# WRITE-UP – CTF UAM – HISPASEC:

# ELCOCHE FANTASTICO EPISODIO 2

*ELABORADO POR: ARSENICS*

**Misión:**

Con Kitt arreglado, mandaste crear una nueva interfaz web que no fuera vulnerable. El problema es que ahora has salido del coche dejándote las llaves en el interior (/facepalm). Debes encontrar el fallo en la nueva interfaz de Kitt para poderlo recuperar.

http://34.253.120.147:1337

Info: La flag tiene el formato UAM{md5 del string encontrado}

Tools:

*-En este caso no se utilizan tools especiales sino lógica & Skills de programación*

**Búsqueda de información:**

Se entra en la web incial donde aparece una casilla que demanda un código para acceder al control de accesso de KITT.


Se prueba de hacer fuzzing sin éxito pero al inspeccionar el elemento se obtine una pista:

```
<!doctype html>
<html>
▶<head>…</head>
▼<body>
  <h1>KITT 2000 - Control de acceso al vehículo</h1>
  ▼<form method="POST" action="/unlock" onsubmit="document.getElementById('unlockbtn').disabled=true;">
    ▼<p>
      <input id="code" type="text" name="code" size="5/"> == $0
      <input id="unlockbtn" type="submit" name="submit" value="Unlock!">
    </p>
  </form>
  <script type="text/javascript">
    document.getElementById("code").focus();

  </script>
  <!-- hint: /use_the_source_michael -->
 </body>
</html>
```

Si esta ya es la información de la source-view! Poniendo añadiendolo a la URL tenemos la recompensa:

```
# Generar el siguiente código (0-99) y actualizar el estado interno (16bits)
def get_next_code():
    global app

    # Si no hay estado, generamos uno aleatorio
    if not "state" in session:
        session['state']=random.randint(2**15, 2**16)
        session['count']=0

    # Obtenemos el estado actual y lo magreamos un poco
    state = session['state']
    state ^= 0xffff
    state = (state >> 15) ^ (state << 2)

    mask1 = 0x1111
    mask2 = 0x8888
    state ^= (state & mask1) << 3
    state ^= (state & mask1) << 2
    state ^= (state & mask2) >> 2
    state ^= (state & mask2) >> 3

    state &= 0xffff

    # El código es un valor entre 0 y 999
    code = state % 1000

    # Guardamos el estado (no olvidarse de la cookie para los próximos intentos)
    session['state'] = state

    return code
```

Se asigna uin state random junto a la cookie del usuario, se realizan algunos cálculos “magreo” para generar un nuevo state siendo el código de la web el módulo de mil de ese nuevo state.

```
if nextcode == code:
    # Código correcto
    session['count']+=1
    if session['count']==TRIES:
        # Número de aciertos superado. Reiniciamos el estado y mostramos la flag
        session.pop('state',None)
        session.pop('count',None)
        return render_template("flag.html", flag=FLAG)
```

Si se acierta 3 veces se reiniciará el estado y se mostrará la flag. Necesitamos conocer ese state inicial que el server se guarda para realizar los cálculos y obtener el código.

## Fallo de la Interfaz

Si se introduce un número cualquiera la web nos contesta " Código incorrecto se esperaba el 731"

Conociendo que ese 731 es el % 1000 de nuestro state. Acabamos de conocer nuestro state actual. Si se realiza el cálculo que se definde en el código fuente como el "magreo" obtendremos un nuevo state que delatará el código que esta esperando la web.

Problema: Hay 66 casos en los que el % de 1000 da como resultado ese XXX. Pues nada podemos morir con el intento...

Ejemplo si tenemos el número 731:

2731 % 1000 = 731 pero también:

12731 %1000 = 731 & 24731 % 1000 = 731 & 65731 %1000 = 731

Ponemos otro número y al obtener el segundo código de la webvemos que dice que esperaba el 462.

Interesante observación de al arimetica modular, vamos a modificar un poquito ese script para ver cuál de esas opciones es la que nos ofrece el código 462.

```python
import time
code0 = 731


def get_next_code(initial_state):

        # Obtenemos el estado actual y lo magreamos un poco
        state = initial_state
        state ^= 65535
        state = (state >> 15) ^ (state << 2)

        mask1 = 4369
        mask2 = 34952
        state ^= (state & mask1) << 3
        state ^= (state & mask1) << 2
        state ^= (state & mask2) >> 2
        state ^= (state & mask2) >> 3

        state &= 65535

        # El código es un valor entre 0 y 999
        codef = state % 1000

        print(" initial_state: " + str(initial_state))
        print("Codef: " + str(codef))

for n in range(66):

    initial_state = (n * 1000 + code0)
    get_next_code(initial_state)
    time.sleep(1)
```

Teniendo en cuenta los siguiéntes parámetros del script:

Code0 = código inicial q nos dió la web = 731

Initial state = una de las 66 opciones posibles del % 1000 de 731

Codef = el código final al que necesitamos llegar que es el siguiente que nos dice que espera 462.

