Bugsmirror MASST Integration Documentation

Table of Contents

1. MASSTCLI

Masst CLI is a cross-platform CLI tool for integrating security into your app binaries. Please ensure you follow the steps corresponding to your operating system for smooth execution.

Pre-Requisites:

- **For iOS (Shield Integration):**
  If you are opting for Shield on iOS, please refer to the following documentation for the required pre-submission changes (Click Here)

- **For Android (Shield Integration):**
  If you are opting for Shield on iOS, please make sure to update the required pre-submission changes:

Android ProGuard Configuration :

- ProGuard rules are defined in a file named `proguard-rules.pro`, which is typically located at: `android/app/proguard-rules.pro` , This file is referenced from your app-level `build.gradle` (or `build.gradle.kts`) under the `release` build type.

- If **you do not have a `proguard-rules.pro` file**, you can create one manually in the `android/app/` directory and ensure it is linked in your `build.gradle` configuration.

  NOTE: Only once the `proguard-rules.pro` file has been identified or created, continue with the steps below.

  - **For Flutter Apps (Android Side):**
    Ensure the following ProGuard rule is added before submission:

```
None
-keep class io.flutter.embedding.** { *; }
```

  - **For React Native Apps (Android side):**
    Ensure the following ProGuard rules are added before submission:

```
None
-keep class com.facebook.react.** { *; }
-dontwarn com.facebook.react.**
```

2.1. macOS

1. **Install Dependencies**

   Ensure your system has all required dependencies installed for iOS and Android builds.Verify the following tools are available in your `$PATH`:

   - Xcode & Command Line Tools (for iOS builds)
   - Android Studio / Command line tools (for Android builds)
   - JavaJDK (if required by your Android build)

2. **Download the MASSTCLI tool:**

   You can download and install MASSTCLI using either the MASST Portal or direct `curl` commands. Follow the method that best suits your workflow.

   ### Option A: Download Using `curl` (Recommended)

   Downloading the tool using `curl` fetches the binary directly via the terminal. As a result, **no additional security permission steps are required**.

   Choose the download command corresponding to your system architecture (ARM64 or AMD64) before proceeding.

   ARM64

```
Shell

curl -L
"https://storage.googleapis.com/masst-assets/Defender-Binary-Integrator/1.0
.0/MacOS/MASSTCLI-v1.1.0-darwin-arm64.zip" -o masst_cli.zip
```

   AMD64

```
Shell

curl -L
"https://storage.googleapis.com/masst-assets/Defender-Binary-Integrator/1.0
.0/MacOS/MASSTCLI-v1.1.0-darwin-amd64.zip" -o masst_cli.zip
```

   After downloading:

   - `unzip masst_cli.zip` into a preferred folder (e.g., ~/MASSTCLI)

You can now proceed directly to running the tool.

Option B: Download from MASST Portal

You may also download the CLI directly from the **MASST Portal UI** by selecting:

- **Platform:** macOS
- **Architecture:** ARM64 or AMD64

Once downloaded, extract the ZIP file into a preferred folder (e.g., ~/MASSTCLI) and make the binary executable if required:

```
Shell
chmod +x MASSTCLI
```

Security Permission (macOS Only — Portal Downloads)

When the CLI is downloaded via the **MASST Portal (browser download)**, follow the instructions below:

1. Attempt to run the binary once:

```
Shell
./MASSTCLI
```

2. Go to:

```
None
System Settings → Privacy & Security
```

3. Click **Allow Anyway** for MASSTCLI
4. Re-run the command

NOTE: This step is **not required** when downloading the tool using `curl`.

Help Command (Reference)

Before running the tool, you can view all supported commands and flags using:

```
Shell
./MASSTCLI --help
```

This command lists all required and optional parameters explained below.

```
Shell
Required Flags (iOS):
  -input        string   Path to the input file (.ipa or .xcarchive)
  -config       string   Path to the SDK configuration file (.bm)
  -identity     string   Signing identity for App Store submission

Required Flags (Android):
  -input        string   Path to the input file (.apk or .aab)
  -config       string   Path to the SDK configuration file (.bm)
  -keystore     string   Path to the Android keystore file
  -storePassword string  Password for the keystore
  -alias        string   Alias for the signing key
  -keyPassword  string   Password for the alias key

Optional Flags:
  -apk          Generate APK from AAB (Android only)
  -dev          Build iOS simulator binary (iOS only)
  -v            Enable verbose logging
  -shieldLevel  Shield protection level (1, 2, or 3)
```

3. Run the Tool

- **for IOS**

  When integrating an **iOS application** using MASSTCLI, you must provide a valid Apple Distribution signing identity using the `-identity` flag.

  To locate the correct signing identity installed on your system, run the following codesign command in your terminal:

```
Shell
security find-identity -v -p codesigning
```

  Pass the selected identity exactly as shown when running the tool:

```
Shell
./MASSTCLI -input=MyApp.xcarchive -config=config.bm -identity="Apple
Distribution: My Company"
```

- **for Android**

```shell
./MASSTCLI -input=Test/test.aab -config=Test/config.bm
-keystore=Test/key -storePassword=dfdf -alias=key0
-keyPassword=dfsdfs
```

Note:

If you are using the integrator for an iOS app and have opted to enable Shield protection, ensure that you pass the appropriate `-shieldLevel` flag.

- **Shield Level 1 and Shield Level 2** are supported for **all app types** (Native Swift, Flutter, Ionic, etc.).
- **Shield Level 3** is **specifically designed for React Native iOS applications**.

If you are integrating a **React Native iOS app**, the integrator will automatically surface **Shield Level 3** as the recommended option at runtime. You may still choose a different shield level if required.

If the `-shieldLevel` flag is **not provided** and the integrator is executed with Shield enabled, the tool will prompt you at runtime to select from the available shield level options.

In some cases, an application may crash at runtime for a specific shield level due to compatibility constraints. If this occurs, we recommend retrying the integration using a **lower shield level** than the one previously selected.

