SECURITY ANALYSIS OF SIGNALS DATA STORAGE MECHANISMS IN IOS VERSION

Michał Glet

Institute of Mathematics and Cryptology, Faculty of Cybernetics, Military University of Technology
e-mail: michal.glet@wat.edu.pl
Poland

Abstract: Signal is one of the most popular instant messengers for iOS mobile devices that offers many security features. Since 2015 when Edward Snowden called Signal "very good", the application has been used by millions of people all over the world. By many people, it is recognized as the most secure among other competitors. It uses very well analysed cryptographic protocols that secure messages during the transport over the network. It keeps sent and received messages in local encrypted database stored at the iOS device. In this paper, I analyse the security of data storage mechanisms used in the iOS version of the Signal communicator, describe how Signal encrypts database and where database files are stored. I show that architecture and assumptions of storage mechanism has some security vulnerabilities that can lead to recovery of encryption key that secures application database. I show some of the possible vectors of attack and give recommendations that can make Signal's database encryption key much more secure.

Key words: Signal, security of data storage mechanisms, iOS, instant messenger, key recover.

1. INTRODUCTION

Signal is an open source instant messaging application available for Android, iOS, and desktops (Windows and macOS). Android and iOS versions offer additional features, such as voice and video calling. It is said that Signal uses end-to-end encryption protocols, which secure messaging, voice and video communication. To secure text messages Signal uses so called Signal Protocol (formerly known as the TextSecure Protocol [1]), which was audited by cryptographic and security community and it is said to be secure [1, 2]. Therefore, communication with the Signal application is thought to be secure by many people.

They think that every message they have sent or received is secure and no one can read it. However, they very often forget that those messages can be stored on their mobile devices. The way that application stores data is as much important to overall security as important is network security. That is why I reviewed the mechanisms that Signal for iOS uses to store data - I wanted to know if my data stored in Signal are private and confident under all circumstances.

In this paper I recalled recent studies about iOS storage security. Then I analyzed source codes of iOS version of Signal to get know how Signal storage mechanism work. Then I have checked in runtime (iOS simulator) where Signal database had been located and if it had been encrypted. Lastly, I showed some possible vectors of attack and gave recommendations that could make Signal storage mechanism much more secure.

2. STATE OF THE ART

Over the past few years, researchers have mainly focused on security analysis of cryptographic protocols and cryptographic primitives that Signal is using while sending text messages and during making calls. Most of these researches conducted that Signal Protocol is good enough to treat communication as secure [1, 2]. Signal Protocol provides property called end-to-end encryption. This mean that messages are transmitted over the network in encrypted form and decryption is performed only on the recipient device. None other parties between the sender and the recipient are able to decrypt messages. From cryptographic point of view Signal uses e.g. the variations of Diffie-Hellman protocol [3], the AES block cipher [4] with 256-bit key working in Cipher Block Chaining mode [5], the Double Ratchet algorithm [1], and the HMAC-SHA2 algorithm [6].

To achieve high level of security and data confidentiality instant messenger applications like Signal must also deal with security of data storage at mobile device. Mobile devices are prone to be lost, stolen or confiscated, so keeping its data safe in such conditions is a crucial task. Without secure enough data storage mechanisms users cannot be sure that messages and other data will stay confident in such circumstances. When we consider security of Signal's data storage mechanisms, there are only few publications about it. One of the most important is work from 2015 [7], where group of researches analysis security of data protection mechanisms in the early version of Signal application. Unfortunately, most of published papers describes Android version of Signal messenger. I did not find in publicly available research databases any publication that deals with security of data storage mechanism in Signal application for iOS (search done 20.10.2019).

Security of data at rest (e.g. application database) is very important especially when we consider applications that are said to provide high level of messages confidentiality and privacy. Security of data in many applications rely only on the security mechanisms introduced in iOS operating system. Unfortunately, many papers show how to bypass iOS security mechanisms and how to get accesses to data and files that are store on the mobile device [8, 9, 10]. Over last few years there were several forensics papers that show how to analyse data in iOS operating system [9, 11, 12]. What is more, there are private companies, like Pegasus [13], that provide forensics software for government agencies that is able to extract any data stored in iOS device. Even more there is software called "Jailbreak" [14]. It is often published as series of iOS kernel patches that escalates 3rd party application privileges in iOS operating system. The main purpose of such a software is to remove system restrictions introduced by Apple in their mobile operating system. Unfortunately, this privilege escalation can also be used to steal files and other data from applications (e.g. from Signal) by simply bypassing iOS security mechanisms and sandbox runtime environment [9, 10, 15].

As we can see, relying only on security mechanisms of iOS operating system is not enough for applications that need to keep its data confident, private and secure.

3. ANALYSIS OF STORAGE MECHANISMS

Security of data at rest should be a vital part of every application that is said to be secure. Unfortunately, it is very often underdeveloped. Data storage mechanisms should ensure that application data are stored in a form that is confident and private. Only the user should be able to read these data. Data stored without any security mechanism are exposed to being stolen by unauthorized entities (e.g. process, attackers, organizations, government agencies).

The simplest way to secure data at rest is to store them in an encrypted form. If you select the encryption method carefully and use it correctly, data will be written in a form that only entities which know the encryption keys and other necessary data will be able to read. This approach, unfortunately, creates a new problem – how to store the encryption keys and other data necessary for the encryption mechanisms.

In this article, I analyze how Signal stores application data and, if it uses encryption, how it stores data needed by the encryption mechanisms. This analysis is crucial to answer if Signal can still be considered a secure application that defends users' privacy against other entities (e.g. process, attackers, organizations, government agencies).

The first step of the analysis of Signal's storage mechanisms was performed by reviewing source codes of the version 2.39.2 build 2.39.2.0. Source codes have been downloaded from Git repository available at https://github.com/signalapp/Signal-iOS (according to state at 15.06.2019). The Signal application was built using XCode 10.2.1 running under the control of the macOS Mojave 10.14.4 operating system. The analysis of the data storage mechanism was carried out by a static analysis of Signal's source code and by analyzing the way that application works in database (storage) subsystem. The second step of the analysis was made using iPhone Xr simulator running under the control of iOS 12.2 operating system. This step consisted of identification where are database files located and reviewed them content to check if they were encrypted.

Signal in the iOS uses mainly the SQLite database system with the SQLCipher add-on to store data. SQLCipher is used to encrypt the contents of the database. The encryption used in the Signal application uses the AES algorithm with a 256-bit key. AES works in a very popular Cipher Block Chaining (CBC) mode.

According to Signal’s source codes that being analyzed, data and control flow depict that database is created during the first application launch and in any situation in which no relevant database files are found.

