Girgit

Migration Guide

verifone

www.verifone.com

Girgit Migration Guide
© 2025 Verifone, Inc.

Comments? Please e-mail all comments on this document to your local Verifone Support Team.

Content

Preface

Purpose

This document outlines the steps for migrating from Verifone X990 device to Neo devices, enabling you to use your existing application with minimal code modifications.

The document will be revised and updated whenever new functionality is developed in a new version of the application.

Objective and Scope

The objective of this guide is to provide the essential steps and permissions necessary to leverage Girgit effectively, thus ensuring that the existing application remain robust and compatible with Neo devices. This document also highlights the potential issues that may arise during the migration process, along with details on how to resolve them.

The scope of this guide includes:

- The migration strategy and system requirements necessary to migrate to Neo devices.
- Changes made to the existing application, APIs, and UI.
- Permissions (mandatory and optional) and privileged access.
- EMV and security configuration details.
- Issues encountered during migration and best practices to resolve them.
- Risks identified and their mitigation strategies.
- Post-migration validation and support.

Out of Scope

The following is not included in the migration solution:

- TMS

Audience

This guide is intended for the Verifone customers, developers, and technical teams who are migrating from Verifone X990 to Verifone Neo devices by using Girgit middleware SDK.

NOTE For customers who are willing to rewrite the application during the transition to Neo devices, we recommend utilizing Payment SDK-SDI instead of Girgit middleware SDK.

Definitions and Abbreviations

The following terms are used in this document:

Abbreviation	Definition
SDK	Software Development Kit
PSDK	Payment Software Development Kit
VRK	VeriShield Remote Key
VHQ	Verifone Headquarters
SDI	Secure Data Interface
EMV	Europay, MasterCard, and Visa
UI	User Interface
AIDL	Android Interface Definition Language
PCI	Payment Card Industry
PIN	Personal Identification Number
P2PE	Point-to-Point Encryption

Related Documentation

To learn more Girgit application, refer to the following documents:

- Girgit Programmers Guide
- X990 Quick Installation Guide: VPN DOC550-004-EN-A
- Android 13 Migration for the older applications (https://developer.android.com/about/versions/13/behavior-changes-13)

Revision History

Date	Version Number	Description
17-02-2025	1.0.0	First Release

1. Introduction

Girgit service application allows the seamless transition of applications from Verifone X990 devices to Verifone Neo devices. It is a middleware SDK (Software Development Kit) developed to streamline the migration process.

Girgit acts as a translation layer that efficiently facilitates the porting of Android solutions with minimal application modifications. It abstracts all SDI API calls and replaces them with VFI service API (X990-SDK). The existing features of X990, which are VF service and system service, are replaced with Girgit system service and Girgit device service.

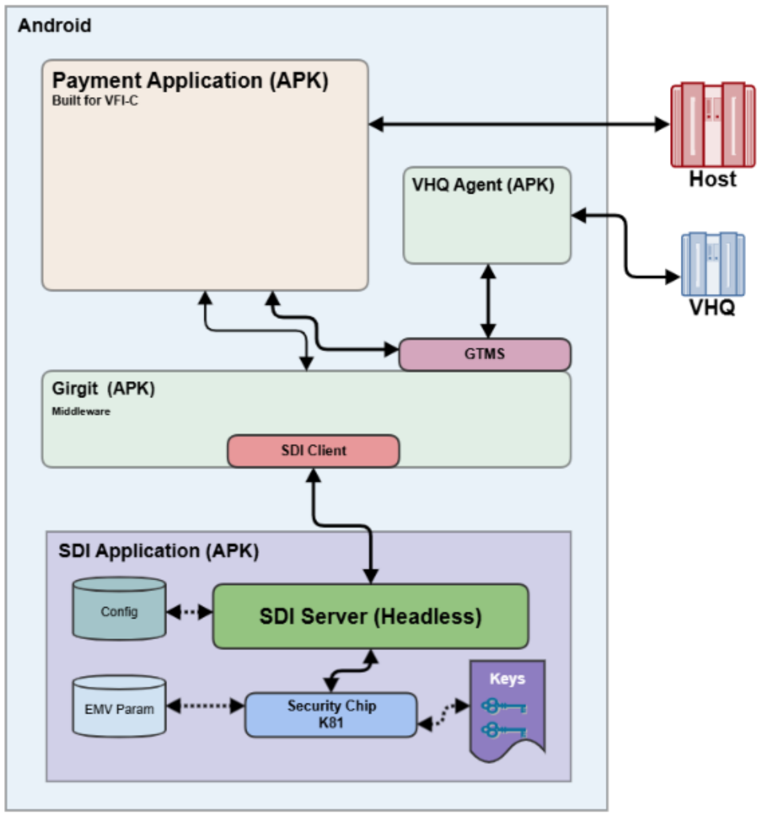

Figure 1: High-level architecture diagram

1.1 Key Features and Functionalities

The key features and functionalities offered by Girgit include:

- Support for Android 13: Girgit enables applications to fully utilize the capabilities of Android 13, ensuring that they remain compatible with the latest system enhancements, user interface improvements, and security protocols. This support allows developers to integrate advanced features and functionalities available in Android 13, providing users with a seamless and up-to-date experience on Verifone Neo devices.
- Increased Memory Capacity: Girgit supports more memory, which facilitates the handling of larger and more complex applications, improving overall efficiency and performance on Verifone Neo devices.

