
# Secure Quick Reliable Login

This first of the four SQRL system documents provides a broad overview of the SQRL system.

Documentation Download Links:

| | |
|---|---|
| SQRL Explained | https://www.grc.com/sqrl/sqrl_explained.pdf |
| SQRL Operating Details | https://www.grc.com/sqrl/sqrl_operating_details.pdf |
| SQRL Cryptography | https://www.grc.com/sqrl/sqrl_cryptography.pdf |
| SQRL On The Wire | https://www.grc.com/sqrl/sqrl_on_the_wire.pdf |

## Contents

# List of Figures

# Glossary of Cryptographic Terminology

*(There will be **no** test on any of this.)*

Anyone who is already well-versed in standard crypto will be familiar with the meaning of the terms on this first page of this three-page glossary. However, those who do not spend their days working to foil the plans of bad guys may appreciate a quick refresher on the meaning of these standard terms:

**AEAD Cipher** – "Authenticated Encryption with Associated Data" – most encryption modes do not also verify that the message has not been altered. An AEAD cipher performs encryption while simultaneously verifying that the message has not been changed. The "with Associated Data" feature allows in-the-clear plaintext to be included within the authenticated block so that it, too, is protected.

**AES-GCM** – "Advanced Encryption Standard Galois/Counter Mode" – AES-GCM is an authenticated encryption with associated data (AEAD) cipher. It employs Rijndael/AES encryption in counter mode to produce a pseudo random stream which is XORed with the plaintext or with the ciphertext to produce the other. It simultaneously provides authentication of the entire message with additional non-encrypted data.

**ASIC** – "Application Specific Integrated Circuit" – Unlike a CPU which is general purpose and programmable with software, an ASIC is designed for a specific purpose. In the context of cryptography, this is usually a chip used to dramatically increase the performance of a fixed "number crunching" task such as hashing input data. Because they are purpose-designed, they are typically the fastest solution, though also the most expensive.

**Authentication** – The terms "identification" and "authentication" are technically distinct. The first identifies, and the second, often separately, authenticates the possibly-inaccurate identification. As we'll see, SQRL securely and inseparably identifies and authenticates. Therefore, this text will use "Authentication" to describe both processes.

**FPGA** – "Field Programmable Gate Array" – An FPGA falls in between a software programmable CPU and a fixed-purpose ASIC. It is firmware-programmable hardware. It can be reprogrammed as needed and it will dramatically outperform a general-purpose CPU, but its "generality" makes it larger (and therefore more expensive) and also slower than ASICs.

**Multifactor** – If a single factor is insufficiently specific or reliable for identifying an individual, multiple differing factors may be used to increase the overall identification reliability and assurance.

**Nonce** – Short for "number used once" – In cryptography, there are many places where the reuse of some critical data must never be allowed to occur. For example, when eavesdropping could occur, the answer to a question might be overheard. So, if the same question were ever asked again, it might be correctly answered, not by someone who actually knew the answer, but by someone who overheard the question and its answer the first time. Secure cryptographic systems use "nonces" so that they never ask the same question twice.

**PBKDF** – "Password Based Key Derivation Function" – Any mathematical process that takes a variable-length password string and returns a fixed-length key. The function must have the property that whenever it is given the same password it will return the same key. Any cryptographic hash function meets this requirement. However, modern PBKDF functions typically add a "salt" input to create per-user versions of the PBKDF and add complexity to deliberately slow down their operation for resistance to ASIC and FPGA acceleration brute-force attack.

**SALT** – Cryptographic systems use the slang term "salt" to refer to any data added to a hash or other cryptographic function which provides additional randomization to, or customization of, an otherwise uniform and standard function.

**SCRYPT** – Scrypt (pronounced "ess-crypt") is a "memory hard" PBKDF (see above) which was deliberately designed by Colin Perceval to thwart the acceleration capabilities of ASIC and FPGA devices by requiring that a large block of RAM memory be committed to each PBKDF operation. Large blocks of RAM memory is not something that any custom hardware devices have in the required quantity, nor are likely to anytime soon.

**URL** – "Universal Resource Locator" – Since it would be possible for someone **not** to know that a URL is that funky string of characters we type into our web browsers to access web pages and to download files, and since SQRL is all about URLs and has them coming and going, it seemed useful, if only in the interest of completeness, to make sure that no one would confuse URLs for the Urals, that famous mountain range running longitudinally through Western Russia. We just wanted to be sure.

# SQRL-Specific Terminology

The development of SQRL required the invention of new ideas and several bits of technology. They needed naming so they could be described and discussed. You might find that scanning though these SQRL-specific definitions will be useful as an introduction to the descriptions on the pages that follow.

**Alt-ID** – Alternate Identity – SQRL automatically provides a single unique "native" user identity to every website. But there may be an occasion when a SQRL user wishes to be appear as someone else. SQRL's built-in Alt-ID facility allows any number of named alternate identities to be used.

**Ask** – Provides a means for a website to present its SQRL user with any high-security question and to receive their SQRL-signed authenticated reply, completely bypassing the hack/attack-prone web browser interface.

**CPS** – Client Provided Session – When the SQRL client resides in the same machine as the web browser being used, SQRL provides the authenticated session received from the web server over SQRL's secure channel, directly to the web browser. This has the highly desirable effect of thwarting every possible form of spoofing and man-in-the-middle attack.

**Cross-Device Sign-in** – SQRL sign-in webpages display a QR code version of their SQRL authentication URLs. This allows a SQRL client in a smartphone (for example), separate from the computer and web browser being signed into, to scan the displayed QR code and perform the authentication of the web browser session. This is termed "cross-device" sign-in as opposed to "same-device" sign in.

**EnHash** – Hash functions take a variable-length input and produce a fixed-size output. They are designed with a "security margin" which reflects a trade-off of security vs performance. Previous cryptographic hashes no longer deliver sufficient security as analytical and computational resources have improved. SQRL's use of hashing is not performance sensitive but it is extremely security sensitive. Therefore, SQRL uses a compound iterative hash function to provide an extremely large margin of security.

**EnScrypt** – The use of SQRL clients is protected by their SQRL password which is used to decrypt their stored SQRL identity file. Being a local password exposes it to brute force attacks. To harden SQRL clients against such attacks, SQRL iterates the memory-hard acceleration-resistant Scrypt PBKDF function to significantly further slow down its operation so that a single password decryption attempt requires many seconds.

**Identity Lock** – To prevent identity abuse, SQRL's identity lock prevents a website account's SQRL identity from being changed once it has been set. Only a higher level of authentication, provided by the identity's "Rescue Code", provides the information required to unlock a website's SQRL identity for removal or replacement.

**IMK** – The "Identity Master Key", derived from the IUK, is always stored encrypted and is briefly decrypted, when needed, by the user's SQRL password. When decrypted, it keys the HMAC256 which hashes the authentication domain (from the website's URL) to produce the per-site public/private key pair.

**IUK** – The "Identity Unlock Key" is obtained randomly when the user's SQRL identity is created. It is always stored encrypted and is decrypted by the identity's Rescue Code. In SQRL's key hierarchy, it is the antecedent of the identity's IMK and has a number of highly-privileged roles including password reset and identity unlock.

**MSA** – "Managed Shared Access" – SQRL defines, and the SSP API fully supports, the management of multiple SQRL identities which can be used to access a single website. Since SQRL identities are not readily shared, this provides a well-managed solution which is normally solved by unmanaged "credential sharing."

**NUT** – In homage to its namesake "squirrel", the SQRL system uses the term "nut" for its cryptographic nonces.

**QuickPass** – SQRL users should use a long and strong local password to unlock their encrypted identity. But that becomes harassment if they're required to constantly reenter that password every time they reauthenticate to another website. SQRL uses a "QuickPass" to allow just the first 'n' characters (by default, 4) to be quickly reentered after their whole password has been used once. "Guessing" is not allowed, and a mistakenly entered QuickPass will then require a full password reentry.

