This paper was released 01 June, 2018. For the most recent revision, related source code visit http://securityaversion.net

BREAKING DOWN SUBRESOURCE INTEGRITY (SRI)

Jaume Boniquet Travila
01 June 2018, Barcelona, SPAIN

Abstract

Today's web sites are rich in content, using multiple architecture coding languages and frameworks. As technology advances, so does the web standard specification.

In order to provide in websites advanced and reactive capabilities to user actions, it is required the use of scripting languages such as JavaScript, Node.js, etc. There are many source codes libraries that already provide standard functions to perform common actions, such as pop-ups, fade in/out windows, and almost any kind of behavior that is desired.

It is common to leverage the use of third party scripts to load in websites, to provide advanced features or functionalities.

As mentioned before, among others, that could JavaScript or Node.js. But loading third party scripts from untrusted sources might be dangerous. In this paper is analyzed how sensitive is to load untrusted source code from uncontrolled locations and how does behave the W3C Subresource Integrity (SRI) to some custom attacks to mitigate this potentially threat.

1. Introduction

It is common to see websites loading external scripts to provide advanced features, menus, time counters, or reactive actions depending the user interaction within a website, but this poses a severe risk to our website integrity, confidentiality and reputation.

The reason is simply. Any source code loaded from untrusted sources (out of your control) is susceptible to be changed or altered without any notification. This means that legitimate scripts loaded from external sites, such as a CDN or a non-custom locations, could be compromised for hostile purposes.

Additionally, not only the risk comes from hostile purposes, but also from unintentional source code edits that could render the service unusable or

with errors facing to all service users (potential customers for example).

This paper focuses in some attack vectors that could potentially make the SRI check useless or bypass it.

2. Related Work.

Related work to the vulnerabilities described in this paper does exist, and a few examples will be described below.

An article [1] published in Troyhunt website describes how the "*Brownsealoud*" product was affected by a cryptominer. The issue was caused because a JavaScript code was loaded directly from a Github repository (Igo Escobar account) and one of the project contributors (web developer) introduced the following source code:

```
<script
src="https://github.com/igorescobar/jQuery-Mask-Plugin/blob/gh-pages/js/jquery.mask.min.js"
type="text/javascript></script>
```

As can be seen, it loads another JavaScript code, which was tampered with and a JavaScript code was added to create a cryptominer that affected over 4.000 websites.

[1] https://www.troyhunt.com/the-javascript-supply-chain-paradox-sri-csp-and-trust-in-third-party-libraries/

After further research, no tools or proof of concepts were found capable to break or compromise the SRI.

3. SRI concepts and overview.

The purpose of this paper is no to explain how does the Subresource Integrity Check (SRI) work or to be a guide explaining detailed concepts of specific features, however a basis is required to understand how the attack vectors may be exploited.

SRI uses an integrity value that may contain multiple hashes separated by a whitespace.

These hashes in reality are cryptographic digest in Base-64 format, by applying a particular hash function to some input (for example a script or a style sheet file). But it is common to use shorthand hash to mean cryptographic digest, so this is what is used in SRI.

The current SRI specification supports the following cryptographic hashes:

- SHA-256
- SHA-384
- SHA-512

The stronger algorithm is used, the longer it will take the client side to calculate the hash for the given resource (it is quite fast, but for huge source code input files, calculating the hash might take some seconds).

In order to generate a hash, several tools can be used for that purpose, but in this case it has been chosen the command line tool called "shasum", which can be called using the following command:

"shasum –b –a 256 <filename>"

Below is an example of a SHA-256 bit hash of the word "test":

9F86D081884C7D659A2FEAA0C55AD015A3BF4F1B2B0B822CD15D6C15B0F00A08

Next is a SHA-384 hash, also using the same text as before.

768412320f7b0aa5812fce428dc4706b3cae50e02a64caa16a782249bfe8efc4b7ef1ccb126255d196047dfedf17a0a9

Finally is a SHA-512 encoded with the same text:

ee26b0dd4af7e749aa1a8ee3c10ae9923f618980772e473f8819a5d4940e0db27ac185f8a0e1d5f84f88bc887fd67b143732c304cc5fa9ad8e6f57f50028a8ff

Note that the hash length increases from 32 characters in the first case, followed by 64 and finally 64.

