6.858 Project: One Time Chat

Jake Barnwell, Andres Perez, Miles Steele

https://github.com/mlsteele/one-time-chat

December 2015

1 Abstract

One Time Chat is a chat service that aims to be information-theoretically secure. Specifically, we propose a chat service that is built on top of a randomly-generated one-time pad. Furthermore, we explore limitations to this model, why one-time pad encryption isn't typically used in practice, and how we aim to resolve these issues.

2 Introduction

Users typically look to public-key cryptography for secure communication. While public-key cryptography is widely used and is sufficiently secure in practice, it is not information-theoretically secure. The assumption behind the security of public-key cryptography is that factorizing integers is thought to be a computationally hard problem. However, this complexity has not been proven, and there are no gaurantees that this presumed one-way function will not be broken in the future.

Fortunately, encryption using a one-time pad has been proven to be completely information-theoretically secure (under the right circumstances). We propose a chat service that is built upon a large[1] one-time pad, allowing two (or more) users to securely communicate with each other with guaranteed confidentiality and integrity.

2.1 Usage of a One-time Pad

Suppose Alice wants to send Bob a message. In order to guarantee to both Alice and Bob that her message m_1 will be securely[2] transmitted to Bob using a one-time pad, they must first agree on a secret pad p_1. After Alice prepares the message, she *encrypts* her message by computing the *ciphertext* $c_1 = m_1 \oplus p_1$, where $\oplus$ is the bit-wise XOR operator. Bob, when he receives the ciphertext c_1, can compute the original message $m_1 = c_1 \oplus p_1$.

This pad p_1 *cannot* be reused, and must be immediately destroyed. Reusing any bits from p_1 breaks the one-time assumption about the pad.

2.2 Drawbacks of the One-time Pad

Though the one-time pads offers, in theory, completely secure communication, care must be taken to implement and use the system correctly. There are three primary issues with one-time pads that must be addressed appropriately to ensure security:

1. The pad must be made up of *random* bits of data. If the pad's bits are not truly random, then an adversary could obtain information about the message by observing the ciphertext. Pseudo-random number generators (PRNGs) are not sufficient for creation of one-time pads, since PRNGs rely heavily or entirely on deterministic functions; even very chaotic functions, like Mersenne twisters, are easily dismantled if an adversary can pinpoint a seed.

2. The users may *never* reuse the same pad bits. Suppose two messages m_1 and m_2 are encrypted using the same pad p, resulting in c_1 and c_2, respectively. Then, we find that

$$c_1 \oplus c_2 = p \oplus m_1 \oplus p \oplus m_2 = m_1 \oplus m_2$$

   Since an adversary knows both c_1 and c_2, she clearly knows $m_1 \oplus m_2$, which means that she has information about both messages. This is not secure.

3. Each time a message is encrypted with a pad, the pad must be the same length as the message. In other words, before two users can communicate with each other, they must have already established a shared pad that is of sufficient length to encrypt all of their intended messages.

[1] In practice, one that will last a lifetime.

[2] that is, in a manner in which no other user will be able to infer the content of the message

Together, these three issues make one-time pads less preferred for encryption than other more popular schemes, even though one-time pads are information-theoretically secure.

3 Design

We believe that much of the inconvenience of using one-time pads can be solved by generating and sharing a pad that is long enough to encrypt all messages that a user could feasibly send in her lifetime. We hypothesize that a pad on the order of 1 terabyte (TB) in size would be enough to encrypt every text message for the rest of her life. [3]

Our design involves two users[4] meeting in person and generating a pad from a local source of randomness, e.g. UNIX's `/dev/random`. We have noticed in practice that this can take a while, but the wait is well worth it, and users could certainly find ways of occupying themselves while the pad is generated. Users then (securely) transfer the generated pad to their local storage device, and part ways.

At this point, the users can store their shared one-time pad on hardware devices and communicate using our chat system. Since both users share the same stream of random bits, they can achieve perfect security under certain threat model assumptions, as long as they synchronize the use of the pad.

