Lab 6: Insecure Storage for iOS - iOS Static RE

Shivali Mate

I. Introduction

THIS lab focused on analyzing the security of an iOS application through static reverse engineering. The main objective was to identify insecure storage practices and sensitive data exposure within the provided IPA file, B649_InsecureStorage.ipa By exploring the contents of the IPA package, inspecting property list files (plists), and analyzing the compiled binary, the goal was to understand how insecure storage can reveal secrets or personal data to attackers. The exercise also provided hands-on experience with standard reverse engineering utilities on a Linux environment and offered insight into secure iOS development practices.

II. Methodology

This section describes the step-by-step static reverse-engineering process performed on the provided B649_InsecureStorage.ipa file. Each subsection corresponds to the specific lab tasks, including commands executed, intermediate outputs, and interpretations.

A. Extracting the Application Files

The provided B649_InsecureStorage.ipa file was first extracted using the unzip utility to view the internal layout. The unzipped folder contained the directory Payload/B649_InsecureStorage.app/. A further listing of the directory confirmed the structure:

```
find . -maxdepth 2 -print
```

The output revealed the standard iOS bundle layout containing Info.plist, Hints.plist, PL.plist, _CodeSignature/CodeResources, and the main executable B649_InsecureStorage. This directory structure as in Fig 1 matches the conventional organization of iOS app bundles, where the .app folder holds all resources, executables, and configuration data.

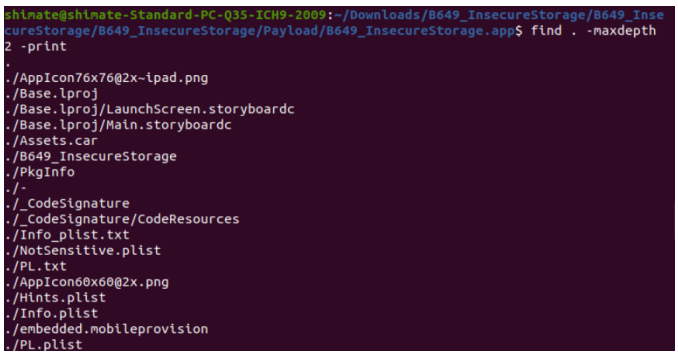

Fig. 1. Extracted application directory structure

B. Identifying the Application Binary

To locate and verify the compiled application binary, the file command was executed:

```
file B649_InsecureStorage
```

This verified that the extracted file is indeed the iOS executable compiled for the ARM64 architecture as in Fig 2. The presence of the Mach-O format indicates that this is the binary the iOS operating system executes.

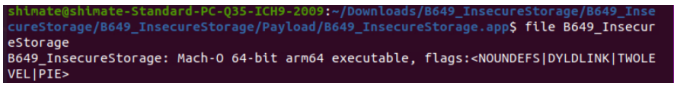

Fig. 2. Identifying the Application Binary

C. Discovering Stored HTTP/HTTPS URIs

To search for possible hard-coded web endpoints or API links, recursive grep and strings commands were used:

```
string B649_InsecureStrorage |
grep -E "http|https"
```

The commands found only Apple-related documentation URLs such as http://www.apple.com/DTDs/PropertyList-1.0.dtd and https://www.apple.com/certificateauthority/0

No custom endpoints or third-party APIs were discovered. As shown in Fig 3, this indicates that the app does not embed insecure HTTP links or transmit data over unencrypted channels.

Fig. 3. URL found in the files

D. Inspecting Permissions and Purpose Strings

The Info.plist file was converted using Python's plistutil module to an text file using the commands

```
plistutil -i Info.plist >
Info\_plist.txt
```

then grep was used to identify declared permissions which is shown in Fig. 4

Two permission keys were identified:

- **NSCameraUsageDescription:** "We need to access the camera to scan QR codes"

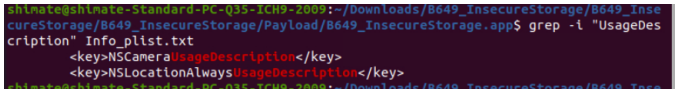

Fig. 4. Camera and Location usage keys

- **NSLocationAlwaysUsageDescription:** "We need to access the location to find the Luddy School"

This information confirms that the developer included proper purpose strings as required by iOS security guidelines.

