# Commerce Driver™

iOS Quick-Start Guide v1.0

# Understanding EMV® Certification

## What is EMV?

EMV, named after the organizations that created the technology standard – Europay, MasterCard and Visa – is a technical standard for the interaction between chip-based "smart cards" and approved payment devices. The standard is now managed by EMVCo, a consortium with control split equally among American Express, JCB, Discover, MasterCard, UnionPay and Visa.

The purpose of the EMV Specifications is to facilitate the worldwide interoperability and acceptance of secure payment transactions. During a card-present payment transaction, payment data – securely stored on a microchip – can either be embedded in a traditional plastic card or mobile device.

## How Does it Work?

EMV devices are able to read data stored on a chip within the card. By using chips as an active part of the payment transaction, EMV cards and devices help prevent credit card fraud from stolen account numbers, cloned payment cards and other security and fraud threats that exist today.

Each chip-based card is embedded with encrypted data. During the transaction authorization process, the encrypted data in the card is used to verify the card's authenticity. Strong cryptographic functions are used to authenticate the card and cardholder to ensure validity.

## Becoming EMV Compliant

The EMV standards define the interaction at the physical, electrical, data and application levels between EMV cards and EMV card processing devices. There are three levels of EMV certification:


### Level 1 – Hardware/Terminal Certification

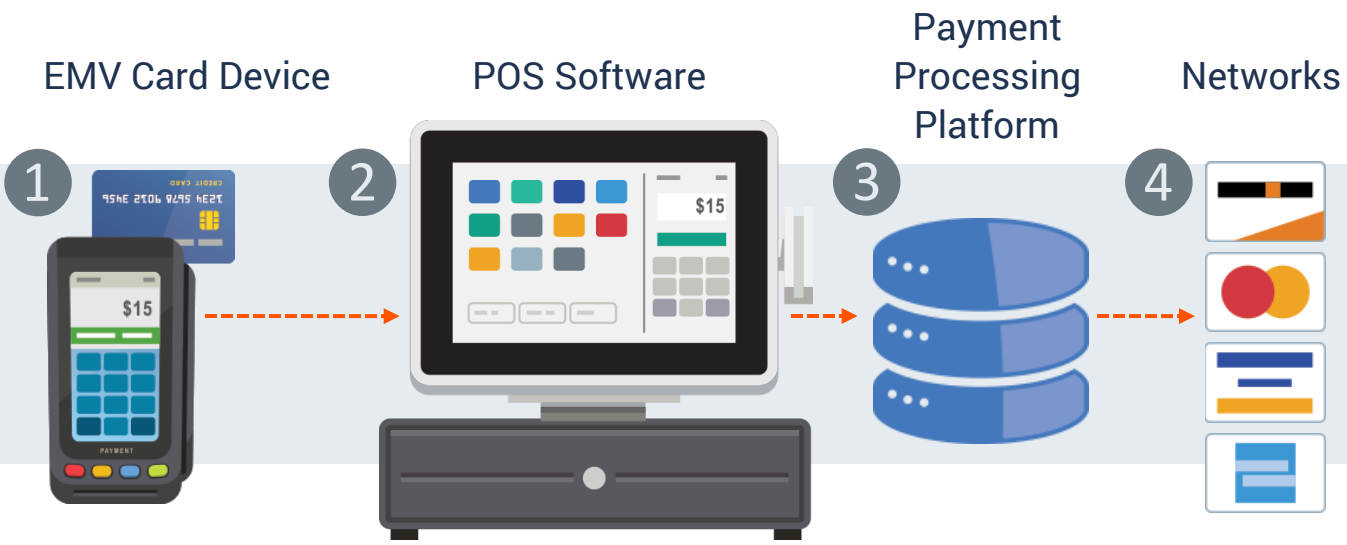
Level 1 certification covers the physical interface between the card acceptance terminal and the EMV card. *Responsible Party: Terminal Vendor*

### Level 2 – Kernel Certification

Level 2 certification covers the software interface between the card acceptance terminal and the chip card. *Responsible Party: Terminal Vendor*

## Level 3 – Payment Application Certification

Level 3 certification covers the software interface between a POS application and the card acceptance terminal. ***Responsible Party: Software Vendor***




Level 3 certification - also called end-to-end (E2E) or network certification - tests each unique EMV path to the networks. The testing flow is as follows:

> ❶ Level 1 and 2 certified card device, ❷ the POS software, any middleware or gateway in use, ❸ the processor, and finally out to ❹ the card brands.

