# NEW AGE APPLICATION ATTACKS AGAINST APPLE's iOS [AND COUNTERMEASURES]

by NITESH DHANJANI

BLACKHAT BARCELONA 2011

## Executive Summary

10 billion app downloads, 100 million iPhones, and 15 million iPads later, it is clear that the iOS revolution is here. The exponential rate of adoption of Apple's iOS platform in the enterprise has been stunning.

The adoption of iPhones and iPads also brings in a new attack surface into the enterprise. As such, it becomes important to establish a clear and actionable process for securely designing enterprise applications that are being designed to run on these devices. These applications are responsible for securely handling financial, healthcare, and utility related data and transactions.

This paper brings together emerging research to illustrate the net-new attack vectors targeting iOS applications. The intended audience for the rest of this paper include technical security analysts and iOS application developers. The following topics are discussed in detail:

- Protocol handling attacks and secure design.
- User Interface (UI) attacks and best practices.
- Abusing and securely design Apple Push Notifications.
- Man in the Middle, Privacy Leaks, Identity Decloaking, and Countermeasures.

In addition to these topics, the Appendix in this document contains a checklist of items to consider when assessing iOS applications. This list includes traditional application security weaknesses that also apply to iOS. Additional items to consider, such as data protection and file encryption applicable to iOS devices, are also presented in the Appendix.

The following steps are recommended to executives whose job it is to securely enable the iOS platform in the enterprise:

- Build a solid team of professionals who have deep knowledge of the iOS platform and the Objective C language.
- Have the capability to keep up to date with emerging attack vectors targeting the iOS platform.
- The information presented in this document as well as the attached Appendix should be used to hook into the enterprise Software Development Lifecycle (SDL). Doing this will help make iOS applications developed in-house and consumed from third parties more secure.

The iOS revolution is here. In many organizations, iPads and iPhones are the tipping point into allowing user owned devices into the perimeter. As such, the iOS devices themselves have become the new corporate perimeter given the critical amount of data and transactional capability they are trusted with. The applications that run on these devices are sure to be an emerging target. The actionable information presented in this document will enable you to kick-start the process and capability needed to enable the platform securely.

## Introduction

In January of 2011, Apple announced that 10 Billion iPhone and iPad Apps have been downloaded from the App Store. In March 2011, Apple announced that it has sold 100 million iPhones and 15 million iPads to date.

The majority of the 10 billion Apps downloaded from the App Store have been written by third party organizations and developers. Besides entertainment, these Apps store and transmit confidential data and are relied upon by individuals and corporations to perform critical transactions in the fields of healthcare, utilities, and finance.

As the market-share of iPhones and iPads continues to gain momentum in the enterprise, threat agents with malicious motives and intentions are likely to target the Apps that run on these devices.

To securely enable the onslaught of Apple's iOS device into the enterprise, organizations must have the capability and talent to ascertain and act upon the net-new attack surface exposed by the application layer pertaining to the platform. As such, this paper brings together recent research that uncovers new attack vectors applicable to iOS applications.

The details on new attack vectors targeting iOS applications presented in this paper will enable enterprises to lower the risk posed by third party and custom developed Apps that are deployed on iPhones and iPads. In addition to the specific attack vectors, this paper also includes an Appendix containing an assessment methodology checklist that can be used to locate vulnerabilities in iOS Apps prior to deployment.

## Protocol Handling Attacks

From initiating Skype phone calls to launching the default email client, we place trust on web browsers to securely launch applications that reside on our desktop.

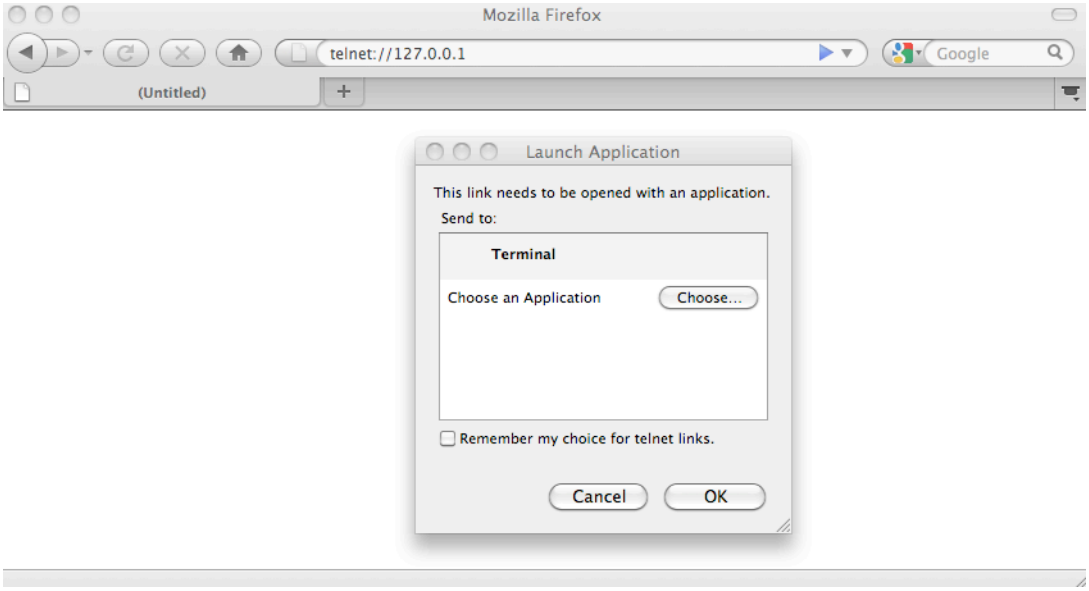


**Figure 1:** The Firefox browser requests the user's permission before executing the Telnet application

The screen-shot in Figure 1 illustrates how the `telnet:` protocol handler is invoked in the Firefox web browser. As shown, the browser makes sure to request permission from the user before launching a local Telnet executable to handle the request.

### URLSchemes and the `tel:` Handler

Apple uses the term URLSchemes to refer to protocol handlers. In the URLScheme Reference document [`http://developer.apple.com/library/safari/#featuredarticles/iPhoneURLScheme_Reference/Introduction/Introduction.html`], Apple lists the default URL Schemes that are registered within iOS. For example, the `tel:` scheme can be used to launch the Phone application.

.

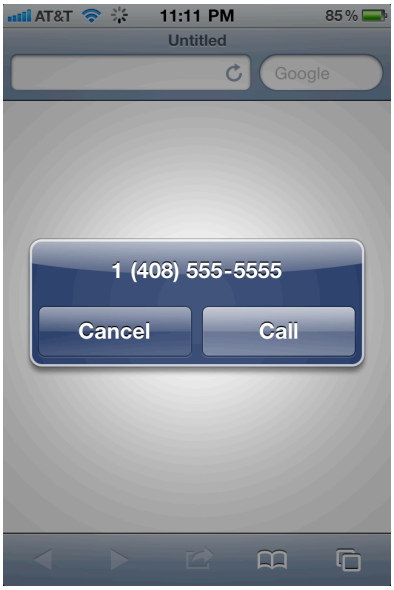


**Figure 2:** Dialog box in Safari (iOS) requesting for user authorization to launch the Phone application

Now, imagine if a website were to contain the following HTML rendered to someone browsing the web using Safari on iOS:

```
<iframe src="tel:1-408-555-5555"></iframe>
```

In this case, Safari requests the user for authorization as shown in Figure 2. This is the recommended behavior since the control protects the user from launching arbitrary phone calls if instructed by malicious websites.

### Insecure URLScheme Implementations

Users that have Skype.app installed and have launched Skype in the past are likely to have their credentials cached.


