KeeeX Metadata Statement Specification

Author : Gabriel Risterucci and the KeeeX Team

Abstract

This document describes the KeeeX langague statements that are injected within files before keeexing. The core idea is to non destructively inject in a file its hash, that takes into account the expected signing parties and the corresponding signatures for this hash. The keeexing metadata are injected in files at positions that do not interfere with the expected behavior, which is most often possible. This results in files that when altered at any position from byte 0 to end of file are detected as corrupt. This also addresses a number of extra features, like chaining files to references and versions, indexing files with their embedded hash, making a file display it's own hash and more.

Version 2.0

This is the version 2 of the KeeeX metadata statement language. This V2 allows for a number of new features, including parallel additive multi hashes, incremental or multi level keeexing, delegated signatures. It also introduces a stricter version of protected statements : the reserved statements.

Statements syntax

General structure

A KeeeX statement is located between two keywords: `keeex` and `xeeek`. The content of a statement follows this syntax:

```
keeex <type> "<value1>"[, "<value2>"][, {extra}] xeeek
```

Values can optionally be found between delimiters (see below). `value2` is optional, depending on the statement. Extra is a key-value structure whose format is described in a following section. It is enclosed in brackets.

Text encoding

Statements must be ASCII-only text. Non ASCII characters (code point over 127) are to be encoded using URL-like escape sequences. The only exception is the `%` character (37) encoded as `%25` to make the transformation fully reversable.

In addition to this encoding, all value strings (`value1` and `value2`) are escaped as JSON strings, which means that doublequotes are escaped, and the escape symbol (`\`) is escaped too.

Some injection methods cause specific modifications to the injected text. If decoding the written metadata requires more conversion than the default, the `kx.decode` extra will be present in the main `self` statement. If present, the requested decoding is done before returning effective data to the user.

Statements location

Some statements can have specific positioning in a file, depending on the format and requirements. The three options are as follow:

- regular statements: placed in an appropriate place to not alter the original data payload
- important statement: the main self statement can (if possible) be placed in a position that might be visible in the underlying payload, in order to preserve it across modifications. This is required to distinguish a modified file from a non-keeexed file
- in free space: some `reserved` statements might want to reserve space. There are two options for these statements: either the file format does not allow flexible length statements, in which case space is reserved when keeexing, or the file format allows for flexible length statements.

Statements value's delimiters

The valid delimiters for statement values are double quote (`"`) and HTML-style escaped double quote (`&quot;`). When one style is used on a set of statements, it should remain consistent across all statements.

The recommended form is to use double quote when possible.

Extra property

The extra property that can be optionally found in a statement allows for defining a key-value object. It is loosely based on JSON with the following differences:

- There are no double quotes
- Keys can only be ASCII without space
- Values can only be number, boolean or string with no double quotes (comma can't be used in values)
- Boolean values are treated differently: if a key is present without value, it is set to true.

Extra properties whose key starts with `kx.` are reserved and can only be set through dedicated program parameters when keeexing a file.

Statement types

This section lists all the supported types of statements, along with their meaning and usage. The only mandatory statement is a `self` statement containing the file's main hash identifier, called IDX in the sequel.

The reason for using the special name IDX for KeeeX 'hashes', is that such a hash is not computed from the entire original file but instead it is a multihash computed from the the original file injected with arbitrary KeeeX statements, but excluding some parts of the file where hashes, signatures and otherwise reserved data. Furthermore the file's IDX is itself injected in the file using encodings that facilitate human readability, content based indexing, human exploitation as a kind of DOI (Digital Object Identifier) and inter file reference and linking.

Although `self` is the only mandatory statement, it is rare to find a file with only this statement, notably because files become fully verifiable solely when their origin or authorship is assessed by non forgeable digital signatures.

Unless stated otherwise, statements are kept in full when computing a file's identifier. This means that for instance the public key of all expected signing parties are involved when computing the IDX, so that a file can never be re-attributed.

Shared extra properties

These properties can be set in the `extra` of any statement.

`kx.height` extra (numeric)

If missing, it defaults to 1. It indicates at which keeexing "height" the statement is located. When keeexing a file, all new statements must be above the largest height already present in the file (if the keeexed file does not contain pre-existing keeex statements, the height parameter should be absent). When verifying a file, only the statements of the highest height are to be used; lower level statements being discarded silently.

In some cases statements of different heights can be linked together, for more details see Continued KeeeX statement.

`self` (file identifier)

This statement contains file identifiers.

The identifier is set in `value1`, without double quotes. There is no `value2`.

There are two kind of identifiers:

- The main identifier is the file's IDX. Its extra has the boolean property `kx.main`. There must be exactly one main `self` per height.
- Extra identifiers, computed using different hash algorithms and encodings.

