Reading Hardcoded Strings

Sensitive information that may be accidentally or improperly stored in an Android application can encompass a wide variety of data types. During a penetration test, it is crucial to examine the application for any indicators that such data may be stored or transmitted insecurely. Below are common examples of sensitive strings that may be discovered within an Android application:

Title	Description
`API keys`	Often used to authenticate a client app's requests to a server. If stored insecurely, API keys can be stolen and misused by malicious actors.
`Database credentials`	If an application interacts with a database, it may require a username and password. These should never be stored in plain text.
`OAuth tokens`	Used for authenticated `communication` between the app and services like Google, Facebook, Twitter, etc. A malicious user with access to these tokens can potentially impersonate the app or user.
`Cryptographic keys`	Employed to encrypt and decrypt sensitive data. If a key is stored insecurely, an attacker might be able to decrypt sensitive information.
`Hardcoded passwords or passphrases`	These may be used to access restricted parts of the application or to enable leftover debugging functionality in production builds.
`Personally identifiable information (PII)`	Includes data such as names, email addresses, social security numbers, and credit card information. This type of data must be handled with strict security measures.
`Sensitive URLs or IP addresses`	Can reveal information about backend servers or services the application interacts with, potentially aiding in targeted attacks.
`Debugging information`	May include verbose error messages, stack traces, or internal implementation details that can assist an attacker in understanding or exploiting the app.

When conducting a security review or penetration test, one should always look for these sensitive pieces of information and recommend secure storage practices. This includes using encryption, storing the data on secure servers, and accessing it through secure connections when necessary.

Analyzing with JADX and APKTool

For Android application static analysis, understanding the underlying Java code is crucial. JADX (Java Decompiler for Android) is a popular open-source tool for decompiling Android applications. It takes APKs and converts them back into readable Java source code. While the decompiled output is not identical to the original source code, it provides a representation close enough to analyze the application's behavior and logic. JADX offers several key features that support effective static analysis, including:

Feature	Description
`File Formats`	Decompiles Dalvik bytecode to java classes from APK, dex, aar, aab, and zip files.
`GUI and CLI`	Provides both a GUI version (`jadx-gui`) for a more visual experience and a CLI version (`jadx`) for scripting and automation.
`Decoding Files`	Decodes `AndroidManifest.xml` and other resources from `resources.arsc`.
`Deobfuscation`	Renames obfuscated methods and variables with meaningful names to make the code easier to read.
`Smali`	Supports Smali representation and debugging through ADB.
`Other Functionalities`	Full-text search, highlighted syntax, jump to declaration, Annotation.

It's essential to highlight the importance of JADX in understanding what an application does under the hood. Reading the decompiled code gives insights into potential vulnerabilities, logic flaws, or suspicious behaviors that might not be visible just by examining the application's behavior. At this point, It is worth mentioning that despite all these features JADX provides, it won't allow code editing and recompilation. As we mentioned earlier, JADX is a CLI tool, but it also provides a GUI, which is the one that we will be using in this section. Below, we will see various examples that will give us a

