BLINDHASH
RESTORING TRUST IN PASSWORDS

BlindHash
Architecture & Deployment

August, 2017

Table of Contents

Foreword

Passwords have become a lucrative target for hackers; administrative passwords can be leveraged to gain deeper access within a company network, while customer passwords can be used to target end users for fraud and identity theft. Passwords grant access to an ever-increasing trove of emails, documents, photos, and sensitive financial data. Since users tend to reuse the same password across sites, the fallout from a breach can be widespread and carry deeply personal consequences.

Responding to a password database breach is a time consuming and complex process which begins with identifying the means of access and remediating any ongoing compromise, resetting passwords and notifying users of the breach. Companies will experience significantly increased customer support requirements as users are notified of the breach and their passwords are reset. Some users may find themselves locked out of their accounts if password reset options are no longer accessible. Many users will struggle to understand how even a “salted and hashed” password breach puts them at risk, and will leave the same password in place on other services. Beyond the direct remediation costs in the immediate aftermath of a breach, companies may suffer damage to their brand and reputation, as well as risk of regulatory scrutiny from agencies such as the FTC.

To try to protect passwords, industry best practice has been to use hashing functions which are designed to add cost and latency to each attempt to verify a password. These hashing functions have become increasingly sophisticated to allow their cost factors to be tuned over time, to best utilize the CPU and memory hardware available on common web servers, and to resist optimizations which an attacker might bring to bear, such as employing GPUs. A password hashing function must run slowly enough on an attacker’s hardware (or botnet) to impose a tangible cost of cracking each password, but crucially the hash must also run fast enough on the defender’s hardware to allow a site’s users to login during heavy load, creating a difficult and dynamic balancing act.

Notably, the cost to crack a password depends not just on the complexity and cost factors of the hash function, but also on the complexity of the password itself. Attackers optimize their attacks by creating sorted dictionaries of the most common passwords, and then testing them against every salted hash in order. Simple passwords employed by the majority of users can be cracked quickly and cheaply even with the most robust hashing. The industry has embarked on myriad attempts to increase password complexity through everything from public education campaigns, creative user interface design, and even draconian password policies which can ultimately lead to user frustration, forgotten passwords, site abandonment, even increased support costs and decreased conversions. This might buy some time in responding to a breach, but unfortunately the end result is unchanged; companies must assume that every stolen salt and hash will ultimately be cracked, and they must instruct their customers to act accordingly.

BlindHash’s *BlindHashing* fundamentally changes this equation by completely preventing offline attacks, regardless of the complexity of the password, even if a site’s password database is stolen. This document provides details on the design and cryptographic principles underlying BlindHash, as well as an implementation guide to help organizations deploy BlindHash to protect their own passwords.

The Limits of Computational Hashing

Standard practice for storing passwords is to generate a random value for each user (called a ‘salt’), hash the salt and password together with a suitably slow hashing function, and save the resulting ‘hash’ along with the ‘salt’. When verifying a password, the application will retrieve the user’s stored salt and hash, perform the same hashing function with the submitted password, and check if the calculated hash matches the stored value. The key pitfalls of this standard password hashing are threefold;

Managing ‘Cost Factors’: The hashing function must complete fast enough to responsively login your users, and yet slow enough to impose a tangible cost to an attacker. This core trade-off, where security can only be increased at the expense of performance, puts an effective upper-limit on the hashing cost which can be employed in large scale web applications. Cost factors must be carefully chosen to account for the peak authentication load a server may experience, in addition to the capacity required to actually serve users beyond simply logging them in. Cost factors must also be actively managed over time (and existing hashes strengthened) to keep pace with the ever-declining cost of computation.

Offline Attacks: When salts and hashes are stolen by an attacker, the attacker can then independently test candidate passwords and verify if they have guessed correctly, by simply calculating the hash for each guess. Once a salt and hash are stolen, the only limit to running this so-called *offline dictionary attack* is the time and computing power available to the attacker. Each guess at the password requires running the hash function once, so the ultimate cost to crack a password depends on the run-time cost of the hash function multiplied by the complexity (the so-called ‘guessability’) of each user’s password.