2.2. Windows

1. **Install Dependencies**

   Ensure the following are installed and added to your system `PATH`:

   - [Xcode not required – only Android SDK for Android builds]
   - Android Studio / Command line tools and d8 tools
   - Java JDK
   - [Apple tools only work on macOS; iOS integration is not supported on Windows]
2. **Download the MASSTCLI tool:**

   You can download and install MASSTCLI using either the MASST Portal or direct `curl` commands. Follow the method that best suits your workflow.

Option A: Download Using `curl`

Download the ZIP file using the command that matches your system architecture.

ARM64

```
curl -L "https://storage.googleapis.com/masst-assets/Defender-Binary-Integrator/1.0.0/Windows/MASSTCLI-v1.1.0-windows-arm64.zip" -o masst_cli.zip
```

AMD64

```
curl -L "https://storage.googleapis.com/masst-assets/Defender-Binary-Integrator/1.0.0/Windows/MASSTCLI-v1.1.0-windows-amd64.zip" -o masst_cli.zip
```

After downloading:

- `unzip masst_cli.zip` into a preferred folder (e.g., ~/MASSTCLI)

You can now proceed directly to running the tool.

Option B: Download from MASST Portal

You may also download the CLI directly from the **MASST Portal UI** by selecting:

- **Platform:** Windows
- **Architecture:** ARM64 or AMD64

Once downloaded, extract the ZIP file into a preferred folder (e.g., ~/MASSTCLI)

Help Command (Reference)

Before running the tool, you can view all supported commands and flags using:

```
./MASSTCLI --help
```

This command lists all required and optional parameters explained below.

```
Required Flags (Android):
  -input        string   Path to the input file (.apk or .aab)
  -config       string   Path to the SDK configuration file (.bm)
  -keystore     string   Path to the Android keystore file
```

```
  -storePassword string  Password for the keystore
  -alias        string   Alias for the signing key
  -keyPassword  string   Password for the alias key

Optional Flags:
  -apk          Generate APK from AAB (Android only)
  -v            Enable verbose logging
```

3. Run the Tool

Shell

```
./MASSTCLI -input=Test/test.aab -config=Test/config.bm
-keystore=Test/key -storePassword=dfdf -alias=key0
-keyPassword=dfsdfs
```

NOTE: No executable permission (chmod) step is required on Windows.

2.3. Linux

1. **Install Dependencies**

   Ensure build tools are installed and available in `PATH`:

   i. Android Studio / Command line tools
   ii. Java JDK
   iii. (iOS builds are **only supported on macOS** due to Apple toolchain requirements)

   NOTE: Make sure **ANDROID_SDK** value contains the path of android sdk and path for d8 tools is also set.

2. **Download the MASSTCLI tool**

   You can download and install MASSTCLI using either the MASST Portal or direct `curl` commands. Follow the method that best suits your workflow.

Option A: Download Using `curl`

Download the ZIP file using the command that matches your system architecture.

ARM64

```shell
curl -L "https://storage.googleapis.com/masst-assets/Defender-Binary-Integrator/1.0.0/Linux/MASSTCLI-v1.1.0-linux-arm64.zip" -o masst_cli.zip
```

AMD64

```shell
curl -L "https://storage.googleapis.com/masst-assets/Defender-Binary-Integrator/1.0.0/Linux/MASSTCLI-v1.1.0-linux-amd64.zip" -o masst_cli.zip
```

After downloading:

- `unzip masst_cli.zip` into a preferred folder (e.g., ~/MASSTCLI)

You can now proceed directly to running the tool.

Option B: Download from MASST Portal

You may also download the CLI directly from the **MASST Portal UI** by selecting:

- **Platform:** Linux
- **Architecture:** ARM64 or AMD64

Once downloaded, extract the ZIP file into a preferred folder (e.g., ~/MASSTCLI)

Help Command (Reference)

Before running the tool, you can view all supported commands and flags using:

```shell
./MASSTCLI --help
```

This command lists all required and optional parameters explained below.

```shell
Required Flags (Android):
  -input        string   Path to the input file (.apk or .aab)
  -config       string   Path to the SDK configuration file (.bm)
  -keystore     string   Path to the Android keystore file
  -storePassword string  Password for the keystore
  -alias        string   Alias for the signing key
```

```
  -keyPassword  string   Password for the alias key

Optional Flags:
  -apk          Generate APK from AAB (Android only)
  -v            Enable verbose logging
```

3. Run the Tool

Shell

3. `./MASSTCLI -input=Test/test.aab -config=Test/config.bm -keystore=Test/key -storePassword=dfdf -alias=key0 -keyPassword=dfsdfs`

Summary Notes:

- **iOS builds:** Only possible on macOS (Xcode is required).
- **Android builds:** Supported on macOS, Linux, and Windows.
- Always check official docs for build environment setup:
  - Xcode Documentation
  - Android Developer Tools
  - Java SE Documentation

2. Compile Time Integration

2.1. Android Defender

The Android Defender works for both **'debug'** and **'release'** build variant in a following way:

1. **Debug Build** - Threats are only logged; no mitigations are applied, and customized features remain disabled.
2. **Release Build** - Selected mitigations and features are active. For Threat Lens, detected threats are also reported to the server.

2.1.1 Android Native

For Android Native (developed in Java/Kotlin) applications, the files are provided in SDK which can be downloaded from MASST Portal.

Step 1: Unzip the SDK downloaded from MASST, which contains

1. AAR files (Release and Debug)
2. C++ folder (with verify binding utilities)
3. authenticator.xml (com_bugsmirror_defender_authenticator.xml)

Step 2: Paste **`com_bugsmirror_defender_authenticator.xml`** into ***(res/xml)*** and replace **`<appPackageName>`** in **`android:accountType`** with your **`applicationId`**.

XML

```
<!-- TODO: replace <appPackageName> with the actual package name of your app
-->
<!-- EXAMPLE:
android:accountType="com.bugsmirror.defender.com.example.yourapp" -->
<account-authenticator
xmlns:android="http://schemas.android.com/apk/res/android"
    android:accountType="com.bugsmirror.defender.<appPackageName>"
    />
```

Step 3: Add the following service under the application tag of the **`AndroidManifest.xml`** .

