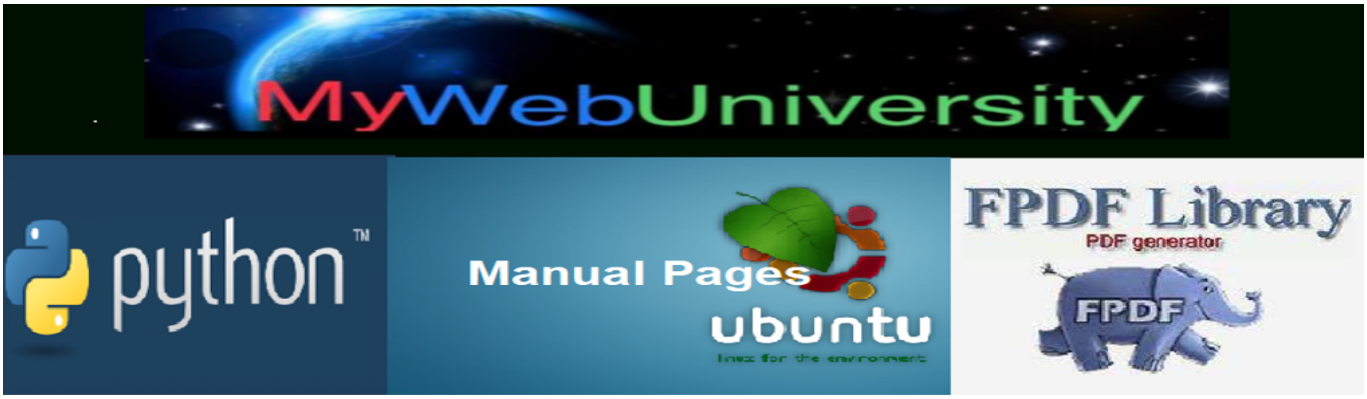

Rocky Enterprise Linux 9.2 Manual Pages on command 'containers-signature.5'

$ 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 implemented by the github.com/containers/image/signature package.

Most users should be able to consume these signatures by using the github.com/contain? ers/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 independent reimplementations.

High-level overview

The signature provides an end-to-end authenticated claim that a container 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 im? age for production) under a specified identity (e.g. an OS base image / specific applica? tion, with a specific version).

A container signature consists of a cryptographic signature which identifies and authenti? cates 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, contains other information about the image.

The signatures do not modify the container image (the layers, configuration, 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 im? age. An image 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 signa? ture? with independent blobs containing the JSON document and a cryptographic signature). 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 identi? fying the cryptographic signature format. It is expected that most formats are suffi? ciently self-describing that this is not necessary and the configured expected public key provides another indication of the expected cryptographic signature format. Such metadata may be added in the future for newly added cryptographic signature formats, if necessary.)

Consumers of container signatures SHOULD verify the cryptographic signature against one or more trusted public keys (e.g. defined in a policy.json signature verification policy file ?containers-policy.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 starting to process the JSON payload.

(Consumers MAY extract identification of the signing key and other metadata from the cryp? tographic signature, and the JSON payload, without verifying the signature, if the purpose is to allow managing the signature blobs, e.g. to list the authors and image identities of signatures associated with a single container image; if so, they SHOULD design the output of such processing to minimize the risk of users considering the output trusted or in any way usable for making policy decisions 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/sig? nature package does):

- ? The blob MUST be a ?Signed Message? as defined RFC 4880 section 11.3. (e.g. it MUST NOT be an unsigned ?Literal Message?, 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 cryptographic validation.

? The signature MUST correctly authenticate the included JSON payload (in particu?
lar, the parsing of the JSON payload MUST NOT start before the complete payload has been cryptographically 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 member in a JSON object).

Any violations of the JSON format or of other requirements in this document MAY be ac?
cepted 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 exactly 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 signature consumers MUST reject signa?
tures 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), con?
sumers MUST reject the signature if the critical object, or any of its subobjects, contain any member 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 incor?
rect types of expected members.

For the same reason, consumers SHOULD accept any members with unrecognized names in the optional object, and MAY accept signatures where the object member is recognized but un?
supported, or the value of the member is unsupported. Consumers still SHOULD reject sig?
natures where a member 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 valid? ity 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 container signature (three words, including the spaces).

Signature consumers MUST reject signatures which do not have this member 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 unexpected member or data.

(Currently only the docker-manifest-digest way of identifying a container image is defined; alternatives to this may be defined in the future, 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-digest.Digest string format.

The value of this member MUST match the manifest of the signed container image, as implemented in the docker/distribution manifest addressing 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 downloading layers of the image).

Implementation notes: * A single container image manifest may have several valid manifest digest values, using different algorithms. * For ?signed? docker/distribution schema 1 ?https://github.com/docker/distribution/blob/master/docs/spec/manifest-v2-1.md? manifests, the manifest 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 unexpected member or data.

(Currently only the docker-reference way of claiming an image identity/purpose is defined; alternatives to this may be defined in the future, 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/reference 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.identity.docker-reference member SHOULD use the fully explicit form (including 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 require critical.identity.docker-reference values with a completely different namespace to the reference used to refer to/download the image (e.g. requiring a critical.identity.docker-reference value which identifies the image as coming from a supplier when fetching it from a company-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-policy.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 policy.json ?containers-policy.json.5.md#signedby?.)

optional

This MUST be a JSON object.

Consumers SHOULD accept any members with unrecognized names in the optional object, and MAY accept a signature where the object member is recognized but unsupported, or the value of the member is valid but unsupported. Consumers still SHOULD reject any signature where a member of an optional object is supported but the value is recognized as invalid.

optional.creator

If present, this MUST be a JSON string, identifying the name and version of the software which has created the signature.

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

? SHOULD define the contents to unambiguously define the software in practice (e.g. it SHOULD contain the name of the software, 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 signature changes in any way; it SHOULD not be possible for two implementations which use a different format or semantics to have the same op? tional.creator value

? SHOULD use a format which is reasonably easy to parse in software (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 op? tional.creator (perhaps augmented with optional.timestamp), and MAY change their process? ing 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 retroac? tively), as long as the semantics of the invalid document, as created by such an implemen? tation, is clear.

If consumers of signatures do change their behavior based on the optional.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 com? pletely reject the signature, but it would be very undesirable for the two kinds of imple? mentations to accept the signature in different 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).