Breach Exposure: Password hashes are designed to be small and portable, but this technical feature also makes them extremely efficient to transfer over the network. Any vulnerability in a site’s defensive perimeter could allow an attacker to exfiltrate millions of salts and hashes in only a matter of seconds, and thus begin their offline attack. Unless we know the password was randomly generated with sufficient entropy (in which case we would call it a ‘token’, not a ‘password’), we must assume once a salt and hash are stolen that the password will eventually be cracked. Crucially, this means that standard password hashing does not actually protect passwords in the event of a breach, but rather hashing is merely a delaying tactic to hopefully allow time to detect and respond to the breach. Since users are highly likely to reuse their passwords across multiple sites, even resetting passwords after a breach does not close the attack window, meaning users must be promptly notified and educated about the risk of ever trying to use that password again.

Introducing BlindHash

BlindHash provides an additive layer of security for your existing password hashes in order to completely prevent offline dictionary attacks, even if your hashes and salts are stolen, and regardless of the complexity of your user's passwords.

BlindHash is designed to overcome the weaknesses inherent in computational hashing by providing durable security guarantees even in the event of a breach. This is accomplished by using a highly scalable cost factor which actually increases the cost to an attacker while simultaneously decreasing runtime latency. Instead of trading security for performance, BlindHash allows increasing security and performance in tandem, fundamentally altering the nature of how passwords are secured.

At the heart of BlindHash is something we call the *BlindHash Data Pool.* Using a cryptographically secure random number generator, we produce a massive array of completely random data. We store this data in racks of solid-state drives, located in secure data centers, and replicated across several geographic locations. The data pool is sized large enough so that simply trying to transfer the entire pool over the network would take years at full line rate. As we increase the size of the data pool by generating more random data and adding more solid state drives, overall performance is increased, both in terms of higher throughput and lower latency for BlindHash requests. At the same time, greater security is achieved, by increasing the amount of data which must be stolen in order to attack a protected password.

To perform a blind hash, you start by taking a user's password, generating a secure random salt, and applying the hashing function of your choice, perhaps *scrypt, bcrypt,* or even *Argon2*. But then instead of storing the salt and hash together in your user database where they could be stolen and attacked offline, we provide a simple API which *entangles* your hash with the BlindHash Data Pool so that an attacker trying to run an offline attack would have to steal both your hashes and salts, as well as the data pool before they could even attempt to crack even a single password. The hashes are effectively *blinded* by the data pool, while the data pool itself does not contain any information whatsoever about the hashes it protects.

In the next sections, we will walk-through how the BlindHash Data Pool is created and maintained, and how multiple sites can securely share and benefit from a single massive data pool. We will show how you can add BlindHash to the back-end authentication function of your site, and detail how the BlindHash Server actually calculates a blind hash response.

Data Pool Architecture

Maintaining the integrity and security of the data pool is the most mission-critical aspect of BlindHash's operation. Multiple layers of protection are deployed at the hardware, filesystem, and application layers, along with complete redundancy within each data center, and across multiple data centers.

The data content of the data pool is generated from a cryptographically secure random number generator and written onto encrypted storage. When this data is first generated, a combination of application-layer checksums, hashes, and parity data are calculated and stored alongside the random data to ensure the integrity of the data pool, and to allow the Data Pool Server to detect and correct any bit errors that may occur. This application-layer protection is in addition to the error correcting and data integrity mechanisms already provided by the hardware and filesystem.

For example, each individual BlindHash operation verifies a checksum stored alongside each 64-byte block as it is read from the data pool. Each instance of the data pool is regularly scanned to verify the cryptographic checksum of every file within the data pool. Parity data spread across each instance of the data pool allows error correction and online recovery in the event of any page-level or block-level failures of an SSD.

Scalability

A key benefit of BlindHash is its inherently scalable nature – a larger data pool spread across an increasing number of solid state drives increases the overall performance (IOPS) of the storage system. Since each BlindHash request is uniformly distributed across the data pool, the load is naturally balanced across all available drives. As IOPS increases, the latency of each individual BlindHash request decreases, while the maximum throughput (Blind Hashes per second) increases. As the size of the data pool increases, it also becomes easier to defend in terms of the amount of data an attacker must steal, and the time available to detect and shut down an attack before enough data can be captured.

