Xamarin Root and SSL Pinning Bypass

Introduction

Many Android applications implement root detection and SSL pinning mechanisms to prevent tampering, protect sensitive functionality, and secure network communications. While effective against casual users, these controls can hinder legitimate security research by restricting analysis on rooted test devices and blocking traffic inspection.

In this post, I describe how both root detection and SSL pinning were bypassed in a Xamarin-based Android application using static reverse engineering techniques. The approach focuses on decompiling the application, extracting and modifying Xamarin assemblies, and rebuilding the app without relying on runtime hooking or dynamic instrumentation frameworks.

This work was conducted strictly for educational and authorized security testing purposes, with the goal of understanding and evaluating mobile application security controls.

Tools Used

The following tools were used throughout the analysis and bypass process:

- `ADB` (installed within Android SDK platform tools)
- `Jadx-Gui`
- `APKTool`
- `pyxamstore`
- `dnSpy`
- `zipalign` (installed within Android SDK platform tools)
- `apksigner` (installed within Android SDK platform tools)

Target Application Overview

The target application is a Xamarin-based Android app that implements managed root/jailbreak detection and SSL pinning logic within its .NET assemblies.

The primary challenge of this assessment was bypassing these security mechanisms, as the checks are enforced at the managed code level rather than through native libraries, making traditional smali-only patching ineffective

High-Level Attack Methodology

The overall approach followed in this research can be summarized in the following steps:

- Extracting the APK from a rooted device
- Reverse engineering application package
- Extracting Xamarin (.NET) assemblies
- Patching the managed root detection logic and SSL pinning Methods
- Rebuilding, aligning, and re-signing the application
- Reverse engineering the APK
- Extracting Xamarin assemblies
- Patching root detection logic
- Rebuilding and resigning the application

Step-by-Step Technical Walk through

Step 1: Extracting the APK from a rooted device

The first step is extracting the target APK from the rooted device to begin static analysis.

To do this, we first need to identify the application package name. In my case, this was done using the following command:

```
> adb shell pm list packages
```

To save time and filter the results, I used `grep` to search for the target package name:

```
> adb shell pm list packages | grep -i example
```

Note: On Windows systems, `Select-String` can be used instead of `grep`.

The output revealed the package name: like `com.example.com`

Next, I retrieved the APK path associated with this package using:

```
> adb shell pm path com.example.com
```

This command returned the full path to the APK file onto the device.

Finally, I pulled the APK from the device to my local machine using:

```
> adb pull <APK_PATH_FROM_PREVIOUS_COMMAND>
```

At this point, the APK was successfully extracted and ready for further reverse engineering and analysis.

Step 2: Reverse Engineering the APK and extracting DLL files

At this stage, the extracted APK is reverse engineered to access its internal structure and resources.

First, `APKTool` was used to decompile the application:

```
> apktool d com.example.com.apk
```

Figure 1 De-Compiled Files

Once the decompilation process is completed, I navigated to the following directory within the extracted application folder:

`app_folder/unknown/assemblies`

This directory contains the Xamarin assemblies bundled with the application.

Figure 2 Assemblies files in app_folder/unknown/assemblies

To avoid modifying the original extracted files, the `assemblies` folder was copied to a new working directory.

Next, the `pyxamstore` tool was used to unpack the Xamarin assemblies into readable `.dll` files:

```
> pyxamstore unpack -d assemblies_folder_path
```

