hacker_Hacker101 CTF：Android挑战文章

翻译

weixin_26722031 于 2020-08-01 15:39:36 发布 559 收藏
文章标签： android python
原文链接：https://medium.com/bugbountywriteup/hacker101-ctf-android-challenge-writeups-f830a382c3ce

hacker

In this article, I will be demonstrating how to solve the *Hacker101 CTF* (Capture The Flag) challenges for the Android category. Hacker101 is a free educational site for hackers, run by HackerOne.

在本文中，我将演示如何解决Android类别的*Hacker101 CTF* (捕获标志)问题。 Hacker101是由HackerOne经营的免费的黑客教育网站。

免责声明 (Disclaimer)

I was motivated to make this article out of a desire to learn more about Android mobile application security. This writeup will obviously contain spoilers and I encourage readers to attempt this CTF before reading this article. Try to solve as many of these challenges as you can and then come back later to read this article if you get stuck or want to see a potentially different approach to solving a challenge. Without any further delay, lets jump in □!

出于使我对学习更多有关Android移动应用程序安全性的渴望而感到兴奋。 该文章显然将包含破坏者，我鼓励读者在阅读本文之前尝试使用CTF。 尝试解决所有这些挑战，如果遇到困难或想了解解决挑战的潜在方法，请稍后再阅读本文。 没有任何进一步的延迟，让我们跳入□！

H1温控器(简易，2个标志) (H1 Thermostat (Easy, 2 Flags))

I started this challenge by downloading the application APK file and installing it on my emulator device using **Android Debug Bridge** (**ADB**)

我通过下载应用程序APK文件并使用**Android Debug Bridge** (**ADB**)将其安装在模拟器设备上而开始了这项挑战

```
adb install thermostat.apk
```

Opening the application showed that it only had a single activity with a thermostat and a gauge, allowing the user to raise or lower the temperature setting.

打开该应用程序表明，它只有一个带有恒温器和仪表的活动，允许用户升高或降低温度设置。

Image for post

Next, I generated a static analysis report for the APK file using the **Mobile Security Framework** (**MobSF**) tool. I started reviewing the report by examining the *AndroidManifest.xml* file.

接下来，我使用**移动安全框架** (**MobSF**)工具为APK文件生成了静态分析报告。 我通过检查*AndroidManifest.xml*文件开始查看报告。

Image for post

Looking through the manifest file, I can see that the application has requested only the **android.permission.INTERNET** permission, which allows the application to create network sockets. The developer has set the attributes **android:usesCleartextTraffic** and **android:allowBackup** to **true** which means the application intends to use cleartext traffic and can have it's contents backed up by the user.

查看清单文件，可以看到该应用程序仅请求了**android.permission.INTERNET**权限，该权限允许该应用程序创建网络套接字。 开发人员已将**android：usesCleartextTraffic**和**android：allowBackup**属性设置为**true**，这意味着应用程序打算使用明文流量，并且可以将其内容由用户备份。

The application also appears to only have two components. A **activity** called *com.hacker101.level11.ThermosActivity* has been declared with an intent filter. A **content provider** called *ProcessLifecycleOwnerInitializer* has also been declared but it is not exported. I decided to examine the java source code for the *com.hacker101.level11.ThermosActivity.*

该应用程序似乎也只有两个组件。 已经使用意图过滤器声明了一个名为*com.hacker101.level11.ThermosActivity*的**活动**。 还声明了一个名为*ProcessLifecycleOwnerInitializer*的**内容提供程序**，但未导出。 我决定检查*com.hacker101.level11.ThermosActivity*的Java源代码。

Looking through the source code of the activity, I noted that a network request was being made. A class called *PayloadRequest* was used inside this network request as seen highlighted in green. Examining the source code for the *PayloadRequest* class, I discover both flags for the challenge.

