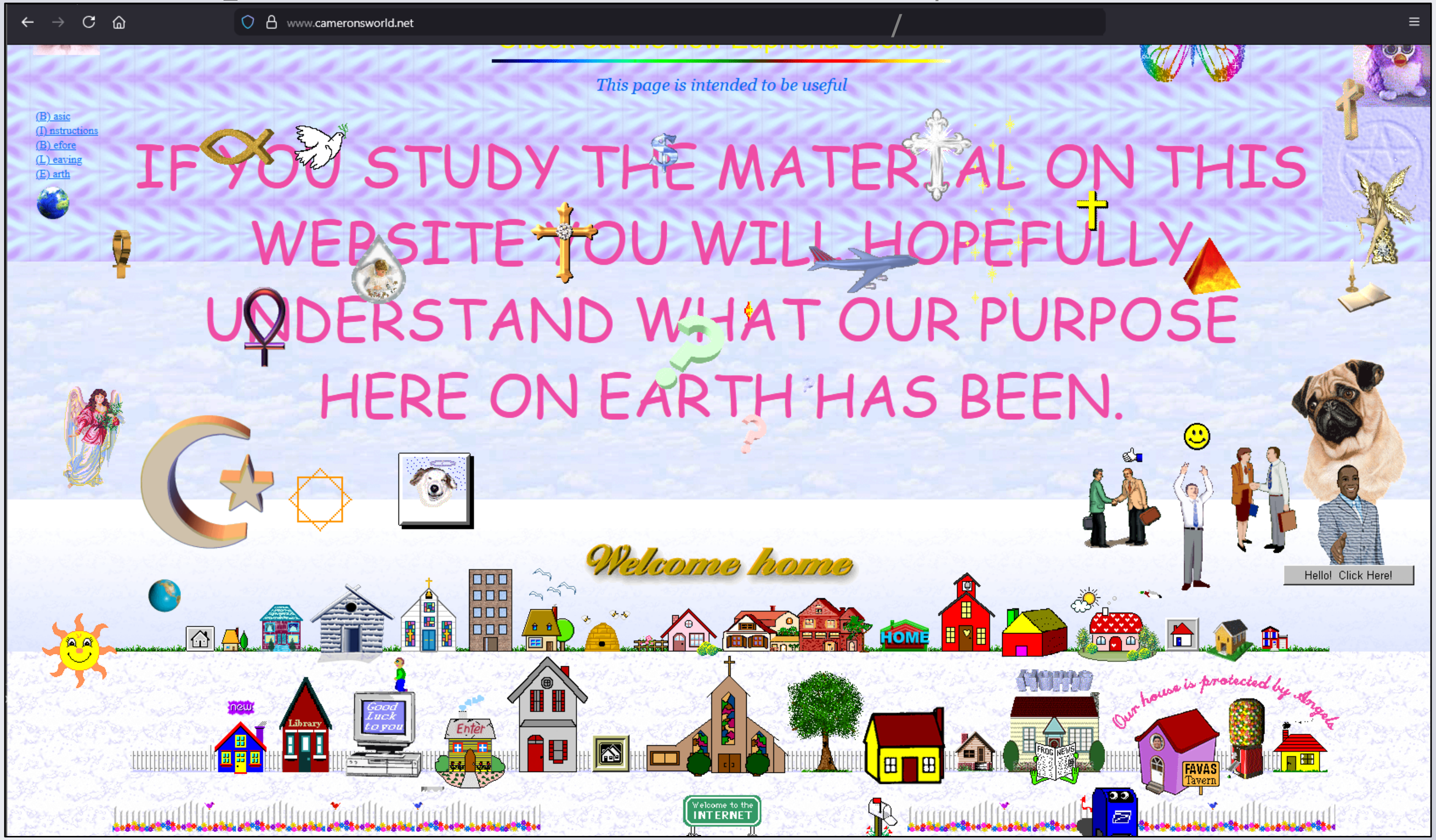
Cross-App Security V

Subresources

The Composable Web

https://www.cameronsworld.net


www.cameronsworld.net
This page is intended to be useful
(B) asic
(I) nstructions
(B) efore
(L) eaving
(E) arth
IF YOU STUDY THE MATERIAL ON THIS WEBSITE YOU WILL HOPEFULLY UNDERSTAND WHAT OUR PURPOSE HERE ON EARTH HAS BEEN.
Welcome home
Hello! Click Here!
new
Library
Good Luck to you
Enter
HOME
Our house is protected by Angels
FAVAS Tavern
Welcome to the INTERNET

Subresource

A resource that a document fetches to complement its primary content, such as images, scripts, or stylesheets.

- LLM citing non-existent definition from the W3C SRI Spec

Subresource as a Destination

- Remember *sec-fetch-dest?*
- According to Fetch API, a *subresource request* has one of the following destinations:
  - The empty string
  - One of: "audio", "audioworklet", "font", "image", "json", "manifest", "paintworklet", "script", "style", "track", "video", "xslt“
- By comparison, the destination for a *non-subresource request* is one of:
  - "document", "embed", "frame", "iframe", "object", "report", "serviceworker", "sharedworker", or "worker"

Subresource Risk

- The composable web is designed to be permissive with respect to subresource requests and embedding
- These represent general exceptions to Same-Origin Policy
- In general, there are two classes of risk:
    1. Risk posed to applications embedding malicious resources
    2. Risk posed to applications embedded by malicious apps

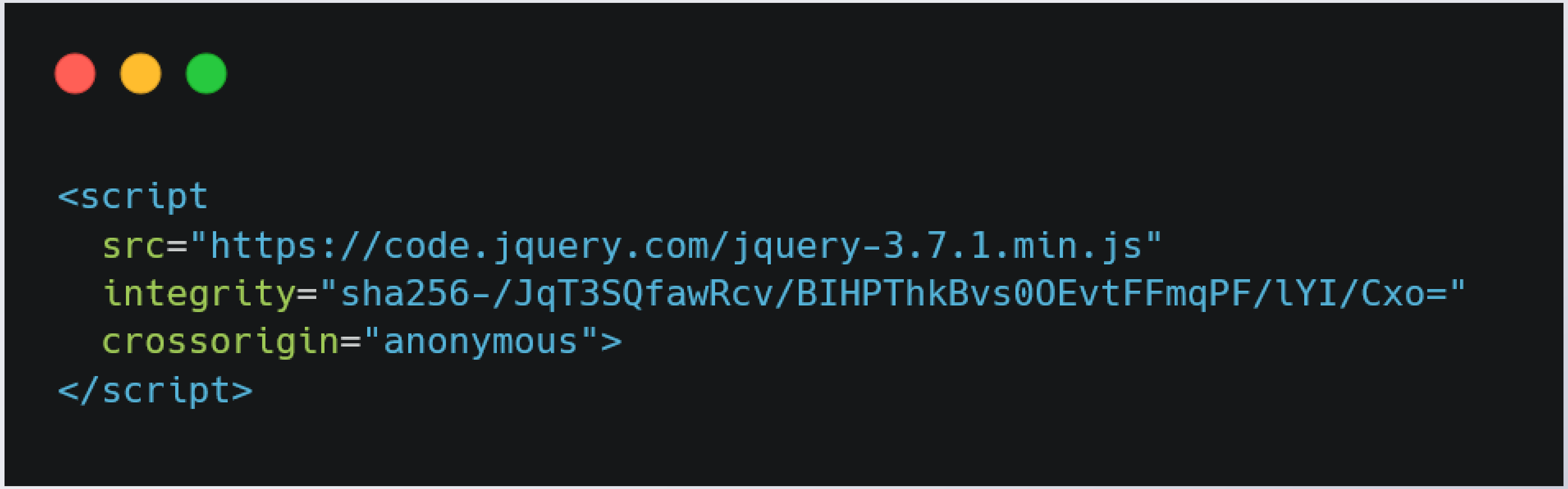
Note: client-side injection attacks (XSS, HTMLi) are attempts to embed malicious resources in an application and will also be examined to some extent in this lesson

Simple Case: Resource Hijacking

- Somewhat common *Supply Chain Attack*
- Scenario:
  - An application embeds a 3rd party script: <*script src*="*https://some-site...*
  - An attacker modifies the script content and injects malicious content
    - How? Compromise of code repo, CDN compromise, web cache poisoning, and so on
  - All applications that embed the script now run the malicious content IN THE CONTEXT OF THEIR OWN ORIGIN!
- This is a common pattern in card-skimming attacks (Magecart)
  - Annoyingly, "card-skimming" attacks and reporting on them almost always fail to indicate the root cause that permitted the attack to happen

Subresource Integrity (SRI)

- A mechanism to validate the integrity by hash comparison performed by the browser
- Requires:
  - Static resource
  - Addition of *integrity* attribute
  - Resource support for CORS



```html
<script
  src="https://code.jquery.com/jquery-3.7.1.min.js"
  integrity="sha256-/JqT3SQfawRcv/BIHPThkBvs0OEvtFFmqPF/lYI/Cxo="
  crossorigin="anonymous">
</script>
```

