BLINDHASH TECHNICAL SPECIFICATIONS WHITEPAPER

BY JEREMY SPILMAN

Table of Contents

Purpose & Scope

This document is intended for technical audiences to provide a complete description of the BlindHashing process as well as implementation details of each individual component in the processing pipeline. The following sections will identify all the steps required for both client and server to complete a BlindHashing request.

This document should allow a third party to implement a BlindHashing client library capable of securely performing BlindHashing to protect password verifiers from offline attack. This document should also allow a third party to create a matching or 'compliant' BlindHashing implementation which, provided the same data pool and application tokens, can then generate for any 'Hash1' input, the same exact 'Salt2' output as BlindHash'd own servers. Test vectors showing expected inputs/outputs for a given application token against a known data pool can be provided.

Areas which this document will not address are the design, management, and operation of the underlying compute, network, and storage systems themselves. Operational concerns such as datacenter logistics, network design, system hardening, software deployment, patch management, resource monitoring, event logging, intrusion detection, failover and load balancing, etc. are not addressed by this document.

BlindHashing Overview

The "client" is a customer's machine which is performing password authentication for a user login. The client machine will perform some hashing of the user's password with a secure random salt, in order to generate a 'Hash1' value. A BlindHashing request containing a client identifier (AppID), 'Hash1' value, and a 'Version ID' is sent from the client to the BlindHashing server via HTTPS.

A 'Request Handler' receives a client request, and parses the request to determine the Client IP, 'AppID', 'Hash1', and 'Version' values. The request is authorized based on the 'AppID' and Client IP, and then dispatched to the appropriate 'Data Pool Handler' via a single-packet UDP request over a dedicated secure channel ('spiped').

Data Pool Handlers receive UDP packets containing data pool read requests from the Request Handlers and calculate the necessary indices to read from the data pool volumes. As reads are completed, they are transformed from raw data pool bits into the so-called 'virtual private data pool' bits using HMAC with a 64-byte key associated with the specified AppID. The Data Pool Handler then assembles the transformed reads into a buffer for a final hashing step. Once all reads are complete, the final hash is performed and the resulting digest ('Salt2') is returned to the 'Request Handler' via UDP over a dedicated secure channel.

The ‘Request Handler’, upon receiving the UDP response from the ‘Data Pool Handler’, assembles a JSON-formatted response to be returned to the client, and then returns the response to the client via the existing TLS channel.

The client, upon receiving the JSON response with the ‘Salt2’ value, performs the final HMAC with ‘Hash1’ and ‘Salt2’ in order to calculate the desired ‘Hash2’ value, and then compares the calculated ‘Hash2’ against the stored ‘Hash2’ value to determine if the supplied password is a match.

The following sections will describe the specific implementation and design considerations for each of these steps in detail.

Client-Side Processing

Overview

A client performing password based authentication of a user’s credentials receives a username and password over a web form submitted via HTTPS using a secure TLS channel. The client queries a back-end database with the username to determine if an account exists, and if so retrieves the associated ‘Salt1’, ‘Hash2’, and hashing meta-data for that username.

Hashing meta-data stored with the user record may specify the desired hashing function and cost factors used by this pre-hashing step. The meta-data may also include a 32-bit integer version number which must be included in the BlindHashing request.

The client first performs a pre-hashing step using the ‘Salt1’ and ‘Password’ in order to calculate ‘Hash1’. Next, a BlindHashing request is dispatched to BlindHash’s servers by making an HTTPS request including the client’s AppID as a 64-byte hex encoded string, the ‘Hash1’ value as a hex-encoded string, and the BlindHashing Version ID.

The JSON-encoded response is parsed to retrieve a 64-byte hex encoded ‘Salt2’, as well as an optional decimal ‘Version ID’, decimal ‘New Version ID’, and a 64-byte hex encoded ‘New Salt2’. The ‘Salt2’ value is converted to 64-byte binary value and HMAC’ed with the 64-byte binary ‘Hash1’ to calculate a ‘Hash2’, which is compared against the ‘Hash2’ stored in the user database. If the ‘Hash2’ values are equal, then the user is successfully authenticated.

If ‘New Version ID’ and ‘New Salt2’ values are included in the BlindHashing response, and only if the user is successfully authenticated, then the ‘New Salt2’ value is converted to a 64-byte binary value and HMAC’ed with the 64-byte binary ‘Hash1’ to calculate a new ‘Hash2’ value which is saved in the user database in place of the existing ‘Hash2’ value, along with the ‘New Version ID’ integer, in place of any existing ‘Version ID’ stored with the hashing meta-data.

