

# Hooking Java Methods

In Android application penetration testing, understanding how apps behave at runtime is essential. This is where dynamic analysis becomes particularly valuable, offering insights that static analysis may overlook. One of the most powerful techniques in dynamic analysis (and the focus of this section) is **Dynamic Code Instrumentation**.

Unlike static analysis, dynamic instrumentation enables us to observe, trace, and modify an application's behavior during execution without altering its original source code. This technique is invaluable for bringing hidden issues and vulnerabilities to the surface, understanding complex behavior, and ultimately assessing the security of an application under real-world conditions.

## Applications of Code Instrumentation

Application	Description
Performance Analysis	Instrumentation allows developers to measure the execution time of different parts of the code, identify bottlenecks, and optimize performance.
Debugging	By inserting logging statements or breakpoints, developers can trace the execution flow and isolate bugs.
Behavioral Analysis	Observing how an application reacts to various inputs provides insights into functionality and areas for improvement.
Security Analysis	In penetration testing, instrumentation reveals how an app interacts with user data and uncovers vulnerabilities that attackers can exploit.

In the following section, we will demonstrate a technique for capturing the return value of a Java method using a tool called Frida—a dynamic instrumentation toolkit widely used by developers, reverse engineers, and security researchers. It enables the injection of JavaScript or native code directly into the memory of running Android processes.

**Method hooking**, a core technique in dynamic analysis, allows testers to intercept method calls to observe inputs and outputs, alter return values, or change parameters. In this section, we will focus specifically on intercepting and logging a method's return value.

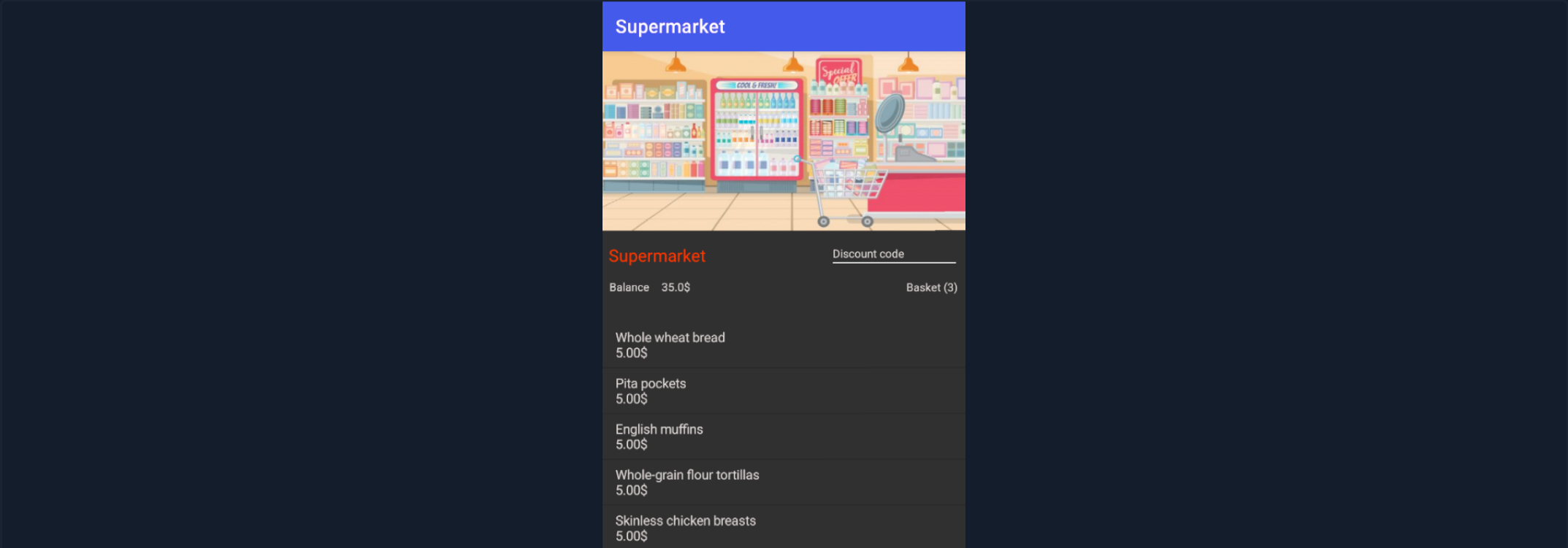
This example will primarily use an Android Virtual Device (AVD), although the process is compatible with any physical or emulated Android device. Let's begin by connecting to the device via ADB and installing the application.