通过查看活动的源代码，我注意到正在发出网络请求。 如绿色突出显示的那样，在此网络请求中使用了一个称为*PayloadRequest*的类。 检查*PayloadRequest*类的源代码，我发现了挑战的两个标志。

It appears that one flag is hashed using **MD5** and then **base64** encoded before being added as a value to a header called *X-MAC*. The other flag is simply added as a plaintext value to the header *X-Flag*. I can use the tool **BurpSuite** to intercept this network request when interacting with the application.

似乎是使用**MD5**对一个标志进行了哈希处理，然后对其进行了**base64**编码，然后再将其作为值添加到名为*X-MAC*的标头中。 另一个标志只是作为纯文本值添加到标头*X-Flag*。 与应用程序进行交互时，我可以使用工具**BurpSuite**拦截此网络请求。

As seen in the image above, the intercepted network request shows the *X-MAC* and *X-Flag* headers with their equivalent values. Nice and easy for the first challenge □!

如上图所示，截获的网络请求显示*X-MAC*和*X-Flag*标头及其等效值。 轻松应对第一个挑战□！

有意锻炼(中等，1个标志) (Intentional Exercise (Moderate, 1 Flag))

Following a similar approach as seen in the previous challenge, I installed the application using ADB and generated a static analysis report of the APK file using MobSF. Opening the application showed an activity with a welcome message and a link called **"Flag"**.

按照上一个挑战中所示的类似方法，我使用ADB安装了该应用程序，并使用MobSF生成了APK文件的静态分析报告。 打开应用程序显示了一个带有欢迎消息和名为**" Flag"**的链接的活动。

Clicking on the link leads to a **"Invalid request"** error. Guess it won't be that easy □.

单击链接将导致**"无效请求"**错误。 猜猜那不是那么容易□。

As seen with the previous challenge, I started by looking at the manifest file for the application. There is only one **activity** declared in the manifest file. What standouts about the activity, is that it has been declared with three **intent filters**. The two intent filters surrounded by green in the image below shows how you create a **deep links** for your app's content (**see references**).

从上一个挑战中可以看出，我首先查看了应用程序的清单文件。 清单文件中仅声明了一项**活动** 。 该活动的突出之处在于，它已通过三个**Intent过滤器**进行了声明。 下图中绿色包围的两个意图过滤器显示了如何为应用程序内容创建**深层链接**（**请参阅参考资料** ）。

> N.B. Deep Links are a concept that help users navigate between **the web** and **applications.** They are basically URLs which navigate users directly to the specific content in **applications.**
>
> NB Deep Links是一个概念，可以帮助用户在**Web**和**应用程序**之间导航。 它们基本上是URL，可将用户直接导航到**应用程序中**的特定内容。

To test the first deep link (i.e. http), I can use **ADB** with the activity manager (am) tool to test that the intent filter URI's specified for deep linking resolve to the correct app activity. The command seen below successfully starts the main activity for the application.

为了测试第一个深层链接(即http)，我可以将**ADB**与活动管理器(am)工具一起使用，以测试为深度链接指定的意图过滤器URI是否可以解析为正确的应用程序活动。 下面显示的命令成功启动了应用程序的主要活动。

```
adb shell am start -W -a "android.intent.action.VIEW" -d "http://level13.hacker101.com" com.hacker101.level
```

I decided to look at the source code for the *MainActivity.* Looking through the java code, I can see that the application creates a **WebView**. Two string variables are also declared, with the variable **str** containing a hardcoded URL.

我决定查看*MainActivity*的源代码。 查看Java代码，可以看到该应用程序创建了一个**WebView** 。 还声明了两个字符串变量，其中变量**str**包含硬编码的URL。

> N.B. The URL for your application will be different.
>
> 注意：您的应用程序的URL将不同。

Entering this URL into a browser brings up the same page seen earlier in the application's main activity.

在浏览器中输入该URL会弹出与该应用程序主要活动中先前所见页面相同的页面。

Clicking the Flag link still returns an invalid request.