Application ID

The ‘Application ID’ or AppID is a 64-byte randomly generated token assigned to the client by the BlindHashing server. Each client may have one or more AppIDs under their account, where an AppID identifies the client making the request, as well as a set of configuration parameters associated with the

AppID. The AppID is considered a shared secret between the client and BlindHashing server and is used to authenticate the client.

A given client may configure multiple 'Applications' and be issued separate AppIDs for each one. Separate AppIDs can be used to provide a degree of separation in terms of authenticating, authorizing, and accounting for any BlindHashing requests.

The configurable settings of an AppID include most importantly, the size the of the data pool used for any BlindHashing requests. Additional settings include, for example, an IP whitelist of allowed clients, rate limits, rate notification thresholds, and preferred server lists.

Pre-Hashing

After retrieving the user's 'Salt1' and 'Hash2' values from the database and before dispatching the BlindHashing Request, the client must perform a *pre-hashing* step. This pre-hashing is intended to ensure the 'Hash1' value sent to the BlindHashing server is an independent and identically distributed uniformly random value, which cannot be used on its own to distinguish a user's password, or perform a dictionary attack against the user's password, regardless of the entropy of the password itself.

At a minimum, the pre-hashing step must use a cryptographically secure hashing algorithm, and generally will combine the user's password with the secure random salt using the HMAC construct.

The 'Salt1' value must be at least 16 bytes of CS-PRNG output which is uniquely generated for each user at the time the password was set. The recommended hashing function to perform this initial step is single iteration of HMAC-SHA512.

There may also be some key stretching applied at this time, for example using 'scrypt', 'bcrypt', or 'PBKDF2'.

Key Stretching

Beyond a minimal single HMAC invocation, the client may also perform some amount of key stretching in the process of calculating the 'Hash1' value. This key stretching may be in place purely as a historical feature, in other words, key stretching which was performed to calculate 'Hash1' values before BlindHashing was even introduced.

Any amount of key stretching may be performed in the process of generating 'Hash1' as long as a minimum of 128 bits of entropy are present in the final hash (salt plus password).

When upgrading an existing database of hashed passwords with BlindHashing, as long as the existing Hash1 values are at least 128-bits, and were calculated using a secure random salt with at least 128-bits, the Hash1 values can be used as-is, allowing the Hash1 values to be blinded offline (without the user's password).

A typical formulation of BlindHashing with key stretching would be;

```
Hash1 = KeyStretch(Salt1, password, costFactor)
Salt2 = BlindHash(AppID, Hash1)
Hash2 = HMAC(Salt2, Hash1)
```

In this case, key stretching is completed prior to issuing the BlindHashing request. This means that total latency is the sum of key stretching latency with BlindHashing latency.

An alternative structure performs key stretching asynchronously while the BlindHashing request is being issued, which reduces the latency to the slower of the key stretching latency or the BlindHashing latency.

Hash1 = KeyStretch(Salt1, password, costFactor)
Hash1' = HMAC-SHA512(Salt1, password)
Salt2 = BlindHash(AppID, Hash1')
Hash2 = HMAC(Salt2, Hash1)

In this case the Hash1' used for issuing the BlindHashing request is the unstretched HMAC of password and salt, and then the BlindHashing request can be made while the key stretching is still completing. Once both key stretching and BlindHashing are complete, the final Hash2 value is the HMAC of Salt2 and the stretched Hash1.

The primary advantage of this approach is to allow any key stretching which is deemed necessary by the customer to occur concurrently with the BlindHashing request. A disadvantage of this approach is that the un-stretched hash is sent over the network, providing somewhat less protection against a compromised TLS channel. One caveat with performing asynchronous key-stretching is that it cannot be used when upgrading existing hashes with BlindHashing. In other words, if you have a set of existing Hash1 values and you must perform an offline upgrade of those existing hashes with BlindHashing, then by definition this will mean the key stretching must be run prior to the BlindHashing request.

BlindHashing Request

A BlindHashing request specifies a 64-byte AppID, an up-to 64-byte Hash1 value, and optionally a 32-bit integer version number. The AppID is a static token, which is randomly generated by the BlindHashing server when the 'Application' is first created.

The version number is a value returned from a previously completed BlindHashing which indicates the configured size of the data pool at the time the request was made.

