

Hooking Native Methods

Native methods—written in C or C++ and compiled into machine code—are often used in Android applications for performance-critical tasks, accessing low-level system resources, or handling sensitive operations. While powerful, these methods can also introduce vulnerabilities or become targets for exploitation.

By this point, you're already familiar with the concept of hooking. In this section, we'll apply hooking specifically to native methods to observe their behavior at runtime. In parallel, we'll use static analysis to disassemble and review native libraries included in the app, allowing us to understand their logic and identify potential vulnerabilities from both dynamic and static perspectives.

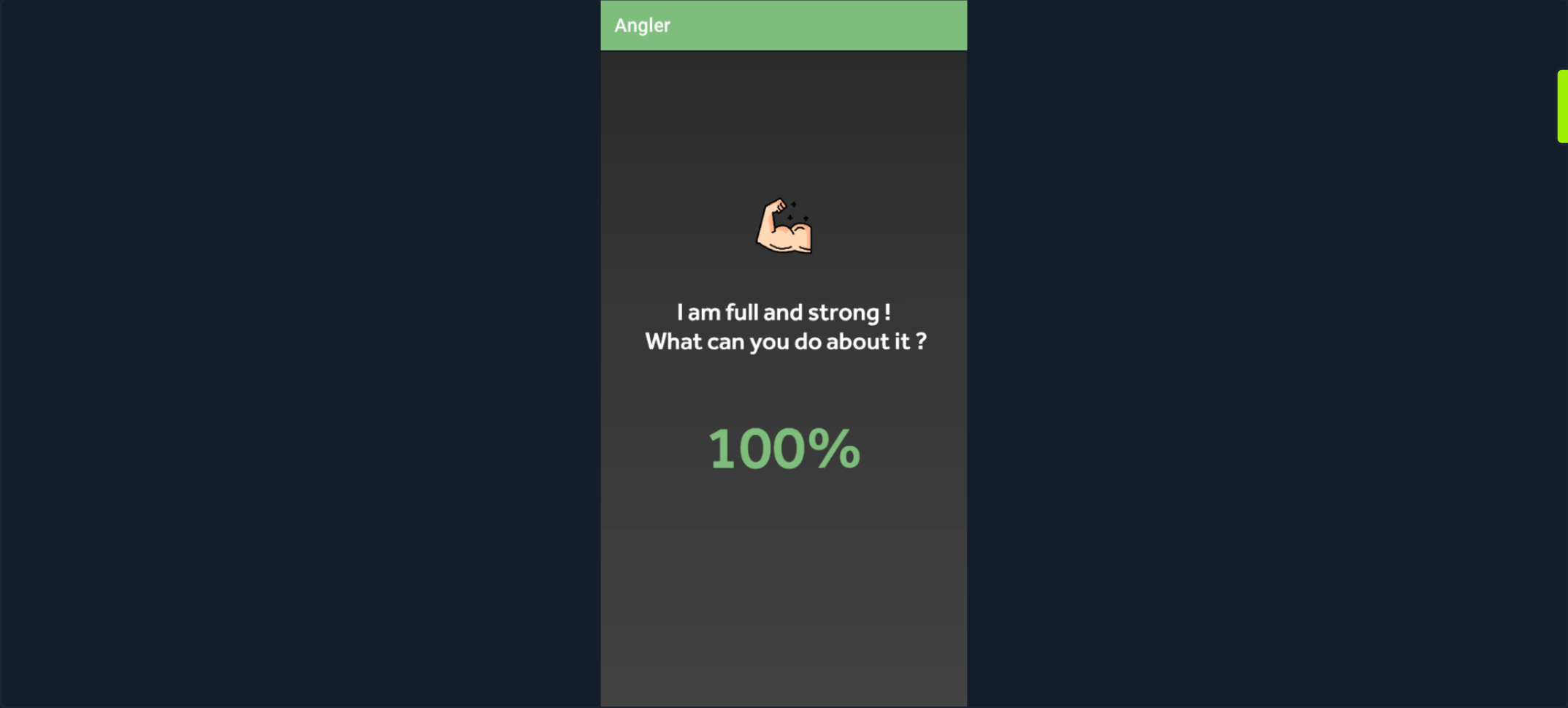
We'll primarily use an Android Virtual Device (AVD) for this example, though any physical or emulated Android device will work. Let's begin by connecting to the device via ADB and installing the application.

