

Práctica 4

Análisis dinámico de apps Android

Tabla de contenido

1. Objetivos	3
2. Ejercicios.....	3
2.1. Actuar sobre apps depurables (android:debuggable="true").....	3
2.2. Instrumentación dinámica de código con Frida	6
2.2.1. Baipasear la pantalla de Login	6
2.2.2. Almacenamiento inseguro de credenciales	6
2.2.3. Baipasear la detección de root.....	7

1. Objetivos

En esta práctica vamos a abordar el estudio de la app InsecureBankv2, ya introducida en la práctica anterior, desde una perspectiva dinámica, es decir, ejercitando la aplicación y actuando sobre la misma en tiempo de ejecución.

Para ello utilizaremos tanto el depurador de Java, *jdb*, como *Frida*, una herramienta de inyección de código que ya hemos estudiado en teoría.

2. Ejercicios

La práctica tiene 4 ejercicios, el primero explotará las posibilidades que ofrece el modo depuración cuando la aplicación lo tiene activo en su manifiesto, los otros tres plantearán retos que, aunque puedan plantearse y resolverse utilizando otros medios, serán resueltos utilizando Frida en esta práctica.

2.1. Actuar sobre apps depurables (android:debuggable="true")

Aunque este punto se ha comentado en clase utilizando Android Studio, o combinando el uso de este IDE con Smalidea, en este ejercicio mostraremos como servirnos del depurador de Java, *jdb* (del inglés Java DeBugger). Como este aspecto no se ha ejemplificado en teoría, se abordará en este ejercicio como un tutorial, planteándose finalmente una pequeña actividad que deberá realizarse y documentarse.

La app InsecureBankv2 se define en su manifiesto como una app depurable (flag *android:debuggable="true"*) tal y como se muestra a continuación:

```
14 <uses-feature android:glEsVersion="0x00020000" android:required="true"/>
15 <application android:allowBackup="true" android:debuggable="true" android:icon="@mipmap/ic_launcher" and
16 <activity android:label="@string/app_name" android:name="com.android.insecurebankv2.LoginActivity">
17 <intent-filter>
18 <action android:name="android.intent.action.MAIN"/>
19 <category android:name="android.intent.category.LAUNCHER"/>
20 </intent-filter>
21 </activity>
```

Podemos aprovechar este hecho para utilizar un depurador e investigar el comportamiento de la aplicación y descubrir alguna información interesante. Utilizaremos *jdp* (Java Debugger) que se instala al instalar la JDK de Java. Seguiremos los siguientes pasos:

1. Averiguaremos el PID de la app que deseamos estudiar. Para ello utilizaremos la orden:

```
adb jdwp
```

Anotaremos el PID que aparecerá por consola. El problema es que si en el dispositivo/emulador hay más de una app depurable aparecerá más de un PID, con lo que tendremos que adivinar cual es el asociado a app que nos interese. Si esto ocurre recurriremos a la orden:

```
adb shell ps
```

Esta orden nos proporciona un largo listado de aplicaciones y procesos en el que buscaremos la línea asociada a la app bajo estudio y tomaremos nota de su PID tal y como sigue:

```
root      14457      2      0      0 0      0 S [kworker/4:1]
u0_a9     14535    1787 4118968 221440 0      0 S com.google.android.gms.persistent
u0_a168   14555    1787 4443820 149476 0      0 S com.android.insecurebankv2
root      14584      2      0      0 0      0 S [kbase_event]
```

2. A continuación, enlazaremos el puerto en el que ejecutaremos el depurador en nuestra máquina con el PID del proceso con el que interactuaremos en el dispositivo o emulador con el que trabajemos, y que hemos determinado en el paso anterior. Asumiendo que el puerto de nuestra máquina es el puerto 7777 y el InsecureBankv2 se ejecuta con un PID de 14555 (según hemos visto anteriormente), utilizaremos la orden:

```
adb forward tcp:7777 jdwp:14555
```

3. Finalmente, lanzaremos a ejecución el depurador en nuestra máquina y lo asociaremos con la app en el emulador/dispositivo mediante la orden:

```
jdb -attach localhost:7777
```

En las máquinas Windows esta orden puede fallar generando una excepción similar a la siguiente:

```
C:\Users\jucar>jdb -attach localhost:7777
java.io.IOException: shmemBase_attach failed: El sistema no puede encontrar el archivo especificado

    at com.sun.tools.jdi.SharedMemoryTransportService.attach0(Native Method)
    at com.sun.tools.jdi.SharedMemoryTransportService.attach(SharedMemoryTransportService.java:108)
    at com.sun.tools.jdi.GenericAttachingConnector.attach(GenericAttachingConnector.java:116)
    at com.sun.tools.jdi.SharedMemoryAttachingConnector.attach(SharedMemoryAttachingConnector.java:63)
    at com.sun.tools.example.debug.tty.VMConnection.attachTarget(VMConnection.java:519)
    at com.sun.tools.example.debug.tty.VMConnection.open(VMConnection.java:328)
    at com.sun.tools.example.debug.tty.Env.init(Env.java:63)
    at com.sun.tools.example.debug.tty.TTY.main(TTY.java:1082)

Fatal error:
Unable to attach to target VM.
```