To be valid, all identifiers for a given file must match the actual file content as computed with respect to the 'reserved' or 'protected' statements.

IDX computation

When computing a file identifier, the content of `value1` is fully skipped in the whole file, including in the statement itself.

Placeholders

A placeholder string is used when creating statements. Such placeholders can be found in the file, and when the file is properly seeded they are replaced with the appropriate value throughout the file.

The placeholders are computed by applying each required combination of algorithms and encoding to the following string:

```
KeeeX Placeholder for self
```

As a special case, the historical placeholder for the sha256 algorithm and the bubble-babble encoding is kept as:

```
Zuzi-ntinen-cudi-bfudab-poru-hsalup-leti-nsehoh-polu-cbazec-dahy-rgogun-roke-hvusin-fato-ztymabf-exar
```

with leading 'Z' in lowercase (the above replacement is intended to prevent the string to be substitued by actual IDX when keeexing this file)

`kx.statementVersion` extra (mandatory numeric)

A numeric value. If missing, it defaults to 1 (the initial version where this property didn't exist). Indicates which version of this specification was used when writing the statements. It is only present in the `extra` of a main `self` statement, and applies to all statements of the same `height`.

`kx.main` extra (boolean)

Indicate that this is the main identifier. All statement blocks must have one main identifier.

`kx.id` extra (numeric)

Identify this identifier for future reference in signatures or external processes. It defaults to 0, and should be skipped in the main `self`. The first non-main `self` has a `kx.id` value of 1.

`kx.alg` extra (string)

Specify the algorithm used to compute the identifier. Defaults to the value specified in the computing IDX section.

Valid hash algorithms are:

- "sha224"
- "sha256"
- "sha512"
- any combination of valid hash algorithms separated with a `+` to use multihash (described briefly in the computing IDX section)

`kx.enc` extra (string)

Specify the encoding used to present the identifier. Defaults to the value specified in the computing IDX section.

Valid encodings are :

- "b58": base58
- "bubble": bubble-babble

`kx.decode` extra (string) {#kxdecode}

The description of an injection method to indicate how data must be decoded to retrieve the correct initial value. This property is named `kx.decode` and defaults to `raw` if not specified. The possible values are:

- `raw` : no special modifications
- `xml` : escapes the characters using basic html entities encoding (at least encode `>` and `&` with their equivalent `>` and `&`)

`kx.stopAt` extra (numeric)

A numeric value, with optional space padding before the number. If this property is present, then the metadata block is only computed up to the given byte of the file (excluded). Such a block is not considered valid if there is no final statement block. The range of bytes used in the computation must include all the statements of the current height. See the details in the section `Continued keeex statement` below.

`name` (file name)

Used to indicate various "readable" names associated to the file. This statement is optional, but can be used to display names in automated processes. The use includes providing alternate names for several locales.

`kx.main` extra (boolean)

Can be used to indicate a default name to display to users.

`prop` (file property)

This allow storage of arbitrary properties in the file. A property is a key-value mapping. The key is set in `value1` and the value in `value2`. Properties whose name starts with `kx.` are reserved and can't be used for user custom properties.

For a list of known key names, see `special_props.md` .

`kx.base64` extra (boolean)

If this property is present, the content of `value2` is expected to be encoded in base64 and must be decoded for the user to retrieve the original data.

`ref` (IDR reference)

Contains a reference to another file (using an IDR). The IDR is set as `value1` and ways to distinguish between multiple reference can be set in the extra.

`kx.previous` extra (boolean)

The special prop named `kx.previous` is used to indicate a reference to a previous version of this file.

`kx.continuePrevious` extra (boolean)

If this property is present on a ref, it must be present on only one ref. The value of the ref must be the main IDX of the previous block (one height lower).

See the details in the section `Continued keeex statement`.

`signature` (digital signature)

Represents a single signature of a file.

Digital signatures are usually made using a private/public keypair. In keeex statements we store three informations:

- an "address", related to the public key
- the "signature" itself
- the type of signature

The notion of address comes from using bitcoin-message signature, and initially represented what is actually called an "address" in this context. It is now a broader term to identify the public part of a signature key, in a way that makes it possible to verify the signature.

The signature value is comprised of two fields: a fixed-size ISO-8601 datetime (down to the millisecond), and the signature itself. They are separated with a `|`. When placing a placeholder, the datetime is set as `-----------------------`. To make this date fixed-size, it is *always* written as UTC with the `Z` timezone indicator.

Example: `2021-08-20T09:18:33.379Z`.

The type of signature is stored as a string value in the extra with the key `kx.sigType`.

IDX computation

When a signature statement is cleaned for computing identifiers, the signature (the `value2`) is removed, including the delimiters (double quote or other).