**ReKeying** – SQRL strongly protects the secrecy of its user's master key. But accidents happen. So SQRL also provides a built-in "rekeying" facility to allow its users to replace a stolen or believed-compromised key when necessary. Rekeying should never be necessary, but it's an available option, if needed.

**Rescue Code** – SQRL's Rescue Code is a randomly-obtained 24-digit value used to encrypt the user's most privileged key. It is never needed during normal use of SQRL, but it can be used to reset a forgotten SQRL password or rekey a user's identity which is believed to have become compromised.

**S4** – "SQRL Secure Storage System" – SQRL fully specifies its identity's encrypted data storage format. This readily-extended format fully supports the range of features SQRL needs and enables seamless identity sharing among SQRL clients. In addition to the user's master keys, this binary object format also contains and protects the user's non-encrypted user preference settings and up to four previously-used and retired master identity keys.

**Same-Device Sign-in** – When the user has a SQRL client or web browser extension installed in their machine, they click "Sign in with SQRL" to trigger authentication. Since they are signing into the browser which is running in the same machine as their SQRL client, this is referred to as "same-device" sign in.

**Secure Single-Factor** – Since SQRL both identifies its user to the web server, and robustly verifies its user's identity with strong cryptographic assurance, it functions as a secure single-factor identity solution with **nothing**, no additional identification assurance "factors," required.

**SQRL Identity** – For the sake of convenience, the term "SQRL Identity" is used interchangeably to refer to both the single master identity the user creates for themselves just once, and to the per-site identity which descends from that master identity. The user works with their master SQRL identity, whereas website receive a "per-site" SQRL identity which is unique to for each user and each website. The specific sense of SQRL identity being referred to should always be clear from its context.

**SQRL Password** – The user's master SQRL identity is always encrypted. Therefore, their password is required to decrypt their identity to RAM memory for use. Keeping the SQRL identity encrypted protects it from theft and abuse, and requiring the password before SQRL's first use prevents someone other than its proper user from coming along and logging in with someone else's SQRL identity.

**SSP API** – The "SQRL Service Provider Application Programming Interface" completely abstracts away all aspects of the SQRL protocol management, placing it behind the SSP API. The SSP API exposes a simple, private, HTTP query/response interface to dramatically reduce the work required to add SQRL support to any new web server.

# What is SQRL?

SQRL (pronounced "squirrel") is an open, free, intellectual property unencumbered, complete and practical system to cryptographically authenticate the identity of individuals across a network. Though principally intended for website visitor identification and account sign-in, its concepts may be extended for related applications. SQRL may be used alongside other traditional website sign-in systems, and it can replace **all** other systems while offering dramatic improvements in usability and security.

Though designed and intended to be a two-party solution – website visitor to website – if needed, SQRL can also be used in a federated third-party mode to provide centralized identity management services to websites (e.g. login.gov).

SQRL supports sign-in on a device containing the SQRL client, and "cross-device" sign-in using a SQRL client on a separate device, such as a smartphone, in the role of secure identity authenticator. SQRL clients currently exist for Windows, Linux/Mac (via WINE), Android, iOS, Chrome, Firefox, Edge.

**This document** endeavors to completely explain SQRL. It assumes that the reader is starting out with no idea what SQRL is or how SQRL works. It is deliberately much less dry than an Internet RFC specification because its audience is intended to be much wider. Non-developers can use this to gain a working understanding of SQRL.

**The following three documents** refine the concepts and topics introduced here to the level required by developers to build their own SQRL systems from scratch.

**If you have not used SQRL,** we strongly recommend that you take a few minutes to acquaint yourself with it before proceeding. You may download GRC's reference SQRL client for Windows, then create a SQRL identity, use SQRL to create an account and/or sign-in to GRC's and other demonstration servers and SQRL's public web forums.

You may obtain GRC's client from the link below, and experiment with SQRL at the sites shown:

| | |
|---|---|
| GRC's reference SQRL client for Windows: | https://www.grc.com/files/sqrl.exe |
| The simplest SQRL demo server to play with: | https://sqrl.grc.com/demo |
| SQRL's public web forums which support SQRL: | https://sqrl.grc.com |
| The demo server for experimenting with MSA: | https://sqrl.grc.com/msa |
| More features at the development server at GRC: | https://www.grc.com/sqrl/demo.htm |

# Implementations

The official **WordPress** SQRL plug-in: https://wordpress.org/plugins/sqrl-login/

SQRL plug-in for **Firefox**: https://addons.mozilla.org/en-US/firefox/addon/sqrl/

For **Chrome**: https://chrome.google.com/webstore/detail/sqrl/adfaiodpchglcmalaiifkcclimpffono

SQRL client for **Android**: https://github.com/kalaspuffar/secure-quick-reliable-login

SQRL server base in **JAVA**: https://github.com/sqrlserverjava/sqrl-server-base

A pluggable SSP API in **GO**: https://github.com/smw1218/sqrl-ssp

# How SQRL Works (in a nutshell)

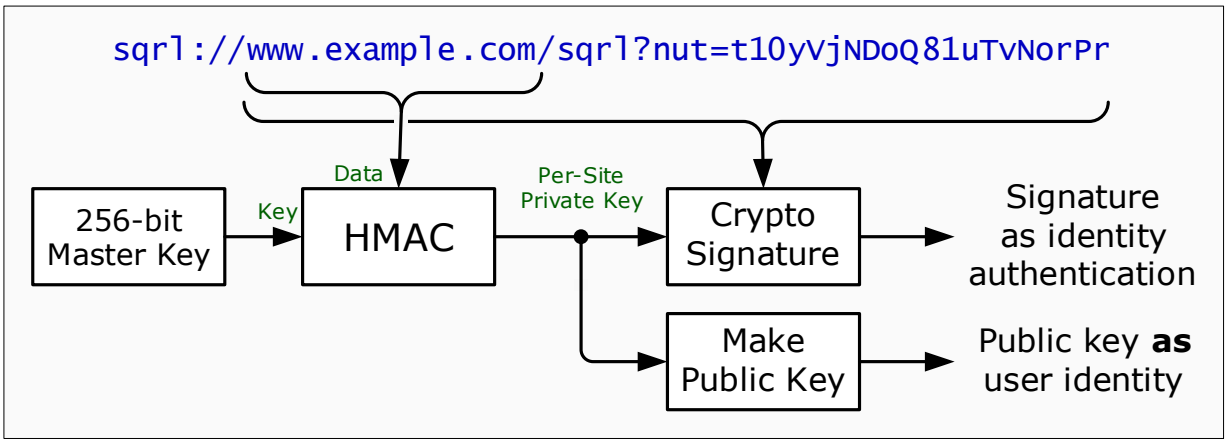



*Figure 1: SQRL authentication URL containing a nonce*

SQRL synthesizes elliptic curve public and private keys from a single master key and the full domain name of the website to which its user wishes to authenticate. This produces unique per-user/per-site key pairs which are used to both assert the user's identity and to verify the user's identity assertion.

- Websites provide a SQRL authentication URL containing a globally unique nonce.
- SQRL clients issue queries to that URL.
- Each website query reply supplies a new nonce for the next query in the chain.
- Each SQRL user has a single 256-bit master key.
- The domain name of the authentication URL is hashed with an HMAC keyed by the user's master key to produce a unique per-site public/private key pair.
- The public key is sent with every query to uniquely identify the user.
- The website retains the public key as the identifier of each SQRL user.
- Being a public key, there is no requirement that it be kept secret.
- The SQRL client uses the matching private key to sign every query sent to the website.
- The website uses the public key (which also identifies the user) to verify each query's signature.
- Since each signed query includes a nonce provided by the website, replay and impersonation attacks can never succeed.

