Урок 4

Безопасность мобильных приложений

Основные принципы защиты данных в приложении

Введение

Многие разработчики при работе над приложением не задумываются о безопасности, и напрасно. Большинство приложений работают с данными пользователя, часто с очень важными. Банковские приложения управляют финансами, чаты содержат личную переписку, даже простое приложение для заметок может содержать данные, которые представляют ценность для злоумышленников. Мы как разработчики должны приложить все усилия, чтобы обеспечить безопасность этих данных.

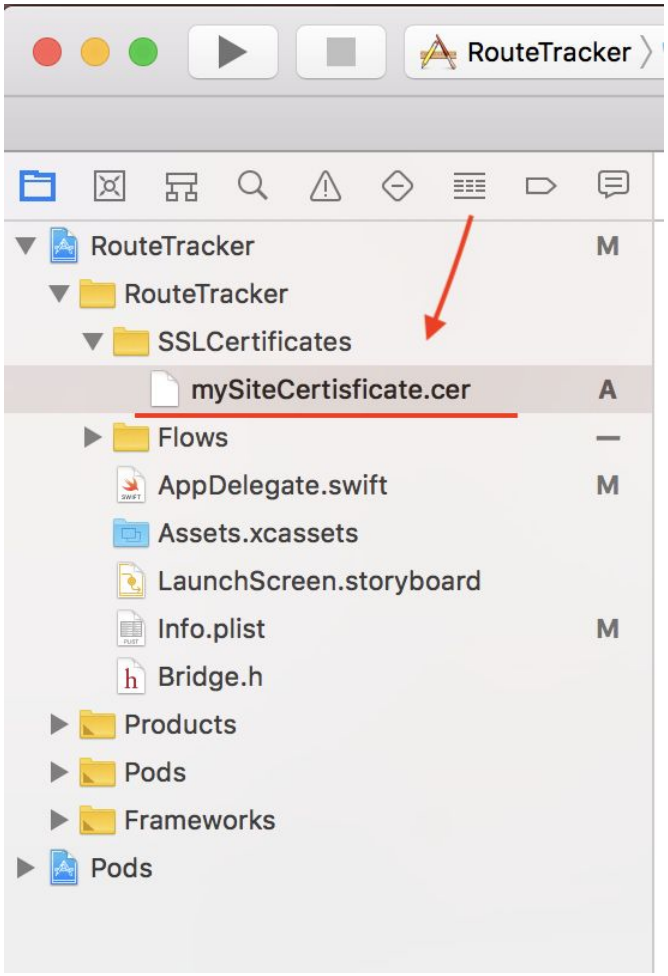
Тема информационной безопасности очень обширна, не зря же существует профессия специалиста по защите информации. В этом уроке мы познакомимся с базовыми техникам, применяемым в iOS-приложениях. Они достаточно просты и легко внедряются для обеспечения базового уровня защиты.

Перехват информации в канале обмена между приложением и сервером

Итак, вы начали работать над новым приложением и ещё только настраиваете проект, бэкэнд находится в зачаточном состоянии. Но уже на этом этапе вы можете сделать первые шаги в сторону усиления безопасности. И первое, о чём нужно позаботиться – чтобы обмен данными с сервером шёл по протоколу **https.** Данные, передаваемые по этому протоколу, будут зашифрованы, и если злоумышленник их перехватит, то не сможет их использовать. Как вы помните, сама Apple настоятельно рекомендует использовать этот протокол. По умолчанию **http** вообще не поддерживается в приложениях. Но мы можем его включить в **info.plist.** Делать это крайне не рекомендуется, и если ваш заказчик этого не понимает, вы должны его в этом убедить.

В ряде случаев злоумышленник может расшифровать https-трафик. Дополнительно повысить уровень безопасности поможет такой **SSL-пиннинг**. Что это такое? При обмене информацией по https на сервере хранится сертификат, подтверждающий, что серверу можно доверять, но проблема в том, что сертификатов, которым мы доверяем, очень много, и если злоумышленник раздобудет себе такой сертификат, он получит наш трафик. Но если мы укажем в приложении, какой конкретно сертификат используется на сервере, злоумышленник нас уже не обманет. Для этого мы можем хранить его копию, его хеш или его публичный ключ. Проще всего первый вариант, но безопаснее и удобнее в перспективе всё же второй или третий. К счастью, в Alamofire есть достаточно простая поддержка всех трех способов. Давайте попробуем реализовать первый способ, воспользовавшись специальным классом **ServerTrustPolicyManager.**

Для начала у вас должен быть файл сертификата с расширением **.cer.** Положите его в любую удобную папку в xcode, название тоже может быть любым.



