Juicebox Protocol

Distributed Storage and Recovery of Secrets Using Simple PIN Authentication

Revision 2 — June 7, 2023

Nora Trapp
Juicebox Systems, Inc

Diego Ongaro
Juicebox Systems, Inc

Abstract

Existing secret management techniques often demand users memorize complex passwords, store convoluted recovery phrases, or place their trust in a specific service or hardware provider. We present a novel protocol that combines existing cryptographic techniques to eliminate these complications and reduce user complexity to recalling a short PIN. Our protocol specifically focuses on a distributed approach to secret storage that leverages *Oblivious Pseudorandom Functions* (OPRFs) and a *Secret-Sharing Scheme* (SSS) combined with self-destructing secrets to minimize the trust placed in any singular server. Additionally, our approach allows for servers distributed across organizations, eliminating the need to trust a singular service operator. We have built a reference open-source implementation of the client and server sides of this new protocol, the latter of which has variants for running on commodity hardware and secure hardware.

Contents

1 Introduction 3
2 Overview 3
  2.1 Configuration 3
  2.2 Realms 4
    2.2.1 Hardware Realms 4
    2.2.2 Software Realms 4
  2.3 Tenants 5
3 Cryptographic Primitives 5
  3.1 Oblivious Pseudorandom Functions (OPRFs) 5
  3.2 Secret-Sharing Scheme (SSS) 5
  3.3 Noise Protocol 6
  3.4 Additional Primitives 6
4 Protocol 6
  4.1 State 6
  4.2 Registration 7
    4.2.1 Phase 1 8
    4.2.2 Phase 2 8
  4.3 Recovery 9
    4.3.1 Phase 1 9
    4.3.2 Phase 2 10
    4.3.3 Phase 3 11
  4.4 Deletion 12
    4.4.1 Phase 1 12
  4.5 Authentication 12
5 Implementation Considerations 13
  5.1 Post-Quantum and OPRFs 13
  5.2 Registration Generations 13
6 Recommended Cryptographic Algorithms 13
  6.1 OPRFs 13
  6.2 SSS 14
  6.3 KDF 14
  6.4 Secret Encryption 14
  6.5 Tag MAC 14
7 Acknowledgements 14
8 References 15

1 Introduction

Services are increasingly attempting to provide their users with strong, end-to-end encrypted privacy features, often with the direct goal of preventing the service operator from accessing user data. In such systems, the user is generally given the role of managing a secret key to decrypt and encrypt their data. Secret keys tend to be long, not memorable, and difficult for a user to reliably to reproduce, by design. The burden of this complexity becomes particularly apparent when the user must enter their key material on a new device.

Techniques like seed phrases [1] provide some simplification to this process but still result in long and unmemorable strings of words that a user has to manage. Alternative approaches to key management such as passkeys [2] reduce the user burden but ultimately require that a user still have access to a device containing the key material.

We present an approach — the *Juicebox Protocol* — that allows the user to recover their secret material by remembering a short PIN, without having access to any previous devices.

Specifically, this protocol aims to:

1. Never give any service access to a user's secret material or PIN
2. Keep user burden low by allowing recovery through memorable low-entropy PINs while maintaining similar security to solutions utilizing high-entropy passwords
3. Distribute trust across mutually distrusting services, eliminating the need to trust any singular server operator or hardware vendor
4. Prevent brute-force attacks by self-destructing secrets after n failed recoveries
5. Allow auditing of secret access attempts

Juicebox provides open-source reference implementations for both the client and server on GitHub [3].

2 Overview

2.1 Configuration

A protocol client aims to distribute their secrets across n mutually distrusting services that implement the *Juicebox Protocol*. For this paper, we will refer to each service that a secret can be distributed to as an abstract *Realm*, elaborated upon in Section 2.2.

The overall security of the protocol is directly related to the set of n realms you configure your client with. Adding a *Realm* to your configuration generally results in a net increase in security, although there can be some caveats.

When adding a *Realm* to your configuration, some important questions to ask are:

- Who has access to the data stored on that *Realm*? (referred to as a *trust boundary* going forward)
- Does that *trust boundary* overlap with other realms in your configuration? If so, adding this *Realm* may reduce your overall security.

