Subresource Integrity

W3C Recommendation 23 June 2016

This version: http://www.w3.org/TR/2016/REC-SRI-20160623/ Latest published version: http://www.w3.org/TR/SRI/ Latest editor's draft: https://w3c.github.io/webappsec-subresource-integrity/ Implementation report: https://github.com/w3c/webappsec-subresource-integrity/wiki/Links Previous version: http://www.w3.org/TR/2016/PR-SRI-20160510/ Editors: Devdatta Akhawe, Dropbox, Inc., dev.akhawe@gmail.com Frederik Braun, Mozilla, fbraun@mozilla.com François Marier, Mozilla, francois@mozilla.com Joel Weinberger, Google, Inc., jww@google.com Participate: We are on Github. File a bug. Commit history. Mailing list.

Please note there may be errata for this document.
The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

This specification defines a mechanism by which user agents may verify that a fetched resource has been delivered without unexpected manipulation.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.
A list of changes to this document may be found at https://github.com/w3c/webappsec-subresource-integrity.
This document was published by the Web Application Security Working Group as a Recommendation. If you wish to make comments regarding this document, please send them to public-webappsec@w3.org (subscribe, archives) with [SRI] at the start of your email's subject. All comments are welcome.
Please see the Working Group's implementation report.
This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited from another document. W3C's role in making the Recommendation is to draw attention to the

specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.
W3C expects the functionality specified in this Recommendation will not be affected by changes to Fetch. The Working Group will continue to track the Fetch specification and document issues that impact this specification.
This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.
This document is governed by the 1 September 2015 W3C Process Document.

Table of Contents

1. Introduction

This section is non-normative.

Sites and applications on the web are rarely composed of resources from only a single origin. For example, authors pull scripts and styles from a wide variety of services and content delivery networks, and must trust that the delivered representation is, in fact, what they expected to load. If an attacker can trick a user into downloading content from a hostile server (via DNS poisoning, or other such means), the author has no recourse. Likewise, an attacker who can replace the file on the Content Delivery Network (CDN) server has the ability to inject arbitrary content.

Delivering resources over a secure channel mitigates some of this risk: with TLS, HSTS, and pinned public keys, a user agent can be fairly certain that it is indeed speaking with the server it believes it's talking to. These mechanisms, however, authenticate *only* the server, *not* the content. An attacker (or administrator) with access to the server can manipulate content with impunity. Ideally, authors would not only be able to pin the keys of a server, but also pin the *content*, ensuring that an exact representation of a resource, and *only* that representation, loads and executes.

This document specifies such a validation scheme, extending two HTML elements with an integrity attribute that contains a cryptographic hash of the representation of the resource the author expects to load. For instance, an author may wish to load some framework from a shared server rather than hosting it on their own origin. Specifying that the *expected* SHA-384 hash of https://example.com/example-framework.js is Li9vy3DqF8tnTXuiaAJuML3ky+er10rcgNR/VqsVpcw+ThHmYcwiB1pbOxEbzJr7 means that the user agent can verify that the data it loads from that URL matches that expected hash before executing the JavaScript it contains. This integrity verification significantly reduces the risk that an attacker can substitute malicious content.

This example can be communicated to a user agent by adding the hash to a script element, like so:

Example 1

```
<script src="https://example.com/example-framework.js"

integrity="sha384-Li9vy3DqF8tnTXuiaAJuML3ky+er10rcgNR/VqsVpcw+ThHmYcwiB1pbOxEbzJr7"
        crossorigin="anonymous"></script>
```

Scripts, of course, are not the only response type which would benefit from integrity validation. The scheme specified here also applies to link and future versions of this specification are likely

to expand this coverage.

1.1 Goals

1. Compromise of a third-party service should not automatically mean compromise of every site which includes its scripts. Content authors will have a mechanism by which they can specify expectations for content they load, meaning for example that they could load a *specific* script, and not *any* script that happens to have a particular URL.
2. The verification mechanism should have error-reporting functionality which would inform the author that an invalid response was received.