En ese caso utilizaremos la orden:

```
C:\Users\jucar>jdb -connect com.sun.jdi.SocketAttach:hostname=localhost,port=7777
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
>
```

El símbolo “>” nos indica que el depurador está ya en ejecución y que podemos comenzar con el proceso de estudio de la app.

Para listar los métodos de la clase *com.android.insecurebankv2.LoginActivity* procederemos como sigue:

```
> methods com.android.insecurebankv2.LoginActivity
** methods list **
com.android.insecurebankv2.LoginActivity <init>()
com.android.insecurebankv2.LoginActivity callPreferences()
com.android.insecurebankv2.LoginActivity createUser()
com.android.insecurebankv2.LoginActivity fillData()
com.android.insecurebankv2.LoginActivity onCreate(android.os.Bundle)
com.android.insecurebankv2.LoginActivity onCreateOptionsMenu(android.view.Menu)
com.android.insecurebankv2.LoginActivity onOptionsItemSelected(android.view.MenuItem)
com.android.insecurebankv2.LoginActivity performLogin()
```

Vemos que la clase contiene un método interesante, el método *createUser()*, pongamos un breakpoint en dicho método usando la orden:

```
> stop com.android.insecurebankv2.LoginActivity.createUser()
Usage: stop at <class>:<line_number> or
      stop in <class>.<method_name>[(argument_type,...)]
> stop in com.android.insecurebankv2.LoginActivity.createUser()
Set breakpoint com.android.insecurebankv2.LoginActivity.createUser()
>
```

En la práctica anterior se planteaba como ejercicio 3 la búsqueda de un botón oculto en la pantalla de Login. Si se ha encontrado dicho botón, podremos pulsarlo y activar el método `createUser()`, lo que nos llevará a activar el breakpoint que hemos puesto en el método. Cuando esto suceda, utilizaremos la orden `step` para ejecutar la siguiente línea de código que, como vemos, ejecuta el método `android.widget.Toast.makeText()`. La orden `“locals”` nos permitirá ver las variables locales existentes en cualquier punto del código. En el asociado al método bajo estudio tendremos lo que sigue:

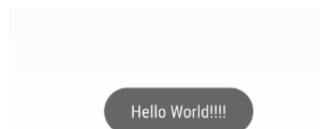
```
main[1] step
>
Step completed: "thread=main", android.widget.Toast.makeText(), line=260 bci=0

main[1] locals
Method arguments:
context = instance of com.android.insecurebankv2.LoginActivity(id=9369)
text = "Create User functionality is still Work-In-Progress!!"
duration = 1
Local variables:
```

Lo interesante es que es posible manipular dichas variables utilizando la orden `“set”` tal y como se muestra a continuación:

```
main[1] set text = "Hello World!!!!"
text = "Hello World!!!!" = "Hello World!!!!"
main[1] run
```

La orden `“cont”` permite continuar con la ejecución del programa hasta la siguiente interacción con el usuario o el siguiente punto de ruptura en el programa. En nuestro caso, la modificación personaliza el texto que muestra un Toast como el siguiente.



La siguiente tabla muestra las órdenes más importantes que se pueden utilizar y explica su uso:

Name	Description
help or ?	The most important JDB command; it displays a list of recognized commands with a brief description.
run	After starting JDB and setting the necessary breakpoints, you can use this command to start execution and debug an application.
cont	Continues execution of the debugged application after a breakpoint, exception, or step.
print	Displays Java objects and primitive values.
dump	For primitive values, this command is identical to print. For objects, it prints the current value of each field defined in the object. Static and instance fields are included.
threads	Lists the threads that are currently running.
thread	Selects a thread to be the current thread.
where	Dumps the stack of the current thread.

Esta tabla está extraída del tutorial <https://www.tutorialspoint.com/jdb>. Más información sobre JDB en <https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html>.

Sigue los pasos planteados en este ejercicio y realiza una captura de pantalla en la que se vea a la app **mostrando un Toast un mensaje con el nombre de los integrantes del grupo**. En el informe que entregues documenta cuál es el texto que originalmente mostraba la app y realiza una captura de pantalla mediante la que mostrarás a la app mostrando el nuevo texto solicitado. No te limites a capturar sólo el Toast, realiza una captura de toda la pantalla del emulador o dispositivo utilizado.