Configurations of realms are often used in *threshold* based operations. A $threshold < n$ allows increased availability of secrets when using a configuration with a larger size n since not all realms are required to be operational or in agreement for the operation to succeed.

A $threshold \geq 3$ is recommended where possible as small n configurations have the weakest security promises.

We require a $threshold > \frac{n}{2}$ which ensures that a majority consensus can always be reached and independent disagreeing sets of realms $\geq threshold$ cannot exist in your configuration.

2.2 Realms

Fundamentally, each *Realm* must adhere to the core protocol as defined here to be compatible. However, different realms may provide different security guarantees influencing the overall security of a user's secret value.

Each *Realm* is assigned a unique 16-byte identifier known as a $Realm_{id}$. For implementation purposes, this could be any value as long as it is universally unique across realms in your configuration. We generally recommend using a random value.

A *Realm* is controlled by an *operator* — the organization or individual who runs the service. The level of trust that must be placed in a given operator varies based on the underlying realm implementation.

Communication with a *Realm* should always occur over a secure protocol such as TLS that ensures the confidentiality and integrity of requests while limiting the possibility of replay attacks.

It should generally be assumed that each *Realm* controls only a share of a user's secret value (via SSS, as described in Section 3.2) and that a singular realm never has access to the full secret material. This guarantee is best achieved by ensuring that realms spanning multiple *trust boundaries* are utilized.

2.2.1 Hardware Realms

We explored a form of *Realm* backed by secure hardware, such as a hardware security module (HSM), as an avenue for creating diversity in *trust boundaries* within a client's configuration. Our primary focus with realms of this nature was on shifting the trust model away from the realm *operator* and hosting provider and onto the hardware *vendor*.

For this purpose, we specifically focused on HSMs that are programmable with non-volatile memory, as encapsulating the protocol operations within the hardware's trusted execution environment (TEE) assures that a malicious operator has no avenue of access. Non-volatile memory is required to prevent an *operator* from rolling back *Realm* state, which could prevent self-destruction of secrets. The HSMs we explored also allow some authorized form of programming, such that an *operator* can prove that a specific and verifiable version of the protocol is being executed within the TEE.

It must be noted that this shift in *trust boundaries* does not come for free. HSMs come with significant tradeoffs in terms of acquisition and operation cost as well as performance when compared to commodity hardware. This makes an HSM product insufficient as a standalone secret storage solution at scale. However, when used in concert with other types of realms — including hardware realms from other vendors — we believe the inclusion of such realms can provide a significant increase in security.

2.2.2 Software Realms

We additionally explored a form of *Realm* that can run on commodity hardware in common cloud providers. We specifically looked at this solution as the ease of deployment has the potential to significantly increase the number of *trust boundaries* that exist within a configuration. This can be particularly convenient to augment costly hardware realms, reducing trust placed on any individual hardware vendor, and even allowing a single organization to operate multiple realms with different *trust boundaries*.

The software solutions we specifically explored by design do not attempt to limit the user's need to trust the operator and additionally require placing some degree of trust in the hosting or database provider.

Since these realms only control an encrypted share of a user's secret value, we believe this is an acceptable tradeoff for the increased accessibility it provides.

It is also important to recognize that given the limited number of distinct cloud providers currently operating, overuse of such realms can potentially put too much secret information in one party's control and jeopardize user secrets.

2.3 Tenants

In general, this protocol assumes that any given *Realm* allows the storage and recovery of secrets from users spanning multiple organizational boundaries. We refer to each of these organizational boundaries as a *tenant*, and the protocol as defined ensures that any individual tenant can only perform operations on user secrets within their organizational boundary.

We encourage this multi-tenanted approach for realms, as we believe it enables a network effect that will broaden the adoption of the protocol. For example, realm operator *Alice*'s users no longer must trust *Alice* if *Alice* additionally distributes their secrets to realm operator *Bob*'s realm. To facilitate this exchange, *Alice* could allow *Bob*'s organization to distribute its user secrets to her realm.

This model can also potentially reduce the costs of running expensive hardware realms by distributing the costs of operation across multiple tenants.

