Mt Wilson

Document Containers

# Background

The need for this format arose when designing data integrity and data encryption formats. The integrity and encryption keys have associated attributes that must be securely bound to those keys so that it’s easy for authorized users to use a key correctly, possible for authorized users to intentionally use a key incorrectly, and hard for an attacker to trick an authorized user into using a key incorrectly. Also, integrity checks and encrypted data have a relationship to the original plaintext, and automated generic processing as well as interoperability across tools designed to work with the same data becomes easier when those relationships can be transmitted declaratively instead of being implied by specific application behavior.

# Requirements

The user stories are design goals, representing the problems that should be solved and value that can be obtained. Usually, there are numerous possible alternative implementations can satisfy a user story.

The user story format is “To <benefit>, as a <role> I want <what>”. Placing the benefit first highlights the reason anyone would undertake effort to implement the feature, makes it easy to group together related stories, and keeps the door open for new ideas on how the benefit can be realized by keeping it separate from the “what”.

## User Stories

To make file management easier, as a user I want a file format that can store multiple related parts that need to stay together.

To keep data and metadata together, as a developer I want to create a single file package containing an original file and its associated metadata.

To bind plaintext content and its metadata together, as a security analyst I want a file format that can declare a canonical way to combine the plaintext and metadata together as input to the digest algorithm.

To allow multiple users to sign a document, as a developer I want a file format that can support related parts such that multiple signatures can cover the same plaintext without interfering with each other.

To make system integration easier, as a developer I want a solution that leverages existing standards and tools.

# Architecture

## Container

There needs to be a container format that allows one or more documents and one or more sets of associated metadata, possibly in different formats, to be related to the documents. The container format itself does not need to include any facility for relating the parts. Its minimum responsibility is to package them together as one file in the file system that the user or application can copy, move/rename, edit, or delete.

Existing formats allow packaging an arbitrary number of files in a flat or hierarchical structure: zip, tar, iso, img, vdi, vhd, vmdk. There are more; these are just a few well-known formats.

Because there are numerous container formats, a specific choice should be left as an implementation detail as long as it meets all the requirements of this document.

## Document format

The document being packaged can be in any format that can be represented as a single file in a file system. It can be text or binary. The architecture must not make assumptions about the format of the document, but features such as metadata and linked data may have specific support for selected document formats. For example a document format which itself is a container such as zip, tar, or img may be used as a directory in a path expression in order to express a link directly to one of its parts.

At a minimum, documents are treated as opaque binary data.

## Metadata

Information about the document that is stored or transmitted separately from the document is metadata.

Existing formats are RFC822 headers, XML, JSON, and RDF.

## Linked data

There needs to be a format for describing relationships among files (parts) within the package (container format). The linked data format must be able to unambiguously identify two entities and a relationship between them. The set of possible relationships must not be constrained by the linked data format itself.

Files in the same container can be identified by a relative or absolute path in UNIX style, where the container is the root for absolute paths. Files outside the container can be identified by a URL, including use of the “file” scheme for other files on the local system.

Linked data can relate metadata to its subject document, two or more documents together as belonging to a collection, and encrypted data to its purpose (is it a document, a signature, a wrapped key, etc.).

Existing formats allow varying levels of ease in expressing relationships:

* HTML can relate two parts within the same document, and the current document to other documents, as a location and relationship. It has a “link” tag and an “anchor” tag that provide this in different contexts. <http://www.w3.org/TR/html5/>
* XML can relate two parts within the same document or from the current document to external resources. The arc link type with from/arcrole/to attributes can express relationships like RDF. <http://www.w3.org/TR/xlink11/>
* RDF provides a data-modeling vocabulary in which relationships are primary concepts, and each RDF item is a triple of subject, relationship, and object. <http://www.w3.org/TR/rdf-schema/> and <http://www.w3.org/TR/rdf-syntax-grammar/> describe an XML syntax.
* JSON-LD provides a data linking capability for the JSON format. <http://www.w3.org/TR/json-ld/>
* Multipart MIME provides a generic capability to loosely link multiple document parts with an identifier and link format. <http://tools.ietf.org/html/rfc2045>

## Integrity

It must be possible to apply integrity protection to any file within the container, including any complete set of metadata. It must also be possible to bind two or more files together with integrity protection, such as a file and its metadata.

