Juicebox Protocol

Distributed Storage and Recovery of Secret Values Using Simple PIN Authentication

Revision 1 — June 2, 2023

Nora Trapp
Juicebox Systems, Inc

Diego Ongaro
Juicebox Systems, Inc

Abstract

Ensuring high adoption of privacy software requires simplicity. Unfortunately, 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 have implemented a novel protocol that combines existing cryptographic techniques to eliminate these complications and reduce user complexity to recalling a 4-digit PIN. Our protocol specifically focuses on a distributed approach to secret storage that leverages *Oblivious Pseudo Random Functions* (OPRFs) and *Shamir's Secret Sharing* (SSS) to minimize the trust placed in any singular server. Additionally, our approach allows for servers to be controlled by any number of organizations eliminating the need to trust a singular service operator.

Contents

1 Introduction

At its core, the *Juicebox Protocol* is a specification for performing secret management operations distributed across a set of *Realms*.

Specifically, the protocol aims to:

1. Never give any *Realm* access to secret values
2. Keep user burden simple by allowing recovery through low-entropy PINs
3. Eliminate the need to trust any singular *Realm* operator or hardware vendor

Additionaly, a key feature of the protocol is that anyone can implement and run instances that conform to it, allowing for distributed trust across different organizational boundaries.

Juicebox provides open source reference implementations for both the client and server on their github [1].

2 Overview

2.1 Realms

For the purposes of this paper, we will refer to each server that a secret can be distributed to as an abstract *Realm*. Fundamentally, each *Realm* must adhere to the core protocol as defined here in order to be compatible. However, different realms may provide different security gurantees 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.

Additionally, realms have the option to generate a 32-byte EC25519 key pair which is used for encrypted communication via Noise, as described in Section 2.5

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.

It should generally be assumed that each *Realm* controls only a share of a user's secret value (via SSS, as described in Section 2.4), and that a singular realm never has access to the full secret material. This gurantee is best achieved by ensuring a variety of realm types are used that span across multiple operators and trust boundaries.

2.1.1 Hardware Realms

One form of *Realm* that we explored is the variant backed by secure hardware, such as a hardware security module (HSM). Realms of this nature allow for a significant reduction in or removal of the user's need to trust a realm's operator, as it is possible to encapsulate all protocol operations within the hardware's trusted execution environment (TEE) such that a malicious operator has no avenue of access. Additionally, it is possible to attest that a specific and verifiable version of software is being executed within the TEE.

This reduction in operator trust does not come for free — the user's trust is transitioned from the realm operator to the HSM vendor. Additionally, HSMs come with significant tradeoffs in terms of acquistion and operation cost as well as performance when compared to commodity hardware. This makes any singular 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 hardware realms can provide a significant increase in security.

2.1.2 Software Realms

Another form of *Realm* that we explored is a light weight software solution that can be easily hosted in common cloud providers. Realms of this nature allow focus on ease of deployment, facilitating further distrubition across organizational boundaries. This can be incredibly convenient to augment the 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 realms 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, over use of such realms can potentially put too much secret information in one party's control and jepoardize user secrets.

2.2 Tenants

In general, this protocol assumes that any given *Realm* allows 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 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 their 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.

2.3 Oblivious Pseudrandom Functions (OPRFs)

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

Our protocol specifically utilizes OPRFs as described in the working draft by Davidson *et al.* [2] as one part of a strategy for registering and recovering secrets on a *Realm* without revealing a user's *PIN* to the *Realm*.

For the purposes of this paper we will define the following abstract functions which map to the corresponding operations in the working draft.

$\boldsymbol{OprfDeriveKey}(\boldsymbol{seed})$ 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*. This message 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 message is sent from the server to the client.

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

$\boldsymbol{OprfEvaluate}(\boldsymbol{key}, \boldsymbol{accessKey})$ Computes the unblinded *result* directly bypassing the blinded exchange.

