hackmd.io

# Security in KERI/ACDC - HackMD

30–39 minutes

---

In KERI we have three classes of uses for signatures each with a different level of security with different security guarantees.

## High

KEL backed either in KEL are anchored to KEL. This means the signatures on the events in the KEL are strongly bound to the key state at the time the events are entered in the KEL. This provides the strongest guarantee of duplicity evidence so that any verifier is protected. The information is end-verifiable and any evidence of duplicity means do not trust. A key compromise of a stale key can not result in an exploit because that would require forging an alternate version of the KEL which would be exposed as duplicity. Key compromise of the current key-state is recoverable with a rotation to the pre-rotated key(s) (single-sig or multi-sig) and pre-rotated keys are post-quantum proof. A compromise of a current key-state is guaranteed to be detectable by the AID's controller because the compromise requires publication of a new KEL event in infrastructure (i.e. witnesses) controlled by the AID controller. This limits the exposure of a successful compromise of the current key-state to the time it takes for the AID controller to observe the compromised publication and execute a recovery rotation.

The ordering of events in the KEL is strictly verifiable because the KEL is a hash chain (block chain). All events are end-verifiable. Any data anchored to these events is also end-verifiable. All these properties are guaranteed when data is anchored to a KEL, i.e., KEL backed. Any information that wants to be end-verifiablly authentic over time should be at this highest level of security. ACDCs when anchored to a KEL directly or indirectly through a TEL that itself is anchored to a KEL have this level of security.

## Medium

BADA-RUN (described in the OOBI spec) This imposes a monotonicity on order of events using a tuple of date-time and key-state. The latest event is the one with the latest date-time for the latest key-state. This level of security is sufficient for discovery information because the worst case attack on discovery information is a DDoS where nothing gets discovered. This is because what gets dicovered in KERI must be end-verifiable (anchored to a KEL). So a malicious discovery (mal-discovery) is no different than a mis-discovery or a non-discovery. The mitigation for such a DDoS is to have redundant discovery sources. We use BADA-RUN for service end-points as discovery mechanisms. We of course could anchor service endpoints to KELs and make them more secure but we want to make a trade-off due to the dynamism of discovery mechanisms and not bloat the KEL with discovery anchors. Because the worst case can be mitigated with redundant discovery its a viable trade-off.

Monotonicity protects against replay attacks and stale key compromise impersonation attacks. It does not provide strict ordering because there may be missed events and stale events

but the last seen event always wins. So the only event that is end-verifiable is the last seen event but that event may be stale.

BADA means best available data acceptance. The specific BADA policy provides guarantees that the latest data-state cannot be compromised via a replay of an earlier data-state where early is determined by the monotonicity of the associated (date-time, key-state) tuple. It also protects against forgery of new data-state given the compromise of any *stale* key-state (not the latest key-state). It does not protect against the forgery of new data-state given a compromised of the latest key-state. It does not provide ordering of all data-states but ensures that the data-state last seen by a given verifier cannot be replaced with an earlier data-state without compromising either the latest key-state of the source or by compromising the storage of the verifier.

To reiterate, because the montonic (date-time, key-state) tuple includes key-state then a verifier is protected from any compromise of an earlier (stale) key-state relative to that which has been last seen by a verifier (latest key-state).

To elaborate, the only key-compromise that can succeed is that which corresponds to the latest key-state seen by the verifier. The exposure to exploit of the source's latest key-state is then bounded by the time it takes for the source to detect key compromise and then perform a rotation recovery. An eco-system governance framework could impose liability on the source for any compromise of the latest data-state for a given latest key-state since it is entirely under the control of the source to protect its key-state from compromise. The difference between BADA and KEL backing is that with KEL backed update a compromise of the current key state is always detectable because the attacker must publish a

new event to the KEL on infrastructure (such as witnesses) controlled by the AID's legitimate controller. Whereas with BADA there is no mechanism that guarantees the AID's legitimate controller can detect that its current key has been compromised.

One way to mitigate this exploit (non-detectability of current key-state compromise) is to periodically KEL anchor a batch of BADA signed data. This then imposes a maximum time for exploit of the current signing key-state. Any BADA verified data that is the latest seen data by a verifier but whose time stamp is older than the current batch time period window could then be reverified against the batch anchor for its appropriate time window. If that data is not included in its time window's batch anchor then this could be flagged by the verifier as likely duplicitous data and the verifier would not trust and could re-request an update from the AID controller. This is a hybrid of KEL-backed and BADA-RUN security.