Hooking Java Methods

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

Performing Streamed Install
Success
```

Running the application, we see that it is an online shopping app that allows users to browse and purchase products.





In the top-right corner of the screen, there is a **Discount code** field where users can enter a promotional code to receive a discount. Let's open the application using JADX to examine its source code.

Hooking Java Methods

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



The source code reveals that class and method names have been obfuscated. However, upon analyzing the **onTextChanged()** method within the app's MainActivity, we can deduce that it listens for input changes in a particular field and stores the input string in the variable **obj**.

Code: java

```
String obj = MainActivity.this.f2075q.getText().toString();
```

The code then performs a decryption operation using keys and algorithms sourced from native code. It compares the user's input to the result of this decryption:

Code: java

```
if (!obj.equals(new String(cipher.doFinal(Base64.decode(stringFromJNI, 0)), "utf-8")))
```

Given that the discount code field is the only visible input on the app's screen, this code likely handles that feature. To investigate further, we'll use Frida to hook into the method that processes the comparison—specifically the return value of **cipher.doFinal()**, which likely contains the correct discount code. Before proceeding with our script, let's briefly review how Frida interacts with Java code.

## 1. Injecting Code into Running Processes

Frida allows for the injection of arbitrary code into active processes. On Android and other Linux-based systems, this is typically done using **ptrace**, though equivalent mechanisms exist on other platforms. Once injected, Frida operates within the target application's runtime, enabling custom scripts to execute alongside the app's native code.

## 2. Bridging Different Runtime Environments

Frida acts as a bridge between its embedded JavaScript engine ([Duktape](#) or [V8](#)) and the target application's [Java Virtual Machine \(JVM\)](#). This bridge allows JavaScript code to perform meaningful actions within the app's Java environment, such as invoking methods, accessing fields, and modifying class behavior at runtime.

### 3. Manipulating Java Objects and Classes

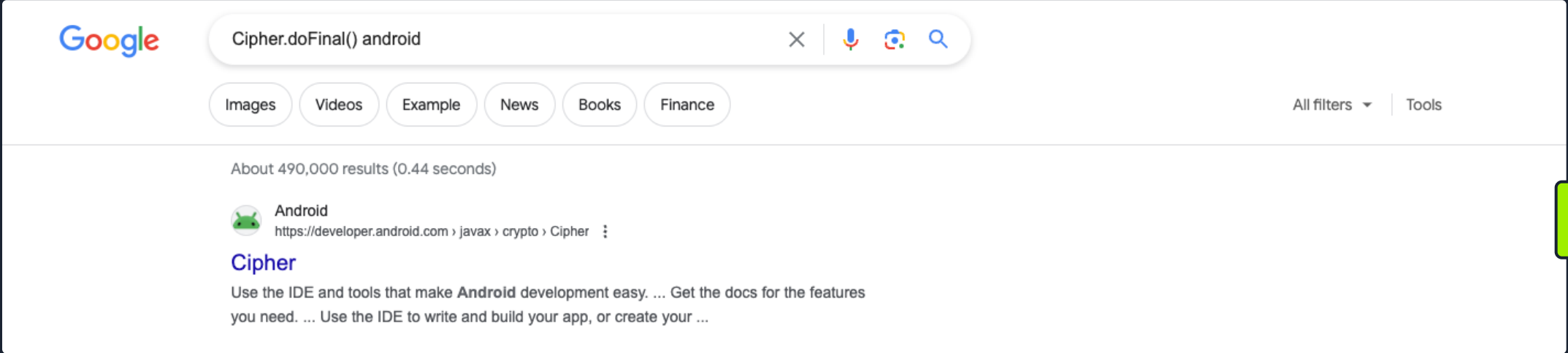
Frida uses a technique known as method hooking, which intercepts calls to specific methods of Java classes. When a method is hooked, Frida redirects the call to a handler function defined in JavaScript. There, we can inspect or modify the arguments, execute the original method, or alter the return value. This capability is crucial for dynamic analysis, enabling real-time inspection and modification of the application's behavior.