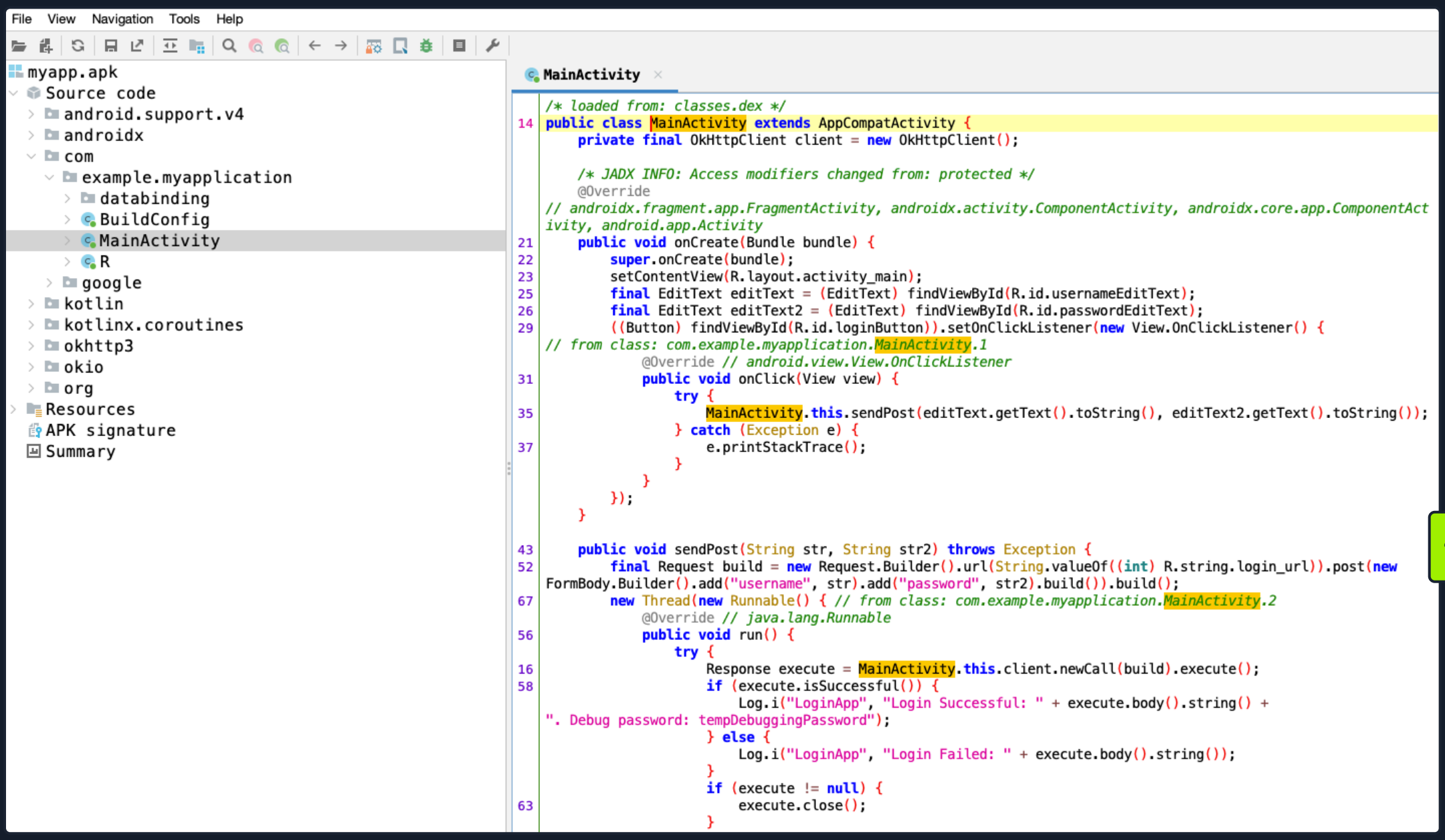
better image of what we can do with this tool, while in some examples, we will also use APKTool.

Reading Strings from Source Code

The `res`, `smali`, and `libs` directories often contain files with hardcoded strings, making them a logical starting point for our examination. These common directories can be analyzed using tools like JADX and APKTool. JADX is available for download from its official Github repository. On Debian-based Linux distributions, JADX-GUI can be installed and launched using the following commands.

Reading Hardcoded Strings

```
rl1k@htb[/htb]$ sudo apt upgrade
rl1k@htb[/htb]$ sudo apt update
rl1k@htb[/htb]$ sudo apt install jadx
rl1k@htb[/htb]$ jadx-gui /full/path/to/myapp.apk
```



In the image above, we have started JADX-GUI and loaded the `myapp.apk`. As we can see in the left-side menu, under the `Source code` -> `com` -> `example.myapplication`, there is the decompiled code of the `MainActivity`. Examination of the app's source code reveals a login functionality over an HTTP post request. In line 58, we see the forgotten hardcoded password `tempDebuggingPassword`, which is left there for debugging purposes. Hardcoded strings like this must be removed from production applications.