During the initialization of the database, data for SQLCipher related to encryption mechanisms are generated:

1) The encryption key and the initialization vector used in the AES-256-CBC algorithm are created. The encryption key and initialization vector are generated in the form of a continuous table composed of 48 bytes. The first 32 bytes are used as the encryption key and the next 16 bytes as the initiating vector. The code responsible for generating the data can be found in the Randomness.m file (SignalCoreKit project):

```
+ (NSData *)generateRandomBytes:(int)numberBytes
{
    NSMutableData *_Nullable randomBytes = [NSMutableData
dataWithLength:numberBytes];
    if (!randomBytes) {
        OWSFail(@"Could not allocate buffer for random bytes.");
    }
    int err = 0;
    err = SecRandomCopyBytes(kSecRandomDefault, numberBytes,
[randomBytes mutableBytes]);
    if (err != noErr || randomBytes.length != numberBytes) {
        OWSFail(@"Could not generate random bytes.");
    }
    return [randomBytes copy];
}
```

As we can see, the iOS internal function SecRandomCopyBytes with the selected default generator kSecRandomDefault is responsible for generating pseudorandom data.

2) The generated encryption key and initialization vector in the form of a 48-byte array is saved in the Keychain system data store [16]. The code responsible for saving the data is in the SSKKeychainStorage.m file (SignalServiceKit project):

```
@objc public func set(data: Data, service: String, key: String) 
throws {

SAMKeychain.setAccessibilityType(kSecAttrAccessibleAfterFirstUnlo
ckThisDeviceOnly)

    var error: NSError?
    let result = SAMKeychain.setPasswordData(data, forService: 
service, account: key, error: &error)
    if let error = error {
       throw KeychainStorageError.failure(description: "\(logTag) 
error setting data: \(error)")
    }
  guard result else {
       throw KeychainStorageError.failure(description: "\(logTag) 
could not set data")
   }
}
```

The encryption key and the initialization vector are saved in the Keychain with the following SAMKeychain.setPasswordData attribute values:

- service = “TSKeyChainService”,
- key = “OWSDatabaseCipherKeySpec”.

In this method, we also see a call of the SAMKeychain.setAccessibilityType function with an argument with the value kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly. Thanks to this, the access to the data of the encryption key and the initiating vector will be possible only after the user’s first unlock of the device after each restart of the device (iOS operating system). After the first unlock, the data will remain available until the next device restarts (iOS operating system). Additionally, after restoring the device state, e.g. from a backup copy, these data will not be available.

Until the device’s first unlock (iOS operating system), the encryption key and the initialization vector remain encrypted by the internal Keychain mechanisms. After the first (after reboot) unlocking by the user of the device (iOS operating system), the data are decrypted and made available to every request until the next device restart (iOS operating system).

After the database initialisation and data generation for the SQLCipher mechanism, the Signal application starts to write and read data from the encrypted database. Each time the Signal application is started, the procedure of passing the encryption key and the initialization vector to the SQLCipher mechanisms is carried out. This procedure consists of the following steps:

1) The Signal application receives the generated encryption key and the initialization vector in the form of a 48-byte array from the Keychain system data store. The code responsible for this is in the SSKKeychainStorage.m file (SignalServiceKit project):

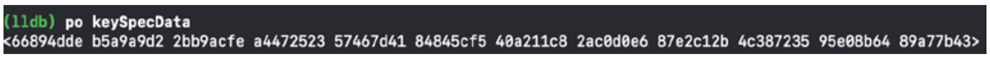
```
@objc public func data(forService service: String, key: String) 
throws -> Data {
    var error: NSError?
    let result = SAMKeychain.passwordData(forService: service,
account: key, error: &error)
    if let error = error {
        throw KeychainStorageError.failure(description:
"\(logTag) error retrieving data: \(error)")
    }
    guard let data = result else {
        throw KeychainStorageError.failure(description:
"\(logTag) could not retrieve data")
    }
    return data
}
```

If the download attempt takes place before the first unlocking of the device, the function call will end in an error - an exception will be thrown.

2) The encryption key and the initialization vector in the form of a 48-byte table are converted to NSString containing the hexadecimal form of the table bytes. Additionally, characters x' and ' are added at the beginning and end of the NSString, respectively. The code responsible for data conversion is in the file YapDatabase.m (project YapDatabase) in the function configureEncryptionForDatabase:

```
NSString *keySpecString = [NSString stringWithFormat:@"x'%@'",
[self hexadecimalStringForData:keySpecData]];
keyData = [keySpecString dataUsingEncoding:NSUTF8StringEncoding];
```

As a result of the conversion, we obtain a 96-character string. In addition, three characters are added, so the total number of characters is 99. Figure 1 presents value of the variable keySpecData from above code. Output is presented in a form of hexadecimal values that represents value of 48 bytes stored in a table, that contains the encryption key and the initialization vector. Variable's value was obtained from the Signal application analyzed in the iPhone Xr simulator.



```
(lldb) po keySpecData
<66894dde b5a9a9d2 2bb9acfe a4472523 57467d41 84845cf5 40a211c8 2ac0d0e6 87e2c12b 4c387235 95e08b64 89a77b43>
```

Fig. 1. Value of the encryption key and initialization vector presented as hexadecimal values, source: own study

3) The resulting 99-character string is passed to the SQLCipher mechanisms by calling the sqlite3_key function from the SQLCipher project. The code responsible for transferring the encryption key and the initialization vector is located in the file YapDatabase.m (project YapDatabase) in the function configureEncryptionForDatabase:

```
int status = sqlite3_key(sqlite, [keyData bytes], (int)[keyData
length]);
if (status != SQLITE_OK)
{
    YDBLogError(@"Error setting SQLCipher key: %d %s", status,
sqlite3_errmsg(sqlite));
    return NO;
}
```

4) If the given data are correct and the database files have not been corrupted, the database is unlocked and the application can read and write data to it.

The Signal application uses the following control commands to initialize the SQLite database and the extension SQLCipher:

```
PRAGMA cipher_plaintext_header_size = 32;
PRAGMA cipher_compatibility = 3
PRAGMA journal_mode = WAL;
PRAGMA auto_vacuum = FULL; VACUUM;
PRAGMA synchronous = NORMAL;
PRAGMA journal_size_limit = 0;
```

Signal uses unencrypted database file headers. The unencrypted header allows the iOS operating system to recognize that the file contains the SQLite database. This is probably due to the WAL mode setting and storage of the database files in the iOS shared container [17]. Thanks to this, the Signal application can use and lock a database file even if it works in the background mode.