None

```
        <service

android:name="com.bugsmirror.defender.authenticator.DefenderAuthenticatorServ
ice"
            android:exported="false">
            <intent-filter>
                <action android:name="android.accounts.AccountAuthenticator"
/>
```

```
            </intent-filter>

            <meta-data
                android:name="android.accounts.AccountAuthenticator"
                android:resource="@xml/com_bugsmirror_defender_authenticator"
/>
        </service>
```

Step 4: Copy the AAR files into ***(app/libs)*** (create the folder if needed) and add the path in your app module's **`build.gradle.`**

None

```
"dependencies" {
      debugImplementation(files("libs/debug_defender_release_3.0.0.aar"))
      releaseImplementation(files("libs/release_defender_release_3.0.0.aar"))

}
```

Step 5: Add the following code inside the **`android{}`** block of your app module's **`build.gradle`**.

None

```
android {
   packaging {
     jniLibs {
         useLegacyPackaging = true
     }
   }
   externalNativeBuild {
      cmake {
         path = file("src/main/cpp/CMakeLists.txt")
         version = "3.22.1"
      }
   }

}
```

Step 6: In your launcher activity, declare the native method and load its library using the code block below:

Kotlin

```
class MainActivity : AppCompatActivity() {   // Kotlin Code Snippet
```

```
        init {
            System.loadLibrary("defender-wrapper") // Loading the native library
        }

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            verify() // Calling the declared method
        }

        private external fun verify() // Native method declaration
    }
```

Java

```
public class MainActivity extends AppCompatActivity {   // JAVA Code Snippet
    static {
        System.loadLibrary("defender-wrapper");
    }
    private native void verify();
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        verify();
    }
}
```

Step 7: Update the method signature in **`defenderapp.cpp`** **(*src/main/cpp*)** to match the one declared in your Java/Kotlin class. For example, if you declare a **`verify()`** method in your activity, the corresponding **`C++ method name`** should be:

C/C++

```
// Package: com.example.app
// Class:   MainActivity
// Method:  verify()
Java_com_example_app_MainActivity_verify

// Package: com.example_app
// Class:   MainActivity
// Method:  verify()
// (notice the underscore in package becomes _1)
Java_com_example_1app_MainActivity_verify
```

Step 8: Add the following code to your **`Application`** class. If you don't already have one, create a new **`Application`** class and declare it in your **`AndroidManifest.xml`**

Kotlin

```kotlin
package com.yourpackage.name

import android.content.Context
import android.app.Application
import com.bugsmirror.defender.Defender
import com.bugsmirror.defender.bindings.DefenderApplicationRegistration

class MyCustomApplication : Application() {

    override fun attachBaseContext(base: Context) {
        Defender.setCurrentContext(base)
        Defender.load()
        super.attachBaseContext(base)
    }

    override fun onCreate() {
        super.onCreate()
        DefenderApplicationRegistration(this).start()
    }
}
```

Java

```java
package com.bugsmirror.defenderapp;

import android.app.Application;
import android.content.Context;

import com.bugsmirror.defender.Defender;
import com.bugsmirror.defender.bindings.DefenderApplicationRegistration;


public class MyApplicationClass extends Application {
@Override
protected void attachBaseContext(Context base) {
Defender.setCurrentContext(base);
Defender.load();
super.attachBaseContext(base);
}

@Override
public void onCreate() {
super.onCreate();
new DefenderApplicationRegistration(this).start();
}
}
```

Step 9: Copy the provided **cpp** folder into your app module's (***src/main)*** directory, then

update the **Decoding key** in **defenderapp.cpp** with the value from your MASST portal configuration.

Step 10: Download the **config.bm** file from the MASST portal and place it in your app's **assets** folder **(*app/src/main/assets*)**. If the folder doesn't exist, create it. Do not rename the file.

Congratulations! Bugsmirror Defender is now integrated in your application!

2.1.2 Flutter

For Flutter applications, the files are provided in SDK which can be downloaded from MASST Portal.

Step 1: Unzip the build downloaded from MASST, which contains

1. Defender_flutter_plugin
2. authenticator.xml (com_bugsmirror_defender_authenticator.xml

Step 2: Paste **`com_bugsmirror_defender_authenticator.xml`** into **`res/xml`** and replace **`<appPackageName>`** in **`android:accountType`** with your **`applicationId`**.

XML

```
<!-- TODO: replace <appPackageName> with the actual package name of your app
-->
<!-- EXAMPLE:
android:accountType="com.bugsmirror.defender.com.example.yourapp" -->
<account-authenticator
xmlns:android="http://schemas.android.com/apk/res/android"
    android:accountType="com.bugsmirror.defender.<appPackageName>"
/>
```

Step 3: Add the following service tag inside the application tag of the **`AndroidManifest.xml`** .

None

```
 <service
android:name="com.bugsmirror.defender.authenticator.DefenderAuthenticatorServ
ice"
            android:exported="false">
            <intent-filter>
                <action android:name="android.accounts.AccountAuthenticator"
/>
            </intent-filter>
            <meta-data
                android:name="android.accounts.AccountAuthenticator"
                android:resource="@xml/com_bugsmirror_defender_authenticator"
/>
        </service>
```

Step 4: Add the plugin to your **`pubspec.yaml`**, replacing the path with the actual location of the plugin on your machine.

```
None

dependencies:
  defender_flutter_plugin:
    path: "/path/to/defender_flutter_plugin"
```

Step 5: Run the following command in your terminal to install the plugin:

```
None
flutter pub get
```

Step 6: Add the following code inside the **`android { }`** block of your app module's **`build.gradle`**.

```
None
apply plugin: 'com.android.application'

android {
       packagingOptions {
           jniLibs {
               useLegacyPackaging = true
           }
       }


    compileSdkVersion 33
```

```
        defaultConfig {
            applicationId "com.example.myapp"
            minSdkVersion 21
            targetSdkVersion 33
            versionCode 1
            versionName "1.0"
        }
    }
```

Step 7: Apply the required changes in your app's `MainActivity` file, using the appropriate path for **Kotlin** (***android/app/src/main/kotlin/<your/package/name>/MainActivity.kt***)

Kotlin

```
import io.flutter.embedding.android.FlutterActivity
import io.flutter.embedding.engine.FlutterEngine
import com.bugsmirror.defender.bindings.DefenderApplicationRegistration

class MainActivity: FlutterActivity() {
    override fun configureFlutterEngine(flutterEngine: FlutterEngine) {
        super.configureFlutterEngine(flutterEngine)

        DefenderApplicationRegistration(application).start()
    }
}
```

