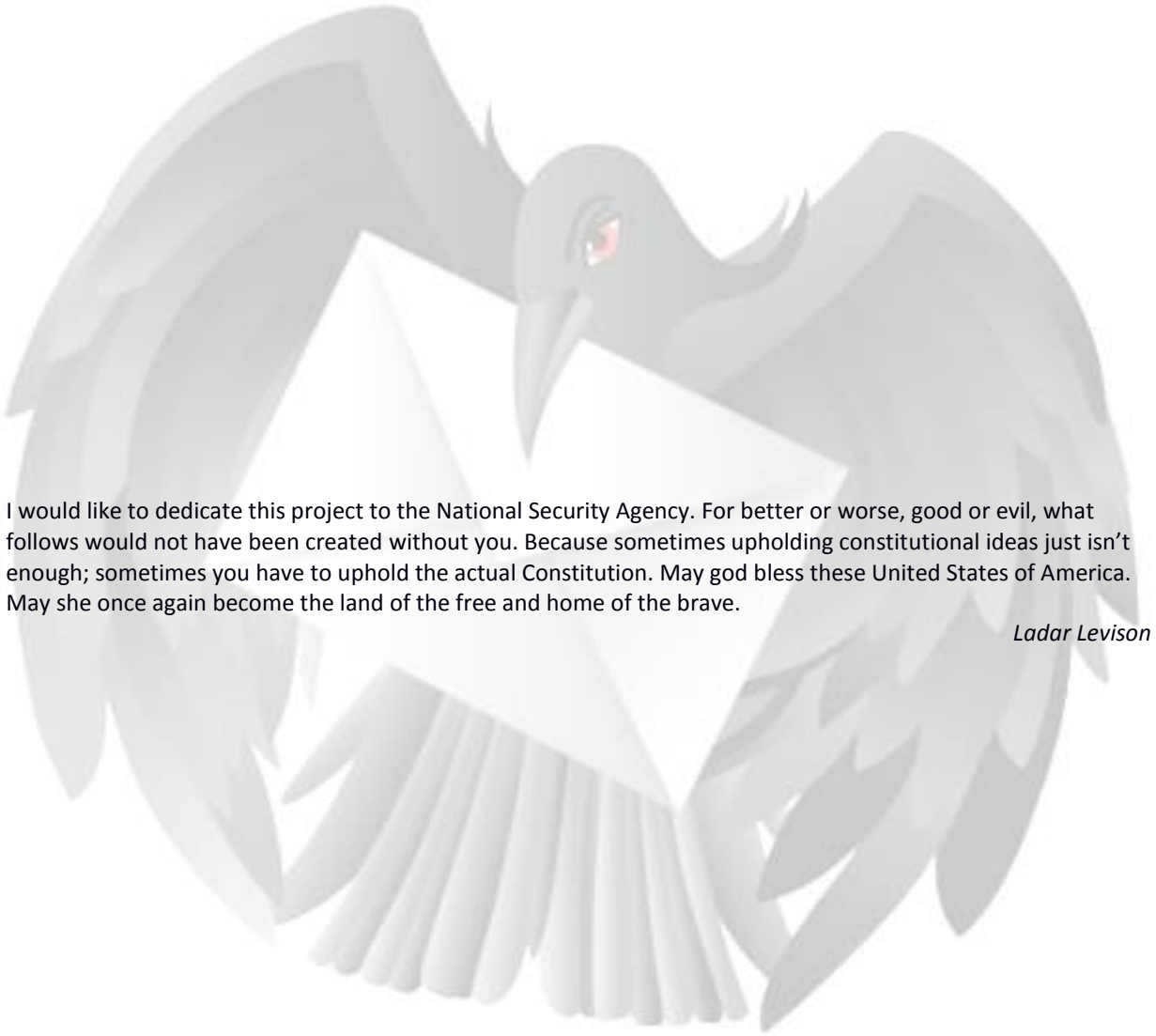
Dark Internet Mail Environment

Architecture and Specifications

June 2018



I would like to dedicate this project to the National Security Agency. For better or worse, good or evil, what follows would not have been created without you. Because sometimes upholding constitutional ideas just isn't enough; sometimes you have to uphold the actual Constitution. May god bless these United States of America. May she once again become the land of the free and home of the brave.

Ladar Levison

Please Note

This is a preliminary draft. We anticipate updated revision(s) will be published later this year. To find the latest version of this document, provide feedback, or contribute to the development effort, please use the links below.

Dive https://darkmail.info/spec

Discuss https://darkmail.info/forums

Develop https://darkmail.info/code

CONTENTS

FIGURES

OVERVIEW

This document is divided into sections that will introduce the reader to the Dark Internet Mail Environment (DIME) terminology, architecture, security, data formats, and protocol specifications.

PART 1: ABSTRACT

The Abstract serves as a short introduction to this document.

PART 2: TERMINOLOGY

The Terminology section defines all DIME-specific terminology as well as other industry standard terms, acronyms and key words used throughout this document.

PART 3: SYSTEM ARCHITECTURE

The System Architecture section introduces DIME, discusses the design goals, and then provides an illustrated guide to the functional components of a complete DIME-enabled mail handling environment.

PART 4: MANAGEMENT RECORD

The Management Record section describes the DNS record used to enable DIME support, advertise policies and provide a cryptographic trust anchor for an organizational domain name.

PART 5: SIGNET DATA FORMAT

The Signet Data Format section describes the data format for user and organizational signets.

PART 6: MESSAGE DATA FORMAT

The Message Data Format section describes the format used to encrypt messages and protect the data they carry.

PART 7: DARK MAIL TRANSFER PROTOCOL (DMTP)

The DMTP section details the unauthenticated protocol specification for message transfers and signet lookups. It provides connection standards, command syntax, and certificate requirements.

PART 8: DARK MAIL ACCESS PROTOCOL (DMAP)

The DMAP section details the authenticated access protocol specification used within the DIME ecosystem.

PART 9: GLOBAL LEDGER

The Global Ledger section details the implementation and use of a distributed immutable reflective ledger for signets.

PART 10: DARK MAIL ALLIANCE

The Dark Mail Alliance section details the creation of the Dark Mail Alliance (DMA) and describes its function and oversight responsibilities, including the processes used to manage DIME infrastructure projects.

PART 11: THREATS

The Threats section details the threats to the privacy functions of DIME and provides a discussion of security considerations not covered elsewhere.

PART 12: ATTACKS AND MITIGATION

The Attacks and Mitigation section details attack scenarios, and provides strategies DIME actors can employ to mitigate specific vectors.

PART 13: KNOWN VULNERABILITIES

The Known Vulnerabilities section will detail any known vulnerabilities as they are discovered.

PART 14: CREDITS

The Credits section provides attributions to the people that helped immensely with this document.

PART 15: REFERENCES

The References section provides a detailed bibliography for the references used throughout this document.

PART 1: ABSTRACT

This document provides the reader with an overview of the Dark Internet Mail Environment (DIME) along with detailed specifications for the data formats and protocols needed for a successful implementation. As revealed in the Overview, these chapters cover the following: Terminology, System Architecture, the Management Record, the Signet Data Format, the Message Data Format, the Dark Mail Transfer Protocol, the Dark Mail Access Protocol, the Global Ledger, the Dark Mail Alliance, a discussion of Threats, Attacks, and Mitigations incorporated into the system design plus a disclosure of any Known Vulnerabilities.

DIME strives to create a secure communications platform for asynchronous messaging across the Internet. The key design element which differentiates DIME from traditional Internet electronic mail (email) is the use of end-to-end encryption. The incorporation of encryption directly into the protocols ensures the secure and reliable delivery of email, while providing for message confidentiality, tamper protection, and a dramatic reduction in the leakage of metadata to processing agents encountered along the delivery path. To the extent possible, we have made DIME resistant to manipulation, but a secure system is only as strong as its weakest link. The goal with DIME has been, wherever possible, to make the security of the system depend on the complexity of a user's password, and the strength of their endpoint's defenses.

This document should serve as an implementation and deployment guide. Our goal is to collect and present, in a singular place, all of the current DIME specifications, standards and best practices. The intended audience is system builders (software developers, integrators), system operators, security researchers and protocol designers. However, this document should provide anyone who reads it with an understanding of the details necessary to design, implement and deploy a secure system that conforms to DIME's strict user-centric requirements for privacy protection.

PART 2: TERMINOLOGY

A number of other terms are used throughout this document which have been taken from related specifications or developed out of colloquial usage. We have attempted to collect the terms with special or unusual means throughout this document and provide definitions for them here.

KEYWORDS

The keywords “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, “OPTIONAL”, “EXPERIMENTAL” and “ASSIGNED” are used throughout this document and are to be interpreted using the definitions provided below. [KEYWORD]

Keywords	Definition
MUST	This word, or the terms "REQUIRED” or "SHALL", assert that the definition is an absolute requirement of the specification.
MUST NOT	This phrase, or the phrase "SHALL NOT", asserts that the definition is an absolute prohibition of the specification.
SHOULD	This word, or the term "RECOMMENDED", asserts that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
SHOULD NOT	This phrase, or the phrase "NOT RECOMMENDED", asserts that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
MAY	This word, or the term "OPTIONAL", asserts that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein, an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides).
EXPERIMENTAL	This word describes elements of the system which are still in active development. Specifications for experimental functionality will likely change in the future and those changes MAY break implementations built using the experimental specifications provided here.
ASSIGNED	This word is used to describe identifiers associated with data format and protocol extensions which have been reserved by individuals, or organizations currently in the process of developing functionality associated with the keyword, but have not submitted specifications for their effort.

ACTORS

When discussing a simple mail transaction, four distinct actors are involved. For this document, an actor is narrowly defined by the cryptographic key associated with the actor. In reality, any of the individual actors below could represent multiple people sharing a single email address, or multiple hosts working collaboratively to support DIME for a single organizational domain.

Actor	Definition
Author	The cryptographic identity associated with the creator of a message.
Origin	Represents the author's service provider and the hosts responsible for providing DIME network services on the author's organizational domain.
Destination	Represents the recipient's service provider and the hosts responsible for accepting a DIME message on behalf of the recipient.
Recipient	The cryptographic identity associated with the recipient of a message.

ACCOUNT MODES

Security is a flexible term. To accommodate the different types of DIME users, and their radically different needs, the concept of an account mode has been created. Each mode represents a defined point along the security-functionality spectrum, and is designed to easily communicate to user's how an account operates. The primary differentiators between each mode are where message encryption (or decryption) occurs, and where a user's private key is stored.

Mode	Definition
Trustful	In this mode, the server handles all privacy issues on behalf of the user. This requires a user to trust the server. The assumption is that accounts operating in this mode will send messages using the Simple Mail Transfer Protocol (SMTP) and receive messages using the Post Office Protocol (POP) or the Internet Mail Access Protocol (IMAP). Webmail systems which handle the encryption functions server-side are also considered to be operating in trustful mode.
Cautious	In this mode, a server is only used to store and synchronize encrypted data. This includes encrypted copies of a user's private keys, and encrypted copies of messages. This mode is designed to provide a user experience comparable to email today, while dramatically minimizing the amount of trust placed in the server. Webmail systems which perform encryption inside a user's browser are considered to be operating in cautious mode.
Paranoid	In this mode, a server will never have access to a user's private keys (encrypted or decrypted). This mode is designed to minimize the amount of trust a user is required to place in their server, at the expense of functionality. This mode does not support webmail access, and does not allow the user to access their account from multiple devices without the assistance of an external method for synchronizing their key ring.

SIGNETS

Signets are broken into two distinct classes: "organizational" signets, which are associated with a domain name, and "user" signets which are associated with an email address. An email address is defined as a mailbox, or local part, in combination with a domain name. Each actor is associated with a signet. The Author and Recipient are associated with user signets, while the Origin and Destination are associated with organizational signets. While it is not a technical requirement, the assumption is that organizational signets will have long lifespans, and that user signets will be rotated frequently.

Category	Definition
Organizational Signet	This signet category holds the public keys associated with a domain name. Optional fields can be used to provide information associated with domain, such identity information, or access information which allow for the auto configuration of user clients. Every DIME-enabled domain will have a unique organizational signet, thus an organization with multiple domains will control multiple organizational signets. Subdomains may have a unique organizational signet or rely on the organizational signet of the parent domain. The private keys associated with organizational signets are used to sign user signets, access message envelope information, and sign outbound messages.
User Signet	This signet category holds the public keys associated with an email address. Optional fields associated with a user signet can be used to provide identity information, or advertise other properties associated with an email address. Every email address is associated with a unique user signet, although a single user who controls multiple email addresses can control multiple user signets. The private keys associated with a user signet are used to decrypt incoming messages and sign outbound messages. Access to a user's current signing key is required to rotate a user signet, and thus publish new public keys for an email address.

The signet data format defines a number of optional fields which have the capable of dramatically increasing the size of a signet. For efficiency, it is possible to extract and store the relatively small portion of a signet required for cryptographic purposes. Because the subset of cryptographic fields have been separately signed, it is possible split a full signet, while retaining the ability to cryptographically validate the resulting core signet.

Type	Definition
Core Signet	The core signet represents the required cryptographic fields associated with a signet. Core organizational signets are self-signed, while core user signets are self-signed by the user, and then counter-signed by the organization.
Full Signet	A full signet includes a complete core signet, including its signatures, along with a number of optional "informational" fields. A full organizational signet is terminated by a second self-signature, while a full user signet is terminated by a second organizational signature.

TERMS

For the reader's benefit, we have attempted to provide definitions for any additional terms used throughout this document that are unusual, or carry with them specific meanings in the context of this document.

Terminology	Definition
Actors	See the Actors section above for definitions of the different actors: Author, Origin, Destination and Recipient.
Alternate Name (AN)	The Alternate Name or Alt Name field supplies additional domains associated with an X.509 certificate.
Advanced Persistent Threat (APT)	Advanced Persistent Threats are organized groups capable of continuous, extremely sophisticated attacks which are capable of devoting significant resources to achieve success.
ASCII	American Standard Code for Information Interchange
Attacker	Any unauthorized party attempting to gain access to message data (content, metadata, etcetera).
Certificate Authority (CA)	Certificate Authority whose signature asserts the validity and trustworthiness of an X.509 certificate.
Consumer	A consumer is a general term used to refer to both a signet resolver and a Message Transfer Agent, or a generic client application capable of sending commands to a server.
Common Name (CN)	The Common Name is the field which provides the domain name associated with an X.509 certificate.
Distinguished Encoding Rules (DER)	The Distinguished Encoding Rules are a binary encoding for X.509 certificates.
DIME	The Dark Internet Mail Environment loosely refers to the collection of format, protocols, and software used to facilitate the automated encryption of email.
DMAP	The Dark Mail Access Protocol is the authenticated protocol used to synchronize keys, submit signet signing requests, access messages and submit outgoing messages.
D/MIME	The Dark Multipurpose Internet Mail Extensions is the encrypted message format used to encapsulate and protect MIME messages within DIME.
DMTP	The Dark Mail Transfer Protocol is the unauthenticated protocol used to retrieve signets and transfer messages across organizational boundaries.
Domain Name System (DNS)	The Domain Name System resolves an arbitrary a string of letters and numbers into the numeric IP address needed to access the hosts associated with the given name.
End-to-End Encryption	Data encrypted between two endpoints, typically the author and recipient. Note that for users in the Trustful account mode, the server is the endpoint.
Fingerprint	**Core Fingerprint.** A SHA-512 hash generated from required cryptographic portion of an organizational or user signet. **Full Fingerprint.** A SHA-512 hash generated from the cryptographic and informational portions of an organizational or user signet. **Ephemeral Fingerprint.** A SHA-512 hash generated by combining the

	current core user signets for two email addresses. **Root Fingerprint.** A SHA-512 hash generated from the required cryptographic portion of the first signet in a user's current chain of custody.
Host	The computer associated with an IP address returned by a DNS name resolution and reachable using TCP/IP. This document assumes a host is a single computer, but it is worth noting that a single host IP address could reach more than one physical computer.
Internet Protocol (IP)	The Internet Protocol is a packet based, routable network of interconnected computers.
Key Ring	The private keys associated with a user's current and former signets.
Key Store (KS)	An authoritative source for organizational and user signets is referred to as a Key Store. Typically, the Key Store is the DMTP host responsible for supplying signets, although the term itself is independent of the protocol and transport used.
Management Record	The DNS record used to enable DIME support, advertise policies and provide a cryptographic trust anchor for an organizational domain name.
Man in the Middle	A type of attack commonly associated with cryptographic communication channels where a perpetrator sits between two victims, emulating each party to the conversation and fooling the endpoints into thinking they are directly connected. This type of attack allows the perpetrator to monitor and/or manipulate the conversation.
MDA	Mail Delivery Agent (sometimes referenced as Message Delivery Agent), the MDA is responsible for delivering messages to a recipient's message store.
MitM	See Man in the Middle above; MitM is a type of attack where a perpetrator sits between two victims to intercept and/or manipulate a conversation.
MS	The Message Store holds all of the messages associated with a user. The message store is sometimes referred to as a mailbox.
MSA	The Message Submission Agent is the message handler responsible for transferring a message from a user's device to the origin host associated with an organizational domain.
MTA	The Mail Transfer Agent (sometimes referenced as Message Transfer Agent) is the handling agent responsible for transferring messages between organizational domains.
MUA	The Mail User Agent is a fancy way of referring to a user's email client, the MUA allows users to send and receive messages.
OCSP	The Online Certificate Status Protocol is used to check with a Certificate Authority and find the real-time revocation status of an X.509 certificate.
OPA	The Organization Privacy Agent handles DIME specific cryptographic operations on behalf of an organizational domain.
Organization	The service provider, corporation or entity associated with a domain name.
Organizational Domain	A domain name which excludes any subdomain. An organizational domain is comprised of a top-level domain extension plus a single additional name typically associated with an organization.
PA	See OPA and UPA, the Privacy Agent is responsible for DIME specific

	cryptographic operations.
PFS	Perfect Forward Secrecy involves protection by an ephemeral key specific to the session or message and that the ephemeral key will remain secure if an associated long-term key is compromised in future.
POK	The Primary Organization Key is the key responsible for organizational level signing operations, which includes signing organizational signets, signing user signets, and signing outbound messages.
RR	A Resource Record refers to the result of a DNS query for a particular domain name.
SR	See Signet Resolver, the resolver retrieves the signet for a domain name or email address and performs the validation required to authenticate the result.
Signature	Signatures are created using EdDSA and the Twisted Edwards curve (and birational equivalent of Curve25519) commonly known as Ed25519. The Ed255219 curve is defined as: $$x^2 + y^2 = 1 (121665/121666)x^2y^2$$
Signet Resolver	The signet resolver is responsible for translating a domain name or email address into a signet, and is analogous a DNS resolver which translates a hostname into an IP address. Specifically, a signet resolver locates the authoritative server for a signet, retrieves it and then cryptographically authenticates the signet.
Signet Ring	The collection of organizational and user signets that have been retrieved and authenticated by a user's signet resolver.
SMTP	The Simple Mail Transfer Protocol (SMTP) is the traditional protocol used for the submission and transfer of messages.
SNI	The Service Name Identifier is a TLS extension which allows a single IP address to host TLS services for multiple domains using different X.509 certificates.
SOK	The Secondary Organization Key is similar to a POK, and can be used to sign user signets and sign outbound messages.
TCP	The Transmission Control Protocol is the network protocol used to ensure the reliable delivery, sequencing and reassembly of IP packets.
TLS	Transport Layer Security refers to a protocol used to provide an encrypted channel between two hosts connected using TCP.
TTL	The Time to Live refers to amount of time a DNS result is considered valid.
UPA	The User Privacy Agent is responsible for the cryptographic operations associated with an email address and on behalf of a user.
User	A person or collection of people represented by a single email address; note that the term is used throughout this document in a manner that is distinct from how it is used in reference to access control systems.

PART 3: SYSTEM ARCHITECTURE

Internet electronic mail (email) is a federated system of written communication which often requires messages to transit through a series of independent services. Email privacy is made challenging by the need to disclose handling information to stations along this path. In addition to the usual protection of content, a design goal for secure email must be to limit the meta-information that is disclosed so that a handling agent only has access to the information it needs to see. The Dark Internet Mail Environment (DIME)[1] achieves this with a core model having multiple layers of key management and multiple layers of message encryption. The system architecture modularizes functionality and that modularity permits a variety of implementation and deployment strategies. The data formats are transport agnostic and should permit transit over alternative infrastructure message transfer services using protocols independent of DIME. Integration with these alternatives is encouraged, with mechanisms provided for extending functionality and supporting protocol alternatives.

The essential challenge in email privacy is protection against compromised handling agents. Simple wiretapping of transit channels is reasonably well protected against by Transport Layer Security (TLS) [TLS]. However, TLS operates over only one Transmission Control Protocol (TCP) hop and email often travels through a significant number of these hops. Every transfer agent, including the immediate submission and delivery agents associated with the author and recipient(s), may become compromised [IMA]:

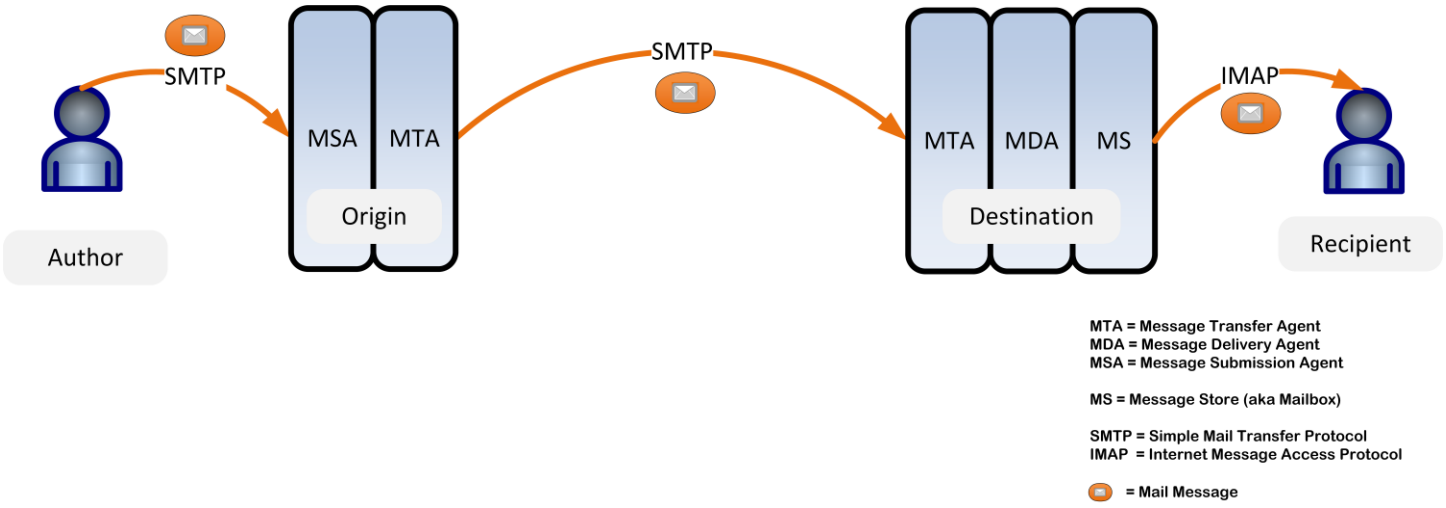


Figure 1 Traditional Email Handling Architecture

When a handling agent is compromised, the attacker could use the breach to gain access to keys, metadata, message content or all three. Hence, mechanisms to protect each are needed. DIME builds upon email's classic distributed architecture, but incorporates end-to-end encryption for the protection of private information. Each party responsible for handling a message is associated with an encryption key, and private information is encrypted for that key.

1 Perhaps sending a message through this service could be called "dropping a dime"?

To facilitate automation of the encryption process organizational mail servers provide encapsulated public key information using an encoded data format. The result is a data object called a "signet." The signet object is retrieved by a signet resolver from the mail host associated with a recipient's email address:

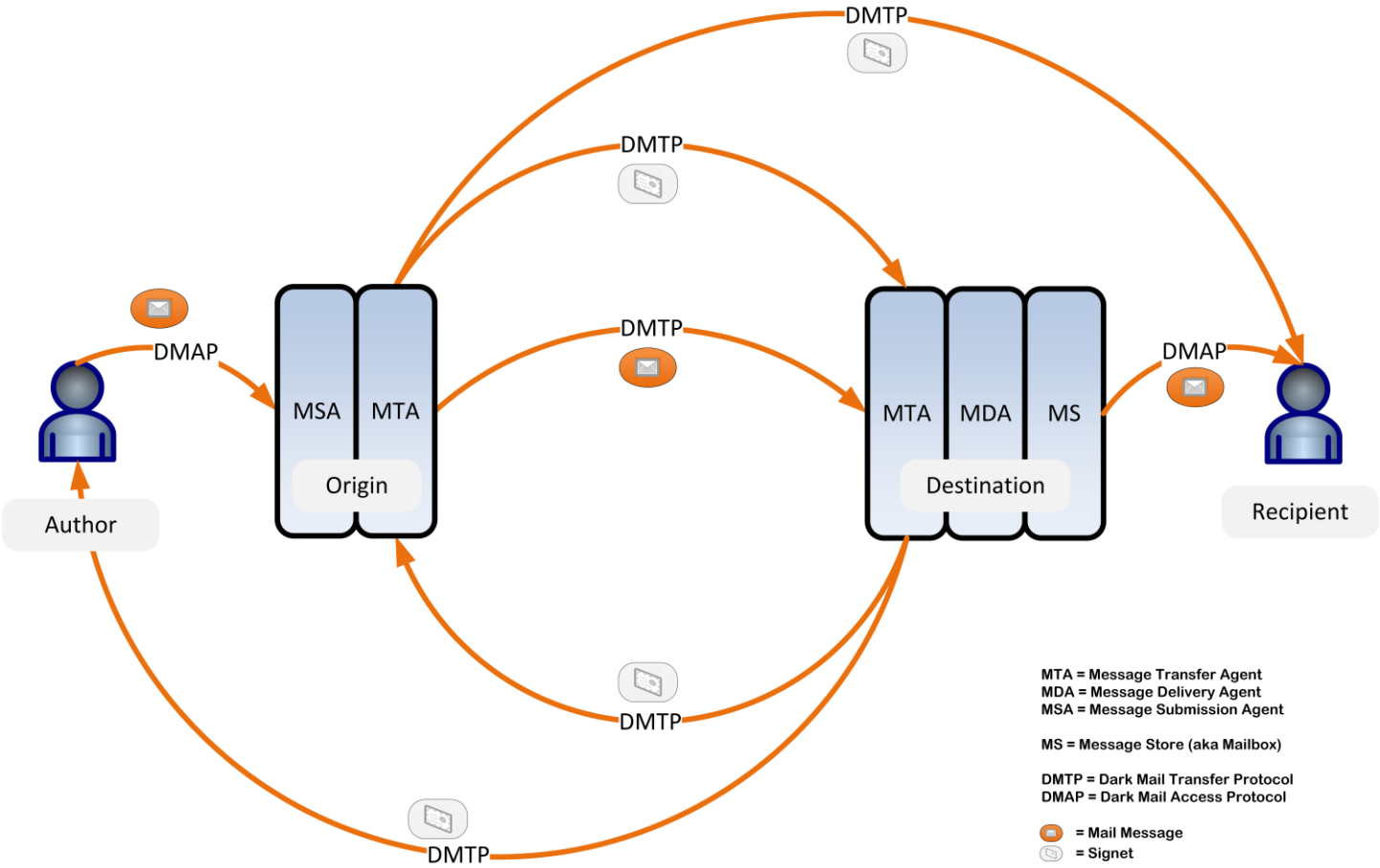


Figure 2 Proposed Email Handling Architecture

DESIGN GOALS

The goal of DIME is to provide a messaging system capable of protecting user privacy. This definition is ambiguous. For clarity, privacy is more precisely defined as the ability to control access to confidential information. In the context of email, confidential information is synonymous with information about a message, in addition to the actual message.

The term security is frequently has also been frequently abused. In our context security is used to discuss the mechanisms a user has to ensure the privacy of a message, and limit the potential for leakage. More security is equivalent with less information exposure, and a greater degree, or increased level of effort, required to breach the protections guarding confidential information. Encryption is the primary mechanism used to secure information, and ensure the privacy of confidential information.

These definitions led to specific deficiencies within the current email infrastructure, and its ability to ensure the security of confidential information. End-to-end encryption appears to ensure the protection of user privacy, but to ensure its ubiquitous use, the following technical goals were identified:

1. Automate key management, including the: creation, rotation, discovery and validation of keys.
2. Transparently encrypt and sign email messages to ensure confidentiality and author non-repudiation.
3. Resist manipulation by Advanced Persistent Threats (APTs).
4. Link security to the complexity of a user's password, and the strength of an endpoint's defenses.
5. Minimize the exposure of metadata to handling agents and service providers.
6. Give control back to the user.

OPERATIONAL DIRECTIVES

The core operational directives for DIME were developed to simplify the adoption of an email system protected by end-to-end encryption, minimize the information exposed to the minimum required for the system to function, and generally provide a protocol framework which is capable of protecting user privacy. At a high level, these core operational directives are achieved through the following elements:

- A handling agent only sees information about its immediate neighbors – the agent from which the message came and the agent to which it goes next. This specifically means that while the a message transits the open Internet, it travels inside a TLS tunnel, and the only information visible to (origin, MTA/MSA) host and target (destination, MTA/MDA) host.
- Author and recipient mailbox addresses are encrypted and then embedded within the message object. The origin host only sees the author mailbox address and the destination host only sees the recipient mailbox address.
- The origin host does not see the recipient mailbox address and the destination host does not see the author mailbox address unless the author and recipient are controlled by the same organization.
- Only the author and recipient can decrypt an entire message. The origin host and destination host only have access to their portion of the encrypted envelope and to the overall message structure.
- Messages are tree structured and content encryption is per leaf with independent keys for each leaf, permitting access to individual parts of the message without having to process other parts. This is especially helpful for clients with limited resources and/or bandwidth when accessing messages in a remote message store. It also permits other handling actions, such as the validation of message signatures, without needing to download the entire message through the use of tree signatures.
- Validation of signet (keys) is accomplished without the use of a formal CA construct, and no single source of information is automatically trusted. The basic validation model is to obtain a signet from a credible primary source and then confirm it with another pre-authenticated source. The two pre-authenticated sources currently available are a management record signed using DNSSEC or a TLS certificate signed by a recognized Certificate

Authority (CA). Both can be cryptographically traced by a signet resolver back to a trusted key that is shipped with the resolver.

- To the extent possible, layers of encryption have been used to mitigate the potential harm a nefarious actor can accomplish with the breach of a single piece of the DIME architecture.
- Public conveyance can be over a variety of transport services. This greatly lowers the barriers to DIME adoption.

This document provides a description of DIME's abstract network service architecture. An abstract network service architecture is distinct from any particular software design that might implement it, or specific scenarios that might derive from it. In particular, implemented software modules might combine or separate abstract network components. For example, the user agent and the message store might be implemented together. Alternatively, the user agent might be split between a simple user interaction module and a remote user 'semantics' module. (This is, in fact, the usual method of providing webmail user services; the variant of webmail that has the server download code to the user's browser dynamically is actually a small operational distinction that does not affect the model.)

FUNCTIONAL COMPONENTS

DIME's additions to the classic email architecture entail a few security-related modules, which are available to authors and recipients. The basic architecture has four categories of components:

- Classic email agents
- Privacy processing agents
- Key stores and signet resolvers
- Encrypted message objects

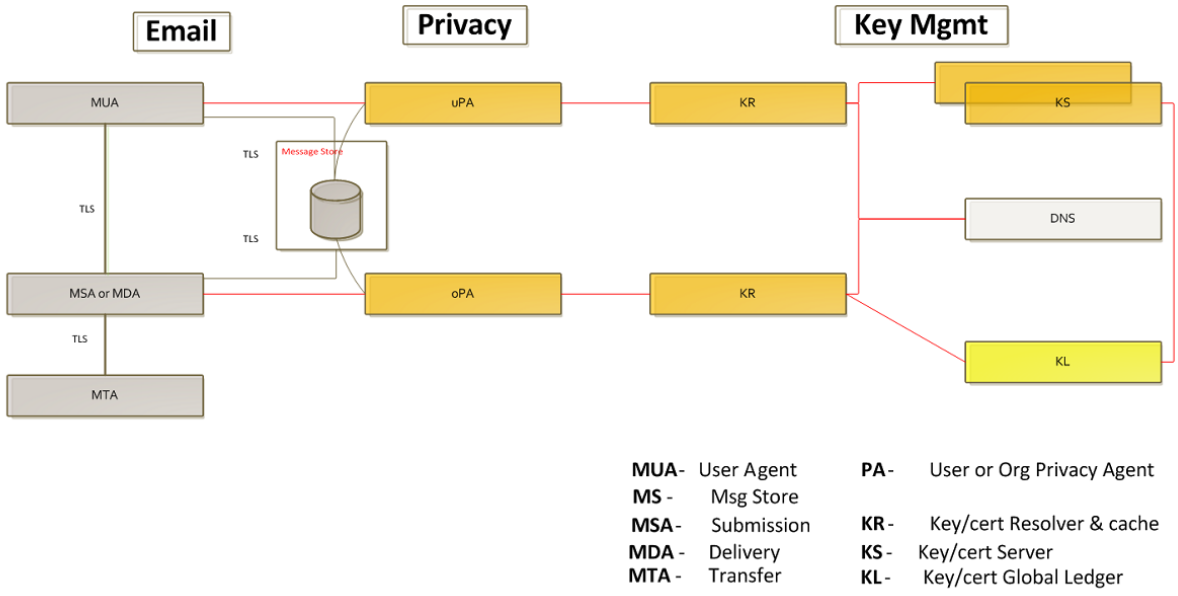


Figure 3 DIME Functional Component

TRANSPORT

DIME can be adapted to a variety of message transport or transfer services, with the choice of channel creating trade-offs between wiretapping and traffic analysis protection, balanced against scaling and interoperability requirements. Using Simple Mail Transfer Protocol (SMTP) to transport messages would ensure maximum reach but would have provided limited protection. The relatively similar Dark Mail Transfer Protocol (DMTP) is designed to provide similar reach (if adopted) with less metadata exposure.

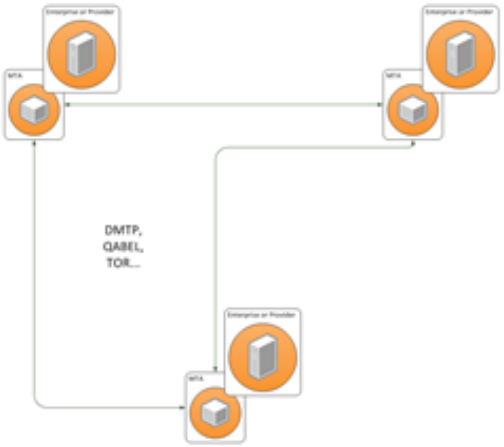


Figure 4 DIME Transport

MESSAGE OBJECT

In terms of handling and protection, each message is encrypted with an ephemeral key accessible by the author and one recipient. The basic message protection model then encrypts each component, called a chunk, to a distinct, ephemeral symmetric key; this includes encrypting each part of the message content (and attachments), with different keys, to

permit separable handling and protection. Access to keys is limited to essential actors: author, origin (submission server), destination (delivery server), and recipient. For example, the origin needs to see information about the destination, but not about the recipient. Messages are decrypted only when the information is needed. A chunk has one or more encrypted key slots. For each actor permitted to decrypt a chunk, there is a separate slot, with its own copy of the symmetric key; the key is encrypted to the actor's signet. A chunk that can be processed by three actors will have three copies of the symmetric key associated with that chunk. The representation of a message is a tree-structured object:

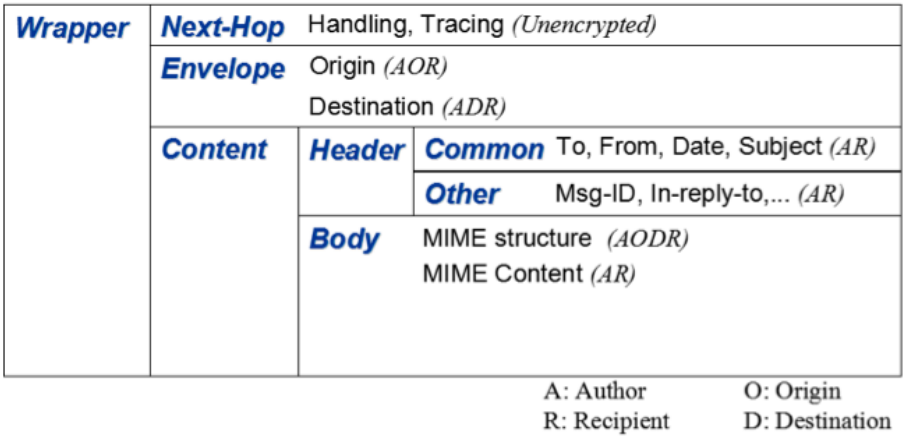


Figure 5 DIME Message Object

The basic structure is:

- *Wrapper* surrounding the entire message
- *Next-Hop* transit handling information, in cleartext, for the currently-active transport
- *Envelope,* with Origin and Destination information, separately encrypted
- *Meta,* including a the commonly used header fields separated from the remaining header fields
- *Message content,* including the body (and possibly attachments) [IMF]
- *Signatures,* including an author and an origin signature

The envelope has a further sub-structure, with each portion being independently encrypted, in order to permit selectively hiding information. The meta section contains headers such as the To, From, Subject and Date traditionally included with messages sent using SMTP [SMTP].

CLASSIC EMAIL AGENTS

The traditional email user and handling agent functional components are present in DIME. These are:

- MUA - Mail User Agent
- MSA - Message Submission Agent
- MDA - Mail Delivery Agent
- MTA -Mail Transfer Agent
- MS - Message Store

Messages in the Message Store (MS) have the content leaf nodes encrypted, with the structure in the clear. This permits selectively accessing leaves, which is needed by resource-limited devices, or clients accessing a remote message store over high-latency/low throughput connections.

PRIVACY PROCESSING AGENTS

DIME message processing semantics and cryptographic functions are handled by two additional system modules: the Organization Privacy Agent and the User Privacy Agent.

ORGANIZATION PRIVACY AGENT

The Organization Privacy Agent (OPA) interfaces with a user's email agent and the rest of the Internet. It facilitates user key management and creates a domain-name based package around the personal addressing and content of messages. It creates a secure transit channel that hides all information about the message using transport layer security, and provides access to the envelope information needed for immediate handling. This is accomplished through three functions:

Signing: The authenticity of a user's signet or the source of an arbitrary message is asserted by a cryptographic signature generated by the Organization.

Encryption: The Organization wraps and unwraps the full user email address, so that only the associated *domain* name is visible to the handling agent.

Channel: The message is transmitted over a channel protected by TLS so that only a message structure is visible during transit. TLS is responsible for providing perfect forward secrecy against network eavesdroppers.

It is worth emphasizing that the message object is encrypted separately from the encryption used for encrypting the transmission channel. Thus, even if a TLS channel is compromised, the only information gained by the attacker is the message structure. Likewise if the organizational encryption key is compromised, without the ability to compromise the TLS tunnel, nothing is gained. For a complete discussion of different potential scenarios see the Attacks and Mitigation section.

USER PRIVACY AGENT

The User Privacy Agent (UPA) provides the cryptographic functions required by a user's email agent. It facilitates key management, the retrieval of signets for recipients, alerts interactive users to potential signet issues, and facilitates the automatic encryption of messages. Because the encryption process can occur automatically it is possible for the UPA to reside on the server, or the user's device. Assuming the UPA resides on the user's device, it performs the following:

Encryption: The message envelope is encrypted for transit, to be unwrapped as necessary by handling agents along the way. As such the author address and the recipient domain are visible to the origin host, while the recipient address and author domain are visible to the destination host.. Only the UPA for the author and

the recipient are able to see both pieces of the envelope. Similarly, only the UPA for the author and recipient are able to access the message content chunks.

SIGNET RETRIEVAL SERVICES

Signet lookup and retrieval services are provided by a purpose-built Key Service (KS), modeled after the Domain Name System (DNS) architecture [DNS]. A Privacy Agent (PA) makes the request to a local signet resolver that in turn locates and queries the appropriate authoritative KS. Lookups use tailored Resource Records (RR) provided by a DNS resolver to locate the KS and validate the retrieved signet.

