Operating Details SQRL

Secure Quick Reliable Login

This second "Operating Details" document assumes that its reader has read and understood the concepts described in the first "SQRL Explained" introductory document. This document digs into the next layer of SQRL's design, including details of the SQRL URL and its support for same-device and cross-device authentication and anti-spoofing considerations.

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

SQRL's Four SQRL System Components

The full SQRL system can best be described and understood as four communicating objects:

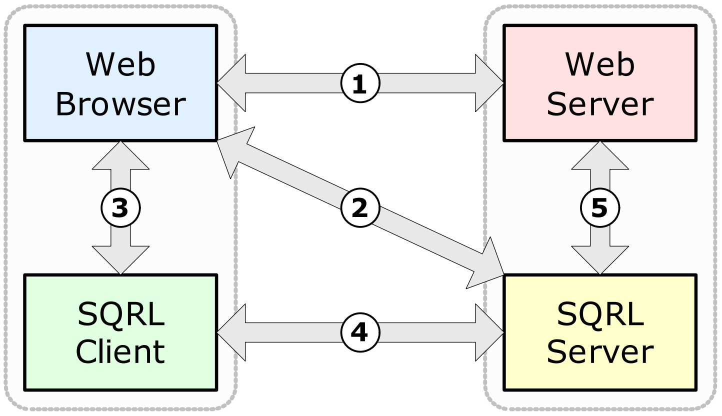

Figure 1: SQRL System Block Diagram

The Web Server and SQRL Server are depicted as separate objects because their respective functions can be, and often are, cleanly divided. Some implementations of SQRL merge the web and SQRL servers into a single service whereas other implementations may choose to have the SQRL services performed outside the web server.

To support the separation of functions, which may make sense when SQRL is being added to existing installations, the SQRL specification formally defines an application programming interface known as the SSP API – SQRL Service Provider API – which performs exactly this function. The API encapsulates all SQRL-specific knowledge and provides the web server with a very simple HTTP query interface. This dramatically reduces the work required to add SQRL support to any web site. The SSP API was first written as IIS ISAPI extension and then ported to 'C' as a free-standing implementation. A pluggable implementation in GoLang has also been created. It is hoped and expected that additional SSP API implementations will be forthcoming since they dramatically simplify the addition of SQRL authentication for any website.

Please note that this document assumes an understanding of web development with a working knowledge of web clients and servers, web browser operation and JavaScript.

As we know, SQRL provides single factor authenticated identity with cryptographic integrity. Rather than repeating the phrase "authenticated identity" over and over throughout this document, we will adopt the shorthand "authentication" and "authenticates" to mean "fully authenticating the user's SQRL identity."

The Case for Spoofing Prevention

When we hear that systems like SMS messaging can no longer be considered safe as an out-of-band means for sending a one-time password, it's clear that attackers are actively exploiting **any** weakness an authentication system might have. Many years of experience have shown that the more automatic an authentication system is, the greater the risk of spoofing due to the "encouraged inattention" of its users. Since SQRL offloads the need for any per-site vigilance from its user, the early years of SQRL's development were marked by a concern that its use would increase its users' susceptibility to spoofing because users would wrongly assume that SQRL fixes everything, including that. We decided SQRL did need to fix that, and now SQRL does.

What, exactly, is CPS?

SQRL's unique "Client Provided Session" (CPS) robustly prevents successful spoofing when SQRL's same-device authentication is used. When the SQRL client indicates to the remote web server being signed-in to that CPS mode is available ①, the web server does **not** authenticate the web browser session which initiated the authentication. Instead, it returns a CPS authentication URL "out of band" to the SQRL client ② which the SQRL client then returns to the user's local web browser ③, which the web browser returns to the remote web server ④ to authenticate its session ⑤.

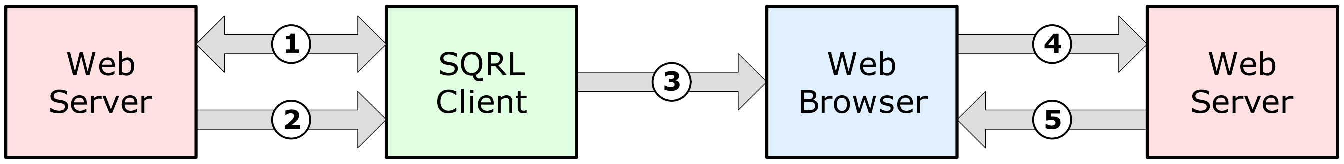

Figure 2: CPS Authentication Flow

1. The web server and the user's SQRL client communicate to establish the user's identity.
2. The web server sends a CPS authentication URL to the user's SQRL client.
3. The user's SQRL client returns the CPS authentication URL to the user's local web browser.
4. The user's local web browser returns the CPS authentication URL to the web server.
5. The web server establishes the user's logged-on session for the browser that returned the URL.

This means that only the web browser physically located in the same system as the SQRL client is able to receive the user's authentication. As a result, any possible remotely located spoofer or middleman is cut out and never receives an authenticated session. Though this is very powerful, its implementation adds significant complexity to SQRL. But it works, and we believe that the absolute protection and assurance it provides is worth the trouble. The system has been well proven and is running on **every** platform – Windows, iOS, Android, Linux, macOS, browser extensions for Firefox, Chrome and Edge's Chromium version.