Далее надо настроить сам **ServerTrustPolicyManager**. Сделаем это в в классе – фабрике сетевого слоя.

```
import Foundation
import Alamofire

final class NetworkFactory {

    private func makeServerTrustPolicyManager() -> ServerTrustPolicyManager {
// Найдём все сертификаты в приложении
        let certificates = ServerTrustPolicy.certificates()
// Политика доверия: доверять серверам с найденными сертификатами
        let serverTrustPolicy = ServerTrustPolicy.pinCertificates(
            certificates: certificates,
            validateCertificateChain: true,
            validateHost: true
        )

// Применим политику доверия к серверу https://example.com
        let serverTrustPolicies = ["https://example.com": serverTrustPolicy]
// Создадим менеджер политик безопасности на основе настроек
             let  serverTrustPolicyManager  =  ServerTrustPolicyManager(policies:
serverTrustPolicies)
        return serverTrustPolicyManager
    }

}
```

Всё делается очень просто, код почти универсальный, и единственное, что в нём меняется от приложения к приложению – адрес сервера. Он автоматически найдёт все файлы и проведёт настройку. Полученный **ServerTrustPolicyManager** надо применить при создании **SessionManager.**

```
    private func makeSessionManager() -> SessionManager {
        let policies = makeServerTrustPolicyManager()

        let configuration = URLSessionConfiguration.default
        let manager = SessionManager(
            configuration: configuration,
            serverTrustPolicyManager: policies
        )

        return manager
    }
```

SessionManager мы уже научились создавать, а здесь просто добавили параметр **serverTrustPolicyManager.**

Если решите использовать публичный ключ, что предпочтительнее, вам достаточно изменить метод вот так:

```
    private func makeServerTrustPolicyManager() -> ServerTrustPolicyManager {
// Найдём все ключи в приложении
        let publicKeys = ServerTrustPolicy.publicKeys()
// Политика доверия: доверять серверам с найденными ключами
        let serverTrustPolicy = ServerTrustPolicy.pinPublicKeys(
            publicKeys: publicKeys,
            validateCertificateChain: true, validateHost: true
        )

// Применим политику доверия к серверу https://example.com
        let serverTrustPolicies = ["https://example.com": serverTrustPolicy]
// Создадим менеджер политик безопасности на основе настроек
             let  serverTrustPolicyManager  =  ServerTrustPolicyManager(policies:
serverTrustPolicies)
        return serverTrustPolicyManager
    }
```

В проекте вы можете хранить вместо самих сертификатов их публичные ключи (в файлах с расширением **.cer**).

Теперь мы максимально защищены от перехвата информации, которой обмениваемся с сервером.

Аутентификация пользователя

Вы, конечно, помните, как мы проходили аутентификацию во «ВКонтакте», и что сервис не позволял передать логин и пароль просто так. Это сделано для обеспечения должного уровня безопасности. Давайте рассмотрим различные уровни, которые мы можем реализовать в приложении.

Как вам известно, аутентификация – проверка, действительно ли пользователь является тем, за кого он себя выдаёт. Самый простой вариант – попросить человека ввести логин. Учитывая, что логин

уникален, мы сможем определить, какой именно человек аутентифицировался в приложении. Этот способ небезопасен: любой человек, зная логин другого, может выдать себя за него.

Поэтому мы запрашиваем у пользователя ещё и пароль. Узнать пару логин/пароль гораздо сложнее, чем только логин. К тому же пароль, как правило, шифруется при хранении на сервере, и даже администрация сервера не может его узнать.

Тем не менее, даже этот вариант ненадёжен. Пароль можно подобрать, особенно если он простой. Или его можно раздобыть у пользователя обманным путём или методами социальной инженерии. Поэтому добавляем двухфакторную аутентификацию: после ввода верных логина и пароля пользователю приходит письмо на электронную почту или SMS с кодом, который необходимо указать в приложении. Это существенно увеличивает надёжность.