Let's walk through how to retrieve this hardcoded information using `APKTool`. First, we'll decompile and decode the APK file.

Reading Hardcoded Strings

```
rl1k@htb[/htb]$ apktool d myapp.apk

I: Using Apktool 2.7.0 on myapp.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /Users/bertolis/Library/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
I: Copying META-INF/services directory
```

A directory called `myapp` is created containing all the decoded resources and the Smali representation of the source code. As a first step, we could run the `grep` command searching for keywords like the following.

Reading Hardcoded Strings

```
rl1k@htb[/htb]$ grep -Rnw './myapp' -e 'password'

<SNIP>
./smali/com/example/myapplication/MainActivity.smali:106:    const-string v0, "password"
./smali/com/example/myapplication/MainActivity$2.smali:97:    const-string v3, ". Debug password: tempDebuggingPassword"
```

The above command revealed the `Debug password: tempDebuggingPassword`, which we found earlier in the `MainActivity` within the decompiled source code with JADX-GUI. Another key-word example could be:

Reading Hardcoded Strings

```
rl1k@htb[/htb]$ grep -Rnw './myapp' -e 'api_key'

./res/values/public.xml:3560:    <public type="string" name="api_key" id="0x7f0f001d" />
./res/values/strings.xml:32:    <string name="api_key">12345678-ABCD-EFGH-IJKL-1234567890AB</string>
./smali/com/example/myapplication/R$string.smali:19:.field public static final api_key:I = 0x7f0f001d
```

The command above revealed the key-value pair `12345678-ABCD-EFGH-IJKL-1234567890AB`, an API key stored in the `Strings.xml`. The key-words `password` and `api_key` are common names used while developing apps. The above commands will save us time while enumerating the app, and often reveal common mistakes. Our next move is to search the decoded files manually for additional hardcoded strings or any other sensitive information. When decompiling an APK with APKTool, the generated Smali files will be stored in the directory `smali`, and more specifically, in our case, under `./myapp/smali/com/example/myapplication/`. Listing the content of this directory reveals the following.

Reading Hardcoded Strings

```
rl1k@htb[/htb]$ ls -l ./myapp/smali/com/example/myapplication/

total 120
-rw-r--r--  1 bertolis  bertolis   619 Sep  6 20:22 BuildConfig.smali
-rw-r--r--  1 bertolis  bertolis  2575 Sep  6 20:22 MainActivity$1.smali
-rw-r--r--  1 bertolis  bertolis  4806 Sep  6 20:22 MainActivity$2.smali
-rw-r--r--  1 bertolis  bertolis  4216 Sep  6 20:22 MainActivity.smali
-rw-r--r--  1 bertolis  bertolis   855 Sep  6 20:22 R$color.smali
-rw-r--r--  1 bertolis  bertolis   629 Sep  6 20:22 R$drawable.smali
-rw-r--r--  1 bertolis  bertolis   712 Sep  6 20:22 R$id.smali
-rw-r--r--  1 bertolis  bertolis   550 Sep  6 20:22 R$layout.smali
-rw-r--r--  1 bertolis  bertolis   609 Sep  6 20:22 R$mipmap.smali
-rw-r--r--  1 bertolis  bertolis  1794 Sep  6 20:22 R$string.smali
-rw-r--r--  1 bertolis  bertolis   554 Sep  6 20:22 R$style.smali
-rw-r--r--  1 bertolis  bertolis   675 Sep  6 20:22 R$xml.smali
-rw-r--r--  1 bertolis  bertolis   696 Sep  6 20:22 R.smali
drwxr-xr-x  3 bertolis  bertolis    96 Sep  6 20:22 databinding
```

Reading the content of one of the `"MainActivity"` files will reveal the Smali representation of the source code from `MainActivity.java`(which contains the hardcoded string we found earlier). In this case, it is the `MainActivity$2.smali`.

