Reversing Shared Objects

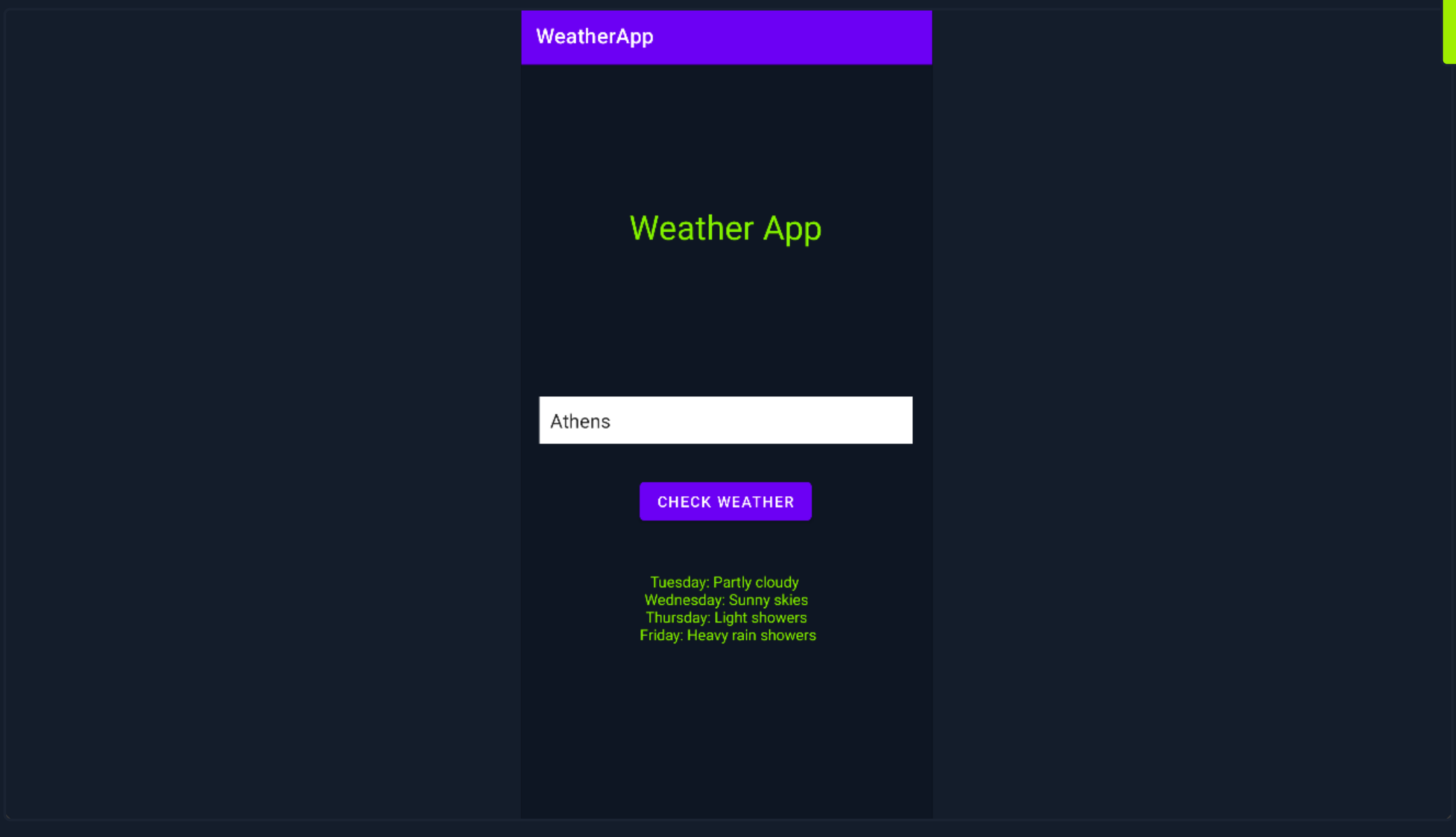
As we briefly discussed in the Native Code section of Android Fundamentals, Android applications sometimes include native C++ code, typically integrated through the Native Development Kit (NDK). Developers may use native code to improve performance, reduce latency, or add a layer of complexity that makes reverse engineering more difficult. When the application is compiled, any native code is packaged into shared object (.so) files within the APK.

These shared objects are susceptible to the same types of security issues we've already been investigating—such as hardcoded credentials, obfuscation logic, or exploitable flaws—only now they may be buried deeper in compiled native code. Analyzing `.so` files is therefore an important part of the broader reverse engineering process and plays a key role in advanced app analysis and malware research, which we'll explore in more detail in future modules.

