Wire Security Review – Phase 2 – iOS Client for Wire Swiss GmbH

Final Report

2018-03-07

FOR PUBLIC RELEASE

Contents

1 Summary

This report is a review of Wire's iOS application security and privacy, describing strengths and limitations, and highlighting a number of minor shortcomings and potential improvements. We also report one low-severity software bug and three medium-severity bugs. The source code audit otherwise reflected adherence to secure software development principles, as well as leverage of the platform's security features.

As noted throughout the report, Wire has adequately addressed the security issues reported.

This review does not cover:

- The core cryptography component Proteus, previously already reviewed [1] .
- The calling mechanism, covered in a separate review.
- Code of third-party dependencies.

The work was performed between August and November 2017, by Jean-Philippe Aumasson (Kudelski Security) and Markus Vervier (X41 D-Sec GmbH). A total of 10.75 person-days were spent.

[1] `https://www.x41-dsec.de/reports/Kudelski-X41-Wire-Report-phase1-20170208.pdf`

2 iOS Client Review

We reviewed security and privacy features of the Wire iOS client, mainly based on the source code review, as well as on dynamic analysis of the official Wire application using a jailbroken device and test devices. The following sections review how the Wire iOS application protects against various technical risks, highlighting potential issues and providing mitigation recommendations. Several observations were made on older versions of the applications (as of August 2017), but all observations reported in this version of the report apply to the version available mid November 2017.

Subject of the review were the following repositories and commits:

- wire-ios-cryptobox commit 8e96cccb35b259187e590ca75e969cde7a6e2efd
- wire-ios-data-model commit 54302381324dfcd017154d4e59590905610d956e
- wire-ios-images commit ba105ea632251fcadd9f388ed319e8300164ace9
- wire-ios-message-strategy commit 22e7c4210e10acfc37963b3f2e2e108496a48190
- wire-ios-share-engine commit 1aba9b92931073d6050cecc18059863cf7ebe1d2
- wire-ios-sync-engine commit dd66a9fc2a056e320a2f62fca277cb233042147f
- wire-ios-system commit db0abf2504f16b4c2ae67e6feaa454d1c08d843b
- wire-ios-transport commit 967db8121677940aa59b101139b6731b61977a3c
- wire-ios-utilities commit 55f2bf21439d49b0018de18de71eb7c9a4658ba7

2.1 PRIVACY

This section summarizes our review of potential privacy leaks.

2.1.1 Logs Leaks

Logs written (in non-debug mode) do not leak critical credentials, but may leak information on the user's identity and its activity. For example, URLs are logged in error messages in `SharedObjectStore.swift`:

```
1 zmLog.error("Failed to write to url: \(url), error: \(error), object: \(object)")
2 zmLog.error("Failed to read from url: \(url), error: \(error)")
3 zmLog.error("Failed to remove item at url: \(url), error: \(error)")
```

Listing 2.1: Logging

Furthermore, a user ID is logged in an error message in `MissingClientsRequestStrategy.swift`:

```
1 zmLog.error("\(userIdString) is not a valid UUID")
```

Listing 2.2: UserID Log

We recommend to review whether such logs are really necessary.

2.1.2 Uninstall Leftovers

After uninstalling the application, all files under `/private/var/mobile/Containers` are removed, as well the localytics folder in file sharing, and the application. One can nonetheless find out that Wire used to be installed on the device, for example by looking at iOS log files, where the mention of "wearezeta" (Wire's original codename) reveals the previous installation of the app.

```
1 lockdownd.log:  0 : <CFString 0x13c603b60 [0x1a1785b68]>{contents = "com.wearezeta.zclient.ios"}
2 lockdownd.log.1:        0 : <CFString 0x14c60b360 [0x19e399b68]>{contents = "com.wearezeta.zclient.ios"}
```

Listing 2.3: Uninstall Leftovers

However, this behavior cannot be modified by the application's developers, as it is a property of the OS.

