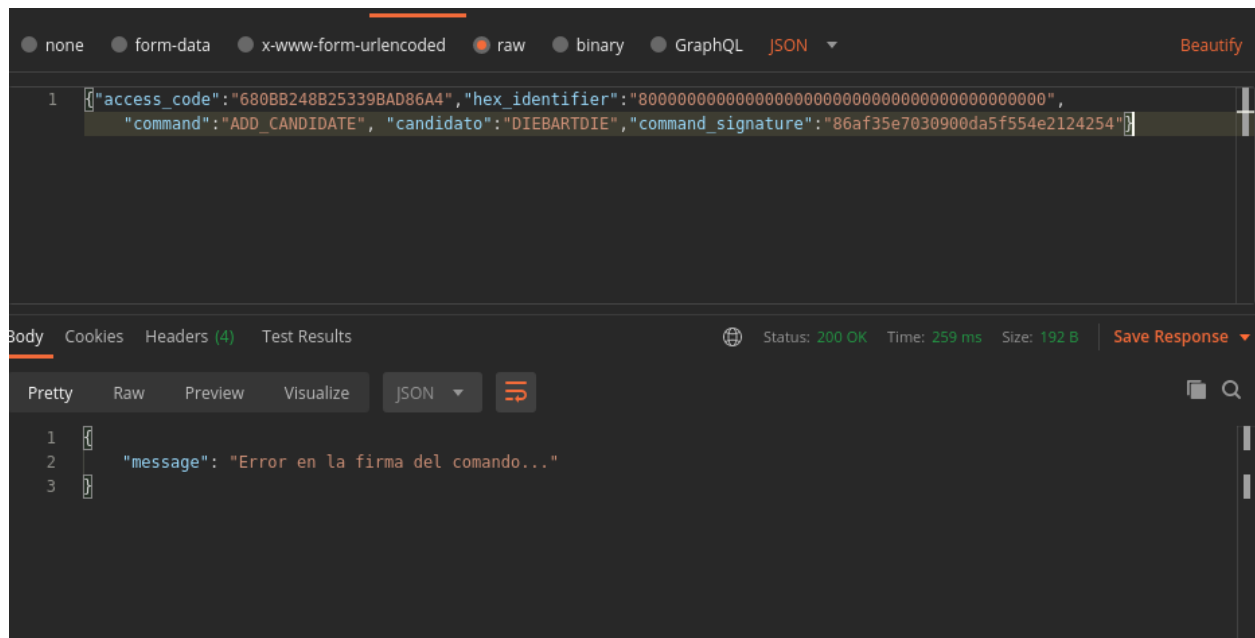
```
authenticate = content["command_signature"]
if access == access_code:
    if authenticate in get_firma(codecs.decode(comments,"hex"), bytes(command,"utf-8")):
        if "ADD_CANDIDATE" in command:
            r = open("./resultados.db","a")
            r.write(candidato + '\n')
            r.close()
            return banner
```

Primero es validado el access_code, enseguida es validada la firma que el usuario provee, contra la firma generada por la petición en la aplicación. De ser valida la firma se escribirá el nombre del candidato en el archivo de ganadores del CTF y se retornará el banner para reclamar la bander.

La siguiente petición es invalida al generar datos erróneos, es decir una firma que no coincide con el valor provisto en command_signature.

http://critical_infrastructure_voting_enforcement_system.hack-defender.mx:3166/modResults

```
{"access_code": "680BB248B25339BAD86A4", "hex_identifier": "80000000000000000000000000000000",
"command": "ADD_CANDIDATE",
"signature": "86af35e7030900da5f554e2124254"}
```



Desconocemos el valor secreto, pero podemos generar un hash valido empleando el padding, por ejemplo, conocemos un hash que se genera con secret + access_code + hex_identifier de getResult y getStatus. MD5 rellena 64 bytes con un padding en caso de que no se complete el bloque con la información brindada por el usuario, así funcionan todos los algoritmos de hashing basados en una construcción Merkle-Damgard. Por ejemplo, la cadena HOLA, internamente tendrá en su bloque de 64 bytes para MD5

HOLA\x80\x00\x00\x00...\x20\x00\x00\x00\x00\x00\x00

Esto, ejemplifica el bloque interno, una vez que se termina la cadena internamente, el algoritmo MD5, rellena el bloque de 0s, comenzando con un 80 en hexadecimal que marcará el final de la cadena y un bloque de los últimos 8 bytes que contienen el tamaño de la cadena contenida en el bloque. Sin embargo es posible en ocasiones, como en este reto manipular el padding, por lo que hacer un hashing de

HOLA

Y

HOLA\x80\x00\x00\x00...\x20\x00\x00\x00\x00\x00\x00

generarían el mismo valor. Esto es importante, ya que en todos los algoritmos de hashing de tipo Merkle Damgard emplean el hash del bloque anterior como valores de inicialización del bloque siguiente. Por lo tanto, los valores de hash generados por la cadena

HOLA\x80\x00\x00\x00...\x20\x00\x00\x00\x00\x00\x00, podrían ayudar a predecir el siguiente estado del hash, conociendo la información que contendría el siguiente bloque...

Existen muchos writeups en línea, sin embargo JSON cambia las cosas y hace que el reto deba ser comprendido antes de usar una herramienta.

https://en.wikipedia.org/wiki/Length_extension_attack

MD5(secret+access_code+hex_identifier+command) sin embargo es poco probable que esta construcción supere los 64 bytes internamente.

MD5(secret+access_code)

MD5(secret+access_code+hex_identifier)

Solo nos queda la duda de el tamaño de secret para poder definir el tamaño del padding y establecer el tamaño de la cadena original en dicho padding.

El assert al principio del código nos da la respuesta.

Siendo d8 el tamaño en bits de llave_secreta + código de acceso, es decir 27. AL enviar las peticiones con y sin padding, se obtiene la misma firma!


```

#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <openssl/md5.h>

int main(int argc, const char *argv[])
{
    int i;
    unsigned char buffer[MD5_DIGEST_LENGTH];
    MD5_CTX c;

    MD5_Init(&c);
    MD5_Update(&c, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA", 64); //Valor arbitrario
    //5f3166cdf8bb2a28cabflac28dafa095
    //Reemplazo de los valores del vector
    c.A = htonl(0x5f3166cd);
    c.B = htonl(0xf8bb2a28);
    c.C = htonl(0xcabflac2);
    c.D = htonl(0x8dafa095);

    MD5_Update(&c, "ADD_CANDIDATE", 13); // NUEVO HASH
    MD5_Final(buffer, &c);
    for (i = 0; i < 16; i++) {
        printf("%02x", buffer[i]);
    }
    printf("\n");
    return 0;
}

```

Obteniendo como resultado un nuevo hash value:

```

f@war-ensemble:~/Documents/hackdef/400$ vim poc.c
f@war-ensemble:~/Documents/hackdef/400$ gcc -o poc poc.c -lssl -lcrypto
f@war-ensemble:~/Documents/hackdef/400$ ./poc
86af35e7030900da5f554e2124254715

```

El cual realiza la firma del comando ADD_CANDIDATE sin la necesidad de conocer la llave privada. Ahora podemos usar el padding y la firma:

