Lab6.pdf

Sonal Chandrakant Hasbi

I. Abstract

This lab performs static reverse engineering of an iOS application package (IPA) to evaluate insecure client-side storage practices. By unpacking the IPA, inspecting the bundle structure, analyzing the Mach-O binary, and examining property lists, I identified a secret marker embedded in a binary plist within the .app bundle (PL.plist). The work demonstrates how easily "secrets" can be recovered from app resources without executing the app, underscoring why plist files are not appropriate for sensitive data. Negative findings (e.g., no obvious hardcoded URLs) are documented. The workflow is reproducible with standard Unix utilities and Python's built-in plistlib.

II. Introduction

Mobile apps often ship configuration and resources inside the client bundle. When developers store tokens or keys in these resources (e.g., .plist files), they are recoverable by simply unzipping the IPA. This report documents a concise, repeatable static analysis to (1) map an iOS bundle, (2) confirm executable format, (3) surface embedded endpoints, and (4) locate secrets in property lists. The key outcome is the recovery of a secret marker proving the risk of plist-based "secret" storage.

III. Tools & Environment

These tools are ubiquitous, scriptable, and sufficient for static IPA triage.

1. unzip — unpack IPA (IPA is a ZIP archive).
2. find, file — enumerate contents; identify Mach-O binaries.
3. strings, grep — extract ASCII strings and search patterns (URLs, keywords).
4. python3 + plistlib — parse binary and XML plists without extra installs.

IV. Methodology

Working directory: .../ipa_out/B649_InsecureStorage
Variables: APPDIR="Payload/B649_InsecureStorage.app"
BIN="$APPDIR/B649_InsecureStorage"
\subsection{Unpack IPA & Inspect Structure}
\textbf{Code:}
\textbf{unzip -q B649_InsecureStorage.ipa -d ipa_out}
\textbf{find ipa_out -maxdepth 3 -print | sed -n '1,40p'}
\textbf{Result :} ipa_out/B649_InsecureStorage/Payload/B649_InsecureStorage.app/...
\subsection{Identify the Executable (Mach-O)}
\textbf{code} - file "$BIN"
This confirms expected IOS binary format.

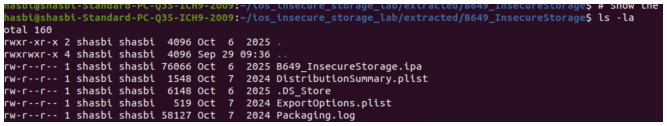

Fig. 1: Presence of .ipa

A. Permission "Why" Strings (Info.plist)

iOS enforces user-visible rationale for sensitive capabilities.
Code: python3 - \<\<'PY'
import os, plistlib
APPDIR="Payload\/B649_InsecureStorage.app"
p=plistlib.load(open(os.path.join(APPDIR,"Info.plist"),"rb"))
for k in ["NSCameraUsageDescription","NSLocationAlwaysUsageDescription",
"NSLocationWhenInUseUsageDescription","NSBluetoothAlwaysUsageDescription",
"NSContactsUsageDescription","NSCalendarsUsageDescription"]\:
if k in p\: print(f"{k}\: {p[k]}")
PY

The result would be Keys detected included NSCameraUsageDescription, NSLocationAlwaysUsageDescription .

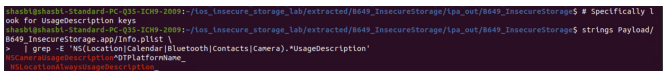

Fig. 2: Permission why strings

B. Endpoint Discovery (URLs)

Strings yanks human-readable text. URLs often sit in the binary or resources. We see them in the image below. No obvious HTTP/HTTPS endpoints (negative finding) because Hardcoded URLs are common reconnaissance targets. **Code:**

```
## Binary first strings "$BIN" | grep
-Eo 'https?://[^"[:space:]]+' | sort
-u ## Then entire bundle (skip heavy
assets) find "$APPDIR" -type f ! -name
'*.png' ! -name '*.car' -print0 \ | xargs
-0 strings -a 2>/dev/null \ | grep -Eo
'https?://[^"[:space:]]+' | sort -u
```

