Wakanim API

My write-up to the successful attempt to reverse engineer the Wakanim API used in the android app.

Why the Android App?

Wakanim does not use a real API on the web version and instead, all the content is rendered server side. They also use Incapsula to protect their site, and it makes it pretty annoying (not impossible) to use a web scraper there. I made a web scraper for it in the past and bypassed Incapsula using puppeteer, remote captcha solving using the Holz API, and some puppeteer-extra plugins. Using the Android App API ensures valid results even if they change their web player and makes it easier to use.

Reversing the API

Requirements

- Windows (not strictly required, but some tools I used are only available on Windows.)
- Android Studio with Emulator installed

  > ⚠ **Warning**
  >
  > Make sure you use a Google APIs Image instead of Google Play Store image because getting root on one of these requires additional steps.

- Wakanim App installed on the Emulator

  > ⓘ **Info**
  >
  > Without the Google Play Store, you can get the App from your regular Android phone or online from something like APK Mirror

- Android debug bridge (ADB) binaries

  > ⓘ **Info**
  >
  > These binaries get shipped with Android Studio and on Windows you usually can find them in `%localappdata%/Android/Sdk/platform-tools`

- Network Traffic analyzer like Fiddler or HttpToolkit (I used Fiddler and the Tutorial expects you to use it if you want to follow.)
- Byte Code Viewer
- Ghidra

Setting up the Analyzer

I at first attempted to just look at all the API requests using a program like Fiddler Classic or HttpToolkit while having the app open on the android emulator. I decided to use Fiddler, but to use it with external devices or in this case the Android emulator, I had to enable it in the Fiddler Settings: `Tools > Options... > Connections > Allow remote computers to connect`. After that, I just set up the emulator to use the proxy with the local IP address "10.0.2.2", and the default fiddler proxy port "8888".

> **Tip**
>
> The IP address "10.0.2.2" is used in the Android Emulator to always represent the host computer, where in my case, Fiddler is running.

> **Info**
>
> If you have problems settings up the Proxy with your android emulator, or it simply does not work, check out Setup a proxy with your emulator

Only doing that just allowed me to analyze unsecured HTTP traffic, but the App uses HTTPS. Fiddler has the capability to decrypt HTTPS traffic, but it is not enabled by default. I enabled it in here: `Tools > Options... > HTTPS > Decrypt HTTPS traffic`. Fiddler then generates a certificate that is used to sign the messages that the Android emulator receives. But because everybody could generate such a certificate, the emulator does not trust it yet. Fixing that, was as simple as, exporting the certificate via `Tools > Options... > HTTPS > Actions > Export Root Certificate to Desktop`, then copying it via

```
adb push %userprofile%/Desktop/FiddlerRoot.cer /sdcard/FiddlerRoot.cer
```

And on the emulator, installing it in the file explorer by clicking on it. I then checked if it is installed in the settings app under `Security & location > Advanced > Encryption & credentials > Trusted credentials > User`. However, I was still not finished yet, because starting with Android 5, user installed certificates are not trusted by default and to work with all apps, is has to be in the "System" Tab instead. Since Android 11, this gets even more enforced[1].

> **Info**

Make sure you select "Apps and VPN" in the installation step or it will not be in the required directory.

Moving the certificate into the system certificate store

User installed certificates are located under `/data/misc/user/0/cacerts-added/<hash>.o`, but it has to be in `/system/etc/security/cacerts/`. As the destination path is in the system partition, I had to start the emulator with the `-writeable-system` flag like this:

```
%localappdata%/Android/Sdk/emulator/emulator.exe -writable-system -avd <AVD Name>
```

> Tip
>
> You can get a list of available AVDs (Android Virtual Device) with:
>
> ```
> %localappdata%/Android/Sdk/emulator/emulator.exe -list-avds
> ```

Then in the last step, I copied the only file in `/data/misc/user/0/cacerts-added/` into the `/system/etc/security/cacerts/` folder using:

```
adb shell cp /data/misc/user/0/cacerts-added/<hash>.o
/system/etc/security/cacerts/
```

> Tip
>
> You can get the name of the certificate found in `/data/misc/user/0/cacerts/` by using
>
> ```
> adb shell ls /data/misc/user/0/cacerts-added/
> ```

And after opening a website that uses HTTPS in chrome on the emulator, I immediately saw the traffic in Fiddler.