The request can be formatted as an HTTPS GET request against a BlindHashing server, with the URL path of "/<AppID>/<Hash1>" or "/<AppID>/<Hash1>/<Version>". The AppID and Hash1 values are hex-encoded, and the Version is decimal encoded.

BlindHashing Response

A BlindHashing response for a successfully completed request will indicate an HTTP status of '200 OK', and a JSON-encoded response body. Each response contains a 64-byte hex encoded 'Salt2', and a 32-bit integer 'Version ID'.

A successful response may optionally contain an additional 64-byte hex encoded 'New Salt2', and a 32-bit integer 'New Version ID'. See: 'Handling Data Pool Upgrades' for details about these fields.

A request may be unsuccessful for a number of reasons;

- Malformed HTTP request
- Missing, malformed, or invalid ‘AppID’
- Unauthorized request for a valid ‘AppID’
- Missing, malformed, or invalid ‘Hash1’
- Internal server error
- Network failure

An unsuccessful request will result in either an HTTPS response with a 5xx status code, or a timeout waiting for a response.

Final Hashing Step

Given a ‘Salt2’ value from a completed BlindHashing request, the final hashing step will key the hash based on ‘Salt2’ to ensure that the stored hash value is not useable to verify a password without the ‘Salt2’ value. In this case, we can use a single iteration of HMAC-SHA512(‘Salt2’, ‘Hash1’) to key the hash.

Note that we are using ‘Salt2’ to key the ‘Hash1’ value, rather than the password string itself. There are operational advantages to using ‘Hash1’ instead of ‘Password’ for the final hashing step. First, it allows the client to clear the user’s password from memory as soon as ‘Hash1’ is calculated and before dispatching the BlindHashing request. Second, it allows applying a Blind Hash offline (where the user password is not present) to an existing database of ‘Hash1’ values.

However, by calculating ‘Hash2’ purely using ‘Hash1’ and ‘Salt2’, this means that the BlindHashing server, knowing both ‘Hash1’ and ‘Salt2’ itself, also could know the ‘Hash2’ value which is stored in the client database. Alternatively, we could include the ‘Salt1’ value as part of the final hashing step, e.g. HMAC-SHA512(‘Salt2’, ‘Salt1’ || ‘Hash1’), or similar, which would ensure the BlindHashing server would not have any knowledge of the client’s stored ‘Hash2’ value, but it is not clear what potential attack this would be defending against.

Handling Data Pool Upgrades

As the data pool stored by the BlindHashing server is expanded, the maximum configurable size of the data pool for any given AppID is likewise increased. The configured data pool size for a given AppID may then be increased either automatically, or manually at some later date. When the configured data pool size of an AppID is changed, the new data pool size is stored associated with a monotonically incrementing version number.

For example, if the current data pool is 16TB, all newly created AppIDs will start with a configured data pool size of 16TB and a Version ID counter equal to 1. If in the future the physical data pool is expanded to 32TB, the existing AppIDs can now be configured with a data pool size up to 32TB. When an AppID is reconfigured to actually use the new 32TB maximum size, the Version ID for that AppID is incremented, and the new data pool size is stored for that AppID associated with the new Version ID. So, in this example, the AppID’s configured set of data pool sizes would be: { “1 16TB”, “2 32TB” }.

If no Version ID is specified in a BlindHashing request, or the Version ID that was specified in the request matches the latest configured version for that AppID, then the BlindHashing request is processed only

using the latest configured size of the data pool, and the response will include only a single 'Salt2' result, along with the 'Version ID' of the latest configured data pool size.

Generally, only new or first-time BlindHashing requests will be sent without a version number (e.g. upgrading existing hashes, enrolling a new user, or storing a password change or password reset). All other BlindHashing requests must specify the Version ID which they were initially run against, in order to ensure a consistent 'Salt2' response.

If a VersionID is specified in the request which is *not* the latest version configured for that AppID, then the BlindHashing request is processed both against the specified version, as well the latest existing version configured for that AppID – effectively two requests processed concurrently by the server. The response will then include both the 'Salt2' and 'Version ID' for the requested version, as well as a 'New Salt2' and 'New Version ID' values corresponding to the latest configured settings for that AppID.

The 'New Salt2' and 'New Version ID' fields in the response indicate to the client that a new (presumably larger) data pool size has been configured since the original Blind Hash was performed. Only if the client is successfully authenticated using the first 'Salt2' value to arrive at the expected 'Hash2' result, then the 'New Salt2' value can then be used to re-blind the hash, and finally, the new 'Hash2' and 'Version ID' values for the authenticated user can be updated in the client database.