An integrity descriptor is a metadata file that names the integrity algorithm, has a list of one or more items that should be protected, and also provides the output file in which the digest or signature is stored. If more than one item is in the list, the items are concatenated (no padding) for the purpose of calculating the digest. The descriptor may support other features such as including a literal, or the length of a file, or calling some other function and using its output as an item in the concatenated list.

It would be nice to be able to apply integrity protection to selected metadata (instead of all of it) such that critical metadata are protected while transient metadata (for example, what page was the user reading when the document was last closed) is free to change without breaking the integrity of the protected metadata. This could also be accomplished by storing the two sets of metadata separately and providing complete integrity protection only to the critical metadata.

Providing integrity protection to the entire container is not required - the ability to selectively protect any part within the container is enough. General network and storage software will maintain its own checksums over the entire container file to detect corruption, but applications that understand the container format and apply integrity protection to its contents can easily determine if any important part has been corrupted or tampered or not.

## Encryption

It must be possible to encrypt any file within the container, including any complete set of metadata.

It must be possible to provide metadata indicating the id of the encryption key and also the cipher algorithm, mode, padding mode, and other relevant information for decrypting the file.

# Profiles

The profiles combine a set of implementation choices into a coherent system that can be built and tested. The choices reflected in a profile may have advantages and disadvantages, but changing the choices by definition means creating a new profile.

Profiles can be compared by creating a table that has profile names across the top, technical questions in the left-most column, and filling it in with the choices made by each profile for each technical question.

## Container: TGZ

The TGZ format is a combination of tar and gzip. An alternative suffix for such files is “.tar.gz”

In benchmarks TGZ performs slightly better than ZIP in terms of compression ratio and speed. GZIP does not compress as well as BZIP2 or LZMA but it’s faster and requires less memory. Both tar and gzip are installed by default in most Linux distributions.

Apache has Java libraries in commons-compress for reading and writing tar files and using GZIP compression, and Java 7 also has built-in support for GZIP compression.

The tar and gzip tools are available for Windows (gnuwin32), and the free 7zip tool for Windows provides a graphical interface to creating .tar.gz files.

The tar and gzip tools are open source. It’s easy to create a new archive:

tar cfz container.tgz files...

It’s also easy to list the contents of a .tgz file without extracting it:

tar tfz container.tgz

Adding the verbose option will also show ownership, file modes, and last-modified dates:

tar tfzv container.tgz

It’s easy to extract files from a .tgz:

tar xfz container.tgz

And it’s also easy to extract a specific file:

tar xfz container.tgz filename

### Media Type

When downloading a container, the media type should be specified. Best practice regarding .tar.gz media type may be changing (formal IETF discussions in Nov 2014).

Currently websites that transmit a .tar.gz or .tgz file for download do so as application/octet-stream with a content disposition of “attachment”. This usually prevents browsers from automatically extracting them.

If the media type application/tar with content-encoding gzip is used, the browser will correctly interpret that as an on-the-wire encoding and automatically unzip the tar before saving. Some browsers will then automatically untar the file.

Here’s an email thread about using Content-Type: application/tar and Content-Encoding: gzip: <http://www.ietf.org/mail-archive/web/media-types/current/msg00609.html>

An “archive/\*” top level media type was discussed at IETF 91 recently (Nov 2014) <http://www.ietf.org/proceedings/91/slides/slides-91-arcmedia-0.pdf> <http://www.ietf.org/proceedings/91/arcmedia.html>

It might be appropriate to use a media type of archive/tar+gz depending on the outcome of the IETF discussions.

## Metadata and Linked data: JSON-LD

The JSON format has readily available parsers for multiple languages as well as an open source command line tool “jq” that can be used by shell scripts. It supports structured data but does not enforce a schema. It can be provided directly to web applications for display.

JSON-LD is a backward-compatible extension of JSON semantics which adds a mechanism for linking objects and adding context information to existing metadata in JSON, and makes it possible to declare relationships based on externally-defined semantics. It defines a minimum vocabulary for processing and all other relationships are defined by the application.

