

hackmd.io

Keri Request Authentication Mechanism (KRAM) - HackMD

46–58 minutes

v0.1.8

Originally I solved this decentralized end-to-end non-interactive authorization problem as part of a proof-of-concept for a privacy preserving lost and find registry and peer-to-peer messaging service. This proof-of-concept was implemented in python in an open source Apache2 project called Indigo-BluePea which may be found here. [Indigo BluePea](#)

Interactive vs. Non-interactive Authentication Design

Authentication mechanism broadly may be grouped into two different approaches, these are: interactive and non-interactive approaches. An interactive mechanism requires a set of requests and responses or challenge responses with challenge response replies for secure authentication. Non-interactive approaches on the other hand pose unique problems because they do not allow a challenge response reply handshake. A request is submitted that is self-authenticating without additional interaction. The main benefits of non-interactive authentication are and path independent end-to-end verifiability. These benefits become more important in decentralized applications that employ zero-trust architectures. (By zero-trust we mean never trust always verify, this means every request is independently authenticated, there are no trusted pre-

authenticated communication channels.)

For non-interactive authentication of some request for access, the most accessible core non-interactive mechanism is a non-repudiable digital signature of the request itself made with asymmetric key-pairs. The hard problem for asymmetric digital signatures is key management. The requester must manage private keys. Indeed this problem of requiring the user to manage cryptographic keys, at least historically, was deemed too hard for users which meant that only federated, token based, authentication mechanisms were acceptable. But given that it's now commonly accepted that users are able to manage private keys, which is a core assumption for KERI in general, then the remaining problem for non-interactive authentication using non-repudiable digital signatures is simply replay attack protection. Indeed with KERI the hardest problem of key management, that is, determining current key state given rotations is already solved.

The closest authentication mechanism to what KERI enables is the FIDO2/WebAuthn standard [FIDO2/WebAuthn](#). The major difference between FIDO2/WebAuthn and KERI is that there is no built-in automated verifiable key rotation mechanism in FID02/WebAuthn. FIDO2/WebAuthn consists of two ceremonies, a registration ceremony and then one or more authentication ceremonies. Upon creation of a key-pair, the user engages in a registration ceremony to register that key pair with a host. This usually involves some MFA procedure that associates the entity controlling the key pair with the public key from the host's perspective. Once registered then individual access may be obtained through an authentication ceremony that typically involves signing the access request with the registered private key. Unfortunately FIDO2/WebAuthn has no in-stride verifiable key rotation mechanism. Should a user ever need to rotate keys, that user must start over with a new registration ceremony to register the new key pair for that user entity. Whereas with KERI rotation is happens automatically with a rotation event

that is verified with the pre-rotated keys. So given one already has KERI verified key state, using FIDO2/WebAuthn to authentical replay requests would be going backwards.

Any access control or authorization mechanism based on KERI must still have some additional registration mechansim in order to link or associate an actual entity as the controller of a given identifier. However, in this case where we are enabling access for the purpose of replaying key event logs, it may not be necessary to link to any external entity. We merely need authenticate to the controller of the identifier regardless of the actual external entity. Thus authentication can be simply restricted to authenticating as the controller of the identifier via its current key state without requiring a separate regisraion step to link to an external controlling entity. With this we can build a base level private authorization policy for a given identifier using a simple authentication mechanism based only on proof of control over the authorized identifier by signing a request or query with the current controlling key pairs of that identifier. This does not preclude an implementation from layering on additional authentication and authorization mechanisms such as external entity associated registration and MFA. But the minimal simple authentication mechanism proposed here is meant to enable this layering without requiring it.

Replay Attack Protection in Non-interactive Authentication

A digital signature made with an assymmetric key pair(s) on a request provides non-repudiable authentication of the requester as the controller of that key pair(s). This assumes that the recipient of the request (requestee) can securely map the identifier of the requestor to the controlling key pair(s). With KERI derivation codes, we have a simple mechanism for attaching and resolving both the identifier of the requestor and the requestor's signature of that

request to that request. Given that the requestee has the current key event log for the requestor, then the requestee may securely verify the mapping from requestor identifier to its associated current key state. Indeed we could view the publication of the key event log of a given potential requestor to a given requestee as an ersatz registration ceremony that then enables later authentication of signed request messages that include both the requestee's identifier and signature as attachments to the request. Presentation is often used to a signed request as an authentication.

The problem is that an attacker may capture the signed request and then resubmit it or replay it from its own host, thereby effectively fooling the requestee into redirecting a copy of the response to the attacker's host in addition to or instead of the original requestor's host. In other words the attacker acts as an imposter. This requires that the attacker intercept the original request and then replay it from a different connection. This is called a replay attack. There are several mechanisms one can employ to protect against this form of replay attack. However the practical choices for non-interactive protection are limited. In general replay attack protection imposes some form of timeliness to any signed request and some form of uniqueness to any signed request.

We are assuming that only the current keys from the current key state may be used for authentication. This requirement imposes one form of timeliness, in that any requests signed with stale keys are automatically invalid. The requestor can ensure that the requestee has its, (the requestor's) latest key state so that it is protected from replay attacks using old stale key signed requests.