**Figure 3:** Skype automatically initiating a call on iOS after being invoked by a malicious website.

Now, what do you think happens when a malicious site renders the following HTML?

```
<iframe src="skype://14085555555?call"></iframe>
```

In this case, Safari throws no warning, and yanks the user into Skype which immediately initiates the call. A screen-shot of this behavior is shown in Figure 3.

The security implications of this is obvious, including the additional abuse case where a malicious site can make Skype.app call a Skype-id who can then decloak the victim's identity (by analyzing the victim's Skype-id from the incoming call).

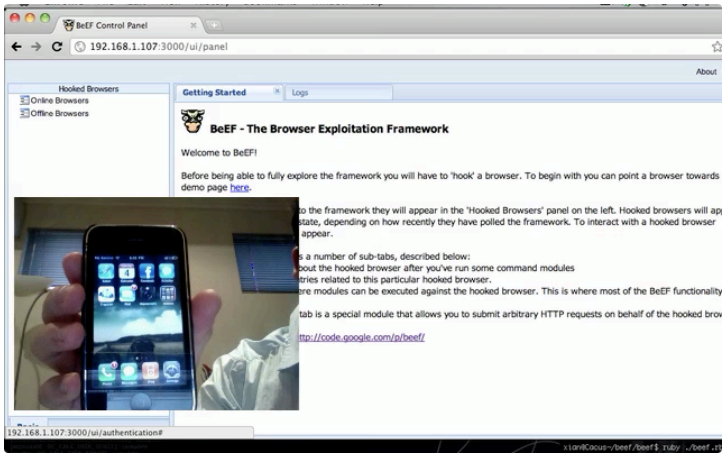


**Figure 4:** The `skype:` handler vulnerability demonstration has been built into the BeEF tool

This vulnerability was reported to Apple and Skype in October 2010. Skype never responded, but fixed this issue in version 3.0 of their iOS application. Meanwhile, the exploit for this issue has been incorporated into the BeEF (Browser Exploitation Framework) project. The demonstration video of this, as illustrated in Figure 4, can be viewed online at `http://www.youtube.com/watch?v=5SVu6VdLWgs`

In addition to the `skype:` example, there have been many cases of custom-developed Apps for various enterprises that expose URLSchemes insecurely. Here are some examples from the field:

- `hr_check://update_employee/EMP_ID=342&NEW_ADD=2%20MAIN%20ST`
- `utility_dashboard://shutdown/device_id=34`
- `health_record://close_case/patient_id=993423`

As you can tell from the list of examples above, the implications of a malicious resource being able to commit the associated transactions can easily spell disaster for the target enterprise.

**Discovering Exposed URLSchemes**

To register a URLScheme, the app needs to list the exposed handlers in its Info.plist file. For example, here is a section from Skype's `Info.plist` file:

```
<key>CFBundleURLTypes</key>
        <array>
                <dict>
                        <key>CFBundleURLName</key>
```

```
                    <string>com.skype.skype</string>
                    <key>CFBundleURLSchemes</key>
                    <array>
                            <string>skype</string>
                    </array>
            </dict>
    </array>
```

If you have a jail-broken phone, you can easily copy over the relevant App's `Info.plist` file for review. Note that you will have to run the following command in order to view it in a text editor:

```
plutil -convert xml1 Info.plist
```

Alternatively, you can capture Info.plist files from your `~/Music/iTunes/Mobile Applications` directory. Just rename the relevant `.ipa` file to `.zip` and unzip it. You should then have access to the `Info.plist` file.

The following resources list URLScheme handlers for popular Apps: `http://handleopenurl.com/scheme` and `http://wiki.akosma.com/IPhone_URL_Schemes/`

### Uncovering Hidden Transactions and Decrypting App Store Binaries

Applications can implement distinct associated transactions with every handler. For example `skype://[phone_number]?call` and `skype://[skype_id]?chat` implement different functionalities (call and chat) even though both of them are invoked by the `skype:` handler.

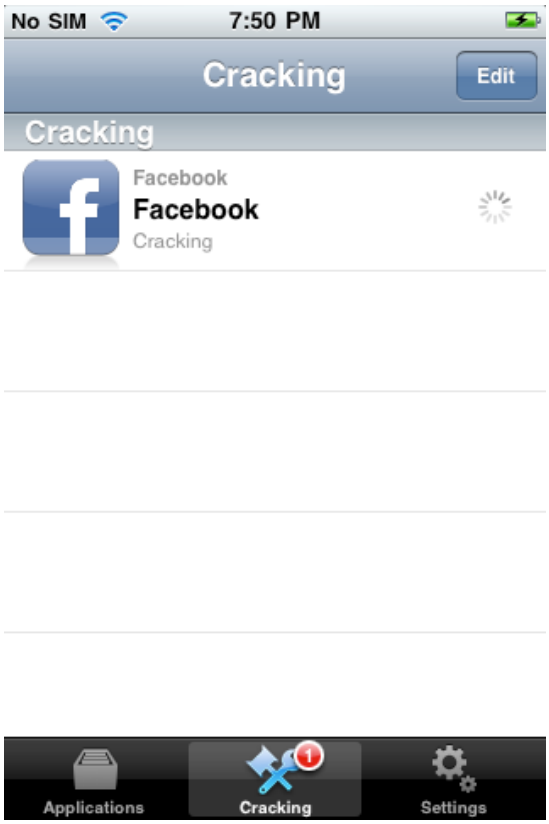


**Figure 5:** Decrypting the Facebook app

Some iOS applications include transactions that are not documented. However, the actual application binary is likely to have the associated patterns hard-coded. A simple strings query on the application binary is likely to expose such hard-coded patterns.

Note that App Store binaries are encrypted. For instructions on how to decrypt the binaries, please see instructions at `http://dvlabs.tippingpoint.com/blog/2009/03/06/reverse-engineering-iphone-appstore-binaries`. Alternatively, you can use a tool such as Crackulous (for jail-broken phones) as shown in Figure 5.

In the case of Facebook.app, the following `strings` command can be run on the decrypted binary:

```
strings Facebook.app/Facebook | grep 'fb:' | more
```

Running this command helped uncover a slew of undocumented URLScheme transactions based on the fb: handler, some of which are presented below:

```
fb://online#offline
fb://birthdays/(initWithMonth:)/(year:)
fb://userset
fb://nearby
fb://place/(initWithPageId:)
fb://place/addfriends
fb://places/(initWithCheckinAtPlace:)/(byUser:)
fb://places/legalese/tagged/(initWithTaggedAtPlace:)/(byUser:)
fb://publish
fb://publish/profile/(initWithUID:)
fb://publish/post/(initWithPostId:)
fb://publish/photo/(initWithUID:)/(aid:)/(pid:)
fb://publish/mailbox/(initWithFolder:)/(tid:)
fb://place/create
fb://map
fb://upload
fb://upload/checkin/(showUploadMenuWithCheckinID:)
fb://upload/profile/(showUploadMenuWithUID:)
fb://upload/album/(showUploadMenuWithUID:)/(aid:)
fb://upload/actions
fb://upload/actions/profile/(initWithUID:)
fb://upload/actions/album/(initWithUID:)/(aid:)
fb://upload/actions/checkin/(initWithCheckinId:)
fb://upload/actions/resume
```

As you can see, the process of uncovering exposed URLScheme requires not only an analysis of the `Info.plist` file, but also a review of the actual binary for associated transaction strings.

