Rocky Enterprise Linux 9.2 Manual Pages on command 'containers-signature.5'

$ man containers-signature.5

container-signature(5)   format   container-signature(5)

Miloslav Trma? March 2017

NAME

    container-signature - Container signature format

DESCRIPTION

    This document describes the format of container signatures, as imple?

    mented by the github.com/containers/image/signature package.

    Most users should be able to consume these signatures by using the

    github.com/containers/image/signature package (preferably through the

    higher-level signature.PolicyContext interface) without having to care

    about the details of the format described below. This documentation

    exists primarily for maintainers of the package and to allow indepen?

    dent reimplementations.

High-level overview

    The signature provides an end-to-end authenticated claim that a con?

    tainer image has been approved by a specific party (e.g. the creator of

    the image as their work, an automated build system as a result of an

    automated build, a company IT department approving the image for pro?

duction) under a specified identity (e.g. an OS base image / specific application, with a specific version).

A container signature consists of a cryptographic signature which iden?tifies and authenticates who signed the image, and carries as a signed payload a JSON document. The JSON document identifies the image being signed, claims a specific identity of the image and if applicable, con?tains other information about the image.

The signatures do not modify the container image (the layers, configu?ration, manifest, ?); e.g. their presence does not change the manifest digest used to identify the image in docker/distribution servers; rather, the signatures are associated with an immutable image. An im?age can have any number of signatures so signature distribution systems SHOULD support associating more than one signature with an image.

The cryptographic signature

As distributed, the container signature is a blob which contains a cryptographic signature in an industry-standard format, carrying a signed JSON payload (i.e. the blob contains both the JSON document and a signature of the JSON document; it is not a ?detached signature? with independent blobs containing the JSON document and a cryptographic sig?nature).

Currently the only defined cryptographic signature format is an OpenPGP signature (RFC 4880), but others may be added in the future. (The blob does not contain metadata identifying the cryptographic signature for?mat. It is expected that most formats are sufficiently self-describing that this is not necessary and the configured expected public key pro?vides another indication of the expected cryptographic signature for?mat. Such metadata may be added in the future for newly added crypto?graphic signature formats, if necessary.)

Consumers of container signatures SHOULD verify the cryptographic sig?nature against one or more trusted public keys (e.g. defined in a pol?icy.json signature verification policy file ?containers-pol?icy.json.5.md?) before parsing or processing the JSON payload in any way, in particular they SHOULD stop processing the container signature

if the cryptographic signature verification fails, without even start?

ing to process the JSON payload.

(Consumers MAY extract identification of the signing key and other

metadata from the cryptographic signature, and the JSON payload, with?

out verifying the signature, if the purpose is to allow managing the

signature blobs, e.g. to list the authors and image identities of sig?

natures associated with a single container image; if so, they SHOULD

design the output of such processing to minimize the risk of users con?

sidering the output trusted or in any way usable for making policy de?

cisions about the image.)

OpenPGP signature verification

When verifying a cryptographic signature in the OpenPGP format, the

consumer MUST verify at least the following aspects of the signature

(like the github.com/containers/image/signature package does):

? The blob MUST be a ?Signed Message? as defined RFC 4880 sec?

tion 11.3. (e.g. it MUST NOT be an unsigned ?Literal Mes?

sage?, or any other non-signature format).

? The signature MUST have been made by an expected key trusted

for the purpose (and the specific container image).

? The signature MUST be correctly formed and pass the crypto?

graphic validation.

? The signature MUST correctly authenticate the included JSON

payload (in particular, the parsing of the JSON payload MUST

NOT start before the complete payload has been cryptographi?

cally authenticated).

? The signature MUST NOT be expired.

The consumer SHOULD have tests for its verification code which verify

that signatures failing any of the above are rejected.

JSON processing and forward compatibility

The payload of the cryptographic signature is a JSON document (RFC

7159). Consumers SHOULD parse it very strictly, refusing any signature

which violates the expected format (e.g. missing members, incorrect

member types) or can be interpreted ambiguously (e.g. a duplicated mem?

ber in a JSON object).