What's so complicated? The challenge is that the page the user is viewing and authenticating from cannot be trusted since it may be from an intercepted or spoofed website. JavaScript running on the untrustworthy page must not be able to obtain the authentication URL returned to the SQRL client from the web server. SQRL achieves this by having the SQRL client run its own little web server, listening on the local machine's localhost port 25519 for incoming web queries. When the user clicks the page's "Sign-in with SQRL" button, after verifying that the local SQRL web server is running JavaScript on the page performs an HREF jump to the SQRL localhost:25519 server. This terminates the sign-in page, including any JavaScript it might have been running, while the web browser waits for a reply from the localhost:25519 web server.

Once authentication is complete the remote web server returns a "CPS authentication URL" to the SQRL client and it then returns an HTTP 302 Found with the provided URL which redirects the user's local waiting web browser to a logged-on page for this user. This securely establishes the session in a fashion that cannot be interfered with or spoofed.

Handling CPS Details

This implementation guide details the solutions to CPS' various requirements. For them to make sense, here is a brief summary of the requirements that needed to be addressed:

- For reasons that will become clear below, it is necessary for SQRL sign-in pages to have JavaScript enabled. Since this is now expected on the Internet, it is not an inconvenience. But SQRL sign-in pages should check for JavaScript and display a notice to their user that JS must be enabled for SQRL. The pages might "grey out" the SQRL controls unless and until JS is present.
- Mobile devices tend to "sleep" or "suspend" applications (such as a SQRL client) after some period of disuse. If this happens the local SQRL client's built-in web server may not be accepting incoming connection requests from the local web browser. Therefore, the "Sign-in with SQRL" button issues a fetch to SQRL's custom "sqrl://" scheme, which will have been registered by the client with the operating system. This has the beneficial effects of launching the SQRL client if it's not already resident and running or awakening any existing instance of a SQRL client so that it is then able to receive incoming queries to its web server.
- After the "sqrl://" scheme has been triggered, the sign-in page's JavaScript jumps the browser's page to the localhost:25519 web server. However, SQRL sign-in pages should not blindly assume that a localhost:25519 web server is running on the user's computer. If pages do not "look before they leap" curious non-SQRL website visitors who click "Sign-in with SQRL" would cause a nasty browser error and have a bad experience. Therefore, the JavaScript on the sign-in page must verify the existence of a running web server before switching its page there. This is accomplished by having the JavaScript request a randomly named image from the root of the localhost:25519 web server. If the SQRL client is present and awake, it will return a tiny valid image which means that the browser is free to change its page location to the page at the root of localhost:25519.
- SQRL clients indicate "CPS mode" by presenting a flag in their authentication negotiation with a SQRL server. When a SQRL server receives that flag, it must **never** sign-in the user's browser session upon a successful SQRL authentication. Instead, it must return a CPS authentication URL to the SQRL client as its only act toward signing in the user. That URL will be used in an HTTP 301 Found redirection which will be returned to the browser, and **that** browser will be signed-in with the user's SQRL identity.
- As we noted above, when CPS mode authentication completes, the SQRL web server returns an HTTP 302 Found reply. Since this reply contains the authentication token, we **must** assure that this is **only** received by the web browser and **not** by any JavaScript that might be running on an untrusted and possibly fraudulent sign-in page. If JavaScript were to use an XML HTTP Request (XHR), or similar, to query the SQRL web server at localhost:25519, it would obtain the HTTP 302 redirect information returned by the web server, which it could abuse. Fortunately, web browser security strongly enforces the differentiation of any and all script-based queries from browser page fetch queries. ALL script-driven queries contain an "Origin:" header which is not under the script's control – and browser page-fetch queries do not. Therefore, all SQRL clients **must** drop **any** query received which contains an "Origin:" header.
- The item above begins… "when CPS mode authentication completes…". But we need to also handle the case of the authentication not completing. In other words, the user clicks "Sign-in with SQRL" all goes well, and the browser jumps to the SQRL localhost:25519 web server to await its HTTP 301 Found redirection. The SQRL client presents a dialog for the user's confirmation. But rather than proceeding, the user elects to cancel the SQRL authentication. Or suppose that the user is unable to authenticate themselves to their SQRL client, or the remote server does not recognize their identity. The dilemma is that the user's browser has already jumped to the SQRL client's web server and is awaiting an HTTP 301 Redirect. We need to give it something rather than refusing the query and showing the user an unfriendly "Connection Refused by Site" error. This need to cancel a pending CPS authentication is handled by having JavaScript on the web page provide a **cancel**lation URL as a parameter to its root page query. The detailed mechanics of this will be discussed during our next discussion of "The SQRL URL."

These points and requirements will be described in greater detail as we dig deeper into the implementation of SQRL.

