# Parte 1: setup di Frida su android (Genymotion)

Si installa il frida-server sul dispostivo emulato android, si scarica dal sito il server e si estrae in una cartella, per installare il server sul device emulato si usa ADB, eseguendo adb devices:

List of devices attached

192.168.157.101:5555 device

Quindi si installa l’applicazione con:

adb push G:\GitHubRepo\AndroidReversing\frida-server-14.2.13-android-x86 /data/local/tmp/frida

si esegue quindi

adb shell

si verifica che Frida sia stato trasferito:

vbox86p:/ # cd /data/local/tmp

vbox86p:/data/local/tmp # ls

frida

vbox86p:/data/local/tmp #

Adesso si danno I permessi al file frida, nello specifico si danno permessi 775:

chmod 775 frida

Quindi si avvia frida

adb.exe shell /data/local/tmp/Frida

Eseguendo il commando:

frida-ps -aU

Si verifia che Frida sta effettivamente funzionando su android

1122 Android Keyboard (AOSP) com.android.inputmethod.latin

1008 Android Services Library android.ext.services

637 Android System android

1312 Blocked Numbers Storage com.android.providers.blockednumber

1843 Calendar Storage com.android.providers.calendar

637 Call Management com.android.server.telecom

1312 Contacts Storage com.android.providers.contacts

1574 Download Manager com.android.providers.downloads

1655 Dynamic System Updates com.android.dynsystem

1716 Email com.android.email

637 Fused Location com.android.location.fused

1698 Key Chain com.android.keychain

1574 Media Storage com.android.providers.media

1771 Messaging com.android.messaging

962 MmsService com.android.mms.service

908 NetworkStack com.android.networkstack

1797 One Time Init com.android.onetimeinitializer

1823 Package installer com.android.packageinstaller

937 Permission controller com.android.permissioncontroller

1289 Phone com.android.dialer

962 Phone Services com.android.phone

962 Phone and Messaging St… com.android.providers.telephony

1677 Print Spooler com.android.printspooler

1048 Quickstep com.android.launcher3

1433 SecureElementApplication com.android.se

969 Settings com.android.settings

637 Settings Storage com.android.providers.settings

1886 Superuser com.genymotion.superuser

1866 System Tracing com.android.traceur

775 System UI com.android.systemui

1753 Work Setup com.android.managedprovisioning

1933 com.android.localtransp… com.android.localtransport

962 com.android.ons com.android.ons

1505 com.android.smspush com.android.smspush

1478 com.genymotion.genyd.… com.genymotion.genyd

1393 com.genymotion.system… com.genymotion.systempatcher

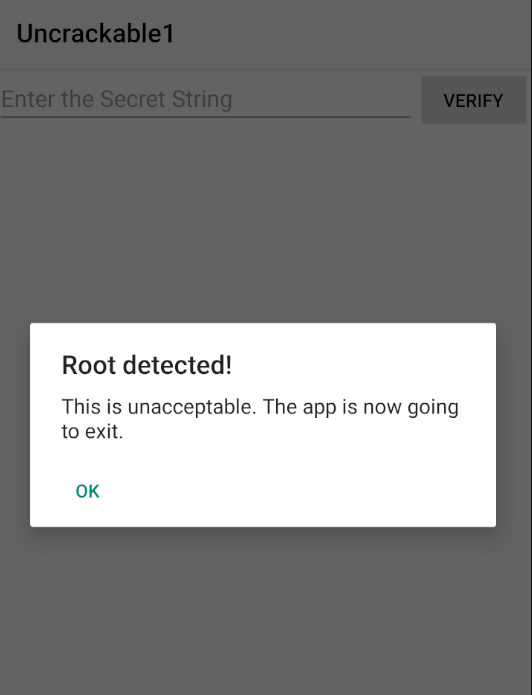
1447 com.genymotion.taskloc… com.genymotion.tasklocker

# Risoluzione del crackme

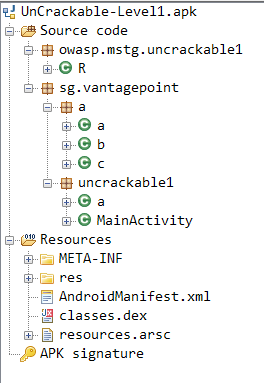
Obiettivo: risolvere owasp mstg 1 per trovare una stringa segreta ed estrarla, si può provare sia con la static analsysis che con la dynamic analysis, in questo caso si procede con analisi dinamica.