SRI: Challenges

- Assumes the computed-hash version is safe… is it?
- May impact dev velocity and conflict with other security practices:
  - JS resources must be static (commitment to version)
  - Are developers really going to fully vet a third-party JS library every update?
  - What takes priority: integrity of trusted JS, or remediated vulnerable JS library versions?
  - Attempts to automate can undermine security
- Can only apply to some types of subresources:
  - JS, CSS, Preload (JS/CSS), fetch() calls

DBG App Test Finding
Active Content Included Without Content Integrity Protection

DBG App Test Finding
Google Tag Manager Use

Protecting the Integrity of Images

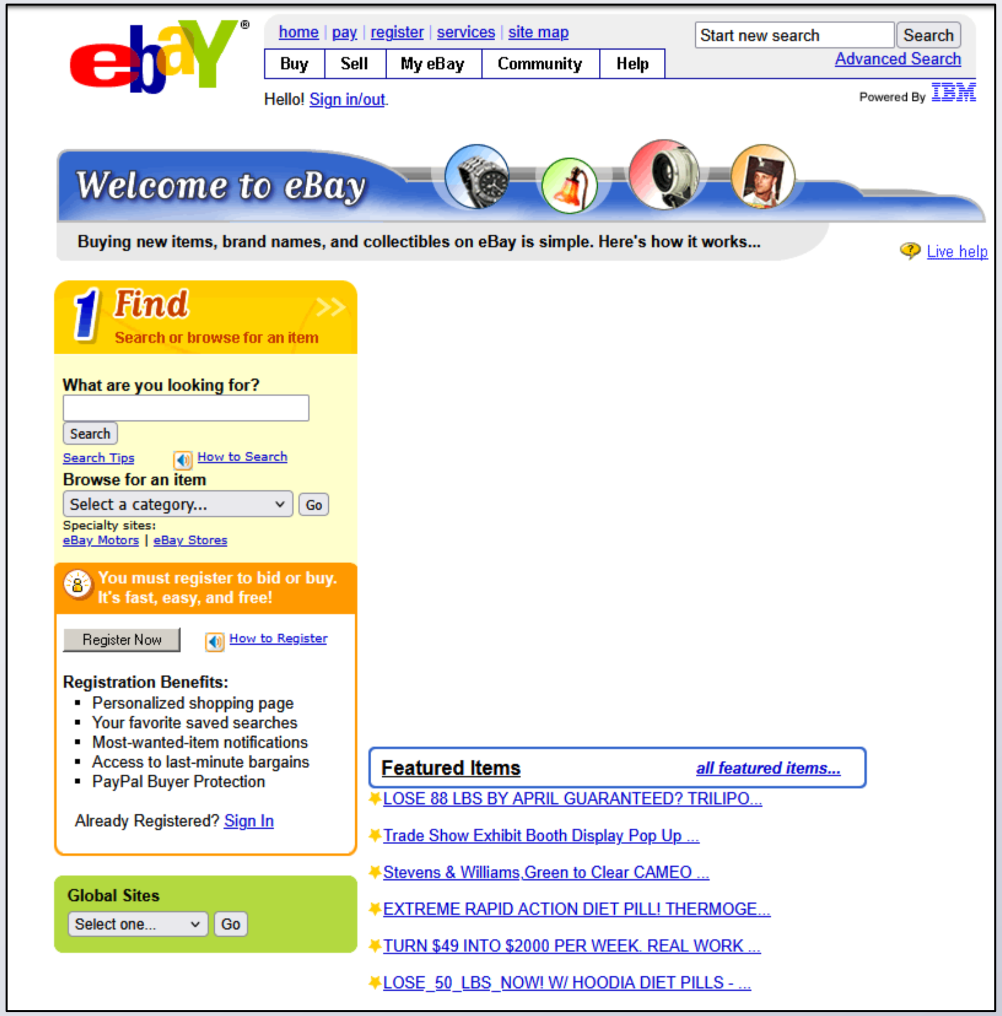
How do I protect my beautiful website?

Example:

- Website Origin: http://www.ebay.com
- Images: http://pics.ebaystatic.com
  - Implicit Trust!

To avoid defacement, the website depends on the security of the image host.

Alternative: self-host



home | pay | register | services | site map

Start new search Search

Advanced Search

Buy | Sell | My eBay | Community | Help

Hello! Sign in/out.

Powered By IBM

Welcome to eBay

Buying new items, brand names, and collectibles on eBay is simple. Here's how it works...

Live help

1 Find

Search or browse for an item

What are you looking for?

Search

Search Tips How to Search

Browse for an item

Select a category... Go

Specialty sites:
eBay Motors | eBay Stores

You must register to bid or buy. It's fast, easy, and free!

Register Now How to Register

Registration Benefits:
- Personalized shopping page
- Your favorite saved searches
- Most-wanted-item notifications
- Access to last-minute bargains
- PayPal Buyer Protection

Already Registered? Sign In

Global Sites

Select one... Go

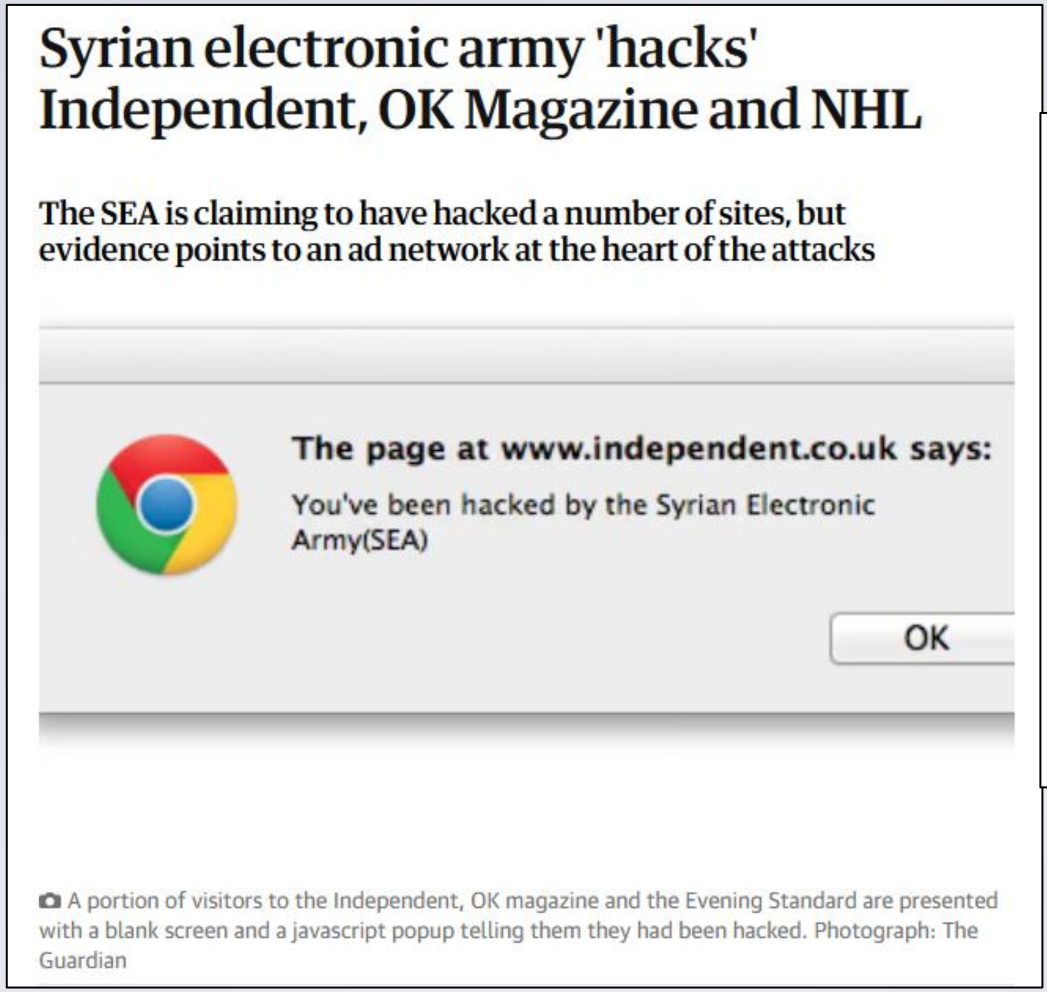
Featured Items all featured items...

- LOSE 88 LBS BY APRIL GUARANTEED? TRILIPO...
- Trade Show Exhibit Booth Display Pop Up ...
- Stevens & Williams,Green to Clear CAMEO ...
- EXTREME RAPID ACTION DIET PILL! THERMOGE...
- TURN $49 INTO $2000 PER WEEK. REAL WORK ...
- LOSE 50 LBS NOW! W/ HOODIA DIET PILLS - ...

DBG App Test Finding
External Resource Inclusion without Validation

https://web.archive.org/web/20050228091153/http://www.ebay.com/

Defacement!