While JADX is commonly used to analyze the Java code of an application, tools like Ghidra, IDA Pro and RADARE2 are preferred for examining shared objects. In our upcoming example, we'll use Ghidra to analyze the native code.

Reading Strings Directly from SO Files

The following example features a weather forecast app that uses a private API key to fetch remote data and display it locally. As seen in several previous labs, uncovering hardcoded or improperly stored API keys is a recurring issue in mobile application security. If a private API key falls into the wrong hands, it can lead to a variety of damaging outcomes depending on the key's privileges and intended use. An attacker could gain unauthorized access to backend systems, incur financial costs by abusing paid services, exfiltrate or manipulate data, disrupt service functionality, or impersonate legitimate users. The screenshot below shows the application's main screen. The screenshot below shows the application's main screen.



The decompiled code in JADX reveals the following.

```
/* loaded from: classes.dex */
public class MainActivity extends AppCompatActivity {
    Button checkWeatherButton;
    EditText cityEditText;
    TextView resultTextView;

    public native String getAPIKey();
```

```
static {
    System.loadLibrary("myapp");
}
```

Reading the Java code of the app, we can understand that the shared library called `myapp` is used, and the native method `getAPIKey()` is initialized. A few lines below, we notice that there is an HTTP request to the URL `http://api.weather.org/data/2.5/weather` using the value retrieved from the native method `getAPIKey()`.

```
        /* JADX INFO: Access modifiers changed from: protected */
        @Override // android.os.AsyncTask
        public String doInBackground(String... strArr) {
            String str = strArr[0];
            try {
                HttpURLConnection httpURLConnection = (HttpURLConnection) new URL(
"http://api.weather.org/data/2.5/weather?q=" + str + "&appid=" + MainActivity.this.getAPIKey()).
openConnection();
```

Let's check if we can retrieve the API key from the shared library. First, we'll extract the files from the APK using either APKTool or `unzip`.

Reversing Shared Objects

```
rl1k@htb[/htb]$ unzip myapp.apk
```

This will extract the file `lib/x86_64/libmyapp.so`, which, as the name implies, is the name of the library (`myapp`) with the prefix `lib`. Issuing the `strings` command to search the file for the API key and using `grep` to filter the output (for the words `api`, `key`, or other possible combinations) unfortunately does not reveal the API key. However, another approach is to use regular expressions to filter the result.

Reversing Shared Objects

```
rl1k@htb[/htb]$ strings lib/x86_64/libmyapp.so | grep -E "[a-zA-Z0-9_-]{60,}"

xmjPceil0E5ekn6QisfF1XLVSxq3n7HkfK9duVJxaqLPxZ4eB9EiYacvgswubvKZ
N12_GLOBAL__N_116itanium_demangle24ForwardTemplateReferenceE
N12_GLOBAL__N_116itanium_demangle26SyntheticTemplateParamNameE
N12_GLOBAL__N_116itanium_demangle24NonTypeTemplateParamDeclE
N12_GLOBAL__N_116itanium_demangle25TemplateTemplateParamDeclE
N12_GLOBAL__N_116itanium_demangle27ExpandedSpecialSubstitutionE
```

The above command will search for strings in the file that are 60 characters or more. If there are no results, we can try lowering the character length until we see some output. In this example, we extract the string `xmjPceil0E5ekn6QisfF1XLVSxq3n7HkfK9duVJxaqLPxZ4eB9EiYacvgswubvKZ`, a potential API key we could use to examine the application's endpoints further.

Reading Hardcoded Strings With Ghidra

Now, let's assume that the output was too big, or that there was was no way to tell which string was the API key. In such cases, we would need an alternative way to retrieve it. Fortunately, this can be done with Ghidra. In Debian-based Linux, Ghidra can be installed as follows.