2.1.3 Telemetry Data

Localytics is used to perform analytics based on user usage data. Users can opt out of this tracking in the settings, but by default data such as the following is tracked:

- Type of connection (WiFi, 4G, etc.).
- The time, size, and type of assets exchanged (photo, audio, video, or other files).
- Calls duration and type (audio or video), etc.

Such data can obviously leak privacy-sensitive information, even if the data is shared anonymously. Wire migrated to Mixpanel during the review. The reviewed code was still using Localytics.

2.1.4 Crash Reports

HockeyApp is used to collect and crash reports from the application and share them with Wire for QA purposes. These reports do not seem to include obvious sensitive information. But stack traces and exception messages may reveal potentially sensitive information. Users can opt out of this tracking in the settings.

2.1.5 Screen Copies

The application does not attempt to protect against screenshots.

By default the iOS app switcher shows a preview of the current Wire screen, and does not attempt to hide sensitive data such as conversations content. However, the application lock mechanism can be activated to require authentication before using the app, and this will hide the content of the application in the app switcher.

2.1.6 Files Metadata

Images sent over the app are stripped of their metadata such as for example geotags. This applies to photos taken from the app or pictures from the image gallery. The code performing the clean-up is in `NSData+MediaMetadata.swift` found in the `wire-ios-image` repository.

Location information is removed from videos, both when sent as a file and when sent from the image gallery. This seems to happen through transcoding of the files from the original format (e.g. .mov) to compressed MP4 format.

Other files uploaded left unchanged. This includes office documents, audio files, PDFs, and other documents.

2.1.7 URL Previews

By default, previews of URLs are generated when an URL is entered, thereby generating a DNS request and subsequent HTTP requests to fetch the preview content. This behavior leaks the address of the sender to the URL's servers, and reveals that the URL was sent.

Wire has addressed this issue by allowing users to disabled URL previews under Settings>Options.

2.1.8 iCloud Backups and iTunes Backups

Wire data is excluded from iCloud and iTunes backups by the application, as implemented in `FileBackupExcluder.swift`.

2.1.9 Screen Snapshots

App switcher screen snapshots are stored under `/private/var/mobile/Containers/Data/Applications/<appID>/Library/Caches/Snapshots/`. However, any information will appear on the snapshot, for example the password length and the clear text value of the last password character on the login page. Such sensitive information should be hidden, using the `applicationDidEnterBackground` method. Please see also section

2.1.5 for further information.

2.1.10 Keyboard Caching

Keyboard caching is disabled for password fields (through `.secureTextEntry=YES`) as well as for username and email fields (through `.autocorrectionType=UITextAutocorrectionTypeNo`). This ensures that sensitive data is not recorded by the OS.

2.2 Cryptography

This section reports on the main cryptographic threats against the application.

2.2.1 Pseudorandom Generator

The PRNG used for cryptographic purposes relies on the iOS SDK's `SecRandomCopyBytes` using the default system secure PRNG (`kSecRandomDefault`), and properly checking for success:

```
1 + (NSData *)secureRandomDataOfLength:(NSUInteger)length
2 {
3     NSMutableData *randomData = [NSMutableData dataWithLength:length];
4     int success = SecRandomCopyBytes(kSecRandomDefault, length, randomData.mutableBytes);
5     Require(success == 0);
6     return randomData;
7 }
```

Listing 2.4: PRNG

It would be slightly better practice, and clearer, to check for `success == errSecSuccess` rather than `success == 0`, although the value of `errSecSuccess` is unlikely to change in future versions of the SDK.

A non-cryptographic and insecure PRNG is used at several places (`arc4random_uniform`), but none of these uses are cryptography- or security-related contexts. After reviewing all these uses, we believe that the insecurity of `arc4random_uniform` does not impact the security of the application.

Wire has addressed these issues by checking for `success == errSecSuccess`[1] and adding a `secureRandomNumber()` method to generate random numbers with stronger randomness[2].

