DISPATCH
Internet-Draft
Intended status: Informational
Expires: 7 January 2026

J. K. Novak
Independent

6 July 2025

SMTP Extensions for End-to-End Encryption and User-Level Signatures
draft-korenenovak-smtp-e2eesign-00

Abstract

This Internet-Draft proposes adding extensions to the SMTP protocol that allow for true End-to-End Encryption and cryptographic signatures between users on a SMTP server. Current DKIM only allows for server verification, while messages sent through secure channels only encrypt traffic between servers, not between users.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://dcrubro.com/files/smtp-ee2esign-latest.txt. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-korenenovak-smtp-e2eesign/.

Discussion of this document takes place on the DISPATCH Working Group mailing list (mailto:dispatch@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/dispatch. Subscribe at https://www.ietf.org/mailman/listinfo/dispatch/.

Source for this draft and an issue tracker can be found at https://github.com/dcrubro/smtp-e2eesign.

Status of This Memo

This Internet-Draft will expire on 7 January 2026.

Table of Contents

1. Introduction

The current version of SMTP, even with protocols like SMTPS, STARTTLS and DKIM, lacks native support for user-level encryption and cryptographic identity. This document proposes three new SMTP extensions — ENCRYPTMESSAGE, SIGNMESSAGE and AUTH HASHEDPASS — that allow for end-to-end encrypted content, sender-level digital signatures and hash-based authentication that prevents SMTP servers reconstructing user private keys.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses the following terms:

* *MUA (Mail User Agent)* and *MTA (Mail Transfer Agent)* as defined in [RFC5321].

* *E2EE*, which stands for _End-to-End Encryption_.

* *Plain-text*, which refers to an unencrypted, human-readable message that can be read by anyone.

3. Security Considerations

This protocol enhances SMTP security by enabling user-level E2EE and cryptographic signatures. However, several risks and considerations apply:

3.1. Downgrade Attacks

Clients and servers MUST ensure that use of ENCRYPTMESSAGE, SIGNMESSAGE and AUTH HASHEDPASS is not silently disabled or ignored in transit. SMTP downgrade attacks are a known vector (e.g., STARTTLS stripping).

3.2. Channel Security

All authentication commands (e.g., AUTH HASHEDPASS, GETSALT) MUST be used *only over secure channels* such as SMTPS or STARTTLS. Servers MUST reject these commands over plain-text connections (normal SMTP).

3.3. Key Authenticity

Clients retrieving public keys via PUBKEY MUST validate that the key is returned from the intended domain. If DNS spoofing or MITM occurs, a forged public key could be inserted.

3.4. Key Compromise

If a user’s private key is compromised, message confidentiality and authenticity are at risk. Clients SHOULD support key rotation and expiration.

3.5. Hash Function Strength

If password-based key derivation is used, the hashing algorithm (Argon2) MUST be strong and salted. SHA-1 and MD5 are not permitted.

3.6. Replay and Injection

Signatures must include timestamping or message-ID binding to prevent replay or message tampering. Signatures over headers MUST include all content-critical headers (To, From, Date, Subject).

3.7. Metadata Leakage

Message headers like "Subject" may leak information about the message. These SHOULD be encrypted along with the main body of the message to avoid leaking information.

4. IANA Considerations

This document requests the registration of the following SMTP extension keywords in the "SMTP Service Extensions" registry:

* Keyword: ENCRYPTMESSAGE Description: Indicates support for user-level message encryption via public key. Reference: This document

* Keyword: SIGNMESSAGE Description: Indicates support for cryptographic message signatures by end users. Reference: This document

* Keyword: AUTH HASHEDPASS Description: An SMTP AUTH method that uses hashed password credentials. Reference: This document

5. Protocol Extensions

Servers that support the extensions defined by this document MUST advertise the ENCRYPTMESSAGE, SIGNMESSAGE and AUTH HASHEDPASS EHLO keywords. These keywords indicate that the server supports the extensions, headers and commands defined by this document.