3.1 Definitions

We define the following terms:

- **User** Person using the chat service.
- **Client** Command line interface and chat software, typically run on a computer.
- **Device** Separate entity that stores the pad and other data; in our case, a Raspberry Pi or, for proof of concept, emulated on the computer.
- **Server** Message-Relay Server that ferries messages between clients.
- **Pad** Very large one-time pad: shared random bits between two users, and associated metadata. Stored on the device.
- **Pad segment** Some small portion of the pad, used to help encrypt a message.

[3] See Section 5 for details.

[4] "Group chat" (i.e. chat among three or more users) is implemented as a simple extension on top of two-way secure chat (see Section 6.4). To send a message to a "group," the client just interprets it as multiple commands to send a message to single contacts.

- **Secrecy** Inability for an attacker to guess any bits of a message.
- **Integrity** Inability for an attacker to forge or modify a message without the receiver knowing.
- **Security** Catch-all phrase used to declare both secrecy and integrity.
- **RNG** Random number generator. A service that (somehow) generates random numbers.
- **HRNG** Hardware random number generator. An RNG service that uses hardware input to help generate "truly" random data.

3.2 Goals

One Time Chat has been designed to address several specific goals that we have decided are important. We enumerate our goals below:

- **Confidentiality**: Messages sent from user A to user B should only be viewable by users A and B.[5] More strongly, an attacker should not be able to infer any amount of information about the message, except for possibly its length.

- **Integrity**: The receiver of a message should be able to know, with an extremely high probability, if an attacker (e.g. a man in the middle) modifies the message that is received. Furthermore, the receiver must be able to know, with an extremely high probability, that the message originated from who she thinks it did.

- **Security Against Malicious Server**: A malicious chat server should not be able to drop, modify, or replay packets without the receiver knowing.

- **Security Against Malicious Client**: If the user is running the chat client on a malicious computer, it *is* possible that the the malicious computer can eavesdrop (e.g. with a keylogger) on the plaintext message typed by the user ryin, modify the message and/or recipient[s], or drop the message entirely before it is even encrypted or sent over the network. Besides these infractions, the client *must not* be able to modify the encrypted message (once it is encrypted), or be able to read any part of the user's pad whatsoever. Furthermore, any infraction made by the malicious client at this point must not negatively impact the security of future messages sent by the user via non-compromised clients.

[5] Or, in the case of a group message, by exactly the users in the group.

Similarly, the malicious client may read the decrypted message from a received packet, or modify the sender tag, since the message will likely be shown on the screen of the client to the user. Besides this, the client should not be able to read or know any bits of either user's pad, nor negatively impact the security of future messages received by the user via non-compromised clients.

3.3 Non-Goals

- **Availability**: We do not provide any guarantees or solutions for availability of One Time Chat. Though it is very important for a chat service to have a high up-time, we did not consider it necessary to show this proof of concept in our chat system. We are certain that if an adversary, or group of adversaries, tried hard enough, they could DOS our server and deny access to users.

- **Server Security**: The security of the server is not that relevant to this project. There are most likely several bugs and vulnerabilities in the server code, leaving it open to attacks of various kinds. This is not an issue, and indeed aligns with our assumptions that a server may be malicious, malformed, or just not very reliable.

4 Threat Model

We assume that the server has bugs and vulnerabilities, is susceptible to attacks, and is generally untrusted. The server may drop or modify packets of its choosing. While we do *allow* for the possibility of the server dropping or modifying every single packet, we typically assume that it relays a reasonable amount of messages untarnished.[6] We can even assume that the server logs every packet that is sent through it; can store them indefinitely; can distribute them to other users, adversaries, servers, or clients; and replay these packets at a later point to try to get them decrypted, somehow.