Si installa l’applicazione nel device emulato:

adb.exe install G:\GitHubRepo\AndroidReversing\UnCrackable-Level1.apk



Si usa jadx-gui per ispezionare il codice e si nota la seguente struttura:



Il codice di mainActivity:

public class MainActivity extends Activity {

private void a(String str) {

AlertDialog create = new AlertDialog.Builder(this).create();

create.setTitle(str);

create.setMessage("This is unacceptable. The app is now going to exit.");

create.setButton(-3, "OK", new DialogInterface.OnClickListener() {

/\* class sg.vantagepoint.uncrackable1.MainActivity.AnonymousClass1 \*/

public void onClick(DialogInterface dialogInterface, int i) {

System.exit(0);

}

});

create.setCancelable(false);

create.show();

}

/\* access modifiers changed from: protected \*/

public void onCreate(Bundle bundle) {

if (c.a() || c.b() || c.c()) {

a("Root detected!");

}

if (b.a(getApplicationContext())) {

a("App is debuggable!");

}

super.onCreate(bundle);

setContentView(R.layout.activity\_main);

}

public void verify(View view) {

String str;

String obj = ((EditText) findViewById(R.id.edit\_text)).getText().toString();

AlertDialog create = new AlertDialog.Builder(this).create();

if (a.a(obj)) {

create.setTitle("Success!");

str = "This is the correct secret.";

} else {

create.setTitle("Nope...");

str = "That's not it. Try again.";

}

create.setMessage(str);

create.setButton(-3, "OK", new DialogInterface.OnClickListener() {

/\* class sg.vantagepoint.uncrackable1.MainActivity.AnonymousClass2 \*/

public void onClick(DialogInterface dialogInterface, int i) {

dialogInterface.dismiss();

}

});

create.show();

}

}

Si procede ad analizzare l’anderoid manifesT:

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1" android:versionName="1.0" package="owasp.mstg.uncrackable1">

<uses-sdk android:minSdkVersion="19" android:targetSdkVersion="28"/>

<application android:theme="@style/AppTheme" android:label="@string/app\_name" android:icon="@mipmap/ic\_launcher" android:allowBackup="true">

<activity android:label="@string/app\_name" android:name="sg.vantagepoint.uncrackable1.MainActivity">

<intent-filter>

<action android:name="android.intent.action.MAIN"/>

<category android:name="android.intent.category.LAUNCHER"/>

</intent-filter>

</activity>

</application>

</manifest>

Da cui si ricava:

package="owasp.mstg.uncrackable1">android:name="sg.vantagepoint.uncrackable1.MainActivity">

Si nota che l’applicazion non richiede particolari permesi e richiede i seguenti requisit:

<uses-sdk android:minSdkVersion="19" android:targetSdkVersion="28"/>

Il metodo onCreate viene eseguito all’avvio e esegue in controlli per il root:

public void onCreate(Bundle bundle) {

if (c.a() || c.b() || c.c()) {

a("Root detected!");

}

if (b.a(getApplicationContext())) {

a("App is debuggable!");

}

super.onCreate(bundle);

setContentView(R.layout.activity\_main);

}

I metodi per cercare il root sono c.a(), c.b() e c.c() contenuti nella classe c

public class c {

public static boolean a() {

for (String str : System.getenv("PATH").split(":")) {

if (new File(str, "su").exists()) {

return true;

}

}

return false;

}

public static boolean b() {

String str = Build.TAGS;

return str != null && str.contains("test-keys");