Encrypting Hash1 Values

Customers have expressed concern with lock-in to the BlindHashing service – in other words, they desire an ability to un-blind their hashes without requiring the user's password in memory, and without having to reset passwords. Customers, in essence, want the ability to recover the original 'Hash1' value for the user's password if needed.

To keep the 'Hash1' value in a database would undermine the entire security premise of BlindHashing. If the 'Hash1' value can be retrieved, the BlindHashing step is no longer required in order to attempt to verify a password, and an offline attack becomes possible.

Keeping 'Hash1' in an encrypted form would only be reasonable if; (1) the ciphertext of 'Hash1' is proveably useless for verifying if a password guess is correct, and (2) the key to decrypt 'Hash1' can be kept offline.

In order to satisfy these points, we recommend encrypting 'Hash1' with an asymmetric algorithm which provides indistinguishability under the chosen plaintext attack (IND-CPA). The unencrypted 'Hash1' value must not be stored, and the private key to allow decryption of 'Hash1' must be kept offline under strict physical security.

Given an encrypted 'Hash1' it also becomes possible to re-blind hashes offline, for example after the data pool size is increased, or when switching to a new AppID if a client-side database breach is discovered.

Server-Side Processing

The BlindHashing service is provided by two distinct components; a 'Request Handler' and a 'Data Pool

Handler'. The following sections will describe in detail how these components perform the steps required to complete a BlindHashing request.

Request Handler

The Request Handler is the front-end of the BlindHashing service, and as such receives HTTPS requests from possibly malicious clients, and must parse these requests, perform authentication, authorization, and accounting, and ultimately provide BlindHashing responses back to the clients.

A large number of Request Handlers may be running behind a suitable L4 or L7 load balancer, and requests from the same or different clients may be processing concurrently across any number of Request Handlers.

The processing steps the Request Handler performs are to;

- Receive BlindHashing requests over HTTPS,
- Parse the specified AppID, Hash1, and Version ID values from the request,
- Authenticate the client making the request,
- Authorize the request based on the AppID, Client IP, and any specified rate limits,
- Increment AppID-specific counters to account for the request,
- Log any unauthorized access and dispatch any appropriate notifications of such,
- Determine the configured organization key, data pool volume, data pool size, and data pool read count in order to complete the request
- Dispatch one or two requests to the appropriate Data Pool Handlers to complete the request
- Wait for response(s) from the Data Pool Handler, and return an HTTP error upon timeout
- Return a JSON-formatted response with the 'Salt2', 'Version ID', and optionally 'New Salt 2' and 'New Version ID' in the response body.

Request Parsing

An HTTPS / TLS session is established between the client machine and our Request Handler, and an HTTP GET request is issued with a URL path of '/<AppID>/<Hash1>/<Version>'. The AppID and Hash1 values are mandatory hex-encoded strings, where AppID must be exactly 64 bytes and Hash1 must be at least 16 bytes and no more than 64 bytes. The Version, if specified, must be a decimal string between 0 and 2^{32}-1.

To parse the response, we split the URL path by the '/' character, and expect either 2 or 3 resulting fields. The first field must be exactly 128 characters ([A-Fa-f0-9]) and convert to 64 bytes of data. The second field must be between 32 and 128 characters inclusive ([A-Fa-f0-9]) and convert to between 16 and 64 bytes of data. The third field, if present, must convert into a 32-bit unsigned integer without overflow. A violation of the parsing constraints results in an HTTP error code response, with an error message indicating the type of parsing error, but without reflecting the requested values in the error response.

There are a great many crucial details around the underlying process of making such a secure, performant, and valid HTTP request which we will take for granted at this time. However, it is worth noting that in communication with a valid peer, there would be no attacker-controlled cleartext or

ciphertext in any part of the request, which might be otherwise used to compromise the security of the TLS channel.

AppID Verification

The 64-byte binary AppID specified in the request is hashed using SHA512, to calculate H(AppID). The resulting 64-byte digest is used to lookup the AppID in a server-side database.

