Universidad de Buenos Aires
Facultades de Ciencias Económicas, Ciencias Exactas y Naturales e Ingeniería

Maestría en Seguridad Informática
Trabajo Final

“Seguridad en aplicaciones móviles: Defensa y ataque a nivel de aplicación y comunicación con Servicios Web”

Autor:

David Arteaga Grigoriev

Director:
Dr. Pedro Hecht

Buenos Aires, Febrero 2021

Cohorte 2017

[Página intencionalmente en blanco]

Declaración Jurada de origen de los contenidos

Por medio de la presente, el autor manifiesta conocer y aceptar el Reglamento de Trabajos Finales vigente y se hace responsable que la totalidad de los contenidos del presente documento son originales y de su creación exclusiva, o bien pertenecen a terceros u otras fuentes, que han sido adecuadamente referenciados y cuya inclusión no infringe la legislación Nacional e Internacional de Propiedad Intelectual.

Firmado

David Arteaga Grigoriev

Licencia

Tabla de Contenido

1. Introducción

El desarrollo de aplicaciones móviles va en constante crecimiento, donde se ha convertido en una tendencia el uso de dispositivos portables para realizar un gran número de tareas relacionadas a trabajo, entretenimiento, financieras, entre otras; es por esto que nace la necesidad de proteger la información almacenada y procesada por las aplicaciones y sus servicios. La gran mayoría de aplicaciones se comunican con un servidor o servicio web (*WS*), con el cual tienen un flujo de información constante de datos para el procesamiento y almacenamiento de los mismos. Teniendo esto en mente, este trabajo final de Maestría en Seguridad Informática desarrollado, tiene como enfoque mostrar aspectos de seguridad vinculados en los procesos de desarrollo de aplicaciones móviles y el consumo de los *WS* expuestos para el servicio de las estas.

Se tomará como foco tres vectores de ataque, y por medio de pruebas de concepto, se dará evidencia de diferentes amenazas y se plantean contramedidas correspondientes para tener en cuenta como requisitos básicos de seguridad para el desarrollo de aplicaciones móviles que mitiguen las amenazas a mostrar.

El primer vector se centra en la protección de las solicitudes hechas por la aplicación hacia un servidor que expone servicios que sean consumidos por la misma aplicación, así estas viajen por canales seguros de comunicación (*SSL/TLS*), la protección de este vector se hará mediante el método de *SSL Pinning*. La prueba de concepto para este caso dejará evidencia de cómo es posible interceptar solicitudes de aplicaciones que carecen de esta medida de protección, por qué esto representa una amenaza y el cómo se pueden visualizar las solicitudes de la aplicación hacia el *WS*. Una de las maneras con las cuales un usuario malintencionado puede capturar las solicitudes de una

aplicación, así se encuentren protegidas por canales seguros, es instalando certificados firmados por su propia Autoridad Certificadora (CA) y configurando un proxy que captura las solicitudes enviadas por la aplicación y haga uso de ese certificado. Al estar confiando en el certificado del proxy es posible capturar las solicitudes enviadas dejando expuestos los parámetros, encabezados, URLs y tipos de solicitudes enviadas al servicio que consume la aplicación [1]. Una de las diferentes maneras para evitar este tipo de amenazas es a través de la implementación de *SSL Pinning* en la aplicación, con la cual se plantea una validación de certificados del *WS* previamente instalados dentro de la aplicación antes de realizar una consulta con dicho *WS* [2].

El segundo vector a tratar en este documento se centra en el código en sí y su protección. Una de las maneras más comunes en que los atacantes o usuarios malintencionados obtienen información de las funciones críticas de una aplicación es interpretando su código. El método de ingeniería inversa es un proceso que a partir de la solución final de una aplicación, para el caso de aplicaciones móviles *APK* e *IPA*, se puede llegar a ver el código de la aplicación con el que fue construida. Se mostrará mediante métodos de ingeniería inversa la importancia de la ofuscación de código en aplicaciones móviles, por qué esto representa una amenaza y formas básicas de ofuscación de código para mitigar esta amenaza. Con este vector se dejará evidencia de cómo un atacante vé el código haciendo ingeniería inversa de una aplicación con código ofuscado y una que no; se comparan los resultados con el código original para poder sacar conclusiones concretas.

Por último y teniendo en cuenta los análisis y resultados que se mostrarán en los vectores anteriores, se dejará en evidencia cómo, mediante métodos de cifrado dentro de las funciones críticas dentro de una aplicación puede, se puede dar una capa de protección extra para la información enviada por los canales de comunicación entre un *WS* y la aplicación móvil. Se piensa usar métodos de

criptografía simétrica para asegurar el intercambio de datos que tendrán las peticiones de la aplicación a los parámetros de las funciones expuestas por el *WS*. Gracias a esto se muestra la importancia del uso de métodos de cifrado de datos para un intercambio íntegro de información dentro de las aplicación en general.

2. Alcance y objetivos

El objetivo general de este documento es brindar de forma detallada, documentación, aportes prácticos mediante pruebas de concepto y teniendo como base los vectores previamente descritos, una guía de aseguramiento básico de aplicaciones móviles que tengan una interacción con un *WS*. Cada vector de ataque representa una amenaza a la integridad de datos que se maneja en la aplicación y en la comunicación con su *WS*. Del mismo modo, cada punto expuesto puede afectar la confidencialidad de la información del usuario y del servicio que hace uso la aplicación. Dicho esto, el alcance del documento se centrará en brindar mecanismos de mitigación de estas amenazas presentes en los vectores de ataque expuestos que son comunes para aplicaciones de hoy en día.

Los objetivos particulares fueron definidos de la siguiente manera:

2.1. Brindar aportes prácticos en el ámbito de la seguridad móvil con pruebas de concepto, entregando repositorios de uso libre que sirvan como punto de partida para un aseguramiento de aplicaciones que hagan uso y procesamiento de información mediante un *WS*.

2.1.1. Desarrollo de *WS* REST para uso en aplicación móvil.

2.1.2. Desarrollo de aplicación móvil Android básica con medidas de aseguramiento mostradas en cada vector propuesto.

2.1.2.1. Mostrar la diferencia de una aplicación con protección de SSL Pinning y una que no con funciones que hagan uso de los recursos del *WS* desarrollado.

2.1.2.2. Mostrar la diferencia de una aplicación con código ofuscado y una que carece de ello mediante métodos de ingeniería inversa. Para esto se crearán dos versiones de aplicación Android con las mismas funciones y uso de recursos del *WS*. Una versión tendrá configuraciones básicas de ofuscamiento, reducción y optimización de código con ProGuard y otra sin estas configuraciones.

2.1.2.3. Mediante el uso de métodos simétricos de cifrado, enviar y procesar la información desde la aplicación al *WS* y viceversa. Mostrar como brinda esto una capa extra de seguridad en el canal de comunicación a nivel de aplicación.

2.2. Integrar y exponer conceptos de seguridad informática en los vectores expuestos como Autoridad Certificadora (CA), ofuscamiento de código, SSL Pinning, cifrado de datos en aplicaciones móviles Android y las amenazas a tener en cuenta en cada uno de estos temas y mostrar mediante una manera práctica la mitigación de los mismos.

2.3. Exponer las conclusiones, puntos de dolor y comparaciones encontradas durante el diseño e implementación de cada prueba de concepto a desarrollar.

3. Primer Vector: Protección ante exposiciones de solicitudes hechas desde la aplicación hacia sus servicios.

3.1. Desarrollo Teórico

Dentro de los vectores más importantes dentro de las aplicaciones móviles de hoy en día está el poder proteger los servicios que la aplicación misma consume, esto con el fin de mitigar riesgos dentro de las funcionalidades que no se llevan dentro de la aplicación y son procesadas por servicios expuestos para el consumo de la misma. Si un atacante puede encontrar hacia qué servicios apunta la aplicación, tiene un vector de ataque con el que puede llegar a afectar tanto el negocio como las funcionalidades inherentes de la aplicación.

De manera ejemplificada se puede tener una aplicación registro de transacciones bancarias, donde dichas transacciones son asociadas para cada usuario con un identificador único. Las transacciones pueden ser albergadas en un servicio externo y la aplicación consulta las transacciones asociadas a un usuario mediante el identificador único en un *servicio web* o *web service* (*WS*) que recibe como parámetro desde la aplicación móvil y devuelve las transacciones asociadas a ese usuario. Si un atacante o usuario malintencionado captura la solicitud y logra entender esta funcionalidad crítica de la aplicación y su servicio, existe el riesgo de que pueda consultar transacciones de otros usuarios alterando las consultas hechas hacia el *WS* cambiando el identificador por ejemplo.

Esto se puede catalogar como una explotación de un riesgo de control de acceso que afecta el desempeño de una función comercial fuera de los límites de un usuario de la aplicación. Una de las maneras de mitigar este riesgo es mediante la implementación de *SSL Pinning* en las aplicaciones móviles.

Relacionado a *SSL Pinning*, OWASP define: “Los usuarios y desarrolladores esperan seguridad de extremo a extremo cuando envían y reciben datos en sus aplicaciones, especialmente datos confidenciales en canales protegidos por VPN, SSL o TLS. Si bien las organizaciones que controlan DNS (Sistema de nombres de dominio) y CA (Autoridad de certificación) han reducido el riesgo a niveles triviales en la mayoría de los modelos de amenazas, los usuarios y desarrolladores sujetos al DNS de otros y a una jerarquía de CA pública están expuestos a cantidades triviales de riesgo "[1]. En el caso de las aplicaciones móviles, una de las prioridades que deben tenerse en cuenta es que las llamadas o solicitudes realizadas a los servicios que consume la aplicación no deben ser interceptadas. Como se contextualizo en el ejemplo anterior, si un usuario, o en el caso específico de un usuario malintencionado, puede capturar las solicitudes de la aplicación al back-end / servicio y los parámetros que le pasa la solicitud, el atacante puede manipular estas solicitudes realizadas, ya sea manipulando los parámetros enviados, los encabezados, el tipo de solicitud HTTP que realiza, pasando parámetros de diferentes tipos a los esperados que se validan en la capa de front-end / aplicación y no en la capa de back-end / servicio, entre otros.

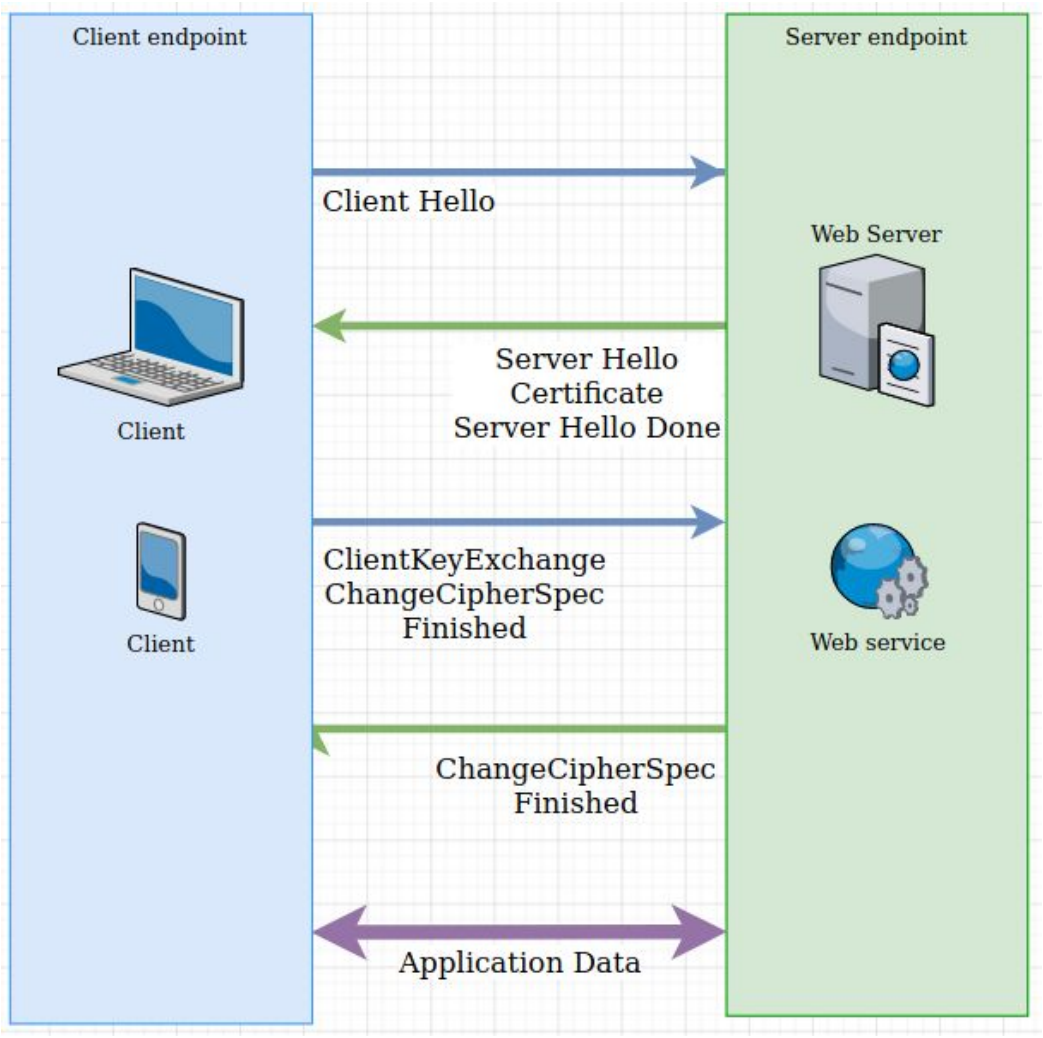

Imagen 1: Protocolo SSL o TLS de intercambio de certificados alto nivel

Una de las formas en que un usuario malintencionado puede capturar las solicitudes de una aplicación, incluso si están protegidas por canales seguros, es instalando certificados firmados por su propia CA y configurando un proxy que utilizando ese certificado para capturar las solicitudes enviadas por la aplicación. Al confiar en el certificado proxy, es posible capturar las solicitudes incluso si pasan por un canal seguro, y exponer parámetros, encabezados, URL y tipos de solicitudes enviadas al servicio consumido por la aplicación. Una forma de evitar este tipo de

amenazas no triviales es a través de la implementación de *SSL Pinning* en la aplicación [2].

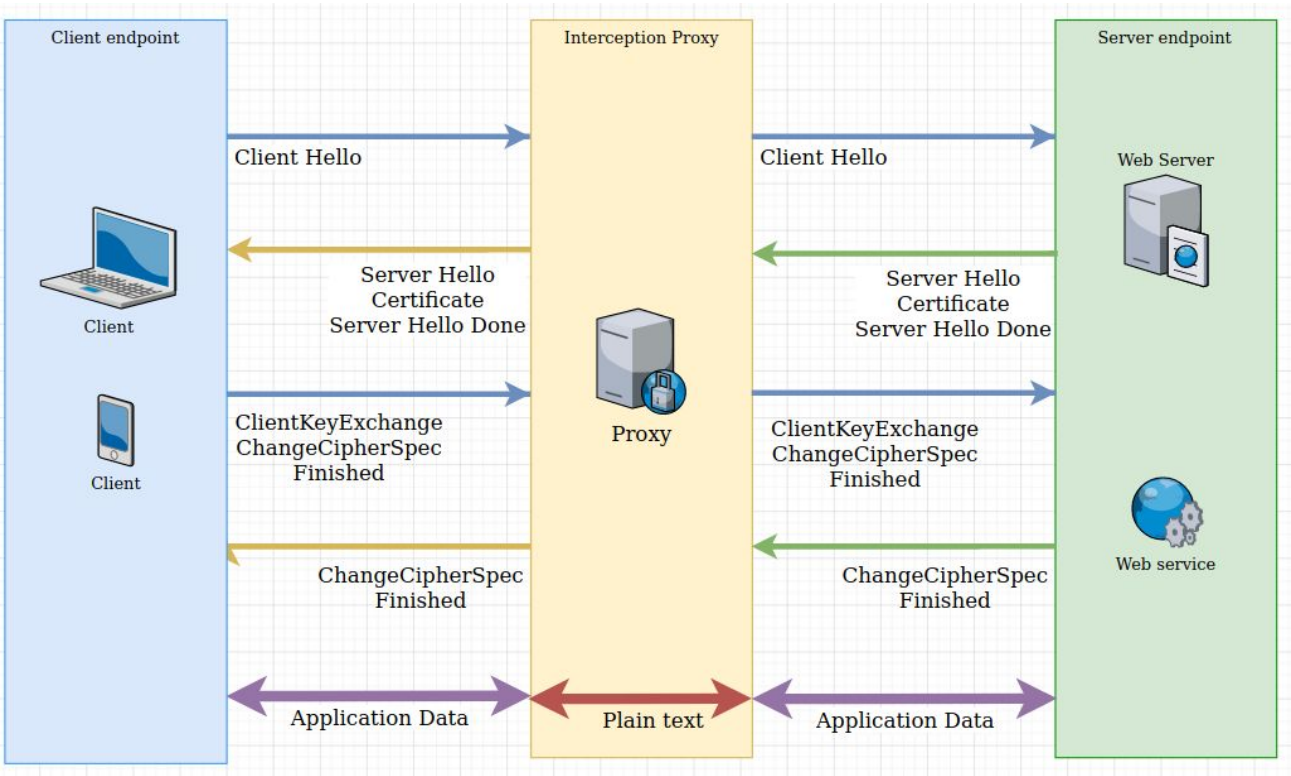

Imagen 2: Protocolo SSL de intercambio de certificados con proxy de intercepción a alto nivel

Dicho esto, *SSL Pinning* toma el certificado o hash de clave pública de un host o servicio; que puede agregarse a una aplicación en el momento del desarrollo y compararse con el servicio publicado cada vez que la aplicación envía una solicitud. También se puede agregar en el primer *handshake* entre la aplicación y el servicio. Es preferible elegir el primer método, ya que precargar el certificado o la clave pública fuera de banda generalmente significa que el atacante no puede detectar el pin. Si se agrega el certificado o la clave pública en el primer encuentro, podría estar fijando el certificado del atacante [2].

El *SSL Pinning* aprovecha la relación existente entre el usuario y una organización o servicio para ayudar a tomar mejores decisiones relacionadas con la seguridad. Como ya tiene información sobre el servidor o servicio, no necesita confiar en mecanismos generalizados para resolver el problema de distribución de claves. Es decir, no necesita ir a un DNS para nombre / dirección o asignaciones de CA para enlaces y estado [7,8].

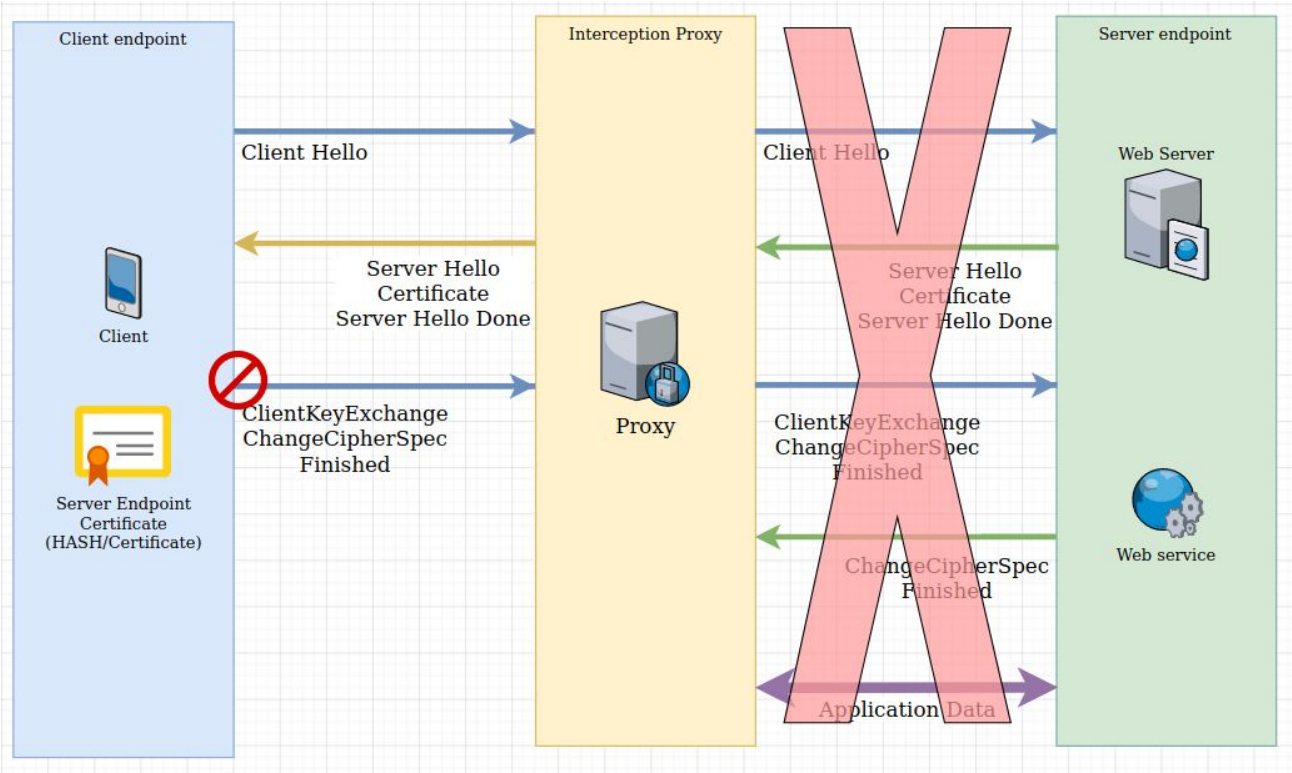

Imagen 3: Protocolo SSL de intercambio de certificados con proxy de intercepción y medida SSL Pinning

Es por esto que cada vez que la aplicación envía una solicitud al host o servicio, compara el bien sea el certificado instalado en la aplicación con el recibido del servicio con el cual se quiere comunicar y enviar la solicitud, si los certificados coinciden significa que es el endpoint con el

cual la aplicación se espera conectar y pueden empezar una conexión segura entre ellos ya que se tiene un conocimiento mutuo de los certificados válidos, de lo contrario, la aplicación no enviará los datos y informar al usuario de un error de conexión.

3.2. Desarrollo Práctico

La siguiente Prueba de concepto (PoC) es un ejemplo sobre la importancia del *SSL pinning* para la protección de exposición de solicitudes hechas desde una aplicación móvil hacia sus servicios. El ejemplo captura una solicitud con un formulario de inicio de sesión y expone las credenciales en texto plano y las modifica a través de una herramienta proxy. El segundo ejemplo incluye una implementación de *SSL Pinning* que evita la intercepción del formulario de inicio de sesión. Para esta PoC se utilizó Burp Suite Community Edition para capturar la solicitud de la aplicación con su función de proxy, el cual se configuró según la topología propuesta en la *imagen 1* para realizar las capturas. La aplicación móvil fue nombrada ***NAFAA*** y desarrollada para Android usando la versión de SDK 29 en el lenguaje JAVA con el IDE de Android Studio versión 4.1. Para el backend, se desarrolló un *WS* tipo *REST* en Python3 y fue nombrada ***NAFWS***.

3.2.1. Desarrollo del Web Service (*NAFWS*)

El *WS* fue desarrollado en Python3 con la librería de Flask [6], que permiten una rápida implementación fácil despliegue de servicios tipo

REST. Para esta *PoC* se desarrolló un servicio HTTP POST, el cual recibe como parámetros dos cadenas de caracteres, la primera es el nombre de usuario definida como “username” y la segunda es la contraseña definida como el parámetro “password”. Ambas son enviadas como información en la solicitud del método POST en el formato JSON. Para este caso en específico, y debido al alcance de la prueba, el *WS* no consta de una base de datos por lo que las credenciales se han dejado embebidas en texto plano en el servicio (no en la aplicación móvil), es decir que la aplicación consume este servicio enviando las credenciales con los parámetros definidos en un formato JSON, estos parámetros son evaluados en el servicio y si las credenciales coinciden con las embebidas en el código devuelve un estado 200 que indica que las credenciales son correctas, caso contrario devuelve un código 401 para unas credenciales incorrectas y un código 400 para casos inesperados.
El link de descarga del *WS* de este proyecto se encuentra en la sección de repositorios.

```
25  @app.route('/login', methods=['POST'])
26  def do_admin_login():
27      if request.is_json:
28          requestJson = request.get_json()
29          print("-----------------Request to login made --> ", request.get_json())
30          if requestJson['password'] == 'password' and requestJson['username'] == 'admin':
31              session['logged_in'] = True
32              return home()
33          else:
34              print("Incorrect Credentials")
35              abort(401)
36      else:
37          print("No JSON?", request)
38          abort(400)
39
```

Imagen 4: NAFWS: Metodo POST WS login

Para la aplicación que expone servicios o ***NAFWS*****,** se configuró un certificado autofirmado generado con la herramienta de OpenSSL con un periodo de duración de 365 días usando el algoritmo RSA con una llave de 4096 bits [6].

```
openssl req -x509 -newkey rsa:4096 -nodes -out
test_cert.pem -keyout test_key.pem -days 365
```

Esta parte es esencial en la configuración del *WS* ya que del certificado se obtiene el hash el cual se va a configurar en la aplicación para hacer la validación de SSL pinning.

