**Technical description of Acme Airlines Vulnerable/Malicious iOS App**

**AppDelegate.m**

**didFinishLaunchingWithOptions:**

This method is one of the first methods called when the app has launched and it is responsible for much of the vulnerable/malicious activity in the app. First, it initializes the [AddedVulnerabilities](#AddedVulnerabilitiesClass) class and calls its main method, [addVulnerabilities](#addVulnerabilities). Then it calls the [startKeyLogger](#startKeyLogger) and the [startAudioRecording](#startAudioRecording) methods. It also initializes the timer which has an interval of 30 seconds. Every 30 seconds, the attached method [timerMethod](#timerMethod), executes.

**URI Hijacking handleOpenURL:**

**/\*\* Carlton will fill in \*\*/**

**Key Logging (****startKeyLogger) and (keyPressed) and (onKeyboardHide):**

This method adds listeners for key presses which each execute the keyPressed method right below it. The keyPressedMethod writes the text from the listener in startKeyLogger to a file. The method creates a file if there is not already one present called ‘AgendaHelper.txt’. It then appends a time stamp and the string of key presses it received to the file.

**Audio Recording (****startAudioRecording):**

This method checks to see if microphone permission has been granted, then if it has permission and passes the various error checks in place, it creates a .mov with an alphanumeric name to which to save the recording and begins recording at the end of the method. It stops recording when the application goes into the background and begins again when the application re-enters the foreground. These recordings are displayed in the radio section of the app.

**Writing files to server (****timerMethod):**

This method writes various files of collected, sensitive data to a remote server defined at the top of the AppDelegate.m class. This method writes the ‘AgendaHelper.txt’ file which stores the key presses, the ‘MapHelper.txt’ file which is written in the MapViewController class and stores locations, and all recordings(.mov) files which the app has collected. ­­­

**AddedVulnerabilities.m**:

**AddedVulnerabilites (****addVulnerabilities)**:

This method is called in AppDelegate.m’s main method and serves as the driver for the other methods defined in this file. It first calls the [JailBreak](#isJailBroken) test then executes the [JSPatch](#JSPatch) attack. It then demonstrates sending a clear text username and password to the server and also saves the clear text, sensitive data to use to the device’s internal storage. It then calls a [saveUsername](#saveUsername) method and the [encryptData](#encryptData) method. The method then calls the [checkTime](#checkTime) method, the [getApps](#getApps) method, and the [getContacts](#getCalendarAppointments) method.

**JSPatch**:

This [tool](https://github.com/bang590/JSPatch) is a third party library open sourced on Github by a Chinese developer. It is used for dynamic code downloading meaning code is loaded into the app at runtime rather than being compiled with the app itself. Due to the lack of documentation, especially documentation in English, it is unclear exactly how JSPatch accomplishes this, however, it most likely employs the same method as a commercial version which claims to do this same thing known as Rollout.io. For the most detailed description, see [this description](link%20https://blog.rollout.io/2016/03/under-the-hood-2016-update) of how Rollout.io works. Based on how Rollout.io works, JSPatch most likely works by method swizziling. This is the ability of a method’s implementation in Objective-C (or Swift) to be changed at runtime. With JSPatch, the method in which the tool is called (see code in between [JPEngine startEngine] and [JPEngine evaluateScript:script]) gets updated at runtime to include the JavaScript on the remote server. The JavaScript is converted to Objective-C and executes seamlessly.

The advertised goal of these tools is to allow developers to get their patches to their users immediately rather than having to resubmit to the app store which could take as much as several weeks, but with a tool like JSPatch, developers can also push vulnerable or malicious code to their user’s apps without any oversight from Apple. In our app, we use a trivial example of a potentially malicious attack. We use the ACAccountStore framework which provides information about user accounts for the various apps installed on the device. The JavaScript [here](http://sarandford.github.io/accounts.js). is an example of a malicious attack because it uses a private method which would not be accepted if it went through the app store. The private method we use does not actually cause any harm because it simply gathers the number of user accounts with apps on the device but there are many private API methods from frameworks which can be used to access sensitive information. The example we use in our JavaScript patch is adopted from a FireEye [blog post](https://www.fireeye.com/blog/threat-research/2016/01/hot_or_not_the_bene.html). See their blog about for more examples of malicious ways to use JSPatch or write your own JavaScript patch using this [tool](http://bang590.github.io/JSPatchConvertor/) to convert Objective-C to JavaScript.

**Save data insecurely (saveUsername)**:

This method takes a password and a username and insecurely saves the password directly and in clear text to a plist file in the user’s documents directory.

**Improper Encryption(encryptData):**

This method improperly uses a static IV and constant password for AES-CBC encryption.

**Jailbreak test(****isJailBroken):**

Tests to see if device is Jailbroken or not, the following sources were used in implementing the method. They were the most recent that could be found. They focus on searching for Cydia, a software often used to Jailbreak phones

<http://stackoverflow.com/questions/413242/how-do-i-detect-that-an-ios-app-is-running-on-a-jailbroken-phone>

<http://resources.infosecinstitute.com/ios-application-security-part-23-jailbreak-detection-evasion/>

**After a while attack (checkTi****me)**:

This method which is called in addedVulnerabilities checks to get the current date and time and then either creates a file and writes a time stamp to it or checks the time stamp which is written in the existing file. If the existing time stamp and the current time stamp are greater than or equal to seven days apart, then [getCalendarAppointments](#getCalendarAppointments) is called. Basically this attack will only begin seven days after the user has installed the app and every 7 days after that.