In addition to the AES-256 algorithm working in CBC mode, SQLCipher also uses other cryptographic algorithms. The code responsible for selecting them and setting the main SQLCipher operating parameters can be found in the sqlite3.c file in the SQLCipher project:

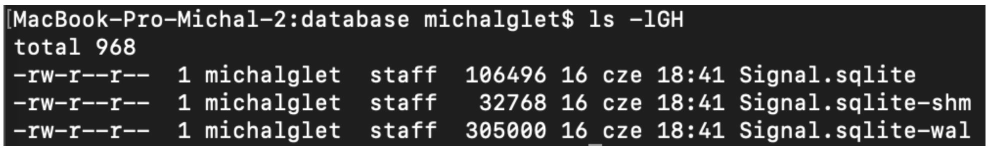
```
rc = sqlcipher_codec_ctx_set_pagesize(ctx, 1024);
if (rc != SQLITE_OK) sqlcipher_codec_ctx_set_error(ctx,
SQLITE_ERROR);
rc = sqlcipher_codec_ctx_set_hmac_algorithm(ctx,
SQLCIPHER_HMAC_SHA1);
if (rc != SQLITE_OK) sqlcipher_codec_ctx_set_error(ctx,
SQLITE_ERROR);
rc = sqlcipher_codec_ctx_set_kdf_algorithm(ctx,
SQLCIPHER_PBKDF2_HMAC_SHA1);
if (rc != SQLITE_OK) sqlcipher_codec_ctx_set_error(ctx,
SQLITE_ERROR);
rc = sqlcipher_codec_ctx_set_kdf_iter(ctx, 64000);
if (rc != SQLITE_OK) sqlcipher_codec_ctx_set_error(ctx,
SQLITE_ERROR);
rc = sqlcipher_codec_ctx_set_use_hmac(ctx, 1);
if (rc != SQLITE_OK) sqlcipher_codec_ctx_set_error(ctx,
SQLITE_ERROR);
```

Thus, in addition to the AES-256-CBC, the SQLCiper in the Signal application can use the HMAC-SHA1 algorithm and the PBKDF2 [18] algorithm based on HMAC-SHA1. The working parameter of the PBKDF2 algorithm has been set at 64,000 iterations, and the page size for the encrypted database is 1024 bytes.

In the shared Signal container, three files related to the database are created:

1. Signal.sqlite,
2. Signal.sqlite-wal,
3. Signal.sqlite-shm.

Figure 2 presents database files created by SQLite with the SQLCipher extension in the Signal application analyzed in the iPhone Xr simulator.



```
[MacBook-Pro-Michal-2:database michalglet$ ls -lGH
total 968
-rw-r--r--  1 michalglet  staff  106496 16 cze 18:41 Signal.sqlite
-rw-r--r--  1 michalglet  staff   32768 16 cze 18:41 Signal.sqlite-shm
-rw-r--r--  1 michalglet  staff  305000 16 cze 18:41 Signal.sqlite-wal
```

Fig. 2. Database files created by the Signal application in the iOS system, source: own study

For these files, Signal assigns the protection level NSFileProtectionCompleteUntilFirstUserAuthentication. The code responsible for this can be found in the OWSPrimaryStorage.m file in the SignalServiceKit project:

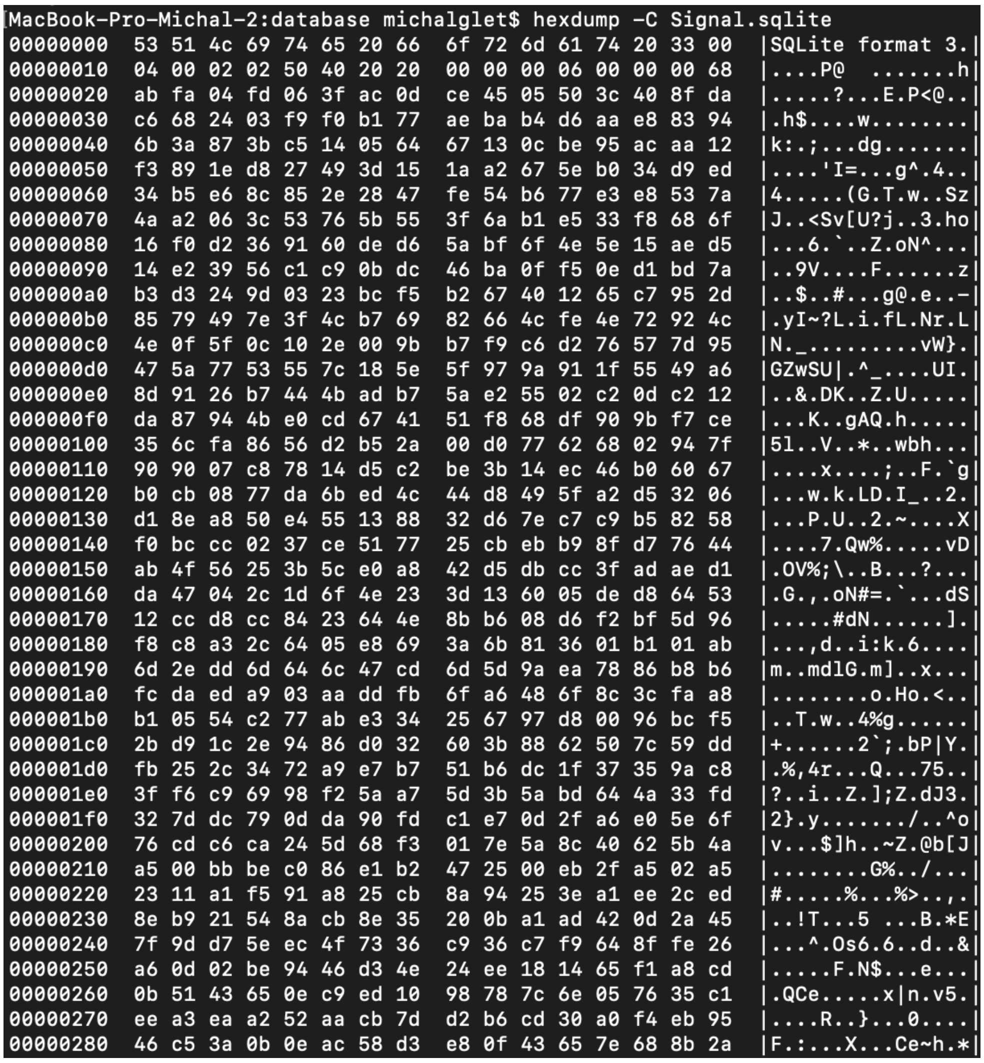
```
+ (void)protectFiles
{
    OWSLogInfo(@"Database file size: %@", [OWSFileSystem
fileSizeOfPath:self.sharedDataDatabaseFilePath]);
    OWSLogInfo(@"\t SHM file size: %@", [OWSFileSystem
fileSizeOfPath:self.sharedDataDatabaseFilePath_SHM]);
    OWSLogInfo(@"\t WAL file size: %@", [OWSFileSystem
fileSizeOfPath:self.sharedDataDatabaseFilePath_WAL]);

    // Protect the entire new database directory.
    [OWSFileSystem
protectFileOrFolderAtPath:self.sharedDataDatabaseDirPath];
}
```