单击标志链接仍然返回无效的请求。

Looking at the source code again, I saw that the application retrieved the data stored in the i**ntent** used to launch the main activity. Examining the manifest file from earlier, I know that this data is the **http://level13.hacker101.com** URL. The application then proceeds to use the java **subString(28)** method to ignore the first 28 characters (i.e. **http://level13.hacker101.com** URL) in the data string retrieved from the intent and appends the remaining string value in the **str2** variable with the hardcoded URL link string in the **str** variable. The application then checks if the string contains a "**?**" and adds it to the end of the string if it does not.

再次查看源代码，我发现应用程序检索了存储在用于启动主要活动的**内容中**的数据。 从前面检查清单文件，我知道此数据是**http://level13.hacker101.com** 网址。 然后，应用程序继续使用java **subString(28)**方法来忽略从Intent检索到的数据字符串中的前28个字符(即**http://level13.hacker101.com** URL)，并将其余的字符串值附加到**str2**变量中。在**str**变量中使用硬编码的URL链接字符串。 然后，应用程序检查字符串是否包含" **？** "，如果没有则将其添加到字符串的末尾。

Image for post

I do not know what the value of **str2** is yet based on my static analysis thus far. The final value of **str** so far is a combination of the **hardcoded URL link**, the the **str2 value** (Which at this point is just an empty string) and the **"?"** at the end of the string.

我不知道**str2**的值是多少 到目前为止，仍基于我的静态分析。 到目前为止， **str**的最终值是**硬编码URL链接** ，**str2值** (此时仅是一个空字符串)和**"？"的组合** 在字符串的末尾。

```
http://34.74.105.127/398abac4c8/appRoot'empty str2 value'?
```

The final block of code consists of creating a message digest using the SHA-256 hashing algorithm. The hash is updated twice. The first is with a key called **s00p3rs3c3rtk3y** and the second is with the **str2 value**. The WebView will then load a newly constructed URL which includes the **str** value (i.e.**URL**), the string "**&hash"** and the **SHA-256 hash** value.

最后的代码块包括使用SHA-256哈希算法创建消息摘要。 哈希值被更新两次。 第一个是使用名为**s00p3rs3c3rtk3y**的密钥 第二个是 **str2值** 。 然后，WebView将加载一个新构建的URL，其中包括**str** 值(即**URL**)，字符串" **&hash"**和**SHA-256哈希**值。

Image for post

The final constructed URL so far can be seen below:

到目前为止，最终构造的URL如下所示：

```
http://34.94.3.143/empty str2 value'?&hash="hash value"
```

Using a tool called **BurpSuite**, I can intercept the request made by the application when it is launched and observe the URL that is constructed by the *MainActivity* source code.

使用名为**BurpSuite**的工具，我可以在启动应用程序时拦截应用程序发出的请求，并观察由*MainActivity*源代码构造的URL。

Image for post

Looking at the host value and GET request made by the application, I can see the full URL.

查看应用程序发出的主机值和GET请求，我可以看到完整的URL。

```
http://35.227.24.107/3ef212b832/appRoot?&hash=61f4518d...etc
```

> N.B. The number value in the URL (i.e. 3ef212b832) has changed due to me downloading a new version of the app.
>
> 注意：由于我下载了新版本的应用程序，URL(即3ef212b832)中的数字值已更改。

Entering this value into the URL will still just bring me to the default WebView with a link to get the flag. I know that this link is used to get the flag somehow and by looking at the page source for the link, I can see that it uses **/flagBearer** as part of the URL address. This is placed right after **appRoot** and could be the missing **str2** value.

在URL中输入此值仍将仅使我进入默认WebView，并带有获取该标志的链接。 我知道此链接用于以某种方式获取标志，通过查看该链接的页面源，我可以看到它使用**/ flagBearer**作为URL地址的一部分。 它放置在**appRoot**之后，可能是缺少的**str2**值。

If I add **/flagBearer** to where the missing **str2** value should be, I have the following URL path.