To elaborate, because BADA only protects that latest data-state it is only appropriate for data where only the latest data-state is important. Unlike token based systems, however, BADA does not invalidate of all latest data-states every time the key-state is updated (rotated) where such invalidation would thereby force a refresh of all latest data-states. This property of not forcing auto-invalidation upon key rotation makes BADA suitable for asynchronous broadcast or gossip of the latest data-state in support of an authenticatable distributed data base. Whereas the forced auto-invalidation of data-state with token based systems limit their usefulness to access control applications (e.g. not data-bases). With BADA, however, an update of data-state only automatically auto-invalidates prior data-states. But that means that it is not useful for verifiable credentials or similar applications

that must have verifiable provenance of old data-states. For example when old data-states must still be valid such as a credential issued under a now stale key-state that must still be valid unless explicitly revoked.

Abeit weaked than KEL backing, BADA is a significant performance and security improvment over token based systems. It also has performance advantages over KEL backing. As a result BADA is approapriate for information that does not benefit from verifiable ordering of all data-states but only the latest data-state such as a distributed data-base.

In BADA-RUN, the RUN stands for Read, Update, Nullify and is a replacement for CRUD in an API. CRUD does not protect from replay attacks on the data-state because a delete erases any record of last seen which destroys the monotonicity of the (date-time, key-state) tuple. Likewise Create makes no sense because the API host cannot create a recored only the remote client can. Moreover, the function of a secure BADA Update must account for both first seen and last seen so a Create verb would be entirely superflous. Nullify deactivates a record without losing the monotonicy of update guarantee. RUN provides the appropriate verbs for any API that applies the security gurantees of a BADA policy to the data-state of its records.

## Low

KRAM (KERI Request Authentication Mechanism) https://hackmd.io/ZbVAbNK1SPyT90-oNwN_cw

The lowest is for ephemeral query/reply mechanisms that protect against replay attacks and key compromise attacks at the moment

not over time. This is done with KRAM which is a non-interactive replay attack algorithm that uses a sliding window of date-time stamps and key state (similar to the the tuple as in BADA-RUN) but the date-time is the replier's not the querier's. KRAM is meant to protect a host when providing access to information to a client from replay attacks by a malicious client. It is not meant to protect the information provided by the host. For that we must use BADA-RUN or KEL backing. Thus, by itself KRAM is not suitable to protect on-disk storage (see below).

## On Disk Storage

Both KEL-backed and BADA-RUN are suitable for storing information on disk because both provide a link between the keystate and date-time on some data when a signature by the source of the data was created. Bada-run is too weak for important data because an attacker who has access to the database on disk can overwrite data on disk without being detected by a verifier hosting the on-disk data either through a replay of stale data (data regression attack) or if in addition to disk access the attacker has compromised a given key-state, then the attacker can forge new data with a new date-time stamp for a given compromised key and do a regression attack so that the last seen key-state is the compromised key-state.

With BADA, protection from a deletion (regression) attack requires redundant disk storage. At any point in time where there is a suspicion of loss of custody of the disk, a comparison to the redundant disks is made and if any disk has a later event given BADA-RUN rules then recovery from the deletion attack is possible.

KRAM on a query is not usable for on disk storage by itself. Because its just a bare signature (the datetime is not of the querier but of the host at the time of a query). But the reply to a query can be stored on disk if the querier applies BADA to the reply. To elaborate, Data obtained via a KRAMed query-reply may be protected on-disk by being using BADA on the reply. This is how KERI stores service endpoints. But KERI currently only uses BADA for discovery data not more important data. More important data should be wrapped (containerized) in an ACDC that is KEL backed and then stored on-disk

In the hierarchy of attack surfaces. Exposure as on disk (unencrypted) is the weakest. Much stronger is exposure that is only in-memory. To attack in-memory usually means compromising the code supply chain which is harder than merely gaining disk access. Encrypting data on disk does not necessarily solve attacks that require a key compromise (because decryption keys can be compromised) and it does not prevent a deletion attack. Encryption does not provide authentication protection. But encryption does protect the confidentiality of data.