6. Hashing and Key Derivation Algorithms

This protocol uses a password-derived key for both authentication (AUTH HASHEDPASS) and private key generation (via GETSALT).

Servers and clients implementing this protocol:

* *MUST support Argon2id* with at least 256-bit output and configurable parameters

* *MUST NOT use PBKDF2, SHA-1, or unsalted hashes*

Salt values used in this scheme MUST be: - At least 128 bits - Unique per user - Randomly generated at account creation and returned by GETSALT

7. Protocol Definitions

This document introduces three new EHLO extensions: ENCRYPTMESSAGE, SIGNMESSAGE and AUTH HASHEDPASS. These keywords advertise the server’ s support for end-to-end encryption, user-level message signing and hash-based authentication as defined in this protocol. Clients can use their presence in the EHLO response to determine whether the remote SMTP server supports these features.

This protocol also defines four new SMTP commands: PUBKEY, SETPUBKEY, GETSALT and RSTSALT.

* *The PUBKEY command* retrieves the public key associated with a specific email identity, allowing the sender to encrypt messages for the intended recipient. The key is returned as a base64 encoded string with a prepended algorithm identifier. It SHOULD be issued before *MAIL FROM* and doesn’t require authentication.

C: PUBKEY alice@example.com
S: 220 ed25519 QUFBQUMzTnphQzFsWkR...

* For an unknown user:

C: PUBKEY bob@example.net
S: 550 No public key available for bob@example.net

* *The SETPUBKEY command* requests the server to set the server-stored public key for the authenticated user to the requested value. This command can be used to modify the public key in case of a password or salt change. It SHOULD be issued before *MAIL FROM* and requires authentication.

C: SETPUBKEY base64(publickey)
S: 250 OK

* For an unauthenticated session:

C: SETPUBKEY base64(publickey)
S: 530 5.7.0 Authentication required

* *The GETSALT command* retrieves the password-derived salt associated with the authenticated user in the current session. This command is available only after successful authentication using the AUTH HASHEDPASS mechanism. The salt is returned as a base64 encoded string. It SHOULD be issued before *MAIL FROM* and requires authentication.

C: GETSALT
S: 220 SALT ODcyZmQwNTFlOTM4OTFh...

* For an unauthenticated session:

C: GETSALT
S: 530 5.7.0 Authentication required

* *The RSTSALT command* requests the server to reset the server-stored salt for the authenticated user to a new, random (128 bits is sufficient) value, and returns it to the client as a base64 encoded string. It SHOULD be issued before *MAIL FROM* and requires authentication.

C: RSTSALT
S: 220 SALT MDk5YTE4ZmJjODlhMTg4...

* For an unauthenticated session:

C: RSTSALT
S: 530 5.7.0 Authentication required

7.1. Authentication Method: AUTH HASHEDPASS

This protocol defines a new authentication mechanism: AUTH HASHEDPASS. This mechanism is intended for use only over secure channels, such as SMTP with STARTTLS or SMTPS (port 465). Servers MUST NOT advertise or accept AUTH HASHEDPASS over unencrypted connections.

Unlike AUTH PLAIN, which transmits the user's cleartext password (typically as a base64-encoded \0username\0password string), AUTH HASHEDPASS transmits a cryptographic hash of that string instead.

The client computes the hash of the \0username\0password string using a secure, salted *Argon2* hashing function. The result is then sent to the server:

C: AUTH HASHEDPASS base64(\0username\0hash(password))
S: 235 Authentication successful

The server compares the received hash to a verifier stored at account creation. This allows the server to authenticate the client *without ever seeing or storing the original password*.

Because the hash is static and reusable, this authentication mechanism MUST be used only over a secure channel (e.g., TLS). This protects the hash from replay or offline brute-force attacks.

7.2. Message Headers

This protocol defines a couple headers that can be added to a message to identify the sender and the mail.

