

# sned – An End-to-End Encrypted File Transfer Service

A Technical Report submitted to the Department of Computer Science


Presented to the Faculty of the School of Engineering and Applied Science  
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree  
Bachelor of Science, School of Engineering

Hamza Mir  
Fall, 2020.

Technical Project Team Members  
Hamza Mir

On my honor as a University Student, I have neither given nor received  
unauthorized aid on this assignment as defined by the Honor Guidelines  
for Thesis-Related Assignments

Signature  \_\_\_\_\_ Date 5/10/2021  
Hamza Mir

Approved \_\_\_\_\_ Date \_\_\_\_\_  
David Wu, Department of Computer Science

# sned

## An end-to-end encrypted file transfer service

Hamza Mir

Department of Computer Science

University of Virginia

Charlottesville, VA USA

hm6ex@virginia.edu

### ABSTRACT

A variety of popular end-to-end encrypted messaging platforms exist, with WhatsApp and Signal being among the most notable. However, there are no ubiquitously used applications that enable end-to-end encrypted file storage or sharing, save for some little-known names such as SendSafely and pCloud. The goal of this project is to create a scalable, end-to-end encrypted file transfer service. The project consists of two components – a client component and a server component. The client either encrypts and requests to send files, or requests to receive and decrypts files that are stored in the server. The server handles client requests to store encrypted files, facilitates the transfer of files between users, and acts as a lookup for mapping usernames to public keys. The server includes multiple computing instances whose traffic is managed by a load balancer.

### 1 Introduction

End-to-end encrypted messaging has seen a surge in usage, due in a large part to the staggering popularity of the WhatsApp messaging application. WhatsApp has reported that it serves over 2 billion users<sup>1</sup>. Additionally, cloud-based file storage and transfer have been successful for years, with big players like Google Drive and Dropbox. However, these services only employ client-server encryption. In contrast to this high popularity of end-to-end encrypted messaging and cloud storage platforms, there are only a few reputable end-to-end-encrypted file transfer and storage solutions.

### 2 Related Works

Despite primarily acting as an instant messaging app, WhatsApp does allow for transferring files between users. WhatsApp uses the Signal protocol, a protocol developed by Open Whisper Systems to enforce end-to-end-encryption on instant messaging platforms<sup>2</sup>. It should be noted that file transfer functionality of WhatsApp is limited, only supporting files no greater than 100 megabytes (MB)<sup>3</sup>. Another disadvantage of the service is that, while it does provide a desktop and web client, its users must use a cell phone number to register<sup>4</sup>.

SendSafely is a service that allows users to transfer to and from one another in an end-to-end encrypted fashion. It works by using OpenPGP to generate two secrets – one generated by a client and one generated by the server. The server sends its secret to the client, but the client does not reveal its secret to the server. These two secrets are combined to create a 256-bit AES encryption key, which is used to encrypt the file the user wishes to transfer. To give another client access to the encrypted file, the sending client sends the other client a link to where the file is hosted and includes the client secret within that link. This link must be sent to the other user out-of-band (via email, for example)<sup>5</sup>.

pCloud is a more general end-to-end encrypted file hosting service, which allows clients to encrypt and upload files to the cloud, which they can then share with other users. pCloud does not disclose in great detail how its system works, but mentions that it uses “industry standard 4096-bit RSA for users’ private keys and 256-bit AES for per-file and per-folder keys”<sup>6</sup>. pCloud also includes features common to many other cloud storage providers, such as file sharing, synchronization, and versioning<sup>7</sup>.

In general, SendSafely and pCloud seem to provide robust services for transferring files securely. However, these products are not well-known by the general public, and it may be possible for a new player to enter this market with a similar service. A feature that appears to be lacking in this space is a convenient command-line interface (CLI) to interact with an end-to-end-encrypted service, which is something this project aims to address. This sort of feature would be particularly useful to software developers and system administrators who work in a command-line often.

