EncryptedSend
A web browser based end-to-end encrypted messaging system

Meixler Technologies, Inc.
October 23, 2019

Abstract
EncryptedSend is a service that allows users to send and receive sensitive messages, files, and documents securely, with end-to-end encryption, using just their web browsers. Encryption takes place in the sender's web browser, and decryption takes place in the recipient's web browser, via client-side javascript running in both users' web browsers. Only encrypted information is sent through EncryptedSend's servers. Unencrypted information and the keys used to encrypt/decrypt this information never leave the sender's web browser or the recipient's web browser. Senders may send information to a recipient through the service, without registering for the service, simply by knowing the recipient's address or URL on the service.

EncryptedSend is operated by Meixler Technologies, Inc. at the URL https://www.encryptedsend.com/.

Introduction
It is common for individuals to have a need to exchange information in a secure and confidential manner. For example, accountants, attorneys, and medical professionals frequently have a need to send and receive confidential information to and from their clients. Because standard SMTP email does not provide a means for encrypting information from the sender's end to the recipient's end of an email transmission, users have sought solutions to use instead of SMTP email, or in conjunction with SMTP email, for exchanging information confidentially and securely.

In 1991, 'Pretty Good Privacy' (PGP) was released, followed later by an open-source compatible program by the name of 'GNU Privacy Guard' (GPG). These programs implement public key encryption using an asymmetric encryption algorithm such as RSA. Using these programs, a sender can encrypt a message for a recipient using the recipient's public key, then send the message to the recipient via SMTP email or via some other transport method. These solutions provide end-to-end encryption, however they require both the sender and the recipient to install the program. These programs also require the user to have a degree of technical know-how and a basic understanding of cryptography.

As the world wide web evolved, cloud-based file sharing services (such as those offered by Citrix® and Dropbox®) have emerged, which claim to allow users to share files securely. These services typically allow users to share files via the following procedure: First, the service allows a sender to upload a file using their web browser to the service's server through an SSL/TLS connection. The SSL/TLS connection ensures that the file is encrypted while in transit from the sender's web browser to the service's server. Then, upon receiving the file at the service's server, the file is immediately encrypted by the service using a symmetric encryption algorithm such as AES, so that the file is encrypted while at rest while stored on the service's server. The AES encryption key is also stored by the service. Later, when the file is requested by the recipient, the service uses the encryption key to decrypt the file. Then, the service allows the recipient to download the file using their web browser from the service's server through an SSL/TLS connection. The SSL/TLS connection ensures that the file is encrypted while in transit from the service's server to the recipient's web browser. However, astute observers will quickly notice that this is not an end-to-end encryption solution. There are points in time when the file is unencrypted while in the service's possession – specifically immediately after the sender uploads the file, and immediately before the recipient downloads the file. Moreover, even though files sent through the service are encrypted while stored by the service, the service has the ability to decrypt these files at any time, because the service also has possession of the encryption keys. Notwithstanding, these services are sometimes seen as more convenient than other solutions, because they can be accessed using a web browser, eliminating the need for the sender and/or recipient to install special software.

More recently, encrypted messaging apps have emerged. Like PGP and GPG, these apps implement true end-to-end security. However, both the sender and recipient must have the these apps installed on their devices in order to share messages and files through this solution, and these solutions are not web browser based.

EncryptedSend aims to provide service that enables users to share files using a web browser, with true end-to-end encryption. In addition, senders are able to send files and messages to recipients without the need for senders to be setup on the service, simply by knowing a unique username, address or URL for a recipient.

Summary of EncryptedSend's Solution

To meet the needs described above, EncryptedSend utilizes a sender's web browser, a recipient's web browser, and a web server. Client-side scripting running within the sender's web browser is used to encrypt messages at the sender's end, and client-side scripting running within the recipient's web browser is used to decrypt messages at the recipient's end. Encrypted messages from a sender are relayed through the server to a recipient.

As such, this solution provides true end-to-end encryption from the sender's device to the recipient's device. Only encrypted messages reach the server. Unencrypted messages are never disclosed to the server, and secret or private encryption keys are never disclosed to the server. The solution is accessed by the sender and the recipient via their web browsers, and does not require senders or recipients to install special software or programs on their devices. Senders can send messages to recipients without being setup with the service, simply by knowing the recipient's unique URL for receiving messages through the service.

Detailed Description of EncryptedSend's Solution

EncryptedSend utilizes the following cryptographic primitives: Elliptic Curve Diffie–Hellman key agreement protocol (ECDHE), Elliptic Curve Digital Signature Algorithm (ECDSA), AES-GCM symmetric encryption, the SHA256 hashing algorithm, and the PBKDF2 password-based key derivation function. All client-side cryptography is implemented using the Web Crypto API (https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API).

To use the service (e.g. to register for the service, or to send an encrypted message through the service, or to receive encrypted messages through the service), a user must first point their web browser to the URL for the EncryptedSend service, https://www.encryptedsend.com/. Upon doing so, the user's browser downloads several .html, .css, and .js from the web server. These files define a single persistent web page which remains loaded in the user's web browser throughout the duration of the user's session with the service. The .html and .css files contain code that is used to present the user interface to the user to allow the user to interact with the service. The .js files contain client-side scripting that runs within the user's web browser and performs three functions: 1) the client-side scripting performs all client-side cryptography operations. 2) the client-side scripting interacts with the server by way of an API hosted by the server. 3) the client-side scripting manages the user interface by displaying and hiding various elements of the user interface as needed, in order to present the user with the elements needed for the operation that the user is performing at any given time.

Appendix 1a shows the process of sending an encrypted message to a recipient through the system. To send a message to a recipient, the sender first points their web browser to a unique URL for the recipient to begin a session with the system. Upon doing so, the user's browser downloads the .html, .css, and .js files which define the single persistent web page which remains loaded in the user's web browser throughout the duration of the user's session with the service. Client-side scripting running within the user's web browser presents the form shown in Appendix 5, for composing a message. In addition, the recipient's certificate is downloaded from the server to the sender's web browser through the API. As shown in Appendix 4b, the recipient's certificate contains the recipient's identifying information and the recipient's public ECDH key, and is digitally signed by both the recipient and the server. These signatures are verified by client-side scripting running within the sender's web browser. The recipient's signature is verified using the recipient's public key contained in the certificate, and the server's signature is verified using the server's public key hardcoded in the client side scripting. After composing a message to the recipient (optionally including one or more attached files) the client-side scripting running within the sender's web browser then uses ECDHE to derive a symmetric encryption key, based on the recipient's public key. The message is then encrypted using the derived key, then the encrypted message is uploaded to the server by the sender's web browser through the API.