The use of DH key exchange as a weak form of authentication is no more secure than an HMAC for authentication. Its sharing secrets and anyone with the secret can impersonate any other member of the group that has the shared secret.

AFAIK very few in the DID world even mention these type of security concerns for did docs standards. This paper Five DID Attacks highlights some attacks to which did:webs should NOT be vulnerable. So when a pull request exposes did:webs to a known attack, it should not be accepted.

## JWS

Section 10:10 for JWS Replay attack protection is for interactive mechanisms where one party produces a nonce, gives it to the signing party then the signing party returns the JWS with the nonce. This is a one time only mechansims. Its only good point-to-point. Is not scalable and can’t be used for broadcast or gossip. Its not BADA. What is needed is a time stamp created by the signer and that time stamp must be in the signed data. Also in the signed data must be the AID of the signer. Then an attached signature on the named and datetime stamped data can be evaluated using BADA rules.

The important question to be answered is what role does the did doc play. Is the did doc the source of the data or the recipient of the data for which the did doc is acting as a relay. We certainly can use JWS for BADA but its only part of the puzzle.

Anything in the unprotected header of a JWS is untrustworthy because it can be forged.

I suggest looking at a keri reply message with attached signature for a service endpoint and then figuring out how to encode that as the body of a JWS with attached signature. That gives you an example of what needs to be replicated. Putting stuff in a protected header of a JWS is problematic for KERI because KERI embeds the SAIDs and AIDs and datetimes as metadata in the signed container so that one can’t do a malleability attack on the container's meta-data with a compromised signature or with an impersonation attack. JWS associates the meta-data to a given signature via the protected header. This exposes JWS to malleability attacks. With ACDC every signer must sign the same

data, there is no per signer meta-data. The security philosophy of JWS with respect to metadata is over a decade old and is outdated. It works for point-to-point. It works for tokens and closed loop verification but its weak for the issue hold-verify-model but is broken for multi-sig.

So the only thing that should NOT be in the payload of the JWS is the signature and the algorithm type fields for that signature algorithm. The payload of a JWS is equivalent to an ACDC or reply message container as a whole. The per signature JWS protected header is not to be used for anything but metadata about the signature itself, i.e. the signature algorithm type. In general the JWS protected header fields were designed as metadata for access control in a closed loop verification policy using tokens.

So if we map JWS appropriately, then its not JWS that is the problem, its making sure to use JWS in the BADA way or in a KEL backed way. But if its KEL backed then there is no need for a JWS because the KEL is already signed. So the only viable reason to use JWS is for data that is compatible with the weaker security guarantees of BADA-RUN but wants to be interoperable with tooling that depends on JWS for evaluating signatures.

Whois data could be used with BADA-RUN whereas did:web aliases should not (leads to an impersonation attack). So we could have a database in the sky aka a did doc that uses BADA-RUN if we modify the DID CRUD semantics to be RUN semantics without necessarily changing the verbs but by changing the semantics of the verbs. Then any data that fits the security policy of BADA (i.e. where BADA is secure enough) can be stored in a DID DOC as a database in the sky. For sure this includes service endpoints for discovery. One can sign with CESR or JWS signatures. The

payloads are essentially KERI reply messages in terms of the fields (with modifications as needed to be more palatable) but semantically the same. The DID DOC just relays those replies. Anyone reading from the did doc is essentially getting a reply message and they then apply the BADA rules to their local copy of the reply message.

To elaborate, we modify the DID CRUD semantics to replicate RUN semantics. Create becomes synonymous with Update where Update uses the RUN update. Delete is modified to use the Nullify semantics. Read data is modified so that any recipient of the Read response can apply BADA to its data (Read is GET). So we map the CRUD of did docs to RUN for the did:webs method. Now you have reasonable security for signed data. If its KEL backed data then just use an ACDC as a data attestation for that data and the did resolver becomes a caching store for ACDCs issued by the AID controller.

Architecturally, a Read (GET) from the did resolver acts like how we handle reply messages for resolving service endpoint discovery from an oobi. The query is the read in RUN and so uses KRAM. The reply is the response to the read request. The controller of the AID updates the did resolvers cache with updates(unsolicited reply messages). A trustworthy did resolver applies the BADA rules to any updates it receives. Optionally the did resolver may applies KRAM rules to any read requests to protect it from replay attacks.