2.4 Shamir's Secret Sharing (SSS)

Shamir's Secret Sharing [3] 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 our purposes, we will define the following abstract functions for creating and reconstructing shares using this algorithm:

$\boldsymbol{CreateShares}(\boldsymbol{threshold}, \boldsymbol{secret})$ Distributes *secret* into *N shares*

$\boldsymbol{RecoverShares}(\boldsymbol{shares})$ Recovers *secret* from *N shares* or returns an error if less than *threshold shares* were recovered

2.5 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. These additional hops introduce the potential for replay of client requests by intermediary parties, potentially allowing a malicious server to make recovery attempts against a user's *secret*.

In order to prevent replay of requests, the *Juicebox Protocol* allows for realms to optionally share a 32-byte EC25519 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 pre-shared 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 Protocol

The *Juicebox Protocol* can be abstracted to three simple operations — *register*, *recover*, and *delete*. These operations, as well as several prerequisites for performing them, are outlined below.

Protocol clients are expected to be configured with *n* mutually distrusting realms, each of which will be referred to as $Realm_i$ for here on. The overall security benefits of the protocol are dependent on sufficient distribution across trust domains.

3.1 Authentication

In order to enforce *Tenant* boundaries, 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, an consequently relies on individual tenants to determine whether or not a user is allowed to perform a given operation.

In order to cede this control to tenants, a realm *operator* must generate a random 32-byte signing key 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] in order 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 identifer (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 $Realm_{i(id)}$

From this point forward, this paper will assume all requests contain valid authentication tokens for a given $Realm_i$ or that an *InvalidAuthentication* (401) error is returned by the *Realm*.

3.2 Additional External Functions

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

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

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

$\boldsymbol{Hash}(\boldsymbol{data}, \boldsymbol{salt})$ 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.

Recommendations on implementing these functions can be found in Section 5

3.3 Storage

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

This record can exist in one of three states:

NotRegistered The user has no existing registration on 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 exceeeds their *allowedGuesses*.

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*
attemptedGuesses starts at 0 and increases on recovery attempts, then reset to 0 on successful recoveries
allowedGuesses the maximum number of guesses allowed before the registration is permanently deleted by the *Realm*
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
maskedTgkShares$_i$ a masked share of the tag generating key
unlockTags$_i$ the key the client provides to demonstrate knowledge of the PIN and release *encryptedSecretShares$_i$*
encryptedSecretShares$_i$ a share of the user's encrypted secret

Standalone, none of the values stored in the *Registered* state expose sensitive user secrets and can be stored by the *Realm* as they see fit for their trust model. This constraint depends on realms existing across trust boundaries to prevent recovery of secret shared values.

3.4 Registration

Registration is a two-phase process that a new user takes to store a PIN-protected secret.

A reference client might expose registration in the following form:

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

pin represents a low entropy value known to the user that will be used to recover their secret, such as a 4-digit pin
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 in order for the *secret* to be recovered. it is generally recommended that $threshold > \frac{n}{2}$ where *n* is the number of realms you are distributing to. we additionally recommend a $threshold \geq 3$

3.4.1 Phase 1

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

For realms that expose a *public key* and implement *Noise*, this request performs the handshake and establishes a *Noise* session prior to any sensitive information being transmitted in subsequent phases.

For realms that don't implement *Noise*, this request has no operation. A client implementation could choose to optimize their behavior by skipping this phase for a given *Realm$_i$* that has no *public key*.

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

3.4.2 Phase 2

Provided a client has completed *Phase 1* (or decided it was safe to omit), the client can proceed to prepare the registration material that will be stored on each *Realm*$_i$. Phase 2 should not be conducted until Phase 1 is completed successfully on all realms to avoid a mismatched registration state across realms.