2.2.2 Cryptobox Integration and Usage

The `wire-ios-cryptobox` repository defines a Swift wrapper around the C version of cryptobox, abstracting out the Proteus functionalities to simple encryption session and encrypt/decrypt functions. We did not find security issues in this component.

2.2.3 Password Protection

TLS-encrypted passwords are hashed using scrypt by the server before being stored, as per the security white paper using parameters $N = 2^{14}, r = 8, p = 1$, with a random 256-bit salt, which we could verify in the server code at `https://github.com/wireapp/wire-server`. This choice of password hashing provides strong protection against password cracking attacks.

2.3 Storage

This section reports on the data stored on the device.

2.3.1 Keychain Items Not Removed Under Certain Conditions

iOS Keychain Services is only used to store the user's authentication cookie, in `kSecAttrAccessibleAfterFirstUnlock` mode (data is accessible between first unlock til reboot). We suggest using the stronger restriction of `kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly`, to make the Keychain data inaccessible on any other device.

Under certain conditions data from the Keychain might not be removed. The data might fail to be deleted without any visible error when calling the function `deleteKeychainIt`

[1] See `https://github.com/wireapp/wire-ios-utilities/pull/30`
[2] See `https://github.com/wireapp/wire-ios-utilities/pull/32/`

ems as defined in `wire-ios-transport/Source/Authentication/ZMPersistentCookieStorage.m` line 134:

```
1  - (void)deleteKeychainItems
2  {
3      dispatch_sync(isolationQueue(), ^{
4          NonPersistedPassword[self.cookieKey] = nil;
5
6          if (KeychainDisabled) {
7              return;
8          }
9
10         [ZMKeychain deleteAllKeychainItemsWithAccountName:self.accountName];
11     });
12 }
```

Listing 2.5: Delete Keychain

If this function is called while the value `KeychainDisabled` is set to `true`, data that might reside inside the Keychain is never deleted. If this could happen depends on the usage of this function and the time it is called.

We recommend to design functions and APIs in such a way that they will never silently fail. This prevents silent error conditions that might prevent for example private data from being deleted.

Wire has addressed the issue by removing the logic preventing the deletion of items[3].

2.3.2 File System

Data stored in the file system is encrypted, meaning that data becomes available to the app after the device boot once the user unlocked the device. The mode used is `NSFileProtectionCompleteUntilFirstUserAuthentication`.

This is not the highest level of protection, but this has the advantage of allowing message previews (otherwise the app would have to wait til the user unlocks the device). We believe that the the loss in security is acceptable. Database files, cryptographic keys, and other files are stored in a shared container under `/private/var/mobile/Contai`

[3] See `https://github.com/wireapp/wire-ios-transport/pull/72`

ners/Shared/AppGroup/<appgroupID>/. A shared container is used rather than a data container in order to allow for share extensions, as discussed by Wire in a blog post [4] .

In particular, messages are stored under com.wearezeta.zclient.ios/ in the store.wiredatabase SQLite database, and prekeys (including the last resort key) and the identity key are stored under otr/.

Screen snapshots (performed by iOS' app switcher), crash reports, logs, and other files are stored in a non-shared data container under /private/var/mobile/Containers/Data/Application/<appID>/.

These are reasonable design choices that prevent third-party applications from accessing the application's data.

2.3.3 PropertyList Files

We did not find sensitive information stored in unencrypted plist files.

2.4 Network

The following issues are related to network security and transport encryption.

2.4.1 TLS Connection

No HTTP content has been detected except when accessing HTTP links provided to the application by external applications, as well as link previews. All traffic otherwise seems to be safely encrypted and served using strong HTTPS components, in particular, traffic to backend servers.

The App Transport Security (ATS) mechanism enforces TLS 1.2 and safe cipher suites. Note however that the configuration of the app sets the NSAppTransportSecurity subkey NSAllowsArbitraryLoads to True, in order to allow for link previews. There is thus no strict enforcement of HTTPS connections.

