Juicebox Protocol

Distributed Storage and Recovery of Secrets Using Simple PIN Authentication

Revision 7 — August 7, 2023

Nora Trapp
Juicebox Systems, Inc

Diego Ongaro
Juicebox Systems, Inc

Abstract

Existing secret management techniques demand users memorize complex passwords, store convoluted recovery phrases, or place their trust in a specific service or hardware provider. We have designed 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 an 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 Realms 4
 2.2 Tenants 4
3 Cryptographic Primitives 4
 3.1 Secret-Sharing Scheme (SSS) 5
 3.2 Oblivious Pseudorandom Functions (OPRFs) 5
 3.3 Threshold OPRFs (T-OPRFs) 5
 3.4 Robust T-OPRFs with Zero-Knowledge Proofs (ZKPs) 6
 3.5 Digital Signature Algorithm (DSA) 6
 3.6 Additional Primitives 6
4 Protocol 7
 4.1 Functionality 7
 4.2 Realm State 9
 4.3 Registration 10
  4.3.1 Phase 1 Registration 10
  4.3.2 Phase 2 Registration 10
 4.4 Recovery 12
  4.4.1 Phase 1 Recovery 12
  4.4.2 Phase 2 Recovery 13
  4.4.3 Phase 3 Recovery 15
 4.5 Deletion 16
  4.5.1 Phase 1 Deletion 16
 4.6 Authentication 16
5 Security Considerations 17
 5.1 Threshold Configuration 17
 5.2 Hardware Realms 17
 5.3 Software Realms 18
 5.4 Realm Communication 18
 5.5 Low-Entropy PINs 18
 5.6 Salting 18
6 Recommended Cryptographic Algorithms 18
 6.1 SSS 18
 6.2 OPRFs 18
 6.3 T-OPRFs 18
 6.4 Robust OPRFs with ZKPs 18
 6.5 DSA 19
 6.6 KDF 19
 6.7 Secret Encryption 19
 6.8 MAC 19
7 Acknowledgments 19
8 References 19

1 Introduction

Services are increasingly attempting to provide their users with strong, end-to-end encrypted privacy features, 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 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 or otherwise backup their key material with a third party they trust.

We have designed the *Juicebox Protocol* to solve these problems. It allows the user to recover their secret material by remembering a short PIN, without having access to any previous devices or placing their trust in any single party.

Specifically, this protocol:

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

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

2 Overview

A protocol client distributes 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 2.1.

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.

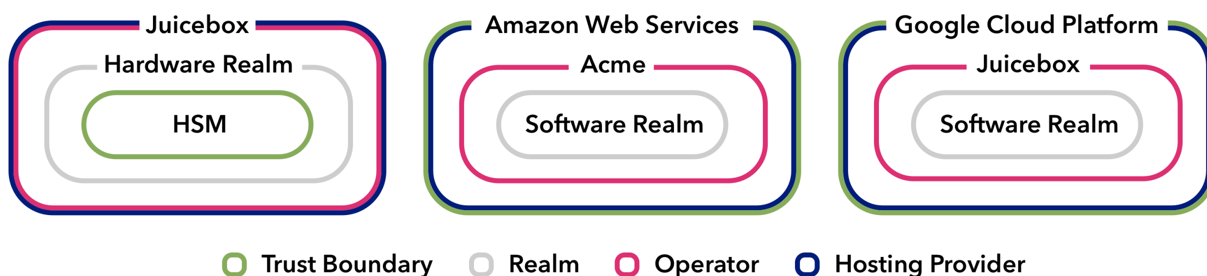

Figure 1: A configuration for a fictional tenant "Acme" demonstrating various realm types, operators, and their trust boundaries.

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 used in *threshold*-based operations. A *threshold* is a definition of how many realms must participate for a secret to be recovered. Configuring a $threshold < n$ allows increased availability of secrets when using a configuration with a larger size since not all realms are required to be operational or in agreement for the operation to succeed.

2.1 Realms

A *realm* is a service capable of storing a distributed share of a user's secret. This section describes the two types of realms that we have implemented and the *trust boundaries* associated with each type.