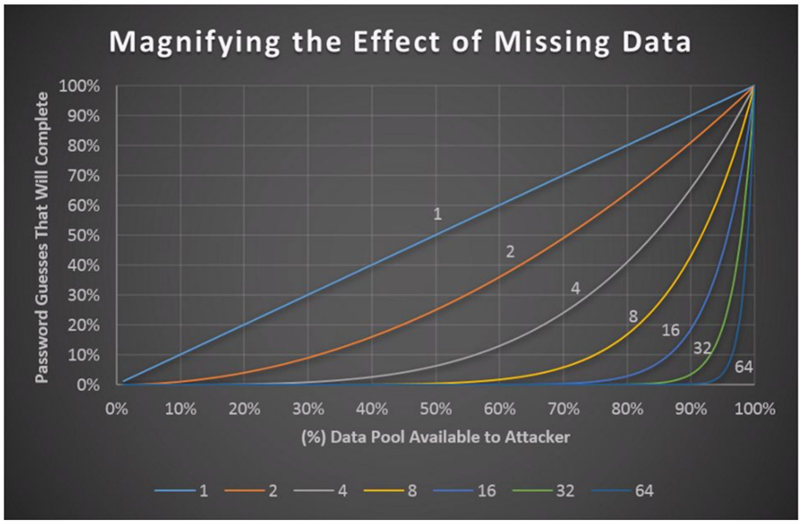
Virtual Private Data Pools

Maintaining an exceptionally large data pool would be too costly for most sites, but BlindHash makes it possible to share a single massive data pool across any number of sites, and each site gains the full security benefit of BlindHash. If two sites were allowed to directly share the same data pool, one site could attack another site by simply using the service to run their attack (effectively an *online* attack). Instead we can generate an unlimited number of *virtual private data pools* by using randomly generated keys to uniquely transform the data pool for each site.

For each site using the BlindHash service, the BlindHash Server generates two cryptographically secure pseudorandom 64-byte values; one is called the 'AppID' and is shared with the site to be used as an authentication token, and the other is a private key *(k)* which is associated with the AppID but kept secret within the data pool. The private key is used to transform a single shared data pool into a private data pool for each AppID. We will also show how this private key can be used to provide additional protection in case your site is ever breached.

Security by Obesity

In order to compute the correct response for a BlindHash request, we read from multiple uniformly distributed locations across the entire data pool. If any one of the reads cannot be completed, the BlindHash request will fail. Increasing the number of independent reads which are made as part of a single request makes any missing data exponentially more likely to stop an offline attack.



For example, imagine an attacker was able to steal “only” 8TB out of a 16TB data pool (50% of the total pool). If we needed to read only one location in the data pool to complete a BlindHash request, the attacker would be capable of completing 50% of their offline guesses. By expanding each BlindHash request into 64 independent data pool reads, now the attacker would only be able to complete the calculation with a probability of (.5)^64, or 5.42 x 10–20. Trying to calculate a blind hash with half of the data pool is like flipping a coin 64 times and needing to get heads every time. Even if an attacker could steal 80% of the data pool, with 64 lookups they would still only be able to check less than 1 in a million guesses.

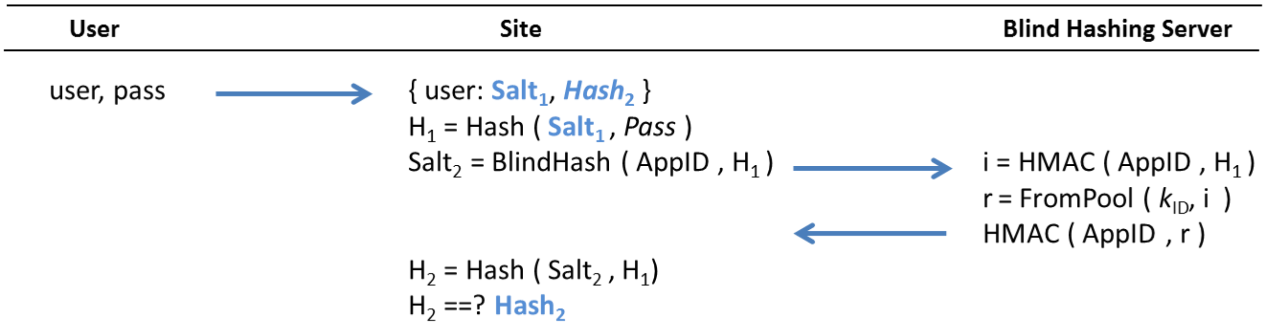
This provides a key defensive advantage – no longer can an attacker steal a small amount of data and then crack a small number hashes. Stealing anything but the vast majority of the data pool is useless.

