Individual Submission
Internet-Draft
Intended status: Informational
Expires: 14 May 2026

M. Vieuille

10 November 2025

KerPass EPHEMSEC One-Time Password Algorithm
draft-mvieuille-kerpass-ephemsec-02

Abstract

This document specifies EPHEMSEC, an algorithm for generating one-time passwords (OTPs) and one-time keys (OTKs). Unlike traditional OTP algorithms that rely solely on static shared secrets, EPHEMSEC uses public key cryptography, which simplifies secure deployment on authentication servers. EPHEMSEC also supports binding the generated OTP/OTK to contextual data. When this context is obtained and injected by a trusted agent, the resulting codes can be made resistant to phishing and man-in-the-middle attacks. Finally, EPHEMSEC includes a built-in time synchronization mechanism that embeds a synchronization hint into the generated output. This allows both parties to deterministically derive the same OTP/OTK value without requiring trial-and-error validation, enabling compatibility with protocols such as Password Authenticated Key Exchange (PAKE) and TLS with Pre-Shared Key (TLS-PSK) that require exact secret agreement.

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 https://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 14 May 2026.

Copyright Notice

Table of Contents

1. Introduction

The concept of one-time passwords has existed for decades. Proprietary systems such as RSA SecurID have been available since the early days of the Internet and have demonstrated the viability of time-based and event-based one-time passwords. HOTP (specified in [RFC4226]) and TOTP (specified in [RFC6238]) are open algorithms that enable the implementation of one-time password generators operating similarly to earlier proprietary systems. These algorithms now form the foundation of various hardware tokens and numerous smartphone authenticator applications that provide a second authentication factor for high-profile web applications.

While OTP authenticator applications are simple to use and widely available, they have had limited impact on web application security. Two factors contribute to this limited success:

1. Deployment of corresponding authentication servers introduces additional risks, as stored credentials can be used to impersonate users if compromised. This problem is particularly serious in today’s cloud environments.

2. The design of traditional authenticator applications does not address specific security challenges of web applications. In particular, modern web browsers do not provide a trusted input interface for entering authentication data, making users vulnerable to phishing attacks that capture credentials.

These problems are the main reasons explaining the lack of attractiveness of one-time passwords for securing web applications. In the current technological ecosystem, one-time password solutions are seen as obsolete, and the current trend is toward using authentication credentials such as [PASSKEY], which address problems 1 and 2 by relying on digital signature key pairs.

The KerPass EPHEMSEC algorithm also addresses problems 1 and 2 but takes a meaningfully different approach than digital signatures. The EPHEMSEC client credential contains a Diffie-Hellman key pair, and the server stores only the public part of this key pair. To obtain a new OTP from EPHEMSEC, a trusted agent obtains the server's ephemeral public key and acquires context information such as the login page URL. This information is passed to the EPHEMSEC client (also known as the Responder; see Section 4.2) in a challenge message. The client first calculates a Diffie-Hellman secret using the received public key and local key pair, then uses this ephemeral secret key to generate an OTP using an algorithm based on principles similar to the HOTP algorithm described in [RFC4226]. This simplified description excludes many important details but should be sufficient to provide an understanding of how public key cryptography is used in EPHEMSEC.

Assuming that both EPHEMSEC OTP and digital signature solutions like [PASSKEY] can solve the server credential storage issue described in problem 1 and the phishing issue described in problem 2 (by having their output depend upon context input provided by a trusted agent), are there differences between the two that could help determine which option is better in a specific context?

One advantage that an OTP authenticator application has over a digital signature authenticator application is simpler integration, resulting from the smaller size of OTPs compared to digital signatures. Integration of smartphone interfaces such as NFC and Bluetooth to computing devices may eventually lessen this advantage, and EPHEMSEC can also generate one-time keys (OTK) that would benefit from the ubiquity of such interfaces.

The main difference between OTP/OTK and digital signature credentials is that OTP/OTK are ephemeral *shared* secrets known by both peers involved in an authentication session, whereas digital signatures can only be generated by the peer controlling the private key. There are contexts such as transaction validation in which the one-way nature of digital signatures is an advantage. However, shared secrets have the distinctive advantage that they can be used as the main credential with protocols like Password Authenticated Key Exchange (PAKE) or TLS-PSK that enable *mutual* authentication of the peers involved in an authentication session. The potential to use EPHEMSEC OTP/OTK for mutual authentication is the distinctive feature of this solution.

OTPs generated by HOTP/TOTP algorithms are also ephemeral shared secrets, but synchronization problems make their integration with PAKE protocols complex (see Section 12). EPHEMSEC has a built-in auto-synchronization feature (see Section 8.2) that addresses this issue.

This document specifies the KerPass EPHEMSEC algorithm, which addresses the fundamental limitations of HOTP/TOTP that have contributed to the low attractivity of OTP authenticator applications for securing web applications. EPHEMSEC solves the server credential storage security problem by using credentials based on public key cryptography, provides a context binding mechanism that enables the generation of OTPs and OTKs resistant to phishing and man-in-the-middle attacks, and produces shared secrets that can be used with mutual authentication protocols such as PAKE and TLS-PSK. Given the ease of implementing OTP/OTK authenticator applications on smartphones, an OTP/OTK generation algorithm with properties similar to EPHEMSEC may be valuable across a wide range of use cases.

2. Requirements Notation

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

3. Functions and Notation

This section defines notation and helper functions used throughout the specification.

* ||: Denotes byte string concatenation.

* byte(v): Converts an integer or single-character input v to a one-byte string.

* size(bs): Returns the length of byte string bs, as an integer.

* BE8(v): Returns the 8-byte big-endian encoding of unsigned integer v.

* U64(bs): Converts an 8-byte string bs into an unsigned 64-bit integer, using big-endian interpretation.