We allow for the possibility that certain clients the user uses are untrusted, but not all of them. We recognize that a malicious client can still read a user's incoming or outgoing plaintext messages, since the user types on the client computer and reads messages on the screen. However, when a user is on a malicious client, future and past messages are not vulnerable,[7] only current ones transmitted via the current client.

We assume that the device itself is fully trusted, and is not malicious. Furthermore, the connection between the device and the client is private.

The pad, located on the device, is randomly generated using an RNG service. We assume the pad is kept secure (only the two relevant users have access to it), and that no one else knows anything of the bits of the pad, except for possibly its length.

If the device is physically compromised (e.g. stolen by an adversary) or the secrecy of the pad is broken, we do not guarantee the security of any messages, past, present, or future. This should be obvious: after all, the pad is the secret key between the users.

The RNG service that creates the pad is trusted, and is truly random.[8] It is not pseudo-random. Lastly, we assume there is a way to securely transmit the pad data to the device, if the pad was not generated on the device.

We assume that the correct use of One-time Pads is information-theoretically secure in its secrecy but provides no integrity guarantees of its own.

The SHA256-based HMAC is assumed to be a computationally hard to reverse. If this assumption is broken, our system still preserves secrecy. We are not sure whether our system still guarantees the integrity of messages if HMAC is broken. See Section 5.2 for details.

Besides the one-time pad and SHA256-based HMAC, we do not rely on other encryption schemes. As such, we are allowed to assume that such encryption schemes are broken.

5 Encryption

In order to securely send a message from Alice to Bob, Alice's device encrypts and signs the message, resulting in what we call a package. A package is defined according to the following scheme (where ||

[6] For example, if Bob never receives a message from Alice, it is difficult for Bob to know if the server is dropping all of Alice's packets or if she just never sent him anything.

[7] For example, if a malicious server logged several previous (encrypted) messages while both users were using non-malicious clients, and Alice was now using a malicious client, the server might send those encrypted packets to the malicious client to try to get them decrypted. Even if the server never sent those stored packets to the recipient, One Time Chat has systems in place to protect against this attack.

[8] Even with the HRNG there may still be biases, but treat it to be unpredictable in our analysis.

means concatenation):

$$\begin{aligned} package &\coloneqq index \mid\mid (p_{body} \oplus body) \\ body &\coloneqq ciphertext \mid\mid tag \\ ciphertext &\coloneqq p_{text} \oplus message \\ tag &\coloneqq HMAC(p_{key},\ ciphertext) \end{aligned}$$

The lengths of the components are as follows:

$$\begin{aligned} |message| &= |ciphertext| + |p_{text}| \\ |body| &= |p_{body}| \\ |tag| &= 32\,bytes \\ |p_{key}| &= 16\,bytes \end{aligned}$$

A package is composed of an index, and a body. The index allows the user receiving the package to infer from where in the pad to construct pad segments $p_{text}, p_{body}, p_{key}$. The index also allows flexibility in the system to drop packages, without desyncronizing the use of the pad. The body is our way of encrypting a message and providing integrity to the packet.

Our system requires $2 \cdot |message| + 48$ bytes of pad to encrypt and securely send a message (see Section 5). With the (very generous) assumption that an average text message is 100 bytes (i.e. 100 characters long), if a user had a 1 TB pad, she could could send over 150000 such texts *per day* for the next 80 years.[9]

With modern technology, this is not an unreasonable amount of storage. A cursory search online reveals that a 64 GB USB drive can be found for roughly $20. Anyone who is genuinely concerned about security would likely be willing to spend that much or more for a lifetime guarantee of secure communication.

The rest of this section aims to explain why we think this scheme accomplishes our goals.

5.1 Confidentiality

It is impossible to deduce *message* given

$$(message \oplus p_{text})$$

as long an adversary has not obtained p_{text}, and it is impossible to deduce *body* from

$$(body \oplus p_{text})$$

[9] Alternatively, she could choose to send up to 4 or 5 Vlad the Impaler articles per day.

