AutoCrypInject v1.0 : Automated Android Asset Security Framework

Author: Sarvesh Aadhitya, Shamalavannan, Karthik

July 30, 2025

Contents

1 Introduction

AutoCrypInject is a framework designed to increase the security of Android applications by automating internal asset encryption and injecting runtime decryption logic. The tool targets a common vulnerability in mobile applications: the storage of sensitive data, intellectual property, or proprietary libraries as plaintext files within the APK's `assets` directory. By making these assets computationally expensive to access without the application's intended logic, AutoCrypInject raises the barrier against static analysis, asset lifting, and reverse engineering.

---

2 Core Features

- **Automated APK Decompilation & Recompilation**: Using Apktool for seamless unpacking and rebuilding of APK files.
- **Selective Asset Encryption**: Allows the user to interactively select specific assets or all assets for encryption using the **AES-256 (CBC mode)** algorithm.
- **Dynamic Code Generation**: Generate a custom Java decryption module tailored to the selected assets and embedding the necessary cryptographic keys.
- **Smali Code Injection**: Converts the Java decryption module into Smali bytecode and intelligently injects it into the decompiled application's source.
- **Automated Bytecode Patching**: Scans the application's Smali code and automatically patches all invocations of `AssetManager.open()`, `AssetManager.openFd()` to redirect through the injected decryption logic.
- **Automated Signing**: Signs the final, repackaged APK with a debug key for immediate deployment and testing.

---

3 System Requirements

Proper functioning of the toolchain requires the following software:

- **Execution Environment**: Python 3.x
- **Python Library**: `pycryptodome`
- **Android Build/Reverse Tools**:
  - Java Development Kit (JDK) 8 or higher
  - `apktool.jar:decompile apk`
  - `android.jar:Android SDK Components`
  - `d8:Android DEX Compiler`
  - `baksmali.jar:DEX files into Smali code`
  - `uber-apk-signer.jar:APK signing tool supporting signature schemes`

---

4 Architecture and Workflow

The tool operates as a sequential pipeline, where the output of each stage serves as the input for the next. The entire process is orchestrated by the main Python script.

4.1 Architectural Flow

`Target APK` → [**1. Decompile**] → `Decompiled APK Files`

↓

`User Selection` → [**2. Encrypt**] → `Encrypted Assets`

↓

`Decryption java source` → [**3. Generate & Convert**] → `java-class-dex-smali`

↓

`Scan open(),openFd()` → [**4. Patch**] → `Modified functions`

↓

`Converted Decryption smali` → [**5. Inject**] → `Inserted Decryption smali`

↓

`Recompile Secured APK` → [**6. Rebuild & Sign**] → `signature & verification`

4.2 Detailed Workflow

1. **Initialization**: The script takes a target APK path as input.
2. **Decompilation**
   - An APK file is like a ZIP file that contains everything an Android application needs to operate: DEX file , manifest file, resources, assets, native library, etc.
   - APK is disassembled into a folder structure containing Smali code, resources, and assets using **apktool**.
3. **Asset Encryption Processing**:
   - The tool lists all files in the `assets/` directory.
   - The user selects which assets to encrypt.
   - Each selected asset is read, encrypted with AES-256, and overwritten.
   - A 16-byte IV is prepended to the ciphertext.
4. **Decryption Module Generation**:
   - A `.java` file is generated containing a class `EncryptedAssetRegistry`.
   - This class holds the AES key and a list of encrypted filenames.
   - It exposes static methods `open()` and `openFd()` that act as proxies for the standard `AssetManager` calls, handling decryption transparently.
5. **Smali Transformation**: The generated `.java` file undergoes 3 step conversion:
   (a) `javac`: Java source → Java bytecode (`.class`)
   (b) `d8`: Java bytecode → Dalvik bytecode (`.dex`)
   (c) `baksmali`: Dalvik bytecode → Smali source (`.smali`)

6. **Injection and Patching**:
    - The new Smali files are moved into the decompiled APK's Smali directory.
    - The script traverses all existing Smali files, searching for specific method call patterns.
    - Calls to `AssetManager.open()` and `AssetManager.openFd()` are rewritten to invoke the static methods in `EncryptedAssetRegistry`.
7. **Rebuild and Sign**
    - The modified application structure is recompiled into an APK using **apktool**
    - The resulting package is signed with **Uber**which is signing tool supporting signature schemes v1 (JAR), v2 (APK Signature Scheme), v3, and v4.
    - It simplifies the post-rebuild APK signing and verification process.
    - Finally, A **secured APK** is ready to launch.