Reversing Shared Objects

```
rl1k@htb[/htb]$ apt install ghidra
```

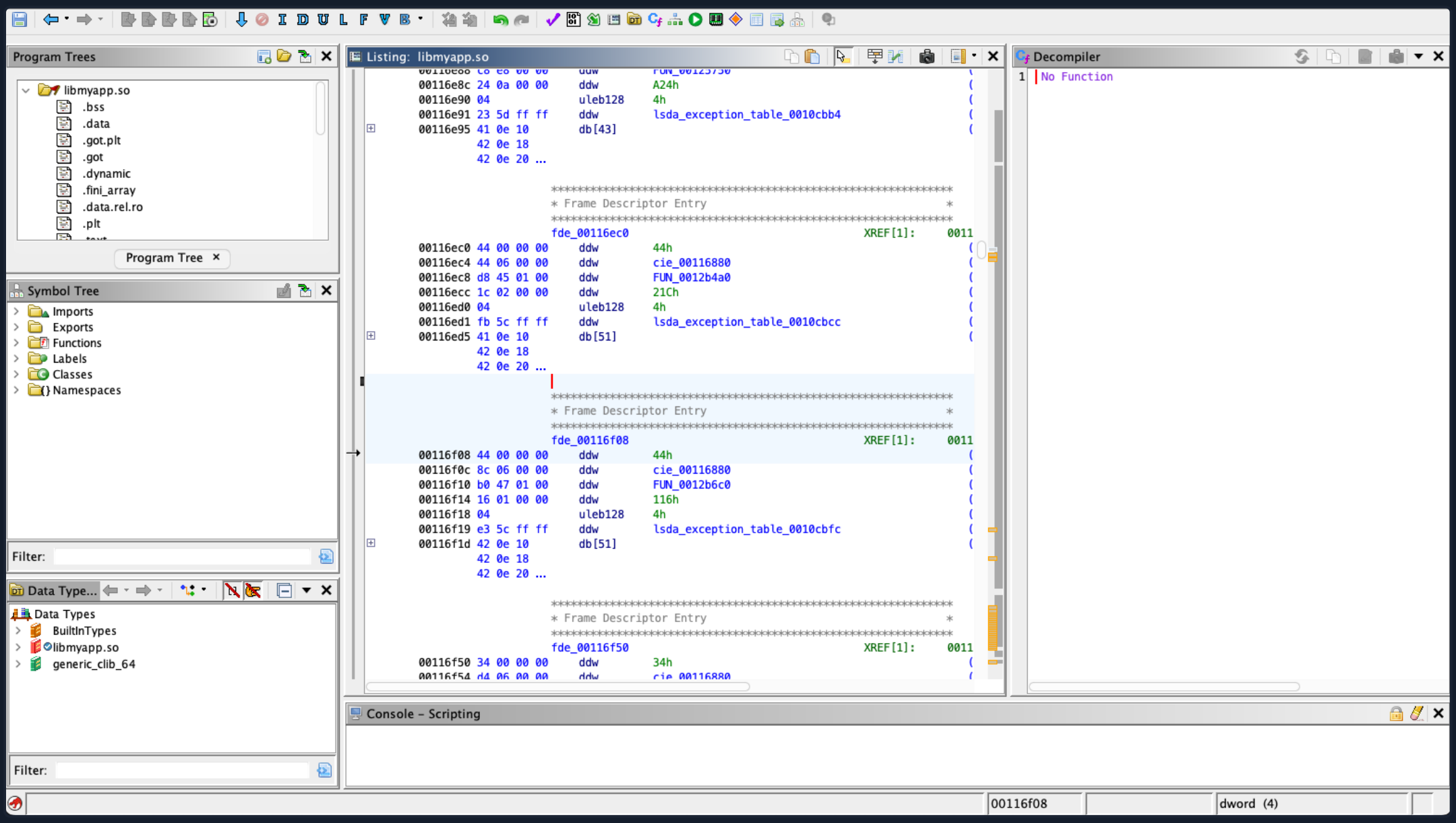
Once it's done installing, we can launch it from the command line.

Reversing Shared Objects

```
rl1k@htb[/htb]$ ghidra
```

When Ghidra is done launching, click on `File` -> `New Project` -> `Non-Shared Project` -> `Next`. Then, give a Project Name and click `Next` again. Once the project is created, click on `File` -> `Import File`, navigate to the `libmyapp.so` file, click `Select File To Import` and click `OK` to the following pop-up windows. Next, double-click on the imported file, click `Yes` and `Analyze` on the next pop-up window. The following image shows Ghidra's layout after

loading a shared library.