The NSFileProtectionCompleteUntilFirstUserAuthentication protection level causes the indicated file / folder to be stored in the permanent memory in an encrypted form. Access to the file / folder is possible after the first (after reboot) unlock of the device by the user. After the first unlock, the file / folder remains available until the next restart of the device (iOS operating system). The code responsible for setting the file / folder protection level can be found in the OWSFileSystem.m file in the SignalServiceKit project:

```
+ (BOOL)protectFileOrFolderAtPath:(NSString *)path
{
    Return [self protectFileOrFolderAtPath:path
fileProtectionType:NSFileProtectionCompleteUntilFirstUserAuthenti
cation];
}
```

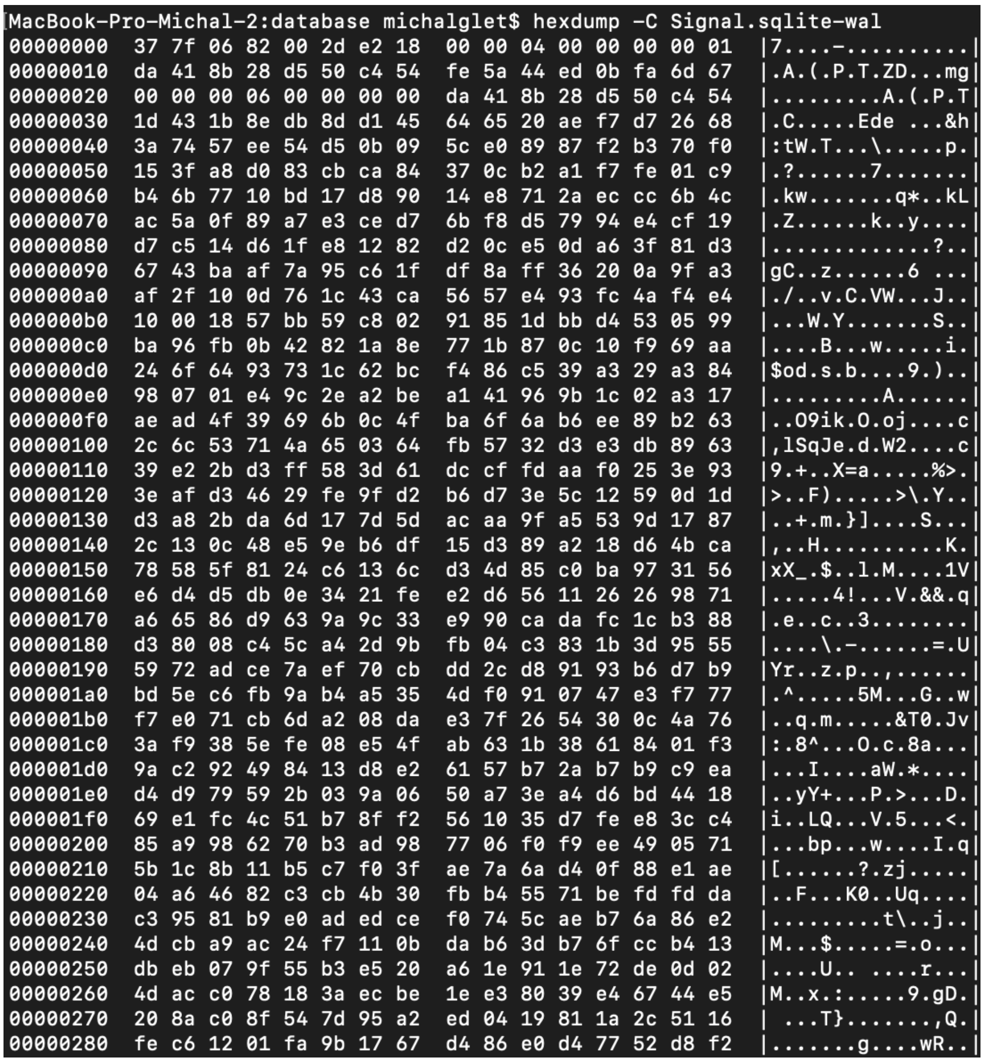
The Signal database is saved in the Signal.sqlite file. Figure 3 presents content of the beginning of Signal.sqlite file. At the left side is presented content offset. Then are presented 32 hexadecimal values (16 bytes of file content). At the right side are presented corresponding ASCII values. We see that the first bytes are not encrypted – we see e.g. text “SQLite format 3”. This is a predicted behavior - the first 32 bytes are written explicitly. The subsequent data look encrypted – we observe heavy and unpredictable noise in hexadecimal and ASCII values. It means that Signal database is stored in an encrypted form.



```
MacBook-Pro-Michal-2:database michalglet$ hexdump -C Signal.sqlite
00000000  53 51 4c 69 74 65 20 66  6f 72 6d 61 74 20 33 00  |SQLite format 3.|
00000010  04 00 02 02 50 40 20 20  00 00 00 06 00 00 00 68  |....P@  .......h|
00000020  ab fa 04 fd 06 3f ac 0d  ce 45 05 50 3c 40 8f da  |.....?...E.P<@..|
00000030  c6 68 24 03 f9 f0 b1 77  ae ba b4 d6 aa e8 83 94  |.h$....w........|
00000040  6b 3a 87 3b c5 14 05 64  67 13 0c be 95 ac aa 12  |k:.;...dg.......|
00000050  f3 89 1e d8 27 49 3d 15  1a a2 67 5e b0 34 d9 ed  |....'I=...g^.4..|
00000060  34 b5 e6 8c 85 2e 28 47  fe 54 b6 77 e3 e8 53 7a  |4.....(G.T.w..Sz|
00000070  4a a2 06 3c 53 76 5b 55  3f 6a b1 e5 33 f8 68 6f  |J..<Sv[U?j..3.ho|
00000080  16 f0 d2 36 91 60 de d6  5a bf 6f 4e 5e 15 ae d5  |...6.`..Z.oN^...|
00000090  14 e2 39 56 c1 c9 0b dc  46 ba 0f f5 0e d1 bd 7a  |..9V....F......z|
000000a0  b3 d3 24 9d 03 23 bc f5  b2 67 40 12 65 c7 95 2d  |..$..#...g@.e..-|
000000b0  85 79 49 7e 3f 4c b7 69  82 66 4c fe 4e 72 92 4c  |.yI~?L.i.fL.Nr.L|
000000c0  4e 0f 5f 0c 10 2e 00 9b  b7 f9 c6 d2 76 57 7d 95  |N._.........vW}.|
000000d0  47 5a 77 53 55 7c 18 5e  5f 97 9a 91 1f 55 49 a6  |GZwSU|.^_....UI.|
000000e0  8d 91 26 b7 44 4b ad b7  5a e2 55 02 c2 0d c2 12  |..&.DK..Z.U.....|
000000f0  da 87 94 4b e0 cd 67 41  51 f8 68 df 90 9b f7 ce  |...K..gAQ.h.....|
00000100  35 6c fa 86 56 d2 b5 2a  00 d0 77 62 68 02 94 7f  |5l..V..*..wbh...|
00000110  90 90 07 c8 78 14 d5 c2  be 3b 14 ec 46 b0 60 67  |....x....;..F.`g|
00000120  b0 cb 08 77 da 6b ed 4c  44 d8 49 5f a2 d5 32 06  |...w.k.LD.I_..2.|
00000130  d1 8e a8 50 e4 55 13 88  32 d6 7e c7 c9 b5 82 58  |...P.U..2.~....X|
00000140  f0 bc cc 02 37 ce 51 77  25 cb eb b9 8f d7 76 44  |....7.Qw%.....vD|
00000150  ab 4f 56 25 3b 5c e0 a8  42 d5 db cc 3f ad ae d1  |.OV%;\..B...?...|
00000160  da 47 04 2c 1d 6f 4e 23  3d 13 60 05 de d8 64 53  |.G.,.oN#=.`...dS|
00000170  12 cc d8 cc 84 23 64 4e  8b b6 08 d6 f2 bf 5d 96  |.....#dN......].|
00000180  f8 c8 a3 2c 64 05 e8 69  3a 6b 81 36 01 b1 01 ab  |...,d..i:k.6....|
00000190  6d 2e dd 6d 64 6c 47 cd  6d 5d 9a ea 78 86 b8 b6  |m..mdlG.m]..x...|
000001a0  fc da ed a9 03 aa dd fb  6f a6 48 6f 8c 3c fa a8  |........o.Ho.<..|
000001b0  b1 05 54 c2 77 ab e3 34  25 67 97 d8 00 96 bc f5  |..T.w..4%g......|
000001c0  2b d9 1c 2e 94 86 d0 32  60 3b 88 62 50 7c 59 dd  |+......2`;.bP|Y.|
000001d0  fb 25 2c 34 72 a9 e7 b7  51 b6 dc 1f 37 35 9a c8  |.%,4r...Q...75..|
000001e0  3f f6 c9 69 98 f2 5a a7  5d 3b 5a bd 64 4a 33 fd  |?..i..Z.];Z.dJ3.|
000001f0  32 7d dc 79 0d da 90 fd  c1 e7 0d 2f a6 e0 5e 6f  |2}.y......./..^o|
00000200  76 cd c6 ca 24 5d 68 f3  01 7e 5a 8c 40 62 5b 4a  |v...$]h..~Z.@b[J|
00000210  a5 00 bb be c0 86 e1 b2  47 25 00 eb 2f a5 02 a5  |........G%../...|
00000220  23 11 a1 f5 91 a8 25 cb  8a 94 25 3e a1 ee 2c ed  |#.....%...%>..,.|
00000230  8e b9 21 54 8a cb 8e 35  20 0b a1 ad 42 0d 2a 45  |..!T...5 ...B.*E|
00000240  7f 9d d7 5e ec 4f 73 36  c9 36 c7 f9 64 8f fe 26  |...^.Os6.6..d..&|
00000250  a6 0d 02 be 94 46 d3 4e  24 ee 18 14 65 f1 a8 cd  |.....F.N$...e...|
00000260  0b 51 43 65 0e c9 ed 10  98 78 7c 6e 05 76 35 c1  |.QCe.....x|n.v5.|
00000270  ee a3 ea a2 52 aa cb 7d  d2 b6 cd 30 a0 f4 eb 95  |....R..}...0....|
00000280  46 c5 3a 0b 0e ac 58 d3  e8 0f 43 65 7e 68 8b 2a  |F.:...X...Ce~h.*|
```