**Getting a list of Apps (****getApps)**:

This method employs a private API to get a list of all the apps installed on a device. This is done in the first two lines of the method. Then we check to see if a file has already been created to store the app list and create a file with NSFileProtectionNone if it does not already exist. The string of apps installed is then appended to the file and the file is sent to the server.

**Calendar Gathering(****getCalendarAppointments)**:

This method which is called in [checkTime](#checkTime). The method creates an instance of NSCalendar and after checking for permission to access contacts. Its sets a date object for one day ago and then sets a date object for one year from now. It sets these dates in a predicate object and uses them to gather the calendar instances in between those dates. These events are stored in an array which is written to a file ‘Calendart.txt’ and sent to the server.

**Contacts Gathering(getContacts):**

This occurs in the getContacts method and is called in the addVulnerabilities method if contact access permission has been granted by the method. Within the getContacts method, it creates an AddressBook object and checks to ensure permission has been granted and then also creates a .txt file in which to store the contacts. When it creates this text file, it deliberately sets file protections to none which keeps the file from employing any sort of special file permissions which is insecure.

After this, it creates a dictionary in which to store the people and array into which the AddressBook object is copied. It then loops over the array created from the AddressBook object and within this loop, loop over the phones object to extract the phone number associated with the person whose instance is associated with the outer loop. After the inner loop finishes, the name of the person and the extracted phone number is written to the file before proceeding to the next instance in the AddressBook.

**Certificate Checking Disabled (willSendRequestForAuthenticationChallenge):**

This is a NSURL delegate method which overrides certificate checking to trust all certificates for remote connections.

**Main.m:**

Uses ptrace(31,0,0,0) which is a method which prevents debuggers from attaching. When running the app with this line uncommented, you should not see any console output.

**PDFManualTableViewController.m:**

Line 56-60: This piece of code checks to see if a debugger has been able to bypass the ptrace protection and attach. If it is detected, the app immediately exits.

**MapViewController.m**

This class is responsible for managing the user map interactions but it also maliciously records the user’s location in the didUpdateLocations method. If a file has not been created yet for MapHelper.txt, it is created. Then a timestamp and the current coordinates and region of the user are written to the file each time this method is invoked.

**RadioTableViewController.m**

When the user opens this view, the controller’s viewDidLoad method dispatches an asynchronous request and begins making a phone call without asking the user for permission to do so.

**AddNoteViewController.m**

In this view, the user is able to enter text in a text box but then when it is saved and reloaded, it is loaded as html in a webview, meaning if a user or attacker enters JavaScript into the text box, they can perform a JS injection on the screen. See the code below the comment “configure the webview to load the text” in this class.

**info.plist**

App Transport Security Settings is set to false to allow connections to websites using HTTP as well as HTTPS.

***noJSObfuscatedBranch***

This branch includes everything except JSPatch in order that it may be obfuscated. The obfuscation makes it more difficult for attackers to access the source code by changing the names of custom methods and classes within the app. Obfuscation is done using a third party library and you can find the instructions below.

An obfuscated app should not be committed to git. Rather you should run these steps to obfuscate your code before release.

1. Install preemptive/PPIOS-Rename from github.
   1. Download from <https://github.com/preemptive/ppios-rename/releases> \*\* at time of writing we use v. 1.0.0
   2. Copy the executable into your PATH e.g. /usr/local/bin
2. Commit all local changes in your source code
3. Build the program
4. OPTIONAL: run the following command:
   1. nm /path/to/program.app/program | less > unObfuscatedClasses.txt
   2. This will give you a basis for comparison to ensure that you have done the obfuscation correctly
5. Run the analyze command as follows
   1. ppios-rename --analyze /path/to/program.app/program –F ‘!AcmeUIApplication’

\*\*RUN THIS COMMAND ONLY ONCE

NOTE: this program can be found in the Products section of your xcode project. If you click on the app, the path to it will be shown in the far right panel in xcode. If that does not work, identify the path by right-clicking and choosing ‘Show in Finder’. Additionally the –F is important for AcmeAirlines. If this class is obfuscated, the app will not be able to run so the –F serves to exclude this class from obfuscation.

1. Run the obfuscate command as follows:
   1. ppios-rename --obfuscate-sources

\*\* RUN THIS COMMAND ONLY ONCE

1. Build the program again
2. OPTONAL: Run the same command as in step 4 but do not write the output to a file. Rather compare the output in your terminal to the txt in unObfuscatedClasses.txt and confirm that the class names are now randomized strings.
3. Release your app

NOTE: Do not commit your changes after this process, at any point if you wish to undo your obfuscation, the best way I found is to simply stash or reset your changes. The documentation lists some other ways to debug if you receive errors. The documentation also lists an alternative way to perform the obfuscation as a part of the build process, however, I found this way to be simpler. Please see the [documentation](https://github.com/preemptive/PPiOS-Rename) for more information.