Con esto vemos que el 462 fué generado por el 26731:

```
 initial_state: 26731
Codef: 462
```

Bien, ahora conocemos nuestro state inicial, ya no será random porque vimos en el source code que se lo guarda y magrea. Si el initial state es el 26731 cuál es el state que genera el 462 ¿??

Cálculemos introduciendo el 26731:

```
def get_next_code():

        # Obtenemos el estado actual y lo magreamos un poco
        state = 26731
        state ^= 65535
        state = (state >> 15) ^ (state << 2)

        mask1 = 4369
        mask2 = 34952
        state ^= (state & mask1) << 3
        state ^= (state & mask1) << 2
        state ^= (state & mask2) >> 2
        state ^= (state & mask2) >> 3

        state &= 65535

        # El código es un valor entre 0 y 999
        codef = state % 1000

        print(" state: " + str(state))
        print("Codef: " + str(codef))

get_next_code()
```

```
root@kali:~/HTB# python3 getluck.py
 state: 44462
Codef: 462
```

Cálculemos el magreo del 44462 para conocer el siguiente state & codef

```python
def get_next_code():

    # Obtenemos el estado actual y lo magreamos un poco
    state = 44462
    state ^= 0xffff
    state = (state >> 15) ^ (state << 2)

    mask1 = 0x1111
    mask2 = 0x8888
    state ^= (state & mask1) << 3
    state ^= (state & mask1) << 2
    state ^= (state & mask2) >> 2
    state ^= (state & mask2) >> 3

    state &= 0xffff

    # El código es un valor entre 0 y 999
    code = state % 1000

    print(str(state))
    print(str(code))

get_next_code()
```

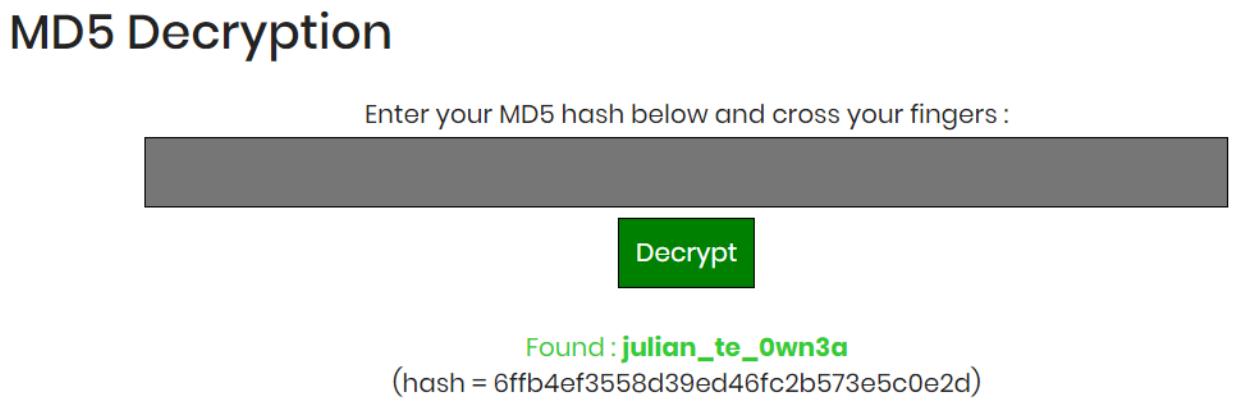
```
root@kali:~/HTB# ./finally.py
17732
732
```

El primer número es el state actual y el segundo el código. Lo introducimos en la web y voilà:


Por fin !!!! pues esto está ya cerca de terminar. Entendiendo la vulnerabilidad sólo tenemos que cálcular el nuevo state e introducir el codef en la web tres veces más

**Quick win:**

Decodeamos el md5 de la flag




Parece que Julian se ha divertido preparando la prueba, lo que no sabe es que yo me he divertido más y por el camino he aprendido cosas nuevas.

Flag: UAM{6ffb4ef3558d39ed46fc2b573e5c0e2d}

***Autoría: Arsenics***