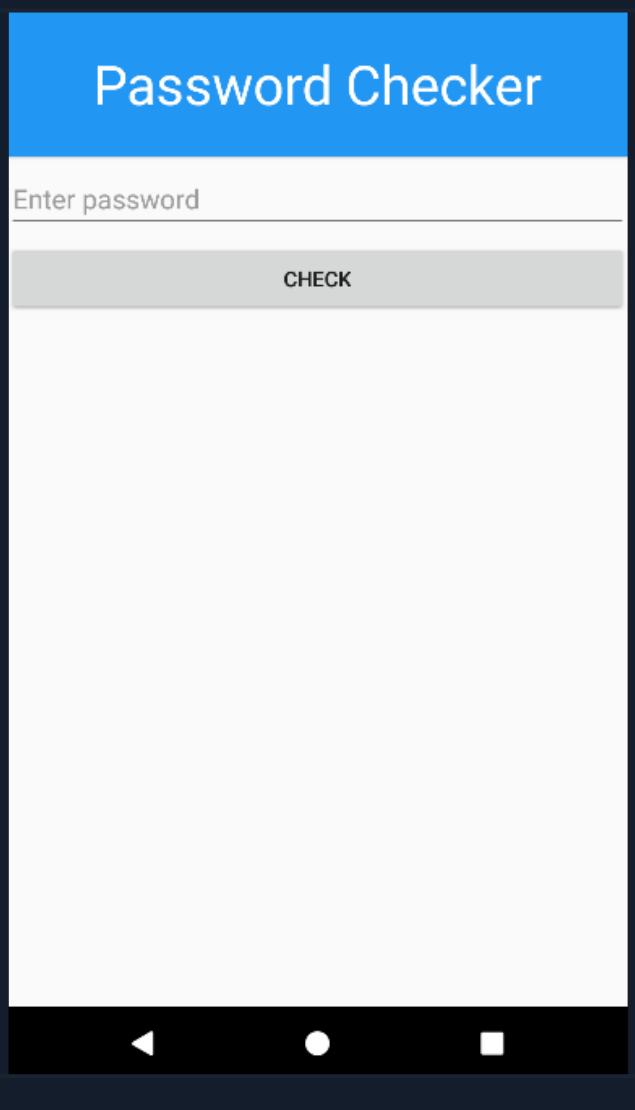
Reversing DLL Files

Unlike applications built with Android Studio—which typically use Java or Kotlin—Xamarin enables developers to create apps using the .NET framework and C#. In Xamarin projects, C# code is first compiled into Intermediate Language (IL) via the .NET compiler, and then translated into platform-specific code. Much like native C++ code is packaged into shared libraries (`.so` files) within the final APK, Xamarin-based applications bundle their compiled code into Dynamically Linked Libraries (`.dll` files), which are stored collectively in a file named assemblies.blob.

Some developers may assume that placing code in DLLs provides a degree of obfuscation. However, just as with the shared objects analyzed earlier, these libraries contain native code, making them unreadable by tools like JADX or APKTool. In the following paragraphs, we will explore how to extract and reverse-engineer a DLL file in order to inspect its source code.

Locating Assemblies Inside APKs

The application shown below includes a feature that confirms whether or not the password you've set is correct.



Now, let's use APKTool to decode and extract the files from the APK.

Reversing DLL Files

```
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...
```

Listing the content of the directory `myapp/smali` reveals the following.

Reversing DLL Files

```
rl1k@htb[/htb]$ ls -l myapp/smali

total 0
drwxr-xr-x    5 bertolis  bertolis   160 Oct 26 15:36 android
drwxr-xr-x   39 bertolis  bertolis  1248 Oct 26 15:36 androidx
drwxr-xr-x    5 bertolis  bertolis   160 Oct 26 15:36 com
drwxr-xr-x    3 bertolis  bertolis    96 Oct 26 15:36 crc640ba5056094cb42e6
drwxr-xr-x    4 bertolis  bertolis   128 Oct 26 15:36 crc6414252951f3f66c67
drwxr-xr-x  193 bertolis  bertolis  6176 Oct 26 15:36 crc643f46942d9dd1fff9
drwxr-xr-x   20 bertolis  bertolis   640 Oct 26 15:36 crc64720bb2db43a66fe9
drwxr-xr-x    3 bertolis  bertolis    96 Oct 26 15:36 crc64a0e0a82d0db9a07d
drwxr-xr-x    6 bertolis  bertolis   192 Oct 26 15:36 crc64ee486da937c010f4
drwxr-xr-x   10 bertolis  bertolis   320 Oct 26 15:36 mono
drwxr-xr-x    3 bertolis  bertolis    96 Oct 26 15:36 xamarin
```