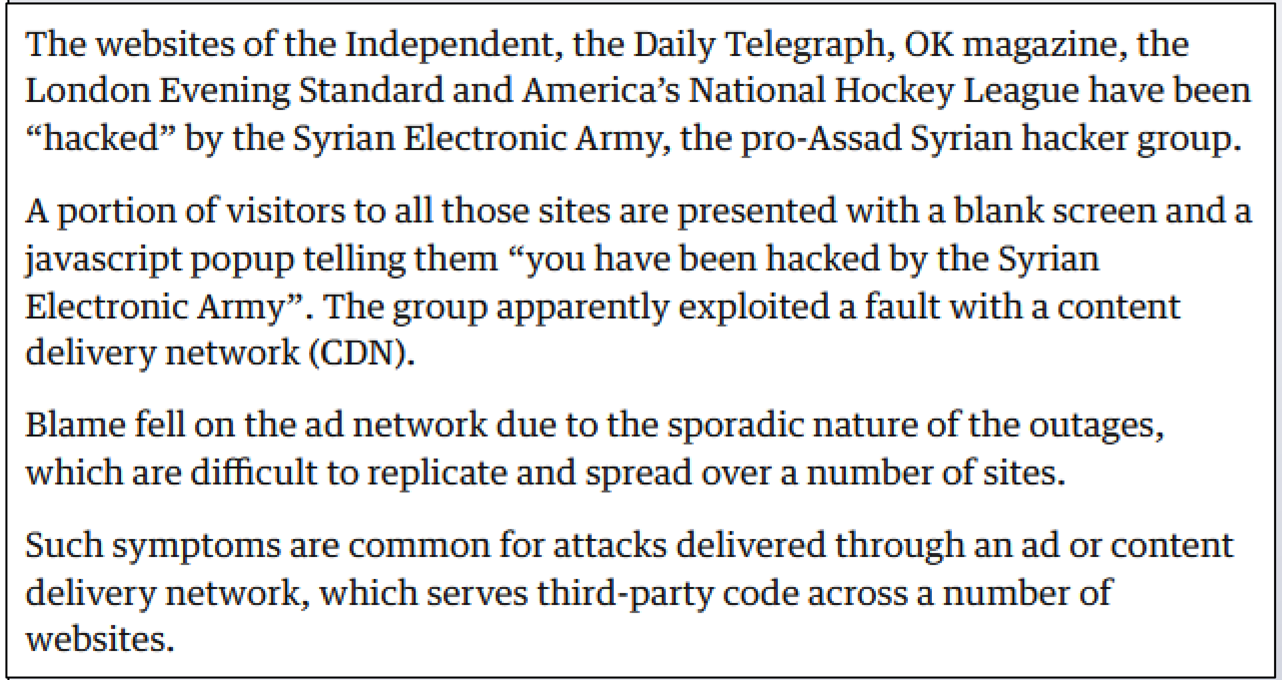
Syrian electronic army 'hacks' Independent, OK Magazine and NHL

The SEA is claiming to have hacked a number of sites, but evidence points to an ad network at the heart of the attacks

The page at www.independent.co.uk says:

You've been hacked by the Syrian Electronic Army(SEA)

A portion of visitors to the Independent, OK magazine and the Evening Standard are presented with a blank screen and a javascript popup telling them they had been hacked. Photograph: The Guardian



The websites of the Independent, the Daily Telegraph, OK magazine, the London Evening Standard and America's National Hockey League have been "hacked" by the Syrian Electronic Army, the pro-Assad Syrian hacker group.

A portion of visitors to all those sites are presented with a blank screen and a javascript popup telling them "you have been hacked by the Syrian Electronic Army". The group apparently exploited a fault with a content delivery network (CDN).

Blame fell on the ad network due to the sporadic nature of the outages, which are difficult to replicate and spread over a number of sites.

Such symptoms are common for attacks delivered through an ad or content delivery network, which serves third-party code across a number of websites.

https://www.theguardian.com/technology/2014/nov/27/syrian-electronic-army-hacks-independent-ok-magazine-and-nhl

Unsolved Problem: JS Execution Scope

- Embedded script content typically runs with full “privileges” to do anything within your app’s execution context
- Some strategies exist to isolate JS, but often require re-architecture:
  - Isolate JS in *iframe* and apply *sandbox* restrictions as appropriate
  - Isolate JS in Web Workers , restricting access to the DOM
    - Similar approach, but for performance: https://github.com/QwikDev/partytown
  - ECMAScript *Realms* define execution spaces and there are ongoing efforts to create standards for managing isolated *Realms*
    - *iframes* and Web Workers create their own *Realms*!
- Best we can do typically: restrict the source of scripts we run and ensure their integrity.

Content Security Policy (CSP)

- CSP is an HTTP response header that implements restrictions for:
  - Embedded content (subresources)
  - Inline content
  - Embedding documents
  - Outgoing actions
- Primary goal: mitigate XSS vulnerabilities
- Other Capabilities
  - Limit impact of client-side injection (restricting outgoing actions)
    - Community consensus: there will always be a way to bypass; focus on preventing injection

DBG App Test Finding
Missing Content Security Policy

CSP: Just the Basics

Directives restrict sources by type

- For example: *script-src*, *style-src, img-src*

Typical values include:

- *'none'*: Block this type of resource entirely
- *'self'*: Permit loading from Same-Origin
- <*URL List*>: Define allow-list of external Origins

Additional unique values:

- *'unsafe-inline'*: Can permit inline (via <*script*>*...*</*script*>, event handlers, and JS URIs) scripts and styles
- *'unsafe-eval'*: Permits additional dangerous JS methods (like *eval()*)
- *'wasm-unsafe-eval'*: Permits Web Assembly (WASM)

CSP: Special Directives

default-src

This directive acts as a "fallback" configuration

base-uri

For relative URLs (ex: <*script src*=*"js/app.js"*>), this can redefine the base URL where they are resolved to

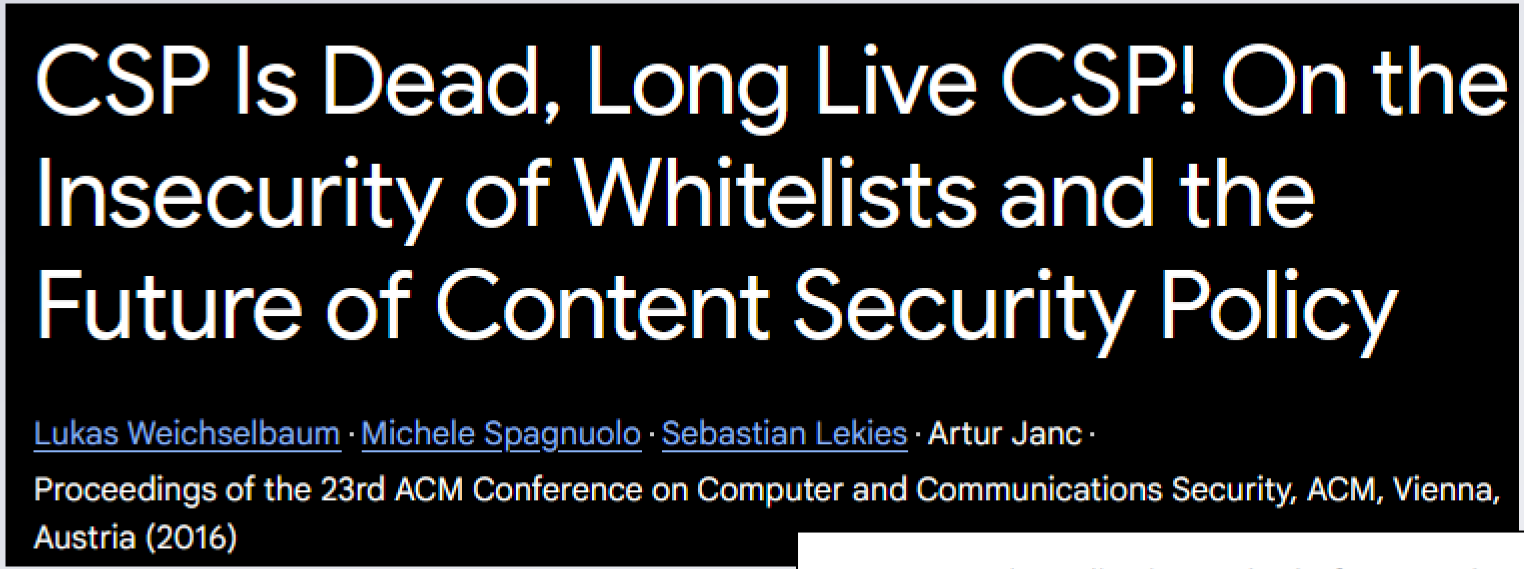
object-src

Restricts potentially risky resources (within <*object*> and <*embed*>)