[4] https://medium.com/@wireapp/the-challenge-of-implementing-ios-share-extension-for-end-to-end-encrypted-messenger-dd33b52b1e97

2.4.2 Pinning

The application performs certificate pinning to enforce the usage of specific CA certificates. The most relevant source code files are, in the wire-ios-transport repository, `ZMServerTrust.m` and `ZMURLSession.m`.

A DigiCert root certificate is hardcoded, pinned in order to validate certificates of hosts matching the following conditions:

```
1     if (   [host hasSuffix:@"prod-nginz-https.wire.com"]
2     || [host hasSuffix:@"prod-nginz-ssl.wire.com"]
3     || [host hasSuffix:@"prod-assets.wire.com"]
4     || [host hasSuffix:@"www.wire.com"]
5     || [host isEqualToString:@"wire.com"]) {
6     pinnedKeys = @[CFBridgingRelease(wirePublicKey())];
7  }
```

Listing 2.6: Pinning

These domains include the testing and production backend servers of the application.

However, the root certificate provided to validate the CDN server's certificate only uses a 1024-bit RSA modulus, which is insufficient (we recommend 2048-bit or greater).

Wire has addressed the issue by directly pinning the 2048-bit public leaf key, the above 1024-bit CA was only in an earlier version of the application that we reviewed in Q2 2017. We still report it here for completeness.

2.5 Platform

This section reports on general security risks related to the iPhone platform.

2.5.1 Permissions

Permissions of the app are defined in `Wire-iOS/Wire-Info.plist`, and include the following: `NSCameraUsageDescription`, `NSPhotoLibraryUsageDescription`, `NSPhotoLibraryAddUsageDescription`, `NSContactsUsageDescription`, `NSLocationWhenInUseUsageDescription`, `NSMicrophoneUsageDescription`.

These are expected permissions to grant access to the camera, photos, contacts list, location, and microphone (for calls). All these permissions are expected, and none seems abusive.

2.5.2 Jailbreak Detection

The app does not include any attempt of jailbreak detection or any anti-debug mechanisms. Anti-debug protection is sometimes used to complicate reverse engineering, but this isn't a concern here since the application being fully open-source.

2.6 Code Quality

This section reports on general issues regarding the code used in the application.

2.6.1 Exploit Mitigation

The application binary has position-independent code (PIE) and implements reference counting for objective-C code (in addition to Swift code). Stack protection is also enabled. These mitigation are best practice and help to defend against memory corruption attacks.

2.6.2 Native Libraries

The following native libraries are directly used by Wire components of the iOS app: libavs, libcryptobox, and libsodium. The first two libraries are provided by Wire, though libavs includes some third-party code. Both libraries were reviewed in a previous phase (partially for AVS).

Libsodium was recently reviewed [5] and found to be "a secure, high-quality library that meets its stated usability and efficiency goals."

[5] https://www.privateinternetaccess.com/blog/2017/08/libsodium-v1-0-12-and-v1-0-13-security-assessment/

2.6.3 Dependencies

Third-party dependencies include frameworks and programs from the following GitHub repositories:

```
    if (   [host hasSuffix:@"prod-nginz-https.wire.com"]
    || [host hasSuffix:@"prod-nginz-ssl.wire.com"]
    || [host hasSuffix:@"prod-assets.wire.com"]
    || [host hasSuffix:@"www.wire.com"]
    || [host isEqualToString:@"wire.com"]) {
    pinnedKeys = @[CFBridgingRelease(wirePublicKey())];
}
```

Listing 2.7: Dependencies

Most of these are coded in Obj-C or Swift and won't directly process user input, but some represent a significant risk, for example ZipArchive's C (de)compression code. ZipArchive for example uses a relatively weak password-based key derivation (in `pwd2key.c`), but this feature is not used in the Wire app and therefore does not create a security risk.

Note however that we have not reviewed the security of all these dependencies.