## Again, but with a bit more detail

Website sign-in pages provide SQRL authentication URLs containing unique nonces. When triggered by its user clicking on a "sqrl://" link on the web page, or scanning the page's QR code, the user's SQRL client issues an HTTPS POST query to that nonce-containing URL. The POST data includes the website's URL and provides the user's public key for that website domain. The POST's query envelope is signed by the matching per-site private key which never leaves the SQRL client. The queried website uses the provided user-identifying public key to verify the query envelope's signature. This allows the website to affirm the user's identity claimed by the public key. Since the signed envelope includes the original URL and its nonce, which the server has never issued before, this simple and straightforward system cannot be spoofed or replayed and it is inherently immune to traditional attacks on both static and dynamic secrets.

Websites only store their user's public keys, so unlike passwords or time-based TOTP authenticators…

**SQRL gives websites no secrets to keep.**

## Visualizing the browser / server / SQRL client interaction

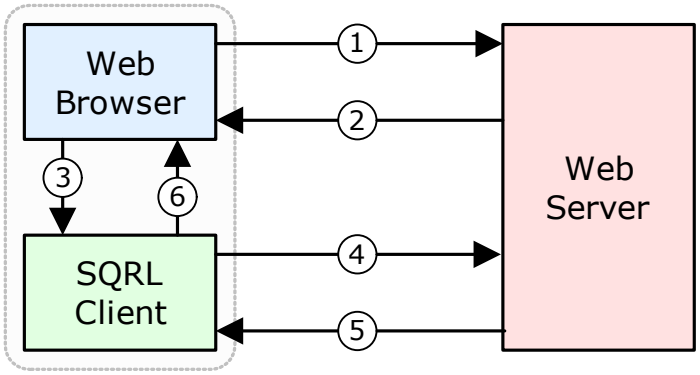



*Figure 2: Browser / Server / SQRL Client Interaction*

1. User's web browser requests a website's sign-in page.
2. Web server returns sign-in page including SQRL URL+Nonce.
3. User clicks "Sign in with SQRL" to invoke local SQRL client, sending it the URL+Nonce.
4. SQRL client sends its user's public (identifying) key for this site domain signed with private key.
5. With user's identity authenticated, website returns signed-in session URL to SQRL client.
6. SQRL client redirects user's browser to the URL of the authenticated signed-in session.

### Introducing client provided session (CPS)

Above, is the essence of the SQRL system. This offers the most powerfully secure anti-spoofing mode of SQRL's operation, known as Client Provided Session (CPS). Unlike traditional browser sign-in, upon successful identification and authentication of the user, the web server does **not** authenticate the browser session that was initiated by the browser in step (1) above. Instead, the web server returns an authentication token URL in its final query reply to the user's local SQRL client which then redirects the user's waiting web browser to the web server provided URL. This "CPS loop" thwarts any known man-in-the-middle or website identity spoofing attacks by providing the resulting authenticated session directly to the user's browser in a way that cannot be intercepted.

### Smartphone authentication

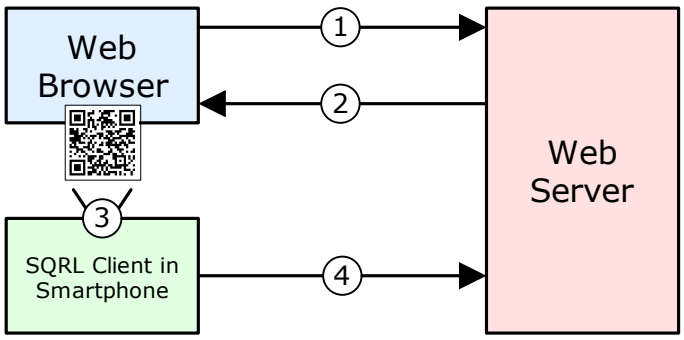



*Figure 3: "Cross-Device" QR Code Authentication*

SQRL's "sqrl://" URL's can also be displayed on website sign-in pages as textual QR codes. This allows SQRL clients residing in smartphones to serve as SQRL authentication token/devices. As shown above, the SQRL client's camera snaps the website's SQRL QR code, synthesizes the user's public and private keys for the site, as before, and queries the web server providing the user's signed identity assertion. The web server signs-in the user to the web browser session to which it provided the SQRL URL. (Not shown is JavaScript running on the sign-in webpage which periodically asks the web server whether its session has been signed-in "out of band".)

# Password Recovery & Hardening

## The SQRL client as a proxy for the user

SQRL effectively signs into websites for its user, serving as an identity proxy for its user. So a SQRL client must prevent its malicious use by anyone who is not its user. The SQRL identity master key and other usage settings must also be protected from theft and misuse. For this reason, SQRL uses a single master password to briefly decrypt (unlock) the user's SQRL identity into RAM. Decrypted identities are never printed, exported or stored to any non-volatile medium. Devices offering biometric identification may re-encrypt the user's decrypted identity for subsequent biometric access if desired.

## Single-party password recovery

How do we design a password-based client unlock that cannot be successfully attacked by brute force, while also allowing the user to change their password as they wish, and providing a means to recover from the inevitable "I just changed my password but now I've forgotten what I changed it to?"

When a SQRL identity is created, SQRL clients create tandem encryptions of the user's master key, one under the user-chosen password and a second encryption under a purely-random 24-digit decimal number:

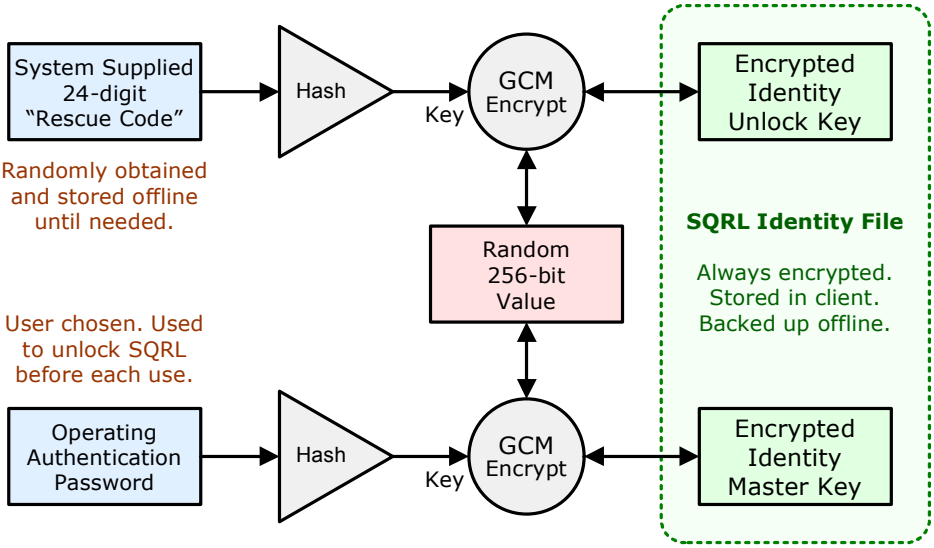



*Figure 4: Simplified (conceptual) Key Flow*

When a user's SQRL identity is created, a maximum-entropy 256-bit master key is derived by the SQRL client. That master key is encrypted using the hash of a randomly-chosen 24-digit decimal number as its key, and stored into the user's identity file. The same 256-bit master key is also encrypted using a hash of the user-supplied password and stored alongside the first encryption.

The 24-digit number is known as the "Rescue Code" because, as we'll see, it is able to rescue its user from several situations including the loss of their password. As the diagram above shows, the user's password would normally be used to decrypt the Identity Master Key. But if, for some reason, it cannot be used, there is a path from the upper "Encrypted Identity Unlock Key" down to the "Encrypted Identity Master Key" allowing it to be reestablished.

SQRL employs the AEAD-mode AES-GCM cipher which allows it to verify decryption keys and bind non-encrypted information into the encrypted authentication to prevent storage tampering.

## Brute-force attack hardening

People pick poor passwords. This may be especially true in a setting where they only wish to prevent the abuse of a device that's within their nominal control, such as their phone or computer. But SQRL amplifies the reward for a successful identity theft because impersonation is not only granted to the one website of a hacked password, but to **all** of the user's online SQRL identity, now and (unless changed) into the future. So, although SQRL provides a means for the recovery of a stolen identity (we'll get to that) hardening SQRL identities against brute force attack is a worthwhile line of defense.