* TLV(tag, bs): Constructs a Tag-Length-Value encoding. Returns byte(tag) || byte(size(bs)) || bs where:

  - tag: An integer identifier in the range 0255

  - bs: A byte string with length 255 bytes

4. EPHEMSEC Roles

EPHEMSEC distinguishes two roles: Initiator & Responder.

4.1. Initiator Role

This role is normally assigned to relying web application.

4.2. Responder Role

This role is normally assigned to authenticator application.

5. EPHEMSEC Parametrization

Each EPHEMSEC protocol instance is characterized by its selection of cryptographic primitives and operational parameters. These include:

* A secure hash function,
* An elliptic-curve Diffie-Hellman (ECDH) function,
* A key exchange pattern defining key contributions from the parties,
* Code output parameters specifying how the OTP/OTK values are formatted.

5.1. Hash Function

The hash function MUST be cryptographically secure (e.g., SHA-512) with digest length between 32 and 64 bytes. . This hash function is used to instantiate the HKDF key derivation function, as specified in [RFC5869].

5.1.1. HKDF(salt, ikm, info, L) → byte[L] Function

This function follows the specification in [RFC5869].

* salt, ikm, and info are arbitrary-length byte strings.
* L is a positive integer indicating the desired output length in bytes.
* The function returns a byte string of length L.

5.2. ECDH(privKey, PubKey) → byte[DHLEN] Function

This function performs an Elliptic-Curve Diffie-Hellman (ECDH) key agreement over a specified elliptic curve group, producing a shared secret of fixed length DHLEN.

* privKey: A private key from an elliptic curve key pair.

* PubKey: A public key from the same elliptic curve group.

The function computes a shared secret using the standard ECDH primitive associated with the curve in use. The result is a byte string of length DHLEN. The specific behavior, including internal encoding and validation, is determined by the curve and cryptographic library in use.

An example of a suitable ECDH function is X25519, as defined in [RFC7748], which yields a 32-byte shared secret.

5.3. Key Exchange Pattern

The key exchange pattern determines which types of keys (ephemeral or static) are contributed by the Initiator and Responder during the ECDH exchanges.

Note that in each of the pattern below, the Initiator always contribute an ephemeral key and the Responder always contribute a static key.

5.3.1. E1S1 Pattern

* Initiator contributes: 1 ephemeral key.

* Responder contributes: 1 static key.

5.3.2. E1S2 Pattern

* Initiator contributes: 1 ephemeral key and 1 static key.

* Responder contributes: 1 static key.

5.3.3. E2S2 Pattern

* Initiator contributes: 1 ephemeral key and 1 static key.

* Responder contributes: 1 ephemeral key and 1 static key.

5.3.4. Rationale for Selecting a Key Exchange Pattern

The choice of key exchange pattern should align with the authentication guarantees provided by the method used to validate the generated OTP or OTK.

If the selected authentication method provides only one-way authentication— verifying the Responder to the Initiator— then there is no benefit in selecting a pattern more complex than E1S1. Patterns such as E1S2 or E2S2 introduce additional computational and communication overhead without improving the resulting security.

If the authentication method provides mutual authentication— for example, using PAKE or TLS-PSK— then the use of E1S2 or E2S2 becomes appropriate. These patterns ensure that both the Initiator and the Responder contribute static keys to the Diffie-Hellman exchange, reinforcing the mutual nature of the derived secret.

5.4. Code Output

The format and characteristics of the OTP/OTK values produced by EPHEMSEC are determined by the parameters T, B, and P.

5.4.1. T Time Window

T defines the duration of a validity window for each generated code, expressed in seconds. T MUST be a positive integer greater than parameter B.

5.4.2. B Encoding Base

B specifies the base used for code encoding. Valid values are:

* 10, 16, or 32: Encodes the output using a human-readable digit set, suitable for OTP.

* 256: Encodes the output in binary form, suitable for OTK.

5.4.3. P Code Size

P defines the number of digits in the generated code, where each digit is an integer in the range [0..B). The valid range of P depends on the encoding base B and the minimum required entropy.

The table below summarizes the minimum and maximum allowed values of P for each supported base, along with the corresponding entropy range.

Base	min P	max P	min Entropy	max Entropy
10	8	15	23 bits	46 bits
16	7	17	24 bits	64 bits
32	6	13	25 bits	60 bits
256	4	65	24 bits	512 bits

Table 1: Code limits

5.5. Naming Scheme

EPHEMSEC uses a structured naming scheme to identify specific protocol instantiations. A valid EPHEMSEC name has the following format:

Kerpass_<Hash>_<ECDH>_<Pattern>_T<T>B<B>P<P>

For example:

Kerpass_SHA512_X25519_E1S1_T600B32P9

Where:

1. <Hash> corresponds to the hash function used (see Section 5.1), e.g., SHA512.

2. <ECDH> denotes the ECDH function used (see Section 5.2), e.g., X25519.

3. <Pattern> specifies the key exchange pattern (see Section 5.3), e.g., E1S1.

4. T<T>B<B>P<P> encodes the code output parameters (see Section 5.4), with <T>, <B>, and <P> replaced by their respective values.

6. EPHEMSEC Credentials

EPHEMSEC requires an *enrollment phase* during which the Responder (typically the authenticator app) registers an account with the Initiator (the relying web application). Implementations may support various enrollment scenarios depending on their operational context and security requirements.

This specification does not define a particular enrollment protocol, but it assumes the following outcomes:

* The Responder registers a static ECDH public key with the Initiator, corresponding to a key pair it controls.

* Both parties derive and retain a shared secret (PSK) that will be used in subsequent EPHEMSEC operations.

6.1. Shared PSK

The *pre-shared key (PSK)* is a byte string shared between the Responder and Initiator. It is established during the enrollment process and MUST be at least 32 bytes in length. The PSK is stored by both parties.