1.2 Introduction to Verifone Components for X990 Customers

This section introduces different Verifone components including VeriShield Remote Key (VRK), EMV Configuration, Security Configuration, Verifone Headquarters (VHQ), and Secure Data Interface (SDI).

1.2.1 Verifone Headquarters (VHQ)

Verifone Headquarters (VHQ) is Verifone's next-generation terminal management software suite that eliminates operational burdens. It empowers the user with management tools for data collection, downloads, remote diagnostics, and content management. It is a platform that enables software installation, updating of OS, diagnostics, and configuration of applications on devices remotely, where different application parameters can be configured, downloaded, and updated in several devices simultaneously. VHQ leverages a wide range of experience to provide best-of-breed estate management capabilities across a variety of platforms and supports growth-oriented, customer-friendly deployment models.

1.2.2 VeriShield Remote Key (VRK)

VRK is a PCI PIN and PCI P2PE certified remote key loading system that facilitates remote key management for encryption keys on payment devices. It streamlines the process of securely deploying and updating encryption keys without requiring physical access to each device.

The VRK solution uses end to end encryption and mutually authenticated end points. The ends are: The VRK HSM (sometimes called the "Key Distribution Host" or the KDH) and the Verifone device (sometimes called the “Key Receiving Device” or the KRD). Intermediate systems are never exposed to sensitive data and have no ability to decrypt it.

1.2.3 Secure Data Interface (SDI)

Secure Data Interface (SDI) provides a secure communication channel between payment devices and transaction processing systems. The payment applications interact with SDI to facilitate access to card reading, EMV, and security functions such as PIN support, key management, access control, card checks etc. It supports card data protection where developers can protect sensitive card holder data inside SDI so that applications will not be exposed to this data. On Android solutions, SDI provides access to the functions of the secure processor, i.e. SDI runs as a system component with privileged access to security domain. It cannot be by-passed when accessing secure domain functions.

1.2.4 EMV Configuration

The Verifone EMV configuration framework supports the payment applications and enable them to configure and conduct contact and contactless transactions. The EMV framework assists in the implementation of different transaction flows and is connected to the application via APIs. EMV configuration involves the initialization and management of transactions, ensuring they meet industry requirements for security and functionality. This process includes configuring terminal parameters and loading the necessary keys to facilitate smooth interactions with EMV-compliant cards.

The configuration data of the application is stored in the XML files including emv-allowed.xml, emv-desired.xml, and cardranges.json.

1.2.5 Security Configuration

The security configurations are paramount to ensuring safe and compliant transaction processing. It includes the encryption and decryption of data using keys. The configuration data is stored in sccfg.json file, which defines the security parameters necessary for authentication and data encryption operations.

2. Migration Strategy

This chapter provides information on the migration process, prerequisites and system requirements, and changes made to the existing APIs used for X990 devices.

2.1 Migration Process

The solution uses a Replatform and device update migration strategy for transitioning applications from the source device (X990) to the target device (Neo). This strategy encompasses a structured approach to ensure seamless application compatibility and optimal performance on the Neo device.

The migration process is divided into four distinct phases, each addressing specific aspects of the transition to the Neo device:

Phase 1: Application Interface and Configuration Updates

In this phase, focus is directed towards the installation and necessary modifications at the application level. Some of the key tasks include:

- Updating the User Interface (UI) components to align with the Neo device's specifications.
- Modifying the manifest file to ensure compatibility with target platform requirements.
- Updating Gradle build configurations/files.

Phase 2: Android Interface Definition Language (AIDL) Transition

This phase encompasses updates related to AIDL, which are crucial for inter-process communication and service binding. This includes:

- Replacing existing AIDL files specific to X990 with those suitable for Neo devices.
- Installing and configuring required software components to support AIDL changes on the Neo platform.

Phase 3: Android OS Version Upgrade

This phase facilitates the migration from Android 8 to Android 13 OS versions. This includes modifications to the manifest file to ensure application compatibility with the updated OS features.

Refer to Chapter 5 for more details.

Phase 4: Package and API Changes

The final phase focuses on the structural aspects of the application to fully enable its operation on Neo devices. It includes the changes made to the existing packages and APIs used for X990 devices.

Refer to Section 2.3 for more information on the list of methods added for the migration.

2.2 Prerequisites

Before beginning the migration process, ensure that the below applications are installed:

1. Girgit
2. GirgitSystemService

These applications contain the necessary frameworks and additional AIDL interfaces (included under AIDL.zip file) to support the migration.

2.3 Changes to Packages and Methods

The following are introduced as optional entities in Girgit as opposed to VFI service (X990):

- Privileged services: Refer to Chapter 4 for more details.
- Girgit external API
- Feature Manager and OpenSDI

2.3.1 Girgit External API

The solution provides 2 new Girgit AIDLs along with an addition of APIs in the existing 5 AIDLs.