### 3 System Design

Two primary components that make up this project – the server component and the client component. The server consists of three subcomponents – an Amazon Web Services (AWS) DynamoDB table, an AWS S3 bucket, and an HTTP webserver.

#### 3.1 DynamoDB Table

This table acts as a NoSQL data store for metadata pertaining to the users registered with the service and requests for transferring files. Each entry in the table is uniquely identified by and indexed

using a composite primary key, made up of a partition (or hash) key and a sort (or range) key. This particular table's partition key is a string which matches "IDENTITY#<username>", "AUTH#<username>", or "TRANSFER#<username>", where "<username>" is the username of one of the registered users on the service. Prepending these "IDENTITY#", "AUTH#", and "TRANSFER#" strings to the username circumvents the need to create three separate tables, which would increase the complexity and cost of the service. The sort key is a numeric string containing the Unix Epoch time in milliseconds. This sort key was chosen to be able to query for transfer requests and get the results in order of ascending datetime.

The table entries whose partition keys begin with "IDENTITY#", "AUTH#", and "TRANSFER#" represent what will henceforth be referred to as identity entries, authenticator entries, and transfer entries, respectively. Each of these entry types has different attributes (other than the primary key) included within them. Identity entries are expected to be created when and only when a new user registers with the service for the first time. The attributes they store (other than the primary key) are two separate RSA public keys of the user – one of these is used for encryption (referred to henceforth as the "encryption private key" and "encryption public key") and the other is used for verifying signatures (known henceforth as the "signing private key" and "signing public key"). These are stored as strings in Privacy-Enhanced Mail (PEM) format. Authenticator entries have only a single attribute – a 32-character alphanumeric string known as an "authenticator". As with identity entries, it is expected that only one authenticator entry for a given username exists at a time, although authenticator entries are frequently deleted and regenerated by the webserver. Transfer entries contain metadata regarding requests to transfer a file from one user to another. The attributes of these entries include a sender, a description, an encrypted AES key, an encrypted AES initialization vector (IV), and a URI. The value of the sender attribute is a string containing the username of the user requesting to transfer a file, and the description is a string composed by the sender to inform the recipient about the nature of the file. The URI is a string generated by the webserver, and represents the key used to access the file the S3 bucket which is associated with the transfer request. The username in the partition key of a transfer entry refers to the intended recipient of the file, not the sender. There can be multiple transfer entries for the same recipient, but their sort key will be different based on when the transfer request was made.

The table does not utilize any of the encryption methods offered by AWS – the entries are stored in plaintext. However, read and write permissions are restricted to an AWS Identity and Access Management (IAM) user whose credentials are accessible to the webserver alone.

### 3.2 S3 Bucket

An S3 bucket is used as the storage location for files uploaded by users. Each file in the bucket can be accessed using a key (known to DynamoDB table transfer entries as the URI), which is an

alphanumeric string generated by the webserver when it makes an upload request to the bucket. Like the DynamoDB table, none of the objects in the bucket are encrypted by AWS, as it is expected that clients have encrypted sensitive files prior to uploading them. Read and write permissions are restricted to the aforementioned IAM user.

### 3.3 HTTP Webserver

The HTTP webserver is the only component of the server that is directly accessible to clients. Clients can make HTTP POST requests using a set of API methods exposed by the webserver. The webserver runs on two separate AWS Elastic Compute Cloud (EC2) instances. A load balancer redirects requests to one of these instances using a round-robin algorithm. The IP address of this load-balancer is what clients directly access, as opposed to the URL of either of the two instances. The load balancer accepts requests over HTTPS, which prevents outside parties from reading metadata sent between clients and the server. Only two EC2 instances are presently active for this project, although more could be instantiated and connected to the load balancer to accommodate additional traffic. The webserver running on each of the instances is a binary executable, and a Unix utility known as "screen" is used to keep these executables running after the terminal session that invoked it has been killed.