Reading Hardcoded Strings

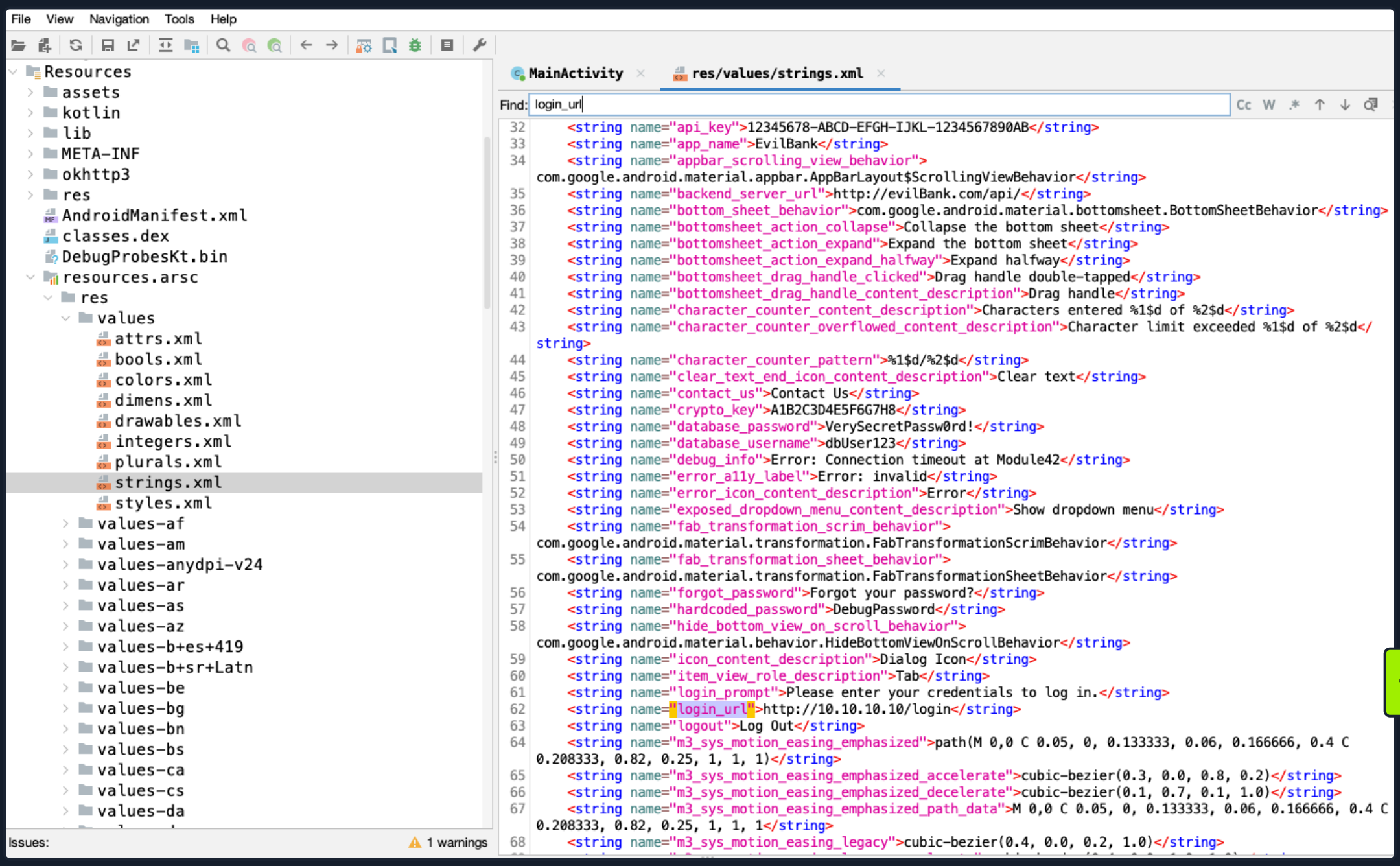
```
rl1k@htb[/htb]$ cat ./myapp/smali/com/example/myapplication/MainActivity\$2.smali

<SNIP>
    move-result-object v3
    invoke-virtual {v3}, Lokhttp3/ResponseBody;->string()Ljava/lang/String;
    move-result-object v3
    invoke-virtual {v1, v3}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
    const-string v3, ". Debug password: tempDebuggingPassword"
    invoke-virtual {v1, v3}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
    invoke-virtual {v1}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;
```

