TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Informatics

Chair of Software Engineering

Security Analysis of Instant Messenger TorChat

Master's Thesis

Student: Rain Viigipuu
Student code: 072125

Supervisor: Alexander Norta, PhD

External Supervisor: Arnis Paršovs, MSc

TALLINN 2015

Abstract

TorChat is a peer-to-peer instant messenger built on top of the Tor network that not only provides authentication and end-to-end encryption, but also allows the communication parties to stay anonymous. In addition, it prevents third parties from even learning that communication is taking place.

The aim of this thesis is to document the protocol used by TorChat and to analyze the security of TorChat and its reference implementation. The work shows that although the design of TorChat is sound, its implementation has several flaws, which make TorChat users vulnerable to impersonation, communication confirmation and denial-of-service attacks.

Contents

1 Introduction

In today's world, secure communication over the Internet is a challenging task. Recently it became known that the architecture of the most popular instant messenger Skype has been redesigned to enable communication surveillance on its users [1]. The U.S. government is using secret warrantless requests to obtain personal information stored by service providers [2] and is obtaining encryption keys from service providers [3] by using legal means. Another aspect recently realized by society is that privacy cannot be achieved just by securing communication content. The communication metadata that shows the parties involved in the communication and their location might be even more sensitive as information than the content of communication and thus it must be protected as well [4].

Therefore, in order to achieve communication privacy, we need a solution that would provide end-to-end encryption between communication parties and make collecting metadata very difficult.

The TorChat [5] instant messenger is a technology built on top of Tor [6] that not only provides end-to-end encryption between TorChat clients, but also hides the location of TorChat users and prevents third parties from determining the parties with whom a TorChat client is communicating.

Allegedly, TorChat has been used to provide secure communication between a doctor and his patients [7], and, as recently revealed by court documents, to protect the location of the mastermind behind the notorious marketplace Silk Road [8].

It is not easy to estimate the size of the current TorChat user base. A study that was carried out on February 4, 2013, made use of the Tor hidden service address harvesting method described in [9], found that among the harvested 39,824 unique Tor hidden service onion addresses, 385 were TorChat clients [10].

The security guarantees provided by TorChat have not been analyzed before and the only source of a description for the TorChat protocol has been the source code of TorChat reference implementation.

The goal of this paper is to describe the protocol used by TorChat, provide security analysis of the TorChat protocol and report on the vulnerabilities found in the audit of the latest TorChat original Python implementation version 0.9.9.553.

Note that there are also other TorChat implementations available. TorChat2 [11], which is a Pidgin IM plugin, jTorChat [12] – TorChat implementation in Java, and Ruby-torchat [13] – TorChat implementation in Ruby. However, this paper focuses only on the TorChat original Python implementation, which is available for MS Windows and is included in most Linux distributions. Therefore, it is believed to have the largest user base. It also serves as a reference implementation of the TorChat protocol, which means that all other implementations have been developed based on the Python implementation.

The rest of the thesis is organized as follows. The next section introduces the Tor anonymity network and Tor hidden services. Section 3 gives an overview of TorChat and its features from the user's perspective. Section 4 documents the protocol as used by TorChat. Section 5 defines the methodology for analyzing the security of instant messengers. Section 6 provides a security analysis of the TorChat instant messenger and its original Python implementation by answering the questions defined in Section 5. Finally, Section 7 provides a summary of findings and Section 8 concludes the work.

2 Tor and Hidden Services

This section provides a short description of Tor and the Tor hidden services feature. The reader is welcome to skip over this section if he is familiar with Tor. The remaining part of the thesis assumes that the reader knows how Tor and its location-hidden services work.

Tor is a software program that allows people to keep their Internet activity private and anonymous. It also provides the tools and platform for developers who can create new applications with built-in encryption, privacy and anonymity features. Those applications not only allow users to use Internet services anonymously, but also allow them to provide different kind of Internet services such as websites, instant messaging services or other services without compromising their privacy.

Privacy on the Internet can be achieved partially by using Internet services over encrypted connections. In this way, only the data moving over the network is private. However, the metadata that reveals the source and the destination of the data and other properties that are not encrypted can still be monitored and analyzed. The Tor network solves that problem by encrypting not only the data moving over the network, but the metadata as well. This makes it impossible for interested parties (governments, intelligence agencies, companies, law enforcement, criminals, etc.) to analyze the network traffic in order to determine the user's behaviour and interests.

To achieve that, the Tor client creates a private secure pathway called the Tor circuit through the Tor network to the destination. The circuit is made out of three Tor nodes. The circuit is extended one hop at a time and each node only knows where the data came from (the previous node) and where it should go (the next node). No single node knows all the nodes in the circuit. The client negotiates a separate set of encryption keys for each hop in the circuit to make sure that the connections could not be traced.

When the user wants to use a regular Internet service anonymously, the connection has to exit from the Tor network at some point. This is done through an exit node,

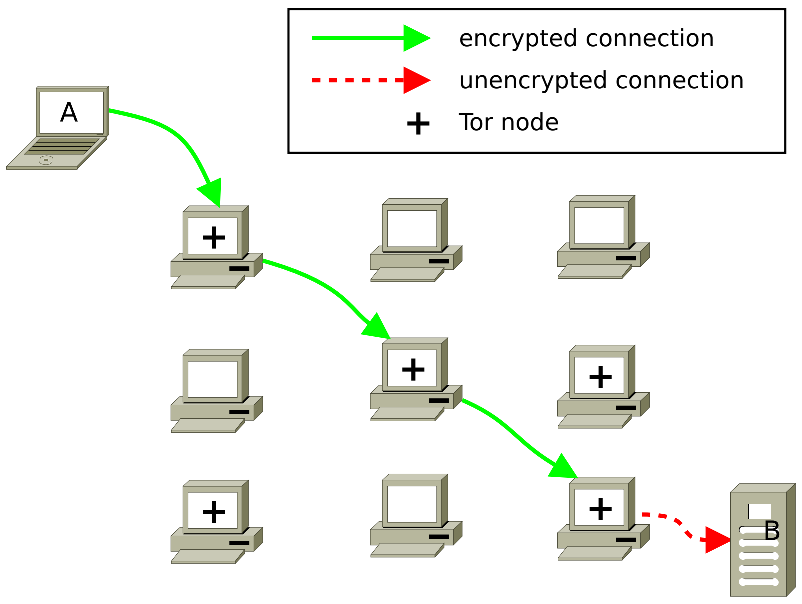


Figure 1: Network traffic moving through the Tor network from A to B.

which is the last node of the Tor circuit. From that node forward the connection is encrypted only when the destination service supports encrypted communication. The way that traffic moves through the Tor network is illustrated in Figure 1.

2.1 Hidden Services

Tor location hidden services is a feature of Tor that enables users to offer various TCP services without revealing their location. Other Tor users can connect to those services without knowing the service provider's network identity and without revealing their own. To achieve this goal, Tor's rendezvous protocol is used [14].

When a Tor client is configured to provide a hidden service and is launched for the first time, it generates a 1024-bit RSA key pair. From the public key it derives a hidden service address that is 16 characters long; this is also called an onion address.

In order to be reachable to other Tor users, a hidden service has to advertise its existence in the Tor network. To do this the hidden service randomly picks some

Tor relays, builds circuits to them and publishes its hidden service descriptor in the public directory. The descriptor contains the public key of the hidden service and a list of introduction points where the service can be reached. Since a full Tor circuit is established between the hidden service and the introduction point, the IP address of the hidden service is hidden from the introduction point.

If someone wants to connect to the hidden service, he needs to know the onion address of the service. After the client has obtained the onion address of the service (by some out-of-band means), he can download the hidden service descriptor from the public directory. From the descriptor the client can extract the public key of the hidden service and the list of introduction points through which the service can be reached. The client chooses one random Tor relay to be the rendezvous point for itself and the service, builds a Tor circuit to it and sends to it a one-time secret (rendezvous cookie).

After the rendezvous point is set up the client puts together an introduction message containing the address of the rendezvous point and the one-time secret. The client connects to one of the hidden service's introduction points via the Tor circuit and sends the introduction message requesting it to be sent to the hidden service.

The hidden service receives the introduction message, decrypts it with its private key and finds information about the rendezvous point and the one-time secret. The hidden service creates a circuit to the rendezvous point and sends a message with the one-time secret.

The rendezvous point notifies the client that a connection has been established with the hidden service. Now, the client and the hidden service can use their circuits to the rendezvous point to communicate with each other. The rendezvous point simply relays the messages from the client to the hidden service and the other way around. Since the messages are end-to-end encrypted, the rendezvous point cannot learn the content of the messages.