如果我将**/ flagBearer**添加到应该缺少的**str2**值的位置，则我具有以下URL路径。

```
http://35.227.24.107/3ef212b832/appRoot/flagBearer?&hash=61f4518d...etc
```

Entering this URL path into my browser presents a new error message which says “**Invalid hash**”.

在我的浏览器中输入该URL路径会显示一条新错误消息，内容为“ **Invalid hash** ”。

This means that **/flagBearer** is the unknown **str2** value and can be confirmed by typing other values in it’s place, which results in a Not Found error.

这意味着**/ flagBearer**是未知的**str2**值，可以通过在其位置键入其他值来确认，从而导致未找到错误。

Despite having the correct URL path, I am stilled presented with a hash error. As seen earlier while statically analyzing the *MainActivity* source code, the **str2** value (i.e. */flagBearer*) is used with the key **s00p3rs3c3rtk3y** to make up the full SHA-256 hash. However, since the data URL path specified in the intent filter (i.e. *http://level13.hacker101.com*) does not contain the string **/flagBearer** and is completely ignored by the the java **subString(28)** method, the **str2** value is left empty. This means the hash is incorrect since there is no value present in **str2** (i.e. */flagBearer*).

尽管具有正确的URL路径，但仍然出现哈希错误。 如前所述，在静态分析*MainActivity*源代码时， **str2**值(即*/flagBearer*)与键**s00p3rs3c3rtk3y**一起使用 组成完整的SHA-256哈希。 但是，由于在意图过滤器(即*http://level13.hacker101.com*)中指定的数据URL路径不包含字符串**/ flagBearer**，并且被java **subString(28)**方法完全忽略，因此**str2** 值保留为空。 这意味着散列是不正确的，因为在**str2**中没有值(即*/flagBearer*)。

This is where the **deep links** come into play. After reading an article titled “*The Zaheck of Android Deep Links!*” (**see references**), I learnt that if there is **insufficient URL validation** being carried out then I can load my own **arbitrary URL**. I know that no URL validation is being performed on the URL used to trigger the intent filter and launch the application’s Main Activity. This means I can provide my own URL with the **/flagBearer** path attached, which will launch the Main Activity and result in the **str2** value being equal to **/flagBearer**. I can accomplish this by using **ADB**, as seen previously above when testing the intent filter URI’s.

这就是**深层联系**发挥作用的地方。 阅读标题为“ *Android深度链接的Zaheck！* ”(**请参阅参考资料**)，我了解到，如果执行的**URL验证不足** ，则可以加载自己的**任意URL** 。 我知道没有对用于触发意图过滤器和启动应用程序的主活动的URL进行URL验证。 这意味着我可以使用**/ flagBearer**提供自己的URL 附加的路径，它将启动Main Activity并导致**str2**值等于**/ flagBearer** 。 我可以通过使用**ADB**来完成此操作，如上面在测试意图过滤器URI时所见。

```
adb shell am start -W -a "android.intent.action.VIEW" -d "http://level13.hacker101.com/flagBearer" com.hack
```

This results in the flag being presented □.

这导致标记显示为□。

Image for post

Another approach to solve this challenge without using the deep link is to simply **create the hash yourself** by combining the key **s00p3rs3c3rtk3y** and the **/flagBearer** string. I used an online tool called **CyberChef** to create the SHA-256 hash.

无需使用深层链接即可解决此难题的另一种方法是，通过组合键**s00p3rs3c3rtk3y** 自己简单地**创建哈希** 和**/ flagBearer**字符串。 我使用了一个名为**CyberChef**的在线工具来创建SHA-256哈希。

Image for post

I then added this new hash to my URL path, giving me the flag.

然后，我将此新哈希添加到我的URL路径，并给了我该标志。

Image for post

Oauthbreaker(中等，2个标志) (Oauthbreaker (Moderate, 2 Flags))

Using ADB and MobSF, I again installed the application and generated a static analysis report. Opening the application, I am greeted with an activity with a button that says authenticate.