The newly added AIDLs are:

1. `ISettingsManager.aidl:` Introduced as a new interface for managing settings within the Girgit framework. It manages various device settings, such as configuring time and date, setting screen brightness, and enabling permissions. Below is the list of APIs under this interface:

S. No	APIs	Description

1.	int settingsSetActions(int settingsType, in Bundle bundle)	Executes setting changes.
2.	Bundle settingsReadActions(int settingsType, in Bundle bundle)	Reads current settings.
3.	boolean settingPCIRebootTime(int hour, int min, int sec)	Sets the PCI reboot time.
4.	long getPCIRebootTime()	Retrieves the PCI reboot time in seconds.
5.	void setScreenLock(boolean isLock)	Locks or unlocks the screen.
6.	boolean setDeviceBrightnessLevel(int level)	Sets the device brightness level.
7.	boolean isShowBatteryPercent(boolean isShow)	Determines whether to show the battery percentage in the status bar.
8.	void enableAlertWindow(String packageName)	Enables alert window permissions for a specific package.
9.	void clearCachesByPackageName(String packageName)	Clears application caches for a specified package.

2. `IFFBase.aidl:` This interface provides methods to retrieve the base IP address and check the connection status with the base unit. Below is the list of APIs under this interface:

S. No	APIs	Description
1.	byte[] getBaseIpAddress()	Retrieves the IP address of the base unit.
2.	boolean isBaseConnected()	Checks if there is an active connection to the base unit.

The existing AIDLs with newly added APIs are:

1. `IDeviceInfo.aidl:` This interface provides several methods for effectively managing and retrieving information about a device. Below is the list of 3 newly added APIs under this interface.

S. No	API	Description
1.	int getDeviceStatus(in Bundle bundle)	Checks the status of various device components such as printers, card readers, pin pads, cameras, and SD cards.
2.	String getButtonBatteryVol()	Retrieves the voltage of the button battery.
3.	Bundle getDeviceInfo()	Provides comprehensive information about the device.

2. `INetworkManager.aidl`: This interface provides a set of methods to manage network-related operations on devices. Below is the list of 8 newly added APIs under this interface.

S. No	API	Description
1.	boolean isMultiNetwork()	Checks if multi-network support is enabled
2.	void setMultiNetwork(boolean enable)	Enables or disables multi-network support
3.	String getMultiNetworkPrefer()	Retrieves the current multi-network preference
4.	boolean setMultiNetworkPrefer(String prefer)	Sets the preferred network order for multi-network.
5.	void setEthernetStaticIp(in Bundle bundle)	Configures a static IP for Ethernet or switches to DHCP.
6.	void setWifiStaticIp(in Bundle bundle)	Configures a static IP for Wi-Fi or switches to DHCP
7.	void setMobilePreferredNetworkType(String type)	Sets the preferred mobile network type for the current SIM card.
8.	String getMobilePreferredNetworkType()	Retrieves the preferred mobile network type for the current SIM card.

3. `ISystemManager.aidl`: This interface provides a set of methods to manage system-level operations on the device. Below is the list of 9 newly added APIs under this interface.

S. No	API	Description
1.	boolean isAdbMode()	Retrieves the status of ADB.
2.	boolean killApplication(String packageName)	Terminates the application specified by the `packageName`.
3.	boolean restartApplication(String packageName)	Restarts the application specified by the `packageName`.
4.	void initLogcat(int logcatBufferSize, int logcatBufferSizeSuffix, in Bundle bundle)	Initializes the logcat configuration with specified buffer size and prefix.
5.	String getLogcat(String logcatFileName, int compressType)	Retrieves the log buffer file.
6.	Bundle getLaunchAppsInfo(long beginTime, long endTime)	Retrieves the usage count of applications within a specified time range.
7.	ISettingsManager getSettingsManager()	Retrieves an `ISettingsManager` object to perform settings-related actions.
8.	Bitmap takeCapture()	Captures the current screen and returns the bitmap data.
9.	void shutdownDevice()	Shuts down the device.

4. `IBeeper.aidl:` This interface provides methods to programmatically manage the beeping functionality of the device. Below is 1 newly added API under this interface.

S. No	API	Description
1.	void startBeepWithConfig(int msec, in Bundle bunble)	Initiates a beeping sound based on the specified configuration

5. `IDeviceService.aidl:` This interface acts as a central point in an application, facilitating access to various peripheral device services associated with a terminal. Below is the list of 3 newly added APIs under this interface.

S. No	API	Description
1.	WirelessConnectListener getWirelessConnectionMgr()	Retrieves the `WirelessConnectionMgr` object.
2.	IGirgitExt getGirgitExt()	Retrieves the `GirgitExt` object.
3.	IFFBase getFFBase()	Provides access to the `FFBase` object.

Refer to Girgit Programmers Guide for more details on each API.

2.3.2 Feature Manager

The solution includes an optional Feature Manager Service, specifically developed for handling non-EMV card transactions. The Feature Manager is a background service responsible for enabling specific features within the Girgit service. One key feature currently managed by this service is the enablement of OpenSDI, which allows for direct APDU command exchanges.

NOTE

Feature Manager Service is non-P2PE complaint. To use this service, contact the Verifone Support Team.

OpenSDI:

In Girgit, the `whitelist.json` file is not included in the default configuration provided with the SDI base and SDI config packages. However, if the user application requires it, you can install this file through a User config package.