The “SQRL” URL

Wherever a website’s sign-in username and password sign-in fields are present, the website may also display a “Sign-in with SQRL” button and a SQRL QR code. Or it might show a smaller, less obtrusive link to a dedicated SQRL sign-in page. Either way, both the sign-in button and the QR code contain a “sqrl://” URL of the form:

`sqrl://{authentication domain}/{page}?nut={unique URL-safe nonce}`

For example: sqrl://www.example.com/sqrl?nut=oOB4QOFJux5Z

SQRL:// → HTTPS://

The “sqrl://” scheme is only used to invoke SQRL clients through scheme-based support in operating systems. The “sqrl://” is replaced with “https://” when the URL is used for Internet communications.

SQRL’s URL Nut

The URL’s ‘nut’ nonce must be unpredictable and single use. It serves double duty as a session nonce to enforce uniqueness for every authentication cycle and as entropy to be signed by the SQRL client. It must be unpredictable to prevent future nut guessing, and it must also be single use so that once a nut has been “consumed” it will not be accepted again. Nuts may be any length, but shorter is better since the entire URL will be encoded into a QR code and lower-density QR codes are smaller, less obtrusive, and scan more reliably. Our recommended nut generator looks like this:

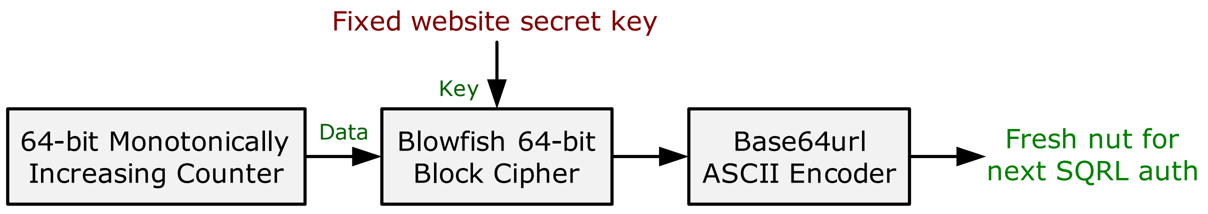
Fixed website secret key
Key
64-bit Monotonically Increasing Counter
Data
Blowfish 64-bit Block Cipher
Base64url ASCII Encoder
Fresh nut for next SQRL auth

Figure 3: Recommended Nut Generator

Bruce Schneier’s excellent Blowfish cipher is chosen since it uses a 64-bit block which is perfect for our purposes. Each website randomly chooses its own unique installation secret key to be used throughout its lifetime. As long as that key remains secret it should never be changed. The secretly keyed Blowfish cipher is fed by a 64-bit counter which the system arranges to only count upwards and to never revert to any previous lower value. 64-bits will never wrap under any conceivable level of usage. Thus, every count ever produced will be unique. This 64-bit counter value is encrypted with Blowfish to feed a Base64Url binary-to-text encoder to produce a 11-character URL-safe text string. This is the SQRL “nut.” Implementations may encode information into longer nuts; “By reference” is preferred.

A fresh, never-before-used nut is issued every time a SQRL sign-in page is displayed. Since single-use is crucial to prevent replay attacks, some means must be provided for assuring that nuts are never reused. This typically means placing the nut onto a “valid nuts” list and removing it once the nut has been consumed or replaced. Since a SQRL authentication requires the use of several authentication cycles, and thus several unique nuts, GRC’s implementation allocates a single “Pending Auth” object for the transaction. It is placed into a traversable linked list with a “last activity” timestamp which allows stale transactions to be pruned. This “Pending Auth” object holds all of the data relevant to the transaction including the most recently issued nut for the pending authentication. As we’ll see later, this “Pending Auth” object holds many other important pieces of authentication data including the IP address of the browser which initiated the authentication. This facilitates one of SQRL’s two important anti-spoofing mitigations. It also holds the MACs (message authentication codes) of any provided by the web server to the SQRL server to prevent in-flight tampering. (Everything the server sends is returned in by the client with its next query, if any, so the server is able to verify the MAC of what the client has returned against the MAC that it created from the data it sent to the client.)

What about random nuts?

The 64-bit counter approach works well for a single server environment. But servers in a large load balanced multi-server environment might need to independently issue SQRL nuts. As long as a high quality source of nut entropy is available, 132 bits of entropy could be encoded into a 22-character SQRL nut (6 bits per encoded char x 22 chars). This approach would sufficiently satisfy SQRL's requirements for unpredictability and single-use and 132-bits of entropy would prevent collisions.

The authentication domain

The "authentication domain" is that portion of the SQRL URL string which is hashed by HMAC-SHA256 to yield SQRL's per-site private key, as shown by this diagram repeated from the SQRL explainer:

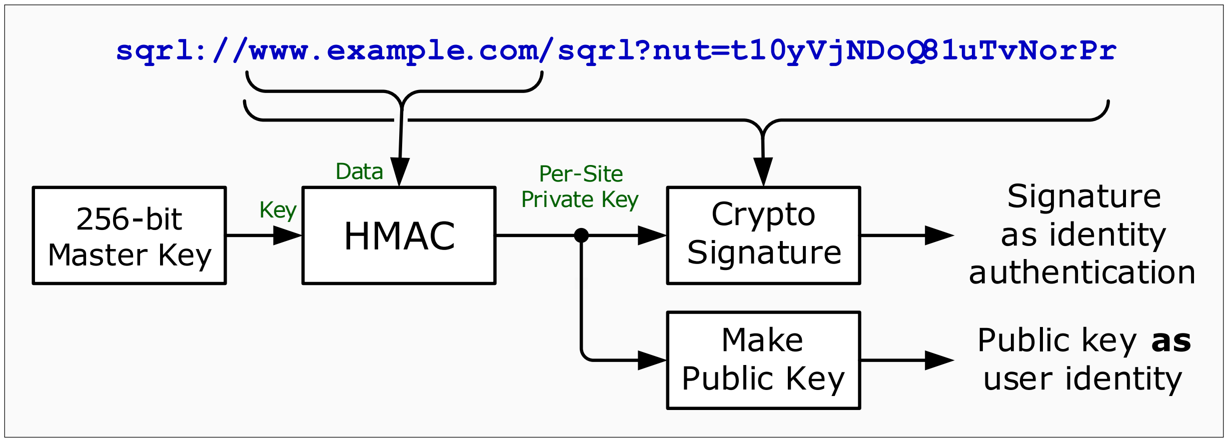
sqrl://www.example.com/sqrl?nut=t10yVjNDoQ81uTvNorPr
Data
Key
256-bit Master Key
HMAC
Per-Site Private Key
Crypto Signature
Signature as identity authentication
Make Public Key
Public key **as** user identity

Figure 4: SQRL Authentication Domain Hashing

Since the user's per-site SQRL identity is entirely dependent upon the precise domain name string hashed, and since issues surrounding username & password, port specification, and alphabetic case can arise, the following authentication domain parsing rules apply:

1. Domain names are 8-bit ASCII "octets" (bytes) compatible with DNS resolvers. Internationalized domain names (IDNs) are represented in standard Punycode ASCII, which is the way DNS sees them. (https://en.wikipedia.org/wiki/Punycode)
2. To preserve DNS domain name case-insensitivity, A-Z uppercase alphabetic must be converted to lowercase.
3. Simple domain names, as shown in the example above, begin immediately after the initial "sqrl://" scheme specification and end immediately before the first "/" is encountered.
4. The formal URL specification provides for a username and password prefix using the format: "username:password@www.example.com". Since URLs are not secure for information transport this is rarely encountered, though "username@domain" is sometimes seen. Therefore, as the URL is parsed from left to right after the scheme://, the start of the domain name will be reset to the right of the last @ encountered.
5. Port number specifications in URLs may be used to override the implicit port 443 assumed for HTTPS connections. And it is foreseeable that a SQRL protocol server might be operating on a non-standard port. But a port specification must not alter the effective SQRL domain. Therefore, any port specification must be removed from SQRL's effective authentication domain.

Path Extension

Some websites, notably GitHub, use the beginning of the URL's path to delineate individual accounts within the website's domain. SQRL's "Path Extension" feature supports separate GitHub-style domains by allowing the SQRL URL to specify the use of additional characters beyond the formal end of the domain name. For example, a sub-site SQRL URL might look like this:

```
sqrl://{authentication domain}/joey?x=5&nut={unique URL-safe nonce}
```

If the SQRL URL's parameter list contains the reserved parameter name 'x', then the decimal value associated with 'x' specifies the number of characters to extend the authentication domain to the right of the domain name – but never into the parameter tail beginning with '?', or past the end of the URL. In the example above x=5, so the five characters, starting with and including the domain-terminating '/' are included at the end of the hashed authentication domain. And, if 'x=10' it only appends '/joey'.

SQRL URL examples

The following examples clarify the use of the preceding URL parsing rules:

SQRL URL	Auth Domain	Rule shown
`sqrl://ExAmPlE.cOm/?nut=...`	`example.com`	domain lowercased
`sqrl://example.com:44344/?nut=...`	`example.com`	ignore port override
`sqrl://jonny@example.com/?nut=...`	`example.com`	ignore username@
`sqrl://Jonny:Secret@example.com/?nut=...`	`example.com`	ignore user:pass@
`sqrl://example.com/jimbo/?x=6&nut=...`	`example.com/jimbo`	extend auth domain
`sqrl://EXAMPLE.COM/JIMBO?x=16&nut=...`	`example.com/JIMBO`	extension's CASE and end extension at '?'

The CANcel parameter

When a SQRL client is used in the same machine as the web browser being signed into, clicking the "Sign-in with SQRL" button causes the browser to execute a page-changing HREF jump to a web server the SQRL client has established on the system's Localhost port 25519. The act of jumping to this resident web server terminates the operation of the sign-in page, and any scripting it may have been running, while the browser waits for the newly requested page to be delivered.

We will discuss what happens then, shortly. But for the moment consider that the user changes their mind and wishes to cancel the SQRL operation. Their web browser has jumped to the built-in SQRL web server and is waiting to be told what to do next. If the user cancels the operation, we need to be able to return the web browser to its original page as if the user had never initiated the authentication. So, if the URL the web browser uses to query SQRL's built-in web server contains a cancel parameter, SQRL's built-in web server can respond to the browser's waiting query with an HTTP 301 Redirect to return the web browser to the page it just left, to back out of a cancelled same-device authentication.