A *realm* is defined by the following information:

id:
A 16-byte identifier uniquely representing this realm across all configured realms.

index:
An integer from *1..N* that uniquely identifies the realm's position in a configuration.

address:
The fully qualified network address for connecting to the service.

publicKey:
An optional 32-byte public key used to establish secure communications for hardware realms, where the realm controls a matching private key. See Section 5.4 for more details.

A *hardware realm* is a type of realm backed by secure hardware — specifically a hardware security module (HSM). Hardware realms provide tight *trust boundaries* as only the HSM and the code it executes must be trusted. This difference is visible in Figure 1.

A *software realm* is a type of realm that runs on commodity hardware in common cloud providers. When paired with hardware realms, they are a valuable low-cost tool for increasing the number of *trust boundaries* within a configuration.

2.2 Tenants

Each *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 utilize this multi-tenanted approach for realms as it reduces the costs of running realms by dividing 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 Secret-Sharing Scheme (SSS)

A secret-sharing scheme is a cryptographic primitive 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:

$\boldsymbol{CreateShares(n, threshold, secret)}$:
: Distributes a *secret* into an ordered list of *n shares* that can be recovered when *threshold* are provided.

$\boldsymbol{RecoverShares(indexedShares)}$:
: Recovers a *secret* from *y indexedShares* where each contains a corresponding *index* from *1..N* and *share*. If an invalid combination of shares is provided, an incorrect *secret* will be returned that is indistinguishable from random.

3.2 Oblivious Pseudorandom Functions (OPRFs)

An OPRF is a cryptographic primitive that enables a server to securely evaluate a function on a client's input. This evaluation ensures 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:

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

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

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

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

3.3 Threshold OPRFs (T-OPRFs)

A T-OPRF combines OPRFs and a secret-sharing scheme into a hybrid primitive that facilitates a highly-efficient oblivious exchange of an *input* across a *threshold* set of servers, allowing the computation of a single secret *result* without revealing that *result* to the servers. The correctness of the *result* can be verified utilizing a *commitment* computed during evaluation and finalization.

For this paper, we will define a T-OPRF exchange with the following abstract functions:

$\boldsymbol{TOprfKeyShares(n, threshold, rootKey)}$:
: Returns *n* shares of a *rootKey* with the specified *threshold*. Each key share is sent to one server. This function is an alias of *CreateShares*.

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

$\boldsymbol{TOprfBlindEvaluate}(\boldsymbol{keyShare}, \boldsymbol{blindedInput})$:
: Performs the evaluation step for the *blindedInput* and returns the *blindedResult*. This *blindedResult* is sent from each server to the client. This function is an alias of *OprfBlindEvaluate*.

$\boldsymbol{TOprfFinalize}(\boldsymbol{indexedBlindedResults}, \boldsymbol{blindingFactor}, \boldsymbol{input})$:
: Performs the finalization step to unblind the set of *indexedBlindedResults* returned from at least *threshold* servers and returns a *result* and a *commitment* that can be used to validate that *result* at a later point. If incorrect or insufficient shares of *indexedBlindedResults* are present, the returned values will be indistinguishable from random. *TOprfFinalize* can be implemented as *OprfFinalize(RecoverShares(indexedBlindedResults), blindingFactor, input)* and splitting the *result*.

$\boldsymbol{TOprfEvaluate}(\boldsymbol{rootKey}, \boldsymbol{input})$:
: Computes the unblinded *result* and *commitment* directly bypassing the oblivious exchange. The *commitment* can be sent to the servers to validate the *result* received from *TOprfFinalize* at a later date. *TOprfEvaluate* can be implemented as *OprfEvaluate(rootKey, input)* and splitting the *result*.

3.4 Robust T-OPRFs with Zero-Knowledge Proofs (ZKPs)

A ZKP is a cryptographic primitive that enables one party to demonstrate the truth of a statement to another party without revealing any additional information. Integrating ZKPs with T-OPRFs contributes to the protocol's robustness against misbehaving realms.

For this paper, we will define a ZKP with the following abstract functions:

$\boldsymbol{TOprfProofGenerate}(\boldsymbol{privateKey}, \boldsymbol{publicKey}, \boldsymbol{blindedInput}, \boldsymbol{blindedOutput})$:
: Returns a *proof* demonstrating that a server performed *TOprfBlindEvaluate* with a trusted *privateKey*, without revealing the *privateKey*.