Hooking Native Methods

```
r11k@htb[/htb]$ adb connect
r11k@htb[/htb]$ adb install myapp.apk

Performing Streamed Install
Success
```

The application presents the user a riddle: **I am full and strong! What can you do about it?.**



Let's examine the application using JADX to see how it works. Reading the source code reveals the **MainActivity** class.

Hooking Native Methods

```
r11k@htb[/htb]$ jadx-gui angler.apk
```

com

example.angler

MainActivity

R

google.android.material

d

d0

d1

d2

e

72

73

74

75

76

77

78

79

80

81

82

83

84

```
public native String getInfo(String str);

@Override // androidx.fragment.app.p, androidx.activity.ComponentActivity, w.g, android.app.Activity
public final void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    setContentView(R.layout.activity_main);
    this.v = (TextView) findViewById(R.id.textView2);
    this.f1753w = (TextView) findViewById(R.id.textView);
    this.f1754x = (ImageView) findViewById(R.id.imageView);
    registerReceiver(this.f1756z, new IntentFilter("android.intent.action.BATTERY_LOW"));
}
```

We notice that in the last line of the `onCreate()` method, a broadcast receiver, is registered dynamically:

Code: java

```
registerReceiver(this.f1756z, new IntentFilter("android.intent.action.BATTERY_LOW"));
```

Reading further into the class, we see that the inner class `a` extends `BroadcastReceiver`.

com

example.angler

MainActivity

R

google.android.material

d

d0

d1

d2

e

e0

e1

e2

f

f0

f1

f2

g

g0

g1

g2

h

h0

h1

h2

i

i0

i1

i2

j

j0

j1

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

/* loaded from: classes.dex */

public class a extends BroadcastReceiver {

public a() {

}

@Override // android.content.BroadcastReceiver

public final void onReceive(Context context, Intent intent) {

PrintStream printStream;

String str;

if (intent.getStringExtra("Is_on").equals("yes")) {

MainActivity mainActivity = MainActivity.this;

int i3 = MainActivity.A;

Window window = mainActivity.getWindow();

window.addFlags(Integer.MIN_VALUE);

window.clearFlags(67108864);

window.setStatusBarColor(mainActivity.getResources().getColor(R.color.purple_200));

d.a r3 = mainActivity.r();

Objects.requireNonNull(r3);

r3.b(new ColorDrawable(mainActivity.getResources().getColor(R.color.teal_700)));

mainActivity.f1754x.setImageResource(R.drawable.please);

mainActivity.v.setTextColor(mainActivity.getResources().getColor(R.color.purple_200));

mainActivity.v.setText("1%");

mainActivity.f1753w.setText(d.d(mainActivity.f1755y));

Toast.makeText(context, "Look me inside", 1).show();

printStream = System.out;

str = MainActivity.this.getInfo(d.d("XDR"));

} else {

printStream = System.out;

str = "I am Strong, no one can defeat me";

}

printStream.println(str);

}

}

static {

System.loadLibrary("angler");

}

public native String getInfo(String str);

As discussed in previous sections, a broadcast receiver can function both as an application component and an interprocess communication (IPC) mechanism. In this case, it acts as a component that listens for a specific system broadcast: `android.intent.action.BATTERY_LOW`. When this intent is received, it influences the app's behavior depending on the device's battery status.

The receiver invokes the native method `getInfo(String str)` in the following line:

Code: java

```
str = MainActivity.this.getInfo(d.d("XDR"));
```

But only if the following condition is satisfied.