6.2. Responder Static Key

During enrollment, the Responder generates a static ECDH key pair and transmits the corresponding public key to the Initiator. The Initiator stores this public key as part of the Responder's account data.

6.3. Initiator Static Key

The Initiator is required to maintain a static ECDH key pair only when using key exchange patterns that require an Initiator static key (e.g., E1S2 or E2S2). The mechanism by which the Responder obtains and establishes trust in the Initiator's static public key is out of scope for this specification.

7. EPHEMSEC Protocol Overview

This section outlines the overall flow of a complete EPHEMSEC OTP/OTK generation exchange between an Initiator and a Responder.

It assumes the following preconditions:

* The two parties have agreed on an EPHEMSEC instantiation (see Section 5).

* The Responder has registered an account with the Initiator, including a shared PSK and necessary public key material (see Section 6).

The protocol operates as a Challenge/Response exchange. The Initiator sends a message to the Responder containing:

* A CONTEXT byte string (e.g., relying party information),

* A freshly generated INONCE,

* Its Diffie-Hellman public key(s) as required by the selected key exchange pattern.

  Note: The structure and transport of this message are out of scope for this specification.

Upon receiving this challenge, the Responder performs the following steps to compute the OTP or OTK:

1. *Obtain nonces* (see Section 8):

   * Read or receive INONCE from the Initiator,

   * Generate PTIME and extract SYNCHINT.

2. *Compute the Diffie-Hellman shared secret Z* (see Section 9): Based on the agreed key exchange pattern and available keys.

3. *Derive the intermediary secret ISK* (see Section 10): Using HKDF with Z, PSK, CONTEXT, SCHEME, INONCE, and PTIME.

4. *Generate the OTP or OTK* (see Section 11):

   * The format is determined by the encoding base B,

   * The last digit or byte encodes SYNCHINT.

The Responder returns the resulting code to the Initiator.

The Initiator, upon receiving the response, uses the included SYNCHINT to reconstruct PTIME and repeat the same derivation steps to validate or use the resulting code.

8. Nonce Acquisition

Each EPHEMSEC session uses two distinct nonces contributed independently by the two parties involved:

* The *Initiator* provides a nonce called INONCE, which ensures session uniqueness from the Initiator’s side.

* The *Responder* provides a time-based nonce called PTIME, which captures the Responder’ s local clock state.

These nonces serve as independent inputs to the intermediary secret derivation process (see Section 10).

8.1. INONCE Initiator Nonce

The Initiator generates a nonce INONCE that contributes to the personalization of the derived intermediary secret (see Section 10). This value acts as an Initiator-specific input to ensure uniqueness of each EPHEMSEC execution.

INONCE MUST be a byte string of length between 16 and 64 bytes. It MUST be unique for each run of the EPHEMSEC algorithm, and MUST NOT be reused across authentication sessions.

The value of INONCE is transmitted from the Initiator to the Responder as part of the authentication request.

8.2. PTIME Responder Time Nonce

The EPHEMSEC Responder derives a pseudo-time value, PTIME, from current time reading. This PTIME acts as a Responder contributed nonce and is used in secret derivation along with an Initiator-contributed nonce.

The challenge lies in enabling the Initiator to reconstruct the same PTIME value computed by the Responder, despite clock skew between the two parties. To address this, the Responder includes a *synchronization hint*, SYNCHINT, in the last digit of the generated OTP or OTK.

Given SYNCHINT, the Initiator can recover the original PTIME as long as clock drift remains within acceptable bounds.

8.2.1. Inputs

The following parameters are used throughout this section:

* time: Current Unix timestamp (seconds since 1970-01-01).
* T: Code validity window (see Section 5.4.1).
* B: Encoding base (see Section 5.4.2).

8.2.2. Responder Function PTime(time) → (PTIME, SYNCHINT)

This function is executed by the Responder to compute the PTIME nonce and the associated synchronization hint:

```
step = T / (B - 1) # floating point division
PTIME = round(time / step)
SYNCHINT = PTIME % B
return PTIME, SYNCHINT
```

8.2.3. Initiator Function SyncPTime(time, SYNCHINT) → PTIME

This function is executed by the Initiator to reconstruct the Responder' s PTIME using its local time and the received SYNCHINT:

```
mintime = time - (T / 2)
step = T / (B - 1) # floating point division
mpt = round(mintime / step)
mphint = mpt % B

Q = mpt // B # integer division
PTIME = Q * B + SYNCHINT

if SYNCHINT < mphint:
    PTIME += B

return PTIME
```

This algorithm works correctly if the clock difference between the Responder and Initiator is less than T / 2. Outside this range, synchronization will fail, resulting in mismatched secrets.

KerPass uses a 600-second time window, allowing up to ±5 minutes clock drift in between Initiator and Responder.

9. Z - Diffie-Hellman Secret Derivation

Each party derives a shared secret Z using the Diffie-Hellman key exchange, based on the agreed EPHEMSEC key exchange pattern (see Section 5.3). Key material is retrieved from received protocol messages and account credential storage.

The result of the Diffie-Hellman exchange is a byte string Z, which is used as part of the key derivation input (see later sections).

Where ephemeral key pairs are used, they MUST be freshly generated for each execution of the EPHEMSEC protocol and MUST NOT be reused across sessions.

EPHEMSEC execution MUST be aborted if any required key is missing or invalid.