For more information about hashes, visit the following [2]reference.

[2] https://en.wikipedia.org/wiki/Secure_Hash_Algorithms

On each browser request with the SRI enabled, it will perform a hash function over the requested resource and them compare both hashed. A resource will only be loaded if I match on of those hashes hardcoded within the code.

An easy way to generate and automatically calculate the hash for a given script file or URI is using the following online service:

https://www.srihash.org/

In the following example we have used the public JQuery library (http://code.jquery.com/jquery-2.2.3.min.js) and the SRI compliant output was as follows:

```
<script
src="https://code.jquery.com/jquery-
2.2.3.min.js" integrity="sha384-
I6F5OKECLVtK/BL+8iSLDEHowS
AfUo76ZL9+kGAgTRdiByINKJaqT
PH/QVNS1VDb"
crossorigin="anonymous"></script>
```

Another way to generate the integrity hashes would be using the following shell command line:

```
openssl dgst -sha384 -binary
FILENAME.js | openssl base64 -A
```

As explained before, mind that the SRI is a feature enabled on client side, to be precise on the web browser. If the web browser does not

supper or has compatibility with the security standard, it will simply not work.

Therefore below is a list of all browsers and its compatibility with SRI.

Besides all the steps described above to make SRI work, there is still one additional requirement to be set. The webserver serving the files must have the "*crossorigin*" or Cross Origin Resource Sharing (CORS) header enabled (this is a requirement set by the W3C organization). Before going deeper, first it will be briefly described how does CORS work and is intended to behave.

CORS is a mechanism that allows restricted resources (such as files, fonts, style sheets, etc) on a web page to be retrieved from another domain from which the first resource was served.

CORS defines a way in which a browser and server can interact to determine whether or not it is safe to allow the cross-origin request.

CORS is supported in the following browsers:

- Chrome 3+
- Firefox 3.5+
- Opera 12+
- Safari 4+
- Internet Explorer 8+

For simple CORS requests, the server only needs to add the following header to its response:

```
Access-Control-Allow-Origin: *
```

Important note: Enabling CORS in a web server with the Allow Origin set to *True* can pose at risk some resources. With that flag enabled means that any website requesting a resource from the server will be served and read. Based on the CORS W3 Specification it is up to the client to determine and enforce the restriction of whether the client has access to the response data based on this header. This configuration is very insecure, and is not acceptable in general terms, except in the case of a public API that is intended to be accessible by everyone.

A secure use of Allow Origin would be by specifying a list of the valid domains that are allowed to read the responses. Therefore, this header shall only be scoped to only those specific resources that are public. The asterisk setting, meaning allows anonymous requests, shall not be enabled for web servers containing private pages or resources that access is controlled through user validation.

As the standard only allow to specify one domain within the header, some custom rules do exist for each specific web server to retrieve the domain name of the resource requester, compare it with a given

list in the web server settings, and send the corresponding Allow Origin header that corresponds (if allowed)

Access-Control-Allow-Origin: https://example.com

Please see Figure #1 for details regarding the web browsers compatibility to interpret CORS headers returned from the servers.

In this test, it was enabled CORS response header in Internet Information Services (IIS) webserver in a Windows server 2012 operating System. To do so, it is required to manually add the following lines of code to the "*web.config*" file or by using the IIS graphical editor.

```
<configuration>
<system.webServer>
<httpProtocol>
<customHeaders>
<add name="Access-Control-Allow-Origin" value="*" />
</customHeaders>
</httpProtocol>
</system.webServer>
</configuration>
```