Заставлять пользователя аутентифицироваться каждый раз, когда он хочет воспользоваться приложением, не очень удобно, поэтому мы должны упростить процесс. К примеру, после первой успешной аутентификации в дальнейшем переложить эту роль на приложение. Для этого на сервере генерируется специальная случайная строка – токен, он передаётся приложению. Приложение сохраняет его и при каждом запросе вместо логина и пароля передает токен. Вы можете спросить, почему бы нам не сохранить логин и пароль и не передавать их каждый раз на сервер. Это небезопасно, потому что злоумышленник может перехватить данные и использовать их в своих целях. Он может использовать и токен, но токен можно заменить даже без участия пользователя.

Кроме того, существуют различные стандарты аутентификации, например, OAuth или OAuth2.0. Там используется не один а два токена. Первый токен нужен для получения информации и работает непродолжительное время, например 30 минут. По истечении этого времени его надо заменить, для чего используется второй токен, который живёт ровно до момента обновления. Этот процесс выглядит так. Вы передаёте на сервер логин и пароль, в ответ получаете два токена: рабочий и для обновления. Используете рабочий токен, пока он не устареет, после этого применяете токен для обновления, и в ответ получаете два новых токена. Чем хорош этот способ? Если злоумышленник получит рабочий токен, он сможет пользоваться им всего 30 минут. Если же он получит токен для обновления и использует его, копия токена у настоящего пользователя окажется недействительной, при первой же попытке использовать его это вскроется, и сервер сделает все токены недействительными. Злоумышленнику и реальному пользователю в этот момент предложат ввести логин и пароль. Реальный пользователь без проблем сделает это, а злоумышленник не сможет, так как он их не знает. Это довольно надёжная защита от хищения аутентификационных данных.

Хранение токенов в приложении у пользователя несет угрозу: если устройство попадёт к другому человеку, он без проблем сможет ими воспользоваться. Для этого можно закрыть доступ к приложению с помощью пин-кода. Это может быть короткий 4-значный код с ограниченным числом попыток ввода (трёх попыток вполне достаточно). В большинстве современных устройств присутствует биометрический механизм аутентификации – отпечаток пальца или распознавание лица. В таком случае после задания пин-кода можно настроить вход с помощью этих технологий.

Получение пользовательского доступа к устройству

Поговорим о рисках, которые сопряжены с тем, что мобильным устройством может завладеть другой человек и получить доступ к данным, даже не обладая специальными знаниями или инструментами.

Раз мы начали говорить о пин-коде, о нём и продолжим. Реализовать ввод пин-кода – это логичный шаг, но реализовывать его нужно правильно. Если мы сделаем обычное поле, во время ввода пин-код можно будет подсмотреть, а включённая по умолчанию автокоррекция легко запомнит введённое пользователем значение и сможет предлагать его как один из вариантов ввода. Поле ввода должно быть как минимум **isSecureTextEntry.** Кроме того, есть шанс, что пользователь установит стороннюю клавиатуру, которая имеет доступ к вводимой с её помощью информации, так что для полей с приватной информацией лучше отказаться от ввода через стандартную клавиатуру и

использовать собственную, а точнее, **view** с кнопками. Конечно, полноценную клавиатуру так реализовать непросто, но для пин-кода лучше сделать цифровой вариант, а цифры в поле ввода заменить точками. Кстати, при вводе логина мы не можем сделать поле **isSecureTextEntry**, так что не забываем отключить все виды автокоррекции для него: **input.autocorrectionType = .no.**

Разобравшись с полями, перейдем к биометрии. Хотя она и максимально защищена самой Apple, в системе можно зарегистрировать несколько отпечатков пальцев и не все они должны принадлежать одному и тому же человеку. Так, например, если кто-то узнает ваш пароль от телефона, он может добавить в базу свой отпечаток и воспользоваться приложением, защищённым этим отпечатком. Но мы можем избежать такого развития событий. **Класс LAContext**, входящий в состав фреймворка **LocalAuthentication**, имеет полезное свойство – **evaluatedPolicyDomainState**. Это слепок всех отпечатков, зарегистрированных в системе. Если в системе будет зарегистрирован новый отпечаток, то и **evaluatedPolicyDomainState** изменится. Нам достаточно сохранить его в **Keychain** после ввода пин-кода и проверять при каждом запуске приложения. Если он изменился, значит кто-то добавил свой отпечаток, и надо отказать в праве входа по отпечатку пальца и запросить ввод пин-кода. Таким образом, человеку, который заполучил пин-код от телефона, потребуется узнать пин-код от приложения, что сложнее.