Since our threat model assumes an adversary does not have access to the pad, it is impossible for an adversary to know what *message* and *body* are. We do not care to provide confidentiality of *index*, because this piece of information does not reveal critical information.

5.1.1 Resistance to Snooping Attack

If an attacker can intercept all network traffic between two clients, she should not be able to recover the original message. We can support this guarantee even under the assumption that the sha256-based HMAC we use for integrity is completely broken.

5.2 Integrity

We argue that *body* provides integrity of the package. An adversary knows where in *package* the *ciphertext* lies (though she doesn't know what *ciphertext* is since it is encrypted with part of p_{body}). Since one-time-pad encryption is malleable, adversaries know that flipping bits in *ciphertext* will flip bits in *message*. For this attack we assume a worst case in which the attacker knows the full plain text of the message and the message is only 1 bit long. However, if an adversary doesn't change *tag*, then with high probability the user would be able to tell that *ciphertext* has been changed, because the modified *ciphertext* would fail to have the same HMAC. We assume, however, that the HMAC we use is hard to fake without a known key.

For the adversary to fool the receiver, she would have to change the tag as well. Because the adversary doesn't know the private key, p_{key}, of the HMAC, we believe an adversary wouldn't be able to to forge the tag. Furthermore we know that if decryption succeeds then the sender is who we think she is because the chances that the pad segments will align is small.

5.2.1 Pad Non-re-use

In any one-time-pad based scheme, it is vitally important that each bit in the pad is only used to encrypt once, ever. Our system defends against pad re-use by having the device manage which bits have been used to encrypt and decrypt during the lifetime of the pad. Because the same pad is stored on two devices, it is critical to make sure that even if the devices cannot communicate, they do not encrypt messages with the same bits as each other. This is why we divide the initial pad in half and dedicate each half for sending from one device (see Section 6.1).

6 Implementation

One Time Chat is implemented using three separately-running python programs. The client software runs on a laptop or desktop which the user places a reasonable amount of trust in. It hosts the chat UI and communicates with the user, pad device, and server. The pad device holds the one time pad data for the user and her contacts, as well as relevant metadata. It communicates through a private, secure channel with the client. The message relay server is an untrusted server used to transfer messages between clients. See Figure 1.

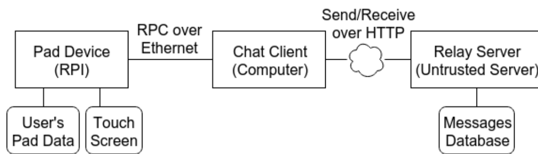

Figure 1: System Diagram

6.1 Pad Structure and Storage

The pad data for a pair of users is stored in two files in the device's file system. Suppose two users, Alice and Bob, have set up a shared pad (see Section 6.2 for details on pad generation) to communicate with each other. Alice's device stores two files: `alice.bob.random.store` and `alice.bob.random.metadata`; Bob has oppositely named files.

The store file (`alice.bob.random.store`) stores the randomly-generated pad bits. Alice and Bob's store files are identical (i.e. `alice.bob.random.store` is equal to `bob.alice.random.store`).

Whenever pad segments are requested to help encrypt/decrypt a message, handler code stored on the device interfaces with the store files to retrieve the appropriate pad bits.

The metadata file `alice.bob.random.metadata` stores certain information about the pad, the contact, and how many bits of the pad have been used for encryption and decryption:

uid User ID of user

rid User ID of contact

store_filename Filename of the random store

metadata_filename Filename of this metadata file

n_bytes Total number of shared pad bytes

rservice Service used to generate the bytes (`random` or `urandom`). In practice this should always be `random`.

split_index Index dividing the two halves of the pad

direction Direction of encryption (1 means encrypt starting at index 0 and moving forward, -1 means encrypt starting at index `n_bytes - 1` and moving backwards).

encrypt_index Index of pad to start encrypting from the next time a pad segment is requested

decrypt_log A log of all decryption requests the device has seen; each decryption request is of the form `i-j` which signifies that pad segment from i to j was requested to decrypt.

decrypt_max Maximum index for a pad segment that has been requested for a decryption. Note that if direction is 1, you *decrypt* backwards (since your contact has direction -1), so this is actually a minimum.

n_eles Number of fields in this metadata, including this one.

checksum Hash to help ensure the metadata wasn't accidentally modified. Not intended to protect against a malicious adversary.