1.2 Use Cases/Examples

1.2.1 Resource Integrity

- An author wishes to use a content delivery network to improve performance for globally-distributed users. It is important, however, to ensure that the CDN's servers deliver *only* the code the author expects them to deliver. To mitigate the risk that a CDN compromise (or unexpectedly malicious behavior) would change that site in unfortunate ways, the following integrity metadata is added to the link element included on the page:
  Example 2
  ```
  <link rel="stylesheet" href="https://site53.example.net/style.css"

  integrity="sha384-+/M6kredJcxdsqkczBUjMLvqyHb1K/JThDXWsBVxMEeZHEaMKEOEct339VItX1zB"
        crossorigin="anonymous">
  ```

- An author wants to include JavaScript provided by a third-party analytics service. To ensure that only the code that has been carefully reviewed is executed, the author generates integrity metadata for the script, and adds it to the script element:
  Example 3
  ```
  <script src="https://analytics-r-us.example.com/v1.0/include.js"

  integrity="sha384-MBO5IDfYaE6c6Aao94oZrIOiC6CGiSN2n4QUbHNPhzk5Xhm0djZLQqTpL0HzTUxk"
          crossorigin="anonymous"></script>
  ```

- A user agent wishes to ensure that JavaScript code running in high-privilege HTML contexts (for example, a browser's New Tab page) aren't manipulated before display. Integrity metadata mitigates the risk that altered JavaScript will run in these pages' high-privilege contexts.

2. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, and *SHOULD* are to be interpreted as described in [RFC2119]. Conformance requirements phrased as algorithms or specific steps can be implemented in any manner, so long as the end result is equivalent. In particular, the algorithms defined in this specification are intended to be easy to understand and are not intended to be performant. Implementers are encouraged to optimize.

2.1 Key Concepts and Terminology

This section defines several terms used throughout the document.
The term digest refers to the base64-encoded result of executing a cryptographic hash function on an arbitrary block of data.
The term origin is defined in the Origin specification. [RFC6454]
The representation data and content encoding of a resource are defined by RFC7231, section 3. [RFC7231]
A base64 encoding is defined in RFC 4648, section 4. [RFC4648]
The SHA-256, SHA-384, and SHA-512 are part of the SHA-2 set of cryptographic hash functions defined by the NIST in “FIPS PUB 180-4: Secure Hash Standard (SHS)”.

2.2 Grammatical Concepts

The Augmented Backus-Naur Form (ABNF) notation used in this document is specified in RFC5234. [ABNF]
Appendix B.1 of [ABNF] defines VCHAR (printing characters).
WSP (white space) characters are defined in Section 2.4.1 Common parser idioms of the HTML 5 specification as White_Space characters.

3. Framework

The integrity verification mechanism specified here boils down to the process of generating a sufficiently strong cryptographic digest for a resource, and transmitting that digest to a user agent so that it may be used to verify the response.

3.1 Integrity metadata

To verify the integrity of a response, a user agent requires integrity metadata as part of the request. This metadata consists of the following pieces of information:

- cryptographic hash function (“alg”)
- digest (“val”)
- options (“opt”)

The hash function and digest *MUST* be provided in order to validate a response’s integrity.

> Note
> At the moment, no options are defined. However, future versions of the spec may define options, such as MIME types [MIMETYPE].

This metadata *MUST* be encoded in the same format as the hash-source (without the single quotes) in section 4.2 of the Content Security Policy Level 2 specification.
For example, given a script resource containing only the string alert(\'Hello, world.\');, an author might choose SHA-384 as a hash function. H8BRh8j48O9oYatfu5AZzq6A9RINhZO5H16dQZngK7T62em8MUt1FLm52t+eX6xO is the base64-encoded digest that results. This can be encoded as follows:

Example 4

```
sha384-H8BRh8j48O9oYatfu5AZzq6A9RINhZO5H16dQZngK7T62em8MUt1FLm52t+
eX6xO
```

Note
Digests may be generated using any number of utilities. OpenSSL, for example, is quite commonly available. The example in this section is the result of the following command line:

```
echo -n "alert('Hello, world.');" | openssl dgst -sha384
-binary | openssl base64 -A
```

3.2 Cryptographic hash functions

Conformant user agents *MUST* support the SHA-256, SHA-384 and SHA-512 cryptographic hash functions for use as part of a request's integrity metadata and *MAY* support additional hash functions.
User agents *SHOULD* refuse to support known-weak hashing functions like MD5 or SHA-1 and *SHOULD* restrict supported hashing functions to those known to be collision-resistant.
Additionally, user agents *SHOULD* re-evaluate their supported hash functions on a regular basis and deprecate support for those functions that have become insecure. See Hash collision attacks.