The directory `xamarin` implies that the application is created using Xamarin. We know that applications built with Xamarin store their code within dynamically linked libraries (DLL files). Let's enumerate the extracted directory `myapp/unknown/assemblies/`.

Reversing DLL Files

```
rl1k@htb[/htb]$ ls -l myapp/unknown/assemblies/

total 34128
-rw-r--r--  1 bertolis  bertolis    15872 Oct 29 21:04 FormsViewGroup.dll
-rw-r--r--  1 bertolis  bertolis    17284 Oct 29 21:04 FormsViewGroup.pdb
-rw-r--r--  1 bertolis  bertolis   166912 Oct 29 21:04 Java.Interop.dll
-rw-r--r--  1 bertolis  bertolis    63144 Oct 29 21:04 Java.Interop.pdb
-rw-r--r--  1 bertolis  bertolis  2183168 Oct 29 21:04 Mono.Android.dll
-rw-r--r--  1 bertolis  bertolis   639024 Oct 29 21:04 Mono.Android.pdb
-rw-r--r--  1 bertolis  bertolis   121856 Oct 29 21:04 Mono.Security.dll
-rw-r--r--  1 bertolis  bertolis    53472 Oct 29 21:04 Mono.Security.pdb
-rw-r--r--  1 bertolis  bertolis   442368 Oct 29 21:04 Myapp.Android.dll
-rw-r--r--  1 bertolis  bertolis    79216 Oct 29 21:04 Myapp.Android.pdb
-rw-r--r--  1 bertolis  bertolis     9728 Oct 29 21:04 Myapp.dll
<SNIP>
```

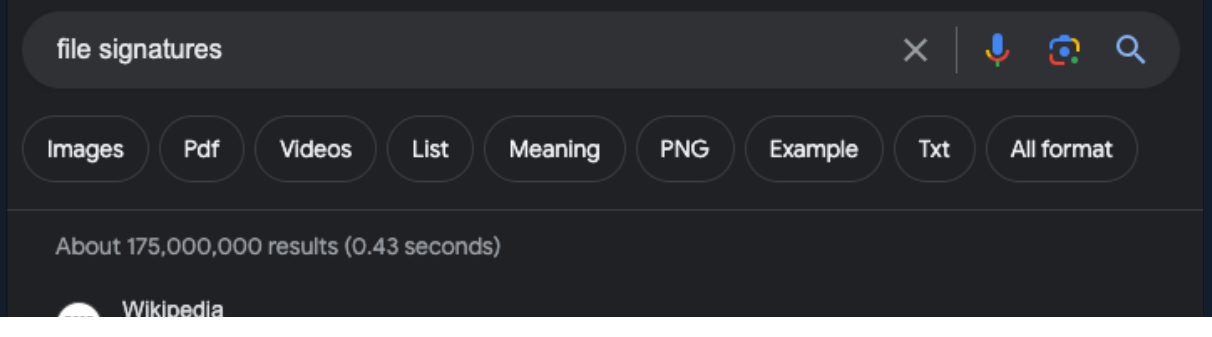
This directory contains the .NET assemblies (DLL files) used by the application. Since the app's name is `myapp`, we could try reading the content of the file `Myapp.dll`, but unfortunately it contains compiled that isn't human-readable. However, the headers of the file `Myapp.dll` reveal the magic number `MZ`.

