

The Problem:

Typical OAuth Applications rely on a client and two servers. One where your web-application is hosted, and the data source (in this case, Hive Manager). Since a typical web-app is loaded in a browser, it is susceptible to having it's memory inspected, and thus should not be trusted with the client secret, or the user's password.

However with a native application, sufficient protections are possible by means of encryption that we can trust the application to securely store the Access Token and Client Secret (though we never want to store the user's password).

The OAuth specification states that a Native Application may secure an Access Token by the following methods:

- capture the response from the authorization server (HiveManager NG) using a redirect URI with a scheme registered with the operating system to invoke the client as a handler.
- manual copy & paste of the credentials
- running a local web server
- installing a user-agent extension
- providing a redirection URI identifying a server-hosted resource under the client's control, which in turn makes the response available to the native application.

The first is ideal, but not all servers (including the current NG) support custom URI schemes. The last option is possible, but we want something quite a bit simpler and perhaps more elegant.

The Solution:

Instead of using the web browser and sending the user outside of the application, we will be implementing a WebKit frame (for web developers, think iFrame) within the application to authorize the app with HiveManager. While this is occurring, we'll be listening for our redirect URI. When the WebKit frame loads our redirect URI, we'll capture the URL and query parameters NG included which contain the Auth Code. We can then have the application make a secondary request to Hive Manager to exchange the Auth Code for an Access Token. We can manage all of this without running a web server locally on the device which can prove problematic in iOS where applications can be put to sleep by the OS when not in the foreground.

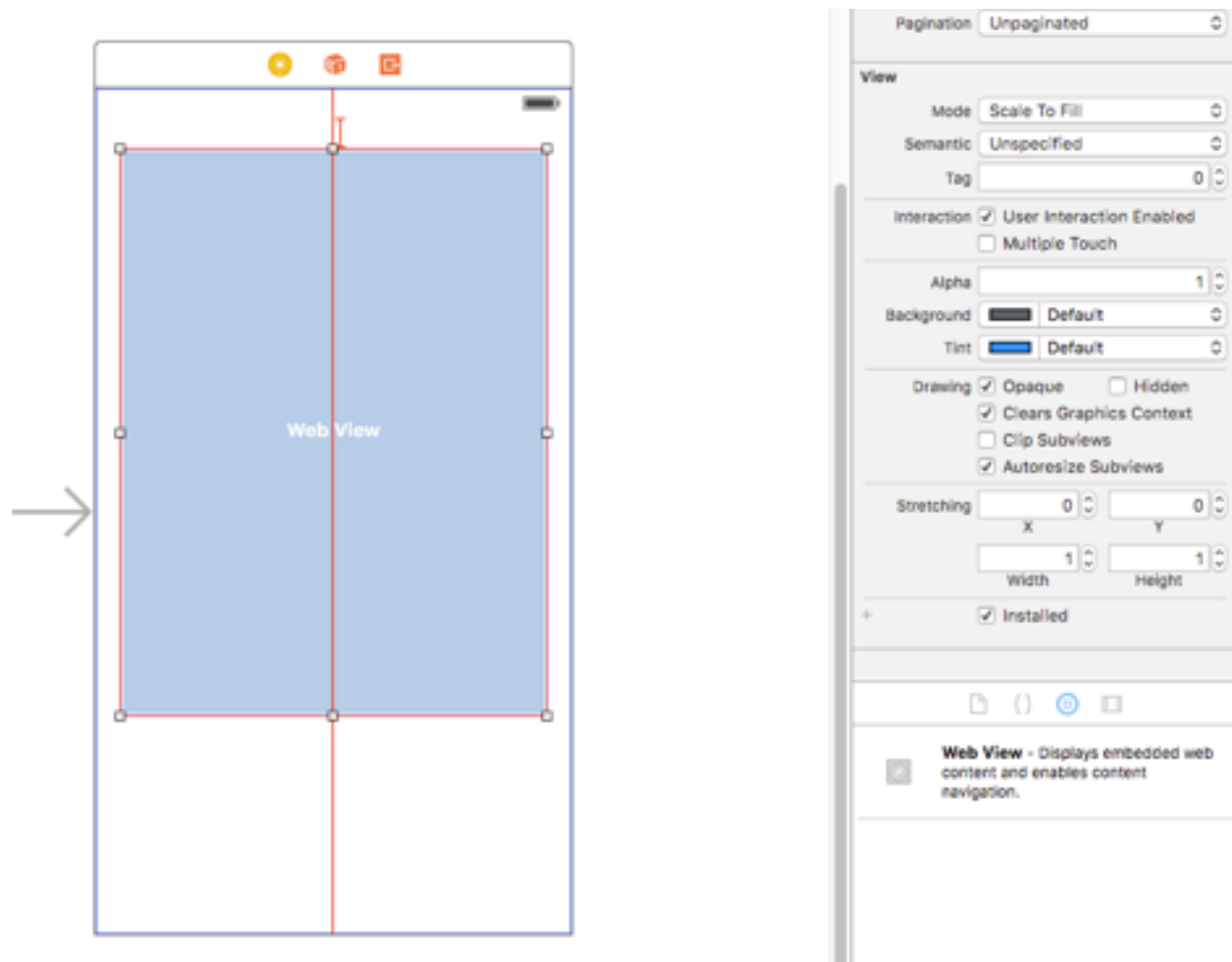
Download the project files [here](#) to have a peek at how we did it.