The client should perform the following actions:

1. Generate a random 16-byte *version* that is used to validate registration consistency across realms
   - $version = Random(16)$
2. Generate a random 16-byte *salt* that is used when hashing the user's *PIN*
   - $salt = Random(16)$
3. Derive an *accessKey* and *encryptionKey* by hashing the user's *PIN*
   - $accessKey, encryptionKey = Hash(PIN, salt)$
   - *accessKey* is the first 32-bytes of the *Hash* result
   - *encryptionKey* is the last 32-bytes of the *Hash* result
   - This hashing operation adds additional entropy to the user's *PIN* before using it in *OPRF* operations.
4. Encrypt the user's *secret* using the derived *encryptionKey*
   - $encryptedSecret = Encrypt(secret, encryptionKey, 0)$
   - Since a new *encryptionKey* is used for every registration, a constant *nonce* of zero can be safely used.
   - This serves as a low-overhead way to ensure that even if realms were to collude the user's *PIN* is required in order to recover their *secret*. However, since a *Realm*$_i$ only ever should have access to a single *secret* share this operation could technically be considered optional depending on your trust concerns.
5. Create shares of *encryptedSecret*
   - $encryptedSecretShares = CreateShares(threshold, encryptedSecret)$
6. Generate a random 32-byte *oprfSeeds*$_i$ for each *Realm*$_i$
   - $oprfSeeds_i = Random(32)$
   - We acknowledge that it is unconventional for *OPRF* key material to be generated on the client. However, in this instance the benefits of doing so outweigh the downsides. Specifically, this change in behavior allows it to be possible to register a secret on a realm without the additional round trip to compute the *OprfResult*. When distributed across 3 or more realms, the performance impacts of this change start to become significant. Since this protocol generally expects clients to have access to a secure random number generator capable of generating a good *oprfSeed*, the primary cause for concern becomes one of implementation and verifying that the key material is not leaked from the client before or after being provided to the *Realm*.
7. Generate a random 32-byte tag generating key *tgk*
   - $tgk = Random(32)$
   - The tag generating key is later used — in combination with the user's *accessKey* — to validate ownership of a secret by deriving an *unlockTag* for each *Realm*$_i$
8. Create shares of *tgk*
   - $tgkShares = CreateShares(threshold, tgk)$
9. Derive the *oprfResult*$_i$ of *accessKey* for each *oprfSeeds*$_i$
   - $oprfResult_i = OprfEvaluate(OprfDeriveKey(oprfSeed_i), accessKey)$
   - Since we have the *OPRF* key material and the *accessKey* we can directly derive the result and skip the blinding process

10. Derive the *maskedTgkShares*
    - $maskedTgkShares_i = tgkShares_i \ XOR \ oprfResult_i$
    - This operation requires that the user to first prove they know their *PIN* to derive the *oprfResult* before they can unmask the *tgkShares* and recover the *tgk*
11. Derive the *unlockTags*
    - $unlockTag_i = MAC(tgk, Realm_{i(id)})$
    - $Realm_{i(id)}$ is the unique 16-byte identifier for each *Realm$_i$* as described in Section 2.1
    - Knowledge of this value during recovery grants the user access to their *encryptedSecretShares$_i$*

A *register2* request is then sent from the client to each *Realm$_i$* that contains the previously determined:

- version
- saltShares$_i$
- oprfSeeds$_i$
- unlockTags$_i$
- maskedTgkShares$_i$
- encryptedSecretShares$_i$
- allowedGuesses

Upon receipt of a *register2* request, *Realm$_i$* creates or overwrites the user's registration record 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.

3.5 Recovery

Recovery is a three-phase process that a new user takes to store a PIN-protected secret.

A reference client might expose recovery in the following form:

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

pin represents the same value used during *register*
threshold 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

3.5.1 Phase 1

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

For realms that expose a *public key* and implement *Noise*, this request additionally performs the handshake and establishes a *Noise* session prior to any sensitive information being transmitted in subsequent phases.