In addition, a did doc can be a discovery mechanism for an ACDC caching server by including an index (label: said) of the SAIDs of the ACDCs held in the resolvers cache.

## Alignment of Data to Security Posture

One more thing. As a general security principle each block of information should have the same security posture for all the sub-blocks in the block. One should not attempt to secure a block that mixes security postures across is constituent sub-blocks. The reason is that the security of the block can be no stronger than the weakest security posture of any sub-block in the block. Mixing security postures forces all to have the lowest common denominator security. The only exception to this rule is if the block of information is purely informational for discovery purposes and where it is expected that each constituent sub-block is meant to be verified independently.

This means that any recipient of such a block with mixed security postures across its constituent sub-blocks must explode the block into sub-blocks and then independently verify the security of each sub-block. But this is only possible if the authentication factors for each sub-block are provided independently. Usually when information is provided in a block of sub-blocks, only one set of authentication factors are provided for the block as a whole and therefore there is no way to independently verify each subblock.

Unfortunately what happens in practice is that users are led into a false sense of security because they assume that they don’t have to explode and re-verify but merely may accept the lowest common denominator verification on the whole block. This creates a pernicious problem for down stream use of the data. A down stream use of a constituent sub-block doesn’t know that it was not independently verified to its higher level of security. This widens the attack surface to any point of down-stream usage. This is a

root cause of the most prevalent type of attack called a BOLA.

Given that for security one must always explode any block into its constituents by security level, then other than discovery, it makes no sense to combine the constituents in the first place. Each constituent block should have its own endpoint with that endpoint protected by the level of security appropriate to the block of information returned by that endpoint. The Did Doc then becomes simply a discovery mechanism consisting of endpoints, or a single endpoint that provides the other endpoints. Since OpenAPI (Swagger) already does this, did:docs are superfluous for did:web and did:webs because a did:webs already includes the domain name. All one has to define is .wellknown on that domain name to get the did:doc (swagger) list of api endpoints each with an atomic security posture, whatever it may be, and we have global interoperability with an exremely lightweight spec. One just has to define the schema using json schema for each endpoint and we have already global interoperability with the web world.

Thats what I would do if it was up to me. But I recognize that the existing did doc community may not be receptive to such a radically simplified approach even though it would not only be way more secure but also way more adoptable for anyone not already invested in did docs.

## Adoption Domains

I see two different adoption domains that did:webs may address. The first domain is the existing DID world for those that are already invested in DID tooling. The second much larger domain is the non DIDified web world for those that are familar with web tooling but

have not invested in DID tooling. Given that broad adoption is important and given that the web world is orders of magnitude greater than the DID world, we should be careful not to sacrifice general web world adoptability merely for DID world adoptability.

Since did:webs has as a dependency KERI under the hood, we should be careful about adding other adoption friction for features that are not needed for general web world adoption. Much of the did doc tooling fits in that category, especially given that a set of open API endpoints with JSON Schema that use KRAM, BADA-RUN, or KEL backing but are all discoverable with the did:webs URN to URL mapping and do not require any did doc tooling other than the did resolver mapping from URN to URL.

Ultimately if the only function left for did tooling is to map from URN to URL then all one needs is the URL in the first place which is equivalend to an OOBI. The whole web world can adopt by adopting the security policies of KERI but expressed using well known web tooling, namely, JSON, JSON Schema, and OpenAPI without ever needing to understand or adopt anything to do with DIDs. So I see did world adoption as a way of bringing the did world forward so that they can help build momentum for web world adoption which ultimately and eventually won't need or benefit from DIDs just AIDs.

## `@context`

Fundamentally, from a secure system design standpoint, its not that someone could use `@context` in a more or less secure way (which even using the best suggestions still has security vulnerabilities) but that when used as intended (which is in an

open world model) that there is no way to guarantee any level of security. Security is about reducing the attack surfaces that malicious users can exploit in ways that are difficult to anticipate. So `@context` is intended as an extensibility mechanism that enables injection of information not under the control of the issuer. Any resolver, or any other intermediary such as a relay can use `@context` to inject arbitrary information and a json-ld verifier will accept it. This means that we would have to add a new protocol that is not normative to json-ld, and that is not-normative to the did method itself to protect against such an injection. A normative did doc itself is not end-verifiable ( it is not signed), one has to trust the did resolver or any other type of intermediary that provides it. Therefore a malicious attacker could impersonate any intermediary and inject in such a way that the end-verifier can't detect such an injection because any intermediate '@context' injection is normatively correct vis-a-vis JSON-LD and the did doc spec.