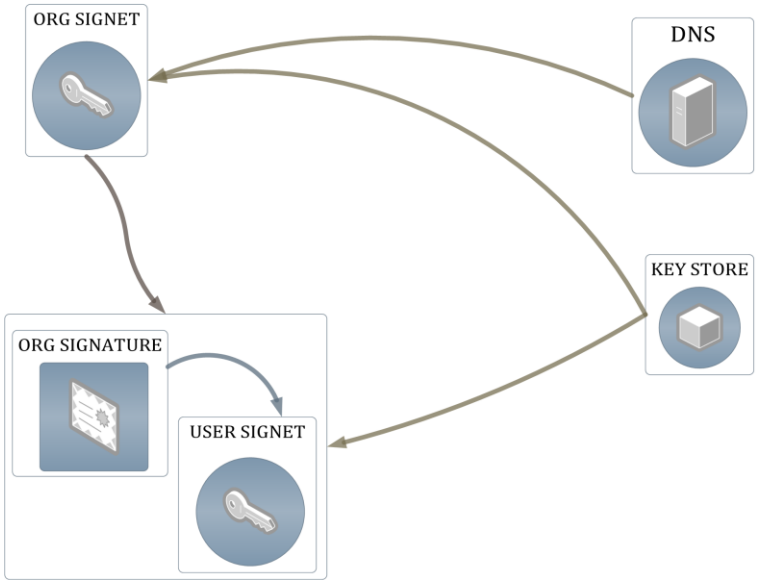


Figure 6 Signet Lookup Services

DIME avoids using a classic Certificate Authority (CA) mechanism for validating the signet's association with a name or address. It does this with a simpler, two-level mechanism:

- An organization with a domain name certifies individual users. The organization's signet is available through (at least) two mechanisms (an authoritative KS and confirmed using the management record signed using DNSSEC). The combination serves as relatively independent confirmation. If the management record lacks a DNSSEC signature, then validation is performed using the global-ledger or by confirming the TLS certificate supplied by the authoritative KS. Only the latter requires that a TLS certificate be signed by a recognized CA.
- A user signet supplied by the organization KS pairs key information with a user email address. It includes a variety of other attributes. Unlike a classic CA-based certificate, a DIME signet is not automatically trusted. Rather the evaluator of it treats it as input, then seeks to confirm the signet using another source, such as the organizational signet signature, or if the signet ring holds a previously authenticated result, by linking the Chain of Custody signature with the previously authenticated user signet.

PART 4: MANAGEMENT RECORD

A Dark Internet Mail Environment (DIME) management record is published in the Domain Name System (DNS) system and serves as the cornerstone for a DIME-enabled organizational domain. The management record advertises policies and hostname information and provides the cryptographic trust anchor for all DIME related functionality. The existence of a management record determines whether messages addressed to a particular domain should be sent using the DIME protocols, or as "naked" messages using the Simple Mail Transfer Protocol (SMTP). Organizational domains lacking a valid management record must be considered "legacy" and Mail Transfer Agents (MTAs) should apply any applicable policies regarding the delivery of naked messages.

INTRODUCTION

The management record is used, primarily, by signet resolvers and MTAs, which we collectively refer to as consumers. A signet resolver will use the management record to locate an organization's Key Service (KS) and validate organizational signets, while an MTA may retrieve a management record when it needs to deliver messages across organizational boundaries.

The only required field, and the primary purpose for a management record, is distributing the Primary Organizational Key (POK). The POK is a public key used for organizational signing operations, and whose corresponding private key is required to sign the organizational signet. The management record may also provide signatures for Transport Layer Security (TLS) certificates. If the management record has been signed using DNSSEC [DNSSEC], the TLS field signatures may be used to validate TLS certificates, in addition to, or in lieu of, a Certificate Authority (CA) signature. If the management record lacks a valid DNSSEC signature, then organizational hosts must still present an TLS certificate signed by a recognized CA.

A management record may also stipulate the signet expiry and refresh periods for a domain, advertise its policy in regards to the accepting/sending messages from legacy domains, and dictate how subdomains should be treated by DIME services. The management record may also provide addressing information for an organization's mail servers.

LOCATION

Signet resolvers must search for the management record using the "DIME" QTYPE [TBD] first. If the request fails, a resolver must also query the target domain using the "TXT" QTYPE [TXT]. When searching for a resource record using the "TXT" QTYPE a resolver must prefix the target domain with "_dime" to avoid conflicts with other uses for the TXT QTYPE [SRV]. The fully qualified domain name for a management record stored using the TXT QTYPE would be "_dime.example.tld," if the organizational domain was "example.tld."

Signet resolvers attempting to locate the applicable management record for email addresses using a subdomain, such as "user@sub.domain.example.tld," must use an increasingly specific search pattern. Starting with the base organizational domain, resolvers must continue by working towards the specific subdomain supplied until they encounter a management record with the "subdomain" policy field equal to the value "strict," or the resolver reaches the specific

fully qualified subdomain under consideration. For the domain "sub.domain.example.tld," a resolver must begin by requesting the DIME resource record for "example.tld" or if necessary, using the TXT QTYPE and the name "_dime.example.tld." Unless a subdomain policy of "strict" is encountered, a resolver must continue by searching for the DIME record using "domain.example.tld" or a TXT record using "_dime.domain.example.tld." As the final step in our example, a resolver must search for a DIME resource record using the fully qualified domain or a TXT resource record using "_dime.sub.domain.example.tld."

If the final search does not return a valid management record, and the nearest ancestor domain returned a management record with a subdomain policy of "explicit," then a signet resolver must assume the subdomain lacks DIME support and consider it a legacy domain. Alternatively, if the nearest ancestor management record supplied a subdomain policy of "loose" then the domain must be considered DIME-enabled, and the ancestor organizational signet and management record must be applied to the target subdomain. The increasingly specific query process ensures management records associated with ancestor domains may assert control over subdomains.

TEXT RECORDS

Operational considerations must be made if the management record is published using the TXT QTYPE. The DNS rules regarding TXT records stipulate that individual strings have a maximum length of 255 characters. As such, management records that exceed 255 must be split across strings. For optimal compatibility, management records must not split individual fields across strings.

While most DNS resolvers allow responses up to 4096 octets using UDP, a handful of non-conformant, but widely deployed DNS implementations truncate DNS responses over UDP at 512 octets (primarily Cisco PIX/IOS implementations). Organizations seeking to interoperate with the widest variety of consumers should ensure their management records have an overall length less than 512 octets.

The authoritative DNS servers for management records should support DNS queries using TCP so that resolvers experiencing problems with UDP responses being truncated may retrieve management records using TCP.

SECURITY

Internet email, like all domain name centric network services, depends upon the reliability and security of the global DNS system. Without DNS, email would cease to function, so despite its shortcomings[2], the DIME security model is dependent upon the reliability and security of the DNS system. For this reason we strongly recommended organizations deploy

2 DNSSEC only provides protection against tampering. It does not prevent attackers from blocking DNS responses entirely, and it does not protect the confidentiality of DNS queries, or the results. By tracking which domains a victim requests the management record for, an eavesdropper is able to track which domains a victim is corresponding with. This deficiency is one of several that will make it difficult to reduce the leakage of metadata beyond the organizational domain level.

DNSSEC to prevent the manipulation of DNS responses for their domain.[3] Consumers encountering a domain protected by DNSSEC must authenticate the entire signature chain, between the root DNS server key and the target domain. If the signature chain fails to validate, a consumer must discard the name server response and immediately notify the user a fatal error occurred with potential security implications.

An attacker could also manipulate unsigned management record responses, substituting the "pok" and "dx" field values to facilitate Man in the Middle (MitM) attacks on domains lacking DNSSEC protection. Attacker could also initiate downgrade attacks against unsigned domains by replacing valid responses with a "name error" [DNS]. The "name error" response may trick a systems into sending a naked message over SMTP that otherwise would have been encrypted as a D/MIME message and sent securely over DMTP.

If a management record is protected using DNSSEC, no other validation paths are required. A management record protected by DNSSEC is considered a pre-authenticated verification source. Authenticating organizational signets using signed management records is the preferred form of validation. If the management record is unsigned, a domain will need to support another pre-authenticated validation source or risk having their organizational signet trigger a warning, or be rejected outright by some consumers. Consult the *Validation* section of the Signet Data Format specification for a complete discussion of possible pre-authenticated validation sources.

EXPIRATION

The expiry value in a management record dictates how long an organizational signet should remain valid, even after the management record has been removed. To prevent attackers with control over the DNS servers for a domain from causing a domain to downgrade into legacy mode prematurely, a resolver must never reduce the amount of time remaining for a cached signet because a smaller "expiry" value was retrieved during a refresh attempt. This rule must only be applied to the expiry value; all other fields may be overwritten by an updated management record even if the resolver ignores the expiry value. This rule should only be applied until the amount of time corresponding to the difference in expiry values has lapsed.

FIELDS

Management records provide information using fields, with each field being comprised of a name/value pair. The table below is provided to indicate the properties associated with each field. The table specifies which fields are required, recommended, or optional, what the type of value each field provides, and whether a default value is applied when the field is absent.

3 Because the number of domains protected by DNSSEC remains a relatively small percentage, and because organizations will benefit significantly from DIME, even if their own management record is unsigned, we have decided not to make DNSSEC a prerequisite.

Management records should use the short version of a field name when specifying values, but may use the long version. Consumers must be capable of parsing and recognizing fields using the long name. Management records must always use the lowercase field names and enumerated values. Consumers must ignore fields using the improper case.

A field is properly defined as a name followed immediately by an equal sign (ASCII value 0x3d) and the desired value. A field ends when the first space (ASCII value 0x20) or semicolon character (ASCII value 0x3b) is reached. A field value may also end because the boundary for a management record is reached, and only the last field value may end without a terminating character. Tab characters (ASCII 0x09) must be treated as spaces, and extraneous whitespace, if discovered, must be ignored.

A single field definition must not span multiple TXT record strings. This means every string must end with either a space or semicolon, with the exception of the last one. This requirement ensures resolvers which concatenate TXT strings together are processed the same as resolvers which automatically insert whitespace between TXT strings.

Fields may be defined in any order. Fields which allow multiple values must specify every value as a fully formed field, using the complete name/value sequence defined above. If an additional value is encountered for a field which does not support multiple values, a resolver must use the first valid field value encountered. Resolvers that encounter additional instances of unique fields may optionally warn users, or silently ignore them.

DEFINITIONS

Management records must only use the fields defined below. If a consumer encounters a management record with an unrecognized field name, or encounters a name without a value, it must reject the entire management record and notify the user a fatal error occurred. If the consumer is searching for a subdomain management record, and encounters an ancestor with an invalid management record, the invalid record should be ignored completely, and consumers should continue searching as if the invalid manage record was never encountered.

Name	Short	Disposition	Multiple	Type	Default (Where Applicable)
primary	pok	Required	Yes	Armored Public Key	
tls	tls	Recommended	Yes	Armored Signature	
version	ver	Optional	No	Numeric	1
refresh	ref	Optional	No	Numeric	1
expiry	exp	Optional	No	Numeric	30
syndicates	syn	Optional	Yes	Hostname Literal	
deliver	dx	Optional	Yes	Hostname Literal	
policy	pol	Optional	No	Enumerated	mixed
subdomain	sub	Optional	No	Enumerated	mixed

DESCRIPTIONS

PRIMARY (pok)

The primary field provides the POK, or more specifically, the public key used to authenticate signatures supplied by a domain's organizational signet. Signet resolvers must ensure the organizational signet they retrieve for a domain name is signed using a POK value found in the management record. While it is possible for a domain to provide multiple POK values in a single management record, a signet resolver must ensure all of the signatures provided by a signet were created using the same private signing key, and that all of the signatures are valid.

If an organization rotates their organizational signet, they should leave the POK used by the previous organizational signet in their management record until the expiry period has lapsed, and any signing keys unique to that signet are no longer in use. This may require resigning messages or user signets. Alternatively, if an organizational signet is rotated, but the former private key has not been compromised, a the previous POK value may be removed from the management record and published as a Secondary Organizational Key (SOK).

All of the POK values must be valid base64 strings precisely 44 characters in length, otherwise a management record must be rejected and the user notified. The field value once decoded must be precisely 33 octets. The decoded octets must begin with the value {0x40}, which indicates the remaining 32 octets represent a compressed Ed25519 public key. [PGP-EdDSA] The public key must be in the compressed little endian format defined by the Ed25519 paper and used by the Ed25519 reference implementation [EdDSA].

The following is an Ed25519 public key, provided in hexadecimal form:

```
Qpub: 0x3f098994bdd916ed4053197934e4a87c80733a1280d62f8010992e43ee3b2406
```

If this same public key was prefixed with an octet value of {0x40} and then converted to a base64 value, the resulting POK field would appear in a management record as:

```
pok=QD8JiZS92RbtQFMZeTTkqHyAczoSgNYvgBCZLkPuOyQG
```

TLS (tls)

The X.509 certificates for DMTP hosts may be validated using the signatures provided by the TLS field. The value for a TLS field represents a 64 octet Ed25519 signature expressed in the form SIG = (R || S), where SIG represents the 64 octet signature created by concatenating the compressed R and S values generated using the EdDSA algorithm and the private key associated with a published POK value. To verify a TLS field, consumers must supply the X.509 certificate, in the binary Distinguished Encoding Rules (DER) format, and confirm the signature validates against a POK value provided in the same management record, using the EdDSA algorithm [EdDSA].

The TLS field value is generated by encoding the 64 octet Ed255219 signature using base64. If a conventional base64 implementation is used, then the 2 trailing pad characters must be stripped off to yield the correct value. Resolvers must reject the entire management record if one the TLS values provided is not a valid base64 string precisely 86 octets in length.

If one, or more, values are provided in the DIME management record, then the TLS certificate received while connecting to a DMTP host must match one of the provided values. This requires organizations using multiple certificates to sign and

provide signatures for all of the certificates used to protect DMTP connections. If an organization is unable to provide TLS signatures in the management record for all of the X.509 certificates in use, it must remove all of the TLS field values from the management record and obtain a signatures for its X.509 certificates from a recognized CA instead.

If a consumer encounters a DMTP host using a certificate that does not match any of the provided TLS field signatures, it must cleanly shutdown the TLS connection and disconnect, treating the connection attempt as a failure. If the threshold for connection failures has not been reached, and additional hostnames are available the consumer should continue onto the next host until it discovers one using a certificate with a valid TLS field signature.

Management records protected by DNSSEC which also provide TLS field signatures may use self-signed certificates. If the management record is not protected using DNSSEC, but still provides a TLS field signature, a consumer should ensure the certificate matches one of the available signatures first, followed by the validation rules required by TLS v1.2 [TLS], which require X.509 certificates to be signed by a recognized CA.[4] Consult the section of the DMTP specification for a complete discussion of TLS certificate validation rules for DIME.

If a TLS certificate generated the following signature values, provided here in hexadecimal form:

```
R: 0x56f90cca98e2102637bd983fdb16c131dfd27ed82bf4dde5606e0d756aed3366
S: 0xd09c4fa11527f038e0f57f2201d82f2ea2c9033265fa6ceb489e854bae61b404
```

Then the corresponding TLS field would appear in a management record as:

```
tls=VvkMypjiECY3vZg/2xbBMd/Sftgr9N3lYG4NdWrtM2bQnE+hFSfwOOD1fyIB2C8uoskDMmX
6bOtInoVLrmG0BA
```

SYNDICATES (syn)

The syndicate field provides the fully qualified domain name of an alternate host authorized to provide signet information for a domain name. The value must be a valid hostname literal and not an IP address. If an IP address is provided as the value, a consumer must reject the management record entirely and notify the user. If the value is a valid hostname literal, but does resolve into a valid IP address, or the DMTP connection attempt fails, then the consumer should ignore value. Consult the section of the DMTP specification for additional details on how hostname literals should be used by a signet resolver.

If a domain was syndicating signets to “syndicate.example.tld” then the syndicate value would appear in a management record as:

4 DNS-Based Authentication of Named Entities (DANE) [DANE] provides similar functionality, but lacks widespread deployment. The primary functional difference is DANE records only support the publication of complete certificates, a public key or a hash value. DIME uses the POK and cryptographic signatures to validate certificates. DANE also requires (by specification and not function) the deployment of DNSSEC, while DIME decided to classify DNSSEC support as a strong recommendation.

```
syn=syndicate.example.tld
```

DELIVER (dx)

Provides the fully qualified domain name for authoritative signet lookups, and for delivering encrypted D/MIME messages[5]. The value must be a valid hostname literal and not an IP address. If an IP address is provided, a consumer must reject the entire management record and notify the user. If the value is a valid hostname literal but does not resolve into a valid IP address, or the connection attempts fail, then the consumer should ignore the value.

If a domain, which supports single protocol mode DMTP connections on the host "dmtp.example.tld," then the corresponding delivery value would appear in a management record as:

```
dx=dmtp.example.tld
```

If a consumer is unable to establish a DMTP single protocol mode connection with any of the hosts provided as deliver field values, or if the management record does not include the deliver field, then a consumer must use the mail exchange (aka "mx") record for the target domain instead. When attempting to setup a DMTP connection to host found in a mail exchange (aka "mx") record, consumers must use dual-protocol mode. For additional details on the processing multiple values, connection failures, and the different the protocol modes, consult the section of the DMTP specification.

VERSION (ver)

The version field controls how a management record should be parsed and validated. Only a single value for this field, "1," is considered valid. If a consumer encounters a different value for this field, then the management record format has changed and a consumer must assume the changes are incompatible with this specification. As a result, consumers must reject management records which supply a value other than "1" with a fatal error. For management records which explicitly stipulate the management record version, the field must appear as:

```
ver=1
```

REFRESH (ref)

The refresh value controls how often a resolver should check whether a cached signet is current. The value is expressed in days. If the field is missing from a management record, then a resolver must assume the refresh period is 1 day. A

5 The decision to provide this information in the management record, rather than use a SRV record, was made because the latter would allow arbitrary port numbers. We felt that delivery hosts should be locked to a specific port, which allows network administrators to detect and/or block nefarious deployments. We also felt the port needed to be lower than 1,024 because binding to ports in this range requires elevated permissions on the most widely deployed operating system. Port 26 was chosen because it is currently unused, and because of its proximity to port 25 (which is used by SMTP).

signet only needs to be refreshed if the period has expired, and a user receives a message, or attempts to send a message to a recipient which will require the use of the signet. The complete refresh field, if provided explicitly should appear as:

```
ref=1
```

EXPIRY (exp)

The expiry value controls the number of days a signet must be considered valid, even if the resolver is unable to refresh the cached signet value. Consumers should continue using cached signets for signature validation, and message encryption up to the number of days specified in the expiry value, starting with the last successful refresh or retrieval of the signet. Signet resolvers should apply the expiry to management records as well, and ensure they continue treating domains as DIME-enabled, even if a management record is removed, for the number of days specified as the expiry value in the last successful retrieval. Note that management record refresh periods must not reduce the amount of time a domain is considered DIME-enabled when it retrieves updated management records. See the section above for details.

If the expiry field is missing a default value of 30 day must be applied. If a management record contains an explicit refresh value larger than the expiry value, then the refresh value should be used as the expiry value. An expiry value of 30 or higher is recommended for domains with no plans of returning to legacy mode. If the default expiry value was specified explicitly by a management record, it would appear as:

```
exp=30
```

POLICY (pol)

Provides the policy applied when transferring messages between origin and destination domains. Message acceptance and delivery must conform to the advertised policy when one of the organizations involved is DIME-enabled. D/MIME messages must be rejected when the delivery does not conform to the policy, or if the organization does not have a valid management record. In the absence of an explicitly defined policy field, resolvers must apply a default policy of mixed. If a resolver encounters a management record with a policy value that does not match one of the three enumerated values, then the entire management record must be rejected. The table below illustrates the appropriate outcome for a message between two domains with each of the possible policy dispositions.

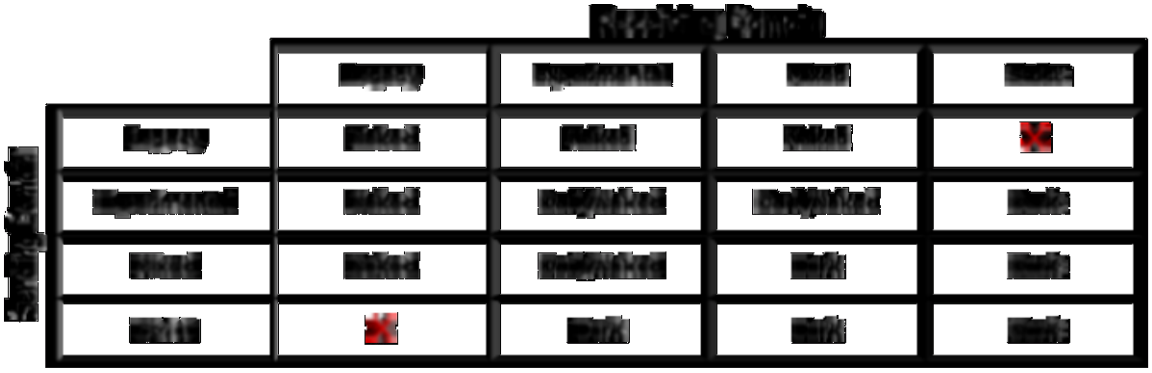


Figure 7 Policy Dispositions

The formal definitions for each of the enumerated policy values are:

- **Experimental.** This organizational domain will be sending both D/MIME and naked messages. Destinations with policies of experimental or mixed should accept both, while those with a policy of strict must reject naked messages. If a domain does not have a management record available then this organization supports the delivery of naked messages. Organizations with a policy of experimental should publish valid signets for all DIME-enabled addresses, or the appropriate error code for valid addresses which are not yet DIME-enabled. Senders with a policy of mixed or experimental may choose to deliver naked messages if they encounter an experimental policy for the destination and the recipient addresses does not have a valid signet available.
- **Mixed.** A mixed policy domain should send D/MIME messages to DIME-enabled domains, and naked messages to legacy domains. Likewise, a mixed policy domain will accept naked messages from legacy domains and D/MIME messages from DIME-enabled domains. Domains in mixed policy mode must ensure they only accept D/MIME messages from other DIME-enabled domains advertising a policies of strict or mixed.

  While mixed policy domains must send D/MIME messages to experimental policy domains by default, they may choose to send a naked message if the signet resolution process for a recipient fails with a permanent error code. Signet resolutions resulting in temporary errors should be retried. Likewise a mixed policy domain must accept both D/MIME and naked messages from domains with a policy of experimental.
- **Strict.** This domain must only accept D/MIME messages and must only send D/MIME messages. If a strict domain encounters a recipient domain without a management record or if signet resolution fails, the send attempt must also fail.

A policy field value of “mixed” would appear in a management record as:

```
pol=mixed
```

SUBDOMAIN (sub)

Determines whether a resolver should apply the management record to subdomain addresses. In the absence of an explicitly defined subdomain field, resolvers must apply a default value of mixed. If a resolver encounters a management record with a subdomain value that does not match one of the three enumerated values, then the entire management record must be rejected.

The formal definitions for each of the enumerated subdomain values are:

- **Explicit.** Subdomains must provide a management record and organizational signet. The absence of management record results in a subdomain being classified as legacy.
- **Mixed.** Subdomains may supply a management record and organizational signet, which are used instead of the parent domain. If the management record is missing, the values and organizational signet of the parent should be applied to the subdomain address.
- **Strict.** Subdomains must always use the management record and organizational signet of the parent domain which supplies this value.

A subdomain field value of "mixed" would appear in a management record as:

```
sub=mixed
```

EXAMPLES

At a minimum, all legal DIME management records must provide a POK value. All other values are optional, with default values being applied in the absence of a version, refresh, expiry, policy and subdomain field. In the absence of a deliver field, a domain's mail exchange (aka "mx") DNS record is used to supply the DMTP host. As a result a simple, but valid, DIME management record might look like:

```
pok=QD8JiZS92RbtQFMZeTTkqHyAczoSgNYvgBCZLkPuOyQG
```

DIME management records should specify values for the recommend field TLS, so that resolvers may validate DMTP connections using the TLS provided upon connection. A simple DIME management record which provides a signed TLS value might look like:

```
tls=VvkMypjiECY3vZg/2xbBMd/Sftgr9N3lYG4NdWrtM2bQnE+hFSfwOOD1fyIB2C8uoskDMmX
6bOtInoVLrmG0BA pok=QD8JiZS92RbtQFMZeTTkqHyAczoSgNYvgBCZLkPuOyQG
```

A complicated DIME management record, with all of the fields specified, might look like:

```
tls=VvkMypjiECY3vZg/2xbBMd/Sftgr9N3lYG4NdWrtM2bQnE+hFSfwOOD1fyIB2C8uoskDMmX
6bOtInoVLrmG0BA pok=QD8JiZS92RbtQFMZeTTkqHyAczoSgNYvgBCZLkPuOyQG pol=mixed
sub=strict dx=dmtp.example.tld syn=mirror.example.tld ref=1 exp=30 ver=1
```

The same complicated DIME management record could also use semicolons as the field delimiter, which would result in it looking like:

```
tls=VvkMypjiECY3vZg/2xbBMd/Sftgr9N3lYG4NdWrtM2bQnE+hFSfwOOD1fyIB2C8uoskDMmX
6bOtInoVLrmG0BA;pok=QD8JiZS92RbtQFMZeTTkqHyAczoSgNYvgBCZLkPuOyQG;
pol=mixed;sub=strict;dx=dmtp.example.tld;syn=mirror.example.tld;ref=1
exp=30;ver=1
```

PART 5: SIGNET DATA FORMAT

This specification details the format and semantics for the signet data format. The Dark Internet Mail Environment (DIME) uses the signet data format to transfer cryptographic information for use in encryption and signing operations. A signet carries with it signatures which must be evaluated by the consumer when determining whether to accept the validity of a signet for an organization or user identity. In addition to the required cryptographic information, a signet may be used to advertise information about the signet owner, or information used to facilitate other non-cryptographic functions commonly supported by DIME implementations.

The signet data format requires a small number of "cryptographic" fields containing public keys and signature data. Only the fields required to facilitate the sending and receiving of encrypted email are required. However the signet specification has also defines a number additional "informational" fields, whose use is entirely optional, but allows for the distribution of various fields along with a signet. These optional fields are designed to provide biographic information, facilitate optional functionality, and improve the overall user experience.

The signet data format also provides a mechanism for providing an unlimited number of "undefined" fields. Undefined fields provide an arbitrary name and data value, and like the informational fields, are entirely optional. If provided, the undefined fields may be used to carry arbitrary data items. Undefined fields may provide information which is useful in a specific context, or to facilitate functionality unrelated to DIME.

GROUPINGS

CLASSES

Signets are broken into two distinct classes: "organizational" signets, which are associated with a domain name, and "user" signets which are associated with an email address. An email address may defined as a mailbox, or local part, in combination with a domain name. Every signet carries a number of data elements organized into individual units called "fields." This specification details a number of "defined" fields, which are all associate with a unique numeric type identifier and used to carry data which conforms to the provided validation rules. Signets may also carry with them an unlimited number of "undefined" fields, which use a single numeric type identifier, with each undefined field providing its own arbitrary name and value.

TYPES

There are two signet types, with the type primarily used to communicate whether a signet consists of the cryptographic fields, or a combination of the cryptographic fields and the informational fields. A "cryptographic signet" indicates that a signet only contains the cryptographic fields, while the term "full signet" indicates both the cryptographic and informational fields are included in a signet. Since the informational fields are entirely optional, it is possible for a signet resolver to request what is technically defined as a full signet, but only receive a cryptographic signet.

MODIFIERS

The term “root signet” is a modifying term, which refers to the first cryptographic signet in a chain of custody. The class, user, and the type, cryptographic, are implied because the chain of custody is built using cryptographic signets, and because only user signets provide custody signatures. Unless specified, root signet references are for the root signet linked to a user’s current signet. The term “identifiable” is also a modifier and is used to indicate that a full or cryptographic signet includes the identity fields.

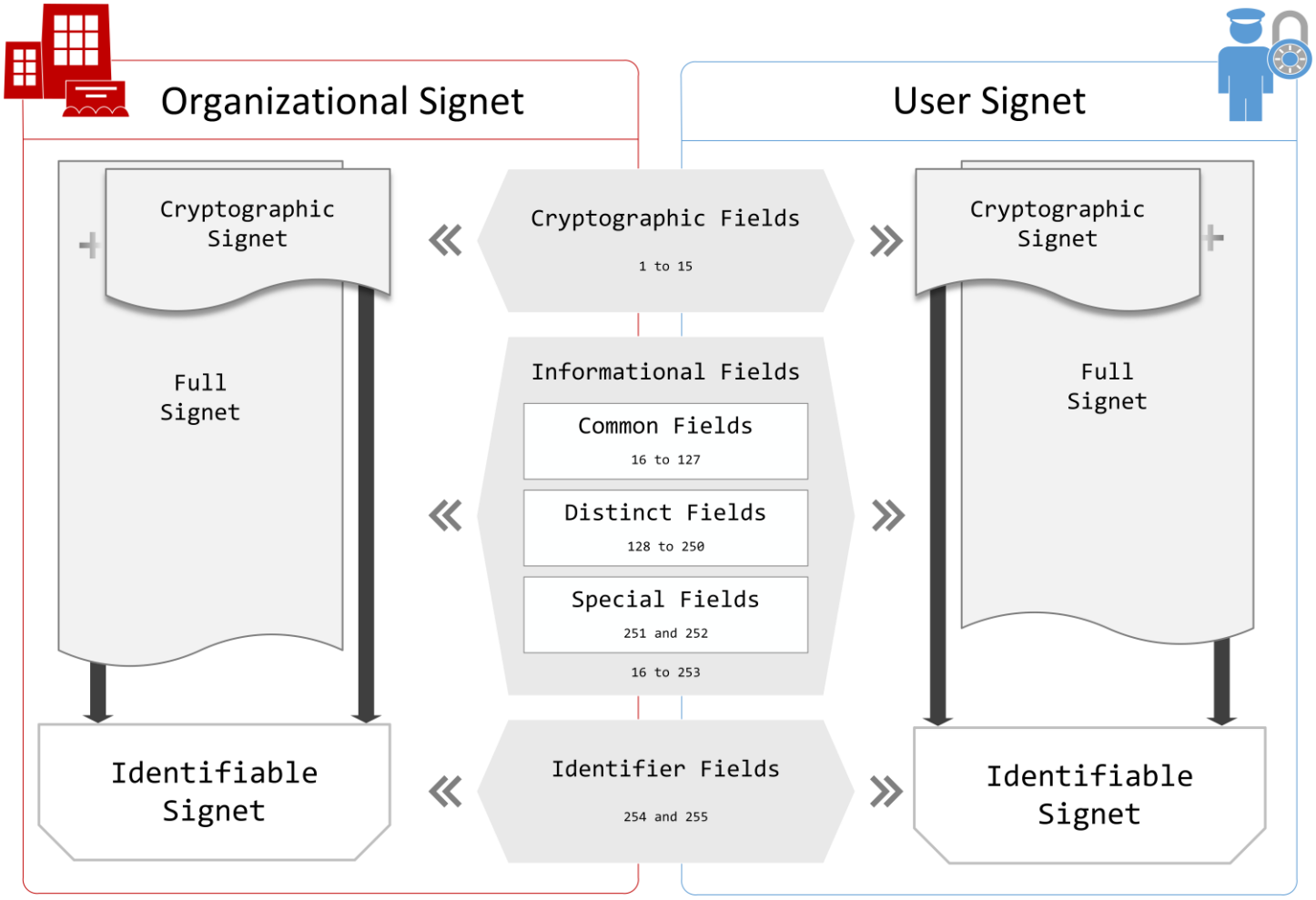


Figure 8 – Signet Groupings

CATEGORIES

Signet fields are broken up into three field categories, with each category associated with a range of numeric field identifiers. Fields always appear in order, and are broken into the categories: “cryptographic,” “informational,” and

"identity." The informational are further subdivided into three subcategories, the "common" fields, which consists of fields shared by both signet classes, the "distinct" fields, which are distinct to each signet class, and the "special" fields which are shared between the classes, but require special processing.

FIELD IDENTIFIERS

The numeric signet field types have been ordered to ensure specific data items are provided in the proper sequence. This ensures certain carefully selected to create specific security, flexibility, and functional properties. They have also been divided into ranges, to aide cognition whenever possible.

RANGES

The cryptographic fields occupy the range 1 through 15 and provide public key information, and the signatures used to assess the validity of a signet. Informational fields occupy the range 16 through 253 and are all optional. The fields 254 and 255 are the identity fields.

Field Ranges	Category
1 – 15	Cryptographic Fields
16 – 253	Informational Fields
254 - 255	Identity Fields

The informational fields are broken into 3 subcategories. Field identifiers shared by organizational and user signets are called "common" and occupy the range 16 to 127. Fields in the range 128 to 250 will always be different, or "distinct," between organizational and user signets. All of the common and distinct fields are variable length value fields, using a 2 octet length parameter.

The undefined field identifier, 251 {oxFB}, and the image field identifier, 252 {oxFC}, are grouped together in the "special" subcategory, because both fields use unique binary layouts.

RESERVED

The unused field identifiers in the cryptographic portion of a signet are reserved for future use. Parsers encountering a signet with a field type that falls outside of these ranges must reject the signet as invalid.

For organizational signets, fields 0 {0x00}, and 5 {0x05} through 15 {0x0E} are reserved for future use. All organizational signets conforming to this specification must ensure they only provide fields in the ranges: 1 {0x01} through 4 {0x04} and 16 {0x0F} through 255 {0xFF}.

For user signets, fields 0 {0x00}, and 7 {0x07} through 15 {0x0E} are reserved for future use. All user signets conforming to this specification must ensure they only provide fields in the ranges: 0x01 through 0x06 and 0x0F through 0xFF.

ORDERING

All fields within a signet must be sorted according to their single octet numeric type, and appear in then appear in ascending order. If a consumer encounters a signet which does not conform to this field order, or if a signet parser encounters a unique field multiple times, then the signet must be considered malformed. If either error occurs among the informational fields, then a parser may choose to discard the informational fields and only retain and use the cryptographic signet.

Undefined fields all carry the same numeric identifier, which is used for each undefined field a signet carries. These undefined fields should be sorted lexicographically by codepoint using the name parameter. However a consumer must not assume this sort ordering has been applied, and act accordingly when searching for a specific undefined field.

Parser implementation which encounter multiple instances of an undefined field with an identical name parameter must only return the first occurrence of the name by default. A parser may provide an alternative method of returning all of the undefined fields matching a particular name using an alternate interface, for use by functionality which explicitly expects to encounter multiple values for the same undefined name.

BINARY LAYOUTS

The signet data format is a binary schema, which relies on numeric values to convey information and facilitate parsing. The binary values defined by this specification will always use network byte order, which is defined as a big endian representation, requiring the most significant byte to be stored in the smallest address, and the least significant byte be stored in the largest address. Implementations running on little endian systems will need to convert the values to ensure proper processing. [6]

SIGNET HEADER

All signets must start with a 5-octet header. The first 2 octets provide a magic number indicating the data type. Signets conforming to this must set the first 2 octets to 1776 {0x06F0} for organizational signets, and 1789 {0x06FD} for user signets.

The remaining 3 octets are used to provide the length of the remaining binary signet data, without the 5 octets used by the object header. Since the length parameter is 3 octets, signets have a technical limitation of 16,777,220 octets or 16,777,215[7] for the signet data plus the 5 octets for the header. Signet resolvers and parser implementations conforming to this specification must be capable of handling signets up to their maximum possible size.

6 The decision to use a big endian number encoding is not final yet. We may still switch to using a little endian encoding.

7 $16,777,215 = 2^{24} - 1$

Signet parsers must reject signets with unrecognized magic numbers, and generate an error when they encounter object headers that begin with an unrecognized identifier. The signet format has been designed to allow for future revisions of this specification to add new defined fields without breaking existing implementation. As a result, the only time the magic number will be altered is when it becomes necessary to alter the cryptographic fields or make an alternation that is incompatible with the current format.

```
[  2 octet  ] [ Magic Number  ]
[  3 octets ] [ Signet Size   ]
[  variable ] [ Signet        ]
```

FIELD TYPES

Signet fields always begin with single octet numeric type identifier, and may provide values using one of three different layouts. This specification provides the correct layout for all 255 possible type values. The possible layouts are a fixed length signature field, a variable length value field, which is used by all of the defined fields which don't provide signatures, and an undefined field layout designed to provide a variable length name in addition to the variable length value.

While parsing signets conforming to signet data formats, a parser must ignore any fields with unrecognized type codes. If an unrecognized field is encountered, it must use the defined field, variable length value layout provided below. With the exception of the required cryptographic fields identified by this specification, and the image field, all other variable length value fields must use a 2 octet length parameter. This scheme will allow parsers to ignore unrecognized fields and continue processing the signet, although a parser may issue a warning message. Future signets specifications which continue using the magic numbers provided by this specification must ensure backwards compatibility.

Parsers adhering to this specification must be able to identify and validate all of the required cryptographic fields described in this revision. An implementation must also be able to parse the length all fields. Parsers should be capable of validating all of the optional field values, but must be ensure fields they properly validate the value of every field whose value is used.

SIGNATURE FIELDS

Fixed length fields are used to provide cryptographic signatures. When encountering signature field, the single octet type must always be followed by a 64 octet signature.

```
[  1 octet  ] [ Type            ]
[ 64 octets ] [ Signature Value ]
```

DEFINED FIELDS

Defined fields use a single octet to provide a field type, which is immediately followed by a length parameter that is 1, 2 or 3 octets. The scheme allows defined fields to provide a variable length value. The number of octets used by the length parameter is determined by field category. Defined cryptographic fields must use a 1 octet length parameter (unless they are a signature field, in which case they use the fixed length format above). All of the informational fields in the common

and distinct ranges, and the signet identifier field use the defined variable length value layout provided here, with a 2 octet length parameter. The image field is the only field which provides a 3 octet length parameter.

For defined fields holding a variable length value, the minimum valid length is 0, while the maximum valid length is determined by the number of octets used by the length parameter. Implementations must be able to handle a value length of 0, and must treat these fields as being the functional equivalent of omitting the field entirely.

```
[  1 octet   ] [ Type       ]
[ variable   ] [ Length     ]
[ variable   ] [ Value      ]
```

UNDEFINED FIELDS

The undefined field layout has been designed for flexibility, allowing implementations to create fields with variable length names and values. Undefined fields are indicated by the single octet type parameter, which will indicate undefined fields using the value 251. The type parameter is followed by the length of the name encoded as a single octet length value. The name value follows the length parameter and must be comprised of valid characters from the UTF-8 encoding standard. Names must always be at least 1 character in length, and should always begin with a capital letter. Name values must be constructed without the use of whitespace characters, and may use up to 255 octets. The name parameter is followed by the value length parameter, which is provided using 2 octets. Implementations must accept undefined fields with a length value of 0. The maximum length of an undefined field value is 65,535 octets. Values may include binary data, with octets of any possible 8 bit value, and signet parsing implementations must be capable of handling binary data in undefined field values.

Implementations should be capable of handling invalid undefined fields where the length of the name is 0, or where the name value includes invalid UTF-8 sequences or whitespace characters. Implementations responsible for signet creation must remove these invalid undefined fields, and consumer implementations must never use the value of an undefined field with an invalid name for any purpose. Implementations may choose whether to provide users with a warning when invalid name values are encountered.

```
            [ Min ] [  Max  ] [ Optional ]
[ Type    ] [  1  ] [   1   ] [          ]
[ Length  ] [  1  ] [   1   ] [          ]
[ Name    ] [  0  ] [ 255   ] [          ]
[ Length  ] [  2  ] [   2   ] [          ]
[ Value   ] [  0  ] [ 65535 ] [          ]
```

CRYPTOGRAPHY

SIGNING KEYS

All public signing keys must begin with the value 64 {0x40}, and then be followed by 32 octets which represent a Ed25519 public key. [PGP-EdDSA] The public key must be in the compressed little endian format defined by the Ed25519

paper and used by the Ed25519 reference implementation [EdDSA]. A decompressed public key must represent a valid point on the Twisted Edwards curve:

$$x^2 + y^2 = 1 (121665/121666)x^2y^2$$

ENCRYPTION KEYS

All encryption key fields must provide an uncompressed public key, which represents a valid point on the secp256k1 curve. Public encryption keys are provided using an uncompressed format, where the two coordinates representing the point, or P = (x, y), are provided as concatenated big endian integers. Each coordinate must be aligned to an 8 bit boundary, and prefixed with the a format identifier consisting of a single octet with a value of 4 {0x04} [PGP-ECC]. This may also be expressed as K_{pub} = 0x04 || x || y, with the value K_{pub} represents a serialized public key provided with a signet.