Appendix 1b shows the process of receiving an encrypted message from a sender through the system. To retrieve a message, the recipient first points their web browser to a URL for the service to begin a session with the system. Upon doing so, the user's browser downloads the .html, .css, and .js files which define the single persistent web page which remains loaded in the user's web browser throughout the duration of the user's session with the service. Client-side scripting running within the user's web browser presents the form shown in Appendix 7, to allow the user to authenticate with the system. The recipient then provides his/her secret key file and password. The format of the user's secret key file can be seen in Appendix 4a. At this point, the client-side scripting derives a decryption key from the password using PBKDF2 with the number of rounds specified in the secret key file. The user's encrypted ECDH private key is then extracted from the secret key file, and the decryption key derived from the password is then used to decrypt the user's ECDH private key. The private key is stored in a Web Crypto API CryptoKey object, with the .extractable property set to false. This enables the user's private key to be used to sign messages or derive symmetric keys, but prevents the private key from being read for the remainder of the user's session. Client-side scripting running within the user's web browser then performs a query to the server through an API to request user account information for the recipient. An authentication process then commences between the web browser and the server, via the API, to ensure that the recipient is in possession of the private key corresponding to recipient's public key sent in the request. This process is carried out by way of a cryptographic process that enables the server to verify that the recipient is in possession of the private key, without the need for the recipient to disclose the private key to the server (this process is explained in more detail later in this section, and is shown in Appendix 2a-2e). Upon successful authentication, account information for the recipient is returned by the API, and this information is displayed for the user by the client side scripting running within the user's web browser. The user can then request to view a log of messages received. Client-side scripting running within the user's web browser then performs a query to the server through an API to request a log of messages received. An authentication process similar to the one described above commences. Upon successful authentication, metadata for each message is returned by the API, along with the encrypted subject, message text, and list of attachments for each message. Client-side scripting running within the recipient's web browser then decrypts the subject and message text of each message using the recipient's private key to derive the symmetric encryption key used by the sender to encrypt these parts. Then, metadata for each message is displayed, along with the decrypted subject, message text, and list of attachments for each message.

At this point, the user can request a file attached to a message. Client-side scripting running within the recipient's web browser then requests the encrypted attachment from the server via the API. An authentication process similar to the one described above commences. Upon successful authentication, the requested encrypted message part is returned by the API, and client-side running within the recipient's web browser decrypts the file and makes the file available to the recipient to open or save.

Appendixes 2a-2d show the architecture of the API hosted by the server. Client-side scripting running within users' web browsers accesses the API through an SSL/TLS connection over HTTPS, using the browser's built-in XMLHttpRequest request (XHR) object. The API allows client web browsers to send requests and commands to the server needed to support system functions such as registering for the service, accessing certificates for message recipients, enabling senders to upload encrypted messages for recipients, accessing logs of messages sent and received, enabling recipients to download encrypted messages that they've received, etc.

Each API operation consists of two phases, where each phase consists of an HTTPS round trip between the browser and the server. Appendix 2a shows phase 1 of an API operation. Phase 1 is the same for all API functions. At the conclusion of phase 1, the client and the server each have a mutually agreed upon ephemeral shared secret and exchange hash. The mutually agreed upon ephemeral shared secret and exchange hash derived in phase 1 are used in phase 2, where the client sends a request or command to the server, and the server returns a response. The procedure used in phase 2 depends on the nature of the request or command sent by the client to the server in phase 2. Appendixes 2b, 2c, and 2d show phase 2 of an API operation for various API functions. As can be seen in Appendixes 2a, 2b, 2c, and 2d, the API mutually authenticates the client and the server during each API operation, using a cryptographic schema involving the permanent keys for the client and the server, the ephemeral keys generated by the client and the server in phase 1, the shared secret derived by the client and the server in phase 1, and the exchange hash derived by the client and the server in phase 1. The ephemeral shared secret is derived from the client and server ephemeral keys in phase 1 using ECDHE. The exchange hash is derived in phase 1 using a SHA256 hash function over the client permanent public key, the server permanent public key, the client ephemeral public key, the server ephemeral public key, and the ephemeral shared secret. Digital signatures over the exchange hash by both the client and the server, each using their permanent private ECDH keys with ECDSA, are used to assure each party that the other is in possession of the permanent private key corresponding to the permanent public key provided in phase 1, without the need for either party to disclose their permanent private key to the other. Thus, the client is able to verify that the server is in possession of the permanent private ECDH key corresponding to a known pinned public key for the server in phase 1, and the server is able to verify that the client is in possession of the permanent private ECDH key corresponding to public key on record for that client in phase 2. The server uses memory to store the shared secret and exchange hash (indexed by the client's permanent public key and ephemeral public key provided by the client in phase 1) for the short duration between the two phases comprising an API request. However, no state information is stored on the server after the completion of an API request. Some API functions (such as retrieving messages for a recipient) require the client to authenticate, while others (such as registering for the service) do not require the client to authenticate. For operations that require the client to authenticate, the user is identified at this point by the permanent public ECDH key presented in the request, so that the user can be authenticated. For operations that do not require the client to authenticate, the same process is used, except that the client may use a randomly generated ECDH key pair in place of the permanent key pair, and the server does not attempt to identify or authenticate the client for these operations. In addition to the authentication function performed by the API, all requests and responses sent between client web browsers and the server through the API during phase 2 are encrypted using AES-GCM with the ephemeral shared secret derived from the client and server ephemeral keys in phase 1. This ensures secrecy, authenticity, and integrity of requests and responses between client web browsers and the server during phase 2. Therefore, all communications between client web browsers and the server are 'double encrypted', first by the API, then secondly by the SSL/TLS channel. In addition, perfect forward secrecy is guaranteed, because the ephemeral shared secret used for AES-GCM by the API in phase 2 is derived from the ephemeral keys. The schema used by the API is also resistant to replay attacks.

Appendix 2e shows a list of the functions supported by the API. Functions that execute queries are generally done using HTTPS GET requests in phase 2, and results of queries are returned in a JSON formatted response, as shown in Appendix 2b. Functions that retrieve parts of messages are generally done using HTTPS GET requests in phase 2, and encrypted truncated message payloads are returned as binary data, as shown in Appendix 2c. Functions that execute commands are generally done using HTTPS POST requests in phase 2, and results of commands are returned in a JSON formatted response, as shown in Appendix 2d. For both phase 1 and phase 2 of an API operation, the server returns a 2xx status code in the HTTPS response header to indicate success, and a 4xx status code in the HTTPS response header to indicate a failure. These same status codes are included in the JSON response as well for phase 1 and phase 2 in cases where a JSON response is returned, along with more detailed status messages, as can be seen in Appendix 2b and 2d.

In order to receive messages through the system, a user must first register for the service and complete a setup process. Appendixes 3a-3c show the steps involved in the setup process. To register for the service, the user first points their web browser to a URL on the server where the service is hosted. Upon doing so, the user's browser downloads the .html, .css, and .js files which define the single persistent web page which remains loaded in the user's web browser throughout the duration of the user's session with the service. Upon selecting an option to register for the service, client-side scripting running within the user's web browser presents the form shown in Appendix 3a. After providing their name, company (optionally), and email address in the form in Appendix 3a, the user then proceeds to the form in Appendix 3b where they enter a passphrase. At this point, client-side scripting running within the user's web browser then generates a random ECDH key pair for the user using the P-256 elliptical curve. Next, a PBKDF2 function with one million iterations of SHA256 hashing is used to derive a 256-bit AES-GCM encryption key from the user's passphrase. This key is then used with an AES-GCM function to encrypt the user's ECDH private key. The encrypted ECDH private key is stored in the secret key file, along with the supporting information shown in Appendix 4a.