---

5 Core Component Analysis

5.1 Cryptography Module

This function implements **AES-256** in **Cipher Block Chaining (CBC)** mode with **PKCS7 padding**. A new, cryptographically secure 16-byte Initialization Vector (IV) is generated for each file to ensure that identical plaintext assets produce unique ciphertexts. The final encrypted file structure is:

$$\text{EncryptedFile} = \text{IV (16 bytes)} || \text{AES-256-CBC(PaddedData)}$$

5.2 Code Generation and Injection

- `generate_encrypted_registry_java()`: Creates the Java source for the decryption logic. The encrypted file list is hardcoded into a `HashSet` for efficient runtime lookups ($O(1)$ average time complexity).
- `convert_java_to_smali_and_inject()`: Orchestrates the multistep conversion from Java source to Smali and places the final files in the target directory. `decoded`$_{a}pk/smali/com/decryptassetmanager/$.

5.3 Smali Patching Engine

The patching mechanism is driven by regular expressions and code analysis.

- `find_and_patch_asset_open()`: Using regex to find calls to the `open` method It captures the invocation type (e.g., `invoke-virtual`) and the registers, then rewrites the line to an `invoke-static` call to our injected method.
- `scan_and_patch_open_fd()`: Patches calls to `openFd`. This is more complex as our wrapper requires a `Context` object. The `find_context_register()` helper function uses heuristics to find the register holding the context:

1. If the class is a known Context subclass (e.g., `Activity`, `Service`), it assumes `p0` (the `this` reference).
2. It checks the method's parameters for a `Context` type.
3. It performs a reverse search from the call site for an `invoke-virtual ... ->getAssets()` call to identify the object from which the `AssetManager` was retrieved.

---

6 Setup and Usage

6.1 Configuration

Before first use, ensure the paths to the required tools are correctly set as global constants at the top of the script:

```
APKTOOL_PATH = 'apktool.jar'
AES_KEY = bytes.fromhex("...") # Change for production use
# Paths for d8, baksmali, and uber-apk-signer may also need adjustment
```

Security Warning: The default `AES_KEY` is a placeholder. For any real-world application, this key **must** be replaced with a unique, securely generated key. Note that storing the key within the APK itself is a form of obfuscation, not true security.

6.2 Execution

usage:

```
python main.py /path/to/my_app.apk
```

1. Run the script from your terminal, passing the path to the target APK file as the sole command-line argument.
2. The script will display the available assets and prompt for input. You can provide a comma-separated(**,**) list of filenames to encrypt every asset or type **all** to encrypt all assets.
3. Patching Smali files to decrypt assets at runtime transparently
4. Creating custom class using Java and converting java → Class → Dex → Smali
5. Rebuilding the APK and Signing with Uber for verification.
6. Finally, the Secured APK is ready to Launch.

.