In this directory, we can also find the Smali representation for all of the application's activities and classes.

Reading Strings from strings.xml

In line 52, we can also see the `R.string.login_url` integer. Breaking it down, we should know several things: the `R.java` file contains the IDs of all the resources in the `res/` directory, `string` is a static class within `R` that provides references to string resources defined acrossthe project, and `login_url` is the ID given to a specific string resource. With the string's ID, we can proceed to search inside the strings.xml file, where the hardcoded strings are typically kept in Android apps. Navigating to the `Resources` -> `resources.arsc` -> `res` -> `values` -> `strings.xml`, we can see the following content.



Using `Control+F` or `Command+F`, we can search using the ID of the string. In the above image we see the value `http://10.10.10.10/login`, which represents the server's IP and a login page that can be further enumerated. Other sensitive information like the `crypto_key`, `database_password`, and `api_key`, are also stored in this file. In production applications, sensitive strings like these should never be stored directly within `strings.xml`—or any other resource file—without proper encryption or obfuscation.

APKTool can extract the `strings.xml` file, typically located under the `./myapp/res/values` directory. The `res` directory contains various application resources, including layouts, images, and other static files. It also includes subdirectories such as `values` and `xml`, which store resource definitions and configuration data. Upon examining the `./myapp/res/values/strings.xml` file, we can identify hardcoded string values referenced in the application's code via the `R.java` class.

Reading Hardcoded Strings

```
rl1k@htb[/htb]$ cat ./myapp/res/values/strings.xml

<SNIP>
    <string name="api_key">12345678-ABCD-EFGH-IJKL-1234567890AB</string>
    <string name="app_name">EvilBank</string>
    <string name="backend_server_url">http://evilBank.com/api/</string>
    <string name="bottom_sheet_behavior">com.google.android.material.bottomsheet.BottomSheetBehavior</string>
    <string name="bottomsheet_action_collapse">Collapse the bottom sheet</string>
    <string name="bottomsheet_action_expand">Expand the bottom sheet</string>
```

As we can see in the results, the API key `12345678-ABCD-EFGH-IJKL-1234567890AB` is included in this file in plaintext.

Reading Strings from network_security_config.xml

Moving on to examining the `xml` directory for strings won't reveal much sensitive information. Still, the file `res/xml/network_security_config.xml` will give us a good picture of the app's connections with any potential API endpoints. Therefore, this file will reveal at least a Domain Name or an IP. The following command will list its contents.

Reading Hardcoded Strings

```
rl1k@htb[/htb]$ cat ./myapp/res/xml/network_security_config.xml

<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <base-config cleartextTrafficPermitted="false">
        <trust-anchors>
            <certificates src="system" />
            <certificates src="@raw/certificate" />
        </trust-anchors>
    </base-config>
    <domain-config cleartextTrafficPermitted="true">
        <domain includeSubdomains="true">192.168.1.8</domain>
    </domain-config>
</network-security-config>%
```