3.2.1 Agility

Multiple sets of integrity metadata may be associated with a single resource in order to provide agility in the face of future cryptographic discoveries. For example, the resource described in the previous section may be described by either of the following hash expressions:

Example 5

```
sha384-dOTZf16X8p34q2/kYyEFm0jh89uTjikhnzjeLeF0FHsEaYKb1A1cv+Lyv4Hk8vHd
sha512-Q2bFTOhEALkN8hOms2FKTDLy7euqP2zFZ1T8LCvX42Fp3WoNr3bjZSAHeOsHrbV1Fu9/A0EzCinRE7Af1ofPrw==
```

Authors may choose to specify both, for example:

Example 6

```
<script src="hello_world.js"

integrity="sha384-dOTZf16X8p34q2/kYyEFm0jh89uTjikhnzjeLeF0FHsEaYKb1A1cv+Lyv4Hk8vHd

sha512-Q2bFTOhEALkN8hOms2FKTDLy7euqP2zFZ1T8LCvX42Fp3WoNr3bjZSAHeOsHrbV1Fu9/A0EzCinRE7Af1ofPrw=="
    crossorigin="anonymous"></script>
```

In this case, the user agent will choose the strongest hash function in the list, and use that metadata to validate the response (as described below in the “parse metadata” and “get the strongest metadata from set” algorithms).
When a hash function is determined to be insecure, user agents *SHOULD* deprecate and eventually remove support for integrity validation using the insecure hash function. User agents *MAY* check the validity of responses using a digest based on a deprecated function.
To allow authors to switch to stronger hash functions without being held back by older user agents, validation using unsupported hash functions acts like no integrity value was provided (see the “Does response match metadataList” algorithm below). Authors are encouraged to use strong hash functions, and to begin migrating to stronger hash functions as they become available.

3.2.2 Priority

User agents must provide a mechanism for determining the relative priority of two hash functions and return the empty string if the priority is equal. That is, if a user agent implemented a function like getPrioritizedHashFunction(a, b) it would return the hash function the user agent considers the most collision-resistant. For example, getPrioritizedHashFunction('sha256', 'sha512') would return 'sha512' and getPrioritizedHashFunction('sha256', 'sha256') would return the empty string.

Note

The getPrioritizedHashFunction is an internal implementation detail. It is not an API that implementors provide to web applications. It is used in this document only to simplify the algorithm description.

3.3 Response verification algorithms

3.3.1 Apply algorithm to response

1. Let result be the result of applying algorithm to the representation data without any content-codings applied, except when the user agent intends to consume the content with content-encodings applied. In the latter case, let result be the result of applying algorithm to the representation data.
2. Let encodedResult be result of base64-encoding result.
3. Return encodedResult.

3.3.2 Is response eligible for integrity validation

In order to mitigate an attacker's ability to read data cross-origin by brute-forcing values via integrity checks, responses are only eligible for such checks if they are same-origin or are the result of explicit access granted to the loading origin via Cross Origin Resource Sharing [CORS].

Note
As noted in RFC6454, section 4, some user agents use globally unique identifiers for each file URI. This means that resources accessed over a file scheme URL are unlikely to be eligible for integrity checks.

Note
Being in a Secure Context (e.g., a document delivered over HTTPS) is not necessary for the use of integrity validation. Because resource integrity is only an application level security tool, and it does not change the security state of the user agent, a Secure Context is unnecessary. However, if integrity is used in something other than a Secure Context (e.g., a document delivered over HTTP), authors should be aware that the integrity provides *no security guarantees at all*. For this reason, authors should only deliver integrity metadata in a Secure Context. See Non-secure contexts remain non-secure for more discussion.

The following algorithm details these restrictions:

1. Let response be the response that results from fetching the resource.
2. If the response type is basic, cors or default, return true.
3. Return false.

Note
The response types are defined by the Fetch specification [FETCH] and refer to the following:

- basic is a same-origin response, and thus the requestor has full access to read the body.
- cors is a valid response to a cross-origin, CORS-enabled request, and thus again the requestor has full access to read the body.
- default is a valid response that is generated by a Service Worker as a response to the request, so its body, too, is fully readable by the requestor.

3.3.3 Parse metadata.

This algorithm accepts a string, and returns either no metadata, or a set of valid hash expressions whose hash functions are understood by the user agent.

1. Let result be the empty set.
2. Let empty be equal to true.
3. For each token returned by splitting metadata on spaces:
    1. Set empty to false.
    2. If token is not a valid metadata, skip the remaining steps, and proceed to the next token.
    3. Parse token per the grammar in integrity metadata.
    4. Let algorithm be the alg component of token.
    5. If algorithm is a hash function recognized by the user agent, add the parsed token to result.
4. Return no metadata if empty is true, otherwise return result.

