

# Understanding Smali

When an Android application is compiled, the Java compiler (**javac**) compiles **.java** files into Java bytecode (**.class** files). Likewise, Kotlin source files are compiled into **.class** files by the Kotlin compiler (**kotlinc**). These are then translated into Dalvik bytecode, and the output is bundled into **.dex** (Dalvik Executable) files. Afterward, the Android Asset Packaging Tool (**aapt**) packages the compiled code, resources, and manifest into a single archive known as an APK file. When extracting and decoding the contents of an APK using APKTool, in addition to unpacking resources, a utility called **baksmali** is used to disassemble the Dalvik bytecode into a human-readable format known as Smali.

Smali is a low-level, assembly-like language used by the Dalvik Virtual Machine and Android Runtime (ART). It symbolically represents the structure and behavior of Android applications and is essential for understanding an app’s inner workings during static analysis. Having access to the disassembled source code in Smali format enables penetration testers to analyze functionality and potentially modify application logic—a process known as Application Patching. For example, security mechanisms such as root detection or license verification can be bypassed by altering Smali code. We will explore application patching in greater detail in upcoming sections.

To gain familiarity with Smali, let’s examine a simple example. The following is the original Java code for a class named **Calculate**:

Code: **java**

```
package com.hackthebox.myapp;

public class calculate {
    public static int add(int a, int b) {
        return a + b;
    }
}
```

The Smali representation of the above snippet is shown below.

Code: **assembly**

```
.class public Lcom/hackthebox/myapp/calculate;
.super Ljava/lang/Object;
.source "calculate.java"

# direct methods
.method public constructor <init>()V
    .locals 0

    .line 3
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    return-void
.end method

.method public static add(II)I
    .locals 0

    add-int/2addr p0, p1

    return p0
.end method
```

Here is a breakdown of the Smali representation:

Instruction	Description
<code>.class public Lcom/hackthebox/myapp/Calculate;</code>	Declares the public class <code>calculate</code> in the specified package.
<code>.method public constructor &lt;init&gt;()V</code>	Declares the default constructor method of the class.
<code>.method public static add(II)I</code>	Declares a static method <code>add</code> that takes two integer parameters (I, I) and returns an integer (I).
<code>.locals 0</code>	Specifies that no local (non-parameter) registers are used in this method.
<code>invoke-direct {p0}, Ljava/lang/Object; -&gt; &lt;init&gt;()V</code>	Calls the constructor of the superclass ( <code>Object</code> ) on the current object ( <code>p0</code> ).
<code>add-int/2addr p0, p1</code>	Adds the integer in register <code>p1</code> to the integer in <code>p0</code> , storing the result in <code>p0</code> .
<code>return p0</code>	Returns the value in <code>p0</code> as the result of the add method.

## Data Types

Smali uses specific shorthand representations for data types, known as **primitive types**. These types can be stored in registers or passed as method parameters. Below is a summary of the commonly used types:

Type	Description
V	Represents the type <code>void</code> . Used in methods that don't return a value.
Z	Represents the type <code>boolean</code> . It holds either of the two values: <code>true</code> or <code>false</code> .
B	Represents the type <code>byte</code> , an 8-bit integer.
S	Represents the type <code>short</code> , a 16-bit integer.
C	Represents the type <code>char</code> , a single 16-bit Unicode character.
I	Represents the type <code>int</code> , a 32-bit integer.
J	Represents the type <code>long</code> , a 64-bit integer.
F	Represents the type <code>float</code> , a 32-bit number.
D	This stands for <code>double</code> , a 64-bit number.

In addition to primitive types, Smali also supports **reference types** for objects and arrays:

Type	Description
L	Represents an object of a specific class. For example, <code>Ljava/lang/String;</code> refers to a <code>String</code> object.
[	Denotes an array. Used in combination with other types. For example, <code>[I</code> indicates an array of integers, and <code>[Ljava/lang/String</code> is an array of <code>String</code> objects.

These are the main data types used in Smali. It's worth noting that the `J` (long) and `D` (double) types take up two registers each because they're 64-bit data types, whereas all other types take up one register.

## Methods and Fields

Mthods are defined using the `.method` directive, followed by a series of modifiers, the method name, and the method's parameters and return type. The following Smali code defines a public static method named `add` that takes two integers as parameters and returns an integer.

Code: `assembly`

```
.method public static add(II)I
```

In the above method `add(II)I`, `II` represents two integer parameters, and `I` represents an integer return type. Methods are invoked using `invoke-` instructions, such as `invoke-virtual`, `invoke-super`, `invoke-direct`, `invoke-static`, and `invoke-interface`. These instructions are followed by the

registers containing the arguments to be passed to the method, as well as the method reference itself. This reference is composed of the package name (`Lpackage/name/`), the class name (`MyClass`), and finally the method signature (`add(II)I`).

Code: `assembly`

```
const v0, 5
const v1, 3

invoke-static {v0, v1}, Lpackage/name/MyClass; ->add(II)I
```

The following snippet shows what the Java equivalent of the method `add(II)I` would look like.