sandbox

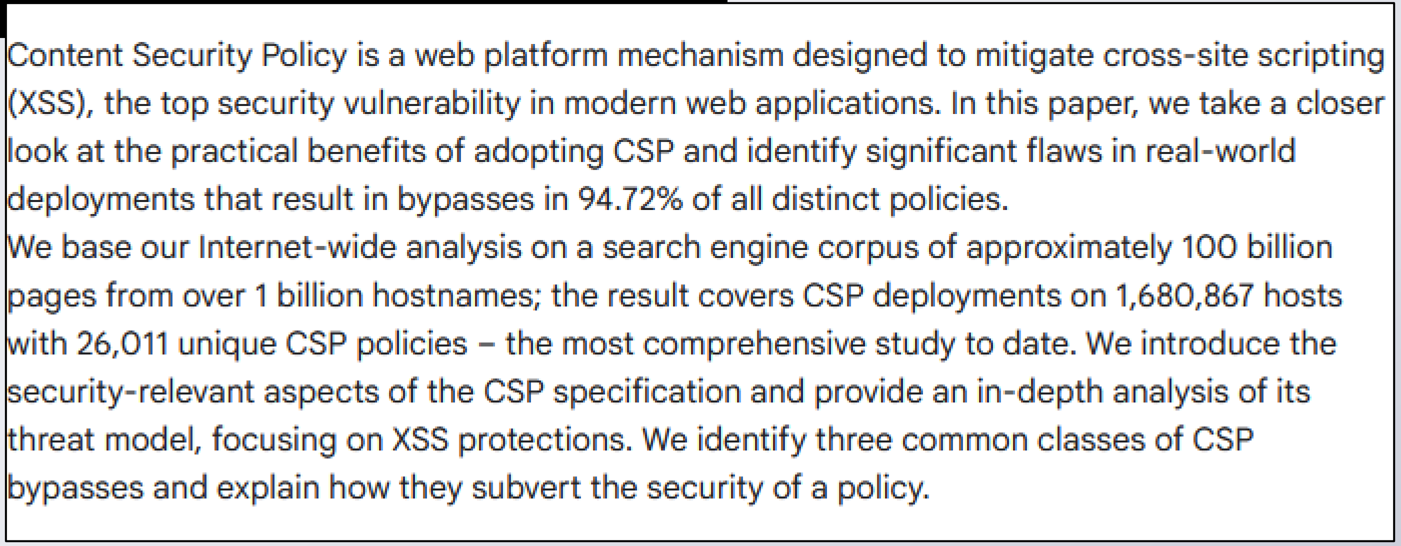
Similar to *iframe sandbox* but at top-level, setting a unique/opaque Origin (we may discuss in future sessions)

CSP: URL/Domain Allow-List Approach


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

Lukas Weichselbaum · Michele Spagnuolo · Sebastian Lekies · Artur Janc ·

Proceedings of the 23rd ACM Conference on Computer and Communications Security, ACM, Vienna, Austria (2016)


Content Security Policy is a web platform mechanism designed to mitigate cross-site scripting (XSS), the top security vulnerability in modern web applications. In this paper, we take a closer look at the practical benefits of adopting CSP and identify significant flaws in real-world deployments that result in bypasses in 94.72% of all distinct policies.

We base our Internet-wide analysis on a search engine corpus of approximately 100 billion pages from over 1 billion hostnames; the result covers CSP deployments on 1,680,867 hosts with 26,011 unique CSP policies – the most comprehensive study to date. We introduce the security-relevant aspects of the CSP specification and provide an in-depth analysis of its threat model, focusing on XSS protections. We identify three common classes of CSP bypasses and explain how they subvert the security of a policy.

https://research.google/pubs/csp-is-dead-long-live-csp-on-the-insecurity-of-whitelists-and-the-future-of-content-security-policy/

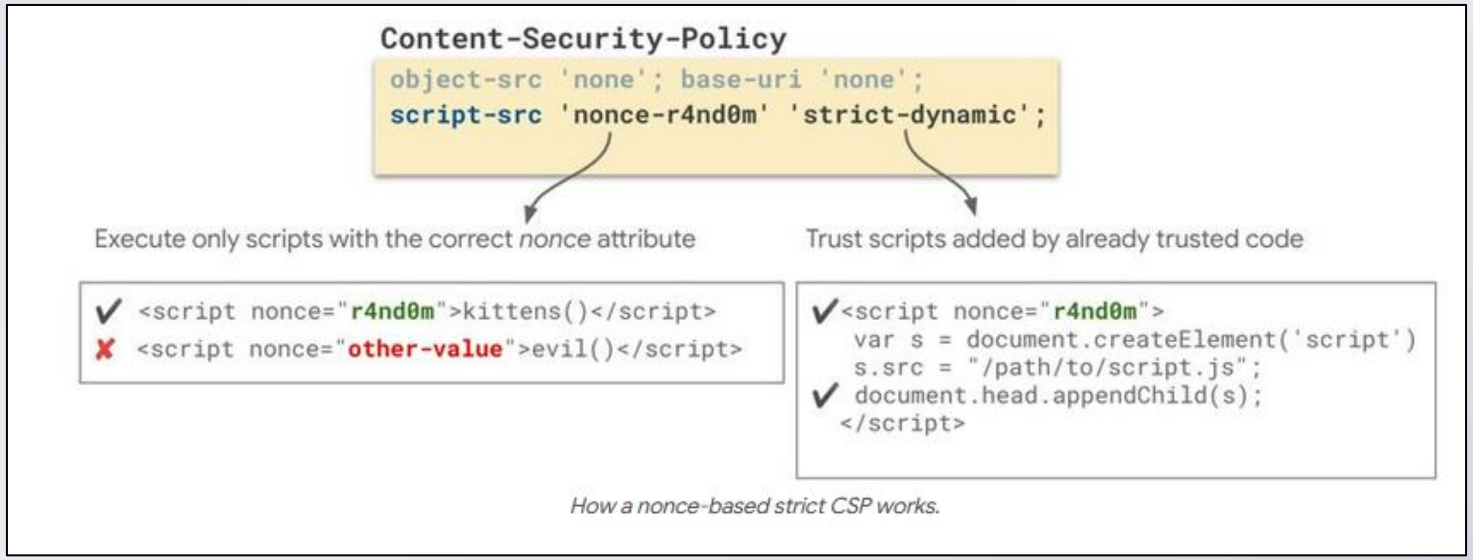
CSP: Common Bypasses

- Over-permissive allow-list (allowed sources permit user content)
- *‘unsafe-inline’* or *‘unsafe-eval’*
- JSONP to abuse *‘self’*
- Hijacking trusted resource
- Open redirects on trusted domains
- *base-uri* abuse
- JS framework script gadget
- CRLF (perhaps not so common anymore)
- Nonce exfiltration

DBG App Test Finding
Insecure Content Security Policy

CSP: Nonces and Hashes

- ***Nonces*** **are server-generated values intended to be random/unguessable and dynamically assigned to permit resources**
- ***Hashes*** **are generated based on the content**
- **Both are validated by the browser**
- ***'strict-dynamic'*** **permits trusted resources to load other resources**



```
Content-Security-Policy
object-src 'none'; base-uri 'none';
script-src 'nonce-r4nd0m' 'strict-dynamic';
```

Execute only scripts with the correct *nonce* attribute
Trust scripts added by already trusted code

```
✓ <script nonce="r4nd0m">kittens()</script>
✗ <script nonce="other-value">evil()</script>
```

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

How a nonce-based strict CSP works.

https://web.dev/articles/strict-csp

CSP: Strict

- **Most effective CSP implementation with the following features:**
  - **NO domain/URL-based allow-list**
  - **All resources must be permitted using a nonce or hash**
  - ***strict-dynamic* can permit trusted resources to load further resources**
  - ***object-src* and *base-uri* must be set to *'none'***
  - ***unsafe-inline* can be used for legacy browsers if necessary**

```
Content-Security-Policy:
  script-src 'nonce-{random}' 'strict-dynamic' https: 'unsafe-inline';
  object-src 'none';
  base-uri 'none';
```

https://web.dev/articles/strict-csp

CSP: Injection Into Trusted Context

HTTP Response: CSP Configuration

```
Content-Security-Policy: script-src 'nonce-ED98652N' 'self'; object-src 'none'; base-uri 'none';
```

Server-Side PHP Code

```
<script nonce="ED98652N">
    var profileData = {
        username: "<?php echo $_GET['name']; ?>",
        isLoggedIn: true
    }
</script>
```