Consider that each BlindHash Request involves receiving a 64-byte hash value and returning a 64-byte result. Ultimately this means that 64-bytes of data will be transmitted upstream from our Data Pool for each request. By imposing a physical limit on the network uplink, we can make a simple calculation;

Days to Steal = Maximum Number of Blind Hashes / Second * 64 Bytes / Data Pool Size;

For example, a 16TB data pool supporting up to 10,000 blind hashes per second, rate limited to 640,000 bps, means that it would take an attacker 289 days to steal at full-line rate. By counting every byte and every packet which traverses our network, and correlating the observed traffic with the expected load based on the number of BlindHash requests actually processed, we can immediately detect and investigate any unexpected traffic. The sheer size of the data pool provides a *window of defense* where attacks can be detected and thwarted long before they become dangerous.

BlindHash Request Flow


User
Site
Blind Hashing Server
user, pass
{ user: $Salt_1$, $Hash_2$ }
H_1 = Hash ($Salt_1$, $Pass$)
$Salt_2$ = BlindHash (AppID , H_1)
i = HMAC (AppID , H_1)
r = FromPool (k_{ID}, i)
HMAC (AppID , r)
H_2 = Hash ($Salt_2$, H_1)
H_2 ==? $Hash_2$

To step through the figure above, first the user submits their username and password to your site's own login form, just as they normally would. The entire BlindHash process is invisible to end-users, allowing sites to maintain complete control of the user experience.

On your site's server, the username is used to retrieve Salt1 and Hash2 from your database. You then perform traditional hashing with Salt1 and the password to calculate H1, just as usual, using the hash function of your choice.

Next, however, instead of storing H1 in your database directly, you perform *BlindHash* on H1. The BlindHash API is a single function which takes a 64-byte 'AppID' and a 64-byte 'Hash'. The AppID identifies your server and authenticates your BlindHash request. The 'Hash' being blinded is H1. The return value of the function is also a 64-byte value, which we call Salt2. You use the returned Salt2 and perform an additional hash, typically a simple HMAC, to calculate H2. The last step is to compare the calculated H2 against the Hash2 from your database. A match means the password was correct and the user should be logged in.

BlindHash is designed to respect your site's privacy and autonomy. Your site maintains complete control over verifying whether a user should be authenticated, and the BlindHash server does not see any personal information about your users. The BlindHash server does not even know if any given login attempt will ultimately succeed or fail. Notably, even a malicious BlindHash server could not cause an invalid login attempt to appear valid, because your site performs the final hash of Salt2 with H1 and verifies the result.

The connection between your server and the BlindHash Server is protected by TLS 1.2 with a pinned certificate, or alternatively, a dedicated encrypted channel such as 'stunnel'. The H1 value sent to the BlindHash Server is keyed by your Salt1 value, meaning the 'H1' value on its own is indistinguishable random data. Since the Salt1 is being used as a key, we recommend you generate salts as 64-byte cryptographically secure random values.

The immediate impact of adding BlindHash is that the values stored in your database, Salt1 and Hash2, can no longer be used on their own to perform an offline attack. Regardless of how simple the password, an attacker cannot even attempt to crack a hash without being able to also execute the BlindHash() function.

BlindHash API

The BlindHash API acts like a deterministic hash function – given any AppID and Hash input, the hash output will always be a 64-byte uniformly distributed pseudo-random value. Given the exact same set of inputs, BlindHash will always produce the exact same hash output. However, flipping even a single bit of the input value will produce a completely uncorrelated output value.

To submit a BlindHash request, issue an HTTPS GET request to a BlindHash server with a path of '/<AppID>/<Hash>'. The <AppID> is a 64-byte token which identifies and authorizes the site making the request, and <Hash> is the hash to be blinded. Values must be hex-encoded. An example request;

```
https://api.BlindHash.co/cdafb61b3473a8df704bfbadaae43a355576715d5a8baf176c54df059c41a7d816faf
3b697d235ec8512d56c44403aa672145029fbcf4529a626e2e8f37b2b3e/f607aeaaa87a1229abec33c5b1e78c98a8
faaefc9954dcd3b6822db010e228e7a44b7852a547bf7c51faa9d45176ab5a743b80b288e0798c96015d448ae6c609
```

The response is a JSON encoded object with two values; 'h' is the blind hash result, a 64-byte value in hexadecimal notation, and 'v' is a positive integer representing the data pool version number.

```
{"h":"612fbd62142f6095d7bd67376411a23e00a1eed5272719aab3cc0f33fd42804f16a820bdd1ebba23a8d3d8eb
476dee140c969dc2987ed516d9cada68ac6fbee4","v":2}
```

Data Pool Versioning

BlindHash regularly increases the total size of the BlindHash Data Pool over time, to support ever higher performance, and higher levels of security. Of course we cannot simply add or change the data within the data pool, because that would alter the result of the BlindHash function. In order to support growing the data pool over time, we only ever *append* new random data to the existing data pool, and never alter existing data.