2.1.1 Hidden service address

In the Tor network, hidden services are identified and accessed by their so-called onion addresses. An onion address is a 16-character hash with the suffix ".onion". The 16-character hash is computed as follows. The SHA-1 hash is calculated from the hidden service's DER-encoded ASN.1 RSA public key (as specified in PKCS.1) [15, Section 0.3]. The hash will be 160 bits long. The first half of the hash (80 bits long) is encoded to Base32, so it will only contain the digits 2-7, the letters a-z, and will be exactly 16 characters long.

The reason why onion addresses are hashes and not human meaningful names is described in Zooko's Distnames essay. The essay argues that a system giving out names in a network protocol has three desirable properties [16]: "human meaningful" – i.e. highly memorable for human beings, "decentralized" – i.e. there is no centralized authority to hold the meaning of a name, and "secure" – i.e. there is only one unique entity to which the name maps.

The essay states that a name system cannot have all three properties at the same time, but only two: decentralized and human meaningful (for example nicknames that people choose for themselves), secure and human meaningful (for example the current DNS system) or secure and decentralized (OpenPGP has these properties).

Onion addresses have been chosen to be secure and decentralized and therefore have the disadvantage of not being meaningful for humans. The reason for that kind of choice is that while the onion addresses are secure, they are also self-authenticating. That means that when a client gets an onion addresses and requests the descriptor from the hidden service directory service, then it also receives the public key of the hidden service and can derive the onion address from it. It allows the client to make sure that it is encrypting and sending the data to the correct hidden service.

3 TorChat

TorChat is a peer-to-peer instant messaging solution built on top of Tor and its location-hidden services feature. It can be used to interactively chat and exchange files between participants.

On a very high level view, TorChat works by making every TorChat client available through the Tor network as a hidden service. The onion address of the hidden service is used as a unique identifier in TorChat. The onion address is a domain name allowing anyone to establish an end-to-end encrypted and authenticated connection over the Tor network with the service behind that address.

TorChat is written in Python and uses the wxPython library to draw GUI objects. To provide end-to-end encryption and anonymity features, TorChat relies on the official Tor client software.

TorChat has been designed to be simple to use. In case of the Windows version of TorChat, everything required to run TorChat is included in the TorChat Windows archive. On other platforms (Linux and Mac), TorChat requires the installation of the Tor client and wxPython libraries. Apart from that there is no configuration required – the Tor hidden service is started by TorChat and the key pair for the hidden service is generated when TorChat is launched for the first time.

However, the last update to the TorChat main code base was made on June 12, 2013. The executable packages have not been updated either. Because of this the latest TorChat Windows package available for download ships with an outdated Tor client that cannot be used anymore to connect to the Tor network.

3.1 Managing Contacts and Conversations

TorChat's user interface is very simple and minimalistic. The TorChat main window holds the contact list and only has one button for changing the status (see Figure 2). The application menu becomes available with a right-click somewhere inside the contact list window (see Figure 2).

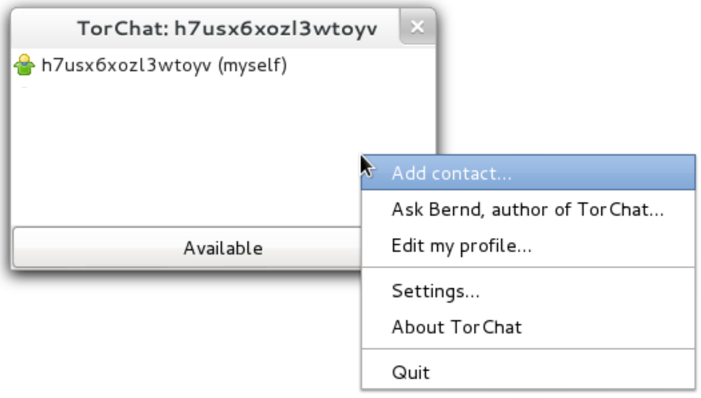

TorChat: h7usx6xozl3wtoyv
h7usx6xozl3wtoyv (myself)
Available
Add contact...
Ask Bernd, author of TorChat...
Edit my profile...
Settings...
About TorChat
Quit

Figure 2: TorChat main window with the application menu opened.

After TorChat is started for the first time it adds to the contact list the user's own TorChat contact, with the profile name set to "myself" (see Figure 2). This means that it is easy for the user to find his TorChat identifier and see whether he is online, i.e., whether his onion address is reachable over the Tor network, since after the TorChat client is started, it may take a while until Tor circuits are established and the hidden service descriptor is published in the Tor directory.

To add a contact the user must select the "Add contact..." entry from the application menu (see Figure 2). The "Add new contact" window will open (see Figure 3). There, the user can enter a peer's TorChat identifier, profile name and introduction message, which will be sent to the contact. If the profile name is omitted (the field is left empty), the profile name will be set to one received from the peer.

To remove a contact the user must select the "Delete contact..." entry from the application menu. After the user confirms the deletion, the remote peer will be informed about a deletion request and the corresponding contact will be deleted from the peer's contact lists. Note that if the peer is not online, the deletion request will not reach the peer. Once the peer comes back online, the contact will be added back to the peer's contact list.

A user can see his contact's profile information by hovering a mouse cursor over the contact's entry in the contact list. The profile information always shows the latest profile name, profile text and avatar received from the contact (see Figure 4).

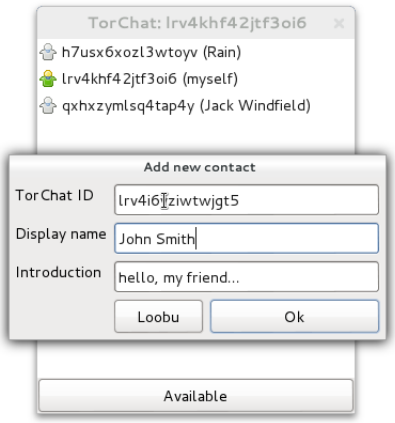


Figure 3: TorChat “Add contact” window.

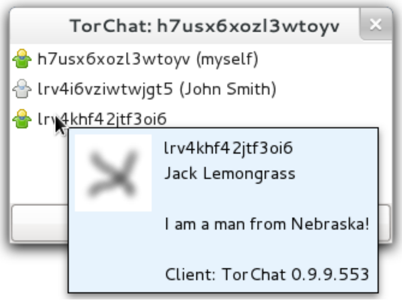
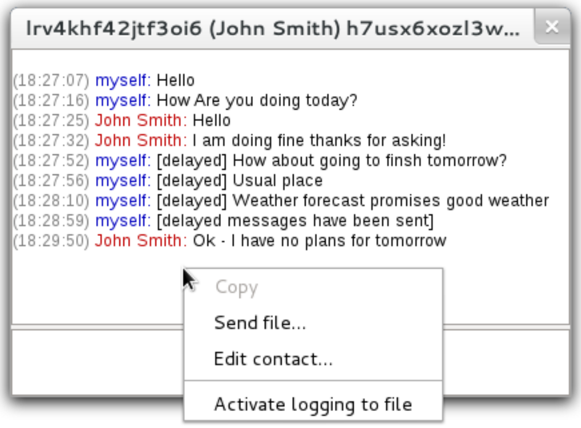

TorChat: h7usx6xozl3wtoyv
h7usx6xozl3wtoyv (myself)
lrv4i6vziwtwjgt5 (John Smith)
lrv4khf42jtf3oi6
lrv4khf42jtf3oi6
Jack Lemongrass
I am a man from Nebraska!
Client: TorChat 0.9.9.553

Figure 4: Contact’s profile information box.

The icon in front of the contact indicates the status of the contact. If the icon is grey, then the contact is offline. If the icon is a blue ball/globe, then this means that TorChat is in the middle of a handshake process with that contact. If the icon is green, then the contact is online. The icon can also be yellow and red, which means that the contact is online, but has set his availability status to “away” or “away for an extended period”, respectively.

To start a conversation with a contact, the user must double-click on the contact’s name. This will open a conversation window with that contact (see Figure 5). The messages entered for a contact who is offline will be saved in the `~/.torchat/<TorChat identifier>_offline.txt` file and sent to the peer when he comes back online. In the conversation window these messages will be marked with the “[delayed]” prefix.