A hash of the AppID is used for two reasons;

- The AppID is a sensitive value which we can avoid storing on the BlindHashing server
- The search function used by the database may not run in constant time, and therefore could be exploited by a client to search for valid AppIDs by matching progressively more bytes and looking for progressively longer delays in receiving the 'AppID Not Found' error response

Using H(AppID) solves both of these issues as it is computationally infeasible to compute the pre-images necessary to pull off a timing attack, and H(AppID) is not sensitive since the AppID itself is 512-bits of randomly generated data.

Client Authentication

Authentication of a BlindHashing request is primarily based on matching H(AppID) to an existing record in the server-side database. If the H(AppID) is not found, then a HTTP 500 status code is returned with a message indicating 'AppID Not Found' is returned.

If the H(AppID) value is found, then the request is considered to be authenticated for that AppID, and the configured settings for the AppID are retrieved from the server-side database.

Request Authorization

Once a request is authenticated for a given AppID, a request is then authorized based on the Client IP address, and any rate limits which may be in place. If an IP whitelist is specified, then the IP is compared against the list of allowed addresses (which may be individual addresses, or subnet ranges) for a match.

If a rate limit is specified, a multi-level token bucket algorithm determines if the request may be processed. A first level token bucket determines a baseline allowed rate, while a second level token bucket allows short term bursts above the baseline, which may also trigger a warning to be sent to the client via SMS, or email.

If both a Client IP whitelist and rate limit exist, the IP whitelist is checked first, and the rate list checked only if the IP is allowed. In either case, a failure results in an error message indicating the reason for the failure.

If the request is not authorized, either a 'Client IP Rejected' or 'Rate Limit Exceeded' counter is incremented for that AppID. Otherwise, if the request is authorized, an 'Authorized Request' counter is incremented for that AppID. Performance counters track the incidence of various such events occurring on a per-AppID basis at 5-minute intervals.

Data Pool Read Request

Once a request is authorized, to prepare for the actual BlindHashing request, the Request Handler retrieves the 64-byte 'Organization Key' which was randomly generated by the BlindHashing server at the time a new client account is first created, and which is shared across all the AppIDs created for that account. The Request Handler also retrieves the data pool volume, data pool size, and data pool read count settings which are configured for the specified version of the AppID, as well as the same settings for the latest configured version of the AppID.

If the 'Version ID' in the BlindHashing request is not equal to the latest configured version, then two Data Pool Read requests are prepared. If no version is specified, or the specified version matches the latest configured version, then only a single Data Pool Read Request is prepared.

Each Data Pool Read Request is a UDP packet with the following format;

- TxID – 4 bytes– A binary value which is copied into the corresponding response packet returned by the Data Pool Handler verbatim.
- UDP Port – 4 bytes – Big Endian unsigned integer indicating the UDP Destination Port for the response packet.
- Read Count – 4 bytes – Big Endian unsigned integer indicating the number of reads to be made from the data pool to complete the BlindHashing Request (valid range: 1 – 128)
- Pool Size – 4 bytes – Big Endian unsigned integer indicating the size of the data pool volume in millions of bytes (MiB).
- Organization Key – 64 bytes – The organization level (not unique per AppID) key used to transform the data pool bits into a virtual private data pool
- Indexer – 64 bytes – The value used to derive the uniformly distributed locations of the data pool to be read from

If two requests are being dispatched concurrently, all values will be the same except for the TxID, Read Count, and Pool Size.

The 'TxID' is used to distinguish one Data Pool Read Request from any other requests which may have been sent from the same Request Handler to the same Data Pool Handler while this request is still being processing. The Data Pool Read Response will include the TxID field verbatim to allow the Request Handler to associate the response with the corresponding request. The TxID is implemented by the Request Handler as a monotonically incrementing counter which will wrap if it overflows. This allows for 2^{32}-1 requests to theoretically be outstanding all at once before any responses from the Data Pool Handler would not be distinguishable.

The 'Indexer' is calculated as the SHA512-HMAC(AppID, Hash1) where AppID is the key, and Hash1 is the value being hashed. The 64-byte binary values (not string representations thereof) are provided as inputs to the underlying SHA512 function, and the 'Indexer' is the 64-byte binary result of the calculation.

The 'Hash1' value is run through an HMAC with the AppID as the key in order to ensure that requests with the same 'Hash1' value issued against two different AppIDs will each have independently

distributed read patterns against the data pool, and provide entirely uncorrelated results, even for two AppIDs running against the same virtual private data pool.

An available Data Pool Handler is selected by the Request Handler based on the 'Data Pool Volume' configured for the AppID. If multiple Data Pool Handlers are available to service requests for a given 'Data Pool Volume' then requests are dispatched round-robin between the available Data Pool Handlers.

