Safely Turn Authentication Credentials Into Entropy (STACIE)
draft-ladar-stacie-00

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 plaintext input. STACIE attempts to make discovering the plaintext input through the use of brute force more difficult by linking the amount of processing to the length of a user’s plaintext password. The shorter the plaintext password the more processing time per attempt with the amount of additional, artificially required, work scaling exponentially for each character.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on November 23, 2015.
Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ........................................... 3
2. Terminology ........................................... 4
3. Encodings ............................................. 4
4. Derivation Process ...................................... 4
   4.1. Hash Rounds ........................................ 6
   4.2. Entropy Extraction .................................. 8
   4.3. Key Derivation ..................................... 10
   4.4. Token Derivation ................................... 12
   4.5. Realm Key Derivation ............................... 13
5. Protocol ............................................... 15
   5.1. Login .............................................. 15
       5.1.1. Login Request .................................. 16
       5.1.2. Login Response ................................ 16
   5.2. Authenticate ....................................... 18
       5.2.1. Authenticate Request .......................... 18
       5.2.2. Authenticate Response ......................... 19
   5.3. Create ............................................. 20
   5.4. Password Changes .................................. 20
   5.5. Fetch Account Keys ................................. 20
   5.6. Store Account Key .................................. 20
6. Operational and Security Considerations .................... 20
   6.1. Servers ........................................... 20
   6.2. Clients ........................................... 21
   6.3. Shared ............................................ 21
7. Feedback ............................................... 22
8. Acknowledgements ....................................... 22
9. Normative References .................................... 22
Appendix A. Test Vectors .................................. 24
   A.1. User Inputs ....................................... 24
   A.2. Server Inputs ..................................... 25
   A.3. Realm Inputs ...................................... 25
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 plaintext 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 a plaintext 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 rely on JavaScript Object Notation (JSON), 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 Skein hash function (SKEIN) is 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 use of the provided 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 in development which need to safely derive a symmetric encryption key, and authenticate with a server using a shared 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 by the subsequent stage. The extraction and key derivation stages require a user’s plaintext password, while the token derivation stages do not. This allows the token derivation stages to be used for authentication purposes.

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. Only storing these values ensures a user’s plaintext password is still required to alter the account credentials, and makes a point in time recovery from compromise possible. Implementations must never store a user’s plaintext password. Client implementations on platforms which provide a secure storage facility, should make use of this capability to protect the master key and verification token if automatic login support is enabled.

*Required Inputs*

The derivation process requires the following inputs:

username
- The normalized username.

password
- The plaintext 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 using precomputed table lookups. Salt values must provide a minimum of 64 octets, and should be less than 1,024 octets. Salt values greater than 64 octets should also 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. If the nonce does exceed 64 octets, it should be aligned along a 32 octet boundary.

bonus
- The fixed number of additional iterations added to beyond what is calculated dynamically using a password’s length.

*Outputs*

rounds
- Required number of hash rounds during each key derivation stage.

master_key
The derived key 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, would be used to derive the various outputs by calling example functions provided in subsequent sections:

```python
# 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)
```

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 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 plaintext 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 calculated number of hash 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 results in the "variable" number of rounds. The "bonus" rounds are added to the "variable" number of rounds to derive the total number of rounds.

The final step in the process involves increasing the total to the minimum value of 8, if the total falls below 8. Alternatively, if the value exceeds 16,777,216 the total is reduced to this maximum value. The maximum value corresponds to the limit imposed by the use of a 3 octet counter during the entropy extraction and key derivation stages.

Token derivation must be performed without leaking any information about the password, including its length. As a result, the token derivation stages must use a fixed 8 rounds.

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

```python
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.
    characters = len(password.decode("utf-8"))

    # Identify the number of Unicode characters.
    dynamic = operator.sub(24, characters)

    # Calculate the difficulty exponent by subtracting 1
    # for each Unicode character in a password.
    dynamic = max(1, dynamic)

    # 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

# Derive the Rounds
rounds = CalculateHashRounds(password, bonus)
```

### 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 seed 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 for the SHA2-512 variation required for all implementations is also available [HMAC-SHA]. The optional Skein hash function should be used by implementations supporting this primitive, and substituted wherever the SHA2-512 function is used. This includes using an HMAC implementation based which implements Skein, and not the Skein-MAC alternative described by the Skein paper [SKEIN]. Future STACIE extensions which define alternative hash algorithms should also specify whether HMAC must be used for seed extraction, or some alternative method.