使用ADB和MobSF，我再次安装了该应用程序并生成了静态分析报告。 打开应用程序，我看到一个带有身份验证按钮的活动。

Image for post

Clicking on this button, the WebView browser on my emulator is opened with an address in the URL bar and a link to authorize my mobile application.

单击此按钮，将打开模拟器上的WebView浏览器，并在URL栏中显示一个地址，并提供一个链接来授权我的移动应用程序。

Image for post

The full address in the URL bar can be seen below.

网址栏中的完整地址如下所示。

```
http://34.74.105.127/81857ddddb/oauth?redirect_url=oauth%3A%2F%2Ffinal%2Flogin&response_type=token&scope=al
```

Clicking on the *Authorize Mobile Application* link brings me to a new activity, with a message saying I have “**Successfully authenticated via OAuth!**”.

单击“ *授权移动应用程序*”链接将我带到一个新活动，并显示一条消息，提示我“**已通过OAuth成功验证！** ”。

Image for post

While exploring the functionality of the application, I decided to open the URL shown in the WebView browser on my emulator. While looking at the source code for this page, I discovered the **first flag**.

在探索应用程序的功能时，我决定在模拟器上打开WebView浏览器中显示的URL。 在查看此页面的源代码时，我发现了**第一个标志** 。

Image for post

Examining the Android Manifest file showed that **two activities** called "*com.hacker101.oauth.Browser*" and "*com.hacker101.oauth.MainActivity*" were declared with **intent filters**. As seen with the previous challenge, these intent filters are used to create deep links.

检查Android Manifest文件显示，使用**意图过滤器**声明了**两个**名为*com.hacker101.oauth.Browser*和*com.hacker101.oauth.MainActivity*的 **活动** 。 从上一个挑战中可以看出，这些意图过滤器用于创建深层链接。

Image for post

After reviewing the manifest file, I started to look at the source code for the *MainActivity.* When the Main Activity is created, a variable called **authRedirectUri** is equal to the value "**oauth://final/**". This is the deep link URL used to bring the user to the *Browser* activity. The data contained in the intent used to launch the Main Activity is retrieved and checked to see if it has any data or if the query parameter **redirect_uri** is null. If the intent parameter **redirect_uri** is not null, then the value of this parameter is assigned to **authRedirectUri**.

查看清单文件后，我开始查看*MainActivity*的源代码。 创建主活动时，名为**authRedirectUri**的变量等于值" **oauth：// final /** "。 这是用于将用户带到*浏览器*活动的深层链接URL。 检索并启动用于启动主活动的意图中包含的数据，并检查其是否具有任何数据，或者查询参数**redirect_uri**是否为null。 如果意图参数**redirect_uri**不为null，则将此参数的值分配给**authRedirectUri** 。

Image for post

Further down, I can see that when I click the button "**Authenticate***",* a URL is constructed which includes the URL encoded **authRedirectUri** value. A new intent is then created and the URL is added as data to this intent. This intent is then used to trigger the **Browser activity**.

再往下看，我看到当我单击按钮" **Authenticate** *"时，*将构建一个URL，其中包括URL编码的**authRedirectUri**值。 然后创建一个新的意图，并将URL作为数据添加到该意图。 然后，此意图用于触发**浏览器活动** 。

Image for post

From this source code, I can see that the user will be **redirected** to whatever the **redirect_uri** parameter value is. To test this, I can assign a value to the **redirect_uri** intent parameter and observe if it successfully redirects me.

从此源代码中，我可以看到用户将被**重定向**到无论**redirect_uri**参数值是什么。 为了测试这一点，我可以为**redirect_uri** intent参数分配一个值，并观察它是否成功重定向了我。

```
adb shell am start -W -a "android.intent.action.VIEW" -d "oauth://final/redirect_uri=https://ctftime.org/"
```

This works and I am successfully redirected to the *ctftime.org* website. The first flag is also shown in the URL.