Code: java

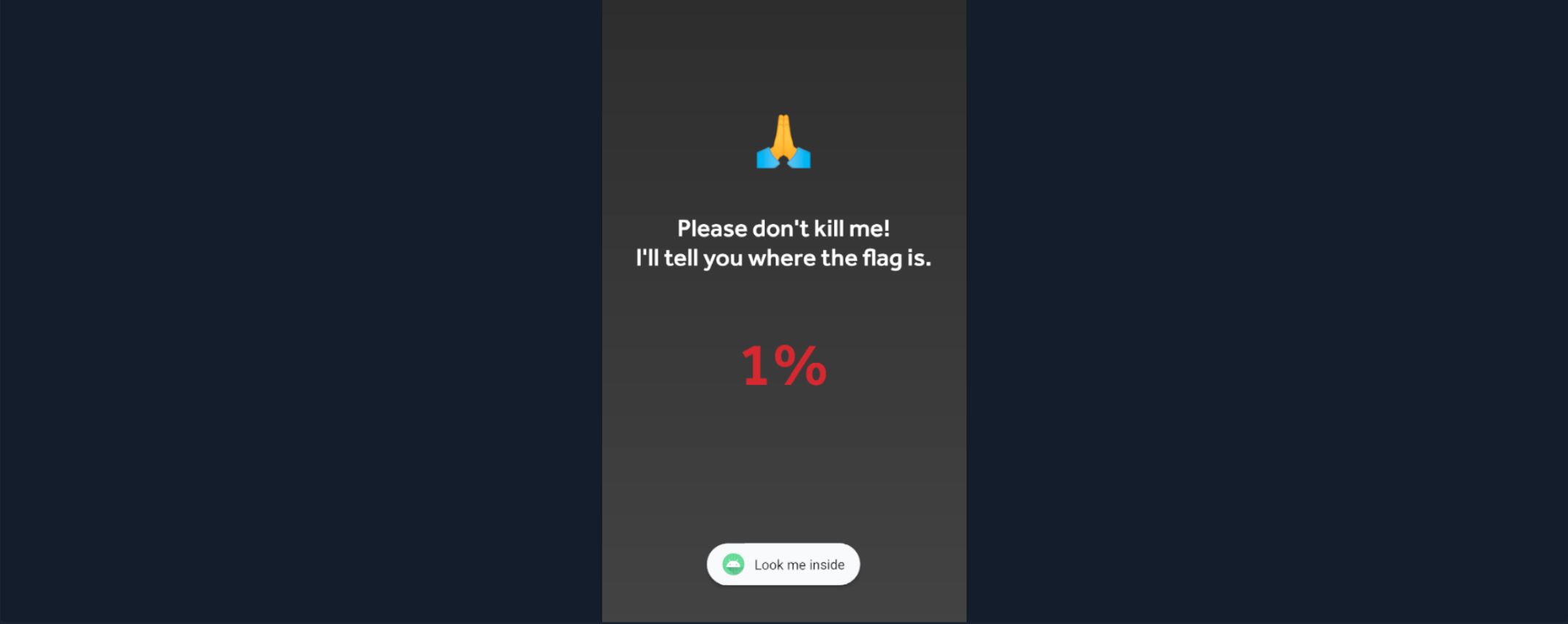
```
if (intent.getStringExtra("Is_on").equals("yes")) {  
    ...  
}
```

This checks whether the Intent contains an Extra named `"Is_on"` with the value `"yes"`.

Since the broadcast can originate from either the system or a third-party app, we can manually trigger it using ADB. Doing so will cause the app to start and call the native method.

Hooking Native Methods

```
r11k@htb[/htb]$ adb shell am broadcast -a "android.intent.action.BATTERY_LOW" --es "Is_on" "yes"  
  
Broadcasting: Intent { act=android.intent.action.BATTERY_LOW flg=0x400000 (has extras) }  
Broadcast completed: result=0
```



However, this screen doesn't reveal much—just a toast message saying "Look me inside." To further investigate, we need to analyze the app's native code. First, decompress the APK to extract the native libraries:

Hooking Native Methods

```
r11k@htb[/htb]$ unzip Angler.apk -d angler

<SNIP>
Archive:  Angler.apk
  inflating: angler/META-INF/com/android/build/gradle/app-metadata.properties
  extracting: angler/assets/dexopt/baseline.prof
  extracting: angler/assets/dexopt/baseline.profm
  inflating: angler/classes.dex
  inflating: angler/lib/arm64-v8a/libangler.so
  inflating: angler/lib/armeabi-v7a/libangler.so
  inflating: angler/lib/x86/libangler.so
  inflating: angler/lib/x86_64/libangler.so
  inflating: angler/AndroidManifest.xml
```

The output confirms the presence of the library file: `angler/lib/arm64-v8a/libangler.so`. Let's examine it with Ghidra.