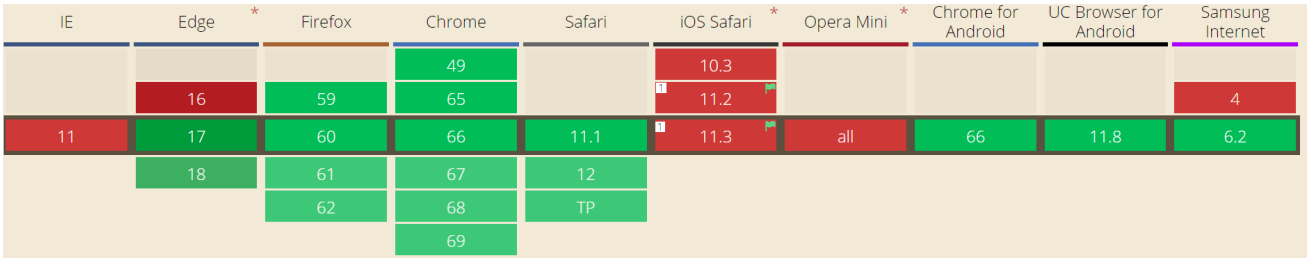

Figure 1 – Browsers SRI compatibility list

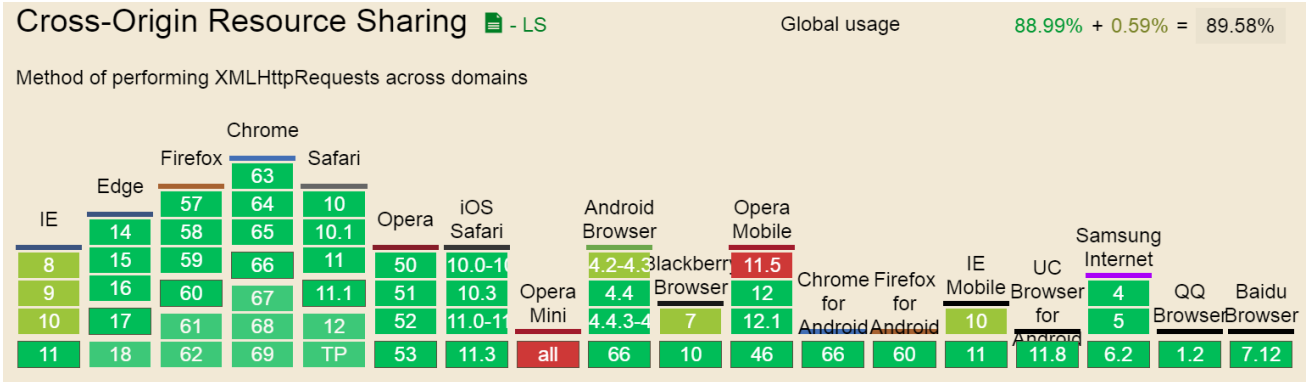

Figure 2 – Browsers CORS compatibility list

4. Advantages and disadvantages of a CDN

As mentioned before, SRI is usually employed when loading resources from external parties. One of the most common services used is a CDN. A Content Delivery Network (CDN) is a geographically distributed group of proxy servers. Its goal is to distribute service and resources in a very fast way, by detecting the origin of the request and serving the content from a geographically near (or any other low latency server). It also provides high performance and availability, capable of serving broad types of content, ranging from static files (scripts, texts, fonts, images, etc), software, raw files, live stream media and social networks.
See Figure #3 for a typical CDN headers returned from the servers.

configuration showing a comparative between a regular network serving content versus a CDN.

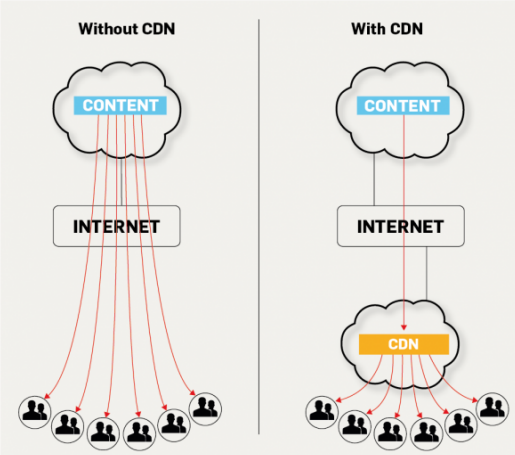

Figure 3 – CDN vs no CDN

Advantages	Disadvantages
Server with less load	CDN's service have associated costs (usually charged per served GB)
More concurrent users	Add layers of complexity to site releases
Content delivered faster	Some CDN's might be blocked by user's firewalls
Geo-location of the content	Content served is out of your control and could potentially be changed without notice
Caching capabilities	Server responses, headers and settings less customizable than in premise web servers