2.7 Implementation Security Issues

Security issues that are resulting from implementation level vulnerabilities are described here.

All of these issues have been addressed by Wire, and we reviewed the relevant patches to confirm their effectiveness.

2.7.1 WIRE-iOS-01: Hardcoded Sec-WebSocket-Key Header Value

Severity:	*LOW*
CWE:	*547*

2.7.1.1 Description

A hardcoded value was found for the WebSocket request header `Sec-WebSocket-Key`. The code defined in `Source/PushChannel/ZMWebSocket.m` line 119 [6] defines value *dGhIIHNhbXBsZSBub25jZQ==*:

```
1     if (   [host hasSuffix:@"prod-nginz-https.wire.com"]
2     || [host hasSuffix:@"prod-nginz-ssl.wire.com"]
3     || [host hasSuffix:@"prod-assets.wire.com"]
4     || [host hasSuffix:@"www.wire.com"]
5     || [host isEqualToString:@"wire.com"]) {
6     pinnedKeys = @[CFBridgingRelease(wirePublicKey())];
7 }
```

Listing 2.8: Hardcoded WebSocket Key

This value is a dummy value specified in RFC6455 [7] . The **RFC!** also makes it mandatory to always use a random base64 encoded nonce for the header value:

> The request MUST include a header field with the name |Sec-WebSocket-Key|. The value of this header field MUST be a nonce consisting of a randomly selected 16-byte value that has been base64-encoded (see Section 4 of [RFC4648]). The nonce MUST be selected randomly for each connection.

As per the WebSocket FAQ [8] , the `Sec-WebSocket-Key` header serves to ensure that "the client can verify that they are indeed talking to a WebSocket server and not to some

[6] `https://github.com/wireapp/wire-ios-transport/blob/967db8121677940aa59b101139b6731b61977a3c/Source/PushChannel/ZMWebSocket.m#L119`

[7] `https://tools.ietf.org/html/rfc6455`

[8] `https://trac.ietf.org/trac/hybi/wiki/FAQ`

other kind of server." In particular, it prevents a caching proxy from replaying a previous WebSocket conversation.

Wire fixed [9] this issue by using a random nonce for each connection.

2.7.1.2 Solution Advice

A random nonce should be generated for each connection and request as specified by RFC6544.

[9] `https://github.com/wireapp/wire-ios-transport/pull/71`

2.7.2 WIRE-iOS-02: Crash from Previews of Malformed PDFs

Severity:	*MEDIUM*
CWE:	*248*

2.7.2.1 Description

When loading certain malformed PDFs to send to a peer, the app will crash, allegedly in file preview generation. For example, a file where we modified the header to include the following object crashed the app:

```
<<
/Pages 0 0 R
/Type /Catalog
/OutputIntents 0 0 R
/Metadata 0 0 R
>>
```

Listing 2.9: Malformed PDF

We have not investigated in details all the possible causes of a crash, but one possible cause is a division by zero in the `generatePreview()` function in `FilePreviewGenerator.swift`:

```
public func generatePreview(_ fileURL: URL, UTI: String, completion: @escaping (UIImage?) -> ()) {
    var result: UIImage? = .none

    ...

    UIGraphicsBeginImageContext(thumbnailSize)
    let pdfRef = CGPDFDocument(CGDataProvider(url: fileURL as CFURL)!)
    let pageRef = pdfRef?.page(at: 1)

    let contextRef = UIGraphicsGetCurrentContext()
    contextRef?.setAllowsAntialiasing(true)

    let cropBox = pageRef?.getBoxRect(CGPDFBox.cropBox)
    let xScale = self.thumbnailSize.width / (cropBox?.size.width)!
    let yScale = self.thumbnailSize.height / (cropBox?.size.height)!
    let scaleToApply = xScale < yScale ? xScale : yScale

    ...
```

Listing 2.10: Preview

Note that there is a similar risk of division by zero when resizing images to thumbnail format, in

```
1   func ScaleToAspectFitRectInRect(_ fit: CGRect, into: CGRect) -> CGFloat
2   {
3       // first try to match width
4       let s = into.width / fit.width
5       // if we scale the height to make the widths equal, does it still fit?
6       if (fit.height * s <= into.height) {
7           return s
8       }
9       // no, match height instead
10      return into.height / fit.height
11  }
```

Listing 2.11: Division By Zero

However, we failed to trigger a crash when sending a 0×0 bitmap file, so can't confirm the root cause. Checks for division by zero were introduced [10] by Wire to mitigate this issue.

