Workgroup: Network Working Group
Internet-Draft:
draft-yasskin-wpack-use-cases-02
Published: 13 April 2021
Intended Status: Informational
Expires: 15 October 2021
Authors: J. Yasskin
Google

Use Cases and Requirements for Web Packages

Abstract

This document lists use cases for signing and/or bundling collections of web pages, and extracts a set of requirements from them.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the WPACK Working Group mailing list (wpack@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/wpack/.

Source for this draft and an issue tracker can be found at https://github.com/WICG/webpackage.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 15 October 2021.

Copyright Notice

Table of Contents

1. Introduction

People would like to use content offline and in other situations where there isn't a direct connection to the server where the content originates. However, it's difficult to distribute and verify the authenticity of applications and content without a connection to the network. The W3C has addressed running applications offline with Service Workers ([ServiceWorkers]), but not the problem of distribution.

Previous attempts at packaging web resources (e.g. Resource Packages and the W3C TAG's packaging proposal) were motivated by speeding up the download of resources from a single server, which is probably better achieved through other mechanisms like HTTP/2 PUSH, possibly augmented with a simple manifest of URLs a page plans to use. This attempt is instead motivated by avoiding a connection to the origin server at all. It may still be useful for the earlier use cases, so they're still listed, but they're not primary.

2. Use cases

These use cases are in rough descending priority order. If use cases have conflicting requirements, the design should enable more important use cases.

2.1. Essential

2.1.1. Offline installation

Alex can download a file containing a website (a PWA) including a Service Worker from origin O, and transmit it to their peer Bailey, and then Bailey can install the Service Worker with a proof that it came from O. This saves Bailey the bandwidth costs of transferring the website.

There are roughly two ways to accomplish this:

1. Package just the Service Worker Javascript and any other Javascript that it importScripts(), with their URLs and enough metadata to synthesize a navigator.serviceWorker.register(scriptURL, options) call, along with an uninterpreted but signature-checked blob of data that the Service Worker can interpret to fill in its caches.

2. Package the resources so that the Service Worker can fetch() them to populate its cache.

Associated requirements for just the Service Worker:

* Indexed by URL: The register() and importScripts() calls have semantics that depend on the URL.

* Signing as an origin: To prove that the file came from O.

* Signing uses existing TLS certificates: So O doesn't have to spend lots of money buying a specialized certificate.

* Cryptographic agility: Today's algorithms will eventually be obsolete and will need to be replaced.

* Certificate revocation: O's certificate might be compromised or mis-issued, and the attacker shouldn't then get an infinite ability to mint packages.

* Downgrade prevention: O's site might have an XSS vulnerability, and attackers with an old signed package shouldn't be able to take advantage of the XSS forever.

* Metadata: Just enough to generate the register() call, which is less than a full W3C Application Manifest.

Additional associated requirements for packaged resources:

* Indexed by URL: Resources on the web are addressed by URL.

* Request headers: If Bailey's running a different browser from Alex or has a different language configured, the accept* headers are important for selecting which resource to use at each URL.

* Response headers: The meaning of a resource is heavily influenced by its HTTP response headers.

* Resources from multiple origins in a package: So the site can be built from multiple components (Section 2.2.4).

* Metadata: The browser needs to know which resource within a package file to treat as its Service Worker and/or initial HTML page.

2.1.1.1. Online use

Bailey may have an internet connection through which they can, in real time, fetch updates to the package they received from Alex.

2.1.1.2. Fully offline use

Or Bailey may not have any internet connection a significant fraction of the time, either because they have no internet at all, because they turn off internet except when intentionally downloading content, or because they use up their plan partway through each month.

Associated requirements beyond Offline installation:

* Packaged validity information: Even without a direct internet connection, Bailey should be able to check that their package is still valid.

2.1.2. Offline browsing

Alex can download a file containing a large website (e.g. Wikipedia) from its origin, save it to transferrable storage (e.g. an SD card), and hand it to their peer Bailey. Then Bailey can browse the website with a proof that it came from O. Bailey may not have the storage space to copy the website before browsing it.