Imagen 5: NAFWS: Configuración certificados WS

3.2.2. Desarrollo de aplicación móvil (*NAFAA*)

Para la aplicación movil se uso la libreria de Okhttp3 [3] como cliente de manejo de solicitudes, esta librería tiene funciones propias de validación de certificados para certificados CAs de entidades confiables, pero al usar un certificado propio en el *WS* la validación se hace igualmente por Okhttp3 pero teniendo en cuenta un manejador de certificados de confianza, el cual obtiene el certificado y se compara con el hash del certificado previamente configurado antes de enviar los parámetros y el cuerpo de la solicitud como se haría normalmente en en la librería para certificados de CAs confiables. El link de descarga del proyecto se encuentra en la sección de repositorios.

Para poder mostrar de una mejor manera la validación de certificados se elaboraron dos clientes que realizan peticiones en la clase Utils.java del proyecto. Ambos clientes usan la librería Okhttp3, con la diferencia de que en uno de ellos no usa la funcionalidades de la clase “CertificatePinner” al momento de su creación. Esta clase es la que permite añadir el hash del certificado esperado, es decir el hash que se genera a partir del certificado del *WS*, que será evaluado posteriormente antes de realizar las solicitudes desde la aplicación móvil hacia el endpoint o *WS* que desea consumir.

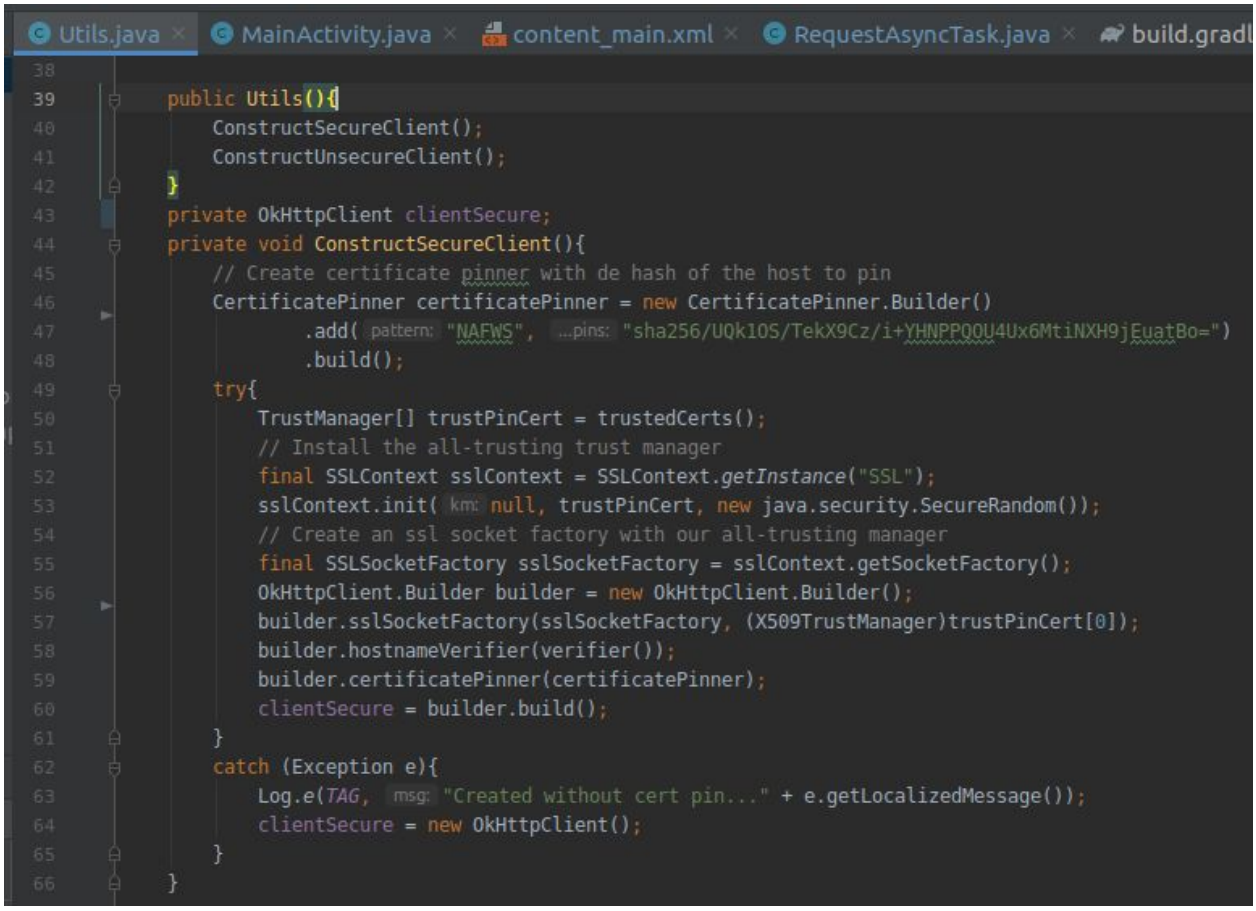

Imagen 6: NAFAA: Okhttp3 construción del cliente seguro y pinning del certificado

Como se puede ver en la imagen anterior, cuando se define el cliente seguro para hacer las peticiones *HTTP* hacia el *WS* se define un objeto “CertificatePinner” en el cual se añade el hash del certificado público como se explicó en la sección 3.1, este se compara cuando se quiera realizar una petición hacia el *WS*, es decir, antes de enviar los parámetros, la aplicación recibe el certificado del *WS* hacia el cual desea realizar la petición, genera el hash del certificado con el certificado público recibido y lo compara con el que se ha puesto en el código previamente en el objeto CertificatePinner. Si los hashes coinciden procede a enviar la solicitud que se desea hacer desde la aplicación con los parámetros que desea enviar, garantizando así que la comunicación entre aplicación y

WS es la que se espera conociendo el hash que debería generar el certificado esperado.

Para el cliente inseguro no se define el objeto “CertificatePinner” por lo tanto no se añade ningún hash de certificado ya que no hará uso de esta funcionalidad.

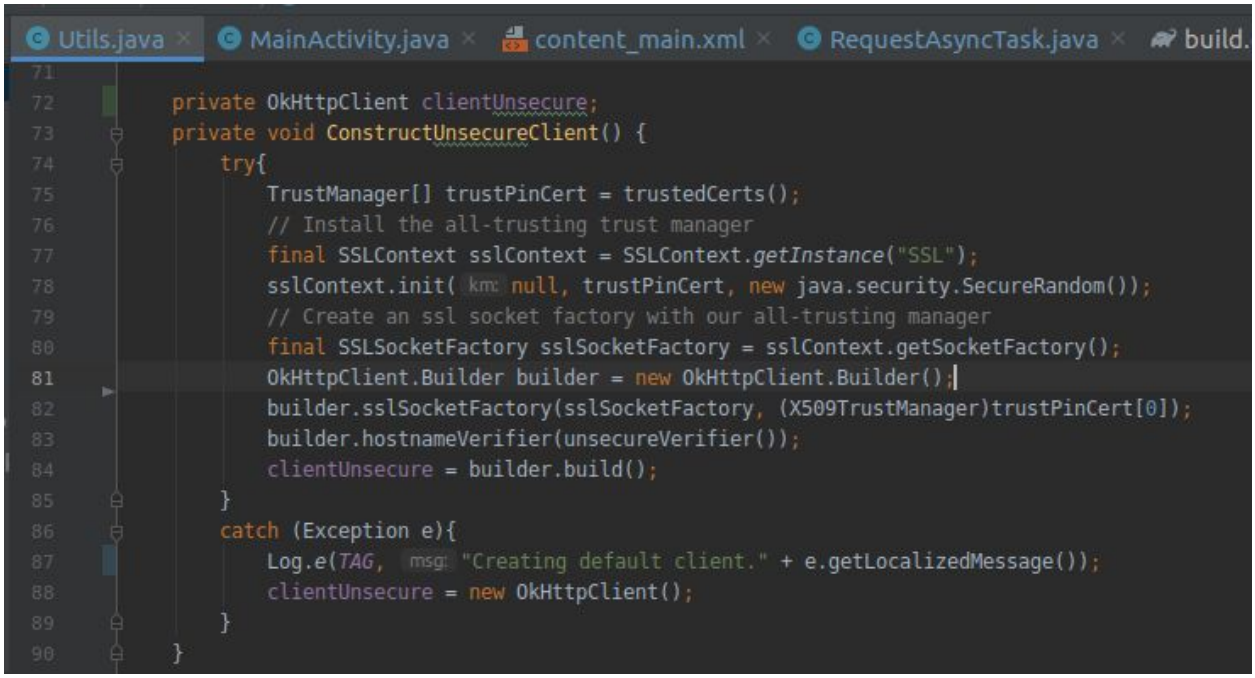

Imagen 7: NAFAA: Okhttp3 construción del cliente sin funcionalidad de ssl pinning

Para la parte gráfica de la aplicación, se definieron dos campos de edición que reciben los parámetros “username” y “password” en los clientes, los cuales son esperados por el *WS* en su servicio POST de login como se explicó en la sección anterior. Además, se agregaron dos botones con los cuales se maneja el evento de envío de solicitudes. Para el botón “Secure Login” se usa el cliente con la función de *SSL Pinning* y caso contrario para el botón “Unsecure Login” que no realiza la validación de certificados al enviar solicitudes.

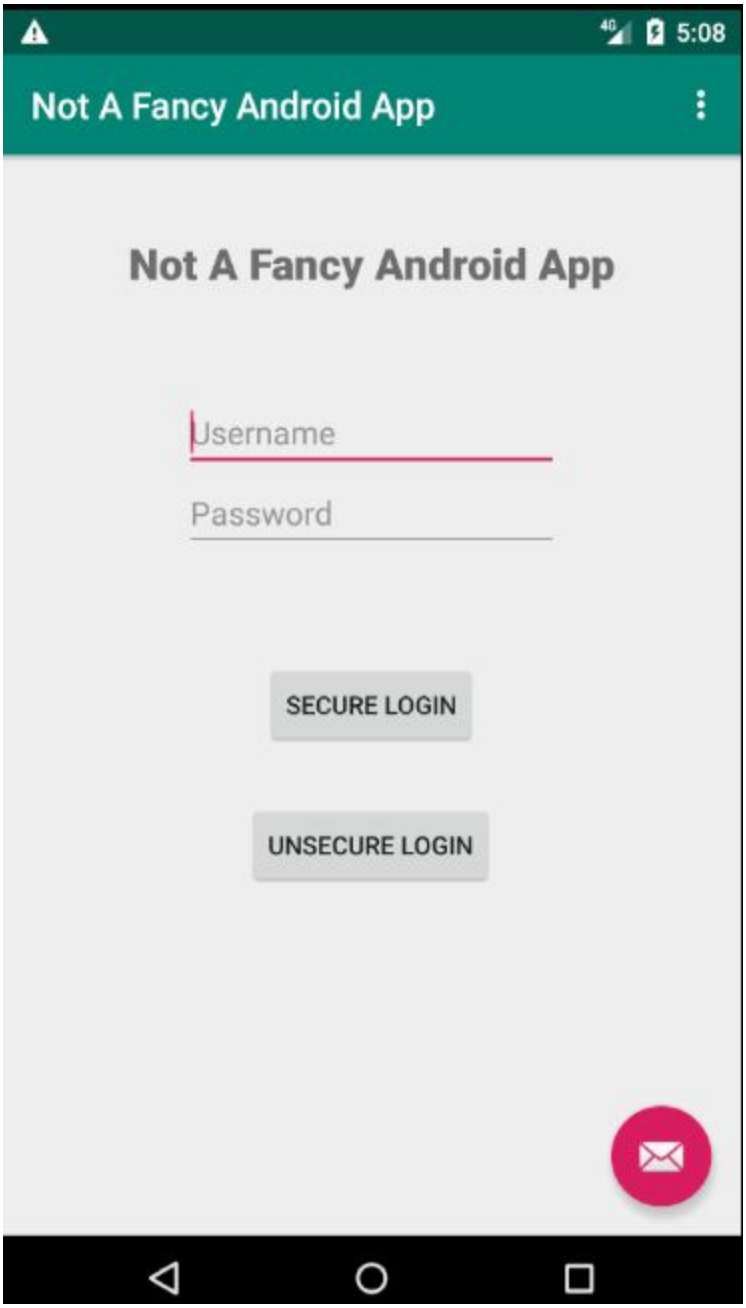

Imagen 8: NAFAA: Parte gráfica login aplicación móvil

Por último se agregó una pantalla de bienvenida que se le muestra al usuario después de hacer un login enviando las credenciales correctas. Si las credenciales no son correctas permanece en la vista principal de login donde se le informa el error de validación de credenciales al usuario.

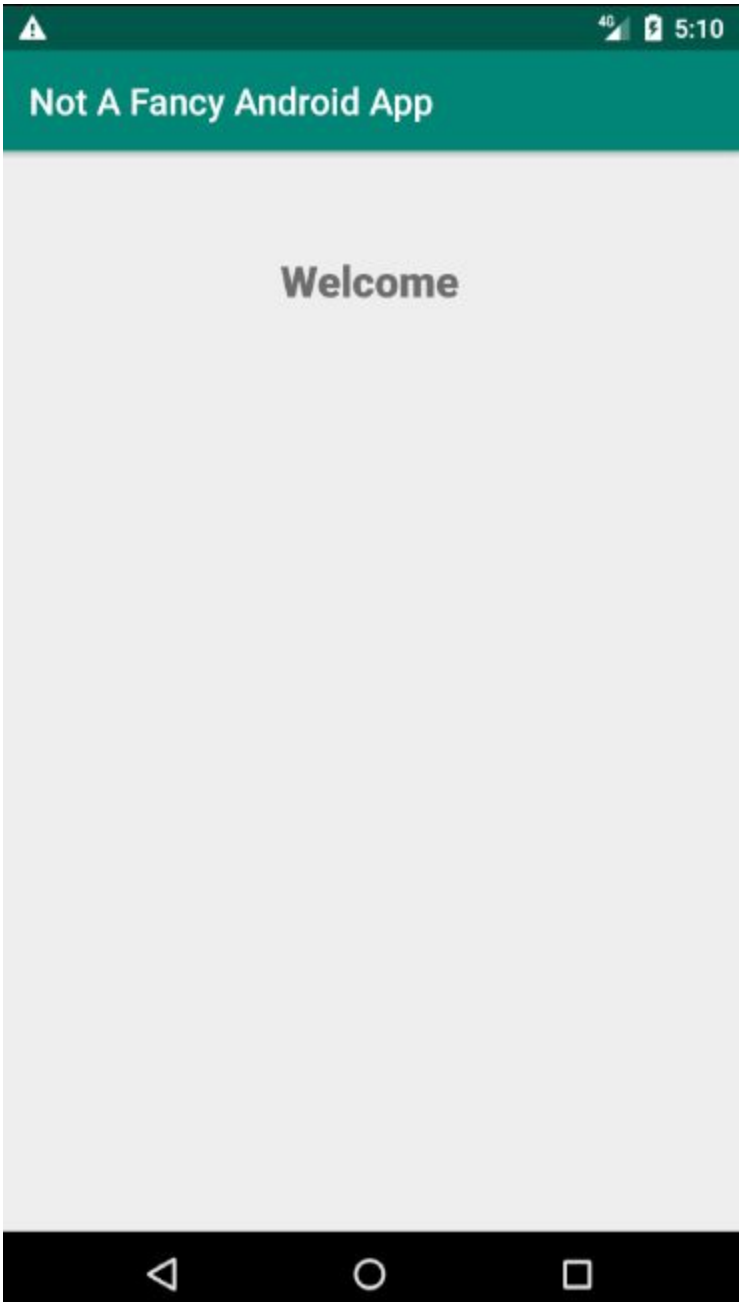

Imagen 9: NAFAA: Parte gráfica profile aplicación móvil

3.2.3. Pruebas funcionales

3.2.3.1. Proxy desactivado

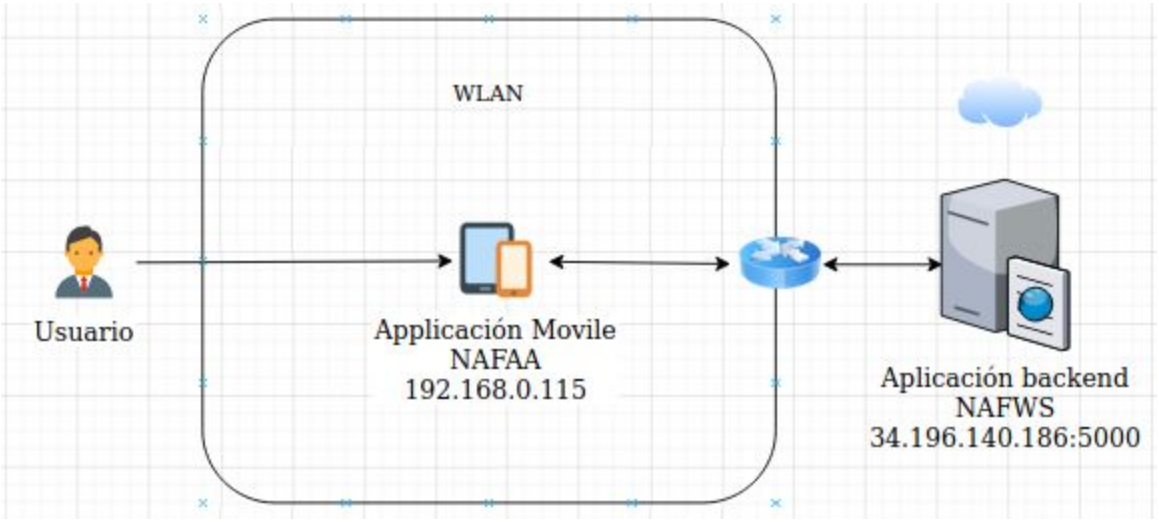

Imagen 10: Topología proxy desactivado

En primera instancia, se corrobora que la aplicación funcione de manera correcta, se desplegó el *WS* el cual está listo para recibir peticiones en el servicio expuesto de “login” como se explicó en la sección 3.2.1 y la aplicación móvil con la configuración del hash del certificado de dicho *WS*. Para esta primera prueba no se configura el proxy y se hace una solicitud directa al endpoint usando ambos clientes implementados con la librería Okhttp3. Para el caso del cliente configurado con el control de *SSL Pinning* se puede ver a través del log de la aplicación el hash del certificado que está “pinneado” y el hash del certificado que recibe del servicio hacia el cual va a realizar la petición. Como no se tiene

configurado el proxy para este escenario los hashes de los certificados concuerdan y es posible hacer una petición con éxito.

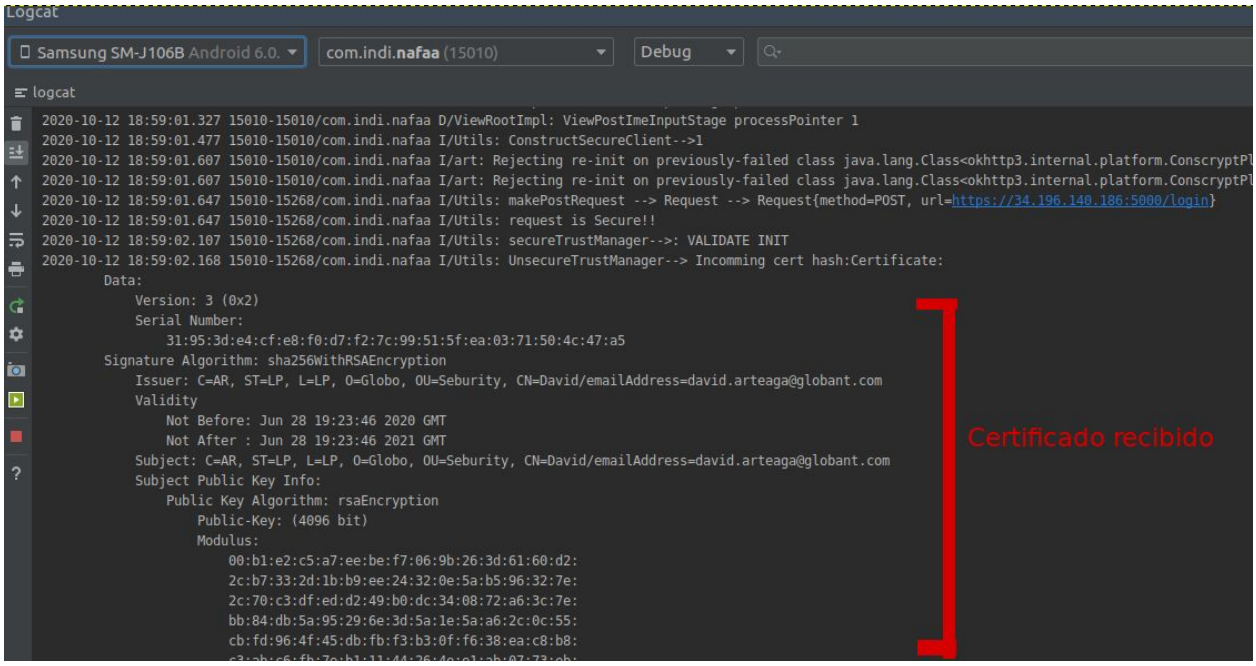

Imagen 11: NAFAA: Prueba funcional proxy desactivado cliente con SSL Pinning, certificado recibido.

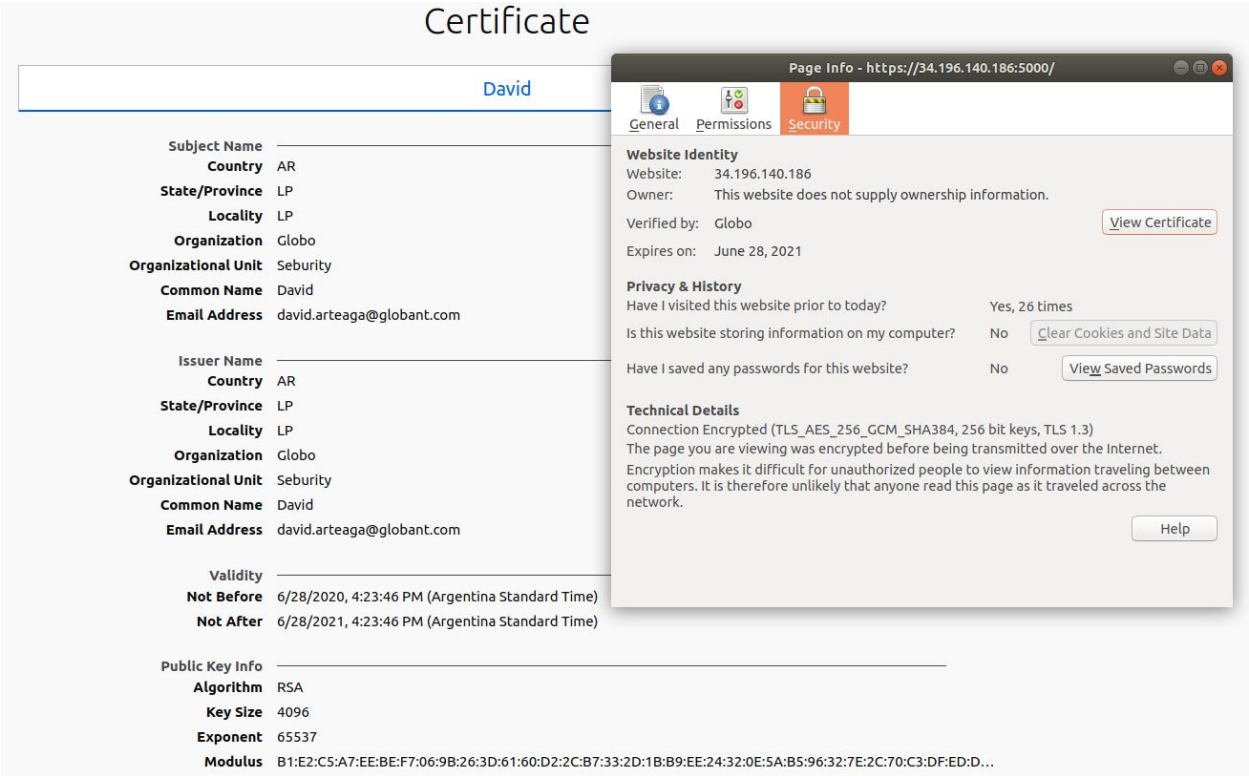

Imagen 12: NAFWS: Certificado configurado en el WS.