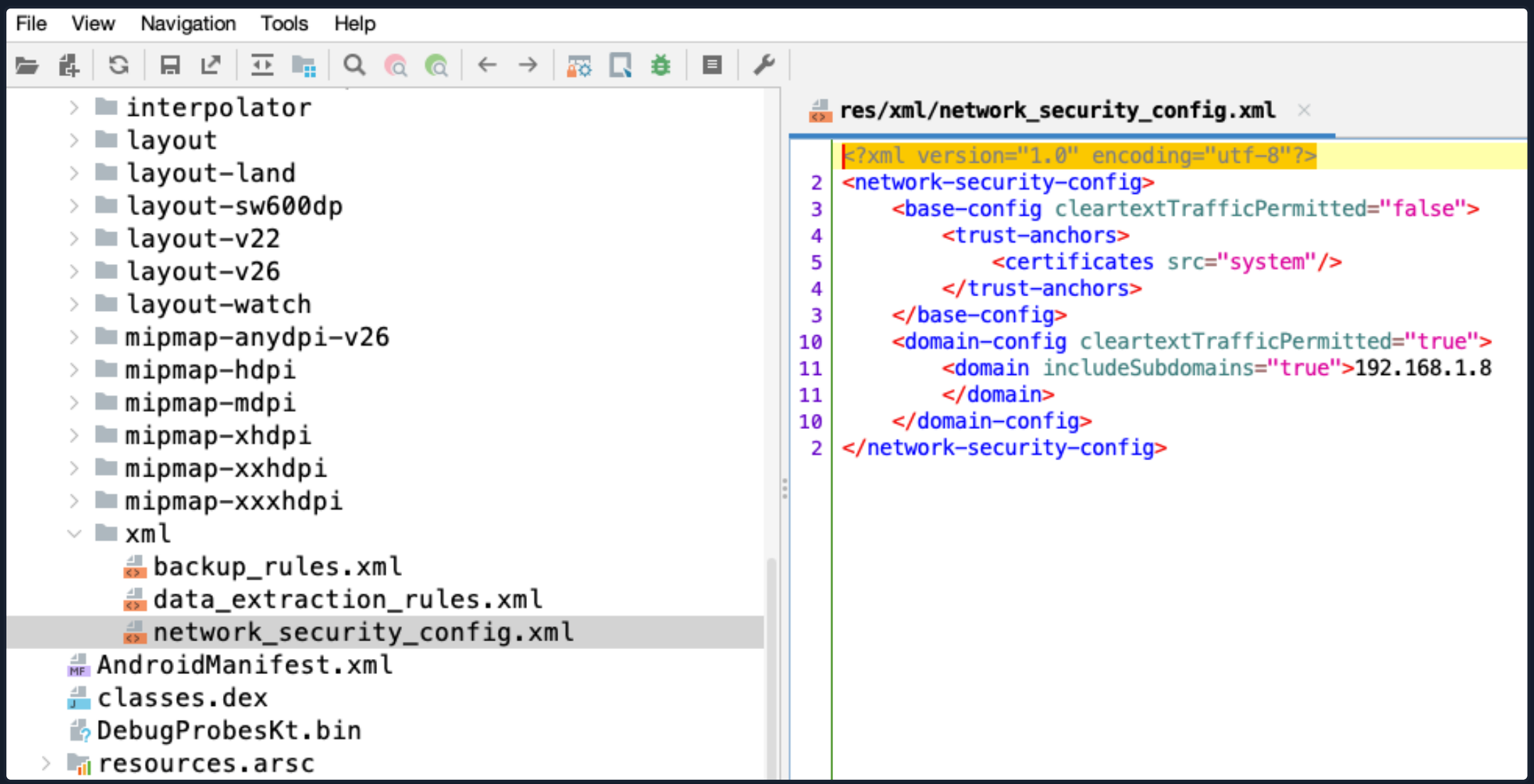
Reading the content of this file reveals that the app does not allow unencrypted communication (HTTP), except for communicating with the URL 192.168.1.8. Having an IP and the insight that an unencrypted connection is used, the attacking surface is getting bigger. The file also indicates that certificates stored in the system can be used for establishing the secure connection or the custom self-signed certificate specified in `<certificates src="@raw/certificate" />`. The certificate can be found in the directory `./myapp/res/raw`.