2.2. Instrumentación dinámica de código con Frida

2.2.1. Baipasear la pantalla de Login

Este es un ejercicio que ya planteamos en la práctica anterior, y que ahora se propone que resolvamos utilizando Frida. Revisa las transparencias de teoría y los ficheros JS que tienes a tu disposición en poliformaT para implementar un JS que te permita baipasear la pantalla de login de la app.

2.2.2. Almacenamiento inseguro de credenciales

La actividad de login permite que los usuarios (auto)rellenen sus credenciales para no tener que introducirlas cada vez que acceden a la app. La siguiente captura muestra el método `fillData`, que es invocado cuando el usuario pulsa (`onClick()`) el botón con texto “Autofill Credentials”.



```

31  /* access modifiers changed from: protected */
32  public void onCreate(Bundle savedInstanceState) {
33      super.onCreate(savedInstanceState);
34      setContentView(R.layout.activity_log_main);
35      if (getResources().getString(R.string.is_admin).equals("no")) {
36          findViewById(R.id.button_CreateUser).setVisibility(8);
37      }
38      this.login_buttons = (Button) findViewById(R.id.login_button);
39      this.login_buttons.setOnClickListener(new View.OnClickListener() {
40          /* class com.android.insecurebankv2.LoginActivity.AnonymousClass1 */
41          public void onClick(View v) {
42              LoginActivity.this.performlogin();
43          }
44      });
45      this.createuser_buttons = (Button) findViewById(R.id.button_CreateUser);
46      this.createuser_buttons.setOnClickListener(new View.OnClickListener() {
47          /* class com.android.insecurebankv2.LoginActivity.AnonymousClass2 */
48          public void onClick(View v) {
49              LoginActivity.this.createUser();
50          }
51      });
52      this.fillData_button = (Button) findViewById(R.id.fill_data);
53      this.fillData_button.setOnClickListener(new View.OnClickListener() {
54          /* class com.android.insecurebankv2.LoginActivity.AnonymousClass3 */
55          public void onClick(View v) {
56              try {
57                  LoginActivity.this.fillData();
58              } catch (UnsupportedEncodingException | InvalidAlgorithmParameterException | InvalidKeyException e) {
59                  e.printStackTrace();
60              }
61          }
62      });
63      /* access modifiers changed from: protected */
64      public void createUser() {
65          Toast.makeText(this, "Create User functionality is still Work-In-Progress!!", 1).show();
66      }
67      /* access modifiers changed from: protected */
68      public void fillData() throws UnsupportedEncodingException, InvalidKeyException, NoSuchAlgorithmException,
69          SharedPreferences settings = getSharedPreferences("mySharedPreferences", 0);
70          String username = settings.getString("EncryptedUsername", null);
71          String password = settings.getString("superSecurePassword", null);
72          if (username != null && password != null) {
73              try {
74                  this.usernameBase64ByteString = new String(Base64.decode(username, 0), "UTF-8");
75              } catch (UnsupportedEncodingException e) {
76                  e.printStackTrace();
77              }
78              this.Username_Text = (EditText) findViewById(R.id.loginscreen_username);
79              this.Password_Text = (EditText) findViewById(R.id.loginscreen_password);
80              this.Username_Text.setText(this.usernameBase64ByteString);
81              this.Password_Text.setText(new CryptoClass().aesDecryptedString(password));
82          } else if (username == null || password == null) {
83              Toast.makeText(this, "No stored credentials found!!", 1).show();
84          } else {
85              Toast.makeText(this, "No stored credentials found!!", 1).show();
86          }
87      }
88  }

```

Por lo que podemos ver en el método *fillData()* las credenciales que se guardan en el fichero *mySharedPreferences.xml*. Tal y como muestra la línea de código 80 el nombre del usuario esta codificado con un cifrado Base64, mientras que la contraseña lo está con un cifrado AES con Cipher Block Chaining (CBC) inicializado con un vector y una clave de cifrado codificados como atributos de la clase *CryptoClass*. Con esto ya se podría obtenerse la contraseña almacenada en el fichero de preferencias, pero lo que se propone en este ejercicio es hacerlo utilizando Frida. Nuestro objetivo es capturar la llamada al método *aesDecryptedString* de la clase *com.android.insecurebankv2.CryptoClass*, modificar su implementación para que, tras realizar la llamada, la contraseña pueda verse sin cifrar. Muestra el código JS del script necesario.

2.2.3. Baipasear la detección de root

Desarrolla una solución basada en Frida para engañar a las pruebas de verificación que despliega la app y conseguir que en lugar de indicar que el dispositivo está ruteado, diga que no lo está.