The webserver was written using the Rust programming language. Notable libraries that are used in the webserver include *actix-web* (Actix), *rust-openssl*, and *rusoto*. Actix is an asynchronous Rust web framework providing useful abstractions for routing, handling requests, sending responses, and more. *rust-openssl* is a wrapper over the tried and tested OpenSSL library, and is used in the webserver to parse public keys stored in the DynamoDB in order to authenticate requests. *Rusoto* is an AWS SDK for Rust, and is used to retrieve and upload data to and from the DynamoDB table and the S3 bucket.

The webserver exposes six API methods. These are: *register*, *lookup*, *authenticate*, *transfer*, *inbox*, and *download*. These can be reached by setting their name as the path in the URL of the client's request (e.g., sending a request to `https://<load-balancer-ip>/transfer` will invoke the *transfer* method). Additionally, requests made to each of these methods must be POST requests containing JSON payloads whose structure matches schemas specified in the webserver. If this condition is not met, the webserver will respond with an HTTP 404 error.

The *register* method is called by a client to register a new user with the service. The method expects a JSON object containing a username, as well as two strings representing two separate RSA public keys in PEM format. Upon receiving the request, the webserver checks the DynamoDB table to see if an identity entry for the given username already exists. It then attempts to parse the provided PEM strings as RSA public keys to verify that they are valid. If both of these conditions are met, the webserver creates a new identity entry in the DynamoDB table with the given username and PEM strings, then returns an HTTP 200 response.

The *lookup* method is called by clients to retrieve the encryption public key of a specified user. The expected payload consists solely of the username of the user whose encryption public key the client wishes to look up. When executing this method, the webserver checks if the given user exists, and then retrieves the encryption public key in PEM format from the identity entry in the DynamoDB table. If the user exists, it replies to the client with an HTTP 200 response whose body consists of the encryption public key.

The *authenticate* method is used by clients to retrieve an authenticator from the server. Authenticators are 32-character alphanumeric strings that are randomly generated by the webserver and included in authenticator entries in the DynamoDB table. Like *register*, this method only expects a username in its JSON payload. Upon invoking this method, the webserver will verify that the username exists. Then, it retrieves the authenticator from the authenticator entry in the DynamoDB table. If this entry is not present, the webserver will generate an authenticator and create a new authenticator entry in the table. Additionally, if such an entry is present but is older than 6 hours, the webserver will delete this entry, create a new authenticator, and add a new entry with the new authenticator.

The *transfer*, *inbox*, and *download* methods all require the client to authenticate themselves to receive a successful response. This involves including a signed authenticator in the payload of requests to these methods. Clients retrieve authenticators using the *authenticate* method, and then sign them using their signing private key. The webserver verifies signed authenticators by looking up the authenticator of the user that made the request in the associated authenticator entry, looking up the signing public key of that user in the identity entry, and verifying the signed authenticator with that public key. The padding scheme used for this verification is the same one used by the client when it signs authenticators (see Section 3.4). If the verification is successful, then the webserver regards the request as legitimate. The 6-hour-long lifespan of the authenticator ensures that malicious actors do not have time to brute force a signed authenticator, and also prevents them from taking advantage of any compromised signed authenticators for more than 6 hours. Regardless of whether the signed authenticator is compromised, attackers will still be unable to read files unless the recipient's encryption private key is compromised. A disadvantage of this design decision is that there may be a scenario in which a client retrieves an authenticator which is about to expire, and sends a signed authenticator to the webserver for a now expired authenticator. The server will reply with an HTTP 401 (Unauthorized) response. However, the client can easily rectify this by simply requesting a new authenticator (by using the *authenticate* method) and resending the request.