The reason we split the pad into two is so that messages can be sent from Alice to Bob and from Bob to Alice asynchronously: if Alice sent m_A with index i_A to Bob, while at the same time Bob sent message m_B to Alice with index i_B, then if both users were reading from the same portion of the pad, there could be interference: both could, for example, try to use the same portion of the pad to encrypt, which breaks the security of a one-time pad. By splitting up the pad, each user has a dedicated set of bits to send (encrypt) from, and the recipient knows where in the pad to decrypt from when she see an incoming message from the sender.

Alice encrypts from index 0 moving forward in the top half of the pad (the index delimiting two halves is stored by the `split_index` field), and Bob encrypts from index `n_bytes - 1` moving backward in the bottom half the pad. Similarly, when Alice decrypts a message (from Bob), she reads Bob's half of the pad, in reverse; and when Bob decrypts a message, he reads Alice's half of the pad in the forward direction. See Figure 2 for a helpful diagram.

The reason we have the encryption directions moving in opposite directions towards the center of the pad is so that the `split_index` can be appropriately

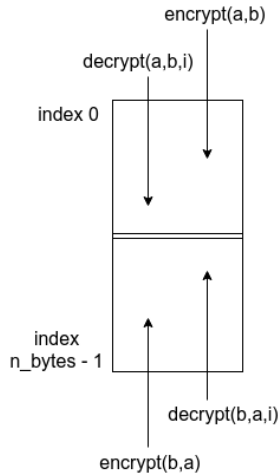

Figure 2: Pad Layout. In this diagram, the function signatures are `encrypt(sender,receiver)` and `decrypt(sender,receiver,index)`. Hence, `encrypt(a,b)` means that `a` is sending to `b` and needs to encrypt, `decrypt(a,b,i)` refers to when `b` receives the message from `a` and wants to decrypt it, starting at index `i`.

modified if, for example, Bob has run out of encryption pad but Alice still has half of hers. However, this feature has not been implemented in the current version.

Obviously, Bob's and Alice's metadata files differ. Each keeps track of messages it has been asked to decrypt, as well as what is the next index of the pad it should use to encrypt.

For example, if Alice has just received a new pad and has requested a pad segment of length 10, the device checks the direction and encrypt index of her metadata (and sees that they are 1 and 0, respectively), and returns `pad[0:10]`, and then updates the encrypt index to 10 so it knows where to start the next time.

When Bob receives the encrypted message (also included is the requested `index`, i.e. a decryption index), he checks his metadata file to see what direction and which half the pad to look at, and then requests the pad segment `pad[decrypt_index:decrypt_index+len(msg)]`. The range of indices is stored into the `decrypt_log`, and `decrypt_max` is updated.

The decryption log and max indexes are stored to help detect dropped packets and replay attacks.

6.2 Pad Generation

We use the `/dev/random` service to generate the random bits of our pads. Because `random` tries to generate bits based on stored entropy, it often blocks while waiting for more entropy, making it very slow to generate more than a few hundred bytes of data. To remedy this situation, we have purchased a hardware RNG (HRNG) device, a USB device that helps seed `random` by generating randomness based on the emissions between two physical diodes. Even then, number generation is very slow for more than several megabytes. However, other more expensive HRNG devices would be much faster.

Note that for development and debugging, we often used `urandom` since it is much faster.