}

public static boolean c() {

for (String str : new String[]{"/system/app/Superuser.apk", "/system/xbin/daemonsu", "/system/etc/init.d/99SuperSUDaemon", "/system/bin/.ext/.su", "/system/etc/.has\_su\_daemon", "/system/etc/.installed\_su\_daemon", "/dev/com.koushikdutta.superuser.daemon/"}) {

if (new File(str).exists()) {

return true;

}

}

return false;

}

}

Il metodo b.a richiamato nell’ onCreate esegue invece:

public class b {

public static boolean a(Context context) {

return (context.getApplicationContext().getApplicationInfo().flags & 2) != 0;

}

}

## Fase attacco:

### 1: Bypassare il root check per accedere al programma vero e proprio.

Se l’applicazione rivela un dispositivo rooted oppure un tentativo di debug si interrompe, questo check viene fatto nell’oncreate:

if (c.a() || c.b() || c.c()) {

a("Root detected!");

}

if (b.a(getApplicationContext())) {

a("App is debuggable!");

}

E il metodo a() della main class cioè:

a("Root detected!");

Si occupa di chiudere l’applicazione

private void a(String str) {

AlertDialog create = new AlertDialog.Builder(this).create();

create.setTitle(str);

create.setMessage("This is unacceptable. The app is now going to exit.");

create.setButton(-3, "OK", new DialogInterface.OnClickListener() {

/\* class sg.vantagepoint.uncrackable1.MainActivity.AnonymousClass1 \*/

public void onClick(DialogInterface dialogInterface, int i) {

**System.exit(0);**

}

});

create.setCancelable(false);

create.show();

}

Si può notare **System.exit(0);** una chiamata di sistema.

Per bypassare questo controllo ed accedere all’applicazone in se si hanno più opzioni:

1. Fare hooking su tutte l funzioni e bypassare il check.
2. Fare hooking su a() della mainactivity.
3. Impedire l’exit dell’applicazione facendo override su exit()

### Opzione 3:

Si scrive un hook javascript che verrà iniettato con Frida:

setImmediate(function(){

    console.log("Avvio dell'hook");

    Java.perform(function(){

        var system = Java.use("java.lang.System");

        system.exit.overload("int").implementation=function(var0){

            console.log("Uscita dall'applicazione");

        }

    });

});

dove

console.log("Uscita dall'applicazione");

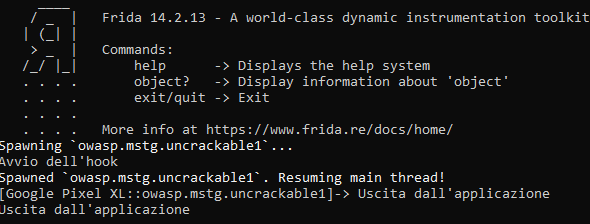
è il codice dell’override del metodo system.exit il cui codice originale era:

Runtime.getRuntime().exit(n)

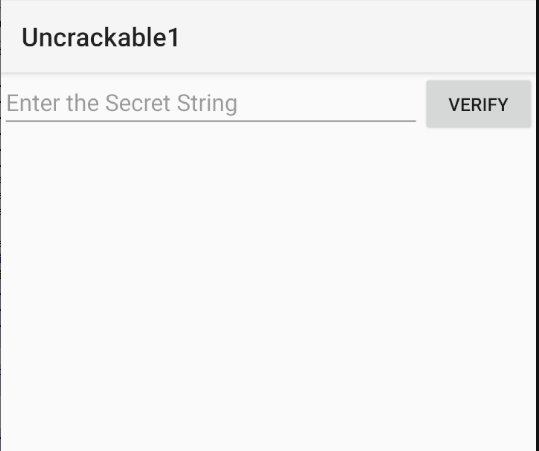
Si inietta quindi l’hook sull’applicazione:

frida -U -f owasp.mstg.uncrackable1 -l G:\GitHubRepo\AndroidReversing\hook\_1.js --no-pause

questo produce il seguente risultato:



Si può notare che nel momento in cui l’applicazione vorrebbe uscire esegue in realtà un console.log, disattivando di fatto il controllo impedendo l’uscita:



### Opzione 1:

Si scrive il seguente hook che fa tornare sempre true a tutti i metodi di check:

setImmediate(function(){

    console.log("Avvio dell'hook");

    Java.perform(function(){

        var c\_class = Java.use("sg.vantagepoint.a.c");

        c\_class.a.overload().implementation=function(){

            console.log("Override del metodo a");

            return false;

        }

    });

    Java.perform(function(){

        var c\_class = Java.use("sg.vantagepoint.a.c");

        c\_class.b.overload().implementation=function(){

            console.log("Override del metodo b");

            return false;

        }

    });

    Java.perform(function(){

        var c\_class = Java.use("sg.vantagepoint.a.c");

        c\_class.c.overload().implementation=function(){

            console.log("Override del metodo c");

            return false;

        }

    });

});

### Opzione 2:

In questo cado si ridefinisce il metodo onClick della schermata di errore, annullando la funzione che permetteva l’uscita dal programma, il problema è che il metodo onClick

public void onClick(DialogInterface dialogInterface, int i) {

System.exit(0);

}

E’ definito come classe anonima, la jadx-gui riesce a trovarla:

/\* class sg.vantagepoint.uncrackable1.MainActivity.AnonymousClass1 \*/

public void onClick(DialogInterface dialogInterface, int i) {

System.exit(0);

}

Si riesce a scrivere un hook che esegue il lavoro:

    console.log("Avvio dell'hook");

    Java.perform(function(){

        var main\_Activity = Java.use("sg.vantagepoint.uncrackable1.MainActivity$1");

        main\_Activity.onClick.implementation=function(){

            console.log("Override del metodo onClick della mainClass");

        }

    });

});

## Fase 2: Recovery della stringa segreta

Rimane quindi da scoprire la stringa segreta da inserire, analizzando il codice si nota banalmente che il check viene fatto nel metodo **verify:**

public void verify(View view) {

String str;

String obj = ((EditText) findViewById(R.id.edit\_text)).getText().toString();

AlertDialog create = new AlertDialog.Builder(this).create();

if (a.a(obj)) {

create.setTitle("Success!");

str = "This is the correct secret.";

} else {

create.setTitle("Nope...");

str = "That's not it. Try again.";

}

create.setMessage(str);

create.setButton(-3, "OK", new DialogInterface.OnClickListener() {

/\* class sg.vantagepoint.uncrackable1.MainActivity.AnonymousClass2 \*/

public void onClick(DialogInterface dialogInterface, int i) {

dialogInterface.dismiss();

}

});

create.show();

}

Il check viene fatto in

if (a.a(obj)) {

create.setTitle("Success!");

str = "This is the correct secret.";

} else {

create.setTitle("Nope...");

str = "That's not it. Try again.";

}}

Il cui metodo a.a(obj) esegue:

public static boolean a(String str) {

byte[] bArr;

byte[] bArr2 = new byte[0];

try {

bArr = sg.vantagepoint.a.a.a(b("8d127684cbc37c17616d806cf50473cc"), Base64.decode("5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=", 0));

} catch (Exception e) {

Log.d("CodeCheck", "AES error:" + e.getMessage());

bArr = bArr2;

}

return str.equals(new String(bArr));

}

public static byte[] b(String str) {

int length = str.length();

byte[] bArr = new byte[(length / 2)];

for (int i = 0; i < length; i += 2) {

bArr[i / 2] = (byte) ((Character.digit(str.charAt(i), 16) << 4) + Character.digit(str.charAt(i + 1), 16));

}

return bArr;

}

}

La challeange consiste nel risolvere questo problema crittografico:

bArr = sg.vantagepoint.a.a.a(b("8d127684cbc37c17616d806cf50473cc"), Base64.decode("5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=", 0));

in cui si esegue la chiamata sg.vantagepoint.a.a(a(b(“8d127684cbc37c17616d806cf50473cc”)) in cui i metodi a e b eseguono:

**Metodo a:**

public static byte[] a(byte[] bArr, byte[] bArr2) {

SecretKeySpec secretKeySpec = new SecretKeySpec(bArr, "AES/ECB/PKCS7Padding");

Cipher instance = Cipher.getInstance("AES");

instance.init(2, secretKeySpec);

return instance.doFinal(bArr2);

}**Metodo b:**

public class b {

public static boolean a(Context context) {

return (context.getApplicationContext().getApplicationInfo().flags & 2) != 0;

}

}

Si procede quindi impostando un hook su:

bArr = sg.vantagepoint.a.a.a(b("8d127684cbc37c17616d806cf50473cc"), Base64.decode("5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=", 0));

nello specifico sul metodo più esterno, sg.vantagepoint.a.a.a

Perchè il metodo eseguirà:

return str.equals(new String(bArr));

Si cerca di scrivere un hook per il metodo a in modo da intercettare:

return instance.doFinal(bArr2);

come prima cosa si prova ad inserire un hook trasparente:

setImmediate(function(){

    Java.perform(function(){

        var system = Java.use("java.lang.System");

        system.exit.overload("int").implementation = function(var0){

            console.log("Override uscita");

        }

        var algorithm = Java.use("sg.vantagepoint.a.a"); //Hook della funzione

        algorithm.a.overload("[B","[B").implementation = function(var0,var1){

            var ret = this.a.overload("[B","[B").call(this,var0,var1);

            console.log("Hook trasparente");

            return ret;

        }

    });

});

Quindi si inserisce un hook che stampa il contenuto di ret, cioè dell’output originale della funzione, si può verificare che nel momento in cui si preme il tasto verify si esegue l’hook.

Si procede a scrivere un hook che scrive il contenuto del return:

setImmediate(function(){

    Java.perform(function(){

        var system = Java.use("java.lang.System");

        system.exit.overload("int").implementation = function(var0){

            console.log("Override uscita");

        }

        var algorithm = Java.use("sg.vantagepoint.a.a"); //Hook della funzione

        algorithm.a.overload("[B","[B").implementation = function(var0,var1){

            var ret = this.a.overload("[B","[B").call(this,var0,var1);

            for(var i = 0;i<ret.length;i++){

                console.log(ret[i])

            }

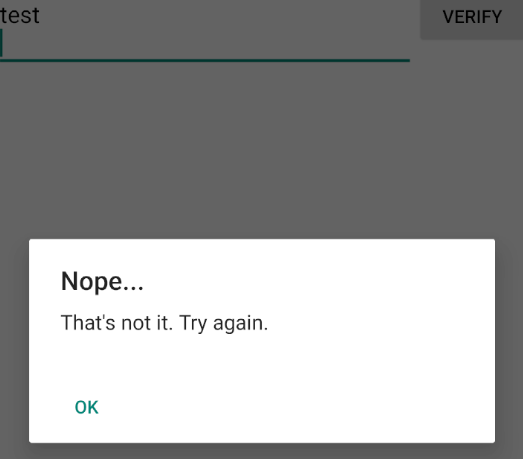
            return ret;

        }

    });

});

Da cui si ha il seguente output:

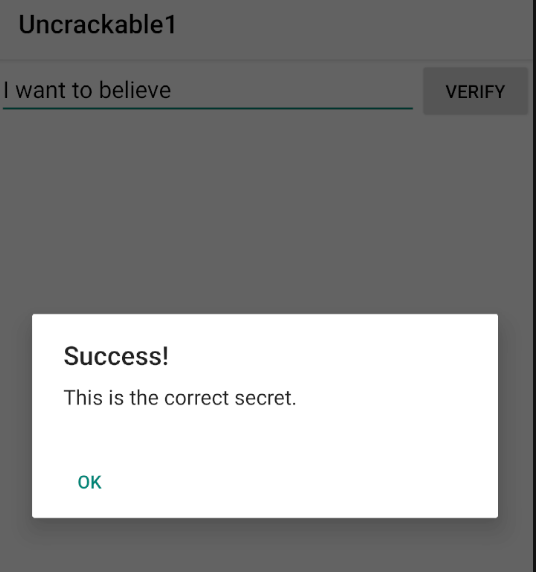


Tuttavia il programma stamperà sulla console attraverso frida la stringa (in byte):