This use case is harder for publishers to support if we specialize Section 2.1.1 for Service Workers since it requires the publisher to adopt Service Workers before they can sign their site.

Associated requirements beyond Offline installation:

* Random access: To avoid needing a long linear scan before using the content.

* Compress stored packages: So that more content can fit on the same storage device.

2.1.3. Save and share a web page

Casey is viewing a web page and wants to save it either for offline use or to show it to their friend Dakota. Since Casey isn't the web page's publisher, they don't have the private key needed to sign the page. Browsers currently allow their users to save pages, but each browser uses a different format (MHTML, Web Archive, or files in a directory), so Dakota and Casey would need to be using the same browser. Casey could also take a screenshot, at the cost of losing links and accessibility.

Associated requirements:

* Unsigned content: A client can't sign content as another origin.

* Resources from multiple origins in a package: General web pages include resources from multiple origins.

* Indexed by URL: Resources on the web are addressed by URL.

* Response headers: The meaning of a resource is heavily influenced by its HTTP response headers.

2.1.4. Privacy-preserving prefetch

Lots of websites link to other websites. Many of these source sites would like the targets of these links to load quickly. The source could use <link rel="prefetch"> to prefetch the target of a link, but if the user doesn't actually click that link, that leaks the fact that the user saw a page that linked to the target. This can be true even if the prefetch is made without browser credentials because of mechanisms like TLS session IDs.

Because clients have limited data budgets to prefetch link targets, this use case is probably limited to sites that can accurately predict which link their users are most likely to click. For example, search engines can predict that their users will click one of the first couple results, and news aggreggation sites like Reddit or Slashdot can hope that users will read the article if they've navigated to its discussion.

Two search engines have built systems to do this with today's technology: Google's AMP and Baidu's MIP formats and caches allow them to prefetch search results while preserving privacy, at the cost of showing the wrong URLs for the results once the user has clicked. A good solution to this problem would show the right URLs

but still avoid a request to the publishing origin until after the user clicks.

Associated requirements:

* Signing as an origin: To prove the content came from the original origin.

* Streamed loading: If the user clicks before the target page is fully transferred, the browser should be able to start loading early parts before the source site finishes sending the whole page.

* Compress transfers

* Subsetting and reordering: If a prefetched page includes subresources, its publisher might want to provide and sign both WebP and PNG versions of an image, but the source site should be able to transfer only best one for each client.

2.2. Nice-to-have

2.2.1. Packaged Web Publications

The W3C's Publishing Working Group, merged from the International Digital Publishing Forum (IDPF) and in charge of EPUB maintenance, wants to be able to create publications on the web and then let them be copied to different servers or to other users via arbitrary protocols. See their Packaged Web Publications use cases for more details.

Associated requirements:

* Indexed by URL: Resources on the web are addressed by URL.

* Signing as an origin: So that readers can be sure their copy is authentic and so that copying the package preserves the URLs of the content inside it.

* Downgrade prevention: An early version of a publication might contain incorrect content, and a publisher should be able to update that without worrying that an attacker can still show the old content to users.

* Metadata: A publication can have copyright and licensing concerns; a title, author, and cover image; an ISBN or DOI name; etc.; which should be included when that publication is packaged.

Other requirements are similar to those from Offline installation:

* Random access: To avoid needing a long linear scan before using the content.

* Compress stored packages: So that more content can fit on the same storage device.

* Request headers: If different users' browsers have different capabilities or preferences, the accept* headers are important for selecting which resource to use at each URL.

* Response headers: The meaning of a resource is heavily influenced by its HTTP response headers.

* Signing uses existing TLS certificates: So a publisher doesn't have to spend lots of money buying a specialized certificate.

* Cryptographic agility: Today's algorithms will eventually be obsolete and will need to be replaced.

* Certificate revocation: The publisher's certificate might be compromised or mis-issued, and an attacker shouldn't then get an infinite ability to mint packages.