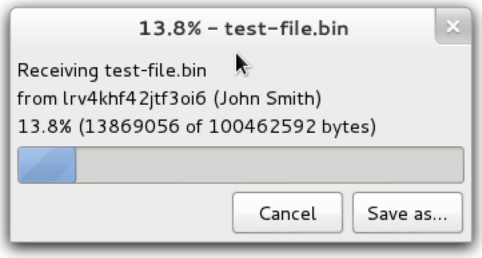

lrv4khf42jtf3oi6 (John Smith) h7usx6xozl3w...

```
(18:27:07) myself: Hello
(18:27:16) myself: How Are you doing today?
(18:27:25) John Smith: Hello
(18:27:32) John Smith: I am doing fine thanks for asking!
(18:27:52) myself: [delayed] How about going to finsh tomorrow?
(18:27:56) myself: [delayed] Usual place
(18:28:10) myself: [delayed] Weather forecast promises good weather
(18:28:59) myself: [delayed messages have been sent]
(18:29:50) John Smith: Ok - I have no plans for tomorrow
```

Figure 5: Conversation window.

The conversation window has its own menu, which can be accessed by right-clicking in the upper part of the window. The menu allows sending files to that contact, edit the profile name shown in the contact list and turn on/off message logging (off by default). If logging is enabled then the conversation with the peer is saved in the `~/.torchat/<TorChat identifier>.log` file.

To send a file the user must select the "Send file..." entry from the application menu. Files sent from contacts are automatically saved as temporary files in `~/.torchat/torchat_incoming_<unique identifier>`. The user receiving the file can click on the "Cancel" button to stop the download and delete the temporary file or click the "Save as..." button, which will save the file and remove the temporary file (see Figure 6).


13.8% – test-file.bin

Receiving test-file.bin
from lrv4khf42jtf3oi6 (John Smith)
13.8% (13869056 of 100462592 bytes)

Figure 6: File transfer window (receiving in progress).

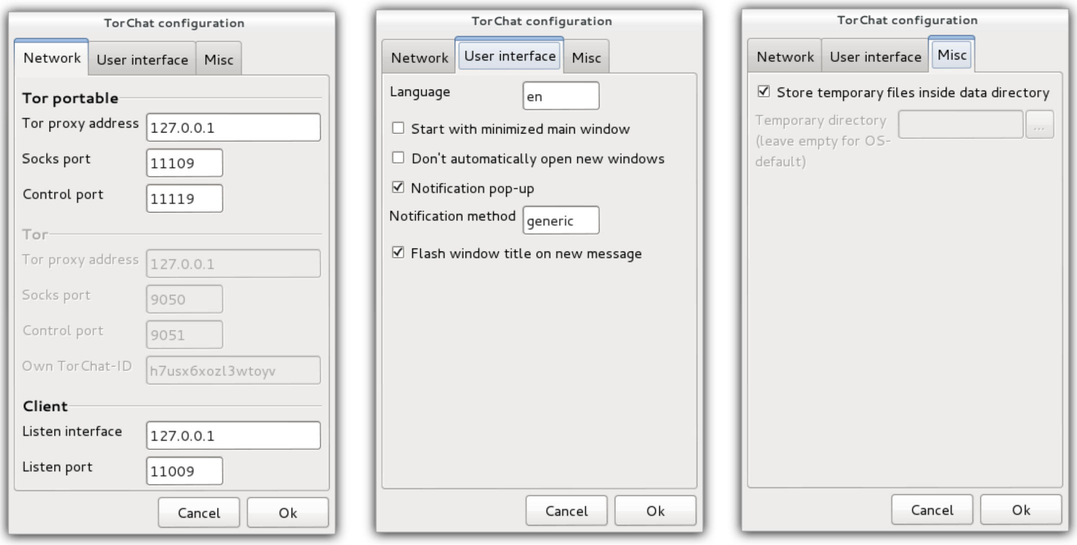
3.2 Configuration Options

TorChat can be configured using the “Settings” dialog of the application menu. The settings are stored in the `~/.torchat/torchat.ini` configuration file, which can also be edited with a text editor.

The TorChat’s settings dialog has three tabs: “Network”, “User interface” and “Misc” (see Figure 7). Under the “Network” tab the user can change the network configuration stored in `torchat.ini`. There is no need to modify the default network settings unless the user wants to run several TorChat clients in parallel [17].

The “User interface” tab can be used to specify whether the TorChat contact window should be minimized when the application starts, whether a new window should be opened automatically, and whether the application should notify the user about new messages. It can also be used to choose the interface language.

Under the “Misc” tab the user can configure whether temporary files should be stored inside the data directory (the folder from which TorChat was executed), in a default location in the operating system or a location specified by the user.



Tor portable
Tor proxy address: 127.0.0.1
Socks port: 11109
Control port: 11119

Tor
Tor proxy address: 127.0.0.1
Socks port: 9050
Control port: 9051
Own TorChat-ID: h7usx6xozl3wtoyv

Client
Listen interface: 127.0.0.1
Listen port: 11009

Language: en
☐ Start with minimized main window
☐ Don't automatically open new windows
☑ Notification pop-up
Notification method: generic
☑ Flash window title on new message

☑ Store temporary files inside data directory
Temporary directory (leave empty for OS-default)

Figure 7: TorChat configuration window.

4 TorChat Protocol

This section describes the protocol used by TorChat. The description provided here has been obtained by examining the source code.

4.1 Handshake

Before two parties can start exchanging messages they must perform the handshake process. The goal of the handshake process is to establish trusted connections between parties, so that after the process is completed, both parties know who they are communicating with, i.e., which Tor hidden service the other party controls.

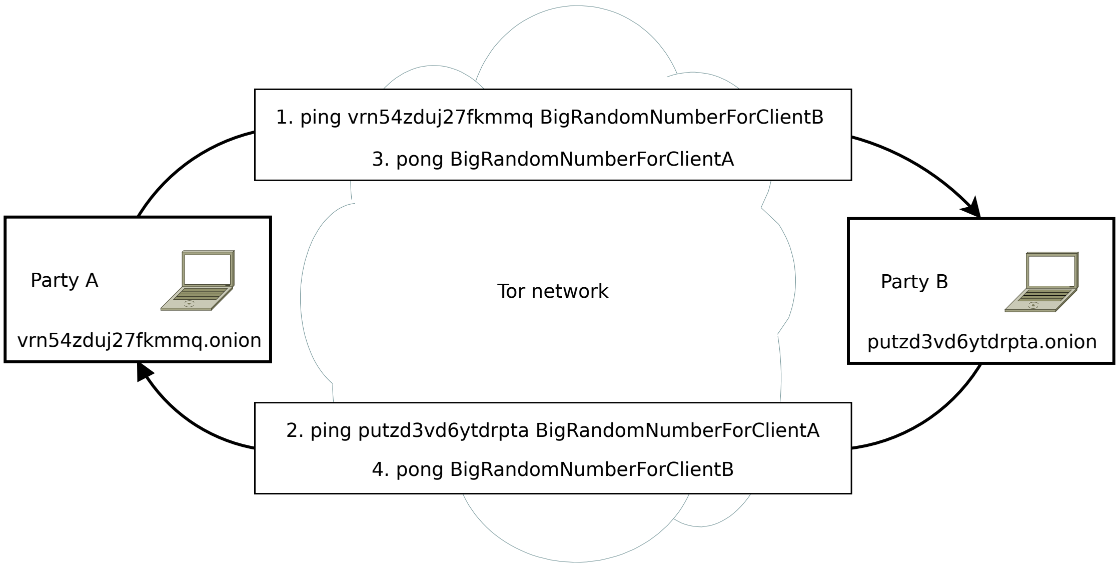

1. ping vrn54zduj27fkmmq BigRandomNumberForClientB
3. pong BigRandomNumberForClientA
Party A
vrn54zduj27fkmmq.onion
Tor network
Party B
putzd3vd6ytdrpta.onion
2. ping putzd3vd6ytdrpta BigRandomNumberForClientA
4. pong BigRandomNumberForClientB

Figure 8: Handshake process.

The handshake process (Figure 8) between TorChat clients party A and B:

1. Party A knows party B's onion address (established by out-of-band means). Party A establishes a TCP connection over the Tor network with party B's onion address to port 11009. Party A sends a `ping` message with its own onion address and a large random number. At this point party A knows with whom he has established a secure outgoing connection. However, party B does not know who is behind the anonymous incoming connection.

2. Party B establishes a TCP connection over the Tor network with party A's onion address received in step 1 to port 11009. Using the established connection, party B sends a `ping` message with its onion address and a new random number generated by party B.
3. Party B sends a `pong` message with the random number received in step 1 to party A. At this point party B knows with whom the secure outgoing connection has been established and party A knows that the anonymous incoming connection belongs to party B, because only party B could have replied with the same random number as sent in step 1.
4. Party A takes party B's random number received in step 2 and sends it in a `pong` message using the outgoing connection established in step 1. At this point party B also knows that the anonymous incoming connection established in step 1 belongs to party A, since only party A could have replied with the random number sent by party B in step 2.