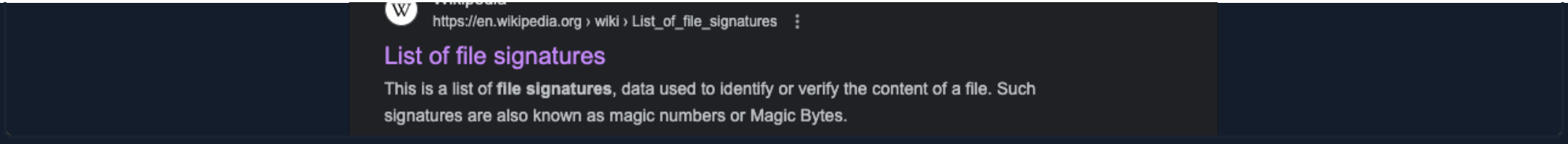
Reversing DLL Files

```
rl1k@htb[/htb]$ vim myapp/unknown/assemblies/Myapp.dll
```

```
MZ<90>^@^C^@^@^@^D^@^@^@ÿÿ^@^@¸^@^@^@^@^@^@^@@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
    Í!¸^ALÍ!This program cannot be run in DOS mode.^M^M
$^@^@^@^@^@^@^@PE^@^@L^A^C^@SD<8c>È^@^@^@^@^@^@^@à^@^B!^K^A0^@^@^^^@^@^@F
@^@^@^@D^@^@^@^@^@^@^@^@<80>^@^@^@B^@^@^@^@^@^@C^@`<85>^@^@P^@^@P^@^@^@
^@ô^B^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@`^@^@^L^@^@^@t<^@^@8^@^@^@^@^@^@
@^@^@^@^@^@ ^@^@^H^@^@^@^@^@^@^@^@^@^@^H ^@^@H^@^@^@^@^@^@^@^@^@^@.text^@
```

Checking online for `file signatures` reveals the following as the first result.





Using the `control+F`, we can search on the website for the `MZ` magic numbers, which eventually brings us to the following cell in the table.

`4D 5A`	`MZ`	0	exe dll mui sys scr cpl ocx ax iec ime rs tsp fon efi	DOS MZ executable and its descendants (including NE and PE)

In the `Description` cell, we can see that this is a DOS MZ executable. This is the executable file format used for .EXE files in DOS. The output of the following command also indicates that this is an executable file for MS Windows.

Reversing DLL Files

```
rl1k@htb[/htb]$ file myapp/unknown/assemblies/Myapp.dll

myapp/unknown/assemblies/Myapp.dll: PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly, for MS Windows
```

Furthermore, our command indicates that it's a .NET assembly, which means we should be able to produce human-readable code using tools like dnSpy or ILSpy. These are .NET decompilers/assembly browsers used to inspect and analyze .NET assemblies (DLL and EXE files). Before we begin our analysis with these tools, let's first examine two other cases of retrieving.NET assemblies from APK files.

Extracting Assemblies from Compressed DLLs

Assume we are testing a similar application and have already extracted the files of the APK using APKTool. Reading the content of the directory `myapp/unknown/assemblies/` reveals the file `Myapp.dll`, but this time, the `file` command produces the following output.

Reversing DLL Files

```
rl1k@htb[/htb]$ file myapp/unknown/assemblies/Myapp.dll

myapp/unknown/assemblies/Myapp.dll: Sony PlayStation Audio
```