The cancel parameter will typically be an encoded version of the sign-in page's URL. So, JavaScript on the SQRL login page will obtain the URL of the current page, base64url-encode it, and append the additional parameter to the sign-in button's URL when the page is loaded.

An example of an actual sqrl:// URL for a "Sign-in with SQRL" button looks like this:

```
sqrl://sqrl.grc.com/cli.sqrl?nut=oOB4QOFJux5Z&
can=aHR0cHM6Ly9zcXJsLmdyYy5jb20vYWNjb3VudC9jb25uZWN0ZWQtYWNjb3VudHMv
```

As mentioned previously, any webpage offering SQRL authentication should offer both a "Sign-in with SQRL" button for a visitor to click, and a SQRL QR code for the visitor to scan. Each of these contains a SQRL URL with a common nut which the webserver associates with the visitor's browser session. But the URLs are NOT identical because the button's HREF contains a URL with a cancel (can=) parameter whereas the SQRL QR code does not. As noted above, the cancel parameter is required to aid in the smooth functioning of SQRL's most powerful anti-spoofing feature: Client Provided Session or CPS.

The QR code *could* also contain the cancel parameter. It wouldn't do any harm. But the cancel parameter's value string will probably be long, and QR code data should be kept as small as possible. Therefore, since the cancel parameter serves no purpose in cross-device SQRL authentication, it's much better to exclude it.

Forming the Cancel parameter

Since the cancel parameter's value will be the full URL of the site's SQRL sign-in page, it may contain characters which are not URL parameter value safe. So, the cancel URL is base64url encoded before being used as the "can=" parameter's value. The SQRL protocol often employs base64url encoding and the optional trailing '=' equals sign padding that often appears at the end of the encoding are unnecessary and should be removed from the end of the encoded value.

The astute web developer might wonder why SQRL clients don't simply use the Referer header in the browser's query as the place to return the browser if the user cancels the operation. It would have been nice if that could have been done. But browsers have become increasingly careful to trim tracking information from their Referer headers. Moreover, whereas the SQRL sign-in page will certainly be HTTPS, the SQRL client's little built-in web server is HTTP, and all browsers trim their Referer data when jumping from a secured to a non-secured page.

What if no cancel parameter?

It's possible that a poorly designed website might fail to provide a cancel parameter to a local SQRL client. If this were to happen, the client would have a dilemma on its hands if the user chose to cancel the operation. It would have accepted a connection and query from the user's web browser, which is now waiting for its reply, but the SQRL client would have nowhere to redirect the waiting browser. In this case, the SQRL client should return a simple webpage to the browser to show its users something like: "You have cancelled the SQRL sign-in. Please press your web browser's BACK button to return to the previous login page. That simple page might also contain JavaScript to pop the browser's page history stack to automatically return the user to the previous page. The SQRL specification requires the use of an explicit cancel parameter since that provides additional cancellation flexibility to the website, and also since the page returned by the SQRL client will not be HTTPS, it will be HTTP and it is foreseeable that browsers or add-ons might proactively block executable script from non-secured origins.

Why HTTP and not HTTPS?

The Referer header note above and the preceding paragraph noted that the web server running in the SQRL client uses HTTP rather than HTTPS and TLS. In other words, when the user clicks the "Sign-in with SQRL" button, the browser is redirected to **http:**//localhost:25519/... This might strike some as reckless, but it is not insecure. Encryption and authentication are only required when communications are subject to interception. But, despite the fact that we're using a protocol normally used for network communication, the use of a system's local networking stack is **not** network communications, it is inter-process communications and it is no less secure than if the protocol were encrypted.

We now know how to parse any sort of complex SQRL URL, what to exclude, what to include, what to lowercase and how to form SQRL URLs for SQRL's use including its optional path extension and cancel parameters.

```
sqrl://steve:badpass@SQRL.grc.com:8080/demo/cli.sqrl?x=5&nut=oOB4QOFJux5Z&
can=aHR0cHM6Ly9zcXJsLmdyYy5jb20vYWNjb3VudC9jb25uZWN0ZWQtYWNjb3VudHMv
```

Armed with this understanding of the SQRL URL format and handling, let's rewind and examine the overall processes of same-device and cross-device authentication.

Authentication Setup

A SQRL user wishing to sign into a website arranges to display the site's SQRL sign-in page. As we have seen, one way or another, that SQRL sign-in page arranges to present a "Sign-in with SQRL" button and to display a SQRL QR code. The "sqrl://" URL carried by those contains a SQRL nut that has never been used before. There are many ways a system can accomplish this, and the specific approach taken is left to the developer. For the sake of clarity, we'll enumerate a few typical systems:

1. The web server might directly obtain a guaranteed unique and unpredictable nut nonce from an encrypted incrementing counter or from a source of high entropy, and directly embed the SQRL URL into the "Sign-in with SQRL" button and also generate the QR code image containing the SQRL URL. In this model the web server would supply a fully formed webpage to the user's browser.
2. Since JavaScript is needed, anyway, to support SQRL sign-in automation, the page's JavaScript can be given the additional task of getting the page setup. So, the web server would send a generic sign-in page containing JavaScript whose task is to customize the page. The JavaScript would query either the webserver or perhaps a separate SQRL server to obtain and register a unique nut. It would set the URL for the sign-in button, and either synthesize the image of a matching SQRL QR code in JavaScript, or set the URL of the QR code's image "src=" parameter so that the web browser queries either the webserver or the separate SQRL server for the QR code image content.