After the handshake is completed, there are two end-to-end encrypted TCP connections between party A and party B and both party A and party B know the identity (onion address) of the other party.

4.2 File Transfers

File transfers are implemented using `file*` protocol messages. The files are split into blocks and sent using `filedata` protocol messages. To speed up file transfer, several blocks can be sent before the acknowledgement `filedata_ok` message is received. The block size used by TorChat is set to 8192 bytes and the number of blocks sent before receiving the acknowledgement is 16.

In general, the TorChat client accepts protocol messages only on incoming connections. The exceptions are all `file*` protocol messages, which are accepted on both incoming and outgoing connections. The TorChat client on the incoming connecton only sends `filename` and `filedata` protocol messages. This is done to avoid delaying the chat messages.

4.3 Protocol Messages

TorChat protocol messages are exchanged over TCP sockets. The messages are separated by a newline character '\n' (`0x0a` in hex). Whenever a full message (ending with a newline character) is received, it should be processed immediately.

The protocol message format is as follows:

```
<command> <optional parameters> <UTF-8 encoded data>
```

The `<command>` part may contain only lowercase characters a-z and the underscore character '_'. The command and encoded data are separated by a space (`0x20` in hex). The data part is a UTF-8 encoded binary string.

Backslash and newline characters inside the data part must be escaped. The backslash character '\' (`0x5c` in hex) must be replaced with two characters "\/" (`0x5c` and `0x2f` in hex) and the newline character '\n' (`0x0a` in hex) must be replaced by two characters "\n" (`0x5c` and `0x6e` in hex). The opposite replacement must be done after receiving a message.

In addition to a command, a message could also contain parameters for the command. The parameters are separated between other parameters and data with a space. Parameters cannot contain a space and are subject to the same formatting rules as commands.

The following subsections describe protocol messages, their purpose, the time when they can be sent and how they should be processed when received by the client application.

The words "must" and "should" are used interchangeably and do not have any special meaning.

4.3.1 not_implemented

If a peer receives an unknown message, he should send `not_implemented` over the outgoing connection. If the outgoing connection does not exist, the connection via which the unknown message was received should be closed.

4.3.2 ping

This message should be sent immediately after establishing an outgoing connection to a peer. It should contain the peer's own address without the ".onion" suffix and a 32-byte random cookie unique for the peer to whom the ping is sent.

Example: `ping vrn54zduj27fkmmq 13883080185381582480815986962179 4`

When a peer receives a `ping` message over the incoming connection he should first check:

1. If the address in the `ping` message is 16 characters long and contains valid Base32 encoding. If the address is not valid then the message should not be processed and the incoming connection must be closed.

2. If a repeated `ping` message contains a different address than the previous `ping` message received over the same connection then the message should not be processed and the incoming connection should be closed.

3. If a `ping` message contains an address that has already been assigned with another incoming connection for which the handshake process has been completed then the message should not be processed and the `not_implemented double connection` message should be sent over the incoming connection via which the `ping` message was received, but the connection should not be closed.

4. If the received `ping` message contains the peer's own address, but the cookie is different from the one expected, the message should not be processed and the connection over which the message was received should be closed.
   Note that after a TorChat client is started it establishes a connection with its own hidden service. If such a bogus message is received when TorChat has already established a connection with itself, the case should be handled by performing the "double connection" check above.

If the checks above succeed then a buddy with the peer's address specified in the `ping` message and the status `STATUS_OFFLINE` should be created. Once the buddy is created, an outgoing connection should be established with the peer's address specified in the `ping` message. Once the outgoing connection is established, the outgoing connection should be assigned to the buddy and the buddy's status should be updated to `STATUS_HANDSHAKE`. Using the outgoing connection established, the peer should send the following messages:

1. his own `ping` message;
2. the `pong` message with the random cookie received in the `ping` message;
3. `client` message;
4. `version` message;
5. `profile_name` and `profile_text` messages (if set);
6. `profile_avatar_alpha` and `profile_avatar` messages (if avatar is set);
7. `add_me` message (if the contact is in the peer's buddy list);
8. `status` message.

If a repeated `ping` message is received, but the buddy already has an outgoing connection over which a `pong` message has been sent then the `ping` message received should be ignored, unless the `pong` reply from the peer has not been received yet, in which case the `ping` message should be sent over the outgoing connection.

4.3.3 pong

This message should be sent over the outgoing connection as a reply to the `ping` message received over the incoming connection. The message must contain the random cookie received in the `ping` message.

Example: `pong 138830801853815824808159869621794`

When a peer receives a `pong` message over an incoming connection, he must process it as follows:

1. Extract the random cookie from the message and find the first buddy who has been `ping`'ed with this random cookie. If a buddy with the random cookie cannot be found the `pong` message should be ignored.

2. If the buddy is found the peer must check whether the address assigned to the buddy is the same address that has been specified in the `ping` message received over this incoming connection. If this check fails, the `pong` message should be ignored.

If these checks succeed the incoming connection should be assigned to the buddy and the previous incoming connection (if it exists and is not the same) should be closed.

4.3.4 client

This message is sent as a reply to the `ping` message (see Section 4.3.2). The message must contain the name of the client software.

Example: `client TorChat`

When received, the peer should assign the received client software name to the contact.

4.3.5 version

This message is sent in an answer to `ping` message (see Section 4.3.2). The message must contain the version string of the client software.

Example: `version 0.9.9.553`

When received the peer should assign the received version string to the contact.

4.3.6 status

This message must be sent as a reply to the `ping` message (see Section 4.3.2) immediately after the user has changed his status or profile information and in every 120 seconds. The message can contain three different status values: “available”, “away” and “xa” where “xa” stands for “away for an extended period”.

Example: `status available`

When receiving this message, the client should update the status information of the contact accordingly. If the `status` message is not received from a peer in 240 seconds the incoming connection should be closed.

4.3.7 profile_name

This message is sent as a reply to the `ping` message (see Section 4.3.2) and to all peers whenever the user changes his profile information. The message is not sent if the user has not specified his profile name or the user changes his profile name to a blank name.

Example: `profile_name John Smith`

If a client receives this message and the profile name for the peer is not set, the client should store the profile name in the contact list and show it next to the peer's TorChat identifier in the contact window. If a client receives this message, but the profile name for the peer is already set, the profile name should not be changed, but the profile name received should be shown when a mouse cursor hovers over the contact's entry in the contact window (see Section 3.1).

4.3.8 profile_text

This message is sent as a reply to the `ping` message (see Section 4.3.2) and to all peers whenever the user changes his profile information. The message is not sent if the user has not specified his profile text or the user changes his profile text to a blank text.

Example: `profile_text Business consultant with extensive experience`

The profile text received in this message should not be stored in the contact list, but should be shown when a mouse cursor hovers over the contact's entry in the contact window (see Section 3.1).

4.3.9 profile_avatar_alpha

This message is sent as a reply to the `ping` message (see Section 4.3.2) and to all peers whenever the user changes his avatar. The message is not sent if the user has not specified his avatar or the user removes his avatar.

This message contains an uncompressed 64*64*8 bit alpha channel. If there is no alpha channel in the avatar, the message has to be sent without the data part. This message has to be sent before sending the `profile_avatar` message.

Example: `profile_avatar_alpha <alpha channel binary data>`

When receiving this message, the client should wait for the peer's `profile_avatar` message to construct the avatar.

4.3.10 profile_avatar

This message is sent as a reply to the `ping` message (see Section 4.3.2) and to all peers whenever the user changes his avatar. The message is not sent if the user has not specified his avatar or the user removes his avatar.

The message contains an uncompressed 64*64*24 bit image. This message must be sent after the `profile_avatar_alpha` message is sent.

Example: `profile_avatar <image data as raw binary data>`

When receiving this message, the client should construct an avatar taking into account the alpha channel received in the `profile_avatar_alpha` message and make the peer's avatar visible when a mouse cursor hovers over the contact's entry in the contact window (see Section 3.1).

4.3.11 add_me

This message is sent as a reply to the `ping` message (see Section 4.3.2) if the peer is in the sender's contact list.

Example: `add_me`

4.3.12 remove_me

This message is sent to the peer when a user removes the peer from his contact list. If the peer is not online, this message is not sent. After removing the peer from the contact list, the peer's offline message queue file should also be wiped.

Example: `remove_me`

When receiving this message, the client should remove the peer from the contact lists and contact window, wipe the peer's offline message queue file and close the incoming and outgoing connections associated with the peer.