Se puede evidenciar que el certificado recibido es el que se tiene configurado en el *WS.* Luego cuando se evalúa el hash del certificado recibido concuerda con el hash que se tiene configurado en la aplicación y realizada esta validación envía los parámetros configurados al *WS* de login.

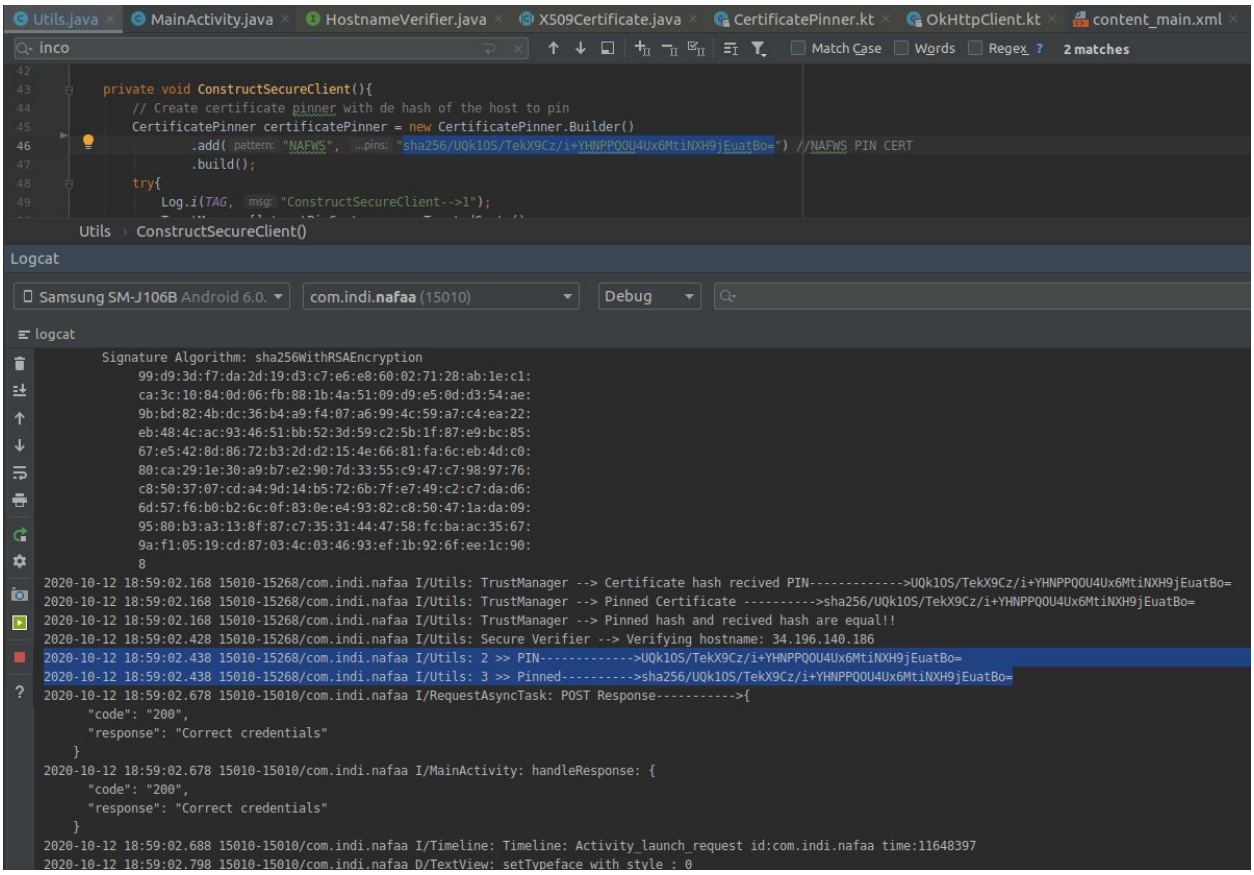

Imagen 13: NAFWS: Prueba funcional proxy desactivado, comparación de hashes.

3.2.3.2. Proxy activado

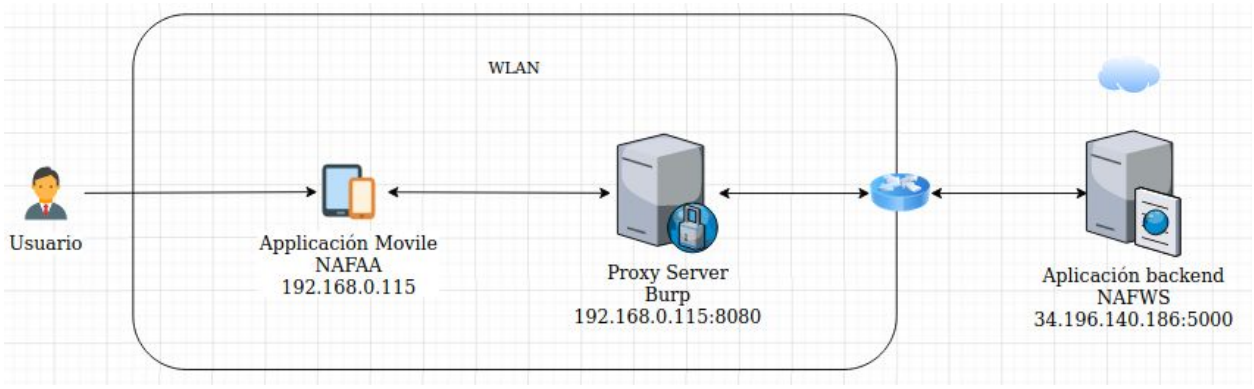

Imagen 14: Topología Primer Vector proxy activado

Para el segundo escenario se despliega el proxy y se configura en el dispositivo móvil según la topología mostrada anteriormente. Para probar la validación de certificados, en primera instancia se realiza una petición a través del cliente implementado de manera segura con el control de *SSL Pinning*. Cómo el certificado que recibe la aplicación será el del proxy el hash no coincide con el hash configurado dentro de la aplicación y no envía los parámetros de la solicitud al *WS*.

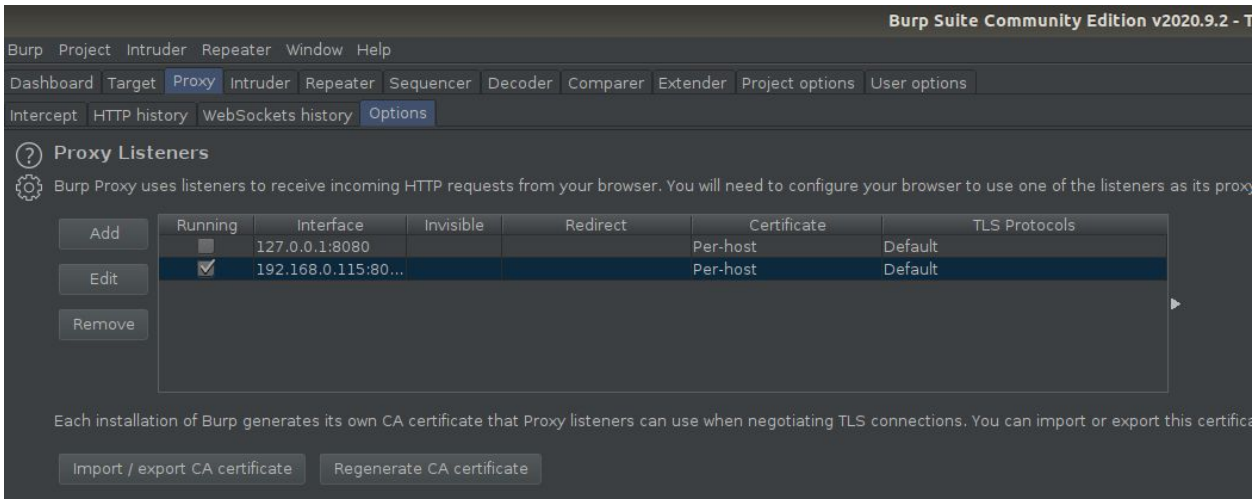

Imagen 15: Configuración Burp proxy.

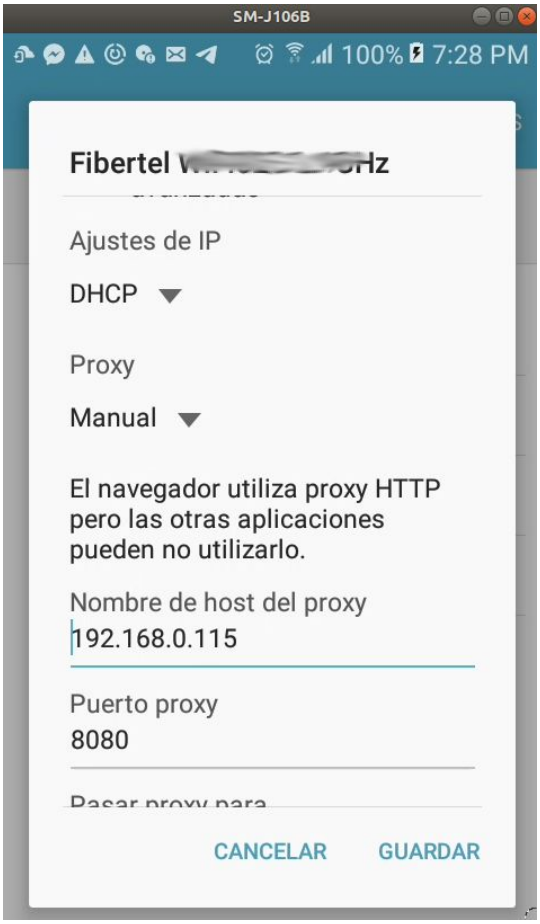

Imagen 16: Configuración proxy dispositivo móvil.

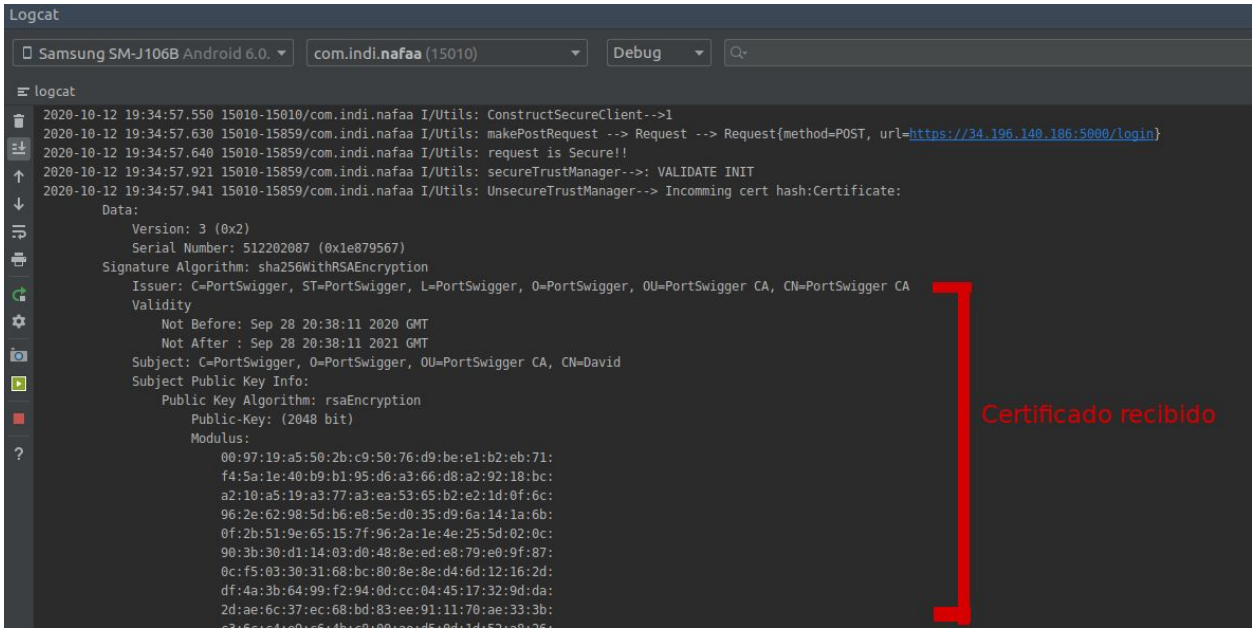

Imagen 17: Prueba funcional proxy activado cliente con SSL Pinning, certificado recibido.

Se puede evidenciar que el certificado recibido no coincide con el que se tiene configurado en el *WS* y se loguea un error de comparación de certificados y NO envía la petición POST junto con los parámetros hacia el *WS* debido a esto. Como la solicitud nunca es enviada hacia el endpoint, debido a este control, el proxy no es capaz de capturar la solicitud ni los parámetros hacia el *WS* que intenta realizar la aplicación.

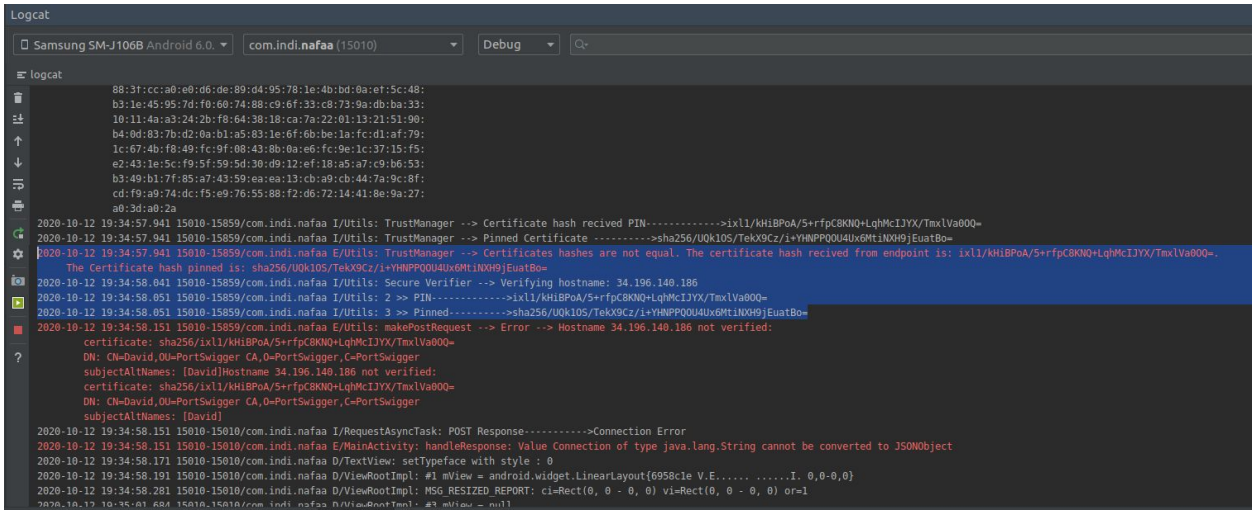

Imagen 18: Prueba funcional proxy activado cliente con SSL Pinning, error validación certificados 1.

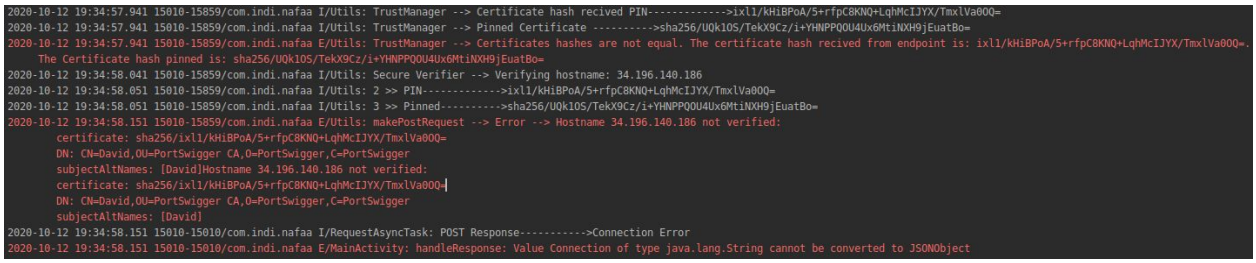

Imagen 19: Prueba funcional proxy activado cliente con SSL Pinning, error validación certificados 2.

Usando el cliente que se configuró sin el control de SSL Pinning, se pueden capturar las solicitudes de la aplicación con el proxy de intercepción integrado en la herramienta Burp Suite. Al no tener que realizar la validación de certificados es posible hacer intercambio de certificados entre el cliente o aplicación móvil (***NAFAA***) y el proxy de intercepción (Burp), de esta manera el proxy es capaz de interceptar la solicitud y se pueden ver los parámetros enviados desde la aplicación al *WS* en texto plano pese a que el *WS* se tiene configurado con un certificado, es decir las peticiones se hacen a un URL con HTTPS. Esto requiere una instalación del certificado dentro del dispositivo móvil.

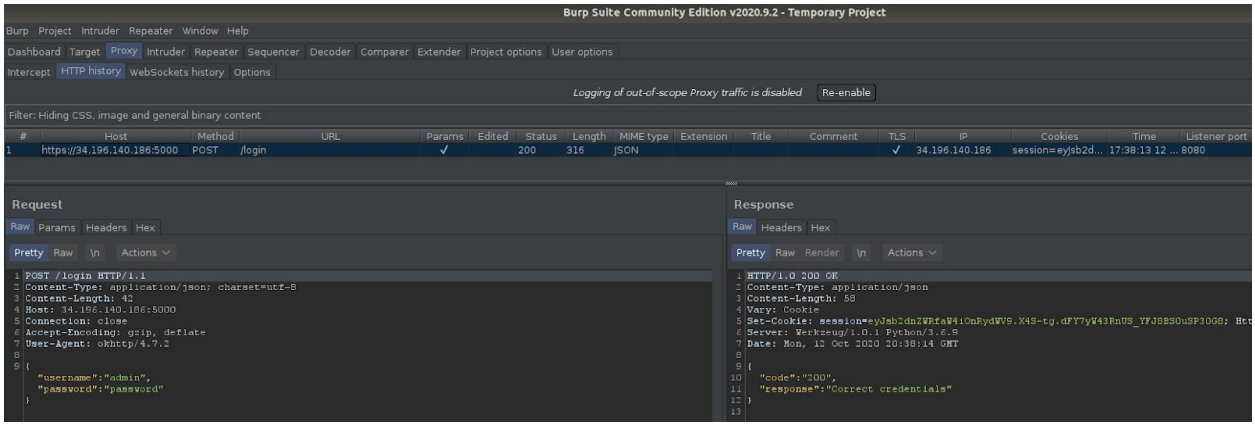

Imagen 20: Intercepción de solicitud burp proxy 1

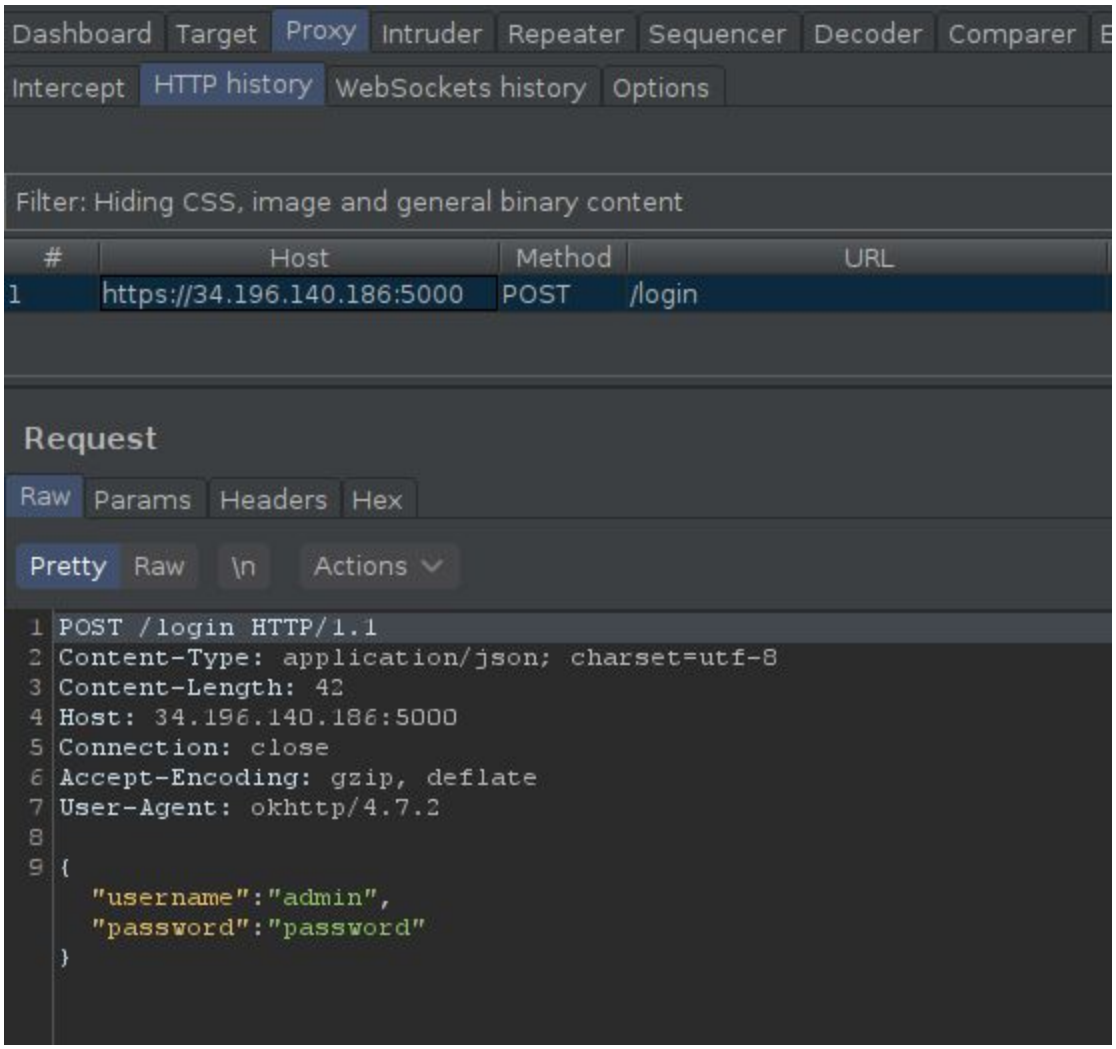

Imagen 21: Intercepción de solicitud burp proxy 2

El riesgo de que las peticiones puedan ser capturadas junto con los parámetros que se envían y hacia el endpoint, es que permite a un atacante la automatización de ataques, para este caso específico de login, el atacante podría automatizar el proceso de login y realizar un ataque de fuerza bruta hacia el endpoint pasando un diccionario de usuarios y password para poder extraer credenciales de usuario. También puede realizar ataques de Fuzzing o pasar información invalida hacia el servidor y en caso de que este no tenga un buen manejo de errores podría llegar a extraer información del mismo o incluso hacer que el servicio dejase de funcionar.

4. Segundo Vector: Protección de código generado para evitar ingeniería inversa.

4.1. Desarrollo Teórico

Ingeniería inversa se entiende como el proceso de análisis de una aplicación de escritorio, web o móvil para extraer información sobre su código fuente con el fin de entender sus funcionalidades desarrolladas y componentes. Generalmente el proceso involucra llegar de un estadio de la aplicación compilada a poder visualizar en su totalidad o parcialmente el código fuente de la misma. Para el caso de las aplicaciones móviles existen diferentes formas para realizar este proceso se explicaran las más conocidas y en el desarrollo práctico se mostrará un ejemplo concreto de ingeniería inversa sobre la aplicación móvil desarrollada en el primer vector. Por último, se compara el resultado de hacer este proceso sobre

la misma aplicación con y sin el control de seguridad de ofuscación de código.