$\boldsymbol{TOprfProofVerify}(\boldsymbol{proof}, \boldsymbol{publicKey}, \boldsymbol{blindedInput}, \boldsymbol{blindedOutput})$:
: Validates a *proof* to confirm that a server generated the correct *blindedOutput* from the *blindedInput* using the *privateKey* associated with a trusted *publicKey*.

3.5 Digital Signature Algorithm (DSA)

A DSA is a cryptographic primitive that facilitates the generation and verification of digital signatures. These signatures rely on a key pair, comprised of a private signing key and a corresponding public verifying key. While the signing key is kept confidential and used for signing messages, the verifying key is openly shared to verify signatures.

For this paper, we will define a DSA with the following abstract functions:

$\boldsymbol{NewSigningKey}()$:
: Returns a random private *signingKey*.

$\boldsymbol{GenerateSignature}(\boldsymbol{signingKey}, \boldsymbol{message}, \boldsymbol{context})$:
: Signs the combined *message* and *context* using the *signingKey* and returns a *signedMessage* that contains the *signature*, *verifyingKey*, and original *message*.

$\boldsymbol{VerifySignature}(\boldsymbol{signedMessage}, \boldsymbol{context})$:
: Verifies the *signature* was created with the private *signingKey* associated with the public *verifyingKey* for the specified *message* and *context*.

3.6 Additional Primitives

In addition to the previously established primitives, the following common primitives are necessary to define the protocol:

$Encrypt(encryptionKey, plaintext, nonce)$:

Returns an authenticated encryption of *plaintext* with *encryptionKey*. The encryption is performed with the given *nonce*.

$Decrypt(encryptionKey, ciphertext, nonce)$:

Returns the authenticated decryption of *ciphertext* with *encryptionKey* or an *error* if decryption fails. The decryption is performed with the given *nonce*.

$KDF(data, salt, info)$:

Returns a fixed 64-byte value that is unique to the input *data*, *salt*, and *info*.

$MAC(n, key, *inputs)$:

Returns an *n*-byte tag by combining the *key* with the provided *inputs*. The *MAC* function should encode the inputs unambiguously.

$Random(n)$:

Returns *n* random bytes. The *Random* function should ensure the generation of random data with high entropy, suitable for cryptographic purposes.

$Scalar(seed)$:

Returns a scalar created from a 64-byte seed value.

$PublicKey(scalar)$:

Computes the *point* representing the public key for a given *scalar*.

4 Protocol

The *Juicebox Protocol* can be abstracted into three simple operations:

Register: 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.

1. In phase 1, the client checks that at least *y* realms are available to store the user's secret, where $y \geq threshold$.
2. In phase 2, the client prepares and updates the registration *state* on each $realm_i$.

Recover: A three-phase operation that an existing user takes to recover a PIN-protected secret.

1. In phase 1, the client checks that a *threshold* of realms with consensus on *version* is available to recover the secret.
2. In phase 2, the client performs a T-OPRF exchange on the identified realms using an *input* derived from the user's *PIN*.
3. In phase 3, the client uses the T-OPRF *result* to derive tags for each realm that proves it knows the correct *PIN*. Then, each realm returns information that the client uses to reconstruct the user's secret.

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

4.1 Functionality

The protocol uses the above operations, along with the previously defined primitives, to provide the following functionality:

Prevent online brute-force attacks:

Realms limit the number of unsuccessful recovery attempts to prevent online brute-force attacks against the user's *PIN* (and OPRF key shares). Each realm independently increments an attempted

guess count in phase 2 of recovery and disallows recoveries beyond the limit specified during registration. This counter is reset to zero in phase 3 of recovery if the client proves it knows the correct *PIN*.

Audit events:

In phase 2 of recovery, a threshold of realms can log that a recovery is being attempted. For a successful recovery, a *threshold* of realms can log during phase 3 that the client proved it knew the correct *PIN* before the client could access their secret.