73,32,119,97,110,116,32,116,111,32,98,101,108,105,101,118,101

Usando una tabella di encoding byte → string si risale al risultato:

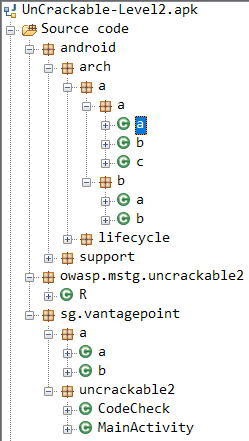
“I want to believe”, la stringa segreta.



Se si volesse permettere al programma di accettare **ogni** stringa e considerarla valida si potrebbe procedere con lo stesso principio.

# Livello 2

Si nota che il programma ha la seguente struttura:



Aprendo il codice della classe Main si può notare:

public class MainActivity extends c {

private CodeCheck m;

static {

System.loadLibrary("foo");

}

/\* access modifiers changed from: private \*/

/\* access modifiers changed from: public \*/

private void a(String str) {

AlertDialog create = new AlertDialog.Builder(this).create();

create.setTitle(str);

create.setMessage("This is unacceptable. The app is now going to exit.");

create.setButton(-3, "OK", new DialogInterface.OnClickListener() {

/\* class sg.vantagepoint.uncrackable2.MainActivity.AnonymousClass1 \*/

public void onClick(DialogInterface dialogInterface, int i) {

System.exit(0);

}

});

create.setCancelable(false);

create.show();

}

private native void init();

/\* access modifiers changed from: protected \*/

@Override // android.support.v4.app.h, android.support.v4.app.z, android.support.v7.app.c

public void onCreate(Bundle bundle) {

init();

if (b.a() || b.b() || b.c()) {

a("Root detected!");

}

if (a.a(getApplicationContext())) {

a("App is debuggable!");

}

new AsyncTask<Void, String, String>() {

/\* class sg.vantagepoint.uncrackable2.MainActivity.AnonymousClass2 \*/

/\* access modifiers changed from: protected \*/

/\* renamed from: a \*/

public String doInBackground(Void... voidArr) {

while (!Debug.isDebuggerConnected()) {

SystemClock.sleep(100);

}

return null;

}

/\* access modifiers changed from: protected \*/

/\* renamed from: a \*/

public void onPostExecute(String str) {

MainActivity.this.a((MainActivity) "Debugger detected!");

}

}.execute(null, null, null);

this.m = new CodeCheck();

super.onCreate(bundle);

setContentView(R.layout.activity\_main);

}

public void verify(View view) {

String str;

String obj = ((EditText) findViewById(R.id.edit\_text)).getText().toString();

AlertDialog create = new AlertDialog.Builder(this).create();

if (this.m.a(obj)) {

create.setTitle("Success!");

str = "This is the correct secret.";

} else {

create.setTitle("Nope...");

str = "That's not it. Try again.";

}

create.setMessage(str);

create.setButton(-3, "OK", new DialogInterface.OnClickListener() {

/\* class sg.vantagepoint.uncrackable2.MainActivity.AnonymousClass3 \*/

public void onClick(DialogInterface dialogInterface, int i) {

dialogInterface.dismiss();