Para los casos específicos de aplicaciones móviles Android, la manera más común de realizar un proceso de ingeniería inversa es decodificando el *APK* generado, o compilado, de la aplicación a tratar, hasta obtener el código fuente lo más cercano posible al original. Entiéndase que un *APK* o *Android Package* es el archivo resultante para las aplicaciones de Android al ser construidas y este es el formato de distribución que se usa para las aplicaciones. Un *APK* puede ser comparado con archivos *JAR* o *Java Archive* que son archivos basados en formatos *ZIP* [9]. Por esta razón los archivos *APK* pueden ser descomprimidos de la misma manera que un archivo *ZIP* y en primera instancia ver los archivos compilados de código binario que hay en una aplicación a tratar [10]. Dentro de los archivos resultantes, una vez descomprimido un *APK*, están los archivos de formato *DEX* (Dalvik Executable File), los cuales son archivos ejecutables del código compilado de la aplicación [11]. Estos archivos no son de lectura humana y resulta bastante complejo tratar de entenderlos o incluso modificarlos para dicho formato. Teniendo esto en mente, durante el proceso de ingeniería inversa se deben pasar estos archivos *DEX* a formatos más sencillos de entender como *SMALI* o *JAR*. Los archivos en formato *SMALI* son archivos legibles para el humano y permiten el análisis de código de una aplicación de manera más sencilla [12].

De forma ejemplificada, si se está desarrollando una aplicación de Android en Java las variables se ven de la siguiente manera:

```
int x = 42
```

Asumiendo que esta sea la primera variable del programa entonces el código *DEX* contendrá la siguiente secuencia hexadecimal que representa la variable y su valor:

```
13 00 2A 00
```

Por último, la representación *SMALI* para esta variable que se puede obtener con diferentes herramientas por un proceso de ingeniería inversa

```
const/16 v0, 42
```

El riesgo de poder realizar un proceso de ingeniería inversa sobre una aplicación es que permite a un atacante o un usuario mal intencionado, el poder entender las funcionalidades integradas dentro de la aplicación, encontrar mecanismos de control implementados en la aplicación, por ejemplo detección de “rooteo” sobre el dispositivo, *SSL Pinning*, donde está almacenando información sensible, entre otros. También permite evaluar variables y constantes dentro del código lo que permite encontrar URLs de endpoints usados por la aplicación, tokens de accesos a APIs que pueda estar usando la misma y/o variables con información sensible como contraseñas o IDs requeridos para autorización. Por último permite evaluar las funcionalidades implementadas dentro de la aplicación y el como esta lleva a cabo dichas funcionalidades integradas.

Una forma de control usada durante el desarrollo de aplicaciones móviles para mitigar el riesgo de ataques por ingeniería inversa, es la ofuscación de código. El término de ofuscar se define como un estado en el cual no

se puede pensar con claridad. Dicho esto, el ofuscar código en aplicaciones móviles, se puede definir como el generar un grado de entropía dentro del código que no permita a un atacante el poder entenderlo de una manera tan sencilla. Es decir, si un atacante realiza un proceso de ingeniería inversa sobre un *APK* de una aplicación con ofuscamiento, el código resultante no se verá igual al que implementó el desarrollador en primera instancia y será mucho más complicado de entender según el grado de entropía generada por el algoritmo de ofuscamiento.

Para ver el efecto de un proceso de ofuscación en JAVA, se toma como ejemplo de en una aplicación con un método de cálculo de salarios para empleados, donde el método recibe un grupo de empleados y actualiza el salario de cada empleado que esté en esa lista.

```
private void ActualizarSalarioGrupo( ListaEmpleados grupo ) {
     while(grupo.HasMore()){
          Empleado empleado = groupo.GetNext(true);
          empleado.ActualizarSalario();
          VerificarRenovacion(empleado);
     }
}
```

Si el método anterior es compilado y se aplica un un proceso ofuscamiento, al momento de que un atacante realice un proceso de ingeniería inversa sobre este va a obtener la siguiente interpretación del mismo método:

```
private void g ( A b ){
     while (b.h()){
          C d = b.j(true)
          d.e();
          f(d);
     }
}
```

Nótese que además de reducir el tamaño de las variables y métodos, que también ayuda a su vez a reducir el tamaño de la aplicación, deja incomprensible el código para un atacante. Si toda la aplicación fue ofuscada se deberá invertir mucho tiempo para poder comprender sus funcionalidades y referencias si se le aplica un proceso de ingeniería inversa.

Existen herramientas que permiten la compresión, ofuscamiento y optimización de código para aplicaciones móviles en específico. Las herramientas dificultan el proceso de lectura de código obtenido por si se llegase a aplicar un proceso de ingeniería inversa sobre la aplicación como se ha mostrado en el ejemplo anterior. Existen herramientos como ProGuard y DexGuard que permiten el ofuscamiento de código para aplicaciones Android [13]. Del mismo modo iXGuard permite el ofuscamiento para codigo Swift y Objectve-C que son los lenguajes usados hoy en dia para el desarrollo de aplicaciones para iOS. Existen diversas herramientas en el mercado para el ofuscamiento de código dependiendo del lenguaje de programación usado para el desarrollo de una aplicación bien sea de escritorio o móvil. Para el caso del desarrollo práctico se usó ProGuard ya que es una herramienta gratuita, integrada con el *IDE* (Integrated Development Environment) de Android Studio, la

cual permite el ofuscamiento, compresión y optimización de aplicaciones Android. DexGuard es una herramienta paga que aparte de las funcionalidades que brinda ProGuard, integra mecanismos seguros de defensa contra ataque dinámicos de aplicaciones, es decir, pruebas durante el tiempo de ejecución de una aplicación. Cabe recalcar que el proceso de ingeniería inversa se considera una prueba estática ya que busca analizar el código de una aplicación descompilandola mientras no está en ejecución. Las pruebas dinámicas de aplicaciones móviles no están dentro del alcance de este documento pero hay que tener presente que este tipo de pruebas permiten la modificación de código mientras la aplicación se está ejecutando en un dispositivo o emulador.

4.2. Desarrollo Práctico

La *PoC* desarrollada para este segundo vector consiste en realizar un proceso de ingeniería inversa sobre el *APK* generado para la aplicación mostrada en el Primer Vector (***NAFAA***). Con esta PoC se deja evidencia de cómo un atacante puede llegar, mediante ingeniería inversa, a un estadio que le permite el análisis de las funcionalidades y procesos de una aplicación analizada.

Para poder dejar evidencia de este riesgo, dentro del proyecto de ***NAFAA*** en el archivo “build.gradle” de la aplicación se definieron tres tipos o variantes de construcción de la aplicación: “release”, “debug”, “releaseDebug”. Para esta *PoC* se usaron las variantes de “release” y “debug” para mostrar las diferencias entre un *APK* con protección de código de ofuscamiento habilitado y un *APK* con esta protección deshabilitada. La variante de “release” usa las opciones de reducción, optimización y ofuscamiento de código con la herramienta ProGuard

cuando se está generando la construcción de un *APK* de la aplicación. El link de descarga del proyecto se encuentra en la sección de repositorios.

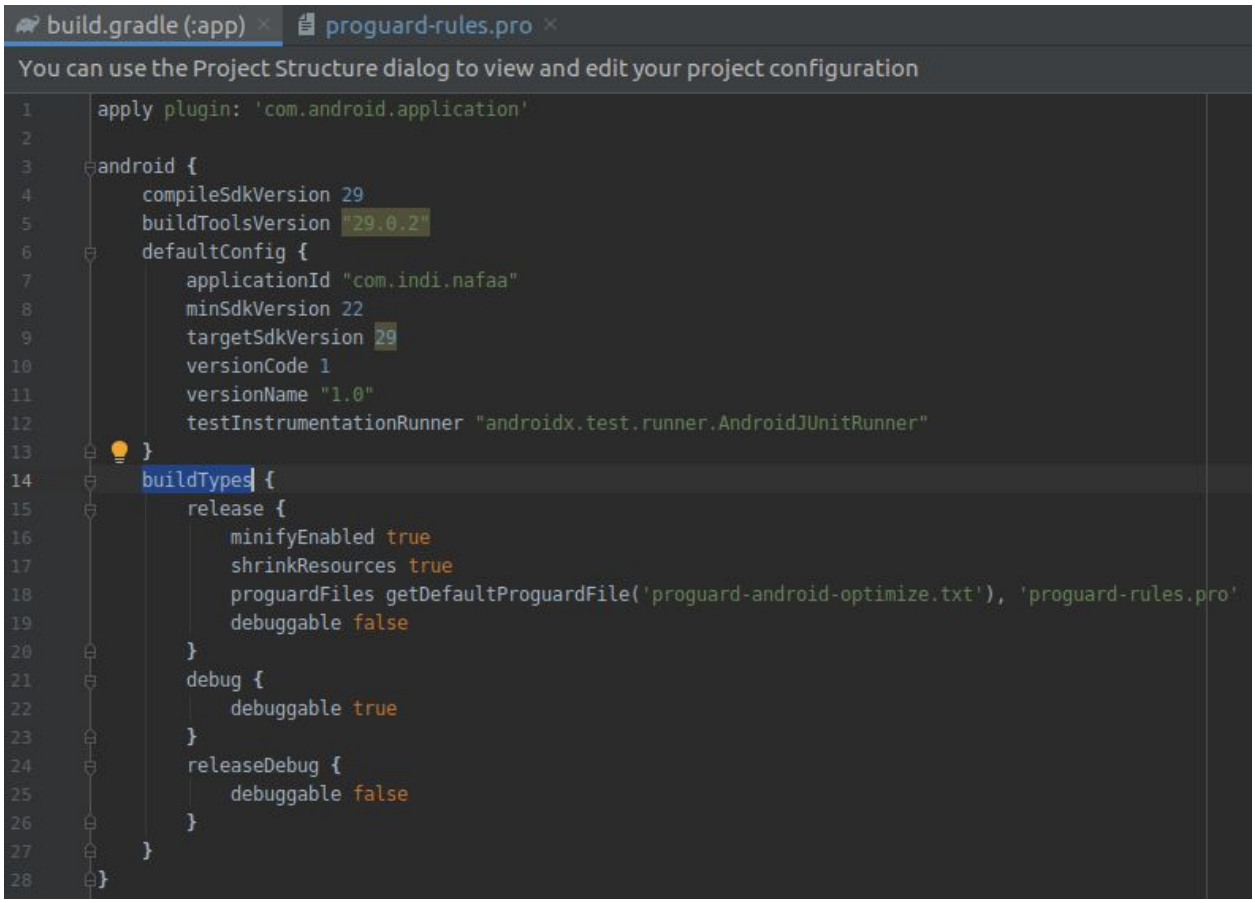

Imagen 22: Variantes de construcción aplicación NAFAA

4.2.1. Generación APK sin código ofuscado

En primera instancia se genera el *APK* sin ninguna protección de ofuscamiento de código con la variante de “debug”, con el fin de mostrar cómo, mediante ingeniería inversa, en esta versión se puede llegar a ver el código de la aplicación tal cual se desarrolló antes de su compilación y el riesgo que esto representa.

Como se mostró en el desarrollo práctico del primer vector, la aplicación ***NAFAA*** se desarrolló usando el lenguaje JAVA con el IDE de Android Studio. Por motivos de prueba el *APK* generado no será firmado ya que no será necesario subirlo al Play Store de Google, sino que será trabajado de manera local. Para generar un *APK* de manera local en Android Studio basta con seleccionar la variante que se quiere construir, para este caso “debug”, y luego generar el *APK* desde la opción de “Build” en el IDE de Android Studio.

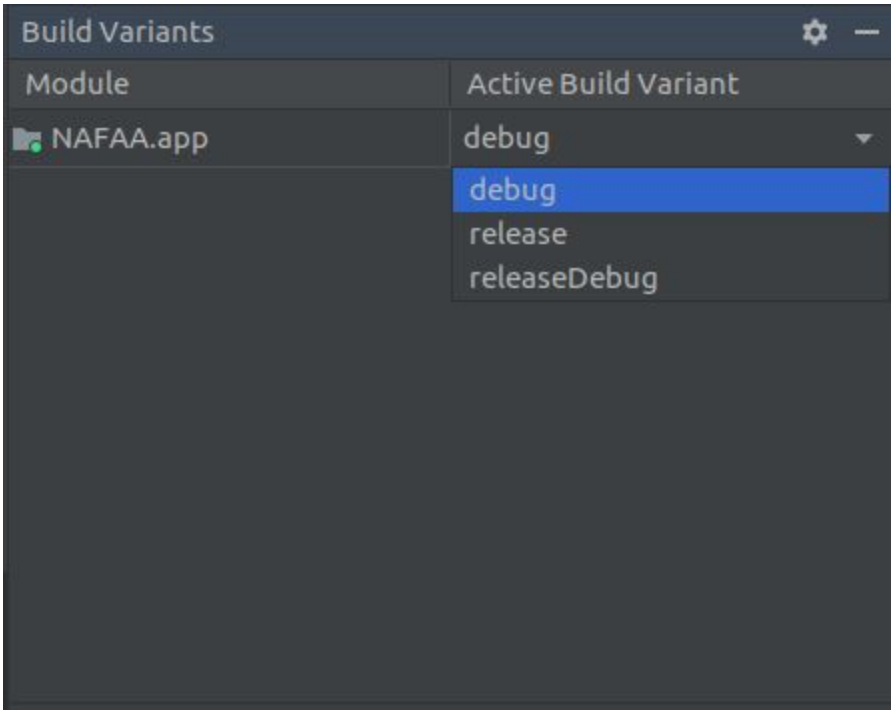

Imagen 23: Selección de variante “debug” para generar APK sin ofuscación

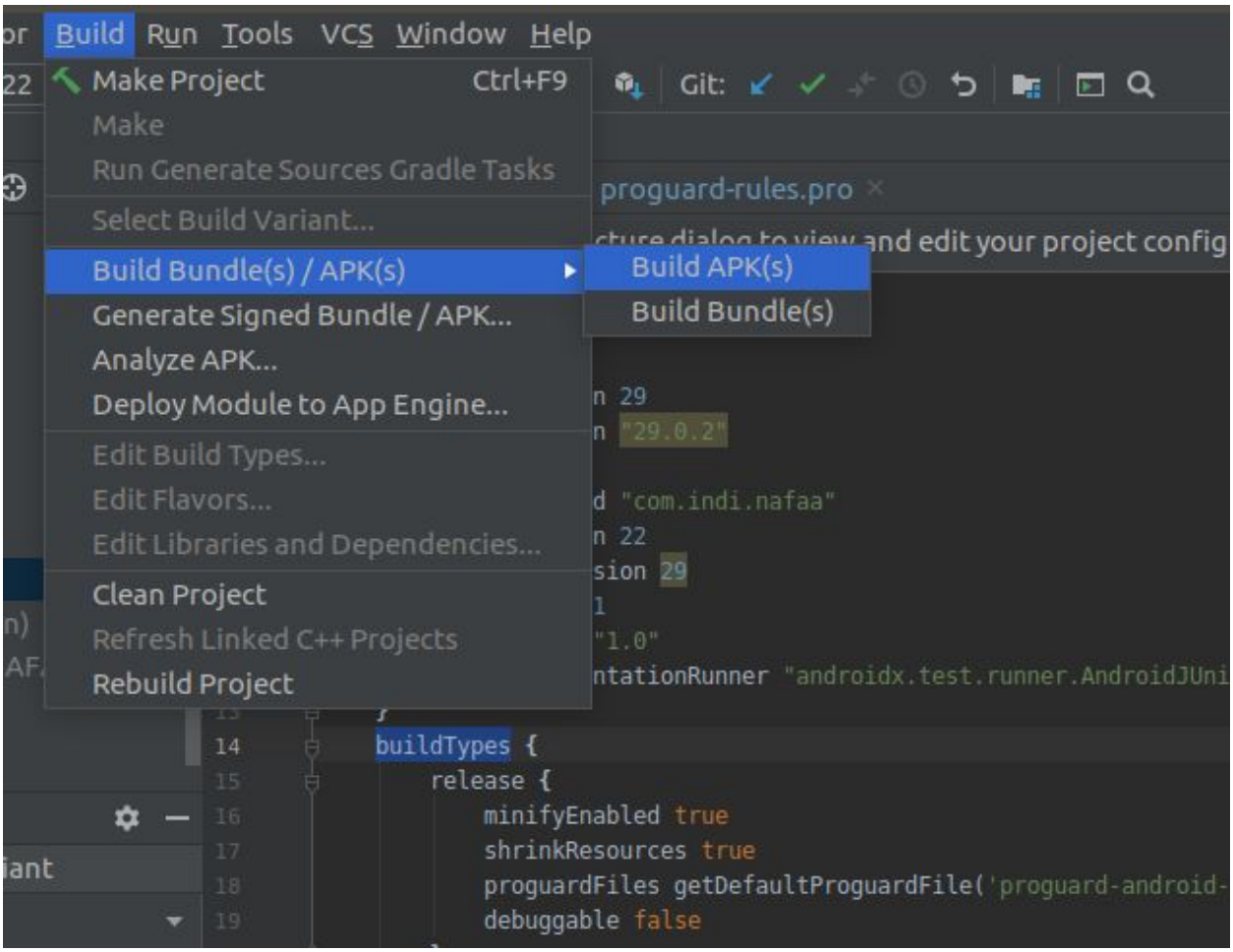

Imagen 24: Generación APK IDE Android Studio

Con este proceso se genera el APK de la aplicación y es posible someterlo a un proceso de ingeniería inversa. El APK generado de la aplicación se encuentra en la ruta del proyecto “~/NAFAA/app/build/outputs/apk/debug”.

Para aplicaciones instaladas en dispositivos es posible extraer los APKs de las aplicaciones por diferentes procesos como backups, extracción de la aplicación vía SSH para dispositivos “rooteados” o con permisos de administración, entre otros. No se mostrarán estos procesos de extracción de APKs desde dispositivos por el alcance establecido en el proyecto y debido a que se pueden

generar directamente desde el IDE, como se mostró anteriormente, para poder evaluarlos.

4.2.2. Proceso de ingeniería inversa APK no ofuscado

Imagen 25: Generación APK IDE Android Studio

Para el proceso de ingeniería inversa, y como se mostró en la sección de Desarrollo Teórico de este vector, los archivos *APK* son tratados como archivos *ZIP*. Es por esto que el *APK* generado se descomprime como si fuese un *ZIP* y se obtiene una carpeta que contiene los archivos binarios en formato *DEX*, el archivo manifiesto que contiene las actividades implementadas en la aplicación, permisos y puntos de entrada de la misma. El *APK* descomprimido también contiene la metadata y archivos de recursos que no son tan imprescindibles para obtener el código de la aplicación. Los archivos que realmente son indispensables en el proceso de ingeniería inversa son los de formato *DEX* que contienen las clases compiladas que se han desarrollado en la aplicación y son aquellos que se quiere evaluar para conocer las funcionalidades que contiene la aplicación. Para el proceso de ingeniería inversa, desde este punto, se hace uso de la

herramienta dex2jar [14]. Esta herramienta es de uso gratuito y permite convertir desde formato *DEX* a *JAR* y de esta manera con cualquier interfaz gráfica de usuario que permita la apertura de archivos *JAR* se puede analizar el código de la aplicación. Para el caso de esta PoC se usó JD-GUI para examinar los *JAR* generados por la herramienta dex2jar.

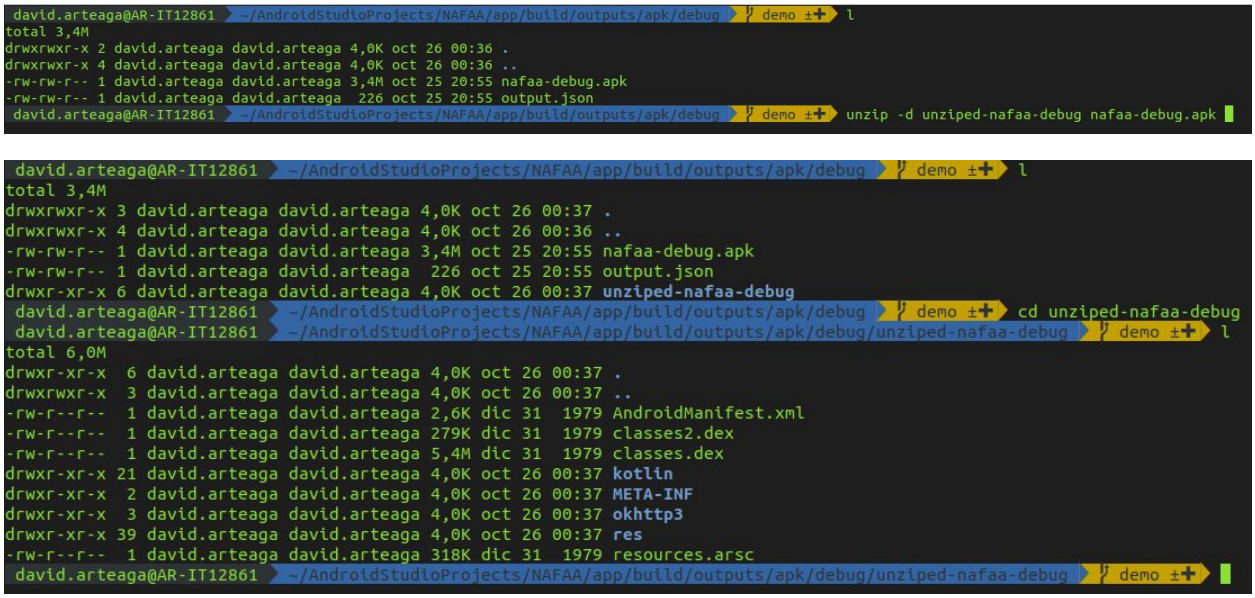

Imagen 26: Descompresión APK no ofuscado

Como se puede evidenciar, para el caso específico de la aplicación desarrollada, al momento de descomprimir el APK se obtienen dos archivos en formato DEX, el primero de ellos contiene las dependencias de servicios integrados y librerías propias de android, el segundo contiene las clases desarrolladas para esta aplicación. Hay que tener en cuenta que esta versión no pasó por un proceso de ofuscación, optimización ni reducción de código y por defecto, los archivos *DEX* tienen un máximo de 8K en tamaño. Dicho esto, si una aplicación tiene las configuraciones por defecto,

entre más código tenga, o funcionalidades desarrolladas dentro de la misma, esta pesará más y tendrá más archivos *DEX* a la hora de descomprimir el *APK*. [16]

Paso siguiente es convertir el archivo de formato *DEX* para a un archivo de formato *JAR* con la herramienta dex2jar como se mencionó anteriormente.

Imagen 27: Proceso de ingeniería inversa pasar de formato dex a jar para APK no ofuscado

Finalizada la conversión, se puede ver que se genera el archivo en formato *JAR* y está listo para ser analizado por cualquier interfaz gráfica de usuario que permita visualizar archivos *JAR*. Como se mencionó anteriormente, para esta *PoC* se usó la interfaz gráfica JD-GUI donde se hará una comparación contra el código en la siguiente sección.