* X-Sender-Encrypted: This header should be added to a message if its contents are encrypted in compliance with this protocol. If it's missing or set to "no", the message is considered plain-text.

X-Sender-Encrypted: yes

* X-Sender-Pubkey: This header is a courtesy header that includes the sender's public key as a raw UTF-8 string with a prepended algorithm identifier.

X-Sender-Pubkey: ed25519 AAAAC3NzaC1lZDI1NTE5AAA...

* X-Sender-Signature: This header is a courtesy header that includes the sender's public key as a raw UTF-8 string with a prepended algorithm identifier. This signature is derived from the raw (byte-for-byte) message contents (after headers), and the optional

headers From, To, Date, Subject and Message-ID (if they're present). This signature can be verified by issuing a PUBKEY command to the sender server for the sender email address and verifing the signature against the encrypted source data and the retrieved public key. This header *MUST not be trusted* and *MUST be validated via PUBKEY* to avoid security pitfalls.

X-Sender-Signature: ed25519 d75a980182b10ab7d54b...

8. Protocol Process

This protocol defines a simple process to exchange encrypted and/or signed mail.

8.1. Message End-to-End Encryption

To provide true E2EE to users, a few conditions must be met.

1. The plain-text (unencrypted) mail should only be available to the sender user and the receiver user, with no middle party being able to access it.

2. The decryption (private) key should only be accessible to the user which receives the encrypted mail.

By protocol design, each user should have their own public/private keypair, which can be generated via a preferred asymmetric encryption algorithm (e.g. RSA or ED25519).

Each user's public key SHOULD be stored by the SMTP server responsible for handling mail for their domain (e.g., the server handling example.com stores public keys for users like user@example.com). These public keys enable senders to encrypt messages such that only the intended recipient — the holder of the corresponding private key — can decrypt them.

The private key — which can be used for decrypting messages that have been encrypted by its corresponding public key — SHOULD only be held by the actual user. No copy of it should exist on the SMTP server or any other place.

8.1.1. Keypair Generation

Generating a keypair poses a problem. Logically, we could just generate a random keypair, store the public key on the SMTP server and keep the private key for ourselves, however this gives users a new responsibility of keeping another secret besides their authentication password.

The ideal scenario should be to use the user's password to derive the private key (and subsequently the public key). However, users cannot realistically be trusted to create secure and random passwords to generate a secure keypair.

Instead, this document proposes the solution of using a random salt, stored by the server. To put this sequence into a proper flow:

1. When the user authenticates with the SMTP server, the server checks its local salt store.

2. If no salt is found, one is randomly generated (128 bits is sufficient) and stored by the server.

3. The server responds to the client (user) with the base64 encoded salt, e.g.:

```
C: AUTH HASHEDPASS base64(\0username\0hash(password))
S: 235 Authentication successful
S: 250 SALT <base64-salt>
```

4. With the gotten salt, the user derives a private key for the desired algorithm using a function KDF by passing it the arguments as such: KDF(password, salt)

5. The user (still in the authenticated session) sets its public key on the SMTP server by using the SETPUBKEY command as described in the "Protocol Definitions" section:

```
C: SETPUBKEY base64(publickey)
S: 250 OK
```

By using this method, the user and server now have their required data, without leaking any sensitive info. Since the salt is a big part of the private key, but still not sufficient to generate it, this effectively makes the private key only generatable by the user that knows the password. Of course, making the password itself longer and more random helps a lot to making the private key even more unguessable.