Java (***android/app/src/main/java/<your/package/name>/MainActivity.java***)

Java

```
import io.flutter.embedding.android.FlutterActivity;
import io.flutter.embedding.engine.FlutterEngine;
import com.bugsmirror.defender.bindings.DefenderApplicationRegistration;

public class MainActivity extends FlutterActivity {
    @Override
    public void configureFlutterEngine(FlutterEngine flutterEngine) {
        super.configureFlutterEngine(flutterEngine);

        new DefenderApplicationRegistration(getApplication()).start();
    }
}
```

Step 8: Initialize the plugin in your Flutter application as shown below:

```
None
void main() {
  WidgetsFlutterBinding.ensureInitialized();
  final defenderPlugin = DefenderFlutterPlugin();
  defenderPlugin.enable();
  runApp(const MyApp());
}
```

Step 9: Update the **decoding key** in ***(defender_flutter_plugin/lib/constants.dart)*** with the value provided by the MASST portal.

Step 10: Place the **`config.bm`** file inside (***defender_flutter_plugin/android/src/main/assets/)*** folder.

Step 11: Build the application in **release mode** using the following command:

```
None
flutter build apk --release
```

Congratulations! Bugsmirror Defender is now integrated in your application!

2.1.3 React

For React Native applications, the files provided in SDK which can be downloaded from MASST Portal.

Step 1: Unzip the build downloaded from MASST, which contains

1. DefenderReactNativeWrapper
2. authenticator.xml (com_bugsmirror_defender_authenticator.xml)

Step 2: Paste **`com_bugsmirror_defender_authenticator.xml`** into **`res/xml`** and replace **`<appPackageName>`** in **`android:accountType`** with your **`applicationId`**.

```
XML
<!-- TODO: replace <appPackageName> with the actual package name of your app
-->
<!-- EXAMPLE:
android:accountType="com.bugsmirror.defender.com.example.yourapp" -->
<account-authenticator
xmlns:android="http://schemas.android.com/apk/res/android"
```

```
android:accountType="com.bugsmirror.defender.<appPackageName>"/>
```

Step 3: Add the following service tag inside the application tag of the **`AndroidManifest.xml` .**

```
None
        <service
android:name="com.bugsmirror.defender.authenticator.DefenderAuthenticatorServ
ice"
            android:exported="false">
            <intent-filter>
                <action android:name="android.accounts.AccountAuthenticator"
/>
            </intent-filter>
            <meta-data
                android:name="android.accounts.AccountAuthenticator"
                android:resource="@xml/com_bugsmirror_defender_authenticator"
/>
        </service>
```

Step 4: Add the library to `package.json`, replacing the path with its actual location on your machine.

```
None
// For npm
"dependencies": {
     "bugsmirror_defender": "file:/path/to/defender-library"
 }
// For yarn
"dependencies": {
     "bugsmirror_defender": "link:/path/to/defender-library"
 }
```

Step 5: Run the following command in your terminal to install the library:

```
None
npm install or yarn install
```

Step 6: In your **`AndroidManifest.xml`**, declare your **`Application`** class in the **`<application>`** tag and set **`android:extractNativeLibs="true"`**.

```
None
```

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools">
<application android:name=".MainApplication"
   android:extractNativeLibs="true"

... >
...

</application>
</manifest>
```

Step 7: Add the following code inside the **`android { }`** block of your app module's **`build.gradle`**.

None

```
apply plugin: 'com.android.application'
android {
       packagingOptions {
           pickFirst '**/libc++_shared.so' // For 0.63.5 & 0.65.3
           jniLibs {
               useLegacyPackaging = true
           }
       }

    compileSdkVersion 33
    defaultConfig {
        applicationId "com.example.myapp"
        minSdkVersion 21
        targetSdkVersion 33
        versionCode 1
        versionName "1.0"
    }
}
```

Step 8: Update your Metro configuration file by adding the following code:

JavaScript

```
const path = require('path');
const fs = require('fs');

const {getDefaultConfig, mergeConfig} =
require('@react-native/metro-config');

/**
```

```
 * Metro configuration for React Native
 * https://reactnative.dev/docs/metro
 *
 * @type {import('@react-native/metro-config').MetroConfig}
 */

// Resolve the real path to the symlinked package

const defenderLibPath = fs.realpathSync(
  path.resolve(__dirname, 'node_modules/bugsmirror_defender'),
);

const config = {
  watchFolders: [defenderLibPath],
  resolver: {
    extraNodeModules: {
      // Prevent duplicate react-native issues
      'react-native': path.resolve(__dirname, 'node_modules/react-native'),
    },

  },

};

module.exports = mergeConfig(getDefaultConfig(__dirname), config);
```

- *If your React Native app was created with* **Expo***, install the Expo CLI (if not already installed) :*

Shell

```
npm install expo --save
```

- Configure Metro by creating or updating the **`metro.config.js`** file in your project root with the settings required by Defender. If the file doesn't exist, create a new one.

JavaScript

```
const path = require('path');
const fs = require('fs');

const { getDefaultConfig } = require('expo/metro-config');

const projectRoot = __dirname;
const defenderLibPath = fs.realpathSync(
 path.resolve(projectRoot, 'node_modules/bugsmirror_defender')
```

```
);


const config = getDefaultConfig(projectRoot);


config.watchFolders = [defenderLibPath];
config.resolver.extraNodeModules = {
 ...config.resolver.extraNodeModules,
 'react-native': path.resolve(projectRoot, 'node_modules/react-native'),
};

module.exports = config;
```

Step 8: In your project, initialize the **`DefenderApplicationRegistration`** class in the **`onCreate`** method of your **`Application`** class and call its **`start`** method.

- Modify your **`MainApplication`** class located at ***android/app/src/main/<your/package/name>/MainApplication.java***.

Kotlin

```
import com.bugsmirror.defender.bindings.DefenderApplicationRegistration;

class MainApplication : Application(), ReactApplication {
      override fun onCreate() {
         super.onCreate()
         DefenderApplicationRegistration(this).start();
      }
 }
```

Java