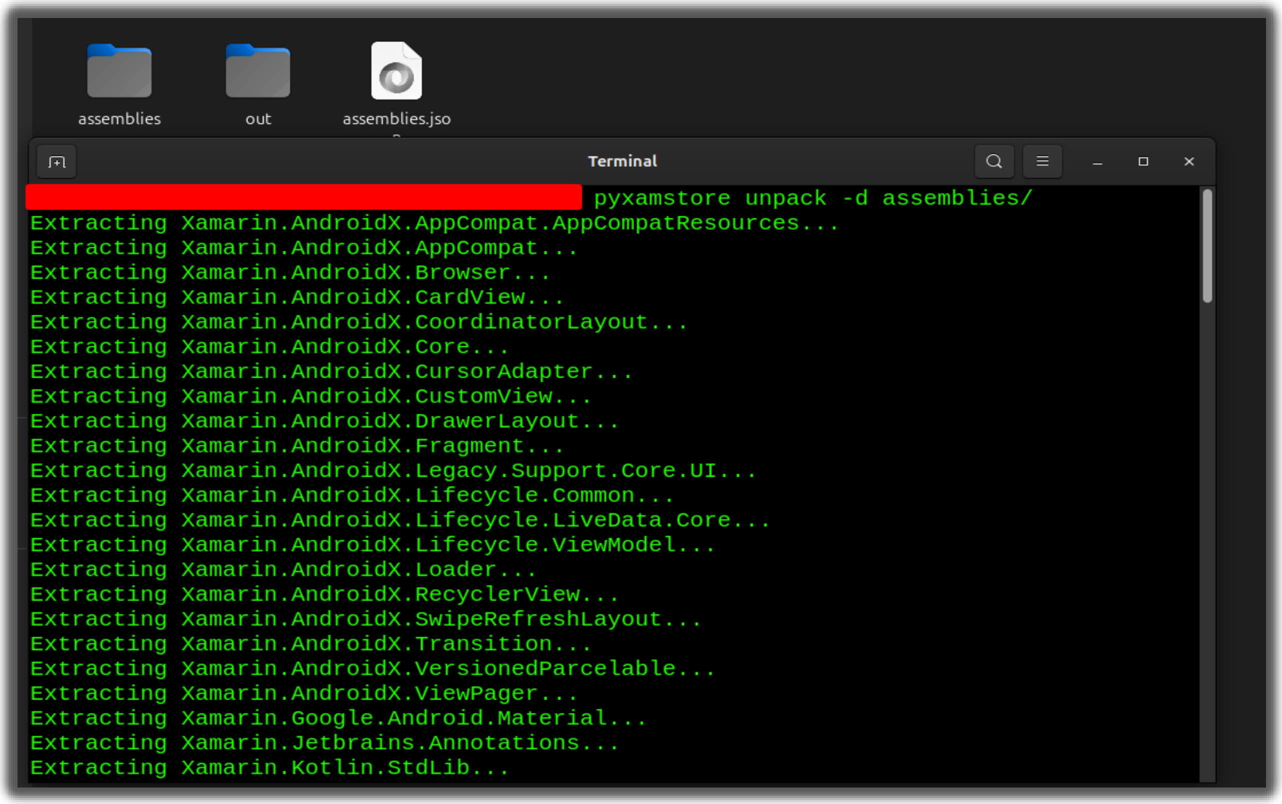


Figure 3 Unpacking to extract DLL Files

After running this command, two important artifacts were generated

- A new directory named `out`, containing the extracted `.dll` files
- An `assemblies.json` file, which will later be required when repacking the patched assemblies

At this point, the managed assemblies were successfully extracted and ready for static analysis and patching, which will be covered in the next step.