If you encounter any issues, feel free to raise one here: <https://github.com/aerohive/OAuthHack/issues> or email: dororke@aerohive.com.

NOTE: The linked sample code does **not** encrypt the Auth Code, and prints URLs including auth codes to the debug console. This is **not** secure, and should be changed in any production application.

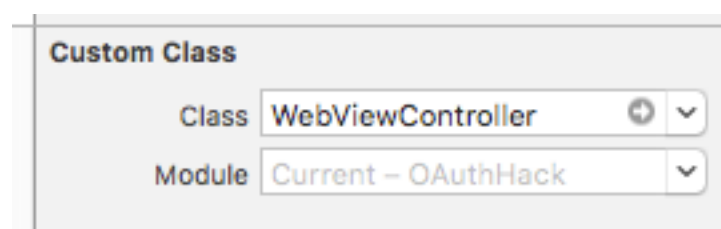
How to do it:

First, open Xcode and create a new iOS project in Swift. Add a new viewController and add a Web View.



Next, create a new CocoaTouch class called WebViewController.

Set the new View Controller to be a subclass of the WebViewController.swift file you just created.

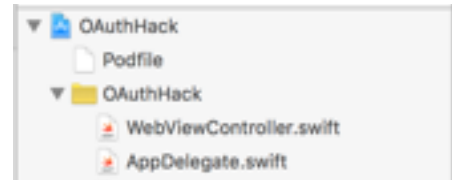


We'll now need to add some modules to make it all work. We will use CocoaPods. If you are unfamiliar with CocoaPods, I highly suggest you watch Google's introduction to the dependency manager before going further.

<https://www.youtube.com/watch?v=iEAjvNRdZa0>

Let's make a Podfile which will tell cocoa pods what to install. Create a new "Empty File" at the root of your project. Call it "Podfile".

Within the podfile, we will add the following text:



```
source 'https://github.com/CocoaPods/Specs.git'
xcodproj 'YOUR_PROJECT_NAME.xcodeproj'
platform :ios, '8.0'
use_frameworks!

pod 'Alamofire', '~> 3.0'
pod 'SwiftyJSON', :git => 'https://github.com/SwiftyJSON/SwiftyJSON.git'
```

Now close your project, and open a terminal window. Navigate to your project. My favorite way to get there is to type 'cd ' into the terminal window, then drop the folder I want to navigate to directly onto the terminal window. Hit enter and viola! You're there.

Now we want to install the CocoaPods for Alamofire and SwiftyJSON. Alamofire is a swift-based networking library, written by the same author of AFNetworking and SwiftyJSON is a powerful library for handling JSON objects.


```
dororke-mac:Python dororke$ cd /Users/dororke/Documents/Egnyte/Private/dororke/Project\ Files/iOS/OAuthHack
dororke-mac:OAuthHack dororke$ pod install
```

Assuming all went well, you should have an xcworkspace file in your project folder in addition to the xcodeproj file. You will need to use the new xcworkspace file from now on because it contains references to the frameworks you just added! Without them, things won't compile.