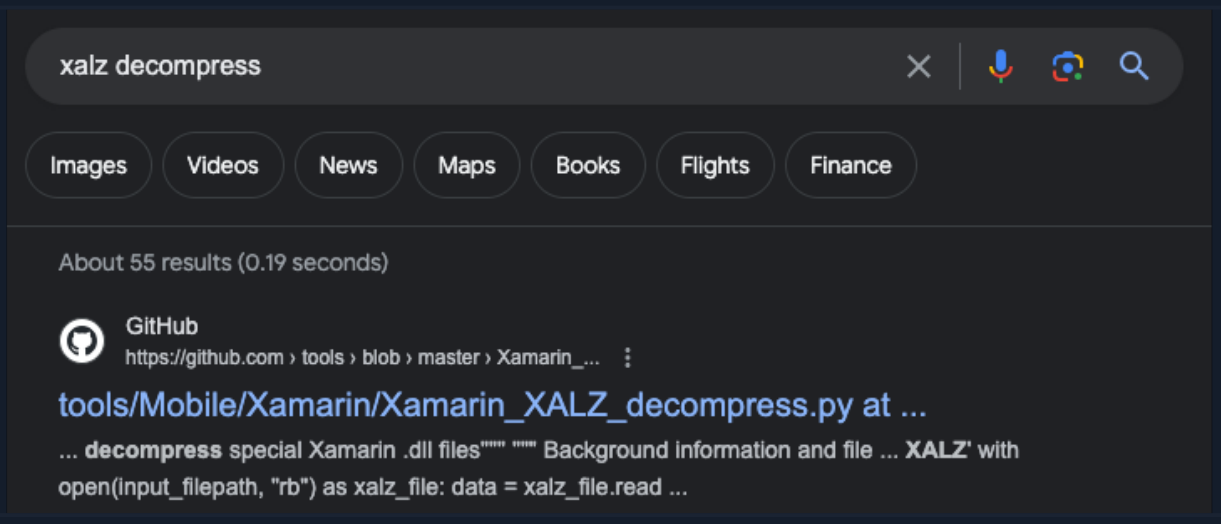
Again, we will check the headers of the file.

Reversing DLL Files

```
rl1k@htb[/htb]$ vim myapp/unknown/assemblies/Myapp.dll
```

```
XALZ^E^@^@^@^@(^@^@ò^CMZ<90>^@^C^@^@^@^D^@^@^@ÿÿ^@^@¸^@^A^@/@^@^A^@^Oó.<80>^@^@^@^N^_°^N^@´
    Í!¸^ALÍ!This program cannot be run in DOS mode.^M^M
$D^@ÄPE^@^@L^A^C^@<82>í<8f>ÖX^@âà^@^B!^K^A0^@^@ ^@^@^@^F^T^@2^N>^@^P^@^D<9e>^@^@^L^@^R^B¸^@
^S^@À^@^E<95>^@^R^B^K^@q^C^@`<85>^@^@^P^C^@^F^H^@^E^G^@^AÄ^@S¸=^@^@SS^@/X^Cå^@^@ß`^@^@^L^@^
@^@Ü<^@^@8^@^A^@^XG ^@^@^H^O^@W^H ^@^@H^P^@£.text^@^@^@^T^^Ø^@^CÐ^@^KC^@s`.rsrc^@<9c>^@^S@é
^@^Z"(^@¢@^@^@@.reloc¬^@^@´^@^AL^@^]&(^@^\B§^@"ð=^H^@^QH3^@Ï^E^@x&^@^@d^V^@^@^A^@^A^@#ñ^Sb^
```

We notice that this time the headers start with the magic number `XALZ`. After reading the following Pull request from Xamarin's GitHub repository, we can conclude that the `XALZ` magic bytes indicate the use of the LZ4 compression algorithm. Searching online for `xalz file header` reveals the following as the first result.



According to the description in the comments of the file `Xamarin_XALZ_decompress.py`, this tool decompresses special Xamarin DLL files. Let's download it to our local machine and run it.

Reversing DLL Files

```
rl1k@htb[/htb]$ wget https://raw.githubusercontent.com/x41sec/tools/master/Mobile/Xamarin/Xamarin_XALZ_decompress.py
rl1k@htb[/htb]$ python Xamarin_XALZ_decompress.py myapp/unknown/assemblies/Myapp.dll myapp_decompressed.dll
```

Running the script on the compressed `Myapp.dll` file will output the new file `myapp_decompressed.dll`. Subseqeuntly, the file type now shows that it's a .NET assembly.

Reversing DLL Files

```
rl1k@htb[/htb]$ file myapp_decompressed.dll

myapp_decompressed.dll: PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly, for MS Windows
```

At this point, we could begin analysis using a .NET decompiler like ILSpy or dnSpy. Now, let's examine the third case of retrieving .NET assemblies from an APK file.