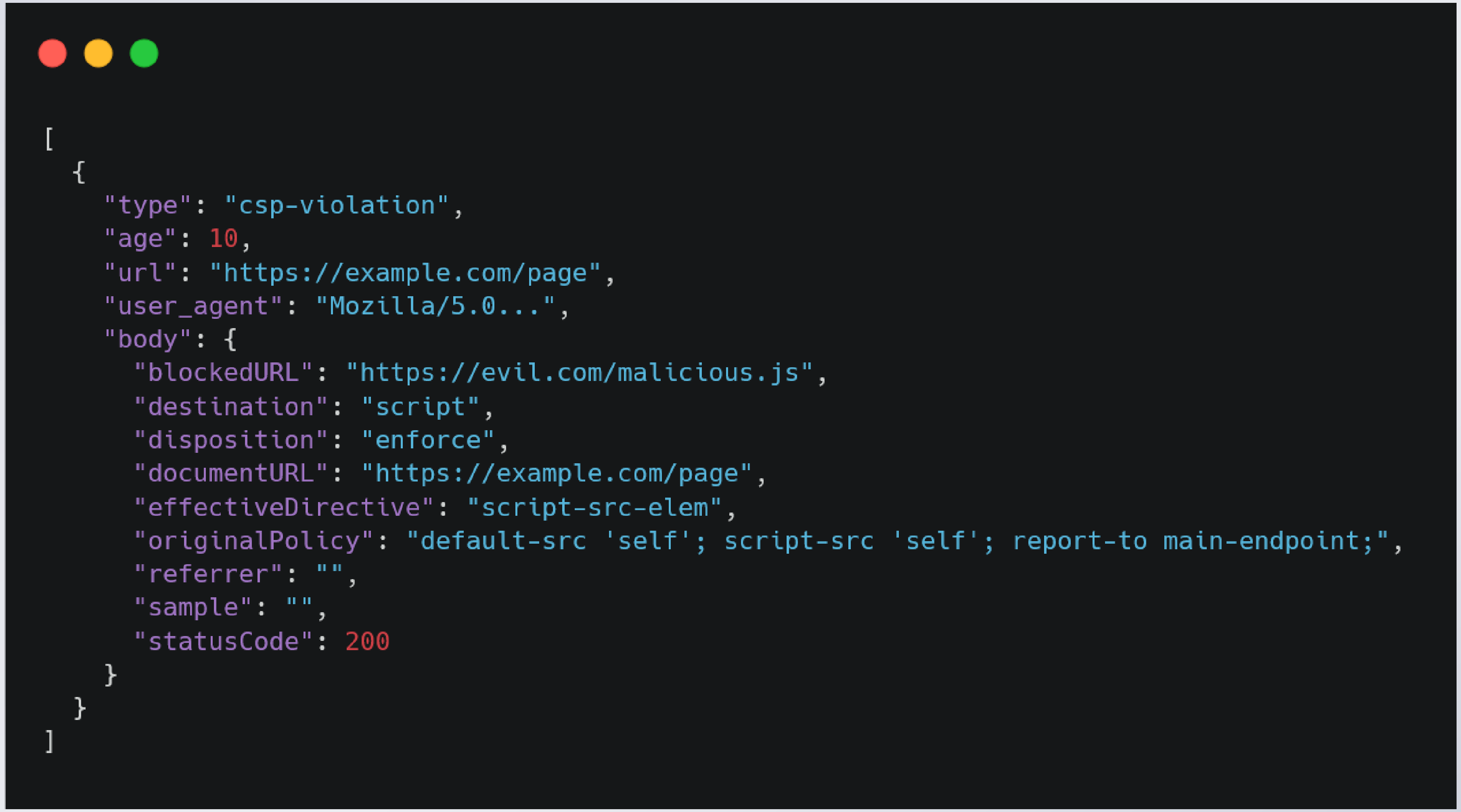
CSP: Preventing Framing Attacks

- CSP provides the *frame-ancestors* directive with the following possible values:
  - *none*: prevents the page from being embedded
  - *self*: prevents the page from being embedded by other Origins
  - <*source-expression-list*>: space-separated list of permitted Origins
- NOT impacted by *default-src* (fallback mechanism)
- The following embedding mechanisms are in scope:
  - <*frame*> (deprecated)
  - <*iframe*>
  - <*object*>
  - <*embed*>

CSP: Reporting

- **Violations of CSP can be logged:**
  - ***Content-Security-Policy: report-uri <uri>***
    - **Deprecated**
  - ***Content-Security-Policy: report-to <uri>***
  - **Via Reporting API:**
    - ***Reporting-Endpoints: main-endpoint= <uri>***
    - ***Content-Security-Policy: report-to main-endpoint;***
- **Reporting is most useful for ensuring functional CSP**
- **Can it identify attacks?**
  - **Possibly, but attackers developing exploits can test in their controlled browser and suppress requests to reporting endpoints**

CSP: Reporting



```
[
  {
    "type": "csp-violation",
    "age": 10,
    "url": "https://example.com/page",
    "user_agent": "Mozilla/5.0...",
    "body": {
      "blockedURL": "https://evil.com/malicious.js",
      "destination": "script",
      "disposition": "enforce",
      "documentURL": "https://example.com/page",
      "effectiveDirective": "script-src-elem",
      "originalPolicy": "default-src 'self'; script-src 'self'; report-to main-endpoint;",
      "referrer": "",
      "sample": "",
      "statusCode": 200
    }
  }
]
```

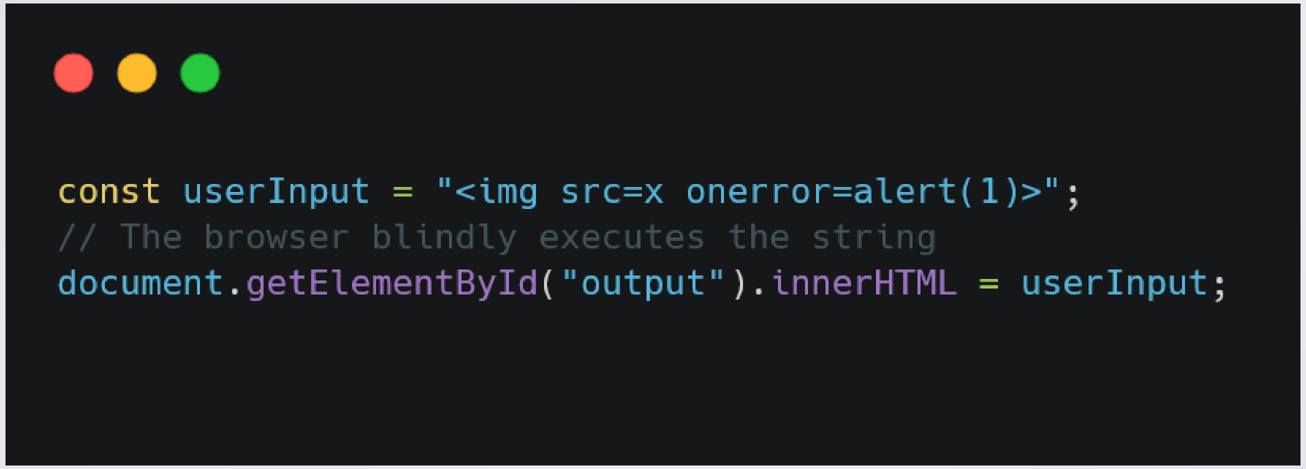
Trusted Types

- **Targets injection sinks to mitigate DOM-based XSS**
  - **Includes *.innerHTML*, *eval()*, *document.write()***
- **When TT enabled, the browser will NOT accept plain strings in sinks**
  - **BLOCKED: *el.innerHTML = "some_string"***
- **Developers must create a *Policy* to neutralize input which returns a "Trusted Type" object**
- **Therefore, a secure centralized policy means the app is protected**
- **TT is enforced with the CSP header:**
  - ***require-trusted-types-for 'script'***

DBG App Test Finding
Trusted Types Not Enforced

Trusted Types: No Enforcement

Client-Side JavaScript



```
const userInput = "<img src=x onerror=alert(1)>";
// The browser blindly executes the string
document.getElementById("output").innerHTML = userInput;
```

Trusted Types: Enforcement

HTTP Response: CSP Configuration

```
Content-Security-Policy: require-trusted-types-for 'script'; trusted-types myPolicy
```

Client-Side JavaScript

```
const myPolicy = trustedTypes.createPolicy('myPolicy', {
  createHTML: (string) => {
    // Basic (WEAK!!) sanitization: strip <script> tags
    return string.replace(/<script>/gi, '');
  }
});

// BLOCKED! Raw String!
document.getElementById("output").innerHTML = "Hello World";

// C. This will SUCCEED
const sanitized = myPolicy.createHTML("<b>Safe Content</b>");
document.getElementById("output").innerHTML = sanitized;
```

Cross-Site Script Inclusion (XSSI)

- Embedding external scripts results in their execution within the loading Origin's context, BUT scripts may be loaded with user's authority with 3rd party
- While you cannot read Cross-Origin scripts directly, their functionality/actions may leak information
- Most attacks abuse JavaScript resources, but there are alternatives

