My CDN Was Compromised!

In this lesson, we'll look at some measures you can take to ensure your users' protection in the case of a CDN compromise.

We'll cover the following

Introduction

Often times, web applications serve some of their content through a content delivery network (CDN), typically in the form of static assets like JavaScript or CSS files, while the main document is rendered by a webserver. This gives developers limited control over the static assets themselves, as they're usually uploaded to a third-party CDN (e.g., CloudFront, Google Cloud CDN, Akamai).

Now, suppose an attacker gained access to your login credentials on the CDN provider's portal and uploaded a modified version of your static assets, injecting malicious code. How could you prevent such a risk for your users?

Using integrity hashes to protect users

Browser vendors have a solution for you, called sub-resource integrity (SRI). SRI allows your main application to generate cryptographic hashes of your static files and tell the browser which file is mapped to what hash. When the browser downloads the static asset from the CDN, it will calculate the asset's hash on-the-fly and make sure that it matches the one provided in the main document. If the hashes don't match, the browser will simply refuse to execute or render the asset.

This is how you can include an asset with an *integrity hash* in your document.

```
...
<script
  src="https://my.cdn.com/asset.js"
  integrity="sha256-Y34u3VVVcO2pZtmdTnfZ+7OquEpJj/VawhuWPB4Fwq3ftcFc0gceft1HNZ
```

```
14eUHT"


></script>
...
```

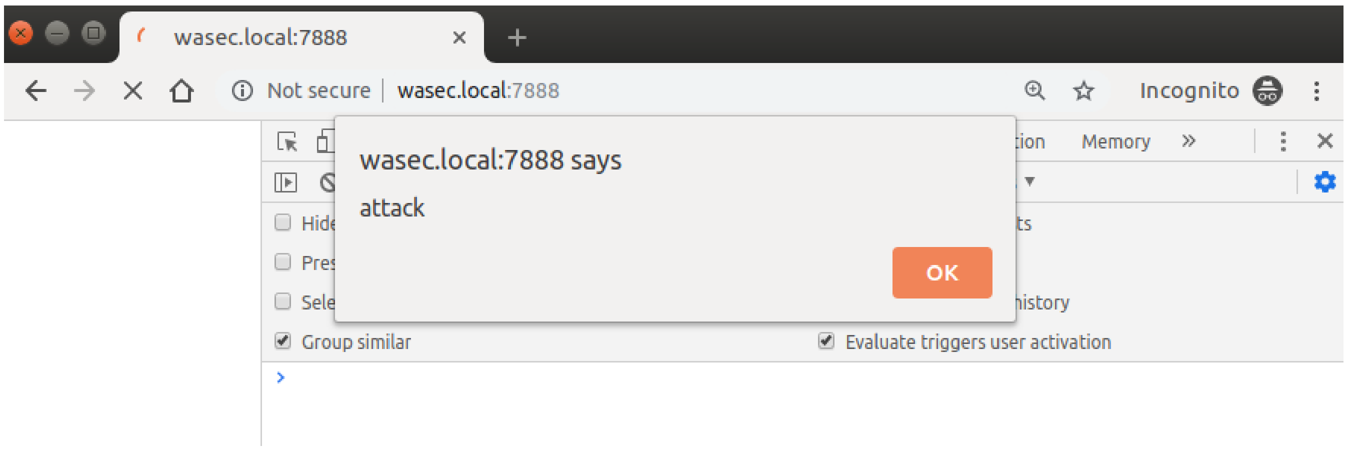
The *integrity hash* can be computed with a simple command.

```
cat $ASSET_FILE_NAME | openssl dgst -sha384 -binary | openssl base64 -A
```

A working example can be found below; after you've ran the web server by clicking the run button, you can visit https://x6jr4kg.educative.run by replacing the educative.run link with the one generated in your app below to see the SRI in action.

```
var qs = require('querystring')
var url = require('url')
var fs = require('fs')

let attack_integrity = "sha256-AN_INTEGRITY_THAT_DOESNT_MATCH"
let asset_integrity = "sha256-Z67eKNNu3z1gzgMcRCqRQo4f4gtT6pM0y6BHe/r5OGY="

require('http').createServer((req, res) => {
  let query = qs.parse(url.parse(req.url).query)

  if (req.url == "/attack.js") {
    res.end(`alert("attack")`)
  }

  if (req.url == "/asset.js") {
    res.end(`alert("valid asset")`)
  }

  let asset_integrity_check = ''
  let attack_integrity_check = ''

  if (query.sri) {
    asset_integrity_check = `integrity="${asset_integrity}"`
    attack_integrity_check = `integrity="${attack_integrity}"`
  }

  res.writeHead(200, {})
  res.end(`<html><body><script src="/asset.js" ${asset_integrity_check}></script><script src="/att
}).listen(7888)
```