SQRL was created in a world where custom FPGA and ASIC hardware was busily mining Bitcoin by hashing at high speed. Therefore, traditional PBKDFs (password-based key derivation functions) using iterations of standard hash functions (which are all now hardware accelerated) was no longer the best solution. SQRL needed a form of brute-force hardening that would be resistant to advanced hardware-assisted forms of acceleration. The SCRYPT, a "memory hard" hashing algorithm designed by Colin Percival formed a good foundation (https://en.wikipedia.org/wiki/Scrypt). Because FPGA and ASIC cores are necessarily memory lean, Scrypt's algorithm uses a large array of memory in such a way that it is not feasible to arrive at the same result as Scrypt without committing that large amount of memory to the task. Since FPGAs and ASICs do not have access to, for example, 16 megabytes of memory per core it is not feasible for custom hardware to usefully accelerate Scrypt.

> How Scrypt operates: Scrypt's input data to be hashed determines the sequence of a pseudo-random number generator. A large array is filled by the output of this generator with values modulus the size of the array. These values are treated as pointers (offsets) into the array. A very long "chain walk" is taken by following the pointers. The path taken during this walk generates data which is hashed to produce the output.

SQRL clients need to operate within modest-memory devices, such as less capable smartphones. So it could not require the use of, for example, 256 megabytes of array space which a busy device might not have available. This meant that while Scrypt was slow and acceleration resistant compared to traditional PBKDF2-style hashing, it was still far too fast when running within acceptable memory constraints. The other consideration was that this is client-side decryption, not a server-side PBKDF function, where a very slow function would cripple a busy server. For a single user entering a single password once per session, where we want maximum brute-force hardening, we **want** a very slow and acceleration-resistant process. In this setting "slow" isn't a problem, it's a beneficial feature.

We also wanted to allow SQRL's users to decide how long their password processing should take. The more time required, the more difficult to brute-force. We wanted user-specified decryption speed. This is provided by iterating over the acceleration-resistant Scrypt function and counting iterations while we wait for the user-specified time to elapse. We call this iterative function "EnScrypt":

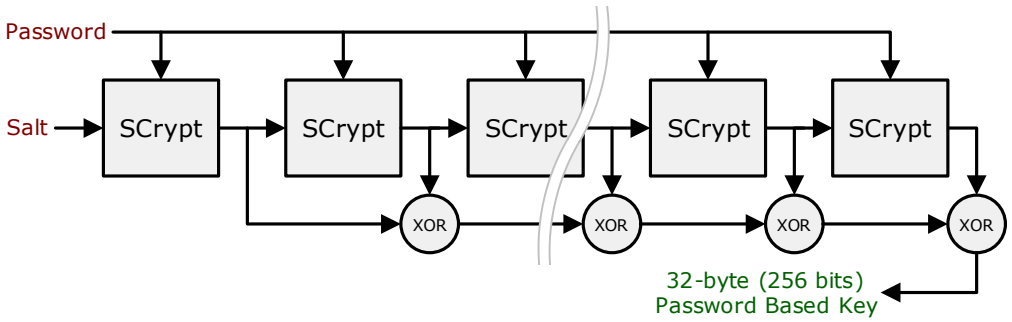



*Figure 5: Sqrl's EnScrypt PBKDF*

As shown above, the "salt" input of successive Scrypt iterations is the output of the previous iteration, and every output is XOR summed to produce the final result. Since each iteration of Scrypt depends upon the output of the previous one, and the result depends upon all of the intermediate products, the process can only be run sequentially and no stage can be bypassed.. The iteration count required during encryption is retained as a parameter for decryption which repeats the process, bounded by the count rather than by time.

# Identity Security

## SQRL's Secure Storage System (S4)

SQRL's design required a custom compartmentalized encrypted storage container to facilitate the secure storage of SQRL identities within an insecure (PC) environment and for secure long-term external backup. SQRL's Secure Storage System is a lightweight, easily implemented solution consisting of a block of binary data managed by a bit of code surrounding an AES/Rijndael cipher using Galois/Counter Mode (AES-GCM) to provide authenticated encryption with associated data (AEAD). This produces a minimal size, inherently portable, always-encrypted identity representation. The incorporation of an encrypted storage format into SQRL's formal specification allows all clients to safely exchange encrypted SQRL user identities. The small size of these identities allows securely-encrypted identity exchange via QR code.

| | |
|---|---|
| **sqrldata** - lowercase signature means binary follows | 8 bytes |
| {length = 125} - inclusive length of entire outer block | 2 bytes |
| {type = 1} - user access password protected data | 2 bytes |
| {pt length = 45} - inclusive length of entire inner block | 2 bytes |
| {aes-gcm iv} - initialization vector for auth/encrypt | 12 bytes |
| {scrypt random salt} - update for password change | 16 bytes |
| {scrypt log(n-factor)} - memory consumption factor | 1 byte |
| {scrypt iteration count} - time consumption factor | 4 bytes |
| {option flags} - 16 binary flags | 2 bytes |
| {hint length} - number of chars in hint | 1 byte |
| {pw verify sec} - seconds to run PW EnScrypt | 1 byte |
| {idle timeout min} - idle minutes before wiping PW | 2 bytes |
| {encrypted identity master key (IMK)} | 32 bytes |
| {encrypted identity lock key (ILK)} | 32 bytes |
| {verification tag} - | 16 bytes |
| {length = 73} | 2 bytes |
| {type = 2} - rescue code data | 2 bytes |
| {scrypt random salt} | 16 bytes |
| {scrypt log(n-factor)} | 1 byte |
| {scrypt iteration count} | 4 bytes |
| {encrypted identity unlock key (IUK)} | 32 bytes |
| {verification tag} - | 16 bytes |
| {length = 54, 86, 118 or 150} | 2 bytes |
| {type = 3} - previous identity unlock keys | 2 bytes |
| {edition >= 1} - count of all previous keys | 2 bytes |
| {encrypted previous IUK} | 32 bytes |
| {encrypted next older IUK (if present)} | 32 bytes |
| {encrypted next older IUK (if present)} | 32 bytes |
| {encrypted oldest previous IUK (if present)} | 32 bytes |
| {verification tag} - | 16 bytes |

*Figure 6: SQRL S4 Secure Storage*

## The need for identity backup

Since SQRL is, by design, a freestanding two-party system without any form of auto-magical cloud connection, there is an absolute requirement for the user to backup their identity for long-term safe keeping. This only needs to be done once to protect the user from any possibility of identity loss. But it **does** need to be done once. SQRL users will be making a large investment in their online SQRL identities, so the loss of that investment must be avoided. Since SQRL identities are derived randomly (like bitcoin addresses) they **cannot** be recreated. SQRL's backup is comprised of two separate pieces: The very small (a few hundred bytes) encrypted identity file and the 24-digit decimal Rescue Code.

A SQRL identity is so small it can be represented conveniently as text for printing and manual entry:

**jNRZ X6J9 RPz6 jKrY JGH7**
**zWE6 xzzi FF8X NBzh i6sz**
**nFqd nVza ttBZ SeQr kPrq**
**P9jv uz8t C8VY mZVV 3q5E**
**xjhe NdjT 6kvF PXef cnTT**
**3qq5 h9y**

*Figure 7: A Complete Textual Identity*

This textual representation uses a custom alphabet to reduce character ambiguity with incremental per-line hashing to catch any previous manual entry errors before the next line is started. This design makes printing identities onto paper practical. No storage medium has withstood the test of time better than paper. (If you have data saved on a once-ubiquitous 8-inch floppy or an Iomega Zip disk, try to read it today.) Everyone typically has some place where important papers are stored. But more than anything, any paper-based backup solution is inherently offline. And "offline" is where the user's identity backup and its associated Rescue Code must remain. The printed identity is stored in encrypted form so that its discovery will not compromise the user's identity because the Rescue Code is required for its decryption. Consequently, if unauthorized discoverability is a possibility, the paper backup and its printed Rescue Code should also be stored apart. Most of today's Internet users have more than one computer and often a smartphone. So, their identity will be naturally backed up as it is spread around among the various devices they use.