9.1. Initiator E1S1 Z Derivation

Inputs:

* ei Initiator ephemeral Keypair
* Sr Responder static PublicKey

Z = ECDH(ei, Sr)

9.2. Responder E1S1 Z Derivation

Inputs:

* sr Responder static Keypair
* Ei Initiator ephemeral PublicKey

Z = ECDH(sr, Ei)

9.3. Initiator E1S2 Z Derivation

Inputs:

* ei Initiator ephemeral Keypair
* si Initiator static Keypair
* Sr Responder static PublicKey

Z = ECDH(ei, Sr) || ECDH(si, Sr)

9.4. Responder E1S2 Z Derivation

Inputs:

* sr Responder static Keypair
* Ei Initiator ephemeral PublicKey
* Si Initiator static PublicKey

Z = ECDH(sr, Ei) || ECDH(sr, Si)

9.5. Initiator E2S2 Z Derivation

Inputs:

* ei Initiator ephemeral Keypair

* si Initiator static Keypair

* Er Responder ephemeral PublicKey

* Sr Responder static PublicKey

Z = ECDH(ei, Er) || ECDH(si, Sr)

9.6. Responder E2S2 Z Derivation

Inputs:

* er Responder ephemeral Keypair

* sr Responder static Keypair

* Ei Initiator ephemeral PublicKey

* Si Initiator static PublicKey

Z = ECDH(er, Ei) || ECDH(sr, Si)

10. 10. ISK Intermediary Secret Derivation

EPHEMSEC derives an intermediary secret key ISK using the HKDF function (see Section 5.1.1).

10.1. 10.1 Inputs

The function uses the following inputs:

* CONTEXT: An implementation-specific byte string (64 bytes), used to encode contextual information (e.g., login page url).

* SCHEME: A byte string representing the EPHEMSEC instantiation (see Section 5.5).

* B: Code encoding base (see Section 5.4.2).

* P: Code size (see Section 5.4.3).

* INONCE: An Initiator-generated nonce, a byte string between 16 and 64 bytes (see see Section 8.1).

* PTIME: Responder-contributed time nonce (see Section 8.2).

* PSK: A shared pre-established secret (32 bytes) (see Section 6.1).

* Z: Diffie-Hellman shared secret derived from the selected key exchange pattern (see see Section 9).

10.2. 10.2 ISK Derivation

The ISK is derived using the following steps:

```
# CONTEXT & SCHEME are used for domain separation
salt = TLV(byte('C'), CONTEXT) || TLV(byte('S'), SCHEME)

ikm = Z || PSK

# INONCE & PTIME are used for output personalization
info = TLV(byte('N'), INONCE) || TLV(byte('T'), BE8(PTIME))

# Output length
if B == 256:
  L = P - 1 # OTK case
else:
  L = 8 # OTP case

ISK = HKDF(salt, ikm, info, L)

return ISK
```

11. OTP/OTK Derivation

The intermediate secret key ISK computed in Section 10 serves as the final source of entropy for generating the OTP (one-time password) or OTK (one-time key). The output format depends on the encoding base B (see Section 5.4.2).

11.1. Inputs

* B: Code encoding base (see Section 5.4.2).

* P: Code size (see Section 5.4.3).

* PTIME: Responder-contributed time nonce (see Section 8.2).

* ISK: Intermediate secret key (see Section 10).

11.2. OTP Derivation (B ∈ {10, 16, 32})

When B is 10, 16, or 32, the code is formatted as an OTP composed of P digits. The first P - 1 digits are derived from ISK, and the last digit is a synchronization hint (SYNCHINT) derived from PTIME.

ISK MUST be exactly 8 bytes long and is interpreted as an unsigned 64-bit integer.

```
# Interpret ISK as a big-endian unsigned integer
maxcode = B ^ (P - 1)
isrc = U64(ISK) % maxcode

# Extract (P - 1) digits in base B
OTP = '' # empty byte string
for i in 0 .. (P - 2):
    digit = byte(isrc % B)
    OTP = digit || OTP
    isrc /= B

# Append 1-digit time synchronization hint
SYNCHINT = byte(PTIME % B)
OTP = OTP || SYNCHINT

return OTP  # byte string of P digits in [0 .. B)
```

Note: The result is returned as a sequence of P integer digits in base B.

Conversion to a human-readable representation (for example, an alphanumeric alphabet) is outside the scope of this specification. Nevertheless, the use of the [Crockford-B32] alphabet is RECOMMENDED, as it is applicable to all EPHEMSEC OTP schemes and has been designed to minimize transcription and input errors.

11.3. OTK Derivation (B = 256)

When B is 256, the output is an opaque binary key. The first P - 1 bytes are taken directly from ISK, and the last byte encodes the synchronization hint.

```
SYNCHINT = byte(PTIME % 256)
OTK = ISK || SYNCHINT

return OTK  # byte string of length P
```

12. Key Exchange Protocol Integration

EPHEMSEC OTPs/OTKs are ephemeral *shared secrets* that can serve as primary credentials in mutually authenticated key exchange protocols, such as:

* Password-Authenticated Key Exchange (PAKE)

* TLS with Pre-Shared Key authentication (TLS-PSK)

Traditional one-time password algorithms like HOTP, as defined in [RFC4226], are unsuitable for these protocols due to their reliance on loose synchronization. Validation servers must compare a received OTP against a range of possible values, which precludes direct use as cryptographic key material.

EPHEMSEC addresses this limitation by appending a *synchronization digit* to each OTP/OTK. This digit enables reconstruction of the time-based nonce PTIME (see Section 8.2), ensuring that both parties derive identical secrets without relying on trial-and-error validation.

To use EPHEMSEC outputs as inputs to a key exchange protocol:

1. *Client Preparation*:

    * Append the synchronization digit to the account identifier.

    * Use this composite identifier to initiate the key exchange protocol.

2. *Server Operation*:

    * Extract the synchronization digit from the received identifier.

    * Remove the synchronization digit to recover the base account identifier.

    * Load the corresponding client credentials using the base identifier.

    * Execute the EPHEMSEC algorithm with the received session parameters, credentials, and synchronization hint to derive the shared secret.

    * Proceed with the key exchange protocol using the derived secret.

13. Security considerations

This section outlines the security properties of the EPHEMSEC algorithm. The analysis presented here is intended to demonstrate why the protocol is expected to meet its security goals, based on widely accepted cryptographic assumptions.