2.7.2.2 Solution Advice

Check the value of the width and height parameters to avoid a division by zero, catch any exception that may be thrown by the `CGPDFDocument` instantiating. Then test using a corpus of malformed PDF files.

[10] `https://github.com/wireapp/wire-ios/pull/1414`

2.7.3 WIRE-iOS-03: Potential Format String Vulnerabilities

Severity:	*MEDIUM*
CWE:	*134*

2.7.3.1 Description

Potentially insecure usage of functions that process format strings was found in several parts of the code.

In `ZMUserSession+UserNotificationCategories.m` the message `localizedStringWithFormat` is used with value that might be controlled by an attacker:

```
1         - (UIMutableUserNotificationAction *)mutableAction:(NSString *)actionIdentifier
2                                       activationMode:(UIUserNotificationActivationMode)activationMode
3                                    localizedTitleKey:(NSString *)localizedTitleKey
4 {
5     UIMutableUserNotificationAction *action = [[UIMutableUserNotificationAction alloc] init];
6     action.identifier = actionIdentifier;
7     action.title = [NSString localizedStringWithFormat:ZMPushActionLocalizedString(localizedTitleKey),
    ↪   nil];;
8     action.destructive = NO;
9     action.activationMode = activationMode;
10    action.authenticationRequired = false;
11    return action;
12 }
13 ...
14
15 - (UIUserNotificationAction *)replyActionDirectMessage:(BOOL)isCallContext
16 {
17    NSString *localizedTitleKey = isCallContext ? @"call.message" : @"message.reply";
18    UIMutableUserNotificationAction *action = [self
    ↪   mutableBackgroundAction:ZMConversationDirectReplyAction localizedTitleKey:localizedTitleKey];
19    if ([action respondsToSelector:@selector(setBehavior:)]) { // This is only available in iOS9
20        action.behavior = UIUserNotificationActionBehaviorTextInput;
21        NSString *sendButtonTitle = [NSString
    ↪   localizedStringWithFormat:ZMPushActionLocalizedString(@"message.reply.button.title"), nil];
22        action.parameters = @{UIUserNotificationTextInputActionButtonTitleKey: sendButtonTitle};
23    }
24    return action;
25 }
```

Listing 2.12: Format Strings

Additionally the same function could be used in file `ZMLocalNotificationLocalization.m` in an insecure way:

```
1 static NSString *localizedStringWithKeyAndArguments(NSString *key, NSArray *arguments)
2 {
3     switch(arguments.count) {
4         case 0:
5             return [NSString localizedStringWithFormat:key, nil];
6             ...
```

Listing 2.13: localizedStringWithFormat Format String

In case an attacker can control a format string, this can lead to memory corruption, information exposure, and other data representation problems.

In the given time it could not be fully verified if all of the functions mentioned above are processing external and potentially untrusted data. However, it is advised to create all internal and external API functions in a safe way.

Wire fixed [11] this issue by adding additional checks and using a safer API.

2.7.3.2 Solution Advice

Format strings should always be specified explicitly for all calls to format string processing functions. In general it is recommended to design all APIs and functional interfaces in a safe to use way. This means that for example the contents of a potentially untrusted string should not have any side effects or contain control data such as format strings.