Store arbitrary secrets:

The user stores a secret of their choosing, which provides flexibility to integrate with external systems. The user can also re-register with a new *PIN*, without changing the secret. To provide this flexibility, the recovered secret cannot be based on the T-OPRF *result*, which is pseudo-random and changes with every registration and *PIN*. Instead, the client stores the user's secret in an encrypted format during registration. In phase 3 of recovery, once the client proves it knows the correct *PIN*, it retrieves the encrypted secret, along with shares of a component of the encryption key.

Increase offline brute-force costs:

As an additional layer of security, the client generates a unique salt per registration and combines this with the user's *PIN* using a KDF. A portion of the KDF output is utilized as the T-OPRF input and as a seed for the key used to encrypt a user's secret. An adversary who has compromised a threshold of realms would need to additionally spend resources to brute-force each user's *PIN* to access the secret. Using a resource-intensive KDF further increases the adversary's costs. The random salt also serves as the registration's version and is retrieved during phase 1 of recovery.

Exclude misbehaving realms:

The protocol adds a layer of robustness to detect misbehaving realms and prevent them from interfering with recovery. This allows the client to proceed with recovery as long as a threshold number of correct realms are available, even if some realms are returning incorrect or adversarial results. It also allows the client to distinguish an incorrect PIN from incorrect realm behavior (otherwise, the T-OPRF *result* appears incorrect for both cases).

The protocol combines a few techniques to provide robustness:

- The client generates a signature of each realm's public OPRF key and ID during registration, then discards the signing key (so that it cannot be used again). During phase 2 of recovery, the client identifies a *threshold* of realms that agree on a verifying key with valid signatures for their public OPRF key corresponding to that verifying key.
- Each realm generates a ZKP in phase 2 of recovery. The proof allows the client to verify that the realm computed its share of the T-OPRF consistently with its OPRF key share.
- During registration, the client unobliviously evaluates the T-OPRF for the user's *PIN* using the root OPRF key. A copy of the resulting public *commitment* is stored with each realm. During phase 2 of recovery, the client identifies a threshold of realms that agree on a *commitment* to recover from. It then verifies that the T-OPRF finalization produces a matching *commitment*. This prevents a colluding *threshold* of realms from substituting different OPRF key shares and signatures without knowing the *PIN*.
- The client stores a secondary *commitment* based on the T-OPRF *result*, the encrypted secret, and that realm's share of the encryption key with each realm during registration. During phase 3, the client verifies that the information about the user's secret returned from each realm is consistent with the *commitment* stored during registration.

4.2 Realm State

Realm$_i$ will store a record indexed by the combination of the registering user's identifier (UID, as defined in Section 4.6) and their *tenant*'s name. This ensures that a given *tenant* may only authorize operations for its users.

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 recover that registration.

NoGuesses:
The user has registered secret information with this *realm*, but can no longer attempt to recover 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 16-byte value that uniquely identifies this registration for this user across all configured *realms*. The version is random so that a malicious realm can't force a client to run out of versions. The version is also used as a salt that is combined with the user's PIN.

oprfPrivateKeys$_i$:
A realm-specific OPRF private key derived by secret sharing a random root key.

unlockKeyCommitment:
A 32-byte value derived during registration from the OPRF result used to verify the *unlockKey*.

oprfSignedPublicKey$_i$:
A signed public key that corresponds to the *oprfPrivateKeys$_i$* for this realm. Wraps the *publicKey*, a *signature*, and the public *verifyingKey* for the *signingKey* used to generate the *signature*.

unlockKeyTag$_i$:
A tag unique to this *realm$_i$* derived during registration from the *unlockKey*. The client will reconstruct this tag during recovery to prove knowledge of the *PIN* and be granted access to the secret.

encryptionKeyScalarShares$_i$:
A single share of the random scalar used to derive the *encryptionKey* for the user's secret. Even if *threshold* shares were recovered, the *encryptionKey* cannot be derived without knowing the user's *PIN*.

encryptedSecret:
A copy of the user's encrypted secret.

encryptedSecretCommitment$_i$:
A MAC derived from the *unlockKey*, *realm$_{id}$*, encryptionKeyScalarShares$_i$, and *encryptedSecret*. During recovery, the client can reconstruct this MAC to verify if *realm$_i$* has returned a valid share and secret.

allowedGuesses:
The maximum number of guesses allowed before the registration is permanently deleted by the *realm*$_i$.

attemptedGuesses$_i$:
The number of times recovery has been attempted on *realm*$_i$ without success. Starts at 0 and increases on recovery attempts, then reset to 0 on successful recoveries.