OK, now that you're got the xcworkspace file opened up, you can begin to use the libraries. Add 'import Alamofire' and 'import SwiftyJSON' to your WebViewController.swift file and UIWebViewDelegate to the list of things this file extends.

```
import UIKit
import Alamofire
import SwiftyJSON

class WebViewController: UIViewController, UIWebViewDelegate {
    @IBOutlet weak var successImage: UIImageView!
```

You will also want to connect the WebView you added in your story board to the `WebViewController.swift` file by selecting the connect view , holding down the control key and dragging from the webView in the story board to the code of `WebViewController.swift`. Create an IBOutlet, and call it 'myWebView'.

Now for the functions. If we are going to do OAuth, we will need a Client ID, Client Secret, and Redirect URL. You can find these in your app profile at <https://developer.aerohive.com/>. Note that since we're overriding the redirect URL with our custom data, you can set your redirect URL to anything you want, **provided it is https**. I used our dev portal as an example.

```
// MARK: Variables for OAuth
let clientID = "your_app_id"
let clientSecret = "your_secret"
let redirectURL = "https://developer.aerohive.com/"
```

Now we will need to create the WebView within the ViewController. Add the following to your project:

```
func createLoginWebView() {
    //===== THIS GENERATES THE WEBVIEW WHEN USERS LOG IN=====
    // Set up the OAuth URL to call
    let authQueryParams = "?client_id="+clientID
    +"&redirect_uri="+redirectURL
    let urlString = "https://cloud.aerohive.com/
    thirdpartylogin"+authQueryParams
    let url = NSURL (string: urlString)
    let requestObj = NSURLRequest(URL: url!);
    myWebView.delegate = self // Allows us to OVERRIDE functions
    myWebView.loadRequest(requestObj);
}
```

The above code first generates the URL we will send the user to in order to authorize your app on HiveManger. The URL is then encoded as a NSURL object and embedded in a NSURLRequest object. The important piece is to set the delegate for myWebView so that myWebView will call the webViewDidFinishLoad function we will implement next.

webViewDidFinishLoad is called each time the WebView completes loading a URL. We will use this function to capture the URLs being followed by the user and identify when the user has encountered the redirect URL. This is important because the redirect URL does not actually have the capability to process the Auth Code issued by NG. It is up to the app to process the Auth Code when the redirect URL is encountered.

If we observe the redirect URL, the entire URL and query parameters are split on '?' just the query parameters are sent to another function for processing.

Add the following below the function you just inserted.

```

func webViewDidFinishLoad(webView : UIWebView) {
    print("Finished!")
    let requestedURL = myWebView.request?.URL?.absoluteString
    //Check if the request URL contains our redirect URL
    // 'lowercaseString' converts both strings to lowercase so it is case
agnostic
    // 'hasPrefix' checks if one string begins with the other.
    if (requestedURL!.lowercaseString.hasPrefix(redirectURL.lowercaseString)){
        self.myWebView.hidden = true // Since we got the auth code, hide the
webView!
        let UrlArray = requestedURL!.componentsSeparatedByString("?") //Split
the URL string into an array on '?'
        let queryString = UrlArray[1] // Grab only the things after the '?'
        let params = parametersFromQueryString(queryString) // We get back a
dict of the query params.
        print(params)
        if params["authCode"] != nil { // Check if the result has authCode
            let accessTokensJSON = getAccessTokenFromAuthCode(params)
        }
        else { // We don't have the Auth Code
            let alert = UIAlertController()
            alert.title = "Hey! There was an error."
            alert.message = params["error"]
            let okAction = UIAlertAction(title: "OK", style:
UIAlertActionStyle.Default) {
                UIAlertAction in
                NSLog("OK Pressed")
                self.createLoginWebView() //reload the login page
            }
            alert.addAction(okAction)
            self.presentViewController(alert, animated: true, completion: nil)
        }
    }
}

```

BTW, if you've been reading from the top this would be a good time to stand up, stretch and maybe grab a glass of water or go for a short walk. The internet will still be here when you get back.