Reading Hardcoded Strings

```
rl1k@htb[/htb]$ ls -l ./myapp/res/raw

total 8
-rw-r--r--  1 bertolis  bertolis  1029 Jun 15 14:55 certificate.der
```

JADX can also reveal this information by navigating `Resources` -> `res` -> `xml` -> `network_security_config.xml`.



Reading Strings from Shared Libraries

applications store their native C++ code in the directory `./myapp/lib/x86_64/`. In this example, the code is inside libmyapplication.so, and although the content of this file is compiled, hardcoded strings can still be retrieved. Using JADX to read the content of this file will not be successful because it

doesn't work with native shared libraries. However, reading the content using Vim reveals the following.

Reading Hardcoded Strings

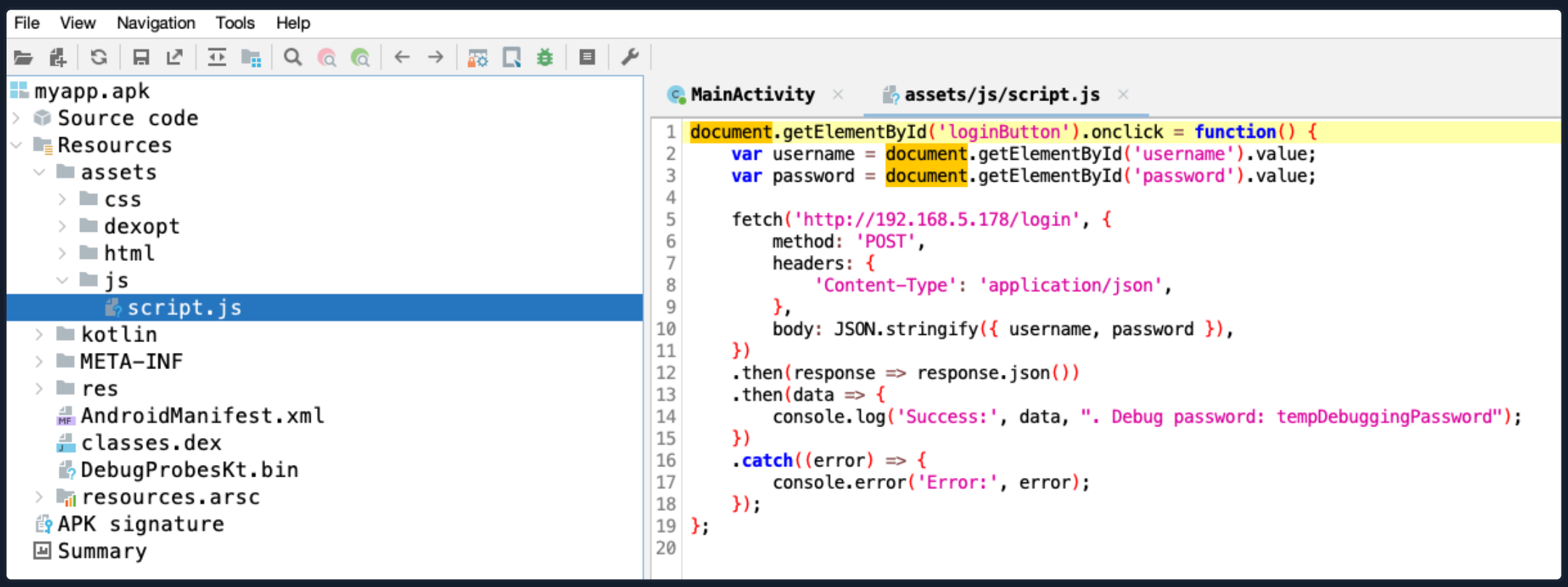
```
rl1k@htb[/htb]$ vim ./myapp/lib/x86_64/libmyapplication.so
```

```
92 ^@rdx^@rbp^@xmm6^@virtual thunk to ^@+=^@>>=^@ul^@%af^@operator*^@operator||^@istream^@ &&^@char8_t^
   d size^@Unknown DWARF encoding for search table.^@rsi^@r12^@unexpected_handler unexpectedly returned
    ^@=^@||^@__int128^@template<^@operator<<^@operator+^@operator+=^@operator++^@string literal^@unsign
   @libunwind: malformed DW_CFA_expression DWARF unwind, reg too big
93 ^@^^@|=^@+^@vE^@operator=^@operator/=^@bool^@libunwind: malformed DW_CFA_restore DWARF unwind, reg (
94 ^@evaluateExpression^@r10^@<<=^@operator^^@ restrict^@decimal64^@libunwind: malformed DW_CFA_undefin
   big
95 ^@terminating with %s exception of type %s: %s^@non-virtual thunk to ^@->^@...^@std::basic_istream<c
   r> >^@iostream^@pixel vector[^@union^@_Unwind_Resume() can't return^@libunwind: malformed DW_CFA_reg
   oo big
96 ^@<<^@yptn^@operator%=^@ const^@libunwind: malformed DW_CFA_offset DWARF unwind, reg (%lu) out of ra
97 ^@DW_OP_fbreg not implemented^@terminating with %s foreign exception^@uncaught^@!=^@operator--^@oper
   am<char, std::char_traits<char> >^@_Unwind_Resume^@libunwind: malformed DW_CFA_def_cfa_sf DWARF unwi
98 ^@xmm12^@<=^@%=^@'block-literal'^@operator-^@basic_istream^@std::istream^@double^@xmm14^@invocation
   tatic_cast^@sizeof... (^@ull^@(^@operator-=^@unsupported x86_64 register^@libunwind: malformed DW_CF
   F unwind, reg too big
99 ^@getULEB128^@malformed uleb128 expression^@DWARF opcode not implemented^@rbx^@&&^@|^@nullptr^@opera
   ^@libunwind: malformed DW_CFA_val_expression DWARF unwind, reg too big
100 ^@xmm4^@hardcodedSensitiveInformation^@terminating^@&^@~^@reinterpret_cast^@l^@, ^@DW_EH_PE_textrel
   orted^@truncated uleb128 expression^@rax^@rcx^@xmm0^@^@operator ^@throw ^@basic_string^@void^@float^
   alformed DW_CFA_restore_extended DWARF unwind, reg too big
```