### Securely Implementing URLSchemes

Applications that expose URLSchemes in their Info.plist files must implement code to handle the invocation. This is done in the `handleOpenURL:` delegate:

```
(BOOL)application:(UIApplication *)application handleOpenURL:(NSURL *)url
{

//Parse URL <-- Careful - do thorough input validation

//Ask for authorization
```

```
//Perform transaction
}
```

Note that as of iOS 4.2, `handleOpenURL:` is deprecated. Use `application:openURL:sourceApplication:annotation:` instead which is more secure because you get the invoker's `BundleID` (`SourceApplication`) and a custom `.plist` (`annotation`). You can use this information to ascertain who is invoking you (i.e. via Safari or another trusted app). Still, be careful: your app can be subject to a *Ricochet Attack* where an external website can abuse an intermediary app to get to you, i.e. `<iframe src="some_other_app://you://boom">`

Also consider following the advise below:

- Perform strict input validation when parsing the incoming parameters.
- Do not allow URLScheme transactions to edit or delete user data or change status of a given transaction.
- Assume that your handler as well as the associated transaction strings are known to the world, i.e. do not rely on the secrecy of the name of your handler or transaction strings.
- When buying a third party enterprise application, be sure to audit it or look through the code and make sure you understand what URLSchemes are exposed and how they are invoked.

**Open Questions to Apple**

Unlike the case of the `tel:` handler, which enjoys the special privilege of requesting the user for authorization before yanking the user away from his or her browsing session in Safari, third party applications can only request authorization after they have been fully launched. A solution to this issue is for Apple to allow third party applications an option to register their URL schemes with strings for Safari to prompt and authorize prior to launching the external application.

This leaves open the following two questions for discussion:

1. *Should Apple audit the security implications of registered URL schemes as part of its App Store approval process?* Apple's `tel:` handler causes Safari to ask the user for authorization before placing phone calls. The most logical explanation for this behavior is that Apple is concerned about their customers' security and doesn't want rogue websites from being able to place arbitrary phone calls using the customer's device.

However, since the Skype application allows for such an abuse case to succeed, and given that Apple goes to great lengths to curate applications before allowing them to be listed in the App Store, should Apple begin to audit applications for security implications of exposed URL Schemes? After all, Apple is known to reject applications that pose a security or privacy risk to their users, so why not demand secure handling of transactions invoked by URL Schemes as well?

2. *Should Apple implement a feature to allow advanced users and enterprise administrators to see the list of exposed URL Schemes?* It is possible to enumerate all the Info.plist files in an iPhone to ascertain the list of exposed URL schemes. However, it may make sense for this list to be available in the Settings section of iOS so users can look at it to understand what schemes their device responds to that can be invoked by arbitrary websites. Perhaps this will mostly appeal to the advanced users, yet it will help keep the application designers disciplined the same way the user location notification in iOS does. This will also make it easier for enterprises to figure out what third party applications to provision on their employee devices based on any badly designed URL schemes that may place company data at risk.

In order to create a registry of exposed URL schemes, Apple cannot simply parse information from Info.plist because it only contains the initial protocol string. In other words, the `skype:` handler responds to `skype://[phone_or_id]?call` and `skype://[phone_or_id]?chat` but only the `skype:` protocol is listed in `Info.plist`, while the actual parsing of the URL is performed in code. Therefore, to implement

this proposed registry system, Apple will have to require developers to disclose all patterns within a file such as `Info.plist`.

## User Interface Spoofing Attacks

Popular web browsers today do not allow arbitrary websites to modify the text displayed in the address bar or to hide the address bar (some browsers may allow popups to hide the address bar, but in such cases, the URL is then displayed in the title of the window). The reasoning behind this behavior is quite simple: if browsers can be influenced by arbitrary web applications to hide the URL or to modify how it is displayed, then malicious web applications can spoof User Interface elements to display arbitrary URLs, thus tricking the user into thinking he or she is browsing a trusted site.

Using an iPhone, browse to the following demo and keep an eye out on the address bar:

```
http://www.dhanjani.com/iphone-safari-ui-spoofing/
```

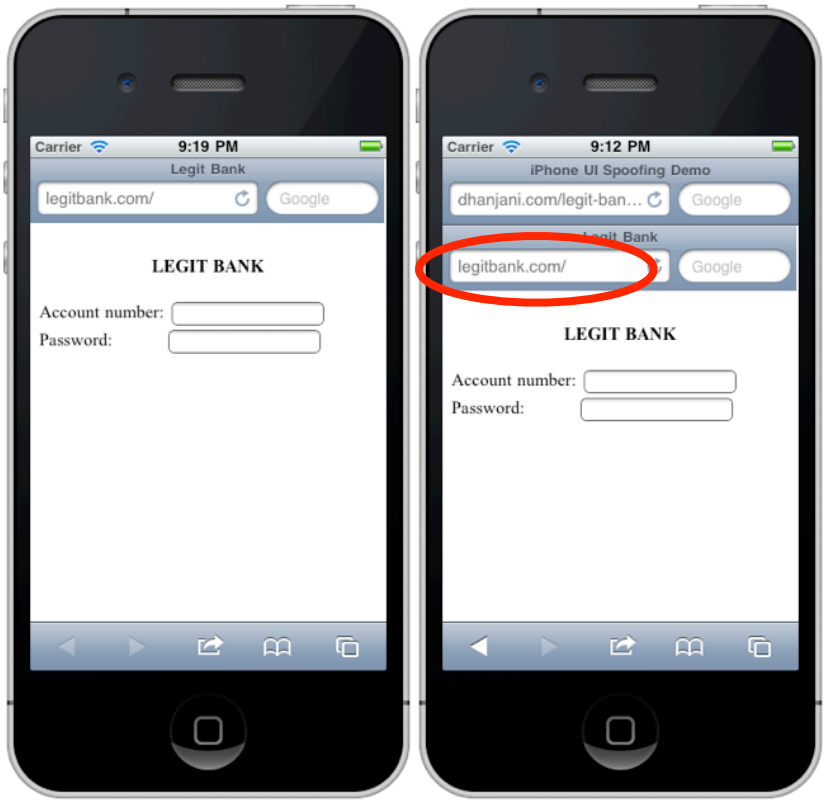


**Figure 6:** Image on the left illustrates the page rendered, which displays the 'fake' URL bar while the real URL bar is hidden above. Image on the right illustrates the real URL bar that is visible once the user scrolls up.

As illustrated in Figure 6, notice that the address bar stays visible while the page renders, but immediately disappears as soon as it is rendered. While this may give the user some time to notice, it is, however, not a reasonably reliable control (and Apple probably did not intend this to be a reliable control).

The primary reason for this behavior is to maximize the screen real-estate on the iPhone. However, since the address bar in Safari occupies considerable space, perhaps Apple may consider displaying or scrolling the current domain name right below the universal status bar (i.e. below the carrier and time stamp). Positioning the current domain context in a location that is unalterable by the rendered web content can provide the users similar indication that browsers such as IE and Chrome provide by highlighting the current domain being rendered.

### Secure Design of Apps that Leverage `UIWebView`

Desktop operating systems most often launch the default web browser of choice when a `http` or `https` handler is invoked (this is most often the case even though the operating systems provide interface elements that can be used to render web content within the applications).

However, in the case of iOS, since most applications are full-screen, it is in the interest of the application designers to keep the users immersed within their application instead of yanking the user out into Safari to render web content. Given this situation, it becomes vital for iOS to provide consistency so the user can be ultimately assured what domain the web content is being rendered from.

To render web content within applications, all developers have to do is invoke the `UIWebView` class. It is as simple as invoking a line of code such as `[webView loadRequest:requestObj];` where `requestObj` contains the URL to render.


**Figure 7:** Twitter app rendering web content on the iPad

The screenshot in Figure 7 illustrates web content rendered by the Twitter app on the iPad. The URL being rendered is no where to be seen.

In such cases, it is clear that developers of iOS applications need to make sure that they clearly display the ultimate domain from which they are rendering web content. A welcome addition to this would be default behavior on part of `UIWebView` to display the current domain context in a designated and consistent location.