Figure 3 shows an example of generating an (insecure) pad from `urandom`.

```
one-time-chat git:(master)
$ py device/generate.py -m 5 alice bob -s urandom                      [1]

[INFO] Generating 5242880 random bytes with OS service 'urandom'
[INFO] Target files: alice.bob.random.store and bob.alice.random.store

[INFO] Wrote megabyte chunk 1/5
[INFO] Wrote megabyte chunk 2/5
[INFO] Wrote megabyte chunk 3/5
[INFO] Wrote megabyte chunk 4/5
[INFO] Wrote megabyte chunk 5/5

[INFO] Random number generation complete!
[INFO]  - amount random data: 5242880 bytes (5120.0 KiB)
[INFO]  - time elapsed: 5.47410202026 seconds
[INFO]  - throughput: 935.313222342 KiB/s

[INFO] 5242880 random bytes have been written to alice.bob.random.store and bob.alice.random.sto
re
[INFO] metadata for alice.bob.random.store and bob.alice.random.store have been written to alice
.bob.random.metadata and bob.alice.random.metadata, respectively
one-time-chat git:(master)
$                                                                      [0]
```

Figure 3: Generating A Pad

6.3 Device (Pad Device)

The pad device is a device dedicated to storing and mediating access to the one time pad data for a user andher contacts. Each user has one device which she is responsible for keeping safe. A user initializes her device with a shared pad for each contact she wishes to communicate with.

Ideally, a Raspberry Pi would be used to run the device code and connect to the client computer via an ethernet cable. We were able to make this work for awhile, but had so much trouble with it that we chose to emulate the device software on the same computer as the client software for development purposes. While this eliminates the security benefits of having a dedicated device, the proof of concept still stands. The code is still implemented in such a way that it is easy to run it on a separate device. For analysis, we assume that the users are utilizing the separate Raspberry Pi device.

6.3.1 Packaging and Unpackaging

The pad device is responsible for encrypting and decrypting messages. The encryption and decryption methods are in `device/crypto.py` with some tests in `device/test_crypto.py`. See Section 5 for the details of the encryption scheme. The cryptography implementation relies on a simple XOR function and on Python's built-in sha256-based HMAC from the `hmac` module.

The two crypto functions used by the device are `package` and `unpackage` which, when given (as arguments) message data and pad data, secure and unlock messages. For transport convenience, packaged messages (i.e. encrypted and signed messages) are encoded in base64 when being transported between the device and client.

The signatures of the `package` and `unpackage` functions are below:

```
package(index, message,
   p_text, p_body, p_tag_key)
unpackage(package,
   p_text, p_body, p_tag_key)
```

where `p_text`, `p_body`, and `p_key` are different (consecutive) segments from the pad.

6.3.2 Confirmation

A malicious client might get a connection to the pad device. In this case, to ensure that the device doesn't use or release pad bits that it shouldn't, a touchscreen on the Raspberry Pi is used to ask the user whether it is okay to use pad bits to encrypt and decrypt messages; see Figure 4 for an example of this. (The touchscreen and software is controlled by the device so it is assumed to be trustworthy.) This allows the user to have full control of the pad bits that are used in the chat; a user can choose to decline if, for example, the message says that the client is requesting 500 bytes of pad when the user had typed a short message like "hello." This also makes malicious-client-based brute force attacks nearly impossible.

This confirmation dialogue is certainly an inconvenience to the user, but it is a trade-off for security that we think a security-conscious user would be willing to make.

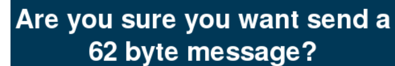

Figure 4: Confirmation dialog on device screen.

6.3.3 RPC Interface

The client communicates with the pad device via an HTTP server running on the device. The server is not accessible to the internet, but is available through the ethernet cable, or as a local service when running on the same machine as the client.

The RPC server is an HTTP server which has a client that can be used as if it were a normal python module from the client. Arguments and return values are sent as JSON. This RPC server currently allows two exposed methods, `package` and `unpackage`, with signatures as follows:

```
package(src_uid, dst_uid, plaintex)
unpackage(src_uid, dst_uid, package)
```

These methods are responsible for asking the user for confirmation, getting pad data from pad storage, feeding it to the crypto functions, and returning the result. Any unexpected errors that occur are hidden from the client so that in the case of a programming error on the device, no sensitive information is leaked through errors.