Though it's hard for someone to understand the meaning of random strings when reading the content of this file, some might be pretty straightforward, like the string `hardcodedSensitiveInformation` we found in the above example.

Reading Strings from JS

Reading hardcoded strings from a web-based Android application can be done using both JADX and APKTool. Web-based apps store their code inside `.js` files under the `assets` directory. Navigating to `Resources` -> `Assets` -> `js` in JADX, we see the file `script.js` containing the following code.



The code above exposes the hardcoded URL `http://192.178.5.178/login` and the plaintext password `tempDebuggingPassword`. These findings can also be obtained using APKTool, which decompiles the application and allows direct inspection of the `./myapp/assets/js/script.js` file.

Reading Hardcoded Strings

```
rl1k@htb[/htb]$ cat ./myapp/assets/js/script.js

document.getElementById('loginButton').onclick = function() {
    var username = document.getElementById('username').value;
    var password = document.getElementById('password').value;

    fetch('http://192.168.5.178/login', {
        method: 'POST',
        headers: {
```

```
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({ username, password }),
    })
    .then(response => response.json())
    .then(data => {
        console.log('Success:', data, ". Debug password: tempDebuggingPassword");
    })
    .catch((error) => {
        console.error('Error:', error);
    });
};
```

Connect to Pwnbox

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

Pwnbox Location

UK 27ms

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!

+ 3 What is the value of the "oauth_token"?

Submit your answer here...

+10 Streak pts Submit myapp_hardcoded_strings.zip

Cheat Sheet

Go to Questions

Table of Contents

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left