The simplest and most practical non-interactive mechanism for ensuring timeliness and uniqueness is to insert a date-time stamp in the request body. This date time stamp is relative to the date time of the requestee not requestor. There are two protection mechanisms, one loose and one tight. In the loose mechanism the

requestee imposes a time window with respect to its date time for any request. The time window is usually some small multiple of network latency of the connection to the requestor. An imposter must intercept and resubmit the replay attack within that window in order to successfully redirect a response to itself. This is loose protection because the requestor is not guaranteed to be able to detect the attack. The requestee may respond to multiple copies of the same signed request but from different source addresses. In addition to timeliness a tight mechanism imposes a uniqueness constraint on the request timestamps. The requestee does this by keeping a cache of all requests from the same requestor identifier (not host address). All requests in the cache must have monotonically increasing timestamps. The requestee only responds once to any request with a given time stamp and any subsequent requests must have later timestamps. This means that a successful replay attack may be detected because the only way that the attacker gets a reply response is if redirects the one and only one response to that request first to itself before possibly forwarding it on to the requestor (if at all). This redirection is detectable because either the requestee does not get the response at all if it is not forwarded or it gets it via a later redirection from some host that is not the originating requestee. Given that the requestor detects the attack it may then take appropriate measures to avoid future interception of its traffic by the attacker.

The monotonic cache even protects against a retrograde attack on the system clock of the requestee. The monotonicity requirements means that even if the system clock is retrograded to an earlier time such that an attacker could replay stale requests, the cache would have later timestamps and therefore prevent response until the system clock was restored to normal time.

The datetime stamp should have fine enough resolution that the monotonicity constraint doesn't limit the rate of requests nor cause it to run past the end of the timeliness window. Microsecond

resolution is adequate for most internet connections but may not be for 40 Gbps LAN connections. An ISO-8601 timestamp with microsecond resolution is a common variant. Most operating systems now also support nano or atto time variants of ISO-8601 timestamps.

Authentication Request Timeliness and Caching

Generally speaking all authentication mechanisms, both interactive and non-interactive, depend on some explicit or implicit datetime stamp for timeliness. Interactive mechanisms that use tokens typically embed the datetime stamp or its hash in the token itself. This is because tokens are bearer authentication mechanisms and need to expire quickly to prevent replay. A common non-token based interactive authentication mechanism requires that the requestee send a challenge response with a nonce to the requestor. The requestor then includes the nonce in its signed challenge reply. This ensures uniqueness of the signed request (as challenge reply). But even then it's still best practice to have a timeout on the authentication because an attacker that intercepts the traffic can still replay the challenge response unless the requestee enforces one and only one reply per request. This is especially problematic in asynchronous networks since the routes by which the set of request, challenge, response, reply messages are exchanged could all be different and each subject to MITM attacks.

One way to mitigate MITM attacks on async interactive authentications using a nonce (challenge response) is for the requestee (host) to keep a cache of challenge responses and only reply once to any nonced signed response. The nonce must be a salty nonce generated by a CSPRNG instead of merely a nonce. The salty nonce must have sufficient cryptographic entropy that it's universally unique, i.e. the host can never generate the same nonce for any two interactive requests. At scale this means either a database of all requests which will grow without bound or there

must be a timeout mechanism on a cache to allow old entries to be deleted.

Absent such a database or timeliness cache, an attacker might capture the reply from the requestee to the signed challenge response and replay the response as a signed bearer token. The attacker can forward the reply to the requestor to hide the fact that it has intercepted the response and reply as a MITM. To reiterate, any intermediary that intercepts the signed response now has a bearer token it can replay unless the requestee enforces one and only one request per nonce. Thus either a database that grows without bound or some timeliness mechanism must be used to enforce the one and only one reply per request.

One timeliness mechanism could be that the requestee imposes a timeout on nonced reply requests that invalidates any stale replayed nonced requests. This approach enables the requestor to detect an attack because there is one and only one response to any request. Given a successful attack, either the requestor does not get the response or it sees that the response has been redirected from some other host that is not the requestee.

Consequently, from a replay attack protection perspective, a nonce with a time window in an interactive authentication is functionally equivalent to a monotonic time stamp in a non-interactive authentication but the latter avoids the overhead of the interaction. Both employ a timed cache (timeliness) tied to each requestor and both employ a mechanism that ensures one and only one response is sent for any signed request (uniqueness).

The typical use case for interactive authentication is not on an asynchronous channel but to use a synchronous channel such that the host can pair each request and response and replay without needing a database. In a sense, a synchronous channel provides a weak form of timeliness given the ordering of messages from the time of the establishment of the synchronous channel. This sort of

works if the host enforces that there may be one and only one outstanding request-challenge-response-reply interaction at a time by caching the latest. So a replay must be the one outstanding request and it is assumed that a prior request could not be replayed without re-establishing the connection. This means the channel is blocked until either the interaction times out or succeeds. It does provide replay attack protection since any message must either be part of a the current interaction or start an new interaction with a request never a signed response.