3 Cryptographic Primitives

As a prerequisite to defining the protocol, we must define several cryptographic primitives that the protocol relies upon. Each of these is abstractly described, as the fundamental details of their implementation may evolve. For specific algorithms that we recommend as of the writing of this paper, see Section 6.

3.1 Oblivious Pseudorandom Functions (OPRFs)

An OPRF is a cryptographic primitive that enables a server to securely evaluate a function on a client's input while ensuring the server learns nothing about the client's input and the client learns nothing about the server's key beyond the output of the function.

For this paper, we will define an OPRF exchange with the following abstract functions:

$\boldsymbol{OprfDeriveKey}(\boldsymbol{seed})$: Returns an OPRF *key* derived from the provided *seed*. The key generated from a specific seed will always be the same.

$\boldsymbol{OprfBlind}(\boldsymbol{input})$: Performs the blinding step for the *input* value and returns the *blindedInput* and *blindingFactor*. This *blindedInput* is sent from the client to the server.

$\boldsymbol{OprfBlindEvaluate}(\boldsymbol{key}, \boldsymbol{blindedInput})$: Performs the evaluation step for the *blindedInput* and returns the *blindedResult*. This *blindedResult* is sent from the server to the client.

$\boldsymbol{OprfFinalize}(\boldsymbol{blindedResult}, \boldsymbol{blindingFactor}, \boldsymbol{input})$: Performs the finalization step to unblind the *blindedResult* using the *blindingFactor* and the *input* and returns the *result*.

$\boldsymbol{OprfEvaluate}(\boldsymbol{key}, \boldsymbol{input})$: Computes the unblinded *result* directly bypassing the oblivious exchange.

3.2 Secret-Sharing Scheme (SSS)

A secret-sharing scheme is a cryptographic algorithm that allows a secret to be divided into multiple shares, which are then distributed among different participants. Only by collecting a minimum number of shares — typically determined by a *threshold* specified during share creation — can the original secret

be reconstructed. This approach provides a way to securely distribute and protect sensitive information by splitting it into multiple fragments that individually reveal nothing about the original secret.

For this paper, we will define the following abstract functions for creating and reconstructing shares:

$\boldsymbol{CreateShares}(\boldsymbol{n}, \boldsymbol{threshold}, \boldsymbol{secret})$: Distributes *secret* into *n shares*
$\boldsymbol{RecoverShares}(\boldsymbol{shares})$: Recovers *secret* from *n shares* or returns an error if $n < threshold$

3.3 Noise Protocol

In some implementations of the *Juicebox Protocol*, such as when utilizing a *Hardware Realm*, it can be necessary to implement additional abstraction layers in communication between the user and the realm software such that client communication cannot securely terminate within the realm software. Often, this might look like a load balancer that services several realms and terminates a TLS connection. This introduces the possibility of an intermediary party intercepting requests before they reach the realm software.

To prevent this, the *Juicebox Protocol* allows for a *Realm* to optionally generate a 32-byte key pair and distribute the public key to its clients. The realm may then implement the NK-handshake pattern of the Noise Protocol [4]. Utilizing this public key allows users to establish a secure session directly with the realm software and encrypt each request with a new ephemeral key, regardless of additional hops a request may take to arrive at the *Realm*.

3.4 Additional Primitives

In addition to the previously established *OPRF* and *SSS* primitives, the following common primitives are necessary to define the protocol:

$\boldsymbol{Encrypt}(\boldsymbol{encryptionKey}, \boldsymbol{plaintext}, \boldsymbol{nonce})$: Returns an authenticated encryption of *plaintext* with *encryptionKey*. The encryption is performed with the given *nonce*.

$\boldsymbol{Decrypt}(\boldsymbol{encryptionKey}, \boldsymbol{ciphertext}, \boldsymbol{nonce})$: Returns the authenticated decryption of *ciphertext* with *encryptionKey*. The decryption is performed with the given *nonce*.

$\boldsymbol{KDF}(\boldsymbol{data}, \boldsymbol{salt})$: Returns a fixed 64-byte value that is unique to the input *data* and *salt*.

