

Lecture 27

CSP nonce and strict-dynamic

ECE 422: Reliable and Secure Systems Design



Instructor: An Ran Chen
Term: 2024 Winter

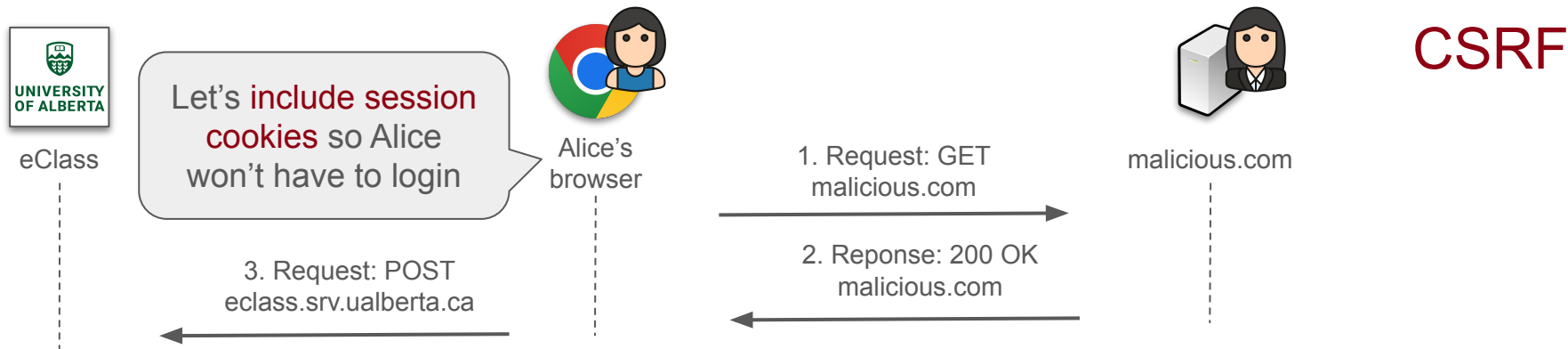
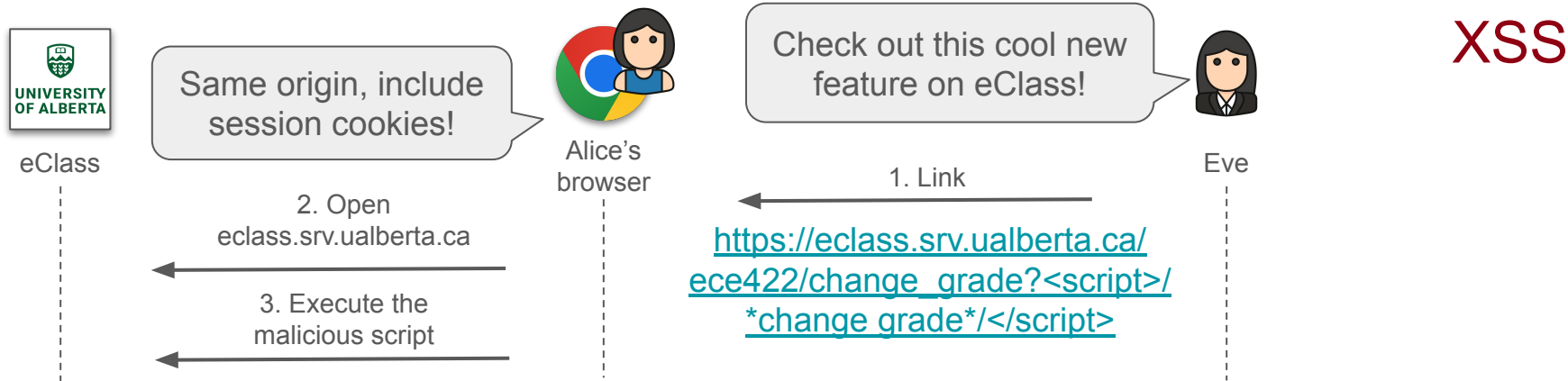
Schedule for today