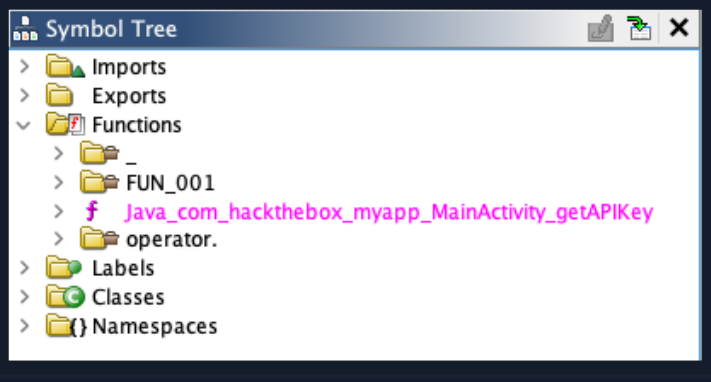


Below is a quick overview of the basic panes we'll be working with in Ghidra while reversing shared libraries.

Pane	Description
`Symbol Tree`	Found on the left side of the window, this presents the symbols within the binary being analyzed in a structured, hierarchical manner. Symbols represent the entities of the program and include functions, variables, labels, and more. This helps to quickly navigate to and identify interesting parts of a binary.
`Listing`	This pane displays the disassembled code of the binary by representing the low-level machine instructions in a human-readable assembly language. Alongside the code, addresses, byte values of the instructions, and other relevant comments or annotations can also be found. This pane provides a granular view of the program's operation at the instruction level.
`Decompiler`	One of the most powerful tools in Ghidra, it can translate the assembly code from the Listing pane into a higher-level representation, making it easier for the user to read and understand.

Checking the functions used in this file under `Symbol Tree` at the left of the window reveals the function

`Java_com_hackthebox_myapp_MainActivity_getAPIKey`.



Let's click on it and check the decompiled code in the `Decompile` window on the right.

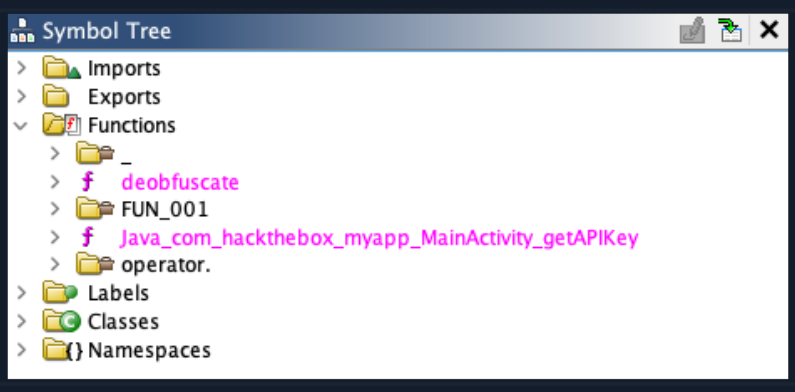
```
local_10 = *(long *)(in_FS_OFFSET + 0x28);
local_27._0_1_ = 'B';
local_27._1_1_ = '9';
local_27._2_1_ = 'E';
local_27._3_1_ = 'i';
uStack_23._0_1_ = 'Y';
uStack_23._1_1_ = 'a';
uStack_23._2_1_ = 'c';
uStack_23._3_1_ = 'v';
uStack_1f._0_1_ = 'g';
uStack_1f._1_1_ = 's';
uStack_1f._2_1_ = 'w';
uStack_1f._3_1_ = 'u';
uStack_1b._0_1_ = 'b';
uStack_1b._1_1_ = 'v';
uStack_1b._2_1_ = 'K';
```

```
  uStack_1b._3_1_ = 'Z';
  uStack_17 = 0;
                    /* try { // try from 001170a0 to 001170a5 has
  uVar1 = (**(code **)(*param_1 + 0x538))(param_1,&local_27);
  if (*(long *)(in_FS_OFFSET + 0x28) != local_10) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return uVar1;
}
```

The above table of characters reveals the string `B9EiYacvgswubvKZ`. Given the name of the function and the fact that this is its return value, we've found a candidate for a possible API key.

Reading Code With Ghidra

In the previous example, we saw how to retrieve hardcoded strings from a shared library using Ghidra. In the following example, we will try to read and understand the code in order to retrieve the API key. Looking at the `Symbol Tree` pane under the `Functions`, we can see the functions `Java_com_hackthebox_myapp_MainActivity_getAPIKey` and `deobfuscate`.



