FIREEYE™

FLARE

# Flare-On 7: Challenge 5 – TKApp.tpk

**Challenge Author: Moritz Raabe (@m_r_tz)**

## Introduction

This challenge targets the Tizen operating system that runs on millions of Samsung devices, including TVs and wearables. Luckily, we can resort to familiar reverse engineering tools for our analysis. FLARE VM contains all of them including the most important tool dnSpy.

You can run (and solve) the challenge via the emulator included with Tizen Studio. While this write-up focuses on static analysis it contains screenshots of the running app. As we will see the app helps someone crazy about big cats organize their day.

## Basic Analysis

Inspecting `TKApp.tpk` with a hex editor or the `file` utility reveals that we are dealing with a Zip archive. Unzipping the archive reveals a file and directory structure of an app – similar to an unpacked Android Package Kit (APK) file. Various image files hint at the theme of this challenge: tigers. In the `bin` directory we notice multiple DLLs. These suggest the use of Xamarin.Forms and the Tizen Wearable Circular UI.

In `tizen-manifest.xml` we notice a reference to `TKApp.dll`. Judging from all files' timestamps, this DLL appears to be the most interesting to us.

We open `TKApp.dll` in a PE viewer such as CFF Explorer and quickly notice that this is a .NET DLL. Before throwing the file into dnSpy for advanced static analysis, a look at the file's strings can provide some additional insights. Figure 1 shows the application running in the Tizen Studio emulator.


**Figure 1: Start screen of `TKApp` running in the Tizen Studio emulator**

While it's helpful to have some understanding of Xamarin.Forms and Tizen.Net it's not really required here. If you want to learn more about these technologies check out https://dotnet.microsoft.com/apps/xamarin/xamarin-forms and https://docs.tizen.org/application/dotnet/index.

# Advanced Analysis of TKApp.dll

Figure 2 shows dnSpy's Assembly Explorer view of the DLL. The app contains multiple resources and classes. They don't appear to be obfuscated.

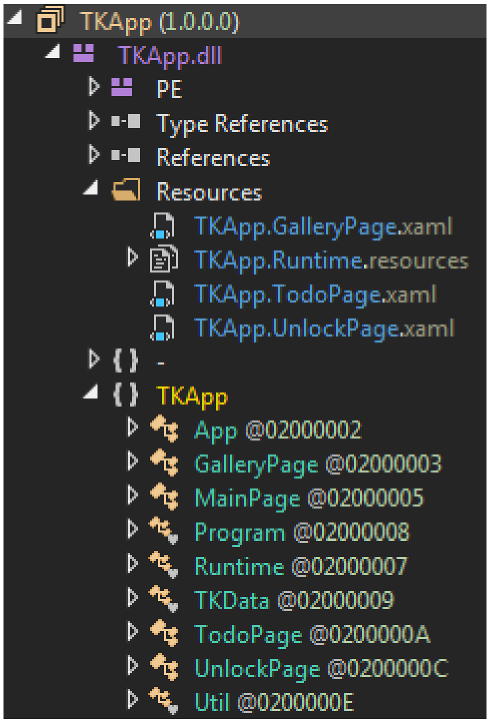


**Figure 2: `TKApp.dll` overview in dnSpy's Assembly Explorer**

A right click in the Assembly Explorer and then selecting "Go to Entry Point" brings us to the file's entry point in the `Program` class. Here we see that the main implementation starts in the `App` class.

Depending on the `App.IsLoggedIn` state, one of the two pages `UnlockPage` or `MainPage` is run. By default, `IsLoggedIn` is `false` and the application displays the `UnlockPage`.

With Xamarin.Forms user interfaces can be defined with an XML-based language called eXtensible Application Markup Language (XAML). The DLL's resources contain three such XAML files.