This only gives the illusion of protection, however, because even on a synchronous channel there may be a MITM that can replay requests out of order. The nonce by itself does not order the requests which means the MITM can reorder requests so change the transaction state. This is technically not a replay attack, it's a first play attack, but can create similar exploits. Therefore, notwithstanding a synchronous channel, all requests must still be cached so the host can ensure one reply per request in the originally intended order.

Therefore except for a couple of extreme corner cases, there is little advantage to interactive authentication over KRAM's non-interactive authentication. KRAM with a timed cache monotonically orders the requests and ensures no replays and make retrograde attacks detectable. For strict monotonic ordering use a sequence number in addition to the timestamp. KRAM would not be feasible without KERI becasue it depends on key state and signed requests relative to that key state. But given one has KERI there is good reason to leverage it to build a truly scalable and secure request authentication method, e.g. KRAM.

This analysis just reinforces that observation that all practical secure authentication replay attack protection mechanisms include both timeliness and uniqueness properties.

Caching by intermediates

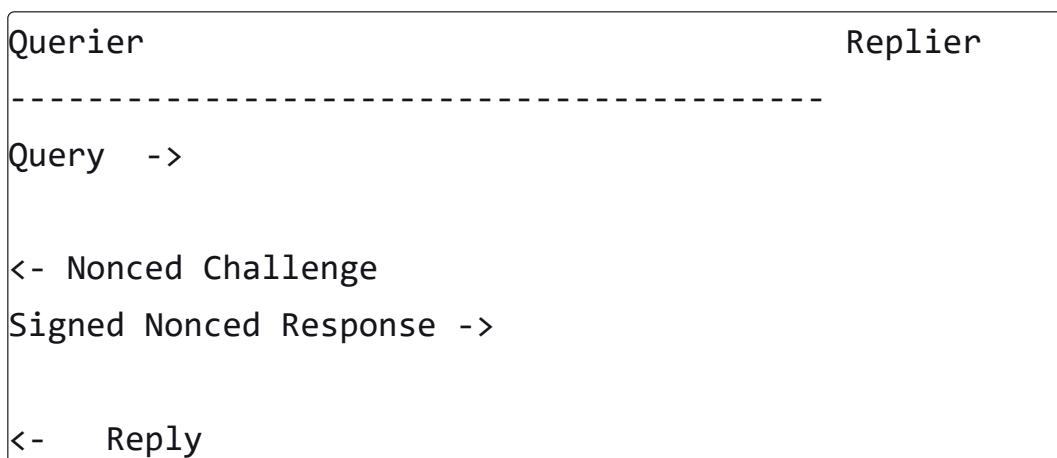
Because generally replay attack protection mechanisms use timeliness, caching of authentication requests by the requestee or some intermediate load balancer may be an anti-pattern. This applies to both interactive and non-interactive authentication requests. Thus, in general request caching of authentication requests must be disabled. Replies, however, may still be cached.

In a zero-trust non-interactive environment where every request is self-contained and self authenticating with an embedded date time stamp and attached identifier and signature couple, request caching would be problematic. This may be viewed as a trade-off where the scalability advantages of non-interactivity exceed the scalability disadvantages of not being able to cache requests.

Examples

There is some terminological ambiguity because of the extra two messages that are the nonced challenge and nonced response to that challenge.

So to better remove ambiguity lets not use request but query. This uses KERI parlance.

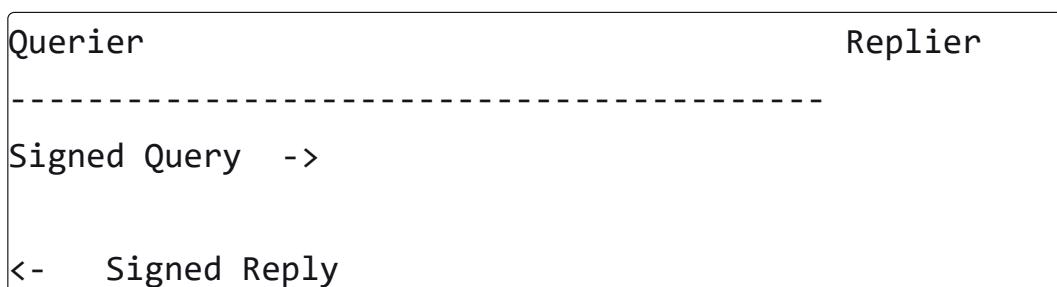


A replay attack is the successful reuse of a prior Signed Response.
A First-play attack uses Signed Responses out of order.