4.3 Registration

The registration operations are exposed by the client in the following form:

$$error = register(pin, secret, allowedGuesses, userInfo)$$

pin:
This argument contains a potentially low entropy value known to the user that will be used to recover their secret, such as a 4-digit PIN.

secret:
This argument contains the secret value a user wishes to persist.

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

userInfo:
This argument contains per-user data that is combined with the random *salt* used to stretch the user's *PIN*.

error:
An error in registration, such as insufficient available realms.

The following sections contain Python code that demonstrates in detail the work performed by each phase.

For this code, we assume that the protocol client has been appropriately configured with:

- n mutually distrusting realms, each of which will be referred to as *realm*$_i$
- $threshold \leq n$ indicating how many realms must be available for recovery to succeed

4.3.1 Phase 1 Registration

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

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

Once a client has completed *Phase 1* on y realms, it will proceed to Phase 2.

4.3.2 Phase 2 Registration

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

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

  stretchedPin = KDF(pin, version, userInfo)
  accessKey = stretchedPin[0:32]
  encryptionKeySeed = stretchedPin[32:64]

  oprfRootPrivateKey = Scalar(Random(64))
```

```
oprfPrivateKeys = TOprfKeyShares(len(realms), threshold, oprfRootPrivateKey)

unlockKeyCommitment, unlockKey = TOprfEvaluate(oprfRootPrivateKey, accessKey)

signingKey = NewSigningKey()

oprfSignedPublicKeys = [GenerateSignature(signingKey, PublicKey(key), realm.id)
                        for key, realm in zip(oprfPrivateKeyShares, realms)]

encryptionKeyScalar = Scalar(Random(64))
encryptionKeyScalarShares = CreateShares(len(realms),
                                         threshold,
                                         encryptionKeyScalar)
encryptionKey = MAC(32,
                    encryptionKeySeed,
                    "Encryption Key",
                    encryptionKeyScalar)

# A `nonce` of 0 can be used since `encryptionKey` changes with each registration
encryptedSecret = Encrypt(secret, encryptionKey, 0)

encryptedSecretCommitments = [MAC(16,
                                  unlockKey,
                                  "Encrypted Secret Commitment",
                                  realm.id,
                                  share,
                                  encryptedSecret)
                          for realm, share in zip(realms, encryptionKeyScalarShares)]

unlockKeyTags = [MAC(16, unlockKey, "Unlock Key Tag", realm.id)
                 for realm in realms]

return (
    version,
    oprfPrivateKeys,
    unlockKeyCommitment,
    oprfSignedPublicKeys,
    encryptionKeyScalarShares,
    encryptedSecret,
    encryptedSecretCommitments,
    unlockKeyTags
)
```

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

- version
- oprfPrivateKeys_i
- unlockKeyCommitment
- $\text{oprfSignedPublicKeys}_i$
- $\text{encryptionKeyScalarShares}_i$
- encryptedSecret
- $\text{encryptedSecretCommitments}_i$

- unlockKeyTags$_i$
- allowedGuesses

Upon receipt of a *register2* request, $realm_i$ creates or overwrites the user's registration state with the corresponding values from the request and resets the *attemptedGuesses* to 0.

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

The registration operation completes successfully with at least y *OK* responses.

4.4 Recovery

The recovery operations are exposed by the client in the following form:

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

pin:

This argument represents the same value used during *register*.

userInfo:

This argument represents the same value used during *register*.

secret:

The recovered secret, as provided during registration, if and only if the correct *PIN* was provided and no *error* was returned.

error:

An error in recovery, such as an invalid *PIN* or the *allowedGuesses* having been exceeded.

The following sections contain Python code that demonstrates in detail the work performed by each phase.

For this code, we assume that the protocol client has been appropriately configured with:

- n mutually distrusting realms, each of which will be referred to as $realm_i$
- $threshold \leq n$ indicating how many realms must be available for recovery to succeed

4.4.1 Phase 1 Recovery

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

The following demonstrates the work a $realm_i$ performs 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)
  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

Once a client has completed Phase 1 on at least *threshold* $realm_i$ that agree on *version*, it will proceed to Phase 2 for those realms. If no realms are in agreement, the client will assume that the user is *NotRegistered* on any realm.

4.4.2 Phase 2 Recovery

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