}

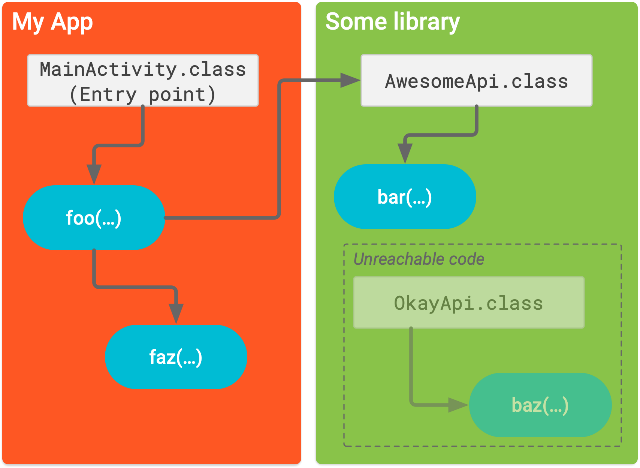
});

create.show();

}

}

Sembra più o meno il codice dell’altro, il programma esegue un root check come il precedente ma si può notare che il check della stringa inserita viene fatto utilizzando la JNI e non le istruzioni Java ad altro livello, questo è un modo noto e abbastanza comune utilizzato dai programmatori Android per nascondere codice e informazioni sensibili in un programma.



La parte che esegue il check è:

String obj = ((EditText) findViewById(R.id.edit\_text)).getText().toString();if (this.m.a(obj)) {

create.setTitle("Success!");

str = "This is the correct secret.";

} else {

create.setTitle("Nope...");

str = "That's not it. Try again.";

}

}

Dove la stringa obj viene passata al metodo m.a, l’oggetto m viene creato:

this.m = new CodeCheck();

dove codeChek() è:

private CodeCheck m;

Metodo dell’omonima classe che riporta il seguente codicE:

public class CodeCheck {

private native boolean bar(byte[] bArr);

public boolean a(String str) {

return bar(str.getBytes());

}

}

Quindi questa questo prende in ingresso una stringa e restituisce bar(string.getBytes()), questo bar è un metodo nativo:

private native boolean bar(byte[] bArr);

Probavbilmente questo metodo risiede nella libreria che viene importata dal main:

static {

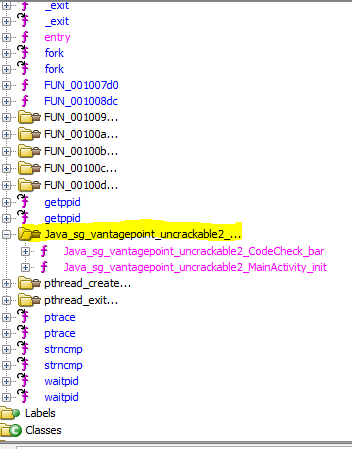
System.loadLibrary("foo");

}

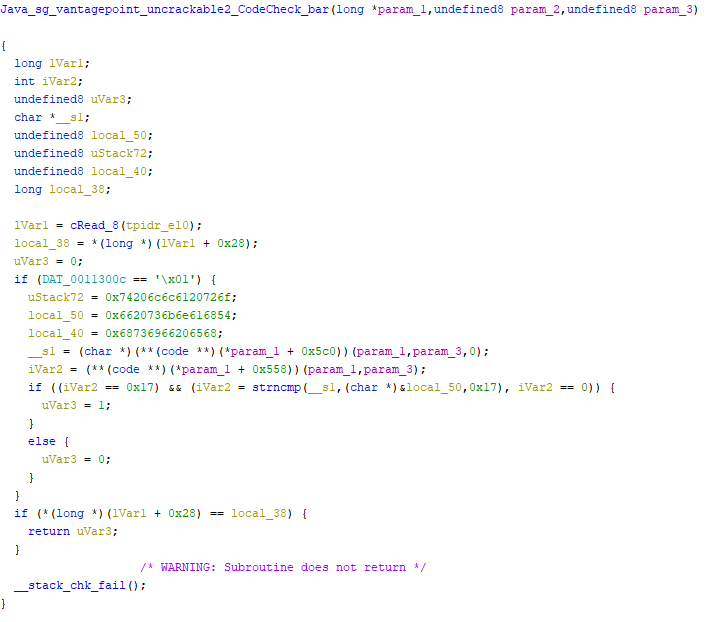
Dove “foo” è una libreria di sistema che può essere identificata come una delle “libfoo.so” presente nelle varie sottocartelle di /lib/ dell’apk.