This approach only works if the channel is synchronous and the Replier caches the current/last interaction (at least the Signed

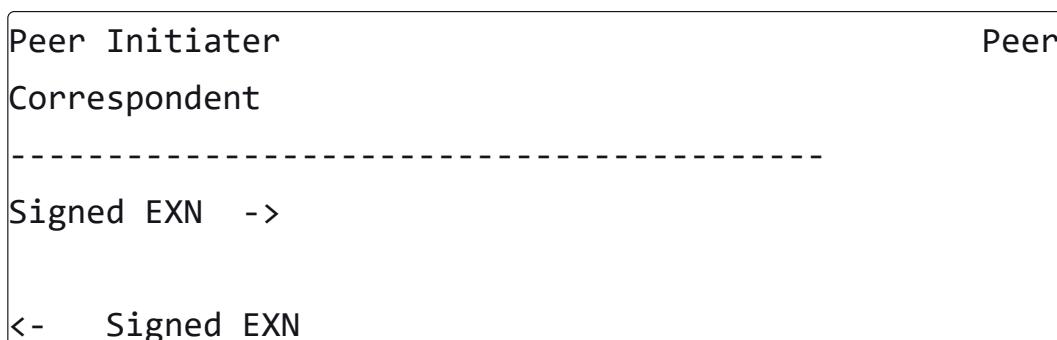
Nonced Response. This means only one interaction per channel is allowed at any time. This does not work on an asynchronous channel unless a database of all interactions is kept which grows without bound unless a monotonic ordering is placed on the interactions so that the database may be pruned of stale interaction which effectively creates a synchronized set of channels. This means at the least putting time stamps in the Query messages and then keeping the latest Interaction for each Querier and dropping any interactions that are earlier. Given this timed ordered cache all the querier has to do differently is sign the query and now you have exactly KRAM. There is no need anymore for the Nonce challenge response set of messages.

With KRAM you have



But the Query MUST have a datetime and should have a SAID The Signed Reply should include the Query's SAID The Replier has a sliding timed window cache that enforces that any Query's datetime must be later than any so far previously received query.

KRAM works just as well on asynchronous channels and peer-to-peer exchanges not simply query-reply between client and server. So KRAM can be applied to exchange messages



Both sides keep a timeliness cache of EXNs from the other party to protect from replay attacks.

Given that routes in the EXNs define an interactive protocol, i.e., a transaction of a given type. Then, the timed cache can be per transaction type per Peer, not just per Peer. The timeout on the timed cache can then be of any size bounded only by the cache's memory across all simultaneous live transactions of a given type for all peers.

Because each EXN includes the SAID of the prior EXN in the transaction, and the first EXN message in the transaction has a datetime stamp, it is not vulnerable to First Play attack.

The transaction itself is ordered and self-synchronizing.

Summary

Simple KRAM

Simple kram uses a short time window relative to the host (i.e the host clock is determinative). All signed client requests must include a timestamp field that lies within the window else they are dropped. This does not require any host cache. A replay attack is only possible inside the time window synchronized to the host clock. A short time window minimizes the opportunity for replay attack. But simple kram may be problematic for multi-sig where the time to collect signatures is longer than the time window. This is what is currently implemented in keripy.

Concisely, simple KRAM works by including in the over-the-wire message a timestamp referenced to the network time clock of the recipient. Network time means time synchronized to well-known ubiquitous network time servers.

Because the timestamp is referenced to the time as seen by the recipient, the originator of the message can't lie about time. The recipient has a moving time window whose size is some small

multiple of the average network latency and drift of network time and surrounds the current time as seen by the recipient. If the timestamp of the received message lies outside this window, it is dropped. This window limits the time during which a replay attack can be mounted. If the average time between subsequent messages sourced by any given originator is larger than this window, then no replay attack is possible. Therefore the protective efficacy of simple KRAM is better the smaller the window.

For example, let t represent the current time seen by the recipient, let d represent clock drift and skew, let l represent average network latency, and let m represent some integer multiple. Then the simple KRAM window would be the interval $[t-d-m*l, t+d]$. The timestamp of the received message must lie within that window. Typical values could be $d = .01$ seconds, $l=1$ second, and $m=3$.

However, simple KRAM may be problematic for multi-sig AIDs for several reasons, which will be discussed later. One has to be careful when building exchange-message-based protocols using multi-sig to avoid violating a very small time window. This is not a problem for full KRAM.

Full KRAM

Full kram is a monotonic timed cache. The last request from any client is cached inside a sliding time window. A new request must both have a later datetime than the cached request and must be inside the host time window. Requests that move outside the sliding time window may be safely deleted thereby pruning the cache. The timed cache can be made more granular by caching a request per message type for a given client or even more granular by caching messages with a specific time window for a given transaction itself not merely the message type or the transactions type. Essentially, creating an expiration datetime as the request datetime plus the host configured timewindow for all instances of a given message at

a given stage in a transaction. The monotonicity of the cached requests protects against replay attacks. The time window bounds the size of the cache. That has not been implemented yet in keripy. The timewindow needs to be controlled by the host relative to the host's clock otherwise the client can attack it.

With a host-controlled sliding window timed monotonic cache, because the monotonicity of the cache protects against a replay attack and the time window merely bounds the memory requirements, one can tune the window per message type (and per AID) and/or per transaction type or per transaction itself, so that messages associated with certain types of transactions or certain transactions themselves can have relatively long time windows (days or weeks) which windows are only bounded by memory and the traffic load for those message from given clients.