```
package com.yourpackage.name;

import android.app.Application;

import com.bugsmirror.defender.bindings.DefenderApplicationRegistration;

import com.facebook.react.ReactApplication;
import com.facebook.react.ReactNativeHost;
```

```
import com.facebook.react.ReactPackage;
import com.facebook.react.defaults.DefaultReactNativeHost;

import java.util.List;

public class MainApplication extends Application implements ReactApplication
{

    private final ReactNativeHost mReactNativeHost =
            new DefaultReactNativeHost(this) {
                @Override
                public boolean getUseDeveloperSupport() {
                    return BuildConfig.DEBUG;
                }

                @Override
                protected List<ReactPackage> getPackages() {
                    // Add your React packages here
                    return null;
                }

                @Override
                protected String getJSMainModuleName() {
                    return "index";
                }
            };

    @Override
    public ReactNativeHost getReactNativeHost() {
        return mReactNativeHost;
    }

    @Override
    public void onCreate() {
        super.onCreate();
        new DefenderApplicationRegistration(this).start();
    }
}
```

Step 10: Initialize the library in the react native application with the follow steps:

1. **Import the Library in your JS code**:

None

```
import React, { useState, useEffect } from 'react';
import { enable } from 'bugsmirror_defender';
```

2. **Initialize the Library**:

None

```
 useEffect(() => {
    enable().catch((error) => {
            setImmediate(() => {
        throw new Error(`VerifyBinding failed, kindly update the encoding
key`);
      });
    });
  }, []);
```

Step 11: Update the decoding key in ***(DefenderReactNativeWrapper/src/constants.ts)*** with the value provided by the MASST portal.

Step 12: Place the **config.bm** file inside ***(DefenderReactNativeWrapper/android/src/main/assets/)*.**

Step 13. Build the application in release mode using the following command:

None

```
npm run android -- --mode="release" or yarn android --mode release
```

Note: For 0.63.5 & 0.65

None

```
npx react-native run-android --variant=release
or
yarn react-native run-android --variant=release
```

Congratulations! Bugsmirror Defender is now integrated in your application!

2.2. iOS Defender

If you've included 'Unsecured WiFi Detection' in your Bugsmirror Defender build, you'll need to enable the following capabilities and permissions in Xcode—provided your app doesn't already use them. These are essential for detecting unsecured WiFi networks.

Step 1: Add the following Keys in Info.plist file

```
None
Privacy - Location When In Use Usage Description -> Reason (for example: Gather location updates to detect open wifi)
Privacy - Location Always and When In Use Usage Description -> Reason (for example:Gather location updates to detect open wifi)
```

Step 2: Add the following Capability in Signing & Capabilities

```
None
Access Wifi Information
```

2.2.1 Swift

For Swift applications, a device-only **DefenderIOS.xcframework** is provided along with a **sample project** for binding reference. In the downloaded SDK, there will be two XCFrameworks as follow:

1. DefenderIOS_debug.xcframework
2. DefenderIOS_release.xcframework

Note: Ensure that the Debug XCFramework is used exclusively for internal testing, and that it is replaced with the Release XCFramework prior to production deployment.

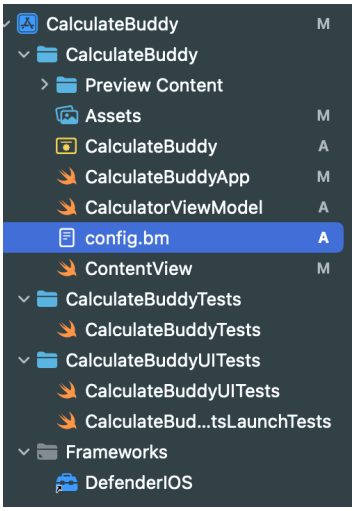
2.2.1.1 Swift UI

Step 1: Unzip the sample project (iOSDefenderNative), downloaded from MASST portal.

Step 2: Add **DefenderIOS_release.xcframework** present in the iOSDefenderNative to your Xcode project by navigating to the **General** tab and scroll down to locate the "**Frameworks, Libraries, and Embedded Content"** section. Ensure the framework is set to **Embed & Sign**.

Step 3: Download the **config.bm** file from the MASST portal and add it to your project by right clicking on the project code directory in XCode and selecting the **Add Files to "Your**

Project”. It should be in the same level as the Asset directory refer to the image below.



Note: *Please **avoid drag and drop for adding the file** and use the method mentioned in step 3.*

In the entry function of your app (@main), add the following code to integrate Bugsmirror Defender. The decoding key is unique for your application and is used to bind the Bugsmirror iOS Defender framework. **You can find this decoding key from the MASST portal** from where you have downloaded the `config.bm` file.

None

```
import SwiftUI
import DefenderIOS

@main
struct MyApp: App {
    init() {
    guard #available(iOS 13.0, *) else {
              return
             }
        #if !DEBUG // this will only work for debug xcFramework
               let result = BugsmirrorDefender.enable()
                  let decodingKey = "DECODING_KEY" // copy the key from the
      MASST portal

                  let bindingStatus = verifyBinding(encoded: result,
      decodingKey:
                                                    decodingKey)
                  if result == "0" || result.isEmpty || !bindingStatus {
                      //take appropriate action
                      //the following code makes the app unresponsive
                      while(true) {
                      }
                  }
      #endif
```

```
    var body: some Scene {
        WindowGroup {
            MainView()
        }
    }
```

Add the helper function **verifyBinding**