2.2.2. Avoiding Censorship

Some users want to retrieve resources that their governments or network providers don't want them to see. Right now, it's straightforward for someone in a privileged network position to block access to particular hosts, but TLS makes it difficult to block access to particular resources on those hosts.

Today it's straightforward to retrieve blocked content from a third party, but there's no guarantee that the third-party has sent the user an accurate representation of the content: the user has to trust the third party.

With signed web packages, the user can re-gain assurance that the content is authentic, while still bypassing the censorship. Packages don't do anything to help discover this content.

Systems that make censorship more difficult can also make legitimate content filtering more difficult. Because the client that processes a web package always knows the true URL, this forces content filtering to happen on the client instead of on the network.

Associated requirements:

* Indexed by URL: So the user can see that they're getting the content they expected.

* Signing as an origin: So that readers can be sure their copy is authentic and so that copying the package preserves the URLs of the content inside it.

2.2.3. Third-party security review

Some users may want to grant certain permissions only to applications that have been reviewed for security by a trusted third party. These third parties could provide guarantees similar to those provided by the iOS, Android, or Chrome OS app stores, which might allow browsers to offer more powerful capabilities than have been deemed safe for unaudited websites.

Binary transparency for websites is similar: like with Certificate Transparency [RFC6962], the transparency logs would sign the content of the package to provide assurance that experts had a chance to audit the exact package a client received.

Associated requirements:

* Additional signatures

2.2.4. Building packages from multiple libraries

Large programs are built from smaller components. In the case of the web, components can be included either as Javascript files or as <iframe>d subresources. In the first case, the packager could copy the JS files to their own origin; but in the second, it may be important for the <iframe>d resources to be able to make same-origin requests back to their own origin, for example to implement federated sign-in.

Associated requirements:

* Resources from multiple origins in a package: Each component may come from its own origin.

* Deduplication of diamond dependencies: If we have dependencies A->B->D and A->C->D, it's important that a request for a D resource resolves to a single resource that both B and C can handle.

2.2.4.1. Shared libraries

In ecosystems like Electron and Node, many packages may share some common dependencies. The cost of downloading each package can be

greatly reduced if the package can merely point at other dependencies to download instead of including them all inline.

Associated requirements:

* External dependencies

2.2.5. Cross-CDN Serving

When a web page has subresources from a different origin, retrieval of those subresources can be optimized if they're transferred over the same connection as the main resource. If both origins are distributed by the same CDN, in-progress mechanisms like [I-D.ietf-httpbis-http2-secondary-certs] allow the server to use a single connection to send both resources, but if the resource and subresource don't share a CDN or don't use a CDN at all, existing mechanisms don't help.

If the subresource is signed by its publisher, the main resource's server can forward it to the client.

There are some yet-to-be-solved privacy problems if the client and server want to avoid transferring subresources that are already in the client's cache: naively telling the server that a resource is already present is a privacy leak.

Associated requirements:

* Streamed loading: To get optimal performance, the browser should be able to start loading early parts of a resource before the distributor finishes sending the whole resource.

* Signing as an origin: To prove the content came from the original origin.

* Compress transfers

2.2.6. Pre-installed applications

Device manufacturers would like to ship their devices with some web applications pre-installed and usable even if the application is first used without an internet connection. Thereafter, the application should use the normal Service Worker update mechanism to stay up to date.

One way to accomplish this would be to pre-create a browser profile in the device's default browser and navigate it to each of the pre-installed apps before recording the device image. However, this means end-users miss the browser's initial setup flow and possibly that any "unique" cookies the sites set are now shared across

everyone who bought the device. It also doesn't help users who change their default browser.

If multiple browsers supported an unsigned web package format, with an option to trust it as if it were signed if it's in a particular section of the filesystem that's as protected as the browser's executable, and if registering a Service Worker from a page inside a package passed the full package contents to the Service Worker's install event, the device manufacturer could provide web packages for each pre-installed application that would work in the user's chosen browser.

Associated requirements:

* Service Worker integration: To pass the package into the install event and from there get its contents into a Cache.