## Integrity: HMAC-SHA256 and RSA-SHA256

Integrity protection: SHA256 over specific files within the container. Integrity protection for a single document using just SHA256 only protects against accidental corruption, because an attacker can edit the document and update the digest. To protect against an attacker, the algorithm should be HMAC-SHA256 using a secret key or an RSA-SHA256 signature using a private key. When using HMAC-SHA256 or RSA-SHA256, the metadata and plaintext should be concatenated in order to bind them. For example, the metadata should include the integrity algorithm name such as “HMAC-SHA256” or “RSA-SHA256” and optionally identify the integrity key id so that in a system with multiple keys, the application can automatically load the right key for verification. Using the linked data feature, the integrity key itself (when using secret keys for HMAC) can be encrypted and stored in the container, wrapped with one or more recipient public keys so they can verify the integrity as well as edit the document and update the integrity protection. This profile specifies SHA256 to prevent downgrade attacks. Other profiles may allow other SHA-2 variants or other algorithms.

An example with a document, a metadata file, and an integrity protection file:

container.tgz

+ document.pdf

+ metadata.json

+ integrity.json

+ integrity.sig

metadata.json

{"can be": "any document metadata"}

integrity.json

{"integrity\_algorithm":"HMAC-SHA256", "key\_id":"a137a43e-03af-486a-ba05-3e0b4c159582", "manifest":["document.pdf", "metadata.json", "integrity.json"], "signature":"integrity.sig"}

integrity.sig

<binary signature output described by integrity.json>

Notice that the file integrity.json is itself included in the integrity manifest. All files mentioned in the manifest are concatenated (without padding) before hashing and the integrity signature covers all of them.

## Encryption: AES-128

Encryption: AES-128 in CBC or OFB modes. This profile specifies AES-128 to prevent downgrade attacks. Other profiles may allow longer-key variants of AES.

An example with a document, a metadata file, and an encryption descriptor file:

container.tgz

+ document.pdf.enc

+ metadata.json

+ encryption.json

metadata.json

{"can be": "any document metadata"}

encryption.json

{"encryption\_algorithm":"AES-128", "mode": "OFB", "padding": "NoPadding", "key\_id":"a137a43e-03af-486a-ba05-3e0b4c159582", "manifest": ["document.pdf.enc"]}

document.pdf.enc

<binary ciphertext of document.pdf described by encryption.json>

Unlike the integrity manifest, each of the files mentioned in the encryption manifest is decrypted separately, and if padding is specified then it applies to each item separately.

## Index: HTML

The index.html file is a standard entry point to websites. For containers that include multiple documents and features to be processed by one or more applications, it makes sense to include an index file that can guide both automated processes and interactive use.

The HTML “link” tag can be used to indicate entry points for specific applications. For example, if a container includes an encrypted document with a signature, the index.html can include a link tag to indicate there is an encrypted text document:

<!DOCTYPE html>

<html>

<head>

<link href="document.enc" type="encrypted/openssl; enclosed=&quot;text/plain&quot;; enclosed-charset=&quot;UTF-8&quot;; alg=&quot;aes-128-ofb&quot;; digest-alg=&quot;sha256&quot;; key-gen=&quot;pbkdf2&quot;; iter=&quot;2052228&quot;" rel="document" title="My Encrypted Document" data-content-id="85a7a2b5-95b2-4fb3-96f2-f2618a485763">

</head>

</html>

If the container is meant for interactive use, the index.html file could also contain a “body” tag with some helpful information, and links to documents the user can browse (text or office documents, or other HTML documents).

Applications reading the index.html file that don’t intend to display its contents can minimally parse it to extract all the link elements in the head section in case they need to find one or more files provided for a specific purpose (rel).

# Notes

The Open Packaging Conventions (OPC) format has a similar purpose, uses ZIP as the container format, and defines special XML files [Content\_Types].xml and \_rels/.rels that are the starting points for generic processing. <http://en.wikipedia.org/wiki/Open_Packaging_Conventions>

The Research Object Bundle (RO Bundle) format has a similar purpose, uses ZIP as the container format, and defines special XML file META-INF/manifest.xml and special JSON file .ro/manifest.json. <https://researchobject.github.io/specifications/bundle/>