Code: `java`

```
public int add(int a, int b) {
    // Method body
}
```

In Smali, variables that belong to a class are known as `fields`. A field declaration in Smali consists of the following parts:

Field Part	Description
<code>Class Type</code>	The type of the class to which the field belongs.
<code>Field Name</code>	The identifier used to access the field within the class.
<code>Field Type</code>	The data type of the field. It defines the kind of data the field will store, such as int, float, String, etc.

A fully qualified field reference looks like this:

Code: `assembly`

```
Lpackage/name/ObjectName; ->FieldName:Ljava/lang/String;
```

The snippet below defines a public field named `myString` of type `String`.

Code: `assembly`

```
.field public myString:Ljava/lang/String;
```

## Registers

In Smali, like many low-level languages, we work directly with registers to perform operations. Dalvik bytecode uses registers instead of a stack to store variables and intermediate results. Understanding how these registers work is crucial to understanding the Smali code. Each register in the Dalvik Virtual Machine (DVM) is 32 bits in size and can hold any type of value that fits within those 32 bits. This includes the integer, boolean, and float data types, as well as references to objects or arrays. For example, consider the following Smali code:

Code: `assembly`

```
const v0, 10
```

In this case, `v0` is a register, and the number `10` is stored in it. For larger data types that require 64 bits, such as `Long` and `Double`, Dalvik uses a pair of registers. The lower 32 bits of the value go into the first register, and the upper 32 bits go into the second register. For example:

Code: `assembly`

```
const-wide v0, 0x100000000L
```

In this case, the 64-bit long integer `0x100000000L` is being stored across two registers, `v0`, and `v1`. This characteristic is important to remember when handling method arguments. Let's consider a non-static method, `LMyObject;->MyMethod(IJZ)V`. This method takes three parameters: an integer (I), a long (J), and a boolean (Z). Plus, since it's a non-static method, there is an implicit first parameter which is a reference to the instance of the object itself (`LMyObject;`). So, in total, this method needs five registers to hold all its parameters. This is because an int and a boolean each require one register, the long needs two registers because of its 64-bit size, and the reference to the object instance takes up another one. So, we end up with a total of five registers needed for all these parameters.

Register	Type
p0	this
p1	I
p2, p3	J
p4	Z

### Method Invocation and Registers

In Smali, `p` and `v` denote two different sets of registers, parameter and local registers, respectively. The `p` registers are used for method parameters. The numbering of `p` registers starts with `p0`, `p1`, `p2`, and so on. In non-static methods, `p0` refers to the object the method was called on (similar to `this` in Java). For static methods, `p0` refers to the first method argument. The `v` registers are used for local variables and temporary computations within the method. The numbering starts with `v0`, `v1`, `v2`, and so on. When invoking a method, arguments are passed in registers. The registers `p0`, `p1`, `p2`, `pn`, refer to the method's parameters. For instance:

Code: `assembly`

```
invoke-virtual {p0, v0}, Lcom/example/MyClass;->myMethod(I)V
```

Here, `p0` typically refers to the object instance on which `myMethod` is invoked (similar to Java's `this`) and `v0` is an integer parameter passed to the method. We also notice that the integer parameter of the method is `v0` and not `p1`. This occurs because Dalvik bytecode treats the `p` and `v` registers as part of the same set. Parameters can be referred to by either `p` or `v` names, as these distinctions are mainly designed to make the Smali code easier to read.

The available number of registers is declared at the beginning of the method using the `.registers` directive. For example, `.registers 5` declares that the method uses 5 registers, both `p` (parameter) registers and `v` (local variable).

Code: `assembly`

```
.method public exampleMethod(Ljava/lang/String;I)V
    .registers 5
<SNIP>
```

The alternate `.locals` directive specifies the number of non-parameter registers in the method.

Code: `assembly`



```
.method public exampleMethod(Ljava/lang/String;I)V
    .locals 5
<SNIP>
```

If a non-static method with one parameter declares `.registers 5`, then `p0` would be `this`, `p1` would represent the parameter, and `v0`, `v1`, and `v2` would be available for local variables. However, internally, `p0` is the same as `v4`, `p1` is the same as `v3`, `v0` is the same as `v2`, `v1` as `v1`, and `v2` as `v0`. The mapping is done in reverse order, starting from the total register count minus one for `p0`/`this`.






 Cheat Sheet

Table of Contents


Extracting and Enumerating APK Files

-  Introduction
-  Disassembling the APK
- [Understanding Smali](#)





Analyzing Application's Source Code

-  Reading Hardcoded Strings
-  Bad Cryptography Implementation
-  Reversing Hybrid Apps
-  Reading Obfuscated Code
-  Deobfuscating Code

Analyzing Native Libraries

-  Reversing Shared Objects
-  Reversing DLL Files

Application Patching


-  Authentication Bypass
-  Modifying Game Apps
-  License Verification Bypass
-  Root Detection Bypass


Skills Assessment

-  Skills Assessment

My Workstation

OFFLINE

 Start Instance

 / 1 spawns left