A single UDP packet is sent to 'localhost' with a Destination UDP Port based on the selected Data Pool Handler. A dedicated secure channel is provided by a 'spiped' daemon—"a utility for creating symmetrically encrypted and authenticated pipes between socket addresses"—which listens on localhost on the specified port, and provides a persistent secure channel carrying packets to the desired Data Pool Handler.

The Request Handler increments a 'Data Pool Read' counter associated with the AppID based on the number of Data Pool Read requests which were dispatched (either 1 or 2) as well as request counters associated with each Data Pool Handler.

Data Pool Read Response

The Data Pool Handler provides asynchronous responses via UDP packets with the following format;

- TxID – 4 bytes – Binary value copied from the Data Pool Request packet
- Salt2 – 64 bytes – Binary value with the completed result of the BlindHashing process

The response is sent from the Data Pool Handler to its 'localhost' with a UDP Port as specified in the Data Pool Request, which again causes the response to be routed by an 'spiped' daemon providing a persistent encrypted link back between the Data Pool Handler and Request Handler processes.

As UDP packets are received by the Request Handler on the designated port, the 68 byte payload is parsed into TxID and Salt2 fields. The TxID is used to lookup the corresponding request which was in process, and dispatch the HTTP response back to the client.

BlindHashing Response

Once a UDP packet is received from the Data Pool Handler over the encrypted channel, the TxID is used to lookup which request it corresponds to. If the request involves only a single Data Pool Read, then a JSON-formatted response with 'Salt2' and 'Version ID' is immediately prepared, and dispatched to the remote client with an HTTP Status Code of 200.

If the request involves two Data Pool Reads, then the 'Salt2' and 'Version ID' are queued to be returned to the client. Once both responses are received from the Data Pool Handler(s) then a single JSON-formatted response containing: 'Salt2', 'VersionID', 'New Salt2', 'New VersionID' is assembled and dispatched to the remote client. The TxID is used to determine which response corresponds to 'Salt2' versus 'New Salt 2'.

The Request Handler also implements a timeout handler which will dispatch a response with an HTTP Status Code '500' indicating an Internal Timeout if the UDP response(s) are not received from the Data Pool Handlers within the expected timeframe.

Data Pool Handler

The Data Pool Handler provides the gateway into performing the necessary reads from the underlying data pool volume in order to calculate the ultimate 'Salt2' value. The service is designed as a UDP listener accepting individual 144-byte requests, and dispatching asynchronous 68-byte UDP responses.

While the operational controls to secure the Data Pool Handler are out-of-scope for this document, it bears mentioning that the crucial requirement of this service is to not leak the underlying data pool bits in any way, and to provide a network traffic pattern which can be easily audited to ensure that the only network traffic was the expected number of Data Pool Read requests and responses.

The processing steps the Data Pool Handler performs are to;

- Receive Data Pool requests over UDP from Request Handlers,
- Parse the specified Tx ID, UDP Port, Organization Key, Indexer, Read Count, and Data Pool Size from the UDP request
- Use the Indexer to generate 'Read Count' number of uniformly distributed values between [0…'Data Pool Size').
- Identify the data pool blocks to be read, and dispatch the reads for those blocks
- As each read completes,
    - Verify the data pool block checksum,
    - Transform the data pool blocks into the virtual private data pool blocks using the Organization Key and HMAC-SHA512
- Extract the desired 64-bytes from the transformed data pool block(s)
- Collect these 64-byte reads into a 64 * 'Read Count' byte buffer, ordered by Read Count
- Once the read buffer is full, perform a final HMAC on the read buffer with the Organizational Key in order to calculate 'Salt2'.
- Dispatch a UDP packet with a Destination UDP Port as specified in the request, and containing the following data;
    - TxID (4 bytes)
    - Salt2 (64 bytes)

Request Parsing

Each Data Pool Read Request consists of a single 144 byte UDP packet sent from a Request Handler over a persistent encrypted channel. The request contains the 6 fields as defined above, and which are parsed as simple binary data, or big endian unsigned integers.

Requests of an invalid length or which specify an out-of-range UDP Port, Read Count, or Data Pool Size are silently discarded.