$\boldsymbol{MAC}(\boldsymbol{key}, \boldsymbol{input})$: Returns a 32-byte tag by combining the *key* with the provided *input*.

$\boldsymbol{Random}(\boldsymbol{n})$: Returns *n* random bytes. The *Random* function should ensure the generation of random data with high entropy, suitable for cryptographic purposes.

4 Protocol

The *Juicebox Protocol* can be abstracted to three simple operations — *register*, *recover*, and *delete*.

The following sections contain Python code that demonstrates the work required for each operation. For this code, we assume that the protocol has been appropriately configured with *n* mutually distrusting realms, each of which will be referred to as $Realm_i$.

4.1 State

$Realm_i$ will store a record indexed by the combination of the registering user's identifier (UID[1]) and their *tenant*. This ensures that a given *tenant* may only authorize operations for its users.

Clients do not require any persistent state to recover their secrets and should just rely on the user's *PIN* input.

[1] As defined in Section 4.5

This record can exist in one of three states:

NotRegistered: The user has no existing registration with this *Realm*. This is the default state if a user has never communicated with the *Realm*.

Registered: The user has registered secret information with this *Realm* and can still attempt to restore that registration.

NoGuesses: The user has registered secret information with this *Realm*, but can no longer attempt to restore that registration.

A user transitions into the *NoGuesses* state when the number of *attemptedGuesses* on their registration equals or exceeds their *allowedGuesses*, self-destructing the registered data.

In the *Registered* state, the following additional information is stored corresponding to the registration:

version: a unique 16-byte value that identifies this registration across all *Realms*

allowedGuesses: the maximum number of guesses allowed before the registration is permanently deleted by the *Realm*

attemptedGuesses$_i$: starts at 0 and increases on recovery attempts, then reset to 0 on successful recoveries

saltShares$_i$: a share of the salt the client generated during registration and used to hash their *PIN*

oprfSeeds$_i$: a random OPRF seed the client generated during registration, unique to this realm and this registration

maskedUnlockKeyShares$_i$: a masked share of the unlock key

unlockTags$_i$: the tag the client provides to demonstrate knowledge of the PIN and release *encryptedSecretShares$_i$*

encryptedSecretShares$_i$: a share of the user's encrypted secret

4.2 Registration

Registration is a two-phase operation that a new user takes to store a PIN-protected secret. A registration operation is also performed to change a user's PIN or register a new secret for an existing user.

A reference client might expose registration in the following form:

$$register(pin, secret, allowedGuesses, threshold, associatedData)$$

pin: represents a low entropy value known to the user that will be used to recover their secret, such as a 4-digit pin[2]

secret: represents the secret value a user wishes to persist

allowedGuesses: specifies the number of failed attempts a user can make to recover their secret before it is permanently deleted

threshold: represents the number of realms that shares must be recovered from for the *secret* to be restored

associatedData: known user data that is factored into the random *salt* used to stretch the user's *PIN*[3]

[2]While the protocol aims to provide strong security guarantees for low entropy pins, using a high entropy value here will provide increased security.

[3]Using a known constant, like the UID, can prevent a malicious *Realm* from returning a fixed *salt* with a pre-computed password table.

4.2.1 Phase 1

The purpose of Phase 1 is to verify that at least y realms are available to store a new registration, where $y \geq threshold$. Ensuring registration succeeds on more realms than your *threshold* increases availability during recovery.

An empty *register1* request is sent from the client to each $Realm_i$.

For realms that expose a *public key* and implement *Noise*, it is recommended to combine this request with the handshake if there is no open connection, as it does not reveal any sensitive information.

A *Realm* should always be expected to respond *OK* to this request unless the Noise handshake fails or a transient network error occurs.

Provided a client has completed *Phase 1* on y realms, the client can proceed to prepare the registration material that will be stored on each $Realm_i$ within that set.

4.2.2 Phase 2

The purpose of Phase 2 is to update the registration state on each $Realm_i$ to reflect the new *PIN* and *secret*.

