Analysis of Payment Service Provider
SDKs in Android

Samin Yaseer Mahmud, K. Virgil English, Seaver Thorn, William Enck, Adam Oest, Muhammad Saad

Presented at Annual Computer Security Applications Conference (ACSAC), 2022
Digital Payment has Revolutionized Commerce

Mobile applications are becoming a predominant form of payment.
Payment Processing SDK

**Step 1:** SDK collects payment information from users through the app and securely forwards them to payment service provider’s (PSP) server.

**Step 2:** PSP server facilitates the payment processing with the bank maintaining appropriate specification and protocols.

**Step 3:** Bank informs the PSP whether the payment request is approved or not, completing transaction accordingly.

**Step 4:** PSP forwards the response back to the SDK; SDK notifies the user showing appropriate message in the app.
Mobile Application Security Verification Standard

- **MASVS** (v1.2) is an industry security standard for mobile applications, proposed by **OWASP**.
- Captures common android security weaknesses
- We designed 28 security checks of 4 broader categories from the standard that are applicable to Payment and SDKs
- Classifies requirements into:
  - MASVS L1 (Ordinary Use Cases)
  - MASVS L2 (Sensitive Use Cases)
- MASVS captures many requirements of PCI DSS
Data Storage Requirements

- Goal is to protect **sensitive user data** in device
- Many requires dataflow analysis for tracking data
  - Mostly **PCI DSS** related
- Others involve looking at syntax in **code, layout, config** files
- Excluded checks:
  - App removes data from system memory
  - App educates user about processed data

<table>
<thead>
<tr>
<th>MASVS Check</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android Keystore not utilized</td>
<td>Syntax (Code)</td>
</tr>
<tr>
<td>Accessed data from External Storage</td>
<td>Syntax (Code, Config)</td>
</tr>
<tr>
<td>Logged sensitive data</td>
<td>Data Flow</td>
</tr>
<tr>
<td>Data shared with third parties</td>
<td>Data Flow</td>
</tr>
<tr>
<td>Keyboard cache enabled</td>
<td>Syntax (Layout)</td>
</tr>
<tr>
<td>Data leaked through IPC</td>
<td>Data Flow</td>
</tr>
<tr>
<td>Data leaked through UI</td>
<td>Syntax (Layout)</td>
</tr>
<tr>
<td>Auto backup turned on</td>
<td>Syntax (Config)</td>
</tr>
<tr>
<td>Screenshot disabling not detected</td>
<td>Syntax (Code)</td>
</tr>
<tr>
<td>Screen lock presence not detected</td>
<td>Syntax (Code)</td>
</tr>
<tr>
<td>Sensitive data stored</td>
<td>Data Flow</td>
</tr>
<tr>
<td>Sensitive data stored unencrypted</td>
<td>Data Flow</td>
</tr>
</tbody>
</table>

MASVS - L1

MASVS - L2
Goal and Key Challenges

**Goal:** To build a novel tool that would enable **dataflow analysis** of an **SDK** without the need of a **host app**

**Challenges:**
- **Entry points** to SDK analysis are not easily inferred
- Existing tools consider **whole application context**

**Goal:** To perform a security evaluation of payment SDKs with industry security standards

**Challenges:**
- Security standards are written in **natural languages**
- Security standards are written with **application** in mind
AARDroid Architecture

Preprocessing Steps:

1: Connecting to the SDK:
   ➔ Create empty android project
   ➔ Include SDK

2: Identifying relevant API:
   ➔ Consider all public API
   ➔ Consider API related with Strings

3: Resolving API Semantics:
   ➔ Incorporate Text analytics
   ➔ Incorporate Data ontology
   ➔ Connect dummy edges from app -> SDK
Resolving API Semantics

Step 1: Generate the AST of the SDK to extract API Semantics

Step 2: Determine if an API is sensitive
- Inspect API method name
- Inspect API parameters
  - E.g., public void processTransaction(String creditCardNumber)

Step 3: Identify sensitive parameter
- Data Ontology

Step 4: Assign sensitivity as High, Medium, Low

---

Table 1: Accuracy of sensitive API identification

<table>
<thead>
<tr>
<th>SDK</th>
<th>Total API</th>
<th>FN</th>
<th>API Count After Filtering</th>
<th>API Reduction</th>
<th>TP</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>redis</td>
<td>15</td>
<td>0</td>
<td>9</td>
<td>40%</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>simplify</td>
<td>18</td>
<td>3</td>
<td>8</td>
<td>55%</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>tranzzo</td>
<td>26</td>
<td>2</td>
<td>9</td>
<td>43%</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>square</td>
<td>30</td>
<td>6</td>
<td>5</td>
<td>84%</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>payjp</td>
<td>47</td>
<td>3</td>
<td>6</td>
<td>88%</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