In addition to the secret key file being generated for the user, a certificate is also generated for the user. As shown in Appendix 4b, the certificate is created in JSON format, and contains the user's first name, last name, company, email address, creation date, verision information and ECDH public key. The certificate is then self-signed by the user, using the user's ECDH private key with ECDSA. This signature can later be verified to prove that the user is in possession of the private key corresponding to the public key contained in the certficate.

The user is then prompted to save the secret key file to their system, as shown in Appendix 3c. After saving the secret key file, the user's web browser executes a 'register1' API call to the server, passing the user's certificate that was created in the previous step. The server then verifies that the public key contained in the certificate is valid, that the creation date is within a given tolerance of the current date and time, and that the user’s self-signed signature is valid given the user’s public key contained in the signature. If all checks pass, the server then sends an email to the user containing a verification URL. The verification URL contains the user’s certificate passed in the ‘register1’ API call, encrypted using AES-GCM with a key known only to the server. The user is then presented with a message, notifying them that a verification email has been sent to the address that they provided. The user then receives a verification email, containing the verification URL. Upon accessing the verification URL, the user’s web browser loads client-side scripting which then executes a ‘register2’ API call to the server, passing the AES-GCM encrypted certificate contained in the verification URL sent in the verification email previously by the ‘register1’ operation. The server then attempts to decrypt the encrypted certificate. If successful, the server then performs the same set of verification steps described above during the response to the ‘register1’ API call. If decryption was successful and all verification checks pass, then this ensures that the individual claiming to have the name and company affiliation appearing in the certificate is in possession of the private key corresponding to the public key in the certificate, and was able to access a verification URL sent by email from EncryptedSend to the email address in the certificate. At this point, the server signs the certificate using the server’s ECDH private key with ECDSA, and this signature is appended to the certificate. Then, a database record is created for the user, containing information from the certificate, and the certificate itself. The user is then presented with a message advising that the setup process is complete and that the account is active. The user is also provided with a URL that they can then provide to their senders, where senders can access a form to send encrypted messages to the user. The user also receives an email from the system containing the same information. Going forward, the user’s encrypted private ECDH key contained in the secret key file that the user created during the setup process is used to authenticate the user with the system, and is used to decrypt messages that the user receives through the system.

As noted previously in this section, Appendix 5 shows a form that is used by a sender to send an encrypted message to a recipient, and the sending process is shown in Appendix 1a. After composing a message to the recipient (optionally containing one or more attached files) client-side scripting running within the user’s web browser creates a payload file containing the encrypted message, as per the format shown in Appendix 6, using the procedure in the following paragraph.

As shown in Appendix 6, the payload file consists of four segments: metadata, the parts data cipher, the signed header, and the payload cipher bytes. To create the payload file, first ECDHE is used to derive a 256-bit symmetric AES-GCM encryption key based on the sender’s private ECDH key and the recipient’s public ECDH key from the certificate. If the sender has not authenticated with the system, then a random ECDH key pair is generated for the sender to use for the ECDHE process. Next, the first 144 bytes of metadata is generated, including the sender and recipient X and Y public ECDH key values. Next, the parts data is collected, as shown on the left of Appendix 6. Each message consists of N parts. The subject of the message is indexed as part 0, the text of the message is indexed as part 1, then any attached files are indexed from part 2 to part N-1. The parts data consists of the name, mime type, SHA256 hash of each part. Each part is encrypted using AES-GCM with the derived key. Next, the parts data cipher is generated using AES-GCM with the derived key to encrypt the parts data. The length (in bytes) of the parts data cipher is then appended to the metadata, followed by 64 (the length, in bytes, of an ECDSA signature). Next, the length (in bytes) of each encrypted message part is appended to the metadata, followed by four zero bytes to mark the end of the metadata. Next, the signed header is generated using ECDSA to create a digital signature over the metadata and the parts data, using the sender’s private key. The payload file is then constructed as shown in Appendix 6 by appending the metadata, the parts data cipher, the signed header, and each encrypted message part. This payload file is then uploaded to the server using the ‘postmessage’ API call. The payload file is then stored on the server. The server then parses the metadata of the payload file to capture the public keys of the recipient and sender, and uses these keys to identify the account records of the recipient and sender (if possible). A record for the message is stored in the database consisting of a unique identifier for the message, a pointer to the recipient, and pointer to the sender (if known), the message size (in bytes), and the timestamp. Finally, the server (optionally) notifies the recipient that they've received a message through the system.

As noted previously in this section, Appendix 7 shows a form that can be used by a registered user to begin a session with the system. After successfully authenticating, the user's private key is stored securely in a Web Crypto API CryptoKey object, and is used to commence API calls to the server such as those shown in Appendix 2e via the procedures shown in Appendixes 2a, 2b, 2c, and 2d, in order to facilitate further operations. At this point, client-side scripting in the user's browser executes a 'getaccountrecord' API call to the server, and the information returned by the API is used to present the user with general account information. The user is also presented with options to view account information, views logs of messages sent and received, and send a message.

After beginning a session with the system as described above, and selecting the option to view received messages, client-side scripting running within the user's web browser executes a 'getmessages' API call to the server, with the string 'received' as the input parameter. The server queries the database to get the message id, sender's name (if known) and timestamp of each message, and this information is returned by the API, and used to populate the log of messages received. The process is similar for viewing a log of messages sent, except the string 'sent' (instead of 'received') is used as the input parameter to the 'getmessages' API call to the server.

Some of the pieces of information shown in the logs of messages presented to the user are not returned by the 'getmessages' API call. For example, this is the case with the subject of each message, the text of each message, and the names of any files attached to each message. These pieces of information come from the encrypted message payloads, not the 'getmessages' API call. To present these pieces of information, client-side scripting running within the user's web browser executes a 'getmessagepart' API call to the server for each message appearing in the log, passing the message id and the index of the requested part number as inputs. Using the metadata in the encrypted payload file of the message, the server parses the encrypted payload file to determine the indexes of the starting and ending bytes of each encrypted part in the encrypted payload file. The 'getmessagepart' API call returns a truncated encrypted payload file, including the metadata, the parts data cipher, the signed header, and the payload cipher bytes for parts 0 and 1, and the payload cipher bytes for any part requested with index 2 or greater, without the payload cipher bytes for any parts that were not requested with index 2 or greater. Client-side scripting running within the user's web browser then derives the symmetric AES-GCM key used to encrypt the encrypted parts of the truncated encrypted payload file, using ECDHE with the recipient's private ECDH key and the sender's public ECDH key contained in the metadata of the truncated encrypted payload file. Using the derived key, client-side scripting running within the user's web browser then decrypts the requested part from truncated encrypted payload file, using the derived key, through the inverse of the process used to create the encrypted payload file. This decrypted information is then used to populate the subject and text for each message shown in the message logs, and is also used to show a list of any files attached to the message.