Hooking Native Methods

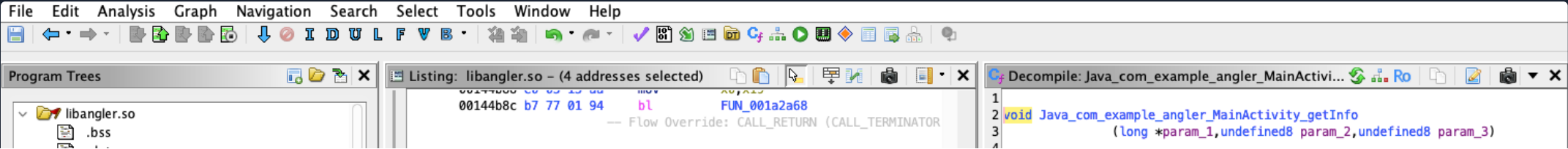
```
r11k@htb[/htb]$ ghidrarun
```

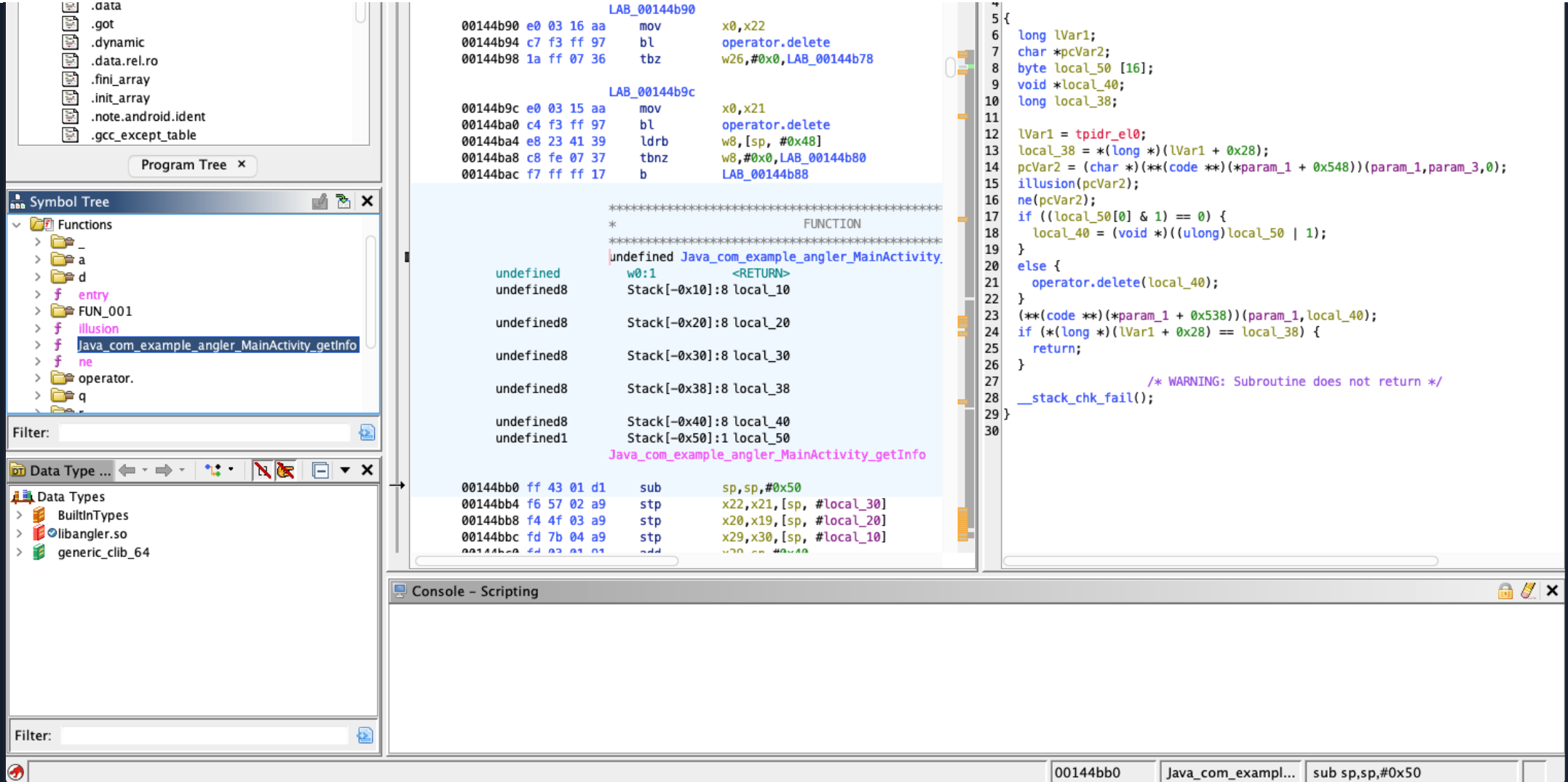
Once Ghidra starts:

- 1. Go to `File` → `New Project`, choose a name (e.g., `Test`), and click `Finish`.
- 2. Then go to `File` → `Import File` and select `libangler.so` from `angler/lib/arm64-v8a/`.
- 3. Accept the default settings in the pop-ups and click `OK`.
- 4. Open the imported file by double-clicking it or clicking the green dragon icon.
- 5. Confirm any prompts and click `Analyze`.