The following demonstrates the work a client should perform to prepare a new registration:

```
def PrepareRegister2(realms, pin, secret, associatedData):
  version = Random(16)

  salt = Random(16)
  saltShares = CreateShares(y, threshold, salt)

  stretchedPin = KDF(pin, salt + associatedData)
  accessKey = stretchedPin[:32]
  encryptionKey = stretchedPin[-32:]

  # A `nonce` of 0 can be safely used since `encryptionKey` changes with each registration
  encryptedSecret = Encrypt(secret, encryptionKey, 0)
  encryptedSecretShares = CreateShares(len(realms), threshold, encryptedSecret)

  oprfSeeds = [Random(32) for _ in realms]
  oprfResults = [OprfEvaluate(OprfDeriveKey(seed), accessKey) for seed in oprfSeeds]

  unlockKey = Random(32)
  unlockKeyShares = CreateShares(len(realms), threshold, unlockKey)

  maskedUnlockKeyShares = [x ^ y for x, y in zip(unlockKeyShares, oprfResults)]

  unlockTags = [MAC(unlockKey, realm.id) for realm in realms]

  return (
      version,
      saltShares,
      oprfSeeds,
      maskedUnlockKeyShares,
      unlockTags,
      encryptedSecretShares
  )
```

A *register2* request is then sent from the client to each $Realm_i$ that contains the prepared:

- version
- allowedGuesses
- $saltShares_i$
- $oprfSeeds_i$
- $maskedUnlockKeyShares_i$
- $unlockTags_i$
- $encryptedSecretShares_i$

Upon receipt of a *register2* request, $Realm_i$ creates or overwrites the user's registration state with the corresponding values from the request.

A *Realm* should always be expected to respond *OK* to this request unless a transient network error occurs.

4.3 Recovery

Recovery is a three-phase operation that an existing user takes to restore a PIN-protected secret.

A reference client might expose recovery in the following form:

$$secret, error = recover(pin, threshold, associatedData)$$

pin: represents the same value used during *register*
threshold: represents the same value used during *register*
associatedData: represents the same value used during *register*
secret: the recovered secret as provided during registration, if and only if the correct *pin* was provided
error: indicates an error in recovery, such as an invalid *pin* or the *allowedGuesses* having been exceeded

4.3.1 Phase 1

The purpose of Phase 1 is to recover the *version* and $saltShares_i$ from each $Realm_i$ and determine a set of realms to restore from.

An empty *recover1* request is sent from the client to each $Realm_i$.

For realms that expose a *public key* and implement *Noise*, it is recommended to combine this request with the handshake if there is no open connection, as it does not reveal any sensitive information.

The following demonstrates the work a $Realm_i$ should perform to process the request:

```
def Recovery1(state, request):
  if state.isRegistered:
    if state.attemptedGuesses >= state.allowedGuesses:
      state.transitionToNoGuesses()
      return Error(NoGuesses)

    return Ok(state.version, state.saltShare)
  elif state.isNoGuesses:
    return Error(NoGuesses):
  elif state.isNotRegistered:
    return Error(NotRegistered)
```

An *OK* response from this phase should always be expected to return the following information from the user's registration:

- version

- saltShares$_i$

Once a client has completed Phase 1 on at least *threshold* *Realm*$_i$ with a majority consensus on the returned *version* and *salt* it can proceed to Phase 2. Only the realms that exist within this majority should be considered for the remaining phases. Provided the initial *threshold* during registration consisted of a majority of realms, this consensus should always be reached as long as the user is registered. If this consensus cannot be reached, the client should assume that the user is *NotRegistered* on any realm.

4.3.2 Phase 2

The purpose of Phase 2 is to increment the *attemptedGuesses* for the user and recover the *maskedUnlockKeyShares* stored during registration along with the *OPRF* result required to unmask them and reconstruct the *unlockKey*. An optimal client can abort Phase 2 as soon as *threshold* *OK* responses are recovered, as this should be sufficient to recover the *unlockKey*.

By design, a client cannot recover their secret or determine the validity of their PIN by performing Phase 2 alone. This ensures that each realm has an opportunity to learn if the client succeeded or failed in their recovery attempt in order to audit their attempt appropriately and self-destruct their secret data if necessary.