Full KRAM employs a strictly monotonically ordered timeliness cache to protect from replay attacks. It includes protection from retrograde attacks on the recipient's clock. The timeliness cache can be granularized to have different caches per transaction type so that interleaved messages from multiple transactions do not interfere with each other. Moreover, when the messages include their own non-timeliness, i.e., numeric ordinal, then both ordinals, timeliness, and numeric, can be employed in concert to account for asynchronous networks that deliver messages out-of-order but that are not replay attacks.

Simply speaking, a timeliness cache stores the timestamp of the last message received from any source AID. Whenever a new message is received, its timestamp must be later than the timestamp of the stored message. If so, the later message is accepted and the cached timestamp for that AID is updated. This timeliness cache may employ a stale cache release where it includes a lagging time window relative to the current network time seen by the recipient. When an AID timestamp cache is empty, new

entries in the cache must be later than the oldest end of the time window. Any cached timestamps older than this lagging time window can, therefore, be safely deleted. A replay attack of an already received message will not be later than a cached timestamp, and if the latest received message is already older than the lagging time window, then any received messages must be sooner than that, so it is not actually a replay.

To protect against retrograde attacks on the receiver's time server or internal clock, the receiver saves a single timestamp of the latest time it sees. If the network time is even older than the latest saved time, the receiver can refuse to accept messages until the clock catches up.

The timeliness cache can be made more granular to save timestamps not merely by source AID but by some vector of attributes such as all of or a subset of the following: (source AID, message type, transaction type, transaction ID).

The timeliness cache can be stored in memory only or persistently on disk. The latter consumes more resources and may be less performant but does not induce retries in the event of a reboot of the receiver's process.

Importantly, the lagging time window for releasing cached timestamps can be quite long in duration. It is really only bounded by memory or disk resources for the cached time stamps. This means it could be days or weeks long.

This can largely relieve any latency or ephemerality problems with group multi-sig AID signed over-the-wire messages. Especially if the cache is granular, i.e. per the vector (AID, transaction type or transaction ID). This way, if a given exchange message is signed with a given timestamp by one member of the group and then not fully signed by a threshold number of other members of the group until days later (assuming the lagging window is days in duration), it could still be accepted by Full Kram.

For example, let t represent the current time seen by the recipient, let d represent clock drift and skew, let l represent the lag duration. Then the full KRAM window would be the interval $[t-d-l, t+d]$. The timestamp of the received message must lie within that window. [NTP Clock Skew](#) Typical values for clock skew are around 10 milliseconds with worst case usually 100 milliseconds. So a value for d would be some integer multiple of 10 milliseconds, say $d = .1$ seconds that is 100 milliseconds. We could let l be on the order of days or weeks, say $l=2$ weeks.

If we have two network nodes with really bad network time clock skew we could set d to be an integer multiple of the worst case clock skew of 100 milliseconds, like say 200 or 300 milliseconds.

The clock's resolution determines how many messages can be received in order in any timeframe before the sender must block to avoid running past the leading edge of the receiver's time window. The timestamp in KERI exchange messages uses a microsecond resolution, which supports up to 1 million messages with unique monotonically increasing time stamps per second per timeliness cache entry. This is discussed in more detail below in the comment labeled time resolution.

When a message is received, the first filter is to test if its timestamp lies within the interval. If not, the message is dropped. The next filter is to check if a cache entry already exists for the source AID, message type, transaction type, and transaction ID. If so, then check to see if the timestamp of the received message is earlier than that of the cached message, if so, drop the received message. If the timestamp is the same or the message is the same, then verify the attached signatures. If verified, then accept (idempotently for the message) and process attachments. If the timestamp is later, then verify the attached signature(s) and accept if verified, and then update the cache, replacing the existing message entry with this new one.

In addition to the on-demand message receive filtering, scan the cache periodically and prune any entries that lie outside the interval $[t-d-1, t+d]$.

The above approach could work with group multi-sig in the following manner: A given message sourced by a multi-sig AID must be verifiably signed by at least one member of the group. If the message is the first received message, it must lie within the receiver's lagging time window with respect to its current time. A new cache is created with the time stamp, message type, and transaction ID. The message itself and its attached signatures are added to a partially signed escrow.

If the message is not the first message received for a given timestamp, message type, and transaction ID, but is the same as the cached one, then the message is accepted by full KRAM, and any attached and verified signatures are added to the partially signed escrow for that message. The same message but with different but verified member signatures is not considered a replay attack. The signatures are added to the database in an idempotent manner i.e., if a signature already exists, nothing changes. As long as the lagging time window nor the escrow timeout expires while collecting a threshold satisfying number of signatures, the message will be accepted and moved out of escrow.

In many cases, merely implementing full KRAM in this way for group multi-sig non-key-event messages, namely, exchange, query, reply, prod, and bare, will solve the current problems.

Summary

Any of the policies works with full KRAM, be it single-level with escrow or two-level policy with single attached signatures and embedded payload signature or with a threshold and pre-protocol to collect signatures. Moreover, with granular full KRAM each transaction type could employ a different acceptance policy.