If a user requests a file attached to a message, client-side scripting running within the user's web browser executes a 'getmessagepart' API call to the server, passing the message id and the index of the attachment. This truncated encrypted payload file returned by the 'getmessagepart' API call, containing the encrypted file requested by the user, is then decrypted by client-side scripting running within the user’s web browser, using the same process described in the previous paragraph, and the user is given the option to open or save the file.

Security Concerns

The first security concern that will come to the mind of most readers is the potential for a malicious scripting attack in the .html, .css, and .js files which define the single persistent web page which remains loaded in the user’s web browser throughout the duration of the user’s session with the service. If the operator of the service (MTI), or an attacker that it is able to compromise the web server, were to exploit the system by inserting malicious code in one of these files, then he/she may be able to capture sensitive information sent or received by users through the system in unencrypted form, or capture the users’ private keys.

The potential for a malicious scripting attack, like the one described above, is the reason for the design of the system based on the single persistent web page which remains loaded in the user’s web browser throughout the duration of the user’s session with the service. This is in contrast to the design typically used in web applications, where a new web page is downloaded by the user’s web browser from the web server after each action by the user, replacing the previously loaded web page. In the design used by EncryptedSend, at no point during the user’s session is a full page refresh done; instead client-side scripting associated with the single persistent web page instructs the web browser to interact with the user and the web server, and to display updated information retrieved by the user’s web browser from the web server through the API as the user interacts with the system. The reason for this design is to provide the user with a means to mitigate a potential malicious scripting attack. To protect himself/herself from this threat, it is necessary for the user to ensure the integrity of these files only once, at the beginning of the session - not repeatedly throughout the session as would be necessary after each full-page refresh. Therefore, if the user is able to verify the integrity of the .html, .css, and .js files which define the single persistent web page which remains loaded in the user’s web browser throughout the duration of the user’s session with the service, then this is sufficient to protect himself/herself from a malicious scripting attack throughout their session with the service. Those users that are able to independently verify the code that defines the single persistent web page used by EncryptedSend can do so using the ‘view source’ function in their web browser. Also, users can verify that this single persistent web page remains loaded in their browser (and that a full page refresh is never done during their session) using the networking function in the developer tools function of their web browser.

To assist users in protecting themselves against a malicious scripting attack by a third-party attacker other than MTI, MTI will use the bitcoin blockchain to publish the SHA256 hash of the root HTML document of the set of files files which define the single persistent web page used by EncryptedSend, and a new hash will be published whenever an update is made to any file in this set. MTI’s will use the bitcoin address `13NHh7kGwusYRNXPBDaW98NfkLUaUKUMAk` for this purpose. Users can find the latest published hash by entering the above address in their preferred blockchain explorer. The latest hash can be found in the op_return field of the most recent transaction originating from this address. Thus, users can verify that the set of files that are currently loaded in their browser have not been compromised by a third-party attacker by using the ‘save page as...’ function in their web browser to save the root HTML document to their system, then taking a SHA256 hash of this file, and comparing this hash to the reference hash published by MTI in the blockchain. If the derived hash matches the published reference hash, then this ensures that the files have not been tampered with by a third-party attacker. It should be noted subresource integrity is utilized in the root document for references to all supporting files. Therefore, ensuring the integrity of the root document via the procedure described above also ensures the integrity of all supporting files, as any change to any supporting file would necessitate a change to its reference in the root document as well, otherwise the change would be detected by the browser when enforcing subresource integrity. The blockchain's immutability property makes it attractive for storing these references hashes, as information written to the blockchain cannot be modified or deleted by an attacker. The private key for the above bitcoin address is stored offline.

The procedure described above can be used to prevent a malicious scripting attack by a third-party attacker other than MTI, however this procedure can not be used to prevent an attack by MTI. If MTI were to 'go rogue' or MTI were to be coerced into performing a malicious scripting attack by a third party, then MTI could simply publish an updated reference hash in the blockchain after inserting malicious code in one or more of the files, and the attack would go undetected using the method described above. To mitigate such an attack by MTI, and to reduce the amount of trust that users must place in MTI, MTI is seeking independent security experts to review this whitepaper and the code, and publish their own stamp of approval in a manner similar to the one used by MTI described above. Interested security experts are invited to contact MTI using the 'contact' link at https://www.encryptedsend.com/.

Another security concern that will come to the mind of most readers is the ability of the system to to validate the authenticity of other users of the system, via the procedure described in the registration process, to ensure that these users are 'who the say they are', and that they are in possession of the private keys corresponding to the public keys appearing in their certificate. If the system is not able to accurately validate the authenticity of its users, then this may allow malicious users to impersonate other users (an impersonation attack). This could result in a sender then unwittingly sending sensitive to a malicious impersonator, thinking that they are in fact sending this information to the impersonated user.

The potential for an impersonation attack exists if an attacker is able to gain access to the server, or if MTI were to 'go rogue', or MTI were to be coerced into performing such an attack. Users concerned about such an attack are encouraged to verify the certificates or the public keys of other users on the system via some out-of-band method, such as phone, email, SMS, fax, etc. Senders can view the certificate of a recipient by accessing the recipient's URL on the system. Likewise, recipients can view their own certificate by logging into their account the system. Therefore, a sender can to confirm that the certificate or public key that he/she sees for a recipient is in fact correct via their preferred out-of-band method.

Consider now the implications of a server breach. If an attacker were to gain access to the server, the attacker would gain access to any encrypted messages stored on the server which were sent by users through the system, but would not have the keys necessary to decrypt these messages. The attacker would be able to launch a malicious scripting attack like the one described above, but this attack would be detected if the user applied the procedure described above to safeguard against this type of attack. The attacker may also gain access the server's private keys used to sign certificates for users, and these keys could be used to falsely sign certificates in order to impersonate users on the system. However, certificates signed after a known breach could be revoked through blacklists and/or whitelists published by the system, and this attack would be detected if the user applied the procedure described above to safeguard against this type of attack. Notwithstanding, industry-standard best practices for hardening servers have been applied to safeguard against an attack on the server.

Yet another concern is a client-side breach. It should be noted that the ability of an attacker to gain access to the user's device can be potentially catastrophic, as this could enable the attacker to gain access to the user's unencrypted messages or the user's private key. Barring a compromise of the user’s device, an attack targeting the user’s web browser can also have serious implications. If an attacker is able to exploit a vulnerability in the user’s web browser, or a vulnerability in the code comprising the set of files which define the single persistent web page which remains loaded in the user’s web browser throughout the duration of the user’s session with the service, then the attacker may be able to gain access to unencrypted messages that the user sends and/or receives through the system, or the attacker may be able to gain access to the user’s private key. Because of this threat, considerable care has been taken to mitigate such an attack. To mitigate cross-site scripting (XSS) attacks, a strict content security policy (CSP) has been applied, which prohibits inline scripting. Also, all content originating from external sources is written to web page elements using the elements’ .innerText attribute (as opposed to the elements’ innerHTML attribute). To mitigate the potential breach of users’ private keys while using the system, private keys are stored in Web Crypto API CryptoKey objects, with the .extractable property set to false, to prevent code from being able to extract the private keys from these objects.