The SDI Server do not deliver payment relevant sensitive data (e.g. a PAN) as clear text to an outside application. Such data elements are either encrypted or obfuscated to prevent unauthorized data access. There are cases where it is necessary to deviate from this rule (e.g. for loyalty cards). The SDI Server manages this with a list of PAN ranges which are excluded from the secure data handling. SDI-Server performs a check of the first digits of the PAN with the PAN's given in the whitelist. Also ranges of PANs are allowed.

Example:

```
[
"4377","88888","78787878","10000","600000-601099"
]
```

The `whitelist.json` file serves two main purposes:

1. It whitelists the set of cards that will block sensitive payment-related data, such as the Primary Account Number (PAN).
2. If there is a need for a clear PAN format, then it can be added in the whitelist.json file.
3. It enables or disables OpenSDI direct SDI commands. If the file contains the value `[OPENSDI]`, OpenSDI is enabled, allowing customers to interact with the card using direct SDI commands. By default, the file contains the values [0,1,2,3,4,5,6,7,8,9].

3. Permissions and Functionality Management

3.1 Common Permission Challenges

If the following runtime exception in the application is encountered, it indicates a permission-related issue that needs to be addressed.

The application throws a `java.lang.RuntimeException` when attempting to start an activity. This is caused by a `java.lang.SecurityException`, indicating that the application does not have the `android.permission.READ_PHONE_STATE` permission.

Issue:

```
by: java.lang.SecurityException: getActiveSubscriptionInfoForSimSlotIndex: uid 10101 does
not have android.
permission.READ_PHONE_STATE.
at android.os.Parcel.createException(Parcel.java:2072)
at android.os.Parcel.readException(Parcel.java:2040)
at android.os.Parcel.readException(Parcel.java:1988)
at
com.android.internal.telephony.ISub$Stub$Proxy.getActiveSubscriptionInfoForSimSlotIndex(ISub
.java:1308)
at
android.telephony.SubscriptionManager.getActiveSubscriptionInfoForSimSlotIndex(SubscriptionM
anager.java:1272)
```

Solution:

To resolve the aforementioned issue, modify the `isSimReady()` method in the `SimUtil.java` class and remove the annotation as shown below:

```
@SuppressLint("MissingPermission")
and change the code as follows


public static boolean isSimReady(Context context, int slot) {
    SubscriptionManager subscriptionManager = (SubscriptionManager)
context.getSystemService(Context.TELEPHONY_SUBSCRIPTION_SERVICE);
    if (subscriptionManager == null) {
        LogUtil.e(TAG, "subscriptionManager is null");
        return false;
    }

    // Permission check added here
    if (ActivityCompat.checkSelfPermission(context, Manifest.permission.READ_PHONE_STATE) !=
PackageManager.PERMISSION_GRANTED) {
        // TODO: Consider calling
        //    ActivityCompat#requestPermissions
        // here to request the missing permissions, and then overriding
        //   public void onRequestPermissionsResult(int requestCode, String[] permissions,
        //                                          int[] grantResults)
        // to handle the case where the user grants the permission. See the documentation
        // for ActivityCompat#requestPermissions for more details.
        return true;
    }
    SubscriptionInfo subscriptionInfo =
subscriptionManager.getActiveSubscriptionInfoForSimSlotIndex(slot);
    if (subscriptionInfo == null) {
        LogUtil.d(TAG, "slot " + slot + " has no SIM");
        return false;
    }

    TelephonyManager mTelephonyManager = (TelephonyManager)
context.getSystemService(Context.TELEPHONY_SERVICE);
    boolean isSimCardExist = false;
    try {
        Method method = TelephonyManager.class.getMethod("getSimState", int.class);
        int simState = (Integer) method.invoke(mTelephonyManager, new Object[]{slot});
        if (TelephonyManager.SIM_STATE_READY == simState) {
            isSimCardExist = true;
        }
    } catch (Exception e) {
        LogUtil.d(TAG, "e:" + e.toString());
        e.printStackTrace();
    }

    LogUtil.d(TAG, "isSimCardExist:" + isSimCardExist);

    return isSimCardExist;
}
```

3.2 Printing Functionality Permissions

To enable image printing in Android 13 version and above, add the below permission in the Android Manifest file:

```
<uses-permission android:name="android.permission.READ_MEDIA_IMAGES" />
```

Modify the code for image printing as shown below:

```
private String createDirectoryAndSaveFile(byte[] imageData) {
        File file = null;
        OutputStream stream = null;
        File storageDir = null;
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
            //Android 13 Specific changes for Image storage
            storageDir =
Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES);
        } else {
            storageDir =
DeviceMaster.getInstance().getAppContext().getExternalFilesDir(Environment.DIRECTORY_PICTURE
S);
        }
        try {
            String contentType = URLConnection.guessContentTypeFromStream(new
ByteArrayInputStream(imageData)).replace(REPLACE_STRING, ".");
            file = new File(storageDir, IMAGE_NAME + Math.random() + contentType);

            stream = new FileOutputStream(file);

        } catch (FileNotFoundException e) {
            LogUtil.e(TAG, "File not found", e);
        } catch (IOException e) {
            LogUtil.e(TAG, "File not created", e);
        } finally {
            if (stream != null) {
                try {
                    stream.write(imageData);
                    stream.flush();
                    stream.close();
                } catch (IOException e) {
                    LogUtil.e(TAG, "Exception while writing file.", e);
                }
            }
        }
        Uri savedImageURI = null;
        if (file != null) {
            savedImageURI = Uri.parse(file.getAbsolutePath());
        }
        return savedImageURI != null ? savedImageURI.toString() : "";
    }
```

4. Privileged Access

This chapter outlines the specific permissions required for on-demand privileged applications. These applications have enhanced access and capabilities, allowing them to utilize specific functionalities not available to standard applications.