## UNLOCKPAGE

Per its XAML file (`Resources/TKApp.UnlockPage.xaml`) the `UnlockPage` contains a password entry field and a button (shown in Figure 1). Per the class' implementation the app loads the `MainPage` if the provided password is correct. To verify this, the program compares the user input to the decoded `TKData.Password` field. The decode function `Util.Decode` shown in Figure 3 is a simple one-byte XOR function using the decimal key 83. The decoded password is `mullethat`.

```
public static string Decode(byte[] e)
{
    string text = "";
    foreach (byte b in e)
    {
        text += Convert.ToChar((int)(b ^ 83)).ToString();
    }
    return text;
}
```

Figure 3: `Util.Decode` function used for the password verification

## MAINPAGE


Figure 4: `MainPage` running in the emulator

In `MainPage` the `GetImage` function stands out. Figure 5 shows that the function generates a SHA256 hash value and decodes Base64 data based on data obtained from the `Util.GetString` function.

```
string text = new string(new char[]
{
    App.Desc[2],
    App.Password[6],
    App.Password[4],
    App.Note[4],
    App.Note[0],
    App.Note[17],
    App.Note[18],
    App.Note[16],
    App.Note[11],
    App.Note[13],
    App.Note[12],
    App.Note[15],
    App.Step[4],
    App.Password[6],
    App.Desc[1],
    App.Password[2],
    App.Password[2],
    App.Password[4],
    App.Note[18],
    App.Step[2],
    App.Password[4],
    App.Note[5],
    App.Note[4],
    App.Desc[0],
    App.Desc[3],
    App.Note[15],
    App.Note[8],
    App.Desc[4],
    App.Desc[3],
    App.Note[4],
    App.Step[2],
    App.Note[13],
    App.Note[18],
    App.Note[18],
    App.Note[8],
    App.Note[4],
    App.Password[0],
    App.Password[7],
    App.Note[0],
    App.Password[4],
    App.Note[11],
    App.Password[6],
    App.Password[4],
    App.Desc[4],
    App.Desc[3]
});
byte[] key = SHA256.Create().ComputeHash(Encoding.ASCII.GetBytes(text));
byte[] bytes = Encoding.ASCII.GetBytes("NoSaltOfTheEarth");
try
{
    App.ImgData = Convert.FromBase64String(Util.GetString(Runtime.Runtime_dll, key, bytes));
    return true;
}
```

**Figure 5: Key functionality of `MainPage.GetImage` function**

The `GetString` function decrypts data using the RijndaelManaged class (AES). The arguments passed to this function are:

- `cipherText`: `Runtime.Runtime_dll` data which the `ResourceManager` obtains from the binary resource named `Runtime.dll`
- `Key`: SHA256 hash value of bytes of `text` variable
- `IV` (initialization vector): bytes of string `NoSaltOfTheEarth`

We use dnSpy to save the resource via right click, “Raw Save…” as shown in Figure 6.