L’analisi procede quindi analizzando la libreria libfoo.so utilizzando **Ghidra** (aggiungi intro su ghidra, nsa ecc.)

Analizzando la libreria si può notare che il tool riesce a trovare le istruzioni native dichiarate:



Si può notare la funzione uncrackable2\_codecheck\_bar e uncrackable\_2\_mainactivity\_init, andando ad ispezionare il codice della prima (codice C):



Il check del secret viene fatto in:

if ((iVar2 == 0x17) && (iVar2 = strncmp(\_\_s1,(char \*)&local\_50,0x17), iVar2 == 0)) {

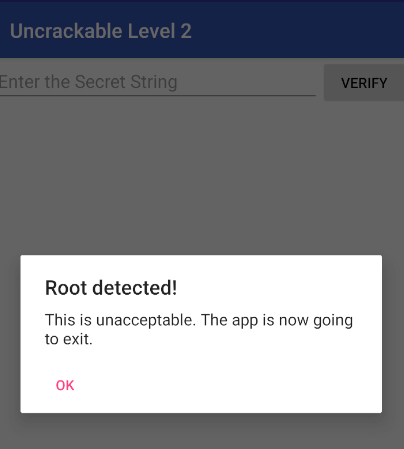
Dove la prima parte dell’if verifica se var2 è uguale a 0x17 mentre la seconda esegue il cmp della stringa con il secret, var2 è:

iVar2 = (\*\*(code \*\*)(\*param\_1 + 0x558))(param\_1,param\_3);

Che determina la lunghezza della stringa in ingresso, il programma eseguirà una strncnp s.s.e la lunghezza della stirnga inserita è 0x17, cioè 23.

## Analisi dinamica

Eseguendo il crackme si può notare che restituisce l’errore di rilevato dispositivo rooted:



Pertanto come prima cosa si proverà a bupassarlo usando frida

setImmediate(function(){

console.log("Avvio dell'hook");

Java.perform(function(){

var system = Java.use("sg.vantagepoint.uncrackable2.MainActivity");

system.a.overload("java.lang.String").implementation=function(var0){

console.log("Uscita dall'applicazione");

}

});

});

Si procede quindi a recuperare la stringa di confronto che verrà utilizzata dentro la libfoo.so a partire dalla chiamata del metodo bar, si scrive quindi un hook per “attaccarsi” alla funzione strncmp per analizzare gli argomenti passati, in cui si presume poter esserci la stringa segreta:

Java.perform(function () {

const sysexit = Java.use("java.lang.System");

sysexit.exit.overload("int").implementation = function(a0) {

console.log("Override dell'uscita");

};

});

Interceptor.attach(Module.findExportByName("libc.so", "strncmp"), {

onEnter: function (args) {

var bt = backtrace(this);

if (bt.includes("libfoo.so")) {

console.log("----------------------------------");

console.log("Estrazione degli argomenti")

console.log(Memory.readUtf8String(args[0]));

console.log(Memory.readUtf8String(args[1]));

console.log(Memory.readUtf8String(args[2]));

console.log("----------------------------------");

}

}

});

function backtrace(c) {

return (Thread.backtrace(c.context, Backtracer.ACCURATE)

.map(DebugSymbol.fromAddress).join("\n") + "\n");

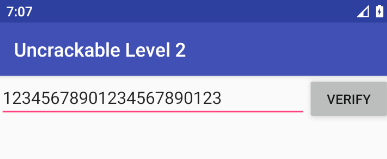
}

Attaccando l’hook al programma:

frida -U -f owasp.mstg.uncrackable2 -l G:\GitHubRepo\AndroidReversing\Level2\hook.js --no-pause

Si va ad inserire una stringa di 23 caratteri:

12345678901234567890123



Nella console di Frida si può osservare l’output