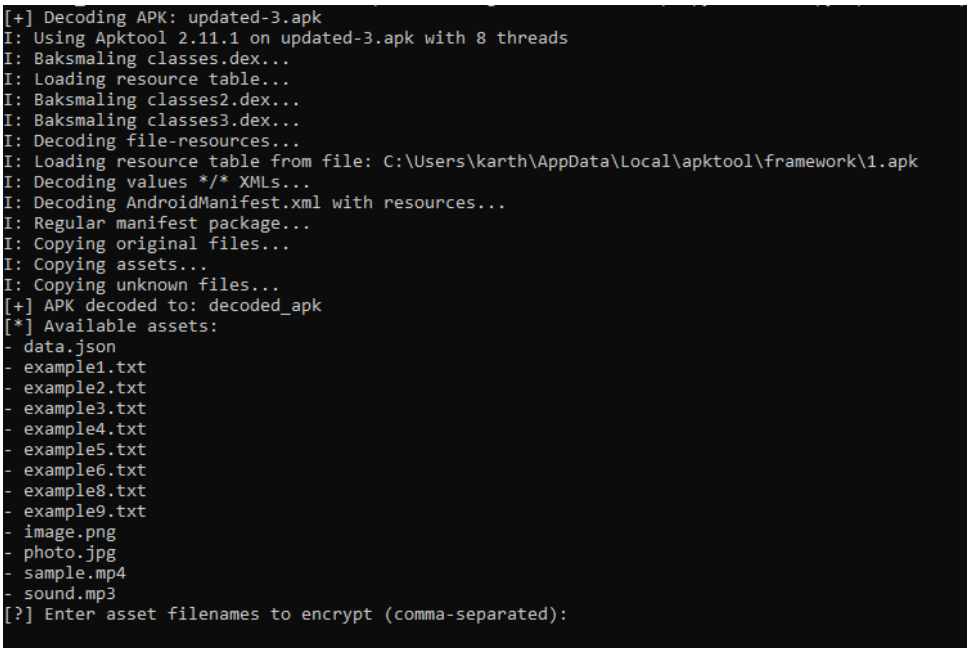

Figure 1: Decompilation

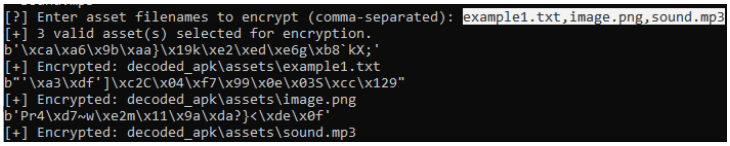

Figure 2: User Selection

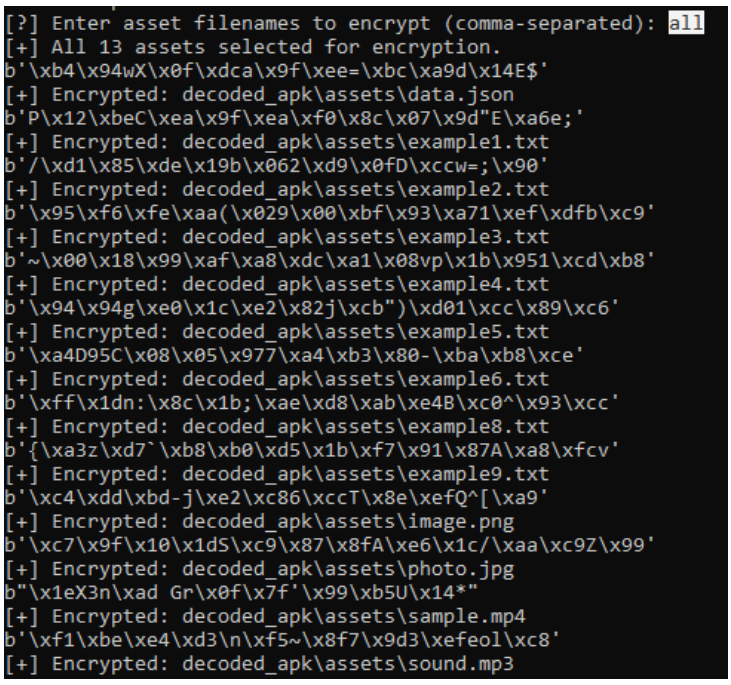

Figure 3: if all files needs to Encrypt

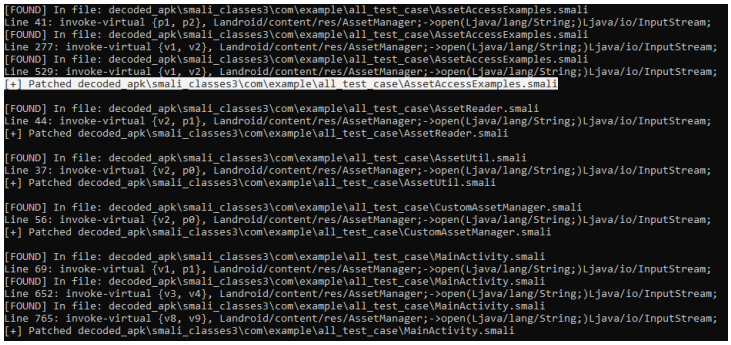

Figure 4: Patched open function

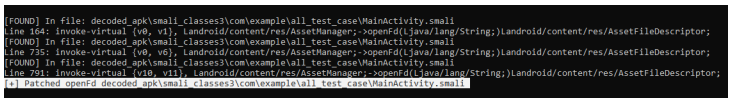

Figure 5: Patched openFd function

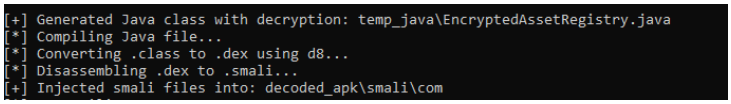

Figure 6: converted java to smali

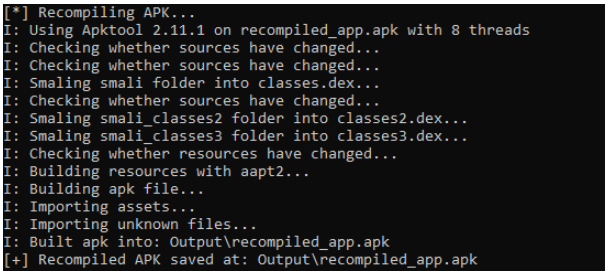

Figure 7: Recompiling the APK

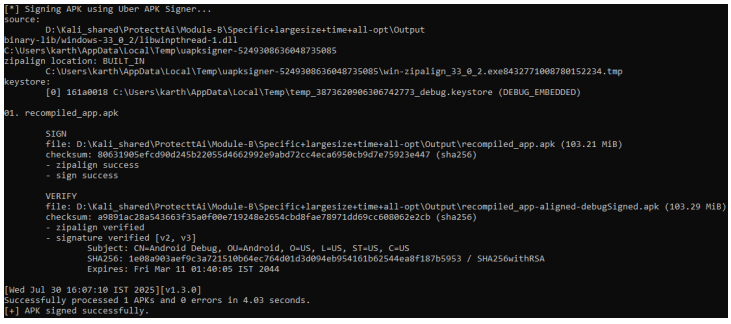

Figure 8: Signing the APK