Calculating Read Locations

We must execute 'Read Count' number of reads into the data pool using the 'Indexer' to calculate a deterministic set of uniformly random i.i.d. indices between [0, 'Data Pool Size'). To perform this task we use the Indexer as the key to a SHA512-HMAC-DRBG as specified in NIST SP 800-90Ar1 to generate a

deterministic sequence of random bits, which are then used to produce uniformly random values in the desired range as follows;

- Deterministic random bits generated by the SHA512-HMAC-DRBG are generated 64 bytes at a time, and consumed 8 bytes at a time, taken as 64-bit big endian unsigned integers
- Uniformity is achieved by taking a new 64-bit unsigned integer from the DRBG output until the taken value is outside the range [0, 2**64 % Data Pool Size). This guarantees the selected random number will be inside [2**64 % Data Pool Size, 2**64) which maps back to [0, Data Pool Size) after reduction modulo Data Pool Size. (This algorithm is taken from BSD's *arc4random_uniform* function)

In this fashion, the 'Indexer' is used to produce a list of 'Read Count' number of uniformly random i.i.d indices into the Data Pool between 0 (inclusive) and 'Data Pool Size' (exclusive). A given 64-byte binary 'Indexer' will always produce the exact same sequence of read locations for a particularly sized data pool.

Calculating Virtual Private Data Pool Blocks

A 'Virtual Private Data Pool' allows a client to perform BlindHashing with, what appears for all intents and purposes to be, their own unique secure random Data Pool, but which underneath is physically a single data pool shared between any numbers of other clients. The transformation from the shared or 'raw' data pool into a 'virtual private' data pool must satisfy several requirements;

- Produce a client-specific set of data pool bits with no loss of entropy or uniformity,
- Produce a virtual private data pool which is completely independent and uncorrelated with the underlying shared data pool
- Produce a data pool which could be provided entirely to the client without risking any disclosure of the 'raw' data pool bits from which they were calculated

Taking the underlying 'raw' data pool as a numbered sequence of 64-byte blocks, we can calculate the corresponding 'virtual private' data pool blocks by calculating the SHA512-HMAC(Organization Key, Data Pool Block || Block Number) for each block. Specifically, the SHA512-HMAC is calculated using the 64-byte binary Organization Key as the HMAC key, and a 72-byte HMAC value consisting of the 64-byte binary 'Data Pool Block' as the HMAC value concatenated with the 64-bit unsigned big endian integer block number.

Therefore, in order to perform a read of 64 bytes from a specific index within a virtual private data pool, in most cases (63 out of 64 times) we will need to convert two blocks of the underlying data pool into the corresponding virtual private data pool blocks, and then select the desired 64-bytes of virtual private data from within those two blocks (128 bytes) of data.

In 1 out of 64 times, the index to be read happens to align perfectly with the start of a data pool block, and therefor only a single SHA512-HMAC needs to be performed to calculate the virtual private data pool block and provide the desired data into the buffer. In the current implementation, however, for timing consistency, two data pool blocks are still read from disk and converted into virtual private data pool blocks, even though just the entire first block of data is copied into the output buffer.

Note that different 'Applications' created by a given client will each be assigned a unique AppID, but will share the same underlying 'Organization Key'. This allows each client to have effectively one 'virtual private data pool' shared between their various applications, and provides the possibility for that client's own data pool to be brought "on-premise" for their own use in serving their own Applications / AppIDs.

Processing Read Data

An output buffer with a length of 64 * 'Read Count' bytes is allocated at the start of each request, which will ultimately contain the result of the completed reads from the client's virtual private data pool. Each index to be read is given a 'Read Number' from 1..'Read Count', and the 64-byte result of each read is copied into the output buffer at a location of 64 * ('Read Number' – 1). So, for example, a 'Read Count' of 64 will result in a 4096 byte output buffer where the result of the first read is copied into locations 0..63 of the buffer, the result of the second read is copied into locations 64..127 of the buffer, etc.

Once all reads are complete and all locations within the output buffer have been filled, the final Salt2 result is calculated by taking the SHA512-HMAC(Organization Key, buffer).

Response Format

Once a request is received and parsed, asynchronous IO reads are dispatched to retrieve the necessary data pool blocks which will be converted into virtual private data pool blocks and ultimately assembled into the output buffer. When the final read is complete, transformed, and inserted into the output buffer, then the final HMAC is performed and a Salt2 value is produced.