The SQRL system also defines a standard binary QR code format for identity interchange. The identity shown above is represented by this QR code:

*Figure 8: QR Code Identity*

SQRL clients can display their user's identities on screen as an encrypted QR code. This allows a user's camera-equipped device to easily import their identity from another of the user's SQRL clients.

## SQRL's "QuickPass"

When using any personal device, such as a PC or a smartphone, where some measure of continuity of use is available, the security thresholds for establishing "it's me" versus "it's still me" differ. Stated less formally: Asking someone sitting alone in front of their private computer, or holding their smartphone, to repeatedly reenter a long and complex password is needless and annoying. Doing so encourages the user to change their password to something much more convenient to enter but also much less secure.


*Figure 9: QuickPass Options*

SQRL clients acknowledge this with a system that allows only the first 'n' characters of the user's full password to be used to reassert the user's continued "yes, it's still me" presence. This shortcut is the user's "QuickPass." Once the user has entered their full password to initiate a session, and their identity has been successfully decrypted, the password's first 'n' characters are used to immediately re-encrypt their identity in RAM using an "EnScrypt time" of one second. This QuickPass data is retained in RAM and is never written to any non-volatile storage medium. During subsequent password challenges, the user may reenter just those first 'n' characters of their password to transiently decrypt their in-RAM identity. Even hitting "Enter" is unnecessary, since the system knows 'n'.

However, if the re-decryption should fail (due to the wrong 'n' characters) the QuickPass RAM is immediately wiped and the user is prompted to reenter their full password. In other words, in return for the convenience of a password abbreviation, zero guessing is allowed. Since QuickPass privilege is intended for the assertion of "I'm still here", any system event which might suggest that the user is not still there – such as screen blanking, system standby, user account switching, or a long period of inactivity – will also cause the QuickPass data to be proactively wiped from RAM with the first of these events. As an example, the upper right side of the Settings & Options dialog of GRC's SQRL client for Windows (shown on the previous page) shows user-configurable triggers which will cause the RAM-based QuickPass information to be wiped. After that, the client will prompt for and require the user's full-strength password. The result is a system that significantly increases ease of use while sacrificing very little practical security and encouraging the use of a long and strong full password.

## Secure SINGLE-Factor Identity Authentication

A traditional username identifies a user, but it is not necessarily unique, and it is not typically secret. Moreover, the frequent need to reset user passwords has resulted in the increasing use of widely known public eMail addresses as user identifiers… making them even easier to guess. Passwords may not be unique either. And while they are more or less secret, they cannot be used to reliably identify their users. Being often user-chosen, they often contain low levels of entropy. Time-based 6-digit TOTP "authentication" tokens provide no user identification, and their value is that they change in a predictable sequence every 30 seconds. So, they can provide some additional real time confirmation of a proposed user who shares the same symmetric secret. The industry has taken to using a collection of many single factors because no one of them can both identify and securely authenticate its user.

Now compare this with SQRL. SQRL's per-user/per-site public key unambiguously identifies a user to a website with a unique 256-bit one-to-one pairing. And the website's use of that public key to verify the signature returned of a never-before-used unique challenge securely authenticates the user with cryptographic veracity. It is, therefore, no exaggeration to state that **SQRL provides secure single-factor identity authentication.**

## Suppressing the Weakest Link

The security of a system is limited by the strength of its weakest link. So, while SQRL might be super-strong and unspoofable, we know that the existing system of usernames and passwords falls far short. Therefore, adding SQRL authentication to an existing web account can dramatically increase its user's ease of use. But, by definition, the addition of any system offering more security cannot increase the overall security of the system if the weaker solutions are allowed to persist. To address this, SQRL clients transmit two, sticky, global "login policy preference" flags, both which are initially **unset**. They are "sticky" in the sense that they are retained as password-protected settings in the SQRL identity file to keep the last setting given to them by their user. They are "global" in the sense that they are not "per-site". They transmit the user's login policy preference to every website where the user signs-in with their SQRL identity. Websites are asked to retain and honor the login policy settings they most recently received from each user. Once a user thoroughly understands how SQRL operates and has grown comfortable with its use, they may, at their sole discretion, enable either or both of these flags: The first flag asks sites to please **disable** all other non-SQRL forms of identity authentication and the second flag asks sites to please **disable** all forms of automated and human authentication recovery.


*Figure 10: Login Policy Preferences*

We all know that "I forgot my password" account recovery is one of the weakest links in login security. So, enabling either of these options will foreclose the use of that often exploited vulnerability. No SQRL user needs to enable either of these to obtain SQRL's login ease of use. But until alternative non-SQRL login authentication has also been disabled, important accounts will remain vulnerable to traditional non-SQRL authentication.

# Tight Identity Binding

In identity jargon we would say that SQRL creates a "tight" user-identity binding. It's "tight" in the sense that the system intends a user to create a single SQRL identity and to use that single identity for every website, possibly for life. Consequently, over time, users will make a significant investment in their online SQRL identity. By comparison, usernames and passwords are comparatively ephemeral. Little investment is made in them, and they can be, and often are, sometimes forced to change for enforcement of security or policy. In the jargon of identity, we would say that usernames and passwords are very "loosely bound" to the identity of their users.

There are common occasions where the inherent looseness of username/password identity binding serves as a feature rather than a liability. "Hey Dad, what's our Netflix password?" is an often heard and entirely reasonable question within households. When multiple users need to share access to a single online account, the inherent laxity of username and password identity binding can be quite handy. This is **not** something SQRL naturally accommodates because SQRL's inherently tight identity binding completely prevents the "sharing" or "borrowing" of SQRL identities.

So, what do we do?

Tighter identity binding feels like the future. While the freedom provided by anonymity is valuable beyond measure, the use of the Internet for transactions that need not or must not be anonymous, such as commerce or voting, require knowing with greater certainty than we can today the true identity of the person at the other end of the connection. While "absolute real world identity" is not something SQRL attempts to offer – yet – SQRL is a strong contender as the starting point for such an online capability. For those instances where it is desirable to share online accounts, SQRL's "Managed Shared Access" not only solves the problem, but also brings much improved management to an otherwise ad hoc free-for-all mess.

## Managed Shared Access

The proper way for websites that wish to allow and encourage the sharing of a single account among multiple users is to offer such a facility directly. SQRL specifies and implements a fully articulated and secure facility for this, known as "Managed Shared Access" (MSA). The SQRL Service Provider (SSP) API described later fully supports MSA, and a working implementation can be experimented with using the demonstration site at GRC: https://sqrl.grc.com/msa.

MSA supports the access management of any number of SQRL users in a simple fashion. The group of users initially contains only the original account owner as a manager of the group. Any users may be designated to be managers of the group, though the group must always have at least one manager designated. Any of the group's managers may add or remove any other users from the group. New users are added to the group by invitation. To do this, one of the group's managers requests an invitation consisting of a long single-use numeric nonce. The manager copies this nonce and sends it to the invitee through any sufficiently secure out-of-band communication channel. Being numeric, it may easily be verbalized and transcribed by its recipient. When the invitee, unknown to the website, signs in with SQRL, rather than creating a new account, they provide the invitation nonce. This securely associates their now-known SQRL identity with the shared group account. They will now be signed-in and a member of the group.

The use of Managed Shared Access may also be ideal for corporate/enterprise users of SQRL who wish to manage the access of disjoint sets of users. MSA allows new users to be securely added to shared groups, and also allows existing users to have their group membership revoked under the supervision of any of the group's managers.

When Managed Shared Access is available for a website, it offers the optimal means for managing the dynamic association of SQRL identities with that site. But it is foreseeable that, especially in the early days of SQRL, websites may not understand or bother to implement MSA. For that, there is a workaround…

## Explicitly Shared Identities

Suppose that an online banking service has added SQRL sign-in but hasn't yet understood the value of adding MSA support. Mom and Dad are both avid SQRL users and both also need to have access to the banking service. Since we know that usernames and passwords are not going away, probably ever, the obvious solution is for them to continue signing in the way they always have, by sharing a username and password.