这可以正常工作，我已成功重定向到*ctftime.org*网站。 第一个标志也显示在URL中。

Image for post

I was still unsure about how to use this exploit, so I decided to start looking at the **Browser activity**. I saw that a private class called *SSLTolerentWebViewClient* is created. Inside this class, the *shouldOverrideUrlLoading()* method is declared which allows the host application a chance to take control when a URL is about to be loaded in the current WebView (**see references**). This explains why the WebView browser is opened on my emulator. The method *SslErrorHandler()* is also declared and is used to simply ignore SSL errors.

我仍然不确定如何使用此漏洞，因此我决定开始研究**Browser活动** 。 我看到创建了一个名为*SSLTolerentWebViewClient*的私有类。 在此类内部，声明了*shouldOverrideUrlLoading()*方法，该方法使主机应用程序有机会在当前WebView中要加载URL时进行控制(**请参见参考资料**)。 这解释了为什么在我的模拟器上打开WebView浏览器。 还声明了方法*SslErrorHandler()*，该方法*仅*用于忽略SSL错误。

Image for post

Moving further down in the source code, I can see that a variable called **str** is declared with the value set to a URL address. This URL address is the success message I saw earlier, which tells me I am authenticated. The data from the intent used to launch the activity and the intent parameter called **uri** are checked to see if it is null. The **str** value is then made equal to the data contained in the **uri** intent parameter.

在源代码中进一步往下看，我可以看到一个名为**str**的变量。 声明为具有设置为URL地址的值。 该URL地址是我之前看到的成功消息，告诉我已通过身份验证。 检查用于启动活动的意图数据和名为**uri**的意图参数是否为空。 然后使**str**值等于**uri** intent参数中包含的数据。

Image for post

Beneath this, I can see that a new WebView is created. Two important pieces of information are noted when the WebView is being created. The first is that the WebView has enabled **JavaScript execution** using *setJavascriptEnabled()*. The second is that the method *addJavascriptInterface()* is declared. This injects a supplied Java object into the WebView and **allows the Java object's methods to be accessed from JavaScript**. This method takes two parameters:

在此之下，我可以看到创建了一个新的WebView。 创建WebView时会注意两个重要信息。 首先是WebView使用*setJavascriptEnabled()*启用了**JavaScript执行** 。 第二个是声明方法*addJavascriptInterface()* 。 这会将提供的Java对象注入WebView，并**允许从JavaScript访问Java对象的方法** 。 此方法有两个参数：

The class instance to bind to JavaScript (i.e. *WebAppInterface*)

绑定到JavaScript的类实例(即*WebAppInterface*)

The name to be used to expose the instance in JavaScript (i.e. *iface*).

用于在JavaScript中公开实例的名称(即*iface*)。

Image for post

This allows me to take control of any methods inside the **WebAppInterface** class. Looking at this class, I can see an interesting method called **getFlagPath()**. This method contains what appears to be a large array of int values as seen below.

这使我可以控制**WebAppInterface**类内的任何方法。 看这个类，我可以看到一个有趣的方法，称为**getFlagPath()** 。 该方法包含一个看似大型的int值数组，如下所示。

Image for post

The code below this appears to perform a variety of operations that result in a path to a html file being created.

此代码下面的代码似乎执行了各种操作，这些操作导致创建html文件的路径。

Image for post

Image for post

To call this method, I can create a simple web page using **Github Pages**. I can then add the following JavaScript seen below to the web page, which will call the **getFlagPath()** method using the "**iface**" name which exposes the class instance.

要调用此方法，我可以使用**Github Pages**创建一个简单的网页。 然后，我可以将下面显示的以下JavaScript添加到网页中，该网页将使用“ **iface** ”名称调用**getFlagPath()**方法，该名称公开了类实例。

Next, I can redirect to my Github Pages website as seen earlier by assigning the **uri** parameter with the sites URL address as it's value when calling the **Browser activity**.

接下来，我可以重定向到我的Github Pages网站，如之前所见，可以通过在调用**Browser活动**时为**uri**参数分配站点URL地址作为其值。

```
adb shell am start -W -a "android.intent.action.VIEW" -d "oauth://final/uri=https://github.website.url" com
```

This results in a path for a HTML page.