After deploying additional data pool capacity, we make that capacity available, first to internal AppIDs used to verify the new data pool operation and performance, and then later to customer production AppIDs. To stage and track how much of the data pool to expose, each AppID keeps a versioned set of the size of its data pool over time.

If no version number is provided, by default a BlindHash request will be issued against the latest deployed version, i.e. the largest data pool size available. Every BlindHash response includes an integer value 'v' which is the latest version of the data pool when that request was issued. This version number is then saved alongside 'H2' in your user database.

Automatic Upgrades

When making a request to verify an *existing* password, you include the version number at the end of the URL, which instructs the BlindHash Server to use the exact sized data pool as existed at that time the original request was made. If a larger data pool has been deployed, the BlindHash Response will also automatically include two additional values; 'new_v' – the latest version number, and 'new_h' – the BlindHash result using the full available capacity of the current data pool.

For example; the following request explicitly specifies a BlindHash against version '2' of the Data Pool;

```
https://api.BlindHash.co/cdafb61b3473a8df704bfbadaae43a355576715d5a8baf176c54df059c41a7d816faf
3b697d235ec8512d56c44403aa672145029fbcf4529a626e2e8f37b2b3e/f607aeaaa87a1229abec33c5b1e78c98a8
```

```
faaefc9954dcd3b6822db010e228e7a44b7852a547bf7c51faa9d45176ab5a743b80b288e0798c96015d448ae6c609
/2
```

In this case the response includes only ‘h’ and ‘v’ indicating that version 2 is still the latest version of the data pool;

```
{"h":"612fbd62142f6095d7bd67376411a23e00a1eed5272719aab3cc0f33fd42804f16a820bdd1ebba23a8d3d8eb
476dee140c969dc2987ed516d9cada68ac6fbee4","v":2}
```

If later we issue the same request after the data pool capacity has been expanded for this AppID, now the response will include the original ‘h’ value using the original sized data pool, but also a ‘new_h’ and a ‘new_v’ as well. We include the ‘new_h’ and ‘new_v’ values within the same response to allow easily upgrading existing hashes to use the new expanded data pool without having to complete an additional BlindHash request or network round-trip.

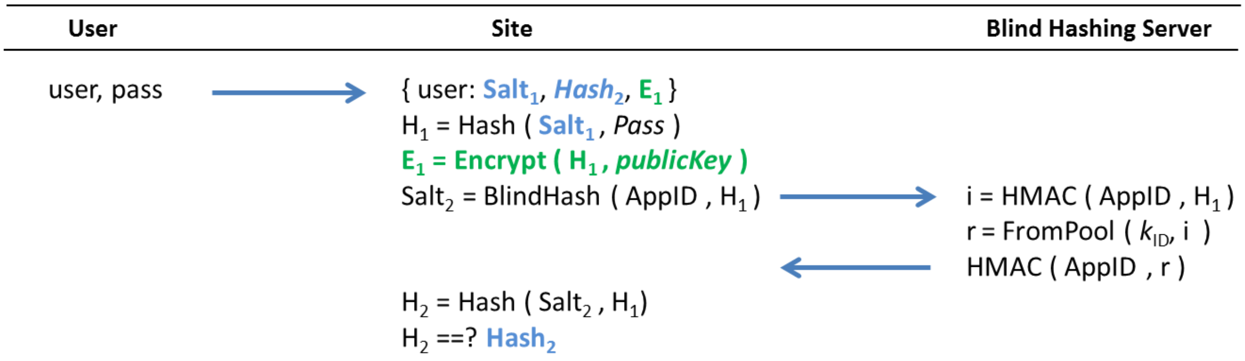
```
{"h":"612fbd62142f6095d7bd67376411a23e00a1eed5272719aab3cc0f33fd42804f16a820bdd1ebba23a8d3d8eb
476dee140c969dc2987ed516d9cada68ac6fbee4","v":2,"new_h":"368555904635836dfbf1a3ca59bca5fb91649
6f453a39f4782ef4f9a23420a7d47d7c6ca048039919ea3c47f44d40e7a85a96e9d87abaf332937d2655a1afca5","
new_v":3}
```

Your site will use the ‘h’ value to complete the authentication request and verify the password. Then, only if the authentication succeeds (the calculated H2 matches your stored Hash2 value) you can then use the ‘new_h’ value to calculate a new H2, and update your database with the ‘new_v’ and Hash2.