In either of those instances, the request for a new nut will either create or update a "Pending Auth" object residing in either the webserver or the SQRL server. Once created, that object will be indexed or searchable by the nut most recently issued for the authentication, and will contain information relating to the pending authentication including a timestamp used to remove unused and expired authentication objects, the IP address of the user's browser, the most recently issued nut, and other management information the servers might require.

At this point our path diverges depending upon whether same-device or cross-device authentication occurs…

Internal vs External Browser Clients

This document's discussion of localhost connections to port 25519, HTTP 301 redirections and so on, assumes that the SQRL client lies outside the web browser, either as a standalone SQRL application or possibly as an operating system service. This is the way GRC's Windows client and the clients for iOS and Android operate, since they are freestanding clients providing SQRL client services to any web browser running on the local machine. But this is not the way the SQRL browser extension for Firefox and Chrome operates, nor would it be the way SQRL clients would operate if they were to be integrated natively into the web browsers themselves.

SQRL clients internal to the browser do not invoke any services of external SQRL clients. They have no need to connect to an external client's webserver over localhost port 25519, receive an HTTP 301 rediretion, or anything else. SQRL-aware web browsers or web extensions operate by parsing the "sqrl://" URLs found on SQRL logon pages and modifying the DOM (Document Object model) contents as needed to assume all of the functions of external clients.

The internal client captures the user's "Sign-in with SQRL" button click, then initiates the connection to the remote SQRL server specified in the "sqrl://" URL. It displays any required user interface prompting to authenticate its user, uses the SQRL protocol to negotiate the user's identity with the remote SQRL server, then changes the browser's page to the URL provided by the server's CPS reply.

SQRL sign-in pages are created with the assumption that **no** internal SQRL support is provided by the web browser or any browser extension. Therefore, if SQRL browser or extension support is present, that support must intercept and override the page's normal behavior which always assumes the use of external SQRL clients.

Same-Device Authentication

SQRL’s same device authentication is all about leveraging the anti-abuse power of client provided session (CPS). The user clicks the website’s “Sign-in with SQRL” button to initiate authentication with the SQRL client which has been installed in, and is running on, the same device as the web browser.

Local SQRL clients residing outside of the browser (external clients) are triggered by **both** of two separate and different mechanisms: triggering the operating-system-registered “sqrl://” scheme and jumping to the running client’s localhost webserver which is listening for HTTP queries on localhost port 25519. (Note that the SQRL client webserver must only bind to the machine’s 127.0.0.1 IP and interface and **not** also to the WAN interface!)

On the desktop, triggering the operating system’s “sqrl://” scheme launches a new instance of the client, and on mobile platforms it awakens the mobile client which might be sleeping or suspended. If the newly launched SQRL client does not find another instance of SQRL already running it will remain running. This allows SQRL clients to be terminated until they are needed and then run on demand. If the newly launched SQRL client **does** find another instance of SQRL already running, it uses private inter-process communication to hand off the SQRL URL to the already-resident process instance.

The newly launched client instance also transfers its “focus rights” to the resident process. Modern operating systems actively prevent “focus stealing” by applications which the user has not brought to the foreground with some manual action. When the user clicks “Sign-in with SQRL” on the browser, it will have focus. But we then want the SQRL client to pop-up a SQRL authorization password or Quickpass prompt, and we want that dialog to have the system and keyboard focus upon appearance so that the user can enter their password without needing to first click on the prompt’s input field.

The http://localhost:25519 web query trigger

The magic sauce of SQRL’s Client Provided Session (CPS) is that the resident SQRL client runs a small web server to which the user’s browser jumps in order to eventually receive an HTTP 301 Redirect URL to a logged-on website session. Therefore, any query to the root of http://localhost:25519/* which does not match any other defined and reserved queries (see below) will be treated as the browser’s request for a 301 Redirect.

The http://localhost:25519/*.gif query

Resident SQRL clients must respond to queries for a GIF image of any name. They only need to return a 1x1 pixel image, but they must return a valid image. This feature allows web pages to probe for the presence of a local running SQRL client and to avoid jumping the user’s browser to a non-existent client web server when a page’s “Sign-in with SQRL” button is clicked. If this “look before you leap” test was not performed, the browser’s attempt to jump to http://localhost:25519/ would result in a messy and uncool “Connection Refused” error from the web browser. Since the resident SQRL client might have been put to sleep or suspended by a mobile operating system, or a desktop client might not be running yet, the JavaScript code running on the page should not give up after a first failure. JavaScript on the page will have also triggered the operating system’s “sqrl://” scheme (described in detail next) so the SQRL client might be loading or waking up and need some time to reply. The page should keep retrying failed image loads (after a short delay) until it succeeds. And under no circumstances should the page’s JavaScript jump the browser to the Localhost server until and unless a /*.gif image load succeeds. To avoid any possible caching, the JavaScript should use a GIF name derived from a high-resolution date timestamp so that every name is guaranteed to be unique.