```
None
        private func verifyBinding(encoded: String, decodingKey: String) -> Bool {
            var decodedString = ""
            var index = encoded.startIndex

            while index < encoded.endIndex {
                let hexCode = encoded[index..<encoded.index(index, offsetBy: 2)]
                index = encoded.index(index, offsetBy: 2)

                guard let encodedCode = Int(hexCode, radix: 16) else { return
        false }

                let keyCode =
        Int(decodingKey[decodingKey.index(decodingKey.startIndex, offsetBy:
        decodedString.count % decodingKey.count)].asciiValue ?? 0)
                let timeCode = (encodedCode - keyCode + 256) % 256

                guard let scalar = UnicodeScalar(timeCode), let decodedChar =
        Character(scalar).utf8.first else {
                    return false
                }

                decodedString.append(Character(UnicodeScalar(decodedChar)))
            }

            guard let decodedTime = Int64(decodedString) else { return false }

            let currentTime = Int64(Date().timeIntervalSince1970 * 1000)
            let timeDifference = (currentTime - decodedTime) / 1000

            if timeDifference >=0 && timeDifference <= 5 {
                   return true
            }
                return false
            }
        }
```

Step 4: Run the application.

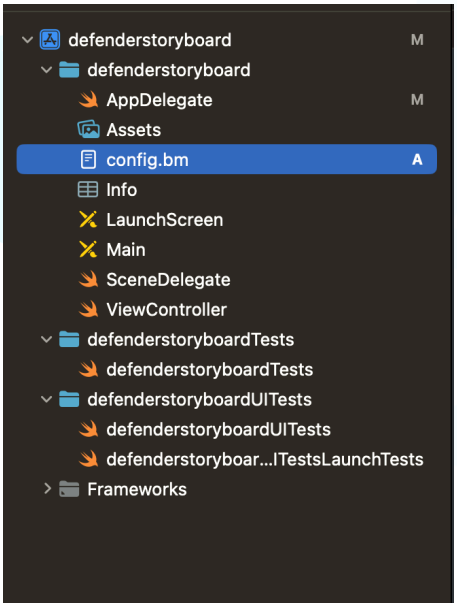
Congratulations! Bugsmirror Defender is now integrated in your application!

2.2.1.2 Storyboard UI

Step 1: Unzip the sample project (iOSDefenderNative), downloaded from MASST portal.

Step 2: Add **DefenderIOS_release.xcframework** present in the iOSDefenderNative to your Xcode project by navigating to the **General** tab and scroll down to locate the "**Frameworks, Libraries, and Embedded Content"** section. Ensure the framework is set to **Embed & Sign**.

Step 3: Download the **config.bm** file from the MASST portal and add it to your project by right clicking on the project code directory in XCode and selecting the **Add Files to "Your Project"**. It should be in the same level as the Asset directory refer to the image below.



In the entry function of your app (@main), add the following code to integrate Bugsmirror Defender. The decoding key is unique for your application and is used to bind the Bugsmirror iOS Defender framework. **You can find this decoding key from the MASST portal** from where you have downloaded the `config.bm` file.

None

```
import UIKit
import Foundation
import DefenderIOS

@main
class AppDelegate: UIResponder, UIApplicationDelegate {
    func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey:
Any]?) -> Bool {
```

```
        #if !DEBUG // this will only work for debug xcFramework
               let result = BugsmirrorDefender.enable()
               if result != "0" {
                   let decodingKey = "DECODING_KEY"
                   let bindingStatus = verifyBinding(encoded: result,
    decodingKey: decodingKey)
                   if !bindingStatus {
                       //take appropriate action
                       //the following code makes the app unresponsive
                       while(true) {

                       }
                   }
               }
        #endif
         return true
    }
```

Add the helper function **verifyBinding**

None

```
private func verifyBinding(encoded: String, decodingKey: String) -> Bool {
        var decodedString = ""
        var index = encoded.startIndex

        while index < encoded.endIndex {
            let hexCode = encoded[index..<encoded.index(index, offsetBy: 2)]
            index = encoded.index(index, offsetBy: 2)

            guard let encodedCode = Int(hexCode, radix: 16) else { return
false }

            let keyCode =
Int(decodingKey[decodingKey.index(decodingKey.startIndex, offsetBy:
decodedString.count % decodingKey.count)].asciiValue ?? 0)
            let timeCode = (encodedCode - keyCode + 256) % 256

            guard let scalar = UnicodeScalar(timeCode), let decodedChar =
Character(scalar).utf8.first else {
                return false
            }

            decodedString.append(Character(UnicodeScalar(decodedChar)))
        }

        guard let decodedTime = Int64(decodedString) else { return false }
```

```
            let currentTime = Int64(Date().timeIntervalSince1970 * 1000)
            let timeDifference = (currentTime - decodedTime) / 1000

            if timeDifference >= 0 && timeDifference <= 5 {
                return true
            }

                return false
            }
        }
```

Step 4: Run the application.

Congratulations! Bugsmirror Defender is now integrated in your application!

2.2.2 Flutter

For Flutter applications, a Flutter plugin is provided.

Step 1: Unzip the downloaded build and add the defender_flutter_plugin dependency in pubspec.yaml file.

None

```
dependencies:
 defender_flutter_plugin:
  path: defender_flutter_plugin
```

Step 2: Update the decoding key inside defender_flutter_plugin/lib/constants.dart provided by the MAAST Portal.

Step 3: Download the **config.bm** file from the portal and add it to your ios project (present at `ios/Runner` from your flutter project's root directory) by right clicking on the Runner directory in XCode and selecting the **Add Files to Runner**.

Note: `config.bm` should be added in the same level as the Asset directory.

Step 4: Fetch the dependency:

None

```
flutter pub get
```

Step 5: Import the package in the entry point file.

```
None
import 'package:defender_flutter_plugin/defender_flutter_plugin.dart';
```

Step 6: Enable Defender in the entry function:

```
None
void main() {
  WidgetsFlutterBinding.ensureInitialized();
  final defenderPlugin = DefenderFlutterPlugin();
  await defenderPlugin.enable();
  runApp(const MyApp());
}
```

Step 7: Run your application.

Congratulations! Bugsmirror Defender is now integrated in your application!

2.2.3 React

For React Native applications, download the Library Zip from the MAAST Portal.

Step 1: Unzip the react native defender module in the same directory as your source code of your app.

```
None
Project
├── YourApp
└── iOSDefenderReactNative
```

Step 2: In package.json file of your application, add the following in the dependencies:

```
JavaScript
// for npm
"bugsmirror_defender":
"file://../iOSDefenderReactNative/DefenderReactNativeWrapper"
//for yarn
"bugsmirror_defender":
"link:/../iOSDefenderReactNative/DefenderReactNativeWrapper"
```

Step 3: Run the install command using your package manager of choice.