SSL Pinning

I thought it would be as easy as that, but after trying to log into an account, I saw that the App does say, that there is a problem with the network connection, even if there is not. That comes from the app using SSL certificate pinning[2]. That just means that it does check if the certificate that was used to sign the messages, is actually the one expected to do it and not, like in our case, something like the Fiddler Root certificate. Using Frida to defeat the SSL pinning[3] is probably one of the easiest solutions. I began by downloading the Frida server archive from their GitHub repository, choosing the latest android server archive with my

emulator architecture, in this case, "frida-server-16.0.8-android-x86_64.xz", unpacking it and using the following command to copy the binary to the emulator file system:

```
adb push ./frida-server-16.0.8-android-x86_64 /data/local/tmp/frida-server
```

and then I ran the server in the background using the following command:

```
adb shell /data/local/tmp/frida-server &
```

The server is running and ready to receive a frida script. I was lazy and just used this one and ran it using:

```
frida -U -l ./frida-script.js -f wakanimapp.wakanimapp
```

I finally could see some API requests, their content and what the server responded. The only thing still not working, was logging into the account, and it still said, that there is a problem with the network connection. If I logged in while the proxy was disabled and re-enabled it later, I could still see all other network requests. But the other routes are completely useless without the access token from the login request. The weird thing was, that it seemed that SSL pinning was disabled and it was, but only for the Java code running. The Frida script gave a hint about this problem:

```
Unpinning Android app...
[+] SSLPeerUnverifiedException auto-patcher
[+] HttpsURLConnection (setDefaultHostnameVerifier)
[+] HttpsURLConnection (setSSLSocketFactory)
[+] HttpsURLConnection (setHostnameVerifier)
[+] SSLContext
[+] TrustManagerImpl
...
[+] Android WebViewClient (SslErrorHandler)
...
Unpinning setup completed
---
  --> Bypassing TrustManagerImpl checkTrusted
  --> Unexpected SSL verification failure, adding dynamic patch...
      Thrown by com.wakanim.wakanimapp.test.wakanimWebclient.WakanimWebClient->p1
      Attempting to patch automatically...
      [+] com.wakanim.wakanimapp.test.wakanimWebclient.WakanimWebClient->p1
(automatic exception patch)
```

The script I used has two methods.

1. It automatically patches common methods of SSL Pinning by some libraries.
2. It waits for a `SSLPeerUnverifiedException` exception to be thrown and just bypasses the whole method where it originated.

The first method actually disabled the default SSL pinning in the Wakanim App, used for almost all API requests, except for the login request. While logging in, the exception gets thrown by their custom check, and just bypassing it makes the request fail. I was confused by why the request failed and checked it out in fiddler. And there was the next hint on what was going wrong:

```
POST https://account.wakanim.tv/core/connect/token HTTP/1.1
Content-Type: application/x-www-form-urlencoded
charset: utf-8
Accept-Language: en
X-DeviceType: Google
X-AppVersion: 7.1.0
X-DeviceVersion: Android SDK built for x86_64
X-SoftwareVersion: 9 P
User-Agent: Dalvik/2.1.0 (Linux; U; Android 9; Android SDK built for x86_64
Build/PSR1.180720.122)
Host: account.wakanim.tv
Connection: Keep-Alive
Accept-Encoding: gzip
Content-Length: 0

(Body would be here if there was any)
```

And yes, that is the whole request. It specified `Content-Type: application/x-www-form-urlencoded` but the Body is empty (`Content-Length: 0`).

Decompiling and Reverse Engineering

So it was time to actually look into the code of it. I started by downloading Byte Code Viewer and I collected the URL of the token endpoint (`https://account.wakanim.tv/core/connect/token`) and the exception that was thrown by the app (`com.wakanim.wakanimapp.test.wakanimWebclient.WakanimWebClient->p1`).