Figure 4 – CDN comparative

5. Types of Attackers

Attackers typically fall into one of three areas:

- Amateur: Amateurs are curious individuals who carry out attacks just to "see if it can be done."
- Expert: Experts attack under the auspices of scientific institutions and universities studying the technology.
- Professional: Professionals attack for financial reward or to obtain sensitive data and compromise a system.

6.1 Attack vector #1: Running SRI protected tags without browsers support

Some testing was performed by disabling such feature in Firefox latest version (In Google Chrome was not possible to disable it as it has no advanced menus for feature personalization). When disabling the SRI feature in a page with it enabled within the source code and a hash calculated, turned out that no issues were caused by the SRI tags, simply, were ignored by the web browser without prompting any message to the user (but it does in the Inspect logs page), however, the script itself, and was executed.

- In order to disable the SRI in Firefox, surf to "*about:config*", then search for the following term "*security.sri.enable*" and change it to "*False*"

Attack vector mitigations:

Major web browsers latest versions (Microsoft Edge, Google Chrome, Firefox and Safari) do support SRI. Only old versions of Internet Explorer, which are still currently in use by some old Windows, do not support SRI. This is not a big problem, but users will not be in that high level of protection. The mitigation for this issue is to update to a newer Operating System version or simply, move to one of the before mentioned web browsers that actually supports it.

6.2. Attack vector #2: Wrong implementation of SRI without CORS enabled

As previously described, CORS is mandatory to be enabled in the webserver that hosts the files to be checked with SRI. The W3C standard clearly states that this is a [3]requirement but sometimes web browsers do differ. An attempt to use an old web browser without support for SRI, and CORS disabled was performed and the results were unsuccessful, meaning that the script executed without being verified by its calculated hash.

On the other hand, the same test was performed in a web browser with SRI support but CORS disabled in the server

[3] https://www.w3.org/TR/SRI/

side, and the script did not executed, simply the "*integrity*" attribute was ignored not being able to be verified using the enclosed hash.

Figure 5 – Web browser console error

Attack vector mitigations:

Reducing the impact for this vulnerability is not really tough at all. We have seen during the test that modern web browsers do support SRI and prevents the execution of the script if there is some error or misconfiguration. Also does not allow to execute the script if the hash does not match with the one embedded in the source code.

In regards to the old web browsers, it showed that the script was executed, the hash was not verified and no warnings were displayed to the user.

Therefore, any critical application making use of SRI with CORS properly enabled in the server side would be useless for clients accessing from old web browser versions (please see Figure 1 – Browsers SRI compatibility list).

Attached is below an W3C approved service to test out web browser SRI capabilities:

http://w3c-test.org/subresource-integrity/subresource-integrity.sub.html

The only mitigation factor is end users to keep up to date their web browser. Actually, most vendors provide auto-update capabilities by default (unless disabled by the user).

Last but not least, mind that usually SRI is deployed in scripts loaded from other domain (commonly retrieved from a CDN for performance purposes), but if for some reason the external resource does not work or is down, the script will not load and very likely will disrupt the normal behavior of the website. In order to avoid this potential issue, [4]Mozilla foundation suggests adding some custom code to load from locally the resource only in case it was not possible to be retrieved from the original source. Below is an example of this failover solution:

```
<script
src="//SomeCDN.com//SomeCode.js
" integrity="sha384-******..."
crossorigin="anonymous"></script>
<script>(window.jQuery) ||
document.write('<script
src="/scripts/SomeCode.js"><\/script
>');
</script>
```

[4] https://hacks.mozilla.org/2015/09/subresource-integrity-in-firefox-43/

6.3 Attack vector #3: Hash collision

A theoretical attack was devised to look for hash collisions. For example, if an attacker gain access to a CDN, an manages to modify or specially craft a specific resource that is being loaded by another service and implements SRI, it will simply not load because the hash would not match.