Note that privileged applications are for customers using certain system settings which are restricted by android due to security and memory constraints.

During migration, ensure that you adhere to the below guidelines for managing privileged access.

NOTE For more details on privileged access, contact Verifone Support Team.

4.1 Key Features and Permissions

***Package naming convention*:**

The application's package name must begin with 'com.priv.'. This naming convention is a requirement for identifying applications that are eligible for privileged access.

Alternatively, you can create a separate service with package name starting with 'com.priv.' This service will handle requests that require elevated permissions.

NOTE For more details on Android permissions, refer to https://developer.android.com/reference/android/Manifest.permission

With this naming convention, the application is granted access to the following permissions and custom APIs for requested features:

4.1.1 Hide Navigation Bar

To hide the navigation bar, set the device_operating_mode to 3 which corresponds to KIOSK mode. Subsequently, send a broadcast intent programmatically to update the mode dynamically.

Example:

```
Settings.Global.putInt(getActivity().getContentResolver(), "device_operating_mode", 3);
Intent intent = new Intent("com.verifone.DEVICE_OPERATING_MODE");
getActivity().sendBroadcast(intent);
```

KIOSK Mode:

The following behavior is enforced when KIOSK mode is enabled:

- The screen operates in immersive mode, ensuring that the navigation or status bar remain hidden when user touches the screen.
- All soft keys are hidden except within the Settings and other system UI views, where the back key is available.
- Hardware keys:
    - The 'Recents' key is always disabled.
    - The Home key is disabled while a payment application is in the foreground but is otherwise available to help return to the home application.
    - The Back key is always available.

4.1.2 Manage User Certificates

Verifone employs a proprietary mechanism for the installation and removal of CA certificates via the **UpdateService**. Privileged customers have the option to utilize this feature for managing certificates.

4.1.3 Manage WIFI Profiles

To connect to the desired Wi-Fi networks, the privileged application must have the following specific permissions in the Android manifest file:

1. ACCESS_FINE_LOCATION

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

2. ACCESS_WIFI_STATE

```
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
```

3. CHANGE_WIFI_STATE

```
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
```

4. NETWORK_SETTINGS

```
<uses-permission android:name="android.permission.NETWORK_SETTINGS" />
```

4.1.4 SIM Card Data

To retrieve SIM card data, following permissions are needed in the Android manifest file:

1. READ_PRIVILEGED_PHONE_STATE: This permission allows the application to retrieve detailed SIM card information.

```
<uses-permission android:name="android.permission.READ_PRIVILEGED_PHONE_STATE" />
```

2. READ_PHONE_STATE: This permission is classified as dangerous by Android. As a result, the application must request the user's permission by implementing the `` `requestPermissions()` `` method to gain access.

```
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
```

4.1.5 Change Device Language

To change the device language, the application requires the following permissions in the Android manifest file:

1. WRITE_SECURE_SETTINGS

```
<uses-permission android:name="android.permission.WRITE_SECURE_SETTINGS" />
```

2. CHANGE_CONFIGURATION

```
<uses-permission android:name="android.permission.CHANGE_CONFIGURATION" />
```

4.1.6 Change Device Date, Time, and Timezone

To change the device date, time, and timezone, the application requires the following permissions in the Android manifest file:

1. SET_TIME

```
<uses-permission android:name="android.permission.SET_TIME" />
```

2. SET_TIME_ZONE

```
<uses-permission android:name="android.permission.SET_TIME_ZONE" />
```

4.1.7 Manage APN

To configure or update the APN settings, the application requires the following permission in the Android manifest file:

1. WRITE_APN_SETTINGS

```
<uses-permission android:name="android.permission.WRITE_APN_SETTINGS" />
```

4.1.8 Manage Screen Brightness

To modify the screen brightness functionality, the application requires the following permission in the Android manifest file:

1. WRITE_SECURE_SETTINGS

```
<uses-permission android:name="android.permission.WRITE_SECURE_SETTINGS" />
```

4.1.9 Set Screen Timeout

To set the screen timeout duration, the application requires the following permission in the Android manifest file:

1. WRITE_SECURE_SETTINGS

```
<uses-permission android:name="android.permission.WRITE_SECURE_SETTINGS" />
```

4.1.10 Manage Private DNS Modes

To manage private DNS modes, the application requires the following permission in the Android manifest file:

1. WRITE_SECURE_SETTINGS

```
<uses-permission android:name="android.permission.WRITE_SECURE_SETTINGS" />
```

4.1.11 Change Vendor Passwords

The features required for changing vendor passwords are facilitated by the **Usermanagement application**. Privileged applications can perform the following actions:

1. Check if the user management is configured using the content provider:

```
private static final String USER_AUTH_CP =
"com.verifone.user.auth.action.USER_AUTH_CONTENT_PROVIDER";
private static final String URI_SCHEME = "content";
private static final String USER_PATH_STATUS = "STATUS";
private static final String COL_NAME_RESULT = "RESULT";
private static final String COL_NAME_TIMEOUT = "TIMEOUT";

ProviderInfo providerInfo = null;
Intent intent = new Intent(USER_AUTH_CP);
List<ResolveInfo> queryIntentContentProviders =
context.getPackageManager().queryIntentContentProviders(intent, 0);
for (ResolveInfo resolveInfo : queryIntentContentProviders) {
    providerInfo = resolveInfo.providerInfo;
}

if (providerInfo != null) {
    Uri.Builder builder = new
Uri.Builder().scheme(URI_SCHEME).authority(providerInfo.authority);
    builder.appendPath(USER_PATH_STATUS);

    Uri contentUri = builder.build();
    try (Cursor cursor = context.getContentResolver().query(contentUri, null, null, null,
null)) {
        if (cursor != null && cursor.getCount() == 1) {
            cursor.moveToNext();
            boolean result = cursor.getInt(cursor.getColumnIndex(COL_NAME_RESULT)) == 1;
            if (result) { return true;
            } else {
                int timeout = cursor.getInt(cursor.getColumnIndex(COL_NAME_TIMEOUT));
                Log.e(TAG, "Operation timed out " + timeout);
                return false;
            }
        }
    }
}
```

2. Start user management activity to authorize or manage users:

```
private static final String INTENT_REQUEST_ACTION =
"com.verifone.user.auth.intent.action.REQUEST";
```

```
private static final String USER_AUTH_PACKAGE = "com.verifone.user.auth";
private static final String USER_AUTH_ACTIVITY =
"com.verifone.user.auth.view.UserAuthActivity";

Intent intent = new Intent("INTENT_REQUEST_ACTION");
intent.setClassName(USER_AUTH_PACKAGE, USER_AUTH_ACTIVITY);
intent.putExtra("INTENT_EXTRA_REQUEST_TYPE", requestType);
intent.putExtra("INTENT_EXTRA_USER_ROLE_TYPE", role);
```

Here, the requestType and role can be:

requestType	Role	Action
MANAGE	Null	Setup/update passcodes
AUTHORIZE	MERCHANT_ADMIN	Used to protect restricted features like factory reset
	MERCHANT_MANAGER	Authorize existing manager
	MERCHANT_CASHIER	Authorize existing cashier

4.1.12 Manage Airplane Mode

To manage the airplane mode, the application requires the following permission in the Android manifest file:

1. WRITE_SECURE_SETTINGS

```
<uses-permission android:name="android.permission.WRITE_SECURE_SETTINGS" />
```

To turn on airplane mode, the applications must execute the following code:

```
Settings.Global.putInt(getActivity().getContentResolver(), Settings.Global.AIRPLANE_MODE_ON,
isEnabled ? 1 : 0);
Intent intent = new Intent(Intent.ACTION_AIRPLANE_MODE_CHANGED);intent.putExtra(STATE,
isEnabled);
getActivity().sendBroadcast(intent);
```

4.1.13 Reboot Device

To reboot the device, the application requires the following permission in the Android manifest file:

1. REBOOT

```
<uses-permission android:name="android.permission.REBOOT" />
```

The privileged application must also execute the following:

```
PowerManager pm = (PowerManager)getContext().getSystemService(Context.POWER_SERVICE);
pm.reboot("reboot-reason");
```

4.1.14 Shut Down Device

To shut down the device, the application requires the following permissions in the Android manifest file:

1. SHUTDOWN

```
<uses-permission android:name="android.permission.SHUTDOWN" />
```

The privileged application must also execute the following:

```
Intent intent = new Intent(Build.VERSION.SDK_INT >= 26
                ? "com.android.internal.intent.action.REQUEST_SHUTDOWN"
                : "android.intent.action.ACTION_REQUEST_SHUTDOWN");
    intent.putExtra("android.intent.extra.KEY_CONFIRM", false);    // request confirmation
from the user before shutting down?(True/False)
    intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    context.startActivity(intent);
```

5. Android Version Changes and Compatibility

5.1 Android 13 and Above Version Changes

In Android 13 and later versions, direct connections to bound services require additional configuration and changes to the manifest file. Specifically, you must include the AIDL application's package name in the manifest file.

Add the following code in the android manifest file before the application tag:

```
<queries>
<package android:name="com.vfi.smartpos.deviceservice" />
<-- other Aidl packages if any required
</queries>
```

6. User Interface (UI) Considerations

As Verifone Neo devices uses soft navigation keys (such as home, recent, and back buttons), the UI layouts provided by X990 cannot be used as it has some hard coded values for width and height.

To resolve this issue, we recommend using **Constraint layout** which works on all kind of screen sizes.

Device Features

Device	Display
Neo Device (V660p)	Large 5.5” LCD display; (720 × 1280) HD IPS LCD touchscreen
X990 (Pinpad)	4” Capacitive touch screen (800 × 480)

7. VHQ Integration Process

In case if issues arise due to the heartbeat and sending responses back to VHQ, follow the steps below:

1) Place the VHQ agent file in the **libs** folder of the presentation project.

2) Add the newly built GAndroidTms library in the **libs** folder of the domain project.