- Key concepts from last classes
- Content Security Policy (CSP)
 - “CSP is Dead” paper by Google in 2016
 - Additional CSP protections in two-fold
 - CSP nonces
 - CSP strict-dynamic
 - Read-only mode
 - CSP deployment
- Final take-homes on CSP

Key concepts from last classes

- Session fixation attack
 - Session fixation prevention: Signing cookies
- Cross-Site Request Forgery (CSRF)
 - Access to user session by ambient authority
 - CSRF prevention: Same Origin Policy (SOP)
- Cross Site Scripting (XSS)
 - Reflected XSS
 - Stored XSS
 - XSS prevention
 - HttpOnly and HTML Escaping
 - Content Security Policy (CSP)

CSRF vs XSS



CSP vs SOP

CSP



Eve



With or without
database



eClass

1. Submit quiz

Question 1 answer:
<script>/*change grade*/</script>

Check Eve's
answer.

2. Request get
eclass.srv.ualberta.ca

3. Respond with Eve's
malicious script embedded



Alice's
browser

4. CSP prevents
the malicious script

SOP



eClass

Different origin, let's
not share the
session cookies!

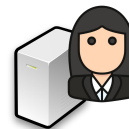


Alice's
browser

1. Request: GET
malicious.com

2. Reponse: 200 OK
malicious.com
with malicious script embedded

3. Request: POST
eclass.srv.ualberta.ca



malicious.com

Schedule for today

- Key concepts from last classes
- Content Security Policy (CSP)
 - “CSP is Dead” paper by Google in 2016
 - Additional CSP protections in two-fold
 - CSP nonces
 - CSP strict-dynamic
 - Read-only mode
 - CSP deployment
- Final take-homes on CSP

Key concepts into practice

Implementing CSP on [UAlberta website](#):

```
<script async  
src='https://www.google-analytics.com/analytics.js'></script>
```

```
/*create /script.js*/
```

```
<script>  
    window.GoogleAnalyticsObject = 'ga'  
    function ga () { window.ga.q.push(arguments) }  
    window.ga.q = window.ga.q || []  
    window.ga.l = Date.now()  
    window.ga('create', 'UA-XXXXXXX-XX', 'auto')  
    window.ga('send', 'pageview')  
</script>
```

Content-Security-Policy:

```
default-src: 'self';  
script-src: 'self' https://www.google-analytics.com
```

What problem can happen?

- analytics.js can call other scripts that do not originate from google-analytics.com
- Those scripts with different origin will be blocked!
- ... get worst when we have nested scripts inside nested scripts inside ...

Challenge: Nested scripts

Problem in CSP's original design: How do we make sure that CSP is respected while new scripts can be executed from trusted sources?

- Intuition: **propagate trust** from the initial script to any nested scripts

This can be achieved with more advanced CSP such as strict-dynamic

- Explicitly trust a script with a *nonce* or a *hash*, which shall be propagated to all the scripts loaded by that root script.
 - *Nonce* = similar to a secure random token
 - No longer need whitelisting resources

CSP Is Dead, Long Live CSP

In 2016, Google published a paper titled “[CSP Is Dead, Long Live CSP](#)” at CCS conference ([presentation video](#) available):

CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy

Lukas Weichselbaum
Google Inc.
lwe@google.com

Michele Spagnuolo
Google Inc.
mikispag@google.com

Sebastian Lekies
Google Inc.
slekies@google.com

Artur Janc
Google Inc.
aaj@google.com

ABSTRACT

Content Security Policy is a web platform mechanism designed to mitigate cross-site scripting (XSS), the top security vulnerability in modern web applications [24]. In this paper,

1. INTRODUCTION

Cross-site scripting – the ability to inject attacker-controlled scripts into the context of a web application – is arguably the most notorious web vulnerability. Since the

- 94.68% of CSP that attempt to limit script execution are ineffective
- 99.34% of hosts use policies that offer no benefit against XSS

Solution: The use of **strict-dynamic** as a value in script-src

Problems in CSP

CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy

Google's findings highlight two problem statements: Majority of the studied CSP are **time-consuming to deploy** and **difficult to operate and maintain**.