This document has not yet undergone comprehensive peer review by the cryptographic and security communities. The security analysis presented should be considered preliminary, and implementers should exercise appropriate caution in security-critical deployments pending further review.

13.1. Adversary Profiles

Two adversary profiles are considered in this analysis:

* *Network Observer*
  This adversary has access to all publicly visible data, including the Initiator's and Responder's static ECDH public keys, session inputs (nonces, ephemeral public keys), and outputs (OTPs/OTKs) from previous EPHEMSEC sessions. It does not have access to any party's private credentials.

* *Credential Leak Attacker*
  This adversary has all the capabilities of a Network Observer, and additionally has read access to the Initiator's credential store — specifically the Responder's shared PSKs and static public keys. It does *not* have access to the Initiator's private keys or control over protocol behavior.

13.2. Output Unpredictability of EPHEMSEC

The primary security goal of an OTP/OTK algorithm is to ensure that outputs are unpredictable — even to attackers with significant passive or partial access.

EPHEMSEC achieves this by combining Diffie-Hellman key exchange and HKDF with session-specific nonces. Specifically:

1. *Unpredictability of the Shared Secret (Z)*
   The ECDH-derived shared secret Z is indistinguishable from random to any adversary lacking private keys, due to the hardness of the Decisional Diffie-Hellman (DDH) problem on the chosen curve.

2. *Unpredictability of the HKDF Input (ikm)*
   The HKDF input key material (ikm = Z || PSK) inherits the unpredictability of Z. Even if the attacker knows the pre-shared key (PSK), the presence of the fresh and secret Z value ensures ikm remains secure.

3. *Security of the Derived Secret (ISK)*
   HKDF acts as a cryptographically strong pseudorandom function (PRF), meaning its outputs — including the intermediary secret ISK — are indistinguishable from random, provided the HKDF ikm secret input is unpredictable. Because each execution of EPHEMSEC uses unique nonces and ephemeral keys, the ISK value changes with every session.

4. *Output Derivation (OTP/OTK)*

   * For OTKs (B = 256): ISK is directly used, preserving its pseudorandomness.

   * For OTPs (B ∈ {10, 16, 32}): ISK is converted to digits via modular arithmetic. This process introduces bias when B = 10, but the bias is mitigated by restricting code sizes (P) to the ranges specified in Table 1 (see Appendix A for analysis).

5. *Prevention of Replay and Forward Prediction*
   The use of unique nonces (INONCE, PTIME) and ephemeral keys ensures that no two executions produce the same output — even for the same account and context. This prevents replay attacks and ensures that attackers cannot predict future codes based on prior sessions.

As a result, even an attacker who observes multiple sessions (and even possesses some server-side credentials) cannot derive or guess new OTPs or OTKs, nor can they reuse prior ones.

13.3. Phishing and MITM Prevention via Context Binding

EPHEMSEC can mitigate web browser phishing and man-in-the-middle (MITM) attacks by binding cryptographic outputs to authentication context through the CONTEXT input (see Section 10.1). This mechanism enables:

1. *Phishing Resistance*:

   * By embedding the login page URL in CONTEXT, the derived OTP/OTK becomes domain-specific.

   * An attacker hosting a fake page cannot reuse intercepted codes, as their context will differ.

2. *MITM Resistance*:

   * Including the server’s TLS certificate hash in CONTEXT ensures the OTP/OTK is tied to the authenticated connection.

   * A MITM with an invalid certificate cannot generate valid codes.

13.3.1. The need for a trusted Agent

Context binding alone is *insufficient* to provide Phishing or MITM resistance.

For Context binding to be efficient, it must be used jointly with a *trusted Agent* (e.g., a browser extension) that:

* Reliably acquire authentication context (e.g., page URL, certificate)

* Securely inject it into EPHEMSEC’s CONTEXT parameter

* Resist spoofing or coercion by attackers

13.4. Mutual Authentication

Because the EPHEMSEC Initiator and Responder share a PSK (see Section 6.1), all OTP/OTK outputs are derived from a secret known to both parties. As a result, these values can serve as credentials in protocols that support mutual authentication, such as PAKE or TLS-PSK.

However, when the E1S1 key exchange pattern is used, the only contribution from the Initiator is an ephemeral key. In this configuration, the mutuality of the shared secret relies solely on the PSK. If the PSK is compromised — for example, by a Credential Leak Attacker — the attacker can impersonate the Initiator to the Responder.

To mitigate this risk, it is RECOMMENDED that mutual authentication deployments use the E1S2 or E2S2 key exchange patterns. These patterns require the Initiator to contribute a static ECDH key, ensuring that mutual authentication depends on key material not accessible to a Credential Leak Attacker. This prevents such an attacker from successfully impersonating the Initiator.

13.5. Time Synchronization Attacks

EPHEMSEC's time-synchronized nature creates a potential attack vector against the protocol's availability. The algorithm requires that clock drift between Initiator and Responder remain within T/2 seconds for successful PTIME recovery (see Section 8.2).

An attacker who can manipulate the time sources or time synchronization mechanisms of either party may cause authentication failures by forcing clock drift to exceed this threshold. Such attacks could result in denial of service against the authentication system.

Future revisions will extend PTIME derivation to support event-based counters alongside time-based synchronization. Setting T = 0 will enable event-synchronized OTP/OTK generation using shared counters, while T > 0 will maintain time-based operation, allowing applications to choose the synchronization method best suited to their threat model.

14. IANA Considerations

No IANA action is required.

15. References

15.1. Normative References

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/rfc/rfc7748>.

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

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

15.2. Informative References

[RFC4226] M'Raihi, D., Bellare, M., Hoornaert, F., Naccache, D., and O. Ranen, "HOTP: An HMAC-Based One-Time Password Algorithm", RFC 4226, DOI 10.17487/RFC4226, December 2005, <https://www.rfc-editor.org/rfc/rfc4226>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/rfc/rfc5869>.