4.3.13 message

This message is sent to the peer whenever a user has entered text in the conversation window or after the peer comes online and there are unsent messages in the offline message queue.

The message may be sent only after the `add_me` message has been sent or when it is known that the peer is in the receiver's contact list.

Example: `message Hello, how are you?`

4.3.14 filename

The `filename` message is a message that initiates file transfer.

The data part of the message contains information about the file being sent:

- id – identifier generated by the sender that uniquely identifies the file transfer.
- file_size – size of the file in bytes.
- block_size – specifies the chunk size that will be sent in `filedata` messages (the actual `filedata` messages can contain blocks in a different size).
- file_name – specifies the name of the file as stored in the sender's system.

Every field is separated by a space.

Example: `filename 2665323703 11 8192 testfile.txt`

4.3.15 filedata

This message is used to transport the actual data in blocks of a fixed size. Every message contains the file transfer identifier, offset in the original file, the lower-case MD5 hash of the current data block and an arbitrary-size block of the data itself. Each message should be replied to with the `filedata_ok` message after the receiver has successfully verified the hash of the data block. The hash of the block is calculated on the basis of the unescaped data block.

The sender should send only a limited number of blocks ahead of the incoming `filedata_ok` messages. For example, the sender should send the fifth block only after the arrival of the first block is confirmed (`filedata_ok` message for the first block has arrived), the sixth only after the arrival of the second block is confirmed and so on. The `filedata` messages must be sent in sequential order.

Example:
`filedata 2665323703 0 ee5a58024a155466b43bc559d953e018 line1\nline2`

4.3.16 filedata_ok

This message is sent as a reply to the `filedata` message once the receiver has successfully verified the `filedata` message. The sender of the file can use this message to update the file transfer progress bar and learn that more blocks can be sent.

The message contains the file transfer identifier and the offset position of the block that has been successfully received.

Example: `filedata_ok 2665323703 0`

4.3.17 filedata_error

The message is sent if there has been a problem receiving the `filedata` message. The problem might be that the hash was wrong or the file offset was not the one that was expected (too much ahead of what it should be, meaning that some blocks have been skipped or lost during a temporary network outage).

The sender must respond to this message by restarting the file transfer at the offset specified in the message.

The message contains the file transfer identifier and the file offset from which the file transfer should be resumed.

Example: `filedata_error 2665323703 0`

4.3.18 file_stop_sending

The message is sent to the file sender if the receiver has canceled the file transfer. After this message is received, the sender should stop sending the file. The user must be notified that the file receiver has canceled the file transfer.

Example: `file_stop_sending 2665323703`

4.3.19 file_stop_receiving

The message is sent to the file receiver if the sender has canceled the file transfer or the sender has received from the receiver the `file_stop_sending` message. Once this message is received, the receiver should not expect any more messages related to the identified file transfer. All the allocated resources should be freed, temporary files should be wiped and the user should be notified about the cancellation.

Example: `file_stop_receiving 2665323703`

5 Analysis Methodology

The security analysis methodology presented in this section is based on EFF's "Secure Messaging Scorecard" [18], but has additional criteria that concern not only the privacy of the content of messages but also the privacy of communication metadata and other issues related to the secure and anonymous use of instant messaging in practice.

5.1 Is the communication protected in transit?

This criterion considers an attacker who controls the network connection between the user and the IM service provider. How is the data protected against such an attacker? What can the attacker learn by observing the connection? Can he learn whether the user is using the IM service? Can he learn with whom the user is communicating? Can he tamper with the communication line to execute an attack, other than the denial-of-service attack?

5.2 Is the communication protected against abuse from the provider?

This criterion considers a malicious IM service provider whose objective is to compromise the user's privacy, communication integrity or even the device on which the IM software is running. With what is the user trusting the service provider?

5.3 Can someone impersonate the user?

This criterion should answer whether the user has means to verify the cryptographic binding between the communication channel and his contact's identity or does he have to trust the IM service provider on that.

5.4 Are past communications secure if the user's keys are stolen?

This criterion considers an attacker who has collected encrypted communication at any point between the user and the intended recipient and then obtains a long-term encryption key from the user in order to decrypt collected past communication. Is forward secrecy used, i.e., is the user's long-term asymmetric key used in the signing mode only to establish a short-term encryption key?

5.5 Is the source code available, crypto design well-documented, open to independent review?

This criterion considers whether the source code of the software is freely available and whether the design of the software is well-documented.

5.6 Can the service be used anonymously?

Can the user register an IM service account anonymously? Is the user's connection IP address concealed from his contacts and from the IM service provider?

5.7 Who has access to the user's profile information?

This criterion considers how is the user's profile information protected against third parties. Is this information available only to the persons in the contact list, to the IM service provider, or someone else?

5.8 Who has access to the user's presence-related information?

This criterion considers both the presence-related information (reveals whether the user is logged into the IM service) and availability-related information (reveals the user's specified status in the IM service, i.e., busy, away, available). Is this information available only to the persons in the contact list, to the IM service provider, or someone else?

5.9 Who has access to information on the user's contacts?

Who, except for the parties involved in the communication, can learn about the parties in contact?

5.10 Is the user protected against denial-of-service attacks?

Is the IM service vulnerable to any attack which could be used by an attacker to deny the user access to the service?

This includes any kind of targeted attacks which result in the messaging client crashing, intensive CPU or memory usage, or in any other way that creates trouble for the user using the service.

5.11 Which forensic evidence does the software leave on the user's device?

This criterion should answer what kind of IM-related information can be extracted from the device in case the device falls into the hands of a third party. Things to look for — contact list, messaging history, logs, keys, file transfer history, offline message queue, etc.

5.12 Is the software available from a trusted source and can its integrity be verified?

How is the software made available? Are the distribution methods and channels trusted? If the binary packages are provided, are there ways to reproduce them? Are there ways to verify package authenticity?

6 Security Analysis

This section gives detailed answers to the questions defined in the previous section.

6.1 Is the communication protected in transit?

TorChat does not implement its own encryption, but fully relies on the confidentiality and authenticity guarantees provided by Tor and the Tor hidden service design.

Adversary eavesdropping on the communication channel between the TorChat client and the Tor network would see encrypted traffic protected by four layers of encryption, where the innermost layer contains end-to-end encrypted TorChat protocol data exchanged between TorChat peers. Every layer is protected by a 128-bit AES key negotiated using 1024-bit RSA and ephemeral DH keys [15].

Therefore, the only information the attacker can learn is that the user is using Tor. This fact is easy to learn since the entry nodes of the Tor network can be identified by their IP addresses.

6.2 Is the communication protected against abuse from the provider?

In the TorChat context we can consider the Tor network to be the IM service provider. There are several entities involved in establishing a connection between two Tor hidden services (TorChat peers).

If the hidden service directory in which TorChat's hidden service descriptor is published is under an adversary's control, the adversary can only learn the same information as the users who are requesting that descriptor. This includes introduction points which can be used to contact the hidden service, the public key (onion address) of the hidden service and the time when the descriptor was published.

An adversary operating node which acts as a rendezvous point between TorChat peers would still have to break the final layer of end-to-end encryption between TorChat peers. An attack by a rendezvous point is complicated even further, because rendezvous points are chosen randomly by the Tor client, both end-points from the perspective of the rendezvous point are anonymous, and a single rendezvous point would most likely route only one direction of TorChat communication.

To sum up, if the security assumptions implied by Tor hold, the IM service provider is not capable of attacking TorChat users.

6.3 Can someone impersonate the user?

The following subsections contain a discussion on several attacks that can be used to impersonate a TorChat user.

6.3.1 Impersonating the Tor hidden service

For authenticity guarantees TorChat exploits the self-authenticating nature of onion addresses, where the onion address represents the public key of a hidden service's long-term 1024-bit RSA key.

As can be seen from the description of the handshake process (see Section 4.1), after the handshake is completed both TorChat peers are sure that the opposite party is in control of the private key corresponding to the onion address.

In order to impersonate a TorChat peer an adversary would have to compromise the 1024-bit RSA key of the hidden service. The RSA key pair is generated by Tor when TorChat is first used. In Unix systems the generated key is stored in the file `~/.torchat/Tor/hidden_service/private_key`, which is readable only by the user.

While the security of 1024-bit RSA is nowadays considered weak [19], we do not know of any cases where the 1024-bit RSA key would have been factored. Since this long-term key is used only for authentication, the use of 1024-bit RSA is tolerable for now.