After the analysis completes, navigate to the `Symbol Tree`, expand the `Functions` folder, and locate `Java_com_example_angler_MainActivity_getInfo`.

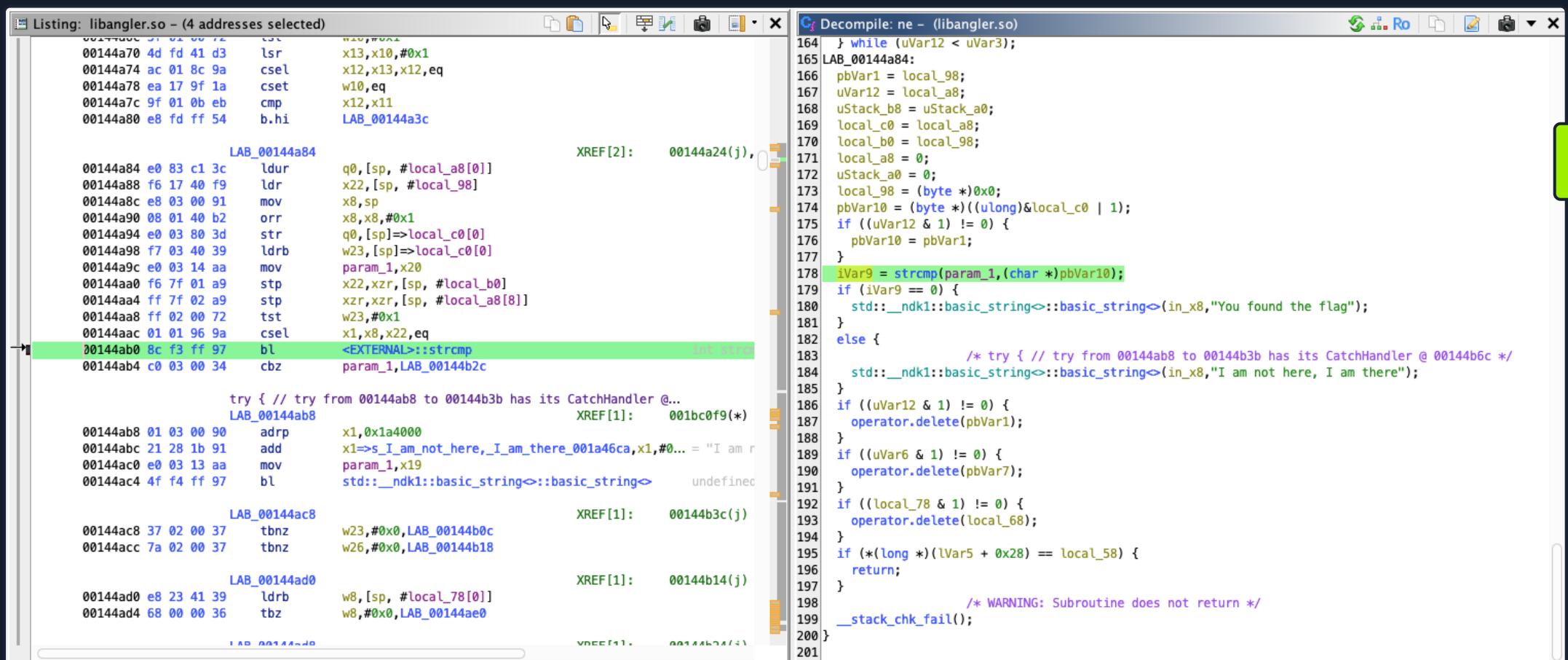
Based on the JNI naming convention, this function corresponds to the native `getInfo()` method we previously identified in the app's Java code using JADX.





Reading the contents of the function `Java_com_example_angler_MainActivity_getInfo` in Ghidra's Decompile window reveals that it calls two other functions: `illusion(pcVar1)` and `ne(pcVar2)`.

Double-clicking `ne(pcVar2)` opens the corresponding code, where we find a key comparison using the `strcmp` function.



The following line compares two strings: `param_1` and `pbVar10`.

Code: `java`

```
iVar9 = strcmp(param_1, (char *)pbVar10);
```

The `strcmp` function returns `0` if the two strings are equal. If this condition is met (`iVar9 == 0`), the code proceeds to construct a message string containing `You found the flag` using `std::__ndk1::basic_string<>`.