But suppose they very much want to use SQRL. The solution, though not "the SQRL way," would be for the two of them to create a third SQRL identity which they share. SQRL clients support, though discourage, the use of multiple user identities in recognition of the fact that there are instances, such as this, where a single user might need to segment their online identity for sharing. In this case, Mom and Dad both have their own SQRL identities and share a "Mom & Dad" identity. They could then also use this explicitly shared identity for any other similar services that do not support SQRL's Managed Shared Access.

Over time, as MSA becomes more prevalent, retiring explicitly shared identities is facilitated by MSA. In this example, once the online banking service got its act together and added MSA, Mom or Dad would sign-in with their shared identity and send invitations to themselves, marking them as managers of the new MSA group. They would each sign in with their individual identities, and use their invitations to join the group. Once they were both present, their manager status with MSA would be used to delete the shared identity. Eventually, the need for any sharing would end, and the shared identity could be permanently retired, or at least shelved in case it might be needed again someday.

Before we leave the topic of tight identity binding, one last requirement needs to be addressed…

## Alternate Identities

There are instances where a SQRL user who is already known to a website might wish to transiently appear as someone who is unknown to the site. In a world with usernames and passwords, such a user would simply invent a new username and password and create an account as someone unknown. Since our goal is not only to improve upon but to also fully replace the functionality of usernames and passwords, SQRL needed to incorporate a solution for the need for "transient" identities.

The obvious solution would be for a SQRL user to create and use another SQRL identity. But SQRL's identity creation process is intended to be used very sparingly. In return for the strong recovery and protection features it offers, it is a bit involved and lengthy with a password and a 24-digit Rescue Code which needs to be written down and then proven by immediate reentry. It is not well suited to the creation of "ad hoc" disposable SQRL identities. The right solution is SQRL's built-in Alternate Identity facility:


*Figure 11: Alternate Identity Entry*

If, when signing into a website, the Alternate identity name field is not blank, whatever text the user has entered is appended to the end of the domain name being hashed to create the user's per-site identity. This presents an entirely new SQRL identity to the website. It can be thought of as a sub-identity, or as a fork of the primary identity. Since the text that's entered determines the sub-identity, they can also be regarded as "named sub-identities." And, if the same text is used anytime in the

future, the same sub-identity will be re-created. So, they can also be used to create persistent alter egos on sites that already know the user's "primary ego."

## Keeping Secrets for the User

Another problem SQRL solves for websites is the need for sites to securely store information on behalf of their users in such a way that no possible security breach or employee compromise can expose their user's data. This means encrypting user data without retaining any key. As we know, SQRL's protocol operates by providing websites with a public key which performs double duty, serving both as the user's identity and as a verifier of their client's signatures. This doesn't provide anything that a site might use as a secret encryption key. SQRL's per-user/per-site private key would be perfect for this, except that it must be kept secret. So SQRL uses the per-site private key, safely, by hiding it behind an "EnHash" function and using its output to key another HMAC function which hashes a site-provided "secret index" to generate an "indexed secret":

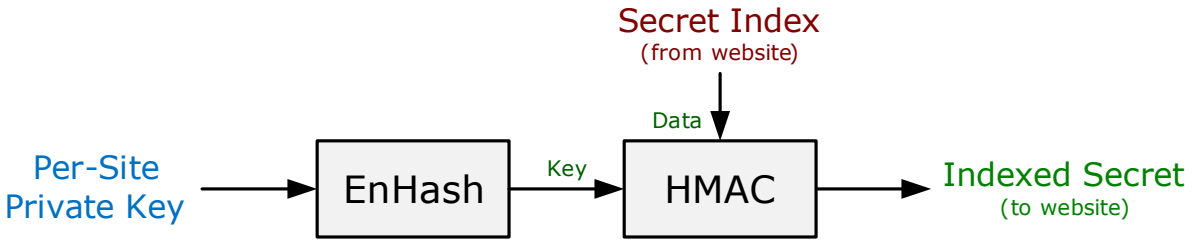



*Figure 12: Synthesizing the Indexed Secret*

This mechanism allows websites wishing to obtain individualized static secrets from their SQRL users to provide a string as a "Secret Index" in a reply to a SQRL client's query. The client will then return a 256-bit "Indexed Secret" which will remain unchanged for the life of the user's identity key. And if the user rekeys their identity (see below) SQRL arranges to provide both the previous and the updated Indexed Secrets so that the website can decrypt and re-encrypt the user's data under their new key.

### The EnHash function

A hash function is a one-way function which performs a lossy scrambling encryption of any variable-length input to produce a fixed-length "digest" output. Internally, hash functions are iterative. They incorporate some fixed number of "rounds" which are deemed to be sufficient to scramble their input enough to achieve their security design objectives. However, we have seen past hash functions fail and fail in the face of greater computation or analytical power. And even "reduced rounds" versions of contemporary hash functions have shown weaknesses.

SQRL's design employs an excessive security margin when such excessive margin is available at negligible cost in time or effort. The modern SHA-256 function is believed to be secure. After all, it was designed by the NSA, what could possibly go wrong? But earlier hashing efforts were also originally believed to be secure and later turned out not to be. There are several places within SQRL where we really (really really) need a one-way function to be absolutely one-way. In those places, we use the invented "EnHash" function:

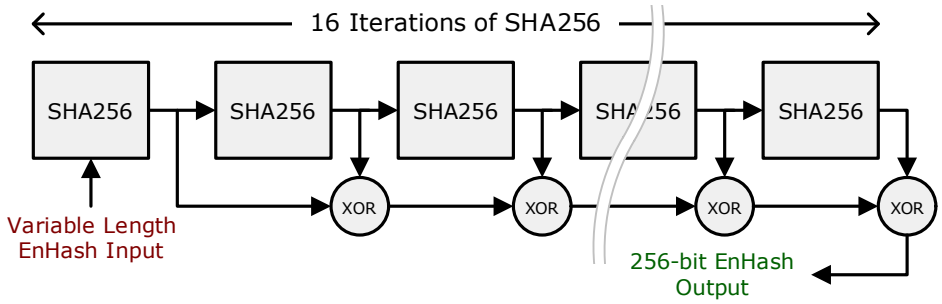



*Figure 13: EnHash Construction*

As shown above, we iterate 16 times over the SHA256 hash. Each successive SHA256 output is fed into the next iteration's input, and all outputs are XOR summed to produce the final result. This construction will be familiar from the earlier description of EnScrypt, which uses similar iterative XOR summing. Our preference would have been to programmatically quadruple the number of rounds used internally by the SHA256 function, from 64 to 256, to achieve a similar "excessive" security margin. But this would have created a custom hash function unsupported by any cryptographic libraries. We chose, instead, to iterate the standard drop-in SHA256 for function-call-level compatibility.

## All Identity Eggs in One Basket

SQRL's greatest design strength is also its greatest design liability: All of a user's identities derive from a single secret master key. That's wonderful **only** as long as that master key remains secret. If anything were to cause that master key to be revealed to anyone, a user's SQRL website identity in the past, and in the future, based upon that (previously) secret key would be vulnerable to wholesale impersonation. Although this might appear to be SQRL's Achilles heel, SQRL's design incorporates countermeasures to dramatically reduce the liability from a compromise of its user's master key.

Before we examine the additional features that make these countermeasures possible, it's worth noting that we are seeing the appearance of inexpensive hardware security modules (HSMs) such as the Yubikey[tm] which are capable of strongly protecting the secrets they keep. SQRL's cryptosystem is well suited to having its most security-sensitive elliptic curve signing operations performed inside an inexpensive consumer dongle or a consumer product's "secure enclave". This protects the user's master key because it is never exported from the dongle. It does its job internally. Although we may not initially be able to rely upon security hardening to keep SQRL's key safely locked inside hardware, in the longer term it is foreseeable that the burden of keeping SQRL's most important secret – the user's identity master key – will be lifted from the user… and from SQRL.