Alternatively, since the onion address consists of the Base32-encoded first 80 bits of SHA-1 digest of the RSA public key, an adversary may try to find a different RSA key which would collide with the same onion address. However, the complexity of such an attack is estimated to require 2^{80} operations, which is nearly the same as factoring the 1024-bit RSA key. In case the attacker can find a different RSA key which produces the same onion address, there would be a race condition with the legitimate user's published hidden service descriptor [20].

6.3.2 Spoofing the `pong` message

As can be seen from the TorChat protocol description (see Section 4.1), a peer authenticates the incoming connection by checking if the random number received in the `pong` message matches the random number that was sent in the `ping` message over the outgoing connection. Thus, if the attacker is able to guess the random number which was sent in the `ping` message, he can spoof the `pong` message and thus impersonate the incoming connection.

In TorChat implementation the random number sent in the `ping` message is generated by calling the `random.getrandombits(256)` method which will return an integer from 0 to $2^{256} - 1$ generated by the MersenneTwister pseudo-random number generator. The Python manual states that it should not be used for security purposes and suggests using a cryptographically secure pseudo-random number `os.urandom()` or `SystemRandom` [21] instead.

Even if the attacker can predict the random number used in the `ping` message, the impact of the attack is limited, since the attack has to be executed at a time when vulnerable victim performs the handshake and before the legitimate peer has answered with his `pong` message. Furthermore, in case of successful impersonation the attacker can only send impersonated messages, but not read the responses. The exceptions are file transfers which are sent over the incoming connection (see Section 4.2). Thus, if the attacker convinced the victim to send a file, the file would be received by the attacker over the victim's incoming connection.

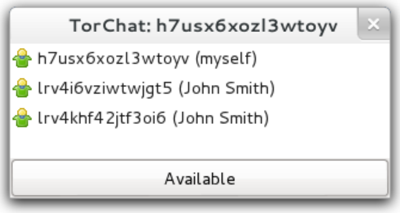
6.3.3 Impersonation at the GUI level

Since TorChat peer identities are onion addresses that are hard to memorize, TorChat provides the possibility to assign an arbitrary name to a contact which will be shown next to the TorChat identifier in the contact list. However, the way it is implemented in TorChat opens the contact list to confusion attacks that may lead to a successful impersonation attack.

As described in Section 3.1, when the user is adding a new contact, he can specify the contact's profile name. If the name is left blank then it is set to the name specified in the `profile_name` message received from the contact.

This allows the remote peer to set the profile name specified by him if the user does not specify one when adding the contact. This is not a problem, since adding the contact is an operation consciously performed by the user and the user has the opportunity to set the profile name. However, if someone adds the user to his contact list, then that someone is added to the user's contact list automatically and the contact's profile name is set to the name sent by that someone in the `profile_name` message. The user has no way to control what gets added to his contact list.

Thus, if the attacker knows with whom the victim is chatting, the attacker can generate a similar-looking onion address [22], set the same profile name, and add the victim to his contact list. This way the victim will have several contacts with the same contact name and a similar TorChat identifier, which the attacker can use to start the conversation and trick the user into thinking that the window with the attacker's conversation is the intended TorChat peer (see Figure 9).


TorChat: h7usx6xozl3wtoyv

h7usx6xozl3wtoyv (myself)

lrv4i6vziwtwjgt5 (John Smith)

lrv4khf42jtf3oi6 (John Smith)

Available

Figure 9: Two similar-looking TorChat contacts. The second one has been added automatically by the attacker.

One thing that the victim can do to determine the false TorChat identity is to look at the order of contacts in his contact list. The contacts in the contact list are ordered so that newly added or edited contacts are moved to the bottom of the list.

This kind of an attack could be prevented if TorChat asked for confirmation before adding a contact to the contact list. Note that arbitrary contacts can be added to the contact list also by exploiting the contact list manipulation flaw described in Section 6.10.3.

6.4 Are past communications secure if the user's keys are stolen?

As already described in Section 6.1, Tor uses layered encryption where every layer provides forward secrecy, thus preventing an adversary who has obtained a long-term private key from decrypting previously collected network traffic.

6.5 Is the source code available, crypto design well-documented, open to independent review?

For both TorChat and Tor, source codes are freely available and open to independent review. The situation is different regarding the documentation.

Tor has a high-level overview of the main features and principles published on Tor's website. The code repository contains specifications which give an in-depth description of the protocol and the operation of the Tor network.

TorChat, on the other hand, does not have much documentation available. There is some general information on its GitHub page and there is also some general documentation regarding the configuration and different working modes available in the "docs" folder together with the source code. The protocol of TorChat is not described in any separate document. This paper aims to fix this issue.

6.6 Can the service be used anonymously?

The anonymity of any person using TorChat can be reduced to the anonymity guarantees provided by Tor and the hidden services design. In literature, several attacks have been described which could be used to locate a Tor hidden service or a Tor user [9]. The most popular type of attack is traffic confirmation attack.

The subsections below discuss some anonymity aspects that affect specifically TorChat.

6.6.1 Deanonymization by a malicious guard node

One of the weaknesses of Tor and other low-latency anonymity networks is that if the attacker can monitor both ends of the communication channel, then he can correlate the data volume and timing information and compromise anonymity. In the Tor network this means that the attacker needs to control the circuit's first and last relay. In case of hidden services it means that the attacker needs to control just the entry node chosen by the hidden service, since the other end is already under the attacker's control.

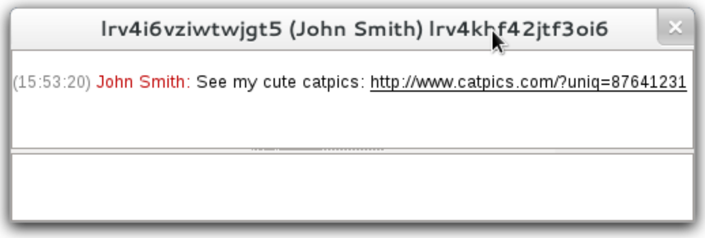
If the Tor client always chose a new entry guard for each circuit then Tor would eventually pick the entry node controlled by the attacker and the user's anonymity would be compromised. To protect against this, the Tor client randomly chooses a fixed relay that will act as an entry guard. The guard node rotation time is set by the configuration parameter `"GuardLifeTime"`; by default it is 60 days [23, Section 3.4.1].

In the configuration file that is shipped with TorChat (`~/.torchat/Tor/torrc.txt`), the number of entry guards is set to six (`NumEntryGuards 6`). Considering that Tor's default value is coming from the directory authority concensus and is currently one or if the number is not found in the consensus file, it defaults to three [23, Section 3.4.1], then TorChat's use of six nodes compared to Tor's default settings considerably increase the risk of TorChat user deanonymization and should therefore be reconsidered.

6.6.2 Deanonymization by message contents

It is a well-known fact that Tor provides anonymity only at the transport protocol level. If the user exposes in his TorChat profile or sent messages information that could be used to identify him, Tor anonymity guarantees are no help.

TorChat users have to be especially careful with the links they receive in conversations. If Tor is used only by TorChat and other applications go directly to the Internet, opening a link from the TorChat window can expose the user's real IP address to the server hosting the website, which, if under the control of an attacker, will allow the attacker to find out the TorChat user's real IP address. To make it easier to slip TorChat makes all links clickable (see Figure 10).


lrv4i6vziwtwjgt5 (John Smith) lrv4khf42jtf3oi6

(15:53:20) John Smith: See my cute catpics: http://www.catpics.com/?uniq=87641231

Figure 10: Clickable links in TorChat.

This is the simplest way to deanonymize a TorChat user. It is not recommended to remove link clickability, since the user will most likely copy and paste the same URL into a browser. However, when the user clicks on a link, he should see a message warning him about the deanonymization risk, with the option to click a checkbox so as not to see the warning again.

6.7 Who has access to the user's profile information?

The user's profile information (profile name, description and avatar) is available to contacts in the user's contact list. However, since contact requests are processed automatically, anyone who knows the user's TorChat identifier can become his contact and receive his profile information.

Even more, a full handshake is not needed as according to the protocol (see Section 4.3), the user's profile information will be disclosed automatically in response to a peer's `ping` message.

This should be prevented by asking for a confirmation before adding a contact to the contact list and by sending profile information only after the peer has been added to the contact list.

6.8 Who has access to the user's presence-related information?

It would be desirable that the user's presence-related information be disclosed only to contacts in the user's contact list. As described in the previous section, TorChat discloses profile information, including availability-related information to anyone who initiates the handshake process.

However, even if TorChat disclosed availability only to the contacts in the contact list, presence-related information is still available to every client in the Tor network who is able to download a hidden service descriptor. The descriptor is updated in the directory server once an hour or whenever its content changes [14, Section 1.4].