It is known that some old hash functions were vulnerable to collisions. For example, MD5 is vulnerable to it and SHA-1 does also not provide enough entropy to generate a secure hash taking in consideration the current CPU processing capabilities.

Specially crafted resources could potentially be forged to match the unalterable hash of the website, but the content be totally different than the original one.

As can be observed in figure #7, a set of different characters can produce the same hash, when theoretically this is impossible.

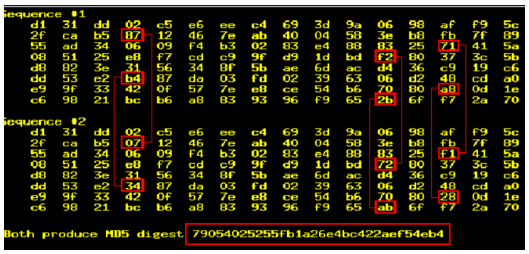

Figure 7 – Sample of a hash with collisions (MD5)

Figure #6 shows a comparison between some of the most common SHA algorithms.

Algorithm	Output size (bits)	Internal state size (bits)	Block size (bits)	Max message size (bits)	Word size (bits)	Rounds	Operations	Collision
SHA-0	160	160	512	$2^{64}-1$	32	80	+,and,or,xor,rotl	Yes
SHA-1	160	160	512	$2^{64}-1$	32	80	+,and,or,xor,rotl	2^63 attack
SHA-256/224	256/224	256	512	$2^{64}-1$	32	64	+,and,or,xor,shr, rotr	None yet
SHA-512/384	512/384	512	1024	$2^{128}-1$	64	80	+,and,or,xor,shr, rotr	None yet

Figure 6 – SHA comparison

Attack vector mitigations:

SRI only allow three hashing algorithms to be used:

SHA-256, SHA-384 and SHA-512.

If we take as example the weakest hashing algorithm (SHA-256), based that outputs 64 characters which can either be a lowercase letter or a number from 0-9. Which should mean that there are 64^36 distinct SHA-256 results.

For comparison, as of January 2015, Bitcoin was computing 300 quadrillion SHA-256 hashes per second. That's 300×1015 hashes per second.

If attempted to perform a collision attack on SHA-256 it will be needed to calculate up to 2128 hashes. At the rate Bitcoin is going, it would take:

2128/(300×1015·86400·365.25)≈ **3.6×1013 years**.

Therefore, the resource manipulation to create a hash collision coincidence is nowadays unfeasible.

Hash collision probabilistic are usually calculated using the Birthday Problem. Details of this theorem are out of the scope of this paper, but included as a [5]reference.

6.4. Attack vector #4: Dynamic code substitution and function overloading

This attack scenario attempts to modify the source code of the website in order to prevent loading an external resource that is being loaded using the secure SRI mechanism. But not also prevent running the script (which could potentially include additional security checks or features not interesting by an attacker) but also it has been attempted to replace the hash or URL pointing to the external resource.

Some different approaches will be tested in order to verify if SRI attributes can be tampered with or removed.

- Remove the entire "script" tag that loads the external resource and performs SRI verification
- Remove the "integrity" attribute, which as already seen in attack #1, in old browsers the script, even if tampered, will be executed.
- Tamper the "integrity" hash value to customize it to an arbitrary value.
- Overload native JavaScript functions to render unusable the SRI functionalities.

Our testing case scenario is a fictional website, where an attacker can introduce some JavaScript code using one of the most common web vulnerabilities, a Stored Cross Site Scripting [6](XSS).