Upon receipt of a *recover1* request, *Realm$_i$* checks the current state of a user's registration and responds appropriately.

NotRegistered The *Realm* will respond with a *NotRegistered* error to this request.
Registered

- If $attemptedGuesses \geq allowedGuesses$ on the stored registration, the *Realm* will immediately update the record to the *NoGuesses* state and respond appropriately.
- Otherwise, the *Realm* will process this request and return an *OK* response.

NoGuesses The *Realm* will respond with a *NoGuesses* error to this request.

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

- version
- saltShares_i

3.5.2 Phase 2

Once a client has successfuly completed Phase 1 on all $\textit{Realm}_i$, it must determine a majority consensus on the returned *version* and *salt*. 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. If this consensus cannot be reached, the client should assume that the user is *NotRegistered*.

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

The client should perform the following actions:

1. Derive an *accessKey* and *encryptionKey* by hashing the user's *PIN*
   - $accessKey, encryptionKey = Hash(PIN, salt)$
   - *accessKey* is the first 32-bytes of the *Hash* result
   - *encryptionKey* is the last 32-bytes of the *Hash* result
   - This process is identical to the one performed during registration
2. Compute *blindedAccessKeys* for each $\textit{Realm}_i$
   - $blindedAccessKeys_i = OprfBlind(accessKey)$

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

- version
- $\text{blindedAccessKeys}_i$

Upon receipt of a *recover2* request, $\textit{Realm}_i$ checks the current state of a user's registration and responds appropriately.

NotRegistered The *Realm* will respond with a *NotRegistered* error to this request.

Registered

- If $attemptedGuesses \geq allowedGuesses$ on the stored registration, the *Realm* will immediately update the record to the *NoGuesses* state and respond appropriately.
- If the *version* on the stored registration does not match the *version* on the request, the *Realm* will respond with a *VersionMistmatch* error to this request.
- Otherwise, the *Realm* will process this request and return an *OK* response.

NoGuesses The *Realm* will respond with a *NoGuesses* error to this request.

The $\textit{Realm}_i$ should perform the following actions to process the request:

1. Compute the *blindedResult* using the $\textit{blindedAccessKeys}_i$ from the request:
   - $blindedResult = OprfBlindEvaluate(blindedAccessKeys_i)$
2. Increment the *attemptedGuesses* on the stored registration as we are now revealing sensitive information about the *OPRF* to the client that could allow them to brute force the *accessKey* if attempts are not limited.

An *OK* response from this phase should always be expected to return the following information:
- blindedResult
- maskedTgkShares$_i$ *(from the user's registration record)*

3.5.3 Phase 3

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

The purpose of Phase 3 is to recover the *encryptedSecretShares* allowing decryption and reconstruction of the user's *secret.* Additionally, upon success this phase resets the *attemptedGuesses* on each *Realm$_i$* to 0. For the latter reason, it is important a client completes this process on *all* realms, even if sufficient material has been received to recover the user's *secret.*

The client should perform the following actions:

1. Compute the *oprfResults* using the *blindedResult* from the response and the *accessKey* derived in Phase 2
   - $oprfResults_i = OprfFinalize(blindedResult_i, accessKey)$
2. Unmask the *maskedTgkShares* from the response
   - $tgkShares_i = maskedTgkShares_i\ XOR\ oprfResults_i$
3. Recover *tgk* from *tgkShares*
   - $tgk = RecoverShares(tgkShares)$
   - If the wrong pin was used the client will recover the wrong *tgk*
4. Derive the *unlockTags*
   - $unlockTag_i = MAC(tgk, Realm_{i(id)})$
   - $Realm_{i(id)}$ is the unique 16-byte identifier for each *Realm$_i$* as described in Section 2.1
   - Knowledge of this value proves to the *Realm$_i$* the user knows their *PIN* and allow it to release the *encryptedSecretShare* without revealing the *PIN*
   - If the *unlockTag$_i$* is invalid due to an incorrect *PIN* the client will not know until after confirming with *Realm$_i$*, ensuring both the client and the realm know the outcome of the recovery operation for any auditing purposes

A *recover3* request is then sent from the client to each *Realm$_i$* that contains the previously determined:
- version
- unlockTags$_i$

Upon receipt of a *recover2* request, *Realm$_i$* checks the current state of a user's registration and responds appropriately.