3.3.4 Get the strongest metadata from set.

1. Let result be the empty set and strongest be the empty string.
2. For each item in set:
    1. If result is the empty set, add item to result and set strongest to item, skip to the next item.
    2. Let currentAlgorithm be the alg component of strongest.
    3. Let newAlgorithm be the alg component of item.
    4. If the result of getPrioritizedHashFunction(currentAlgorithm, newAlgorithm) is the empty string, add item to result. If the result is newAlgorithm, set strongest to item, set result to the empty set, and add item to result.
3. Return result.

3.3.5 Does response match metadataList?

1. Let parsedMetadata be the result of parsing metadataList.
2. If parsedMetadata is no metadata, return true.
3. If response is not eligible for integrity validation, return false.

4. If parsedMetadata is the empty set, return true.
5. Let metadata be the result of getting the strongest metadata from parsedMetadata.
6. For each item in metadata:
    1. Let algorithm be the alg component of item.
    2. Let expectedValue be the val component of item.
    3. Let actualValue be the result of applying algorithm to response.
    4. If actualValue is a case-sensitive match for expectedValue, return true.
7. Return false.

This algorithm allows the user agent to accept multiple, valid strong hash functions. For example, a developer might write a script element such as:

Example 7

```
<script src="https://example.com/example-framework.js"

integrity="sha384-Li9vy3DqF8tnTXuiaAJuML3ky+er10rcqNR/VqsVpcw+ThHmYcwiB1pbOxEbzJr7

sha384-+/M6kredJcxdsqkczBUjMLvqyHb1K/JThDXWsBVxMEeZHEaMKEOEct339VItX1zB"
        crossorigin="anonymous"></script>
```

which would allow the user agent to accept two different content payloads, one of which matches the first SHA384 hash value and the other matches the second SHA384 hash value.

Note
User agents may allow users to modify the result of this algorithm via user preferences, bookmarklets, third-party additions to the user agent, and other such mechanisms. For example, redirects generated by an extension like HTTPS Everywhere could load and execute correctly, even if the HTTPS version of a resource differs from the HTTP version.

Note
This algorithm returns false if the response is not eligible for integrity validation since Subresource Integrity requires CORS, and it is a logical error to attempt to use it without CORS. Additionally, user agents *SHOULD* report a warning message to the developer console to explain this failure.

3.4 Verification of HTML document subresources

A variety of HTML elements result in requests for resources that are to be embedded into the document, or executed in its context. To support integrity metadata for some of these elements, a new integrity attribute is added to the list of content attributes for the link and script elements. A corresponding integrity IDL attribute which reflects the value each element's integrity content attribute is added to the HTMLLinkElement and HTMLScriptElement interfaces.

> Note
> A future revision of this specification is likely to include integrity support for all possible subresources, i.e., a, audio, embed, iframe, img, link, object, script, source, track, and video elements.

3.5 The integrity attribute

The integrity attribute represents integrity metadata for an element. The value of the attribute *MUST* be either the empty string, or at least one valid metadata as described by the following ABNF grammar:

```
integrity-metadata = *WSP hash-with-options *( 1*WSP
hash-with-options ) *WSP / *WSP
hash-with-options  = hash-expression *("?" option-expression)
option-expression  = *VCHAR
hash-algo          = <hash-algo production from [Content Security
Policy Level 2, section 4.2]>
base64-value       = <base64-value production from [Content Security
Policy Level 2, section 4.2]>
hash-expression    = hash-algo "-" base64-value
```

The integrity IDL attribute must reflect the integrity content attribute.
option-expressions are associated on a per hash-expression basis and are applied only to the hash-expression that immediately precedes it.
In order for user agents to remain fully forwards compatible with future options, the user agent *MUST* ignore all unrecognized option-expressions.

> Note
> Note that while the option-expression has been reserved in the syntax, no options have been defined. It is likely that a future version of the spec will define a more specific syntax for options, so it is defined here as broadly as possible.

3.6 Element interface extensions

3.6.1 HTMLLinkElement

```
partial interface HTMLLinkElement {
                    attribute DOMString integrity;
};
```

3.6.1.1 Attributes

integrity of type DOMString integrity

3.6.2 HTMLScriptElement

```
partial interface HTMLScriptElement {
                    attribute DOMString integrity;
};
```

3.6.2.1 Attributes

integrity of type DOMString integrity

3.7 Handling integrity violations

The user agent will refuse to render or execute responses that fail an integrity check, instead returning a network error as defined in Fetch [FETCH].

> Note
> On a failed integrity check, an error event is fired. Developers wishing to provide a canonical fallback resource (e.g., a resource not served from a CDN, perhaps from a secondary, trusted, but slower source) can catch this error event and provide an appropriate handler to replace the failed resource with a different one.