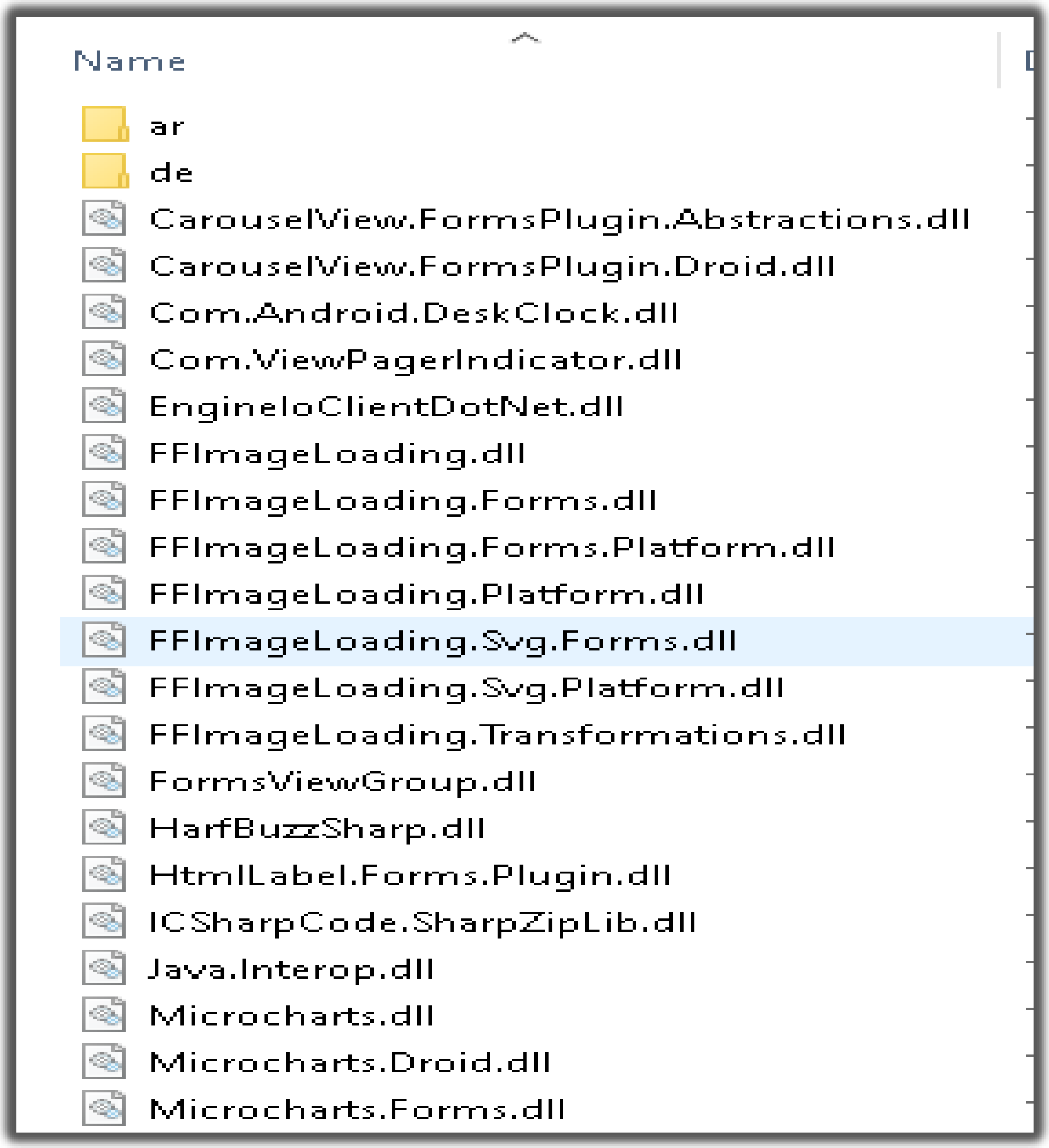


Figure 4 Unpacked DLL Files

Step 3: Static Analysis and Patching the Root Detection Logic

With the Xamarin assemblies extracted, the next step was to analyze and patch the root detection logic.

Using **`dnSpy`**, I imported the suspected `.dll` files that potentially contained the root and debugger detection mechanisms. The analysis started by locating the **`MainActivity`** and following the application execution flow to identify where security checks were triggered.

During the analysis, multiple root-related methods were identified, including:

- `isDeviceRooted`

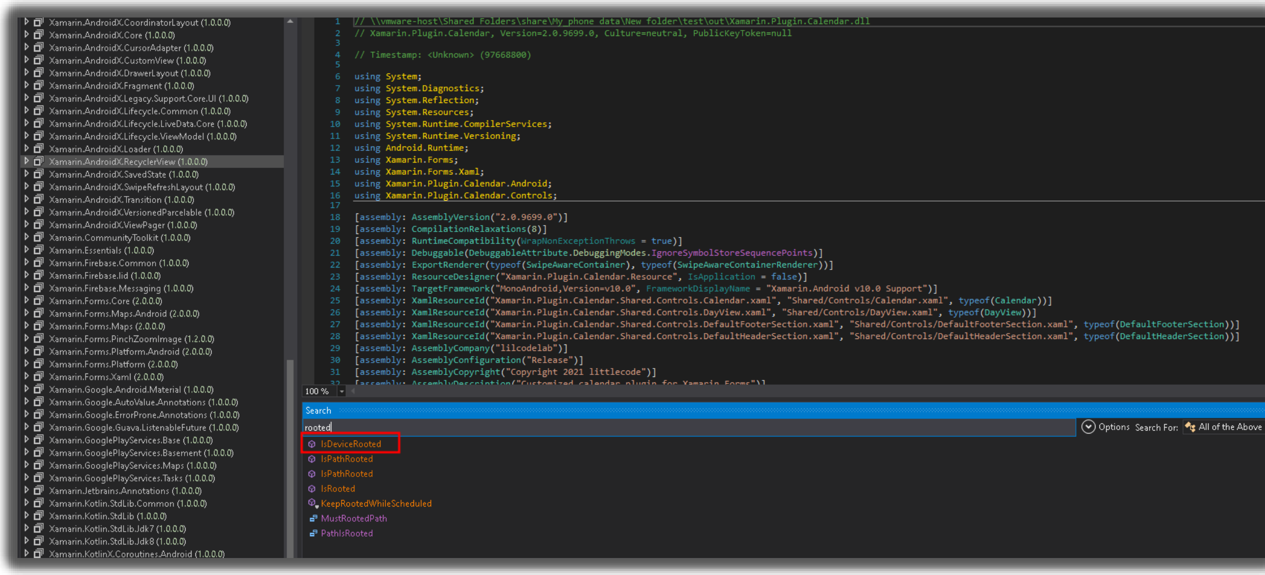


Figure 5 Static Analysis to Get the root detection mechanism

As an alternative approach, dnSpy's search functionality was also used to quickly locate relevant detection logic by searching for common keywords such as:

`rooted`, `hooked`, `frida`, `superapk`, `magisk`, `integrity`.