The *transfer* method called to request a file transfer from one user to another. The expected payload includes the sender's username, the recipient's username, a description of the file contents, a signed authenticator, an encrypted AES key, an encrypted AES IV, and the file itself. Since JSON values cannot include raw bytes, the file must be encoded in ASCII or UTF-8. Therefore,

raw PNG image data, for example, can't be included in the payload. To circumvent this, the client encodes the file data in base64 format before uploading it. When the method is invoked, the webserver checks if the sender username exists, and then verifies that the signed authenticator is valid. It then checks if the recipient username exists. If these conditions are met, the server generates a random URI for the file and creates a transfer entry in the DynamoDB table with the URI and the information provided in the payload. Then, the file is uploaded to the S3 bucket using the URI as the key.

The *inbox* method returns pending transfer requests for a given user. The expected payload includes the requesting user's username and a signed authenticator. When invoked, the webserver checks if the given username exists and verifies that the signed authenticator is valid. Then, it retrieves all the transfer requests in which the specified user is the recipient. This is done by querying the DynamoDB table for all entries whose partition key is equal to "TRANSFER#<username>", where "<username>" is the username specified in the payload. It is guaranteed that these entries will be returned from the table in ascending order of datetime, since a Unix epoch timestamp is used as the sort key. The results of this query are serialized into JSON object containing an array of JSON "InboxItem" objects. Each InboxItem includes the timestamp, sender, description, encrypted AES key, and encrypted AES IV of one of the transfer entries. They are ordered by timestamp. This JSON object is used as the body of the webserver's response. If there are no transfer requests for the given recipient, a JSON object containing an empty array will be sent as the response.

The *download* method allows clients to download files that other users have requested to transfer to them. The expected payload includes the username of the user requesting to download the file, a signed authenticator, and an ID. The ID is a non-negative integer representing the index of one of the user's pending transfer request. An ID of 0 represents the earliest pending transfer request for the user, an ID of 1 represents the second earliest, etc. Users can determine the order of pending transfer requests, and thus their IDs, by calling the webserver's *inbox* method. When the method is called, the webserver checks that the user exists, the signed authenticator is valid, and the provided ID is valid. As with the *inbox* method, the webserver queries the DynamoDB table for all entries whose partition key is equal to "TRANSFER#<username>". It retrieves the URI of the entry whose index is equal to the ID supplied by the user. It then uses this URI to download the associated file from the S3 bucket, and generates a response with the raw file data as its body. Finally, the webserver deletes the transfer entry from the DynamoDB table, deletes the file from the S3 bucket, and sends a response (which includes the file) to the client.

### 3.4 Client

The sned client is also written in Rust, and makes use of the *rust-openssl* library to encrypt and decrypt using RSA and AES. The client is an executable binary that acts as a CLI to simplify

making properly formatted requests to the webserver. There are five commands exposed by the CLI – *register*, *lookup*, *inbox*, *transfer*, and *download*. Each of these commands accepts different flags and arguments, and each will invoke the webserver methods of the same name.

The *register* command accepts a username as its argument. When it is invoked, the client generates two new 4096-bit RSA key pairs – one keypair for encryption/decryption and another for signing/verifying signatures. It then invokes the *register* method on the webserver using the user-provided username string and the newly generated public keys. If the server responds with a successful status code, then the public keys, private keys, and username are each written to separate files in the same directory as the executable. If any command (other than *register* or *lookup*) is called, and these files are not present, the command will fail. Additionally, if these files are already present when the *register* command is run, it will fail.

The *lookup* command takes a username as an argument. It then invokes the *lookup* method on the webserver and prints the returned encryption public key to standard output (stdout) if the given username exists. This command was included to provide an out-of-band mechanism for users to verify that the server is behaving properly. If a malicious actor were to gain access to the server, they could lie about what public keys are associated with each user, instead providing different public keys whose corresponding private keys they have access to. Then, when users make transfer requests using these false public keys, the malicious actor controlling the server will be able to intercept and decrypt the files. The *lookup* command allows users to verify that this is not happening. To do this, they can ask another user to send them their encryption public key (through some medium other than the sned webserver) and diff it with the output generated by *lookup*. If the keys do not match, it is possible the server has been compromised.