I know that "best practice" would be to only run one's own resolver in one's own infrastructure so that its more protected than some resolver running in someone else's infrastructure. Moreover, given that we have a universal resolver that is meant to be run anywhere, a verifier can't guarantee that best practices have been followed, if for no other reason that we have made it easy to violate them by virtue of publishing implementations that do not use them. But even if one runs their own resolver in their own infrastructure it is still a much weaker security posture than one that is end-verifiable. One can have trusted intermediaries but such intermediaries are still subject to various attacks. Attacks on such intermediaries are well know exploits for DNS resolvers and BGP resolvers even when in one's own infrastructure. Running

one's own DID resolver is not very strong protection relatively speaking. Furthermore, `@context` injection because it is normatively correct makes such an attack undetectable downstream. To reiterate, because `@context` is a normatively correct mechanism that allows arbitrary injection of information, a successful attack is undetectable. Protocols that enable undetectable, but successful attacks are pathologically broken protocols from a security standpoint. The harm from such an attack is unbounded because the attack can persist indefinitely without detection and hence mitigation.

Another way of appreciating the problems of non-normative protection mechanisms is by viewing them as a form of ad-hoc security. Because one can't reliably reason about ad-hoc security, one can't build secure systems on top of such mechanisms. At the very least one must incur the cost of continual manual monitoring of all implementations of ad-hoc mechanisms. We should be wary of any such approaches.

KERI has two important security guarantees for KEL-backed data.

1. Any successful attack must begin with the successful compromise of one or more private keys.
2. Any successful attack is detectable as duplicity. This is true because it requires publication to the KEL in a way that is observable to the controller of the KEL or any other verifier of the KEL.

A normative mechanism that is correct but allows injection of arbitrary data violates both the first and second properties of KERI. The first because the information can be injected without compromising at least one private key of the issuer. The second,

because any such injection is undetectable downstream and may hide in plain sight despite any attempt to make it KEL backed.

The book, Practical Cryptography by Ferguson and Schneier, defines two fundamental security system design principles:

1. Complexity is the worst enemy of security.
2. Correctness must be a local property

`@context` violates principle 2 in about as salient a fashion as one can violate it.

The best way to protect against that type of attack is to convert a did doc into an ersatz verifiable credential that elides the `@context`. Then the ersatz VC can be securely attributed to its issuer not any intermediary such as a did resolver. Any end user can then verifity the ersatz VC without the `@context` and then the end-user can choose to inject an `@context` and thereby inject whatever information comes with it and then do expansion to an RDF graph. This makes the ersatz VC as did doc, end-verifiable with no possibility of malicious injection mid-stream and only the end-user can inject the `@context` after verification.

Thus the presence of `@context` of a did:doc provided by an intermediary would be a sign of exploit and must be discarded. Arbitrary injection then cannot happen in any intermediary but is guaranteed to only happen post verification by the end user. This gives us a strong guarantee than removes such an attack from the attack surface regardless of how any intermediary is implemented or how well the intermediary follows best practices for non-normatively protecting against exploit of `@context`. It's off the table. It's therefore simplified our security posture and we can reason about it correctly.

Now if we are NOT going to make a did doc an ersatz verifiable credential, then we can't trust any information it provides so we must verify every item of information it provides. This means the did doc itself is merely a discovery mechanism and all the information in it must eventually lead to the discovery of end-verifiable KEL backed data. Furthermore, as mentioned above, because `@context` injection is not KEL backable it has no place in a such a did doc e.g. one that is meant to be secured by KERI mechanisms. It would violate KERI's security guarantees.