```
def PrepareRecovery2(pin, userInfo, realms, version, salt):
  stretchedPin = KDF(pin, salt, userInfo)
  accessKey = stretchedPin[0:32]
  encryptionKeySeed = stretchedPin[32:64]

  blindedAccessKey, blindingFactor = TOprfBlind(accessKey)

  return (
    accessKey,
    encryptionKeySeed,
    blindedAccessKey,
    blindingFactor
  )
```

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

- version
- blindedAccessKey

The following demonstrates the work a $realm_i$ performs 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()

    blindedResult = TOprfBlindEvaluate(state.oprfPrivateKey, request.blindedAccessKey)
    blindedResultProof = TOprfProofGenerate(state.oprfPrivateKey,
                                            state.oprfPublicKey,
                                            request.blindedAccessKey,
                                            blindedResult)

    state.attemptedGuesses += 1

    return Ok(blindedResult,
              blindedResultProof,
              state.oprfSignedPublicKey,
              state.unlockKeyCommitment,
              state.allowedGuesses,
              state.attemptedGuesses)
  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
- blindedResultProof
- $\text{oprfSignedPublicKeys}_i$
- unlockKeyCommitment
- allowedGuesses
- attemptedGuesses

This phase will proceed until a client has completed it on at least *threshold* $realm_i$ that:

1. agree on an *unlockKeyCommitment* and *verifyingKey* for the *oprfSignedPublicKeys*
2. each have a valid $oprfSignedPublicKeys_i$:
   ```
   VerifySignature(oprfSignedPublicKey, realm.id)
   ```
3. each have a valid *blindedResultProof* for the *blindedResult*:
   ```
   TOprfProofVerify(
     blindedResultProof,
     oprfSignedPublicKey.publicKey,
     blindedAccessKey,
     blindedResult
   )
   ```

If this cannot be completed on *threshold* realms, the client may need to re-register or try again later.

If completed successfully, the client will attempt to recover the *unlockKey*.

The following demonstrates the work the client performs to reconstruct and validate the *unlockKey*:

```
def RecoverUnlockKey(
  blindingFactor,
  accessKey,
  unlockKeyCommitment,
  blindedResults,
  allowedGuesses,
  attemptedGuesses,
  realms
):
  indexedBlindedResults = [(realm.index, result)
                           for realm, result in zip(realms, blindedResults)]
  ourUnlockKeyCommitment, unlockKey = TOprfFinalize(
    indexedBlindedResults,
    blindingFactor,
    accessKey
  )

  if ConstantTimeEquals(ourUnlockKeyCommitment, unlockKeyCommitment):
    return unlockKey
  else:
    guessesRemaining = min([x - y for x, y in zip(allowedGuesses, attemptedGuesses)])
    return Error.InvalidPin(guessesRemaining)
```

If the *unlockKey* could be recovered successfully, the client will proceed to Phase 3. Otherwise, an *InvalidPin* error will be returned by the client.

4.4.3 Phase 3 Recovery

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

```
def PrepareRecovery3(realms, unlockKey):
  unlockKeyTags = [MAC(16, unlockKey, "Unlock Key Tag", realm.id)
                   for realm in realms]
  return unlockKeyTags
```

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

- version
- unlockKeyTags_i

The following demonstrates the work a $realm_i$ performs to process the request:

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

    if !ConstantTimeEquals(request.unlockKeyTags, state.unlockKeyTags):
      guessesRemaining = state.allowedGuesses - state.attemptedGuesses

      if guessesRemaining == 0:
        state.transitionToNoGuesses()

      return Error.BadUnlockKeyTag(guessesRemaining)

    state.attemptedGuesses = 0

    return Ok(state.encryptionKeyScalarShare,
              state.encryptedSecret,
              state.encryptedSecretCommitment)
  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:

- $\text{encryptionKeyScalarShares}_i$
- encryptedSecret
- $\text{encryptedSecretCommitments}_i$

A *BadUnlockKeyTag* 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 performs to do so:

```
def RecoverSecret(encryptionKeySeed,
                  encryptionKeyScalarShares,
                  encryptedSecret,
```

```
                encryptedSecretCommitments,
                realms,
                threshold):
    indexedEncryptionKeyScalarShares = []

    for share, commitment, realm in
      zip(encryptionKeyScalarShares,
          encryptedSecretCommitments,
          realms):
      ourCommitment = MAC(16,
                          unlockKey,
                          "Encrypted Secret Commitment",
                          realm.id,
                          share,
                          encryptedSecret)
      if ConstantTimeEquals(ourCommitment, commitment):
        indexedEncryptionKeyScalarShares.append((realm.index, share))

    if len(validEncryptionKeyScalarShares) < threshold:
      return Error.Assertion()

    encryptionKeyScalar = RecoverShares(indexedEncryptionKeyScalarShares)
    encryptionKey = MAC(32,
                        encryptionKeySeed,
                        "Encryption Key",
                        encryptionKeyScalar)

    secret = Decrypt(encryptionKey, encryptedSecret, 0)
    return secret
```

4.5 Deletion

The delete operation is exposed by the client in the following form:

$$error = delete()$$

error:

An error in delete, such as a transient network error.

This operation does not require the user's *PIN* as a user can always register a new secret effectively deleting any existing secrets.

4.5.1 Phase 1 Deletion

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.

4.6 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) [4] 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 *$realm_{i(id)}$*
4. Don't contain an *iss* matching the *tenantName* in the *kid*

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 Security Considerations

5.1 Threshold Configuration

While any $threshold \leq n$ is valid, we recommend a $threshold > \frac{n}{2}$ which ensures that there can be only at most one valid secret for a user at a time, avoiding uncertainty during Phase 1 of recovery.

Additionally, a $threshold > 1$ (and consequently $n > 1$) should always be used, as the security guarantees this protocol provides only apply when secrets are distributed across multiple realms.

5.2 Hardware Realms

We specifically utilize HSMs that are programmable with non-volatile memory. 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 the self-destruction of secrets. The HSMs we use 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.

Hardware realms assume that a combination of relatively opaque hardware and firmware is secure, which — outside of the *Juicebox Protocol* — makes them not ideal as a standalone secret storage solution. However, when used in configuration with other types of realms — including hardware realms from other vendors — these risks can be mitigated.

5.3 Software Realms

Since these realms only control an encrypted share of a user's secret, we believe it is an acceptable tradeoff that they require extending the *trust boundary* to include the realm's *operator* and *hosting provider*. It is important to recognize that given the limited number of distinct *hosting providers* currently operating, overuse of such realms can potentially put too much secret information in one party's control and jeopardize user secrets.

5.4 Realm Communication

Communication with a *realm* always occurs over a secure protocol that ensures the confidentiality and integrity of requests while limiting the possibility of replay attacks. Towards this end, all requests to a realm are made over TLS.

Hardware realms terminate this TLS connection outside of their *trust boundary*. This allows a single load balancer to service multiple HSMs but necessitates an additional layer of secure communication between the client and the HSM. For this layer, we use the *Noise Protocol* [5] with an NK-handshake pattern with the realm's *public key*. Specifically, we use the protocol name `Noise_NK_25519_ChaChaPoly_BLAKE2s`.

5.5 Low-Entropy PINs

While the protocol provides strong security guarantees for low entropy PINs, using a higher entropy PIN provides increased security if a *threshold* of realms was compromised.

5.6 Salting

The *register* and *recover* operations accept a *userInfo* argument that is mixed into the *salt* before passing it to the *KDF*. Using a known constant, like the UID, for this value can prevent a malicious *realm* from returning a fixed *salt* with a pre-computed password table.

6 Recommended Cryptographic Algorithms

6.1 SSS

The protocol relies on a secret-sharing scheme to ensure a *realm* does not gain access to the user's secret. We utilize the scheme defined by Shamir [6] over the Ristretto255 group [7].