The *inbox* command accepts no flags or arguments. It calls the webserver's *authenticate* method to retrieve an authenticator, which it then signs using its signing private key. The padding scheme used when signing the authenticator is the Probabilistic Signature Scheme (PSS) defined in RFC 2437<sup>8</sup>. SHA-256 is used for the digest and masking function of the scheme. The client then invokes the *inbox* method of the server with the signed authenticator as its payload, and prints the response (a list of *InboxItems*) as a neatly formatted table to stdout. The table contains four columns – ID, timestamp, sender, and description. The AES key and IV are omitted from the table since viewing them is of little use to the user.

The *transfer* command accepts a *-p* (or *--plaintext*) flag, and expects three arguments – a recipient, a description, and a file path. When the command runs, the client reads the file specified by the file path into memory, and then generates a random 32-byte key and 16-byte IV to create a 256-bit AES encryptor. If the plaintext flag is not set, the file data is encrypted using the AES encryptor. Then, the encryption public key of the recipient is

retrieved by calling the *lookup* method on the webserver. This key is used to encrypt the AES key and IV. The padding used for this encryption is the Optimal Asymmetric Encryption Padding (OAEP) scheme defined in RFC 2437. This provides stronger security than the encryption padding scheme defined in PKCS #1 v1.5, in part by using random seeds to create different ciphertexts for the same plaintext<sup>9</sup>. The client then encodes the file data as base64 so that it is in a format that can be accommodated by JSON. The client calls the *authenticate* webserver method to retrieve an authenticator, which it signs using its signing private key to generate a signed authenticator. The client's username, the recipient's username, the description, the signed authenticator, the encrypted AES key, the encrypted AES IV, and the base64-encoded file data are serialized into a JSON string and included in the request to the webserver's *transfer* method. By encrypting the AES key and IV (which are used to encrypt the file) with the recipient's encryption public key, it is guaranteed that only the recipient will be able to decrypt the AES key/IV and thus decrypt the file. Using a symmetric key like AES to encrypt files (as opposed to RSA) allows for significantly faster encryption/decryption, especially for large files.

The *download* command takes two arguments – an ID and a file path. It also accepts a *-p/--plaintext* flag. When run, this command invokes the *inbox* method of the webserver to retrieve the encrypted AES key and IV of the *InboxItem* with the given ID. The client then calls the *authenticate* method to retrieve an authenticator, which it signs, and invokes the *download* method of the webserver using the ID, the client's username, and the signed authenticator as its payload. Once the file has been downloaded, it is decoded from base64 (recall that all uploaded files are in base64 format) to a byte vector. If the plaintext flag is not set, the encrypted AES key and IV are decrypted using the client's encryption private key using the OAEP padding scheme. With the now decrypted AES key and IV, the file data is decrypted, and written to the file located at the specified file path.

## 4 Procedure

The example provided here shows how a user, named Alice, would transfer a file to another user, named Bob. First Alice has to generate an RSA key-pair and register herself with the service:

```
alice$ ./sned register alice
```

And so does Bob:

```
bob$ ./sned register bob
```

Note that “./sned” is the executable. If Alice wants to transfer a file to Bob, she can use the *transfer* command, supplying Bob's username, a description, and a file path as arguments.

```
alice$ ./sned transfer bob "my file" file.txt
```

If Bob wants to check if he has any pending transfer requests, he can call the *inbox* command.

```
bob$ ./sned inbox
```

Assuming Alice is Bob’s only correspondent, this command will output the following table:

ID	TIMESTAMP	SENDER	DESCRIPTION
0	1605566380697	alice	my file

**Figure 1: Table output when Bob runs the *inbox* command**

Bob sees that Alice’s transfer request has an ID of 0, and uses that information to download Alice’s file, also specifying a file path for the output file.

```
bob$ ./sned download 0 output_file.txt
```

The original contents of “file.txt” from Alice’s machine will be written to “output.txt” on Bob’s machine.