## Recursive DID Resolvers

### A lesson learned from the KERI DHS work that I abandoned:

The idea was to provide a trusted discovery mechanism for KELs via a distributed hash table. What I realized is that making that discovery mechansim truly trustable was going to be hard work as we had to account for all the corner cases for exploit. And in the end the solution was that no matter what, anything discovered from the DHS had to be end-verifiable independent of the DHS. But when you step back and say wait I am trying to build a trustable discovery mechanism but I have to ensure that anything I discover is already end-verifiable then why does the discovery mechanism have to be trustable in the first place. The worst that can happen is that my discovery is DDoSed . And if the worst case is a DDos then I don’t need a trusted discovery mechanism (i.e. signed discovery data) but merely redundant discovery to mitigate DDoS. So I feel like anything we do that signs or anchors the DiD Doc itself is just re-making the same mistake. Did Doc don’t need to be trustable if they are just discovery mechanisms because

everything that you discover is itself end-verifiable.

### DID Method Updates are Meta-Data Only

For the sake of adoption, we want the did:webs method to output a did:doc and also a directory of files. But if they are not going to verify the contents of the did:doc then they aren't benefiting much from did:webs. But we can give then a library that takes that did:doc and says "verified" or not or elides unverifiable information from the did:doc. I am hoping that the library which they never see is OK. Its not pushing KERI up the stack except that the library has to be run locally on their end (edge) that is consuming the did:doc.

So they don't need to know that the so-called anchored files are actually ACDCs. All they see is the index of SAIDs and the payloads of the ACDCs after verification. Its a slight of hand. It looks like we are giving them a did doc that is trustable but under the hood the actual did doc is a discovery mechanism that must be verified before it can be used and verification means verifying the KEL backed data whatever form it takes and then rendering that data into a directory of files. The file directory could be indexed by the saids of the payloads of the ACDCs not the ACDCs themselves so they would never see an ACDC if they configure it so. Only the end-verification library every sees any KERI anywhere.

So their tooling sees exactly what they expect to see. But we aren't adding the complexity of a trustable discovery mechanism aka a trustable did doc but instead are using did docs as untrustable discovery mechanisms for end-verifiable data. All they see if the rendered output after end-verification.

The difference is really in what the issuer has to do to issue a did:doc. The did:doc the issuer issues is an untrustable discovery mechanism. This can be done in parts if its untrusable. Essentially updating vectors for discovery. If its not then the issuer has to assemble a correct did:doc and sign it or anchor it. But if all the issuer has to do is update the discovery vectors then that can be done piecemeal. Technically the did:resolver. is just capturing those vectors. A verifier must then follow the vectors end-verify them and then render the results. Where the verifier sits must always be local to the end. So one way to do this is that the Updates using RUN (converted from CRUD) of the did method is just updating did resolver metadata not pushing completed signed or anchored did docs. The resolver then end-verifies the metadata and renders a did doc. This is for a local resolver. A non-local resolver would act as a relay (like a recursive dns relay) and only be asked to relay its meta-data but not to render the actual did doc. An ACDC is a container for its payload. The ACDC as container therefore is a form of meta-data. A malicious resolver can delete ACDC meta-data but can't forge it.

An issuer of did:webs metadata does not have to worry about versioning did:docs or about stale did doc issuance because the metadata vectors are discovery vectors and the end verifier looks up the latest KEL infor automatically and dynamically at end-verifiable rendering time.

This allows a network of untrusted did resolvers to share metadata and only a local resolver (the one at the edge) does the final verification and then rendering of the did:doc. If a metadata vector is un-verifiable then it never gets into the final rendering of the did:doc. This is a scalable architecture and looks like recursive

dns. But unlike recursive dns, you can’t poison its cache because it never renders un-verifiable data. And the only place that end-verification is incurred is at the end. The issuer can then update any number of intermediate resolvers with the did metadata vectors without having to know about the end-resolver used by any user.

So it means an attacker doesn’t know which resolver to attack and the worst case attack is a ddos on some subset of intermediate resolvers which is mitigated by the issuer directly or indirectly updating multiple intermediate resolvers with its metadata.

Security is enhanced because the did method end-verifier and renderer behavior is guaranteed by a library that is normatively implemented. This means attacks must start with attacks on the code supply chain for the library not local access policies. Many OSes now provide TEEs for protections on run-time library services. The worst that an attack on any intermedate relay can do is DDoS. This approach lends itself to and may benefit from formal verification of the code library.

Issuers do not have the burden of versioning did:docs. Whenever the end verifier runs it produces the latest end-verifiable version possible and renders it. Issuers just update meta-data on the method API of any resolver. Resolvers can share the meta-data recursively. This enables operation at scale.