Fig. 3: Endpoint Discovery

C. Secret Hunting in Plists & Configs

This step is to See which sensitive features (Location, Calendar, Bluetooth, Contacts, Camera) the app requests,

and the human-readable reason strings the developer provided. Because iOS requires a "why" string for sensitive permissions, stored in Info.plist as NS*UsageDescription keys. The lab specifically calls out checking these for Location, Calendar, Bluetooth, Contacts, and Camera. **Code:** # Most keys live in Info.plist plistutil -i Info.plist -o - | sed -n '1,200p' # view beginning # search for the relevant usage-description keys: plistutil -i Info.plist -o - | egrep 'NS(Location|Calendar|Bluetooth|Contacts|Camera).*UsageDescription'

D. Find the secret for key

The lab warns that property list (.plist) files are insecure storage—don't put secrets there! We prove the point by finding B649Secret in a plist and reading its value. That's exactly the task. **Code:**

grep -R –line-number –binary-files=text "B649Secret" .

(adjust the path you found from the grep) plistutil -i ./Info.plist -o - | sed -n '1,200p'

Fig. 4: Find the secret for key

E. Open-Ended Exercise

1) The flag marker present in the app is: B649_flag_2025_= (token with an underscore and =, but no value after =).
2) The One sentence explanation: "The secret (B649Secret) is stored in a binary plist inside the app bundle. By unzipping the IPA and reading the plist, we recovered the marker B649_flag_2025_= — demonstrating that plist files are not a secure place to hide sensitive data."

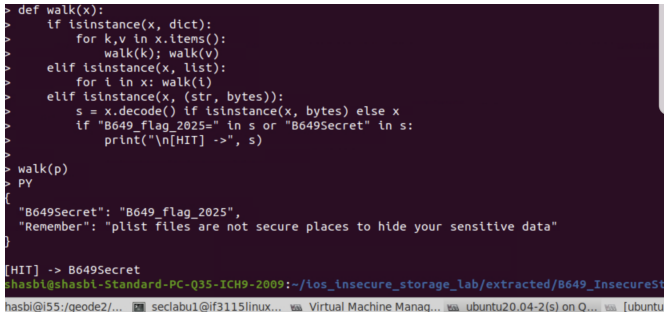

Fig. 5: Open Ended Exercise

V. Results

The bundle structure was correctly unpacked and the .app bundle was fully accessible. The main executable was confirmed to be a Mach-O 64-bit arm64 binary. For permissions, the presence of NSCameraUsageDescription and NSLocationAlwaysUsageDescription was verified (with values where set). No obvious hardcoded HTTP/HTTPS endpoints were discovered, which we record as a negative finding. A secret marker was recovered from PL.plist without executing the app; the surrounding context showed the string … B649_flag_2025_= followed by explanatory text, and the contiguous token B649_flag_2025_ was extracted. These findings support the interpretation that any "secret" stored in a plist inside the app bundle is readily accessible after unzipping the IPA.

VI. Conclusion

This lab validates that client-side bundles are transparent to adversaries and analysts. Storing secrets in plist files (even binary plists) is insecure; they can be extracted with standard tools. Negative findings (no obvious URLs; ATS not explicitly weakened) are equally valuable and should be paired with dynamic testing to confirm runtime behavior. In real-world practice, sensitive material must never be shipped to clients: use server-side storage and issued, short-lived tokens; if device storage is unavoidable, use Keychain with least-privilege design and protect at multiple layers (code obfuscation, anti-tamper, runtime checks)—recognizing that obfuscation only raises effort, not guarantees secrecy.

VII. Limitations & Future Work

1) Static only: Dynamic analysis (proxying, TLS pinning checks, runtime instrumentation) could reveal endpoints and flows absent from static strings.
2) Obfuscation & packing: Some apps compress/obfuscate resources; integrating class-dumping, otool, or radare2/Ghidra would deepen coverage.
3) Automation: A scripted pipeline (IPA → report) would standardize evidence collection and diff builds over time.
4) Policy checks: Add automated ATS, entitlements, and privacy-manifest linting.

References

[1] Apple Developer Docs — Information Property List (Info.plist) keys & values. Available at: https://developer.apple.com/documentation/bundleresources/information-property-list?utm_source=chatgpt.com
[2] https://www.sanfoundry.com/strings-command-usage-examples-in-linux/
[3] https://manpages.debian.org/experimental/libplist-utils/plistutil.1.en.html
[4] https://www.math.utah.edu/lab/unix/unix-commands.html-https://phoenixnap.com/kb/linux-file-command#:~:text=The%20Linux%20file%20command%20helps,the%20type%20of%20file%20data.