Но даже если человек, получивший чужой телефон в руки, не смог попасть в приложение, остаётся ещё одно место, где он потенциально может увидеть чужие персональные данные. Это список запущенных приложений, который доступен по двойному нажатию кнопки Home. В момент, когда приложение сворачивается в фон, делается скриншот, который можно посмотреть в списке запущенных приложений. Чтобы на него не попала какая-либо важная информация, надо скрыть данные пользователя. Для этого можно сделать прозрачным **view** текущего контроллера или, наоборот, реализовать контроллер-шторку, который закроет всю важную информацию. Это делается в методе **applicationWillResignActive** в **AppDelegate.** Главное – не забудьте отменить эти изменения в методе **applicationDidBecomeActive,** иначе пользователь так и будет смотреть на пустой экран после запуска приложения из фона.

Получение привилегированного доступа к устройству

К сожалению, доступ к устройству может получить не простой пользователь, а человек с навыками взлома. В этом случае меры предосторожности не помогут. Но мы можем предпринять дополнительные меры защиты.

Во-первых, злоумышленник может даже без доступа к приложению получить доступ к данным, которые сохранены в файловой системе устройства. Именно поэтому так важно хранить всю приватную информацию в **Keychain**, а не в файлах, **UserDefaults, CoreData, Realm.** Но многие неопытные разработчики, даже используя **Keychain**, допускают ошибку. **Keychain** – это контейнер, информация в котором шифруется, но не всегда. При использовании приложения она расшифровывается. Мы можем указать, когда доступен или недоступен **Keychain:**

- пока устройство разблокировано;
- после перезагрузки и первой разблокировки;
- всегда.

Очевидно, если выбрать второй и третий вариант, данные будут доступны практически всегда или вообще всегда. Этот вариант мало чем отличается от **UserDefaults** и небезопасен, поэтому надо выбирать только первый режим. Так как при использовании разных библиотек настройка режимов производится по-разному, продемонстрировать универсальный пример нет возможности.

Воспользовавшись **Keychain**, вы можете обезопасить приватные данные. Но есть ещё одно место, где хранится информация, даже когда вы не подозреваете об этом, – кэш сетевых запросов и ответов. По умолчанию всё, что вам отправил сервер, будет сохранено в кэш, а кэш – в файловой

системе. В результате злоумышленник может получить доступ к этим данным. Чтобы отключить кэш, достаточно небольшого кусочка кода:

```
let cache = URLCache(
    memoryCapacity: 0,
    diskCapacity: 0,
    diskPath: nil)
URLCache.shared = cache
```

Этот код необходимо написать до того, как вы начнёте отправлять запросы в сеть. Идеально, если он будет исполнен перед созданием **SessionManager.**

Заключение

Не забывайте, что это только базовые принципы, которые должны быть применены по умолчанию. Но их достаточно для большинства приложений. Если вы пишете клиент для популярных сервисов, содержащих приватные данные, например банка или соцсети, вам потребуется отдельно изучить тему защиты данных. Также помните, что полной защиты не существует: вы можете усложнить жизнь злоумышленнику, но не остановить. Если кто-то захочет получить доступ к данным, он их обязательно получит.

Практическое задание

К сожалению, наше приложение не использует приватные данные пользователя, и у нас нет авторизации на сервере, поэтому мы можем применить очень мало техник для защиты.

1. Настроить параметры автокоррекции у текстовых полей на экране авторизации и восстановления пароля.
2. Настроить показ шторки при уходе приложения в фоновый режим.

Дополнительные материалы

1. Основные практики обеспечения безопасности iOS-приложений.
2. 10 самых опасных угроз для веб-приложений по версии Owasp в 2017 г.

Используемая литература

1. Сайт библиотеки ReactiveX.
2. Репозиторий библиотеки ReactiveX.