[11] `https://github.com/wireapp/wire-ios-sync-engine/pull/687`

2.7.4 WIRE-iOS-04: Message Nonce Used to Index Asset Cache

Severity:	*MEDIUM*
CWE:	*123*

2.7.4.1 Description

A cache is used to store assets such as images or other data sent via encrypted Proteus messages.

The cache is defined in file `AssetCache.swift` of repository `wire-ios-data-model` and has several subclasses such as for example `ImageAssetCache` and `FileAssetCache`.

When data from a message is added to the cache the message nonce is usually used. One example is file `LinkPreviewAssetDownloadRequestStrategy.swift` of repository `wire-ios-message-strategy`:

```
1    func handleResponse(_ response: ZMTransportResponse!, forMessage message: ZMClientMessage) {
2        guard response.result == .success else { return }
3        let cache = managedObjectContext.zm_imageAssetCache
4
5        let linkPreview = message.genericMessage?.linkPreviews.first
6        guard let remote = linkPreview?.remote, let data = response.rawData else { return }
7        cache?.storeAssetData(message.nonce, format: .medium, encrypted: true, data: data)
```

Listing 2.14: Message Nonce

If the message nonce could be controlled by an external party, unrelated messages from different conversations could be stored in the cache. This could lead to confusion where two messages have the same nonce. This is of course an invalid state, however it should be checked if this situation could happen. This is still subject of ongoing reviews.

Wire changed [12] the asset key generation in order to avoid collisions.

[12] `https://github.com/wireapp/wire-ios-data-model/pull/436`

2.7.4.2 Solution Advice

To rule out any possibility of invalid cache behavior, it is recommended to add additional unique data to the caching key. Also collisions in the insertion of cache entries should cause an error - where possible.

2.8 AUTHENTICATION

Working and secure authentication is one of the most important things for application security. We reviewed the authentication process in he Wire iOS app and describe our findings in the following.

2.8.1 Tokens Content and Signature

For authentication to the Wire backend an access token is used. This access token is retrieved using a long term cookie value that is stored by the app. The app stores this value encrypted in the Keychain. It is encrypted by a random key using AES-256 in CBC mode in extension `zmEncryptPrefixingIVWithKey` of `NSData`, defined in file `wire-ios-utilities/Source/NSData+ZMSCrypto.m`. The key used for encryption is stored in the user defaults database (NSUserDefaults) and generated randomly as defined in file `wire-ios-utilities/Source/NSUserDefaults+SharedUserDefaults.m`, line 62:

```
1 //create new key
2 NSMutableData *newKey = [NSMutableData dataWithLength:kCCKeySizeAES256];
3 int success = SecRandomCopyBytes(kSecRandomDefault, newKey.length, (uint8_t *) newKey.mutableBytes);
4 Require(success == 0);
5 key = newKey;
```

Listing 2.15: Token Generation

In general the process of key generation is considered to be reasonably secure. The key is also stored in the user defaults database of the Wire application, which is local. Data stored in the Keychain is therefore not accessible, even if the Keychain is extracted and used on another device.

3 About

Kudelski Security
route de Genève, 22-24
1033 Cheseaux-sur-Lausanne
Switzerland

Kudelski Security is an innovative, independent Swiss provider of tailored cyber and media security solutions to enterprises and public sector institutions. Our team of security experts delivers end-to-end consulting, technology, managed services, and threat intelligence to help organizations build and run successful security programs. Our global reach and cyber solutions focus is reinforced by key international partnerships. Kudelski Security is a division of Kudelski Group.

For more information, please visit `https://www.kudelskisecurity.com`.

X41 D-Sec GmbH
Dennewartstr. 25-27
D-52068 Aachen
Germany

X41 D-Sec is an expert provider for application security services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world class security experts enables **X41 D-Sec** to perform premium security services.

Fields of expertise in the area of application security are security centered code reviews,

binary reverse engineering and vulnerability discovery. Custom research and a IT security consulting and support services are core competencies of **X41 D-Sec**.

For more information, please visit `https://www.x41-dsec.de`.