4.2.3. Análisis de resultados *APK* no ofuscado

Como primera parte del análisis de la aplicación no ofuscada, se puede comparar que la estructura de clases y paquetes del *JAR* obtenido es bastante similar a la original. La principal diferencia radica en las clases generadas por configuración del proyecto a través de la compilación que son BuildConfig.java y el archivo de

referencia de recursos, R.class. Del mismo modo se pueden visualizar las clases principales desarrolladas en la aplicación (MainActivity.class, ProfileActivity.class, RequestAsyncTask.class y Utils.class). Se puede evidenciar que dichas clases encuentran bajo el mismo paquete de recursos y son estas clases las que contienen todas la funcionalidades de la aplicación. [15]

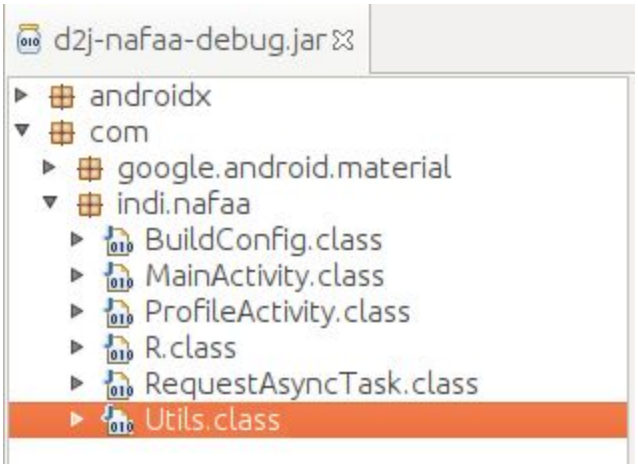

Imagen 28: Estructura de clases para JAR de APK no ofuscado

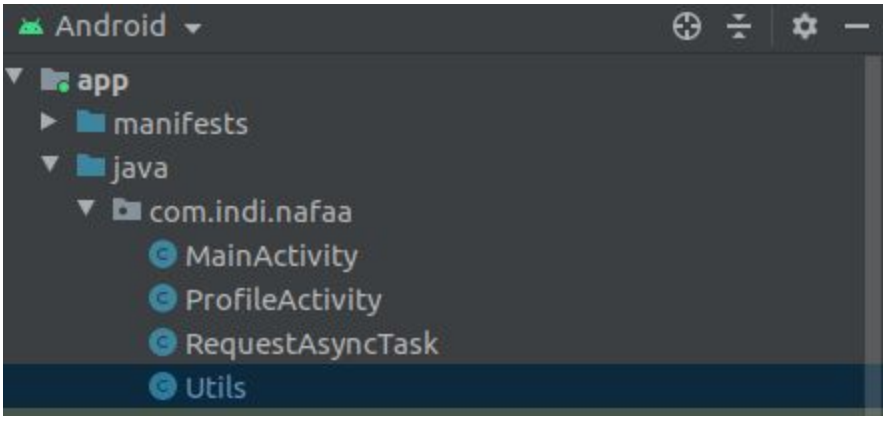

Imagen 29: Estructura de clases original aplicación Android Studio

Como segunda parte del análisis de la aplicación no ofuscada, se compara la clase Utils. Tanto del *JAR* obtenido como dentro del

código original con el que se generó la aplicación, en la clase Utils se encuentra implementado el control de seguridad de *SSL Pinning* como se mostró en el primer vector en la sección de desarrollo práctico de desarrollo de la aplicación móvil. En esta clase se encuentra configurado el hash del certificado del *WS* y como se puede evidenciar en las imágenes 6 o 31 , el código del *JAR* obtenido es idéntico al original con diferencias mínimas de indentación de código, lo que permite a un atacante el poder evaluar las funcionalidades de la aplicación como si estuviese leyendo el código original con el que se generó el *APK*. Para este caso en específico, un atacante puede encontrar el método que implementa la protección de *SSL Pinning* y eventualmente entendiendo el cómo se implementó este control sobrepasar con métodos de pruebas dinámicas o recompilando la aplicación con un hash diferente al original. Del mismo modo, para esta aplicación, se puede ver las variables usadas dentro del código. Si una aplicación tuviese llaves privadas o API Keys embebidos para el uso de la misma, quedarían expuestos a lectura de un atacante.

Utils.class

```
import okhttp3.CertificatePinner;
import okhttp3.MediaType;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.RequestBody;
import okhttp3.Response;

public class Utils {
  private static final String BASE_URL = "https://34.196.140.186:5000/";

  public static final MediaType JSON = MediaType.get("application/json; charset=utf-8");

  public static final String LOGIN_SERVICE = "login";

  private static final String TAG = "Utils";

  private OkHttpClient clientSecure;

  private OkHttpClient clientUnsecure;

  public Utils() {
    ConstructSecureClient();
    ConstructUnsecureClient();
  }

  private void ConstructSecureClient() {
    CertificatePinner certificatePinner = (new CertificatePinner.Builder()).add("NAFWS", new String[] { "sha256/UQk1OS/TekX9Cz/i+YHNPPQOU4Ux6MtiNXH9jEuatBo=" }).build();
    try {
      Log.i("Utils", "ConstructSecureClient-->1");
      TrustManager[] arrayOfTrustManager = secureTrustedCerts();
      SSLContext sSLContext = SSLContext.getInstance("SSL");
      sSLContext.init(null, arrayOfTrustManager, new SecureRandom());
      SSLSocketFactory sSLSocketFactory = sSLContext.getSocketFactory();
      OkHttpClient.Builder builder = new OkHttpClient.Builder();
      builder.sslSocketFactory(sSLSocketFactory, (X509TrustManager)arrayOfTrustManager[0]);
      builder.hostnameVerifier(verifier());
      builder.certificatePinner(certificatePinner);
      this.clientSecure = builder.build();
    } catch (NoClassDefFoundError noClassDefFoundError) {
      this.clientSecure = new OkHttpClient();
    } catch (Exception exception) {}
  }
```

Imagen 30: Clase Utils para JAR de APK sin ofuscación

Utils.java

```
import okhttp3.CertificatePinner;
import okhttp3.MediaType;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.RequestBody;
import okhttp3.Response;

public class Utils {

    public static final String LOGIN_SERVICE = "login";
    public static final MediaType JSON = MediaType.get("application/json; charset=utf-8");
    private static final String TAG = "Utils";
    private static final String BASE_URL = "https://34.196.140.186:5000/";
    private OkHttpClient clientSecure;
    private OkHttpClient clientUnsecure;

    public Utils(){
        ConstructSecureClient();
        ConstructUnsecureClient();
    }

    private void ConstructSecureClient(){
        // Create certificate pinner with de hash of the host to pin
        CertificatePinner certificatePinner = new CertificatePinner.Builder()
                .add( pattern: "NAFWS",  ...pins: "sha256/UQk1OS/TekX9Cz/i+YHNPPQOU4Ux6MtiNXH9jEuatBo=") //NAFWS PIN CERT
                .build();
        try{
            Log.i(TAG,  msg: "ConstructSecureClient-->1");
            TrustManager[] trustPinCert = secureTrustedCerts();
            // Install the all-trusting trust manager
            final SSLContext sslContext = SSLContext.getInstance("SSL");
            sslContext.init( km: null, trustPinCert, new java.security.SecureRandom());
            // Create an ssl socket factory with our all-trusting manager
            final SSLSocketFactory sslSocketFactory = sslContext.getSocketFactory();
            OkHttpClient.Builder builder = new OkHttpClient.Builder();
            builder.sslSocketFactory(sslSocketFactory, (X509TrustManager)trustPinCert[0]);
            builder.hostnameVerifier(verifier());
            builder.certificatePinner(certificatePinner);
            clientSecure = builder.build();
        }
        catch (NoClassDefFoundError defFoundError){
            //HOOKED
            clientSecure = new OkHttpClient();
        }
        catch (Exception e){
```

Imagen 31: Clase Utils original aplicación Android Studio

4.2.4. Generación *APK* con código ofuscado

Para generar un *APK* con código ofuscado de este proyecto se selecciona la variante de “release” la cual tiene configuradas las reglas básicas de ProGuard para ofuscamiento, reducción y optimización de código.

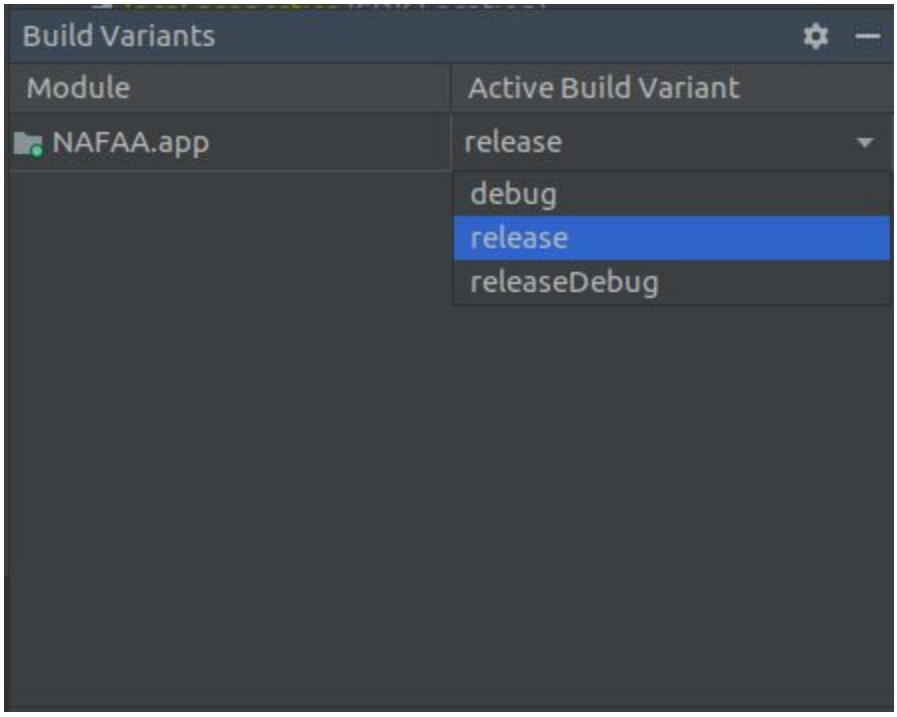

Imagen 32: Selección de variante “release” para generar APK con ofuscación

Recordemos que las configuración dentro de la variante de “release” se realizan en el archivo de build.gradle del proyecto de la aplicación al momento de implementar las diferentes variantes. Para este caso se usaron las reglas por defecto de ofuscamiento de ProGuard [4]. Para algunos casos para aplicaciones con diferentes funcionalidades que usen librerías de terceros se deben modificar las reglas para que no afecte la funcionalidad de las librerías de terceros usadas e invocadas dentro de la aplicación [16].

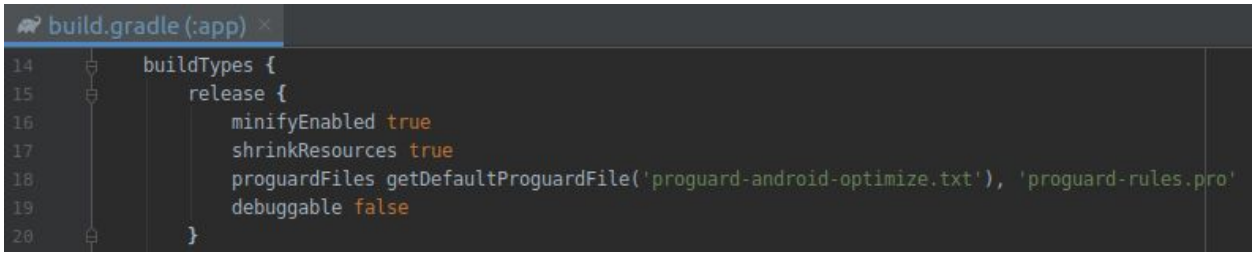

Imagen 33: Configuración reglas ProGuard variante “release”

En caso de tener que agregar excepciones en el ofuscamiento del código, como se mencionó anteriormente, por ejemplo para algunas librerías de terceros, se agregan en el archivo ‘proguard-rules.pro’ dentro del proyecto, el cual se invoca dentro de la regla de la variante, para este caso “release”, en el momento de la definición de los archivos que ProGuard usará como se puede ver en la imagen 33. Para este caso en específico se agregó una regla que imprima las reglas aplicadas dentro de un archivo temporal.

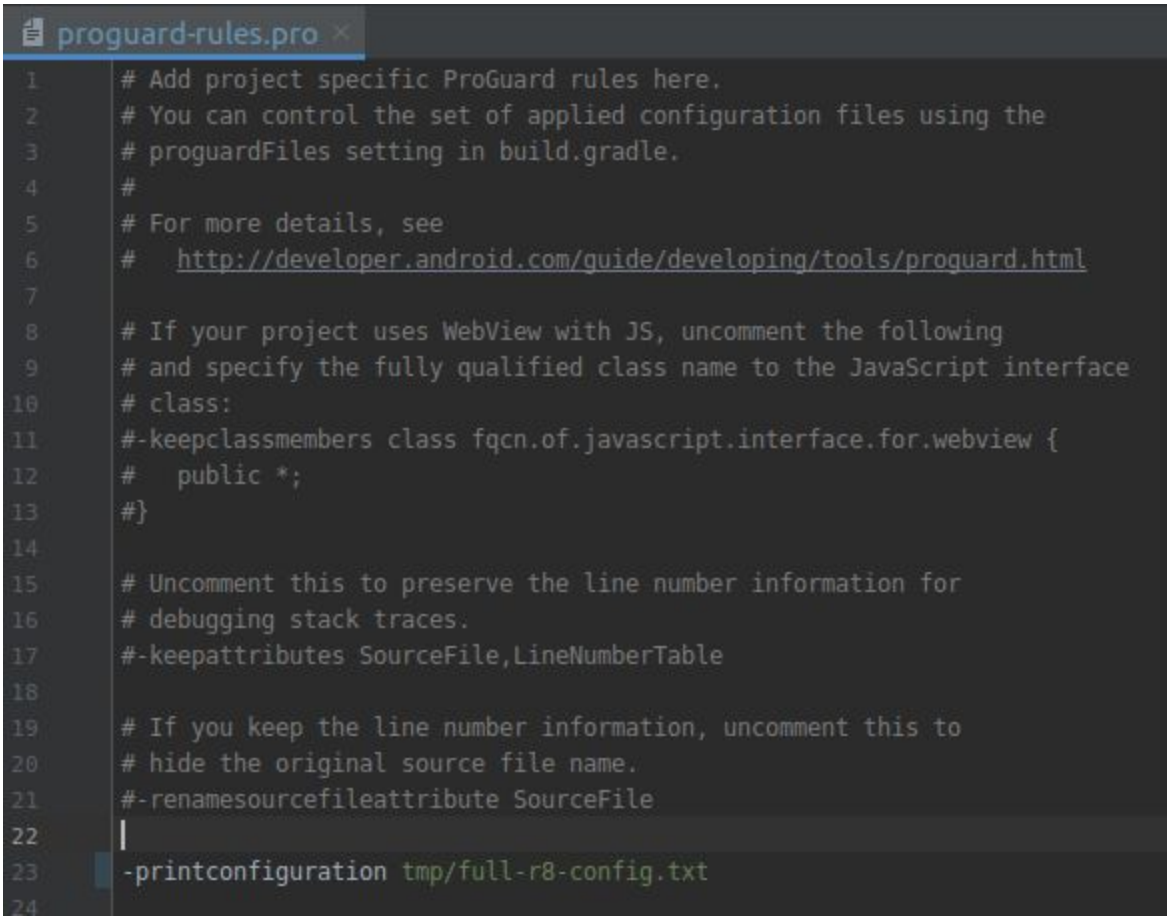

Imagen 34: Configuración reglas ProGuard variante “release”

Las reglas aplicadas dentro de este proyecto son las que trae por defecto la herramienta de ProGuard y como se puede evidenciar excluye de la ofuscación clases nativas y librerías core propias de Android para mantener un correcto funcionamiento y correlación de clases de la aplicación.[17]

Aunque dentro de las reglas definidas por defecto se excluyen clases y algunas librerías nativas, las clases principales junto con las funcionalidades de la aplicación desarrollada, serán ofuscadas y a la hora de evaluar el código después de hacer el proceso de

ingeniería inversa, se verá el efecto de las reglas de ofuscación aplicadas.

Open ▾ full-r8-config.txt ~/AndroidStudioProjects/NAFAA/app/tmp Save

```
-keep class androidx.core.app.CoreComponentFactory

-keep public class * extends androidx.versionedparcelable.VersionedParcelable
-keep public class android.support.**Parcelizer { *; }
-keep public class androidx.**Parcelizer { *; }
-keep public class androidx.versionedparcelable.ParcelImpl

-keepattributes *Annotation*

-keepclassmembers enum androidx.lifecycle.Lifecycle$Event {
    <fields>;
}

-keep !interface * implements androidx.lifecycle.LifecycleObserver {
}

-keep class * implements androidx.lifecycle.GeneratedAdapter {
    <init>(...);
}

-keepclassmembers class ** {
    @androidx.lifecycle.OnLifecycleEvent *;
}
-keepclassmembers,allowobfuscation class * extends androidx.lifecycle.ViewModel {
    <init>();
}

-keepclassmembers,allowobfuscation class * extends androidx.lifecycle.AndroidViewModel {
    <init>(android.app.Application);
}

-keep @android.support.annotation.Keep class * {*;}

-keepclasseswithmembers class * {
    @android.support.annotation.Keep <methods>;
}
```

Imagen 35: Reglas defecto de herramienta ProGuard

Por último, una vez se definan las reglas que se usarán y se tengan configuradas en la variante se procede a generar el *APK* de la misma manera que se realizó para generar el *APK* sin ofuscar como se muestra en la imagen 24.

4.2.5. Proceso de ingeniería inversa APK ofuscado

Una vez generado el *APK* ofuscado se realiza el proceso de ingeniería inversa de la misma forma como se realizó para la

aplicación sin ofuscar siguiendo los pasos mostrados en la imagen 25.

En primera instancia se descomprime el *APK* como si fuese un archivo *ZIP* que permite extraer los archivos de código fuente en formato DEX de la aplicación

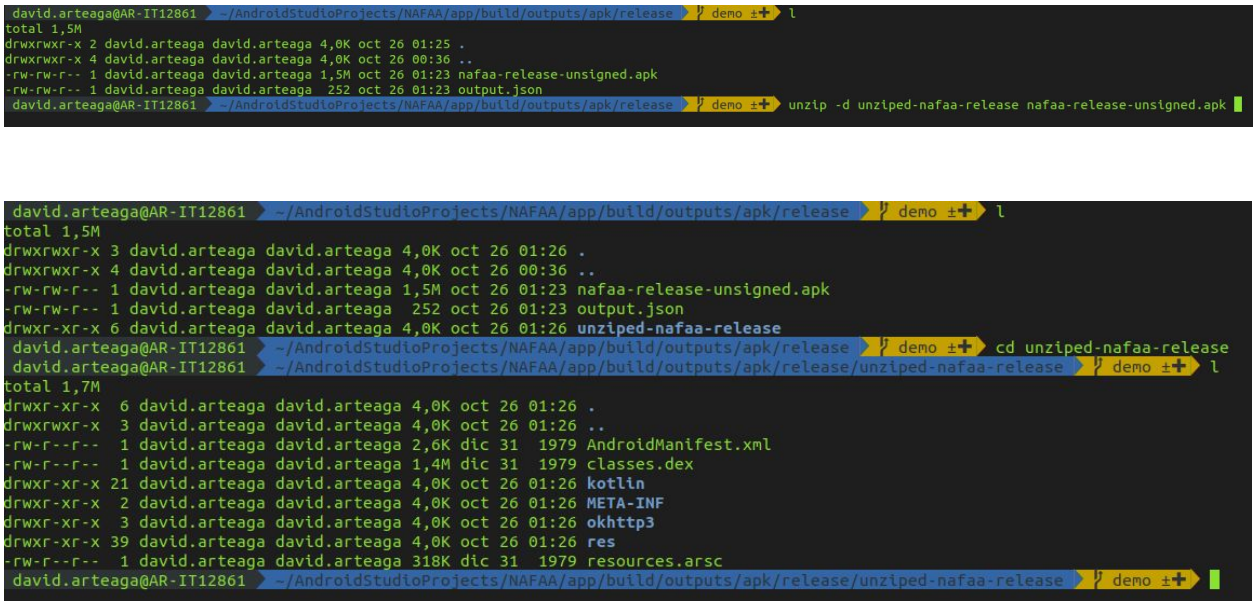

Imagen 36: Descompresión APK ofuscado

Nótese que para este caso, al momento de descomprimir el *APK* de la aplicación, se ve solo un archivo en formato *DEX,* se debe a las configuraciones dadas al momento de generar esta variante con las reglas de ofuscamiento, optimización y sobre todo de reducción de código.

Una vez se obtiene el archivo en formato *DEX* se pasa a formato *JAR* usando la herramienta dex2jar del mismo modo como se mostró para el APK sin ofuscar.

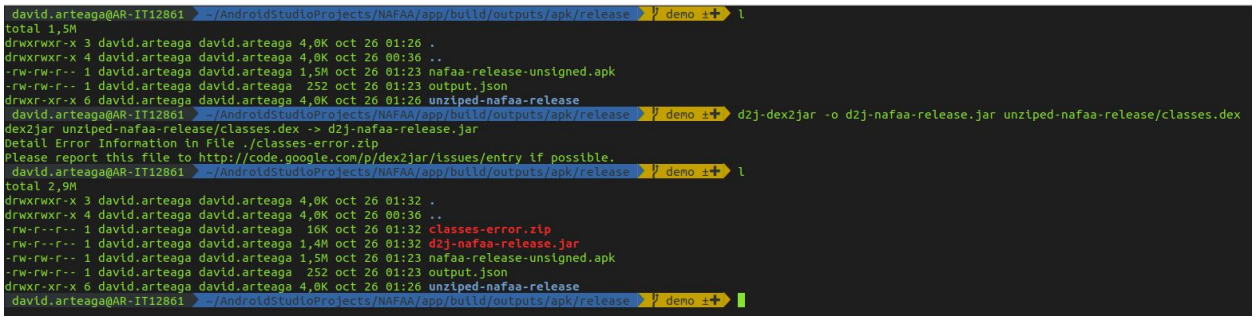

Imagen 37: Descompresión APK ofuscado

Completado este proceso se obtiene el *JAR* que permite el análisis del código para esta variante del *APK* generado de la aplicación ***NAFAA***. El análisis se realiza nuevamente con la herramienta de visualización gráfica para archivos *JAR* de JD-GUI.

4.2.6. Análisis de resultados *APK* ofuscado

Como primera parte del análisis se puede evidenciar la diferencia en la estructura del proyecto frente a la estructura que se tiene originalmente y la que se había encontrado para la aplicación sin ofuscar. Algunas de las clases y librerías nativas de android están incluidas debido a la reducción de código que se configuró para esta variante del APK. Del mismo modo se puede ver que muchas de las clases y paquetes dentro de la estructura están definidas solo por una letra debido a la ofuscación de código. Las unicas clases que mantienen el nombre son aquellas clases que extienden de AppCompatActivity, cómo es el caso de “MainActivity” y “ProfileActivity”.

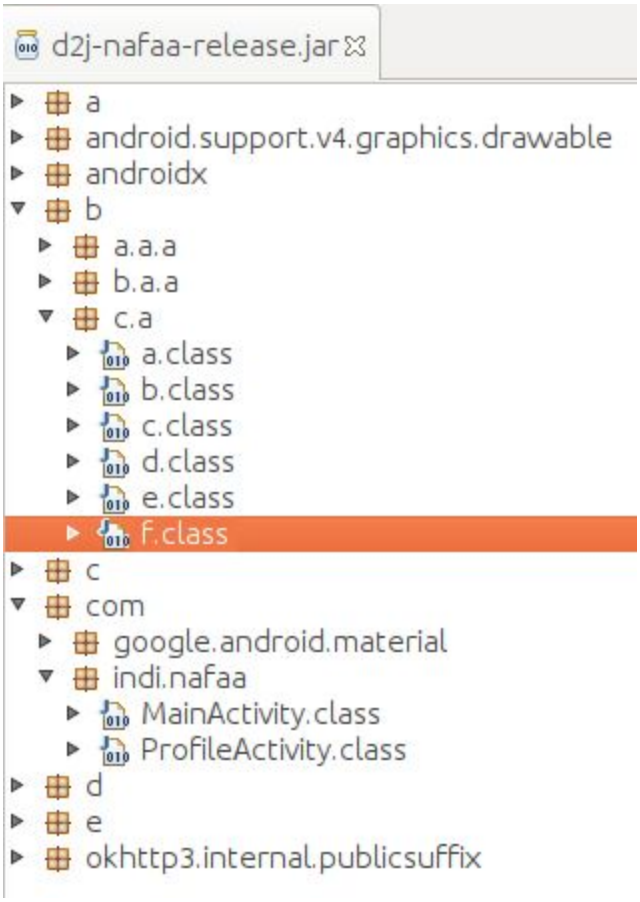

Imagen 38: Estructura de clases para JAR de APK ofuscado

Como segunda parte del análisis y como se realizó para el *JAR* obtenido de la aplicación no ofuscada, se compara la clase “Utils”, en la cual se implementó el control de seguridad de *SSL Pinning*, se puede constatar que para el caso de la *JAR* obtenido desde el *APK* con ofuscación, la clase “Utils” se ha dividido en diferentes clases con un renombramiento a una sola letra. En el caso del hash con el que se realiza la validación del certificado, se encuentra en la clase nombrada “f” y se puede ver, que pese a que se ve el hash del certificado como un string, no se logra entender del todo como este objeto es usado por la aplicación. Nótese que toda la clase “Utils” ha cambiado y es muy difícil para un atacante

el poder deducir la funcionalidad de este control y sus otras funcionalidades desarrolladas en general. Debido al conocimiento previo del funcionamiento de la aplicación se pudo encontrar el hash en este fragmento de clase nombrada “f” pero como se mencionó anteriormente la clase “Utils” en sí se dividió en diferentes pedazos y aunque se posible ver el hash no es evidente el uso que se le dá lo largo de la aplicación ni el momento en cual se está invocando dicho objeto. Esto aumenta considerablemente la posibilidad de que un atacante entienda el código de la aplicación y sus funcionalidades y el poder elaborar una forma para saltar los control implementados en la misma.