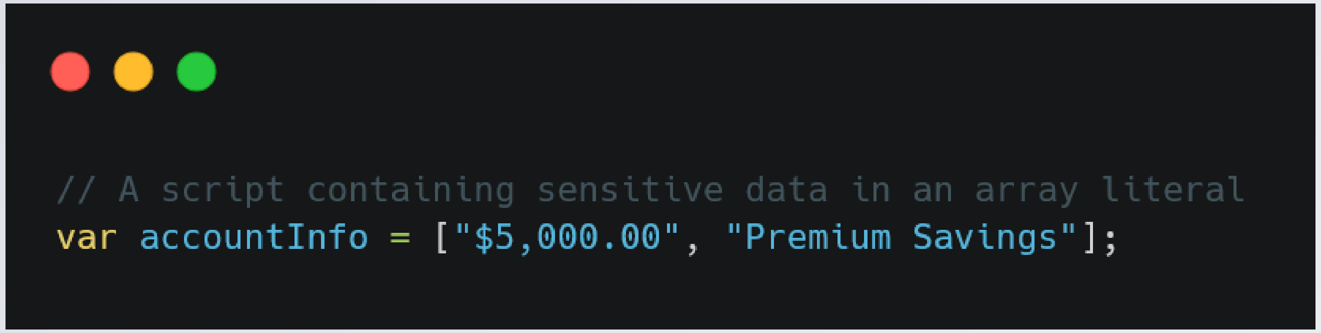
Cross-Site Script Inclusion (XSSI)

Traditional information leaking mechanisms:

- Global variables/functions (set on *window*)
- Prototype Pollution/Monkeypatching
  - Loading page uses JS to “poison” prototypes of objects used by loaded script
- JS Errors
  - Modern browsers restrict error details, but some error leakage can occur
- Timing Side-Channel (execution and resource loading)
- Monitoring global side effects
  - If the embedded script interacts with the DOM, can we observe?
  - Mutation Observers and Event Listeners can be useful here
- JONSP (technically, by design)

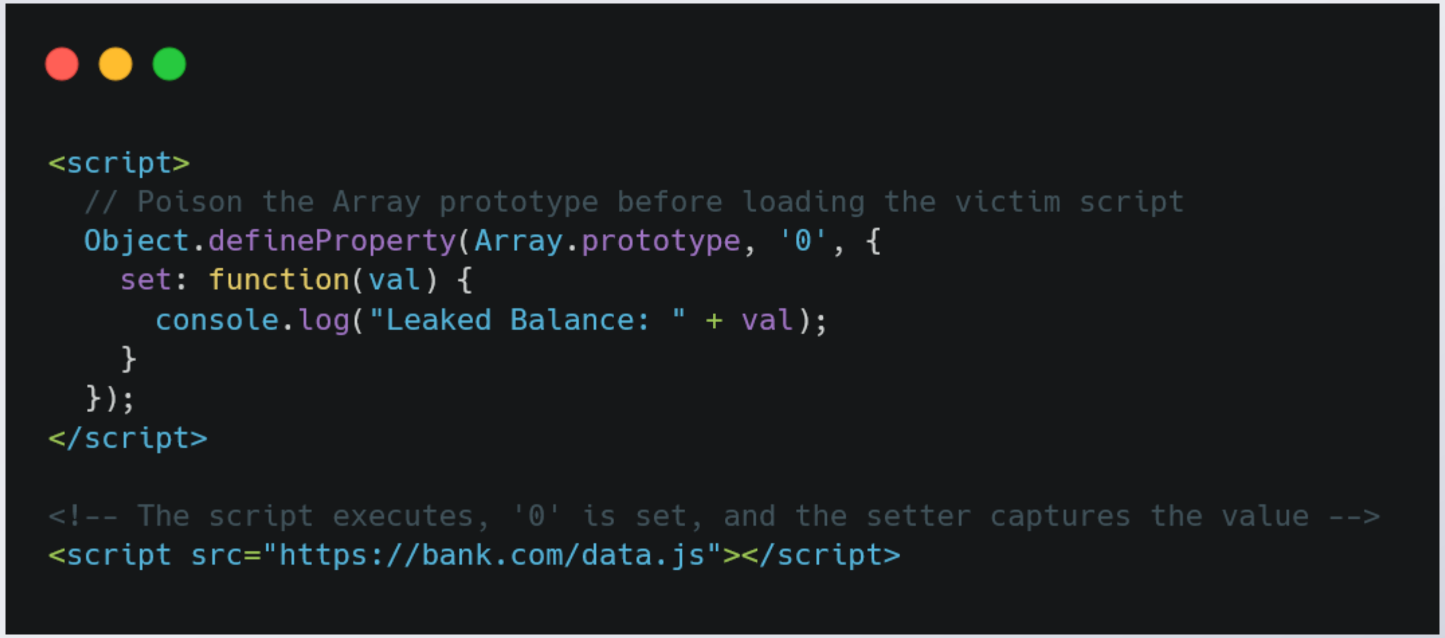
XSSI: Prototype Pollution (Obsolete)

Target application: ***trusted-site.com/sensitive-data-in-script.js***



```javascript
// A script containing sensitive data in an array literal
var accountInfo = ["$5,000.00", "Premium Savings"];
```

Malicious application: ***some-other-origin.com***

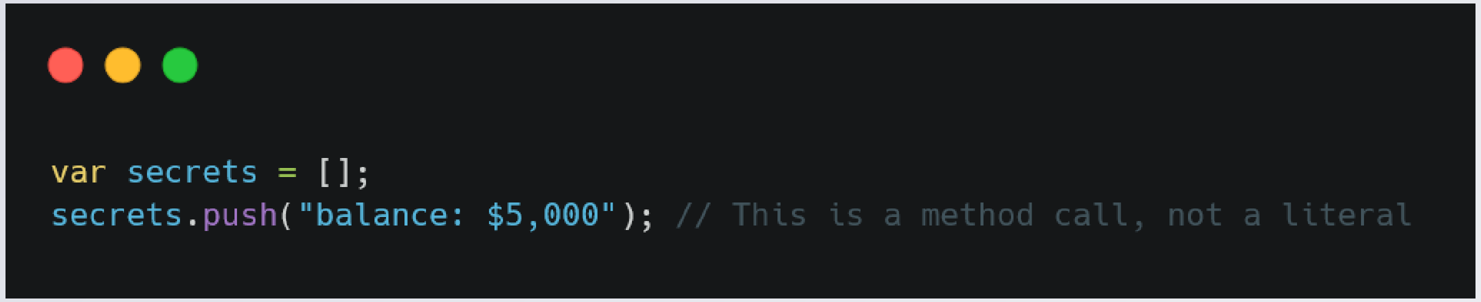


```html
<script>
  // Poison the Array prototype before loading the victim script
  Object.defineProperty(Array.prototype, '0', {
    set: function(val) {
      console.log("Leaked Balance: " + val);
    }
  });
</script>

<!-- The script executes, '0' is set, and the setter captures the value -->
<script src="https://bank.com/data.js"></script>
```

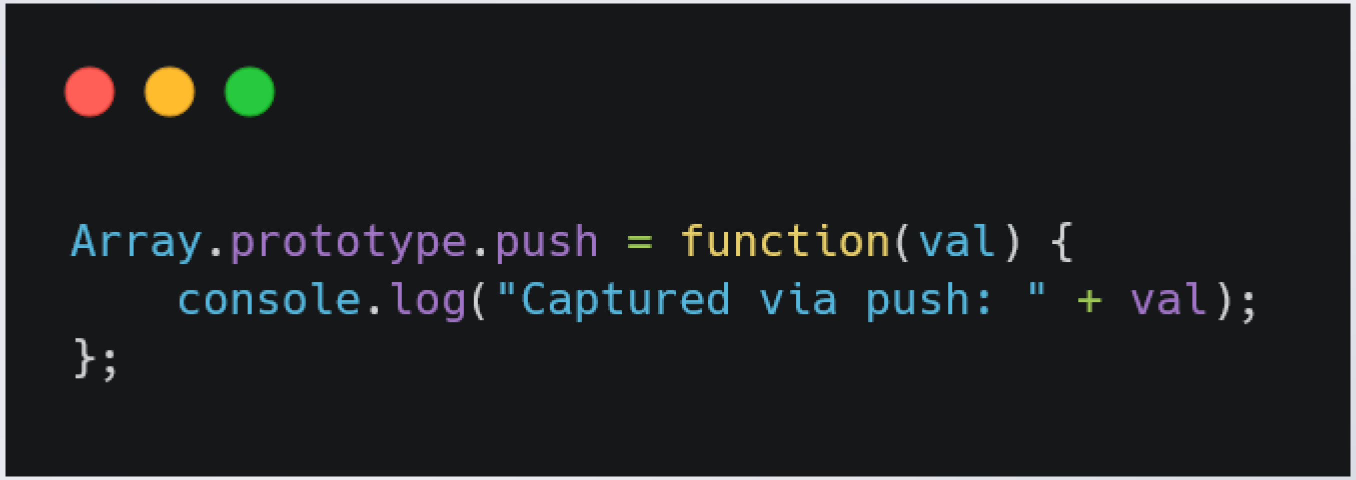
XSSI: Prototype Pollution (Method Override)