### 4. Runtime Type Information

**Runtime Type Information** is a technology that allows applications (and, by extension, tools like Frida) to dynamically query and interact with objects, regardless of their compile-time types. This is crucial for Frida's operation, as it needs to inspect, modify, and interact with various objects and classes at runtime. Frida leverages [Runtime Rype Information \(RTTI\)](#), available within the JVM, to interact with Java objects and perform the previously mentioned operations.

## Building the Script

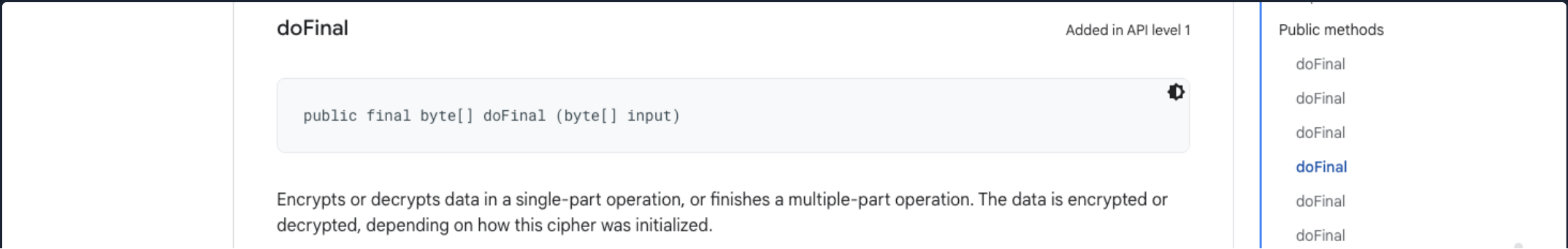
Now that we have an understanding of how Frida uses JavaScript to hook into method calls, the next step is to gather the information needed to hook the `Cipher.doFinal()` method. Specifically, we need to identify the method;s argument types and the full package name of the `Cipher` class. A quick search for `Cipher.doFinal()` `android` provides the following as the first result.



This is Android's official documentation for the Class `Cipher`. Right at the top of the screen, we notice the class's package name is `javax.crypto.Cipher`.



As we continue to read through the documentation, we come to the section [Public Methods](#). It includes the method signatures available within the `Cipher` class, showcasing the diversity of overloads for handling different operations. Looking at the table on the right-side of the page, we can see seven different methods with the name `doFinal()`.



The bytes in the `input` buffer, and any input bytes that may have been buffered during a previous `update` operation, are processed, with padding (if requested) being applied. If an AEAD mode such as GCM/CCM is being used, the authentication tag is appended in the case of encryption, or verified in the case of decryption. The result is stored in a new buffer.

Upon finishing, this method resets this cipher object to the state it was in when previously initialized via a call to `init`. That is, the object is reset and available to encrypt or decrypt (depending on the operation mode that was specified in the call to `init`) more data.

Note: if any exception is thrown, this cipher object may need to be reset before it can be used again.

Parameters	
<code>input</code>	<code>byte</code> : the input buffer

Returns	
<code>byte[]</code>	the new buffer with the result

doFinal

getAlgorithm

getBlockSize

getExemptionMechanism

getIV

getInstance

getInstance

getInstance

getMaxAllowedKeyLength

getMaxAllowedParameterSpec

getOutputSize

getParameters

getProvider

init

init

init

init

We already know that the specific `doFinal()` method used in the app takes only one argument, which suggests it is the `public final byte[] doFinal(byte[] input)` variant. To write an effective Frida script, it's essential to identify both the method's signature (including the number and type of parameters) and the class that implements it. In this case, the method accepts a single parameter of type `byte[]`.

With both the method signature and the class's package name confirmed, we can now write our hook. Create a file named `snippet.js` and add the following JavaScript code:

Code: `js`