By collecting presence-related information, the adversary can build an activity graph and use it to correlate TorChat activity with other activities, thereby discovering the real identity of the TorChat user.

There is no solution for that and the only recommendation is to make sure that TorChat is online all the time.

It is interesting to note that someone who is in the user's contact list can distinguish between a network outage and a TorChat client restart, since the random number used to authenticate the peer is not regenerated if the connection between peers breaks up.

6.9 Who has access to information on the user's contacts?

Thanks to Tor and its hidden service design, the TorChat peers communicating should be the only parties that know about the communication taking place and for anyone else, it should be very difficult to find out who is communicating with whom.

However, there exists a basic communication confirmation attack that can be performed due to the way the TorChat protocol handles the received `ping` messages that contain a TorChat identifier from a peer with whom the handshake process has already been established.

As can be seen in the protocol description (see Section 4.3.2), if the `ping` message is received with a peer identifier with whom the handshake process has already been established, TorChat will ignore the `ping` message and send `not_implemented double connection` using the connection over which the `ping` message was received. This can be used by an adversary to test if the victim has established a handshake with some other TorChat peer.

The simple fix would seemingly be to just ignore the double `ping` message without informing the sender about a double connection. However, the proper fix would be to handle the double `ping` message in a way that would prevent the attacker from distinguishing between a case where the double `ping` message is ignored and a case where TorChat tries to establish a back-connection to the address specified in the spoofed `ping` message.

6.10 Is the user protected against denial-of-service attacks?

Since TorChat relies on the Tor hidden service design, any denial-of-service attack against Tor hidden services is also applicable to TorChat. In order to make the TorChat client unavailable one can use general attacks against the Tor hidden services described in [9].

The subsections below focus on application-level denial-of-service attacks which apply specifically to TorChat.

6.10.1 Memory exhaustion through network read

TorChat does not limit the length of any message and buffers bytes into the memory until a command separator (newline character) is received. This can be used trivially by the attacker to exhaust the available memory in the victim's system by sending an endless stream of data over the victim's outgoing connection (but not the incoming connection as the victim's incoming connection expects to receive a keepalive `status` message every 120 seconds). How fast the memory in the victim's system is exhausted depends on the memory capacity of the victim's system and the speed at which the victim can receive data over the Tor network.

6.10.2 Memory exhaustion through a chat message

A more efficient but also a less invisible memory exhaustion attack can be performed by sending a large chat message in the `message` command. For instance, a 20 MB `message` command displayed in the chat window will cause TorChat to consume around 1 GB of memory.

6.10.3 Attacking via the `profile_name` message

As described above, TorChat does not enforce a size limit to the messages received. This allows an attacker to send an arbitrarily large profile name. A large profile name will cause the TorChat GUI to hang while GTK tries to update the contact list window. This will also prevent the victim from removing the attacker from the contact list since the profile name is shown in the "Confirm deletion" window, which will not be created by GTK if wider or taller than 32767 pixels. The victim can remove the attacker's contact by first changing the attacker's profile name in the "Edit contact" menu.

TorChat also fails to validate the profile name before writing it into the contact list (`buddy-list.txt`). The file stores every contact on a separate line. The line starts with the contact's TorChat identifier (onion address) and continues with the contact's profile name separated from the TorChat identifier with a space.

The lack of validation allows adding arbitrary lines into the contact list by sending out a profile name that contains escaped newline characters.

The contact list is saved right after the `profile_name` message is received; however, injected lines will be read only after TorChat is restarted. The injected lines will be lost if a contact is added to or removed from the contact list before TorChat is restarted.

An attacker can exploit this to cause an effective DoS attack by adding thousands of contacts into the contact list. On start-up TorChat will try to create and establish a connection with all the contacts in the contact list, which will cause extensive Tor activity and will prevent the TorChat GUI from starting. The only way how the victim can use TorChat again is to manually clean up the `buddy-list.txt`.

6.10.4 Attacking via multiple `add_me` messages

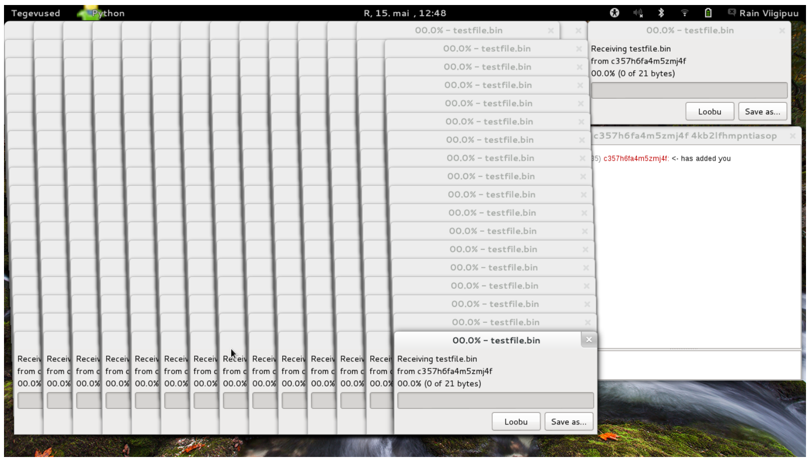
As stated before, TorChat does not ask for the user's consent before processing a peer's `add_me` request. The attacker can exploit this by flooding the user with dummy contacts and messages.

There has been a discussion about adding a block list feature to TorChat [24]. However, no solution has been implemented. The blacklist approach might not be effective since the attacker can introduce a new TorChat identity at an insignificant cost. A more appropriate solution would be the whitelist approach were the user is asked for confirmation before a contact is added to his contact list.

6.10.5 Attacking via multiple `filename` messages

Peers in the user's contact list can send files which will be automatically accepted by TorChat and downloaded in a separate window (see Section 3.1).

An attacker can initiate several file transfers by sending multiple `filename` messages. This will fill up the victim's screen with file transfer windows and cause the memory in the victim's system to be exhausted very quickly, resulting in a very efficient denial-of-service attack (see Figure 11).



```
Receiving testfile.bin
from c357h6fa4m5zmj4f
00.0% (0 of 21 bytes)
```

```
35) c357h6fa4m5zmj4f: <- has added you
```

Figure 11: Denial-of-service attack with multiple file transfer windows.

The impact of the attack could be significantly reduced if file transfers were shown in a single file transfer window and if arbitrary contacts were not added to the contact list without the user's consent.

6.11 Which forensic evidence does the software leave on the user's device?

All user-specific TorChat-related files are stored in the `~/.torchat/` directory. This includes `buddy-list.txt` that contains the user's contacts, `torchat.ini` that contains TorChat configuration parameters including the user's profile name and profile text, `avatar.png` (if set by the user), logged conversations in `<TorChat identifier>.log` files (logging is not enabled by default), an offline message queue in files that are in the form `<TorChat identifier>_offline.txt`, and temporary files received from other peers in the form `torchat_incoming_<unique identifier>`.

The `~/.torchat/Tor/` directory stores the Tor hidden service long-term private key in the `hidden_service/private_key` file, the Tor configuration in the `torrc.txt` file and the Tor cache in the `tor_data/` directory.

TorChat uses secure erasing (wiping) in several situations. Any temporary files received are wiped as soon as the user saves the file or cancels the download. Offline message queue files are wiped as soon as messages are delivered or the contact is removed from the contact list.

6.12 Is the software available from a trusted source and can its integrity be verified?

It is important to obtain the TorChat application from a trusted source to be sure about its integrity. The original source for TorChat is available on TorChat's project page in GitHub [5]. Since GitHub is available over a secure connection and the history of all versions is preserved in GitHub, it is safe to assume that TorChat obtained from GitHub has not been modified by an adversary.

GitHub's download section [25] contains prebuilt packages of TorChat releases available for MS Windows and Debian Linux distribution.

6.12.1 Linux

TorChat is also available in several Linux distribution package repositories:

- Debian Linux stable version 8 has the latest version of TorChat 0.9.9.553 [26].
- Ubuntu Linux version 14.04 [27] and the latest version 15.04 [28] both have TorChat version 0.9.9.553.
- Arch Linux has TorChat 0.9.9.553 available in their Arch User Repository (AUR) [29].

Both Debian and Ubuntu apply minor patches to the versions they provide through package repository: the Spanish translation of TorChat, a change in the way how the SOCKS proxy Python module is found in the system and an upgrade to the wxWidgets Python library version from 2.8 to 3.0. Only Ubuntu 14.04 is missing the patch needed to upgrade the wxWidgets Python module. Arch Linux does not apply any patches and provides an unmodified version of TorChat.