Signature subject

By default, a signature is computed on the `main` IDX of the file. Optionally, an `extra` property named `kx.id` can be used. In that case, the signature is computed on the statement that matches this `kx.id`.

The whole signature subject is the content described in the previous paragraph prefixed with the timestamp string described above.

Signature placeholder

It is possible to keeex a file without computing all signatures at the same time; in this case, a placeholder of the appropriate size is set in the `value2` field. The placeholder should be clearly indicated as such and reserve enough space. As described above, it is preceded by a placeholder timestamp.

To compute the signature placeholder itself, once the expected length `L` is known we create a string of that length that ends with the address and is padded to the left by `placeholder`. If the signature length is shorter than the address field, the word `placeholder` is kept in full at least once.

`bitcoin` signature

These signatures are based on bitcoin-message signatures. This is the format typically used to sign classic bitcoin transactions, and includes the mechanisms to convert a public key to an "address" (which is used as the `address` field) and the conversion of the signature output to a base64 string.

All implementations are compatible with this kind of signatures.

`x509` signature

These signatures use standard protocols to create digital signatures. The public key is backed by a full certificate. There is no check of the certificate's emitter when performing a signature; this is left to the verification process.

For these signatures, the `address` is the full X509 certificate in base64 with the BEGIN/END delimiters (PEM). The `signature` itself is also stored in base64.

Complex addresses

Some signature can have a special address string which requires further interpretation before verification/signature. These cases are described in the sections below, according to the following:

- If a `signature` statement `extra` contains the `kx.indirect` boolean property, then the `address` field is used differently. See Indirect signature
- If a `signature` statement `extra` contains the `kx.delegate` boolean property, then the `address` field is checked differently. See Signature delegation

- If both `kx.indirect` and `kx.delegate` are present, `kx.indirect` is resolved first.

`kx.indirect` extra (boolean) {#kxindirect}

Instead of storing the actual signature address in the statement, an indirect reference is used. This reference relies on a known third-party to "link" this reference to an actual address.

The reference is a JSON string representing the following object:

```
{
  "ref": "<text reference>",
  "trustedPeer": "<trusted third-party address>"
}
```

In the presence of this kind of address, the actual address used is resolved by looking for a `reserved` statement with the `kx.indirectSignature` extra and whose first value matches the `<text reference>`. Once such a statement is found, the second value is read and interpreted as the concatenation of the effective address to use and a signature by the trusted third party separated by `|`.

The actual content of a complete `value2` entry for this `reserved` statement looks like:

```
<address to use>|<signature of the below block by third party>
```

Effectively, when signing, this statement is updated with the appropriate informations. The signature is computed using the trusted third-party private key, and signs the following: `<IDX>|<text reference>|<effective address>` (the three fields are separated by `|`).

This process has the following properties:

- The text reference and the trusted peer at the time of writing the metadata are known, and are part of the IDX
- The effective address used is confirmed by the trusted peer on a file per file basis and can't be reused spuriously
- Effective signatures can use an address that is not known at the time of keeexing

`kx.delegate` extra (boolean) {#kxdelegate}

A signature can be done with a different address than the one in reference. (this feature is only available for bitcoin type signatures)

If present, two cases arise: either we use the initial address, in which case nothing special is done for either signing or verifying, or we use a delegation. In the second case, the effective address is resolved by:

- searching for a `reserved` statement with the extra prop `kx.delegation` and the initial address in `value1`
- processing `value2` as a sequence of delegation information in the format specified below separated by a comma. Each object (if valid) replaces the current address by another address, ultimately providing the address to use for the signature

Each delegation information object is a string that represents the concatenation of the following elements separated by a `|` character:

- original address in its text representation as usually used in a `signature` statement
- destination address
- signature role, if applicable (if not, an empty string is used)
- a date indicating the "not before" validity date of the delegation, encoded in ISO8601 (only the date portion) (inclusive)
- a date indicating the "not after" validity date of the delegation, same as the previous field, also inclusive
- the signature of all the previous fields, without any concatenation characters

These data all have a known maximum size and allow for allocation of space in advance for multiple level of delegation if required.

`kx.id` extra (numeric)

When a signature is not computed on the main IDX, this property indicates on which statement the signature is computed. It can reference either a `self` or a `reserved` statement. For `reserved` statements, the signature is applied to `value2`.

`kx.role` extra (string)

A signature can have a role indication, which is user-provided and of no significance to the usual signature process. Such a role is indicated in the `kx.role` extra as an ASCII-string whose length is limited to 50 bytes (usual string encoding applies). For technical reasons, role strings cannot contain the following characters: `,` (comma), `}` (closing curly bracket), and `|` (vertical bar).