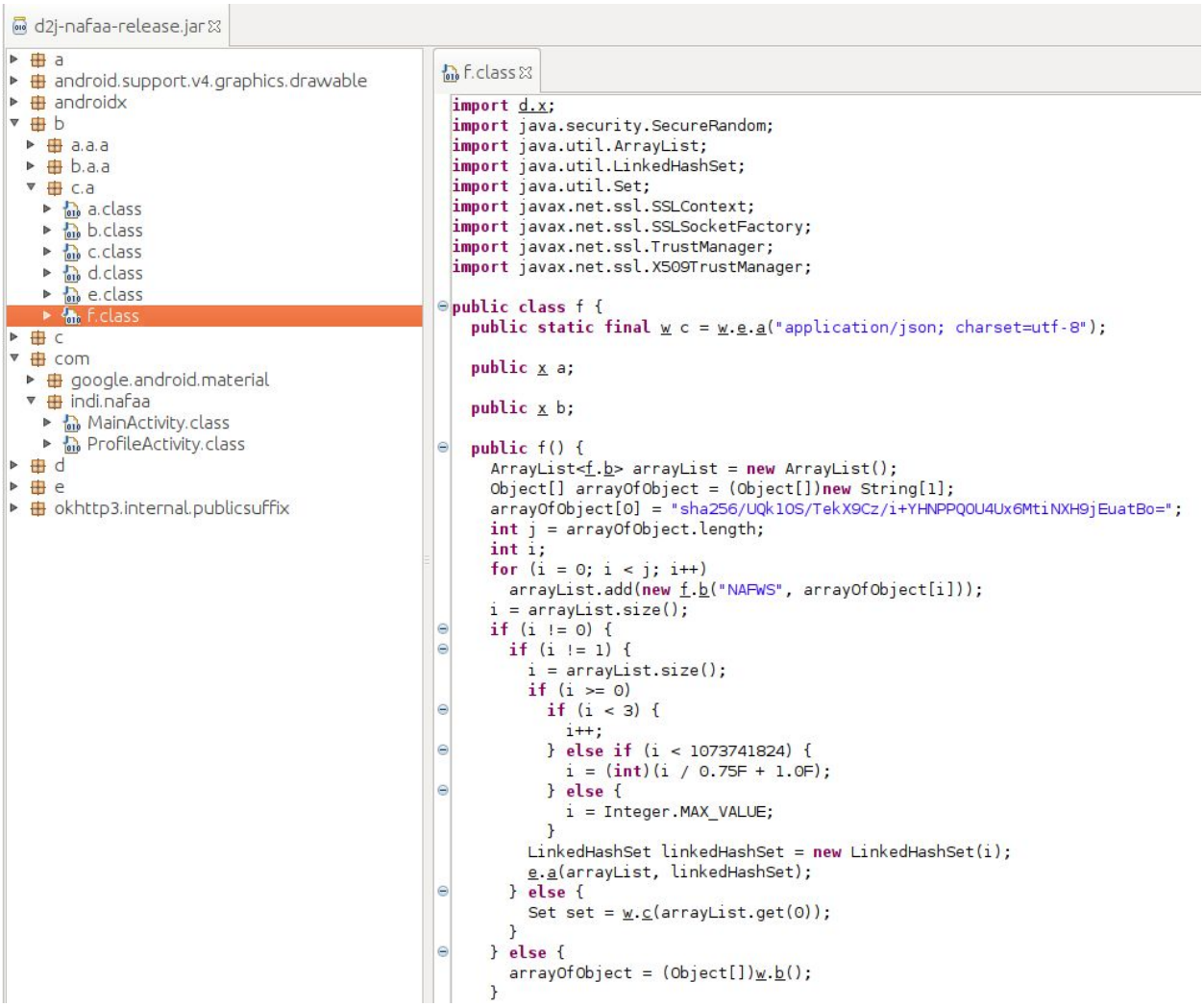

Imagen 39: Fragmento clase Utils para JAR de APK con ofuscación

Por último, se puede verificar que aunque el nombre de las clases de “MainActivity” y “ProfileActivity” no cambiaron, el código implementado dentro de las mismas, pasó por un proceso de ofuscación. Hay que recordar que dentro de las reglas que tiene por defecto la herramienta ProGuard, las clases nativas y librerías core de Android se mantienen para garantizar un correcto funcionamiento de la aplicación. Las clases “MainActivity” y “ProfileActivity” al extender de clases nativas de tipo

“AppCompatActivity” que son nativas de Android, los métodos que implementan esta extensión no se verán afectados por la ofuscación, sin embargo otros métodos implementados que son funcionalidades propias de la aplicación, como por ejemplo el método de “loginMethod”, se van a ver afectadas por el proceso de ofuscamiento.

MainActivity.class

```
public class MainActivity extends l {
  public TextView p;

  public void a(String paramString) {
    try {
      int i = Integer.parseInt((new JSONObject(paramString)).get("code").toString());
      StringBuilder stringBuilder = new StringBuilder();
      stringBuilder.append("handleResponse: ");
      stringBuilder.append(paramString);
      Log.i("MainActivity", stringBuilder.toString());
      if (i == 200) {
        startActivity(new Intent((Context)this, ProfileActivity.class));
        return;
      }
      if (i == 401) {
        this.p.setVisibility(0);
        return;
      }
      Toast.makeText((Context)this, "Ups, something went wrong...", 1).show();
      return;
    } catch (JSONException jSONException) {
      StringBuilder stringBuilder = a.a("handleResponse: ");
      stringBuilder.append(jSONException.getLocalizedMessage());
      Log.e("MainActivity", stringBuilder.toString());
      Toast.makeText((Context)this, "Ups, something went wrong...", 1).show();
      return;
    }
  }

  public void a(String paramString1, String paramString2, boolean paramBoolean) {
    f f = new f();
    JSONObject jSONObject = new JSONObject();
    try {
      jSONObject.put("username", paramString1);
      jSONObject.put("password", paramString2);
      paramString1 = jSONObject.toString();
      (new a(this)).execute(new Object[] { f, paramString1, Boolean.valueOf(paramBoolean) });
      return;
    } catch (JSONException jSONException) {
      jSONException.printStackTrace();
      return;
    }
  }
```

Imagen 40: Clase MainActivity para JAR de APK ofuscado

MainActivity.java

```java
public class MainActivity extends AppCompatActivity {

    private static String TAG = "MainActivity";

    private TextView tvBadAuth;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        FloatingActionButton fab = findViewById(R.id.fab);
        fab.setOnClickListener((view) → {
                Snackbar.make(view, text: "Not today...", Snackbar.LENGTH_LONG)
                        .setAction( text: "Action", listener: null).show();
        });

        final EditText etUsername = findViewById(R.id.et_main_username);
        final EditText etPassword = findViewById(R.id.et_main_password);
        tvBadAuth = findViewById(R.id.tvBadAuth);
        Button btnLoginSecure = findViewById(R.id.btnSecureLogin);
        btnLoginSecure.setOnClickListener((v) → {
                loginMethod(etUsername.getText().toString(), etPassword.getText().toString(), secure: true);
        });

        Button btnLoginUnsecure = findViewById(R.id.btnUnsecureLogin);
        btnLoginUnsecure.setOnClickListener((v) → {
                loginMethod(etUsername.getText().toString(), etPassword.getText().toString(), secure: false);
        });
    }

    public void loginMethod(String username, String password, boolean secure){
        Utils utils = new Utils();
        JSONObject jsonObject = new JSONObject();
        try {
            jsonObject.put( name: "username", username);
            jsonObject.put( name: "password", password);
            Object[] params = {utils, jsonObject.toString(), secure};
            new RequestAsyncTask( mainActivity: this).execute(params);
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }
```

Imagen 41: Clase MainActivity original aplicación desde Android Studio

5. Tercer Vector: Protección de datos con cifrado en capa de aplicación por ECDH y DES.

5.1. Desarrollo Teórico

Para los vectores anteriores se han mostrado dos riesgos, uno relacionado a la exposición de los endpoints a los cuales la aplicación consume los servicios web expuesto y el otro para el código en sí de la aplicación. Teniendo en cuenta que los controles propuestos en los vectores anteriores ayudan a la mitigación de los riesgos expuestos, no son métodos infalibles contra atacantes; por esto, se propone el uso de cifrado de punto a punto a nivel de la capa de aplicación para el resguardo de la información enviada desde la aplicación hacia el *WS* y brindar una capa de protección extra en la comunicación de los datos.

En este vector se muestra cómo se realizó el cifrado a nivel de aplicación usando el algoritmo de encriptación simétrica *Advanced Encryption Standard* (*AES*) usando como llave un secreto compartido generado entre los pares usando el protocolo de acuerdo de llaves *Elliptic-Curve Diffie-Hellman* (*ECDH*). Se han escogido *AES* para el cifrado y *ECDH*, para generar la llave en común entre pares, por términos de eficiencia ya que hay que tener en cuenta que estos procesos impactan en cierto modo en el rendimiento de la aplicación en general, puesto que el proceso de encriptado hace uso de los recursos de procesamiento del dispositivo y del endpoint para poder pasar un texto plano a un texto cifrado y de igual

forma para revertir el proceso. Además, antes de poder empezar a cifrar la información el cliente y servidor deben hacer intercambio de sus llaves públicas para poder generar la llave compartida. El algoritmo simétrico *AES* es uno de los algoritmos recomendados dentro de la documentación oficial de Android para el manejo de encriptación en aplicaciones móviles debido a su eficiencia, potencia y también debido a su compatibilidad con otras tecnologías al ser un estándar bastante conocido y usado [18]. Para el caso de la PoC se integra con el servicio de backend de Flask que se ha usado en los vectores anteriores, desarrollado en Python y el cliente Android en Java también usado y desarrollado anteriormente.

Para que la aplicación móvil y el *WS* pueda comunicarse de manera segura a través del algoritmo simétrico *AES*, se debe generar una llave compartida que permita encriptar y desencriptar la información desde cualquiera de los extremos. Para esto se debe tener la misma llave tanto en el cliente móvil como en el servidor que expone el *servicio web* y es por esto que para generar una llave compartida de manera segura se usa el protocolo de acuerdo de llaves *Elliptic-Curve Diffie-Hellman* (*ECDH)*. Este protocolo es de la familia de criptosistemas de llave pública, que está basada en la estructura algebraica de las curvas elípticas sobre campos finitos y la seguridad de esta radica en la dificultad de resolución del problema de logaritmo discreto de curva elíptica [19].

La criptografía en curvas elípticas (*ECC*) implementa la mayoría de capacidades que tienen los criptosistemas asimétricos de encriptación, firma e intercambio de llaves. Para este caso específico se usaron las curvas elípticas para realizar intercambio de llaves. En primera instancia se deben generar un par de llaves, pública y privada, tanto en el servidor que expone los *WS* como en el cliente mobile que quiere consumirlos.

Teniendo las llaves públicas generadas se realiza el intercambio de las mismas y se genera un secreto en común o llave secreta compartida y poder empezar a encriptar datos entre ellos.

Para poder generar este par de llaves y entendiendo las curvas elípticas como un conjunto de puntos que satisfacen una ecuación de dos variables de grado dos de la forma: $y^2 = x^3 + ax + b$, se escoge una curva en común entre los pares y un punto “G” de la curva que también es conocido por los pares y es denominado como el punto generador. Cada par escoge un número “d” aleatorio que esté dentro del rango de valores de la curva escogida. El valor “d” es la llave privada y solo debe ser conocida por el par que escogió dicho valor. La llave privada, o “d”, se multiplica por el punto generador “G” y se obtiene otro punto de la curva que representa la llave pública del par que está realizando la operación. Cuando cada par genera su llave pública “Q”, son intercambiadas entre los pares y nuevamente realizan la operación multiplicativa sobre la llave pública recibida y la llave privada del par que la recibe. En el siguiente gráfico se puede visualizar el proceso *Diffie-Hellman* aplicado en curvas elípticas explicado anteriormente teniendo como pares a Alice y Bob. Al final del proceso se puede evidenciar que el secreto compartido es el resultado de la operación multiplicativa entre la llave privada de Bob, la de Alice y el punto generador “G”.

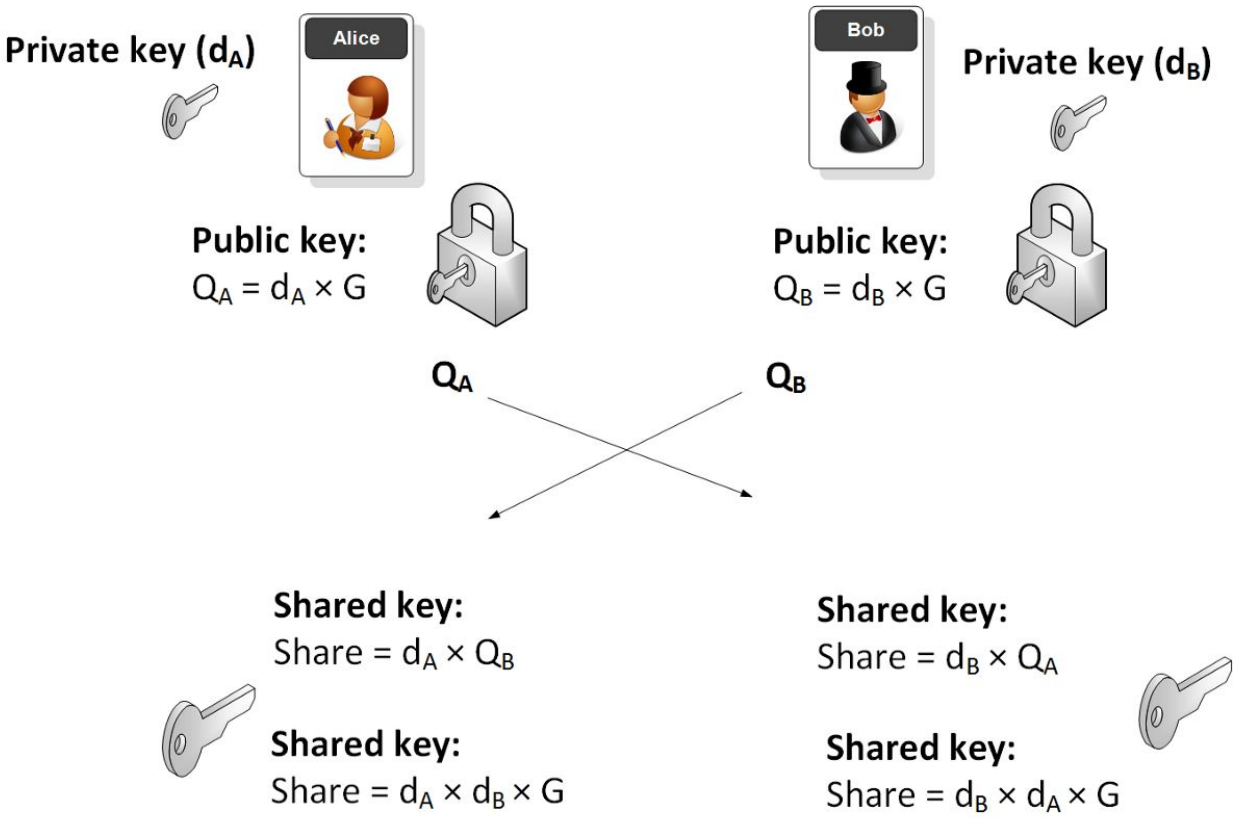

Imagen 42: Generación Secreto compartido ECDH [21]

Para el caso de la PoC se escogió la curva elíptica ***ecp256r1*** o **secp256r1**, también conocida como ***NIST P-256 Elliptic Curve*** [22]

5.2. Desarrollo Práctico

Para el último vector planteado en la PoC, se agregó la funcionalidad de encriptamiento y desencriptamiento tanto en al servidor como en la aplicación móvil de Android. Esta funcionalidad permite el encriptado de textos planos usando una llave compartida generada entre servidor y la aplicación mediante *ECDH* y encriptando los textos planos con *AES*

usando dicha llave compartida como se explicó en la sección anterior. Con esta medida se cifran los datos que son enviados entre la aplicación móvil y el *WS* que expone los servicios a una capa de aplicación. El flujo empieza desde la aplicación móvil quien en primera instancia solicita las llaves públicas del servidor Flask que expone los servicios. Del mismo modo en esta solicitud el cliente envía su llave pública al servidor el cual expone un nuevo endpoint que intercepta la solicitud de la aplicación y retorna el punto de la curva elíptica que representa la llave pública del servidor. Ambas partes, tanto cliente como servidor, con las llaves públicas contrarias pueden generar una llave compartida y empezar el proceso de cifrado entre las siguientes solicitudes que se quieran hacer.

5.2.1. Desarrollo nuevo endpoint en Web Service (*NAFWS*)

Para el caso del *WS* se desarrolló un módulo nuevo donde se manejan todos los nuevos procesos de seguridad. Este módulo contiene las funciones de encriptado y desencriptado usando el algoritmo *AES*. En este módulo también se maneja la generación de las llaves pública y privada tomando la curva elíptica “**secp256r1**” del servidor cuando se inicializa por primera vez el servicio. También se integró la funcionalidad de generar la llave compartida una vez se reciba la llave pública del cliente. Para el caso del servidor se uso la libreria de Python de “tinyec” que contiene la definición de varias curvas elípticas, la cual ayuda con la generación de la llave pública con la curva seleccionada y la llave privada que es un número aleatorio entre 1 y el campo de la curva. El módulo se encuentra dentro del proyecto del *WS* bajo el nombre de “ecdh_security_module.py”. Recordar que el repositorio del código, tanto

de la aplicación móvil como del *WS* se encuentra disponible en la sección de repositorios.

ecdh_security_module.py nafws.py

ecdh_security_module.py > generateKeys

```python
import tinyec.ec as ec
import tinyec.registry as reg
import secrets
from Crypto.Cipher import AES
import hashlib
import base64
import json

curve = ""
server_public_key = ""
server_private_key = ""
shared_key = ""

BS = 16
pad = lambda s: s + (BS - len(s) % BS) * chr(BS - len(s) % BS)
unpad = lambda s : s[0:-ord(s[-1])]

def generateKeys():
    global curve
    global server_private_key
    global server_public_key
    print("generateKeys --> Init")
    curve = reg.get_curve('secp256r1')
    #Generating Server private
    server_private_key = secrets.randbelow(curve.field.n)
    #server_private_key = 1 # Prueba con key harcodeada
    #print("Z(p): ", curve.field.n)
    print("generateKeys --> Server private key: ", server_private_key)
    # Generate Server publickey from the private key and Generator point
    server_public_key = server_private_key * curve.g
    print("generateKeys --> Server public key: ", server_public_key, "type: " , type(server_public_key))
```

Imagen 43: Función de generación de llaves

La generación de las llaves, pública y privada del servidor, solo se lleva a cabo al momento de la inicialización del *WS*. Mientras que la generación del secreto compartido se hace cada vez que un cliente se pone en contacto con el servicio y entrega su llave pública y solicita la llave pública generada por el servicio.

Para poder generar la llave compartida se recibe la llave pública del cliente, que en este caso se recibe en el request de la aplicación móvil, el

cual es un punto en la curva “**secp256r1**” o “**prime256v1**” o “**NIST P-256 EC**”.

Imagen 44: Método de generación de secreto compartido NAFWS

Después generar una llave secreta compartida entre el servidor y el cliente se usa esta llave para la encriptación de mensajes entre los pares usando el algoritmo de encriptación simétrica *AES* en modo ECB, el cual no toma un vector de inicialización, usando la llave compartida generada para inicializar el módulo *AES* de la librería Crypto.Cipher en Python. El método recibe un texto en plano y retorna el texto cifrado usando un padding de 16 bits por su llave de 256 que espera un texto plano múltiplo de 16.

ecdh_security_module.py × nafws.py

ecdh_security_module.py > encrypt

```
58
59  def encrypt(text):
60      print("encrypt() --> Intentando encriptar: ", text)
61      padded_text = pad(text)
62      print("encrypt() --> padded_text: ", padded_text ,"type: ",type(padded_text))
63      padded_text_bytes = bytes(padded_text, "utf-8")
64      print("encrypt() --> padded_text bytes: ", padded_text_bytes ,"type: ",type(padded_text_bytes))
65      key_unhex = compress_point2(shared_key)
66      print("encrypt() --> AES KEY UNHEX: ",  key_unhex)
67      key_hex = bytes.fromhex(compress_point2(shared_key))
68      print("encrypt() --> AES KEY HEX: ",  key_hex)
69      cipher = AES.new(key_hex, AES.MODE_ECB)
70      encrypted = cipher.encrypt(padded_text_bytes)
71      print("encrypt() --> encrypted_text: ", encrypted)
72      print("encrypt() --> encrypted_text2: ", base64.b64encode(encrypted))
73      print("encrypt() --> encrypted_text3: ", base64.b64encode(encrypted).decode("utf-8"))
74      return base64.b64encode(encrypted).decode("utf-8")
75
76  def decrypt(text):
77      key_unhex = compress_point2(shared_key)
78      print("decryt() --> AES KEY UNHEX: ",  key_unhex)
79      key_hex = bytes.fromhex(compress_point2(shared_key))
80      print("decryt() --> AES KEY HEX: ",  key_hex)
81      cipher = AES.new(key_hex, AES.MODE_ECB)
82      print("decrypt() --> Intentando desencriptar raw:", text, ";bytes:", bytes(text, "utf-8") ,"; len: " ,len(text))
83      decrypted_data_raw = cipher.decrypt(base64.decodebytes(bytes(text, "utf-8")))
84      print("decrypt() --> Decrypted_text_raw:", decrypted_data_raw, type(decrypted_data_raw))
85      decrypted_data_decoded = decrypted_data_raw.decode("utf-8")
86      print("decrypt() --> Decrypted_text_decoded:", decrypted_data_decoded, type(decrypted_data_decoded))
87      decrypted_data_unpadded = bytes(__unpad(decrypted_data_decoded),"utf-8")
88      print("decrypt() --> Decrypted_text_unpadded:", decrypted_data_unpadded, "type:", type(decrypted_data_unpadded))
89      return json.loads(decrypted_data_unpadded.decode("utf-8"))
```

Imagen 45: Método de encriptado y desencriptado NAFWS

Para hacer uso de este módulo, se exponen 2 nuevos servicios en la clase principal del servidor “nafws.py” usada en los vectores anteriores. Para realizar la construcción del secreto compartido entre los pares el servidor expone el servicio en la ruta “/ecdh”. Este servicio recibe la llave pública del cliente, y dado caso que sea posible generar el secreto compartido inicialmente desde el lado del servidor usando el nuevo módulo de seguridad, devuelve la llave pública del servidor que se crearon al momento de la inicialización del *WS*. Hay que recordar que la llave pública representa un punto en la curva elíptica y es por esto que este servicio devuelve dos enteros que representan las coordenadas X y Y del punto de la curva.

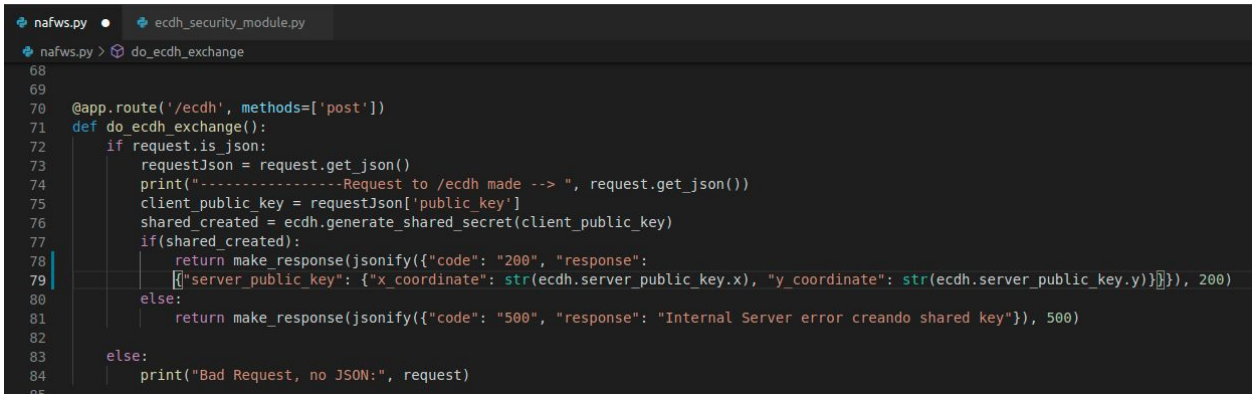

Imagen 46: Endpoint para generar secreto compartido NAFWS

Del mismo modo, se expone a su vez un nuevo servicio que recibe la información del login cifrada y la desencripta con la llave compartida previamente generada y devuelve la respuesta según la validación de parámetros dada en la ruta “/login_encrypted”. La validación se mantiene estática en el código del servidor como se realizó en los vectores anteriores donde si se recibe como “username” el valor de “admin” y en el parámetro de “password” se recibe el valor de “password” se retorna un booleano que permite o deniega el acceso del usuario según el resultado de esta validación. La respuesta también se envía encriptada usando la llave compartida donde será desencriptada en la aplicación móvil.