Complications of multi-sig AID sourced messages and KRAM

The primary complication of multi-sig sourced AID messages with KRAM is deciding what policy a given KRAM instance applies to accepting a message sourced from a multi-sig AID. One policy would be to accept the message if any one of the members signed it and then escrow the message in order collect signatures from copies of the message signed by other members. The second would be to require that the received message include a threshold satisfying number of signatures which in turns requires a pre-protocol to collect signatures before sending. The pre-protocol obviates the need for an escrow to collect signatures.

Both policies depend on a lagging time window that is long enough to collect signatures from the members. The timestamp in the message must lie within the lagging window, but this lagging window may be set to be large enough that loose coordination between the members is practical.

For simple KRAM, the only practical policy is to accept the message if it is signed by any one of the group members and its timestamp lies within the much tighter time window. But then the message is not protected by the threshold of the group multi-sig. Employing an escrow to collect signatures after acceptance does not help because all the to-be escrowed signatures must still arrive within the narrow time window, which means the coordination of the group members must be tight enough to fit in this window. This is the root problem of using simple KRAM with group multi-sig AIDs.

Two-Level Simple KRAM

A viable modification to the single-sig acceptance policy with simple KRAM is to use a hierarchical or two-level approach. In the two-level approach the first level employs simple KRAM to authenticate the over-the-wire exchange message as a wrapper using single-sig

from any member and the second level authenticates the embedded payload using the full multi-sig threshold. The second level does not employ KRAM. To clarify, the authentication of the over-the-wire conveyance using the exchange message as a wrapper is different from the authentication of an embedded payload so wrapped.

To further elaborate, in the two-level method, the over-the-wire conveyance is authorized, given a single signature from any group member, and must satisfy the simple KRAM replay attack protection window. Whereas the embedded payload, which is not subject to KRAM replay attack protection, is separately signed from its exchange message wrapper and is only authenticated when the signatures on the embed satisfy the multi-sig threshold. This requires each member to generate and sign an exchange wrapper with a timestamp separately, which is subject to KRAM, and also sign the embedded payload, which is not subject to KRAM. Once past simple KRAM the embedded payload goes into an escrow to collect signatures asynchronously from the group members.

The embedded payload may be a tunneled exchange message that just leverages the processing of exchange messages. The tunnel is meant to cross through simple KRAM.

A limitation of this approach is that the escrow timeout for exchange messages must be long enough to support loose coordination when collecting signatures from the group members. If this is days or weeks, then the escrow must also keep partially signed messages in escrow for days or weeks.

This is the current supported approach. This allows escrow of the embedded payloads to collect signatures asynchronously until the threshold is satisfied. It has the drawback that each member must sign twice, the wrapper and the embedded payload. Whereas the full KRAM solution for group multi-sig messages described above only requires a signature on the wrapper.

Two-Level Simple KRAM with pre-protocol

A variant of the two-level method with simple KRAM is to use a pre-protocol to collect the signatures on the embedded payload so that a designated member can send a single-sig signed wrapped with multi-sig signed payload all at once. This is the motivation for the above-referenced HAMI issue. This approach avoids having to deal with the escrow timeout while collecting signatures. The HAMI protocol uses a different mechanism for collecting signatures asynchronously given very loose coordination time frames.

Time Windows

We should avoid transaction-specific windows but use transaction typed windows. Transaction specific windows have the potential for a prune attack (see below). If some transactions of a given type need longer or shorter KRAM cache windows with respect to other transactions of the same type, then we want to add a modifier to transaction types, that is, the window type. This way, any transaction of a given window class gets the same window size.

So we would have two tables. A lagging window size table. Indexed hierarchically as follows. For the non transactioned message types (qry, rpy, pro, bar) we have: KEY = MessageType.WindowType and VALUE = window size For the transactioned message type, (exn) we have: KEY = MessageType.TransactionType.WindowType and VALUE = window size

The corresponding replay caches have the following structure: For non-transactioned message types

KEY = SourceAID.MessageType.WindowType.MessageID and
VALUE=Timestamp

For transactioned message types

KEY =

SourceAID.MessageType.TransactionType.WindowType.TRansactionID.MessageID

and VALUE = timestamp

The TransactionID and MessageID are SAIDs (ie hashes) Each message has a SAID, d field and each exchange message has a prior message SAID, p field. Which means the SAID of the first message in a multi-message exchange transaction can be used as the TransactionID.

Putting a salty nonce in the modifier q block of the first exchange message of a transaction makes the transaction ID universally unique even when all the other fields are the same. Why not then use universally unique transaction IDs for replay attack protection? Unfortunately, we still need a timestamp to know when we can prune any given transaction. Otherwise, we must store all transactions forever in order to prevent replay attacks. The timestamp orders transactions monotonically so that we create the property of "stale" transactions that can both be pruned and not replayed.

In KERI v2, we have a xip transaction inception message that normatively defines the first message in a given transaction. The exchange exn message then populates its x field with this transaction ID. But in v1 we just have to book keep which exchange message was first and verify the other messages by walking the prior hash links back to the first.