2.2.7. Protecting Users from a Compromised Frontend

If an attacker gains control over a frontend server, any user who visits that server while they have control can have their web app upgraded to a hostile version. On the other hand, native applications either control their own update process or delegate it to an app store, which allows them to protect users by requiring that updates are signed by a trusted key. This protection isn't perfect---it's a Trust-On-First-Use mechanism that doesn't protect users who first install the application while the attacker controls the server they get it from, and attackers can bypass it by compromising the app's build system---but since both of those risks also apply to web apps, it does make the attack surface for native applications smaller than for web apps.

Not all application developers should choose to require signed updates, since doing so adds the risk of losing the signing key, but having this option gives security-sensitive applications like Dashlane an incentive to build native apps instead of web apps.

It has been difficult to add a signature requirement for web app upgrades because we haven't had a way to sign web resources. Web Packaging is expected to provide that, so we'll be able to consider the best way to do it.

Both HTTP Strict Transport Security (HSTS, [RFC6797]) and HTTP Public Key Pinning (HPKP, [RFC7469]) have established ways to pin assertions about a site's security for a bounded time after a visit. We could do the same with a web app's signing key.

Note that HPKP has been turned off in Chromium because it was difficult to use and made it too easy to "brick" a website. To reduce the chance of bricking the website, this key pinning design

could require an active Service Worker before enforcing the pins. It could also avoid the need for users to take manual action to recover from a lost signing key by allowing a new key to be used if it's seen consistently for a site-chosen amount of time, instead of waiting for the whole pin to expire. However, these mitigations don't guarantee that browsers would find the tradeoffs more acceptable than they did for HPKP.

One can think of a CDN as a potentially-compromised frontend and use this mechanism to limit the damage it can cause. However, this doesn't make it safe to use a wholly-untrustworthy CDN because of the risk to first-time users.

Associated requirements:

* Signing without origin trust: To let a backend system vouch for the content. This would likely be augmented with origin trust by receiving the signed content over TLS.

* Streamed loading: To get optimal performance, the browser should be able to start loading early parts of a resource before the server finishes sending the whole resource.

2.2.8. Installation from a self-extracting executable

The Node and Electron communities would like to install packages using self-extracting executables. The traditional way to design a self-extracting executable is to concatenate the package to the end of the executable, have the executable look for a length at its own end, and seek backwards from there for the start of the package.

Associated requirements:

* Trailing length

2.2.9. Packages in version control

Once packages are generated, they should be stored in version control. Many popular VC systems auto-detect text files in order to "fix" their line endings. If the first bytes of a package look like text, while later bytes store binary data, VC may break the package.

Associated requirements:

* Binary

2.2.10. Subresource bundling

Text based subresources often benefit from improved compression ratios when bundled together.

At the same time, the current practice of JS and CSS bundling, by compiling everything into a single JS file, also has negative side-effects:

1. Dependent execution - in order to start executing *any* of the bundled resources, it is required to download, parse and execute *all* of them.

2. Loss of caching granularity - Modification of *any* of the resources results in caching invalidation of *all* of them.

3. Loss of module semantics - ES6 modules must be delivered as independent resources. Therefore, current bundling methods, which deliver them with other resources under a common URL, require transpilation to ES5 and result in loss of ES6 module semantics.

An on-the-fly readable packaging format, that will enable resources to maintain their own URLs while being physically delivered with other resources, can resolve the above downsides while keeping the upsides of improved compression ratios.

To improve cache granularity, the client needs to tell the server which versions of which resources are already cached, which it could do with a Service Worker or perhaps with [I-D.ietf-httpbis-cache-digest].

Associated requirements:

*Indexed by URL

*Streamed loading: To solve downside 1.

*Compress transfers: To keep the upside.

*Response headers: At least the Content-Type is needed to load JS and CSS.

*Unsigned content: Signing same-origin content wastes space.

2.2.11. Archival

Existing formats like WARC ([ISO28500]) do a good job of accurately representing the state of a web server at a particular time, but a browser can't currently use them to give a person the experience of that website at the time it was archived. It's not obvious to the author of this draft that a new packaging format is likely to improve on WARC, compared to, for example, implementing support for WARC in browsers, but folks who know about archiving seem

interested, e.g.: https://twitter.com/anjacks0n/status/950861384266416134.