Conclusion

We have presented a solution that enable users to send and receive sensitive messages, files, and documents securely, with end-to-end encryption, using just their web browsers. In addition, we’ve outlined several security concerns that users should be aware of when using the system, and offered suggestions on how to mitigate these concerns.

We’ve made a reasonable effort to review and test the code that drives EncryptedSend, however it’s impossible to be sure that any software is bug free.

For this reason, we’ve published this whitepaper that explains in detail how the system functions. Also, the client-side code, which handles all of the cryptography in EncryptedSend, can easily be accessed using the developer tools in any modern web browser. We invite security experts and researchers to review the whitepaper and the code.

While we can’t guarantee that our code is bug-free, we do promise that any bugs reported to us will be fixed in a timely manner, and that any of our users that were impacted by a bug that we learn of will be notified promptly. Security experts and researchers interested in helping with this effort are invited to contact us.

Contact

MTI can be contacted at https://www.meixler-tech.com/contact.php.

Appendix 1a – Sending Process

Sender		Server
Sender points web browser to unique contact URL for recipient HTTPS GET https://domain.tld?contact=x		
	---------->	
		HTML, CSS, and Client-side scripting files.
	<----------	
Client-side scripting running within sender's web browser requests recipient's contact info and ECDH public key via the API based on recipient's id in URL above.		
	---------->	
		Server returns recipient contact info and ECDH public key via the API.
	<----------	
Sender composes message to recipient and optionally attaches one or more files. Client-side scripting running within sender's web browser derives a symmetric encryption key based on recipient's public key. Message is encrypted using derived key. Encrypted message payload uploaded to server via API.		
	---------->	
		Server stores encrypted message payload and optionally notifies recipient of message received.

Appendix 1b – Receiving Process

Recipient		Server
Recipient points web browser to URL for service HTTPS GET https://domain.tld/		
	----------->	
		HTML, CSS, and Client-side scripting files
	<-----------	
Recipient provides secret key file. Client-side scripting running within recipient's web browser extracts recipient's ECDH key pair from secret key file, then requests user account information.		
	----------->	
		Server API authenticates recipient by cryptographically verifying that recipient is in possession of the private key corresponding to the public key sent in the recipient's request. Upon successful authentication, server API queries database to fetch user account information and returns this information.
	<-----------	
Client-side scripting running within recipient's web browser displays user account information returned by the API. User requests log of messages received. Client-side scripting running within recipient's web browser requests log of messages received from server via the API.		
	----------->	
		Server API authenticates recipient by cryptographically verifying that recipient is in possession of the private key corresponding to the public key sent in the recipient's request. Upon successful authentication, server API queries database to fetch metadata for each message. Server API returns metadata, along with the encrypted subject, text, and list of attached files for each message.
	<-----------	
Client-side scripting running within recipient's web browser decrypts subject and text of each message using recipient's private key to derive the symmetric encryption key used by the sender to encrypt these parts. Metadata of each message is then displayed, along with		

decrypted subject, text, and list of attached files for each message.
Recipient requests file attached to a message.
Client-side scripting running within recipient's web browser requests encrypted file from server via the API.

----------->

Server API authenticates recipient by cryptographically verifying that recipient is in possession of the private key corresponding to the public key sent in the recipient's request. Upon successful authentication, server API returns attached file encrypted by the sender using the recipient's public ECDH key.

<-----------

Client-side scripting running within recipient's web browser decrypts attached file using recipient's public key and makes the file available to the user to open or save to their system.

Appendix 2a – API Process, Phase 1

<u>Client</u>

Generate ephemeral key pair
HTTPS GET Request:
/api/query.php?phase=handshake
&clientpermanentpublickeyx=X&clientpermanentpublickeyy=X
&clientephemeralpublickeyx=X&clientephemeralpublickeyy=X

-------->

<u>Server</u>

Verify received keys are valid
Generate ephemeral key pair
Compute shared secret
Generate exchange hash
Sign exchange hash
Temporarily store shared secret and exchange hash referenced by clientpermanentpublickey and clientephemeralpublickey
HTTPS GET Response:

```
header('Content-type: application/json; charset=utf-8');
header("HTTP/1.1 200");
{
  "status":200,
  "status_message":"success",
  "handshakeresponse":
     {
       "serverpermanentpublickeyx":"X",
       "serverpermanentpublickeyy":"X",
       "serverephemeralpublickeyx":"X",
       "serverephemeralpublickeyy":"X",
       "serversignedexchangehash":"X"
     }
}
```

<--------

Verify that server permanent public key matches pinned key
Verify received keys are valid
Compute shared secret
Generate exchange hash
Verify server's signature
Sign exchange hash

Appendix 2b – API Process, Phase 2 for echo, getcertificate, getaccountrecord, getmessages, getmessages functions

<u>Client</u>

Encrypt plaintext request using AES-GCM with shared secret created in phase 1
HTTPS GET Request:
/api/query.php?phase=handshake
&clientpermanentpublickeyx=X&clientpermanentpublickeyy=X
&clientephemeralpublickeyx=X&clientephemeralpublickeyy=X
&encryptedrequest=X

-------->

<u>Server</u>

Verify received keys are valid
Retrieve shared secret and exchange hash referenced by clientpermanentpublickey and clientephemeralpublickey
Verify client's signature
Decrypt request using AES-GCM with shared secret
If request requires authentication, identify user by the permanent public key sent in the request and authenticate
Process request
Encrypt response using AES-GCM with shared secret
HTTPS GET Response:

```
header('Content-type: application/json; charset=utf-8')
header("HTTP/1.1 200")
{
  "status":200,
  "status_message":"success",
  "encryptedqueryresponse":"X"
}
```

<--------

Decrypt response using AES-GCM with shared secret

Appendix 2c – API Process, Phase 2 for getmessagepart function

<u>Client</u>

Encrypt plaintext request using AES-GCM with shared secret created in phase 1

HTTPS GET Request:
/api/query.php?phase=handshake
&clientpermanentpublickeyx=X&clientpermanentpublickeyy=X
&clientephemeralpublickeyx=X&clientephemeralpublickeyy=X
&encryptedrequest=X

-------->

<u>Server</u>

Verify received keys are valid
Retrieve shared secret and exchange hash referenced by clientpermanentpublickey and clientephemeralpublickey
Verify client's signature
Decrypt request using AES-GCM with shared secret
If request requires authentication, identify user by the permanent public key sent in the request and authenticate
Process request
Encrypt response using AES-GCM with shared secret
HTTPS GET Response:
header('Content-type: application/octet-stream')
header("HTTP/1.1 201")
[byte stream]

<--------

Decrypt response using AES-GCM with shared secret

Appendix 2d – API Process, Phase 2 for postmessage, register1, register2 functions