Breach Recovery

With traditional password hashing, if an attacker is able to pierce the perimeter defenses and gain access to the salts and hashes, the site has effectively been breached. With no way to recover or invalidate the stolen data, the site is forced to take immediate action to reset passwords and notify their customers. However, when password hashes are protected with BlindHash, a compromise against the site alone, or a compromise of the data pool alone does not expose the passwords to any risk of an offline attack. An attacker would need to compromise both the site and the BlindHash Data Pool together in order to even start an offline attack. This independent and additive layer of security is particularly effective against insider threats, or in the case where a site’s administrative credentials are compromised.

If your site ever were to experience a breach, with BlindHash you can also perform a simple breach recovery procedure which makes any stolen salts and hashes forever and completely useless. *Breach Recovery* effectively invalidates any stolen data an attacker might possess, and ensures customer passwords remain completely safe. To achieve this additional level of protection, we will add one step to the BlindHash Request Flow;



User | **Site** | **Blind Hashing Server**

user, pass →

{ user: Salt_1, $\textit{Hash}_2$, E_1 }

$H_1 = \text{Hash}(\text{Salt}_1, \textit{Pass})$

$E_1 = \text{Encrypt}(H_1, \textit{publicKey})$

$\text{Salt}_2 = \text{BlindHash}(\text{AppID}, H_1)$ →

$i = \text{HMAC}(\text{AppID}, H_1)$

$r = \text{FromPool}(k_{ID}, i)$

← $\text{HMAC}(\text{AppID}, r)$

$H_2 = \text{Hash}(\text{Salt}_2, H_1)$

H_2 ==? Hash_2

The added step is simply to keep a public key on your site which is used to encrypt H1 when a new password is registered, and to keep a copy of the encrypted result E1 in your database along with Salt1 and Hash2. The corresponding private key, which would allow the recovery of H1, must be kept in a secure offline location, so it is not subject to attack over the network.

In the event a breach is detected, after the vulnerability is patched and systems are re-established, you first request a new AppID from the BlindHash Server. Next, as an independent background process and without requiring end-users to login, you decrypt each E1 value to recover H1, run BlindHash() using the new AppID and H1, calculate a new H2, and update the 'Hash2' in your database.

Once all your hashes have been re-blinded using your new AppID, you can instruct the BlindHash Server to delete the old AppID, and destroy the corresponding private key *(k)*. Once the private key is destroyed, it will be forever impossible to execute a BlindHash using the old AppID, and any stolen Hash2 values which were based on that AppID are turned into nothing more than useless random data.

In this fashion, even if an attacker can steal your hashes and salts, the breach can be fully contained. Users' passwords remain entirely safe, without requiring a password reset or even for users to re-login to your site.

For particularly sensitive applications, it may even be advisable to *proactively rotate* your AppID using the same process, which could effectively guard against even an undetected breach.

BlindHash Request Processing

Within BlindHash's servers, the first step in processing a BlindHash request is to validate the AppID and authorize the request. After the AppID token is validated, additional site-specific validation steps may include verifying the source IP address of the request against a whitelist, or enforcing a rate limit on the allowed number of requests per second.

Once a request is authorized, we execute a HMAC-SHA512 using the specified 'Hash' and 'AppID'. The 64-byte result is used as a random seed in order to generate 64 uniformly distributed pseudorandom indices, or 'offsets' to locations within the data pool. We then dispatch requests across the data pool to read 64 bytes of data starting at each of the calculated offsets, collecting a total of 4096 bytes into an array.

As data is read from each of the specified locations within the data pool, we use HMAC-SHA512 with the 64-byte private key (*k)* associated with the AppID, to transform each data pool block being read into the virtual private data pool block for that AppID. Notably, we use a cryptographic one-way function rather than symmetric encryption to generate the virtual private data pool blocks, because we do not want the operation to be reversible. This also supports the special case where a customer may wish to obtain a copy of their own virtual private data pool. In this case, we can provide the virtual private data pool to the customer, and still be certain that even with the private key *(k)* the original shared data pool could never be reconstructed.