The following demonstrates the work a client should perform to prepare for Phase 2:

```
def PrepareRecovery2(realms, pin, associatedData, version, salt):
  stretchedPin = KDF(pin, salt + associatedData)
  accessKey = stretchedPin[:32]
  encryptionKey = stretchedPin[-32:]

  blindedAccessKeys, blindingFactors = zip(*[OprfBlind(accessKey) for _ in realms])

  return (
    accessKey,
    encryptionKey,
    blindedAccessKeys,
    blindingFactors
  )
```

A *recover2* request is then sent from the client to each *Realm*$_i$ that contains the previously determined:

- version
- blindedAccessKeys$_i$

The following demonstrates the work a *Realm*$_i$ should perform to process the request:

```
def Recovery2(state, request):
  if state.isRegistered:
    if state.attemptedGuesses >= state.allowedGuesses:
      state.transitionToNoGuesses()
      return Error(NoGuesses)
    if request.version != state.version:
      return Error(VersionMismatch)

    oprfKey = OprfDeriveKey(state.oprfSeed)
    blindedResult = OprfBlindEvaluate(oprfKey, request.blindedAccessKey)

    state.attemptedGuesses += 1
```

```
    return Ok(blindedResult, state.maskedUnlockKeyShare)
  elif state.isNoGuesses:
    return Error(NoGuesses):
  elif state.isNotRegistered:
    return Error(NotRegistered)
```

An *OK* response from this phase should always be expected to return the following information:

- blindedResult
- $\text{maskedUnlockKeyShares}_i$

Provided at least *threshold OK* responses have been received from Phase 2, a client can safely proceed to Phase 3.

4.3.3 Phase 3

The purpose of Phase 3 is to recover the *encryptedSecretShares* allowing decryption and reconstruction of the user's *secret*. Additionally, this phase tells each $Realm_i$ the result of the operation so it can be audited appropriately.

Upon success this phase resets the *attemptedGuesses* on each $Realm_i$ to 0. For this reason, a client should complete this process on *all* realms that Phase 2 was performed on, even if sufficient material has been received to recover the user's *secret*. Otherwise, secret material may prematurely self-destruct.

The following demonstrates the work a client should perform to prepare for Phase 3:

```
def PrepareRecovery3(
  realms,
  accessKey,
  blindingFactors,
  blindedResults,
  maskedUnlockKeyShares
):
  oprfResults = []
  for blindedResult, blindingFactor in zip(blindedResults, blindingFactors):
    oprfResults.append(OprfFinalize(blindedResult, blindingFactor, accessKey))

  unlockKeyShares = [x ^ y for x, y in zip(maskedUnlockKeyShares, oprfResults)]
  unlockKey = RecoverShares(unlockKeyShares)
  unlockTags = [MAC(unlockKey, realm.id) for realm in realms]

  return unlockTags
```

A *recover3* request is then sent from the client to each $Realm_i$ that contains the previously determined:

- version
- unlockTags_i

The following demonstrates the work a $Realm_i$ should perform to process the request:

```
def Recovery3(state, request):
  if state.isRegistered:
    if request.version != state.version:
      return Error(VersionMismatch)

    if !ConstantTimeCompare(request.unlockTag, state.unlockTag):
```

```
      guessesRemaining = state.allowedGuesses - state.attemptedGuesses

      if guessesRemaining == 0:
        state.transitionToNoGuesses()

      return Error(BadUnlockTag(guessesRemaining))

    state.attemptedGuesses = 0

    return Ok(state.encryptedSecretShare)
  elif state.isNoGuesses:
    return Error(NoGuesses):
  elif state.isNotRegistered:
    return Error(NotRegistered)
```

An *OK* response from this phase should always be expected to return the following information from the user's registration state:

- encryptedSecretShares$_i$

A *BadUnlockTag* response from this phase should always be expected to return the previously determined:

- guessesRemaining

Upon receipt of *threshold OK* responses, the client can reconstruct the user's *secret*.

The following demonstrates the work a client should perform to do so:

```
def RecoverSecret(encryptionKey, encryptedSecretShares):
  encryptedSecret = RecoverShares(encryptedSecretShares)
  secret = Decrypt(encryptionKey, encryptedSecret, 0)
  return secret
```

4.4 Deletion

Delete is a single-phase operation that reverts a user's registration state to *NotRegistered*.