<u>Client</u>

Encrypt plaintext command and data using AES-GCM with shared secret created in phase 1
HTTPS POST to /api/query.php
HTTPS POST Payload
Clientpermanentpublickeyxbytes +
clientpermanentpublickeyybytes +
Clientephemeralpublickeyxbytes +
clientephemeralpublickeyybytes +
clientsignedexchangehash +
encryptedcommandanddata

-------->

<u>Server</u>

Verify received keys are valid
Retrieve shared secret and exchange hash referenced by clientpermanentpublickey and clientephemeralpublickey
Verify client's signature
Decrypt request using AES-GCM with shared secret
If request requires authentication, identify user by the permanent public key sent in the request and authenticate
Process request
Encrypt response using AES-GCM with shared secret
HTTPS GET Response:

```
header('Content-type: application/json; charset=utf-8')
header("HTTP/1.1 200")
{
  "status":200,
  "status_message":"success",
  "encryptedqueryresponse":"X"
}
```

<--------

Decrypt response using AES-GCM with shared secret

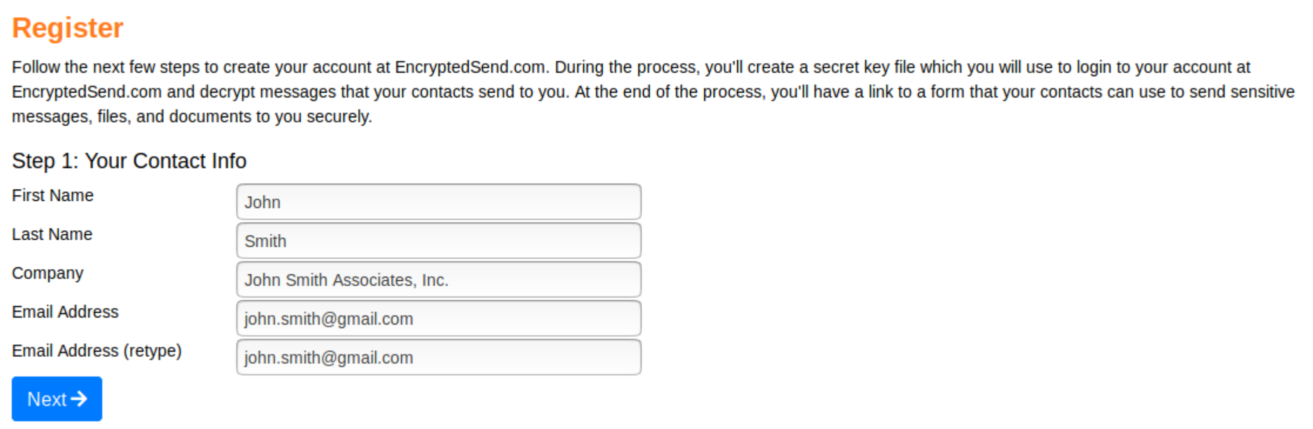
Appendix 2e – API Functions

Operation	echo
Phase 2 API Schema	Appendix 2b
Phase 2 HTTPS Method	GET
Phase 2 HTTPS Response Content-Type	application/json
Plaintext Request	command=echo &value=str
Plaintext Response	JSON formatted response containing string sent in request
Authentication Required	No
Description	Returns the string provided as the input parameter. Used for testing purposes.
------------	------------
Operation	getcertificate
Phase 2 API Schema	Appendix 2b
Phase 2 HTTPS Method	GET
Phase 2 HTTPS Response Content-Type	application/json
Plaintext Request	command=getcertificate &shortcode=x
Plaintext Response	JSON formatted response containing the requested recipient's certificate
Authentication Required	No
Description	Used to retrieve a recipient's certificate, given the first few bytes of the X value of the recipient's public ECDH key.
------------	------------
Operation	getaccountrecord
Phase 2 API Schema	Appendix 2b
Phase 2 HTTPS Method	GET
Phase 2 HTTPS Response Content-Type	application/json
Plaintext Request	command=getaccountrecord
Plaintext Response	JSON formatted response containing account information of authenticated user
Authentication Required	Yes
Description	Used to retrieve account information for user, given the user's public key, after authenticating.
------------	------------
Operation	getmessages
Phase 2 API Schema	Appendix 2b
Phase 2 HTTPS Method	GET
Phase 2 HTTPS Response Content-Type	application/json
Plaintext Request	command=getmessages &direction=received
Plaintext Response	JSON formatted response containing records of messages received

Authentication Required	Yes
Description	Used for showing logs of messages received
------------	------------
Operation	getmessages
Phase 2 API Schema	Appendix 2b
Phase 2 HTTPS Method	GET
Phase 2 HTTPS Response Content-Type	application/json
Plaintext Request	command=getmessages &direction=sent
Plaintext Response	JSON formatted response containing records of messages received
Authentication Required	Yes
Description	Used for showing logs of messages sent
------------	------------
Operation	getmessagepart
Phase 2 API Schema	Appendix 2c
Phase 2 HTTPS Method	GET
Phase 2 HTTPS Response Content-Type	application/octet-stream
Plaintext Request	command=getmessagepart &messageid=x&part=y
Plaintext Response	Byte stream containing truncated encrypted message payload including requested part
Authentication Required	Yes
Description	Used for retrieving parts of messages sent and received by users
------------	------------
Operation	postmessage
Phase 2 API Schema	Appendix 2d
Phase 2 HTTPS Method	POST
Phase 2 HTTPS Response Content-Type	application/json
Plaintext Post Data	'postmessage' + 0x00 + encrypted message payload file
Plaintext Response	JSON formatted response indicating success or failure
Authentication Required	No
Description	Used for sending encrypted messages
------------	------------
Operation	register1
Phase 2 API Schema	Appendix 2d
Phase 2 HTTPS Method	POST
Phase 2 HTTPS Response Content-Type	application/json
Plaintext Post Data	'register1' + 0x00 + URL-encoded account data
Plaintext Response	JSON formatted response indicating success or failure
Authentication Required	No
Description	Used for registering new users. Server sends email to user with verification URL. Verification URL contains AES-GCM encrypted user account information. AES-GCM key is known to server only.