[RFC6238] M'Raihi, D., Machani, S., Pei, M., and J. Rydell, "TOTP: Time-Based One-Time Password Algorithm", RFC 6238, DOI 10.17487/RFC6238, May 2011, <https://www.rfc-editor.org/rfc/rfc6238>.

[KerPass] "KerPass open source project", n.d., <https://github.com/amvtek/KerPass>.

[PASSKEY] "passkeys-Specifications", n.d., <https://passkeys.dev/docs/reference/specs/>.

[Crockford-B32] "Crockford Base 32 Alphabet", n.d., <https://www.crockford.com/base32.html>.

Appendix A. OTP Sampling Bias

EPHEMSEC generates P (see Section 5.4.3) digits OTPs from 64-bit pseudo-random integers as described in Section 11. This final operation may introduce a non-uniform distribution in the generated OTP when the code base B (see Section 5.4.2) does not evenly divide 2^64, which occurs for all bases except B = 10.

We apply the Bias Formula below to evaluate the bias for code base 10.

The bias reaches its maximum when the code size P is at its maximum value. While EPHEMSEC supports a maximum P of 15, only 14 digits are used to encode the pseudo-random value. Therefore:

bias = 1 / floor(2^64 / 10^14) 5.4e-6

This value is negligibly close to zero, making the code base 10 bias insignificant.

For comparison, [RFC4226] generates 6-digit base-10 OTPs from 31-bit pseudo-random integers, resulting in:

bias = 1 / floor(2^31 / 10^6) 0.5e-3

While this value is also close to zero, it is approximately 100 times larger than EPHEMSEC's code base 10 bias.

A.1. Bias Formula

For integers m and M where m M and m > 0, when sampling uniformly from [0, M) and taking remainder modulo m, remainder distribution may not be uniform.

To quantify this non uniformity we can calculate a bias coefficient using:

* bias = 0 when M % m = 0

* bias = 1/floor(M/m) when M % m ≠ 0

The bias value varies in [0,1] and indicates the relative overrepresentation of certain remainders. When bias = 0, distribution is uniform.

A.1.1. Derivation

Let X be a uniformly distributed random variable that generates integers in the range [0,M). Let R = X % m, where R is a random variable that maps X to the remainder of the division by m.

We want to determine the probability distribution of R and quantify any bias that arises.

Since X is uniformly distributed, the probability of X taking any specific value v in [0, M) is 1/M.

Using the results from Lemma 1 below, we can determine the probability distribution of R.

A.1.2. Case 1: When M is divisible by m (M % m = 0)

From Lemma 1, each remainder r in [0, m) occurs exactly M/m times among the values [0, M).
Therefore, R is uniformly distributed with:

p(r) = (M/m) × (1/M) = 1/m for all r in [0, m)

In this case, there is no bias since all remainders are equally likely.

A.1.3. Case 2: When M is not divisible by m (M % m = s where 0 < s < m)

From Lemma 1, the remainders have different frequencies:

* Remainders r in [0, s) occur floor(M/m) + 1 times each

* Remainders r in [s, m) occur floor(M/m) times each

This leads to a non-uniform distribution:

* p(r) = p1 = (floor(M/m) + 1)/M for r in [0, s)

* p(r) = p2 = floor(M/m)/M for r in [s, m)

A.1.4. Bias Quantification

The bias arises because p1 > p2. We can express this relationship as:

$p1 = p2 \times (1 + 1/\text{floor}(M/m)) = p2 \times (1 + \text{bias})$

where:
bias = 1/floor(M/m)

This bias factor measures the non-uniformity of the distribution:

* bias = 0 when M is divisible by m (uniform case)

* bias $\in$ (0, 1] when M is not divisible by m, with larger values indicating greater non-uniformity

* The bias is maximized when floor(M/m) = 1, giving bias = 1

A.2. Lemma 1: Distribution of Remainders in [0, M)

Let m and M be integers where m M and m > 0.

For any integer r in [0, m), the number of integers v in [0, M) that satisfy v % m = r is:

1. When M is divisible by m (M % m = 0):

    * Exactly M/m values for every r in [0, m)

2. When M is not divisible by m (M % m = s where 0 < s < m):

    * floor(M/m) + 1 values for r in [0, s)

    * floor(M/m) values for r in [s, m)

A.2.1. Proof:

For r in [0, m) let count(r) be the count of v satisfying v % m = r for v in [0, M)

Any v, such that v % m = r can be written as v = qm + r where q 0
To keep v in [0, M), we must have q (M - 1 - r)/m, hence
count(r) = floor((M -1 - r)/m) + 1

When M = km (divisible case):

- count(r) = floor((km - 1 - r)/m) + 1 = k = M/m

When M = km + s (non-divisible case):

- For r < s: count(r) = k + 1
- For r s: count(r) = k

Appendix B. Design Rationale

The EPHEMSEC algorithm has undergone several implementation iterations, initially starting by modifying the HOTP algorithm described in [RFC4226] to achieve the desired security properties.

The following sections explain the rationale behind key design choices.

B.1. Use of HKDF

HKDF was chosen over the [RFC4226] HMAC approach for two reasons:

1. The HKDF domain separation feature enabled by the HKDF salt parameter (see Section 10.2) provides the necessary context binding used for phishing protection.

2. HKDF allows adjusting output size, which reduces the complexity of supporting the range of OTP/OTK sizes supported by EPHEMSEC (see Table 1).

B.2. Usage of PSK

The inclusion of a pre-shared key (PSK) ensures that generated OTP/OTKs are always a *mutual* secret, which the E1S1 pattern (see Section 5.3.1) alone cannot guarantee.