6.2 OPRFs

The protocol utilizes OPRFs based on 2HashDH as defined by Jarecki *et al.* [8]. These operations are performed over the Ristretto255 group [7] and utilize the SHA-512 hashing algorithm [9].

6.3 T-OPRFs

The protocol utilizes T-OPRFs based on 2HashTDH as defined by Jarecki *et al.* [10]. These operations are performed over the Ristretto255 group [7] and utilize the SHA-512 hashing algorithm [9].

6.4 Robust OPRFs with ZKPs

The protocol utilizes ZKPs to allow a client to verify the server used a specific private key during the execution of the protocol. Our implementation of these proofs specifically utilizes a Chaum-Pedersen DLEQ with a Fiat-Shamir transform, as defined by Jarecki *et al.* [8]. These operations are performed over the Ristretto255 group [7] and utilize the SHA-512 hashing algorithm [9].

6.5 DSA

The protocol utilizes a cryptographic signature to validate the public key used by each *realm*. We utilize Ed25519, an Edwards-curve Digital Signature Algorithm (EdDSA) [11].

6.6 KDF

The protocol relies on a *KDF* function to add entropy to the user's *PIN*. When an expensive *KDF* is utilized, this provides an additional layer of protection for low entropy PINs if a *threshold* of realms were to be compromised. For this reason, we utilize *Argon2* [12].

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. This only works if you can assure new devices for that user only get more performant, which may be difficult to guarantee.

For the common case, we have evaluated performance across popular smartphones and browsers circa 2019 and defined the following recommended parameters:[1]

- 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.

6.7 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. We utilize *ChaCha20* and *Poly1305* [13].

6.8 MAC

The protocol relies on a *MAC* function to compute various values for future validation. We utilize *BLAKE2s-MAC-256* [14].

7 Acknowledgments

- The protocol is heavily based on design and feedback from Trevor Perrin and Moxie Marlinspike.
- 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 [15].

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 (GitHub)

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

[2] C. Brand, A. Czeskis, et al., “Client to authenticator protocol (ctap),” 2019. [Online]. Available: https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html (FIDO Alliance)

[3] “Juicebox-systems github organization.” [Online]. Available: https://github.com/juicebox-systems

[4] M. B. Jones, J. Bradley, and N. Sakimura, “Json web token (jwt),” RFC Editor, 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7519 (RFC 7519)

[5] T. Perrin, “The noise protocol framework,” 2018. [Online]. Available: https://www.noiseprotocol.org/noise.html

[6] A. Shamir, “How to share a secret,” *Commun. ACM*, vol. 22, pp. 612–613, 1979.

[7] H. de Valence, J. Grigg, et al., “The ristretto255 and decaf448 groups,” Internet Engineering Task Force, 2023. [Online]. Available: https://datatracker.ietf.org/doc/draft-irtf-cfrg-ristretto255-decaf448/07/

[8] S. Jarecki, A. Kiayias, and H. Krawczyk, “Round-optimal password-protected secret sharing and t-pake in the password-only model,” 2014. [Online]. Available: https://eprint.iacr.org/2014/650 (Cryptology ePrint Archive, Paper 2014/650)

[9] T. Hansen, and D. E. E. 3rd, “Us secure hash algorithms (sha and sha-based hmac and hkdf),” RFC Editor, 2011. [Online]. Available: https://www.rfc-editor.org/info/rfc6234 (RFC 6234)

[10] S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu, “Toppss: cost-minimal password-protected secret sharing based on threshold oprf,” 2017. [Online]. Available: https://eprint.iacr.org/2017/363 (Cryptology ePrint Archive, Paper 2017/363)

[11] S. Josefsson, and I. Liusvaara, “Edwards-curve digital signature algorithm (eddsa),” RFC Editor, 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8032 (RFC 8032)

[12] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, “Argon2 memory-hard function for password hashing and proof-of-work applications,” RFC Editor, 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9106 (RFC 9106)

[13] Y. Nir, and A. Langley, “Chacha20 and poly1305 for ietf protocols,” RFC Editor, 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7539 (RFC 7539)

[14] M.-J. O. Saarinen, and J.-P. Aumasson, “The blake2 cryptographic hash and message authentication code (mac),” RFC Editor, 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7693 (RFC 7693)

[15] J. Lund, “Technology preview for secure value recovery,” Signal Blog, 2019. [Online]. Available: https://signal.org/blog/secure-value-recovery/