Because of the time scales involved in archival, any signatures from the original host would likely not be trusted anymore by the time the archive is viewed, so implementations would need to sandbox the content instead of running it on the original origin.

Associated requirements:

* Indexed by URL

* Response headers: To accurately record the server's response.

* Unsigned content: To deal with expired signatures.

* Time-shifting execution

3. Requirements

3.1. Essential

3.1.1. Indexed by URL

Resources should be keyed by URLs, matching how browsers look resources up over HTTP.

3.1.2. Request headers

Resource keys should include request headers like accept and accept-language, which allows content-negotiated resources to be represented.

This would require an extension to [MHTML], which uses the content-location response header to encode the requested URL, but has no way to encode other request headers. MHTML also has no instructions for handling multiple resources with the same content-location.

This also requires an extension to [ZIP]: we'd need to encode the request headers into ZIP's filename fields.

3.1.3. Response headers

Resources should include their HTTP response headers, like content-type, content-encoding, expires, content-security-policy, etc.

This requires an extension to [ZIP]: we'd need something like [JAR]'s META-INF directory to hold extra metadata beyond the resource's body.

3.1.4. Signing as an origin

Resources within a package are provably from an entity with the ability to serve HTTPS requests for those resources' origin [RFC6454].

Note that previous attempts to sign HTTP messages ([I-D.thomson-http-content-signature], [I-D.burke-content-signature], and [I-D.cavage-http-signatures]) omit a description of how a client should use a signature to prove that a resource comes from a particular origin, and they're probably not usable for that purpose.

This would require an extension to the [ZIP] format, similar to [JAR]'s signatures.

In any cryptographic system, the specification is responsible to make correct implementations easier to deploy than incorrect implementations (Section 3.1.12).

3.1.5. Random access

When a package is stored on disk, the browser can access arbitrary resources without a linear scan.

[MHTML] would need to be extended with an index of the byte offsets of each contained resource.

3.1.6. Resources from multiple origins in a package

A package from origin A can contain resources from origin B authenticated at the same level as those from A.

3.1.7. Cryptographic agility

Obsolete cryptographic algorithms can be replaced.

Planning to upgrade the cryptography also means we should include some way to know when it's safe to remove old cryptography (Section 3.2.6).

3.1.8. Unsigned content

Alex can create their own package without a CA-signed certificate, and Bailey can view the content of the package.

3.1.9. Certificate revocation

When a package is signed by a revoked certificate, online browsers can detect this reasonably quickly.

3.1.10. Downgrade prevention

Attackers can't cause a browser to trust an older, vulnerable version of a package after the browser has seen a newer version.

3.1.11. Metadata

Metadata like that found in the W3C's Application Manifest [W3C.WD-appmanifest-20170828] can help a client know how to load and display a package.

3.1.12. Implementations are hard to get wrong

The design should incorporate aspects that tend to cause incorrect implementations to get noticed quickly, and avoid aspects that are easy to implement incorrectly. For example:

* Explicitly specifying a cryptographic algorithm identifier in [RFC7515] made it easy for implementations to trust that algorithm, which caused vulnerabilities.

* [ZIP]'s duplicate specification of filenames makes it easy for implementations to check the signature of one copy but use the other.

* Following Langley's Law when possible makes it hard to deploy incorrect implementations.

3.2. Nice to have

3.2.1. Streamed loading

The browser can load a package as it downloads.

This conflicts with ZIP, since ZIP's index is at the end.

3.2.2. Signing without origin trust

It's possible to sign a resource with a key that has some effect on trust other than asserting that the origin's owner vouches for it. These keys could be expressed as raw public keys or as certificates with other key usages.

3.2.3. Additional signatures

Third-parties can vouch for packages by signing them.

3.2.4. Binary

The format is identified as binary by tools that might try to "fix" line endings.

This conflicts with using an [MHTML]-based format.