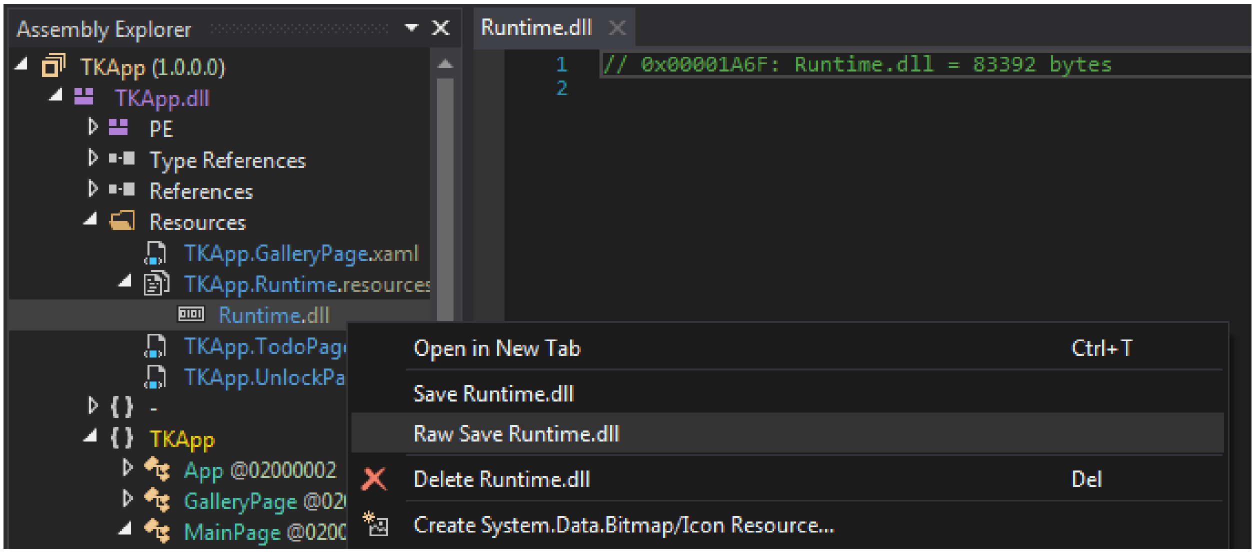


Figure 6: Saving the resource via right click, “Raw Save Runtime.dll”

To decrypt the saved resource data, we only miss the key derived from the fields `App.Desc`, `App.Password`, `App.Note`, and `App.Step`.

We use dnSpy’s cross references feature via right click, “Analyze” on each field to inspect where the respective value is set. The Analyzer view for the `App.Desc` field is shown in Figure 7.

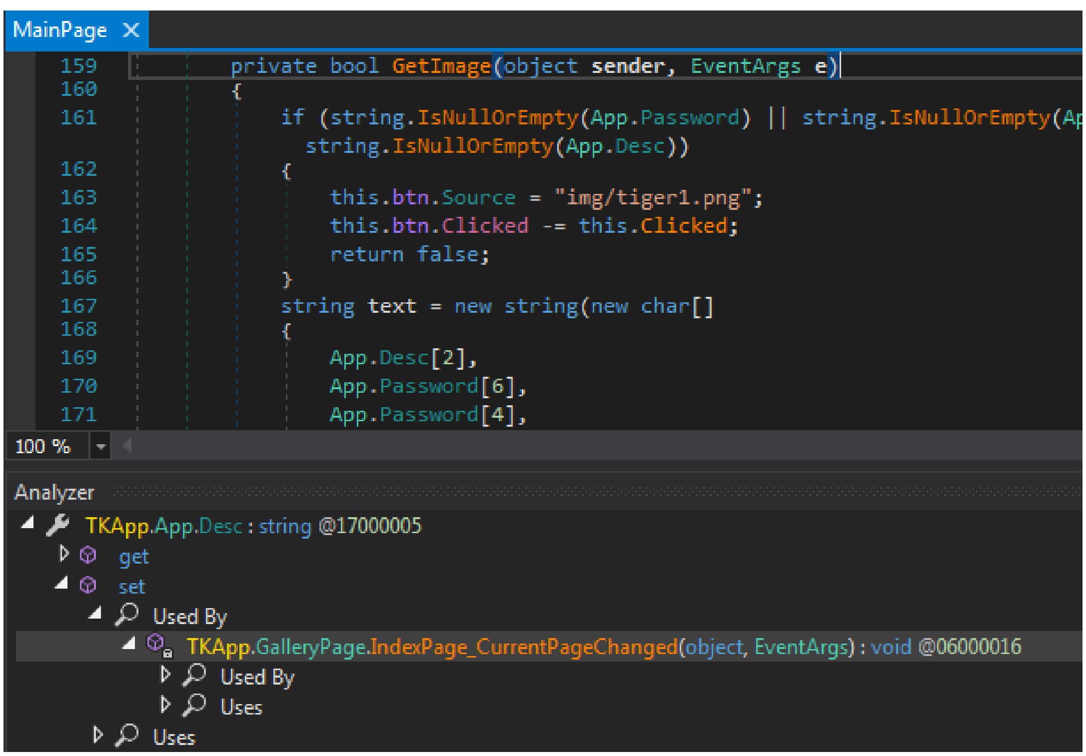


Figure 7: Usage of field `App.Desc` in Analyzer view (bottom)

## APP.DESC

In the `GalleryPage` the Desc field is set from EXIF data of the file `res\gallery\05.jpg`. Inspecting the file using an EXIF tool or using the Windows Properties view reveals the image description value `water`.