3) Update the gradle file as follows:

```
// In the build.gradle file of domain
implementation files('libs/GAndroidTmsLib_debug-1.0.0.0.aar')
// In the build.gradle file of presentation
implementation files('../domain/libs/GAndroidTmsLib_debug-1.0.0.0.aar')
implementation files('libs/vhq-agent-api-debug-4.3.34.0.aar')
implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:1.5.30"
```

4) In the code, ensure that you pass **true** once the file download is completed.

```
if (GAndroidTMSLib.getInstance() != null && VHQService.getHandleTemp() != null) {
LogUtil.d(TAG, "Execute updating VHQ status");
GAndroidTMSLib.getInstance().sendFileStatus(true);
}
```

8. Disabling Recent Apps Button

To disable the recent apps button, add the following code in BaseActivity of the Application, so that it reflects for the whole application:

```
@Override
protected void onPause() {
super.onPause();
android.app.ActivityManager activityManager =
(android.app.ActivityManager)getApplicationContext()
.getSystemService(Context.ACTIVITY_SERVICE);
activityManager.moveTaskToFront(getTaskId(), 0);
}
```

9. EMV Configuration

The Verifone EMV configuration framework supports the payment applications and enable them to configure and conduct contact and contactless transactions. The configuration data of the application is stored in the XML files including **emv-allowed.xml**, **emv-desired.xml**, and **cardranges.json**.

emv-allowed.xml:

The default EMV configuration file is emv-allowed.xml. This configuration is used by the ADK EMV contact and contactless frameworks to judge if a desired L2 kernel is allowed to load.

NOTE	This configuration is device specific and cannot be overloaded with a User config package or removed with a User config removal package.

emv-desired.xml:

For customization of EMV configuration, emv-desired.xml file is used. This file is not allowed in EMV configuration package. Use a user config package to install this file.

This file contains device specific lists of L2 kernels – contact and contactless – that a main application needs to select. Intentionally, there is no default setting. This is to avoid a change of the kernel set loaded after an update of the ADK components. Instead, a main application must provide its own emv-desired.xml and install it either as a user config package or together with the user-signed main application.

NOTE	Available certifications decide which kernel version is allowed to use on which terminal.

cardranges.json:

This file is used for mapping of different cards within the allowed card ranges. The card range configuration serves to realize card related checks inside the SDI Server because the outside application will not get all relevant data in clear.

Each card range must be configured as element of a JSON array which can be called either `staticRanges` or `dynamicRanges`. These are customer specific range categories and are not relevant for the matching and validation algorithm.

Example:

```
{
    "details": {
        "name": "PaymentCore Template",
        "templateVersion": "1.0.2",
        "generated": "2021-07-7T11:28:00.603+13:00"
    },
    "defaults": {
        "minPanLength": 13,
        "maxPanLength": 19,
        "expiryChecking": true,
        "serviceCodeCheck": true,
        "luhnChecking": true,
        "startDateChecking": false,
        "startDatePosition": 8,
        "startDateFormat": "YYMM"
    },
    "staticRanges": [
        {
            "low": "000000000000",
            "high": "999999999999",
            "products":["DEBIT"],
            "minPanLength": 14,
            "maxPanLength": 19,
            "expiryChecking": true,
            "serviceCodeCheck": true,
            "luhnChecking": true,
            "panMasking": "6-4fix"
        },
        {
            "low": "18000",
            "high": "18009",
            "products":["JCB"],
            "minPanLength": 13,
            "maxPanLength": 16,
            "expiryChecking": true,
            "serviceCodeCheck": true,
            "luhnChecking": true
        },
        ...  // n more entries
    ],
    "dynamicRanges": [
        {
            "low": "5116545113",
            "high": "5116545113",
            "products":["DEBIT", "MASTERCARD-DEBIT"],
            "minPanLength": 13,
            "maxPanLength": 19,
            "expiryChecking": true,
            "serviceCodeCheck": true,
            "luhnChecking": true
        },
        {
            "low": "423953000",
            "high": "423953999",
            "products":["DEBIT", "VISA-DEBIT"],
            "minPanLength": 13,
            "maxPanLength": 19,
            "expiryChecking": true,
            "serviceCodeCheck": true,
            "luhnChecking": true
        },
        ... // n more entries
    ]
}
```

The `defaults` sections can contain default values for several range attributes which should be used by the SDI Server for ranges where a given attribute is missing. Following range attributes can be configured in the `defaults` section:

- maxPanLength
- minPanLength
- expiryChecking
- luhnChecking
- startDateChecking
- serviceCodeCheck
- startDatePosition
- startDateFormat

The `details` section provides general information about the validation table such as name, templateVersion and generated to give the name, the version and the generation date of the validation table. The `details` section is returned by the SDI Server with getValidationInfo command.

NOTE

cardranges.json is not provided with SDI default configuration coming along with the SDI base package and SDI config package. If desired by user application, the file can be installed with a User config package.

10.1 Mandatory and Optional EMV Configuration

Mandatory File:

The below mandatory file is crucial for enabling basic EMV functionality.

- emv-allowed.xml: The default EMV configuration file. It outlines the core settings needed for EMV operations, ensuring compliance and interoperability with EMV standards.

Optional Files:

The below optional files provide additional customization and flexibility.

- emv-desired.xml: Used for customization of EMV configuration.
- cardranges.json: Used for mapping of different cards within the allowed card ranges.