Prune Attack and Classified Windows

The reason we want window durations to be classified is that when durations are per transaction, not a class of transactions, then when a given transaction is not in the cache, we don't know what default window to apply to decide if we can create a new cache value. If we apply a default that is longer than the one the transaction itself uses, then when we prune it, we have a gap where the replay is possible.

So we want to unambiguously classify every message as belonging to a window duration class. If we can assume that all messages of a given type have the same duration, then the message type is synonymous with the window type. Likewise, if all transactions of a given type have the same window duration, then the transaction type is synonymous with the window type.

However when they are not synonymous, then we need a way to explicitly unambiguously associate a given message with a window size from information in the packet itself. One way to do this would be to use the route, r field, or a field in the route modifier q block when we can't simply use either the message type or the combination of message type and transaction id to determine the window size.

Typically we would use the route r field for the transaction type. So we would either extend the route path to have a window type differentiator or better use a field in the route modifier q block to provide that differentiator.

I suggest then defining a vector of attributes as the key for each cache entry will work. In keripy lmdb we use a tuple to derive the key for an given database entry. So as long as the tuples are well structured you can mix and match in the same database. Some entries have more elements in thier tuple than others. This just changes the b-tree location. The only place this complicates things is the the pruning of the cache has to walk the tree to find all the entries. Which again is not hard to do. We do it in other places.

So for those cache entries that need custom window sizes you define a tuple that includes sufficient detail to differentiate them. Like source AID, message type, and for exns, then also transaction type, and then transaction ID. the final element in either case (exn or not) is the window duration. or the window duration goes in the database entry not its key space. In a b tree, essentially anything in the key is also retrievable as a value. Onc can retrieve whole

branches as a collection of values.

Time Resolution and Throughput

In a timeliness cache used for replay attack protection, there is a limit on throughput that is a function of the clock resolution. For KERI exchange messages the clock resolution of the timestamp is microseconds. Most modern operating systems support clocks with resolutions in nanoseconds. But microseconds should be good for the foreseeable future. When they no longer are then we can version KERI to support higher resolution timestamps.

Limitations

Given a microsecond clock there are 1M time slots available per second for monotonically increasing timestamps in a given cache. Each cache entry gets its own 1M per second set of slots. So it's inherently parallelizable.

Lets say worst case we only have 1 cache entry per source AID for all exchange messages from that source AID. This means that in any given second that Source AID can send at most 1M messages to a given Dest AID. When it runs out of slots it stops sending and waits for the clock to tick up, and then a new set of slots are created. If we have a cache entry per transaction, then each transaction gets 1M time slots per second. So highly parallelizable.

Let's be more specific. Recall that the receiver's KRAM acceptance window is $[t-d-1, t+d]$ where d is some multiple of the clock skew, and 1 is the lagging window which could be on the order of days or weeks.

So the worst case scenario is that the sender sends enough messages fast enough to fill all the time slots and runs ahead of the leading edge of the window at $t+d$. This is only going to happen on very high speed networks. Given there are 1M messages per second and the minimum size of a signed exchange message is

about 300 bytes then we have a throughput of 300 MegaBytes per second. Or 2.4 Gigabits per second. So, a high-speed network.

Let's suppose the worst case, which is that on such a high-speed network, there is zero network latency (every millisecond of network latency effectively provides an extra 1000 time slots because the receiver's clock has incremented by a millisecond after the source message was timestamped and before it was received. On such a high-speed network, a reasonable worst-case clock skew is 10 ms, so we can set d to 100 ms.

Given the 10 ms worst-case clock skew, the senders' clock could be 10 ms ahead of the receivers' clock. The leading edge of the receiver's window is at $t+d = t + 100$ ms. This means that there are 90 ms worth of microsecond time slots ahead of the sender before it runs ahead of the receiver's leading edge. This means the sender could send as many as 90,000 messages in that 90 ms to the receiver before the sender ran ahead of the leading edge and would have to block waiting for network time to tick up. But this is only true if network latency is truly zero. Recall that each ms that a message takes to traverse the network, the receiver's clock has added another 1000 time slots.

So the sender has to batch send at least 90,000 messages at once, each with a monotonically increasing timestamp where it increments each message's timestamp by one microsecond before any messages have timestamps that could possibly run ahead of the leading edge of the receiver's time window and would therefore be dropped by the receiver's full KRAM timeliness clock

Practical considerations

High-volume transaction infrastructure is usually measured in tens of thousands of transactions per second. So, with reasonable values for the time window clock skew, we have a capacity of 90,000 pipelined transaction messages per second if we do not

partition the traffic by transaction but lump all transactions into one timeliness cache per source AID. This means we are multiplexing and pipelining the transactions into one ordered batch.

Consider that in a practical, interactive transaction, at each stage of the transaction, the sending party waits for a response from the other party before sending its next message. So the one-way turnaround and traversal time (parsing and verifying SAIDs and signatures) on a stage in an interactive exchange would have to take less than 1 microsecond per one way to consume all the available 1-microsecond slots and, therefore, be blocked at the sender or dropped at the receiver. A typical turnaround time for an exchange message will be in the tens of milliseconds. So, it is highly unlikely that any given transaction would ever run into any clock skew based limitation. We are, therefore only worried about the case where there are 90,000 simultaneous transactions between a given sender and receiver that are all concurrent and share the same timeliness cache.