Two scripts are included in the page you're opening, one that's *legitimate* and one to simulate an attacker's attempt to inject malicious code in one of your assets. As you can see, the attacker's attempt proceeds without any issue when SRI is turned off.

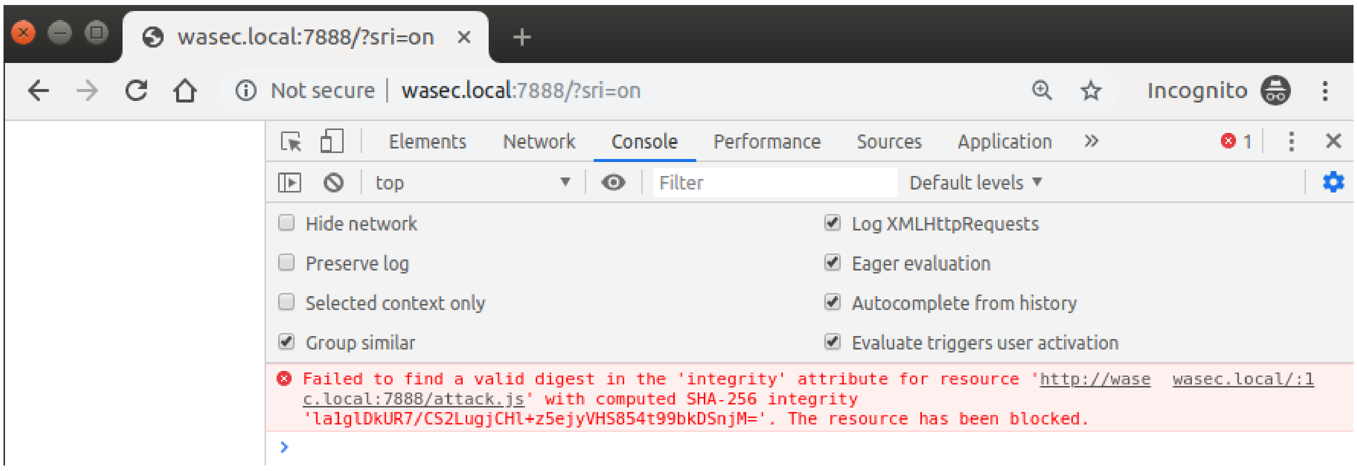

wasec.local:7888
Not secure | wasec.local:7888
Incognito
Memory

```
wasec.local:7888 says
attack
```

OK
Group similar
Evaluate triggers user activation

Attack successful with SRII turned off

By visiting https://x6jr4kg.educative.run/?sri=on we get a completely different outcome, as the browser realizes that there's a script that doesn't seem to be genuine, and doesn't let it execute:



```
Failed to find a valid digest in the 'integrity' attribute for resource 'http://wase  wasec.local/:1
c.local:7888/attack.js' with computed SHA-256 integrity
'lalglDkUR7/CS2LugjCHl+z5ejyVHS854t99bkDSnjM='. The resource has been blocked.
```

Attack not executed with SRI turned on

This is what our HTML looks like when SRI is turned on.

```html
<html>
<body>
    <script src="/asset.js" integrity="sha256-Z67eKNNu3z1gzgMcRCqRQo4f4gtT6pM0y6BHe/r5OGY="></script>
    <script src="/attack.js" integrity="sha256-AN_INTEGRITY_THAT_DOESNT_MATCH"></script>
</body>
</html>
```

A clever trick from browser vendors and users are secured should anything happen to the files hosted on a separate CDN. Clearly this doesn't prevent an attacker from attacking your main resource, but it’s an additional layer of security you couldn’t count on until a few years ago.

In the next lesson, we’ll learn what EV certificates are and why they don’t work.