Target application: *trusted-site.com/sensitive-data-in-script.js*



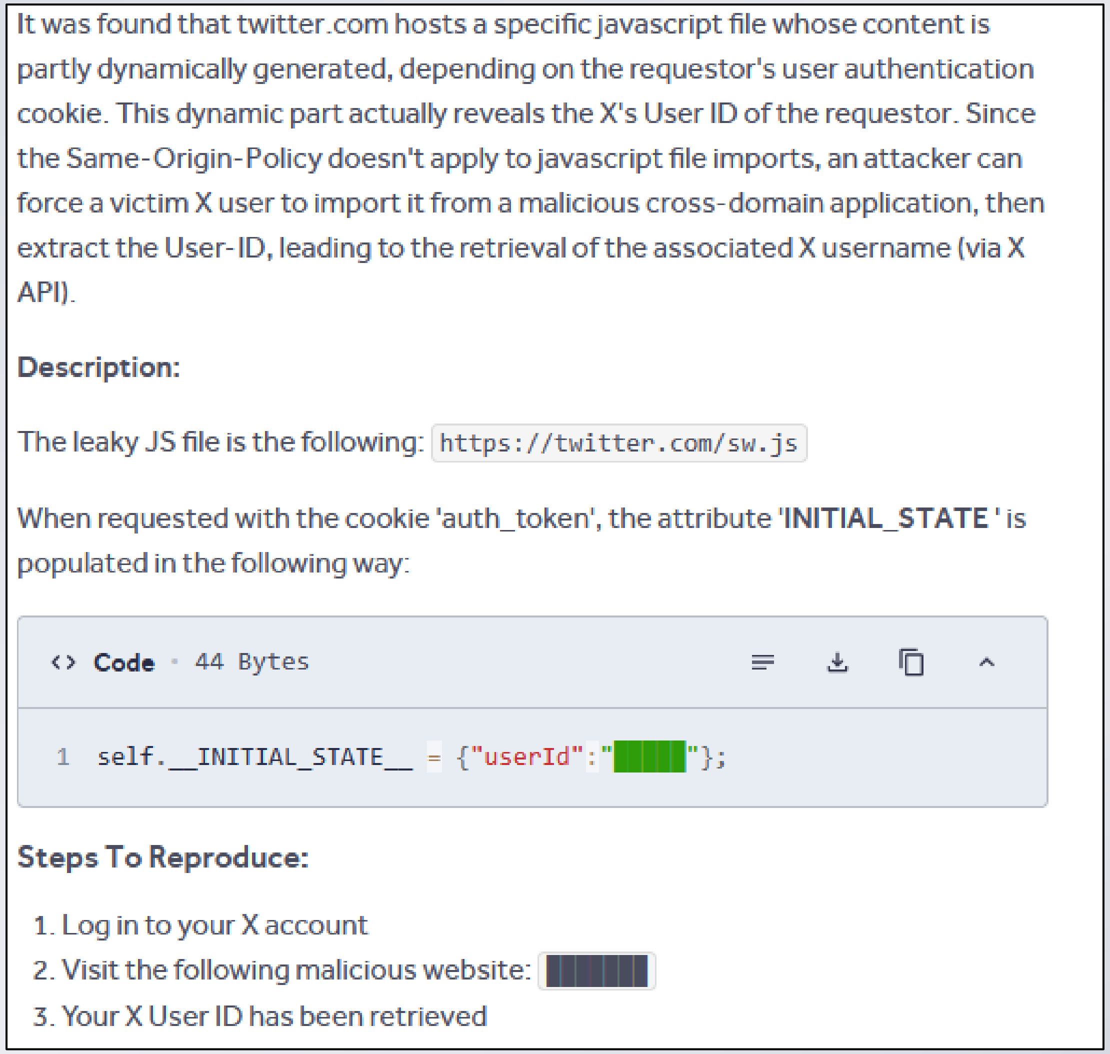
```javascript
var secrets = [];
secrets.push("balance: $5,000"); // This is a method call, not a literal
```

Malicious application: *some-other-origin.com*



```javascript
Array.prototype.push = function(val) {
    console.log("Captured via push: " + val);
};
```

XSSI: Recent Case Study (Twitter/X)



It was found that twitter.com hosts a specific javascript file whose content is partly dynamically generated, depending on the requestor's user authentication cookie. This dynamic part actually reveals the X's User ID of the requestor. Since the Same-Origin-Policy doesn't apply to javascript file imports, an attacker can force a victim X user to import it from a malicious cross-domain application, then extract the User-ID, leading to the retrieval of the associated X username (via X API).

Description:

The leaky JS file is the following: `https://twitter.com/sw.js`

When requested with the cookie 'auth_token', the attribute '**INITIAL_STATE**' is populated in the following way:

<> **Code** • 44 Bytes

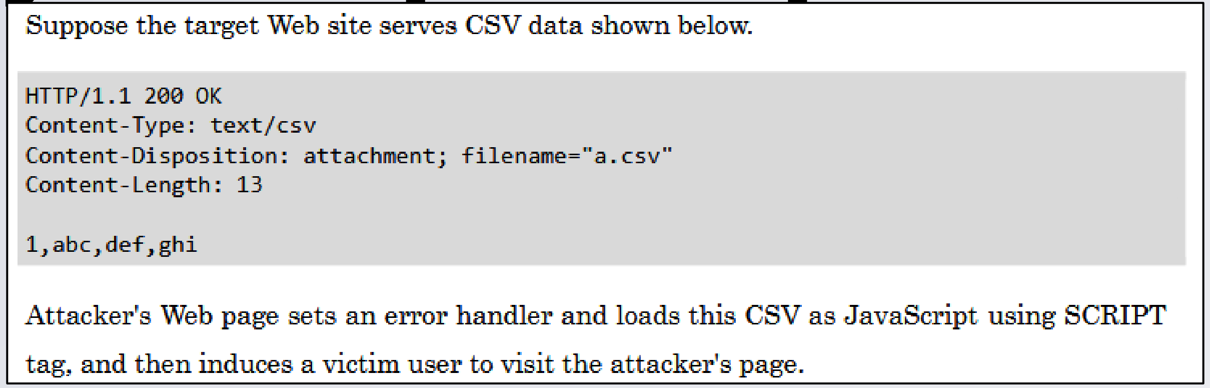
```
self.__INITIAL_STATE__ = {"userId":"[redacted]"};
```

Steps To Reproduce:

1. Log in to your X account
2. Visit the following malicious website: [redacted]
3. Your X User ID has been retrieved

https://hackerone.com/reports/2244229

XSSI: Non-JavaScript Example 1 (Historic)

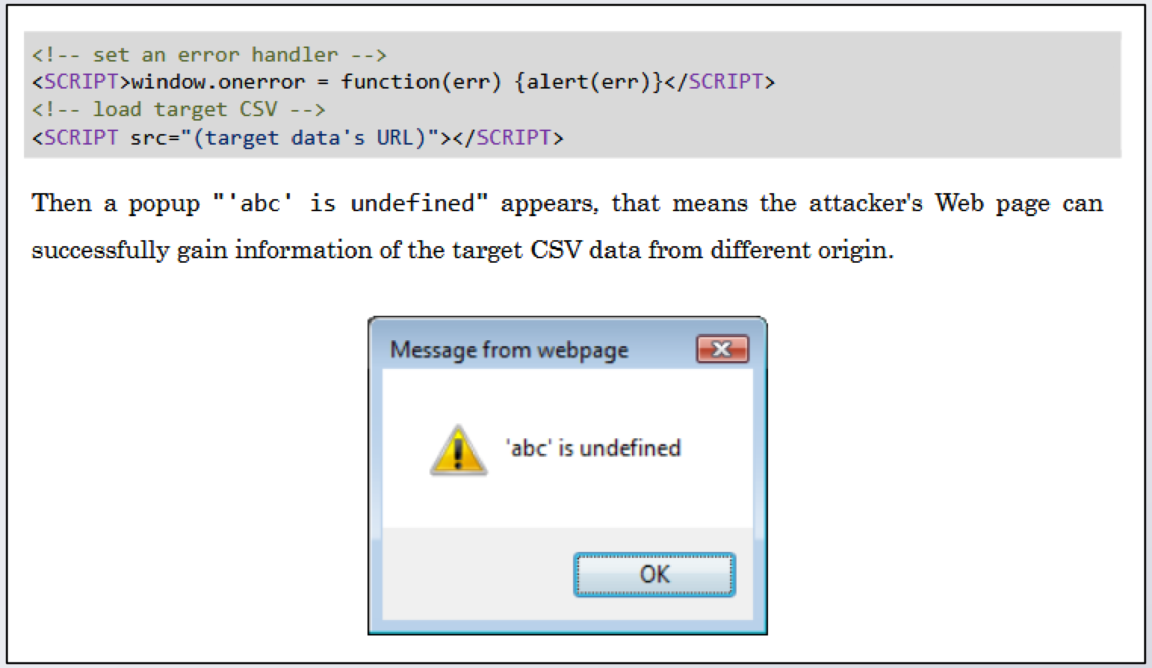


Suppose the target Web site serves CSV data shown below.

```
HTTP/1.1 200 OK
Content-Type: text/csv
Content-Disposition: attachment; filename="a.csv"
Content-Length: 13

1,abc,def,ghi
```

Attacker's Web page sets an error handler and loads this CSV as JavaScript using SCRIPT tag, and then induces a victim user to visit the attacker's page.



```
<!-- set an error handler -->
<SCRIPT>window.onerror = function(err) {alert(err)}</SCRIPT>
<!-- load target CSV -->
<SCRIPT src="(target data's URL)"></SCRIPT>
```

Then a popup "'abc' is undefined" appears, that means the attacker's Web page can successfully gain information of the target CSV data from different origin.