Graph: Reduction (%) in API count after filtering

41 SDKs in increasing order of API count before filtering
Payment SDK Analysis

SDK Data Set
• No specific market for hosting SDK
• 50 android Payment SDK
• SDKs collected from Google Pay and Apple Pay’s website

AARDroid Analysis
• We ran our 28 MASVS checks on 50 Payment SDKs
• 9 SDK code were obfuscated, 6 dataflow checks were skipped on these
• 24 SDK did not have a UI, 4 UI checks were skipped on these
• Average analysis time ~8 minutes

Manual Validation
• We manually validated 50 SDKs over a two-week period with JD-GUI, Fernflower
Highlighted Findings

- **3 SDKs** save unencrypted sensitive information such as credit card number and CVC. A fourth SDK stores a CVC in encrypted form.
  - PCI-DSS standards prohibit such storage.
- **11 SDKs** rely on outdated cryptographic primitives for sensitive functionality.
  - e.g., DES, Blowfish, SHA1, MD5
- **10 SDKs** do not follow industry standards on UI for taking credit card data from users.
  - 4 SDK display CVC in plain text
  - 3 SDK displays unmasked Credit card number (not reported)
  - 6 does both
Highlighted Findings

• **26 SDKs** use WebViews, all of which have at least one improper configuration, that introduces unnecessary attack surface.
  – 20 enables JS, 8 allows JS Bridge, 23 allow local file access, 21 do not clear WebView cache

• **37 SDKs** do not meet at least one of the basic (MASVS-L1) security requirements, **12 SDKs** do not meet at least one of the more advanced (MASVS-L2) security requirements.
Case Study

- Stores **unencrypted** credit card number
- Stores Encrypted CVC
- Stored **encryption key** on Shared preference **base64 encoded**
- Less secure key size (RSA 1024)
- UI leaks CVC and **un-masked** Credit card number
- Misconfigured WebView with all excess privileges
- **16 out of 28** MASVS requirements not met

Listing 1: Code snippet of persisting encrypted CVC in Dotpay

```java
public void addCreditStoreSecurityCode(String credit_card_security_code) {
    try {
        String secure_code = this.RSAEncrypt(credit_card_security_code);
        L.e("secure_code" + secure_code);
        this.sharedPreferences.edit().putString("credit_card_security_code", secure_code).commit();
        this.setOnClickCVVDataAvailable(true);
    } catch (NoSuchAlgorithmException var3) {
        var3.printStackTrace();
    }
}
```

Listing 2: Code snippet of insecurely instantiating and storing RSA key for CVC encryption

```java
KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
keyPairGenerator.initialize(1024);
KeyPair keyPair = keyPairGenerator.generateKeyPair();
PublicKey publicKey = keyPair.getPublic();
PrivateKey privateKey = keyPair.getPrivate();
this.sharedPreferences.edit().putString("publicKey", Base64.encodeToString(publicKey.getEncoded(), 2)).commit();
this.sharedPreferences.edit().putString("privateKey", Base64.encodeToString(privateKey.getEncoded(), 2)).commit();
```
Responsible Disclosure

**Goal:** Get feedback from SDK developers regarding the findings.

- Reached out **46** SDK vendors
- **27** acknowledged receiving it
  - 13 were auto-generated
- **7** vendors are investigating the issues (no further response)
- **1** claimed to release patches (BlueSnap)
- **1** claimed them *less severe* (Google Wallet)
- **1** claimed deprecated and informed affected merchants (PayPal)
- **4** acknowledged majority of the concerns (**26 out of 37**)

**Insights:**
- Some issues are *intentional business decision*
- Some issues are *not appropriate* for SDK
Summary

• Static analysis tool for analyzing stand-alone SDKs
• Evaluate 50 Payment SDKs against OWASP’s MASVS
• Identified several security weaknesses
• Identified gap between security standards and their implementation in payment SDKs
• Responsibly disclosed to SDK vendors
Questions

- Static analysis tool for analyzing stand-alone SDKs
- Evaluate 50 Payment SDKs against OWASP’s MASVS
- Identified several security weaknesses
- Identified gap between security standards and their implementation in payment SDKs
- Responsibly disclosed to SDK vendors

Samin Yaseer Mahmud
Ph.D Candidate, NCSU
Expected Graduation: May, 2023
smahmud@ncsu.edu
https://saminmahmud.com

I am looking for full-time roles in industry!