           Safely Turn Authentication Credentials Into Entropy (STACIE)
                         draft-ladar-stacie-01

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.

Status of This Memo

Copyright Notice

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 an 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

In this document, the key words "MUST", "MUST NOT", "REQUIRED",
"SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY",
and "OPTIONAL" are to be interpreted as described in BCP 14, RFC 2119
[KEYWORDS] and indicate requirement levels for compliant STACIE
implementations.

3. Encodings

This document represents all of the request and responses using
standard JavaScript Object Notation [JSON]. When an object value
must always be text, the native UTF-8 representation is supplied.
Otherwise the value is armored using the base64 encoding scheme
defined in RFC 4648, 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
token derivation stages to be used for authentication purposes,
because tokens 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 a user's password must only store
the master key and the verification token. 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 means a user can
recover from an endpoint compromise by restoring the security of
their endpoint, and 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 master key and verification token.

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. If the nonce 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
   Required number of hash rounds used for the extraction and key
   derivation stages.

master_key
   The derived key value required to decrypt and use the realm
   specific keys.

password_key
   The output from the second key derivation phase, and required to
   authenticate password update requests.

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 = CalculateHashRounds(password, bonus)

# Extract the Seed
seed = ExtractEntropySeed(rounds, username, password, salt)

# Keys
master_key = HashedKeyDerivation(seed, rounds, username, password, \
  salt)
password_key = HashedKeyDerivation(master_key, rounds, username, \
  password, salt)

# Tokens
verification_token = HashedTokenDerivation(password_key, username, \
  salt)
ephemeral_login_token = HashedTokenDerivation(verification_token, \
  username, salt, nonce)

# Derive the Realm Key
realm_key = RealmKeyDerivation(master_key, realm, salt)

# Extract the Cipher and Vector Keys
vector_key = ExtractRealmVectorKey(realm_key)
tag_key = ExtractRealmTagKey(realm_key)
cipher_key = ExtractRealmCipherKey(realm_key)
```

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 the
length, and is primarily intended to offset improvements in computer
performance in the future.

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 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 should be used. The result of this calculation
is used as the "dynamic" exponent, which is used to raise the base 2,
and resulting value is the "variable" number of rounds. The "bonus"
rounds are 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 3 octet
counter employed during the entropy extraction and key derivation
stages.

Because the token derivation must be performed without leaking any
information about the password, including its length, they employ a
fixed 8 rounds.

Example

The following Python code demonstrates the proper method for deriving
the number of rounds:

```
def CalculateHashRounds(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 should be used as the HMAC key.

When the provided salt is not 128 octets, then a key must be derived
using the hash function. The 128 octet key is derived 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 by providing the password 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 ExtractEntropySeed(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 can 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:

```python
def HashedKeyDerivation(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

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:

```python
def HashedTokenDerivation(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 seperate data into
different realms based on the data type. Every realm is protected by
a unique encryption key. The realms are isolated to allow seperable
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 combinatio n 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 seperate the keys for a given realm:

```python
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."

    hashed = SHA512.new(master_key + label + salt).digest()
    realm_key = str().join(chr(operator.xor(ord(a), ord(b))) \
      for a,b in zip(hashed, shard))

    return realm_key

def ExtractRealmVectorKey(realm_key):
    vector_key = realm_key[0:16]

    return vector_key

def ExtractRealmTagKey(realm_key):
    tag_key = realm_key[16:32]

    return tag_key

def ExtractRealmCipherKey(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 should 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. Specicially, 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**

6.2. **Deep Password Change**

6.3. **Hybrid Password Change**

7. **Protocol**

7.1. **Login**

The 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 required to validate the session or connection.

7.1.1. 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" }
}
```

7.1.2. 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
specifications for the password based method for key derivation and
authentication. 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 stored secret, which results
   in a session token. Server implementations must only allow a
   single a validation attempt per nonce value.

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 higher than 1,024. If this attribute is missing, a client
   must assume a default value of 0.

The authenticate object has the following parameters:

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, and optionally implement the skein algorithm. 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] 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 required.

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" }
  ]
}
```

7.2. Authenticate

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.

7.2.1. Authenticate Request

Required Parameters

username
   The normalized username.

nonce
   A randomly generated value, which may be combined with the
   verification token to create an ephemeral login token. Every
   nonce value must only be used by 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"
    }
}
```

7.2.2. 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." }
```

7.3. Create

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: "The registration is 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" }
    }
```

7.4. Password Changes

   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 Specific Shard Values

   Fetch the realm shard values. The result may be narrowed to a
   specific realm, and serial number.

7.6. Add Realm Specific Shard Value

   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.

9. IANA Considerations

   This document has no actions for IANA.

9.1. 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].

9.2. 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
which is independent of the actual length. Clients may round any
artificial processing delays to aligned boundaries, which would also
make correlation more difficult.

9.3. 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 the 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].

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>.

[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 = "AADgUtNbxGHrQEI3hLFx6otzATOda5IeP7-a_wxJUEE" \
        "UXJ3xSwis3mph6D7iqTfJXwFQDN9gqVAdsxWw_zLC00jM"
```


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"
    realm-key = "exoUw4lFSz_RU0uTSQTM22jEdjaP-rvjvrXMbhyqNPq8o9vL" \
        "Rg9pcuKaAj_JFzQenY13XGKwxPHKULrVjrCJKQ"

    verification-token = "-Eu5mUcA7ko2BysV965hrf9bvMlh_S_iiI3tfMr" \
        "0Qc7hf4oPmBCdGOU9VCeQ1qBrga-WyR-rko5l0-feoWuuuA"
    ephemeral-login-token = "8YEH_6kBdAdR5vlBaxs3KR3pZ429bEzF3AVF" \
        "hkA0P2WPt2h94omJq-d8NhX0rNLBESn2yTu_z0ugJcSVLyz5iQ"

    tag-key = "aMR2No_6u-O-tcxuHKo0-g"
    vector-key = "exoUw4lFSz_RU0uTSQTM2w"
    cipher-key = "vKPby0YPaXLimgI_yRc0Hp2Nd1xisMTxylC61Y6wiSk"

    decrypted-data = "Attack at dawn!"
```


Author's Address

    Ladar Levison
    Lavabit LLC

    Email: ladar@lavabit.com