(Reconstructed) `WakanimWebClient.class`

```
  ...

  private void p1(X509TrustManagerExtensions trustManager, HttpsURLConnection
connection, String... arguments) {
    String body = arguments.length > 0 ? arguments[0] : null;
    int returnCode = process(body, connection, trustManager);
    if (returnCode != 0) {
      StringBuilder errorMessage = new StringBuilder();
      errorMessage.append("Failed to sanitize request. \n");
      errorMessage.append(returnCode);
```

```
      throw new SSLPeerUnverifiedException(errorMessage.toString());
    }
  }

  public static native int process(String body, URLConnection connection,
X509TrustManagerExtensions trustManager);

  ...
```

Looking into the method and refactoring it a little bit reveals, that as I guessed, the signature verification does not happen in Java but in a native library. To be able to find out what which variables are, I again used Frida with the following script:

```
setTimeout(() =>  {
  Java.perform(() => {
    const wakanimWebClient = Java.use(
      "com.wakanim.wakanimapp.test.wakanimWebclient.WakanimWebClient"
    );
    wakanimWebClient.p1.implementation = (trustManager, connection, arguments) =>
{
      const returns = this.p1(trustManager, connection, arguments);
      console.log("WakanimWebClient -> p1", arguments);
      return returns;
    };
  });
}, 0);
```

> Tip
>
> You pretty much can do this with every function you find with the decompiler, and that with only some slight modifications. I will continue using this with modifications without providing the exact source of it.

The main reason I hooked the function was that I wasn't sure what the third argument was. After executing it and trying to log in, we see something that looks like we would expect the missing body of the token endpoint would look like (Modified for readability):

```
client_id=wakanim.android.test2
&grant_type=password
&response_type=code+id_token+token
&client_secret=FA2P0X10
&username=EMAIL
&password=PASSWORD
&scope=email+openid+profile+offline_access+read
&redirect_uri=wakanimandroidapp://callback
```

```
&nonce=2c24dd32-af93-4ba5-aa83-f54fbb489f8b
&state=463ee844-0284-4cc2-83fc-1188c8e1997d
```

But it would be too simple just using that and getting an access token. If we do that, we get the same `{"error":"invalid_client"}` response as without a body. So before digging into the native library, I thought I would search where p1 actually gets called from. It was as easy as searching for the token URL path (`/core/connect/token`) that is found also in `WakanimWebClient` and stored as a static variable. After searching for usages of it, I traced it back to `WakanimWebClient$i1.class`.

(Decompiled) `WakanimWebClient$i1.class`

```
public j a(String... var1) {
  ...

  connection.setConnectTimeout(60000);
  connection.connect();
  WakanimWebClient var43 = this.c;
  WakanimWebClient.c(var43, WakanimWebClient.x(var43), connection, new String[]
{var1[0]});
  responseCode = connection.getResponseCode();

  ...
}
```

> Tip
>
> When working with decompiled code it is good to know that,
>
> 1. the standard java HTTP implementation lets you write the request body between connection.connect() and calling any response related method, like in this case the getResponseCode() method
> 2. the java compiler is simplifying a lot of stuff if more information is not required and in this case, it is also obfuscated which makes working with it more unpleasant

At the first look it looks pretty complicated but after analyzing it a bit it just simplifies to following pseudocode:

(Reconstructed) `WakanimWebClient$i1.class`

```
public Response makeRequest(String requestBody) {
  ...
```

```
    connection.setConnectTimeout(60000);
    connection.connect();
    webClient.p1(webClient.getTrustManager(), connection, requestBody);
    int responseCode = connection.getResponseCode();

    ...
}
```

So that is the last proof that we actually need to tackle the native lib. But what lib? Finding that out was pretty easy, because in the constructor of the `WakanimWebClient` class, there is this expression `System.loadLibrary("libsanitize")`

Troubleshooting

Manually setting the Proxy of the Android Emulator

In my case, getting the Emulator to use a Proxy was harder than expected because it just did not want to use the proxy. I gave it in the proxy settings but, there is an easy workaround using adb. If you type in the following command, it sets the address you provide to the global proxy of the android system.

Enable:

```
adb shell settings put global http_proxy "10.0.2.2:8888"
```

Disable:

```
adb shell settings put global http_proxy ":0"
```

---

1. https://httptoolkit.com/blog/android-11-trust-ca-certificates/↩
2. https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning↩
3. https://httptoolkit.com/blog/frida-certificate-pinning/↩