Unlike a simple hash, HMAC requires a 128 octet key value 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 salt value is not 128 octets, then a key must be derived using the hash function. The key is derived by digesting the salt value concatenated 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 outputs must be concatenated, which then forms the precise 128 octet value needed as an HMAC key.

The HMAC primitive also requires a "message" which is created using the plaintext password, provided repeatedly, with the number of repetitions dictated by the "rounds" variable. The digest produced by the HMAC function becomes a 64 octet seed value, and a required input for the master key derivation stage.

*Example*

The following Python code demonstrates the proper method for extracting the entropy seed value:

```python
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 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()
else:
    key = salt

# Initialize the HMAC instance using the key created above.
hmac = HMAC(key, None, SHA512)

# Repeat the plaintext 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

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

### 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 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 previous 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 confirms the value translates into the verification token it has stored in its authentication database.

Each key derivation stage repeats the hash process by the variable number of iterations dictated by the rounds variable. Provided the hash function remains one-way, this 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.

Hashes are generated by digesting the input seed generated in the previous stage, concatenated together 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 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

# Key Derivation Stages
master_key = HashedKeyDerivation(seed, rounds, username, password, \
    salt)
password_key = HashedKeyDerivation(master_key, rounds, username, \
    password, salt)

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 process uses the password key to derive the verification token shared with a server during account creation or during password updates. The process is repeated using the verification token instead of the password key, and in combination with a nonce value, which generates the ephemeral login tokens required to authenticate a session or connection.

Like the key derivation stages defined above, the seed value 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, if deriving an ephemeral token. A counter value is also appended, with value representing a 3 octet big endian integer value, and corresponding to a 0 based count of the current round. The output of 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 key derivation, with the seed value either the extracted seed, or the master key, 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
```

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

4.5. Realm Key Derivation

Realm specific keys are used to access encrypted user data. The realm label specifies the category and/or type of data protected by a given key. Protocols which incorporate STACIE may use a single realm, or separate data into different realms. Each realm is protected by a unique encryption key.
The unique key for a realm is derived by hasing the concatenated master key, realm label, and the realm shard. The resulting hash is then combined with the shard value using a bitwise exclusive "or" operation. The result is a "realm key" containing the initialization vector and the cipher key. The vector key is the produced by combining the first 16 octets with the subsequent 16 octets using a bitwise exclusive "or" operation. The last 32 octets provide the cipher key.

*Required Inputs*

The master key, as previously described, is combined with the following required inputs:

realm
The category and/or type of data.

shard
A non-secret fragment required to derive the key associated with a given realm.

*Outputs*

realm_key
The realm specific key distilled from the provided inputs, it holds both the vector and cipher keys.

realm_vector_key
The key used to unlock the initialization vectors for a given realm.

realm_cipher_key
The key used by the symmetric cipher to decrypt user data associated with a given realm.

*Example*

The following code, written in Python, would be used to derive keys for a given realm:
def RealmKeyDerivation(master_key, realm="", shard=""):  
    if len(realm) < 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.")  
    
    hashed = SHA512.new(master_key + realm + shard).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):  
    realm_vector_key = str().join(chr(operator.xor(ord(a), ord(b))) \  
    for a,b in zip(realm_key[0:16], realm_key[16:32]))  
    return realm_vector_key

def ExtractRealmCipherKey(realm_key):  
    realm_cipher_key = realm_key[32:64]  
    return realm_cipher_key

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

# Extract the Cipher and Vector Keys  
realm_vector_key = ExtractRealmVectorKey(realm_key)  
realm_cipher_key = ExtractRealmCipherKey(realm_key)

5. Protocol

5.1. 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.
5.1.1. 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*

```json
{ login:
  { username: "user-alias@example.tld" } }
}
```

5.1.2. 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 Advanced Encryption Standard [AES] is being used in the Galois Counter Mode [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. Support for the "chacha" primitive is optional. If provided, the use of poly1305 as the authenticator is implied, and the returned cipher object may contain parameters specifying the key and nonce sizes in bits, with default values of 256 bits, and 64 bits respectively.