Fig. 3. First 640 bytes (in hexadecimal and ASCII values) of the Signal.sqlite file, source: own study

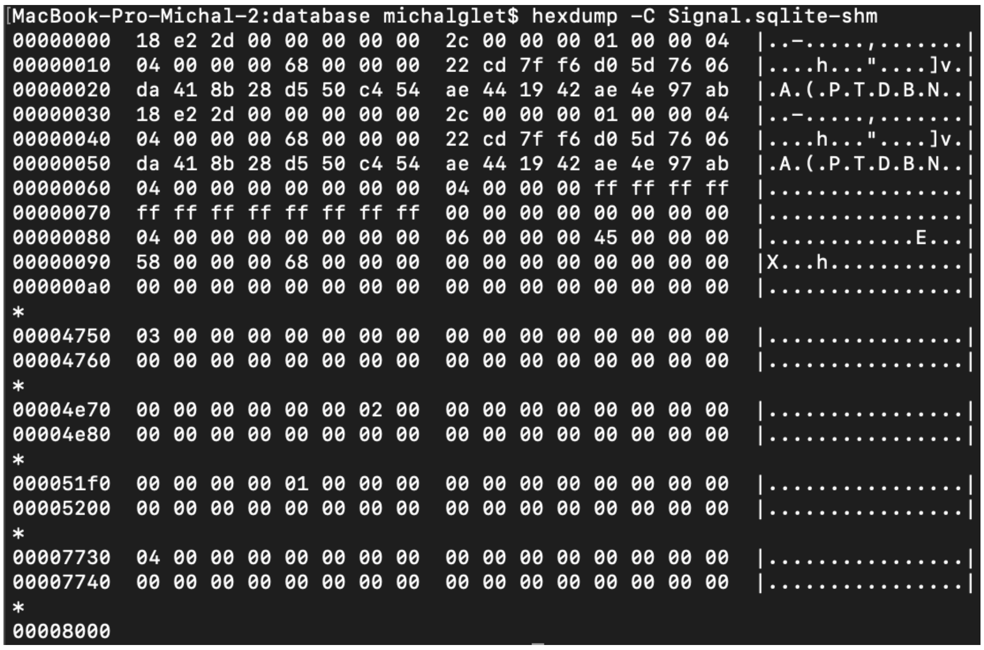
Signal.sqlite-wal is an auxiliary file used by the SQLite working in the WAL mode. This file is used to perform atomic commit and rollback operations. When using the SQLite API correctly, this file is deleted when the last connection to the database is closed. However, if this connection is not closed properly, or if there is aby other problem with the Signal application, the file will stay in the iOS operating system's file system. Figure 4 presents content of the beginning of Signal.sqlite-wal file. At the left side is presented content offset. Then are presented 32 hexadecimal values (16 bytes of file content). At the right side are presented corresponding ASCII values. Initial analysis shows that the first 40 bytes of this file are not likely to be encrypted (little noise, many repetitive sequences of values, e.g. 00). The further data look encrypted – we observe heavy and unpredictable noise in hexadecimal and ASCII values. It means that content of Signal.sqlite-wal file is encrypted. This is very important because the file stores information related to database commits and rollbacks, hence it could, if it were not encrypted, be a potential source of data leakage.



```
[MacBook-Pro-Michal-2:database michalglet$ hexdump -C Signal.sqlite-wal
00000000  37 7f 06 82 00 2d e2 18  00 00 04 00 00 00 00 01  |7....-..........|
00000010  da 41 8b 28 d5 50 c4 54  fe 5a 44 ed 0b fa 6d 67  |.A.(.P.T.ZD...mg|
00000020  00 00 00 06 00 00 00 00  da 41 8b 28 d5 50 c4 54  |.........A.(.P.T|
00000030  1d 43 1b 8e db 8d d1 45  64 65 20 ae f7 d7 26 68  |.C.....Ede ...&h|
00000040  3a 74 57 ee 54 d5 0b 09  5c e0 89 87 f2 b3 70 f0  |:tW.T...\.....p.|
00000050  15 3f a8 d0 83 cb ca 84  37 0c b2 a1 f7 fe 01 c9  |.?......7.......|
00000060  b4 6b 77 10 bd 17 d8 90  14 e8 71 2a ec cc 6b 4c  |.kw.......q*..kL|
00000070  ac 5a 0f 89 a7 e3 ce d7  6b f8 d5 79 94 e4 cf 19  |.Z......k..y....|
00000080  d7 c5 14 d6 1f e8 12 82  d2 0c e5 0d a6 3f 81 d3  |.............?..|
00000090  67 43 ba af 7a 95 c6 1f  df 8a ff 36 20 0a 9f a3  |gC..z......6 ...|
000000a0  af 2f 10 0d 76 1c 43 ca  56 57 e4 93 fc 4a f4 e4  |./..v.C.VW...J..|
000000b0  10 00 18 57 bb 59 c8 02  91 85 1d bb d4 53 05 99  |...W.Y.......S..|
000000c0  ba 96 fb 0b 42 82 1a 8e  77 1b 87 0c 10 f9 69 aa  |....B...w.....i.|
000000d0  24 6f 64 93 73 1c 62 bc  f4 86 c5 39 a3 29 a3 84  |$od.s.b....9.)..|
000000e0  98 07 01 e4 9c 2e a2 be  a1 41 96 9b 1c 02 a3 17  |.........A......|
000000f0  ae ad 4f 39 69 6b 0c 4f  ba 6f 6a b6 ee 89 b2 63  |..O9ik.O.oj....c|
00000100  2c 6c 53 71 4a 65 03 64  fb 57 32 d3 e3 db 89 63  |,lSqJe.d.W2....c|
00000110  39 e2 2b d3 ff 58 3d 61  dc cf fd aa f0 25 3e 93  |9.+..X=a.....%>.|
00000120  3e af d3 46 29 fe 9f d2  b6 d7 3e 5c 12 59 0d 1d  |>..F).....>\.Y..|
00000130  d3 a8 2b da 6d 17 7d 5d  ac aa 9f a5 53 9d 17 87  |..+.m.}]....S...|
00000140  2c 13 0c 48 e5 9e b6 df  15 d3 89 a2 18 d6 4b ca  |,..H..........K.|
00000150  78 58 5f 81 24 c6 13 6c  d3 4d 85 c0 ba 97 31 56  |xX_.$..l.M....1V|
00000160  e6 d4 d5 db 0e 34 21 fe  e2 d6 56 11 26 26 98 71  |.....4!...V.&&.q|
00000170  a6 65 86 d9 63 9a 9c 33  e9 90 ca da fc 1c b3 88  |.e..c..3........|
00000180  d3 80 08 c4 5c a4 2d 9b  fb 04 c3 83 1b 3d 95 55  |....\.-......=.U|