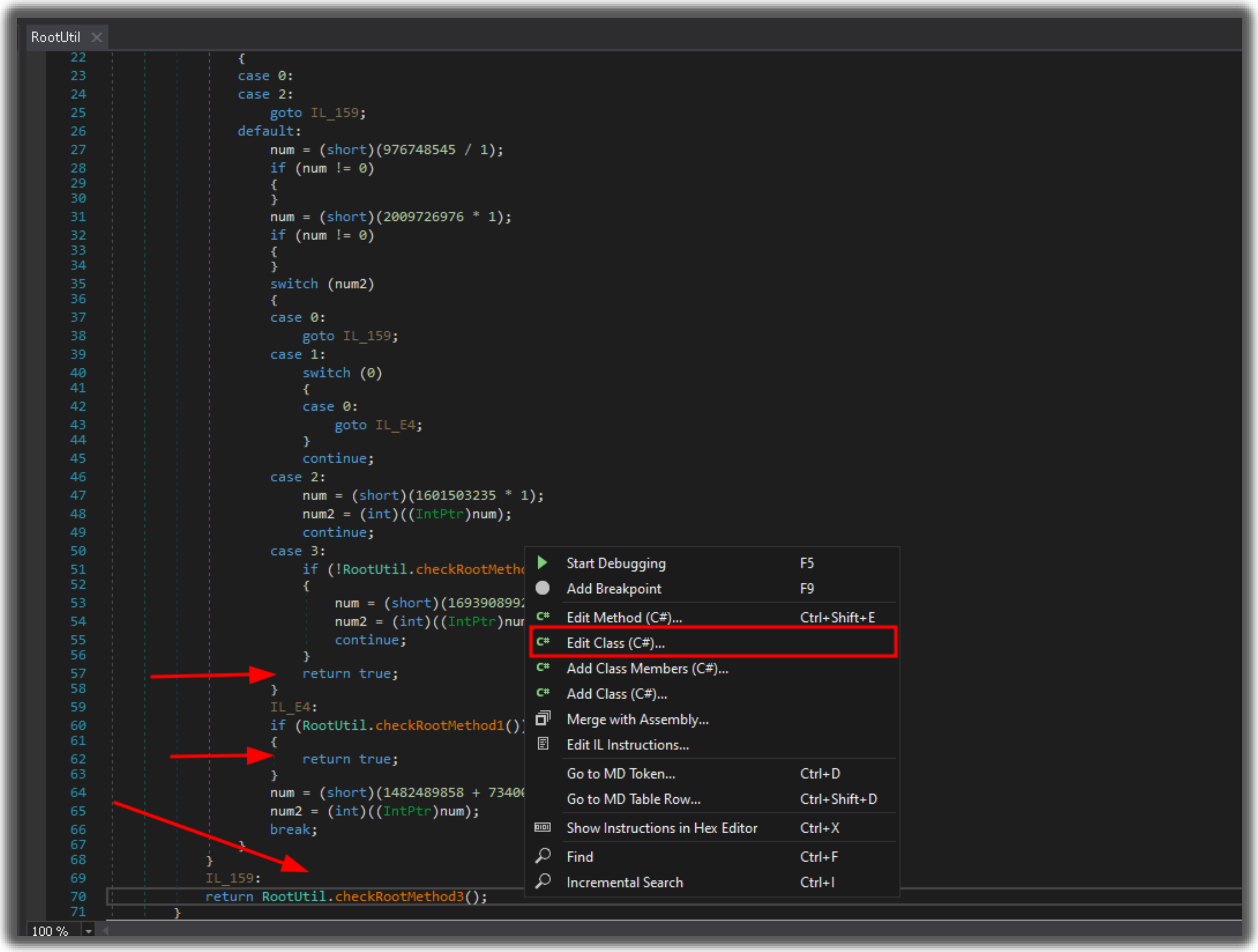
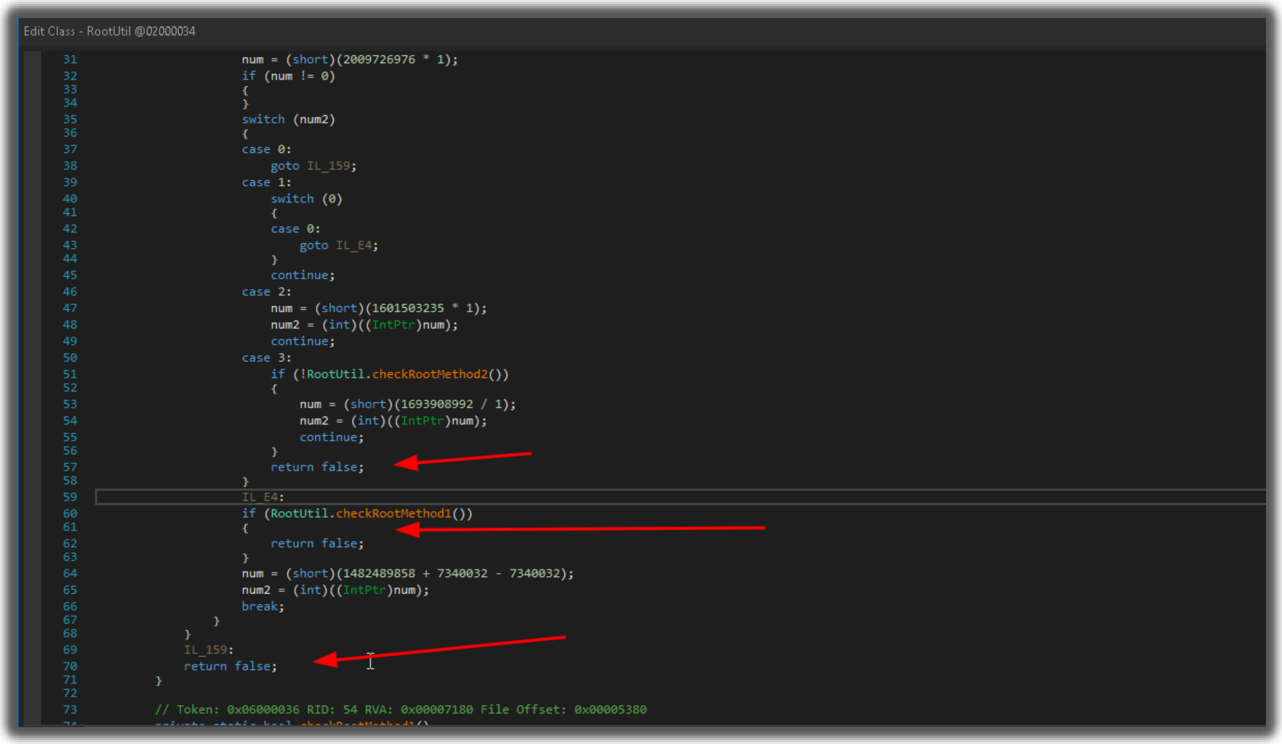