SIGNATURES

When encountering a signature field, it is important to note precisely what data must be used to generate, and then validate a signature. A given signature must always be taken over every field which precedes it. The fields must be provided in ascending order, according to their numeric field identifier, and the native binary encoding form. This description encompasses all of the octets which precede a signature, including field identifiers and length parameters, with two exceptions: the 5 octets used by the signet object header, and the 1 octet used by a signature's field identifier. Note this exception only applies to the current signature field. If a signet includes signatures with lower numeric field identifiers, they are included in their entirety in the current signature.

All signatures must be created and validated using the EdDSA[8] algorithm and the Twisted Edwards curve:

$$x^2 + y^2 = 1 (121665/121666)x^2y^2$$

This particular Twisted Edwards curve is more commonly known by the colloquial name Ed25519, and is birationally equivalent to the Montgomery curve colloquially known as Curve25519[9] [EdDSA]. This signing algorithm generates two parameters as output, the R and S, which are compressed into 32 octets each, using a little endian encoding scheme defined in the EdDSA paper and used by the Ed25519 reference implementation. These parameters are expressed as SIG = (R || S), where SIG represents the 64 octet signature value supplied by the signet signature fields.

8 Specifically, the parameters for EdDSA are: b = 256; H is SHA2-512; q is the prime $2^{255} - 19$; the 255 bit encoding of $F_{2^{255}-19}$ is the usual little endian encoding of { 0, 1, ..., $2^{255} - 20$}; ℓ is the prime 2^{252} + 27742317777372353535851937790883648493; d = -121665/121666 $\in F_q$; and B is the unique point (x, 4/5) $\in$ for which x is positive. This collection of parameters is known colloquially as Ed25519-SHA2-512. [EdDSA]

9 Curve25519 is the Montgomery curve $v^2 = u^3 + 486662u^2 + u$ over the same field. The equivalence is $x = \sqrt{486664}u/v$ and y = (u - 1)/(u + 1) [EdDSA]

SPLITTING

A signet may carry up to 3 organizational signatures. These signatures are provided between each of the field categories, and are generated using all of the signet fields which precede them. This allows a signet resolver, or Privacy Agent (PA) to split a signet, and extract a subset of the fields, while a retaining signet object that remains cryptographically verifiable.

A full signet may be extracted from an identifiable full signet by removing the identity fields. Alternatively a cryptographic signet is obtainable from a full signet by removing the informational fields.

Signet resolvers may choose to split full signets and only store the cryptographic signet when encountering excessively large signets. Alternatively, a PA may split stale signets and retain only the cryptographic signets when encountering a new organizational or user signet.

Signet resolvers should store at least the cryptographic signet in a user's Signet Ring to facilitate chain of custody verifications in the future. Storing the full signet, along with storage of the identity fields is optional.

Key Servers (KS) must ensure they store all of a user's cryptographic signets between a user's current root signet and their current signet. This information is need by the DMTP signet history command (see) and is used by a signet resolver to validate the chain of custody between a stored user signet, and a current user signet.

FINGERPRINTS

Several different types of signet fingerprints are used by different aspects of the system. All fingerprints are generated using the SHA2-512 algorithm. A cryptographic fingerprint is used to retrieve a specific signet or verify that a stored signet is still current. The root and ephemeral fingerprints may be used to manually verify a signet using an alternate communications channel.

CRYPTOGRAPHIC

A cryptographic fingerprint is used by the DIME protocols and formats to identify a specific signet. This is the type of fingerprint used by the DIME protocols and supplied in the envelope of an encrypted D/MIME message. Unless otherwise noted, any reference to a fingerprint will be to a cryptographic fingerprint.

A cryptographic fingerprint is formally defined as a SHA2-512 hash of a cryptographic signet. The fingerprint is taken over all of the fields which make up a cryptographic signet in their binary encoding form. This includes the signature fields. Only the 5 octet signet object header must be omitted from a cryptographic fingerprint.

EPHEMERAL

An ephemeral fingerprint may be used by two people in real-time contact, and is generated by concatenating the full signets for both users, and then generating a SHA2-512 hash over the combination in its native binary form. Only the signet object header and the identity fields are omitted.

To determine which order the full signets should be combined, an implementation must generate a cryptographic fingerprint for both users, then order them based on the numeric value of the fingerprint output. This ordering is then applied to the full signets to calculate the ephemeral fingerprint.

ROOT

A root fingerprint is specific to user signets, and refers to the cryptographic fingerprint of the first signet a user's chain of custody. This is the fingerprint that a user may supply to others, and is suitable distribution using a static medium like paper, or posted on a user's personal website. A root fingerprint is suitable for asynchronous manual verification because it will remain unchanged until there is a break in a user's chain of custody.

An individual may verify the current signet for a user by retrieving root signet for a given email address, confirm the root fingerprint matches the supplied fingerprint, and then validate the chain of custody between the manually verified root signet and a user's current signet.

CRYPTOGRAPHIC SIGNETS

A cryptographic signets represents the smallest valid form of signet object, and is a subcomponent of all other signet forms (full and identifiable). Cryptographic signets are also incredibly small which makes retrievals fast, and storage efficient. The typical cryptographic organizational signet will be less than 256 octets, while most cryptographic user signets will be less than 512 octets.[10]

A cryptographic signet is compromised of the required fields needed to support the end-to-end to encryption of email messages. This requires providing public signing and encryption keys along with the signatures required to assess their validity.

ORGANIZATIONAL SIGNETS

The cryptographic fields for an organizational signet occupy the range 1 {0x01} to 4 {0x04} and are used to provide the relevant encryption and signing keys associated with a domain name.

Field	Label	Status	Multiples	Type
1	Primary-Organizational-Key	Required	No	Signing Key
2	Secondary-Organizational-Key	Optional	Yes	Signing Key
3	Encryption-Key	Required	No	Encryption Key
4	Organizational-Signature	Required	No	Signature

10 Organizational signets could include multiple secondary signing keys, and user signets could include alternate encryption keys. This would increase their size beyond 256, and 512 octets respectively, although the size would likely remain small.

PRIMARY ORGANIZATIONAL KEY

Provides the Primary Organizational Key (POK) associated with a domain name, which is a 32 octet compressed Ed25519 public key, prefixed with a 1 octet format identifier. The private key associated with a POK value must also be used to the organizational signatures signet, and a consumer must independently validate all of the signatures provided by an organizational signet using the supplied POK value. A consumer must also ensure the POK value supplied by an organizational signet matches 1 of the POK field values in the management record.

The Primary Organizational Key (POK) is authorized for all organizational signing operations. If an organization chooses to use their POK for signing user signets and outbound messages, then consumers may validate the signatures using a DNS query, without retrieving the organizational signet.

When displaying the value of this field, the label "Primary-Organizational-Key" should be used and the key encoded using base64.

```
[  1 octet  ] [ Type              ]
[  1 octet  ] [ Length            ]
[  1 octet  ] [ Format Identifier ]
[ 32 octets ] [ Public Key        ]
```

The following is an Ed25519 public key, provided in hexadecimal form:

```
Qpub: 0x3f098994bdd916ed4053197934e4a87c80733a1280d62f8010992e43ee3b2406
```

If this same public key was supplied as the POK value by an organizational signet, it would appear as the following base64 value:

```
Primary-Organizational-Key: ASFAPwmJlL3ZFu1AUxl5NOSofIBzOhKA1i+AEJkuQ+
47JAY
```

SECONDARY ORGANIZATIONAL KEY

The Secondary Organizational Key (SOK) field provides an alternative signing key, along with a flags octet to indicate which potential signing functions the key is authorized to perform. The value holds a single octet flags parameter, followed by an Ed25519 public key. The SOK field is the only cryptographic field which may be included in the cryptographic portion of an organizational signet multiple times.

The first octet for this field provides the permissions octet. This octet contains a collection of bit positions, which if enabled indicate the appropriate operation is authorized. At least one of the first 3 bit positions must be enabled. If any of the reserved flags have been enabled, the field value must be ignored and any associated signature verification operations must fail.

```
[  1 octet  ] [ Type              ]
[  1 octet  ] [ Length            ]
[  1 octet  ] [ Permissions       ]
[  1 octet  ] [ Format Identifier ]
```

```
[ 32 octets ] [ Public Key          ]
```

The bits in the permissions octet authorize the secondary key to sign the listed data types:

```
[   1 ] [ 0x01 ] [ User Signets                          ]
[   2 ] [ 0x02 ] [ Outbound Messages                     ]
[   4 ] [ 0x04 ] [ TLS Certificate                       ]
[   8 ] [ 0x08 ] [ Software                              ]
[  16 ] [ 0x0F ] [ Reserved                              ]
[  32 ] [ 0x20 ] [ Reserved                              ]
[  64 ] [ 0x40 ] [ Reserved                              ]
[ 128 ] [ 0x80 ] [ Reserved                              ]
```

The following is an Ed25519 public key, provided in hexadecimal form:

```
Qpub: 0x3f098994bdd916ed4053197934e4a87c80733a1280d62f8010992e43ee3b2406
```

When displaying the value of this field, the label "Secondary-Organizational-Key" should be used. If this public key was supplied as a SOK value in an organizational signet, and the SOK value was authorized to sign user signets and outbound messages, it would become the following base64 value:

```
Secondary-Organizational-Key: AiIDQD8JiZS92RbtQFMZeTTkqHyAczoSgNYvgBCZ
LkPuOyQG
```

ENCRYPTION KEY

The encryption key field is used to provide an uncompressed secp256k1 public key. Once an encryption key has been published as part of an organizational signet, the corresponding private key will be required to access the envelope information for any D/MIME messages handled by the organization's mail servers.

The encryption key field provides its public key in an uncompressed format, with the point values that make up the point P = (x, y), provided as two concatenated big endian numbers, aligned to the 8 bit boundary, and prefixed with the format identifier 0x04 [PGP-ECC]. The complete public key value is expressed as B = 0x04 || x || y, with the value of B representing a serialized public key.

The following is a secp256k1 public key, with the values of X and Y provided in hexadecimal form:

```
X = 0x6df18fcf75f52c09bd7cb0d56d601ff404a8d2fa610f127c21f51e4bea6233d1
Y = 0x362c92d78981499d09b2102fe7f8a227dd551e23aea5ff396235bf14af0749b6
```

When displaying the value of this field, the label "Encryption-Key" should be used, and the binary data encoded as a base64 string. If the public key provided above were display, it would appear as:

```
Encryption-Key: A0EEbfGPz3X1LAm9fLDVbWAf9ASo0vphDxJ8IfUeS+piM9E2LJLX
iYFJnQmyEC/n+KIn3VUeI66l/zliNb8UrwdJtg
```

ORGANIZATIONAL SIGNATURE

The organizational signature field provides a 64 octet signature, generated using the signet fields 1 through 3, and the private portion of the POK. This signature allows a full organizational signet to be split, and the cryptographic signet extracted, while retaining in a form signet form which can still be cryptographically verified.

USER SIGNETS

The first 6 user signet fields make up the cryptographic signet, and provide all of the necessary public keys support the message encryption functionality provided by DIME. The field definitions are:

Field	Label	Status	Multiples	Type
1	Signing-Key	Required	No	Signing Key
2	Encryption-Key	Required	No	Encryption Key
3	Alternate-Encryption-Key	Optional	No	Encryption Key
4	Custody-Signature	Required	No	Signature
5	User-Signature	Required	No	Signature
6	Organizational-Signature	Required	No	Signature

SIGNING KEY

Must provide a valid 32 octet Ed25519 public key in compressed little endian form (see). The corresponding private key is used to generate a self-signature, which is included in a signet signing request, and becomes the user signature field (see). The corresponding private key must also be used to create the chain of custody signature when the signet is rotated (see), and for generating the full and tree signatures included with outbound messages. When displaying the value of this field, the label "Signing-Key" should be used and the key information encoded using base64.

ENCRYPTION KEY

Must provide a valid public key representing a point on the secp256k1 curve and prefixed with the format identifier 4 {0x04} (see). The corresponding private key will be needed to access messages encrypted to this signet, as described in the next chapter. When displaying the value of this field, the label "Encryption-Key" should be used and the key converted into a base64 string.

ALTERNATE ENCRYPTION KEY

Alternate key fields must always begin with two octets, the first provides security level claims for the alternate encryption key, while the second indicates which alternate encryption cipher suite the key should be used with.

The first octet, which provides an indication of what security level applies to the alternate key. These claims must be treated as advisory unless the following exception is applicable. This because when a UPA is retrieving a signet from a foreign source, it has no way of determining, verifiably, whether the claimed security level is accurate. A policy of requiring accurate security level claims within user signet signing requests is recommended for all Key Service (KS) implementations.

This revision specifies two alternate ciphersuites. Which suite the provided value should be used with is indicated by the second octet. Currently the value 1 {0x01} indicates an alternate secp256k1 public key, while a value of 2 {0x02} indicates a public key on the curve colloquially known as Curve41417 [DANGER]. The values 192 {0xC0} through 239 {0xEF} must be used by non-standard, or experimental ciphersuites. The values 240 {0xF0} through 255 {0x255} must never be used. A cryptographic signet which supplies an alternate encryption key where the second octet in the reserved range should be considered invalid, and a warning must be generated, which a user may elect to ignore.

The remaining octets are used to store actual key material, and are dependent upon the indicated ciphersuite. When displaying the value of this field, the label "Alternate-Encryption-Key" should be used.

SECURITY LEVELS

When a user signet claims a security level for an alternate encryption key, the information must be treated with skepticism, used carefully, and considered only as an advisory. The exception is when the author and recipient belong to confined group where the members are trusted to provide accurate security level claims, or they belong to the same organization and the KS is trusted to ensure a signet provides authentic clearance level claims. For consumers where this exception does not apply, there is no guarantee a user's signet is reporting an accurate security level. This policy may considered useful inside organizations, or confined groups, which handle particularly sensitive materials and want to allow users to force the specialized handling with enhanced security precautions for specific messages.

Security levels, even in an advisory role, may provide guidance to authors, and allow them to make informed decisions about the sensitivity of any materials sent. A security level octet uses a bit mask to the values. Only most significant security level should be considered relevant. The currently defined security levels for private keys are:

```
[  0 ] [ 0x00 ] [ Unprotected                                          ]
[  1 ] [ 0x01 ] [ Sensitive                                            ]
[  2 ] [ 0x02 ] [ Secret                                               ]
[  4 ] [ 0x04 ] [ Top Secret                                           ]
[  8 ] [ 0x08 ] [ Top Secret // Special Access                         ]
[ 16 ] [ 0x16 ] [ Top Secret // Extremely Compartmented
                  Information // Special Access                        ]
```

The definitions for these security levels is:

- **Unprotected.** Server side encryption, trustful account mode, requires absolute trust in the service provider.
- **Sensitive.** Client side encryption, cautious account mode, but thin and thick clients are supported, allowing for web access.
- **Secret.** Client side encryption, cautious account mode, thin client support is disabled for content protected by an alternate encryption key, mandating that a thick client be used to access the attachment and display sections of a message.

- **Top Secret.** Client side encryption, cautious account mode, thin client support is disabled, mandating that a thick client is always used, multiple devices are allowed.
- **Top Secret // Special Access.** Client side private key storage, paranoid account mode, mandates that a single thick client is always used.
- **Top Secret // Extremely Compartmented Information // Special Access.** Hardware security module must be used for key storage and encryption, paranoid account mode, mandates the use of a singular purpose built access device.

SPECIAL ACCESS

The remaining bits in a security level octet are used to indicate a private key is used with special access program. The recommended policy is for implementations to limit the use of special access program claims, and only allow signets with these bits enabled when an alternative encryption key has a Top Secret // Special Access clearance or higher. We also recommend that client implementations only consider the special access bits if the appropriate Top Secret // Special Access has been indicated, and if the signet conforms to the exception detailed above regarding the trustworthiness of security level claims. The currently defined special access program labels are:

```
[  32 ] [ 0x20 ] [ Special Access // Yankee White          ]
[  64 ] [ 0x40 ] [ Special Access // Shadow Hunter         ]
[ 128 ] [ 0x80 ] [ Special Access // Underclass Appelbaum ]
```

CUSTODY

When rotating a user signet, this field must contain a 64 octet Ed25519 signature for the first 3 user signet fields. The signature must use the previous signing key when generating this signature. If this is the first user signet ever created, or if the private signing key for the previous signet is unavailable, this field must be omitted. Consumers must reject a signet if the value supplied by this field is invalid. If the field is missing, or empty, then a signet resolver must issue a security error if a previous user signet is stored in a user's signet ring. If a previous user signet is available, and this field is populated with a signature, then a resolver must independently validate the newly retrieved signet contains a valid chain of custody signatures linking the stored signet with the current one (see). When displaying the value of this field, the label "Custody" should be used and the signature information encoded using base64.

USER SIGNATURE

Must provide a 64 octet Ed25519 signature for the binary data stream comprising the first 4 fields in the user signet, which validates using the public Ed25519 signing key stored in field 1 (Signing-Key). Consumers must reject a signet if the value supplied by this field is invalid. When displaying the value of this field, the label "User-Signature" should be used and the signature information encoded using base64.

ORGANIZATIONAL SIGNATURE

Must provide a 64 octet Ed25519 signature for the binary data stream comprising the first 5 fields in the user signet, which validates against the Ed25519 POK, or an authorized SOK found in the associated organizational signet. Consumers

must reject a signet if the value supplied by this field is invalid. When displaying the value of this field, the label "Organizational-Signature" should be used and the signature information encoded using base64.

FULL SIGNETS

A full signet includes a cryptographic signet, plus the optional informational fields. Unlike the required cryptographic fields which are strictly defined, parsers must ignore unrecognized informational field types. Assuming a full signet is syntactically and structurally valid, a parser should also ignore fields any invalid content, and use only the valid field values. All informational fields, except for the last 3, use the variable length value field type, with a 2 octet length parameter.

This also means that any unrecognized informational field types must also use the variable length value field type and provide a 2 octet length parameter. This scheme guarantees backwards for any signet object which provides a magic number defined by this specification. Signet parsers must be capable of separating semantics from syntax, which means they must be capable of parsing out the length of an unrecognized or unsupported informational field and advancing over its value. This allows future revisions to add informational fields which are semantically ignored.

The informational fields provided with a full signet have been divided into the following ranges (see *and*):

COMMON FIELDS

The two signet types (user and organizational signets) include fields that are common between them. Implementations must not vary the use of these common fields between the user and organizational signet formats. The following table lists the currently defined common fields:

Field Identifier	Label	Status	Multiples	Type
16	Name	Optional	No	Text
17	Address	Optional	No	Text
18	Province	Optional	No	Text
19	Country	Optional	No	Text
20	Postal-Code	Optional	No	Text
21	Phone	Optional	No	Text
22	Language	Optional	No	Text
23	Currency	Optional	No	Text
24	Cryptocurrency	Optional	No	Text
25	Motto	Optional	No	Text
26	Website	Optional	No	Text
32	Message-Size-Limit	Optional	No	Text

The common fields defined above are described below. These descriptions will not be repeated in the class specific, distinct field sections.

NAME

Should provide a UTF-8 string of characters containing an organization or user's preferred name, as they want it presented. If applicable, when displaying this field, the label "Name" should be used.

ADDRESS

Should provide a UTF-8 string of characters corresponding to an organization, or user's physical address. When displaying the value of this field, the label "Address" should be used.

PROVINCE

Should provide UTF-8 string of characters corresponding to an organization or user's province, or the principal administrative division of the signet owner's country. This is more commonly called the state, region, territory, district, or canton depending on the locale. The contents of this field should not be abbreviated. When displaying the value of this field, the label "Province" should be used, unless the client is sophisticated enough to consider the locale and supply the appropriate colloquial term.

COUNTRY

Should provide a UTF-8 string of characters corresponding to an organization or user's country. The contents of this field should not be abbreviated.[11] When displaying the value of this field, the label "Country" should be used.

POSTAL CODE

Should provide a UTF-8 string of characters corresponding to an organization or user's postal code. The contents of this field should not be abbreviated. When displaying the value of this field, the label "Postal-Code" should be used.

PHONE

Should provide an organization or user's phone number, multiple values may be supplied, separated by semicolons. An optional identifier may appear at the beginning of a value. If an identifier is provided it must be terminated by the colon {0x3A} symbol. Identifiers must not exceed 16 UTF-8 characters, must not include a colon or semicolon, or a white space character. Identifiers which violate these rules should be ignored, and may result in the specific phone number, or the entire field being ignored. If the identifier is invalid, or missing, a default identifier value of "Phone" should be used.

The phone number field parameter may begin with a plus "+" {0x2B} symbol to indicate an international phone number, and should be followed by a complete the dialing prefix, country code, and phone number. If the phone number does not begin with a plus "+" {0x2B} symbol, it must be a national number. When providing a national number, a portion of the

11 The suggestion has been made to make this a 2 or 3 letter country code and use the alpha-2 or alpha-3 list to determine the actual country name. This would limit the number of recognized countries and require implementations to translate the country code into the local language, but would make it possible to programmatically recognize the value. The jury is still out on this.

leading digits may be enclosed inside parentheses "( )" {0x28} and {0x29} to indicate they digits are a trunk prefix and may only be required when calling from a different trunk code. [E123]

The plus "+" {0x2B} symbol may only be used by a phone parameter once, and must be the first character supplied. Parentheses "( )" {0x28} and {0x29} may only be used when the plus "+" {0x2B} symbol is absent. If the opening parentheses "(" {0x28} is used, it must be followed by one or more digits and a closing parentheses ")" {0x29}. The parentheses "( )" {0x28} and {0x29} sequence may only be used once in each phone parameter.

A sample phone field, without an identifier, which supplies a national phone number would appear as:

```
4108546334
```

While the same phone number could also appear as:

```
(410)8546334
```

If this number was provided with an identifier and a second phone value was also provided with an identifier, it could appear as:

```
DIRECT:(410)8546334;OFFICE:2024561414
```

Note the first phone parameter supplies the optional parentheses while the second value does not.

LANGUAGE

Should provide a string of characters containing an organization or user's preferred language identifier in order of preference.[12] A semicolon terminates the string value and provides an optional separator if multiple values have been provided. The string which follows the semicolon should be considered a secondary language identifier and used if the preceding value is unsupported. The sequence may repeat until either the signet owner's list of preferred languages is exhausted or the length limit for the field value is reached. When displaying the value of this field, the label "Language" should be used.

The string values provided by this field should appear in form of a language tag, optionally followed by a subtag, which is typically used to indicate the specific country or region. The language and subtag values must be separated by a dash. [LANGUAGE] Convention dictates that language tags should be provided in lowercase form [ISO639-1], script codes in lowercase form but with the first letter capitalized [ISO15924] while regional and country subtags should be provided using all uppercase letters. [ISO3166-1]

The value may be used to select a signet owner's preferred language. The value also provides guidance when formatting dates, times, numbers and currency amounts. When a consumer encounters multiple language identifiers, it should select

12 The language field value is sometimes referred to the "locale" by other protocols and standards.

the first fully supported value it encounters. If none of the identifiers are fully supported, a consumer should examine the list a second time, and discard the subtag when making comparisons, considering only the language identifier. The first supported language it encounters should be selected. If this field is missing, and a signet has supplied a value for the field "Country", then its value may be considered as an alternative.

For user signets where the Language and Country fields are missing, invalid or their values unsupported, a consumer may fall back to considering the associated organizational signet using the same logic described above. If all of the preceding logic fails to yield a supported language identifier, then implementations may also consider the organizational domain and apply the appropriate language preference the domain belongs to a regional or country specific Top-Level-Domain (TLD). If an implementation is unable to determine the language preference, the default value "en-US" should be used.

Implementations should recognize all 2 character language identifiers established by the ISO 639-1 standard. [ISO639-1] Support for the more comprehensive 3 character language tags established by ISO 639-2 [ISO639-2], the 4 character script codes established by ISO 15924 [ISO15924], location subtags is optional.

Language tags should be selected and interpreted using the ISO 639-2 registry maintained by the Library of Congress (LOC) [LOC-LANG], while location subtags, if supported, should be interpreted using the registry maintained by the Internet Assigned Numbers Authority (IANA). [IANA-LANG]

The following value would indicate the language is English and the country is the United States of America:

```
en-US
```

The following, more complicated example, indicates a preference for English, localized for the United States of America, followed by any available English representation, then any German representation, and finally by any available French representation.

```
en-US;en;de;fr
```

CURRENCY

A sequence of 3 uppercase characters [ISO4217] which correspond to the code used by a signet owner's preferred form of currency. A semicolon terminates the value, and serves as an optional separator, allowing for multiple currency codes to be supplied, based on the signet owner's order of preference. Currency codes should be in preferential order, with the most preferred appearing first, and the least preferred appearing last. Only 3 character codes should be used, and they should be interpreted using the A.1 currency codes table maintained by the Swiss Association for Standardization (SNV). [SNV-CURRENCY] Values which are not included in the A.1 table, or which include an invalid character should be ignored. Matching 3 character codes with lower case characters is optional.

If the currency field is missing, or the supplied values are invalid, then the following should be applied as the default currency preference ordering: "USD;EUR;CHF;GBP;JPY;CAD;AUD;CNY;NZD;RUB;BRL;MXN" which was derived by the major independent national currencies based on trading volume, the size of the country's population, along with relative

strength and stability. This default preference ordering may be overridden if the value of the Country signet field is recognized, and is associated with a national currency.

When displaying the value of this field, the label "Currency" should be used. The following currency field example indicates a preference for United States Dollars, followed by Swiss Francs, European Euros, and finally Indian Rupees:

```
USD;CHF;EUR;INR
```

CRYPTOCURRENCY

A UTF-8 string which should correspond to a signet owner's preferred cryptographic currency. The 3 character cryptocurrency identifying type must be separated from the payment address information by a colon. A semicolon terminates the cryptocurrency string, and provides an optional separator. If a UTF-8 string follows the semicolon, it should be interpreted as a less preferred cryptocurrency type presented in the same identifier, colon, value form. The sequence may repeat until the signet owner's list of preferred cryptocurrencies is exhausted or the length limit for the field value is reached. If the field value includes an invalid UTF-8 codepoint, the entire field must be ignored, otherwise if an individually delimited value is provided without an identifier, or if the identifier is invalid/unrecognized, only the specifically delimited value should be ignored. When displaying the value of this field, the label "Cryptocurrency" should be used.

Support for this field is optional, but if an implementation does support it, then the following defined cryptocurrency symbols must be recognized:

Symbol	Name	Website
BLK	Blackcoin	https://www.blackcoin.co/
BTC	Bitcoin	https://bitcoin.org/
DRK	Darkcoin	https://www.darkcoin.io/
LTC	Litecoin	https://litecoin.org/
PPC	Peercoin	http://www.peercoin.net/
STR	Stellar	https://www.stellar.org/
XMR	Monero	https://monero.cc/
XRP	Ripple	https://ripple.com/currency/

In the event this field is empty, a consumer should assume the preferred cryptocurrency is Bitcoin. The value of this field should match the following example:

```
BTC:19gy9ifMJuHoRbVpXBgtf6NTAT6PiDb8SQ
```

MOTTO

A UTF-8 string of characters corresponding to a signet owner's motto or vision statement. When displaying the value of this field, the label "Motto" should be used. The value should be less than 256 UTF-8 characters. Parsers have the option of truncating the value at 256 characters. The following is a possible motto value, provide as an example:

```
For protection from the forces of evil. Use as directed.
```

WEBSITE

A UTF-8 string of characters corresponding to a signet owner's website. The value for this field must be a valid Hypertext Transfer Protocol (HTTP) Universal Resource Locator (URL) or HTTP Secure (HTTPS) URL. If the field does not contain a valid HTTP or HTTPS value, it must be ignored. The URL should use HTTPS, although this requirement remains optional. When displaying the value of this field, the label "Website" should be used.

MESSAGE SIZE LIMIT

A number, represented as a string of digits in text form, with values between 0 {0x30} and 9 {0x39}. The string represents the size limit for incoming messages. When provided by an organizational signet, the value applies to all of the email addresses associated the signet. This includes the target domain, and depending on the subdomain policy in the management record, any subdomains that might exist. If the field is provided in a user signet, the size limit only applies to individual email address associated with the signet.

The minimum legal value is 1 megabyte (aka mebibyte). When organizational and user signets both provide legal values for this field, then the smaller of the two values takes precedence. If the field contains a character which is not in the range {0x30} through {0x39}, if the numeric value is less than 1,048,576, or if the value is greater than 4,294,967,295, it should be ignored. When displaying the value of this field, the label "Message-Size-Limit" should be used.

DISTINCT ORGANIZATIONAL FIELDS

The following table lists the defined fields which apply only to organizational signets.

Field Identifier	Label	Status	Multiples	Type
128	Contact-Abuse	Recommended	No	Text
129	Contact-Admin	Recommended	No	Text
130	Contact-Support	Recommended	No	Text
131	Web-Access-Host	Recommended	No	Text
132	Web-Access-Location	Recommended	No	Text
133	Web-Access-Certificate	Optional	No	Text
134	Mail-Access-Host	Recommended	No	Text
135	Mail-Access-Certificate	Optional	No	Text
136	Onion-Access-Host	Optional	No	Text
137	Onion-Access-Certificate	Optional	No	Text
138	Onion-Delivery-Host	Optional	No	Text
139	Onion-Delivery-Certificate	Optional	No	Text
200	Services	Optional	No	Text

CONTACT ABUSE

A UTF-8 string corresponding to the email address for the organization's abuse contact. If this field is omitted the mailbox name “abuse” is combined with the organizational domain name for the signet to derive an abuse contact. This field must provide a value, or an organization must capable of receiving complaints using the default address. When displaying the value of this field, the label “Contact-Abuse” should be used.

CONTACT ADMIN

A UTF-8 string corresponding to the email address for the organization's administrative contact. When displaying the value of this field, the label “Contact-Admin” should be used.

CONTACT SUPPORT

A UTF-8 string corresponding to the email address for the organization's support contact. When displaying the value of this field, the label “Contact-Support” should be used.

WEB ACCESS HOST

A UTF-8 string of characters corresponding to the DNS name (not IP address) of the web access hostname which offers Hyper Text Transfer Protocol Secure (HTTPS) and provides web based access to user email accounts. A semicolon terminates the hostname string, and provides an optional separator. The UTF-8 string which follows the semicolon should be considered a second web access hostname. The sequence may repeat until either the list of web access hostnames is exhausted or the length limit for the field value is reached. The final web access hostname may terminate with a semicolon, but its inclusion is optional. When displaying the value of this field, the label “Web-Access-Host” should be used.

WEB ACCESS LOCATION

A UTF-8 string of letters or numbers corresponding to a HTTPS resource location for the organizational webmail system. When displaying the value of this field, the label “Web-Access-Location” should be used.

WEB ACCESS CERTIFICATE

A base64 string which provides the encoded Ed25519 signature for the TLS certificate supplied by the web access host over HTTPS. A semicolon terminates the TLS certificate signature string, and provides an optional separator. The base64 string which follows the semicolon should be considered a second base64 TLS certificate signature. The sequence may repeat until either the list of valid TLS certificate signatures is exhausted or the length limit for the field value is reached. The final base64 TLS certificate signature may terminate with a semicolon, but its inclusion is optional. When displaying the value of this field, the label “Web-Access-Certificate” should be used.

MAIL ACCESS HOST

A UTF-8 string of characters corresponding to the DNS name (not IP address) of the mail access hostname which offers connectivity using the Dark Mail Access Protocol (DMAP). A semicolon terminates the hostname string, and provides an optional separator. The UTF-8 string which follows the semicolon should be considered a second mail access hostname.

The sequence may repeat until either the list of mail access hostnames is exhausted or the length limit for the field value is reached. The final mail access hostname may terminate with a semicolon, but its inclusion is optional. When displaying the value of this field, the label "Mail-Access-Host" should be used.

MAIL ACCESS CERTIFICATE

A base64 string which provides the encoded Ed25519 signature for the TLS certificate supplied by the mail access host for DMAP connections. A semicolon terminates the TLS certificate signature string, and provides an optional separator. The base64 string which follows the semicolon should be considered a second base64 TLS certificate signature. The sequence may repeat until either the list of valid TLS certificate signatures is exhausted or the length limit for the field value is reached. The final base64 TLS certificate signature may terminate with a semicolon, but its inclusion is optional. When displaying the value of this field, the label "Mail-Access-Certificate" should be used.

ONION ACCESS HOST

A UTF-8 string of characters corresponding to the onion hostname for mail access. A semicolon terminates the hostname string, and provides an optional separator. The UTF-8 string which follows the semicolon should be considered a second onion access hostname. The sequence may repeat until either the list of onion access hostnames is exhausted or the length limit for the field value is reached. The final onion access hostname may terminate with a semicolon, but its inclusion is optional. When displaying the value of this field, the label "Onion-Access-Host" should be used.

ONION ACCESS CERTIFICATE

A base64 string which provides the encoded Ed25519 signature for the TLS certificate supplied by the onion access host. A semicolon terminates the TLS certificate signature string, and provides an optional separator. The base64 string which follows the semicolon should be considered a second base64 TLS certificate signature. The sequence may repeat until either the list of valid TLS certificate signatures is exhausted or the length limit for the field value is reached. The final base64 TLS certificate signature may terminate with a semicolon, but its inclusion is optional. When displaying the value of this field, the label "Onion-Access-Certificate" should be used.

ONION DELIVERY HOST

A UTF-8 string of characters corresponding to the onion hostname for mail delivery and signet lookups using the Dark Mail Transfer Protocol (DMTP). A semicolon terminates the hostname string, and provides an optional separator. The UTF-8 string which follows the semicolon should be considered a second onion delivery hostname. The sequence may repeat until either the list of onion access hostnames is exhausted or the length limit for the field value is reached. The final onion delivery hostname may terminate with a semicolon, but its inclusion is optional. When displaying the value of this field, the label "Onion-Delivery-Host" should be used.

ONION DELIVERY CERTIFICATE

A base64 string which provides the encoded Ed25519 signature for the TLS certificate supplied by the onion delivery host. A semicolon terminates the TLS certificate signature string, and provides an optional separator. The base64 string which

follows the semicolon should be considered a second base64 TLS certificate signature. The sequence may repeat until either the list of valid TLS certificate signatures is exhausted or the length limit for the field value is reached. The final base64 TLS certificate signature may terminate with a semicolon, but its inclusion is optional. When displaying the value of this field, the label "Onion-Delivery-Certificate" should be used.

SERVICES

Provides a semicolon delimited list of domain level services and protocol extensions support by an organization. The services field is designed to allow the advertisement of alternate communication protocols or extend the mail protocols beyond what is defined by this specification.

The text field should provide a semicolon, ";" {ox3b}, delimited list of four character values corresponding to recognized identifiers. The final value may terminate with a semicolon, ";" {ox3b}, but its inclusion is optional. For example, an organizational domain which supports the Extensible Messaging and Presence Protocol (XMPP) [XMPP] may advertise this protocol using the services value:

```
XMPP
```

The colon, ":" {ox3a} character, may be used to delineate dependent identifiers within a single value. The complete value string should be unique, although a protocol identifier may be repeated using a different, unique, collection of dependent identifiers. For example, if an organizational domain supported multi-user chatrooms [XMPP-CHAT], along with end-to-end for instant messages (using Off-The-Record (OTR) [OTR]), the services field would be:

```
XMPP;XMPP:OTR;XMPP:XEP0045
```

Individual identifiers should all be 3 to 6 uppercase letters, although if a value exceeds 7 uppercase characters, the entire identifier must be compared as a single token. If any of the identifiers included with a value are unrecognized, the entire value should be ignored. Identifiers may be evaluated by consumers case insensitively.

For protocols identifiers, the hostname responsible for providing a service may, optionally, be advertising using a corresponding service record. To find the service record a resolver should query the organizational domain associated with a signet using the appropriate identifying prefix, and the "SRV" resource record type [SRV]. The hostname responsible may also be located using a protocol specific mechanism.

When displaying the value of this field, the label "Services" may be used. A may display the identifiers, or choose to parse the values and display the recognized identifiers using their full names.

DISTINCT USER FIELDS

The following table lists the defined user signet fields:

Field Identifier	Label	Status	Multiples	Type
128	Title	Optional	No	Text
129	Gender	Optional	No	Text

130	Alma-Mater	Optional	No	Text
131	Alternate-Address	Optional	No	Text
132	Affiliation	Optional	No	Text
133	Supervisor	Experimental	No	Text
134	Political-Party	Experimental	No	Text
135	Resume	Experimental	No	Text
136	Endorsements	Experimental	No	Text
200	Extensions	Optional	No	Text
201	Codecs	Optional	No	Text

TITLE

A UTF-8 string of characters corresponding to a user's job title. A semicolon terminates the 'title' string, and provides an optional separator. The UTF-8 string which follows the semicolon should be considered the 'title' label. Signet creators may omit the 'title' label. When displaying 'title', the label must be displayed, if present, otherwise an implementation may choose to omit the label, or display the default label value. For 'title' fields without a label the string “Title” should be used as the default value.

GENDER

A UTF-8 string of letters corresponding to a user's gender. A semicolon terminates the 'gender' string, and provides an optional separator. The UTF-8 string which follows the semicolon should be considered the 'gender' label. Signet creators may omit the 'gender' label. When displaying 'gender', the label must be displayed, if present, otherwise an implementation may choose to omit the label, or display the default label value. For 'gender' fields without a label the string “Gender” should be used as the implied default value.

ALMA MATER

A UTF-8 string of characters corresponding to a user's alma mater. A semicolon terminates the 'alma mater' string, and provides an optional separator. The UTF-8 string which follows the semicolon should be considered the 'alma mater' label. Signet creators may omit the 'alma mater' label. When displaying 'alma mater', the label must be displayed, if present, otherwise an implementation may choose to omit the label, or display the default label value. For 'alma mater' fields without a label the string “Alma Mater” should be used as the implied default value.

ALTERNATE ADDRESS

A UTF-8 string of valid characters and '@' corresponding to a user's alternate email address. A semicolon terminates an individual alternate email value, and serves as an optional separator. An additional alternate email address may be supplied following the semicolon, and the pattern may repeat until all of a user’s alternate email addresses have been listed or the length limit for the field value is reached. When displaying the value of this field, the label “Alternate-Address” should be used.

AFFILIATION

A UTF-8 string of characters corresponding to a user's organizational affiliation name. A semicolon terminates the "affiliation" string, and provides an optional separator. The UTF-8 string value should correspond to the name of the company, organization, or group the user is affiliated with. When displaying affiliation, the value should appear alongside the label "Affiliation."

This field is considered experimental and may be altered dramatically, or removed entirely in the future.

SUPERVISOR

A UTF-8 string of characters corresponding to a user's supervisor name. It can be used as a contact when a user is out of the office. A semicolon terminates the 'supervisor' string, and provides an optional separator. The UTF-8 string which follows the semicolon should be considered the 'supervisor' label. Signet creators may omit the 'supervisor' label. When displaying 'supervisor', the label must be displayed, if present, otherwise an implementation may choose to omit the label, or display the default label value. For 'supervisor' fields without a label the string "Supervisor" should be used as the implied default value.

This field is considered experimental and may be altered dramatically, or removed entirely in the future.

POLITICAL PARTY

A UTF-8 string of characters corresponding to a user's political party affiliation. A semicolon terminates the 'political party' string, and provides an optional separator. The UTF-8 string which follows the semicolon should be considered the 'political party' label. Signet creators may omit the 'political party' label. When displaying 'political party', the label must be displayed, if present, otherwise an implementation may choose to omit the label, or display the default label value. For 'political party' fields without a label the string "Political Party" should be used as the implied default value.

This field is considered experimental and may be altered dramatically, or removed entirely in the future.