------------	------------
Operation	register2
Phase 2 API Schema	Appendix 2d
Phase 2 HTTPS Method	POST
Phase 2 HTTPS Response Content-Type	application/json
Plaintext Post Data	'register2' + 0x00 + AES-GCM encrypted user account information contained in URL sent in verification email by register1 operation
Plaintext Response	JSON formatted response indicating success or failure
Authentication Required	No
Description	Used upon verification. Input is decrypted and stored in record for new user

Appendix 3a – Registration Process, Step 1

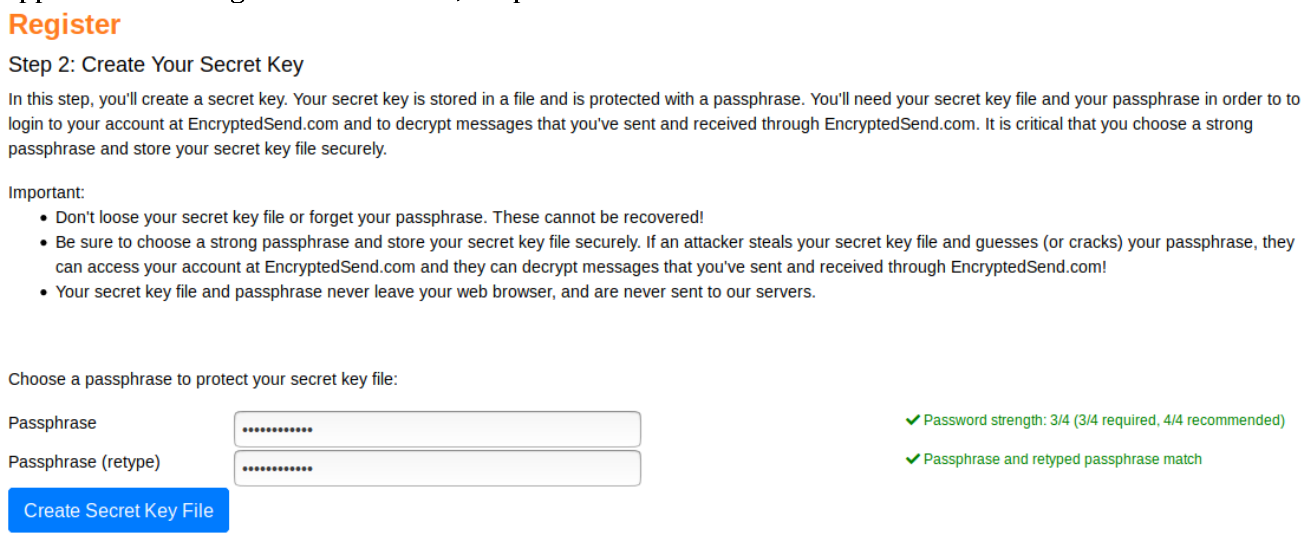

Register

Follow the next few steps to create your account at EncryptedSend.com. During the process, you'll create a secret key file which you will use to login to your account at EncryptedSend.com and decrypt messages that your contacts send to you. At the end of the process, you'll have a link to a form that your contacts can use to send sensitive messages, files, and documents to you securely.

Step 1: Your Contact Info

First Name	John
Last Name	Smith
Company	John Smith Associates, Inc.
Email Address	john.smith@gmail.com
Email Address (retype)	john.smith@gmail.com

Appendix 3b – Registration Process, Step 2



Register

Step 2: Create Your Secret Key

In this step, you'll create a secret key. Your secret key is stored in a file and is protected with a passphrase. You'll need your secret key file and your passphrase in order to to login to your account at EncryptedSend.com and to decrypt messages that you've sent and received through EncryptedSend.com. It is critical that you choose a strong passphrase and store your secret key file securely.

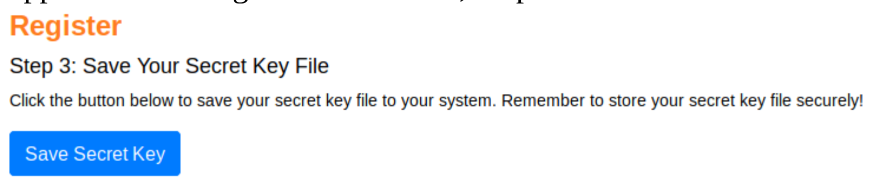
Important:

- Don't loose your secret key file or forget your passphrase. These cannot be recovered!
- Be sure to choose a strong passphrase and store your secret key file securely. If an attacker steals your secret key file and guesses (or cracks) your passphrase, they can access your account at EncryptedSend.com and they can decrypt messages that you've sent and received through EncryptedSend.com!
- Your secret key file and passphrase never leave your web browser, and are never sent to our servers.

Choose a passphrase to protect your secret key file:

Passphrase	••••••••••••	✓ Password strength: 3/4 (3/4 required, 4/4 recommended)
Passphrase (retype)	••••••••••••	✓ Passphrase and retyped passphrase match

Appendix 3c – Registration Process, Step 3



Register

Step 3: Save Your Secret Key File

Click the button below to save your secret key file to your system. Remember to store your secret key file securely!

Appendix 4a – Secret Key File Format and Certificate Format

<u>Secret Key File</u>
Version Data (12 bytes)
PBKDF2 iterations (4 bytes)
PBKDF2 salt (16 bytes)
ECDH Public Key X (32 bytes)
ECDH Public Key Y (32 bytes)
IV (12 bytes)
Encrypted ECDH Private Key (48 bytes)

Appendix 4b - Certificate Format and Certificate Signature Verification

```
Server-signed certificate:
{
          "payload":"eyJwYX...n0=",
          "serversignature":"qBf4ER...w=="
}
```