Figure 6 Editing the Method Return or Logic

This helped uncover additional root detection mechanisms scattered across the assemblies.

Once the relevant methods were identified, `dnSpy`'s Edit Class feature was used to modify the logic. The return values of the root and debugger detection methods were patched to always return `false`.



Edit Class - RootUtil @02000034

```
                    num = (short)(2009726976 * 1);
                    if (num != 0)
                    {
                    }
                    switch (num2)
                    {
                    case 0:
                        goto IL_159;
                    case 1:
                        switch (0)
                        {
                        case 0:
                            goto IL_E4;
                        }
                        continue;
                    case 2:
                        num = (short)(1601503235 * 1);
                        num2 = (int)((IntPtr)num);
                        continue;
                    case 3:
                        if (!RootUtil.checkRootMethod2())
                        {
                            num = (short)(1693908992 / 1);
                            num2 = (int)((IntPtr)num);
                            continue;
                        }
                        return false;
                    }
                    IL_E4:
                    if (RootUtil.checkRootMethod1())
                    {
                        return false;
                    }
                    num = (short)(1482489858 + 7340032 - 7340032);
                    num2 = (int)((IntPtr)num);
                    break;
                }
            }
            IL_159:
            return false;
        }

        // Token: 0x06000036 RID: 54 RVA: 0x00007180 File Offset: 0x00005380
```

Figure 7 Modified Function

After saving the modified assemblies, the root detection logic was successfully neutralized, completing the patching phase

Step 4: Patching the SSL Pinning Logic

After successfully bypassing the root detection logic, the next step was to disable the SSL pinning mechanism implemented at the managed code level.

Using the same static analysis approach in `dnSpy`, I searched through the extracted Xamarin assemblies for certificate validation–related methods. This was done by looking for common SSL and networking keywords such as:

`HttpClientHandler, ServerCertificate, ValidationCallback, SslPolicyErrors`

During this process, I identified the following method responsible for enforcing SSL certificate validation:

`ServerCertificateCustomValidationCallback`

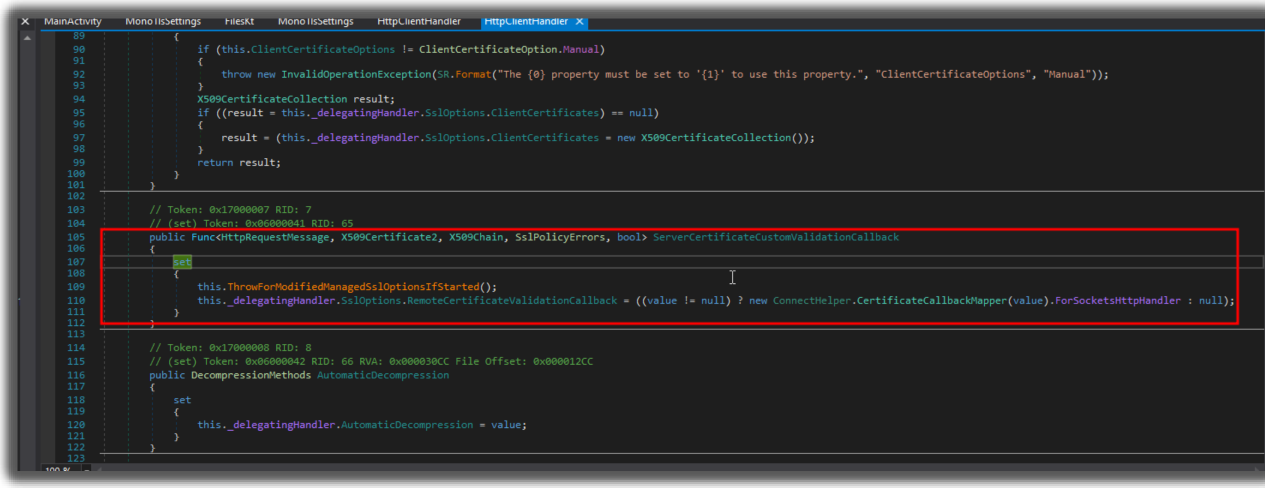


Figure 8 ServerCertificateCustomValidationCallback Function

This method is part of the `HttpClientHandler` implementation and is used to validate the server's TLS certificate during HTTPS connections. The application relied on this callback to enforce SSL pinning by rejecting untrusted or intercepted certificates.

To bypass SSL pinning, the method was patched using dnSpy's **Edit Class** feature. The validation logic was modified so that the callback always returns `true`, effectively instructing the application to trust all certificates, regardless of their validity or trust chain.

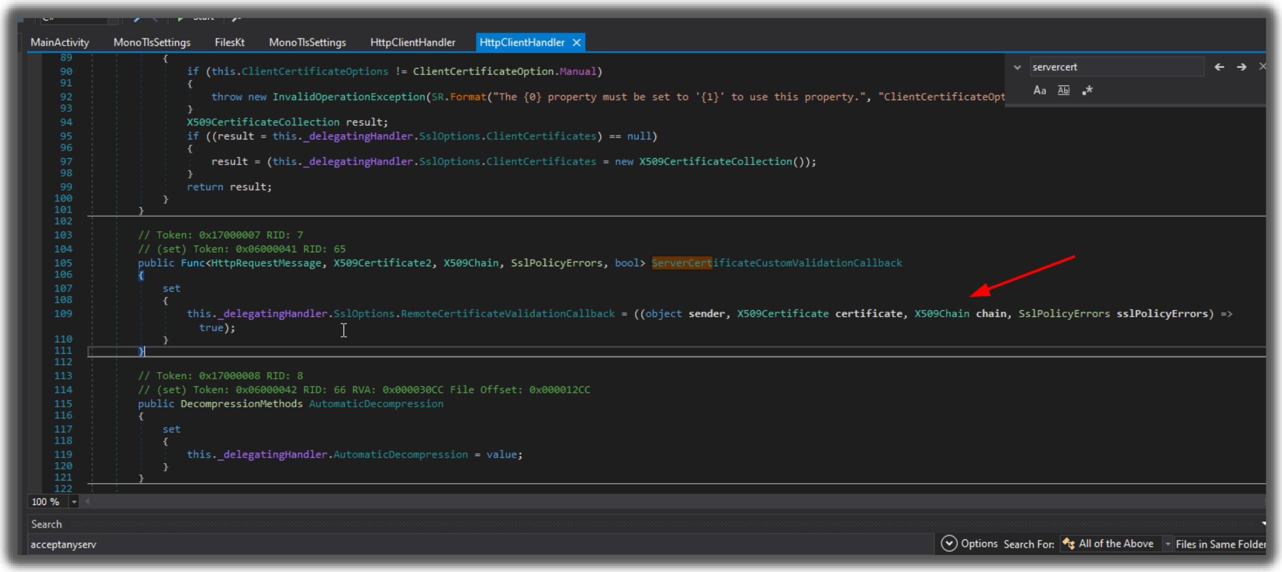


Figure 9 After Modifications to bypass SSL Pinning

By forcing the certificate validation callback to always return `true`, SSL pinning was completely disabled. This allowed HTTPS traffic to be intercepted and analyzed without triggering certificate validation errors.