Extracting Assemblies from .blob Files

Assuming once again that we are testing a similar app, and we have already used APKTool to extract the files from the APK. This time, listing the content of the directory `myapp/unknown/assemblies/` outputs the following files.

Reversing DLL Files

```
rl1k@htb[/htb]$ ls -l myapp/unknown/assemblies/

-rw-r--r-- 1 bertolis bertolis 5118997 Oct 30 11:00 assemblies.blob
-rw-r--r-- 1 bertolis bertolis    3067 Oct 30 11:00 assemblies.manifest
```

In this example, the DLL files are bundled in a single file called `assemblies.blob`. In the latest version of Xamarin, compressing and bundling the DLL files into an `assemblies.blob` file is the default option for optimizing the app's performance, size, and startup time. Checking the file type indicates that this is not an executable file.

Reversing DLL Files

```
rl1k@htb[/htb]$ file myapp/unknown/assemblies/assemblies.blob

myapp/unknown/assemblies/assemblies.blob: Sony PlayStation Audio
```

Opening `assemblies.blob` with ILSpy or dnSpy will result in an error, since the expected format is an uncompressed DLL file containing IL (Intermediate Language) code. Searching online for `"extract dll from assemblies.blob"`, we find this article. It suggests tool pyxamstore for extracting and decompressing DLLs from the `assemblies.blob` file. Let's clone the repository and try it out.

Reversing DLL Files

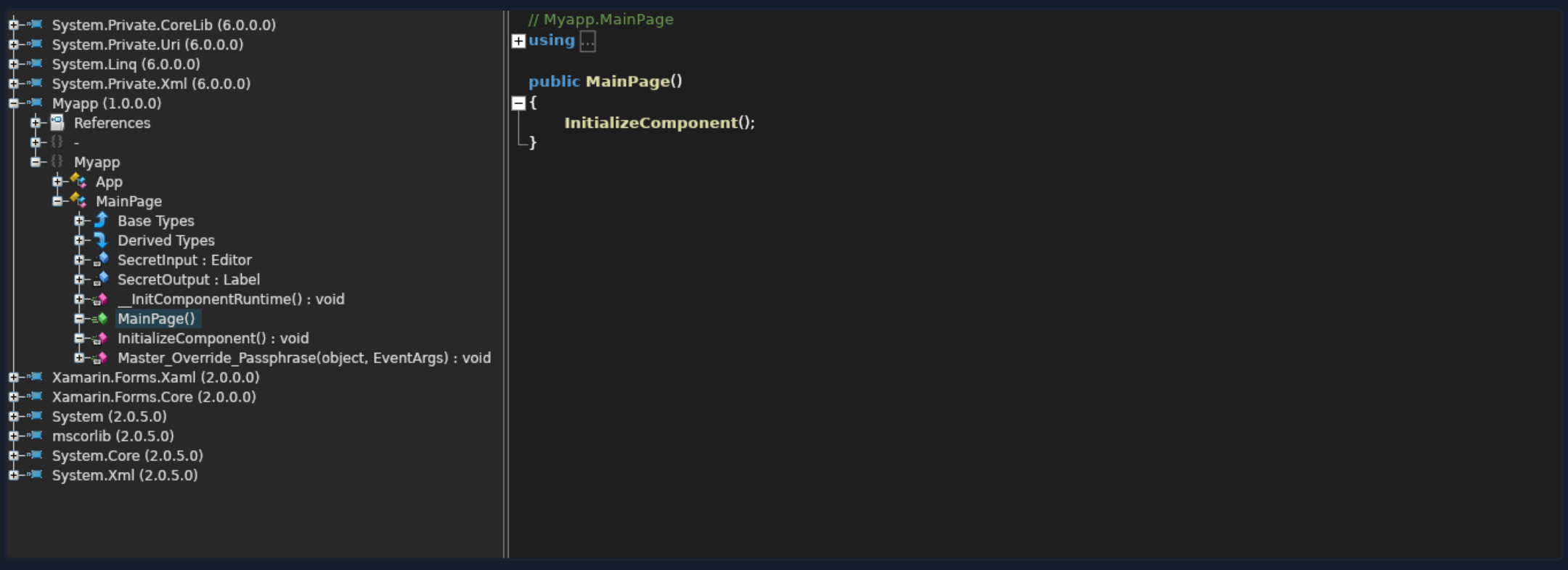
```
rl1k@htb[/htb]$ sudo pip3 install git+https://github.com/jakev/pyxamstore.git
rl1k@htb[/htb]$ pyxamstore unpack -d myapp/unknown/assemblies/

Extracting Myapp.Android...
Extracting FormsViewGroup...
Extracting Myapp...
Extracting Xamarin.AndroidX.Activity...
Extracting Xamarin.AndroidX.AppCompat.AppCompatResources...
Extracting Xamarin.AndroidX.AppCompat...
Extracting Xamarin.AndroidX.CardView...
<SNIP>
```