## Identity Lifecycle

Situations could arise where "rekeying" a SQRL identity is prudent, useful or necessary. And this is true even when a master key has not been stolen. As more of our lives move online, governments and law enforcement are becoming more insistent upon viewing whatever they wish of our online lives. And access to our identities becomes more valuable to malicious actors. A SQRL user might be coerced into revealing their SQRL access password to customs, border agents or law enforcement officers. Because such use requires the brief decryption of SQRL's Identity Master Key, depending upon the circumstances, the user may be concerned that their SQRL identity may no longer be trustworthy and may wish to abandon it in favor of a freshly "rekeyed" identity.

SQRL is able to protect its user's online identity from unauthorized change, while allowing for its authorized user to "rekey" their identity if that should ever become necessary for any reason. This could occur if "I think malware crawled into my computer and stole my identity" or "border agents had my phone overnight and now I'm not sure I can trust my online identity anymore."

More formally stating these requirements:

1. The valid owner of an online identity, who believes that it may have been compromised and can no longer be trusted for any reason, must be able to "rekey" their identity then update and replace their obsolete and untrusted identity at remote websites.

2. Anyone else who obtains access to a user's identity, by any means whatsoever, must be absolutely prevented from updating, changing, or removing that identity at any websites where it is already known and authorized to identify its user.

Points 1 and 2 appear to be mutually exclusive: The owner of an identity has privileges that someone who obtains full access to that same identity does not have. How is this possible?

## SQRL's Rescue Code

Page 9 describes how SQRL's 2-party system arranges to allow a forgotten password to be reset. This is facilitated by SQRL's 24-digit Rescue Code which serves as a random, maximum entropy, symmetric key used to encrypt and, if needed later, decrypt a copy of the user's key if the user is no longer able, for whatever reason, to use their password for that purpose. The diagram on that page carries the legend "Simplified (conceptual) Key Flow" because it depicted only one key when, in fact, there are actually two keys:

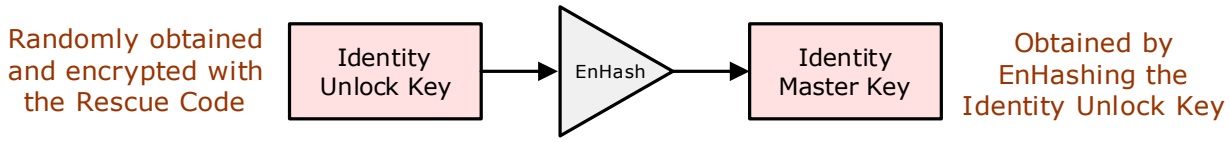



*Figure 14: SQRL's Key Hierarchy*

As this diagram shows, the "Identity Master Key" (IMK) descends from the "Identity Unlock Key" (IUK) through the very-strongly-one-way EnHash function. It is this primary IUK that is obtained at random. Neither of these keys is ever stored unencrypted in non-volatile storage. When either key is needed it is transiently decrypted in RAM for use, then wiped to zero immediately after use.

This construction allows for password recovery, since the Rescue Code can be used to briefly decrypt the IUK, which is then "Enhashed" to produce the decrypted IMK. A new password can then be chosen and used to re-encrypt the IMK for subsequent use by the user.

This creates a key hierarchy, with the decrypted IUK able to re-create the IMK. But the reverse is not true. The strong one-way EnHash function absolutely protects the IUK from attack and discovery, even if the IMK, which must be transiently decrypted whenever the user's SQRL identity is used, were to become compromised.

### The Rescue Code is never online.

That bears repeating: The user's identity Rescue Code is never online. The essence of SQRL's 2-party security is that there is an online "working secret" and a (very far) offline "master secret".

SQRL's "working secret" (the IMK) must be made transiently available to the SQRL client for SQRL to generate its per-site public and private keys. When the user's SQRL password "unlocks" the client, what's actually happening is that the IMK is decrypted into RAM by the user's password so that an identity authentication can be performed.

By comparison, SQRL is designed to **never** need the identity's "master secret" (the IUK) during daily use. Although the IUK is stored with the user's identity, it is always stored strongly encrypted by the identity's (entirely random) Rescue Code, which is written down somewhere safe and deliberately left at home or kept with other important papers in a safe place. Since the Rescue Code will never be kept on the person travelling, or stored in their device, it is not possible for its user to be coerced into divulging something that is not present and which everyone knows no SQRL user will have on their person or in their device. And since the only thing the rescue code is needed for is changing things for long term identity lifecycle management, there is also no valid reason for **any** authority to demand it, or to require that travelers carry it with them.

## Identity "Association" & Lock

Let's back up a bit to define a new term. Before using SQRL to sign-in to a website we first need to "associate" a user's SQRL identity with an account on that site. SQRL provides exceptional account association security by introducing the concept of an "Identity Lock." SQRL's identity lock allows new identity/website account associations to be freely created by SQRL users as they associate their SQRL identities with websites. **However, once such an association has been created it is "locked" and cannot be casually (or maliciously) removed or changed.**

One of the special powers given to SQRL's "Rescue Code" is its ability to "unlock" existing identity associations. The Identity Lock utilizes Diffie-Helman Key Agreement (DHKA) and elliptic curve signing in a unique way described in detail later in this document. It is an important anti-abuse/anti-hacking technology because it prevents mischief that could ensue if SQRL identities could be easily removed or replaced by entities other than their authentic user.

Since it is possible that an attacker might obtain a SQRL user's identity and access password, SQRL needed a system that would prevent even a seemingly authentic user from changing their own previously set SQRL identity association – since it might not really be the identity's authentic user. Therefore, SQRL's Identity Lock system, which can only be unlocked by the identity's Rescue Code, provides that…

- Any SQRL client may establish new website identity associations, but…
- ONLY a SQRL client which has been temporarily loaded with its identity's Rescue Code can make any alterations to, including removal of, any preexisting identity association.

Since the Rescue Code is never stored by any SQRL client, **any** form of attack on or compromise of the SQRL client cannot also compromise the identity's Rescue Code – since it is not present during the normal daily use of SQRL. If the user briefly needs the services of their Rescue Code – which is never required for SQRL's use – it can be loaded into the client's volatile memory and removed immediately after performing its required task.

## Identity "ReKeying"

When a SQRL user has some reason to believe that their current online identity has been compromised and can no longer be trusted, they select their SQRL client's "Rekey Identity" function. This requires the use of their decrypted Identity Unlock Key, so they must provide their current identity's Rescue Code. Their SQRL client will randomly generate a new replacement Identity Unlock Key and provide them with a new Rescue Code protecting their new identity. Their new Rescue Code must, of course, be printed onto paper and stored safely and securely.

Here's the neat trick: Their SQRL identity now contains BOTH identity keys, the newly created key and their previous identity key. Anytime their client attempts to authenticate the user to a website, it will present public keys for that site generated by **both** the new and the old identity keys, and the client will also "co-sign" its queries using both private keys.

If this is the first time a newly rekeyed user is visiting that website, the website will recognize its SQRL user by their **previous** properly-signed identity and will immediately update its records with the details of the user's **current** properly-signed identity, thus automatically and autonomously replacing and removing all record of the user's previous identity in favor of the newer identity.

As this recently rekeyed SQRL user moves around the Internet visiting sites and using SQRL, every website visited will admit them as though nothing happened while quietly updating their records to the new identity, thus replacing and forgetting the old identity.

Although identity rekeying is intended to **never** be necessary, SQRL recognizes that "stuff happens", so it provides a means for making the process simple and transparent if lightning should strike.

Previous keys are only 32-bytes long, so the SQRL identity storage specification allows for the storage of up to four previous identity keys. If a website indicates that it doesn't recognize either of the user's current or most recent previous identity, the SQRL client will retry with the current identity and the next most recent previous identity key, and so on until all available previous identity keys have been tried. Again, identity rekeying is designed and intended to be exceedingly rare so even one previous identity key should be sufficient. SQRL provides for four because it was simple to provide plenty of extra allowance.