At this point, both root detection and SSL pinning mechanisms were successfully bypassed using purely static patching techniques, without relying on runtime hooking frameworks such as `Frida` or `Xposed`

Step 5: Repacking the Patched Assemblies Using pyxamstore

After patching the required `.dll` files, the next step was to repack the modified assemblies back into Xamarin's original format.

Using `pyxamstore`, and from within the directory containing the `out` folder and the `assemblies.json` file, the following command was executed:

```
> pyxamstore pack
```

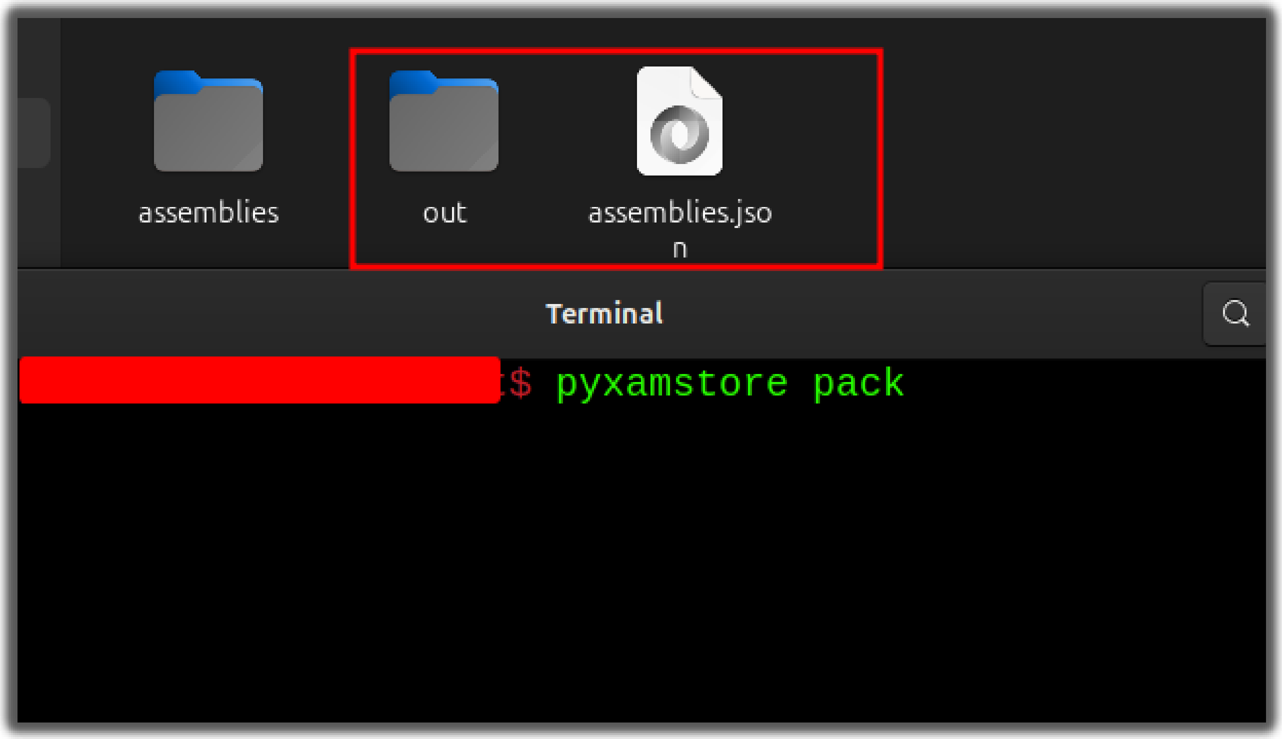


Figure 10 Packing the Patched DLL Files

Important: *This command must be executed from the same directory that contains both the* ***`out`*** *folder and the* ***`assemblies.json`*** *file.*

Upon successful execution, two new files were generated:

- `assemblies.blob.new`
- `assemblies.manifest.new`

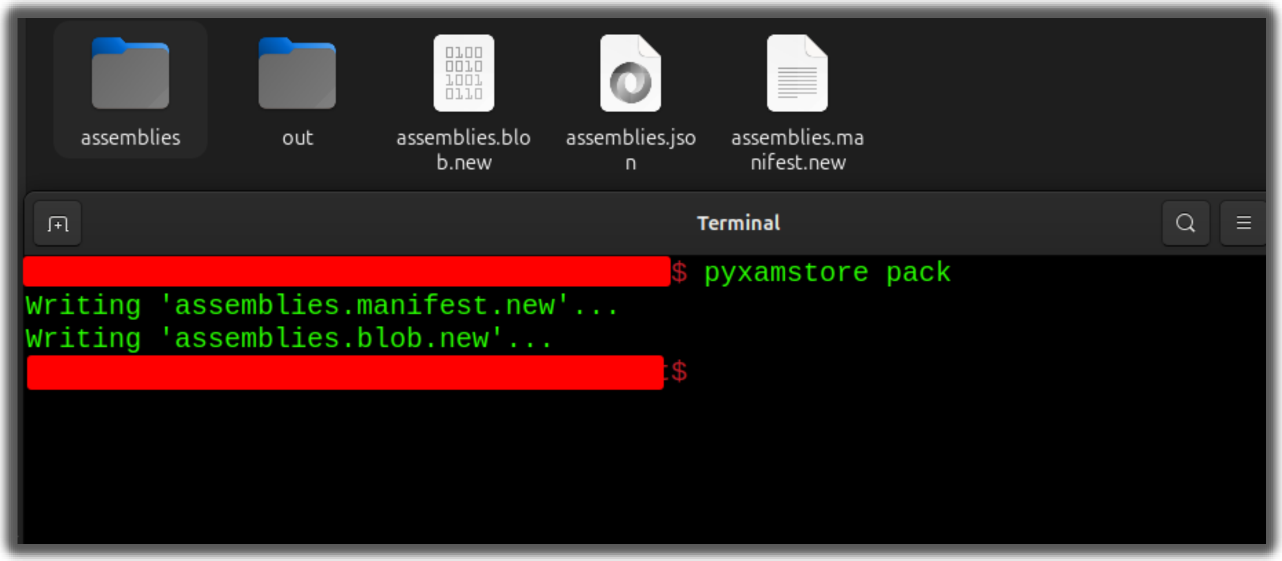


Figure 11 New Assemblies after Patching and Packing

These newly generated files were then used to replace the original files located in:

`app_name/unknown/assemblies`

With this step completed, the patched Xamarin assemblies were successfully repacked and integrated back into the application structure.

Step 6: Rebuilding, Aligning, and Signing the APK

With the patched assemblies in place, the final step was to rebuild the APK and prepare it for installation.

- **Rebuilding the APK**

The application was rebuilt using APKTool:

```
> apktool b app_folder
```

After a successful build, the rebuilt APK is located at:

`app_folder/dist`

- **Aligning the APK**

To optimize the rebuilt APK for proper memory alignment, the zipalign tool was used:

```
> zipalign -v 4 the_extracted_apk out.apk
```

This produces a new aligned APK named `out.apk`

- **Signing the APK**

Before installation on a device, the APK must be signed using `apksigner`. A keystore is required for signing. If a keystore is not available, it can be generated using:

```
> keytool -genkeypair -v -keystore keystore-name.jks -alias alias -keyalg RSA -keysize 2048 -validity 10000
```

Once the key store is ready, the APK can be signed:

```
> apksigner sign --ks-key-alias alias -ks ~/keystore-name.jks out.apk
```

Note: `apktool`, `zipalign`, `apksigner`, and `keytool` are available via Android Studio or the Android SDK Platform Tools from the official Android website.

After completing these steps, the APK is fully rebuilt, patched, aligned, and signed, making it ready for installation on a device using

```
> adb install out.apk
```

Step 7: IP-tables Traffic Routing to Burp Suite

In this final step, we redirect the device's network traffic to Burp Suite by configuring iptables rules on the Android device. This allows all HTTP and HTTPS requests to be transparently intercepted without modifying proxy settings at the application level.

Figure 12: IP-Tables Routing

1. Flush Existing NAT Rules

Start by clearing up any existing NAT rules to avoid conflicts:

```
> adb shell "iptables -t nat -F"
```

2. Redirect Outbound Traffic to Burp Suite

Add DNAT rules to redirect all outbound HTTP and HTTPS traffic to Burp Suite:

```
> adb shell "iptables -t nat -A OUTPUT -p tcp --dport 80 -j DNAT --to-destination 172.16.35.138:8080"
> adb shell "iptables -t nat -A OUTPUT -p tcp --dport 443 -j DNAT --to-destination 172.16.35.138:8080"
```

These rules ensure that any application traffic destined for ports 80 or 443 is transparently forwarded to Burp, regardless of the app's internal networking logic.

3. Enable Masquerading (SNAT)

To ensure proper return routing and prevent connection issues, enable masquerading for both HTTP and HTTPS traffic:

```
> adb shell "iptables -t nat -A POSTROUTING -p tcp --dport 80 -j MASQUERADE"
> adb shell "iptables -t nat -A POSTROUTING -p tcp --dport 443 -j MASQUERADE"
```

Masquerading rewrites the source address, allowing responses from Burp Suite to be correctly routed back to the originating application.