00000190  59 72 ad ce 7a ef 70 cb  dd 2c d8 91 93 b6 d7 b9  |Yr..z.p..,......|
000001a0  bd 5e c6 fb 9a b4 a5 35  4d f0 91 07 47 e3 f7 77  |.^.....5M...G..w|
000001b0  f7 e0 71 cb 6d a2 08 da  e3 7f 26 54 30 0c 4a 76  |..q.m.....&T0.Jv|
000001c0  3a f9 38 5e fe 08 e5 4f  ab 63 1b 38 61 84 01 f3  |:.8^...O.c.8a...|
000001d0  9a c2 92 49 84 13 d8 e2  61 57 b7 2a b7 b9 c9 ea  |...I....aW.*....|
000001e0  d4 d9 79 59 2b 03 9a 06  50 a7 3e a4 d6 bd 44 18  |..yY+...P.>...D.|
000001f0  69 e1 fc 4c 51 b7 8f f2  56 10 35 d7 fe e8 3c c4  |i..LQ...V.5...<.|
00000200  85 a9 98 62 70 b3 ad 98  77 06 f0 f9 ee 49 05 71  |...bp...w....I.q|
00000210  5b 1c 8b 11 b5 c7 f0 3f  ae 7a 6a d4 0f 88 e1 ae  |[......?.zj.....|
00000220  04 a6 46 82 c3 cb 4b 30  fb b4 55 71 be fd fd da  |..F...K0..Uq....|
00000230  c3 95 81 b9 e0 ad ed ce  f0 74 5c ae b7 6a 86 e2  |.........t\..j..|
00000240  4d cb a9 ac 24 f7 11 0b  da b6 3d b7 6f cc b4 13  |M...$.....=.o...|
00000250  db eb 07 9f 55 b3 e5 20  a6 1e 91 1e 72 de 0d 02  |....U.. ....r...|
00000260  4d ac c0 78 18 3a ec be  1e e3 80 39 e4 67 44 e5  |M..x.:.....9.gD.|
00000270  20 8a c0 8f 54 7d 95 a2  ed 04 19 81 1a 2c 51 16  | ...T}.......,Q.|
00000280  fe c6 12 01 fa 9b 17 67  d4 86 e0 d4 77 52 d8 f2  |.......g....wR..|
```

Fig. 4. First 640 bytes (in hexadecimal and ASCII values) of the Signal.sqlite-wal file, source: own study

Signal.sqlite-shm - this is another file used by the SQLite working in the WAL mode. This file is used as a shared memory for indexes to the WAL file for all connections to the Signal database. Figure 5 presents content of the Signal.sqlite-shm file. At the left side is presented content offset. Then are presented 32 hexadecimal values (16 bytes of file content). At the right side are presented corresponding ASCII values. Initial analysis shows that content of this file is not likely to be encrypted. From offset 0x80 to the offset 0x8000 (end of file) almost every byte has value 0.



```
[MacBook-Pro-Michal-2:database michalglet$ hexdump -C Signal.sqlite-shm
00000000  18 e2 2d 00 00 00 00 00  2c 00 00 00 01 00 00 04  |..-.....,.......|
00000010  04 00 00 00 68 00 00 00  22 cd 7f f6 d0 5d 76 06  |....h..."....]v.|
00000020  da 41 8b 28 d5 50 c4 54  ae 44 19 42 ae 4e 97 ab  |.A.(.P.T.D.B.N..|
00000030  18 e2 2d 00 00 00 00 00  2c 00 00 00 01 00 00 04  |..-.....,.......|
00000040  04 00 00 00 68 00 00 00  22 cd 7f f6 d0 5d 76 06  |....h..."....]v.|
00000050  da 41 8b 28 d5 50 c4 54  ae 44 19 42 ae 4e 97 ab  |.A.(.P.T.D.B.N..|
00000060  04 00 00 00 00 00 00 00  04 00 00 00 ff ff ff ff  |................|
00000070  ff ff ff ff ff ff ff ff  00 00 00 00 00 00 00 00  |................|
00000080  04 00 00 00 00 00 00 00  06 00 00 00 45 00 00 00  |............E...|
00000090  58 00 00 00 68 00 00 00  00 00 00 00 00 00 00 00  |X...h...........|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00004750  03 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00004760  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00004e70  00 00 00 00 00 00 02 00  00 00 00 00 00 00 00 00  |................|
00004e80  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000051f0  00 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00005200  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00007730  04 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00007740  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00008000
```