The content of this sample website is not important at all, except the line of code that loads an external resource with SRI. For that purpose, we will use the generic source code already existing in the [7]official Mozilla documentation:

Sample SRI code from Mozilla

```
<script
src="https://example.com/example-
framework.js" integrity="sha384-
oqVuAfXRKap7fdgcCY5uykM6+R9GqQ
8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY
8wC" crossorigin="anonymous">
</script>
```

Before stepping deeper in this attack, a quick introduction of a XSS vulnerabilities is required in order to fully understand it.

[5] https://en.wikipedia.org/wiki/Birthday_problem

[6] https://www.sans.org/top25-software-errors

[7] https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity

A XSS is type of injection attack where an attacker can inject malicious source code (usually JavaScript or HTML) in order this to be executed in the user's web browser. Usually this attacks goes unnoticed to either, the web application/webserver (unless an Application Firewall - [8]WAF or an [9]IDS/IPS device is set between the communications to inspect and analyze the traffic in search of malicious patterns) and in user's end. Actually, the end user's web browser thinks that the execute script is legitimate and the malicious code can perform any kind of action, from stealing cookies or tokens, sensitive information retrieval to malware distribution.

[1] This first approach, it is assumed that we can load a previous custom JavaScript code somehow (for example an XSS) before the execution of the original source code.

Using the following script it was possible to eliminate existing elements (to be precise the third element).

Malicious script

```
<script>
  var myObject = {
   "FirstElement": "#1",
   "SecondElement": "#2",
   "ThirdElement": "#3"
};

delete myObject.ThirdElement;
console.log(myObject);
</script>
```

When the script gets executed, effectively the third element was eliminated and not shown in the response console as can be seen below:

Script execution result

```
{FirstElement: "#1", SecondElement: "#2"}

        FirstElement:"#1"
        SecondElement:"#2"
```

However, this sample does only work with known JavaScript elements such as variables or variable arguments, not in native attributes inside a *"<script>"* code.

[2] The second approach, instead of deleting specific JavaScript elements, it will be attempted to overload native JavaScript functions and methods with custom code in order to replace the standard way the code will be interpreted. That way, it is possible to change how does behave any JavaScript function for a custom code.

JavaScript does NOT natively support method overloading. So, if it sees/parses two or more functions with a same names it'll just consider the last defined function and overwrite the previous ones.

[8] https://en.wikipedia.org/wiki/Web_application_firewall
[9] https://en.wikipedia.org/wiki/Intrusion_detection_system

Method overload

```
var InitialMethod = function(){
console.log("Initial");
}

InitialMethod = (function(initial){

function extendedMethod(){
        initial();
        console.log("Extended!");
        }
        return extendedMethod;
})(InitialMethod);
```

The script above can successfully replace an existing method and change its behavior to perform any desired action. Mind that is required to analyze the source code of the web site or external resources, and search for the specific methods wanted to tamper with. That way it would be possible to alter the logic of any external resource method. This could definitely be useful for a typical hacker attack or post exploitation scenario, but these tests attempts to disable the SRI and this is not possible for the same reasons as the previous described test scenario.

[3] The third approach consists of breaking the standard flow of native JavaScript properties. This test is quite similar to scenario #1, but with a notable difference. Instead of directly deleting a property, which in some cases can introduce code compiling issues, the property will be unloaded from the web browser DOM just before the SRI script is executed, and once executed, we restore to the web browser's DOM the removed property.

```
        //let's suppose
example.framework.js asumes there
is an object called "Object". If we
manage to generate an exception at
runtime, the rest of the script will not
be processed because of the
exception.

        //Saving the object in Object2.
        var window.Object2 =
window.Object;

        //deleting the object
        delete window.Object;

        //The object is restoresd to the
DOM
        document.addEventListener("
DOMContentLoaded",
function(event) {
          window.Object =
window.Object2;
        });
```

This approach introduces an additional step, which makes things even harder for an attacker. As explained before, the element must be unloaded just before the SRI is interpreted, and re-enabled just before. This means that the attacker has to be able to inject code on two different places, making it quite harsh.

With this type of attack, it was also not possible to disable the SRI, but

turned out o be a very convenient way for post exploitation purposes.