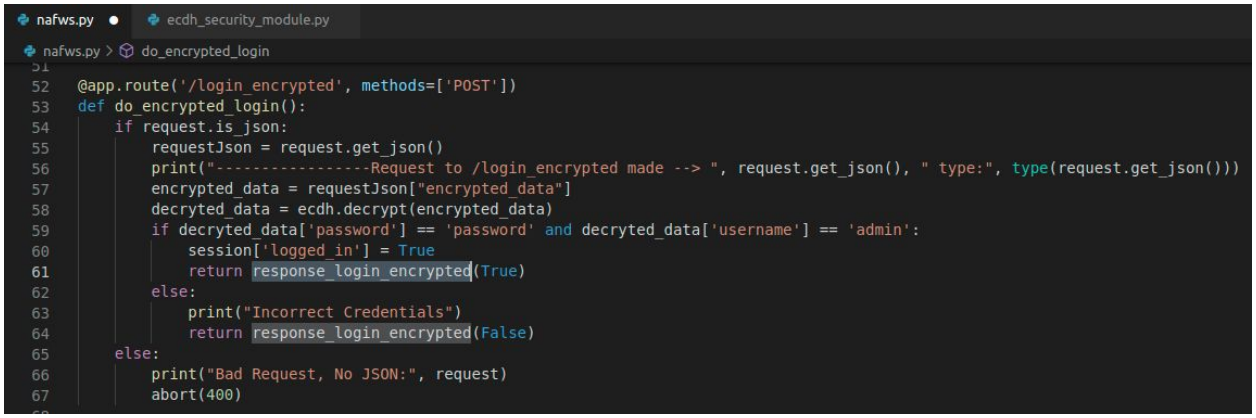

Imagen 47: Endpoint que recibe login encriptado NAFWS

Imagen 48: Método de respuesta de verificación encriptado NAFWS

Por último, hay que tener en cuenta que se dejó la convención que el servicio devuelve una respuesta en formato JSON con el parámetro “encrypted_data” que es el parámetro que contiene la respuesta del servicio y la aplicación debe desencriptar con la llave compartida para poder evaluar la respuesta del servicio. Del mismo modo, el servidor espera recibir una solicitud en formato JSON con el parámetro “encrypted_data” que contiene la información de la solicitud cifrada.

5.2.2. Desarrollo nueva funcionalidad Aplicación Móvil (NAFAA)

Para la aplicación móvil se desarrolló igualmente un módulo que permite el manejo y gestión de llaves así como las funciones de encriptado con *AES* dada la llave compartida generada con la llave pública del servicio y la llave privada del cliente/aplicación móvil. Para el nuevo módulo “SecurityModule.java” de la aplicación móvil, se usaron las librerías nativas de Java de “Java.security” y “Javax.cypto”. Con estas librerías y usando la clase de KeyPairGenerator, se puede generar la llave compartida, especificando la instancia de Curvas Elípticas con la curva “**prime256v1**”. Con estas configuraciones es posible realizar posteriormente, el proceso de generar la llave compartida escogiendo la misma curva elíptica que se definió en el *WS*. Inicializando la clase de

“KeyAgreement” con el parámetro de la llave privada generada para la aplicación, es posible
generar de manera sencilla el secreto compartido después de que se reciba la llave pública del servidor.

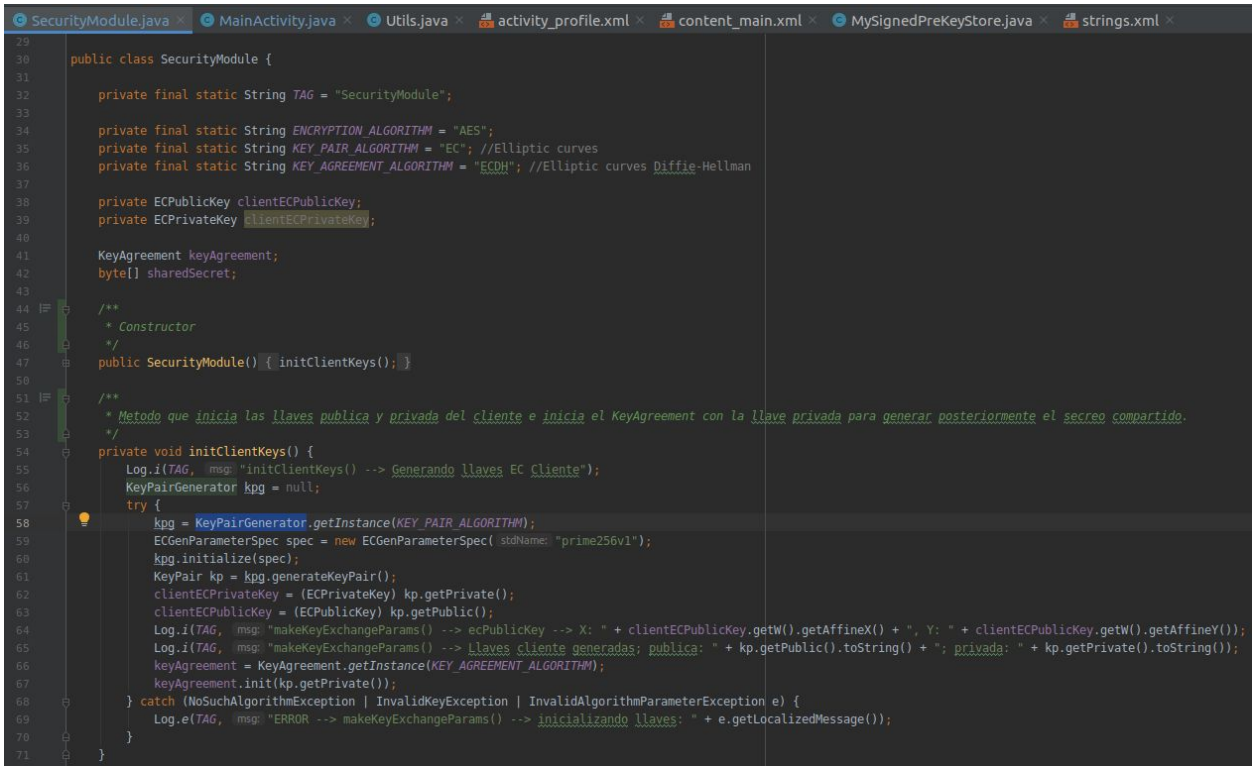

Imagen 49: Método de respuesta de verificacion encriptado NAFWS

Con la inicialización de las llaves del lado del cliente es posible recibir la llave pública del servidor y generar una llave compartida entre los pares para poder empezar a recibir y enviar datos cifrados. Con la finalización de fase para el KeyAgreement que se especificó durante la inicialización con el algoritmo de *ECDH*, se genera el secreto compartido o sharedSecret como se especificó en el código, que será usado para inicializar la librería de encriptación en *AES*.

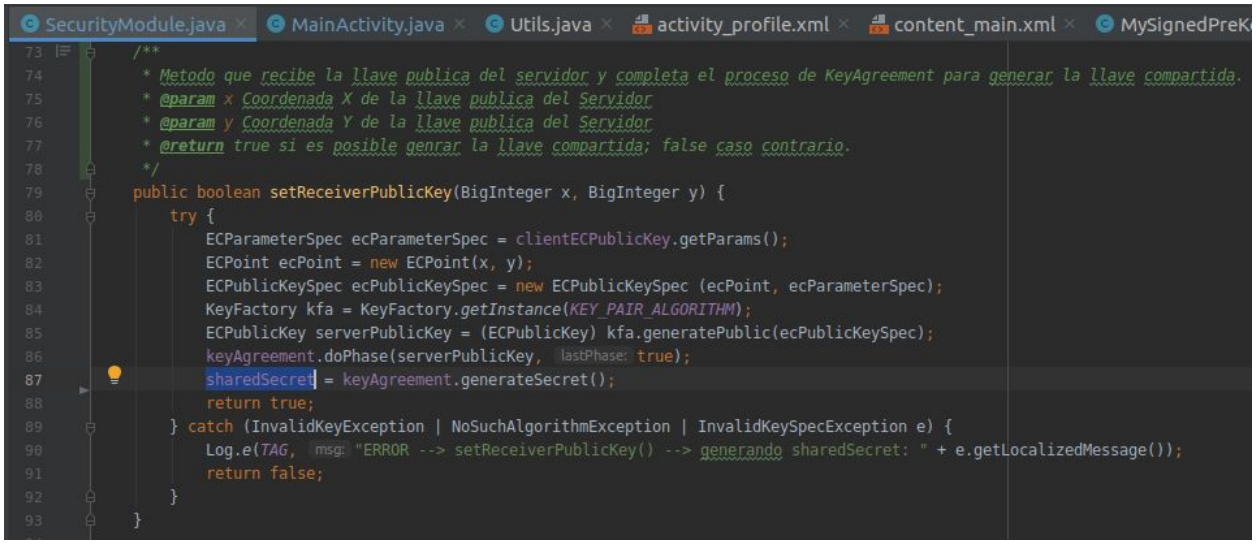

Imagen 50: Método de generación de llave compartida NAFAA

Teniendo el secreto compartido entre los pares es posible inicializar el módulo de encriptación con *AES* que recibe como parámetro la llave compartida y permite encriptar textos planos que reciba el método de “encrypt” como parámetro.

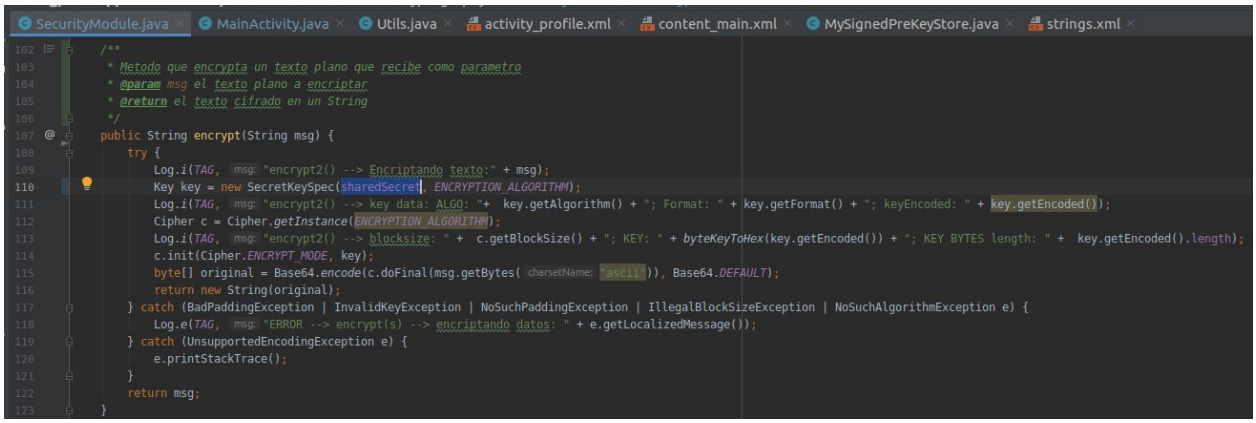

Imagen 51: Método de encriptado NAFAA

Del mismo modo, se inicializa la misma librería de Cipher en modo decrypt con la llave compartida para poder desencriptar los mensajes cifrados recibidos. Por defecto, si la

instancia de Cipher especificada sólo se llama con el parámetro *AES*, Java usa *AES* en modo ECB y no es necesario indicar un vector de inicialización.

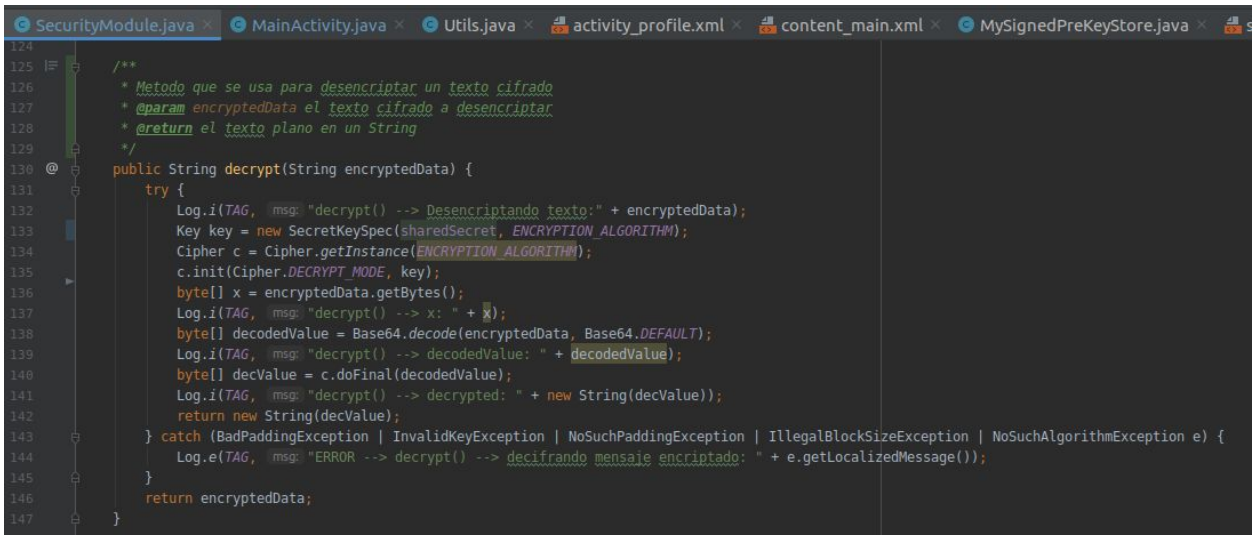

Imagen 52: Mobile - Método de desencriptado NAFAA

Para la parte gráfica se integró un nuevo botón en el menú inicial de login de la aplicación. El botón de “Encrypted Login” es el que hace uso de la nueva funcionalidad donde en primera instancia llama al servicio de “/ecdh”, para realizar el intercambio de llaves públicas y poder generar la llave compartida para poder luego hacer uso del servicio “/login_encrypted” expuesto por el servicio. Ya cuando se consume este último servicio la información viaja encriptada en el parámetro “encrypted_data”

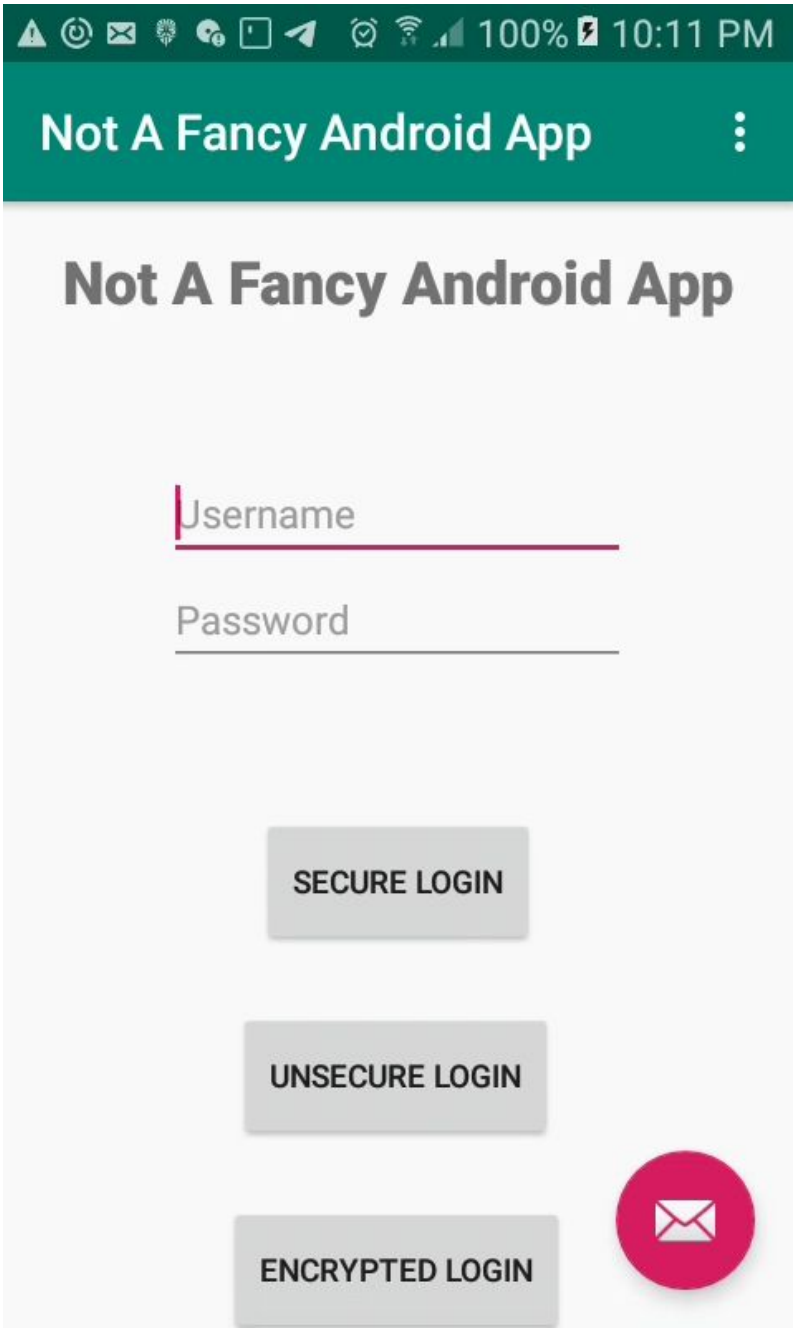

Imagen 53: Mobile - Nuevo botón funcionalidad encriptado NAFAA

5.2.3. Pruebas funcionales

Para realizar las pruebas funcionales y con el fin de mostrar cómo viaja la información cifrada a nivel de capa aplicación, este vector no tiene activada el mecanismo de protección de SSL Pinning mostrado en el Primer Vector. Sin esta protección activada es posible realizar la captura de solicitudes mediante el proxy intermedio y demostrar el cómo se lleva a cabo el intercambio de información entre la aplicación y el *WS*.

La topología para esta prueba mantiene la topología igual a la usada en el desarrollo práctico del Primer Vector para el caso de tener el proxy activado. Teniendo esto en cuenta, las pruebas se realizaron teniendo la misma configuración en el dispositivo y las mismas herramientas usadas en el Primer Vector para capturar las solicitudes entre el dispositivo y el backend.

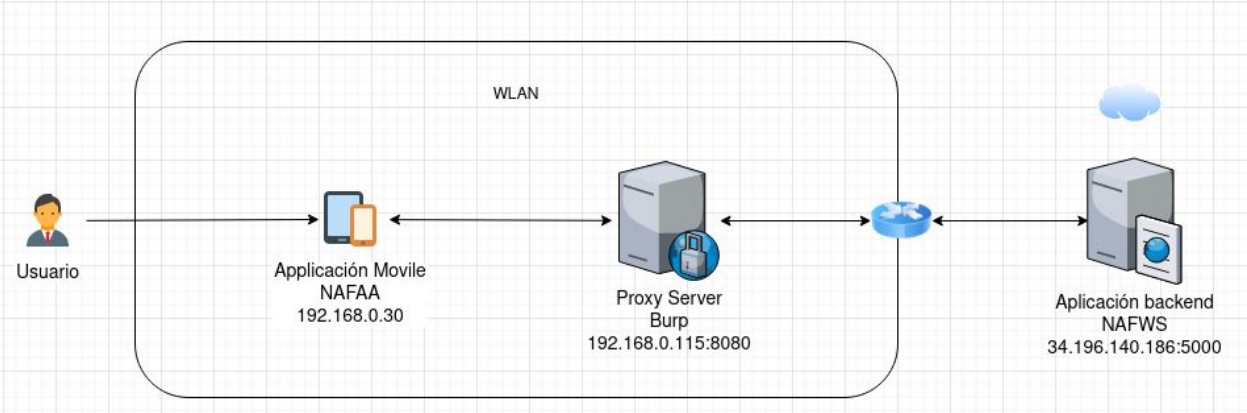

Imagen 54: Topología Tercer Vector proxy activado

Inicialmente se inicializa el servidor que expone los *WS* y este genera las llaves tanto pública como privada y deja la llave pública a disposición de solicitudes entrantes. Hay que tener en cuenta que la información logueada es para un servicio de prueba y bajo un ambiente productivo como buena práctica se sugiere evitar loguear este tipo de información.

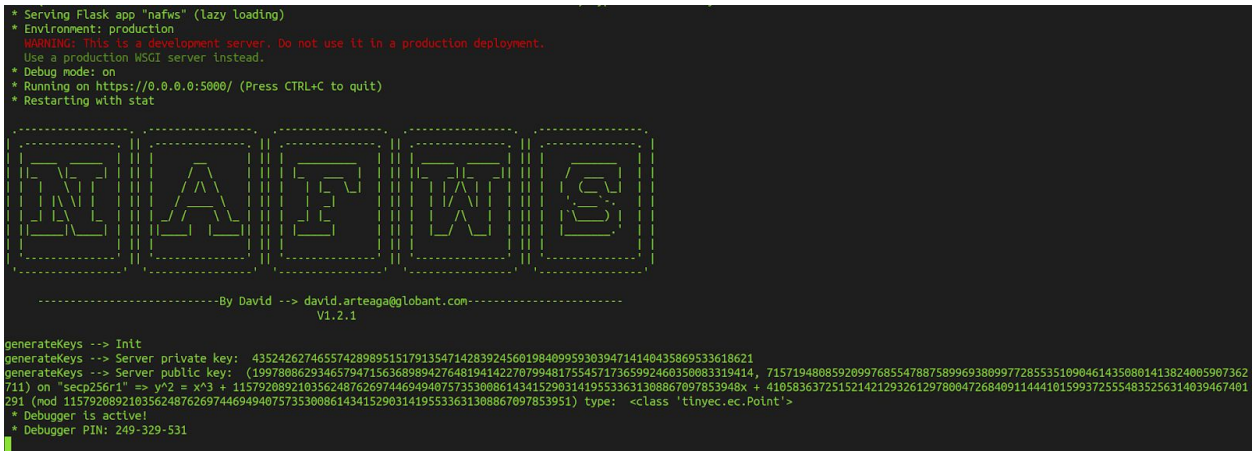

Imagen 55: Inicialización del servidor Llave pública y privada visible

Del mismo modo en la parte de la aplicación mobile se puede ver en el log como se generan la llaves antes de enviar la solicitud y se puede ver que se agrega la coordenada del punto que representa la llave pública generada a la solicitud POST del *WS* en la ruta “/ecdh” con el fin de realizar el intercambio de llaves y generar una llave secreta compartida. Este devuelve la llave pública que se generó al momento de la inicialización.

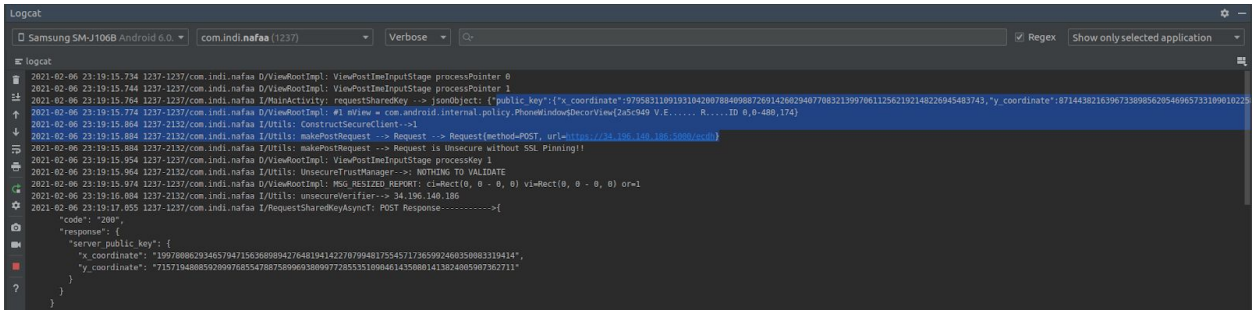

Imagen 56: Log Mobile - Request desde mobile a ws /ecdh enviando la llave pública recibiendo la publica del servidor

Con estos datos cada parte puede generar la llave secreta compartida como se ha explicado anteriormente con el método de *ECDH*, tanto en la aplicación móvil como en el servidor.