3.8 Elements

3.8.1 The link element for stylesheets

Whenever a user agent attempts to obtain a resource pointed to by a link element that has a rel attribute with the keyword of stylesheet, modify step 4 to read:
Do a potentially CORS-enabled fetch of the resulting absolute URL, with the mode being the current state of the element's crossorigin content attribute, the origin being the origin of the link element's Document, the default origin behavior set to taint, and the integrity metadata of the

request set to the value of the element's integrity attribute.

3.8.2 The script element

Replace step 14.1 of HTML5's "prepare a script" algorithm with:

1. Let src be the value of the element's src attribute and the request's associated integrity metadata be the value of the element's integrity attribute.

4. Proxies

Optimizing proxies and other intermediate servers which modify the responses *MUST* ensure that the digest associated with those responses stays in sync with the new content. One option is to ensure that the integrity metadata associated with resources is updated. Another would be simply to deliver only the canonical version of resources for which a page author has requested integrity verification.
To help inform intermediate servers, those serving the resources *SHOULD* send along with the resource a Cache-Control header with a value of no-transform.

5. Security Considerations

This section is non-normative.

5.1 Non-secure contexts remain non-secure

Integrity metadata delivered by a context that is not a Secure Context, such as an HTTP page, only protects an origin against a compromise of the server where an external resources is hosted. Network attackers can alter the digest in-flight (or remove it entirely, or do absolutely anything else to the document), just as they could alter the response the hash is meant to validate. Thus, it is recommended that authors deliver integrity metadata only to a Secure Context. See also securing the web.

5.2 Hash collision attacks

Digests are only as strong as the hash function used to generate them. It is recommended that user agents refuse to support known-weak hashing functions and limit supported algorithms to those known to be collision resistant. Examples of hashing functions that are not recommended include MD5 and SHA-1. At the time of writing, SHA-384 is a good baseline.
Moreover, it is recommended that user agents re-evaluate their supported hash functions on a regular basis and deprecate support for those functions shown to be insecure. Over time, hash functions may be shown to be much weaker than expected and, in some cases, broken, so it is important that user agents stay aware of these developments.

5.3 Cross-origin data leakage

This specification requires the CORS settings attribute to be present on integrity-protected cross-origin requests. If that requirement were omitted, attackers could violate the same-origin policy and determine whether a cross-origin resource has certain content.

Attackers would attempt to load the resource with a known digest, and watch for load failures. If the load fails, the attacker could surmise that the response didn't match the hash and thereby gain some insight into its contents. This might reveal, for example, whether or not a user is logged into a particular service.
Moreover, attackers could brute-force specific values in an otherwise static resource. Consider a JSON response that looks like this:

Example 8

```
{'status': 'authenticated', 'username': 'admin'}
```

An attacker could precompute hashes for the response with a variety of common usernames, and specify those hashes while repeatedly attempting to load the document. A successful load would confirm that the attacker has correctly guessed the username.

6. Acknowledgements

Much of the content here is inspired heavily by Gervase Markham's Link Fingerprints concept, as well as WHATWG's Link Hashes.
A special thanks to Mike West of Google, Inc. for his invaluable contributions to the initial version of this spec. Additionally, Brad Hill, Anne van Kesteren, Jonathan Kingston, Mark Nottingham, Dan Veditz, Eduardo Vela, Tanvi Vyas, and Michal Zalewski provided invaluable feedback.

A. References

A.1 Normative references

[ABNF] D. Crocker, Ed.; P. Overell. IETF. Augmented BNF for Syntax Specifications: ABNF. January 2008. Internet Standard. URL: https://tools.ietf.org/html/rfc5234 [CORS] Anne van Kesteren. W3C. Cross-Origin Resource Sharing. 16 January 2014. W3C Recommendation. URL: http://www.w3.org/TR/cors/ [FETCH] Anne van Kesteren. WHATWG. Fetch Standard. Living Standard. URL: https://fetch.spec.whatwg.org/ [MIMETYPE] Ned Freed; Nathaniel S. Borenstein. IETF. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. Draft Standard. URL: https://tools.ietf.org/html/rfc2046 [RFC2119] S. Bradner. IETF. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119 [RFC4648] Simon Josefsson. IETF. The Base16, Base32, and Base64 Data Encodings. Proposed Standard. URL: https://tools.ietf.org/html/rfc4648 [RFC6454] A. Barth. IETF. The Web Origin Concept. December 2011. Proposed Standard. URL: https://tools.ietf.org/html/rfc6454 [RFC7231] R. Fielding, Ed.; J. Reschke, Ed.. IETF. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7231
↥