A reference client might expose delete in the following form:

$$delete()$$

It is important to note that *delete* does not require the user's *pin*, since a user can always register a new secret effectively deleting any existing secrets.

4.4.1 Phase 1

An empty *delete* request is sent from the client to each *Realm*$_i$.

Upon receipt of a *delete* request *Realm*$_i$ sets the user's registration state to *NotRegistered*.

A *Realm* should always be expected to respond *OK* to this request unless a transient network error occurs.

4.5 Authentication

To enforce *tenant* boundaries and prevent unauthorized clients from self-destructing a user's secret, a given *Realm*$_i$ requires authentication proving that a user has permission to perform operations.

A *Realm*$_i$ aims to know as little as possible about users and consequently relies on individual tenants to determine whether or not a user is allowed to perform operations.

To delegate this control to tenants, a realm *operator* must generate a random 32-byte signing key ($signingKey = Random(32)$) for each *tenant* they wish to access their $Realm_i$. This signing key should be provided an integer version v and the tenant should be provided a consistent alphanumeric name *tenantName* that is shared by both the realm *operator* and the *tenant*.

Given this information, a *tenant* must vend a signed JSON Web Token (JWT) [5] to grant a given user access to the realm.

The header of this JWT must contain a *kid* field of *tenantName:v* so that the $Realm_i$ knows which version v of *tenantName*'s signing key to validate against.

The claims of this JWT must contain an *iss* field equivalent to *tenantName* and a *sub* field that represents a persistent user identifier (UID) the realm can use for storing secrets. Additionally, an *aud* field must be present and contain a single hex-string equivalent to the $Realm_{i(id)}$ a token is valid for.

A $Realm_i$ must reject any connections that:

1. Don't contain an authentication token
2. Aren't signed with a known signing key for a given *tenantName* and version v matching the *kid*
3. Don't have an *aud* exactly matching their $Realm_{i(id)}$

The operations defined in the prior sections assume all requests contain valid authentication tokens for a given $Realm_i$ or that an *InvalidAuthentication* (401) error is returned by the *Realm*.

5 Implementation Considerations

5.1 Post-Quantum and OPRFs

While we have not performed extensive exploration into the state of post-quantum OPRFs, we believe that using a PQ-confidential transport ensures that even if an attacker was able to record the OPRF outputs today, they would not be able to later utilize an *unlockTag* gleaned from them in any meaningful way. The user's *secret* is still protected within the Phase 3 response. This assumes that the protocol has sufficiently evolved by this point, or that the user has otherwise re-registered their secret rotating their *oprfSeed*, such that the *unlockTag* could not be provided directly in new interaction with a *Realm*.

5.2 Registration Generations

The current protocol assumes that each user realm stores either 0 or 1 record per user. While this approach is simple, it does have a downside in the specific scenario of a user re-registering. If the re-registration succeeds locally but fails globally (it succeeds on some realms but not enough to reach a threshold), then the user may be unable to recover their secret using either the old or new PIN. They will need to try again later to register successfully.

This downside could be resolved by adding the concept of *generations* to realms allowing users to store n registrations on a realm. During recovery, a user could fall back to the latest generation that still has at least *threshold* realms available to recover from, so a failed registration would not erase their existing registration.

6 Recommended Cryptographic Algorithms

6.1 OPRFs

The protocol relies on multiple *OPRF* functions to ensure a *Realm* does not gain access to the user's PIN.

We recommend utilizing OPRFs as described in the working draft by Davidson *et al.* [6] with the *Ristretto255* curve as defined by Valence *et al.* [7]. Note that other cipher suites could also be potentially suitable depending on hardware and software constraints. In particular, we recognize that certain HSMs may place restrictions on available cipher suites.

6.2 SSS

The protocol relies on a secret-sharing scheme to ensure a *Realm* does not gain access to the user's secret.

We recommend utilizing the scheme defined by Shamir [8], but other schemes are viable.

6.3 KDF

The protocol relies on a *KDF* function to add entropy to the user's *PIN*, which provides an additional layer of protection if a *threshold* of realms were to be compromised. While the specific hashing algorithm is up to the client, we recommend utilizing *Argon2* [9].