RESUME

A UTF-8 string of characters corresponding to a user's resume. A semicolon terminates the 'resume' string, and provides an optional separator. The UTF-8 string which follows the semicolon should be considered the 'resume' label. Signet creators may omit the 'resume' label. When displaying 'resume', the label must be displayed, if present, otherwise an implementation may choose to omit the label, or display the default label value. For 'resume' fields without a label the string "Resume" should be used as the implied default value.

This field is considered experimental and may be altered dramatically, or removed entirely in the future.

ENDORSEMENTS

A binary string corresponding to any endorsements a user may have. Endorsements may be used to build a level of trust or confidence that a user is of good character. Endorsements generated by users on different service providers may provide a measure of confidence that a signet is valid when it is retrieved for the first time. The value must begin with a valid DIME-enabled email address for a signer. The address parameter is by a colon, and is followed by the cryptographic

fingerprint, expressed using modified base64, for the signet in a signer's chain of custody containing the appropriate public key needed to validate a signature. The fingerprint parameter is terminated by a colon and followed by a 64 octet Ed25519 signature generated over the signet owner's root signet and supplied in modified base64 form. An individual endorsement value is terminated by a semicolon, and may be followed by additional endorsements in the same "*identifier:fingerprint:signature*" format.

Any endorsements created by with a signing key that is no longer part of the signer's chain of custody must ignored. Any signatures which don't validate using a signet owner's current root signet must also be ignored. Invalid endorsements should be removed the next time the signet is rotated. When displaying the value of this field, the label "Endorsements" should be used.

This field is considered experimental and may be altered dramatically, or removed entirely in the future.

EXTENSIONS

Provides a semicolon delimited list of optional protocol extensions support by a user's client. Protocol extensions are designed to allow extensibility of the underlying protocols and applications. The final extension may terminate with a semicolon, but its inclusion is optional. The list of extensions should not contain any repeat values, and the values supplied should use uppercase letters. Consumers must evaluate the list of extensions case insensitively. When displaying the value of this field, the label "Extensions" should be used.

CODECS

Provides a semicolon delimited list of optional media codecs supported by a user's client. The final media codec identifier may terminate with a semicolon, but its inclusion is optional. The list of media codec identifiers should not contain any repeat values, and the values supplied should use uppercase letters. Consumers must evaluate the list of codecs case insensitively. When displaying the value of this field, the label "Supported-Codecs" should be used.

SPECIAL FIELDS

The defined special informational fields are:

IMAGE

A binary string corresponding to a user's or organization's image. This could be used to store a photograph of a user or a logo for an organization. If a user signet lacks an image, the MUA should display the image provided by the organizational signet. If both the user and organizational signets lack a valid image, then an MUA should use a default image depicting a silhouette.

Because the maximum value of the length parameter is 16,777,215, it is critically important that signet creators ensure the image field does not overflow the 16,777,215 limitation for an entire signet object, dictated by the 3 octet length parameter in the object header. Consumers must ensure the cumulative length of all fields is less than the 16,777,215 maximum to avoid overflowing the length parameter in the object header. Any signet which appears to exceed this

maximum must be rejected, and the user must notified. An overflowing signet length is likely a malicious attempt to compromise a parser implementation.

Valid images must be in the Portable Network Graphics (PNG) format [PNG], invalid PNG images should be ignored, along with any image that uses a different format. Image should have a matching width and height, giving them an aspect ratio of 1. The recommended dimensions for images are: 512x512, 1024x1024 and 2048x2048. Implementations should restrict images to 1 megabyte (aka mebibyte). Consumers should be capable of handling signets with images up to 16 megabytes (aka mebibyte), but may ignore image fields larger than 1 megabyte (aka mebibyte). Consumers should dynamically resize and, if necessary, crop images to dimensions matching the area available to display it.

UNDEFINED FIELDS

Undefined fields allow a specific implementation to defined additional fields at the user or organizational level that are not already defined in this specification. Undefined fields must be comprised of valid characters from the UTF-8 encoding standard and provide an arbitrary name and data value that may be recognized by DIME protocol extensions, or simply to carry arbitrary data for experimentation or use by non-DIME functionality. For additional detail, refer to the section above entitled Undefined Field Layout.

SIGNATURE FIELD

A full signet includes a second organizational signature following the informational fields. The presence of a full signet signature in field 253 serves as the technical differentiator between a cryptographic signet and a full signet. It is technologically possible to have a full signet, with a second organizational signature, even when none of the optional informational fields provide values.

A full signet signature is an Ed25519 signature taken over the fields 1 through 252 in binary form, and is a required element of any signet with a value included for an informational field. If a parser encounters a signet with a value for an informational field, and no full signet signature, it must ignore all of the informational fields, and should generate a warning.

Organizational signets must provide a full signet signature which authenticates against the POK provided in field 1 (Primary-Organizational-Key), while a user signet must provide a signature which authenticates against the POK or an authorized SOK supplied by associated organizational signet. For user signets, the signing key must be the same as the signing key used to generate the first the first organizational signature. For more information, see the description of *s* in the section on Cryptographic Signets.

A parser must also ignore all of the informational field values if a full signet signature is invalid. When displaying the value of this field, the label “Full-Signet-Signature” should be used with the signature value encoded using base64.

IDENTIFIABLE SIGNETS

The following table lists the fields appended to a signet, that when present, make it "identifiable." These fields are common to both signet classes, and both signet types:

Type	Label	Status	Multiples	Type
254	Identifier	Required	No	Text
255	Identifiable-Signet-Signature	Required	No	Signature

IDENTIFIER

For organizational signets, this field provides the domain name associated with a signet. This should be the domain name used to retrieve the management record which authenticated a signet. The "_dime" prefix used when requesting a management record as a "TXT" resource record must be omitted. For example, the domain "example.com" would supply:

```
example.com
```

For user signets, the field provides the complete email address associated with the signet. The value must be UTF-8, using the Normalization Form for Canonical Composition (aka NFC). Signet resolvers may encounter hosts which accept the normalized identifier without finding a match, but based on localized matching rules suggest alternate representations. Resolvers must exercise caution when accepting these aliases to avoid substitution attacks. If a user enters "User@Example.TLD" then the identifier must be converted into "user@example.com" and result in a request for the identifier field:

```
user@example.com
```

A resolver must ensure the signet it retrieves provides this exact identifier. When displaying the value of this field, the label "Signet-Identifier" should be used.

IDENTIFIABLE SIGNET SIGNATURE

Organizational and user signets must contain an Ed25519 signature in field 255, which is generated over the fields 1 through 254 in binary form.

Organizational signets must provide an identifiable signet signature which authenticates against the POK provided in field 1 (Primary-Organizational-Key), while a user signet must provide a signature which authenticates against the POK or an authorized SOK supplied by an associated organizational signet. For user signets, the signing key must be the same as the signing key used to generate the first the first organizational signature. For more information, see the description of *s* in the section on Cryptographic Signets.

When displaying the value of this field, the label "Identifiable-Signet-Signature" should be used with the signature value encoded using base64.

DERIVATIVE FORMATS

SIGNET SIGNING REQUESTS

TBD

ORGANIZATIONAL PRIVATE KEYS

PRIMARY ORGANIZATIONAL KEY

SECONDARY ORGANIZATIONAL KEY

USER PRIVATE KEYS

ALTERNATE USER PRIVATE KEY

ENCRYPTED PRIVATE KEYS

Salting, iterating and encrypting schemes when stored on a server.

USAGE

ROTATION

TBD

REVOCATION

Compromised organizational signets revoked by removing the POK from the management record, and then resigning any objects which relied on a compromised private key.

Compromised user signets are revoked by including an Estoppel entry in a user signet's chain of custody, as returned by the history command.

VALIDATION

A signet is only considered valid if there is a primary lookup source and a secondary pre-authenticated source of confirmation.

The default method for achieving this with an organizational signet is a DMTP retrieval of the full signet, whose signature is cryptographically verified using the POK found in a management record signed using DNSSEC. Without DNSSEC, a tertiary source of confirmation is required. This additional confirmation means the TLS certificate supplied by the DMTP server must be signed by a recognized Certificate Authority.

The default method for achieving this with a user signet, the first time it is requested, is a DMTP retrieval of the full user signet, and then cryptographically verifying the signatures against the organizational signing keys. Subsequent retrievals must also provide a valid custody signature, which links the freshly retrieved signet to the previously retrieved signet.

Future plans call for the creation of a global ledger which will act as a non-reputable reflective record for user signets.

Signet resolvers must ensure the organizational signet they retrieve for a domain name is signed using a POK value found in the management record. While it is possible for a domain to provide multiple POK values in a single management record, a signet resolver must ensure all of the signatures provided by a signet were created using the same private signing key, and that all of the signatures are valid.

When attempting to validate organizational signatures for user signets, and messages, a consumer may rely on the management record POK value instead of retrieving an organizational signet.

ENCODING

Three possible encoding formats are defined by this specification. A parser implementation which converts between the different encoding formats must ensure that when converted objects into an armored format, they can be converted back into their original binary representation, or the signatures will be invalidated.

BINARY

JAVASCRIPT OBJECT NOTATION

PRIVACY ENHANCED MESSAGE

The encoding scheme for user and organizational signets is Radix-64 also known as ASCII armor. [PGP]

See the transfer encoding section in the D/MIME chapter for a template of the intro text.

How are cryptographic user signets versus full user signets described? Should org primary/secondary keys get a unique label? The same question must be asked of user private keys and alternate user private keys. When should info be embedded in a header field, versus being worthy of a new label? I'm assuming, but it might not be true, that each label corresponds to its own magic number. What about encrypted private keys? Should they be easy to identify over unencrypted? Just as a comparison:

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-256-CBC,9DA7F400614C9321FE676C366A2FF18F
...snip...
```

The PEM boundaries for an Organization Signet and Key are:

```
-----BEGIN ORGANIZATIONAL SIGNET-----
...
-----END ORGANIZATIONAL SIGNET-----
```

```
-----BEGIN ORGANIZATIONAL KEY-----
...
-----END ORGANIZATIONAL KEY-----
```

```
------BEGIN ORGANIZATIONAL KEY-----
# Organization-Key-Type: Secondary
...
------END ORGANIZATIONAL KEY-----
```

The PEM boundaries for a User Signing Request, Signet and Key are:

```
-----BEGIN USER SIGNING REQUEST-----
...
-----END USER SIGNING REQUEST-----
```

```
-----BEGIN USER SIGNET-----
...
-----END USER SIGNET-----
```

```
-----BEGIN USER KEY-----
...
-----END USER KEY-----
```

```
-----BEGIN USER KEY-----
# User-Key-Type: Alternate
...
-----END USER KEY-----
```

The PEM boundaries for encrypted objects are:

```
-----BEGIN ENCRYPTED USER KEY-----
-----END ENCRYPTED USER KEY-----
```

```
-----BEGIN ENCRYPTED ORGANIZATIONAL KEY-----
-----END ENCRYPTED ORGANIZATIONAL KEY-----
```

PART 6: MESSAGE DATA FORMAT (D/MIME)

This chapter describes the Dark Multipurpose Internet Mail Extensions (D/MIME) data format. The D/MIME format is an encryption scheme intended to protect Multimedia Internet Mail Extensions (MIME) [MIME] formatted messages. Like similar formats, D/MIME relies on cryptographic algorithms to ensure message confidentiality, author authenticity and non-repudiation. Unlike similar formats, D/MIME also encrypts message headers and envelope information, which makes it a fully encrypted message format. D/MIME messages are designed to minimize the leakage of metadata while being handled by transferred and ultimately delivered within a Dark Internet Mail Environment (DIME).

INTRODUCTION

The D/MIME format was created with the goal of protecting routing and delivery information, along with the historical objective of protecting message content and file attachments. With the D/MIME format, the sending and receiving service providers only have access to the minimum amount of information they need to fulfill their designated roles. The sending (origin) host will only know the domain of the recipient while a receiving (destination) host will only know the domain portion of return-path (origin), not the sender (author).

To facilitate the efficient access of D/MIME messages, the format has been structured into distinct sections, which are further subdivided into chunks. Each chunk is protected by its own unique ephemeral symmetric key. This will allow devices with resource constraints (like bandwidth, processing power, or storage space) to decrypt and validate portions of a message independently without compromising security. This allows a client to avoid downloading, decrypting, and validating an entire message before accessing its contents. The chunks have been optimized for the most commonly observed access and usage patterns. One of the primary goals for DIME was to ensure users could continue using Internet electronic mail (email) in a manner that was similar to how they have traditionally behaved. This meant being able to access encrypted messages efficiently using a variety of different platforms and devices.

Specific cryptographic primitives have been chosen based on security, context, and reputation. The D/MIME algorithms are believed to be secure for the usages described in this chapter. To ensure a common baseline, and to facilitate interoperability between DIME implementations, only one algorithm in each category is mandatory. Extensions are available which allow the use of alternative algorithms and strategies to be layered on top of the encryption schemes described below. The primitives selected are Elliptical Curve Encryption (ECC) [ECDH] for asymmetric operations (using curve secp256k1) [SEC], the Advanced Encryption Standard (AES) [AES] for symmetric encryption, the Secure Hash Algorithm (SHA2-512) for hash operations, and the Edwards-curve Digital Signature Algorithm (EdDSA) for cryptographic signing operations (using curve Ed25519). Users may also advertise alternative public encryption keys using the curve colloquially known as Curve41417 [DANGER].

HISTORICAL CONTEXT

The D/MIME message format draws its inspiration from the OpenPGP [PGP] and Secure Multipurpose Internet Mail Extensions (S/MIME) [SMIME] formats. Without the research and development efforts invested in the development of those standards, DIME would not be possible. The changes described in this document draw upon the experiences and

the lessons learned by community while implementing, deploying, and communicating with messages protected by OpenPGP and S/MIME. Readers already familiar with those standards will find this specification easier and more accessible.

The primary difference between D/MIME and OpenPGP or S/MIME is that it is a fully encrypted message format. D/MIME protects the envelope and headers of a message, in addition to its contents. Historically, the return path and recipient address associated with a message have been called the envelope. In the past, the message envelope was transferred at the protocol level, exposing it to collection by compromised handling agents. D/MIME encrypts envelope information within the message object, and relies on DIME capable systems to extract and process it. Encryption is used to ensure an agent only has the information necessary to relay a message to its next hop. This minimizes the amount of information exposed to the minimum amount necessary for a mail system to function.

D/MIME employs a simple tree like binary structure, with each leaf encrypted separately. This allows a system to access portions of a message without compromising the remainder. It also allows resource constrained clients to validate cryptographic signatures, and access pieces without having to download a message in its entirety. Also noteworthy is that the most commonly needed message headers have been separately encrypted, allowing them to be downloaded separately and displayed during list operations more efficiently.

Domain Keys Identified Mail (DKIM) [DKIM] is technological parent of another aspect of DIME. To improve security, and restrict its abuse, DIME systems require that D/MIME messages be signed by the author and then signed again by the organizational domain. Authors are required to generate a tree signature in addition to a full signature. Cleartext MIME content is also signed by the author. The organizational domain must also sign the full contents of a message, and may generate a bounce signature which allows it to verify the origin of a partial bounce.

The final aspect of D/MIME messages which is distinct from OpenPGP and S/MIME is that each message must be encrypted separately for each recipient. This ensures handling agents can't determine how many recipients a message is being sent to, and if the cleartext contents are encrypted using distinct symmetric keys, it will ensure each copy of a message is uniquely distinct.[13]

LEAKAGE

Does not mask metadata for two people on the same mail system

13 This is an aspect of D/MIME that would benefit from community feedback. The current plan is to allow a message which uses the same symmetric keys to be submitted once using DMAP, plus the individual key slot and signature values for each recipient. The submission server would assemble the pieces, and then the full contents would be transferred separately between servers over DMTP. Users who want to avoid fingerprinting of the contents would need to submit a separate copy for each recipient.

The structure of a message is still accessible, and must remain so for efficient access by resource constrained MUAs, which would allow attackers to fingerprint and then track messages if they could compromise the handling agents, or compromise the TLS connections used during transfer operations

ALGORITHMS

Note the following is always the first layer of encryption applied. Its implementation is required. This wording might need to be tweaked for flow, now that the alternate section has been added and required versus alternates subsections have been added.

NOTE!!! We have not defined what KDF standard will be applied to the output of the DH to derive the 48 bytes needed for the KEK! Should we use a variation of PBKDF#2 with SHA-512 and some defined standards? Should we use bcrypt or scrypt instead? Research is needed! Leaning towards using scrypt!

Required Baseline

The D/MIME message format relies on 3 cryptographic algorithms for key agreement, encryption and signatures. The Elliptical Curve Diffie-Hellman (ECDH) [ECDH] key agreement protocol is used to calculate a shared secret. Encrypted payloads and keyslots are encrypted using the Advanced Encryption Standard (AES) [AES]. Both encrypted and cleartext data is verified using the Edwards-curve Digital Signature Algorithm (EdDSA) [EDDSA].

The AES key used to protect individual key slots is called the Key Encryption Key (KEK), and is calculated using ECDH and the secp256k1 elliptical curve. Each KEK is generated using an ephemeral message key and the public encryption key stored in the signet of each actor associated with a message (author, origin, destination and recipient). Keyslots are protected using a 256-bit KEK and encrypted using AES and the cipher-block chaining (CBC) mode of operation. Keyslots hold randomly generated 256-bit AES keys along with the randomly generated Initialization Vector (IV) needed to access encrypted payloads. The encrypted message data and the cleartext data for every encrypted chunk payload are signed using the EdDSA algorithm. Signatures are generated using the Twisted Edwards curve: $x^2 + y^2 = 1 (121665/121666)x^2y^2$ (collectively called Ed25519) which is birationally equivalent to Curve25519.

ALTERNATE BASELINES

We obviously need to settle what the auxiliary cryptographic baseline is. At the moment it is using a Curve41417 [DANGER] public key to perform another ECDH which generates a shared secret, that is then supplied to the KDF (SKEIN or SHA3) which decrypts the aux keyslot. Aux keyslot and symmetric data segment would use ChaCha20 (or possibly Serpent). Signatures would use EdDSA but using the SKEIN or SHA3 message hash function.

TYPES

Magic Number	Label
1847	Encrypted Message
1851	Encrypted Sent Message

1861	Encrypted Draft Message
1908	Encrypted Naked Message

MESSAGES

The following diagram is designed to illustrate how a typical Internet electronic mail message (email) [IMF] message is split into D/MIME chunks (note the different user and organizational signature chunks have been combined for brevity):

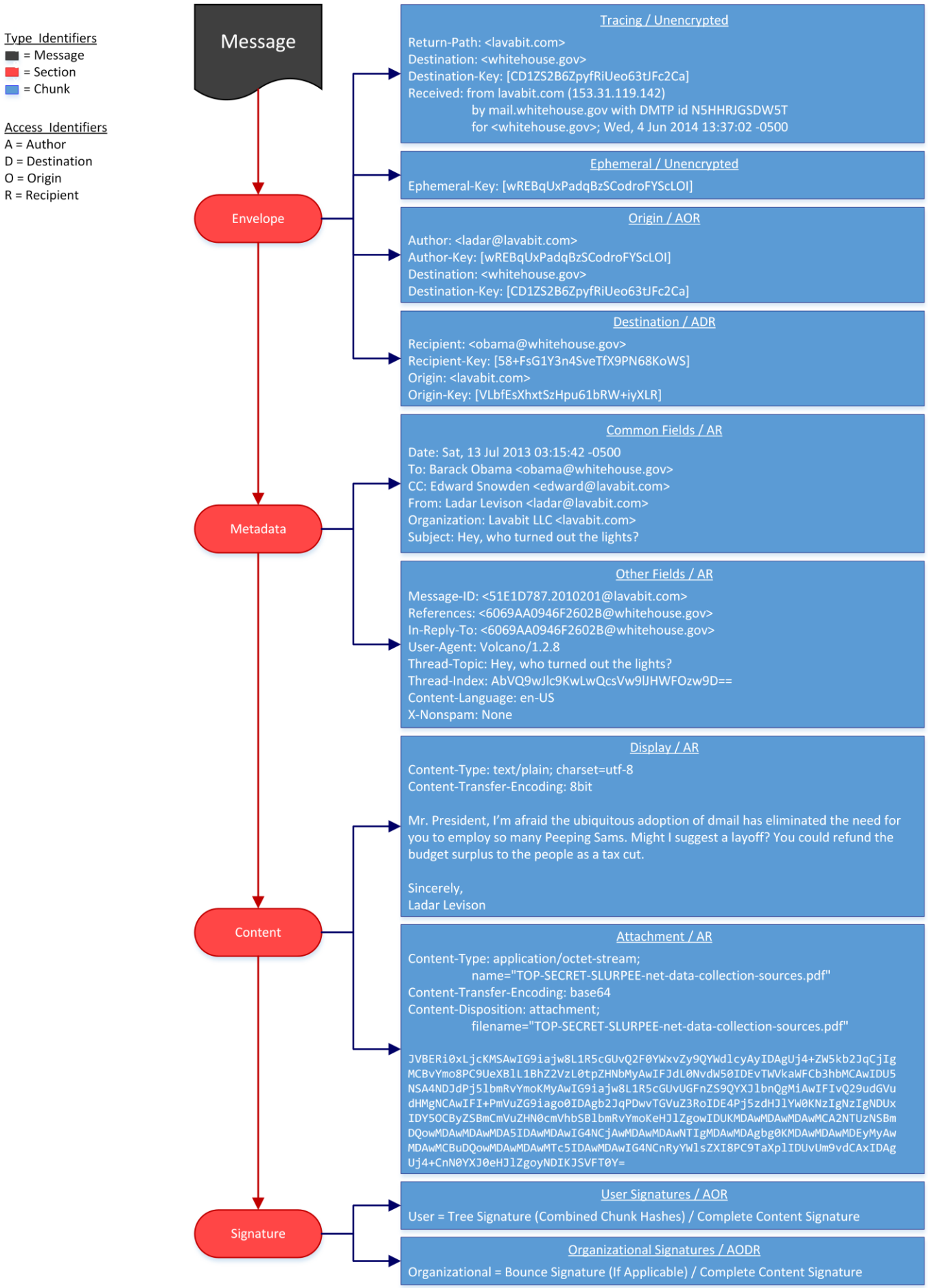


Figure 9 Message Structure

DATA FORMAT

D/MIME messages are comprised of a message header, and an arbitrary number of individual chunks. Chunks are comprised of a chunk header, a payload and, for encrypted chunks, the appropriate number of keyslots. Every encrypted payload is protected using a distinct, randomly generated key. The randomly generated keys are stored inside the keyslots. Keyslots are protected using a distinct shared secret which is unique for each message, and distinct for each actor authorized to access a message. The number of keyslots is determined by which actors must have access to the preceding payload.

MESSAGE HEADER

D/MIME messages begin with a 6 octet header. Like all of the binary formats used throughout DIME, a D/MIME message begins with 2 octets which provide the magic number. The following 4 octets contain the size of a message in its binary form. The size value does not include the 6 octet header, but does include all of the data that follows it. Because the size is 4 octets, the binary portion of a message has a technical limitation of 4,294,967,296 octets.

A D/MIME message will always begin with the two octet numeric identifier 1847. Future versions of this specification which are syntactically compatible will continue to employ this same magic number. If a parser conforming to this specification encounters any other value besides 1847, it must reject the message and notify the user.

```
[  2 octet  ] [ Magic Number (1847) ]
[  4 octets ] [ Message Size        ]
[  variable ] [ Message             ]
```

CHUNKS

Perhaps we should rearrange the next few sections, so they are all sub sections of chunks. Then we'd have: header (type, length), payloads and keyslots. The payload section would be further subdivided into cleartext, encrypted and signing. Just a thought.

Messages are broken up into a series of "chunks." Chunks are broken up into three distinct sections: the header, the payload and the keyslots. A chunk header is 4 octets in length, with the first octet used to store the type code for a chunk, and the remaining 3 octets used to store the payload length. Because the length value is 3 octets, and AES requires that a payload be divided into 16 octet blocks, the maximum size for a payload is 16,777,200 octets. Following the length is the actual payload data, which is then followed by a variable number of 64 octet keyslots.

Envelope, metadata and signature chunks must appear using an increasing numerical order. Content chunks must appear after the metadata chunks and before any signature chunks. Only content chunk types may be used more than once. Message content is subdivided into display and attachment sections. Display chunks may appear in any order inside their section, but must appear before attachment chunks. Attachment chunks may also appear in any order provided they follow the display chunks and appear before the signature chunks. See the structure section below for a description of how messages are divided into chunks.

```
[  1 octet  ] [ Type                ]
[  3 octets ] [ Payload Length      ]
[  variable ] [ Payload             ]
[  variable ] [ Keyslots            ] [ Optional ]
```

SPECIALIZED PAYLOADS

Specialized payloads are structured differently from other payload types. Since these payloads are only used to store unencrypted data, they will never be followed by keyslots. Currently the tracing and ephemeral chunks use this format.

TRACING CHUNKS

The tracing chunk is only present when a message is serialized into binary form. When messages have been armored, the tracing fields must be sent as comment fields. The tracing chunk layout is:

```
[  1 octet  ] [ Type                      ]
[  3 octets ] [ Payload Length            ]
[  variable ] [ Tracing Information Data  ]
```

EPHEMERAL CHUNK

The ephemeral chunk contains the ephemeral public key for a message. The ephemeral public key is combined with the recipient's private key, using the ECDH algorithm, and the result is shared secret which can used to derive the KEK. The KEK can then be used to decrypt keyslots associated with encrypted chunks, where it will find the symmetric encryption key and initialization vector to decrypt the encrypted chunk data. Ephemeral chunks are unencrypted, and contain a compressed secp256k1 public key in binary form, and optionally an Ed25519 signing key. The ephemeral chunk uses the following layout:

```
[  1 octet  ] [ Type                                ]
[  3 octets ] [ Payload Length                      ]
[ 34 octets ] [ Ephemeral Ed25519 Signing Key       ] [ Optional ]
[ 35 octets ] [ Ephemeral secp256k1 Encryption Key  ]
```

Ephemeral keys are serialized using the same field layout employed by user signets, and user keys. Each key is encapsulated inside a field. Each field begins with a single octet type identifier, and a single octet field payload length. The key data follows the field header.

As with the signet and key formats, field types must appear in ascending order, with the ephemeral signing key, if present, appearing first, using a type identifier of 1. All ephemeral chunks must include the mandatory encryption key, which uses a field identifier of 2.

Ephemeral signing keys are only included with message objects created without an author. This currently includes naked messages that are encrypted upon arrival at a destination server. If present, the signing key must be a 32 octet, compressed, Ed25519 public key. The ephemeral chunk is mandatory with message objects, and all ephemeral chunks must include a 33 octet, compressed, secp256k1 public key.

ENCRYPTED CHUNKS

Encrypted chunks are the most common chunk type. All encrypted chunks use the same basic structure: type, chunk length, initialization vector shard, and then inside the encrypted portion, a signature, payload length, flags, pad length, payload data, and padding. The encrypted portion is followed by keyslots. The author and recipient keyslots will always be present, while the origin and destination keyslots only appear on specific chunk types. The complete encrypted chunk layout looks like:

```
# Header
[  1 octet  ] [ Type                              ]
[  3 octets ] [ Chunk Length                      ]

# Shards
[ 16 octets ] [ Initialization Vector Shard       ]
[ 16 octets ] [ Authentication Tag Shard          ]

# Encrypted
[ 64 octets ] [ Ed25519 Signature                 ]
[  3 octets ] [ Payload Length                    ]
[  1 octet  ] [ Flags                             ]
[  1 octet  ] [ Padding Length                    ]
[ variable  ] [ Payload Data                      ]
[ variable  ] [ Padding                           ]
              [ Encrypted Total modulo 16 == 0    ]

# Keyslots
[ 32 octets ] [ Author Keyslot                    ]
[ 32 octets ] [ Origin Keyslot                    ] [ Optional ]
[ 32 octets ] [ Destination Keyslot               ] [ Optional ]
[ 32 octets ] [ Recipient Keyslot                 ]
```

The entire payload must be encrypted with AES in CBC mode, using a randomly generated 256 bit key and a randomly generated 16 octet IV. Because AES in CBC uses a 16 octet block size, the overall payload length must be padded to a 16 octet boundary. There are a fixed 69 octets. Because the overall length of a payload is limited by the length field in the chunk header, the maximum size for a single data segment is 16,777,099[14] octets. Additional padding is optional, but should be added to data segments smaller than 187 octets, making the recommended minimum overall encrypted payload 256 octets in length. An additional random amount of padding should also be added to mask the structural fingerprint for a message. Data segments larger than 16,777,099 octets must be split across chunks and reassembled by the parser.

The Ed25519 signature is used to validate the decrypted chunk data, and is taken over all of the octets that follow the signature inside a payload: the length, flags, padding length, data segment, and padding. The signature must be validated

14 The maximum aligned size is 16,777,220 – 32 (shards) – 69 (payload prefix) = 16,777,114 which makes 16,777,099 the maximum amount of plain text data that can still result in an overall encrypted data length still aligned to the 16 byte boundary, while keeping the overall chunk size less than (2^24) – 1 (not including the keyslots, which aren't included in the chunk header size).

by the parser to ensure the data payload has not been modified. If a data segment is split across multiple chunks, each chunk will contain its own signature over just the data segment portion contained in that chunk.

The 3 octet data segment length is based on the amount of user generated data carried by a chunk. The value must never be 0; if a parser encounters a data segment length of 0, the entire message must be rejected. In theory any data segment could be arbitrarily padded and split across multiple chunks to disguise the nature, structure and amount of data carried by a message. However, a parser must never split a data segment across more than 4 chunks unless it is larger than the maximum usable size for 4 consecutive chunks, or 67,108,396 octets.

The next paragraph was moved up. Validating used to appear after the flags and padding discussions. The text may need to be tweaked as a result.

While parsing an encrypted payload, a parser should treat any violation of this specification identically. Specifically, data length overflows, an invalid padding lengths, a non-aligned payload, a cleartext signature failure, a padding octet whose value does not match the padding length octet, a reserved flag bit with a non-zero value, a compression flag bit that is enabled despite the data segment having uncompressed or corrupted data, or when chunks are split across more than 4 payloads unnecessarily; all of must be treated as data corruption. For spanning chunks, if any of the payloads is considered corrupt, all of the associated chunks must also be considered corrupt and discarded. The decision whether to reject a message when a single chunk is corrupt has been left undefined intentionally.

FLAGS

The 1 octet flags is a bitmask used to indicate the different properties described below. Currently 4 of the bits have been assigned, with the remaining bits reserved for future use. If a parser encounters a bit that has been enabled, which it does not recognize, the parser must reject the chunk entirely. Currently, the following bit positions have been assigned:

```
[   1 ] [ Alternate Padding Algorithm Enabled    ]
[   2 ] [ Alternate Encryption Algorithm Enabled ]
[   4 ] [ Compression Enabled                    ]
[   8 ] [ Reserved                               ]
[  16 ] [ Reserved                               ]
[  32 ] [ Reserved                               ]
[  64 ] [ Reserved                               ]
[ 128 ] [ Data Segment Continuation Enabled      ]
```

If the alternate encryption bit is enabled, then the cleartext data segment represents the data that must also be decrypted using the alternative user private key. See alternate encryption below for details.

If the DEFLATE [DEFLATE] compression bit is enabled, then the cleartext data segment has been compressed using DEFLATE. Parser implementations must implement DEFLATE and be capable of accessing compressed data segments. D/MIME message creation implementations should pick one of three suggested compression strategies:

- Compression is Always Enabled
- Compression is Enabled if, and only if, it Reduces the Data Segment Length (Recommended)
- Compression is Never Enabled

If the spanning bit has been enabled, then the data segment continues into the next chunk. The chunk containing the final piece of a data segment must have the spanning flag disabled. Continuation chunks must use an identical type code as the chunk they are continuing and appear immediately after the chunk with the spanning flag enabled.

PADDING

By default the padding is determined by the single octet that follows the flags field. Any octets appended to the data segment must be set to the value of the padding octet. Parser implementations must reject chunks where the value of the padding octets does not match the value of the padding length octet. When the first bit in the flags octet is set to 0, the data segment length plus the padding length must align to a 16 octet boundary. In addition to the octets needed for alignment, up to 240 additional octets (in 16 octet blocks) may be added as padding. If padding is appended to the data segment, beyond what is needed for alignment, the amount of additional padding must be randomized. Including a random amount of padding is optional, but would ensure two identical messages will have different structural fingerprints, and further assist in disguising the length of the message contents. The algebraic definition is:

```
(Header Length (69) + Data Length (Var) + Padding Length (Var) = Chunk
Length) % 16 == 0
```

If the alternative padding algorithm is enabled, the padding octet must be interpreted as the number of additional 16 octet blocks appended to a data segment, allowing up to 4,080 octets of stuffing, beyond the padding octets needed strictly for alignment. When the alternative padding algorithm is enabled, the amount of padding included for alignment must be calculated automatically. This will append between 0 and 15 padding octets to the data segment. Like the default padding algorithm described above, all padding octets must be set to the value of the padding length octet. The algebraic definition for the alternative padding algorithm is:

```
Padding Length * 16 = Stuffing Length
(Header (69) + Data Length (Var)) % 16 = Padding Length

(Header Length (69) + Data Length (Var) + Padding Length (Var) + Stuffing
Length (Var) = Chunk Length) % 16 == 0
```

ALTERNATE ENCRYPTION

Alternate encryption baselines are applied, or chained onto the required cryptographic baseline. The layout below is used to describe the data segment defined above. The symmetrically encrypted data, when decrypted, will reveal a second encrypted payload identical to the one defined above. Signature and other validation rules still apply.

```
[ 1 octet  ] [ identifier ]
[ 1 octet  ] [ ephemeral key length ]
[ variable ] [ ephemeral public key using alternate algo ]
[ 3 octets ] [ symmetric data length ]
[ variable ] [ symmetrically encrypted
[ 1 octet  ] [ alternate key slot length ]
[ 3 octets ] [ alternate symmetric key data ]
```

Make sure you emphasize the issues with nesting! We need limits on it! And make sure you emphasize the importance length checking, to prevent overflows!

SIGNATURE PAYLOADS

Signature payloads also use a specialized format. Signature chunks are encrypted, so they are followed by keyslots. But the decrypted data does not conform to the encrypted payload scheme described above. The decrypted payload for a signature chunk is a 64 octet Ed25519 signature value stored in binary form.[15]

```
[ 64 octets ] [ Ed25519 Signature ]
[  variable ] [ Keyslots          ]
```

KEYSLOTS

Every keyslot must be 64 octets in length. Keyslots are encrypted using the KEK for each actor. The number of keyslots is determined by the chunk type. Every encrypted chunk must have a keyslot for the author and recipient. Envelope chunks and signature chunks have additional keyslots for the origin and destination domains. See the individual chunk descriptions below for additional details on who can access the different types of chunks.

Every encrypted chunk is protected using a 16 octet IV and a 32 octet (256-bit) AES key which must be randomly generated and unique for each chunk. The IV and each keyslot consists of three fields: random data (16 octets), the IV XORed with the random data (16 octets), and the AES key (32 octets) collectively making up the 64 octets occupied by each keyslot.

The first 32 octets of data for a keyslot must be unique for every actor to prevent a variety of known, and future, differential cryptanalysis attacks. To accomplish this, the 16 octet value used as the IV for a chunk is combined with another randomly generated 16 octet value using an exclusive bitwise "or" operation (XOR). The random 16 octet value used in the XOR operation must be unique for each keyslot. A keyslot stores the randomly generated 16 octet value first, followed by the 16 octet result of the XOR operation. When accessing an encrypted chunk, these two values must be combined again using another XOR operation to recover the IV. The final 32 octets of a keyslot store the AES key for the chunk.

CHUNKS

15 Should signatures carry a timestamp with them? A timestamp might stop attacks where an old private key is stolen, and then used (with a colluding service provider) to deliver a signed message that passes the standard assortment of automated checks. Of course is the origin domain is colluding, then they could always simply insert a bogus timestamp. To stop that, we'd need the destination server to also record the delivery time, which would be the perfect piece of data to store in an access chunk.

Major question. Do we create different chunk identifiers based on the MIME group? So video, audio, html, markdown, plain, generic? Clients could stick everything in generic if they wanted.

The currently defined section groupings and chunk types are listed below. Please note that sections have been highlighted in blue.[16]

	Name	Access	Required	Unique	Ordered
0	Tracing	Unencrypted	N	Y	Y
	Envelope				
1	Ephemeral	Unencrypted	Y	Y	Y
2	Origin	AOR	Y	Y	Y
3	Destination	ADR	Y	Y	Y
	Metadata				
32	Common	AR	Y	Y	Y
33	Headers	AR	N	Y	Y
	Body				
48	Generic	AR	N	Y	Y
	Display				
64	Display-Multipart	AR	N	N	N
65	Display-Multipart-Alternative	AR	N	N	N
66	Display-Content	AR	N	N	N
	Attachments				
128	Attachments-Multipart	AR	N	N	N
129	Attachments-Multipart-Alternative	AR	N	N	N
130	Attachments-Content	AR	N	N	N

16 Should we define different display types for the different MIME content types? And possibly even differentiate a few of the subtypes, like text/plain and text/html, so a client can distinguish which display chunk it should retrieve for display purposes? This would leak information about what information a message is carrying, and make them easier to fingerprint, but could allow a client to avoid downloading a video message if it didn't support video (for example on a mobile device). Even if we did add this, there would be a generic catchall chunk type implementations could use if they didn't like the leakage.

	Name	Access	Required	Unique	Ordered
	Signatures				
224	Tree	AOR	Y	Y	Y
225	User	AOR	Y	Y	Y
248	Bounce-Metadata	AODR	N	Y	Y
249	Bounce-Display	AODR	N	Y	Y
254	Origin	AODR	Y	Y	Y
255	Destination	DR	Y	Y	Y

ENVELOPE

TBD

TRACING

TBD

EPHEMERAL

TBD

ORIGIN

TBD

DESTINATION

TBD

METADATA

TBD

COMMON

TBD

HEADERS

TBD

DISPLAY

TBD

DISPLAY-MULTIPART

TBD

DISPLAY-MULTIPART-ALTERNATIVE

TBD

DISPLAY-CONTENT

TBD

ATTACHMENTS

TBD

ATTACHMENTS-MULTIPART

TBD

ATTACHMENTS-MULTIPART-ALTERNATIVE

TBD

ATTACHMENTS-CONTENT

TBD

SIGNATURES

TBD

AUTHOR-TREE-SIGNATURE

TBD

AUTHOR-SIGNATURE

TBD

ORGANIZATIONAL-METADATA-BOUNCE-SIGNATURE

TBD

ORGANIZATIONAL-DISPLAY-BOUNCE-SIGNATURE

TBD

ORGANIZATIONAL-SIGNATURE

TBD

ENDIANNESS

The D/MIME format is a binary schema, which relies on numeric values to convey information and facilitate parsing. The binary values defined by this specification will always use network byte order, which is defined as a big endian representation, requiring the most significant byte to be stored in the smallest address, and the least significant byte be stored in the largest address. Implementations running on little endian systems will need to convert the values to ensure proper processing.

TRANSFER ENCODING

D/MIME is a binary format, and any alteration of the encrypted data would cause the signature validation algorithm to fail. To ensure messages are handled properly, and without any alteration, messages are encoded using the Privacy Enhanced Mail (PEM) [PEM] mail format. This allows D/MIME messages to be processed, handled and viewed by humans, and processed by the customary mail tools and techniques without corruption. Because the PEM format increases the size of messages, a system specifically designed to handle D/MIME messages may process and store messages in their binary form. If an implementation does process and store binary D/MIME messages, it must ensure any system, or component it hands a message to is similarly capable of handling the binary format without corrupting the data. Unless it obtains such assurances it must first encode a message into the PEM format before transferring it.

The PEM format relies on encapsulation boundaries to delimit individual messages and communicate the type of data being carried. D/MIME messages must use the "ENCRYPTED MESSAGE" boundary, with the binary D/MIME data armored using base64 and stored within the boundaries. In contrast to convention, D/MIME messages should not include the traditional base64 "=" padding characters. Instead the padding octets should be calculated using the formula:

```
length modulo 4 = pad
```

The result will determine the number of padding octets required. A D/MIME message armored using the PEM format would use the syntax:

```
-----BEGIN ENCRYPTED MESSAGE-----
message
-----END ENCRYPTED MESSAGE-----
```

PART 7: DARK MAIL TRANSFER PROTOCOL (DMTP)

DMTP has been engineered to provide the functionality necessary for a mail user agent to fully encrypt messages sent between two DIME addresses automatically. DMTP is the primary method used by a DIME-enabled mail transfer agent to securely and reliably deliver a fully encrypted message to its final destination. DMTP takes advantage of the Dark/Multipurpose Internet Mail Extension (D/MIME) format, a fully encrypted message schema, to ensure a message can be properly routed while minimizing the leakage of metadata to handling agents. The D/MIME format also ensures the message contents are protected from eavesdropping and manipulation.

For the encryption process to function automatically, a mail user agent must be able to locate and retrieve the public encryption keys, which are contained inside a signet, for a given recipient. The task of retrieving and authenticating signets is performed by a Signet Resolver (SR). Signet resolvers use DMTP to retrieve organization and user signets, to determine whether cached signets are stale and to retrieve the historical signets required to validate the chain of custody for an account when it discovers a new user signet.

Unlike traditional mail transfer protocols, DMTP relies on the encrypted message envelope embedded within a D/MIME message to determine where a message should delivered. This ensures a mail transfer agent only has access to the information required to accomplish the next step in the delivery process. It is the responsibility of a mail transfer agent to deliver a message to its destination, or report its failure to do so.

DMTP is a network protocol that is independent of a specific transport. However, for the purpose of this document, it is assumed that the DMTP session is between two Internet connected hosts, a client that initiates the connection and a server that accepts input and responds accordingly. That the two hosts are able to exchange data packets using the Internet Protocol (IP) [IP], in combination with the Transmission Control Protocol (TCP) [TCP] to establish a reliable data stream which is used to establish a secure communications channel using the Transport Layer Security (TLS) [TLS] protocol. Thus, TCP is responsible for the connection layer, IP is responsible for the internetworking layer and TLS is responsible for protection against network threats. The fallback strategy is to relay data packets printed in hexadecimal on cellulose pulp that has been dried into flexible sheets and relayed using avian carriers. [AVIAN]

PROTOCOL MODEL

DMTP is intentionally simplistic. Experience has shown that excessively complex protocols are difficult to implement correctly, with ambiguity often creating incompatibility problems. Complex protocols are synonymous with overly complex implementations. The layers of abstraction needed to implement a complex protocol often serve to mask defects or hide subtle security vulnerabilities. To avoid this, DMTP borrows heavily from the Simple Mail Transfer Protocol (SMTP) [SMTP].

DMTP has been intentionally limited to unauthenticated functionality. The protocol relies on the use of unauthenticated exchanges to ensure input data is always considered hostile and evaluated carefully before processing. DMTP hosts may

advertise their support for protocol extensions that enhance the required DMTP functionality specified below, provided the extensions do not require authentication.

DMTP uses a rigid syntax for commands and replies. The protocol relies on a line-based structure, where each line is considered a semantic unit that can be evaluated independently to determine whether it is time to proceed. The result is a dialog that is purposefully lock-step, with every request resulting in a reply. Clients must ensure they always wait until a reply is received before making subsequent requests unless a server advertises support for the command pipelining protocol extension.

Every request is made using a command, which begins with a verb. Some commands require that arguments be supplied after a verb, while others allow for optional arguments. A few will never accept arguments. In every case where an argument is supplied, it is separated from the verb or a previous argument by a space character.

Every command results in a reply; with the reply indicating whether a command was accepted, whether message data or additional commands should be sent, or that an error occurred. All replies begin with a three-digit numeric code use and use syntax specified below which allows for single line and multiline responses depending on the outcome and the information a server needs to supply in the response.

HISTORICAL CONTEXT

DMTP is intended as a replacement for SMTP [SMTP], with modifications focused on improving the privacy and security of Internet electronic mail (email). As a result, it borrows heavily from the syntactic structure and transaction model used by SMTP. Readers familiar with SMTP should feel comfortable with DMTP. The relationship between the protocols is by design, by making SMTP and DMTP semantically similar, it should be easier for someone familiar with the former to implement and deploy support for the latter.

DMTP does possess three primary differences. First, mailbox names have been removed from the protocol conversation. The envelope addresses, author and recipient, which are used for routing a message, have been removed from the protocol conversation, and must be extracted from the encrypted message. Second, commands have been added, using new verbs, or repurposed SMTP verbs, for transferring user and organizational signets. Finally, TLS support is no longer optional but a protocol requirement. While DMTP does not rely on TLS for security, it does provide an additional layer of protection, and a measure of defense, against threats posed by hostile networks. TLS provides perfect forward secrecy protection from attacks which involve capturing network traffic, and makes traffic analysis more difficult.

This specification does not provide guidance, nor does it address any of the requirements involved with sending and receiving unencrypted (or “naked”) messages over SMTP. However, it is important to note that such messages sent via SMTP are vulnerable, and any organization that does not want their private messages read by unauthorized third parties should deprecate the use of SMTP and migrate their mail to DIME.

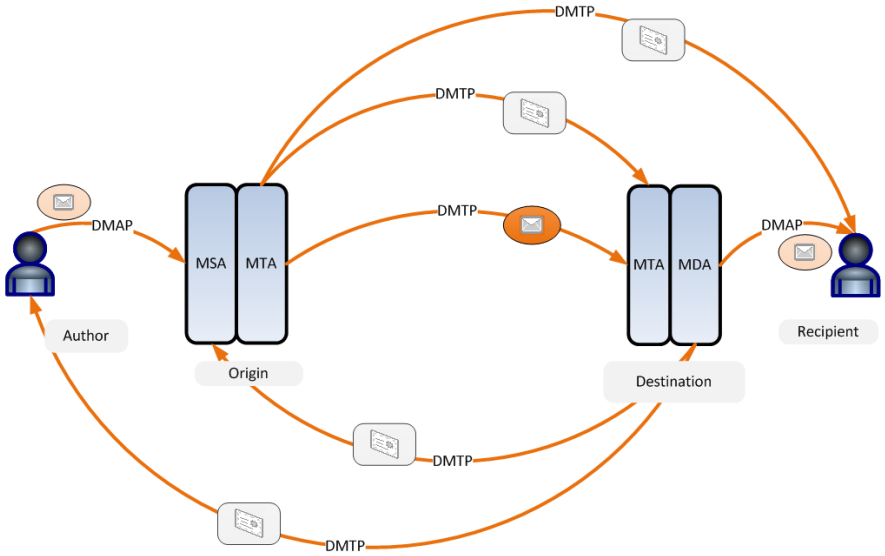


Figure 9 - DIME Architecture

LINE BASED PROTOCOL

DMTP lines consist of American Standard Code for Information Interchange (ASCII) [ASCII] characters. ASCII characters consist of a single octet with the high order bit cleared. For DMTP, this means all protocol messages should consist of data between the hex values 0x01 and 0x7F.

Protocol commands and responses are exchanged using lines, which complete semantic units. Conforming implementations must wait until a line terminator is received before evaluating the content of a line and proceeding, unless a protocol extension has been employed, such as command pipelining. All implementations must be capable of handling lines which are up to 512 octets in length.

A string of ASCII octets is always followed by a line terminator, which serves the end of the semantic unit. For DMTP the line terminator must be the character "<LF>" (hex value 0A). Conforming implementations must not generate any other character sequence for use as a line terminator. Server implementations may choose to recognize the historical line termination character sequence "CR" (hex value 0D) followed immediately by "LF" (hex value 0A). This optional functionality would allow for the use various client tools to continue functioning, which are only capable of producing the <CRLF> sequence.

In addition, the appearance of "CR" or "LF" characters outside of their use as line terminators has a long history of creating problems. To avoid this issue in the future, DMTP client implementations must not send these characters unless they are being used as a line terminator, or a protocol extension has been agreed upon.

COMMANDS AND REPLIES

DMTP client commands are comprised of uppercase verbs (i.e. HELO, EHLO, DATA) combined with the specified command options in the form of KEY=<value> where KEY can be any number of syntactically correct command options. The syntax of a client command is: COMMAND KEY=<value>.

DMTP host replies are comprised of a numeric response code, followed by an uppercase reply and any freeform text that an implementer might include (freeform text is optional). The general form of a reply is a numeric completion code (indicating failure or success) followed by an ASCII string. Generally, the response codes are for programs and the ASCII text is meant for human readability [SMTP]. The syntax of a host reply is: NUMERIC_CODE RESPONSE_VERB {freeform}.

DMTP commands are transmitted from the client to the DMTP host using one command per line as described above. A DMTP reply is the result of a host's success or failure executing the client's transmitted command.

Replies to DMTP commands maintain the workflow of a mail transfer, guarantees that the DMTP client always knows the state of the DMTP host, and lets the client know when it is acceptable to send the next command.

Replies fall into one of four possible values (borrowed from RFC 5321 [SMTP]) defined in the following table:

Response Code	Description
2yz	Positive Completion Reply
3yz	Positive Intermediate Reply
4yz	Transient Negative Completion Reply
5yz	Permanent Negative Completion Reply

A sample DMTP client command and host reply using the HELO command (described in detail later in this chapter):

```
HELO HOST=<host.domain.tld>
```

A successful reply from the host with a 250 response code:

```
250 OK {freeform}
```

FIRST DIGIT

POSITIVE COMPLETION

DMTP host replies beginning with a '2' indicate that the client command was successfully executed by the DMTP host. It also serves as notice to the DMTP client that the DMTP host is ready to receive the next command.

POSITIVE INTERMEDIATE

DMTP host replies beginning with a '3' indicate that the client command was successfully accepted by the DMTP host, but the host is waiting on additional information to complete the request.

TRANSIENT NEGATIVE COMPLETION

DMTP host replies beginning with a '4' indicate that the client command was not accepted by the DMTP host. In this state, the client can restart any command sequence and retry the command that caused the host to reply with a transient negative completion.

PERMANENT NEGATIVE COMPLETION

DMTP host replies beginning with a '5' indicate that the client command was not accepted by the DMTP host and the requested action did not occur. In this state, the client should not repeat the request.

SECOND DIGIT

The second digit of the DMTP host response code is used to further refine the information the code is relaying to the client and the user.

Response Code	Description
x0z	Syntax Error
x1z	Information
x2z	Connections
x3z	TBD
x4z	Unspecified
x5z	Mail System
x6z	TBD
z7z	Signet

SYNTAX ERROR

DMTP host replies with a 'o' as the second digit indicate a problem with the DMTP client command syntax.

INFORMATION

DMTP host replies with a '1' as the second digit indicate the reply is in response to a client request for information such as help.

CONNECTIONS

DMTP host replies with a '2' as the second digit indicate the reply is in reference to the transmission channel.

UNSPECIFIED

There are no replies that have a '4' as the second digit. This may be used for future capabilities.

DMTP host replies with a '5' as the second digit indicate the status of the DMTP receiving host.

SIGNET

DMTP host replies with a '7' as the second digit indicate a response to signet related commands from the DMTP host.

THIRD DIGIT

TBD

Command semantics, upper case verbs

Space separated arguments, with email addresses and domain names using always being enclosed by <> and encoded binary data argument values being enclosed by []... and equal signs used to separate argument names from the value.

Responses, success versus error, theory and severity

Single vs multiline replies

In the replies, kill the ok/error etc. Fixed message text. Plus optional, trailing freeform box "+[*freeform*]" "-[*error*]" or "=[*alert*]" ... do let supporting systems enable/disable this using the verbose command? Aka "VERB" from SMTP.

MAIL TRANSACTIONS

Message transactions.

OBJECTS

Signets and Messages

Modifications - tracing

DELIVERY

Addressing

Validation steps

Mail stores

Bounces

CACHING

How to handle the caching of signet lookups.

CONNECTIONS

A consumer begins the process of initiating a DMTP connection by retrieving the management record for a target domain name. If no management record is discovered, then a consumer should rely upon its local cache until any previous entries have reached their expiration. If no management record is found, or the cached record has expired, a consumer must conclude the target domain is not DIME-enabled and does not support DMTP. If a Mail Transfer Agent (MTA) is attempting to establish a DMTP connection for the purpose of message delivery, it should consider the error temporary and apply the retry logic supplied below.

If a management record is found, and it contains a valid delivery field (dx) value, consumers should first attempt to resolve and connect to the provided values using port 26 in single protocol mode. If multiple delivery field values are encountered, a conforming implementation must attempt at least three unique host names before considering the single protocol mode attempt a failure and continuing on to try dual protocol mode. It is recommended that consumers attempt the supplied host names in a random order, independent of what order they appear in a management record.

If the management record does not contain a valid delivery field, or the consumer is unable to establish a DMTP connection in single protocol mode, it must fall back into dual protocol mode. To find the dual protocol hosts for a domain, consumers must query the target domain name for a mail exchange (mx) resource record. If a valid mail exchange record is found, a conforming client must attempt the connection using port 25, and if that fails, may attempt the connection using port 587. If a TCP connection is established, and then the consumer a consumer should apply rules specified below for DMTP hosts operating in dual protocol mode. A conforming client must attempt at least three unique mail exchange resource record host names before continuing.

If a consumer is unable to establish a connection using the logic above, it may consider the attempt a failure, or optionally attempt to establish a DMTP connection using the target domain name. If a consumer attempts to establish the DMTP connection using the target domain name, it should attempt a connection on port 26 first, and apply the rules associated with single protocol mode. If the connection fails using port 26, then a consumer should attempt a dual protocol connection using port 25, and if that fails, may choose to also attempt a dual protocol mode connection using port 587.

Valid delivery field values and mail exchange resource records must always be fully qualified host names that resolve to A or AAAA resource records. The use of IP addresses, or a CNAME is prohibited and conforming implementations should ignore such values.

If an MTA is unable to locate a valid management record, or establish a DMTP connection using any of the supplied host names, it should consider the error temporary. If the policy field in the management record indicates a domain is operating in experimental mode, then a mail transfer agent may continue using SMTP [SMTP]. Otherwise a conforming MTA must queue and periodically retry the delivery attempt for at least 72 hours. The algorithm used to schedule retries is intentionally undefined, but a conforming implementation must ensure it will retry delivery at least once every 12 hours. An MTA should use a gradually increasing delay between delivery attempts, provided the interval never exceeds

12 hours. If a consumer is unable to establish a DMTP connection during the required 72 hour period, it must consider the error permanent and report its failure to deliver the message back to the author.

An MTA may choose to report temporary failures after 4 hours, but must continue making delivery attempts for a during the entire 72 hour period unless a user intervenes.

All DMTP connections must be secured using TLS v1.2 [TLS] and the cipher suite ECDHE-RSA-AES256-GCM-SHA384[17] [TLS-ECDHE]. The required cipher suite is uniquely identified during a TLS handshake by the octet values oxCo, ox30.[18]

CERTIFICATES

DMTP connections must always be secured using TLS [TLS]. This will require that servers be configured to supply an X.509 certificate during the connection. The certificate provides a signed RSA public key, along with a number of other attributes. Certificates supplied by DMTP hosts must use RSA keys that are least 2048 bits, and keys of at least 4096 bits are strongly recommended. DMTP client implementations must support RSA keys up to 8192 bits in length, and should support RSA keys of 16384 bits in length. If a conforming DMTP consumer encounters a host using an RSA key that is shorter than 2048 bits, it should complete the TLS handshake and immediately shutdown the connection using the QUIT command specified below.

DMTP hosts must allow consumers to specify the intended host name for the connection using the Server Name Identifier (SNI) extension in single protocol mode, and as an argument to the STARTTLS command when operating in dual protocol mode. If a DMTP host is configured with a TLS certificate containing a Common Name (CN) or Alt Name (AN) attribute matching the supplied host name, it must supply the matching certificate. If the DMTP host does not have a matching TLS certificate, it must allow the connection to proceed using a default TLS certificate. Every DMTP host must be configured with a default TLS certificate.

If the management record provided TLS field values, then consumers must validate TLS certificates against the supplied values. TLS field values are Ed25519 signatures [EDDSA], and generated using the target domain's Primary Organizational Key (POK). If a TLS field value is found, a certificate must be confirmed against one of the supplied field values. Note that it is possible for a management record to supply more than one TLS field value, in which case all of the supplied values must be compared until a matching entry is found. If none of the supplied signatures can be validated, then a consumer must terminate the DMTP connection and notify the user of an error with possible security implications. TLS certificates must be converted to a concrete data stream using the Distinguished Encoding Rules (DER).

17 The NIST name, and the one reused by the referenced TLS standard is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384.

18 Should we require ECDHE-ECDSA-AES256-GCM-SHA384 instead? That would allow us to avoid RSA altogether. The problem is that currently very few CA's publish certificates signed using ECDSA. Alternatively, should we make support for the DHE variants optional? That is DHE-RSA-AES256-GCM-SHA384 or DHE-ECDSA-AES256-GCM-SHA384?

If a management record is validated by a DNSSEC signature, and the certificate was validated against a TLS field value, then a consumer must accept certificates that would normally be rejected using strict validation. This means consumers must accept certificates which have been validated using a signed management record that are self-signed, expired and/or lack a matching Common Name (CN) or Alt Name (AN) attribute.

If a certificate is confirmed using the TLS field value, then a consumer should not perform the Online Certificate Status Protocol (OCSP) check [OCSP]. OCSP checks are discouraged if the certificate can be validated using the TLS field even if the management record is not signed using DNSSEC because the request could inadvertently leak information about which domains a host is contacting. All other X.509 validation rules should be applied according to the TLS v1.2 specification regardless of whether the certificate is validated using the management record.

DNSSEC Validation	TLS Field Validation	X.509 Validation	OCSP Check	Result
Yes	Matches	N/A	N/A	Pass
Yes	None	Passed	Yes	Pass
Yes	None	Failed	N/A	Fail
Yes	Mismatch	N/A	N/A	Fail
No	Matches	Passed	Skip	Pass
No	Matches	Failed	N/A	Fail
No	None	Passed	Yes	Pass
No	None	Failed	N/A	Fail

SINGLE PROTOCOL MODE

If a consumer is using the host name supplied by the delivery field in the management record, it must connect to the provided host using port 26. The connection should be initiated using TLS [TLS]. Consumer should supply the host name provided by the delivery field, or "dx" value, in the management record using SNI TLS extension [TLS-SNI]. Connections to port 26 must be specifically for DMTP and upon successfully connecting, consumers must see a banner that starts with the sequence 220 and contains the string "DMTP". Single protocol mode greetings must match the pattern:

```
220 <domain.tld> DMTP {freeform}
```

If a consumer does not encounter the appropriate DMTP protocol banner once the TLS connection has been established, it must immediately shutdown the connection and treat the host name the same way it would if the TLS connection had never succeeded. Consumers conforming to this specification must make TLS connection attempts to at least three valid, and unique, delivery field host names before continuing. If fewer than three unique and valid delivery field host names values are found, then a consumer should try all of the unique and valid host names it encounters.

DUAL PROTOCOL HOSTS

When attempting a DMTP connection using a hostname supplied by a mail exchange (mx) resource record, a consumer should assume the host is operating in dual protocol mode and attempt an unencrypted TCP [TCP] connection using port 25. If port 25 fails, a consumer may attempt the TCP connection using port 587. If either port results in a TCP connection, the consumer should confirm whether a host supports DMTP by parsing the banner greeting before proceeding. A DMTP

capable host operating in dual protocol mode must greet consumers with a banner that starts with 220 and contains the string "DMTP". Dual protocol mode greetings must match the pattern:

```
220 <domain.tld> ESMTP DMTP {freeform}
```

If the appropriate banner is encountered, a consumer must immediately elevate the TCP connection into DMTP mode using the STARTTLS command syntax specified below. If the consumer does not encounter the appropriate dual protocol banner, it must fail immediately and continue as if the TCP connection never succeeded. To elevate a successful TCP connection into DMTP mode, consumers must initiate a TLS handshake using the STARTTLS command syntax:

```
STARTTLS HOST=<domain.tld> MODE=DMTP
```

If a dual protocol host encounters a MODE parameter for a consumer attempting to elevate a connection into DMTP mode, but is unable to negotiate a TLS connection using the cipher suite specified above, then the host must ensure a STARTTLS command fails. If a consumer does find that it connected to a host that allows elevation into DMTP, without using the required cipher suite, it must immediately issue a QUIT command and shutdown the connection. Consumers that encounter this scenario should alert the user to a possible security threat before proceeding.

A connection which has been successfully elevated into DMTP mode must receive a reply with the status code 250, and a response string which contains "DMTP". The response should match the following pattern:

```
250 OK DMTP {freeform}
```

Dual protocol implementations may choose to allow consumers to issue the STARTTLS command with the MODE parameter missing, or with a MODE parameter that supplies the value SMTP. Hosts that support TLS connections in SMTP mode must ensure the connection does not allow consumers to use any commands using the DMTP protocol syntax. Attempts to issue DMTP commands must result in a response code of 501, denoting an invalid syntax. The lone exception to this rule is the MODE command, which must result in response indicating the connection is in SMTP mode. For consumers which successfully establish a TLS connection but remain in SMTP mode, the greeting should be a 250 response code matching the following pattern:

```
250 OK ESMTP {freeform}
```

Note that both STARTTLS responses supplied above indicate the current protocol mode after completing the TLS handshake using the third token in the response. Thus consumers conforming to this specification must consult the server response returned by a host in response to the STARTTLS command and ensure the connection is using the appropriate protocol mode before proceeding.

Once a connection has been secured using the STARTTLS command a host conforming to this specification must reset all state information for the connection. This includes discarding the hostname values provided as parameters to the HELO or EHLO commands. If a consumer is proceeding with the connection in SMTP mode then it must issue the HELO or EHLO command before attempting to transfer a message.

TIMEOUTS

Consumers must provide a timeout mechanism for unresponsive server connections, while the enforcement of connection timeouts remains optional for server implementations. Timeouts should be calculated based on the amount of time that has lapsed since a complete line has been transmitted or received. If an implementation is unable to track timeouts based on when the last complete line DMTP protocol line was sent or received, the recommended alternative is to rely on the amount of time since any DMTP characters were sent or received. We strongly recommended avoiding a strategy of relying on the time elapsed since a TLS message, or TCP packet was observed. It is possible for TLS connections to exchange TCP packets indefinitely without ever exchanging any DMTP protocol data.

Server implementations lacking the ability to track timeouts based on the last DMTP character transmitted, or lack support for timeouts altogether, will waste resources on paralyzed client connections. However, if a consumer lacks support for tracking timeouts based on DMTP protocol data, it could result in unnecessary user distress. For a Mail User Agent (MUA), this could result in lengthy send operations, as the User Privacy Agent (UPA) waits for a signet resolution to complete. If the consumer is a Mail Transfer Agent (MTA), this issue could result in messages being rejected, or bounced, because they were delayed beyond the expiry threshold for a stale user signet.

CONSUMERS

Below are the recommended timeouts a consumer should use for the different categories of possible DMTP operations. However, a more sophisticated implementation may choose to use timeouts based on a higher level of granularity than what's provided here. In our experience, such an implementation should be patient when it comes to waiting on commands which involve a large transmission, whether its sending a message or receiving a signet, and with commands which involve the setup process for a DMTP connection, such as the TCP connection setup, a TLS handshake, or the receipt of an initial greeting once the TLS channel has been established. These operations could employ multiple systems, any of which could be suffering from congestion. The following timeouts are intended only to be recommendations, with the one exception being the amount of time between when an MTA finishes transmitting a message and the receiving host acknowledges its acceptance. An MTA must wait at least 8 minutes for such an acknowledgement and may want to wait longer if a message was particularly large.

	Timeout Range *Recommended*
Connection Setup Operations *TCP s*etup, TLS handshakes, Connection *Banners*	4 to 8 minutes
Protocol Mode Elevation Commands	4 to 8 minutes

STARTTLS	
Global Commands *HELO, EHLO, MODE, RSET, NOOP, HELP*	2 to 8 minutes
Mail Processing Commands *MAIL, RCPT, DATA*	8 to 16 minutes
Signet Retrieval Commands *SGNT, HIST, VRFY*	1 to 4 minutes
Connection Termination Command *QUIT*	1 to 2 minutes

SERVERS

A server implementation must use a timeout of at least 1 minute. Servers should employ timeouts between 4 and 20 minutes, with a timeout of 10 minutes being recommended. We recommend that servers that normally employ a timeout shorter than 1 minute, increase their timeout to 4 minutes while processing a mail transaction. This means increasing the timeout after a successful MAIL command, until the transaction is concluded.

While the practice of enforcing timeouts based on the overall time for a DMTP command to complete is not recommended, if a server does employ this strategy, it must ensure consumers are allowed a minimum of 30 minutes to complete the transmission of messages following a successful DATA command, and a similar minimum of 30 minutes to finish receiving a multiline response following a successful SGNT, HIST, or VRFY command.

Sophisticated server implementations may want to dynamically adjust their timeouts based on network congestion to differentiating between TCP congestion and a client that ceases to transmit packet acknowledgements. It may also want to differentiate between the timeout it employs while receiving or transmitting data, and the time it waits for an idle connection to send a DMTP command.

The above timeouts are intended to apply while server is operating normally and should not apply to servers which are in the process of shutting down.

TERMINATION

A DMTP connection should, under normal conditions, only be terminated in response to a consumer sending the QUIT command. Consumers sending this command should wait for the server to acknowledge the receipt of a QUIT command transmit a positive reply.

A server must not intentionally choose to unilaterally terminate a DMTP connection under normal operating conditions unless a consumer has exceeded the configured timeout, or is in the process of shutting down. Specifically, servers must not terminate DMTP connections in response to an unknown command, because of syntax violations, or because a command was sent out of order.

If a server does encounter a situation where it needs to unilaterally close a DMTP connection, it must first transmit a line starting with the status code 421, to indicate the abnormal closure. Presumably, a consumer will receive and process the response as a reaction to a previously transmitted command, or asynchronously as the response to its next command. A server should follow this transmission by attempting to cleanly shutdown the TLS connection. DMTP clients must always be prepared to handle the abnormal shutdown of a connection. This means gracefully handling a DMTP reply which starts with the status code 421, or being notified the of a TLS shutdown.

If an MTA experiences an abnormal shutdown during a message transfer operation, it must treat the delivery attempt as if a response code of 451 was returned, and ensure the message delivery attempt is retried.

Sometimes abrupt communications failures can result in the unexpected closure of connections. Despite being a violation of this specification, this situation will inevitably, and unavoidably, arise and must be handled gracefully. The robustness of DIME depends upon implementations being able to handle failure and retry the aborted operation using a different DMTP host, or when the original host comes back online.

GLOBAL COMMANDS

The following commands must always be available to consumers regardless of the protocol mode or connection state: HELO, EHLO, MODE, RSET, NOOP, HELP, VERB, and QUIT. This includes a connection to a dual protocol host that has not been elevated into DMTP mode.

For signet resolvers, issuing a HELO or EHLO command is not recommended, and should be avoided to prevent the unnecessary leakage of meta-information about a consumer. If the consumer is an MTA, it must send either the HELO or EHLO command before attempting a mail transaction. If the MTA is connected to the host using the dual protocol mode, it must send, or resend, the HELO or EHLO after the connection has been elevated to DMTP.

HELO

Consumers may use the HELO command at any time. If the host already has a host name stored for the current connection, it must replace the stored value with the newly issued host name. Unlike the EHLO command below, the HELO command does not list the supported protocol extensions in its reply. This command requires a single parameter in the form of a fully qualified domain name. If the consumer does not have a meaningful host name to supply, it should send an address literal. If a host name is supplied, it should resolve to an address literal matching the current connection. The HELO command uses the following syntax:

```
HELO HOST=<host.domain.tld>
```

Successfully issued HELO commands must result in a single line reply, with a response code of 250 in the form:

```
250 OK {freeform}
```

EHLO

The EHLO command is identical to the HELO command above with one notable exception. A successful EHLO command will result in a reply that lists the protocol extensions supported by a DMTP host. If the host does not support any protocol extensions, then it will result in a reply that is identical to the HELO command. This command requires a host name as the first argument, and uses the syntax:

```
EHLO HOST=<host.domain.tld>
```

The EHLO response may use the multiline response structure. The additional lines will provide keywords, with each corresponding to a protocol extension. DMTP hosts operating in dual protocol mode must return the DMTP and STARTTLS keywords in their response to the EHLO command for connections that have not issued the STARTTLS command and successfully completed a TLS handshake. The following is a potential EHLO response returned by a dual protocol host on a connection that has not been elevated into DMTP mode:

```
250-DMTP
250-STARTTLS
250-PIPELINING
250-SIZE 33554432
250 OK {freeform}
```

In contrast, the EHLO response for single protocol connections, or a dual protocol mode connection that has been elevated into DMTP mode, should never include the DMTP or STARTTLS keywords. The following is a potential EHLO response returned to a consumer over a DMTP connection:

```
250-PIPELINING
250-SIZE 33554432
250 OK {freeform}
```

Note that when a consumer connects to a dual protocol host, it must discard the list of protocol extensions returned by an EHLO command submitted before the connection was elevated. Dual protocol hosts are likely to send a list of protocol extensions after a connection has been elevated into DMTP that is distinctly different from the list sent before elevation.

MODE

The MODE command is the only DMTP command that a dual protocol host should accept before a connection is elevated into DMTP. The MODE command accepts no arguments and is used by consumers to confirm the protocol mode for the current connection. A consumer may issue the MODE command using the following syntax:

```
MODE
```

The response to a MODE command matches the reply issued after a successful STARTTLS command. The 250 response code should be followed by the current protocol mode for the connection in the third token. A DMTP connection must result in a reply matching the syntax:

```
250 OK DMTP {freeform}
```

In contrast a connection operating in SMTP mode, must reply with the following response regardless of whether the connection has been secured using TLS:

```
250 OK ESMTP {freeform}
```

For legacy servers which lack support for DMTP, the MODE command should result in a 500 response code to indicate the MODE command was unrecognized. Hosts which are DMTP capable, but currently have DMTP support disabled, should reply using the 502 response code to indicate the MODE command was recognized, but currently has DMTP support disabled.

RSET

The RSET command is used to reset the state information for a connection. It operates in a fashion that is similar to the STARTTLS command specified above, with the exception that the RSET command does not destroy a host name which was supplied as an argument to the HELO or EHLO commands. The RSET command accepts no arguments and uses the following syntax:

```
RSET
```

A RSET command that is accepted by a server, will return a 250 response code to indicate the state table was successfully reset, with the reply conforming to the syntax:

```
250 OK {freeform}
```

If a consumer encounters a response code other than 250, it must clear the state table by disconnecting from the DMTP and reconnecting.

NOOP

The NOOP command is used by consumers to keep a DMTP connection alive, and should result in no operation being carried out by either host. The NOOP command does not require an argument, but servers must accept NOOP commands that supply command arguments provided they conform to the limitations specified above. This means the entire command line, including the line terminator, must be 512 octets or less in length and only contain valid ASCII character values. The command uses the syntax:

```
NOOP {freeform}
```

Valid NOOP commands must result in a reply using the 250 response code and match the pattern:

```
250 OK {freeform}
```

HELP

The HELP command is used by administrators on interactive DMTP connections to retrieve the list of commands supported by the DMTP host. Support for this command is optional. The command itself does not accept an argument, and is issued using the syntax:

```
HELP
```

Servers with the HELP command enabled may use the multiline response structure and must reply using a response code of 214. It is recommended that server implementations always return the list of available DMTP commands in alphabetical order. The following reply includes a listing of the DMTP commands a host is required to support:

```
214-DATA
214-EHLO
214-HELO
214-HELP
214-HIST
214-MAIL
214-MODE
214-NOOP
214-QUIT
214-RCPT
214-RSET
214-SGNT
214 VRFY {freeform}
```

A DMTP host may choose to disable support for the HELP command. To indicate this, a DMTP server should reply using the 502 response code to indicate the HELP command was recognized but has been disabled:

```
502 COMMAND DISABLED
```

QUIT

The QUIT command terminates a connection gracefully. A DMTP host must initiate send the appropriate response and subsequently initiate a controlled shutdown of the TLS connection. The QUIT command accepts no arguments and uses the syntax:

```
QUIT
```

A DMTP server must acknowledge its receipt of a QUIT command by transmitting a reply using the 221 response code:

```
221 BYE {freeform}
```

If a consumer does not receive a reply from the DMTP server a timely fashion, it may choose to begin the shutdown process in accordance with the TLS protocol specification. [TLS]

MESSAGE TRANSFER COMMANDS

The following commands are used to transfer messages between organizations using atomic mail transactions. The commands have been constructed for securely and reliably delivering messages while minimizing the amount of metadata a compromised handling agent is capable of leaking. The commands described in this section must not be sent by a consumer, or accepted by a server, until either the HELO or EHLO command have been sent. These commands require a consumer to provide its fully qualified host name, and for a server to indicate its acceptance of the value by replying with a successful status code. If any of the commands in this section are submitted before a successful HELO or EHLO, a server must respond with the status code 503 to indicate an invalid sequence of commands.

A mail transaction is an atomic transaction requiring a consumer to send, and a server accept all three commands specified in this section, in the sequence: MAIL, RCPT and DATA. If the RCPT or DATA commands are received out of order, then a server must respond with a 503 status code to indicate an invalid sequence of commands.

If the RSET command is received before a mail transaction is completed, then any pending mails transactions must be aborted. A conforming MTA must ensure it retains responsibility for a message until it receives a successful response to the DATA command. This concludes the mail transaction and transfers responsibility for delivering message to the destination host. If a message is accepted by a destination, and it encounters a problem delivering a message, it must generate and deliver a bounce back to the origin domain.

MAIL

The MAIL command is used to start a new mail transaction. The command has two required arguments. The FROM argument must be sent first and is followed by the FINGERPRINT argument. FROM is used to provide the origin domain name for a pending message, while FINGERPRINT provides the full fingerprint for the origin signet required by a destination host to authenticate the organizational signature attached to the pending message.

A destination host must ensure it has a cached copy of referenced origin signet referenced before replying with a successful status code. If it the origin signet has not already been stored, a destination host may choose to delay sending a response to the MAIL command until it has successfully retrieved, and authenticated the origin signet. However, if this simultaneous retrieval attempt does not completed within 4 minutes, a destination host must reply with the status code of 470. A destination should immediately reply with the status code 470 if it prefers, or is unable, to perform a simultaneous origin signet lookup. The response code 470 is used to indicate an origin signet is temporarily unavailable, and that an MTA must queue the message and reattempt the transfer in the future. If a destination host repeatedly tries and fails to retrieve an origin signet for 72 hours, it should return the response code 570 to any MAIL command referencing the origin signet in question. The response code of 570 is used to indicate the prolonged failure to retrieve the origin signet. The destination host should continue making retrieval attempts until it succeeds, or if an additional 72 hours lapses without encountering a reference to origin signet in question.

A server must only respond to a MAIL command with a success response if a new mail transaction is started. If an MTA sends the MAIL command before completing the pending transaction has been completed, a server must abort the

previously started transaction before evaluating the newly submitted MAIL command. As a result, servers must produce identical results for MAIL commands regardless of any potentially pending mail transactions. This also means the outcome of a MAIL command resulting in an error must be semantically equivalent to the outcome of an RSET command resulting in success; both must result the pending mail transaction being aborted without starting a new transaction.

The syntax used to submit a MAIL command with its two required parameters is:

```
MAIL FROM=<domain.tld> FINGERPRINT=[fingerprint]
```

If an MTA attempts a MAIL command before it submits a valid HELO or EHLO command, then a server must respond with a response code of 503 to indicate the invalid command sequence:

```
503 INVALID COMMAND SEQUENCE {freeform}
```

If the submitted MAIL command references an origin signet which is unavailable on the destination host, a server should reply using the status code 470, which must result in the message being queued and retried. The syntax for the 470 status message is:

```
470 ORIGIN SIGNET UNAVAILABLE {freeform}
```

If a destination has been attempting to retrieve the reference origin signet for at least the previous 72 hours, then it should indicate a permanent origin signet failure using the status code 570 and the syntax:

```
570 ORIGIN SIGNET UNAVAILABLE {freeform}
```

If the origin domain lacks a management record, or the authoritative server for the origin domain returns an error when the referenced signet is requested, then a DMTP host should respond using the error code 575 to differentiate it from an origin signet timeout:

```
575 INVALID ORIGIN SIGNET
```

If a destination host does have the referenced origin signet available in its cache, it should allow the transaction to proceed by returning a response code of 250 using the syntax:

```
250 OK {freeform}
```

If the fingerprint does not match what the destination has in its cache for domain.tld, this command would initiate a side channel lookup.

RCPT

The RCPT command is used to confirm a message is being delivered to the correct host and was created using a current and available destination signet. It requires two arguments, TO parameter and the FINGERPRINT parameter. The TO parameter must provide a target domain that the destination host is configured to accept messages for. The

FINGERPRINT parameter provides the full fingerprint for the destination signet used to encrypt the recipient information. The RCPT command uses the following syntax:

```
RCPT TO=<domain.tld> FINGERPRINT=[fingerprint]
```

If the destination host needs to reject a message because the fingerprint indicates the recipient information was encrypted using an expired or otherwise invalid destination signet, it should respond with a status of 576, clear any state information associated with the mail transaction and use the syntax:

```
576 INVALID DESTINATION SIGNET {freeform}
```

If a RCPT command is submitted twice in a single mail transaction, the second attempt must be rejected using the 431 response code. A DMTP mail transaction is only capable of being associated with a single recipient, so if a RCPT was already accepted, the resource limits would be exceeded by accepting a second RCPT command. The limitation is a byproduct of the D/MIME format, which intentionally limits the envelope to a single recipient, which prevents anyone from discovering how many people a message was originally addressed to. This requires that a message be transferred separately for each recipient as standalone mail transactions. To indicate a rejection resulted from this limitation server should use the following response syntax:

```
431 DESTINATION LIMITS EXCEEDED {freeform}
```

If the RCPT parameters indicate a recognized destination domain and the fingerprint indicates the embedded recipient information will be accessible, a server should reply using the status code 250 to allow the MTA to proceed with the transaction by sending the message data. The success response syntax is:

```
250 OK {freeform}
```

DATA

The DATA command is used to transfer a D/MIME message to the destination host. Provided the MTA has successfully issued MAIL and RCPT commands, a DATA command should result in a 354 response code, indicating the destination host is ready to receive the message. A client should proceed to transmit the D/MIME message in its ASCII armored form. The transmission sequence is terminated by the string "<LF>.<LF>" which is sent to indicate the transmission has been completed. The sender must then wait at least 8 minutes for a reply, presumably a sender should wait at least 1 additional minute for every megabyte used by the transmitted message. If the DMTP host responds with the code 254, then the mail transaction is complete. A response code between 400 and 499 will indicate the current attempt failed, but the issue was temporal and the sender should retry the transmission later. A response code over 500 indicates a permanent failure, that the error is likely to persist, and that an origin host should proceed to notify the author of the failure. Once the 254 code has been sent, responsibility for the transmitted message shifts from the origin MTA to the destination MTA. If a DMTP host, after having transmitted the 254 response code, discovers that it is unable to deliver a message, then it must bounce the message back to the origin to ensure the author is properly notified of the failure.

Alternatively, if a message is delivered, but the 254 response code is never received by the sender, because it disconnected before the 254 response was received, when it retransmits the message, it is possible the retransmission will only result in the message being duplicated in a recipient's mailbox. Sophisticated server implementations may want to detect this issue by tracking the cryptographic hashes for any recently delivered messages and compare those hashes against incoming messages. If a duplicate message is detected, then a host may return the response code 255 which indicates the message successfully delivered on a previous attempt.

The transmission process begins with a DATA command. The default DATA command does not allow arguments to be included. The syntax for the command is:

```
DATA
```

If the DMTP host is ready to receive the message it will respond using the 354 banner shown below:

```
354 READY TO RECEIVE MESSAGE
```

Once a sender sees the 354 response, the sender may begin transmitting the message. The sequence "<LF>.<LF>" is used to terminate the message transmission:

```
-----BEGIN ENCRYPTED MESSAGE-----
Bv0AAdcBhvAmjVKiMZmjF8gTnXNTDZ4C1W8MSWfh5NLIdzquujQCBJkg4dcp7m8tjp7JFrWkowv
1bp1a1pIJNyIbbh
Y0CpFaF4z2L8mjcJq5Pl+J/lF4iKrJc7tJYWCueGeJiYgQci0vKUiRHqyr1wkjMUbmdY954udPi
AVzHJplUj6ZtjdA
bSeJhM4nrLzQe5wXR6n8fMDsHtJvZNb1PZSMycs7rMoNDEY6pjjo8Y70k0E3jLy9SHcCBhA78k9
y8JEDzT7M7Udi8o
wooUGwENp3upYuhxd/bzoZg53TdQbNM2RKcGKozSQK2gHKpFI59gjwcZBUxhZGFyGwRDYXZlHhJ
Db3VudHJGggY2FX
FXppcGNvZGUgd2l0aG91dCBjYXZlcyADNDExfjd0pQ0k4DXvBfUNNFxir+IzghryyCr67G9jEa4
4VD8Q1EW1xC/TF2
mfylpmL2iueTyPz50kAY9Qd/EWxhZGFyQGxhdmFiaXQuY29tgI7gaXq2Nu7dVKmu8i78jjB1uOe
U8VbjZQUM9L79Wu
dMC2yD4vW76cGkb8hrGL/y8H0IshRpNeOAM
-----END ENCRYPTED MESSAGE-----
.
```

Upon receiving a message, a server must ensure the D/MIME is structurally correct and contains a valid organizational signature from the origin domain before accepting a message. If the message is does not contain a valid D/MIME binary structure, the DMTP host should immediately return a 451 error code:

```
451 DATA CORRUPTED
```

If the organizational signature for the message is missing, or invalid, then a server must return the 578 error code:

```
578 INVALID ORIGIN SIGNATURE
```

A server may also decrypt the destination portion of the D/MIME message and confirm the validity of the recipient, and whether the recipient signet used to encrypt the message is either current, or is within the expiry threshold for stale user

signets. Alternatively, a DMTP host may also accept a message and commit to bouncing it later if these checks fail. If the message is validated before responding, and the recipient mailbox is invalid, or not affiliated with the destination domain provided by the RCPT command, then a DMTP host must respond using the 510 error code:

```
510 INVALID RECIPIENT
```

If the message was encrypted using an expired user signet, then a DMTP server must respond using the 586 error code:

```
586 INVALID RECIPIENT SIGNET
```

If received message passes all of the checks described above, then the message should be queued for delivery and the 254 response code returned to the sender along with a cryptographic hash of the binary message data received, and transmitted in its base64 encoded form, without padding, yielding a message transaction identifier that is precisely 86 bytes long:

```
254 ACCEPTED=MUYwQkNENDY0MzE3OEQyOTAxRDEwMjlFThDQUZEOTM4NkY5NFE5RDE5NTUxMg
```

If the host tracks the cryptographic hashes for recently accepted messages, and a duplicate message is detected, then it should return the 255 response code to indicate the message has already been delivered. This response must only be returned if the message has already been delivered to the mailbox. If the previous transfer attempt failed, then it must not be considered a duplicate. Successful deliveries result in a response using the syntax:

```
255 DUPLICATE=MUYwQkNENDY0MzE3OEQyOTAxRDEwMjlFRUREOTM4NM4NkY5NjERDE5NTUxMg
```

Regardless of the response code, a sender must consider the mail transaction terminated. If it intends to retransmit the message, or begin the transmission of a different message, it must begin the command sequence again using the MAIL command.

SIGNET TRANSFER COMMANDS

SGNT

The SGNT command is used to retrieve user and organizational signets from an authoritative source using DMTP. The command requires a consumer to supply the DOMAIN argument, and may be submitted along with the FINGERPRINT argument. The DOMAIN argument must be sent first, and a FINGERPRINT value, if supplied, must be sent second. The SGNT command syntax for retrieving an organizational signet is:

```
SGNT DOMAIN=<domain.tld>
```

An almost identical syntax is used for retrieving user signets, with the syntactical exception that a mailbox is supplied using the USER argument:

```
SGNT USER=<mailbox@domain.tld>
```

A consumer may want to retrieve a specific version of a user or organizational signet, possibly because the fingerprint was supplied using the MAIL command above, or because it is trying to retrieve a signet referenced elsewhere. To retrieve a specific signet a consumer would use the second optional argument, which accepts a full fingerprint for the requested signet, after it has been base64 encoded, and the padding bytes removed. The resulting values for the FINGERPRINT argument should be exactly 86 bytes. The complete syntax for SGNT command syntax when retrieving a specific organizational signet:

```
SGNT DOMAIN=<domain.tld> FINGERPRINT=[fingerprint]
```

When a consumer is requesting a specific user signet, it may supply either the full fingerprint or the cryptographic fingerprint for the user signet it wants to retrieve. Note that a server must always return a full signet in response to the SGNT command, even if a cryptographic fingerprint is submitted. The syntax for retrieving a specific user signet is:

```
SGNT USER=<mailbox@domain.tld> FINGERPRINT=[fingerprint]
```

A conforming server implementation must only return organizational signets for domains in which it is the authoritative source. If the requested organizational signet is available, it must be returned in its ASCII armored form, and if the fingerprint argument is omitted, a host must return what is considers the current organizational signet for the supplied domain name. When returning an organizational signet, a server must use the multiline syntax and the 270 response code:

```
270------BEGIN ORGANIZATIONAL SIGNET-----
270-
BvAAAWEBQt1Wjk8S+DkuEbOLgfQTvVyS7Ae7NjwonNLI+TRoDYUCBOYleb/SnE7FZjZYsjv+
270-
BpyT6l4bHZj3Pd0s9QGE0rXCy9PWsCPwAmFC2aVVcG3NTaONDtmz3LS1lKgkFv9B/wB8hkLT
270-
dCBMTEMbGzEyMyBIaWRkZW4gQnVua2VyIEJvdWxldmFaW5ndWxhcmlhHwUwMDAxMSALMw5PU
270-
EqMnb0cbDDFBatu9tTMAi7ERNkWGLqWda2IG0EWjp7QF/qC0byTh7Is+YexkCT+xz0yL3ALb
270-
dC5jb22AFjee+3raziK2GZYoFErVAsJKXbRay9fY/GNihmZgd9SBZrJUnu8XA99RKQrlnn12
270------END ORGANIZATIONAL SIGNET-----
270 OK {freeform}
```

Like organizational signets, a conforming server implementation must only return user signets for domains in which it is the authoritative source. If the requested user signet is available, it must be returned in its ASCII armored form, and if the fingerprint argument is omitted, a host must return what it considers to be the current user signet. When returning a user signet, the full signet must always be returned, even if the consumer supplies a cryptographic fingerprint. When returning a user signet, the server must use the multiline syntax and the 280 response code:

```
280------BEGIN USER SIGNET-----
280-Bv0AAdcBhvAmjVKiMZmjF8gTnXNTDZ4C1W8MSWfh5NLIdzcp7m8jklKZtjp7JFrWkNyIbb
280-hjaxY0CpFaF4z2L8mjcJq5Pl+J/lF4iKrJc7tJYWCueGeJiYgQci0vKUiRHqyY9Zo5dPiA
280-6ZtjmmdAbSeJhM4nrLzQe5wXR6n8fMDsHtJvZNb1PZSMycs7rMoNDEY6pjjocCBhA79y8J
280-7AeM7Udi8oGAwooUGwENp3upYuhxd/bzoZg53TdQbNM2RKcGKozjwcZBUxhZGFyGwRDZlH
280-IHdpdGggY2F2ZXMfFXppcGNvZGUgd2l0aG91dCBjYXZlcyADNXBXvBfUNNFxir+IzghyCr
280-8Q1E3j7EqVW1xC/TF2KGmfylpmL2iueTyPz50kAY9Qd/EWxhZGFyQGxhdmFiagI7gaXq8i
```

```
280-8Vbjj47aXZzQUM9L79WuqTuLdMC2yD4vW76cGkb8hrGL/y8H0IshRpNeOAM
280------END USER SIGNET-----
280 OK {freeform}
```

If a fingerprint parameter is provided, then a host must return the signet matching the fingerprint, or an error. If no signet is available for the requested address, then a server must also return a 486 or 576 error. A server may optionally apply fuzzy matching logic to a non-matching identifier based on common alternate representations of a domain or email address and suggest the true identifier using the temporary error code 486. Multiple potential matches are allowed, with each potential identifier provided on a separate line. The following example shows a possible server request using an international character:

```
SGNT USER=<ũser@example.tld>
```

In a situation where the precise email local part identifier does not match a user signet, but several similar identifiers exist on the system, the possible response might be:

```
486-IDENTIFIER=<ūser@example.tld>
486-IDENTIFIER=<ŭser@example.tld>
486-IDENTIFIER=<ůser@example.tld>
486-IDENTIFIER=<űser@example.tld>
486-IDENTIFIER=<ųser@example.tld>
486 IDENTIFIER=<user@example.tld>
```

Whether suggestions are returned based on greedy matching is optional, and precisely what logic is applied to an identifier is localized to the host locale, and a signet resolver must exercise caution when accepting such suggestions. If no matching users are encountered, then a permanent 576 error must be returned:

```
576 SIGNET UNAVAILABLE
```

If the domain or address is valid, but the signet is unavailable, a server may choose to return the error code 476 instead. If the domain advertises a policy of experimental in its management record, then a consumer may choose to send the message using SMTP if this error is received. Otherwise clients must either retry the request later, or return an error to the message author.

```
476 SIGNET TEMPORARILY UNAVAILABLE
```

If the domain or email address is submitted valid identifier but does not precisely match the available does not precisely match an , but the signet is unavailable, a server may choose to return the error code 476 instead. If the domain advertises a policy of experimental in its management record, then a consumer may choose to send the message using SMTP if this error is received. Otherwise clients must either retry the request later, or return an error to the message author.

HIST

Allows a resolver to retrieve the chain of user signets between a trusted signet fingerprint (START) and a recently encountered user signet (END). If both fingerprint values are valid, then the host should return only the cryptographic signets published by the user between the two values. If the end fingerprint value is missing, the server must provide all of the cryptographic signets through the current user signet. This command must not be used to retrieve organizational signets.

The HIST command has one required argument, and two optional arguments. The USER argument is required and used to provide the email address being queried. The USER argument must always come first. If provided, the START argument must follow the USER argument, and provides the cryptographic fingerprint for a user signet at the start of a chain of custody query. The final argument is STOP and if included, provides a cryptographic fingerprint for the last user signet that needs to be returned. If the START parameter is missing, then a DMTP server should return the first signet from the user's current chain of custody. If the STOP parameter is missing, then the server should provide all of the user signets between the START value and the current user signet. If both arguments are missing, then a server must return the entire chain of custody for the current user signet.

```
HIST USER=<mailbox@domain.tld> START=[fingerprint] STOP=[fingerprint]
```

A DMTP server must be capable of providing the cryptographic signets in a user's chain of custody, from the root, all the way to the current user signet. A server may provide user signets beyond a user's current chain of custody, but should only return these if provided a starting fingerprint that reaches past the current user signet's root. Results are provided using the multiline syntax, and the 290 response code:

```
290------BEGIN USER SIGNET-----
290-Bv0AAQUBhvAmjVKiMZmjF8gTnXNTDZ4C1W8MSWfh5NLIdzqp7m8jklKZtjpwvYgpIJNyIbb
290-jaxY0CpFaF4z2L8mjcJq5Pl+J/lF4iKrJc7tJYWCueGeJiYgQci0vKMUbmdY9Zog5HJLN6Z
290-tjmmdAbSeJhM4nrLzQe5wXR6n8fMDsHtJvZNb1PZSMycs7rMoNDEY6jLy9S8JEDR8bF4R7A
290-M7Udi8oGAwooUGwENp3upYuhxd/bzoZg53TdQbNM2RKcGKozSQK2gHKpFI59gjwc
290------END USER SIGNET-----
290 OK {freeform}
```

If the start or end fingerprint values fall outside of the current user signet's current chain of custody, then a server may return the 576 response code. A server should also return the 576 response code if the user signet requested is unavailable. In a situation where the precise email local part identifier does not match a user signet, but several similar identifiers exist on the system, a host may use the 486 response to suggest possible matches:

```
486-IDENTIFIER=<ūser@example.tld>
486-IDENTIFIER=<ŭser@example.tld>
486-IDENTIFIER=<ůser@example.tld>
486-IDENTIFIER=<űser@example.tld>
486-IDENTIFIER=<ųser@example.tld>
486 IDENTIFIER=<user@example.tld>
```

Whether suggestions are returned based on greedy matching is optional, and precisely what logic is applied to an identifier is localized to the host locale, and a signet resolver must exercise caution when accepting such suggestions. If no matching users are encountered, then a permanent 576 error must be returned:

```
576 SIGNET UNAVAILABLE
```

VRFY

Allows a consumer to determine whether a signet is current. If a signet has been rotated, then the response will return the current signet. Signet resolvers should use "refresh" value provided by a domain's management record to determine how often it should confirm that a signet is current. This command must accept cryptographic fingerprints for users and full fingerprints for organizations, and should reject requests where the consumer supplies a full fingerprint for a user signet. The VRFY command requires the DOMAIN and the FINGERPRINT arguments, and uses the following syntax:

```
VRFY DOMAIN=<domain.tld> FINGERPRINT=[fingerprint]
```

If the organizational signet is current, the following is returned:

```
271 ORGANIZATIONAL SIGNET CURRENT {freeform}
```

Or to verify that a user signet is still current:

```
VRFY USER=<mailbox@domain.tld> FINGERPRINT=[fingerprint]
```

If a user address was supplied and the signet is still current:

```
281 USER SIGNET CURRENT {freeform}
```

Otherwise an update is returned using the same syntax as the SGNT command. Where an organizational signet uses the 270 response code:

```
270------BEGIN ORGANIZATIONAL SIGNET-----
270-AWEBQt1Wjk8S+DkuEbOLgfQTvVyS7Ae7NjwonNLI+TRoDYUCBOYl/SnE7p0FZjZYsA6W9j
270-BpyT6l4bHZj3Pd0s9QGE0rXCy9PWsCPwAmFC2aVVcG3NTXsQ5VhYPjK/l3aONDtmz3LS1l
270-dCBMTEMbGzEyMyBIaWRkZW4gQnVua2VyIEJvdWxldmFyZB4PUG9zdC1TaW5ndWxhcmlhHw
270-EqMnb0cbDDFBatu9tTMAi7ERNkWGLqWda2IG0oTP22njpchB2KEWjp7QF/qC0byTh7Is+Y
270-dC5jb22AFjee+3raziK2GZYoFErVAsJKXbRc2Zxu1Z3oXAJ1ay9fY/GNihmZgd9SBZrJUn
270------END ORGANIZATIONAL SIGNET-----
270 OK {freeform}
```

Or if a user address was supplied that was updated, the 280 response code is used to return the updated user signet:

```
280------BEGIN USER SIGNET-----
280-Bv0AAdcBhvAmjVKiMZmjF8gTnXNTDZ4C1W8MSWfh5NLIdzquujQCBJkg4dcp7m8jklKkow
280-hjaxY0CpFaF4z2L8mjcJq5Pl+J/lF4iKrJc7tJYWCueGeJiYgQci0vKUiRHqyr1wkjMU/5
280-6ZtjmmdAbSeJhM4nrLzQe5wXR6n8fMDsHtJvZNb1PZSMycs7rMoNDEY6pjjo8Y70k0E3jL
280-7AeM7Udi8oGAwooUGwENp3upYuhxd/bzoZg53TdQbNM2RKcGKozSQK2gHKpFI59gjwcZBR
280-IHdpdGggY2F2ZXMfFXppcGNvZGUgd2l0aG91dCBjYXZlcyADNDExfjd0pQ0k4DXBXvBfUn
```

```
280-8Q1E3j7EqVW1xC/TF2KGmfylpmL2iueTyPz50kAY9Qd/EWxhZGFyQGxhdmFiaXQuY29tgQ
280-8Vbjj47aXZzQUM9L79WuqTuLdMC2yD4vW76cGkb8hrGL/y8H0IshRpNeOAM
280------END USER SIGNET-----
280 OK {freeform}
```

RESPONSE CODES

Code	Description
214	HELP
221	BYE
250	OK
254	ACCEPTED=*identifier*
255	DUPLICATE=*identifier*
270	OK
271	ORGANIZATIONAL SIGNET CURRENT
280	OK
281	USER SIGNET CURRENT
290	OK
291	USER SIGNET CURRENT
354	READY TO RECEIVE MESSAGE
421	CONNECTION REQUIRES ABNORMAL TERMINATION
431	DESTINATION LIMITS EXCEEDED
450	ACCESS DENIED
451	DATA CORRUPTED
470	ORIGIN SIGNET UNAVAILABLE
486	IDENTIFIER=*identifier*
500	COMMAND SYNTAX ERROR
501	ARGUMENT SYNTAX ERROR
502	COMMAND DISABLED
503	INVALID COMMAND SEQUENCE
510	INVALID RECIPIENT
570	ORIGIN SIGNET UNAVAILABLE
575	INVALID ORIGIN SIGNET
576	INVALID DESTINATION SIGNET
578	INVALID ORIGIN SIGNATURE
586	INVALID RECIPIENT SIGNET

PROTOCOL EXTENSIONS

SIZE

TBD

BINARY

TBD

UNICODE

TBD

PIPELINING

TBD

SURROGATE

Indicates that the true destination host indicated by the TLS SNI extension, or as an argument to the STARTTLS command, could not be reached. However, the current host will act as a surrogate to accept and relay the D/MIME message onto its destination when the host becomes available. This extension allows individuals to host a DIME server at home, without revealing the destination host address literal to a consumer, and allows consumers to access DMTP services for the target domain even when the destination host is offline. Surrogates, depending on the configured policies for the domain, will be capable of handling signet retrieval requests and accepting encrypted messages for delivery when the true destination host comes back online.

PART 8: DARK MAIL ACCESS PROTOCOL (DMAP)

The Dark Mail Access Protocol (DMAP) specification will not be released with this draft, but will be added to this document in the future. DMAP will be an authenticated protocol and is intended for use between end user MUA and organizational servers. A few of the key elements currently being planned for DMAP are:

- The authentication mechanism will be rely on cryptography, allowing a user to prove they know the account password, without the server ever receiving it. Sometimes called ZKPP. This information will also be used to derive the keys needed to decrypt account data. Details still under development.
- DMAP will handle the submission of messages. The org signature will need to be appended before the message is relayed to another domain, although this doesn't need to be handled by the DMAP implementation.
- DMAP server will accept user signet signing requests and, if once approved/signed, will notify the user's client that a new signet has been published.

Some of the elements still in development, which may be removed:

- Facilitate the synchronization of a user's encrypted key ring. The key ring is responsible for storing private key information.
- Store encrypted copies of user signet rings. The signet ring is used to store management records, organizational signets, and user signets.
- Allow authenticated users behind a firewall to proxy their signet lookups, at the expense of privacy.
- Retrieve encrypted log entries. Log entries may contain security alerts[19], outbound delivery reports[20], service provider notifications, and system broadcasts.

Some of the elements being removed, which are currently available with IMAP:

- The protocol will not include server-side search because all email is encrypted on the server.
- The "fetch" command, or its equivalent will be dramatically simplified, selectors will be limited to retrieving specific chunks, or entire messages.[21]

19 Such as information about failed login attempts, or the IP addresses recently used to access the account.

20 A simple record which provides the outcome of a message submitted for relay to another domain. A simple success, error X, or pending indication. If this information is stored in a user's encrypted log queue, then a service provider won't need to keep the information stored in plain text log files, just so they can handle technical support requests. Of course most of them probably will either way.

PART 9: GLOBAL LEDGER

Currently in the design phase, the future implementation of the global ledger will enhance or replace the need for DNSSEC acceptance and deployment. Since the adoption of DNSSEC may continue to be slow, the introduction of the DIME global ledger will provide a non-reputable external record of user signet publications that a client can consult independently of a provider and thus detect when their provider might be complicit in an attack on their account. External sources also provide non-reputable evidence of a possible service provider MitM attack.

A few of the key elements currently being planned for the Global Ledger are:

- A defined set of trusted global ledger hosts deployed around the world with operational oversight from the Dark Mail Alliance (DMA).
- Allow for DMTP lookups of user and organizational signets providing a non-reputable record of signets.
- Provide key management that is redundant across sources to aid in the detection of compromised servers.
- Stand as the de facto source for public signets for all DIME implementations.

---

21 A handful of IMAP clients use the "chunking" feature of IMAP, allowing them to download a result in pieces. Should DMAP allow clients to pull D/MIME chunks in pieces? Consider the maximum possible chunk size is 16 MB – any MIME body parts larger than 16 MB will be split across chunks. The encryption scheme and chunk layout dictate that an entire chunk be downloaded before decryption. A complete chunk will also be needed to verify the signature taken over the cleartext.

PART 10: DARK MAIL ALLIANCE

The purpose of the Dark Mail Alliance (DMA) is to bring the world an ubiquitous end-to-end encrypted email standard. The DMA will be responsible for evangelizing the DIME amongst implementers and providers, and manage/coordinate the deployment of infrastructure projects like the global ledger, which are provided by independent DMA member organizations (or individuals)

The DMA partners will work to bring other members into the alliance and assist them in implementing the DIME standards. The DMA will also hold the rights to any DIME intellectual property in trust, such as the DIME trademarks. The DMA will also work to develop and maintain an open source reference implementation of the DIME standards to address privacy concerns regarding back doors.

PART 11: THREATS

A user's concern for private email exchanges can involve protection of basic content or extend to their social network information – who they exchange mail with, and when – and can vary by the amount of trust they place in their email service provider. Dark Internet Mail Environment (DIME) builds upon classic Internet Mail [IMA] and provides strong privacy protection using encryption, covering metadata, overall message structure, and individual message content including attachments. DIME also ensures message authenticity, integrity and verifiable non-repudiation.

Privacy exposure can be due to passive or active third-party impostors, with wiretapping that captures message traffic over the wire, compromise of a mail handling host or a key management host, or collaboration by a host operator. DIME's design provides a range of protections that combine to defend against each of these categories of threats.

Key management by end users, and even system operators, is a major barrier to the use of security-related services. Therefore, to the extent possible, DIME's encryption details are designed to operate automatically. Great care has been taken to make it difficult for an attacker to subvert the automated aspects of the system undetected. Because error messages and security warnings can be confusing to users, the system provides for alternate mechanisms so clients can overcome common anomalies without compromising security or requiring user intervention. The goal is to create a system sufficiently resilient so that the occurrence of a non-recoverable security error is most likely to be due to system compromise, or because someone in a privileged network position is attempting to carry out an attack.

Service providers occupy a trusted position in the DIME ecosystem. However, a client can choose among three service trust levels to considerably narrow this dependence. In particular, it determines a server's access to the user's private keys using account modes (Trustful, Cautious and Paranoid).

This document discusses the privacy goals for the DIME protocols and formats, how those goals are achieved and what assumptions are made [SPARROW]. A core goal is attending to different types of users and their trust of an associated organization server. We highlight assumptions, and detail specific aspects of the design intended to mitigate common attack vectors. Unless otherwise noted, this document assumes the "Cautious" account mode is being utilized.

THREATS

VENUES

The primary concern is unauthorized information disclosure, that is, situations where the user loses control over the release of their private information. Different types of information need different types of protection. A related concern is authentication of the participants in an exchange, both end users and service providers, so that fraudulent content is avoided.

The types of information compromise of concern include:

Author spoofing: Whether the purported creator and submitter of a message is the actual agent of action.

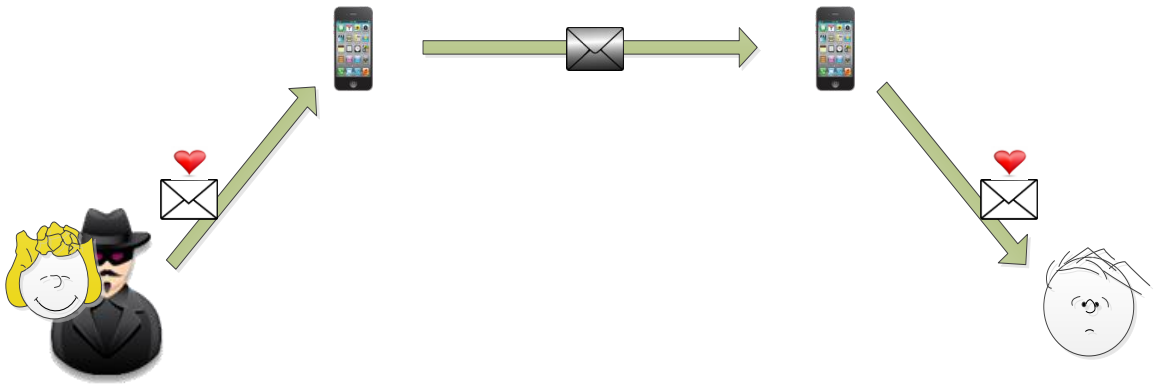


Figure 10 – Author Spoofing

Service provider spoofing: Knowing that the intended provider (mail, key, DNS) is the actual provider.

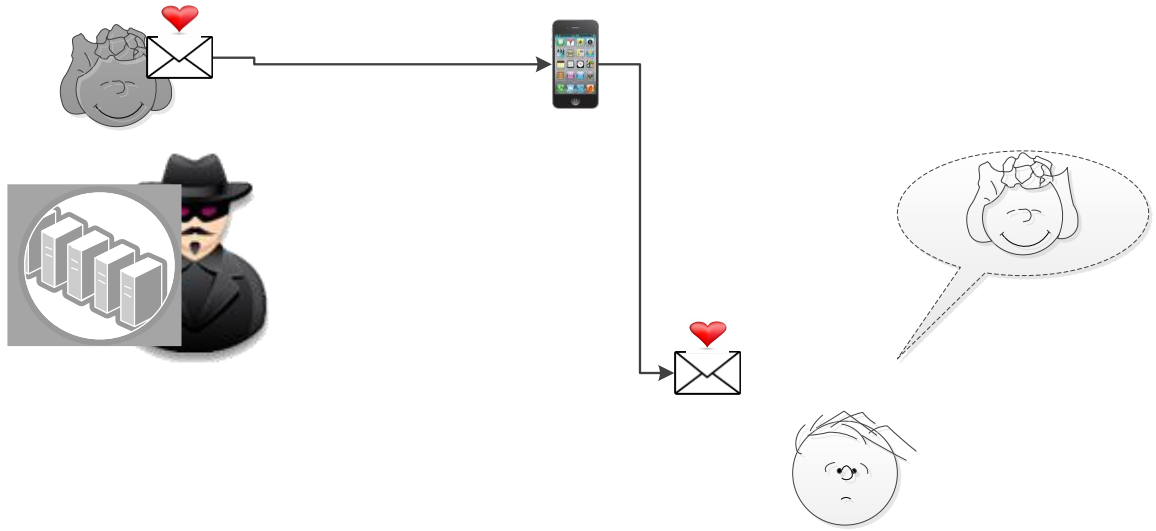


Figure 11 – Service Provider Spoofing

Message content disclosure: Limiting disclosure only to authorized parties or recipients.

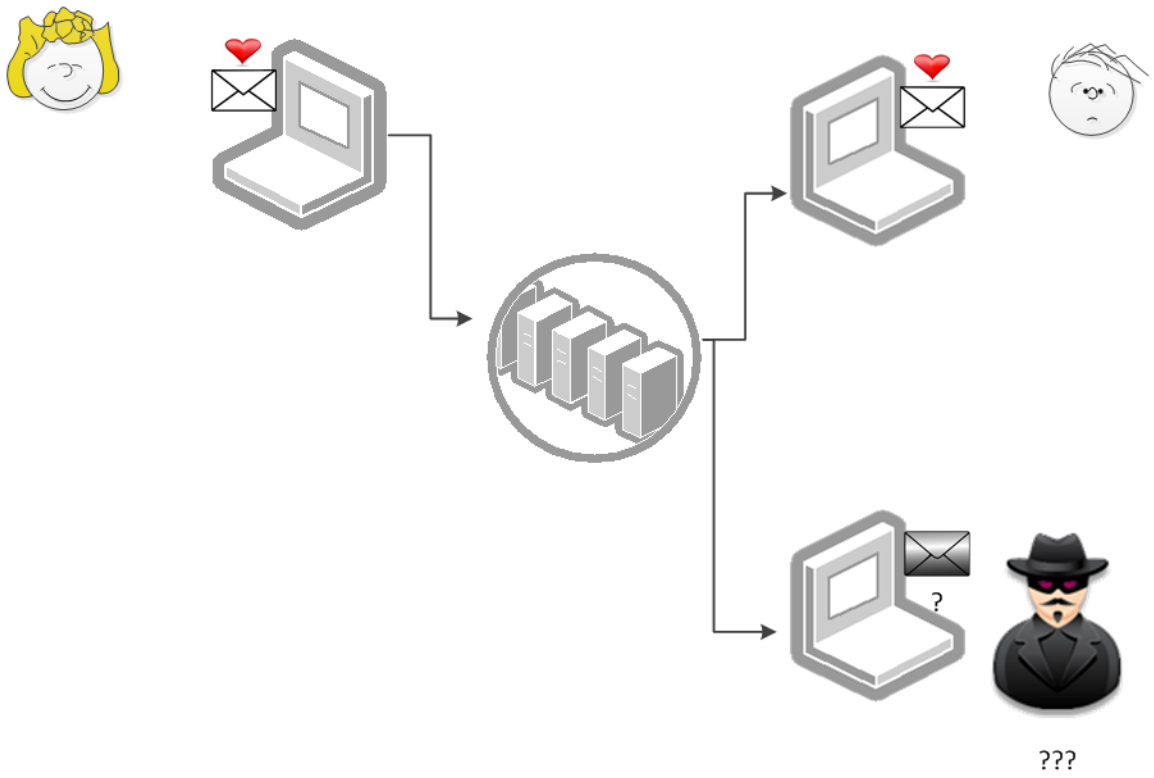


Figure 12 – Message Content Disclosure

Message structure disclosure: Even without knowing the detailed content, knowing about message size, attachment structure, and attachment data types can help an attacker.

Metadata disclosure: Any other structured data, involving participant and message attributes, which can be stored and subjected to social and network traffic analyses. This includes relationships and activity. Who is talking with whom; when and how actively?

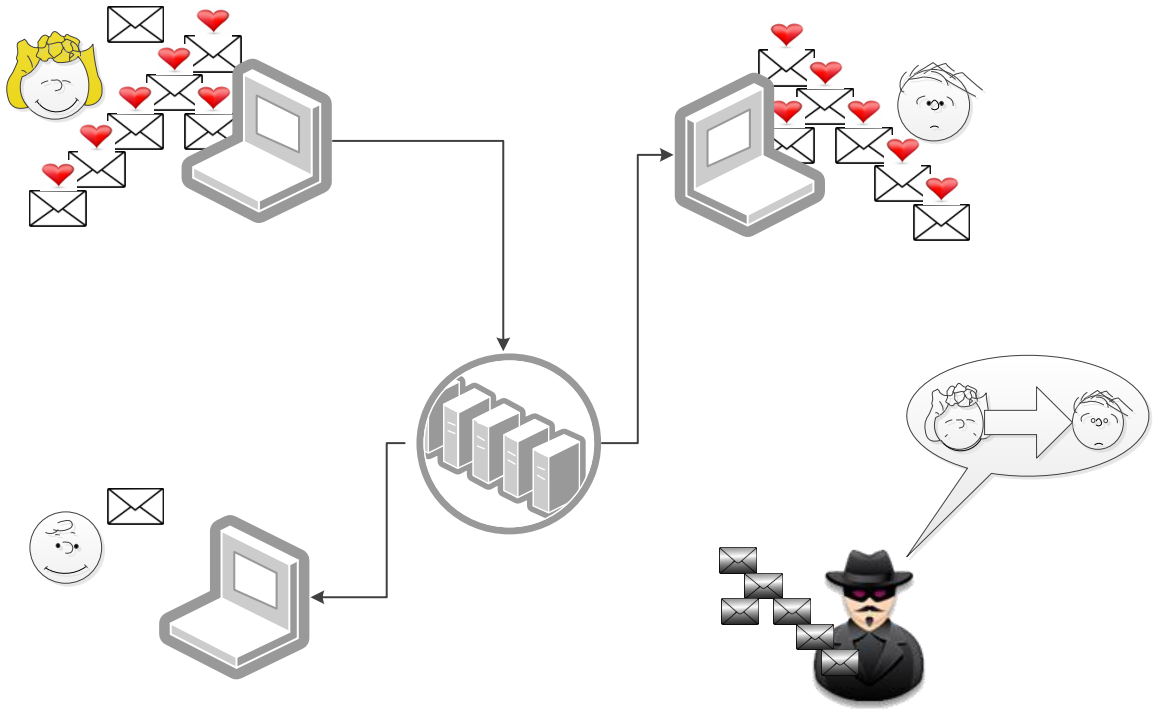


Figure 13 – Metadata Disclosure

VECTORS

A variety of avenues can be exploited to achieve unauthorized disclosures:

Password: The basic unit of local authentication within a system.

"[A challenge is] how to authenticate securely with the service provider without revealing the password (since the password is probably also used to encrypt the private key and other secure storage, so it is important that the service provider does not have cleartext access as with typical password authentication schemes)." [SPARROW]

Key:	"[P]ublic-key encryption to allow[s] a user to send a confidential message to the intendant recipient, and for the recipient to verify the authorship of the message. Unfortunately, public-key encryption is notoriously difficult to use properly, even for advanced users. The very concepts are confusing for most users: public key versus private key, key signing, key revocation, signing keys versus encryption keys, bit length, and so on. This is where we are now: we have public key technology that is excessively difficult for the common user, and our only methods of key validation have fallen into disrepute." [SPARROW]
Organizational Signet:	Information tied to a specific domain name, including the public keys associated with that domain name. The authoritative source and verification information for an organizational signet is advertised using a DIME management record in the DNS system and is considered authentic when retrieved from an authoritative DMTP server and validated by the DIME management record. No further validation steps are necessary if the management record was signed using DNSSEC. The organizational signet may also carry with it policy information for the domain. Compromising the private keys associated with an organizational signet or replacing an organizational signet with a fraudulent one could allow an attacker to generate fake user signets and spoof the organization identity.
User Signet:	Information included with a person's public key that helps others verify that a key is genuine or valid; it can carry related profile information for the entity being identified. Assessing signet validity is a distinct step. Compromising the user signet resolution process could allow an attacker to advertise fraudulent public keys allowing them to spoof an identity or access encrypted message contents only if the victim later uses the spoofed ID. A user signet is considered authentic when a retrieved from an authoritative DMTP host and the signature is authenticated using the keys contained within organizational signet.
Domain name:	Domain names are basic unit of global identification on the Internet. Domain names are associated with records of information through public queries of the Domain Name Service. Trusting DNS servers, or at least DNS records, is the foundation for email service. The primary long-term path for improving that trust is the widespread adoption of DNSSEC.
Transmission Channel:	Monitoring traffic across a transmission link (wiretapping) can be simply passive copying or it can be active spoofing via a man in the middle attack (MitM) that relays messages between both ends, making them believe that they are talking directly to each other over a private connection. TLS is the primary means of protection against wiretapping; MitM protection requires that the server's X.509 certificate is validated using a CA from a certificate authority, or using a management record signed using

DNSSEC, authenticating the server's affiliation with the owner of the target domain name.

Client: A compromised end-point permits the attacker to impersonate the user or, at least, to see all of the user's data. It could also permit the attacker to steal keys and passwords, obtain cleartext message information, or otherwise weaken on-going services to facilitate later interceptions by introducing malware. Additionally, a poorly implemented client or MUA could break the cryptographic mechanisms employed by DIME.

Mail Server: A compromised mail server (MSA, MTA, MDA, MS) can access any mail information that is in the clear or that the server is able to decrypt. Depending upon the capabilities of the client and the account mode, the amount of trust a user must place in their mail server can be greatly reduced.

Key Server: Compromising a server that holds private encryption keys permits an attacker to decrypt data and thereby break DIME's protection. Redundant sources for signet information can aid in the detection of compromised key servers attempting MitM attacks.

DNS Server: A compromised server can permit creation of false records under a target domain name. DNSSEC authenticates records, independent of the server providing them.

Gateway: Transition between a protected email environment, such as DIME, and an unprotected one, such as naked Internet mail, usually requires operation of service gateways. They create opportunities for spoofing and downgrade attacks.

Persistence: Advanced Persistent Threats (APT) typically entails an attacker with a privileged network position, ability to perform extensive and long-term data collection and apply massive computational resources. This creates its own line of attack, beyond those vectors normally of concern.

MITIGATION STRATEGIES

DIME minimizes information that is exposed to intermediaries along the mail-handling path, including what is available to the initial origin and destination service providers. Content is protected by multiple layers of encryption reducing reliance on single-points of failure for providers of keys and signets.

MESSAGE PROTECTION

A message is a hierarchical object, comprising several distinct handling-related and payload components. This permits efficient handling of distinct portions over limited channels and by clients with limited capabilities, as well as permitting separable protection. Only a thin "outer" component of the message transits with unencrypted information.

In terms of handling and protection, each copy of a message is between the author and one recipient. The basic message handling model has twolevels, with an *organization* component and a *user* component. The organization provides public-facing services, at the granularity of a domain name. An individual user's involvement with a message, such as their full email address, is visible only to their associated organization server and the other end user associated with this message.

The basic message protection model encrypts the entire message, as well as each component, using a different key for each portion that is encrypted. This permits independent handling of different message components and protects envelope information by encrypting those portions with different user and organizational keys.

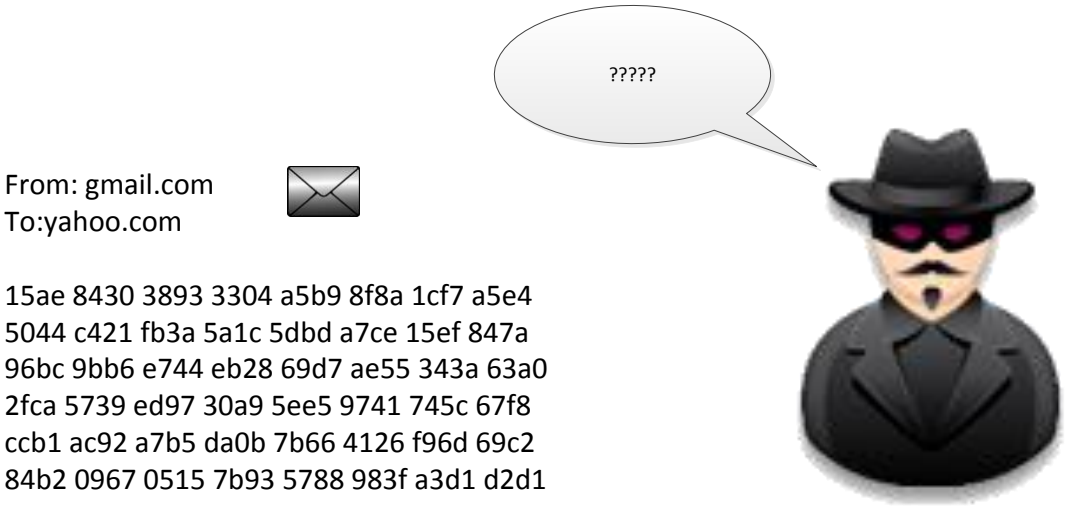


Figure 14 – Basic Message Protection

ACCOUNT MODES

A user's reliance on an associated organization server can be at three different service trust levels, selectable by the user:

Trustful: Comparable to the level of trust placed in a service provider for typical email services historically. In effect, the service handles all privacy issues on behalf of the user. DIME provides protection for messages in transit over the Internet, but the end-user's service provider is fully trusted. In particular, the server has direct access to the user's private keys. Users access email using traditional access via

SMTP and IMAP over SSL. Although it is implementation specific, it is recommended that the user's private key be protected using the user's password.

Cautious: In this mode the server holds encrypted copies of a user's private keys and messages. This is convenient for multi-platform users, while reducing the amount of information a compromised service provider can disclose. Because the service provider never has access to the decrypted private key, they are unable to access a user's messages, or publish new user signet without triggering a break in the chain of custody. This mode is designed to facilitate the adoption of DIME without requiring end users to modify their behavior to obtain the additional benefits of encryption without the traditional encryption costs.

Paranoid: This mode provides the server with almost no user security information. In particular, the server never has access to the user's private keys, even in encrypted form.

A thin client is more dependent upon the service provider, since it has few, or none of its own, independent capabilities. Webmail is typically an example of complete reliance on the provider, since any software running on the client comes from the provider; however, a proper thin client implementation that performs encryption in the user's browser will not have complete access to all user information. In the event a thin client is exploited by an attacker to contain malicious code, it could circumvent security to gain access to user information. The recommended approach is a thick client independently obtained and installed and fully under the control of the user.

ATTACK VECTOR MITIGATION

The following discussion explores the likely approaches for preventing or detecting problems in each part of the system subject to attack.

PASSWORD

User access to a server is controlled through an account password. It is used to authenticate with a server; *however is never sent to the server.* Rather the password is used to derive information that is sent. The server only stores a pre-nonced hash and account key pair, with the private account key being encrypted by the password on the user's device. Hence, if the server is compromised it cannot reveal the password, or even provide the required elements to successfully spoof authentication. The amount of entropy associated with a user's password is improved with user specific salts, and the number of hash rounds being varied based on plaintext length. Organizations can further improve passwords by imposing a variable number of additional hash rounds.

SIGNET

Signet Assignment: Signets are associated with an organizational domain or a user address based on the semantic context of a signet resolver query.

NETWORK PACKET CAPTURE

FORWARD SECRECY

"Traditional schemes for forward secrecy are incompatible with the asynchronous nature of email communication, since with email you still need to be able to send someone a message even if they are not online and ephemeral key generation requires a back and forth exchange between both parties.

"...Another possible approach is to use traditional encryption with no support for forward secrecy but instead rely on a scheme for automatic key discovery and validation in order to frequently rotate keys. This way, a user could throw away their private key every few days, achieving a very crude form of forward secrecy." [SPARROW]

Network level packet captures are useless with DIME because all connections are protected using TLS v1.2 and require the use of a cipher suite which provides for perfect forward secrecy (PFS). If an organization can protect their TLS private key, then they can ensure attackers are also unable to MitM the organization's TLS connections and can achieve PFS at a wire level.

PFS for message objects, as the description above suggests, is far more difficult, and contrary to the nature of email. However, a DIME user using the "paranoid" account mode could still obtain PFS for messages by routinely rotating their signet, and destroying the private keys associated with their former signet once the expiry threshold has been reached. Because the private keys were never synchronized with the server, the user can be assured that deletion means the keys could never be recovered, thereby providing PFS even if the messages were intercepted and recorded by a server.

SIGNET AND KEY MANAGEMENT

BASIC MANAGEMENT AND OPERATION

No single source of key information is automatically accepted by the entity making the query. It always must have a confirmation.

Key creation:

1. Trustful Mode: The user signet and the corresponding private keys are generated on the server. The server appends the organization signature plus optional attributes such as name, address, telephone, etc. and a second organizational signature. The two organization signatures allow the cryptographic portion of the user signet to be split from the optional attribute portion. The server stores this signet internally and makes it available via DMTP.
2. Cautious Mode: The desktop client generates a Signet Signing Request (SSR) and the corresponding private keys and submits to server over DMAP with the private keys encrypted. The server appends the organization signature plus optional attributes such as name, address, telephone, etc. and a second organizational signature. The server stores this signet internally and makes this available via DMTP. The encrypted private keys are available to the desktop client via DMAP.
3. Paranoid Mode: The desktop client generates the SSR and submits the SSR to the

server over DMAP. The server appends attributes (such as names, address, telephone, etc.) and organization signature. The server stores this signet internally and makes this available via DMTP. The encrypted private key is stored on the desktop client and never transferred to the server.

Signet discovery: The desktop client performs a lookup of the management record for a domain using DNSSEC. If there is a DIME management record (MR), it retrieves the Primary Organization Key (POK) from the MR and the organizational signet via a DMTP connection. The organizational signet is validated against the POK retrieved via DNS. If a DIME MR is not signed using DNSSEC, the DMTP server must use a TLS certificate validated by a recognized certificate authority (CA).
The DMTP server will respond to queries for user and organizational signets. Note that some clients might not be able to make direct TCP/TLS connections to a DMTP server because of firewall rules; they will need to proxy requests through their local DIME key server, presumably over an authenticated DMAP connection. This could create an additional avenue for metadata to leak, such as what signet a user retrieves.

Signet validation: A signet is validated by a confirming query via DNS, in addition to the primary means of obtaining and validating it. For an organization-level signet, the secondary query can be via a pre-authenticated source (recognized CA) or DNSSEC. For a user-level signet, confirmation is through a chain of custody if the signet is already in the user's signet cache, in addition to confirmation of an organization signature.

Signet availability: Organization and user signet availability will vary based on deployment decisions and user configuration options. From an organizational viewpoint, access to the organization's private key will be required for signing operations and decryption of delivery information. This will require every DMTP server to have access to the private key, or for more sophisticated deployments, access to a centralized key server that performs all of the organizational level cryptographic operations. Note the trustful/cautious/paranoid modes for end-users; they can choose to share the unencrypted private keys with the server, just the encrypted private keys, or nothing at all. Which option they choose will determine how they can access their account, and where user level cryptographic operations occur.

Signet revocation: To revoke a potentially compromised user signet, a user simply needs to publish a replacement public signet and wait the specified time-to-live for the compromised signet to expire. Once the TTL expires, servers will have to query for the signet again. When an organizational signet is compromised, all existing user signets must be resigned and republished. Because of the potential overhead for large organizations, this issue further stresses the requirement that each organization must protect their corresponding organizational private keys at all costs. If an organizational signet is NOT

compromised, but simply changed, the previous organizational keys can be added to the new organizational signet as secondary keys[22].

Key rollover: A chain of custody is established for a sequence of signets. As a new signet is introduced, it is signed by its predecessor signets. This permits automatic acceptance of a new signet when the previous one is already in a user's signet ring. It is based on the reasonable assumption that the owner of the new signet had access to the private key associated with the trusted signet.

ORGANIZATIONAL_SIGNET

An organizational signet is generated by a system administrator who installs the key into a DMTP configuration and associates the signet with a domain. The administrator configures the DNS for the domain in question to provide the associated validation record. Because of the manual process associated with publishing new organizational signets, the assumption is they will change infrequently. While user signets will have TTL values specified in minutes, organizational signets would use TTL values measured in days; the recommendation is organizations will change signets every 1 to 3 years and have high TTLs (16-32-64 days).

An organizational signet, and its associated private keys, is used to:

- Sign user signets
- Sign outbound messages
- Decrypt 'recipient' chunk on received messages
- Decrypt 'author' chunk for outbound messages before signing
- Decrypt 'author' chunk for bounce message
- Validate signatures before accepting bounces

USER SIGNET

A user signet is generated automatically by a user's client submitted using DMAP. The public signet is published on an authoritative DMTP server. Whether or not the user's private keys are shared with their organization's server depends on the account mode (trustful, cautious, and paranoid). In trustful mode, each device the user has can get access to keys through the organization's DMAP server. In paranoid mode, the user must use an independent mechanism when using multiple devices for synchronizing keys.

To minimize the amount of data exposed by a compromised private key, users are encouraged to have their signets rotated automatically. The time period recommended will likely vary by user, but could range from a handful of days

22 In this context, this can be considered an estoppel (i.e. a revocation).

to a period of weeks. Users who suspect their private keys have been compromised can trigger a manual signet rotation ahead of the scheduled rotation.

To provide a robust validation model, a potential sender has multiple avenues for confirming that a specific public key belongs to a user address. The primary basis is that a public key was retrieved from an organization's authoritative key store, and contains an organization signature that can be traced to a verifiable and trusted organizational signet. This constitutes basic authenticity and typically means the key can be trusted unless: the lookup request(s) was subverted or the organization is complicit in an attack (assuming the organization's key has not also been compromised). Additional verification paths are designed to allow detection of such attacks.

A verifiable chain of custody can illustrate that the owner of an address may have changed recently; this can be used by people with a previous trusted signet in their local cache. Finally, the use of the optional global ledger can provide a non-reputable external record of user signet publications that a client can consult independently of a provider and thus detect when their provider might be complicit in an attack on their account. External sources also provide non-reputable evidence of a possible MitM attack by a user's organization or service provider.

A user signet, and its associated private keys, is used to:

- Decrypt inbound messages.
- Sign outbound messages.
- Sign new public signets before submitting them to the organization's server for publication.

DOMAIN NAME

The DNS system controls whether a domain supports DIME and provides the trusted anchor for organizational and user signets. In effect, compromising the DNS records would permit an attacker to gain authoritative control over a domain's identity. The primary long-term path for ensuring the validity DNS information and responses is DNSSEC.

TRANSMISSION CHANNEL

TLS is the primary means of protecting against wiretapping and the tampering of data in transit. For TLS to provide MitM protection a server certificate must be validated with an X.509 certificate signed by a certificate authority or against a TLS field provided by an MR signed using DNSSEC.

CLIENT

Client implementations will perform the user level cryptographic operations. Like email today, we anticipate a large variety of DIME client implementations will be created. They will likely range from thick applications that run on desktop and mobile devices, to thin clients written in JavaScript that are loaded from a web server at runtime. Because the user level cryptographic functions are performed by the client for cautious and paranoid users, it is important that these client implementations properly implement the cryptographic primitives and conform to the user interface and implementation standards supplied. These standards will ensure client implementations follow a baseline for the secure handling of sensitive information like passwords and private keys. Clients will also be

responsible for communicating to users which inbound and outbound messages are protected by encryption because they involve DIME-enabled domains, versus those that were sent naked using traditional mail protocols.

MAIL SERVER: MSA, MTA, MDA

Email content and data structure are protected by a proper DIME implementation; however, it is still the responsibility of the mail server organization to follow security best practices and secure the mail server.

KEY SERVER

Key management that provides redundant sources can aid in detection of compromised servers. Sources can be authoritative servers or be replicated through syndication to a partner domain's servers or in the future to the global ledger.

DNS SERVER

Distinct from using DNSSEC to authenticate DNS content, the responsibility for securing a domain's DNS servers remains with the organization.

Primary protection is accomplished by DNSSEC. However, if DNSSEC name validation cannot be used, it is still possible to reach a trusted state by publishing a DNS record AND using a TLS certificate that has been signed by a trusted Certificate Authority.

GATEWAYS

SMTP gateways provide the ability for DIME users to exchange messages with users at domains that do not support DIME. These gateways accept incoming SMTP messages from non-DIME domains and encrypt them using a user's current key before storing it on the server. Likewise, outbound messages can be relayed through a gateway to an SMTP host. It should be possible to translate, without any information loss, between the SMTP MIME format and the D/MIME message format. It is worth noting that organizations can choose to disable SMTP access at a domain level, or allow users to disable SMTP access at a user level. It is also important to understand that because SMTP messages may be transmitted in the clear in a worst-case scenario, and rely on TLS for protection in a best-case scenario, that users understand when they are sending out messages to a DIME-enabled domain versus when they send naked messages via traditional email.

PERSISTENCE

For network level protection, DIME relies on TLS cipher suites that provide perfect forward secrecy. For message level protection, we assume that most users will want to retain persistent access to their historical message corpus. This implies retaining private keys to facilitate the future decryption of messages or alternatively, clients storing messages in their decrypted form locally before deleting a given private key.

HUMAN FACTORS

System security is often compromised through social engineering and other challenges with user and operator behavior. Simply implementing DIME does not replace good user education and competent operational security. Bad passwords, poor protection of private keys, and situational factors (such as leaving a laptop, no matter how short the length of time, unattended at the airport) cannot be mitigated by DIME. Depending on the implementation, examples of efforts to mitigate human factors include tailoring the user's interface, such as flagging information that is to be more or less trusted, and compensatory computation, as might be used to counteract a shorter password.

PART 12: ATTACKS AND MITIGATIONS

TBD

THIS PAGE INTENTIONALLY LEFT BLANK

PART 14: CREDITS

AUTHOR

LADAR LEVISON

Ladar Levison is the Founder of Lavabit, LLC, which served as a place for free and private email accounts. By August of 2013, Lavabit had grown to over 410,000 users. Levison created Lavabit because he believes that privacy is a fundamental, necessary right for a functioning, free and fair democratic society. On August 8, 2013, he made the bold decision to shut down his business after refusing to become "complicit in crimes against the American people." Presently, Levison is serving as the lead architect for the Dark Internet Mail Environment. Levison continues to vigorously advocate for free speech and the right to privacy, speaking at conferences, and collaborating on projects which are working to give back control of the Internet to the people.

CONTRIBUTORS

DAVE CROCKER

David H. Crocker is a principal with Brandenburg InternetWorking. He designs network-based applications businesses and distributed system architectures. His focus is on the creation of Internet-based businesses built on a solid foundation of customer benefit and revenue potential.

Dave worked in the ARPA and NSF CSNet network research community during the 1970s and early 1980s, and led product development efforts at MCI and various Silicon Valley companies, into the 1990s. He then founded several startup companies, serving as CEO for one. Dave has developed and operated two national email services, designed two others, and was CEO of a community non-profit ISP. His senior management product efforts cover email clients and servers, core protocol stacks for TCP/IP and OSI, network management control stations, and knowledge management tools for product support. For his work on email, Dave was a co-recipient of the 2004 IEEE Internet Award.

Dave has been leading and authoring Internet standards for forty years, covering Internet mail, instant messaging, security, facsimile and EDI. He has also contributed to work on Internet commerce, domain name service, emergency services, and TCP/IP enhancements. He has authored more than 50 IETF Requests for Comments. Dave served as an Area Director for the Internet Engineering Task Force (IETF) variously overseeing network management, middleware and the IETF standards process. He has also been a member of the IETF's administrative and legal oversight bodies (IAOC/Trust).

UNNAMED CONTRIBUTORS

The DIME team would like to thank the gracious help of numerous, yet unnamed, contributors without whose dedication and time this publication would not be possible.

ATTRIBUTION

The document author's borrowed heavily from referenced RFCs and other sources for several sections. The team provides full attribution to the extent possible; however, if a reader notices an unintentional missing attribution, please notify the author for correction. The DIME team owes a debt of gratitude to the hard work of the many Internet revolutionaries that got us to this point.

PART 15: REFERENCES

[AES] National Institute of Standards and Technology, "Specification for the Advanced Encryption Standard (AES)", FIPS 197, November 2001.

[ASCII] Cerf, V., , RFC 20, October 1969.

American National Standards Institute (formerly United States of America Standards Institute), “USA Code for Information Interchange”, ANSI X3.4-1968, 1968.

[AVIAN] Waitzman, D., , RFC 1149, April 1990.

[DANE] Hoffman, P., Schlyter, J., , RFC 6698, August 2012.

[DANGER] Bernstein, D., Tanja, L., , September 2013.

[DEFLATE] Deutsch, P., , RFC 1951, May 1996.

[DKIM] Allman, E., et al., , RFC 4871, May 2007.

[DOMAIN] Mockapetris, P., , STD 13, RFC 1034, November 1987.

Mockapetris, P., , STD 13, RFC 1035, November 1987.

[DNSSEC] Arends, R., Austein, R., Larson, M., Massey, D., Rose, S., , RFC 4033, March 2005.

[E123] International Telecommunications Union, , February 2001.

[ECDH] Blake-Wilson, S., et al., , RFC 4492, May 2006.

[EdDSA] Bernstein, D., , September 2011.

[GCM] Dworkin, M., , SP 800-38D, November 2007.

[IMA] Crocker, D., , RFC 5598, July 2009.

[IMF] Resnick, P., , RFC 5322, October 2008.

[IP] Postel, J., , RFC 791, September 1981.

[ISO639-1] International Organization for Standardization, , ISO 639-1:2002, July 2002.

[ISO639-2] International Organization for Standardization, , ISO 639-2:1998, October 1998.

[ISO3166-1] International Organization for Standardization, , ISO 3166-1:2013, November 2013.

[ISO4217] International Organization for Standardization, , ISO 4217:2008, October 2008.

[ISO15924] International Organization for Standardization, , ISO 15924:2004, January 2004.

[LOC-LANG] The Library of Congress, , March 2015.

[KEYWORD] Bradner, S., , RFC 2119, March 1997.

[LANGUAGE] Phillips, A. and M. Davis, , BCP 47, RFC 5646, September 2009.

[IANA-LANG] Internet Assigned Numbers Authority, , March 2015.

[MIME] Freed, N., Borenstein, N., , RFC 2045, November 1996.

Freed, N., Borenstein, N., , RFC 2046, November 1996.

Moore, K., , RFC 2047, November 1996.

Freed, N., Klensin, J., , RFC 4289, December 2005.

Freed, N., Borenstein, N., , RFC 2049, November 1996.

[OCSP] Myers, M. et al., , RFC 2560, June 1999.

[PEM] Linn, J., , RFC 1421, February 1993.

[PGP] Callas, J. et al., , RFC 4880, November 2007.

[PGP-ECC] Jivsov, A., , RFC 6637, June 2012.

[PGP-EdDSA] Koch, W., , March 2014.

[PNG] Boutell, T., , RFC 2083, March 1997.

Portable Network Graphics, , ISO/IEC, 15948.

[SEC] , September 2000.

[SHS] National Institute of Standards and Technology, , FIPS 180-2, August 2002.

[SMIME] Ramsdell, B., Turner, S., , RFC 5751, January 2010.

[SMTP] Klensin, J., , RFC 5321, October 2008.

[SNV-CURRENCY] Swiss Association for Standardization, , January 2015.

[SPARROW] Sparrow, E., .

[SRV] Gulbrandsen, A., Vixie, P., Esibov, L., , RFC 2782, February 2000.

[TCP] Postel, J., , RFC 793, September 1981.

[TLS] Dierks, T., Rescorla, E., , RFC 5246, October 2008.

[TLS-ECDHE] Rescorla, E., , RFC 5289, August 2008.

[TLS-SNI] Blake-Wilson, S. et al., , RFC 3546, June 2003.

[TXT] Rosenbaum, R., , RFC 1464, May 1993.

[XMPP] Saint-Andre, P., , RFC 6120, March 2011.

Saint-Andre, P., , RFC 6121, March 2011.

[XMPP-CHAT] Saint-Andre, P., , February 2012.

[XMPP-OTR] Goldberg, I., Borisov, N., , September 2012.

APPENDIX A: DATA TYPE IDENTIFIERS

Magic Number	Label
1215	User Signet Signing Request
1776	Organizational Signet
1789	User Signet
1847	Encrypted Message
1851	Encrypted Sent Message
1861	Encrypted Draft Message
1908	Encrypted Naked Message
1947	Encrypted Organizational Key
1952	Organizational Key
1976	Encrypted User Key
2013	User Key

Symbol	Name	Website
BLK	Blackcoin	https://www.blackcoin.co/
BTC	Bitcoin	https://bitcoin.org/
DRK	Darkcoin	https://www.darkcoin.io/
LTC	Litecoin	https://litecoin.org/
PPC	Peercoin	http://www.peercoin.net/
STR	Stellar	https://www.stellar.org/
XMR	Monero	https://monero.cc/
XRP	Ripple	https://ripple.com/currency/

APPENDIX B: COMMON ENCODINGS

BASE64URL ENCODING

This document represents encodes binary data using the base64 encoding scheme defined in RFC 4648, with the URL and filename safe character set defined in Section 5, and known as base64url. In addition to the standard base64url conversion, all trailing pad characters, line breaks, white space, and other non-printable control characters should be removed, as permitted by Section 3.2. [BASE]

[BASE] Josefsson, S., The Base16, Base32, and Base64 Data Encodings, RFC 4648, October 2006.

NOTES ON IMPLEMENTING BASE64URL ENCODING WITHOUT PADDING

This section was adapted from draft-ietf-jose-json-web-signature, Appendix C.

[JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature (work in progress), March 2015.

This section describes how to implement the base64url encoding and decoding functions without padding based upon standard base64 encoding and decoding functions that do use padding. To be concrete, example code written in C# is provided showing how to convert between standard base64 into the base64url encoding. These functions should provide an adequate template for implementations in other languages.

To encode binary octets into a base64url string:

```
import operator, base64

def base64url_encode(binary):
    # Encodes a string using the standard base64 method, and
    # converts the output into the proper format.

    # Encode the binary input using standard base64 method.
    output = base64.b64encode(binary)

    # Swap '+' (plus) with '-' (minus).
    output = output.replace('+', '-')

    # Swap '/' (slash) with '_' (underscore).
    output = output.replace('/', '_')

    # Remove the padding '=' (equal).
    output = output.replace('=', '')

    # Remove line breaks and other whitespace.
    return output.join((output.split()))
```

To decode a base64url string back into an array of binary octets:

```
import operator, base64
```

```
def base64url_decode(string):
    # Converts the string into the standard base64 format, and
    # then uses the standard method to convert the string.

    # Swap '-' (minus) with '+' (plus).
    string = string.replace('-', '+');

    # Swap '_' (underscore) with '/' (slash).
    string = string.replace('_', '/');

    # Determine whether padding should append to the string.
    if operator.mod(len(string), 4) == 3:
        string = str.join(string, "=")

    elif operator.mod(len(string), 4) == 2:
        string = str.join(string, "==")

    # Finally, convert the string using a standard base64 decoder.
    return base64.b64encode(string)
```

As per the example code above, the number of '=' padding characters that needs to be added to the end of a base64url encoded string without padding to turn it into one with padding is a deterministic function of the length of the encoded string. Specifically, if the length mod 4 is 0, no padding is added; if the length mod 4 is 2, two '=' padding characters are added; if the length mod 4 is 3, one '=' padding character is added; if the length mod 4 is 1, the input is malformed.

The following octet sequence, expressed in hexadecimal form:

```
0x03ecffe0c1
```

Results in the following string after being converted into the base64url format:

```
A-z_4ME
```

MULTIPRECISION INTEGERS

Multiprecision integers (also called MPIs) are unsigned integers used to hold large integers such as the ones used in cryptographic calculations.

An MPI consists of two pieces: a two-octet scalar that is the length of the MPI in bits followed by a string of octets that contain the actual integer.

These octets form a big-endian number; a big-endian number can be made into an MPI by prefixing it with the appropriate length.

Examples (all numbers are in hexadecimal):

The string of octets [00 01 01] forms an MPI with the value 1. The string [00 09 01 FF] forms an MPI with the value of 511.

Additional rules:

The size of an MPI is ((MPI.length + 7) / 8) + 2 octets.

The length field of an MPI describes the length starting from its most significant non-zero bit. Thus, the MPI [00 02 01] is not formed correctly. It should be [00 01 01].

Unused bits of an MPI MUST be zero.

Also note that when an MPI is encrypted, the length refers to the plaintext MPI. It may be ill-formed in its ciphertext.

RADIX-64 CONVERSIONS

As stated in the introduction, OpenPGP's underlying native representation for objects is a stream of arbitrary octets, and some systems desire these objects to be immune to damage caused by character set translation, data conversions, etc.

In principle, any printable encoding scheme that met the requirements of the unsafe channel would suffice, since it would not change the underlying binary bit streams of the native OpenPGP data structures. The OpenPGP standard specifies one such printable encoding scheme to ensure interoperability.

OpenPGP's Radix-64 encoding is composed of two parts: a base64 encoding of the binary data and a checksum. The base64 encoding is identical to the MIME base64 content-transfer-encoding [RFC2045].

The checksum is a 24-bit Cyclic Redundancy Check (CRC) converted to four characters of radix-64 encoding by the same MIME base64 transformation, preceded by an equal sign (=). The CRC is computed by using the generator 0x864CFB and an initialization of 0xB704CE. The accumulation is done on the data before it is converted to radix-64, rather than on the converted data. A sample implementation of this algorithm is in the next section.

The checksum with its leading equal sign MAY appear on the first line after the base64 encoded data.

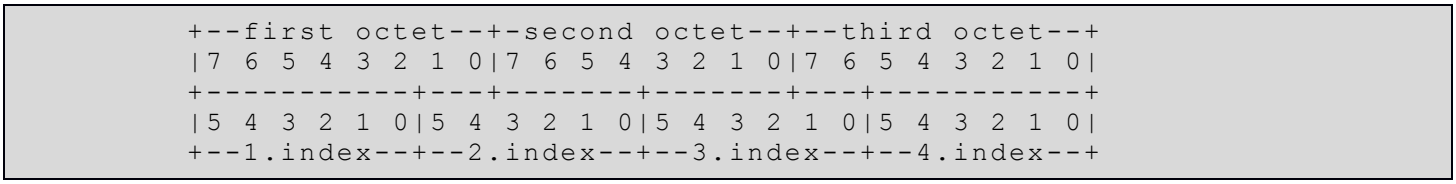
Rationale for CRC-24: The size of 24 bits fits evenly into printable base64. The nonzero initialization can detect more errors than a zero initialization.

An Implementation of the CRC-24 in "C"

```
#define CRC24_INIT 0xB704CEL
#define CRC24_POLY 0x1864CFBL
typedef long crc24;
crc24 crc_octets(unsigned char *octets, size_t len)
{
    crc24 crc = CRC24_INIT;
    int i;
    while (len--) {
        crc ^= (*octets++) << 16;
        for (i = 0; i < 8; i++) {
            crc <<= 1;
            if (crc & 0x1000000) crc ^= CRC24_POLY;
        }
    }
    return crc & 0xFFFFFFL;
}
```

ENCODING BINARY IN RADIX-64

The encoding process represents 24-bit groups of input bits as output strings of 4 encoded characters. Proceeding from left to right, a 24-bit input group is formed by concatenating three 8-bit input groups. These 24 bits are then treated as four concatenated 6-bit groups, each of which is translated into a single digit in the Radix-64 alphabet. When encoding a bit stream with the Radix-64 encoding, the bit stream must be presumed to be ordered with the most significant bit first. That is, the first bit in the stream will be the high-order bit in the first 8-bit octet, and the eighth bit will be the low-order bit in the first 8-bit octet, and so on.



Each 6-bit group is used as an index into an array of 64 printable characters from the table below. The character referenced by the index is placed in the output string.

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

The encoded output stream must be represented in lines of no more than 76 characters each.

Special processing is performed if fewer than 24 bits are available at the end of the data being encoded. There are three possibilities:

1. The last data group has 24 bits (3 octets). No special processing is needed.

2. The last data group has 16 bits (2 octets). The first two 6-bit groups are processed as above. The third (incomplete) data group has two zero-value bits added to it, and is processed as above. A pad character (=) is added to the output.
3. The last data group has 8 bits (1 octet). The first 6-bit group is processed as above. The second (incomplete) data group has four zero-value bits added to it, and is processed as above. Two pad characters (=) are added to the output.

DECODING RADIX-64

In Radix-64 data, characters other than those in the table, line breaks, and other white space probably indicate a transmission error, about which a warning message or even a message rejection might be appropriate under some circumstances. Decoding software must ignore all white space.

Because it is used only for padding at the end of the data, the occurrence of any "=" characters may be taken as evidence that the end of the data has been reached (without truncation in transit). No such assurance is possible, however, when the number of octets transmitted was a multiple of three and no "=" characters are present.

EDDSA POINT FORMAT

The EdDSA algorithm defines a specific point compression format. To indicate the use of this compression format and to make sure the key can be represented in the Multiprecision Integer (MPI) format of [RFC4880] the octet string specifying the point is prefixed with the octet 0x40. This encoding is an extension of the encoding given in [RFC6637] which uses 0x04 to indicate an uncompressed point.

For example, the length of a public key for the curve Ed25519 is 263 bit: 7 bit to represent the 0x40 prefix octet and 32 octets for the native value of the public key.

TEST VECTORS

To help implementing this specification a non-normative example is given.

SAMPLE KEY

The secret key used for this example is (which holds K and Q):

```
Kprv: 0x1a8b1ff05ded48e18bf50166c664ab023ea70003d78d9e41f5758a91d850f8d2
Qpub: 0x3f098994bdd916ed4053197934e4a87c80733a1280d62f8010992e43ee3b2406
```

The public key encoded in the MPI format is:

```
Qmpi: 0x0107403f098994bdd916ed4053197934e4a87c80733a1280d62f8010992e43ee3
      b2406
```

SIGNATURE ENCODING

The MPIs representing the R and S value are encoded as MPIs. Note the compressed version of R and S as specified for EdDSA ([ED25519]) were used.

The signature below was created using this sample key:

```
d: 0xf6220a3f757814f4c2176ffbb68b00249cd4ccdc059c4b34ad871f30b1740280
```

The EdDSA signature function and yields this signature:

```
R: 56f90cca98e2102637bd983fdb16c131dfd27ed82bf4dde5606e0d756aed3366
S: d09c4fa11527f038e0f57f2201d82f2ea2c9033265fa6ceb489e854bae61b404
```

The MPI encoding rules require a prefix octet of oxoo, which yield a signature value of:

```
Smpi: 0x010056f90cca98e2102637bd983fdb16c131dfd27ed82bf4dde5606e0d756aed
      33660100d09c4fa11527f038e0f57f2201d82f2ea2c9033265fa6ceb489e854bae
      61b404
```

APPENDIX C: WHAT NEEDS DOING

Short Authenticating String section

Consolidate the language on signing and encryption key encoding in the signet specification.

Possibly reorganize the field types into: fixed, var(x) (aka variable length value), and var(1)/var(2) (aka variable length name and variable length value, aka undefined fields).

The different types of textual informational fields. Literals, semicolon delimited and identifier colon value fields.

Figure out if any additional rules need to be supplied for the identifier field beyond using utf-8 in Normalization Form C.

Provide a list of valid error codes for the DMTP commands.

Create the ABNF.

Consolidate the various encodings into its own chapter, so the information can be provided once. This means providing id strings and details for the PEM format, and then write up the translation rules for the JSON forms.

Write up the details for the optional pipelining and binary DMTP extensions.

Figure out if we should use radix-64 for everything, or stick with modified base64 (b64 w/o padding). Also consider Z85.

Finish describing the d/mime format. This means chunk descriptions. Tree and bounce signature descriptions are important, and figure out if we want to create a structure chunk in the meta section.

Finish writing the details for aux/alt encryption chunks. Make it clear that aux/alt must not be used on envelope, meta, or signature chunks.

Create a section on how private keys should encoded, encrypted and then stored.

Consolidate and formalize the signet validation rules into a checklist/binary decision tree. Including the rules regarding a primary, plus pre-authenticated source for automated trust acceptance. Guidelines for handling single source signets based on the different account mode. Cover the rules for validating a signet structure, independent of the content. Ensuring no reserved fields identifiers have been used. That all of the defined fields appear in numerical order. Then cover the required content validation rules. Check the text informational fields to ensure they only contain valid utf-8 codepoints. Ensure the encryption keys represent valid points on the signing or encryption curves. Rules for validating the chain of custody. Rules for the identity field comparisons when retrieving a signet. Trimming excessively long text fields.

Add details on compressing/decompressing Ed25519 public signing keys.

Provide a standardized list of properties for signet fields, something like: Identifier (type number), Label, Disposition, Type (binary or text), Format (applicable to text fields only, with the values: literal, semicolon delimited values, identified value pairs, and semicolon delimited identified value pairs), Length Limit, Regular Expression Validator (applicable to text

fields only), Validation Policy (trim, ignore, invalidate info fields only, invalidate entire signet), History (Defined by Revision 1, Updated by Revision 2).

Provide a standardized list of properties for message chunks.

For org signets, the services field doesn't define a list of recognizable identifiers, nor does it cite one. No comprehensive/IANA/standards body list of SRV identifiers currently exists, although IANA does provide a protocol service name registry.

The supported codecs list needs a collection of defined identifiers, for example, GIF, JPG, BPG, TIFF, BMP, vector formats, SVG, SVGZ, audio codecs/containers, MP3, WAV, AAC, WMA, FLAC, and video codecs/containers, AVI, MP4, 3GP, FLV, MKV, WMV, DIVX, WEBM/

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: November 19, 2018

L. Levison
Lavabit LLC
May 18, 2018

Safely Turn Authentication Credentials Into Entropy (STACIE)
draft-ladar-stacie-04

Abstract

This document specifies a method for Safely Turning Authentication Credentials Into Entropy (STACIE) using an efficient Zero Knowledge Password Proof (ZKPP), and is provided as a standalone component suitable for use as a building block in other protocol development efforts. The scheme was created to fill the emerging need for a standard which allows a single low entropy password to be used for user authentication and the derivation of strong encryption keys. The design is modular, and is conservative in its use of an arbitrary one-way cryptographic hash function. The security of the scheme depends on the difficulty associated with reversing the hash function output back into the plain text input. STACIE attempts to make discovering the plain text input through the use of brute force more difficult by correlating the amount of processing to the length of a user's plain text password. The shorter the plain text password, the more processing is required, with the amount of additional, artificially required, work scaling exponentially for each character.

Table of Contents

1. Introduction

A number of emerging client/server protocols are currently being developed which rely on endpoint encryption schemes for protection against server compromises and pervasive surveillance efforts. All of these protocols share a common need for the ability to authenticate users based on their account password, without having to share a plain text password with the server. While several proposals have emerged which rely on a Zero Knowledge Password Proof (ZKPP), none of them provide a standardized method for deriving a symmetric encryption key suitable for use with Authenticated Encryption with Associated Data (AEAD) ciphers using the same user password.

This specification describes a standalone scheme which solves these problems by Safely Turning Authentication Credentials Into Entropy (STACIE). Unlike previous efforts, STACIE can uniquely provide a configurable level of resistance against off-line brute force attacks aimed at recovering the original plain text password, or the derived encryption keys. Client side key stretching ensures attackers capable of eavesdropping on connections protected by Transport Layer Security (TLS), or with access to the authentication database on the server, will be unable to derive a user's password or their symmetric encryption keys.

STACIE is intended for use as a standalone component in other client/server protocol and application development efforts. While the protocol examples provided below are simplified, the abstract mechanism should easily translate into other encapsulation and encoding formats. Likewise, STACIE has been designed in a modular fashion, making it capable of using any arbitrary, but suitably strong, one-way cryptographic hash function. To ensure interoperability among different implementations, the Secure Hash Algorithm (SHA2-512) [SHS] MUST be implemented, while support for the newer Secure Hash Algorithm (SHA3-512) [PBH] and the Skein hash function (Skein-512) [SKEIN], are OPTIONAL.

For improved security, STACIE has been designed to provide extension points making it possible for specifications to extend the scheme with support for alternate authentication factors. The goal of this

specification is to accommodate a large variety of security requirements, while remaining conservative in its assumptions and its use of any particular cryptographic primitives.

To accommodate the unpredictable pace of improvements in computer hardware and processing power, STACIE includes a mechanism which allows system operators to increase the difficulty level and processing required by clients for key derivation beyond what is mandated by this specification.

The purpose of this document is to discourage the proliferation of multiple schemes for use by the variety of protocols currently in development which need to safely derive a symmetric encryption key, and authenticate a user with the server using a single low entropy password. While STACIE introduces strategies designed to strengthen key material against a variety of recently revealed threats, and provides a measure of protection associated with deficiencies in the randomness of human input, it is not intended as a call to change or update existing protocols and specifications.

2. Terminology

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 RFC 2119 [KEYWORDS] when, and only when, they appear in all capitals, as described by RFC 8174 [CAPITALIZATION].

3. Encodings

This document represents all of the request and responses using standard JavaScript Object Notation [JSON]. When an object value is text, the native UTF-8 representation is supplied. Otherwise the value is armored using the base64 encoding scheme defined in RFC 4648 [BASE], with the URL and filename safe character set defined in Section 5, and assigned the identifier "base64url." In addition to the standard base64url conversion, all trailing pad characters, line breaks, white space, and other non-printable control characters MUST be removed, as permitted by Section 3.2. [BASE] For the examples in this document, line breaks only appear when the sample value exceeds the available space.

4. Derivation Process

STACIE employs a multistage process which includes an extraction stage, two key derivation stages, and two token derivation stages. The stages MUST progress in a linear order because the output for each stage is used as an input for the subsequent stage. The

extraction and key derivation stages require a user's plain text password, while the token derivation stages do not. This allows the derived tokens to be used for authentication, because they can be generated and verified by a server without access to the plain text password.

Implementations MUST never store a user's plain text password. Client implementations which need the ability to authenticate and access encrypted user data without user input MUST store the verification token, and the individual realm hash. These values provide the ability to authenticate with a server, and access the realm specific encryption keys without additional user input. By storing just these values, an implementation ensures a user's plain text password is still REQUIRED to alter account credentials. This allows a user to recover from an endpoint compromise by updating their password, allowing for a point in time recovery.

Client implementations with support for automatic login capabilities on platforms which provide a secure storage facility SHOULD make use of this capability to protect the verification token, and realm hashes.

Required Inputs

The derivation process requires the following inputs:

username
: The normalized username.

password
: The plain text user password.

Optional Inputs

salt
: An additional non-secret, per-site, or per-user source of random entropy. The salt value ensures output independence and provides protection against computational reuse and precomputed table lookups. Salt values MUST provide a minimum of 64 octets, and SHOULD be less than 1,024 octets, with 128 octets the RECOMMENDED length. Salt values SHOULD be aligned along a 32 octet boundary.

nonce
: An array of randomly generated octets created by a server for each login attempt, which MUST be combined with the verification token to derive the ephemeral login token. The nonce value MUST be a minimum of 64 octets, and SHOULD be less than 1,024 octets, with

128 octets the RECOMMENDED length. Nonce values SHOULD be aligned along a 32 octet boundary.

bonus
: The fixed number of additional iterations added to the iteration count calculated dynamically based the password's length.

Outputs

rounds
: The REQUIRED number of hash rounds used for the extraction and key derivation stages.

master_key
: The key value REQUIRED for deriving realm keys, and used to derive the password key.

password_key
: The second key derivation output, and REQUIRED for deriving the verification token. The password is also used to authenticate password updates.

verification_token
: The persistent token stored on a server during account creation, or following a password update and then used to authenticate ephemeral login tokens in the future.

ephemeral_login_token
: The ephemeral token value which proves knowledge of the verification token for a singular login attempt, and is REQUIRED to authenticate a session or connection.

Example

The following code, written in Python, demonstrates how to derive the various outputs by calling the example functions provided in subsequent sections:

```
# Derive the Rounds
rounds = RoundsDerivation(password, bonus)

# Extract the Seed
seed = SeedDerivation(rounds, username, password, salt)

# Keys
master_key = KeyDerivation(seed, rounds, username, password, \
  salt)
password_key = KeyDerivation(master_key, rounds, username, \
  password, salt)

# Tokens
verification_token = TokenDerivation(password_key, username, \
  salt)
ephemeral_login_token = TokenDerivation(verification_token, \
  username, salt, nonce)

# Derive the Realm Key
realm_key = RealmKeyDerivation(master_key, label, shard, salt)

# Extract the Cipher and Vector Keys
vector_key = RealmVectorKeyExtraction(realm_key)
tag_key = RealmTagKeyExtraction(realm_key)
cipher_key = RealmCipherKeyExtraction(realm_key)

# Encryption and Decryption
encrypted_data = RealmEncrypt(vector_key, tag_key, cipher_key, \
  secret_message)
decrypted_data = RealmDecrypt(vector_key, tag_key, cipher_key, \
  encrypted_data)
```

4.1. Hash Rounds

To improve the security of short passwords, STACIE requires client implementations to calculate the appropriate number of iterations, or "rounds" used for string concatenation during the seed stage and the number hash rounds REQUIRED during the key derivation stages. The rounds variable is based on the number of characters, with short passwords requiring more rounds than long passwords. The variable number of rounds was designed to make systematically checking all of the possible plain text inputs more expensive in the event any of the derived tokens are compromised. It does not inherently provide security for predictable passwords which might be easily guessed.

To ensure the formula used to calculate the number of rounds, and the required processing remains effective against brute force attacks in

the future, a fixed number of "bonus" rounds MAY be added beyond what is required. The number of bonus rounds is dictated by the server configuration and MUST be added to the number calculated based on length. The bonus variable is primarily intended to offset improvements in computer performance in the future, for implementations which rely on hash algorithm after they've been deprecated.

When calculating the number of dynamic hash rounds clients MUST first determine the number of Unicode "characters" in a password, which is distinct from the number of octets. Many character encodings, such as UTF-8 use a variable number of octets per character, and the number of octets MAY change based on the input method editor. For consistency, the password MUST be converted into the UTF-8 encoding, and the number of Unicode characters determined. Because UTF-8 is capable of representing the same hashed characters using multiple octets, and using different binary values based on the normalization form, it is critical that the length used for this calculation is always based on the number of Unicode characters. This will ensure the number of rounds remains deterministic.

To determine the number of rounds, a client MUST subtract the number of Unicode characters from the constant value 24. If the result is negative, the value 1 MUST be used. The result of this calculation is used as a "dynamic" exponent, which is raised using the base 2, and the result is the "variable" number of rounds. The "bonus" rounds MUST be added to the "variable" number to derive the total number of rounds.

If the combined value of the dynamic and bonus values is less than 8, the value 8 MUST be used. Alternatively, if the value exceeds 16,777,216 the value MUST be reduced to this maximum value. The maximum value corresponds to the limit imposed by the use of the 3 octet counter employed during the entropy extraction and key derivation stages.

Token derivation employs a fixed, 8 rounds, to avoid leaking information about the password length.

Example

The following Python code demonstrates the proper method for deriving the number of rounds:

```
def RoundsDerivation(password, bonus):
    # Accepts a user password and bonus value, and calculates
    # the number of iterative rounds required. This function will
    # always return a value between 8 and 16,777,216.