Problem 1: Time-consuming

- Solution? Leverage **nonces** to improve the deployment of CSP

Problem 2: Difficult to operate and maintain

- Solution? Use **strict-dynamic** to improve the usefulness of CSP

Problem 1: Time-consuming

Fact: Inline scripts are widely used by web developers, because of its advantages:

- Easy to use
 - E.g., Embedded into web pages, avoiding additional JavaScript files
- Easy to access
 - E.g., JavaScript executed within the context of the HTML file
- Still bad practice (from front-end engineers point of view)

However, they also make CSP deployment tedious and time-consuming:

- Migrating from an existing site is difficult: **a lot of work to remove**
- **Performance optimization required** when deploying: inline scripts have better performance, fewer HTTP requests

CSP nonces

A **CSP nonce** is a **randomly** generated token that is used **exactly one time**.

- Generate random numbers to give allow specific scripts when CSP is enabled

Why? Analogous as session IDs but for scripts

- As long as nonce is valid, trusted scripts can be loaded and executed
- Generated on every page load, so attackers cannot reuse the same token

How does it work?

- Generate nonce for every request to web server
- Declare nonce in the CSP header script-src
- Add it to scripts tags

Example of CSP nonces

Inline script without CSP nonce

```
Content-Security-Policy:  
script-src: 'self'
```

```
<script>  
... </script>
```

Inline script with CSP nonce

```
Content-Security-Policy:  
script-src: 'self' 'nonce-rAnd0m'
```

```
<script nonce="rAnd0m">  
... </script>
```

- Without the CSP nonce attribute, the script will not execute
- nonce attribute informs the browser that the script is safe:
 - If and only if once attribute value matches the one in the Content-Security-Policy header

Example of CSP nonces

Inline script without CSP nonce

```
Content-Security-Policy:  
script-src: 'self'
```

```
<script>  
... </script>
```

Inline script with CSP nonce

```
Content-Security-Policy:  
script-src: 'self' 'nonce-rAnd0m'
```

```
<script nonce="rAnd0m">  
... </script>
```

Why a CSP nonce for every inline script block?

- Browser does not distinguish between developer-written and injected JavaScript

Another example of CSP nonces

External script without CSP nonce

Content-Security-Policy:

```
script-src: 'self' https://*.google-analytics.com
```

```
<script  
src='https://www.google-analytics.com/  
analytics.js'></script>
```

External script with CSP nonce

Content-Security-Policy:

```
script-src: 'self' 'nonce-rAnd0m'
```

```
<script  
src='https://www.google-analytics.com/  
analytics.js' nonce='rAnd0m'></script>
```

- Same idea applies to external scripts
- Adding the nonce attribute to the script tag provides another solution to add <https://www.google-analytics.com/analytics.js> to CSP

Questions on CSP nonce



Given Content-Security-Policy: script-src 'nonce-rAnd0m'

- Is `<script src='https://trusted.com/script.js'></script>` allowed?
☐ Yes ☐ No
- Is `<script src='https://trusted.com/script.js' nonce='rAnd0m'></script>` allowed?
☐ Yes ☐ No
- Is `<svg nonce='rAnd0m' onload='alert(document.cookie)'>` allowed?
☐ Yes ☐ No

Questions on CSP nonce



Given Content-Security-Policy: script-src 'nonce-rAnd0m'

- Is `<script src='https://trusted.com/script.js'></script>` allowed?
☐ Yes ☐ No, inline scripts are prevented
- Is `<script src='https://trusted.com/script.js' nonce='rAnd0m'></script>` allowed?
☐ Yes, CSP nonce allows the execution ☐ No
- Is `<svg nonce='rAnd0m' onload='alert(document.cookie)'>` allowed?
☐ Yes ☐ No, nonce can only allow inline scripts

Questions on CSP nonce



Given Content-Security-Policy: script-src 'nonce-rAnd0m'

- Is `<script src='https://trusted.com/script.js'></script>` allowed?