Developers leveraging `UIWebView` to render HTML data should display the URL. In this age of URL shorteners, it is vital for users to be able to quickly locate the URL of the website they are browsing.

## Abusing Push Notifications

Millions of iOS users and developers have come to rely on Apple's Push Notification Service (APN). This section will briefly introduce details of how APN works and present scenarios of how insecure implementations can be abused by malicious parties.

Apple's iOS allows some tasks to truly execute in the background when a user switches to another app (or goes back to the home screen), yet most apps will return and resume from a frozen state right where they left off. Apple's implementation helps preserve battery life by providing the user the illusion that iOS allows for full-fledged multitasking between third party apps.

This setup makes it hard for apps to implement features that rely on full-fledged multitasking. For example, an Instant Messaging app will want to alert the user of a new incoming message even if the user is using another app. To enable this functionality, the APN service can be used by app vendors to push notifications to the user's device even when the app is in a frozen state (i.e. the user is using another app or is on the home screen).

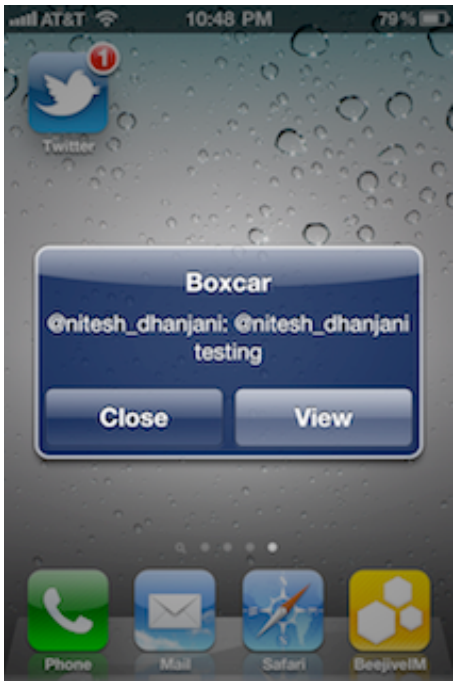


**Figure 8:** Badge and alert notification in iOS

There are 3 types of push notifications in iOS: alerts, badges, and sounds. Figure 8 shows a screen-shot illustrating both an alert and a badge notification.

iOS devices maintain a persistent connection to the APN servers. Providers, i.e. third party app vendors, must connect to the APN to route the notification to a target device.

**Implementing Push Notifications**

Before we take a look at how push notifications can be abused or place enterprise data at risk, it important to first comprehend the steps required to implement push notifications.

For explicit details on how to implement push notifications in your app, please see Apple's Local and Push Notifications Programming Guide available at `https://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/`.

This section briefly lists the steps needed to implement push notifications with an emphasis on security implications.

*1. Create an App ID on Apple's iOS Provisioning Portal.* This is straightforward: just click on "App IDs" in the provisioning portal and then select "New App ID". Next, enter text in the Description field that describes your app. The "Bundle Seed ID" is used to signify if you want multiple apps to share the same Keychain store.

The Bundle Identifier is where it gets interesting: you need to enter a unique identifier for your "App ID". Apple recommends using a reverse-domain name style string for this.

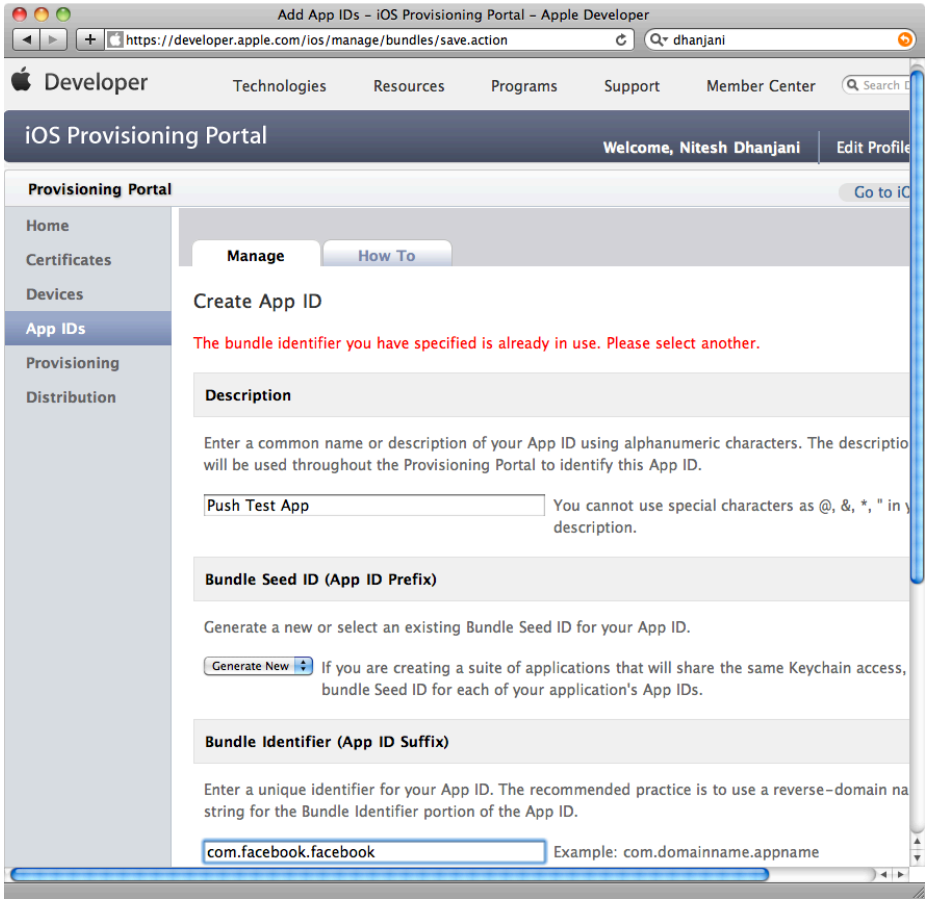


**Figure 9:** iOS provisioning portal rejects "com.facebook.facebook" Bundle Identifier

The "Bundle Identifier" is significant from a security perspective because it makes its way into the provider certificate we will create later (also referred to as the topic). The APN trusts this field in the certificate to figure out which app in the target user's device to convey the push message to. As Figure 9 illustrates, an attempt to create an "App ID" with the "Bundle Identifier" of `com.facebook.facebook` is promptly refused as a duplicate, i.e. the popular Facebook iOS app is probably using this in its certificate. *Therefore, if a malicious entity were to bypass the provisioning portal's restrictions against allowing an existing Bundle Identifier, then he or she could possibly influence Apple to provision a certificate that can be abused to send push notifications to users of another app (but the malicious user would still need to know the target user's device-token discussed later in this document).*

*2. Create a Certificate Signing Request (CSR) to have Apple generate an APN certificate.* In this step, Keychain on the desktop is used to generate a public and private cryptographic key pair. The private key stays on the desktop. The public key is included in the CSR and uploaded to Apple. The APN certificate that Apple provides back is specific to the app and tied to a specific App ID generated in step 1.

*3. Create a Provisioning Profile to deploy your app into a test iOS device with push notifications enabled for the App ID you selected.* This step is straightforward and described in Apple's documentation linked above.

*4. Export the APN certificate to the server side.* As described in Apple's documentation, you can choose to export the certificate to `.pem` format. This certificate can then be used on the server side of your app infrastructure to connect to the APN and send push notifications to targeted devices.