## 5. Results

The system was tested by the running commands listed above in the “Procedure” section. This resulted in the expected outcome – files were encrypted and uploaded to the server by Alice, and downloaded and decrypted by Bob. Requests which failed to include valid signed authenticators were rejected by the server, as expected. The service was also tested with more than two registered users and more than two file transfer requests. All commands and API methods functioned correctly under these conditions. The performance of sned was compared to that of WhatsApp. The entire process of encrypting, encoding, and uploading a 70MB file took sned 41 seconds to complete. Downloading, decoding, and decrypting it took 28 seconds. WhatsApp took 45 seconds to upload that same file, and 15 seconds to download it. This was also tested for a 140MB file. It took sned 1 minute and 40 seconds to transfer the file to the server, and 53 seconds to download it. This file exceeded WhatsApp’s file size limit, so it could not be uploaded.

## 6. Conclusions

The sned system was successfully designed to provide an end-to-end encrypted file sharing service. Users with basic familiarity of Unix are able to interact with the server by using a convenient CLI which abstracts away the encryption/decryption of files, the encoding/decoding of files, and the creation of HTTP requests to interact with the server. Users that frequently use a terminal will likely benefit the most from this service, since they can interact with it in an ecosystem that they are familiar with and use often. The system uses 4096-bit RSA and 256-bit AES to ensure files cannot be accessed by unauthenticated parties. The performance of sned is comparable to WhatsApp, the most popular end-to-end encrypted messaging service. It is also capable of transferring files larger than WhatsApp allows.

## 6. Future Work

Various improvements could be made to increase the ease-of-use and utility of sned. The service could benefit from including a graphical user interface (GUI) version of its client. This would entice more laypeople to use the product, eliminating the need for users to be comfortable with a command-line. Additionally, the project could be expanded to allow for file storage in addition to file transfer. That is, users could be allowed to store files indefinitely and give access to or revoke access from other users. Finally, JSON payloads for requests to the server could be replaced with binary payloads by using a protocol like ProtoBuf. This would increase performance by eliminating the requirement for encoding file data as base64 prior to uploading it. This step not only takes time, but inflates the original file.

## REFERENCES

- [1] Jon Porter. Feb 12, 2020. WhatsApp now has 2 billion users. *The Verge*. <https://www.theverge.com/2020/2/12/21134652/whatsapp-2-billion-monthly-active-users-encryption-facebook>
- [2] Jon Evans. Nov 18, 2014. WhatsApp Partners With Open Whisper Systems To End-To-End Encrypt Billions Of Messages A Day. *TechCrunch*. <https://techcrunch.com/2014/11/18/end-to-end-for-everyone/>
- [3] WhatsApp: How to share more than 100MB files? Jul 30, 2020. *BGR*. <https://www.bgr.in/how-to/whatsapp-how-to-share-more-than-100mb-files-on-the-messaging-app-905827/>
- [4] Using one WhatsApp account on multiple phones, or with multiple phone numbers. *WhatsApp*. <https://faq.whatsapp.com/general/verification/using-one-whatsapp-account-on-multiple-phones-or-with-multiple-phone-numbers?category=5245245>
- [5] Security Overview. *SendSafely*. <https://www.sendsafely.com/security/>
- [6] Encryption. *pCloud*. <https://www.pcloud.com/features/crypto.html>
- [7] pCloud. *pCloud*. <https://www.pcloud.com/>
- [8] B. Kaliski & J. Staddon. October 1998. PKCS #1: RSA Cryptography Specifications Version 2.0. *IETF*. <https://tools.ietf.org/html/rfc2437>
- [9] RSA signature and encryption schemes: RSA-PSS and RSA-OAEP. *CryptoSys*. [https://www.cryptosys.net/pki/manpki/pki\\_rsaschemes.html](https://www.cryptosys.net/pki/manpki/pki_rsaschemes.html)