Fig. 5. Content (in hexadecimal and ASCII values) of the Signal.sqlite-shm file, source: own study

The Signal's application data are stored in the SQLite database. The Signal application manufacturer has used two mechanisms to secure access to this data. The first one is the data encryption mechanism based on the extension of the SQLite database called SQLCipher. The second one is the mechanism for encrypting database files stored in the device's permanent memory and denying access to them until the first unlock of the device (iOS operating system) by the user.

The encryption key and the initialization vector for the first mechanism (SQLCipher) are stored in the system storage of the iOS – the Keychain system. Access to these data is possible from the moment the user first unlocks the device (iOS operating system) until the next device restart (restarting the iOS operating system). Thus, does this mechanism ensure a sufficient level of security? In the next section I show some possible vectors of attack.

4. VECTORS OF ATTACK

Previous section showed that protection of database encryption key and initialization vector is based on the internal security mechanisms of the iOS operating system. The analyzed implementation uses native iOS Keychain mechanism to store and protect highly sensitive data used for database encryption. It means that our messages, contacts, info about received and made calls stored by Signal is as secure as secure is iOS operating system. Because of serious operating system vulnerabilities that are currently known (e.g. [19, 20, 21, 22, 23]) and those that will appear, applications that are said to be secure should not rely only on native iOS security mechanisms, but should provide its own security mechanisms that will resist hacking into iOS system or bypassing its security mechanisms. This section presents possible vectors of attack and describes requirements under which attacks can succeed. Every attack has also assigned simple difficulty level that show how difficult it is to perform it and vulnerability level to show how big impact it can produce.

Main vulnerability discovered in data storage mechanism is a possibility of recovery of the encryption key and initialization vector used by Signal to encrypt its database. Possible vectors of attack mainly depend on the unlock state of the attacking iOS device. Two situations must be considered. In the first one the attacker deals with device that was unlocked be a user after iOS operating system restart (encryption key and vector data can be read from the Keychain). In the second one the attacker deals with device that was not unlocked by the user after iOS operating system restart (encryption key and vector data are not available in the Keychain).

In the first scenario user must enter passcode after iOS device restart. After the first unlocking of the device, the encryption key and initialization vector data can be read from the Keychain using API provided by operating system. Therefore, the data may be accessed by for example:

- Spying software.
- Malware based on publicly known and unknown vulnerabilities in the iOS operating system.
- Any application working on iOS with installed “Jailbreak” software.
- An attacker who knows the passcode or is able to force the user to unlock the phone (e.g. using biometrics).
- Government agencies and state authorities that have access to tools and software that break the passcode of the iOS operating system - and it is mainly against them that Signal is supposed to protect the data in the database.

In the second scenario device has not been unlocked by the user after rebooting. In this case access to the encryption key and initiating vector can be obtained by:

- An attacker who knows the passcode or is able to force the user to unlock the phone.
- Government agencies and state authorities that have access to tools and software that break the passcode of the iOS operating system - and it is mainly against them that Signal is supposed to protect the data in the database.

Similar conclusions can be drawn for the second security mechanism, which also is based on the user's first unlocking the device (iOS operating system) - it is not a mechanism that fully protects database files.

Table 1. Vulnerabilities summary

Vulnerability	**Device unlock state**	**Jailbreak installed**	**Difficulty level**	**Security impact**
Recovery of database encryption key and initialization vector	Device unlocked after restart – user has entered passcode	No	Moderate	High
		Yes	Easy	
	Device locked after restart – user has not entered passcode	No	High	
		Yes	Easy	
Stealing database files	Device unlocked after restart – user has entered passcode	No	Moderate	Medium
		Yes	Easy	
	Device locked after restart – user has not entered passcode	No	High	
		Yes	Easy	

Jailbreak installed means that there are known vulnerabilities in the version of iOS operating system installed on the device that being attacked and that kernel patches for privileges escalation have been installed. Kernel patches have been installed by user or remotely with use of software such as Pegasus.

Difficulty level classification:

- Easy – a vulnerability is easily turned into an attack, an attacker must have IT and developer skills, there is no need to have other resources (money, computing power, etc.).
- Moderate – a vulnerability can be turned into an attack, but it requires professional knowledge, high IT and developer skills and other resources like money, computing power and time for researches.

- High – it is likely that vulnerability can be turned into an attack, but it requires high professional knowledge, high IT and developer skills and high amount resources like money, computing power and time for researches. This difficulty level means that there is a need to use many resources for attack to succeed. In many cases resources spent are much higher than possible gains, so in this cases attack will mainly be conducted by government agencies.

Security impact classification:

- Medium – means that after performing a successful attack sensitive data will probably remain private and confidence for long period of time. In Signal's case it means that data will remain secure as long as an attacker will not find the encryption key, e.g. in brute-force attack (checking every possible key value).
- High – means that after performing a successful attack sensitive data cannot be considered private and confidence anymore.

5. RECOMMENDATIONS

Signal is storing its data in encrypted database. The encryption key and initialization vector are stored in native iOS Keychain mechanism. Database files are secured with native iOS mechanisms. The access to encryption data and database files are given after first device unlocking by the user. Previous section depicts some possible key recover attacks that are mainly related with security of the iOS operating system. This section gives simple recommendation that would make described attacks inefficient.

Firstly, Signal and other applications that has to ensure privacy and confidentiality of stored data cannot rely only on security mechanisms provided by operating systems. Each application should have its own security mechanisms that suits application needs. These mechanisms can replace or cooperate with security mechanisms provided by iOS. Such a custom mechanism can consist of:

- Procedure that allows user to select desired security level. User can choose lower security that uses only iOS security mechanisms or higer security that uses custom mechanisms at the expense of worse user experience (e.g. user must enter password when opening application).
- Procedure of secure encapsulation the encryption key before storing it in iOS operating system. It can for example consist of Key Encryption Key algorithms [24], Key Derivation Functions [25] or Password-based Encryption algorithms [18].
- Procedure that checks if data used for encapsulating the encryption key is good enough in security terms (e.g. if user password is long enough to make brute-force attack infeasible).