这将导致HTML页面的路径。

I can simply add this path for the HTML file to the end of the address seen below.

我可以简单地将HTML文件的此路径添加到以下地址的末尾。

```
http://34.94.3.143/cb1f155695/path-to-flag.html
```

This gives me the second flag □.

这给了我第二个标志□。

移动WebDev(中等，2个标志) (Mobile WebDev (Moderate, 2 Flags))

Once again, using ADB and MobSF, I installed the application and generated a static analysis report. Opening the application, I am greeted with an activity which allows me to refresh the page and edit the pages content.

再次使用ADB和MobSF，安装了该应用程序并生成了静态分析报告。 打开该应用程序，我看到了一个活动，该活动使我可以刷新页面并编辑页面内容。

Clicking on the edit button displays a new page which shows the files I can edit.

单击编辑按钮将显示一个新页面，其中显示了我可以编辑的文件。

Clinking on the file allows me to edit it.

叮当响文件使我可以对其进行编辑。

If I click save and return to view, I can see that my edit has been applied.

如果单击“保存”并返回查看，则可以看到我的修改已应用。

After looking at the functionality of the application, I moved on to examining the Android Manifest file for the application. Only **one activity** called *MainActivity* has been declared.

在查看了应用程序的功能之后，我继续检查该应用程序的Android Manifest文件。 仅声明了一个称为*MainActivity*的**活动** 。

I decided to look at the source code for the *MainActivity.* Reading down through the source code, I first notice a variable called **HmacKey** with a string value which appears to be a private key.

我决定查看*MainActivity*的源代码。 通读源代码，我首先注意到一个名为**HmacKey**的变量，它的字符串值似乎是一个私钥。

Further down, I see a method called *Hmac()* which has not been implemented yet, as hinted by the exception message.

再往下看，我看到一个称为*Hmac()*的方法，该方法尚未实现，如异常消息所提示。

Below this, I find the rest of the source code which facilitates the functionality of the application as seen earlier. I can see the different URLs used for viewing and editing content. I can also see that JavaScript has been enabled for the WebView that displays content.

在此之下，我找到了其余的源代码，这些源代码促进了应用程序的功能，如前所述。 我可以看到用于查看和编辑内容的不同URL。 我还可以看到已经为显示内容的WebView启用了JavaScript。

I wasn't really sure what to do with the HMAC key at first, so I ended up trying some different approaches. For example, my initial instinct was to see if I could perform an XSS attack since I could edit the content of the page and JavaScript execution was enabled in the WebView. I presumed if I could execute some JavaScript, a flag might present itself. I added some JavaScript to the *index.html* page as seen below.

起初我不太确定该如何使用HMAC密钥，因此我最终尝试了一些不同的方法。 例如，我的最初目的是查看是否可以执行XSS攻击，因为我可以编辑页面的内容，并且在WebView中启用了JavaScript执行。 我猜想如果我可以执行一些JavaScript，可能会出现一个标志。 我在*index.html*页面中添加了一些JavaScript，如下所示。

After saving the file and viewing the contents page in my browser, I can see the XSS attack works but no flag is displayed.

保存文件并在浏览器中查看内容页面后，我可以看到XSS攻击有效，但未显示任何标志。

After looking around some more, I eventually found a clue on the *edit.php* page. A comment left by the developer referred to a page called *upload.php*.

看了更多之后，我最终在*edit.php*页面上找到了一条线索。 开发人员留下的评论指向一个名为*upload.php*的页面。

As the name suggests, when I visit this page I can upload a file.

顾名思义，当我访问此页面时，我可以上传文件。

Looking at the page source, I can see that the upload form accepts zip files.

查看页面源代码，我可以看到上传表单接受zip文件。

Image for post

When I attempt to upload a zip file however, I am presented with the following error.

但是，当我尝试上传zip文件时，出现以下错误。

Image for post

It appears that I need to provide a HMAC signature when uploading my zip file.