Although it was not possible to disable the SRI, a collateral effect can be achieved using a slight variation in this code. If an external resource, a JavaScript file is securely loaded using SRI and we cannot avoid that, we can attempt to make the external resource to fail on execution time, so the script will not load at all.

When a JavaScript element has been unloaded from the DOM, if we manage to check whether the removed element is present (by using a comparative element as the source code below), the JavaScript will crash and not load the rest of the code.

Crash JavaScript execution
if(test.test){console.log("down");} console.log("up")

We can verify with the previous code snippet, that if "*test.test*" does not exists, the interpreter will print "down" and will never print "up" because causes a massive error at execution time, therefore, rendering the rest of the script useless.

Attack vector mitigations:

None. It was not possible to remove or replace SRI specific attributes, even though an XSS was a requirement to perform any of these attacks.

6.5. Attack vector #5: Browser extension dynamic content interception and injection

This attack consists of coding a specific web browser add-on that would emulate a CDN, but actually will intercept the web traffic, locate the resources we want to manipulate, and inject them again. All this made automatically using pattern matching.

To develop a custom plug-in for this purpose is out of the scope of this paper, but there is already some Open Source projects than can be taken as example to develop this.

https://github.com/Synzvato/decentraleyes

Attack vector mitigations:

It would be a requirement to have installed a specific rogue web browser add-on, so the first prevention mechanisms would be not to install suspicious add-ons, specially the ones not provided through the official Chrome/Mozilla stores, which are likely to contain untrusted code.

7. Conclusions.

There are many myths about evading SRI by dynamically substituting the content or by overloading functions in JavaScript, which at a first glance may seem doable, but all the approaches tested in this research failed or required the use of other severe web application vulnerabilities in order to break down the SRI.

SRI proved to be a pretty stubborn security feature against all the attacks performed to it, however, each website is coded in a different manner, and in specific scenarios, it can not be discarded to be bulletproof as by manipulating the page or an complex attack aided with web vulnerabilities, may lead to a weak spot to violate the SRI security mechanism.

The vulnerability mitigations stated in this document are an approach to increase the overall web application and JavaScript security against tested potential vulnerabilities.

A high level summary for each attack vector tested is described as follows:

#1: Running SRI protected tags without browsers support:

- To successfully exploit this attach, the client needs to run an old web browser with no support for SRI.
- In Google Chrome there is no way to deactivate the SRI, so no social engineering attacks are feasible with this web browser.
- Running an up to dated web browser always provides the highest level of security, not only for the latest patches but also for the top notch security features embedded in them.

#2: Wrong implementation of SRI without CORS enabled.

- CORS is mandatory to be enabled for SRI to work.
- No warnings are shown to the end user for errors related to CORS or wrong SRI hashes
- If for some reason the SRI fails to verify the hash, the external resource will not be executed, potentially causing issues in the website. Mozilla recommends to locally add the resource and call it when it fails to be loaded from an SRI call or the resource for some reason was not possible to be retrieved from the CDN.

#3: Hash collision.

- Hash collision attacks are unfeasible. Thousands of years are required to find a simple collision.

- Even though SHA-256 being the weakest algorithm, with the current processing capabilities, it is not possible to find a hash collision.

- Old vulnerable hashing algorithms are not used in the SRI specification.

#4: Dynamic code substitution and function overloading.

- An underlying severe vulnerability, such as a XSS is required to perform these attacks.
- At the very end it was not demonstrated a successful SRI remova.
- Alternative underlying JavaScript attacks were discovered, that could potentially be of use for a post exploitation web application attack

- Several ways to tamper with JavaScript were devised, but none could manipulate "script" tag attributes to remove the SRI properties.

#5: Browser extension dynamic content interception and injection.

- Theoretically, removing the SRI would be feasible by using this attack vector

- Installing a rogue web browser add-on is required to successfully exploit it

The following table shows for each attack vector covered in this paper, the Mitigation difficulty, which measures the difficulty of implementing new countermeasures to solve the vulnerability. The Exploitation like hood, which means how easy or difficult is for the average user to exploit the described vulnerability. Finally, the last row called Mitigation conclusion takes in account all the other rows and the mitigation vectors suggested to estimate the solutions effectiveness.