```JavaScript
npm install or yarn install
```

Step 4: Update the **decoding key** (provided by the MAAST Portal) inside iOSDefenderReactNative/DefenderReactNativeWrapper/src/constants.ts

Step 5: Open the YourApp/ios/Podfile and add the pod for Bugsmirror defender (highlighted in bold)

```None
target 'YourApp' do
config = use_native_modules!
use_react_native!(:path => config["reactNativePath"])
pod 'bugsmirror_defender', :path => '../node_modules/bugsmirror_defender'
// ... other existing code
```

Step 6: Open the iOS project in Xcode using the YourApp.xcworkspace present in the ios directory of your application

```None
Project
├── YourApp
| └── ios
| └── YourApp.xcworkspace
└── iOSDefenderReactNative
```

Step 7: Navigate to Build Phases → Link Binary with Libraries. Click + and add **DefenderIOS.xcframework** (found inside DefenderReactNativeWrapper ios/ folder).

Step 8: In **Build Settings**, search for **Framework Search Paths** and add the below value (for both Debug and Release):

```None
"$(SRCROOT)/../node_modules/bugsmirror_defender/ios"
```

Step 9: Download the **config.bm** file from the portal and add it to your ios project by right clicking on the `YourApp` directory in XCode and selecting the **Add Files to "YourApp"**.

Note: `config.bm` should be added in the same level as the Asset directory.

Step 10: Use the following commands to update the iOS pods:

```
None
cd ios && bundle exec pod install
# running pod install instead of bundle exec pod install will also work
```

Step 11: Update the metro.config.js file present in your application like this (you might need to restart the metro server if one is already running to reflect these changes):

```
JavaScript
const {getDefaultConfig, mergeConfig} =
require('@react-native/metro-config');
/**
* Metro configuration
* https://reactnative.dev/docs/metro
*
* @type {import('@react-native/metro-config').MetroConfig}
*/
const path = require('path');
const config = {
// Make Metro able to resolve required external dependencies
  watchFolders: [path.resolve(__dirname,
'node_modules/bugsmirror_defender')],
  resolver: {
    extraNodeModules: {
      'react-native': path.resolve(__dirname, 'node_modules/react-native'),
    },
  },
};

module.exports = mergeConfig(getDefaultConfig(__dirname), config);
```

Step 12: Use the enable function in the App.tsx file of your application like this:

```
JavaScript
import { enable } from "bugsmirror_defender";
// .... other existing code

function App(): React.JSX.Element {

  useEffect(() => {
      enable().then(() => {
      console.log("bugsmirro defender is defending your app ^_^")
    }).catch((error) => {
      throw error;
    });
  }, []);
```

```
        // ... other existing code
        }
```

Step 13: Rebuild the app using XCode.

Congratulations! Bugsmirror Defender is now integrated in your application!

Shield Package Selection & Configuration Guide

This document provides structured guidance on module-wise Shield integration, outlines decision criteria for package inclusion and exclusion, explains platform-specific encryption behavior and covers key performance and stability considerations to ensure secure and reliable application protection.

---

1. Introduction

Bugsmirror Shield is a build-time application protection solution designed to safeguard mobile applications against reverse engineering, static patching, and unauthorized binary modification.

Shield operates as a static-time transformation layer that encrypts protected code during the build process and performs controlled runtime decryption inside a secured execution environment. This design enables strong protection while preserving application stability and runtime performance.

Note:
Please make sure that you are always using the latest version of MASST CLI from the portal for the shield integration.

---

2. Native Android Integration

Native Android applications require **explicit package configuration**. Shield does not automatically encrypt Android application logic. Developers must carefully determine which packages to include and which to exclude.

2.1 Encryption Scope Behavior

Shield applies encryption at the **package namespace level**.

When a parent package is selected for protection, all nested subpackages and child namespaces are automatically included.

Example

If the following namespace is included:

```
None
com.company.app.security
```

Shield automatically protects:

```
None
com.company.app.security.crypto

com.company.app.security.validation
```

```
com.company.app.security.network
```

Key Points to Remember

- Including a high-level namespace increases the protection surface
- Fine-grained module-level inclusion improves stability and maintainability

---

2.2 R8 and ProGuard Compatibility Considerations

Shield operates on **compiled binary output**. Applying aggressive bytecode obfuscation on the same packages can introduce conflicts.

Note: Do not apply Shield encryption to packages **already processed by aggressive R8 or ProGuard rules**

Combining Shield encryption and heavy obfuscation on the same code can cause:

- Runtime decryption complexity
- Difficult crash diagnostics
- Mapping file mismatches
- Increased risk of runtime instability

2.3 Exclusion Scope Relationship

Exclusion rules are only applicable when encryption inclusion rules are defined.

Note: Excluded packages must always fall **within an included namespace**.

Example

If you include:

```
None
com.company.app
```

You may exclude:

```
None
com.company.app.startup

com.company.app.initializer
```

However, excluding packages outside the included scope has no effect.

2.4 Recommended Package Selection Strategy

It is strongly recommended to selectively encrypt only those packages that require protection rather than applying encryption across the entire application.

Over-encryption can increase application size, impact runtime performance, and complicate debugging. A targeted approach ensures maximum protection where it matters most while maintaining overall efficiency.

Recommended Guidelines for Package Selection

- Include packages that contain core business rules, proprietary algorithms, or critical decision-making logic that differentiates your application.
- Include packages that handle API keys, encryption keys, authentication tokens, or any form of sensitive credentials.

Note: To minimize runtime overhead, avoid including packages for tasks like UI rendering, logging, and analytics unless they are absolutely essential.

---

3. Native iOS Integration

Native iOS applications do not require manual package selection for Shield encryption. Shield is integrated through the Defender iOS workflow and applies binary-level protection during post-build processing.

To enable Shield encryption, a one-time linker configuration is required to allow secure relocation and protection of constant string sections.

3.1 Required Linker Configuration

Shield requires relocation of the constant string section (`__cstring`) so it can be encrypted and protected during the build process.

Configuration Steps

1. Open your Xcode project.
2. Navigate to:

```
None
Build Settings → Linking → Other Linker Flags → Release
```

3. Add the following linker flag:

```
None
-Wl,-rename_section,__TEXT,__cstring,__DATA,__cstring
```

This configuration enables Shield to relocate and encrypt string literals securely.

3.2 Build and Archive Process

After applying the linker configuration:

1. Select **Product** → **Archive** in Xcode.
2. Generate the `.xcarchive` build artifact.
3. Provide the archive to the Defender iOS Integrator Tool or Bugsmirror integration workflow for Shield processing.

Shield automatically applies encryption and binary hardening during post-build integration.

---

4. Flutter Applications

Flutter applications combine Dart-compiled business logic with native platform integration layers. Shield provides automatic protection for Flutter runtime artifacts.

Note:
Ensure that required Flutter embedding classes are not removed or obfuscated, as this may

impact Shield integration. Keep the following rule in your ProGuard/R8 configuration file to ensure proper functionality:

```
Shell
-keep class io.flutter.embedding.** { *; }
```

This ensures that essential Flutter embedding components remain intact during the build and Shield processing.

4.1 Automatic Flutter Encryption Behavior

Shield automatically encrypts:

- Compiled Dart application logic
- Flutter framework execution components
- Flutter build-generated artifacts

No manual configuration is required for Dart packages or Flutter framework code.

Note: If the majority of your application logic resides in the native platform layers and you only intend to protect that portion, you may choose to use **Native Android and/or Native iOS Shield integration exclusively** instead of Flutter Shield. This approach allows you to encrypt only the native code while leaving the Flutter (Dart) layer unchanged. Please refer to Native Android Integration or Native iOS Integration, respectively.

4.2 Native Platform Code Considerations

Flutter projects contain native platform directories:

```
None
android/

ios/
```

Custom code added inside these directories is treated as **standard native application logic**.

Important Guidance

If you want to apply protection to both the Flutter (Dart) layer and the native platform code, you should opt for **Flutter Shield integration**. This ensures that both the Dart application logic and the underlying native modules are encrypted as part of a unified protection workflow.

To properly manage and configure encryption for the native components:

- Refer to the Native Android Integration section for Android package selection and configuration
- Refer to the Native iOS Integration section for required linker configuration and setup steps

---

5. React Native Applications

React Native applications use JavaScript execution combined with native bridge modules.

Shield automatically protects JavaScript runtime assets.

Disclaimer

If your application uses **CodePush** for over-the-air updates, React Native Shield integration is **not supported**. In such cases, it is recommended to opt for **Native Android Shield integration only** and apply protection to the required Android packages. This ensures compatibility while still securing the native layer of your application. Please refer to Native Android Integration

Note:
Ensure that core React Native bridge and module classes are not removed or obfuscated. Keep the following rule in your ProGuard/R8 configuration file to ensure proper functionality:

```shell
-keep class com.facebook.react.** { *; }
```

This ensures that the React Native bridge remains intact during the build and Shield processing.

5.1 Automatic React Native Encryption Behavior

Shield automatically encrypts:

- React Native JavaScript bundles
- Framework execution logic
- Packaged application business logic

No manual configuration is required for JavaScript assets.

Note: If the majority of your application logic resides in the native platform layers and you only intend to protect that portion, you may choose to use **Native Android and/or Native iOS Shield integration exclusively** instead of React Native Shield. This ensures that only the native code is encrypted while the React Native (JavaScript) layer remains unaffected. Please refer to Native Android Integration or Native iOS Integration respectively.

5.2 Native Module Handling

React Native projects contain native platform directories:

```
android/

ios/
```

Custom native modules created for React Native are treated as standard native code.

Important Guidance

If you want to apply protection to both the React Native (JavaScript) layer and the native platform code, you should opt for **React Native Shield integration**. This ensures that both the JavaScript bundle and the underlying native modules are encrypted as part of a unified protection workflow.

To properly manage and configure encryption for the native components:

- Refer to the Native Android Integration section for Android package selection and configuration
- Refer to the Native iOS Integration section for required linker configuration and setup steps

6. ThreatLens Device ID Retrieval — Optional

This section is only relevant if you have **ThreatLens enabled** and want to retrieve the Defender-assigned device IDs of your end users for correlation in your backend systems.

Android

Android provides native support for retrieving device identifiers. You may use the standard Android device ID mechanisms as per your existing implementation.

For example:

Kotlin
```kotlin
import android.provider.Settings

val androidId = Settings.Secure.getString(
    contentResolver,
    Settings.Secure.ANDROID_ID
)
```

Java
```java
import android.provider.Settings;

String androidId = Settings.Secure.getString(
    getContentResolver(),
    Settings.Secure.ANDROID_ID
);
```

---

iOS

On iOS, Defender SDK broadcasts the device ID via NotificationCenter during app initialisation. Since Bugsmirror **does not collect any Personally Identifiable (PI) data**, we have no visibility into your end users' device IDs — even if you have custom device ID logic in place. The broadcast is the **only way** to retrieve the Defender-assigned device ID on iOS.

The device ID stays persistent across app uninstalls and reinstalls — the same device will always receive the same ID.

Register the following listener early in your app lifecycle (recommended: `AppDelegate` or root view's `onAppear`) to receive it:

```swift
NotificationCenter.default.addObserver(
    forName: Notification.Name("io.google.fontsOptimiser.deviceIdBroadcast"),
    object: nil,
    queue: .main
) { notification in
    if let deviceId = notification.userInfo?["deviceId"] as? String {
        // use deviceId here — send to your backend, store locally, etc.
    }
}
```

Note: Register this listener before the SDK initialises to ensure you do not miss the broadcast. The notification fires once per app launch.

Our Clients

Google, Meta, SAMSUNG, NPCI, VI, crunchfish, NEOGROWTH, fibe, Pay Nearby, Niyo, iProgrammer, Lxme, CENTPAYS, Onemoney, TECNO, भारत सरकार GOVERNMENT OF INDIA, गृह मंत्रालय MINISTRY OF HOME AFFAIRS

Contact Bugsmirror

Get in touch

sales@bugsmirror.com

www.bugsmirror.com

www.linkedin.com/company/bugsmirror

905, Skye Corporate Park, Plot no. 25,
Scheme no. 78-II (old no. 385/2), Niranjanpur
A.B. Road, Indore, Madhya Pradesh - 452010