*5. Code iOS application to register for and respond to notifications.* Your iOS app must register with the device (which in turns registers with the APN) to receive notifications. The best place to do this is in `applicationDidFinishLaunching:`, for example:

```
[[UIApplication sharedApplication]registerForRemoteNotificationTypes:
(UIRemoteNotificationTypeBadge | UIRemoteNotificationTypeSound)];
```

The code above will make the device initiate a registration process with the APN. If this succeeds, the `didRegisterForRemoteNotificationsWithDeviceToken:` application delegate is invoked:

```
- (void)application:(UIApplication *)app
  didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)devToken
{
    const void *devTokenBytes = [devToken bytes];
    self.registered = YES;
    [self sendProviderDeviceToken:devTokenBytes]; // custom method
}
```

*Notice that the delegate is passed the `deviceToken`. This token is NOT the same as the UDID which is an identifier specific to the hardware.The `deviceToken` is specific to the instance of the iOS installation, i.e. it will migrate to a new device if the user were to restore a backup in iTunes onto a new device.* The interesting thing to note here is that the deviceToken is static across apps on the specific iOS instance, i.e. the same `deviceToken` will be returned to other apps that register to send push notifications.

The `sendProviderDeviceToken:` method sends the deviceToken to the provider (i.e. your server side implementation) so the provider can use it to tell the APN which device to send the targeted push notification to.

In the case where your application is not running and the iOS device receives a remote push notification from the APN destined for your app, the `didFinishLaunchingWithOptions:` delegate is invoked and the message payload is passed. In this case, you would handle and process the notification in this method. If your app is already in the running, then the `didReceiveRemoteNotification:` method is called (you can check the `applicationState` property to figure out if the application is active or in the background and handle the event accordingly).

*6. Implement provider communication with the APN.* With the provider keys obtained in step 4, you can implement server side code to send a push notification to your users.

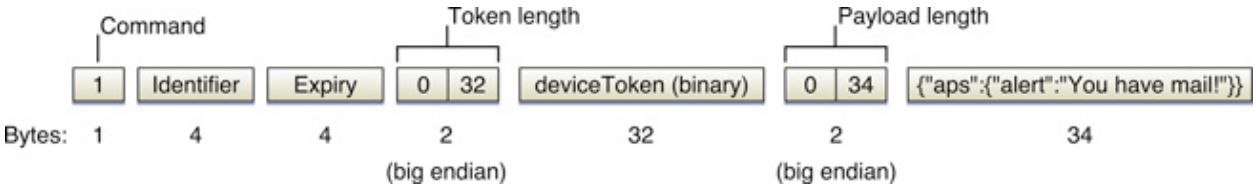


**Figure 10:** (enhanced) Notification format for push notification payloads

As illustrated in Figure 10, you will need the deviceToken of the specific user you want to send the notification to (this is the token you would have captured from the device invoking your implementation of the `sendProviderDeviceToken` method described in Step 4). The Identifier is an arbitrary value that identifies the notification (you can use this to correlate an error-response. Please see Apple's documentation for details). The actual payload is in JSON format.

Now that we have established details on implementing push-notifications into custom Apps, lets discuss applicable security best practices along with abuse cases.

**Best Practices to Counter Abuse Cases**

Now that we have established details on implementing push-notifications, let us discuss applicable security best practices along with abuse cases.

*1. Do not send company confidential data or intellectual property in the message payload.* Even though end points in the APN architecture are TLS encrypted, Apple is able to see your data in clear-text. There may be legal ramifications of disclosing certain types of information to third-parties such as Apple.

*2. Push delivery is not guaranteed, so don't depend on it for critical notifications.* Apple clearly states that the APN service is best-effort delivery.

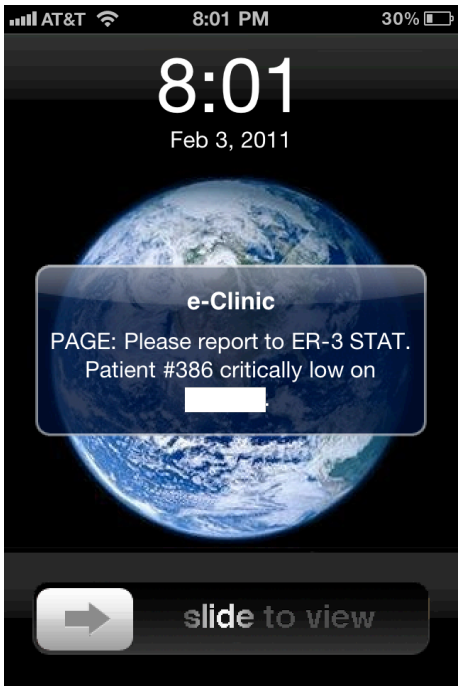


**Figure 11:** Do not rely on push for critical notifications

As shown in Figure 11, the push architecture should not be relied upon for critical notifications.

*3. Do not allow the push notification handler to modify user data.* The application should not delete data as a result of the application launching in response to a new notification. An example of why this is important is the situation where a push notification arrives while the device is locked: the application is immediately invoked to handle the pushed payload as soon as the user unlocks the device. In this case, the user of your app may not have intended to perform any transaction that results in the modification of his or her data.

*4. Validate outgoing connections to the APN.* The root Certificate Authority for Apple's certificate is Entrust. Make sure you have Entrust's root CA certificate so you are able to verify your outgoing connections (i.e. from your server side architecture) are to the legitimate APN servers and not a rogue entity.

*5. Be careful with unmanaged code.* Be careful with memory management and perform strict bounds checking if you are constructing the provider payload outbound to the APN using memory handling API in unmanaged programming languages (example: `memcpy`).

*6. Do not store your SSL certificate and list of `deviceToken`s in your web-root.* There have been known instances where organizations have inadvertently exposed their Apple signed APN certificate, associated private key, and `deviceToken`s in their web-root.

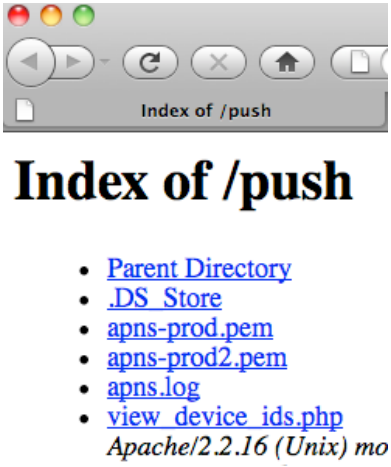


**Figure 12:** Screen-shot of APN certificates and deviceTokens being exposed

The illustration in Figure 12 is a real screen-shot of an iOS application vendor inadvertently exposing their APN certificates as well as a PHP script, which lists all deviceTokens of their customers.


**Figure 13:** Prank scenario depicting rogue push notifications targeting the Facebook app once it's provider certificate has been compromised

In this abuse case, any malicious entity who is able to get at the certificates and list of `deviceToken`s will be able to send arbitrary push notifications to this iOS application vendor's customers (see Figure 13). For example, the `jpoz-apns` Ruby gem can be used to send out concurrent rogue notifications:

```
APNS.host = 'gateway.push.apple.com'

APNS.pem  = '/path/to/pwn3d/pem/file'

APNS.pass = ''

APNS.port = 2195

stolen_dtokens.each do |dtoken|
    APNS.send_notification(dtoken,'pwn3d')

end
```

It is highly recommended to protect the cert with a passphrase (it is assumed to be null in the above example). Also, it goes without saying that any sample server side code that has the passphrase embedded in it should be kept out of the web-root. In fact, there is no good reason why any of this information should even reside on a host that is accessible from the Internet (incoming) since the provider connections to the APN need to be outbound only.

## Man in the Middle, Privacy Leaks, Identity Decloaking, and Countermeasures

Many iOS applications use HTTP to connect to server side resources. To protect user-data from being eavesdropped, iOS applications often use SSL to encrypt their HTTP connections.

In this section, we will take a look at how unencrypted network connections can place user data at risk and how the transmittal of data to third party analytics service providers can violate privacy principles. We will also take a look at relevant countermeasures and best practices.