☐ Yes

☒ No, because inline scripts are prevented

Why can't the attacker figure out the nonce?

- Nonce changes on each page load, so it is unpredictable

Security issues with nonces

For security concerns, nonce should be have the following attributes:

- Hidden
 - Nonces should only be used in script tags, and nowhere else
- Unpredictable
 - Assign a 128 bit nonce (i.e., same length as a session identifier)
- Unique for each page reload
 - Avoid nonce reuse or data exfiltration

Problem 2: Difficult to operate and maintain

Fact: CSP needs to maintain a whitelist of domains.

- For example, to include google-analytics:

Content-Security-Policy:

script-src: https://*.googletagmanager.com

img-src: https://*.google-analytics.com https://*.analytics.google.com https://*.googletagmanager.com
https://*.g.doubleclick.net https://*.google.com https://*.google.

connect-src: https://*.google-analytics.com https://*.analytics.google.com
https://*.googletagmanager.com https://*.g.doubleclick.net https://*.google.com

However, this can be difficult to operate and maintain:

- Operational standpoint: third-party scripts can add other scripts
 - E.g., nested scripts inside nested scripts inside ...
- Maintenance standpoint: Domains may change, constant updates required

strict-dynamic

strict-dynamic is a possible value inside script-src directive

- Used in combination with nonces

Why? **Trust propagation** to all the scripts loaded by the root script

- Allows any script to be included by any script with nonce attribute
- Solution to nested scripts inside nested scripts inside ...

How does it work?

- Declare strict-dynamic in script-src with nonce as part of the CSP header

Example of strict-dynamic

Given a script `/script-loader.js` that loads other scripts:

Content-Security-Policy:

```
script-src: 'self' 'nonce-rAnd0m' 'strict-dynamic'
```

```
<script src='/script-loader.js'  
nonce='rAnd0m'> </script>
```

- 'strict-dynamic' allows `/script-loader.js` to load additional scripts
- No need to specify whitelist in CSP headers anymore
- As long as attackers cannot figure nonce, strict-dynamic will provide security

Schedule for today

- Key concepts from last classes
- Content Security Policy (CSP)
 - “CSP is Dead” paper by Google in 2016
 - Additional CSP protections in two-fold
 - CSP nonces
 - CSP strict-dynamic
 - Read-only mode
 - CSP deployment
- Final take-homes on CSP

Content-Security-Policy-Report-Only

Content-Security-Policy-Report-Only allows developers to experiment with policies by monitoring (but not enforcing) their effects.

- Server sends Content-Security-Policy-Report-Only header instead of Content-Security-Policy
- Violation of policies presented in a report

Why? To test a policy without breaking the application

- Problem with testing CSP: If we miss something (e.g., attribute events), the website will break → unhappy customers
- Report-only mode offer a solution to this problem

Content-Security-Policy-Report-Only

How does it work? Offering an iterative process for improving CSP

- Observe how the site behaves and watch for CSP violations
- Improve the desired policy by implementing new CSP

Example: Do not enforce CSP, but violations are reported to a provided URL

Content-Security-Policy-Report-Only:

```
default-src 'self';  
report-to https://example.com/report
```

Example of Content-Security-Policy-Report-Only



Content-Security-Policy-Report-Only:

```
default-src 'none';  
  
style-src *.example.com;  
  
report-to /reports
```

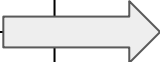
```
<!doctype html>  
<html lang="en-US">  
  <head>  
    <meta charset="UTF-8" />  
    <title>Sign Up</title>  
    <link rel="stylesheet" href="css/style.css" />  
  </head>  
  <body>  
    Welcome to eClass.com  
  </body>  
</html>
```

Can you spot the violation?