After all of the 64 independent reads complete, we will have filled the 4096 byte array with random data from the virtual private data pool. The final step is to execute an HMAC-SHA512 using the AppID and the filled 4096 byte array, to produce a single 64-byte result, which is returned in the HTTP response. Hashing the 64 reads all together in this manor ensures that the entire set of reads must succeed in order to produce the correct result, while the actual bytes read from the data pool are never exposed outside the BlindHash server.

Integration and Deployment

BlindHash is designed to be easy to integrate with your existing authentication code, and simple to stage and deploy into production. BlindHash is developing click-to-install plugins for standard web platforms such as Wordpress, which will deploy BlindHash without a single line of custom code. BlindHash also provides open source client libraries which allow integrating BlindHash with any existing authentication framework with just a few lines of code.

Client Libraries

BlindHash client libraries are available on GitHub (https://github.com/BlindHash) with separate repositories for each programming language. Each repository contains a Quick Start guide showing the commands required to setup and perform BlindHash, as well as examples of integrating BlindHash into an existing authentication framework. Client libraries may also be obtained through your language's package manager, such as 'npm' or 'NuGet' when available.

Each client library exposes a simple API call to perform BlindHash. Internally, the library will maintain redundant connections to BlindHash's Data Pool servers, issue BlindHash requests and parse the response. The client library will also track request latency, and optionally report latency statistics for display in the Web Admin Panel.

Side-by-Side Mode

When staging an initial deployment, instead of directly replacing the ‘Hash1’ value in your database with the new ‘Hash2’ value, keep your existing Hash1 values stored as-is, and add a new column or field to store the new blinded hash ‘Hash2’ value separately in your database.

By keeping your old schema intact, your existing authentication code can continue to function unchanged, unaware of the new ‘Hash2’ data. Then as you progressively deploy updated authentication code, you can track and assess the real-world performance and reliability of the BlindHash API in your environment. The original ‘Hash1’ value will also act as a fail-safe to allow users to login, even if a BlindHash request fails. At any point in the future, you can then either drop or encrypt the ‘Hash1’ value to enable the security of BlindHash.

Staged Rollouts

A common approach for large deployments is to stage a roll-out, both in terms of which servers are running the new authentication code, as well as which existing users within your system should have their passwords protected by BlindHash.

A simple approach for staging deployment across your user-base is for your authentication code to first attempt to retrieve the ‘Salt1’, ‘Hash1’, and ‘Hash2’ values for a specified user. Then, perform your initial hashing in any case, but only perform the Blind Hash if a ‘Hash2’ value exists. Otherwise, simply use the ‘Hash1’ value to verify the password as usual. For applications where strict timing consistency is required, you can perform the Blind Hash even if a ‘Hash2’ value is not stored for the specified user, calculate a ‘Hash2’ value using the result, but still perform the ultimate comparison against ‘Hash1’ for users which do not have a stored ‘Hash2’.

With this approach, your authentication code does not handle the special-case of adding BlindHash for any existing users. Instead, we will use a separate command-line program specifically for this purpose, which will allow you to enable BlindHash for a given set of existing users. In this case the program will select a set of users, retrieve their existing ‘Hash1’, perform a Blind Hash with ‘Hash1’ and calculate ‘Hash2’, and then store the calculated ‘Hash2’ value for the user. Since BlindHash can be added to an existing password hash without requiring access to the user’s password, this tool can be run on-demand and as needed, including rate limiting to ensure a smooth roll-out across even massive user bases.

Version Tracking

BlindHash maintains the security and integrity of the Data Pool, and over time we deploy additional storage capacity and expand the Data Pool with new random data. To expose the expanded data pool to existing applications, we use a simple incrementing integer, or VersionID, which you store along with each ‘Hash2’ value.

When you first create an AppID with BlindHash, your application will use the full capacity of the Data Pool, as it exists at that time, and the VersionID will be set to ‘1’. As additional Data Pool capacity is rolled out, your App may be configured to either enable this new capacity automatically, or require manual confirmation through our Web Admin Panel. With each new ‘generation’ of the Data Pool that is deployed, the VersionID is incremented.

By default, a BlindHash request will always use the latest version of the data pool if no VersionID is specified. Previous generations of the Data Pool are always accessible by simply specifying the desired VersionID at the end of the URL (…/<VersionID>) or providing a Version ID when making the API call.

We recommend serializing the VersionID as a 16-bit unsigned integer (ushort) and storing it along-side the 64-byte 'Hash2' value. When you retrieve the 'Hash2' value for an existing user, the version number will come with it, and you simply pass it along to the BlindHash API.