It's important to note that SQRL's Identity Lock prevents malicious rekeying of existing identity keys. The Rescue Code is required for identity rekeying because only it can provide the cryptographic information required to unlock the current identity at the website's end. This uses several additional keys that we have not discussed yet. It's sufficient for now to understand that as long as the Rescue Code is not available, rekeying of SQRL identities cannot be accomplished.

### "Taking back" a compromised identity

As explained above, when any website encounters a SQRL client presenting both a recognized and verified previous identity accompanied by a not-yet-known current identity, it immediately abandons, forgets and replaces all record of the user's previous identity with the user's new current identity. After this has been done, the website will no longer have any record of the user's previous identity, and anyone attempting to use it will be completely unknown and ignored.

## This allows a SQRL user to "take back" a compromised identity.

After rekeying their identity, it will be necessary for a SQRL user to revisit each website at which they wish to prevent the possible abuse of their previous identity key. This generally means prioritizing visits to their most important websites first. Each such visit will remove their previous, possibly-compromised identity from the site visited, thus preventing any subsequent misuse and abuse of their previous identity at that site.

Over time, as additional sites are visited, all remnants of the previous identity will be erased from the Internet and SQRL, this absolutely private, untrackable, 2-party identity authentication system, will have successfully, if somewhat manually and deliberately, dealt with a situation that should never occur in the first place. But if it does, the SQRL user is able to reassert control over their identity, effectively retaking it from whomever might have caused its compromise.

### Preventing malicious use before rekeying

SQRL's Identity Lock system has one additional feature that might come in handy: It is possible for any SQRL user who is not in possession of their identity's Rescue Code (which should be the case for ALL SQRL users all the time) to quickly and easily disable the use of SQRL at any website they wish. Such a disable command can only be reversed by the use of the identity's Rescue Code, which can either reenable the use of SQRL with the user's current key, or as part of a rekeying update event.

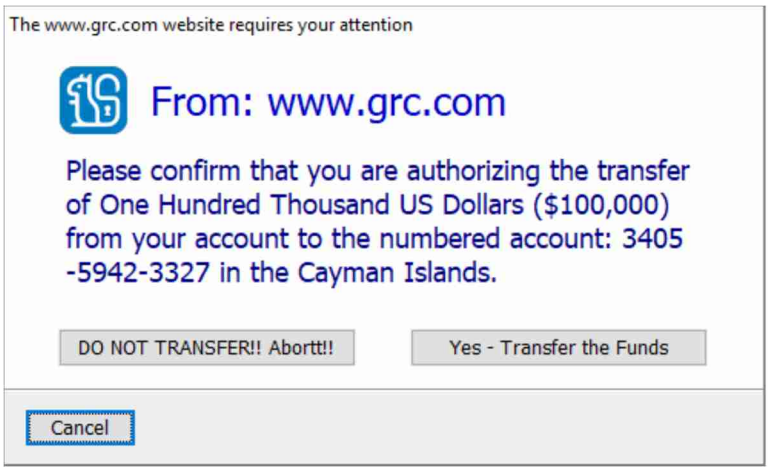
We noted before that someone travelling with their SQRL identity might have reason to believe that it had been compromised. In that case, they will want to rekey their identity and "take it back". But if they are on the road with their Rescue Code securely stored at home, any rekeying will have to wait until they return home. This creates a vulnerability if they are unable to quickly rekey. However, even without their Rescue Code, they can visit their most important sites to disable all subsequent use of SQRL… even for themselves. In that fashion, although they are denying their own access, they are also denying access to anyone who they believe might have obtained their identity. Once they have access to their Rescue Code they can choose whether they wish to reenable SQRL access under their old identity or reenable access by rekeying that identity.

SQRL's identity lifecycle management features are designed to never be required. Once SQRL identity keys can be moved into hardware dongles or strongly protected within software enclaves, the chance of SQRL identity theft by malware will disappear. However, the risk of coerced compromise will always exist. Therefore, the SQRL system provides a solution to a problem that will hopefully never occur. But if it should, it can "turn back the clock" on the compromise and allow a SQRL user to reassert their control over their identity.

# "ASK"ing High-Security Questions

During SQRL's multi-year development we often hit upon features that we were sorely tempted to include. Since this was a "from scratch" effort, SQRL could include anything. Auto form-fill?, hey why not? Or more automated website sign-in?, that would be cool! Or maybe move more website account management into SQRL for standardization? Why not?? And so on. After flirting with each idea we would inevitably remember that SQRL is for authentication, not form-filling, or anything else, and that to aid its adoption it should remain as lightweight and readily implemented as possible. Everything written above supports that. There are already other ways to perform most of those "extra" functions. And that's what makes SQRL's "Ask" feature special and unique. This one non-authentication feature seemed so useful, uniquely suited to SQRL, and unavailable through **any** other means, that it survived all attempts at pruning. It is part of SQRL, and it's called "Ask"…

In response to a SQRL client's query, a website may return a free-format question as text, and optionally the labels of one or two buttons. When a SQRL client receives this in the site's reply, it will display a special "Ask" dialog and await the user's response.




If the user presses cancel, the SQRL client simply drops the Ask reply and that's the end of it. But if one of the labelled buttons are pressed, the SQRL client submits another query to the website containing the user's response. Note that since every SQRL client reply always includes the entire content of the website's previous response, all signed by the user's private key, the question being asked, and its response, are cryptographically tamper-proof.

The unique "Ask" feature is present because it provides an "out-of-band" (out-of-browser) channel for a website to obtain highly security-sensitive and important confirmations from its user. Browsers have become such spoofing platforms, with no end of that in sight, that this seemed like something worthy of inclusion into SQRL.

# No History Kept

The most often asked question is whether SQRL clients retain any sort of log or record of the websites where they have been used. The thought is that if rekeying was needed at some point in the future, having a master list of every website where the old key had been used would greatly aid the task of visiting all of those sites for automatic key replacement.

No one disagrees with that premise, because it's correct. But there are problems with the idea. For one thing, SQRL users will likely have their SQRL identities installed on all of the computers and smartphones they use. One of SQRL's major benefits is that it's a two-party system without any need

to communicate with "headquarters". There is no headquarters, and that's by design. So, if SQRL clients were keeping logs of the sites where they were used, they would all need to be synchronized through some mechanism. The other problem is that if a record of past SQRL usage was being kept by the SQRL client, and bad guys got control of the client, they would also obtain a list of all of the websites where the user's presumably-stolen SQRL identity could immediately be used. Clearly, this list is not something we want the bad guys to ever get their hands on. They would immediately scan it for the biggest targets of opportunity and be greatly facilitated in their attack.

The good news is, whether or not such a feature is added, nothing needs to change about SQRL's core operation. So, some future version of SQRL, or a SQRL client which adds those features, could be something to consider for the future. But its implementation would need to be secured.

## No Centralized Point of Failure

In a world without SQRL, users wishing to minimize the many pitfalls of traditional passwords have been left with no alternative but password managers. LastPass, one of the leading password managers, has suffered multiple service outages affecting their entire userbase and leaving their users with no ability to log in any of their websites. Outages caused by equipment failures, routing table mistakes and malicious denial-of-service attacks have become commonplace in today's Internet.

So, although it may go without saying, it is still worth pointing out that another benefit of SQRL's two-party solution is that there is no middle man and no centralized point of failure or attack that could adversely affect the lives of the system's users. For the future, that seems almost as important as SQRL's many other benefits.

## Please see the other SQRL implementation documents for more:

Documentation Download Links:

| | |
|---|---|
| SQRL Operating Details | https://www.grc.com/sqrl/sqrl_operating_details.pdf |
| SQRL Cryptography | https://www.grc.com/sqrl/sqrl_cryptography.pdf |
| SQRL On The Wire | https://www.grc.com/sqrl/sqrl_on_the_wire.pdf |

Thank you for your attention and interest in this promising system and solution.