At that time, we dispatch a UDP packet from the Data Pool Handler to the specified UDP Destination Port with a payload of;

- TxID – 4 bytes – copied from the Data Pool Read Request
- Salt2 – 64 bytes – the final result

Data Pool Management

The Data Pool is nothing but cryptographically secure random data, dumped to disk and stored in a format which allows the Data Pool Handlers to ensure the integrity of the bits being read. We must carefully protect against undetected bit-flips, as an undetected and uncorrected bit-flip would cause inconsistency (non-determinism) in the BlindHashing function which, in the case of password hashing, would result in valid passwords appearing invalid.

Therefore our design goals for generating and storing the data pool are simply to encapsulate the random bits of the data pool in a structure to allow cryptographic validation of the integrity of the data pool at a point in time, as well as run-time application-layer validation of all bits being used for each BlindHashing request.

Data Generation

The random data which forms the basis of the data pool must be securely generated in such a way that no fraction 's' of the underlying data pool bits can be used to complete a BlindHashing request with a

probability of successfully completing the request greater than s^n where 's' is the percentage of the data pool, and 'n' is the read count. For example, assuming a read count of 64 lookups, then possessing 80% of the data pool bits must not allow completing BlindHashing requests at an average rate greater than .8**64 or ~6.27e-7 (about 6 requests successfully completed out of every 10 million).

Additionally, it must not be possible to obtain n-bits of the underlying data pool in any fashion other than physically stealing n-bits of the pool itself. In other words, to the greatest extent possible, the data pool bits should not be themselves derived from a random key with a deterministic random bit generator and a counter (e.g. NIST SP800-90A).

However, since the size of the data pool can be quite large, and the collection of true random entropy quite performance limited, a hybrid approach using DRBGs with very frequent re-keying can produce the desired size random files drawn from a CS-PRNG with a large set of rapidly discarded secure random keys. Crucially, a single known key should not be used to generate a significant portion of the data pool for risk that the key itself could be stolen, and subvert the fundamental assumptions around the bounded retrieval capabilities of an attacker.

Data Storage, Checksums, and Parity

A data pool volume is conceptually a single monolithic byte array, but in practice we store the data pool on a standard file system as a set of numbered files, where each file contains 1 billion bytes (1 GiB) of the data pool's random data.

The storage sub-system provides several degrees of data integrity protection, including checksums and ECC maintained by the physical drive itself, as well as checksums at the file-system level. We also interleave our own application layer checksums into the data pool as follows.

Each data pool file contains 1 billion bytes of random data, or 15,625,000 data pool blocks of 64 bytes each. Each data pool block is stored on the filesystem as the 64 bytes of data pool entropy followed by a 2 byte 'crc16' calculated from the preceding 64 bytes. Therefore each data pool file is in total 1,031,250,000 bytes long, containing a total of 1 GiB of data pool data, and 31.25 MiB of checksum data.

For each individual data pool file, 5% of parity data (50 MiB) is calculated and stored (using 'par2' Reed-Solomon encoding). This parity data is useful for more rapid recovery of the file if a 'crc16' value ever fails to verify during the course of performing reads from the data pool.

When any given 64-byte data pool block is to be read by the Data Pool Handler, 66 bytes are actually read from disk, and the 'crc16' is calculated for the first 64-bytes and then compared to the last 2 bytes. If the calculated checksum does not match, the associated data pool file is flagged as damaged and attempted to be repaired.

Integrity & Signing

The 'sha512sum' of each data pool file (with the interleaved checksums in place), as well as each associated parity file, is calculated and stored in a 'spec' file located in the same directory. A GPG signature of the 'spec' file is also stored.

When a Data Pool Handler process is first started, the process discovers any available data pool volumes, and then verifies the integrity of the volume by first verifying the GPG signature of the 'spec' file, and then ensuring that the sha512sum of each data pool file matches the specification.

If a recovery action is ever initiated, the recovery process first calculates the sha512sum of the parity file and ensures it matches the 'spec' before attempting to use the parity data to repair the damaged data pool file. After a completed repair, the sha512sum of the repaired file is compared against the spec to ensure a match before clearing the flag indicating the file is damaged / offline.

Redundancy & Availability

Data Pool Handlers search through a specified list of mount points for all available files which makeup a given data pool volume. The Data Pool Handlers will identify if multiple copies of the same file are available across multiple mount points, and load balance reads directed at those files across the available volumes.

If reads from a given file or mount point begin to fail due to a timeout, software, or hardware error, or if an application layer checksum fails to verify, the affected files are marked as damaged / offline, and reads will be directed toward the same files on any remaining mount points.