    # Identify the number of Unicode characters.
    characters = len(password.decode("utf-8"))

    # Calculate the difficulty exponent by subtracting 1
    # for each Unicode character in a password.
    dynamic = operator.sub(24, characters)

    # Use a minimum exponent value of 1 for passwords
    # equal to, or greater than, 24 characters.
    dynamic = max(1, dynamic)

    # Derive the variable number of rounds based on the length.
    # Raise 2 using the dynamic exponent determined above.
    variable = pow(2, dynamic)

    # If applicable, add the fixed number of bonus rounds.
    total = operator.add(variable, bonus)

    # If the value of rounds is smaller than 8, reset
    # the value to 8.
    total = max(8, total)

    # If the value of rounds is larger than 16,777,216, reset
    # the value to 16,777,216.
    total = min(pow(2, 24), total)

    return total
```

4.2. Entropy Extraction

STACIE starts by deriving a fixed-length pseudorandom seed value which is "extracted" by "concentrating" the low-entropy user password into a short, but cryptographically strong pseudorandom value. Future extensions which incorporate a second authentication source that results in a quality pseudorandom value for the seed value may find this stage unnecessary.

Unlike the key and token derivation stages, the entropy extraction stage uses the Hashed Message Authentication Code [HMAC] algorithm, which is also defined by National Institute of Standards and Technology (NIST) as a Federal Information Processing Standard (FIPS)

[HMAC-FIPS]. Test vectors based on SHA2-512 are available [HMAC-SHA].

Implementations supporting the OPTIONAL SHA3-512 or Skein-512 hash functions MUST use an HMAC implementation bsaed on the appropriate SHA3-512 or Skein-512. Implementations SHOULD NOT use the Skein-MAC alternative described by the Skein paper [SKEIN]. Future STACIE extensions MAY provide alternative methods for seed extraction.

Unlike a simple hash, HMAC requires a 128 octet key value. The key value for the entropy extraction stage is derived from the salt value. If no salt value is available the username MUST be hashed and used as a substitute for the salt value. If the provided salt value is precisely 128 octets, then it MUST be used as the HMAC key.

When the provided salt is not 128 octets, then a key MUST be derived using an appropriate hash function, which provides a 128 octet value by digesting the salt value concatenated together with a counter variable. The process is performed twice, with the counter variable set to the values 0 and 1, respectively. The counter is digested as a 3 octet big endian integer value. The two hash digest output values MUST be concatenated to form the 128 octet HMAC key value.

The HMAC primitive also requires a "message" which is created using the plain text password, which MUST be provided to the HMAC primitive repeatedly, with the precise number of repetitions dictated by the "rounds" variable. The digest produced by the HMAC function becomes the 64 octet seed value used for the master key derivation stage.

Example

The following Python code demonstrates the proper method for extracting the entropy seed value:

```
   def SeedDerivation(rounds, username, password, salt=None):
       # Concentrates and then extracts the random entropy provided
       # by the password into a seed value for the first hash stage.

       # If if an explicit salt value is missing, use a hash of
       # the username as if it were the salt.
       if salt is None:
           salt = SHA512.new(username).digest()

       # Confirm the supplied salt meets the minimum length of 64
       # octets required, is aligned to a 32 octet boundary and does not
       # exceed 1,024 octets. Some implementations may not handle salt
       # values longer than 1,024 octets properly.
       elif len(salt) < 64:
           raise ValueError("The salt, if supplied, must be at least " \
             "64 octets in length.")
       elif operator.mod(len(salt), 32) != 0:
           warnings.warn("The salt, if longer than 64 octets, should " \
             "be aligned to a 32 octet boundary.")
       elif len(salt) > 1024:
           warnings.warn("The salt should not exceed 1,024 octets.")

       # For salt values which don't match the 128 octets required for
       # an HMAC key value, the salt is hashed twice using a 3 octet
       # counter value of 0 and 1, and the outputs are concatenated.
       if len(salt) != 128:
           key = \
               SHA512.new(salt + struct.pack('>I', 0)[1:4]).digest() + \
               SHA512.new(salt + struct.pack('>I', 1)[1:4]).digest()
       # If the supplied salt is 128 octets use it directly as the
       # key value.
       else:
           key = salt

       # Initialize the HMAC instance using the key created above.
       hmac = HMAC(key, None, SHA512)

       # Repeat the plain text password successively based on
       # the number of instances specified by the rounds variable.
       for unused in range(0, rounds):
           hmac.update(password)

       # Create the 64 octet seed value.
       seed = hmac.digest()

       return seed
```

4.3. Key Derivation

There are two successive key derivation stages. The master key is first, and requires the extracted seed value derived in the previous stage, along with the calculated number of rounds, the username, password, and if available, the salt value. The master key MUST be kept private. It provides the secret material needed to derive the realm specific subkeys used to encrypt data on the client.

The second key derivation stage provides the password key. It uses an identical process as the master key stage, with the exception of the seed value being replaced by the master key value derived in the first stage. The password key MUST be kept private until it comes time for a user to update their password. Password updates require sharing the password key with a server, which can then confirm the value translates into the current verification token, before updating the values stored in the authentication database. This ensures a that a compromised authentication database can't be used by an attacker to alter user passwords.

Each key derivation stage repeats the hash process by the variable number of iterations dictated by the rounds variable. Assuming the hash function remains securely one-way, this strategy ensures key derivation requires a linear computational process. The amount of processing time is a product of the difficulty imposed by the rounds variable and a client's computational performance. The linear nature of the process means the time required for individual rounds MAY be shortened but the rounds MUST NOT be processed in parallel.

Hash values are generated by concatenating the input seed (or master key value) together with the with the username, salt, password and counter value. Successive rounds repeat the process, using an incremented counter value, and include the output of the previous round prepended to the input. The counter value MUST be digested as a 3 octet big endian integer value, and represents a 0 based value corresponding to the current round.

Example

The following Python code demonstrates the proper method for key derivation, with the seed value either the extracted seed, or the master key, depending on the stage:

```
def KeyDerivation(seed, rounds, username, password, salt=""):
    # Hash the input values together using the input values, and
    # repeat the process, with the number of iterations dictated by
    # the rounds variable.

    count = 0
    hashed = ""

    while count < rounds:
        hashed = SHA512.new(hashed + seed + username + salt + \
            password + struct.pack('>I', count)[1:4]).digest()
        count = operator.add(count, 1)

    # The last digest output is returned as the key value.
    return hashed
```

4.4. Token Derivation

The token derivation process is distinct from the key derivation process because it is repeatable without knowing a user's password. The password key is combined with other inputs to derive the verification token, and the verification token is then shared with the server, which can use it to authenticate future login attempts. To prevent replay attacks, the verification token is combined with a nonce value, and using the same token derivation process, a unique ephemeral login token is generated for each session or connection.

Like the key derivation stages defined above, the seed value in the sample code below represents the output from the previous stage, which is either the password key or the verification token. This value is concatenated together with the salt value, if applicable, and a nonce value (when deriving the ephemeral token). A counter value is also appended, with the value representing a 3 octet big endian integer value, and corresponding to a 0 based count of the current round. The output for each round is prepended to the input of successive rounds, with a fixed 8 rounds performed during each token derivation stage.

__Example__hashed

The following Python code demonstrates the proper method for token derivation, with the seed value either the password key, or the verificiation token, depending on the stage:

```
def TokenDerivation(seed, username, salt="", nonce=""):
    # Hash the input values together using the input values, and
    # repeat the process eight times.

    count = 0
    rounds = 8
    hashed = ""

    # Confirm the nonce, if it was provided, meets the minimum
    # length of 64 octets, does not exceed 1,024 octets, and is
    # aligned along a 32 octet boundary. Implementations may not
    # handle nonce values larger than 1,024 octets properly.
    if len(nonce) > 0 and len(nonce) < 64:
        raise ValueError("Nonce values must be at least " \
          "64 octets in length.")
    elif operator.mod(len(nonce), 32) != 0:
        warnings.warn("The nonce value, if longer than 64 octets, " \
          "should be aligned to a 32 octet boundary.")
    elif len(nonce) > 1024:
        warnings.warn("The nonce should not exceed 1,024 octets.")

    while count < rounds:
        hashed = SHA512.new(hashed + seed + username + salt + \
            nonce + struct.pack('>I', count)[1:4]).digest()
        count = operator.add(count, 1)

    return hashed
```

4.5. Realm Key Derivation

Realm specific keys are used to access and authenticate symmetrically encrypted user data. The realm label specifies the category and/or type of data protected by a given realm key. Protocols which incorporate STACIE MAY use a single realm, or separate data into different realms based on the data type. Every realm is protected by a unique encryption key. The realms are isolated to allow separable handling, and isolation, such that if one realm key is compromised, it is possible for the remaining realms to remain secure, provided the master key was not compromised, or the attacker is unable to gain access to the shard values for other realms.

The shard value is a randomly generated string of 64 octets, provided after successful authentication, which allows a client to derive a realm key. Because the shard is stored on the server, an endpoint compromise won't yield the necessary information to decrypt any locally stored data, after the user updates their credentials. This

will mitigate the damage that would occur when a device with cached data is lost or stolen.

The unique key for a realm is derived by concatenating, then hashing the master key, realm label, and salt. The resulting digest is then combined with a realm shard value using the bitwise exclusive "or" operation. The result is a "realm key" which contains the concatenated vector key, tag key, and cipher key values. The vector key is comprised of the first 16 octets, the tag key is protected by the subsequent 16 octets, and the cipher key is comprised of the final 32 octets.

Required Inputs

The master key, as previously described, is combined with the following required inputs:

label
: The realm label, a predefined lowercase string describing the category and/or type of data.

The salt is only required if a salt value was used to derive the master key:

salt An additional non-secret, per-site, or per-user source of random entropy. The salt value increases the unpredictability of the output. Salt values MUST provide a minimum of 64 octets, and SHOULD be less than 1,024 octets, with 128 octets the RECOMMENDED length. Salt values SHOULD be aligned along a 32 octet boundary.

Outputs

realm_key
: The realm specific key distilled from the provided inputs, and is the combination of the vector, tag and cipher key values.

vector_key
: The key used to unlock the initialization vectors for a given realm.

tag_key
: The key used to unlock the authentication tags for a given realm.

cipher_key
: The key used by the symmetric cipher to decrypt user data associated with a given realm.

Example

The following Python code demonstrates how to derive and then separate the keys for a given realm:

```
def RealmKeyDerivation(master_key, label="", shard="", salt=""):

    if len(label) < 1:
        raise ValueError("The realm label is missing or invalid.")
    elif len(shard) != 64:
        raise ValueError("The shard length is not 64 octets.")
    elif len(master_key) != 64:
        raise ValueError("The master key length is not 64 octets.")

    # The salt value is optional, but if supplied, must be a minimum
    # of 64 octets in length, and no more than 1,024 octets in
    # length. It should be aligned to a 32 octet boundary. Some
    # implementations may not handle salt values longer than 1,024
    # octets properly.
    elif len(salt) != 0 and len(salt) < 64:
        raise ValueError("The salt, if supplied, must be at least " \
          "64 octets in length.")
    elif len(salt) != 0 and operator.mod(len(salt), 32) != 0:
        warnings.warn("The salt, if longer than 64 octets, should " \
          "be aligned to a 32 octet boundary.")
    elif len(salt) > 1024:
        warnings.warn("The salt should not exceed 1,024 octets."

    realm_hash = SHA512.new(master_key + label + salt).digest()
    realm_key = str().join(chr(operator.xor(ord(a), ord(b))) \
      for a,b in zip(realm_hash, shard))

    return realm_key

def RealmVectorKeyExtraction(realm_key):
    vector_key = realm_key[0:16]

    return vector_key

def RealmTagKeyExtraction(realm_key):
    tag_key = realm_key[16:32]

    return tag_key

def RealmCipherKeyExtraction(realm_key):
    cipher_key = realm_key[32:64]

    return cipher_key
```

5. Encryption

STACIE requires client implementations to support the Advanced Encryption Standard [AES] using 256 bit key values. To ensure data integrity, and protect against manipulation by a malicious server, AES MUST be employed using the Galois Counter Mode [GCM]. The binary format specifies a 34 octet envelope, followed by a payload aligned to a 16 octet boundary. The payload includes a 4 octet prefix, and a variable amount of padding appended as a suffix for alignment purposes.

5.1. Envelope

Symmetrically encrypted buffers are preceeded by an envelope, consisting of the realm serial number, the initialization vector shard, and the authentication tag shard. The serial number is a 2 octet big endian integer corresponding to the realm key used to derive the key values associated with a given buffer. It is possible for a realm to have buffers encrypted using different serial numbers. The number MAY be increased when users update their password. The serial number is followed by a 16 octet initialization vector shard, which MUST be randomly generated whenever data is encrypted. The vector shard is combined with the vector key using a bitwise exclusive "or" operation to produce the initialization vector used for a given cipher text. The final envelope value is a 16 octet tag shard, which like the vector shard, MUST be combined with the tag key using a bitwise exclusive "or" operation to produce the authentication tag for a given cipher text.

Envelope Parameters

serial
: The serial number is a 2 octet big endian integer which delineates which shard value for a given realm MUST be used to derive the realm key.

vector_shard
: The randomly generated 16 octet value generated during encryption, and then combined with the vector key to using a bitwise exclusive "or" operation. The result is the initialization vector for a given cipher text.

tag_shard
: A 16 octet authentication tag is created during the encryption process, and then combined with the tag key using a bitwise exclusive "or" operation to create the tag shard. To produce the authentication tag for a cipher text, the tag key MUST be combined

with the tag shard using to another bitwise exclusive "or" operation when the buffer is decrypted.

5.2. Payload

The envelope data is immediately followed by the encrypted payload, which consists of the encrypted plain text value, a 4 octet prefix, and up to 255 octets of padding appended after the plain text. The entire encrypted/decrypted payload, including the prefix and suffix, MUST align to a 16 octet boundary. The prefix begins with a 3 octet big endian integer which denotes the length of the plain text value, and is is followed by a single octet pad value. The pad value indicates how many additional octets have been appended to the plain text value t0 align the payload to the 16 octet boundary. The amount of padding MUST include the requisite 0 to 15 octets required to align the payload, but MAY also include a random amount of OPTIONAL padding in 16 octet increments. Specifically, the pad value MAY include an additional 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 178, 192, 208, 224, or 240 octets beyond those required for alignment. The padding octets appended after the plain text value, or suffix, MUST match the value of the padding octet in the prefix.

size
: The length of the plain text value represented as a 3 octet, big endian integer.

pad
: The amount of padding appended to the plain text value generated 16 octet value generated during encryption, and then combined with the vector key to using a bitwise exclusive "or" operation. The result is the initialization vector for a given cipher text.

buffer
: A plain text value worthy of protection.

padding
: Up to 255 octets of padding, with the padding octets all set to the pad value.

Example

The following Python code demonstrates how to encrypt a plain text value:

```
   def RealmEncrypt(vector_key, tag_key, cipher_key, buffer, serial=0):

       count = 0

       if serial < 0 or serial >= pow(2, 16):
           raise ValueError("Serial numbers must be greater than 0 " \
               "and less than 65,536.")
       elif len(cipher_key) != 32:
           raise ValueError("The encryption key must be 32 octets " \
               "in length.")
       elif len(vector_key) != 16:
           raise ValueError("The vector key must be 16 octets in " \
               "length.")
       elif len(buffer) == 0:
           raise ValueError("The secret being encrypted must be at " \
               "least 1 octet in length.")
       elif len(buffer) >= pow(2, 24):
           raise ValueError("The secret being encrypted must be at " \
               "less than 16,777,216 in length.")


       vector_shard = get_random_bytes(16)

       iv = str().join(chr(operator.xor(ord(a), ord(b))) \
           for a,b in zip(vector_key, vector_shard))

       size = len(buffer)
       pad = (16 - operator.mod(size + 4, 16))

       while count < pad:
           buffer += struct.pack(">I", pad)[3:4]
           count = operator.add(count, 1)

       encryptor = Cipher(algorithms.AES(cipher_key), modes.GCM(iv), \
           backend=default_backend()).encryptor()
       ciphertext = encryptor.update(struct.pack(">I", size)[1:4] \
           + struct.pack(">I", pad)[3:4] + buffer) \
           + encryptor.finalize()

       tag_shard = str().join(chr(operator.xor(ord(a), ord(b))) \
           for a,b in zip(tag_key, encryptor.tag))

       return struct.pack(">H", serial) + vector_shard + tag_shard \
           + ciphertext
```

The following Python code demonstrates how to decrypt and validate the cipher text created by the encryption function above:

```
   def RealmDecrypt(vector_key, tag_key, cipher_key, buffer):

       count = 0

       # Sanity check the input values.
       if len(cipher_key) != 32:
           raise ValueError("The encryption key must be 32 octets in "\
               " length.")
       elif len(tag_key) != 16:
           raise ValueError("The tag key must be 16 octets in length.")
       elif len(vector_key) != 16:
           raise ValueError("The vector key must be 16 octets in " \
               "length.")
       elif len(buffer) < 54:
           raise ValueError("The minimum length of a correctly " \
               "formatted cipher text is 54 octets.")
       elif operator.mod(len(buffer) - 34, 16) != 0:
           raise ValueError("The cipher text was not aligned to " \
               "a 16 octet boundary or some of the data is missing.")

       # Parse the envelope.
       vector_shard = buffer[2:18]
       tag_shard = buffer[18:34]
       ciphertext = buffer[34:]

       # Combine the shard and key values to get the iv and tag.
       iv = str().join(chr(operator.xor(ord(a), ord(b))) \
           for a,b in zip(vector_key, vector_shard))

       tag = str().join(chr(operator.xor(ord(a), ord(b))) \
           for a,b in zip(tag_key, tag_shard))

       # Decrypt the payload.
       decryptor = Cipher(algorithms.AES(cipher_key), \
           modes.GCM(iv, tag), backend=default_backend()).decryptor()
       plaintext = decryptor.update(ciphertext) + decryptor.finalize()

       # Parse the prefix.
       size = struct.unpack(">I", '\x00' + plaintext[0:3])[0]
       pad = struct.unpack(">I", '\x00' + '\x00' + '\x00' + \
           plaintext[3:4])[0]

       # Validate the prefix values.
       if operator.mod(size + pad + 4, 16) != 0 or \
           len(plaintext) != size + pad + 4:
           raise ValueError("The encrypted buffer is invalid.")

       # Confirm the suffix values.
```

```
       for offset in xrange(size + 4, size + pad + 4, 1):
           if struct.unpack(">I", '\x00' + '\x00' + '\x00' + \
               plaintext[offset: offset + 1])[0] != pad:
               raise ValueError("The encrypted buffer contained " \
                   an invalid padding value.")

       # Return just the plain text value.
       return plaintext[4:size + 4]
```

6. Password Changes

6.1. Shallow Password Change

Required Inputs

The derivation process requires the following inputs:

new_master_key
: The master key created using the new password.

new_salt
: The salt value associated with the new password. Note this value SHOULD be different following a password change.

realm_key
: The realm specific key distilled using the previous password, salt, and current shard value.

label
: The realm label, a predefined lowercase string describing the category and/or type of data.

Outputs

realm_shard
: A replacement shard value, which will result in the same realm key being derived when combined with the new master key.

Example

The following code, written in Python, demonstrates how to derive a new realm shard value during password changes:

```
 def RealmShardRotation(new_master_key, new_salt, realm_key, label):

     if len(new_master_key) != 64:
         raise ValueError("The master key is not 64 octets.")
     elif len(new_salt) < 1:
         raise ValueError("The salt is missing or invalid.")
     elif len(realm_key) != 64:
         raise ValueError("The previous realm key is not 64 octets.")
     elif len(label) < 1:
         raise ValueError("The realm label is missing or invalid.")

     realm_hash = SHA512.new(new_master_key + label + new_salt).digest()
     realm_shard = str().join(chr(operator.xor(ord(a), ord(b))) \
         for a,b in zip(realm_hash, realm_shard))

     return realm_shard
```

6.2. Deep Password Change

6.3. Hybrid Password Change

7. Protocol

7.1. Create User

When the birds mate with the bees a new account is born.

```
{ register:
    { username: "user-alias@example.tld" }
}

{ recruit:
  { username: "user-alias@example.tld",
    salt: "Wb4vfzSpBpDRKafDlhhba3KhjIh09_4-IAl22XOcaI2z9O0QNdvNxFiRBM
      qsyr4yD90OmDxBckHJzijGF7d1PEsrGwlGEb9YCVpNvKiIgLeAPxz1OB7mn03wL
      RCfzYA8Ab8kvkinoZjHVnr6Fd34RS6bYB-mBB5WX2iQ-TBKZlE",
    bonus: "131072",
    hash: "sha2" }
}

{ error: "Registration is currently disabled." }
{ error: "The requested username is unavailable." }
{ error: "A dramatic increase in cosmic radiation means registration
    is temporarily unavailable." }

{ enroll:
  { username: "user-alias@example.tld",
    salt: "Wb4vfzSpBpDRKafDlhhba3KhjIh09_4-IAl22XOcaI2z9O0QNdvNxFiRBM
      qsyr4yD90OmDxBckHJzijGF7d1PEsrGwlGEb9YCVpNvKiIgLeAPxz1OB7mn03wL
      RCfzYA8Ab8kvkinoZjHVnr6Fd34RS6bYB-mBB5WX2iQ-TBKZlE",
    verification-token: "egf9dS64Z5b5qmrW4JYT86iNxDwHM5PvLF7DkyufIUwX
      2bAZ8p7iDcHNLVbT53_zZUMWgxWIxAxmWw6d8nAv9Q" }
}
```

7.2. Login

The login process begins by submitting a "login" request with the response providing an array of method objects each with the parameters REQUIRED to compute the secret values needed for key derivation and the tokens used for authentication. This includes the password object which provides the nonce value REQUIRED to generate the ephemeral login token used to validate the session or connection.

7.2.1. Login Request

A login request supplies a single username parameter, which is REQUIRED, and ensures equivalent inputs always provide a common, deterministic outcome.

Required Parameters

username
: The username value provide must be submitted to the server for normalization, canonicalization and alias mapping to ensure a

deterministic result. The specific rules applied are determined by the account policies and system locale for the server. Typically, this will include lower-case characters, decomposing ambiguous characters, adding, removing or altering the domain name component, and mapping aliases to a real username.

Example

```
{ login:
    { username: "user-alias@example.tld" }
}
```

7.2.2. Login Response

The response provides an array of method objects corresponding to different authentication mechanisms along with any requisite parameters. A disposition attribute indicates whether a particular method is OPTIONAL or REQUIRED. Currently, STACIE only provides details for key derivation using passwords. Future specifications MAY extend this scheme to support common alternate, or additional methods, including second factor mechanisms, which is indicated by the presence of multiple method objects marked as REQUIRED.

If a user or site specific salt value is available, it MUST be returned in the password object. The salt provides a non-secret random value which ensures independence between different uses of the same password at different points in time. The salt value is particularly important for sites with a policy of stripping the domain portion off usernames, as a unique salt will ensure independence between accounts with an identical username and password, but residing on different systems.

The singular method defined by this specification is the password mechanism, which provides an object containing the following parameters specified below.

Required Parameters

username
: The username returns the normalized username in a form suitable for use as an input parameter to the cryptographic hash function. Presumably, this will involve matching the value provided by the client with a static username identifier to ensure a deterministic output.

salt

The salt provides additional entropy for the cryptographic hash function. The salt value SHOULD be randomly generated and unique for every username. A minimum of 64 octets SHOULD be returned, with additional octets allowed in 32 octet increments. Clients MUST be capable handling salt values up to 1,024 octets in length.

nonce
The nonce MUST be combined with the verification token, which results in the ephemeral login token. Server implementations MUST ensure a unique nonce is used for each authentication attempt.

Optional Parameters

bonus
The bonus value mandates an arbitrary number of additional hash rounds a client MUST perform during each stage, in addition to the base rounds, and MAY be used by system operators to mitigate improvements in computing performance, or simply provide additional security sensitive accounts. Clients must accept and support values between 0 to 1,024. Implementations MAY provide support for values higher than 1,024. If this attribute is missing, a client MUST assume a default value of 0.

hash
The hash value provides an object which identifies the one-way hash function, along with any parameters specific to the supplied primitive. This specification defines the hash objects for the "sha2" and "skein" primitives. Clients MUST support the SHA2 algorithm. Support for SHA3 or Skein is OPTIONAL. If the hash object is missing, a client SHOULD assume the SHA2 algorithm with block and digest attribute values of 512 bits. If a SHA2 or Skein object is returned without block or digest values, a client MUST assume the default value of 512 bits.

cipher
The cipher value provides an object which identifies the symmetric cipher used to encrypt and decrypt data retrieved from the server along with any algorithm specific parameters. This specification mandates that all implementations MUST be capable of supporting the "aes" primitive using the "gcm" block mode with a 256 bit key. If the cipher object is missing, clients MUST assume that AES [AES] is being used in the GCM [GCM] block mode, with a 256 bit key. These same default values MUST be used if the cipher object specifies AES, but lacks values for the mode and key attributes.

disposition
An enumerated value, with values of OPTIONAL and REQUIRED. If this value is missing, REQUIRED is presumed as the default value.

If two or more method objects are marked as REQUIRED, then 2 factor authentication is implied.

Example

```
{ methods:
  [ password:
    { username: "user@example.tld",
      salt: "lyrtpzN8cBRZvsiHX6y4j-pJOjIyJeuw5aVXzrItw1G4EOa-6CA4R9Bh
        VpinkeH0UeXyOeTisHR3Ik3yuOhxbWPyesMJvfp0IBtx0f0uorb8wPnhw5BxD
        JVCb1TOSE50PFKGBFMkc63Koa7vMDj-WEoDj2X0kkTtlW6cUvF8i-M",
      nonce: "oDdYAHOsiX7Nl2qTwT18onW0hZdeTO3ebxzZp6nXMTo__0_vr_AsmAm
        3vYRwWtSCPJz0sA2o66uhNm6YenOGz0NkHcSAVgQhKdEBf_BTYkyULDuw2fSk
        bO7mlnxEhxqrJEc27ZVam6ogYABfHZjgVUTAi_SICyKAN7KOMuImL2g",
      bonus: "131072",
      hash: "sha2",
      cipher: "aes",
      disposition: "required" }
  ]
}
```

7.3. Password Authentication

The process for a password based authentication concludes by submitting an "authenticate" request with an ephemeral login token. The response provides a keys array, with objects corresponding to the various realm specific keys specific to the protocol. These values are combined with the master key to derive the symmetric keys for the various realms used to encrypt data on a client.

7.3.1. Authenticate Request

The authenticate object is submitted to a server for validation.

Required Parameters

username
: The normalized username.

nonce
: A randomly generated value, which MUST be combined with the verification token to create an ephemeral login token. Every nonce value MUST only be used for one authenticate request. Failed login attempts require a new nonce value to retry the login attempt.

token

The ephemeral login token needed to authenticate a session or token.

Example

```
{ authenticate:
    { username: "user@example.tld",
      nonce: "oDdYAHOsiX7Nl2qTwT18onW0hZdeTO3ebxzZp6nXMTo__0_vr_AsmAm
        3vYRwWtSCPJz0sA2o66uhNm6YenOGz0NkHcSAVgQhKdEBf_BTYkyULDuw2fSk
        bO7mlnxEhxqrJEc27ZVam6ogYABfHZjgVUTAi_SICyKAN7KOMuImL2g",
      token: "-Eu5mUcA7ko2BysV965hrf9bvMlh_S_iiI3tfMr0Qc7hf4oPmBCdGOU
        9VCeQ1qBrga-WyR-rko5l0-feoWuuuA"
    }
}
```

7.3.2. Authenticate Response

If the authentication attempt was successful the server will return an array of realm shards.

Required Parameters

index
: The an incrementing counter corresponding to each shard value.

label
: A protocol specific string containing the realm where the key value is used.

shard
: The random bytes which are combined with the master key to derive a realm specific key value.

Example

```
{ realms: [
    { index: "1",
      label: "mail",
      shard: "gD65Kdeda1hB2Q6gdZl0fetGg2viLXWG0vmKN4HxE3Jp3Z0Gkt5prqS
        mcuY2o8t24iGSCOnFDpP71c3xl9SX9Q",
      }
  ]
}
```

However, if the authentication request is unsuccessful and the server is willing to allow the client another attempt, it will return a

login response with a unique nonce value. A nonce value MUST only be used once regardless of whether the attempt is successful. The following example only contains the required parameters.

Example

```
{ methods:
  [ password:
    { username: "user@example.tld",
      salt: "lyrtpzN8cBRZvsiHX6y4j-pJOjIyJeuw5aVXzrItw1G4EOa-6CA4R9Bh
        VpinkeH0UeXyOeTisHR3Ik3yuOhxbWPyesMJvfp0IBtx0f0uorb8wPnhw5BxD
        JVCb1TOSE50PFKGBFMkc63Koa7vMDj-WEoDj2X0kkTtlW6cUvF8i-M",
      nonce: "vQmxYp9sznZJ1M62AxSGe3cQgMqTmVw92E1qfNR_Fl_u2zVFEiyV5dV
        2abGEhsWPDkHsxtJGj-NTEF1vet1mlgfD67mQO1IPG7RfxPmEAJwAWGWkbgPG
        kQI2tpfAs5LqQai-Any3I95Kq-eTPIP8ykQYXKW8qO-DJCw5SmmCrJs" }
  ]
}
```

Or if the server does not want to allow any further attempts to access the account, it MAY also return an error message.

```
{ error: "The authentication attempt failed." }
```

7.4. Password Change

Update the verification token, and salt values on the server. Note the salt value is only updated if user specific salt values are being used. Alter any existing realm specific shard values, and if required add new randomly generated realm specific shard values.

7.5. Fetch Realm Shards

Fetch the realm shard values. The result MAY be request a specific realm, and serial number.

7.6. Add a Realm Shard

Add a shard, for a given realm, to the account using the next available serial number.

8. Security Considerations

Client and server implementations SHOULD follow the recommendations provided here to avoid leakage, and improve difficulty.

8.1. Servers

Username Enumeration

To avoid enumeration and avoid leaking the list of valid user accounts, servers SHOULD respond to authenticate requests with valid and invalid usernames in the same fashion. Because salt values are typically unavailable in this situation, servers SHOULD normalize and return the username along with a dynamically derived salt value generated by combining the username with a site specific value. This will ensure a consistent salt value is returned on subsequent requests for the same invalid username. Servers MAY choose to return an error if the username contains invalid characters, or was provided with an unrecognized domain name.

Salt Values

To ensure STACIE provides the maximum amount of protection, implementations SHOULD generate unique, random salt values for every user, and then rotate the salt value every time the password is updated. This will ensure independence between common inputs, and strengthen the security analysis underpinning the design [HKDF].

8.2. Clients

Side Channels

A properly implemented client SHOULD ensure it's impossible for an attacker to correlate the duration between client request/responses with the plain text password length. Several mitigation strategies are possible, including submitting authentication requests independently of when users input their password. Adding random delays between hash rounds, which are independent of system load and processor speed, or using a constant duration for password processing independent length, are also possible. Clients MAY round any artificial processing delays to aligned boundaries, which would also make correlation more difficult.

8.3. Shared

Transport Security

STACIE implementations MUST support TLS using a ciphersuite capable of protecting against network eavesdroppers, data tampering and ensure the confidentiality of messages. Protocols incorporating STACIE as a component MUST provide recommendations sensitive to their intended context, but SHOULD encourage the use of TLS version 1.2, or later, and limit implementations to ciphersuites capable of providing

perfect forward secrecy. Server deployments SHOULD ensure they provide valid TLS certificates, and client implementations SHOULD ensure they properly validate server certificates using the procedures described in RFC 6125 [TLS-PKIX] or optionally, using the procedures described in RFC 6698 [TLS-DANE].

As of this writing, the RECOMMENDED ciphersuite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384, identified by the octet values {0xC0, 0x30}, or the equivalent ECDSA variant, TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384, which is identified by the octet values {0xC0,0x2C}. [TLS-GCM]

Specific requirements and recommendations will need to be updated over time, based on what is widely deployed, and MAY need altering based on future vulnerability discoveries. To obtain contemporary guidance, or find additional recommendations, implementers and system operators SHOULD consult the Recommendations for Secure Use of TLS and DTLS [TLS-UTA].

9. IANA Considerations

This document has no actions for IANA.

10. Feedback

The preceding document was excreted with the assistance of a diarrhoetic. As such, feedback is both welcome, and encouraged.

11. Acknowledgments

The genesis for STACIE was the authentication and key derivation method used by Lavabit LLC to authenticate client connections and protect the user specific private keys. Improvements were made while adapting the original server based scheme to operate on clients being developed for the Privacy Respecting Internet Mail Environment (PRIME). The author would also like to acknowledge and thank the One Password Protocol [ONEPW] developed for Firefox Sync and the HKDF [HKDF] specification for inspiring some of the improvements incorporated into STACIE.

The improvements were all focused on providing operational flexibility, extensibility, while improving the security characteristics of short, relatively simple passwords commonly chosen by bipedal hominids. Acknowledgment must also be given to the large online services which allowed their password databases to be publicly scrutinized. Analysis of these databases proved invaluable while selecting the constants used by STACIE, and allowed the author to see

how variations effected the dynamic difficulty level for a random sampling of real passwords.

The goal for STACIE was to ensure it provided sufficient resistance against brute force attacks for the vast majority of passwords which will inevitably be used. Admittedly the term "sufficient resistance" is very subjective, and is constantly being shifted by advances in technology. Thanks should be given to the critics. Their complaints led to a modular hash algorithm, and the strategy of combining a dynamically calculated difficulty with a policy based bonus. Hopefully these decisions will ensure the survival of users with short password who inevitably get stuck on the long tail. STACIE is not a substitute for long, truly random, and incredibly complex passwords used by any evolved hominids capable of remembering them.

The author would also like to thank Stacie for inspiring the name. Her resistance to having a computer bear her name, inevitably, led to something far better.

12. Normative References

[AES] National Institute of Standards and Technology, "Advanced Encryption Standard (AES), FIPS 197", November 2001, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

[BASE] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", October 2006, <https://www.ietf.org/rfc/rfc4648.txt>.

[CAPITALIZATION] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", May 2017, <https://www.ietf.org/rfc/rfc8174.txt>.

[GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, SP 800-38D", November 2007, <http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf>.

[HKDF] Krawczyk, H., "Cryptographic Extraction and Key Derivation: The HKDF Scheme", May 2010, <https://eprint.iacr.org/2010/264>.

[HMAC] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", February 1997, <https://www.ietf.org/rfc/rfc2104.txt>.

[HMAC-FIPS]
National Institute of Standards and Technology, "The Keyed-Hash Message Authentication Code (HMAC), FIPS 198-1", July 2008, <http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf>.

[HMAC-SHA]
Nystrom, M., "Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512", December 2005, <https://www.ietf.org/rfc/4231.txt>.

[JSON] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", December 2017, <https://www.ietf.org/rfc/rfc8259.txt>.

[KEYWORDS]
Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", March 1997, <https://www.ietf.org/rfc/rfc2119.txt>.

[ONEPW] Boulange, R., "One Password Protocol", May 2014, <https://github.com/mozilla/fxa-auth-server/wiki/onepw-protocols>.

[PBH] National Institute of Standards and Technology, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, FIPS 202", August 2015, <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>.

[SHS] National Institute of Standards and Technology, "Secure Hash Standard, FIPS 180-2", August 2015, <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.

[SKEIN] Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., and J. Walker, "The Skein Hash Function Family", November 2008, <http://www.skein-hash.info/sites/default/files/skein1.1.pdf>.

[TLS-DANE]
Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", August 2012, <https://www.ietf.org/rfc/rfc6698.txt>.

[TLS-GCM] Rescorla, E., "TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)", August 2008, <https://www.ietf.org/rfc/rfc5289.txt>.

[TLS-PKIX] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", March 2011, <https://www.ietf.org/rfc/rfc6125.txt>.

[TLS-UTA] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of TLS and DTLS", February 2015, <https://www.ietf.org/id/draft-ietf-uta-tls-bcp-11.txt>.

Appendix A. Test Vectors

This appendix provides test vectors. Binary values are provided using the base64url encoding, with line breaks added as necessary.

A.1. Inputs

```
# User Inputs
password = "password"
username = "user@example.tld"

# Server Inputs
bonus = 131072
salt = "lyrtpzN8cBRZvsiHX6y4j-pJOjIyJeuw5aVXzrItw1G4EOa-6CA4R" \
    "9BhVpinkeH0UeXyOeTisHR3Ik3yuOhxbWPyesMJvfp0IBtx0f0uorb8w" \
    "Pnhw5BxDJVCb1TOSE50PFKGBFMkc63Koa7vMDj-WEoDj2X0kkTtlW6cU" \
    "vF8i-M"
nonce = "oDdYAHOsiX7Nl2qTwT18onW0hZdeTO3ebxzZp6nXMTo__0_vr_" \
    "AsmAm3vYRwWtSCPJz0sA2o66uhNm6YenOGz0NkHcSAVgQhKdEBf_BT" \
    "YkyULDuw2fSkbO7mlnxEhxqrJEc27ZVam6ogYABfHZjgVUTAi_SICy" \
    "KAN7KOMuImL2g"

# Realm Inputs
realm = "mail"
shard = "gD65Kdeda1hB2Q6gdZl0fetGg2viLXWG0vmKN4HxE3Jp3Z" \
    "0Gkt5prqSmcuY2o8t24iGSCOnFDpP71c3xl9SX9Q"

# Encrypted Data
encrypted_data = "AACS5PQoBg4ON1Xt6aUSddMxTTIKGdbGSelUkIbUkUj" \
    "prZv9ekAwPRrJOUqJqWGhdgEvCzSkZwr-kvNZo6f2IW1a
```

A.2. Outputs

```
rounds = 196608
seed = "5f-3mTGTSf-sFPfMkGqHTyydDjJU-cqahwDmHWyh6DLQ2oLBlz3ht" \
    "PTZS6V-TYVBiwJxuTYmQv3fCZN3Fb8brg"

master_key = "SDt67ZfTr8c1KO1Ym6BI69i7TQNNq5J2irym6gPQlEo0MGc" \
    "5x-b43bi1uXJDF4rhJJvfl9NFBQkDQ_X_2n66RA"
password_key = "lYmvC3qutKIb6QrnxnTi_WuJR_PSiyMZ0CdH18DAxHIgw" \
    "jj0_e4W6X8bKckKNGugWMMXmNgXDYb_7LlvtfN3HQ"

verification_token = "-Eu5mUcA7ko2BysV965hrf9bvMlh_S_iiI3tfMr" \
    "0Qc7hf4oPmBCdGOU9VCeQ1qBrga-WyR-rko5l0-feoWuuuA"
ephemeral_login_token = "8YEH_6kBdAdR5vlBaxs3KR3pZ429bEzF3AVF" \
    "hkA0P2WPt2h94omJq-d8NhX0rNLBESn2yTu_z0ugJcSVLyz5iQ"

realm_key = "v53LS2JFjE-ErqJ2UWTe0O-dYxtYMUQzevxXczVVkQzcRPSS" \
    "4sdBHPaKBniqxxr7SWaQR3moXN2tzJJhJ_p5Dw"

tag_key = "751jG1gxRDN6_FdzNVWRDA"
vector_key = "v53LS2JFjE-ErqJ2UWTe0A"
cipher_key = "3ET0kuLHQRz2igZ4qsca-0lmkEd5qFzdrcySYSf6eQ8"

decrypted_data = "Attack at dawn!"
```

Author's Address

Ladar Levison
Lavabit LLC

Email: ladar@lavabit.com