### Man in The Middle Attacks and Testing

First, let us study sample Objective-C code to illustrate how HTTP(S) connections are established and how to locate insecure code that can leave the iOS application vulnerable to Man in the Middle attacks. Next, we will discuss how to configure an iOS device to allow for interception of traffic through an HTTP proxy for testing purposes.

The easiest way to initiate HTTP requests in iOS is to utilize the `NSURLConnection` class. Here is sample code from a very simple application that takes in a URL from an edit-box, makes a GET request, and displays the HTML obtained.

```
//This IBAction fires when the user types in a URL and presses GO
- (IBAction) urlBarReturn:(id)sender
{
    //htmlOutput is the UITextView that displays the HTML
    htmlOutput.text=@"";

    //urlBar is the UITextField that contains the URL to load
    NSURLRequest *theRequest=[NSURLRequest requestWithURL:[NSURL
    URLWithString:urlBar.text] cachePolicy:NSURLRequestUseProtocolCachePolicy
    timeoutInterval:60.0];

    NSURLConnection *theConnection=[[NSURLConnection alloc]
    initWithRequest:theRequest delegate:self];

    if(!theConnection)
        htmlOutput.text=@"failed";
}

- (void)connection:(NSURLConnection *)connection didReceiveResponse:
(NSURLResponse *)response
{
    //receivedData is of type NSMutableData
    [receivedData setLength:0];

}

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)
data
{
    [receivedData appendData:data];

    NSString *tempString = [[NSString alloc] initWithData:data
    encoding:NSUTF8StringEncoding];

    htmlOutput.text = [NSString stringWithFormat:@"%@
    %@",htmlOutput.text,tempString];

    [tempString release];
```

```
}

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)
error
{
    [connection release];

    [receivedData release];

    NSLog(@"Connection failed! Error: %@ %@", [error localizedDescription],
    [[error userInfo] objectForKey:NSURLErrorFailingURLStringErrorKey]);

    htmlOutput.text=[NSString stringWithFormat:@"Connection failed! Error %@
    %@",[error localizedDescription], [[error userInfo]
    objectForKey:NSURLErrorFailingURLStringErrorKey]];
}


- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    NSLog(@"Succeeded! Received %d bytes of data",[receivedData length]);

    [connection release];

    [receivedData release];

}
```

The result is a simple iOS application that fetches HTML code from a given URL.


**Figure 14:** Simple iOS app using NSURLConnection to fetch HTML from a given URL.

In the screen-shot in Figure 14, notice that the target URL is https. `NSURLConnection` seamlessly establishes an SSL connection and fetches the data. When reviewing the source code of an iOS application, it is recommended to analyze code that uses `NSURLConnection`. It is important to understand how the connections are being initiated, how user input is utilized to construct the connection requests, and if SSL is being used or not. Watch for `NSURL*` usage, including invocations to objects of type `NSHTTPCookie`, `NSHTTPCookieStorage`, `NSHTTPURLResponse`, `NSURLCredential`, `NSURLDownload`, etc.

`74.125.224.49` is one of the IP addresses bound to the host name `www.google.com`. When browsing to `https://74.125.224.49`, the browser should display a warning due to the fact that the Common Name field in the SSL certificate presented by the server (`www.google.com`) does not match the host and domain component of the URL.

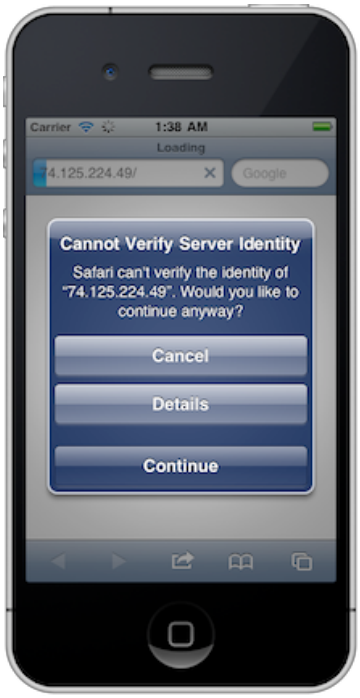


**Figure 15:** Safari on iOS warning the user due to mismatch of the Common Name field in the certificate.

As presented in the screen-shot in Figure 15, Safari on iOS does the right thing by warning the user in this situation. Common Name mismatches and certificates that are not signed by a recognized certificate authority can be signs of a Man in the Middle attempt by a malicious party that is on the same network segment as that of the user or within the network route to the destination.

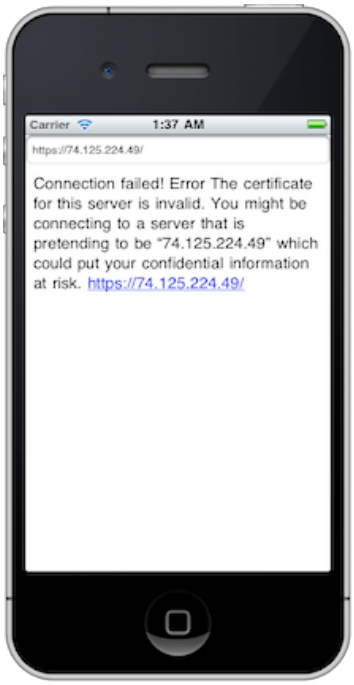


**Figure 16:** `NSURLConnection's connection:didFailWithError:` delegate is invoked to throw a similar warning.

The screen-shot in Figure 16 shows what happens if we attempt to browse to `https://74.125.224.49` using our sample app discussed earlier, the `connection:didFailWithError:` delegate is called indicating an error, which in this case warns the user of the risk and terminates.

This is fantastic. Kudos to Apple for thinking through the security implications and presenting a useful warning message to the user (via `NSError`).

Unfortunately, it is quite common for application developers to override this protection for various reasons: for example, if the test environment does not have a valid certificate and the code makes it to production. The code below is enough to override this protection outright:

```
- (BOOL)connection:(NSURLConnection *)connection
canAuthenticateAgainstProtectionSpace:(NSURLProtectionSpace *)protectionSpace
{
    return [protectionSpace.authenticationMethod
    isEqualToString:NSURLAuthenticationMethodServerTrust];
}


- (void)connection:(NSURLConnection *)connection
didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *)challenge
{

    [challenge.sender useCredential:[NSURLCredential
    credentialForTrust:challenge.protectionSpace.serverTrust]
    forAuthenticationChallenge:challenge];
}
```

There is also a private method for `NSURLRequest` called `setAllowsAnyHTTPSCertificate:forHost:` that can be used to over-ride the SSL warnings, but Apps that use it are unlikely to get through the App Store approval process (Apple prohibits invocations of private API).

*For those responsible for reviewing an organization's iOS code for security vulnerabilities, it is highly recommended to watch for such dangerous design decisions that can put your client's data and your company's data at risk.*

As part of performing security testing of applications, it is often useful to intercept HTTP traffic being invoked by the application. Applications that use `NSURLConnection`'s implementation as-is will reject your local proxy's self-signed certificate and terminate the connection. You can get around this easily by implanting the HTTP proxy's self-signed certificate as a trusted certificate on your iOS device [Note: This is not a loop-hole against the precautions mentioned above: in this case we have access to the physical device and are legitimately implanting the self-signed certificate]. Once you have your iOS device or simulator setup using the self-signed certificate of your HTTP proxy, you should be able to intercept HTTPS connections that would otherwise terminate. This is useful for fuzzing, analyzing, and testing iOS applications for security issues.

**Privacy Leaks and Identity Decloaking Attacks**

Many iOS apps leak the UDID (Unique Device ID) to third-party analytics service providers without the consent of the users.

