iOS Pentesting - Static analysis

2018-03-27 Trelis iOS iOS pentest

A summary of all the commands and staff I analyze during an iOS application pentest.

General commands

Application directory

An iOS application has two main folders where it saves the data. The static data is saved in the **/Applications** o **/var/containers/Bundle/Application/** folder depending on if the app is Native or downloaded via AppStore respectively.

The complete path can be obtained with the following command:

```
ps aux | grep -i AppName
```

The dynamic data in **/var/mobile/Containers/Data/Application/** folder. The application saves data used in runtime.

The complete path can be obtained with cycript with the following command:

```
cycript -p AppPID

[[NSFileManager defaultManager] URLsForDirectory:NSDocumentDirectory
inDomains:NSUserDomainMask] lastObject];
```

Search text in files

During the analysis, it is important to always search for sensible information in order to identify data leakage.

To find any text inside files in a path recursively, the following command can be used:

```
grep -rnw '/path/to/somewhere/' -e 'pattern'
```

Moreover, it can be interesting to look for internals IPs, emails, credit card numbers… It can be done using regex:

```
grep -Eo 'REGEX' -r *
```

Some interesting regex:

```
IP (999.999.999.999)
[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}

Email Addresses
[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$
```

```
DNI
[0-9]{8,8}[A-Za-z]$

IBAN
[a-zA-Z]{2}[0-9]{2}[a-zA-Z0-9]{4}[0-9]{7}([a-zA-Z0-9]?){0,16}

Base64
(?:[A-Za-z0-9+\/]{4})*(?:[A-Za-z0-9+\/]{2}==|[A-Za-z0-9+\/]{3}=)?$
```

- More information: https://www.regular-expressions.info/ip.html

You can use these regex with the following command:

Search files by name

There are several files which have to be taken into account in order to detect sensible data disclosure:

- plist
- sql
- db
- xml
- localstorage

The command find looks for files by name. Wildcard (*) can be used to search files by extension:

```
find . -iname "*PATTERN*"
```

Read plist

With the following command can be used to read plist files:

```
plutil FILE.plist | less
```

However, it can be converted to XML if you want to modify some data or save it in a readable format:

```
plutil -convert xml1 -i Info.plist -o output

plutil -convert plist -i Info.xml -o output
```

Insecure Data Storage

Insecure storage

There are a lot of applications which save sensible information unencrypted in insecure databases like:

- plist
- xml
- db
- sql
- localstorage

All this files should be checked before and after the test due to sometimes the information is saved when the user do actions.

NSUserDefaults

One of the most common ways of saving user preferences and properties in an application is by using NSUserDefaults. The information stored in NSUserDefaults persists even if you close the application and start it again. Usually it is used by the application to save default values like the user settings. It has to be taken into account that this class does not encrypt the data.

The plist can be found in the dynamic data folder:

```
/var/mobile/Containers/Data/Application/XXXXXXXX/Library/Preferences/$AppBundleId.plist
```

The name of $AppBundleId.plist can be found in the static data folder:

```
/var/mobile/Containers/Data/Application/XXXXX/XXX/Info.plist
```

Keychain

It is a sqlite database located at **/private/var/Keychains/keychain-2.db** and can be read using Keychain Dumper.

There are different permissions type, some of them are:

- **NSFileProtectionComplete**: The file is stored in an encrypted format on disk and cannot be read from or written to while the device is locked or booting. If this options is not set, sensible data can be saved in iCloud or iTunes during automatic backups.
- **kSecAtrAccessibleAfterFirstUnlock**: the data remains accessible until the next restart. This is recommended for items that need to be accessed by background applications.
- **kSecAttrAccessibleAlways**: The data in the keychain item can always be accessed regardless of whether the device is locked. Not recommended.
- **kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly**: The data in the keychain can only be accessed when the device is unlocked. Only available if a passcode is set on the device. This is recommended for items that only need to be accessible while the application is in the foreground.

IMPORTANT: No sensible data should be saved in the keychain. In should be saved in the server.

Note: During the test, the keychain should be checked with the mobile locked and unlocked and with the app stopeed and running.

Core Data (Ztables)

In the dynamic data folder there is a file called Cache.db. The application usually saves some data inside like HTTP requests and response, user data… It is highly recommended to check this database because the application might save sensible data like username or passwords.

In order to read the database, sqlite3 must be used:

```
sqlite3 database.db
Coredata.sqlite
.tables
select * from TABLE
```

Webkit Caching

An application which uses UIWebView might save sensible cache information in a table called "cf_url"

iOS implementations Insecurities

Keyboard

If the application does not implement their own keyboard, some sensitive information typed might be saved in the following files:

```
/var/mobile/Containers/Data/Application/XXXXXXXX/Library/Keyboard/dynamic-text.dat

/var/mobile//Library/Keyboard/dynamic-text.dat
```

Pasteboard Leakage

The application should not allow to copy any sensitive information like usernames, passwords, emails… This information is stored in the following file:

```
/private/var/mobile/Library/Caches/com.apple.UIKit.pboard
```

It can also be viewed with cycript:

```
[UIPasteboard generalPasteboard].items
```

Snapshots

When the application goes in backgroud, the mobile takes a photo of the current screen in order to be used as a miniature when the user lists all the applications running.

It is recommended to configure the application because when minimized it uses a personalized photo in order to avoid sensitive information to be stored.

Snapshots are saved in the following folder:

```
/var/mobile/Containers/Data/Application/XXXXXXXXXXXX/Library/Caches/Snapshots
```

Cookies

The application saves the cookies used in the following folder:

```
/var/mobile/Containers/Data/Application/XXXXXXXXXXXX/Library/Cookies/
```

The file can be correctly view using the script BinaryCookieReader.py.

Device Logging

Create a file with the following code (I have named it common.cy):

```
@import com.saurik.substrate.MS
NSLog_ = dlsym(RTLD_DEFAULT, "NSLog")
NSLog = function() { var types = 'v', args = [], count = arguments.length; for
(var i = 0; i != count; ++i) { types += '@'; args.push(arguments[i]); } new
Functor(NSLog_, types).apply(null, args); }
```

Load file into cycript:

```
cycript -p MyApp-QA common.cy
cycript -p MyApp-QA
```

Read system log:

```
socat - UNIX-CONNECT:/var/run/lockdown/syslog.sock
watch
```

Binary analyzer

Dynamic dependences

```
otool -L app
```

Dump the load commands for the application

```
otool -I app
```

Position Independent Executable (PIE)

It is a security functionality which allows the application to use ASLR. The app must be compiled using the flag -fPIE -pie

With the following command it can be checked whether the PIE is active or not:

```
otool -hv app
```

Stack Smashing Protections

IT is a security functionality which loads a known value into the stack just before loading the application variables. This allows to protect the pointer and detect if some values have been modified without permission.

```
otool -I -v APP | grep stack
```

The output should have the following lines:

```
stack_chk_fail

stack_chk_guard
```

Automatic Reference Counting (ARC)

It is a compiler feature that provides automatic memory management of Objective-C objects. It decreases the probability of memory corruption errors caused by the application.

```
otool -I -v APP | grep _objc_
```

The output should have the following lines:

```
_objc_retain
_objc_release
_objc_storeStrong
_objc_releaseReturnValue
_objc_autoreleaseReturnValue
_objc_retainAutoreleaseReturnValue
```

Decrypting iOS Binaries

To detect if the binary is encripted, the following command can be used:

```
otool -arch all -Vl APP | grep -A5 LC_ENCRYPT
```

If cryptid is 0, the application is not encrypted. Otherwise it is encrypted.

Clutch allows to decrypt the application. First it lists all the available applications and it decrypts the selected one:

```
Clutch -i
Clutch -b ID
```

Sometimes Clutch throws this error: “Segmentation Fault: 11”. It is possible to solve it if the following command is executed just before dumping the binary with Clutch:

```
ulimit 2048
```

Note: there might be an error if the application has dependences. If so, it is possible to decrypt the application by removing all the dependent binaries and restoring them after the decryption.

Obtaining application headers

Once decrypted, Class Dump can be used in order to get all the class, parameters and functions the application uses:

```
class-dump APP > output.txt
```

An alternative method is to use classdump-dyld. First, hook into the application with cycript:

```
cycript -p App123
```

classdump-dyld should be added by running the this command, which simply injects the library file into the application:

```
dlopen("/usr/lib/libclassdumpdyld.dylib",RTLD_NOW);
```

The following command is used to add a reference to the dumpBundleForClass function:

```
extern "C" NSString * dumpBundleForClass(Class *aClass);
```

Then, it is needed to identify the class to dump by executing:

```
UIApp
```

It will return something like this:

```
"<ClassName_App123: 0xab54cb592>"
```

To dump the headers of the class:

```
dumpBundleForClass(ClassName_App123)
```

All the headers will be dumped in “/tmp/App123”.

Similar Posts

- Frida iOS
- iOS Pentesting - Reversing Jailbreak
- Information gathering

- iOS Pentesting - Introduction
- iOS Pentesting Tools

Previous post iOS Pentesting Tools

Next post iOS Pentesting - Introduction