Let's click on the first on the `Java_com_hackthebox_myapp_MainActivity_getAPIKey` and check the `Decompiler` pane.

```
undefined8 Java_com_hackthebox_myapp_MainActivity_getAPIKey(long *param_1)

{
  undefined8 uVar1;
  undefined *puVar2;
  long in_FS_OFFSET;
  basic_string local_40;
  undefined local_3f [15];
  undefined *local_30;
  uint local_28;
  undefined4 uStack_24;
  undefined4 uStack_20;
  undefined4 uStack_1c;
  undefined4 *local_18;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_18 = (undefined4 *)operator.new(0x30);
  local_28 = 0x31;
  uStack_24 = 0;
  uStack_20 = 0x20;
  uStack_1c = 0;
  local_18[4] = 0xb9998c96;
  local_18[5] = 0xa9b3a7ce;
  local_18[6] = 0xcc8e87ac;
  local_18[7] = 0x94b7c891;
  *local_18 = 0xaf959287;
  local_18[1] = 0x93969a9c;
  local_18[2] = 0x9acabacf;
  local_18[3] = 0xaec99194;
  *(undefined *)(local_18 + 8) = 0;
                    /* try { // try from 0012012a to 00120138 has its CatchHandler @ 001201c7 */
  deobfuscate(&local_40);
  puVar2 = local_30;
  if (((byte)local_40 & 1) == 0) {
    puVar2 = local_3f;
  }
```

The above snippet is the high-level representation of the assembly code of the shared object. As we can see, various local variables and memory allocations are set up for the execution. Let's break them down so we can better understand what is happening. The `local_18` is dynamically allocated 0x30 (48 bytes) bytes of memory, filled with some hexadecimal values, likely representing the obfuscated API key. The `deobfuscate();` function is called with the address of `local_40` as its argument. Although there is no direct connection between `local_18` and `deobfuscate()`, we could still assume that the value passed is eventually the hexadecimal values contained in the `local_18` variable. A logical explanation for why this is not clear could be that due to an oversight in the reverse-engineering process, there might have been a function that populates `local_40` using the data from `local_18` before the `deobfuscate()` function is called, which is not shown. Let's double-click on the `deobfuscate()` to further examine its functionality.

```
basic_string * deobfuscate(basic_string *param_1)

{
  ulong uVar1;
  byte *in_RSI;
  ulong uVar2;
  byte *pbVar3;
```

```
*(undefined (*) [16])param_1 = ZEXT816(0);
*(undefined8 *)(param_1 + 0x10) = 0;
if ((*in_RSI & 1) == 0) {
  pbVar3 = in_RSI + 1;
  uVar2 = (ulong)(*in_RSI >> 1);
}
else {
  uVar2 = *(ulong *)(in_RSI + 8);
  pbVar3 = *(byte **)(in_RSI + 0x10);
}
if (uVar2 != 0) {
  uVar1 = 0;
  do {
                    /* try { // try from 00120097 to 0012009e has its CatchHandler @ 001200b7 */
    std::__ndk1::basic_string<char,std::__ndk1::char_traits<char>,std::__ndk1::allocator<char>>::
    push_back((basic_string<char,std::__ndk1::char_traits<char>,std::__ndk1::allocator<char>> *)
              param_1,~pbVar3[uVar1]);
    uVar1 = uVar1 + 1;
  } while (uVar2 != uVar1);
}
return param_1;
}
```

Inside the `deobfuscate()` function, we can see that the memory structure of the variable `in_RSI` is checked. We can assume that this is the obfuscated API key since, in many calling conventions, the `RSI` register is used to pass the function's second argument. If its least significant bit is 0, the string is short and stored inline. Otherwise, the string is long and stored in dynamically allocated memory. The function then goes through each string character and appends its bitwise negation to `param_1`. In simpler terms, it's flipping the bits for each character. This is a really simple obfuscation method that should not be used in production. The value `param_1` is then returned to the `Java_com_hackthebox_myapp_MainActivity_getAPIKey()` function.

Now that we know the code's functionality in the shared library, we can try to retrieve the initial API key. To shape the obfuscated string, we will need to take the hexadecimal values stored in the variable `local_18` and arrange them in the correct order. Before starting, however, we need to keep in mind that the x86-64 system architecture uses the little-endian format. This means, for example, that the value `0xaf959287` will become `0x879295af`. Reading the content of the `Java_com_hackthebox_myapp_MainActivity_getAPIKey()` function reveals the following hexadecimal values.

```
local_10 = *(long *)(in_FS_OFFSET + 0x28);
local_18 = (undefined4 *)operator.new(0x30);
local_28 = 0x31;
uStack_24 = 0;
uStack_20 = 0x20;
uStack_1c = 0;
local_18[4] = 0xb9998c96;
local_18[5] = 0xa9b3a7ce;
local_18[6] = 0xcc8e87ac;
local_18[7] = 0x94b7c891;
*local_18 = 0xaf959287;
local_18[1] = 0x93969a9c;
local_18[2] = 0x9acabacf;
local_18[3] = 0xaec99194;
```

Let's take the above values and place them in sequental order. Since the arrays start from `0` and not `1`, our frst value will be `*local_18 = 0xaf959287`, followed by `local_18[1] = 0x93969a9c`, etc. We will also apply big-endian formatting to reverse the existing little-endian formatting. Finally, we will remove the `x0` prefix to produce the following hex sequence.