10. Security Configuration

The security configurations are paramount to ensuring safe and compliant transaction processing. It includes the encryption and decryption of data using keys. The configuration data is stored in sccfg.json file, which defines the security parameters necessary for authentication and data encryption operations.

sccfg.json:

sccfg.json contains the default configuration for the ADKSEC security service, which is used by the SDI Server for secure operations. The file is provided with SDI default configuration coming along with SDI base package and it contains the following security schemes:

- ADE (SRED compliant) scheme used for cardholder sensitive data encryption.
- PED scheme for PIN block encryption (online transactions) and host message MAC calculation.
- E2E scheme for using VSS script authenticate.vso, which allows secure SDI message authentication/encryption.

These security schemes require several keys loaded to device.

Example sccfg.json (with one security scheme for ADE):

```
{
  "adksecconfig":
  {
    "hosts":
    [
      {
        "name": "X990-DUKPT-AES-ECB",
        "description": "this is ADE DUKPT configuration",
        "scheme": "schemeADE",
        "module": "ADE",
        "settings":
        {
          "encMode": "MODE_CBC",
          "IVType":  "ZERO",
          "padding": "NONE"
        }
      }
    ],
    "serviceCfg":
    {
      "secSchemes":
      [
        {
          "name": "schemeADE",
          "settings":
          {
            "padding": "PKCS7",
            "KeyManagementType": "DUKPT",
            "KeyAddressTable":
            [
              {"description": "| 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8 |  9 | 10 | <-
KeySetId (10 slots)"},
              {"encryptData": ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]}
            ]
          }
        }
      ]
    }
  }
}
```

The below table demonstrates the host names corresponding to the Sec Module and Scheme that should be used in the sccfg.json.

Type	Scheme	Algorithm	Enc mode	Host name	Sec Module
PIN	MSK	TDES		X990-PIN-MSK-TDES	TDES-MSK
	DUKPT			X990-PIN-DUKPT-TDES	TDES-DUKPT
Master/Work key	DUKPT	TDES	ECB	X990-DUKPT-TDES-ECB	TDES-DUKPT
			CBC	X990-DUKPT-TDES-CBC	
		AES	ECB	X990-DUKPT-AES-ECB	AES-DUKPT
			CBC	X990-DUKPT-AES-CBC	

	MSK	TDES	ECB	X990-MSK-TDES-ECB	TDES-MSK
			CBC	X990-MSK-TDES-CBC	
		AES	ECB	X990-MSK-AES-ECB	Not implemented by SEC
			CBC	X990-MSK-AES-CBC	
MAC	MSK	TDES	ECB	X990-MAC-MSK-TDES-ECB	TDES-MSK
			CBC	X990-MAC-MSK-TDES-CBC	
			X99	X990-MAC-MSK-TDES-X99	
			X919	X990-MAC-MSK-TDES-X919	
	DUKPT		ECB	X990-MAC-DUKPT-TDES-ECB	TDES-DUKPT
			CBC	X990-MAC-DUKPT-TDES-CBC	
			X99	X990-MAC-DUKPT-TDES-X99	

			X919	X990-MAC-DUKPT-TDES-X919	
		AES	ECB	X990-MAC-DUKPT-AES-ECB	AES-DUKPT
			CBC	X990-MAC-DUKPT-AES-CBC	
			X99	X990-MAC-DUKPT-AES-X99	
			X919	X990-MAC-DUKPT-AES-X919	

NOTE

If desired by user application, sccfg.json can be installed with a User config package to overload default file of SDI base package.

11.1 Mandatory and Optional Security Configuration

Mandatory	**Optional**
scccfg.json: ADKSEC default configuration for SDI server.	scccfg.json: This file can be modified and replaced/overloaded by a User config package as per the requirement of the user application.

11. Post-Migration Support

Girgit Support Process

When an existing customer needs a new feature or identifies a defect, they must initiate the process by creating a ticket on the JIRA Service Desk (JSD) platform. JSD (https://jiraservicedesk.verifone.com) is a customer facing JIRA portal that is used as an external/internal ticketing system. Once a JSD ticket is submitted, it is assigned to the Girgit development team, who will work to resolve the issue according to the specified requirements.

Girgit Release

All the latest Girgit releases will be released in the Verifone Cloud, where the customer can log in and download the latest packages.

Contact Detail

For addressing any residual questions or concerns post-migration, please reach out via email at sakthimani.p@verifone.com.

12. Troubleshooting AIDL Connection

If there is a failure to establish an AIDL connection, then this can result in the application not functioning as expected due to the inability to communicate properly with the necessary services.

To resolve this issue, perform a restart of the terminal. Restarting the terminal can reset any underlying processes or configurations that may be hindering the AIDL connection, allowing it to re-establish successfully.

Verifone

University Drive
Coral Springs,
FL 33065, USA
Fax: 4545 233
Phone: 001 454 2333

www.verifone.com

Thank you!

We are the payments architects who truly understand commerce.

As payment architects we shape ecosystems for online and in-person commerce experiences, including all the tools you need... from gateways and acquiring to fraud management, tokenization and reporting.

As commerce experts, we are here for you and your business. With our payment devices, our systems & solutions and our support. Everywhere. Anytime. So that your customers feel enabled, recognized and well taken care of, even beyond their expectations.

Verifone. Creating omni-commerce solutions that simply shape powerful customer experiences.