Attack vector	Mitigation difficulty	Exploitation like hood	Mitigation conclusion
#1: Running SRI protected tags without browsers support	**Low**	**High**	**Complete**
#2: Wrong implementation of SRI without CORS enabled	**Low**	**High**	**Complete**
#3: Hash collision	**Low**	**High**	**Complete**
#4: Dynamic code substitution and function overloading	**Low**	**High**	**Complete**
#5: Browser extension dynamic content interception and injection	**Low**	**Medium**	**Partial**

8. Further Research.

Further research shall be done to expand the attack vectors devised during this research. SRI is now in the mainstream release of all major web browsers, and everyday is gaining more adepts, but still is a feature fairly not well known.

An approach to further research on the first attack vector would consist of analyzing the source code of the web browser and find vulnerabilities in it that could lead to an exploitation of the SRI.

No additional actions for further research can be devised for the second attack vector, as the standard dictates the SRI requirements and effectively, the tested web browsers forced them.

The third attack vector also cannot be expanded, as the limitation is caused by the current computing capabilities, unable to find a mathematical collision in a reasonable period of time. Maybe in the future some weaknesses are found in one of the supported hashing algorithms that could lead to calculate rogue hashes.

The fourth attack vector looks promising, and further research on it is doable. JavaScript has a plethora of functions and other hacks are prone to be found.

Finally, the fifth attack vector, was not able to be tested with real source code, but from a theoretical point of view looks promising, as could potentially

intercept and replace arbitrary source code from a web application, including manipulating the SRI attributes.

9. Acknowledgments

The author would like to thank the reviewers for their helpful comments and advice. Finally, thanks to my family and friends on both continents who helped me in ways unknown to them.

10. Acronyms and definitions.

SRI: Subresource Integrity

W3C: World Wide Web Consortium

JS: JavaScript

CSS: Cascading Style Sheets

URI: Uniform Resource Indentifier

OS: Operating System

JQuery: Cross platform JavaScript library to simplify client side scripting of HTML

Node.JS: Open source cross platform JavaScript run-time environment that executes JavaScript code in server-side.

HTML: Hypertext Markup Language

CORS: Cross-Origin Resource Integrity

Hash: Mathematical encoding and unique alphanumeric value representation

CDN: Content Delivery Networks

URL: Uniform Resource Locator

HTTP: Hypertext Transfer Protocol

HTTPs: Hypertext Transfer Protocol Secure

CTF: Capture the Flag

SHA-512: International Organization for Standardization

MITM: Main in The Middle

CSP: Content Security Policy

IIS: Internet Information Services

CDN: Content Delivery Network

WAF: Web Application Firewall

IDS/IPS: Intrusion Detection System / Intrusion Prevention System

11. References

[1.] https://blog.lukaszolejnik.com/making-third-party-hosted-scripts-safer-with-subresource-integrity/

[2.] https://news.ycombinator.com/item?id=14041697

[3.] https://hacks.mozilla.org/2015/09/subresource-integrity-in-firefox-43/

[4.] http://www.elladodelmal.com/2016/03/subresource-integrity-sri-fortifica-tu.html

[5.] https://en.wikipedia.org/wiki/Subresource_Integrity

[6.] https://www.w3.org/

[7.] https://en.wikipedia.org/wiki/Secure_Hash_Algorithms

[8.] https://www.srihash.org

[9.] https://hacks.mozilla.org/2016/04/how-to-implement-sri-into-your-build-process/

[10.] https://es.wikipedia.org/wiki/Internet_Information_Services

[11.] https://docs.microsoft.com/en-us/iis/configuration/system.webserver/httpprotocol/customheaders/

[12.] https://en.wikipedia.org/wiki/SHA-2

[13.] https://www.w3.org/wiki/CORS_Enabled

[14.] https://stackoverflow.com/questions/1653308/access-control-allow-origin-multiple-origin-domains