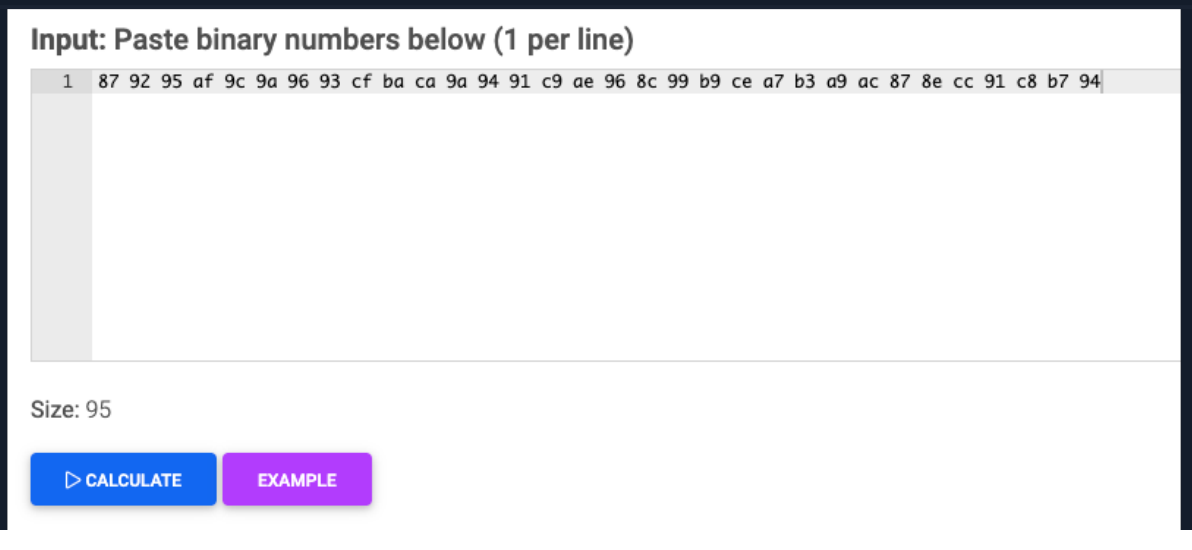
Reversing Shared Objects

```
879295af 9c9a9693 cfbaca9a 9491c9ae 968c99b9 cea7b3a9 ac878ecc 91c8b794
```

Now, we can use the online calculator—or any other calculator with bitwise NOT operation—to get the initial string value of the API key. Before pasting the above sequence to the calculator, make sure it's in the following format.

Reversing Shared Objects

```
87 92 95 af 9c 9a 96 93 cf ba ca 9a 94 91 c9 ae 96 8c 99 b9 ce a7 b3 a9 ac 87 8e cc 91 c8 b7 94
```



Then, press the blue `Calculate` button and check the `Ascii Result` section.

Ascii Result:

```
x
m
j
P
c
e
i
l
0
E
5
e
k
n
6
Q
i
s
f
F
1
X
L
V
S
x
q
3
n
7
H
k
```

The result of NOT operation in Ascii

As we can see, the API key `xmjPceil0E5ekn6QisfF1XLVSxq3n7Hk` is now visible. Fortunately for us, it only took a few steps to retrieve it successfully. However, other more complicated obfuscation methods can be used to make the process of reading the source code even more difficult. We will showcase examples of these in later in the module.

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 Analyze the APK file found inside the attached "myapp_shared_libs_1.zip" archive. What is the API key value?

Submit your answer here...

+10 Streak pts | Submit | myapp_shared_libs_1.zip

+ 5 Analyze the APK file found inside the attached "myapp_shared_libs_2.zip" archive. What is the API key value?

Submit your answer here...

+10 Streak pts | Submit | myapp_shared_libs_2.zip

Previous | Next

Cheat Sheet

Go to Questions

Table of Contents

License Verification Bypass

Root Detection Bypass

Skills Assessment

Skills Assessment

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left