8.1.2. Encrypting and Sending Mail

Encrypting mail and sending it to the receiver is fairly straightforward. We can use this flow to securely send mail to the receiver without any intermediate party being able to read it:

1. The sender (locally) writes out the e-mail with their desired content.

2. The sender (e.g. at the "example.net" domain) connects to the receiver’s domain’s SMTP server and issues an EHLO. If the EHLO responds with the ENCRYPTMESSAGE and SIGNMESSAGE extensions, we can continue with encryption.

```
C: EHLO example.net
S: 250-example.com
S: ...
S: 250-ENCRYPTMESSAGE
S: 250-SIGNMESSAGE
S: 250-AUTH HASHEDPASS
```

3. The sender issues the PUBKEY command for the receiver. If the SMTP server responds with *220*, the receiver has a public key on the server, which we can temporarily store. Else, if the server responds with *550* (the SMTP server does not have a public key for that user), we can fallback to sending normal, plain-text mail.

```
C: PUBKEY receiver@example.com
S: 220 ed25519 QUFBQUMzTnphQzFsWkR...
```

4. With this public key, we can proceed to encrypt the parts of the mail that we want to secure in *DATA* (e.g. the "Subject" header, the message body, attachments, etc.). We also add the X-Sender-Encrypted: yes header to inform the receiver that the message is encrypted. The message MUST be ASCII-armored to ensure best compatability with SMTP (in base64 format).

5. If the receiver SMTP server provides SIGNMESSAGE in its EHLO, we can proceed to sign the message. In the signature function, we can sign the agreed-upon, encrypted parts of the message (e.g. the "Subject" header, the "Date" header (or UNIX timestamp), the "From" header, the "To" header, the message body, etc.). We then append the signature to the mail in the X-Sender-Signature header.

6. Optionally, we can also provide our public key in the X-Sender-Pubkey header. This is optional, because a receiver SHOULD issue their own PUBKEY request to the sender’s SMTP server instead of trusting the provided one.

7. We can now submit the mail to the receiver’s SMTP server normally, using the encrypted content instead.

The whole flow in a SMTP session should look something like this:

User/Client side:

```
   C: EHLO example.net
   S: 250-example.com
   S: ...
   S: 250-ENCRYPTMESSAGE
   S: 250-SIGNMESSAGE
   S: 250-AUTH HASHEDPASS

   C: PUBKEY receiver@example.com
   S: 220 ed25519 QUFBQUMzTnphQzFsWkR...

   C: QUIT
   S: 221 2.0.0 Bye
```

Sender SMTP server side:

```
   C: EHLO example.net
   S: 250-example.com
   S: ...
   S: 250-ENCRYPTMESSAGE
   S: 250-SIGNMESSAGE

   C: MAIL FROM:<sender@example.net>
   S: 250 2.1.0 OK
   C: RCPT TO:<receiver@example.com>
   S: 250 2.1.0 OK

   C: DATA
   S: 354 End data with <CR><LF>.<CR><LF>

   Headers and encrypted message data here

   C: <CR><LF>.<CR><LF>
   S: 250 OK: queued as 12345

   C: QUIT
   S: 221 2.0.0 Bye
```

8.1.3. Reading Mail

When a receiver wants to read their mail, they can do so like this:

1. Authenticate into their SMTP server (e.g. using AUTH HASHEDPASS) as described above.

2. Request their own *SALT* by issuing a GETSALT command.

3. Rederive their private key from their password and received salt.

4. Use a mail retrieval protocol like *IMAP* or *POP3* and decrypt/verify the mail locally using their private key and the sender’s public key (gotten by issuing a PUBKEY <sender email> to the sender’s SMTP server).

8.1.4. Key Rotation

Users MAY want to occasionally rotate their keypairs to secure their mail. This can be done with the following flow:

1. Authenticate into their SMTP server (e.g. using AUTH HASHEDPASS) as described above.

2. Requesting a new *SALT* by issuing a RSTSALT command.

3. Derive a new private key from their password and received salt.

4. Issue the new public key (derived from the private key) to the SMTP server by issuing a SETPUBKEY command to it.

5. The old private key MAY be saved locally to keep access to old mail.

9. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, DOI 10.17487/RFC5321, October 2008, <https://www.rfc-editor.org/rfc/rfc5321>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

Acknowledgments

Thanks to the email security community for inspiration, guidance, and decades of pain to learn from.

Author’s Address

Jonas Korene Novak
Independent
Email: info@dcrubro.com