The http://localhost:25519/stop.bmp   /favicon.ico   /sqrl.ico

The SQRL client’s web server may choose to serve additional images to support any other pages it may wish to serve, such as a manual authentication cancellation instruction page if the authenticating website does not provide a CANcel (&can={…}) parameter in its localhost:25519 jump query so that the client has nowhere to redirect the web browser after a user-cancelled or failed authentication.

The sqrl:// scheme access trigger

The sqrl:// scheme trigger addresses different needs on desktop or mobile operating systems.

Under Windows, and presumably macOS and Linux, applications are actively prohibited from "stealing" input focus from each other. When the user clicks the "Sign-in with SQRL" button on their browser, the browser has the system's input focus. A password prompt dialog which pops-up due to the browser's localhost:25519/ query will neither have focus nor any way to programmatically obtain it. But when the SQRL unlock password appears, the user wants to be able to immediately enter their password without first clicking on the pop-up password dialog. By launching a new instance of the SQRL client, **that** new instance will have focus rights and can programmatically hand them over to the running client, thus allowing the user to immediately enter their password.

The scheme's role on mobile systems is to awaken any possibly suspended and sleeping SQRL client. Mobile systems tend to "sleep" applications after some period of disuse. But the SQRL client needs to be able to respond to the /*.gif image probe request and then to the browser's jump to the root of its web server. To do this the client must be awoken. The sign-in page's trigger of a "sqrl://" scheme performs this task.

The "Same IP" check

Early in the development of SQRL it was clear that SQRL's sign-in automation would open it to spoofing attack. The most worrisome attack is the easiest to perpetrate: A SQRL user unknowingly signs into an untrustworthy website which obtains a SQRL URL link and QR code from, for example, Citibank's website. If the user is in a hurry and/or never appreciated the necessity of **always** double-checking the domain name they are signing into, they'll enter their Quickpass and authenticate **their** identity to the SQRL URL the malicious site obtained from Citibank, *thus signing the malicious site into Citibank **under their identity.***

A closely related spoofing attack is more involved and also more insidious: A malicious website obtains a look-alike domain name for a popular website, say amaz0n.com. A spoofed e-mail induces the user to visit amazon.com but takes them, instead, to "amaz0n.com" and presents a lookalike site. Using some excuse, they are asked to login. The SQRL user now sees the real "amazon.com" they are being asked to login to and, even if they **did** carefully double-check the site, they see what they expect to see… amazon.com. Yet the malicious site would, again, be logged on as them.

In these scenarios, unlike when using a username or e-mail and password, the use of SQRL prevents the malicious actor from obtaining the user's authentication credentials. They cannot sign-in again later, nor anywhere else using the same credentials. They only receive a single session sign-in. But that's still not desirable for a system that hopes to significantly advance the state of the art for remote network identification and sign-in.

This section has the name "The Same IP check" because both of the scenarios described above share a common feature: When the malicious server requests a SQRL URL link and QR code from the valid website, it will do so from **its** IP address, not from the user's IP. When the user's local SQRL client then connects to the website's SQRL server to use the URL provided to it by the malicious server, the SQRL client's IP will not match the IP which originally fetch the URL. The "Same IP" address test will fail, the remote SQRL website will immediately fail the authentication and return an error code to the client.

In short: When the user's web browser and SQRL client occupy the same machine for same-device authentication, the IP address making the initial request for the authentication SQRL URL, and the IP address of the SQRL client that attempts to use that SQRL URL, should **always** be the same. This is a sanity-test, in addition to CPS, that SQRL applies to all same-device authentications. Smartphones signing into a website on their own browser are using same-device mode, so they also obtain the combined anti-spoofing strength of "Same IP" and CPS.

However, a smartphone on its own cellular network being used to sign-in to a session on another computer using a SQRL QR code will be in cross-device mode where its SQRL client IP will **not** be expected to match the IP of the other machine. This means that cross-device mode cannot guarantee either "Same IP" or CPS for anti-spoofing protection. However, if the smartphone is preferentially using a Wi-Fi network, it might assume that it **does** present the same public IP as the browser it is signing in. So, in that case, the very valuable "Same IP" protection might be available.

Since SQRL servers must be informed whether or not to expect an IP match or a mismatch, and since SQRL clients always know whether they are being used in same-device or cross-device mode, and also perhaps whether they are using a cellular or local Wi-Fi network, all SQRL clients include a **`noiptest`** flag when they need to suppress the SQRL server's normal behavior of checking for SQRL URL and SQRL client IP match.

Cross-Device Authentication

Compared to same-device authentication, SQRL cross-device authentication is simple. It is also super sexy and almost magical in the way it works:

While any SQRL sign-in page is being displayed and is visible in the foreground, that page's JavaScript is periodically querying (polling) the website's SQRL server or service using the 'nut' received to ask the server whether any change should be made in the currently displayed page. If a site wishes, it might choose to have its SQRL sign-in JavaScript establish a single persistent connection to await a page change reply, but this may overtax busy systems. In any event, the specific mechanism is left to the implementor.