Any violations of the JSON format or of other requirements in this doc?
ument MAY be accepted if the JSON document can be recognized to have
been created by a known-incorrect implementation (see optional.creator
?#optionalcreator? below) and if the semantics of the invalid document,
as created by such an implementation, is clear.

The top-level value of the JSON document MUST be a JSON object with ex?
actly two members, critical and optional, each a JSON object.

The critical object MUST contain a type member identifying the document
as a container signature (as defined below ?#criticaltype?) and signa?
ture consumers MUST reject signatures which do not have this member or
in which this member does not have the expected value.

To ensure forward compatibility (allowing older signature consumers to
correctly accept or reject signatures created at a later date, with
possible extensions to this format), consumers MUST reject the signa?
ture if the critical object, or any of its subobjects, contain any mem?
ber or data value which is unrecognized, unsupported, invalid, or in
any other way unexpected. At a minimum, this includes unrecognized
members in a JSON object, or incorrect types of expected members.

For the same reason, consumers SHOULD accept any members with unrecog?
nized names in the optional object, and MAY accept signatures where the
object member is recognized but unsupported, or the value of the member
is unsupported. Consumers still SHOULD reject signatures where a mem?
ber of an optional object is supported but the value is recognized as
invalid.

JSON data format

An example of the full format follows, with detailed description below.
To reiterate, consumers of the signature SHOULD perform successful
cryptographic verification, and MUST reject unexpected data in the
critical object, or in the top-level object, as described above.

```
{
    "critical": {
        "type": "atomic container signature",
```

```
            "image": {

                                                        "docker-manifest-digest":
"sha256:817a12c32a39bbe394944ba49de563e085f1d3c5266eb8e9723256bc4448680e"

            },

            "identity": {

                "docker-reference": "docker.io/library/busybox:latest"

            }

        },

        "optional": {

            "creator": "some software package v1.0.1-35",

            "timestamp": 1483228800,

        }

    }
```

critical

> This MUST be a JSON object which contains data critical to correctly
> evaluating the validity of a signature.
>
> Consumers MUST reject any signature where the critical object contains
> any unrecognized, unsupported, invalid or in any other way unexpected
> member or data.

critical.type

> This MUST be a string with a string value exactly equal to atomic con?
> tainer signature (three words, including the spaces).
>
> Signature consumers MUST reject signatures which do not have this mem?
> ber or this member does not have exactly the expected value.
>
> (The consumers MAY support signatures with a different value of the
> type member, if any is defined in the future; if so, the rest of the
> JSON document is interpreted according to rules defining that value of
> critical.type, not by this document.)

critical.image

> This MUST be a JSON object which identifies the container image this
> signature applies to.
>
> Consumers MUST reject any signature where the critical.image object
> contains any unrecognized, unsupported, invalid or in any other way un?

expected member or data.

(Currently only the docker-manifest-digest way of identifying a con?
tainer image is defined; alternatives to this may be defined in the fu?
ture, but existing consumers are required to reject signatures which
use formats they do not support.)

critical.image.docker-manifest-digest

This MUST be a JSON string, in the github.com/opencontainers/go-di?
gest.Digest string format.

The value of this member MUST match the manifest of the signed con?
tainer image, as implemented in the docker/distribution manifest ad?
dressing system.

The consumer of the signature SHOULD verify the manifest digest against
a fully verified signature before processing the contents of the image
manifest in any other way (e.g. parsing the manifest further or down?
loading layers of the image).

Implementation notes: * A single container image manifest may have sev?
eral valid manifest digest values, using different algorithms. * For
?signed? docker/distribution schema 1 ?https://github.com/docker/dis?
tribution/blob/master/docs/spec/manifest-v2-1.md? manifests, the mani?
fest digest applies to the payload of the JSON web signature, not to
the raw manifest blob.

critical.identity

This MUST be a JSON object which identifies the claimed identity of the
image (usually the purpose of the image, or the application, along with
a version information), as asserted by the author of the signature.

Consumers MUST reject any signature where the critical.identity object
contains any unrecognized, unsupported, invalid or in any other way un?
expected member or data.