This process must be completed individually for:

1. each device the POS application is using
2. each version of software
3. each processor
4. each network

The cost and complexity of managing certification to each unique EMV path can quickly become overwhelming. There is a better way. You can create EMV approved iOS based Point of Sale (POS) applications in a snap with the Commerce Driver™ from EVO Snap*

# EVO Snap* Commerce Driver™

The Commerce Driver™ makes implementing EMV technology faster and easier by combining technical, operational and strategic components into a fully integrated, tested and production-ready solution.

Meet all EMV Level 3 compliance requirements and instantly enable PCI-compliant transactions with point-to-point encryption using the Commerce Driver from EVO Snap*.

## How it Works

Like a printer driver, the pre-certified Commerce Driver™ SDK installs alongside your software application - adding EMV transaction processing to your iOS-based POS systems. The Commerce Driver facilitates all transactional communication with the EVO Payments global processing platforms and approved hardware devices to isolate payment data and keep it separate from the software application.


Advantages of the Commerce Driver™:

1. Easy to install…similar to a printer driver
2. Pre-certified…decreasing your time to market
3. Reduces PCI Compliance scope and liability for merchants
4. Provides Point to Point Encryption for all transactions
5. EVO maintained…no ongoing maintenance costs

6. Available for both US and EU

# Commerce Driver™ Specifications

## Authentication Methods

* Chip & PIN
* Chip & Signature

## System Requirements

* iOS 7+

## Networks

* Bluetooth
* USB
* Ethernet
* Wireless

## Managed API Calls

* Login
* Authorize
* Capture
* Authorize & Capture w/Duplicate Transaction Override
* Resubmit
* Transaction Lookup
* Void w/Forced Void
* Batch Capture w/Forced Void
* Local Settings File w/Authentication Parameters

## Pre-Certified Hardware

| | Payment Support | | | | Security and Compliance | | Device Connectivity | | | Platform Support | | | Univ SDK |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Smart Chip Capable (EMV) | MagStripe Capable | NFC Capable | PIN Capable | P2PE | Tokenization | USB | BlueTooth | Audio Jack | Apple Support | Android Support | Windows Mobile Support | Global Support |
| **Ingenico** | | | | | | | | | | | | | |
| iPP320 | ● | ● | ● | ● | ● | ● | ● | | | | | ● | ● |
| iPP350 | ● | ● | ● | ● | ● | ● | ● | | | | | ● | ● |
| iCMP | ● | ● | ● | ● | ● | ● | | ● | | ● | ● | ● | ● |
| **EVO** | | | | | | | | | | | | | |
| IT m-50 | | ● | | | ● | ● | | | ● | ● | ● | ● | |
| IT m-100 | ● | ● | ● | ● | ● | ● | ● | | ● | ● | ● | ● | |

# Commerce Driver™ Integration

## Getting Started

Getting started with the Commerce Driver™ is a snap. Once you've selected your desired Platform, Network and Hardware, the setup is similar to a direct Web Services integration. However, Commerce Driver™ will need to be hosted locally.

1. Create transaction data objects in your POS.
2. Pass the transaction data to the Commerce Driver™.
3. Commerce Driver™ initiates commands in the terminal and gathers the tender/EMV data and sends it to the platform.
4. EVO sends a return response to the Commerce Driver™ with details for the receipt.

## Installation

1. Download the Commerce Driver™ SDK(s) appropriate for your selected device manufacturer(s).
2. Unzip the SDK and add .framework file to the project directory (e.g.: /<ProjectName>/src).

## Framework Integration

1. Import the framework into your project by clicking file->add files to <ProjectName> and navigate to the directory where you placed the .framework file. Select the file and ensure the target project is checked and click 'Add'.
2. Add the required system frameworks listed in EVOCommerceDriver.framework/EVOCommerceDriver.h by selecting your project in the project navigator, selecting your project target in the left column, select "General" at the top, and scrolling to the "Linked Frameworks and Libraries" section. Click the "+" to add the listed frameworks by searching.
3. Add the –ObjC flag to your projects selecting your project in the project navigator, selecting your project target in the left column, select "Build Settings" at the top, and searching for "Other Linked Flags". Add the flag by double clicking the field and typing "-ObjC" and hitting return.