If we reach that limit, the receiver simply makes its timeliness cache more granular. Instead of one cache entry per AID per message type, it can have one cache entry per AID per message type per transaction type or even more granular to one cache entry per message type per transaction type per transaction ID. Now, it would have 90,000 slots per transaction. It would be entirely surprising that anytime soon, any infrastructure would need to process a single transaction between two parties with 90,000 interactive stages that all happen in 10 ms, i.e., a full round trip since we are not pipelining transactions on a single cache entry. Therefore, each round trip must happen in less than 1 microsecond to fully consume all the available time slots for that cache entry.

Therefore we can safely conclude that microsecond clock resolution on our timeliness caches is more than adequate for the foreseeable future.

Asynchronous Out-of-order Messages

The above analysis assumes that message from the same sender are delivered in order at the receiver. In an asynchronous network with routers and gateways using the Internet there may be multiple paths through the network, so a given set of messages when conveyed asynchronously may show up out of order due to different latency for different paths. This means that the timeliness cache would drop a message with an earlier timestamp that showed up later than a message with a later timestamp.

There are several ways to address this problem.

The first is to recognize that interactive transactions induce a natural ordering because the two parties take turns. One solution therefore is to use a timeliness cache entry per transaction. That means that a given sender won't send a subsequent message unless:

1. a retry timer has timeout out so it sends a retry of a previously sent message.
2. the other party has responded
3. a transaction stage timeout timer expires so the sender gives up on the transactions and sends a Nack.

So in the case of interactive transactions with one transaction per cache entry,

1. is not a problem because if it's truly a retry, it won't matter to the transaction if it's the first try or some later retry that is delivered first. Since there are retries, then it won't matter if a message is lost because it will send a retry.
2. It is not a problem because if the other party has responded, then the first party knows that its original try was already accepted by KRAM before it sends another message.
3. in this case, the transaction timeout needs to be long enough to

account for the worst-case path latency difference, and the sender, once it decides to cancel a transaction, can't reverse that decision. It must restart the transaction. This way even if the first message is not lost and the nack shows up after the first message, and the other party responds, the response is ignored, and then when the other party gets the nack it also cancels the transaction.

Therefore out-of-order messages are only a problem when we are pipelining messages from multiple transactions into the same timeliness cache entry to maximize throughput. In this case out-of-order messages between transactions could interfere with each other. But then again this is only a problem if the transactions have no reliable retry mechanism. If the transaction itself is unreliable, then one should be using a reliable synchronous channel where one is guaranteed that messages are delivered in order and no messages are lost.

But for the use case where the transactions include retry capability, i.e. the transaction is reliable, then even if multiple transactions are pipelined interleaved with messages from other transactions it won't matter if some messages are delivered out of order as long as the rate of out-of-order messages is small relative to the number of interleaved transactions because the retries will repair any dropped messages.

To conclude, when one is forced to use asynchronous transport then it is recommended that one use non-interleaved cache entries, i.e. one cache entry per transaction. When one is forced to use interleaved transactions on the same cache entry then it is recommended that the one use a reliable channel where messages are delivered in-order. Otherwise one has to depend on a reliable transaction with retries to overcome any out-of-order messages on an unreliable channel when using and interleaved cache entry.

Afterword

A little bit of historical perspective may help. In the early days of computing, there were no network time servers. Consequently timestamps had no meaning. Each party could just lie about time so time was untrustable. Historically a nonce was just a sequence number. To be a nonce it only had the property of being used once. There are other ways to create a nonce. A datetime can be a nonce, as can a hash of a datetime or the hash of a sequence number etc. In the early days CSPRNGs (cryptographic strength pseudo random number generator) were both rare and computationally expensive. So a Salty Nonce was hard. What was common were private networks (the internet did not exist yet). Private networks could have really strong synchronization properties. So given these conditions, using extra messages that do not depend on a time stamp but use a simple nonce (not a Salty Nonce) was reasonable. Moreover, in those days (and for many protocols still used in the wild) signing did not mean digital signatures with asymmetric crypto but merely an HMAC (a hash using a shared secret key). There using an interaction with an extra set of messages i.e. a nonced challenge response for replay attack protection, made some sense. Often some CS professor puts a simple mechanism in a text book as an academic exercise to teach a principle and then it gets adopted in the real world by former students not because it was well thought out but because it was familiar.

But today where we assume a KERI context, we can already assume CSPRNGs, assymetric digital signatures tied to keystate of AIDs, cryptographic strength hashes, salty nonces, robust ubiquitous access to network time servers and a requirement to work over asynchronous public networks. The cryptographic operations have gotten relatively cheap given multiple exponential Moore's law increases in computation performance per cost. What has not scaled relatively exponentially is bandwith per cost. Synchronous networks do not scale as well as asynchronous

networks and almost all communication now happens over public networks. So refactoring the solution to support asynchronous networks at scale and eliminate 1/2 of the packets is a huge design win.