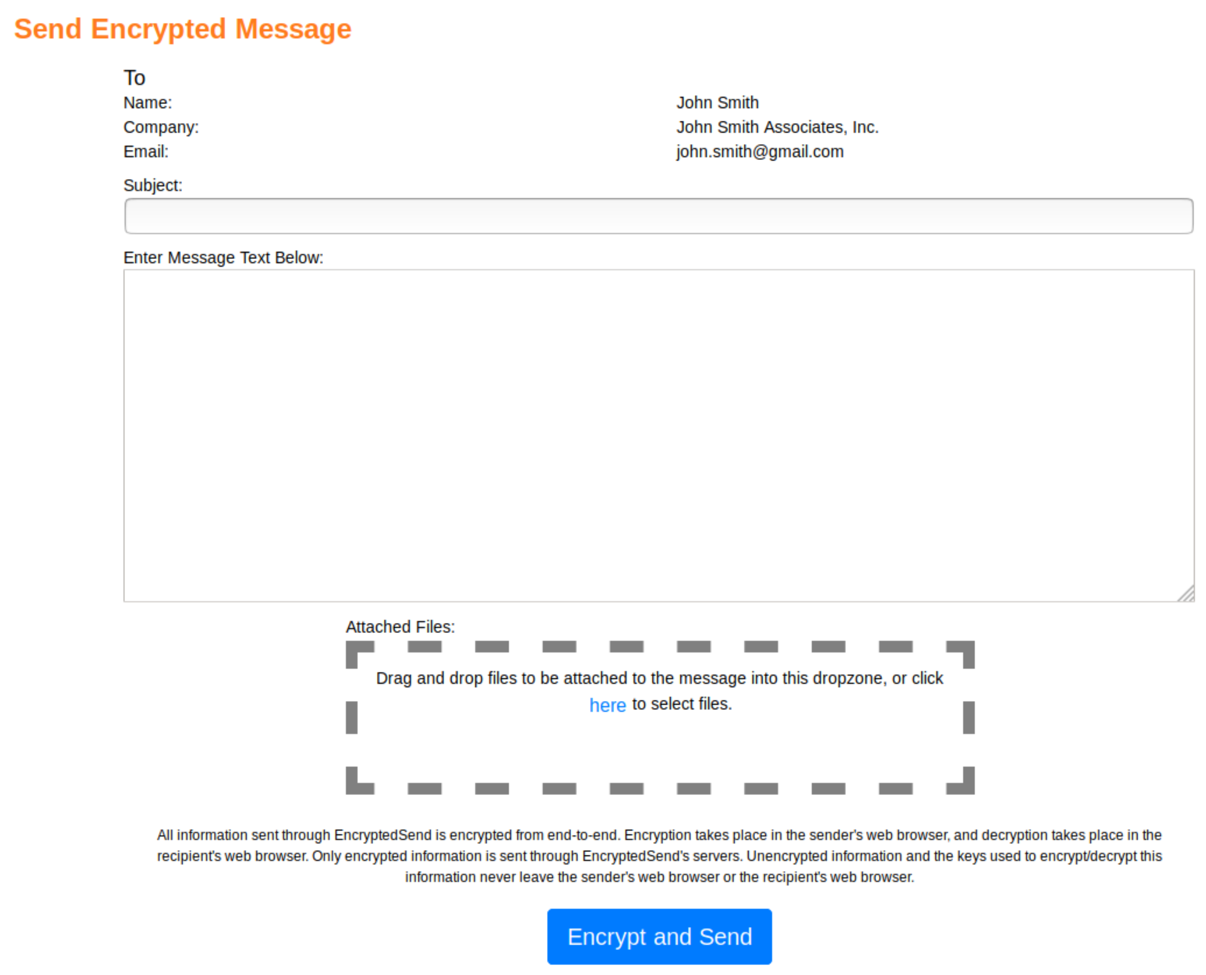
```
payload is base-64 decoded, resulting in client-signed certificate:
{
          "payload":"eyJmaX...Q==",
          "clientsignature":"RJrCqD...w=="
}
```

```
payload is base-64 decoded, resulting in certificate data:
{
          "firstname":"John",
          "lastname":"Smith",
          "company":"XYZ, Inc.",
          "email":"john.smith@gmail.com",
          "datecreated":"1571412271",
          "certificateversion":"1.1.1",
          "publickeyxhex":"2496fcafb1af19dc1a8daf6f89fd1195101b5348c06c136de3d9d4a274dd983f",
          "publickeyyhex":"450cf9f1bb66040b04b1e21549a087ee5eef5a34d7bd845fbc981de3aaf0448c",
          "chkAgreetotermsatsignup":"yes"
}
```

To verify the server’s signature, the server’s public key is used to verify the serversignature over the payload in the server-signed certificate.

To verify the client’s signature, the client’s public key from the certificate data is used to verify the clientsignature over the payload in the client-signed certificate. This confirms that the client is in possession of the private key corresponding to the public key in the certificate.

Appendix 5 – Message Composition Form



Send Encrypted Message

To

Name:	John Smith
Company:	John Smith Associates, Inc.
Email:	john.smith@gmail.com

Subject:

Enter Message Text Below:

Attached Files:

Drag and drop files to be attached to the message into this dropzone, or click here to select files.

All information sent through EncryptedSend is encrypted from end-to-end. Encryption takes place in the sender's web browser, and decryption takes place in the recipient's web browser. Only encrypted information is sent through EncryptedSend's servers. Unencrypted information and the keys used to encrypt/decrypt this information never leave the sender's web browser or the recipient's web browser.

Appendix 6 – Encrypted Message Payload File Format

<u>Parts Data</u>
part[0].name + part[0].type + part[0].hash
part[1].name + part[1].type + part[1].hash
…
part[N-1].name + part[N-1].type + part[N-1].hash

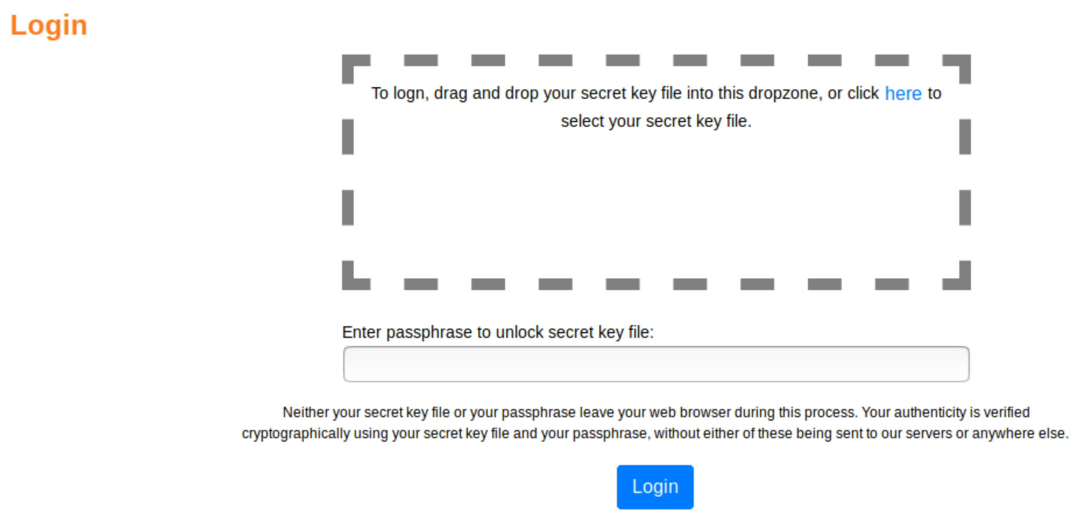
<u>Payload Bytes</u>

<u>Metadata Bytes</u>

Version Data (12 bytes)
Payload Date (4 bytes)

Sender Public Key X (32 bytes)
Sender Public Key Y (32 bytes)
Recipient Public Key X (32 bytes)
Recipient Public Key Y (32 bytes)
Length of parts data cipher (4 bytes)
Length of signed header (4 bytes)
Length of partcipherbytes[0] (4 bytes)
Length of partcipherbytes[1] (4 bytes)
…
Length of partcipherbytes[N-1] (4 bytes)
0x00 0x00 0x00 0x00 (4 bytes)
<u>Parts Data Cipher</u>
AESGCM(DerivedKey, Partsdata)
<u>Signed Header</u>
sign(senderprivatekey, metadatabytes + partsdata)
<u>Payload Cipher Bytes</u>
AESGCM(DerivedKey, partplaintextbytes[0]) (partcipherbytes[0])
AESGCM(DerivedKey, partplaintextbytes[1]) (partcipherbytes[1])
…
AESGCM(DerivedKey, partplaintextbytes[N-1]) (partcipherbytes[N-1])

Appendix 7 - Form Used To Begin Session With System



Login

To logn, drag and drop your secret key file into this dropzone, or click here to select your secret key file.

Enter passphrase to unlock secret key file:

Neither your secret key file or your passphrase leave your web browser during this process. Your authenticity is verified cryptographically using your secret key file and your passphrase, without either of these being sent to our servers or anywhere else.