After running the tool, a new directory named `out` is created containing the decompressed DLL file. We can now open the file `out/Myapp.dll` using ILSpy (since dnSpy can only be installed in Windows operating systems.)

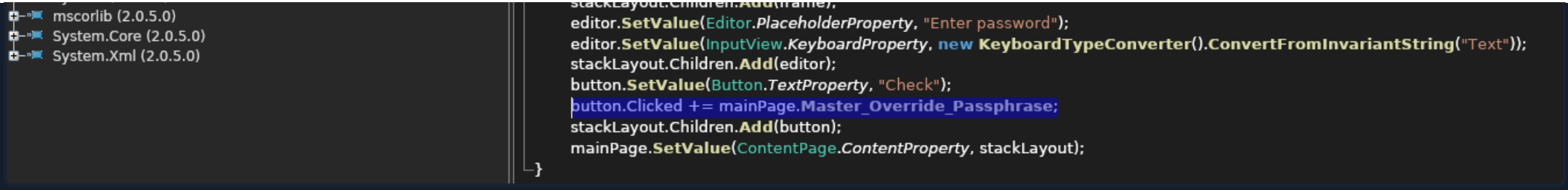
Reversing DLL Files

```
rl1k@htb[/htb]$ wget https://github.com/icsharpcode/AvaloniaILSpy/releases/download/v7.2-rc/Linux.x64.Release.zip
rl1k@htb[/htb]$ unzip Linux.x64.Release.zip
rl1k@htb[/htb]$ unzip ILSpy-linux-x64-Release.zip
rl1k@htb[/htb]$ cd artifacts/linux-x64
rl1k@htb[/htb]$ ./ILSpy
```