`kx.ts` extra (boolean)

A signature can have the `kx.ts` extra property set; in this case, it is timestamped when created. To do this, we use KeeeX's API to perform timestamping of the hash of the following elements concatenated as a string:

- signed identifier (usually the main IDX)
- signature timestamp inserted into the signature placeholder
- signature address
- signature

This way, it is possible to get timestamp information when in possession of the file by recreating this hash.

`kx.continueAddress` extra (boolean)

When using a continuation block (with `kx.stopAt` in the main self) a signature can be marked as valid for the continuation block with this property. In that case the address used for that signature can be used in a continuation block. Multiple signatures can be marked this way to allow for multiple continuation addresses.

See the details in the section `Continued keeex statement` below.

`protected` (reserved space)

A protected statement is kind of a free statement with no specific constraints on its content.

IDX computation

When a file is cleaned for computing identifiers, the whole statement (starting from the beginning of the `keeex` keyword up to the end of the `xeeek` keyword) is removed.

`license` (user license)

The `license` statement is made of two values that indicate the license's IDX in `value1` and the signature from KeeeX's private key in `value2`.

For "newer" license files, the signature is the exact same as the `value2` value in the license file so it would also contain a timestamp.

`reserved` (reserved space with restriction)

These statements are similar to `protected` statements, but only the content of `value2` is cleaned when computing a hash, meaning that all data surrounding this value is part of the file

identifier. It is highly advisable to reserve space when using such a statement, as changing a statement size can lead to broken files.

IDX computation

When computing the file identifiers, the content of `value2` is removed, including the delimiters (in a way similar to that for handling `signature` statements).

`kx.id` extra (numeric)

Add a property named `kx.id` in the extra of each `reserved` statement to identify them formally. This sequence follows the identifiers set in the `self` and is used likewise so that a signature can target a given data. In the case of `reserved` statements, the signature applies to the following:

```
${mainIDX}|${value2}
```

The full length of `value2` is used in the signature.

`kx.htmlComment` extra (boolean)

When present, the content of `value2` is surrounded by `-->` and `<!--` and these are removed before retrieving the actual data. It is useful to "insert" dynamic data in HTML files.

`kx.transaction` extra (boolean)

When present, `value1` identifies a blockchain (or blockchain-like structure) known by KeeeX verifiers, and `value2` indicates a reference on said structure.

`kx.base64` extra (boolean)

If this property is present, the content of `value2` is expected to be encoded in base64 and must be decoded for the user to retrieve the original data.

`kx.flexible` extra (boolean)

If a `reserved` statement has this extra property, it is allowed to change size depending on the effective content. The absence of this property indicates that the statement can't change size, and if the actual data is shorter than the reserved space padding must be used to keep it at the same length.

`kx.indirectSignature` extra (boolean)

Indicates that this statement contains informations pertaining to an indirect signature. For more informations see Indirect signature.

`kx.delegation` **extra (boolean)**

Indicate that this statement contains informations pertaining to a delegated signature. For more informations see Signature delegation.

Computing identifiers (IDX, self) {#computeIDX}

When computing the identifier of a file, we process all data except for specific parts using hashing algorithms. Files generated using the version 1 of the keeex statements use sha256 encoded in bubble-babble. Version 2 defaults to using sha512+sha256 encoded in bubble-babble, and removes the use of a salt at the end of the file.

The input for the hash algorithms is the whole file except for:

- cleaned part of some statements, as described above for signatures, protected and reserved statements, are skipped
- the current file identifiers (either the placeholders, or the effective identifiers) are skipped

Computing a "multihash" result is fully described in the file named "multihash.md", but boils down to this:

- for each hash algorithm, process the (cleaned) file as usual
- in the order specified append the end of the first hash as input data for the second hash and so on
- the output of the last hash is the result

All incremental hashes thus depend upon the entire file contents (modulo cleaning) plus the hashes of the previous steps.

Special statements groups

Continued keeex statement {#continued_statements}

In some cases it might be required to keeex a file once with most informations, then allow some extra data to be added. Ideally we want to be able to "seal" such a statement block.

To do so, we can mark a metadata block with the `kx.stopAt` property in the main `self` `extra`. This value indicates up to which byte of the file (excluded) the IDX must be computed against.

This must include the keeex statements themselves. A block that have the `kx.stopAt` property is considered valid *only* if the next continuation block (identified by a `ref`) is itself valid, recursively.

Following statement blocks are at higher height (using the regular height mechanism) and must contain a `ref` statement with the `kx.continuePrevious` extra property. This `ref` must be the IDX of the previous block. Such a statement block is valid only if at least one of its signature's address matches one of the addresses used in the previous level, indicated by the `kx.continueAddress` extra property.

A followup continuation block is only valid if the previous statement about signature is true and either it does not contain a `kx.stopAt` property in its main `self` extra, or if it is also followed by another continuation block.