6.12.2 Windows

For MS Windows users, the TorChat page in GitHub provides an executable file that contains everything needed to run TorChat out-of-box, including the Tor client. The Windows executable file has allegedly [30] been compiled using PyInstaller [31].

However, the latest Windows executable file ships with an outdated version of the Tor client, 0.2.2.39 [32]. This Tor version has several vulnerabilities including heartbleed, which allows a malicious Tor guard node [33] to compromise the TorChat long-term private key. Fortunately, this version of Tor is too outdated to even connect to the Tor network [34].

7 Summary of Findings

Below is a list of security issues found, ranked by significance.

1. TorChat processes contact requests and updates the contact list without asking the user's consent. This allows an attacker to harvest profile (Section 6.7) and availability (Section 6.8) information, allows executing contact confusion and impersonation attacks at the GUI level (Section 6.3.3) and makes denial-of-service attacks (Section 6.10.2, 6.10.3, 6.10.4, 6.10.5) easier to execute.
2. Due to the way in which the TorChat handshake process is implemented, it is possible for an attacker to determine if two TorChat clients who are online have established a TorChat handshake, thus allowing the attacker to execute a communication confirmation attack (Section 6.9).
3. TorChat does not enforce a length limit on received protocol messages and their parts. This allows an attacker to execute efficient denial-of-service attacks against the TorChat client (Section 6.10.1, 6.10.2, 6.10.3).
4. TorChat fails to validate the contents of received `profile_name` messages before writing them into the contact file. An attacker can exploit this to add arbitrary contacts to the victim's contact list and execute a permanent denial-of-service attack, resulting in the victim's TorChat client failing to start (Section 6.10.3).
5. TorChat uses a cryptographically insecure pseudo-random number generator to generate random numbers used in the TorChat handshake process. This may allow an attacker to impersonate an incoming connection from a victim's contact. Thus, the attacker would be able to send messages on behalf of that contact and receive file transfers designated for that contact (Section 6.3.2).
6. TorChat runs Tor with non-default parameters which, compared to other Tor users, makes TorChat users easier to deanonymize with a malicious Tor guard node (Section 6.6.1).
7. TorChat makes links automatically clickable without warning the user about possible deanonymization attacks (Section 6.6.2).

8 Conclusions

The objectives set in the introduction were achieved. The TorChat protocol has been documented, reference implementation has been audited, and as a result of security analysis, several security considerations were revealed that need to be considered when using TorChat.

The designer of TorChat has made several smart design choices by exploiting the self-authenticating nature of Tor hidden services to provide authentication between TorChat peers and by leaving the encryption and anonymity part to be handled by the well-tested and widely used Tor software.

However, several flaws were found in the implementation of TorChat that make TorChat users vulnerable to denial-of-service attacks and prevent TorChat from achieving the privacy guarantees it could theoretically provide.

Despite the flaws found, the use of TorChat might still be secure in a scenario where the peer's onion address does not became known to an adversary interested in attacking the person behind the TorChat address.

Fortunately, the fixes for the vulnerabilities found can be implemented in the code relatively easily and without needing to change the design of TorChat.

References

[1] Ryan Gallagher. Timeline: How the World Was Misled About Government Skype Eavesdropping. July 2013. http://www.slate.com/blogs/future_tense/2013/07/12/skype_surveillance_a_timeline_of_public_claims_and_private_government_dealings.html.

[2] James Risen and Nick Wingfield. Web's Reach Binds N.S.A. and Silicon Valley Leaders. June 2013. http://www.nytimes.com/2013/06/20/technology/silicon-valley-and-spy-agency-bound-by-strengthening-web.html.

[3] Glenn Greenwald, Ewen MacAskill, Laura Poitras, Spencer Ackerman, and Dominic Rushe. Microsoft handed the NSA access to encrypted messages. July 2013. http://www.theguardian.com/world/2013/jul/11/microsoft-nsa-collaboration-user-data.

[4] Privacy International. What is metadata? https://www.privacyinternational.org/?q=node/53 (last visited 16.05.2015).

[5] Bernd Kreuss. TorChat, January 2014. https://github.com/prof7bit/TorChat.

[6] The Tor Project, Inc. Tor: Overview. https://www.torproject.org/about/overview.html.en (last visited 16.05.2015).

[7] Bernd Kreuss. Interview with Bernd Kreuss of TorChat, October 2013. http://www.reddit.com/r/onions/comments/1l9k8m/interview_with_bernd_kreuss_of_torchat/ccnmivi.

[8] Joe Mullin. Silk Road trial: FBI reveals what's on Ross Ulbrichts computer, January 2015. http://arstechnica.com/tech-policy/2015/01/silk-road-trial-fbi-reveals-whats-on-ross-ulbrichts-computer/.

[9] Alex Biryukov, Ivan Pustogarov, and Ralf-Philipp Weinmann. Trawling for Tor Hidden Services: Detection, Measurement, Deanonymization. In *IEEE Symposium on Security and Privacy'13*, pages 80–94, 2013.

[10] Alex Biryukov, Ivan Pustogarov, and Ralf-Philipp Weinmann. Content and popularity analysis of Tor hidden services. *CoRR*, abs/1308.6768, 2013.

[11] Bernd Kreuss. Torchat2 readme. `https://github.com/prof7bit/TorChat/blob/torchat2/README.markdown`.

[12] jTorchat, October 2014. `https://github.com/jtorchat/jtorchat`.

[13] torchat - the Ruby implementation, August 2012. `https://github.com/meh/ruby-torchat`.

[14] The Tor Project, Inc. Tor Rendezvous Specification, November 23, 2015. `https://gitweb.torproject.org/torspec.git/tree/rend-spec.txt?id=58325d15cd2dba9672fbc94bc56e08856d945796`.

[15] Roger Dingledine and Nick Mathewson. Tor Protocol Specification, August 12, 2015. `https://gitweb.torproject.org/torspec.git/tree/tor-spec.txt?id=5a79d67a45454ab5b7413478702acb93dfa867e2`.

[16] Wikipedia. Zooko's triangle. `http://en.wikipedia.org/wiki/Zooko%27s_triangle` (last visited 16.05.2015).

[17] Howto: Run more than one instance of TorChat, May 2008. `https://github.com/prof7bit/TorChat/blob/torchat_py/torchat/doc/howto_second_instance.html`.

[18] Electronic Frontier Foundation. Secure Massaging Scorecard, January 2015. `https://www.eff.org/secure-messaging-scorecard`.

[19] Elaine Barker and Allen Roginsky. Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths. January 2011. `http://csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf`.

[20] Roger Dingledine. Two relays serving same hidden service. April 2011. `http://archives.seul.org/tor/relays/Apr-2011/msg00022.html`.

[21] Python documentation: 9.6. random — Generate pseudo-random numbers. `https://docs.python.org/2/library/random.html`.

[22] Shallot, July 2012. https://github.com/katmagic/Shallot.

[23] The Tor Project, Inc. Tor directory protocol, version 3, October 22, 2015. https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt?id=35c87554c30bbaaf9641b56bdf4e61a1e636a70b.

[24] Bernd Kreuss. Issue 13: Implement a block list, 2008. https://code.google.com/p/torchat/issues/detail?id=13.

[25] Bernd Kreuss. TorChat Download page in Github, September 2012. https://github.com/prof7bit/TorChat/downloads.

[26] Debin Linux - Package: torchat (0.9.9.553-1.1). https://packages.debian.org/jessie/torchat.

[27] Ubuntu Linux 14.04 Trusty - Package: torchat (0.9.9.553-1). http://packages.ubuntu.com/trusty/web/torchat.

[28] Ubuntu Linux 15.04 Vivid - Package: torchat (0.9.9.553-1.1). http://packages.ubuntu.com/vivid/web/torchat.

[29] Arch Linux - Package Details: torchat 0.9.9.553-2. https://aur.archlinux.org/packages/torchat/.

[30] Bernd Kreuss. TorChat SVN commit r292, December 2010. https://code.google.com/p/torchat/source/detail?r=292.

[31] PyInstaller, March 2015. https://github.com/pyinstaller/pyinstaller/wiki.

[32] Bernd Kreuss. TorChat change log, September 2012. https://github.com/prof7bit/TorChat/blob/torchat_py/torchat/src/changelog.txt.

[33] Roger Dingledine. OpenSSL bug CVE-2014-0160, April 2014. https://blog.torproject.org/blog/openssl-bug-cve-2014-0160.

[34] Tor Binary Too Outdated Again #62 , October 2014. https://github.com/prof7bit/TorChat/issues/62.