With the SQRL sign-in page waiting, the SQRL user scans the page's displayed QR code which displays the domain name to which the user's identity will be authenticated. As will be discussed extensively below, because SQRL's cross-device authentication mode lacks the protection provided by Client Provided Session (CPS) and probably also "Same IP", asking the cross-device authenticating user to carefully confirm the domain their identity will be provided to is very important.

Once the user confirms that they wish to authenticate to the indicated domain, the SQRL authentication handshakes will complete and the SQRL server will sign the user into the web browser session which first requested the "nut" and which has been "pinging" for any update. Upon receiving the next update query from the web browser, the SQRL server will instruct it to change the page as appropriate, which will show the user newly signed-in and ready to proceed.

What makes this authentication mode so striking is that from the user's perspective, they didn't touch the page. They let their smartphone "see" the page's SQRL QR code, they authenticated themselves to their smartphone and double-checked the authentication domain, and without touching the machine being logged in to, the page updated itself to reveal that they are now logged in. The effect is quite dramatic.

That's the good news.

The bad news is that, as we have seen above, because it may lack the protections provided by either CPS or "Same IP", SQRL's cross-device authentication is also **much** more dangerous to use.

To drive this point home very clearly:

The undeniable problem with cross-device authentication is that a website can EASILY spoof ANY inattentive SQRL user!

And, in fact, as that second example above shows using a lookalike site "amaz0n", even a more attentive user, who didn't notice that they were at "amaz0n" rather than "amazon" could be fooled.

Any time **`noiptest`** is used, some form of strong anti-spoofing countermeasure should be employed. A useful though admittedly burdensome mitigation would be to show the SQRL user, in big bold letters, the domain name to which they are providing their SQRL identity and then require them to manually enter the domain name they are authenticating to into an input field on their smartphone. In the first example above, the user would be required to enter **`citibank.com`** into their phone. Anything less will likely result in abuse of SQRL's ease of use in cross-device mode where Same IP and CPS are absent.

However, the resistance to this within the user community has been insurmountable because this makes SQRL's magical instant authentication and sign-in much less magical and instant.

If you are a web developer you may be thinking that there **must** be some way to solve this, some way around this. If you come up with anything, you'll be a hero. This problem has received a great deal of time and attention to no avail. Several crucial pieces are missing from cross-device mode which are all present to merge synergistically in same-device mode:

1. The user's web browser jumps to the local SQRL client web server. In doing so, any possible code running on the page that has been jumped away from, malicious or not, is terminated while the browser waits for a reply from the local SQRL client's web server.
2. Rather than signing the user into the browser session that requested and probably displayed the original SQRL sign-in page, which might not be trustworthy, the web server being signed into returns an authentication URL directly to the user's SQRL client.
3. The user's browser, and nothing else (no possible JavaScript), receives that authentication URL directly from the SQRL client in an HTTP 301 Redirect. This causes it to jump to the page provided. Nothing can intercept this redirection to the signed-in page.

The obvious way to mimic this functionality with a smartphone in cross-device mode on a different IP would be to have the authenticating website return for display an authentication URL which the user would then manually enter into the browser's URL address field. But no one thinks that's practical. However, ...

OS Support for Cross-Device CPS

If SQRL wins the day to become the preferred website sign-in system, operating systems could easily incorporate very simple assistive support for SQRL's cross-device authentication to imbue it with the "unspoofability" of SQRL's full-strength Client Provided Session (CPS) protection:

A keystroke or menu click would prime the system's camera to receive a **CPS** URL. Upon completing a cross-device authentication, the website's CPS URL would be displayed on the smartphone's screen and presented to the waiting PC. Upon seeing the URL, the operating system would send the URL to the system's registered browser, causing it to open a new signed-in page. The original waiting SQRL sign-in page, which has been polling for an update, would receive a "close yourself" command from the SQRL server, and the authentication would be complete.

Since the CPS URL is being delivered to the user's smartphone and then to the computer in front of them, we obtain the same benefits as same-device CPS authentication in a cross-device mode.

Same-Device or Cross-Device?

With Android and iOS mobile platforms increasingly replacing laptops and desktops to become true *personal* computers, where individuals are authenticating themselves to websites visited on their devices, SQRL's support for **same-device** authentication – with full CPS and "Same IP" protections – on those mobile platforms, is crucial.

SQRL's cross-device authentication excites people because it is new and because no one has seen anything like it before. But when we stop to think about it, no one today is using their mobile devices to log into websites on the web browser of another computer. Is it cool that we can with SQRL? Yes, absolutely. But when a SQRL client with the user's SQRL identity is present on those systems, it's actually easier to just click the "Sign-in with SQRL" button and you're done.

So, while there will be many cool instances where cross-device sign-in will be useful, the lack of SQRL's protections against spoofing in cross-device mode will be significantly mitigated by the fact that same-device sign-in will almost certainly be SQRL's most often used mode.