(Currently only the docker-reference way of claiming an image iden?
tity/purpose is defined; alternatives to this may be defined in the fu?
ture, but existing consumers are required to reject signatures which
use formats they do not support.)

critical.identity.docker-reference

This MUST be a JSON string, in the github.com/docker/distribution/ref?

erence string format, and using the same normalization semantics (where

e.g. busybox:latest is equivalent to docker.io/library/busybox:latest).

If the normalization semantics allows multiple string representations

of the claimed identity with equivalent meaning, the critical.iden?

tity.docker-reference member SHOULD use the fully explicit form (in?

cluding the full host name and namespaces).

The value of this member MUST match the image identity/purpose expected

by the consumer of the image signature and the image (again, accounting

for the docker/distribution/reference normalization semantics).

In the most common case, this means that the critical.identity.docker-

reference value must be equal to the docker/distribution reference used

to refer to or download the image. However, depending on the specific

application, users or system administrators may accept less specific

matches (e.g. ignoring the tag value in the signature when pulling the

:latest tag or when referencing an image by digest), or they may re?

quire critical.identity.docker-reference values with a completely dif?

ferent namespace to the reference used to refer to/download the image

(e.g. requiring a critical.identity.docker-reference value which iden?

tifies the image as coming from a supplier when fetching it from a com?

pany-internal mirror of approved images). The software performing this

verification SHOULD allow the users to define such a policy using the

policy.json signature verification policy file format ?containers-pol?

icy.json.5.md?.

The critical.identity.docker-reference value SHOULD contain either a

tag or digest; in most cases, it SHOULD use a tag rather than a digest.

(See also the default matchRepoDigestOrExact matching semantics in pol?

icy.json ?containers-policy.json.5.md#signedby?.)

optional

This MUST be a JSON object.

Consumers SHOULD accept any members with unrecognized names in the op?

tional object, and MAY accept a signature where the object member is

recognized but unsupported, or the value of the member is valid but un?

supported.   Consumers still SHOULD reject any signature where a member of an optional object is supported but the value is recognized as in? valid.

optional.creator

If present, this MUST be a JSON string, identifying the name and ver? sion of the software which has created the signature (identifying the low-level software implementation; not the top-level caller).

The contents of this string is not defined in detail; however each im? plementation creating container signatures:

> ? SHOULD define the contents to unambiguously define the soft?
> ware in practice (e.g. it SHOULD contain the name of the soft?
> ware, not only the version number)

> ? SHOULD use a build and versioning process which ensures that
> the contents of this string (e.g. an included version number)
> changes whenever the format or semantics of the generated sig?
> nature changes in any way; it SHOULD not be possible for two
> implementations which use a different format or semantics to
> have the same optional.creator value

> ? SHOULD use a format which is reasonably easy to parse in soft?
> ware (perhaps using a regexp), and which makes it easy enough
> to recognize a range of versions of a specific implementation
> (e.g. the version of the implementation SHOULD NOT be only a
> git hash, because they don?t have an easily defined ordering;
> the string should contain a version number, or at least a date
> of the commit).

Consumers of container signatures MAY recognize specific values or sets of values of optional.creator (perhaps augmented with optional.time? stamp), and MAY change their processing of the signature based on these values (usually to accommodate violations of this specification in past versions of the signing software which cannot be fixed retroactively), as long as the semantics of the invalid document, as created by such an implementation, is clear.

If consumers of signatures do change their behavior based on the op?

tional.creator value, they SHOULD take care that the way they process the signatures is not inconsistent with strictly validating signature consumers. (I.e. it is acceptable for a consumer to accept a signature based on a specific optional.creator value if other implementations would completely reject the signature, but it would be very undesirable for the two kinds of implementations to accept the signature in differ‐ ent and inconsistent situations.)

optional.timestamp

If present, this MUST be a JSON number, which is representable as a 64-bit integer, and identifies the time when the signature was created as the number of seconds since the UNIX epoch (Jan 1 1970 00:00 UTC).

signature   Container   container-signature(5)