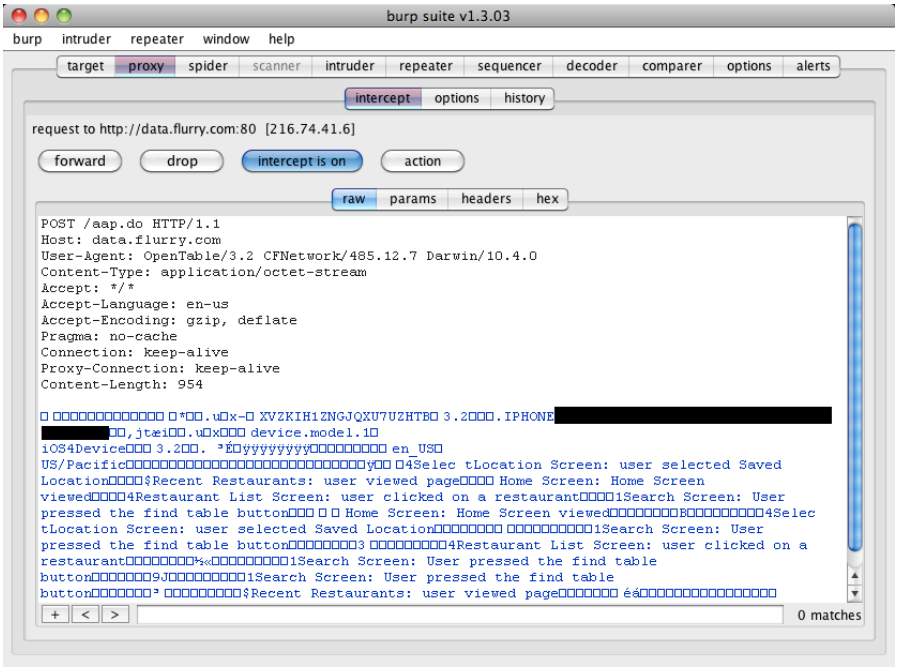


**Figure 17:** The OpenTable app sends the UDID to `data.flurry.com`

As illustrated in Figure 17, the OpenTable app sends the actual device UDID to `data.flurry.com`. This information is also sent in clear-text allowing malicious users on the same unprotected wireless network as the victim to capture this information.

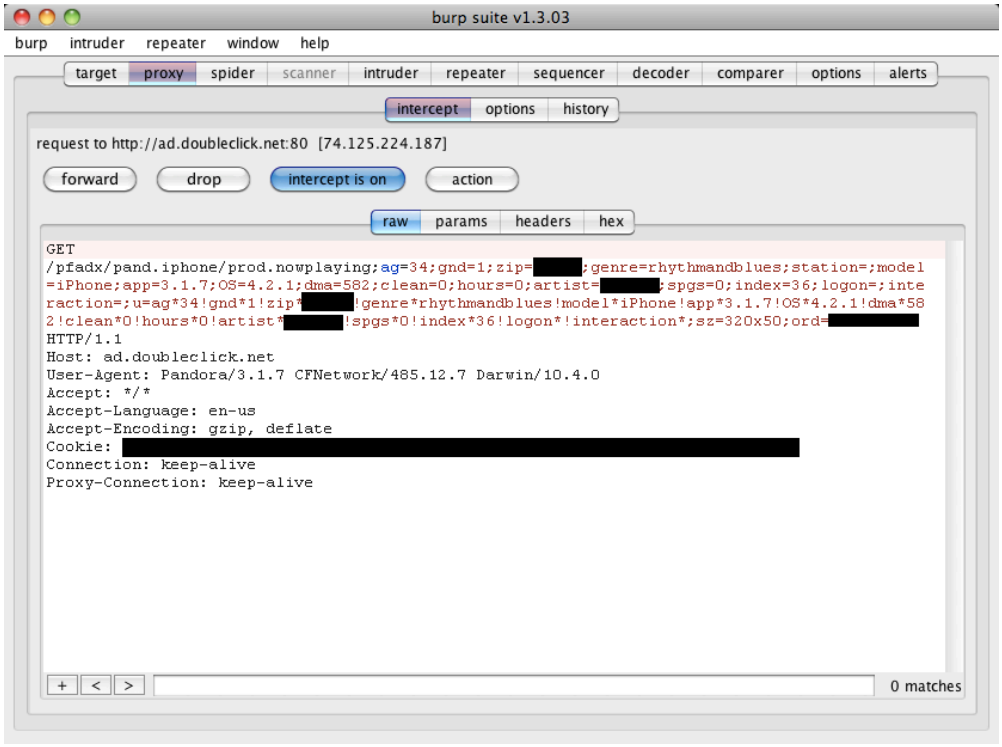


**Figure 18:** The Pandora app sends the user's age and zip code to `ad.doubleclick.net`

As illustrated in Figure 18, the Pandora app sends the user's age and zip code to `doubleclick.net`. Again, this is done in clear-text allowing malicious entities near the victim to potentially capture this information. These malicious entities, in addition to the third party analytics providers, have the ability to piece together detailed information about the end user based on the UDID, thus jeopardizing reasonable privacy expectations.

In addition to privacy leakage, unencrypted network channels are also susceptible to decloaking attacks. Consider the situation where the victim is using a wireless hotspot at a Starbucks. A malicious entity can easily perform a Man in the Middle attack (or alternatively, setup a fake access point and lure the victim to attach to it) and inject the following HTML in the response stream:

```
<iframe src="fb://profile/"></iframe>
```

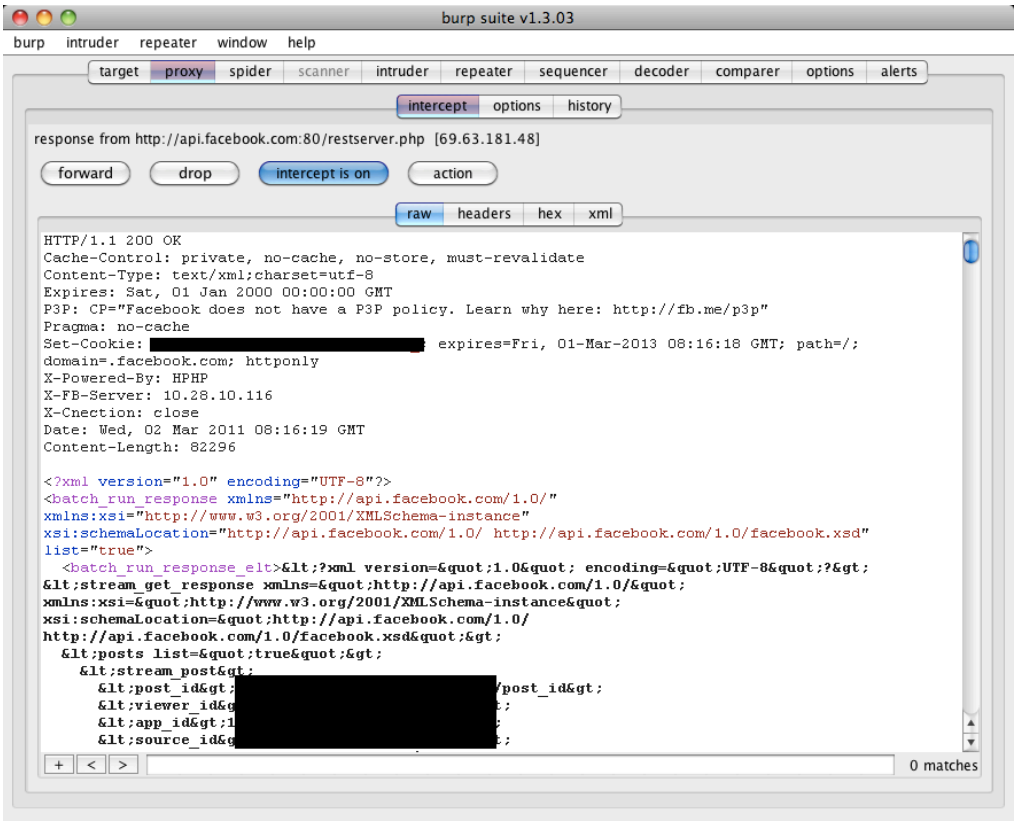


**Figure 19:** URLScheme injection on the Facebook app can be abused to decloak a victim's identity

Based on our discussion of how iOS handles URLSchemes, the `iframe` will cause the victim to be yanked out of his or her Safari browsing session onto his or her own profile on the Facebook app. Since the Facebook app does not use SSL, the malicious entity will be able to capture the victim's real identity and contents of the victim's Facebook wall (as illustrated in Figure 19).