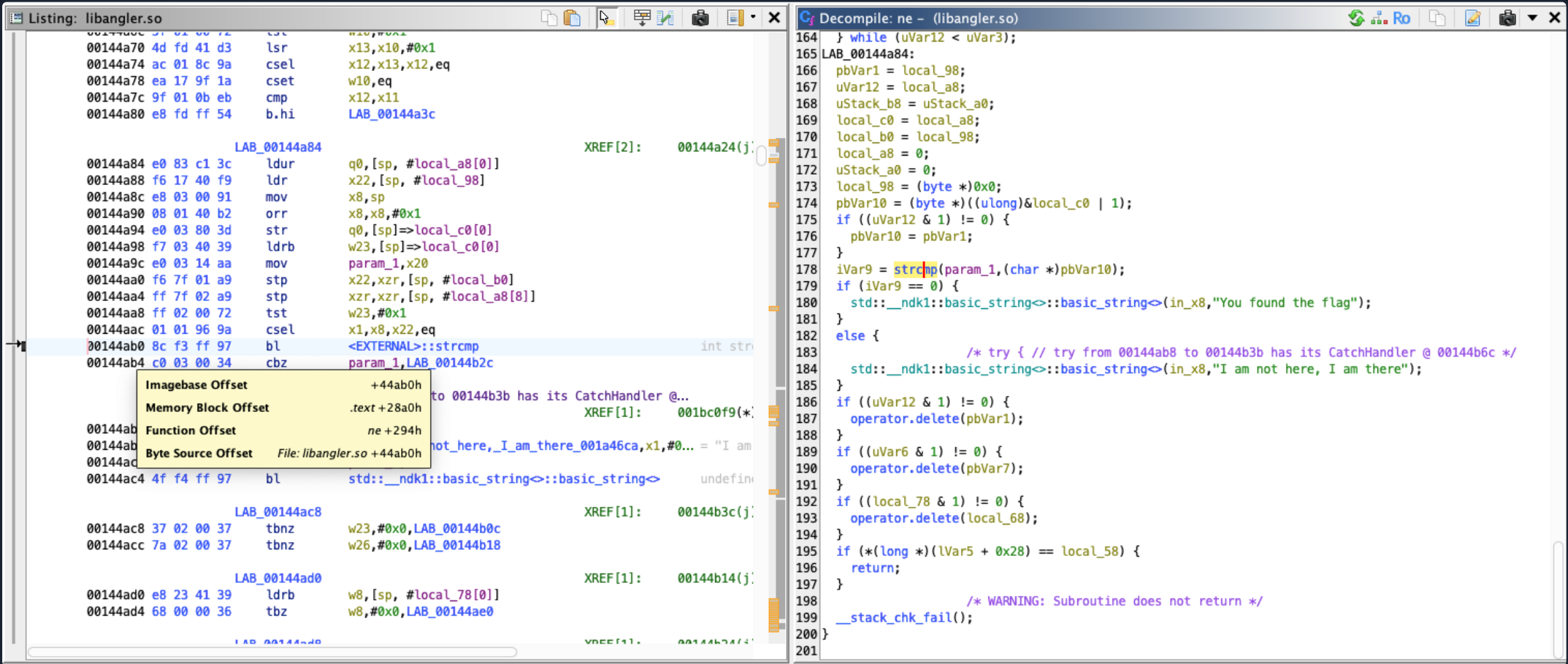
Referring back to the Java code in `MainActivity`, we saw that the `param_1` value passed to `strcmp` comes from the return value of the method `d.d("XDR")`, as used in the line:

Code: `java`

```
str = MainActivity.this.getInfo(d.d("XDR"));
```


To fully understand what the native function is comparing `param_1` against, we need to determine the value of `pbVar10`, the second argument passed to `strcmp`. This requires the identification of the memory address where `strcmp` is invoked during execution.

Due to runtime protection mechanisms like ASLR (Address Space Layout Randomization), we cannot determine the absolute address of `strcmp` statically in Ghidra. However, we can calculate it at runtime by adding the offset of the `strcmp` call—which we can get by hovering over the instruction in the `Listing` window—to the base address of the native library, which we'll retrieve dynamically when the app is running.



Now that we know the offset (`44ab0h`), we can start crafting our JS script. In a file called `snippet.js`, we add the following code.