3.2.5. Deduplication of diamond dependencies

Nested packages that have multiple dependency routes to the same sub-package, can be transmitted and stored with only one copy of that sub-package.

3.2.6. Old crypto can be removed

The ecosystem can identify when an obsolete cryptographic algorithm is no longer needed and can be removed.

3.2.7. Compress transfers

Transferring a package over the network takes as few bytes as possible. This is an easier problem than Compress stored packages since it doesn't have to preserve Random access.

3.2.8. Compress stored packages

Storing a package on disk takes as few bytes as possible.

3.2.9. Subsetting and reordering

Resources can be removed from and reordered within a package, without breaking signatures (Section 3.1.4).

3.2.10. Packaged validity information

Certificate revocation and Downgrade prevention information can itself be packaged or included in other packages.

3.2.11. Signing uses existing TLS certificates

A "normal" TLS certificate can be used for signing packages. Avoiding extra requirements like "code signing" certificates makes packaging more accessible to all sites.

3.2.12. External dependencies

Sub-packages can be "external" to the main package, meaning the browser will need to either fetch them separately or already have them. (#35, App Installer Story)

3.2.13. Trailing length

The package's length in bytes appears a fixed offset from the end of the package.

This conflicts with [MHTML].

3.2.14. Time-shifting execution

In some unsigned packages, Javascript time-telling functions should return the timestamp of the package, rather than the true current time.

We should explore if this has security implications.

3.2.15. Service Worker integration

When a web page inside a package registers a Service Worker, that Service Worker's install event should receive a reference to the full package, with a way to copy the package's contents into a Cache object. ([ServiceWorkers])

4. Non-goals

Some features often come along with packaging and signing, and it's important to explicitly note that they don't appear in the list of Requirements.

4.1. Store confidential data

Packages are designed to hold public information and to be shared to people with whom the original publisher never has an interactive connection. In that situation, there's no way to keep the contents confidential: even if they were encrypted, to make the data public, anyone would have to be able to get the decryption key.

It's possible to maintain something similar to confidentiality for non-public packaged data, but doing so complicates the format design and can give users a false sense of security.

We believe we'll cause fewer privacy breaches if we omit any mechanism for encrypting data, than if we include something and try to teach people when it's unsafe to use.

4.2. Generate packages on the fly

See discussion at WICG/webpackage#6.

4.3. Non-origin identity

A package can be primarily identified as coming from something other than a Web Origin.

4.4. DRM

Special support for blocking access to downloaded content based on licensing. Note that DRM systems can be shipped inside the package even if the packaging format doesn't specifically support them.

4.5. Ergonomic replacement for HTTP/2 PUSH

HTTP/2 PUSH ([RFC7540], section 8.2) is hard for developers to configure, and an explicit package format might be easier. However, experts in this area believe we should focus on improving PUSH directly instead of routing around it with a bundling format.

Trying to bundle resources in order to speed up page loads has a long history, including Resource Packages from 2010 and the W3C TAG's packaging proposal from 2015.

However, the HTTPWG is doing a lot of work to let servers optimize the PUSHed data, and packaging would either have to re-do that or accept lower performance. For example:

* [I-D.vkrasnov-h2-compression-dictionaries] should allow individual small resources to be compressed as well as they would be in a bundle.

* [I-D.ietf-httpbis-cache-digest] tells the server which resources it doesn't need to PUSH.

Associated requirements:

* Streamed loading: If the whole package has to be downloaded before the browser can load a piece, this will definitely be slower than PUSH.

* Compress transfers: Keep up with [I-D.vkrasnov-h2-compression-dictionaries].

* Indexed by URL: Resources on the web are addressed by URL.

* Request headers: PUSH_PROMISE ([RFC7540], section 6.6) includes request headers.

* Response headers: PUSHed resources include their response headers.

5. Security Considerations

The security considerations will depend on the solution designed to satisfy the above requirements. See [I-D.yasskin-dispatch-web-packaging] for one possible set of security considerations.

6. IANA Considerations

This document has no actions for IANA.