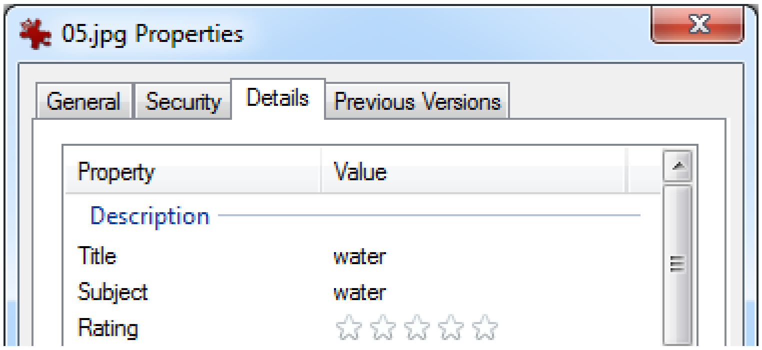


**Figure 8: Image description value in EXIF data of 05.jpg**

## APP.PASSWORD

When analyzing the `UnlockPage` we've already identified that the `Password` value is `mullethat`.

## APP.NOTE

The app sets the `Note` field in the function `TodoPage.SetupList` shown in Figure 9. The value is the `Note` property of the first unfinished Todo object. The selected Todo depends on the Boolean variable `isHome`.

```
private void SetupList()
{
    List<TodoPage.Todo> list = new List<TodoPage.Todo>();
    if (!this.isHome)
    {
        list.Add(new TodoPage.Todo("go home", "and enable GPS", false));
    }
    else
    {
        TodoPage.Todo[] collection = new TodoPage.Todo[]
        {
            new TodoPage.Todo("hang out in tiger cage", "and survive", true),
            new TodoPage.Todo("unload Walmart truck", "keep steaks for dinner", false),
            new TodoPage.Todo("yell at staff", "maybe fire someone", false),
            new TodoPage.Todo("say no to drugs", "unless it's a drinking day", false),
            new TodoPage.Todo("listen to some tunes", "https://youtu.be/kTmZnQOfAF8", true)
        };
        list.AddRange(collection);
    }
    List<TodoPage.Todo> list2 = new List<TodoPage.Todo>();
    foreach (TodoPage.Todo todo in list)
    {
        if (!todo.Done)
        {
            list2.Add(todo);
        }
    }
    this.mylist.ItemsSource = list2;
    App.Note = list2[0].Note;
}
```

**Figure 9: Decompiled `TodoPage.SetupList` function**

The app sets `isHome` to true only if a specific condition is satisfied. So, we assume that the multiple items from the `else` branch are what is expected here. Hence, `keep steaks for dinner` is the value we need. If the decryption using this value fails, we can also try the alternative value `and enable GPS` since these are the only two options. Challenge authors must be careful to not allow for too many shortcuts like this 😉. For extra credit figure out how the `isHome` check works and where “home” is.

Running the app with the correct settings results in the `TodoPage` screen shown in Figure 10.


**Figure 10: `TodoPage` running with correct configuration**

## APP.STEP

The app sets the `Step` field from the application metadata referenced by the key `its`. The value is specified in the `tizen-manifest.xml` file. We noticed this from carefully inspecting the file in the beginning or by grepping the entire unzipped directory for the key name. The metadata value is `magic`.

Combining the individual characters from the recovered fields results in the following string:

```
the kind of challenges we are gonna make here
```

The AES decryption key is the SHA256 hash of this string:

```
248e9d7323a1a3c5d5b3283dcb2b40211a14415b6dcd2a86181721fd74b4befd
```

AES decrypting the data with this key results in a Base64 encoded string. The Base64 decoded string contains a JFIF marker which indicates JPEG image data. Figure 11 shows how to decrypt, Base64 decode and render the data from `Runtime.dll` using CyberChef. The exported CyberChef recipe in Chef format is shown below.

```
AES_Decrypt({'option':'Hex','string':'248e9d7323a1a3c5d5b3283dcb2b40211a14415
b6dcd2a86181721fd74b4befd'},{'option':'UTF8','string':'NoSaltOfTheEarth'},'CB
C','Raw','Raw',{'option':'Hex','string':''})
From_Base64('A-Za-z0-9+/=',true)
Render_Image('Raw')
```