4. If using the Ingenico version, you will need to add the supported external accessory protocols key to your projects Info.plist. Select your project in the project navigator, and your project target in the left column, then select “Info” at the top. Add a new key by clicking the “+” that appears while hovering over an existing key. The key should be named “Supported external accessory protocols”, its type is Array. Click the arrow to the left of the key so that it points down and again click the “+” on the key adding a sub row that should say “Item 0”. Set the value of this row by double clicking to “com.ingenico.easypayemv.spm-transaction”

# API Integration

1. **Import the Commerce Driver**. In your class, import the EVOCommerceDriver with:

```
#import <EVOCommerceDriver/EVOCommerceDriver.h>
```

2. **Configure the API**. The API can be configured two (2) ways:
    a. Using a configuration file:
        i. Add a “configuration.plist” file to your project
        ii. Add the following keys to the file: **baseURL**, **serviceID**, **workflowID**, **applicationProfileID**, and **serviceKey**
        iii. Create an ESConfiguration object with:

```
NSString* configPath = [[NSBundle bundleForClass:[self class]]
pathForResource:@"EVOSnapConfig" ofType:@"plist"];

ESConfiguration* config = [ESConfiguration
configurationWithDictionary:[NSDictionary
dictionaryWithContentsOfFile:configPath]];
```

        iv. Initialize EVOCommerceAPI with:

```
EVOCommerceAPI *api = [[EVOCommerceAPI alloc]
initWithConfiguration:config];
```

    b. Using a dictionary:
        i. Create an ESConfiguration object with:

```
ESConfiguration *config = [ESConfiguration
configurationWithDictionary:@{ESConfigurationBaseURL:@"",
ESConfigurationServiceID:@"",
ESConfigurationServiceKey:@"",
ESConfigurationWorkflowID:@"",
ESConfigurationApplicationProfileID:@""}];
```

ii. Initialize EVOCommerceAPI with:

```
EVOCommerceAPI *api = [[EVOCommerceAPI alloc]
initWithConfiguration:config];
```

3. **Begin making calls**. To begin making calls, first call:

```
[api signOnWithUserName:<UserName> password:<Password>
completion:^(ESSignOnResponse *response, NSError *error){}];
As long as your EVOCommerceDriverAPI object isn't released, the session will
persist. To sign out, call [api removeAuthorization];
```

## Swiper Integration

1. **Import the Commerce Driver**. In your class, import the EVOCommerceDriver with:

```
#import <EVOCommerceDriver/EVOCommerceDriver.h>
```

2. **Add the ESSwiperDelegate protocol to your interface**. E.g.:

```
@interface <YourClassName> () <ESSwiperDelegate>
```

If ESSwiperController supports EMV, use the ESSwiperControllerDelegate

3. **Add these required methods to your class**:

```
swipeTransactionAmount:, swiper:didSwipeCard:
```

If ESSwiperController supports EMV, you will also need to add:

```
swiper:authorizeEMVTender:completion:,
swiper:didFinishEMVTransactionWithResult:tender:completion:
```

4. **Start the swiper by calling**:

```
[ESSwiperController initWithDelegate] or [ESSwiperController
initWithDelegate:loggingEnabled:]. E.g.: ESSwiperController *swiper =
[[ESSwiperController alloc] initWithDelegate:self loggingEnabled:YES];
```

# Request Examples

1. **Authorization Example**:

```
(void)swiper:(ESSwiper *)swiper didSwipeCard:(id<ESSwipedCardTender>)cardTender{
ESTransactionRequest *request = [ESTransactionRequest new];
request.tender = cardTender;
request.customerPresent = YES;
request.employeeID = @"1";
request.laneID = @"1";
request.reference = @"";
request.orderNumber = @"";

[_api authorizeTransaction:request completion:^(ESBankcardTransactionResponse
*response, NSError *error){}];
}
```

2. **EMV Authorization Example**:

```
(void)swiper:(ESSwiperController*)swiper
authorizeEMVTender:(id<ESEMVCardTender>)tender completion:(void(^)(NSString
*result))completion{
ESTransactionRequest *request = [ESTransactionRequest new];
request.tender = tender;
request.customerPresent = YES;
request.employeeID = @"1";
request.laneID = @"1";
request.reference = @"";
request.orderNumber = @"";

[_api authorizeTransaction:request completion:^(ESBankcardTransactionResponse
*response, NSError *error){
completion(@"8A023030");
}];
}
```

3. **Query Family Example**:

```
ESQueryTransactionsFamilies *request = [ESQueryTransactionsFamilies new];
Request.queryTransactionsParameters.merchantProfileIDs =
@[_api.merchantProfileID];
request.pagingParameters.pageSize = 5;
request.pagingParameters.page = 0;
[_api queryTransactionFamilies:request completion:^(NSArray *results, NSError
*error){}];
```

# Reference Documents

For additional assistance, please visit the EVO Snap* Support site at http://www.evosnap.com/support/ for more information.

Sample code available at http://github.com/evo-snap/#