The incremental cost of mixing the PSK (see Section 6.1) into the HKDF ikm input (see Section 10.2) is expected to be low, and the PSK improves the secrecy of ikm. Therefore, it is always used even when the pattern (e.g., E1S2; see Section 5.3.2) already provides the desired *mutual* secrecy.

B.3. Usage of Nonces

The design requires that both the Initiator and Responder contribute a nonce and are responsible for ensuring that their respective nonces are not reused.

Although both parties currently contribute a nonce, the Initiator’s nonce may not be strictly required. Since the Initiator already contributes an ephemeral public key, this element provides sufficient entropy and freshness for the session. The Initiator’s nonce is therefore retained primarily to simplify the security analysis and preserve symmetry in the protocol design. Future revisions of this specification may remove it if it is shown to be redundant.

B.3.1. Transmission of the Responder Nonce

When using OTP output, bandwidth between the Responder and the Initiator is severely constrained. Therefore, a synchronization digit SYNCHINT is used to "compress" the Responder nonce PTIME (see Section 8.2).

When using OTK output, it is assumed that the application has addressed the bandwidth limitations that necessitate OTP output, and therefore PTIME could potentially be transmitted in uncompressed form.

However, explicit support for uncompressed transmission was not included for two reasons:

1. The computational cost of running the synchronization algorithm to recover PTIME from SYNCHINT is very low.

2. Having the server rely on SYNCHINT to reconstruct PTIME limits the Responder’s ability to reuse its own nonces, which provides additional security benefits.

Appendix C. Test Vectors

A minimal set of test vectors is provided for OTP and OTK schemes that use the SHA512 hash function and the X25519 ECDH function.

The [KerPass] open source project provides an extensive set of vectors that allow testing EPHEMSEC output generated using common hash functions and curves.

C.1. Kerpass_SHA512_X25519_E1S1_T600B10P8 vector

```
   {
     "scheme": "Kerpass_SHA512_X25519_E1S1_T600B10P8",
     "context": "a33698e33f8ac3aae8c3b1527cc019e5b3e34e5569883283b1a75fbfbe8f3d7d1d5c17d1
0d90533d33305a90560bfe",
     "psk": "55a439e62aefd92eebb7f4bc7e5f1626ef4545fa499252dd8658b61547f9df0e",
     "shared_secret": "0803020700020800",
     "otp": "83270280",
     "init_nonce": "d8421a5a86f2caf8d45e6ef5c62d3b5421ba7a",
     "init_time": 4134179892,
     "init_static_key": "",
     "init_ephemeral_key": "50d2af0ee110bd06604eb03e57a37763c9194400015290f3ef612be18ecd0
583",
     "init_remote_static_key": "f1e1116982475e7fd252a2ca69c1459b3a4439029fa3f33059d7e657b
3318912",
     "init_remote_ephemeral_key": "",
     "resp_time": 4134179984,
     "resp_synchro_hint": 0,
     "resp_static_key": "fad1034e60389317dd0e405c84fff2d17248fa1112d66a1979515ddf65a44a06
",
     "resp_ephemeral_key": "",
     "resp_remote_static_key": "",
     "resp_remote_ephemeral_key": "0fc4c28b9d85a476a1b664050901c78c9b4ff69bde707829bd1e05
e5602a3d55",
     "hkdf_salt": "432fa33698e33f8ac3aae8c3b1527cc019e5b3e34e5569883283b1a75fbfbe8f3d7d1d
5c17d10d90533d33305a90560bfe53244b6572706173735f5348413531325f5832353531395f453153315f543
630304231305038",
     "hkdf_info": "4e13d8421a5a86f2caf8d45e6ef5c62d3b5421ba7a54080000000003b23d1c",
     "hkdf_secret": "31215fcaafce8df63bd3a21371d15233501da05b5c52248fb6d4cafbdad20d3655a4
39e62aefd92eebb7f4bc7e5f1626ef4545fa499252dd8658b61547f9df0e"
   }
```


C.2. Kerpass_SHA512_X25519_E1S2_T600B16P8 vector


```
   {
     "scheme": "Kerpass_SHA512_X25519_E1S2_T600B16P8",
     "context": "037f9aae0e39c770b7c216572382e1b78ccff597a27aeb004cd14c881fe1944eb630e8af
0f9840b1ef3ffaf2cf697fb0a08d9cacd5e8c5ff4a1be583b4",
     "psk": "a2096639c4a3b6de9bb4d76bfe5b047545eaa3ccb8b67839c55e580ba54feb0d",
     "shared_secret": "0e0f0b02070e020e",
     "otp": "EFB27E2E",
     "init_nonce": "a5ddd81c502e054357c58ac483b8d39ce1e7",
     "init_time": 568923577,
     "init_static_key": "77166d9d14b2cc221a3829f600aad1f2f18872e8a7e7cb7aae2def72b69bff58
",
     "init_ephemeral_key": "90533d33305a90560bfef99d07c496ca92ef5001b62441860509af3d226d8
ba4",
     "init_remote_static_key": "dc94881a2063e4d30a0e7fe377838202e542b46d8836e57187e9abef9
9216078",
     "init_remote_ephemeral_key": "",
     "resp_time": 568923459,
     "resp_synchro_hint": 14,
     "resp_static_key": "dcad0c9e9aafe811761199f7e2286fda5036e5f5fc6f98c5f4b97fce71532ef8
",
     "resp_ephemeral_key": "",
     "resp_remote_static_key": "176198d01d29325b022a5efff4d315bbf9fb4a07755d19581f8577547
97f6b54",
     "resp_remote_ephemeral_key": "ecb63fa01c06b8d2c4de23d1abc4a5d57647481fbed22fb5866fda
4c7989103c",
     "hkdf_salt": "433d037f9aae0e39c770b7c216572382e1b78ccff597a27aeb004cd14c881fe1944eb6
30e8af0f9840b1ef3ffaf2cf697fb0a08d9cacd5e8c5ff4a1be583b453244b6572706173735f5348413531325
f5832353531395f453153325f543630304231365038",
     "hkdf_info": "4e12a5ddd81c502e054357c58ac483b8d39ce1e754080000000000d906ee",
```

```
     "hkdf_secret": "ca2a1d1f71d686ad52f6ef62571d38a5d579f96a86150861e736ea3e71aeba59c85c
ae9ec5404221a87423eaf952c151bc74b0213fa8a41fba0786ccf315fd5ba2096639c4a3b6de9bb4d76bfe5b0
47545eaa3ccb8b67839c55e580ba54feb0d"
   }
```