E. Finding the Secret Associated with "B649Secret"

The next step involved scanning for any sensitive data or secret keys stored in property list files:

```
plistutil -i PL.plist -o -
grep -R "B649Secret" .
```

The analysis revealed the following entry:

```
<key>B649Secret</key>
<string>B649_flag_2025</string>
```

The plaintext flag was found directly inside a plist file. Storing secrets in plist files is insecure because they are part of the app bundle and can be read by anyone who extracts or inspects the IPA. Such values should instead be stored securely in the iOS Keychain or fetched from a server at runtime. This finding successfully demonstrates the main vulnerability targeted by the lab, insecure storage of sensitive data.

F. Open-Ended Exploration and Additional Findings

Further exploration was performed on the application binary and configuration files using Unix commands and static-analysis techniques.

Bundle Structure: The find command (Fig 1) revealed key files, including configuration plists, image assets, and compiled storyboard files, confirming the internal organization of an iOS IPA.

Binary Inspection:

```
strings B649\_InsecureStorage |
egrep -i 'NSUserDefaults|
CFPreferences|Keychain'
```

identified the string and a note referencing the storage path.

```
_OBJC_CLASS_$_NSUserDefaults
found at location
/var/mobile/Containers/Data/
Application/<UUID>/
Library/Preference
```

This proves that the application likely uses NSUserDefaults for local data persistence, a method known to be insecure for storing confidential information.

Transport Security Configuration: The following command checked the app's network-transport settings:

The output returned <no ATS key>, implying that App Transport Security (ATS) is enabled by default, enforcing HTTPS communications.

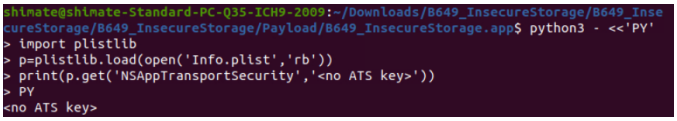

Fig. 5. App Transport Security key check

III. Tools and Learning Experience

The following tools were used during the lab:

- **strings:** extracts printable ASCII strings from binary files to reveal hidden URLs, keys, or text constants.
- **plistutil / plistlib:** parse binary or XML-formatted property list files used extensively in iOS apps to store configuration and user data.
- **grep:** searches recursively through files for specified patterns such as http or keyword names.
- **file:** identifies file type and architecture to confirm that a binary is a Mach-O executable.
- **Python scripting:** provided an alternative to parse plist files when command-line tools failed.

IV. Findings and Discussion

The analysis revealed that the IPA file followed the standard iOS structure containing the app payload, signature, and provisioning profile. Examination of the plist files exposed a plaintext secret `B649_flag_2025` and confirmed the use of `NSUserDefaults`, both of which represent insecure storage practices. These files can be easily extracted through static reverse engineering, highlighting the risk of exposing sensitive information.

The study reinforced the importance of using secure mechanisms like the iOS Keychain or backend token retrieval. While the static tools used were effective, future work could integrate automation or visualization tools to streamline analysis and correlate static results with runtime behavior.

V. Conclusion

This lab demonstrated how static reverse engineering can uncover insecure storage within iOS apps. Using simple Unix tools and Python scripts, hidden data and permission configurations were recovered from the app bundle. The key finding was the plaintext secret stored in a plist, underscoring weak data protection. Overall, the lab enhanced understanding of IPA file structures, insecure storage risks, and the need for secure data handling in mobile application development.

References

[1] Apple Developer Documentation, *Information Property List Files*. https://developer.apple.com/documentation/bundleresources/information_property_list/managing_your_app_s_information_property_list

[2] Sanfoundry, *strings Command Usage Examples in Linux*. https://www.sanfoundry.com/strings-command-usage-examples-in-linux/

[3] Debian Manpages, *plistutil - Convert Property List Files*. https://manpages.debian.org/experimental/libplist-utils/plistutil.1.en.html

[4] PhoenixNAP, *Linux file Command Explained*. https://phoenixnap.com/kb/linux-file-command

[5] University of Utah, *UNIX Command Reference*. https://www.math.utah.edu/lab/unix/unix-commands.html