Determining the appropriate configuration parameters for Argon2 is highly dependent on the limitations of your client hardware. Additionally, since users may register and recover secrets across multiple devices a given user is specifically limited by the weakest device they expect to use. An intelligent client could potentially adjust a user's hashing strength based on the performance of their registered devices, assuming user devices only get more performant. This is of course not a valid assumption in many common cases.

For the common case, we have evaluated performance across popular smartphones and browsers circa 2019 and determined the following recommended parameters:[4]

- Utilize Argon2id to defend against timing and GPU attacks
- Utilize parallelism of 1 (limited primarily by browser-based threading)
- Utilize 32 iterations
- Utilize 16 KiB of memory (limited primarily by low-end Android devices)

We believe this combination of parameters provides a reasonable balance between performance — a user will not wait minutes to register a secret — and security.

A client may always re-register utilizing new parameters to provide stronger guarantees in the future.

6.4 Secret Encryption

The protocol relies on an authenticated *Encrypt* and *Decrypt* function to ensure that the user's PIN is required to access the secret value, even if secret shares are compromised. While the specific encryption algorithm is up to the client, we recommend utilizing *ChaCha20* and *Poly1305* [10].

6.5 Tag MAC

The protocol relies on a *MAC* function to compute an *unlockTag* for a given realm. While the specific algorithm is up to the client, we recommend utilizing *HMAC* over *BLAKE2s-256* [11].

7 Acknowledgements

- The protocol is heavily based on design and feedback from Trevor Perrin and Moxie Marlinspike.
- The protocol builds on concepts closely related to those explored by Jarecki *et al.* in their PPSS [12] primitive and Davies *et al.* in their *Perks* [13] design.

[4] Parts of this evaluation were performed in 2019 at the Signal Foundation as part of their Secure Value Recovery project.

- Some of the ideas utilized in this design were first suggested by the Signal Foundation in the future-looking portion of their *"Secure Value Recovery"* blog post [14].

8 References

[1] M. Palatinus, P. Rusnak, A. Voisine, and S. Bowe, "Mnemonic code for generating deterministic keys," 2013. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki

[2] T. F. Alliance, 2022. [Online]. Available: https://fidoalliance.org/specifications/

[3] "Juicebox-systems." [Online]. Available: https://github.com/juicebox-systems

[4] T. Perrin, "The noise protocol framework," 2018. [Online]. Available: http://www.noiseprotocol.org/noise.html

[5] M. B. Jones, J. Bradley, and N. Sakimura, "Rfc 7519: json web token (jwt)," 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7519

[6] A. Davidson, A. Faz-Hernandez, N. Sullivan, and C. A. Wood, "Oblivious pseudorandom functions (oprfs) using prime-order groups," 2023. [Online]. Available: https://www.ietf.org/id/draft-irtf-cfrg-voprf-21.html

[7] H. d. Valence, J. Grigg, et al., "The ristretto255 and decaf448 groups," 2023. [Online]. Available: https://www.ietf.org/id/draft-irtf-cfrg-ristretto255-decaf448-07.html

[8] A. Shamir, "How to share a secret," 1979. [Online]. Available: https://web.mit.edu/6.857/OldStuff/Fall03/ref/Shamir-HowToShareASecret.pdf

[9] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: the memory-hard function for password hashing and other applications," 2015. [Online]. Available: https://www.password-hashing.net/argon2-specs.pdf

[10] Y. Nir, and A. Langley, "Rfc 7539: chacha20 and poly1305 for ietf protocols," 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7539

[11] M.-J. O. Saarinen, and J.-P. Aumasson, "Rfc 7693: the blake2 cryptographic hash and message authentication code (mac)," 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7693

[12] S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu, 2016. [Online]. Available: https://eprint.iacr.org/2016/144.pdf

[13] G. T. Davies, and J. Pijnenburg, 2022. [Online]. Available: https://eprint.iacr.org/2022/1017

[14] J. Lund, "Technology preview for secure value recovery," 2019. [Online]. Available: https://signal.org/blog/secure-value-recovery/