- If an application requires user to enter a password it must provide a procedure that defends application against dictionary or brute-force attacks.
- Alternatively, procedure of retrieving the database encryption key can use zero-knowledge proofs schemas [26] and secret sharing algorithms [27] to share some parts of encryption key with server.

All these procedures are quite simple to introduce in an existing application. Developers of secure applications should think seriously about custom security mechanisms similar to given procedures. This is one of ways to make data stored in their applications much more secure. One could say that there aren't any known vulnerabilities in the newest version of iOS (13.1.3 in time of writing this paper) and that database security provided by Signal is strong enough. However, it is not quite true. As a history of iOS shows, it is very highly probable that soon some vulnerabilities will become available and the difficulty level of an attack will drop to an easy level. What is more, mobile device can be stolen or confiscated, and an attacker can wait some time till such vulnerabilities become available. Then he can compromise iOS security, recover Signal's database encryption key and get access to user's private data.

6. CONCLUSION

The Signal analysis carried out for iOS leads to one simple conclusion - database files, encryption key, and initiation vector for Signal 2.39.0 are as secure as the iOS operating system is secure. As we all know, there is hardware and software, like Pegasus from NSO Group, that probably can break the security mechanisms even of the newest iOS version (13.1.3) and steal data from the iPhone storage. Then, how to improve the security of data stored in the Signal application in the iOS operating system? The simplest and fastest solution seems to be the introduction of its own mechanism for protection of the encryption key, which will be based for example on the password entered by the user when the Signal application is launching. This mechanism may, for example, use cryptographic algorithms with a key type Key Encryption Key - the key stored in the Keychain (the current mechanism) will be additionally encrypted with a key created on the basis of a password entered by the user. When reading data from the Keychain, they will have to be decrypted with a key created on the basis of the same password entered by the user. Further processing would not change - the decrypted key would be used to encrypt/decrypt data using the SQLCipher mechanisms.

REFERENCES

[1] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, D. Stebila. A Formal Security Analysis of the Signal Messaging Protocol. Cryptology ePrint Archive. International Association for Cryptologic Research (IACR), 2016.

[2] N. Kobeissi, K. Bhsrgavan, B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. European Symposium on Security and Privacy (EuroS&P) , 2017.

[3] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. Public Key Cryptography (PKC), 2006.

[4] J. Daemen, V. Rijmen. The Design of Rijndael. AES - The Advanced, Encryption Standard. Springer, 2002.

[5] M. Dworkin. Recommendation for Block Cipher Modes of Operation. National Institute of Standards and Technology (NIST) Special Publication 800-38A, 2001.

[6] S. Turner, L. Chen. Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms, RFC 6151, 2011.

[7] C. Rottermanner, P. Kieseberg, M. Huber, M. Schmiedecker, S. Schrittwieser. Privacy and data protection in smartphone messengers. Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services. ACM. 2015.

[8] L. FENG, L. KE-SHENG, CH. CHANG, Y. WANG. Research on the Technology of iOS Jailbreak. Sixth International Conference on Instrumentation & Measurement, Computer, Communication and Control (IMCCC), 2016.

[9] Y.-T. Chang, K.-C. Teng, Y.-C. Tso, and S.-J. Wang. Jailbroken iphone forensics for the investigations and controversy to digital evidence. Journal of Computers, vol. 26, no. 2, 2015.

[10] Jailbreaking iOS 11 And All Versions Of iOS 10. 30 March 2018. [Online]. Available: https://articles.forensicfocus.com/2018/03/30/jailbreaking-ios-11-and-all-versions-of-ios-10/, access: 20.10.2019.

[11] C. J. D’Orazio and K. K. R. Choo. Circumventing iOS security mechanisms for APT forensic investigations: A security taxonomy for cloud apps. Future Generation Computer Systems, 2016.

[12] M. Epifani, P. Stirparo. Learning iOS Forensics. Packt Publishing, 2015.

[13] Pegasus, https://en.wikipedia.org/wiki/Pegasus_(spyware), access 15.06.2019.

[14] iOS Jailbreak, https://en.wikipedia.org/wiki/IOS_jailbreaking, access 20.10.2019.

[15] J. Zdziarski. Hacking and securing iOS applications: stealing data, hijacking software, and how to prevent it. O'Reilly Media, 2012.

[16] https://developer.apple.com/documentation/security/keychain_services, access 15.06.2019.

[17]https://developer.apple.com/library/archive/documentation/General/Conceptual/ExtensibilityPG/ExtensionScenarios.html, access 15.06.2019.

[18] B. Kaliski, PKCS #5: Password-based cryptography specification, version 2.0, IETF Network Working Group, RFC 2898, 2000.

[19] CVE-2018-4465, A memory corruption issue was addressed with improved memory handling. This issue affected versions prior to iOS 12.1.1, macOS Mojave 10.14.2, tvOS 12.1.1, watchOS 5.1.2., https://www.cvedetails.com/cve/CVE-2018-4465/, access 20.10.2019.

[20] CVE-2018-4461, A memory corrupt, ion issue was addressed with improved input validation. This issue affected versions prior to iOS 12.1.1, macOS Mojave 10.14.2, tvOS 12.1.1, watchOS 5.1.2., https://www.cvedetails.com/cve/CVE-2018-4461/, access 20.10.2019.

[21] CVE-2018-4447, A memory corruption issue was addressed with improved state management. This issue affected versions prior to iOS 12.1.1, macOS Mojave 10.14.2, tvOS 12.1.1, watchOS 5.1.2., https://www.cvedetails.com/cve/CVE-2018-4447/, access 20.10.2019

[22] New ‘unpatchable’ iOS exploit could lead to permanent jailbreak for iPhone 4s to iPhone X. https://9to5mac.com/2019/09/27/ios-unpatchable-ios-exploit-jailbreak-iphone-x/, access 20.10.2019.

[23] iOS 12.4 jailbreak publicly released after Apple mistakenly unpatches vulnerability. https://9to5mac.com/2019/08/19/ios-12-4-jailbreak-released/, access 20.10.2019.

[24] M. H. Etzel, D. W. Faucher, D. N. Heer, D. P. Maher, R. J. Rance. Data encryption key management system. Patent number US6577734B1, 1995.

[25] L. Chen. Recommendation for Key Derivation Using Pseudorandom Functions (Revised). National Institute of Standards and Technology (NIST) Special Publication 800-108, 2009.

[26] Zero-knowledge proof, https://en.wikipedia.org/wiki/Zero-knowledge_proof, access 20.10.2019.

[27] Secret sharing, https://en.wikipedia.org/wiki/Secret_sharing, access 20.10.2019.

[26] Signal iOS source codes, https://github.com/signalapp/Signal-iOS, access 15.06.2019.

Information about the author:

Michał Glet – is an Assistant at the Faculty of Cybernetics at the Military University of Technology in Warsaw, Poland. His research interests include cybersecurity, cryptography, cryptoanalysis, malware analysis, software reverse engineering, and software development.

Manuscript received on 10 September 2019