NotRegistered The *Realm* will respond with a *NotRegistered* error to this request.

Registered
- If the *version* on the stored registration does not match the *version* on the request, the *Realm* will respond with a *VersionMistmatch* error to this request.
- If the *unlockTags$_i$* on the stored registration does not match the *unlockTags$_i$* on the request, the *Realm* will respond with a *BadUnlockTag* error to this request.
- Otherwise, the *Realm* will process this request and return an *OK* response.

NoGuesses The *Realm* will respond with a *NoGuesses* error to this request.

If returning an *OK* response, *Realm$_i$* should perform the following actions:

1. Reset the *attemptedGuesses* to 0 as the user has proven they know their *PIN*

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

- $\text{encryptedSecretShares}_i$

If returning a *BadUnlockTag*, *Realm$_i$* should perform the following actions:

1. Compute *guessesRemaining*
   - $guessesRemaining = MAX(allowedGuesses - attemptedGuesses, 0)$
2. If $guessesRemaining = 0$ update the stored record to the *NoGuesses* state preventing future attempts and erasing any stored material

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* by performing the following actions:

1. Reconstruct *encryptedSecret* from *encryptedSecretShares*
   - $encryptedSecret = RecoverShares(encryptedSecretShares)$
2. Decrypted *encryptedSecret* using the previously derived *encryptionKey* from Phase 2
   - $secret = Decrypt(encryptionKey, encryptedSecret, 0)$

3.6 Deletion

Delete is a single-phase process 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.

3.6.1 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 responsd *OK* to this request unless a transient network error occurs.

4 Implementation Considerations

4.1 Post-Quantum and OPRFs

While we have not performed extensive exploration into the state of post-quantum OPRFs, the data stored long-term in a user's registration at rest is sufficiently hardened against quantum attacks. Since the protocol limits replay of any requests that contain OPRF primitives, we believe it may sufficiently render such data useless in a post-quantum environment.

4.2 Registration Generations

Earlier versions of this protocol included a concept of generations. These were needed primarily to prevent the re-use of server-derived OPRF keys, but they also allowed recovering from partial failures in registration. The current protocol does not include generations and assumes that each realm stores either

0 or 1 records per user. This approach is much simpler, but it does have a downside in the specific scenario of a user re-registering (for example, changing their PIN). 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.

5 Recommended Cryptographic Algorithms

5.1 PIN Hashing

The protocol relies on a *Hash* function to add entropy to the user's *PIN*. While the specific hashing algorithm is up to the client, we recommend utilizing *Argon2* [6].

Determining the appropriate configuration parameters for Argon2 is highly dependant on the limitiations of your client hardware. Additionally, since user's 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 smart phones and browsers circa 2019 and determined the following recommended parameters:

- Utilize Argon2id to defend against timing and GPU attacks
- Utilize a parallelism of 1 (limited primarily by browser based threading)
- Utilize 32 iterations
- Utilize 16kbs 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.

5.2 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* [7].

5.3 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 *BLAKE2s-256* [8].

5.4 OPRF Cipher Suite

We recommend utilizing the *Ristretto255* curve as defined by Valence *et al.* [9], but 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 Acknowledgements

- Trevor Perrin for helping design and review the majority of the cryptography details in the protocol
- The Signal Foundation for suggesting many of the approaches used here in the future looking portion of their *"Secure Value Recovery"* technology preview [10]

7 References

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

[2] 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

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

[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. 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

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

[8] 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

[9] 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

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