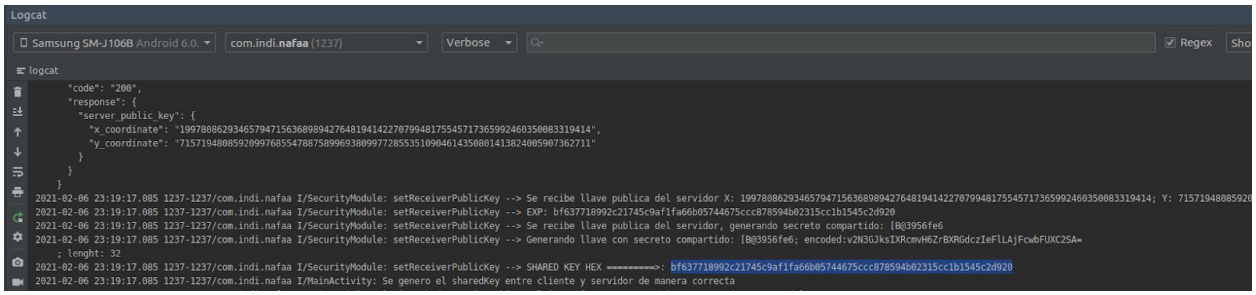

Imagen 57: Log Mobile - Generación de secreto compartido

Imagen 58: Log WS - Generación de secreto compartido

Desde el proxy se puede ver la solicitud con la llave pública del cliente en el request POST y la llave pública del servidor como respuesta a esta solicitud. Nótese que antes de enviar la respuesta el servidor ya generó el secreto compartido y debido a esto retorna

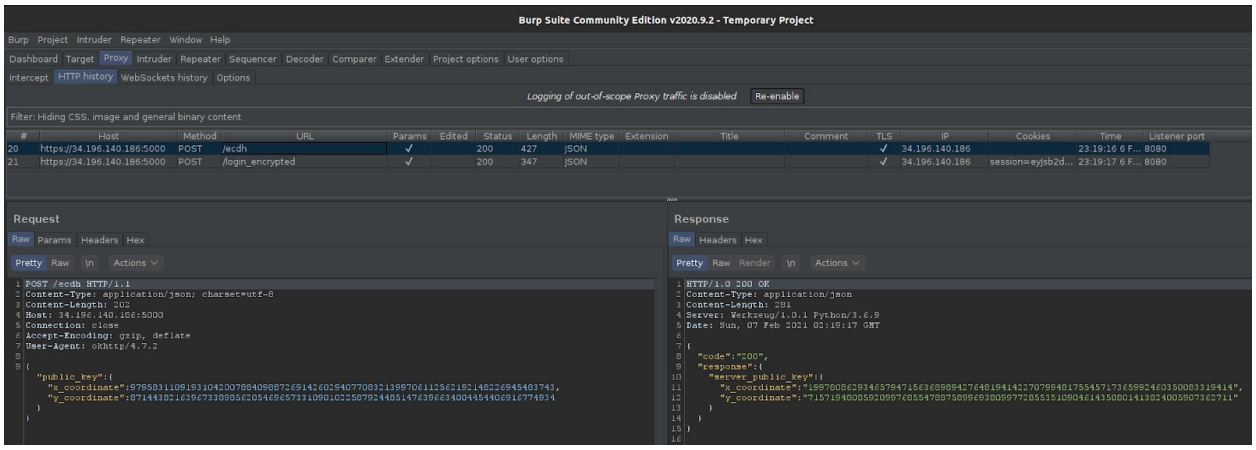

Imagen 59: Proxy - Captura request /ecdh llaves publicas

Ambas partes teniendo el mismo secreto compartido se pueden empezar a enviar información cifrada con el algoritmo escogido AES el cual tiene en cuenta el secreto compartido como llave de inicialización para el proceso de encriptación y desencriptación. Para el caso de la PoC se envía la información de “username” y “password” ingresados en la aplicación de manera cifrada para que sean evaluados por el backend para el servicio expuesto en la ruta “/login_encrypted”.

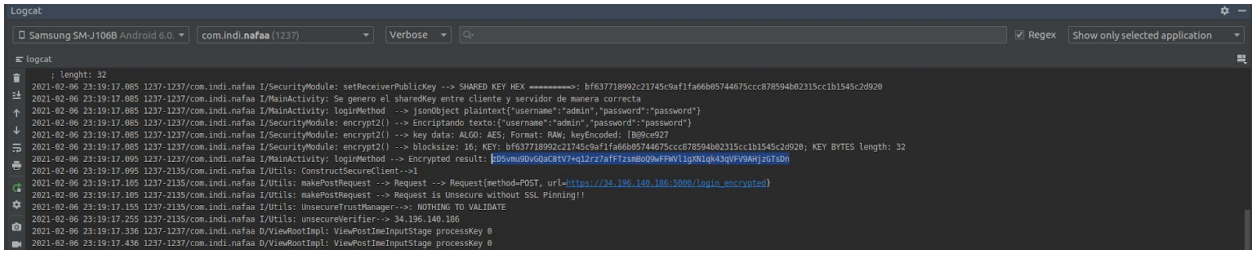

Imagen 60: Log Mobile - Encriptación de información de solicitud de login

El servidor recibe la solicitud hecha en la ruta “/login_encrypted” y procede a desencriptar el mensaje recibido con la llave secreta compartida y usando el mismo algoritmo AES. Una vez de desencripta la

información se valida los parámetros enviados y se genera una respuesta. Al igual que la solicitud, la respuesta se encripta en AES teniendo en cuenta la llave secreta compartida.

Imagen 61: Log WS - Procesamiento solicitud encriptada

La aplicación móvil recibe la respuesta encriptada del servicio y procede a desencriptarla para poder determinar el resultado de los parámetros enviados por el usuario. En caso de ser correctos se redirige al usuario a la pantalla de bienvenida; caso contrario se notifica que los parámetros ingresados no son válidos.

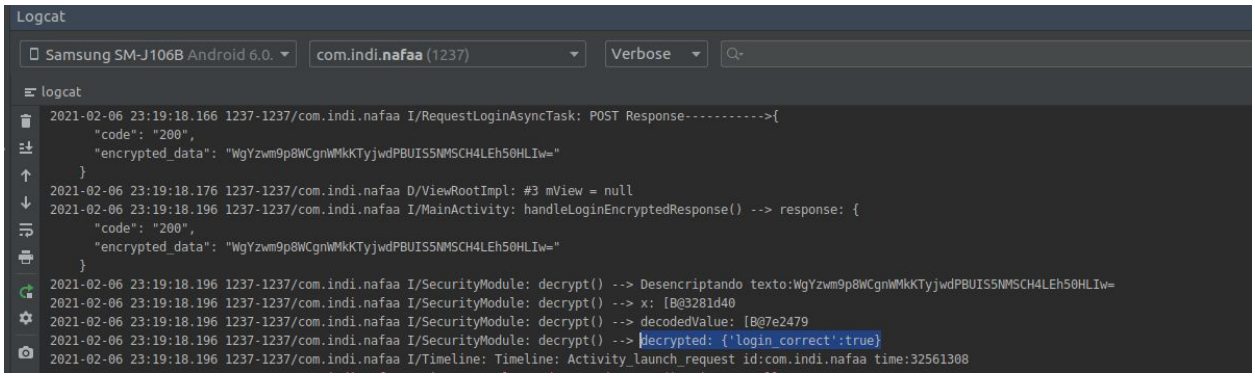

Imagen 62: Log Mobile - Procesamiento respuesta de solicitud encriptada

Por último, se puede evidenciar en la captura de solicitudes desde proxy la información enviada entre parte de manera cifrada con la convención JSON planteada desde el diseño de la aplicación para la ruta “/encrypted_login”.

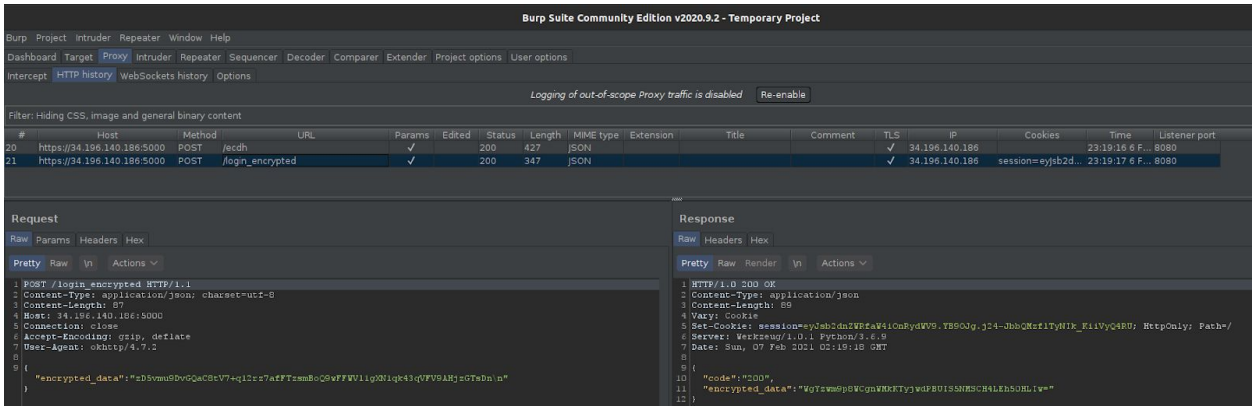

Imagen 63: Proxy - Captura de request “/encrypted_login”

Para concluir este vector, si un atacante es capaz de interceptar las solicitudes que envía la aplicación no va a poder manipular los parámetros enviados para generar ataques automatizados, debido a que la información que va a procesar el servidor se encuentra encriptada y si este la modifica recibirá un error del servidor. Además de no poder generar una solicitud que pueda desencriptar si no conoce la llave secreta compartida así envie una solicitud con los parámetros encriptados con *AES*.

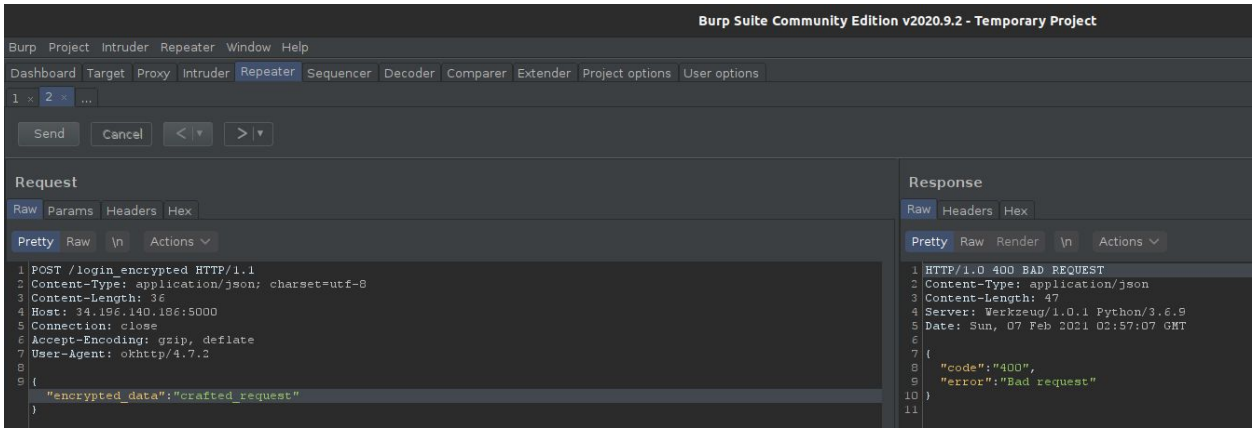

Imagen 64: Proxy - Envio solicitud “encrypted_login” manual con parámetros aleatorios

Imagen 65: Log WS - Procesamiento de solicitudes erróneas con parámetros aleatorios

6. Repositorios

Los repositorios de la prueba de concepto generada se encuentran de manera pública en un repositorio de Github. Se puede descargar el código directamente desde los links de repositorios o usar herramientas de versionado git para la descarga y sincronización de repositorios.

- ***NAFAA***: https://github.com/choringa/NAFAA_pub
- ***NAFWS***: https://github.com/choringa/NAFWS_pub

Recuerde que ***NAFAA*** es un proyecto desarrollado en Java - Android Nativo y se desarrolló en el IDE de Android Studio. Mientras que ***NAFWS*** es un proyecto desarrollado en Python y no es necesario un IDE específico para su edición.

7. Conclusiones

Teniendo en cuenta el objetivo principal se deja, de manera satisfactoria y detallada, documentación y aportes prácticos dentro del ámbito de desarrollo seguro de aplicaciones móviles y consumo de servicios. Del mismo modo se deja evidencia de tres vectores de ataque y protección a tener en cuenta a la hora de desarrollar y desplegar una arquitectura de este tipo. Se muestra a través de los casos de prueba, desarrollos prácticos y pruebas funcionales la mitigación de amenazas que se pueden presentar en aplicaciones móviles y su interacción con los servicios dispuestos para su uso.

De la misma manera se realizó de manera práctica casos reales para vectores comunes de ataque en las aplicaciones móviles. Para poder llevar esto a cabo se desarrolló un servicio que expone diferentes web services para demostrar los objetivos prácticos de cada vector propuesto. Además de los servicios web, también se desarrolló una aplicación móvil capaz de interactuar con dicho servicio y complementar los objetivos específicos propuestos de desarrollo y prueba en cada vector.

Para el primer vector mostrado, hay que tener en cuenta que el control de seguridad de *SSL Pinning* es sencillo de implementar y representa una medida extra para un atacante, haciendo más difícil la captura de las solicitudes enviadas al *WS* con el cual se comunica la aplicación. Sin embargo, existen diversos mecanismos que permiten sobrepasar esta medida, como técnicas de inyección de código en tiempo de ejecución con herramientas como FRIDA o ingeniería inversa que permiten la recopilación de la aplicación cambiando el hash del certificado, por ejemplo. Aunque son técnicas muy avanzadas y requieren una inversión de tiempo, esfuerzo y conocimientos por parte de un atacante es recomendable seguir implementando la medida de control de *SSL Pinning* en las aplicaciones móviles para dificultar el proceso de captura de solicitudes y poder proteger las solicitudes que realiza la aplicación hacia su *WS*.

Dentro del desarrollo del segundo vector se deja evidencia de cómo mediante un proceso de ingeniería inversa un atacante puede llegar a ver el código original con el cual se compiló una aplicación, en este caso Android, con el fin de entender el código de la misma y poder evaluar sus funcionalidades y posiblemente saltear controles de seguridad implementados en esta. Uno de los controles que ayuda a complicar la lectura de código de una aplicación es mediante la ofuscación de código, que como bien se demostró es bastante sencillo de implementar y genera un mayor nivel de entropía en el código resultante cuando se hace un proceso de ingeniería inversa sobre una aplicación que ha pasado por este proceso.

Para el tercer vector, se muestra de manera efectiva el uso de la encriptación simétrica para la comunicación entre pares cifrada a nivel de aplicación con el algoritmo AES y cómo generar una llave secreta compartida para poder iniciar el proceso de encriptación entre los pares y establecer una comunicación con información cifrada. Se mostró de manera efectiva cómo es posible generar un secreto compartido entre pares

usando protocolos de acuerdo de llaves como *ECDH* entre 2 lenguajes de programación diferentes.

Por último, se expusieron a lo largo de todo el documento, diferentes conceptos de seguridad como Autoridad Certificadora (CA), ofuscamiento de código, SSL Pinning, cifrado de datos dentro de una parte teórica y se hicieron uso de estos conceptos expuestos en las partes teóricas. Dejando de este modo un documento base de buenas prácticas de desarrollo con una prueba de concepto funcional que integra diferentes conceptos de seguridad tanto en aplicaciones móviles como en servicios web.

8. Glosario

- WS: Web Service.
- CA: Certificate Authority.
- SSL: Secure Socket Layer.
- TLS: Transport Layer Security.
- APK: Formato aplicaciones móviles Android.
- IPA: Formato aplicaciones móviles iOS.
- REST: Representation State Transfer; tipo de arquitectura de servicios web.
- VPN: Virtual Private Network.
- DNS: Domain Name System.
- HTTP: Hypertext Transfer Protocol.
- Endpoint, Backend: Punto final de transferencia de datos (servidor de servicios).
- PoC: Proof of Concept, Prueba de Concepto.
- SSL Pinning: Mecanismo de protección por validación de certificados.
- IDE: Integrated Development Environment.

- Java: Lenguaje de programación orientado a objetos
- Python: Lenguaje de programación tipado.
- Swift: Lenguaje de programación usado para el desarrollo de aplicaciones iOS.
- Objective-C: Lenguaje de programación usado para el desarrollo de aplicaciones iOS.
- JSON: JavaScript Object Notation; Formato de notación.
- NAFAA: Aplicación Android de prueba.
- NAFWS: Web Service Python de prueba.
- Okhttp3: Librería cliente de envío de solicitudes Android.
- Burp Suite: Proxy usado para pruebas.
- Fuzzing: Tipos de ataque de prueba con parámetros inválidos.
- Fuerza bruta: Ataque de prueba y error con una lista con un número extenso de valores para probar.
- DEX: Dalvik Executable File; archivos binarios de un APK.
- SMALI: Archivo resultante tras realizar ingeniería inversa.
- ProGuard: Programa usado para ofuscación de código en Android.
- iXGuard: Programa usado para ofuscación de código en iOS.
- DexGuard: Programa usado para ofuscación de código en Android.
- SSH: Secure Shell.
- Dex2jar: Programa para realizar proceso de ingeniería inversa.
- API: Application Programming Interface.
- AES: Advanced Encryption Standard.
- ECDF: Elliptic Curve Diffie-Hellman.
- ECC: Elliptic curve cryptography.

9. Bibliografía

[1] Walton, J., Steven, J., Manico, J. and Wall, K. (2018). *Certificate and Public Key Pinning* [online] Disponible en: https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning [Accedido 10 May. 2020].

[2] Dolan, M. (2017). *Android Security: SSL Pinning.* [online] Disponible en: https://medium.com/@appmattus/android-security-ssl-pinning-1db8acb6621e [Accedido 12 May. 2020].

[3] Developers.android.com. (2020). *Cómo reducir, ofuscar y optimizar tu app.* [online] Disponible en: https://developer.android.com/studio/build/shrink-code [Accedido 28 May. 2020].

[4] Kaliciński, W. (2018). *Practical ProGuard rules examples.* [online] Disponible en: https://medium.com/androiddevelopers/practical-proguard-rules-examples-5640a3907dc9 [Accedido 22 May. 2020].

[5] Kriuchkov, T. (2019). *Mobile cybersecurity encryption on Android.* [online] Disponible en: https://www.apriorit.com/dev-blog/612-mobile-cybersecurity-encryption-in-android [Accedido 26 May. 2020].

[6] Grinberg, M. (2017). *Running Your Flask Application Over HTTPS.* [online] Disponible en: https://blog.miguelgrinberg.com/post/running-your-flask-application-over-https [Accedido 21 Jun. 2020].

[7] Buck, A. (2019). *How & What Are SSL Certificates Used For?.* [online] Disponible en: https://www.verticalrail.com/kb/ssl-certificates/ [Accedido 23 Jun. 2020].

[8] Digicert. (2020). *¿Qué son SSL, TLS y HTTPS?.* [online] Disponible en: https://www.websecurity.digicert.com/es/mx/security-topics/what-is-ssl-tls-https [Accedido 23 Jun. 2020].

[9] Fisher, T. (2020). *What Is an APK File?.* [online] Disponible en: https://www.lifewire.com/apk-file-4152929 [Accedido 17 Ago. 2020].

[10] File-Extension.org. (2020). *What is apk file? How to open apk files?.* [online] Disponible en: https://www.file-extensions.org/apk-file-extension [Accedido 17 Ago. 2020].

[11] Source.android.com. (2020). *Dalvik Executable format.* [online] Disponible en: source.android.com/devices/tech/dalvik/dex-format.html [Accedido 19 Ago. 2020].

[12] Tumbleson, C., Wiśniewski, R. (2019). *Apktool: A tool for reverse engineering Android apk files.* [online] Disponible en: https://ibotpeaches.github.io/Apktool/ [Accedido 19 Ago. 2020].

[13] Guardsquare.com. (2017). *DexGuard vs. ProGuard.* [online] Disponible en: https://www.guardsquare.com/en/blog/dexguard-vs-proguard [Accedido 11 Sep. 2020].

[14] Pan, B. (2019). Dex2Jar: Tools to work with android .dex and java .class. [online] Disponible en: https://github.com/pxb1988/dex2jar/ [Accedido 11 Sep. 2020].

[15] OphoneSDN.com. (2010). The Structure of Android Package (APK) [online] Disponible en: https://web.archive.org/web/20110208193918/http://en.ophonesdn.com/article/show/354 [Accedido 29 Sep. 2020].

[16] Developer.android.com. (2020). Enable multidex for apps with over 64K methods [online] Disponible en: https://developer.android.com/studio/build/multidex [Accedido 5 Oct. 2020].

[17] Panhalkar, S. (2020). Android Default ProGuard Rules [online] Disponible en: https://proandroiddev.com/android-default-proguard-rules-guide-20058ba7a486 [Accedido 15 Oct. 2020].

[18] Developer.android.com. (2020). Cryptography. [online] Disponible en: https://developer.android.com/guide/topics/security/cryptography [Accedido 15 Oct. 2020].

[19] Nakov, S. (2018). Practical Cryptography for Developers. [online] Disponible en: https://wizardforcel.gitbooks.io/practical-cryptography-for-developers-book/content/asymmetric-key-ciphers/elliptic-curve-cryptography-ecc.html [Accedido 17 Oct. 2020].

[20] Sullivan, N. (2013). A primer on elliptic curve cryptography. [online] Disponible en: https://arstechnica.com/information-technology/2013/10/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/2/ [Accedido 19 Oct. 2020].

[21] Buchanan, B. (2019). Handshaking Keys for Privacy with ECDH … Let’s “Go” Create It!. [online] Disponible en:

https://billatnapier.medium.com/little-protects-you-on-line-like-ecdh-lets-go-create-it-a14188eabded [Accedido 19 Nov. 2020].

[22] Adalier, M y Teknik, A. (2019). Efficient and Secure ECC Implementa Ronof CurveP--256. [online] Disponible en: https://csrc.nist.gov/csrc/media/events/workshop-on-elliptic-curve-cryptography-standards/documents/presentations/session6-adalier-mehmet.pdf [Accedido 22 Dic. 2020].

[23] National Institute of Standards and Technology. (2013). FIPS PUB 186-4 Digital Signature Standard. [online] Disponible en: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf [Accedido 22 Dic. 2020].

[24] Nakov, S. (2018). Practical Cryptography for Developers. [online] Disponible en: https://cryptobook.nakov.com/asymmetric-key-ciphers/ecc-encryption-decryption [Accedido 22 Dic. 2020].

[25] Eddie, R. (2015). How does ECDH arrive on a shared secret?. [online] Disponible en: https://crypto.stackexchange.com/questions/21169/how-does-ecdh-arrive-on-a-shared-secret [Accedido 22 Dic. 2020].

[26] Legrandin, G. (2019). ECC. [online] Disponible en: https://pycryptodome.readthedocs.io/en/latest/src/public_key/ecc.html [Accedido 27 Dic. 2020].

[27] Developer.android.com. (2020). KeyGenParameterSpec. [online] Disponible en: https://developer.android.com/guide/topics/security/cryptography [Accedido 4 Ene. 2021].

[28] Brown, D. (2010). SEC 2: Recommended Elliptic Curve Domain Parameters. [online] Disponible en: https://www.secg.org/sec2-v2.pdf [Accedido 21 Dic. 2020].

[29] Cryptosys.net. (2020). More details on Elliptic Curve Cryptography (ECC). [online] Disponible en: https://www.cryptosys.net/pki/eccrypto.html [Accedido 23 Dic. 2020].

[30] Oracle. (2020). Java Security Standard Algorithm Names. [online] Disponible en: https://docs.oracle.com/en/java/javase/15/docs/specs/security/standard-names.html [Accedido 15 Ene. 2021].

[31] Brown, D y tsif. (2020). ECDSA secp256r1. [online] Disponible en: https://github.com/forevertz/ecdsa-secp256r1 [Accedido 15 Ene. 2021].