似乎在上传zip文件时需要提供HMAC签名。

> N.B. The **HMAC algorithm** can be used to verify the **integrity** of information passed between applications or stored in a potentially vulnerable location. The basic idea is to **generate a cryptographic hash** of the actual **data** combined with a shared **secret key**. The **resulting hash** can then be used to check the transmitted or stored message to determine a level of trust, without transmitting the secret key.
>
> 注意**HMAC算法**可用于验证在应用程序之间传递或存储在潜在漏洞位置的信息的**完整性** 。 基本思想是**生成**与共享**密钥**组合的实际**数据 的加密哈希** 。 然后，可以将**所得的哈希**用于检查传输或存储的消息以确定信任级别，而无需传输密钥。

Now that I know the purpose of the HMAC private key, I can proceed to create a HMAC signature. To achieve this I used a simple python script available online (**see references**) that computes an HMAC signature.

既然我知道了HMAC私钥的用途，我就可以继续创建HMAC签名了。 为此，我使用了一个在线提供的简单python脚本(**请参阅参考资料**)来计算HMAC签名。

Image for post

Executing this script generates my **HMAC signature**, as seen below.

执行此脚本将生成我的**HMAC签名** ，如下所示。

Image for post

Using Burpsuite, I can then upload my zip file again and this time add my HMAC signature to the POST request as another section in the body of the request.

然后，使用Burpsuite，我可以再次上传zip文件，这次将我的HMAC签名添加到POST请求中，作为请求正文中的另一部分。

Image for post

This is successful and results in the **first flag** being displayed.

这是成功的，并导致显示**第一个标志** 。

Image for post

Upon successfully uploading the zip file, a message is displayed which says that the zip file has been extracted to a folder called */temp* but needs to be **copied** to the */content* folder.

成功上传zip文件后，将显示一条消息，提示该zip文件已被提取到名为/ *temp*的文件夹中，但需要将其**复制**到/ *content*文件夹中。

Image for post

After some searching online for zip file upload directory traversal vulnerabilities, I discovered a vulnerability called the **Zip Slip vulnerability**.

在网上搜索zip文件上传目录遍历漏洞后，我发现了一个名为**Zip Slip漏洞的漏洞** 。

> N.B. A Zip Slip vulnerability allows attackers to create Zip archives that use **path traversal** to overwrite important files on affected systems, either destroying them or replacing them with malicious alternatives.
>
> 注意：Zip Slip漏洞允许攻击者创建Zip存档，这些存档使用**路径遍历**覆盖受影响系统上的重要文件，或者销毁它们或将其替换为恶意替代文件。

To test if this vulnerability exists, I used a zip slip file provided by **Snyk** on their Github (**see references**). I used my python script to create a HMAC signature again but this time with the *zip-slip.zip* file. I then used BurpSuite to intercept the POST request and add the HMAC signature as seen before. Upon successfully uploading the zip file, I received the **second flag**!

为了测试此漏洞是否存在，我使用了**Snyk**在其Github上提供的zip滑动文件(**请参阅参考资料**)。 我使用python脚本再次创建了HMAC签名，但这一次是使用*zip-slip.zip*文件。 然后，我使用BurpSuite拦截了POST请求，并添加了HMAC签名，如之前所示。 成功上传zip文件后，我收到了**第二个标志** ！

Image for post

结束语 (Closing Remarks)

I really enjoyed solving these challenges and found them very useful for teaching how to exploit vulnerabilities that can be found in Android applications. I hope HackerOne will continue to release more Android based CTF challenges in the future and thanks for reading till the end □!

我真的很喜欢解决这些挑战，并发现它们对于教如何利用Android应用程序中的漏洞非常有用。 我希望HackerOne将来会继续发布更多基于Android的CTF挑战，并感谢您阅读本书至尾。

> 翻译自: https://medium.com/bugbountywriteup/hacker101-ctf-android-challenge-writeups-f830a382c3ce

hacker