Example of Content-Security-Policy-Report-Only

Content-Security-Policy-Report-Only:

```
default-src 'none';  
style-src *.example.com;  
report-to /reports
```



```
<!doctype html>  
<html lang="en-US">  
  <head>  
    <meta charset="UTF-8" />  
    <title>Sign Up</title>  
    <link rel="stylesheet" href="css/style.css" />  
  </head>  
  <body>  
    Welcome to eClass.com  
  </body>  
</html>
```

```
{"csp-report": {  
  "blocked-uri": "http://example.com/css/style.css",  
  "disposition": "report",  
  "document-uri": "http://example.com/signup.html",  
  "effective-directive": "style-src-elem",  
  "original-policy": "default-src 'none'; style-src  
cdn.example.com; report-to /reports",  
  "referrer": "",  
  "status-code": 200,  
}
```

- **blocked-uri**: URI of the resource that was blocked from loading by CSP
- **effective-directive**: directive whose enforcement caused the violation.

Schedule for today

- Key concepts from last classes
- Content Security Policy (CSP)
 - “CSP is Dead” paper by Google in 2016
 - Additional CSP protections in two-fold
 - CSP nonces and hashes
 - Strict-dynamic
 - Read-only mode
 - CSP deployment

CSP deployment

Deploying CSP in 5 steps:

- Step 1: Convert attributes into inline
- Step 2: Generate nonce
- Step 3: Add nonce attribute
- Step 4: Test CSP in report only mode
- Step 5: Remove report only from CSP header

CSP deployment

Step 1: Convert attributes into inline

- No longer need to change inline scripts, but event handling attributes are still disabled by CSP
- Most error-prone step, still require manual work
 - No automated tool available, great idea for a product
- Example: Replacing the onClick JavaScript event with an event listener

```
<button onClick='doSomething()' />  
<script>  
    doSomething() = () => { ... };  
</script>
```



```
<button id='foo' />  
<script>  
    document.getElementById('foo')  
        .addEventListener('click, () => { ... }');  
</script>
```

CSP deployment

Step 2: Generate nonce

- Generate random 128 bit nonces that is sent to JavaScript templates
- Open source project are also available:
 - [Django-csp](#) from Mozilla
- Example: `_make_nonce(self, request, length=16)`

```
def _make_nonce(self, request, length=16):  
    if not getattr(request, '_csp_nonce', None):  
        request._csp_nonce = get_random_string(length)  
    return request._csp_nonce
```

CSP deployment

Step 3: Add nonce attribute

```
<script>  
    doSomething() = () => { ... };  
</script>
```



```
<script nonce='{{csp_nonce}}'>  
    doSomething() = () => { ... };  
</script>
```


CSP deployment

Step 4: Test CSP in report only mode

- Report-only mode to check policy violation reports
- Policy violation reports often tend to be very noisy, solve large numbers of failures

Content-Security-Policy-Report-Only:

```
script-src 'nonce-{random}' 'strict-dynamic'  
report-uri https://example.com/report
```

Step 5: Remove report only from CSP header

Content-Security-Policy:

...

CSP Level 3

[The World Wide Web Consortium \(W3C\)](#) posted a Working Draft of [CSP Level 3](#) in February 21, 2024:

- strict-dynamic is part of Level 3

The last version [CSP Level 2](#) was posted in December 15, 2016:

- CSP nonces and hashes are part of Level 2
- Content-Security-Policy-Report-Only was also new to Level 2

Content Security Policy Level 3

[W3C Working Draft, 21 February 2024](#)



▼ More details about this document

Final take-homes on CSP

- XSS are still relevant in real-world web applications
- XSS: convert user's data into code
- Always sanitize user's data: Escaping the input based on the context
- Use CSP to prevent almost all XSS attacks
- CSP nonces and strict-dynamic make it easier to implement CSP
- CSP report-only mode makes it easier to test CSP