```
// Initiates a Frida script to interact with Java classes and methods.
Java.perform(function () {
  // Accesses the Cipher class from the Java Cryptography API.
  var Cipher = Java.use('javax.crypto.Cipher');

  // Hooks into the doFinal method of the Cipher class that processes a byte array.
  Cipher.doFinal.overload('[B').implementation = function (input) {
    // Executes the original doFinal method with the given input and stores the result.
    var result = this.doFinal(input);

    // Creates a new String object from the result byte array assuming it's UTF-8 encoded.
    var decryptedString = Java.use("java.lang.String").$new(result, "UTF-8");

    // Logs the decrypted string to the console.
    console.log("Decrypted string: " + decryptedString);

    // Returns the decryption result to ensure the app's functionality remains unaffected.
    return result;
  };
});
```

Let's analyze the the script.

Instruction	Description
<code>Java.perform(function () {...})</code>	The <code>Java.perform</code> is a Frida function that ensures the enclosed code is executed within the Java Virtual Machine (JVM). This is necessary for interacting with Java classes and methods in the targeted application.
<code>var Cipher = Java.use('javax.crypto.Cipher')</code>	The <code>Java.use</code> function is part of Frida's Java API, allowing the script to interact with Java classes. This line creates a reference to the <code>javax.crypto.Cipher</code> class from the Java Cryptography API that provides functionality for encryption and decryption operations.
<code>Cipher.doFinal.overload('[B').implementation = function (input) {...}</code>	This line of code hooks into the <code>doFinal</code> method of the <code>Cipher</code> class, specifically targeting the overload that takes a byte array ( <code>[B]</code> ) as input. The <code>overload</code> function specifies which version of the method to hook based on its parameters, and the <code>implementation</code> instruction is used to define a new implementation for the hooked method.

<pre>var result = this.doFinal(input)</pre>	Inside the custom implementation, the snippet first calls the original <code>doFinal</code> method with the provided input (the encrypted data) to perform decryption. The result of this operation, which is a decrypted byte array, is stored in the variable <code>result</code> .
<pre>var decryptedString = Java.use("java.lang.String").\$new(result, "UTF-8")</pre>	After the decryption, the script creates a new <code>String</code> object from the decrypted byte array, assuming the data is encoded in UTF-8. This step converts the binary data into a format readable for logging.
<pre>console.log("Decrypted string: " + decryptedString)</pre>	The decrypted string is logged and printed to the console.
<pre>return result</pre>	Finally, the script returns the original decryption result so that the app's functionality is not disrupted by the Frida script. This allows it to continue working with the decrypted data as if the hook was not in place.

Once the script is created, we can move on to setting up Frida. As a client-server tool, Frida requires the server to be installed on the device and the client to be installed on our host machine. Attention should be given to installing the same version on both sides, or there will be difficulties with running the snippets. Start by downloading the correct version from the official [GitHub project](#).

Hooking Java Methods

```
r11k@htb[/htb]$ wget https://github.com/frida/frida/releases/download/16.1.11/frida-server-16.1.11-android-arm64.xz  
  
<SNIP>  
frida-server-16.1.11-android-ar 100%[=====>] 14,85M 1,66MB/s in 17s  
2024-02-24 22:47:00 (907 KB/s) - 'frida-server-16.1.11-android-arm64.xz' saved [15566872/15566872]
```

Attention should also be given to selecting the correct architecture and matching Frida version. Since we've already installed Frida `16.1.11` on our host machine, it's important to use the same version for the Frida server running on the Android device. Additionally, the server binary must match the CPU architecture of the device—common architectures include `arm`, `arm64`, `x86`, and `x86_64`. In this example, the device uses an ARM64 architecture, so we'll download `frida-server-16.1.11-android-arm64.xz`.

Once downloaded, we'll decompress the file, rename it for convenience, push it to a world-writable directory on the device, assign execution permissions, and start the server.

Hooking Java Methods

```
r11k@htb[/htb]$ sudo apt install xz-utils  
r11k@htb[/htb]$ unxz frida-server-16.1.11-android-arm64.xz  
r11k@htb[/htb]$ mv frida-server-16.1.11-android-arm64 frida-server  
r11k@htb[/htb]$ adb push frida-server /data/local/tmp/  
r11k@htb[/htb]$ adb shell chmod +x /data/local/tmp/frida-server  
r11k@htb[/htb]$ adb shell /data/local/tmp/frida-server &  
  
[1] 99344
```

The output of the last command indicates the process has started and is running in the background. Issue the following command to confirm the Frida server's version.

Hooking Java Methods

```
r11k@htb[/htb]$ adb shell /data/local/tmp/frida-server --version  
  
16.1.11
```

After confirming the server version `16.1.11`, we can now install the same version of Frida client locally.

Hooking Java Methods

```
r11k@htb[/htb]$ pip3 install frida-tools  
r11k@htb[/htb]$ pip3 install frida==16.1.11  
  
<SNIP>  
Installing collected packages: frida  
  Attempting uninstall: frida
```