Now that we've had a short break, the next thing to do in the OAuth flow is exchange the Auth Code for an Access Token. This is precisely what `getAccessTokenFromAuthCode` does. It is passed an array of parameters which only includes the Auth Code, but we add our redirect URI to this before making the request.

We use Alamofire to make an asynchronous POST request to the Aerohive Cloud, hence why everything happens in closures. For more about closures look here:

http://www.tutorialspoint.com/swift/swift_closures.htm

We also use SwiftyJSON to handle the JSON response in an elegant way.

Check out [SwiftyJSON](#) and [Alamofire](#) if you have not worked with them already.

```

// This function will exchange an AuthCode for an Access Token
func getAccessTokenFromAuthCode(var params: [String:String]) {
    print("Trying to exchange the Auth Code for an Access Token...")
    let headers = [
        "X-AH-API-CLIENT-SECRET": clientSecret,
        "X-AH-API-CLIENT-ID": clientID,
        "X-AH-API-CLIENT-REDIRECT-URI": redirectURL
    ]
    let url = "https://cloud.aerohive.com/services/acct/thirdparty/accesstoken"
    let requestURL = url
    params["redirectUri"] = redirectURL // Add the redirect URL to our query parameters.
    Alamofire.request(.POST, requestURL, headers: headers, parameters:params ).responseJSON { response in
        // Check that the error is nil
        guard response.result.error == nil else {
            // got an error in getting the data, need to handle it
            print("error calling GET on "+requestURL)
            print(response.result.error!)
            print("Headers: ")
            print(headers)
            return
        }
        if let value: AnyObject = response.result.value {
            // handle the results as JSON, without a bunch of nested if loops
            let result = JSON(value)
            print (result)
            if result["error"]["status"].intValue > 200 { // There was an error
                print("There was an error:" + result["error"]["status"].stringValue)
                // NOTIFY THE USER THAT THE API RETURNED AN ERROR (404,401,403,500,
                etc.)

                let alert = UIAlertController()
                alert.title = "Hey! Error: " + result["error"]["status"].stringValue
                alert.message = result["error"]["message"].stringValue
                let okAction = UIAlertAction(title: "OK", style:
UIAlertActionStyle.Default) {
                    UIAlertAction in
                    NSLog("OK Pressed")
                    self.createLoginWebView() //reload the login page
                }
                alert.addAction(okAction)
                self.presentViewController(alert, animated: true, completion: nil)
            }
            else{
                print("Success!")
                self.nowIHaveTheAccessTokens(result)
            }
            print (result)
        }
        dispatch_async(dispatch_get_main_queue()) {
            // Here's where we synch back up with the UI.
        }
    }
}
}

```

If the HTTP request completes without errors, the JSON is parseable, and we are not given an error code in the response JSON from the API, we call `nowIHaveTheAccessTokens` and pass the `SwiftJSON` object.

This is where it is left to you, the developer to store the VHM ID(s), regional data center URL(s), access token(s), expiry date, and refresh token(s) in a secure fashion for future use.

Note that it is not necessary to create a new access token every time the app is launched. On launch, the application should check for a stored Access Token, validate that it is not expired,

and use it to gain access to HiveManager. If the Access Token is expired, the user will need to log in again.

Here is the function used in the sample. Note that while the example does not encrypt the data, and prints Access Tokens to the console and screen, this is for demonstration purposes only and not advisable for a production application.

```
func nowIHaveTheAccessTokens(authResult: JSON) {
    print ("These are the droids we're looking for!!")
    print (authResult)
    var accessTokens = [String:String]()
    for VHM in authResult["data"] {
        accessTokens[authResult["data"][VHM.0]["ownerId"].stringValue] =
authResult["data"][VHM.0]["accessToken"].stringValue
    }
    let alert = UIAlertController()
    alert.title = "Hey, we got it!"
    alert.message = "Token 1: " + accessTokens.values.first!
    let okAction = UIAlertAction(title: "OK", style: UIAlertActionStyle.Default)
{
    UIAlertAction in
    NSLog("OK Pressed")
    self.myWebView.hidden = false // Unhide the webview.
    self.createLoginWebView() //reload the login page
}
    alert.addAction(okAction)
    self.presentViewController(alert, animated: true, completion: nil)
}
```

To download the completed project, click [here](#).