

CBOR Manifest Serialisation

Open issues

- Should the primary object be an array or a map?
- Should the graph described in draft-moran-suit-manifest-02 be replaced with trees?
 - How are they overridden?
 - How are they represented?
 - Are there any use cases that break?
- Should one digest be used for the whole manifest, or is there a good reason to support more than one?
- Should more sections be severable?
 - Should it be possible to encrypt severable sections?
- COSE has no algorithm identifiers for digests, so they cannot be reused
 - May need another SUI draft for COSE

Open Issues (Cont'd)

- Should encryption of the manifest be addressed explicitly, or should that be handled one level higher?
- Encoding of Processing Step
- Encoding of Directive
- Encoding of Conditions
- Extension encoding
- IANA implications for use of enums throughout the manifest

Primary structure array vs map

- Most fields are mandatory
 - Version
 - digestInfo*
 - Nonce
 - Sequence
 - Preconditions
 - Resources
 - Targets
- Those that aren't used cost 1 byte (nil or 0-length object)
- Out of 12 fields, 7 are mandatory

Tree-based process description proposal

- Things that are installed (assets) get their own section
 - Digest
 - Size
 - Component identifier
- Description of how to obtain an asset:
 - Component identifier
 - List of processing steps
 - Resources are a processing step with URIs or a ComponentIdentifier instead of inputs

Tree-based process description (cont'd)

```
Asset = [  
    component: ComponentIdentifier, ; where to store the asset  
    encoding:      bstr / nil, ; the format of the asset  
    parameters:    bstr / nil, ; any specialized arguments for installing the asset  
    size:          uint / nil, ; size of the resource  
    digest:        Digest      ; digest of the processed resource  
]  
ComponentIdentifier = [* bstr ]  
Digest              = [  
    digestAlgorithm : DigestIdentifiers / nint, ; Digest identifier  
    rawDigest :      bstr                       ; Raw bytes of the digest  
    ? digestParameters : bstr                   ; optional extra arguments to the algorithm  
]
```

Tree-based process description (cont'd)

```
AssetInstallationInfo = [  
    component :      ComponentIdentifier  
    processors :     [ * Processor ],  
    ? allowOverride : bool,  
    ? extensions :   { * int => bstr }  
]  
  
Processor          = [  
    type:           &( resource: 1, decrypt: 2, decompress: 3,  
                        undiff: 4, relocate: 5, unrelocate: 6) / nint,  
    inputs:         UriList / ComponentIdentifier / {int => int},  
    parameters:    bstr      ; Note: this needs to have more detail instead of bstr  
]
```

Description of installation process in manifest

- Why not leave container information in the payload(s)?
 - Reject early if unsupported
 - Important for low-bandwidth
- Why not enums for aggregate formats?
 - Lots of specialized parameters.
 - Many possible configurations.
 - Registration space of accepted enums becomes large

Description of installation process in manifest

- Example:
 - Raw binary payload: no arguments
 - Encrypted binary payload:
 - key identifier
 - algorithm identifier
 - Encrypted, compressed binary payload:
 - key identifier
 - encryption algorithm identifier
 - compression algorithm identifier
 - Encrypted, compressed, delta payload
 - key identifier
 - encryption algorithm identifier
 - compression algorithm identifier
 - precursor digest
 - precursor component ID, storage location
 - delta algorithm identifier

Description of installation process in manifest

- Each enum would need its own parameter structure
- Easy to miss a reasonable combination of supported steps
- Describe each step instead
 - Each step can have a defined structure
 - All steps can be represented in the same way
 - How is flow described?
- Flow of data between steps is a tree, not a linear sequence
 - Delta makes it clear that flow is at least a tree
 - A resource shared between two steps makes it clear that flow is a graph
 - Graphs make constrained processing hard
 - Use multiple trees instead of a graph

Description of installation process in manifest

- For each asset, a tree defines the installation process
- To reduce nesting depth in the parser, the tree is encoded as a list, where
 - output identifiers are the index of the processor in the processor list
 - inputs are a map of indices into the processor list.
- Resources are encoded as a processor with no inputs
- Assets designate a single input node
- Output nodes can be marked as non-overridable
- Dependent manifests can override any installation tree not marked as non-overridable

Examples:

- Raw Binary payload
 - Installation Information:
 - Component Identifier: [bstr(0)]
 - Resource:
 - Parameters: List of URIs
 - Asset Information
 - Component Identifier: [bstr(0)]
 - Encoding: raw-binary

Examples:

- Delta payload
 - Installation Information:
 - Component Identifier: [bstr(1)]
 - Processors:
 - Delta: {1 => 1, 2 => 2}
 - Resource: URIs
 - Resource: [bstr(1)]
 - Asset Information
 - Component Identifier: [bstr(1)]
 - Encoding: raw-binary
 - inode: bstr(2)

Current changes

Highlights of changes

- Primary structure is array (not map)
- Text is severable
- Multiple payloads allowed
- Resources (before installation) separated from assets (after installation)
- Description of installation process in the manifest
- Conditions divided into preconditions and postconditions
- Added component identifier (replaces storage identifier)
- Timestamp removed, added sequence number

WIP status

- This serialization is still in development
- There is ongoing discussion on the mailing list
- See Open Issues for more information

Severable Text

- Text is for humans
- Text is not used by devices to make decisions
- Devices don't need to receive text
- Text is still needed in management systems

Severable Text (cont'd)

- Text lives outside the authenticated container

```
AuthenticatedManifest = [
```

```
  authenticatedManifest: COSE_Mac / COSE_Sign,
```

```
  text: bstr .cbor TextMap
```

```
]
```

```
TextKeys = &{
```

```
  uninitialised: 0
```

```
  manifestDescription: 1
```

```
  payloadDescription: 2
```

```
  vendorName: 3
```

```
  modelName: 4
```

```
  payloadVersion: 5
```

```
)/ nint
```

```
TextMap = { * TextKeys => tstr }
```

Severable Text (cont'd)

- Inside the authenticated container, text is authenticated with a digest

Multiple payloads

- Payloads had three components:
 - A resource identifier
 - Installation instructions (cryptographic info)
 - An asset description
- Required extension for advanced uses (e.g. Delta)
- Aliases, Dependencies, Payloads are now all resource references
- Assets are now separate from their resource identifier
- Installation instructions are now separate (examples below)

Resources separated from assets

- All resource references are effectively the same
- No need to distinguish between alias, dependency, payload
- Resources define a local or remote input
 - URI
 - Digest
- Assets define an installed image:
 - Size
 - Digest
 - Location

Conditions divided into preconditions and postconditions

- Some conditions need to be checked before installation
 - Identifiers, time, precursors, custom
- Some conditions need to be checked after installation
 - Content not identified by an asset digest, custom
- Two choices:
 - Two lists
 - makes it easier for devices to know what to do in each step of the process
 - typically costs 1 byte
 - Duplicate condition identifiers
 - typically smaller serialization
 - duplication of identifiers may increase integer encoding size
 - more registration and maintenance with duplicate identifiers

Component Identifier

- Storage Identifier may not be adequate for all use cases
- Devices can be aggregates of one or more processors with two or more different storage systems
- Component Identifier allows designating the storage system
- Storage systems and hardware components can be nested
- Component Identifier needs to be a list to handle this.
- Storage Identifier can be merged into this list as the last element.