Code: js

```
// Delay execution to ensure module is loaded
setTimeout(function() {
  // Dynamically find module base address
  var libanglerBase = Module.findBaseAddress('libangler.so');
  if (!libanglerBase) {
    console.log('libangler.so module not found!');
    return;
  }

  // Calculate specific strcmp call address using offset from Ghidra
  var strcmpCallOffset = ptr('0x44ab0');
  var strcmpCallAddress = libanglerBase.add(strcmpCallOffset);

  // Attach interceptor to the specific strcmp call
  Interceptor.attach(strcmpCallAddress, {
    // Hook entering the function
    onEnter: function(args) {
      console.log('');
      console.log('Parameter 1:', Memory.readCString(args[0]));
      console.log('Parameter 1:', Memory.readCString(args[1]));
    },
    // Hook leaving the function
    onLeave: function(retval) {
      console.log('Specific strcmp call returned:', retval.toInt32());
    }
  });
}, 1000); // 1 second delay to wait for module load
```

Notice that we changed the offset from `44ab0h` to `0x44ab0`. This is due to JavaScript using the prefix `0x` to denote hexadecimal numbers. Additionally, the `h` suffix is a notation used in some assembly or legacy languages representing hexadecimal format.

Let's break down the script in more detail to understand what each part does.

Instruction	Description
<code>setTimeout</code>	This delays the execution of the script by 1000 milliseconds (1 second). The delay ensures that the native module (<code>libangler.so</code>) has been fully loaded into memory before the script attempts to locate it. If we try to attach the interceptor too early, the module may not exist in memory yet, causing the script to fail.
<code>Module.findBaseAddress('libangler.so')</code>	Searches for the base memory address of the <code>libangler.so</code> library loaded into the target process. This base address is essential for calculating the absolute location of the target function call (<code>strcmp</code>) by adding the offset obtained from Ghidra.
<code>ptr('0x44ab0')</code>	Converts the hexadecimal string <code>'0x44ab0'</code> into a <code>NativePointer</code> object. This value represents the offset within the library where the <code>strcmp</code> call is located.
<code>libanglerBase.add(strcmpCallOffset)</code>	Adds the offset to the base address of the library, producing the absolute address of the specific <code>strcmp</code> call that we want to hook.
<code>Interceptor.attach(...)</code>	Hooks into the calculated memory address. When this address is reached during the app's execution (i.e., when <code>strcmp</code> is called), the <code>onEnter</code> and <code>onLeave</code> callbacks defined in this section will be triggered.
<code>Memory.readCString(args[0])</code> <code>Memory.readCString(args[1])</code>	Reads the two string arguments passed to <code>strcmp</code> from memory. These are C-style strings (null-terminated), and Frida provides this helper to read them correctly.
<code>console.log(...)</code>	Outputs the arguments and the return value to the terminal. This allows us to see exactly what strings are being compared and what the result of the comparison is.

Now that we understand what the script is doing, and we already know the package name (`com.example.angler`), let's go ahead and execute it using Frida:

Hooking Native Methods

```
r11k@htb[/htb]$ frida -U -l snippet.js -f com.example.angler

----
/ _ |   Frida 16.1.11 - A world-class dynamic instrumentation toolkit
| (| |
> _ |   Commands:
/_/ |_|   help      -> Displays the help system
. . . .   object?    -> Display information about 'object'
. . . .   exit/quit  -> Exit
. . . .
. . . .   More info at https://frida.re/docs/home/
. . . .
. . . .   Connected to Android Emulator (id=emulator-5554)
Spawned `com.example.angler`. Resuming main thread!
[Android Emulator 5554::com.example.angler ]->
```

Frida will attach and wait for the `strcmp` function to be called, so we send the broadcast again.

Hooking Native Methods

```
r11k@htb[/htb]$ adb shell am broadcast -a "android.intent.action.BATTERY_LOW" --es "Is_on" "yes"

Broadcasting: Intent { act=android.intent.action.BATTERY_LOW flg=0x400000 (has extras) }
Broadcast completed: result=0
```

Once the broadcast is received, the application will automatically start and execute the native `getInfo()` method, which eventually calls `strcmp`. When this happens, the hooked function is triggered, and Frida prints the intercepted values to the terminal.