When you request a Blind Hash for a prior version of the Data Pool, the response will automatically include the 64-byte 'hash' for the requested Data Pool version, as well as an additional 64-byte 'new_hash' which is calculated using the latest version of the Data Pool, and the corresponding VersionID as 'new_vid'. In this case, your application will verify the user's password by calculating 'Hash2' using the original 'hash' value, and then only if the authentication is successful, calculate and store a new 'Hash2' using 'new_hash'. This allows you to upgrade your Blind Hashes as the Data Pool grows over time, without incurring the latency of an additional network round-trip.

Example Code

To add BlindHash to an existing authentication code path, we will make a few simple modifications to check if BlindHash should be used, apply the Blind Hash, and handle Data Pool upgrades as needed.

The following pseudo-code shows how existing 'login' and 'register' functions might look using PBKDF2-based iterative hashing;

```
struct PasswordInfo {
       byte[] Salt;
       byte[] Hash;
}
bool login(string username, string password) {
       PasswordInfo pwInfo = GetPassInfo(username);
       byte[] hash1 = PBKDF2(password, pwInfo.Salt, ITERATIONS);
       return slowEquals(pwInfo.Hash, hash1);
}
void register(string username, string password) {
       PasswordInfo pwInfo = new PasswordInfo();
       pwInfo.Salt = CS-PRNG.GetBytes(64);
       pwInfo.Hash = PBKDF2(password, pwInfo.Salt, ITERATIONS);
       pwInfo.Save();
}
```

To setup BlindHash, first we will modify our 'PasswordInfo' structure to include fields for 'Hash2' and 'Version'. The 'GetPassInfo()' function will populate this structure for the specified user, based on your particular data store. Note that 'Hash2' and 'Version' may be serialized into a single field in your database, and that field should be optional or null-able, and empty by default.

```
struct PasswordInfo {
       byte[] Salt;
       byte[] Hash;
       byte[] Hash2;
       ushort Version;
}
```

Next, we will add code in the ‘login’ function to perform BlindHash. Note that users without a ‘Hash2’ value will be authenticated using the original ‘Hash’ value as described above to support staged rollouts.

```
bool login(string username, string password) {
        PasswordInfo pwInfo = GetPassInfo(username);
        byte[] hash1 = PBKDF2(password, pwInfo.Salt, ITERATIONS);

        // If BlindHash is not active for this user, use ‘hash1’
        if (pwInfo.Hash2 == null)
                return slowEquals(pwInfo.Hash, hash1);

        // Perform the Blind Hash and check the result
        BlindHash.HashResult blindHash = BlindHash.BlindHash(APP_ID, hash1, pwInfo.Version);
        if (blindHash.Complete) {
                // Verify the password using ‘hash2’
                byte[] hash2 = HMAC-SHA512(blindHash.Hash, hash1);
                bool success = slowEquals(pwInfo.Hash2, hash2);

                // If password was valid, perform an upgrade if needed
                if (success && result.HasUpgrade) {
                        pwInfo.Version = blindHash.NewVersion;
                        pwInfo.Hash2 = HMAC-SHA512(blindHash.NewHash, hash1);
                        pwInfo.Save();
                }
                return success;
        }

        // If the Blind Hash did not complete, use ‘hash1’
        return slowEquals(pwInfo.Hash, hash1);
}
```

These changes are all that are required to support an initial deployment using ‘side-by-side’ mode with automatic fallback to using ‘Hash1’ for authentication. The ‘BlindHash.BlindHash’ function will automatically connect to your preferred server list (configured through the Web Admin panel), keep-alive the TLS sessions to ensure fast response times, and retry any timeouts on redundant servers. The client library will also keep a histogram of request latencies, and optionally provide the latency histogram on the BlindHash Web Admin panel.

Enhancing the ‘register’ function to protect all new user registrations with BlindHash involves similar code, except without any need to handle a Data Pool upgrade;

```
void register(string username, string password) {
        PasswordInfo pwInfo = new PasswordInfo();
        pwInfo.Salt = CS-PRNG.GetBytes(64);
        pwInfo.Hash = PBKDF2(password, pwInfo.Salt, ITERATIONS);

        BlindHash.HashResult blindHash = BlindHash.BlindHash(APP_ID, pwInfo.Hash);
        if (blindHash.Complete) {
                pwInfo.Version = blindHash.Version;
                pwInfo.Hash2 = HMAC-SHA512(blindHash.Hash, pwInfo.Hash);
        }
        pwInfo.Save();
}
```