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": "YKtlO8W2Fp5jujGbds9uJdbFY9CUAr7t1dnZcAcQjb/BYNX4BAynRFd7uAbdnByfY9Wz075+RG5h6xBBAr",
      "nonce": "Qe7K0GarZRsBlHeSN9383LcOln6adOdx12jzDwus+aeOkHWEsqFZESc6BFi7noOBpqhJdhChvWh6leY12w9",
      "bonus": "8",
      "hash": "sha2",
      "cipher": "aes"
      "disposition": "required"
    }
  ]
}
```

5.2. 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.

5.2.1. 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*

```json
authenticate:
  { username: "user@example.tld",
    nonce: Qe7K0GarZrMbl8SN938Lc0L6adOddxi12jzDwus+aeOkHEesqtFZE
          Sc6f7I7noOPqvh1phCnvWh61eYI2w9,
    token: 5Kcju+GZtCpyz81x4fKL4o/Klw/05epHBp+Ylpyo7R1bmrm2EkrRTCD
           CVw5wrWc9CRK8r5RsL+H0EwnWGu1N
  }
}
```

5.2.2. Authenticate Response

*Required Parameters*

realm
A protocol specific string containing the realm where the key value is used.

key
The random bytes which are combined with the master key to derive a realm specific key value.

index
The an incrementing counter corresponding to each key value.

*Example*

```json
{ keys: [
  { "realm": "example"
    "key": "8f_7y-rPUZsN11vryzDM_U4CRp2PZux42AeMGNF1_GkvR4fuB6c6"
    "rqYmZl8F_C1HXoalXoW-1llto5Sw5w"
    "index": "1"
  }
]
}
5.3. Create

When the birds mate with the bees a new account is born.

5.4. Password Changes

Update the verification key on the server. And alter the realm specific values in the account key ring.

5.5. Fetch Account Keys

Grab a copy of all the account keys. Provides the same data returned after a successful login, but can also be narrowed to a specific realm.

5.6. Store Account Key

Save a new account key in the next available sequence number for a given realm.

6. Operational and Security Considerations

Client and server implementations should follow the recommendations provided here to avoid leakage, and improve difficulty.

6.1. 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].
6.2. 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 plaintext 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.

6.3. 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].
7. Feedback

The preceding document was excreted with the assistance of a diarrhoetic. As such, feedback is both welcome, and encouraged.

8. Acknowledgements

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.

9. Normative References
National Institute of Standards and Technology, "Advanced Encryption Standard (AES), FIPS 197", November 2001,


Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, SP 800-38D", November 2007,

Krawczyk, H., "Cryptographic Extraction and Key Derivation: The HKDF Scheme", May 2010,

Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", February 1997,


Nystrom, M., "Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512", December 2005,

Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", March 2014,

Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", March 1997,

Boulange, R., "One Password Protocol", May 2014,
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. User Inputs

    password = "password"
    username = "user@example.tld"
A.2. Server Inputs

bonus: 131072
salt: HQp0A0l4IzkyplLVY8Cnp03-D67E2bk04WDqNOiSzIMNbmCjGlMR
KxBh9UV5t1gXRyqYsFJWns0hwvlwA"
nonce: sJWAhD50ulipapa63FE4dGI-W3PDACaQtA49vQBOG9_UYhgNMz
mLuSEBEQy15Iv2Wn_lvSizRmKfky51Fpp7Q

A.3. Realm Inputs

realm: mail
shard: g065Kdeda1hB26gdZl0fetGg2viLWXG0vmKN4HxE3Jp3Z
0Gkt5prqSmcuY2o8t24iGSCOnFDpP71c3x19NXQ

A.4. Outputs

rounds: 196608
seed: ZpkPVoGLJcsWIhXNL0fNHlJ8RZhh6cvbJe6-0PJ5nxVFSqIKGEih
FcyYACbj05kTe0Tk4op8Wwuz51fy9pUg
master-key: ydoBTjnDlFZTajWUvvGDmi-ICgJbKkhXanx6202TCVW0LY
RcqIn4Qkb1GK3mQfTLKp3iC5iW4qWE54W3b7Q
password-key: irmdngy1NiNZX0QfjeCjPgcAv3SeneP7BDP68fE4rih
F1hT05L1_TtbZ0bpH1xTWncbH9QNdLZ7vM_SqZmg
verification-token: 3biXbu28SNKDLgnz80FOR2IYAtDKxpBGpM_Cmq4
1bfemvYIDXuhi8BQsJjXEqb3-opUt9_dKQzyGHcysSK7-g
ephemeral-login-token: s2mdosElo0VemczsnfwMWacJsV-pa6HmfA9
ZveSMUJmOKyozGc0BJ551LT-Jcu7pYKdhyZVgOHzTJp7yg
realm-vector-key: UpbIOSVXWoFAL0_D7u-Yeg
realm-cipher-key: FOobztKrYXOMnTTjvl1307gT---1WMB07v308FbheNzU

Author’s Address

Ladar Levison
Lavabit LLC

Email: ladar@lavabit.com