6.3.4 Logging

The device logs suspicious occurrences in a device-local log file. All log entries have a time-stamp and description of the error as well as additional information about the messages involved.

Any time the message integrity check fails (i.e. due to a mismatched HMAC), the event is logged.

If a message is received which uses a pad index that was used before, this event is logged on the device (in addition to instructing the client to display a warning); See Figure 5 for an example of this.

If a pad index appears to have been skipped, the event is logged (in addition to instructing the client to display a warning).

This log exists on the device and hence cannot be tampered with maliciously. It is inaccessible from anywhere that is not the client. The hope is that a security-conscious user would frequently review this log. Additionally, this log could provide some help in forensics and damage analysis in the event of a known compromise.

```
Cursor: 249


Cursor: 249


Cursor: 249

andres: there is no need to listen to anything i say about the pad secrecy
WARNING: Possible reuse detected. An attacker may be injecting packets.
andres: the hpad is perfectly secure

Cursor: 252


Cursor: 252
```

Figure 5: Reused Pad Warning shown on Client

6.4 Chat Client

The chat client presents a command line interface to the user for sending and receiving messages. It communicates with the device to request encryption and decryption, and exchanges encrypted messages with a relay server. Figure 6 shows a use case of the client.

To start the client a user gives it as parameters the URL of the chat server, the URL of the device RPC channel, and her username.

```
$ client/client.py otcs.dev:9050 9051 alice                                   [0]
Relay server address: http://otcs.dev:9050
Pad   device address: http://localhost:9051
User ID: alice
Welcome to One Time Chat. Type 'help' for help.

Cursor: 90
send bob Did Charlie set up his device yet?
Message sent.

Cursor: 90

bob: yes

Cursor: 259
gs alice,bob Ok. Hey, do you believe in information theoretically secure cryptography?
Message sent.

Cursor: 259

bob: Sure.
charlie: No. That's crazy.

Cursor: 262
help
=== help ===
to send type send [target] [message]
to send to many users type gsend [user1],[user2],...,[userN] [message]
to recieve messages press enter.
to see your user id type id.
to clear the screen type clear.
to quit type quit or q or exit.
============
```

Figure 6: Example Client Use

The messaging-related client commands are listed below.
Send a message to a contact:
send [target] [message]
Send to multiple contacts:
ms user1,user2,... [message] (multisend)
Define a group alias:
group [groupname] user1,user2,...
Send message to users in a group:
gs [groupname] message
Receive messages: Press enter.

6.5 Message Relay Server

The relay server is an untrusted server which relays messages between chat clients. It is implemented as a simple HTTP server with a single table for messages.

The server is not designed to be secure at all, but to be the minimum to demonstrate that the rest of the system could work. It serves over HTTP with no authentication at all. This is consistent with our threat model of not trusting the server, but if One Time Chat were to be effectively used, then a server with additional security measures would be required. The implementation is in `server/server.py` and stores messages for delivery in `server/database.db`.

The server provides the following endpoints for clients:

```
/check: Make sure server is up.
/send: Send a message between users.
  - sender_uid
  - recipient_uid
  - contents
/getmessages: Get messages for a user.
  - recipient_uid
  - start_ref
```

Each message is assigned a reference number which clients keep track of in order to fetch only messages they have not seen before from the server. The latest ref is fetched from the server each time the client starts.

7 Conclusion

At the end of the day, we have implemented our own chat service, and our own cryptography. In other words, we have broken the first rule of cryptography, which is *don't write your own cryptography*. As we iterated through the design of our protocol we learned how subtle details in protocol could lead to compromises in confidentiality and integrity. As it stands, we believe our system is fairly secure under our threat model, and that our approach in sharing a large pad on a secure device is a step towards information-theoretically secure communication.