Enterprises must assess third party Apps and custom developed Apps to make sure that secure network channels are being used and that private information is not being inadvertently leaked to third party organizations.

## Protecting Data at Rest

Mobile devices such as the iPad and the iPhone are frequently lost or stolen. As such, it becomes important to leverage data protection features in the iOS platform to encrypt the data at rest so that malicious entities who have physical access to the devices are not able to access the data. In this section, we will quickly go through the file encryption functionality provided in the iOS platform. Even though this particular topic does not contain net-new research, it is worthy of discussion since it is a top of mind concern for many business owners.

### Leveraging Data Protection

In iOS 4.0 onwards, every file is protected using a different file key. There also exists a 'file system key' that is used to protect every file. However, this file alone is not enough to decrypt the file. The 'file system key' is implemented to be able to quickly wipe the device remotely, i.e. theoretically speaking, only the 'file system key' would need to be deleted to make the files unreadable.

The key point to take away is that Apps can choose to also tie the user's passcode to the encryption mechanism. Apps can specifically request that certain files use the user enabled passcode to encrypt files and keychain entries. *This is an added on protection mechanism to protect files even if the device has been compromised, i.e. jail-broken without entering the passcode or a sandbox violation*.

In order to implement passcode protection, the following steps are recommended:

- To protect files on the file system, use the `NSDataWritingFileProtectionComplete` option when calling `writeToFile` and/or the `setAttributes:ofItemAtPath:error:` method and add the `NSFileProtectionKey` attribute with the `NSFileProtectionComplete` value to protect the file using the user passcode key.
- To protect entries in the keychain,use `kSecAttrAccessibleWhenUnlocked` or `kSecAttrAccessibleAfterFirstUnlock` attributes when calling `SecItemAdd` or `SecItemUpdate`.

### Preventing Data Exposure in Cached Screenshots

When the user presses the home button while using an app, iOS takes a screenshot of the app to simulate the zoom-out animation. The screenshot is also used to when the user re-launches the app to simulate the zoom-in animation. This presents a risk for devices that may not have a user passcode set in the situation where the user presses the home button while critical data is being displayed and the device is lost or stolen.

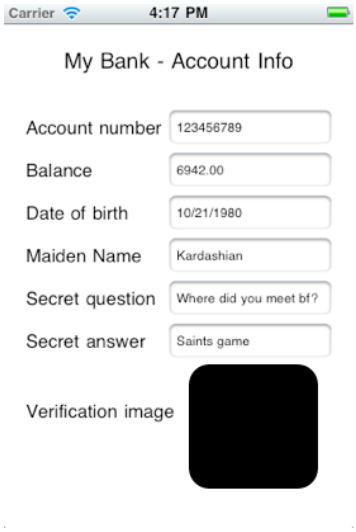


**Figure 20:** Banking app cached screenshot exposing confidential data

The screenshot images are stored in the App's cache directory on the file system. An example screenshot is illustrated in Figure 20.

A simple work-around to protecting critical data from appearing in the screenshot cache is to set window.hidden to YES in the `applicationDidEnterBackground`: delegate and window.hidden to NO in the `applicationWillEnterForeground:` delegate. This simply blanks out the UI before the screenshot is taken and redraws it when the app is re-launched. Alternatively, you can choose to hide certain UI elements instead of the entire window.

## Conclusion

The iOS revolution is here. Enterprise users are demanding the secure enablement of their iPhones and iPads at work. To accomplish this, organizations should take into account the new attack vectors presented in this document to build a solid and repeatable application assessment methodology into their Software Development Lifecycle (SDL). In addition to the new attack vectors, the Appendix included in this document is a checklist of items to consider when assessing iOS applications - this list also includes traditional application security weaknesses that also apply to iOS. Additional items to consider, such as data protection and file encryption applicable to iOS devices, are also presented in the Appendix.

## Appendix: iOS Application Security Assessment Checklist

- Input and output validate every dynamic input (user input, external HTML or database feed, URLs)
- Audit traditional unsafe methods that deal with memory (`memcpy`, `strcpy`, etc)
- Watch out for format string vulnerabilities
- grep for password strings and hard coded credentials / secrets
- grep for `NSURL`, `CFStream`, `NSStream` to locate network connections
- grep for SQL strings and SQLLite queries
- Look for `setAllowsAnyHTTPSCertificate` and `didReceiveAuthenticationChallenge` to see if certificate exceptions are being bypassed
- Locate calls to `NSLog` to see what data is being logged
- Check implementation of URLSchemes in `handleOpenURL`
- Make sure that information is being secured in the Keychain (`kSecAttrAccessibleWhenUnlocked` or `kSecAttrAccessibleAfterFirstUnlock` attributes when calling `SecItemAdd` or `SecItemUpdate`) and the file system (`NSDataWritingFileProtectionComplete`).
- Make sure `NSUserDefaults` is not being used to store critical data
- Take a look at the server side code and web-root, including implementations and payloads sent to the APN. Make sure APN certs are protected by a pass-phrase
- Pay attention to `UIWebView` implementations: Where is the HTLM being rendered from? Is the URL always visible?
- Make sure copy-paste functionality is disabled in sensitive fields (PHI, PII)
- Make sure UI fields that display critical data hide themselves in `applicationWillTerminate` and `applicationWillEnterBackground` to prevent screenshot caching
- Run the app and monitor data (Jailbreak/SSH or a tool such as PhoneView)
- Decrypt the binary and run `strings`
- Install Burp CA and monitor + fuzz HTTP/HTTPS traffic
- Watch out for leakage of UDID and/or PII/PHI to third party analytics services or in clear-text
- Make sure the server side architecture does not rely upon the iOS device to truthfully state its location (since this data can be intercepted and modified)

## About the Author

Nitesh Dhanjani is a well known information security researcher and speaker. Dhanjani is the author of "Hacking: The Next Generation" (O'Reilly), "Network Security Tools: Writing, Hacking, and Modifying Security Tools" (O'Reilly), and "HackNotes:Linux and Unix Security" (Osborne McGraw-Hill). He is also a contributing author to "Hacking Exposed 4" (Osborne McGraw-Hill) and "HackNotes:Network Security" (Osborne McGraw-Hill).

At Ernst & Young, Dhanjani is a Senior Manager in the Advisory practice, responsible for helping some of the largest corporations establish enterprise wide information security programs and solutions. Dhanjani is also responsible for evangelizing brand new technology service lines around emerging technologies and trends such as social media, cloud computing, and virtualization.

Prior to E&Y, Dhanjani was Senior Director of Application Security and Assessments at Equifax where he spearheaded security efforts into enhancing the enterprise SDLC, created a process for performing source code security reviews & threat modeling, and managed the attack & penetration team. Before Equifax, Dhanjani was Senior Consultant at Foundstone's Professional Services group where, in addition to performing security assessments, he contributed to and taught Foundstone's Ultimate Hacking security courses.

Dhanjani holds both a Bachelor's and Master's degree in Computer Science from Purdue University.