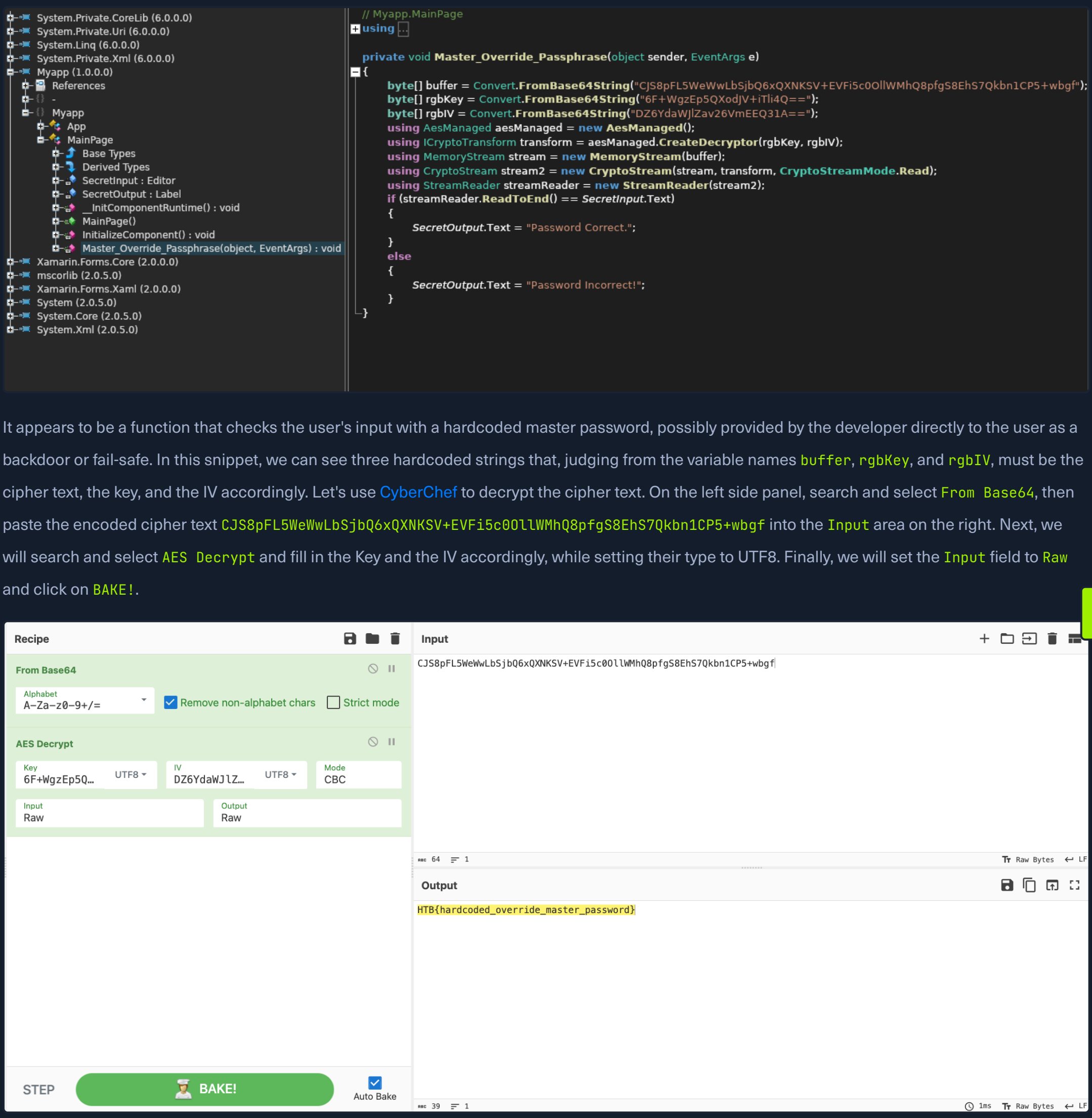
When the program starts up, click on `File` -> `Open` and select the `out/Myapp.dll`. It will load after a few moments, and we can then navigate through the project. The following image shows the source code of the function `MainPage()`, which is likely the constructor of the class `MainPage`.



This function calls the `InitializeComponent()`. Let's click on it to see its source code.



At the end of this function, we can see the line `button.Clicked += mainPage.Master_Override_Passphrase;`. Let's click on this as well.



It appears to be a function that checks the user's input with a hardcoded master password, possibly provided by the developer directly to the user as a backdoor or fail-safe. In this snippet, we can see three hardcoded strings that, judging from the variable names `buffer`, `rgbKey`, and `rgbIV`, must be the cipher text, the key, and the IV accordingly. Let's use CyberChef to decrypt the cipher text. On the left side panel, search and select `From Base64`, then paste the encoded cipher text `CJS8pFL5WeWwLbSjbQ6xQXNKSV+EVFi5c0OllWMhQ8pfgS8EhS7Qkbn1CP5+wbgf` into the `Input` area on the right. Next, we will search and select `AES Decrypt` and fill in the Key and the IV accordingly, while setting their type to UTF8. Finally, we will set the `Input` field to `Raw` and click on `BAKE!`.

The hardcoded password `HTB{hardcoded_override_master_password}` has been successfully decrypted.

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 the hardcoded password?

Submit your answer here...

+10 Streak pts | Submit | myapp_dll.zip | Hint

Previous | Next

Cheat Sheet

Go to Questions

Table of Contents

- 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