Hooking Native Methods

```
r11k@htb[/htb]$

<SNIP>
. . . .   Connected to Android Emulator (id=emulator-5554)
Spawned `com.example.angler`. Resuming main thread!
[Android Emulator 5554::com.example.angler ]->
```

```
[Android Emulator-5554:com.example.angler] >
Parameter 1: HTB
Parameter 1: 4854427b796f755f3472335f676f6f645f34745f6830306b316e397d
```

The hook is successful. Both parameters are printed, with the first being **HTB** and the second **4854427b796f755f3472335f676f6f645f34745f6830306b316e397d**. Based on its format and character set, we can identify with certainty that the second value is a hex-encoded string. To decode it and reveal its actual content, we'll update the script to include a conversion function that transforms hexadecimal into ASCII.

Here is the updated **snippet.js**.

```
Code: js

// Delay execution to ensure module is loaded
setTimeout(function() {
  // Dynamically find module base address
  var libanglerBase = Module.findBaseAddress('libangler.so');
  if (!libanglerBase) {
    console.log('libangler.so module not found!');
    return;
  }

  // Calculate specific strcmp call address using offset from Ghidra
  var strcmpCallOffset = ptr('0x44ab0');
  var strcmpCallAddress = libanglerBase.add(strcmpCallOffset);

  // Attach interceptor to the specific strcmp call
  Interceptor.attach(strcmpCallAddress, {
    // Hook entering the function
    onEnter: function(args) {
      var param1 = Memory.readCString(args[0]);
      var param2 = Memory.readCString(args[1]);
      var flag = hexToASCII(param2);
      console.log('');
      console.log('Parameter 1:', param1);
      console.log('Parameter 2:', flag);
    },
    // Hook leaving the function
    onLeave: function(retval) {
      console.log('Specific strcmp call returned:', retval.toInt32());
    }
  });
}, 1000); // 1 second delay to wait for module load

// Define a function to convert hexadecimal to ASCII string
function hexToASCII(hex) {
  var str = ''; // Initialize an empty string for the result
  for (var i = 0; i < hex.length; i += 2) { // Iterate over hex string two characters at a time
    var v = parseInt(hex.substr(i, 2), 16); // Convert each hex pair to decimal
    if (v) str += String.fromCharCode(v); // Convert decimal to character and append to result string
  }
  return str; // Return the resulting ASCII string
}
```

In this iteration, the function **hexToASCII(hex)** has been added, which converts the hexadecimal string to ASCII. Save the changes and run Frida once more.

```
Hooking Native Methods

r11k@htb[/htb]$ frida -U -l snippet.js -f com.example.angler

x INT 22s ≡

<SNIP>
. . . . Connected to Android Emulator (id=emulator-5554)
```

```
Spawned `com.example.angler`. Resuming main thread!  
[Android Emulator 5554::com.example.angler ]->
```

Again, we send the broadcast.


Hooking Native Methods

```
r11k@htb[/htb]$ adb shell am broadcast -a "android.intent.action.BATTERY_LOW" --es "Is_on" "yes"  
  
Broadcasting: Intent { act=android.intent.action.BATTERY_LOW flg=0x400000 (has extras) }  
Broadcast completed: result=0
```

At last, the output reveals the ASCII representation of the second parameter.

Hooking Native Methods

```
<SNIP>  
. . . . . Connected to Android Emulator (id=emulator-5554)  
Spawned `com.example.angler`. Resuming main thread!  
[Android Emulator 5554::com.example.angler ]->  
Parameter 1: HTB  
Parameter 2: HTB{h0ok1ng_n4t1v3_funct10ns!}
```



Connect to Pwnbox

Your own web-based Parrot Linux instance to play our labs.

Pwnbox Location

UK

28ms

Terminate Pwnbox to switch location

Start Instance

/ 1 spawns left

Questions

Cheat Sheet

Answer the question(s) below to complete this Section and earn cubes!

+ 5

What is the value of the second parameter passed to the strcmp function?

Submit your answer here...

+10 Streak pts

Submit

hook_native_method.zip

← Previous

Next →

Cheat Sheet

Go to Questions

Table of Contents

Enumerating and Exploiting Installed Apps

- Introduction
- Enumerating Local Storage
- Exported Activities
- Insecure Logging
- Pending Intents
- Exploiting WebViews
- Insecure Library Load Through Deep Linking

Dynamic Code Instrumentation

- Hooking Java Methods
- Altering Method Values
- Hooking Native Methods
- Bypassing Detection Mechanisms
- Authentication Token Manipulation

Intercepting HTTP/HTTPS Requests

- Intercepting API Calls
- IDOR Attack
- SSL/TLS Certificate Pinning Bypass

Skills Assessments

- Skills Assessment

My Workstation

OFFLINE

▶ Start Instance

∞ / 1 spawns left