```
Found existing installation: frida 16.2.1
Uninstalling frida-16.2.1:
  Successfully uninstalled frida-16.2.1
Successfully installed frida-16.1.11
```

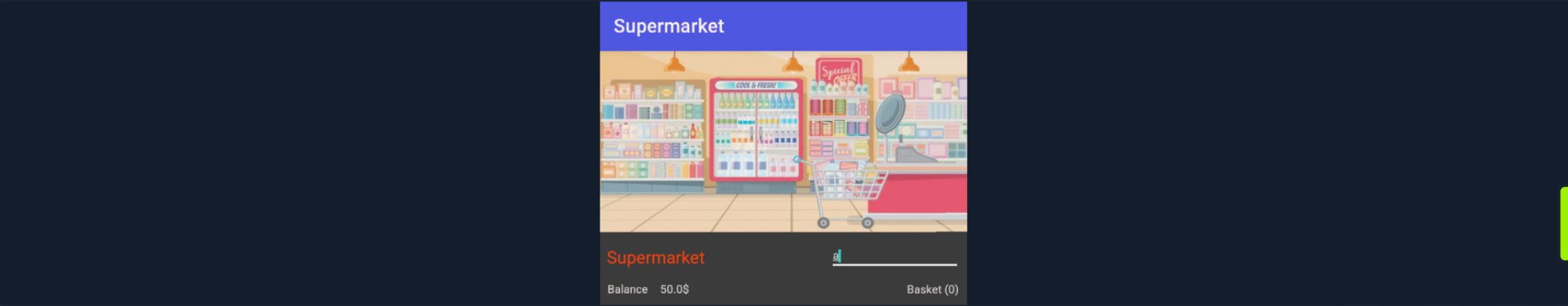
Now that everything is set up, issue the following command to hook the method.

Hooking Java Methods

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

----
/ _ |  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.supermarket`. Resuming main thread!
[Android Emulator::com.example.supermarket ]->
```

When the app starts, enter a random input to invoke the `onTextChanged()` method



Back in our host machine's terminal, the decrypted string is displayed—indicating a successful hook of the `doFinal()` method.

Hooking Java Methods

```
r11k@htb[/htb]$

<SNIP>
. . . .      Connected to Android Emulator (id=emulator-5554)
Spawned `com.example.supermarket`. Resuming main thread!
[Android Emulator::com.example.supermarket ]-> Decrypted string: HTB{Fr1d4_ho0k1ng_1ntro!}
```

Connect to Pwnbox

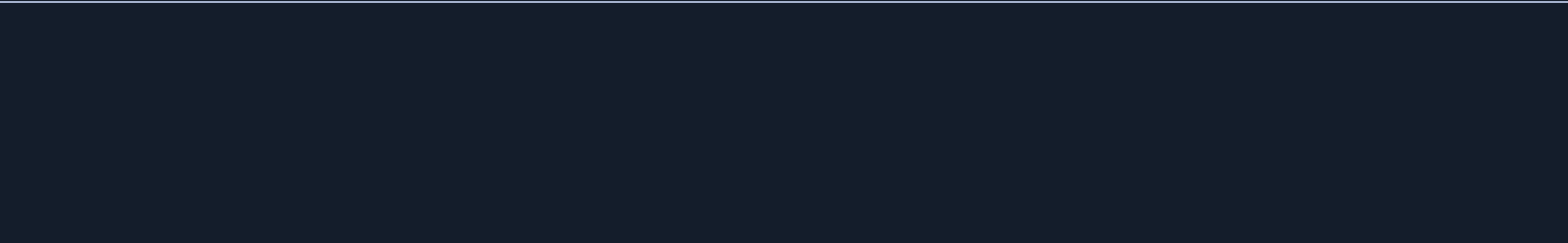
Your own web-based Parrot Linux instance to play our labs.

Pwnbox Location

UK

36ms

Terminate Pwnbox to switch location



Start Instance

∞ / 1 spawns left

Waiting to start...

Enable step-by-step solutions for all questions

Questions

Cheat Sheet

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

+ 5

What is the value of "Discount code" ?

Submit your answer here...

+10 Streak pts

Submit

hook\_java\_method.zip



← Previous

Next →

Cheat Sheet

Go to Questions


Table of Contents

Enumerating and Exploiting Installed Apps


Introduction
<div></div> Enumerating Local Storage
<div></div> Exported Activities
<div></div> Insecure Logging
<div></div> Pending Intents
<div></div> Exploiting WebViews
<div></div> Insecure Library Load Through Deep Linking


Dynamic Code Instrumentation


<div></div> <a href="#">Hooking Java Methods</a>
<div></div> Altering Method Values
<div></div> Hooking Native Methods
<div></div> 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