7. Informative References

[I-D.burke-content-signature]
Burke, B., "HTTP Header for digital signatures", Work in Progress, Internet-Draft, draft-burke-content-signature-00, 7 March 2011, <https://tools.ietf.org/html/draft-burke-content-signature-00>.

[I-D.cavage-http-signatures]
Cavage, M. and M. Sporny, "Signing HTTP Messages", Work in Progress, Internet-Draft, draft-cavage-http-signatures-12, 21 October 2019, <https://tools.ietf.org/html/draft-cavage-http-signatures-12>.

[I-D.ietf-httpbis-cache-digest]
Oku, K. and Y. Weiss, "Cache Digests for HTTP/2", Work in Progress, Internet-Draft, draft-ietf-httpbis-cache-digest-05, 2 July 2018, <https://tools.ietf.org/html/draft-ietf-httpbis-cache-digest-05>.

[I-D.ietf-httpbis-http2-secondary-certs]
Bishop, M., Sullivan, N., and M. Thomson, "Secondary Certificate Authentication in HTTP/2", Work in Progress, Internet-Draft, draft-ietf-httpbis-http2-secondary-certs-06, 14 May 2020, <https://tools.ietf.org/html/draft-ietf-httpbis-http2-secondary-certs-06>.

[I-D.thomson-http-content-signature]
Thomson, M., "Content-Signature Header Field for HTTP", Work in Progress, Internet-Draft, draft-thomson-http-content-signature-00, 2 July 2015, <https://tools.ietf.org/html/draft-thomson-http-content-signature-00>.

[I-D.vkrasnov-h2-compression-dictionaries]
Krasnov, V. and Y. Weiss, "Compression Dictionaries for HTTP/2", Work in Progress, Internet-Draft, draft-vkrasnov-h2-compression-dictionaries-03, 5 March 2018, <https://tools.ietf.org/html/draft-vkrasnov-h2-compression-dictionaries-03>.

[I-D.yasskin-dispatch-web-packaging]
Yasskin, J., "Web Packaging", Work in Progress, Internet-Draft, draft-yasskin-dispatch-web-packaging-00, 30 June 2017, <https://tools.ietf.org/html/draft-yasskin-dispatch-web-packaging-00>.

[ISO28500] "WARC file format", ISO 28500:2017, 2017, <https://www.iso.org/standard/68004.html>.

[JAR] "JAR File Specification", 2014, <https://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html>.

[MHTML] Palme, J., Hopmann, A., and N. Shelness, "MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)", RFC 2557, DOI 10.17487/RFC2557, March 1999, <https://www.rfc-editor.org/rfc/rfc2557>.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <https://www.rfc-editor.org/rfc/rfc6454>.

[RFC6797] Hodges, J., Jackson, C., and A. Barth, "HTTP Strict Transport Security (HSTS)", RFC 6797, DOI 10.17487/RFC6797, November 2012, <https://www.rfc-editor.org/rfc/rfc6797>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/rfc/rfc6962>.

[RFC7469] Evans, C., Palmer, C., and R. Sleevi, "Public Key Pinning Extension for HTTP", RFC 7469, DOI 10.17487/RFC7469, April 2015, <https://www.rfc-editor.org/rfc/rfc7469>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/rfc/rfc7515>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/rfc/rfc7540>.

[ServiceWorkers] "Service Workers Nightly", W3C ED, <https://w3c.github.io/ServiceWorker/>.

[W3C.WD-appmanifest-20170828] Caceres, M., Christiansen, K., Lamouri, M., Kostiainen, A., and R. Dolin, "Web App Manifest", World Wide Web Consortium WD WD-

appmanifest-20170828, 28 August 2017, <https://www.w3.org/TR/2017/WD-appmanifest-20170828>.

[ZIP] "APPNOTE.TXT - .ZIP File Format Specification", 1 October 2014, <https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT>.

Appendix A. Acknowledgements

Thanks to Yoav Weiss for the Subresource bundling use case and discussions about content distributors.

Author's Address

Jeffrey Yasskin
Google

Email: jyasskin@chromium.org