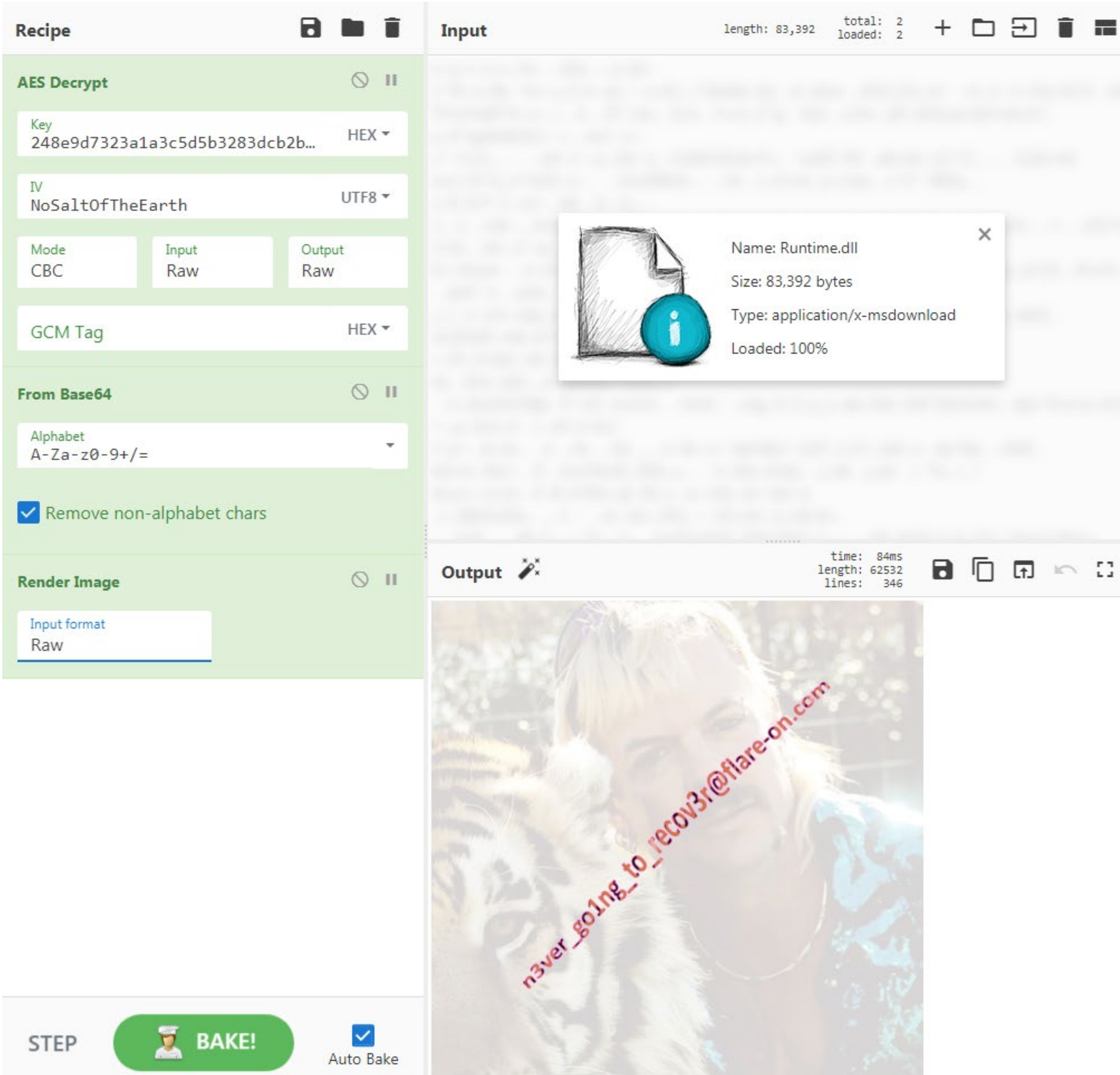


**Figure 11: AES decrypting, Base64 decoding and rendering the data from `Runtime.dll`**

# Challenge Flag

As shown in the CyberChef output pane the flag for this challenge is:

`n3ver_go1ng_to_recov3r@flare-on.com`

Figure 12 shows how the app displays the decrypted and Base64 decoded image after successfully following all required steps.


**Figure 12: `GalleryPage` displaying the decoded image data with the challenge flag**