C.3. Kerpass_SHA512_X25519_E2S2_T600B32P9 vector


```
   {
     "scheme": "Kerpass_SHA512_X25519_E2S2_T600B32P9",
     "context": "9e2d582a0afac79709941506151e9e6ce20cef6da6c3aa457b",
     "psk": "3ec69559723db8a20cdace8bee0bf6534a1de8902786fc3b3626e4897f261dee",
     "shared_secret": "13010a191603100111",
     "otp": "K1ASP3G1H",
     "init_nonce": "716fbb9b9872e0b49dce2723a52ade9d5fdc59",
     "init_time": 2246779725,
     "init_static_key": "6a846ddc06f426e60dd4ae3fbd5b9b06286abc3862e473855e89a138c56ea0b2
",
     "init_ephemeral_key": "aa2b0e1b2e4704f08a377ba26e4475aaf1aab010fb8c6ad4bf63890816d3d
3f2",
     "init_remote_static_key": "acc4416cd58c4a1a43272253c75a8f27708174f24548466757e13374e
161c74b",
     "init_remote_ephemeral_key": "d32119b92d8a61d0c204a4355c86e19c7027376c2e42be84ffa907
a1aaec9e6a",
     "resp_time": 2246779996,
     "resp_synchro_hint": 17,
     "resp_static_key": "6716e0cf047f6721662c501ad15f9884cada29210c019c7df566a0e71b67b1dd
",
     "resp_ephemeral_key": "d45cf4b5736ef3f3fb0d2a98459a304e595276243d4af91aa5ef2366d44ad
e2e",
     "resp_remote_static_key": "f7c192278ea1c2323e92cc13da9ac5f0a8d53ac879eadbd774413c666
59d865e",
     "resp_remote_ephemeral_key": "86e53308c5bcefe25707f0ffee26cbdc260fc98d22fc9f9e53f4c1
994347c27c",
     "hkdf_salt": "43199e2d582a0afac79709941506151e9e6ce20cef6da6c3aa457b53244b6572706173
735f5348413531325f5832353531395f453253325f543630304233325039",
     "hkdf_info": "4e13716fbb9b9872e0b49dce2723a52ade9d5fdc59540800000000006eb4bb1",
     "hkdf_secret": "36d0bdc70509bb9cff6f2d899e4b9dd261c91da06604f43e408e96da1253827fc37f
09bdd64436333faab1150a0bd26d9c8ef3d5935e175b9db3c5ebacdbe7633ec69559723db8a20cdace8bee0bf
6534a1de8902786fc3b3626e4897f261dee"
   }
```


C.4. Kerpass_SHA512_X25519_E1S2_T1024B256P33 vector

```
   {
     "scheme": "Kerpass_SHA512_X25519_E1S2_T1024B256P33",
     "context": "c58ac483b8d39ce1e7037f9aae0e39c770b7c216572382e1",
     "psk": "aad1f2f18872e8a7e7cb7aae2def72b69bff58a2096639c4a3b6de9bb4d76bfe",
     "shared_secret": "8479fca69f6d1dbc9164f20eaa4d799ba3b8dae83f915864766fa66d4cae17422c
",
     "otp": "",
     "init_nonce": "5b047545eaa3ccb8b67839c55e580ba54feb0da5ddd81c502e054357",
     "init_time": 3762980154,
     "init_static_key": "6fda5036e5f5fc6f98c5f4b97fce71532ef8d077166d9d14b2cc221a3829f600
",
     "init_ephemeral_key": "4e5569883283b1a75fbfbe8f3d7d1d5c17d10d90533d33305a90560bfef99
d07",
     "init_remote_static_key": "2f92f4b84f1b107ca934e641f34bfae2b2251e9b70876cd1a256301f8
f3ccc0e",
     "init_remote_ephemeral_key": "",
     "resp_time": 3762980663,
     "resp_synchro_hint": 44,
     "resp_static_key": "c496ca92ef5001b62441860509af3d226d8ba4dcad0c9e9aafe811761199f7e2
",
     "resp_ephemeral_key": "",
     "resp_remote_static_key": "c52b20d6825127ecb33dfff5c1d9a0ed7f57b41eda1dd6855d7ad6268
372fc70",
     "resp_remote_ephemeral_key": "6a5a0dd77082720ebd55bcab514a68be8db117af5a17fbb06a7e03
5e44959307",
     "hkdf_salt": "4318c58ac483b8d39ce1e7037f9aae0e39c770b7c216572382e153274b657270617373
5f5348413531325f5832353531395f453153325f5431303234423235365033333",
     "hkdf_info": "4e1c5b047545eaa3ccb8b67839c55e580ba54feb0da5ddd81c502e0543575408000000
0037da8f2c",
     "hkdf_secret": "04b954ad9d6d42e39217b6a32ae83bc52ba662a39e2bcf18240e6b07034a8c4c86e7
a23622ce6b4cd46665d4309b624dd16fa39e57c86c7daf115ddd17f24039aad1f2f18872e8a7e7cb7aae2def7
2b69bff58a2096639c4a3b6de9bb4d76bfe"
   }
```


Author’s Address

Marc Vieuille
Email: marc.vieuille@polytechnique.org