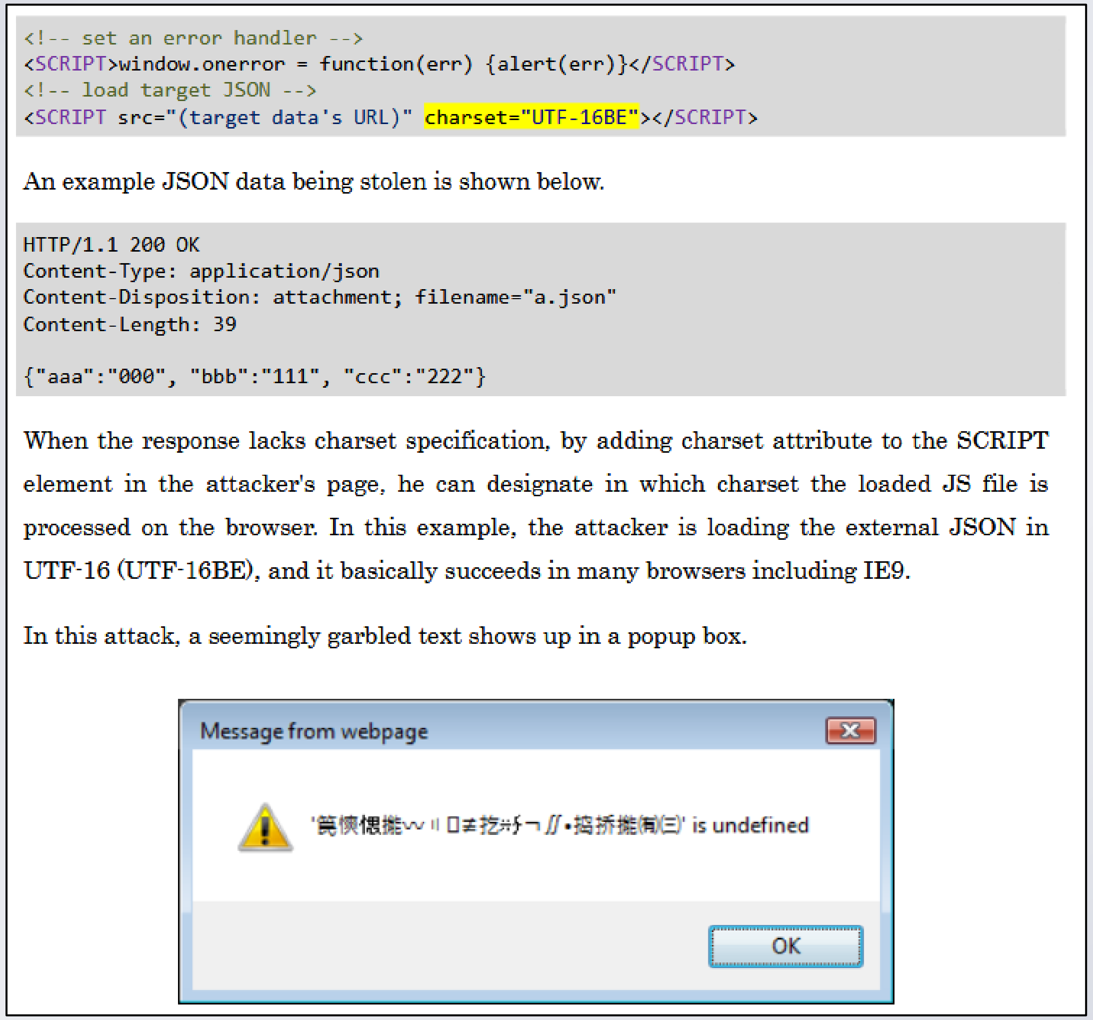
Message from webpage

'abc' is undefined

OK

From: Takeshi Terada's Identifier based XSSI Attacks (2015): https://www.mbsd.jp/Whitepaper/xssi.pdf

XSSI: Non-JavaScript Example 2 (Historic)



```
<!-- set an error handler -->
<SCRIPT>window.onerror = function(err) {alert(err)}</SCRIPT>
<!-- load target JSON -->
<SCRIPT src="(target data's URL)" charset="UTF-16BE"></SCRIPT>
```

An example JSON data being stolen is shown below.

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Disposition: attachment; filename="a.json"
Content-Length: 39

{"aaa":"000", "bbb":"111", "ccc":"222"}
```

When the response lacks charset specification, by adding charset attribute to the SCRIPT element in the attacker's page, he can designate in which charset the loaded JS file is processed on the browser. In this example, the attacker is loading the external JSON in UTF-16 (UTF-16BE), and it basically succeeds in many browsers including IE9.

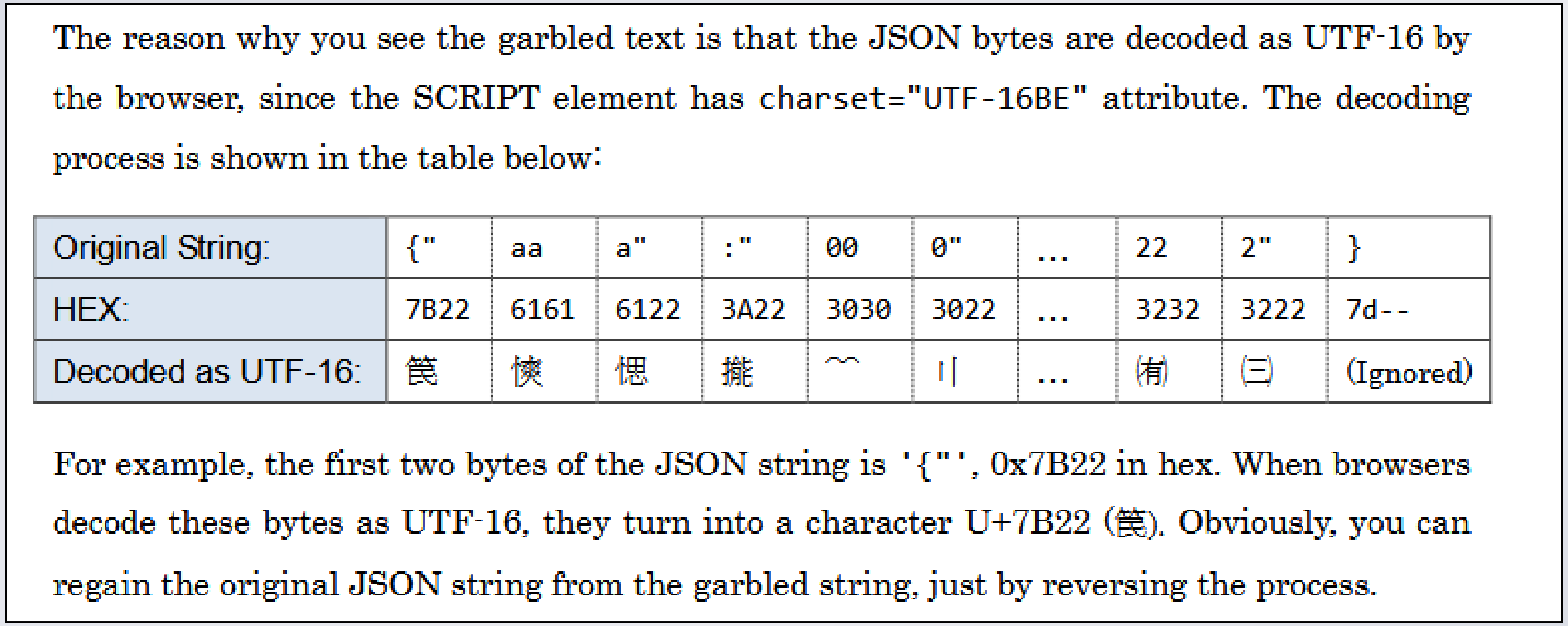
In this attack, a seemingly garbled text shows up in a popup box.

Message from webpage

'箂慡慢∺∠〰〢∬∠扢戢∺∠ㄱㄢ∬∠捣挢∺∠㈲㈢' is undefined

From: Takeshi Terada's Identifier based XSSI Attacks (2015): https://www.mbsd.jp/Whitepaper/xssi.pdf

XSSI: Non-JavaScript Example 2 (Historic)



The reason why you see the garbled text is that the JSON bytes are decoded as UTF-16 by the browser, since the SCRIPT element has `charset="UTF-16BE"` attribute. The decoding process is shown in the table below:

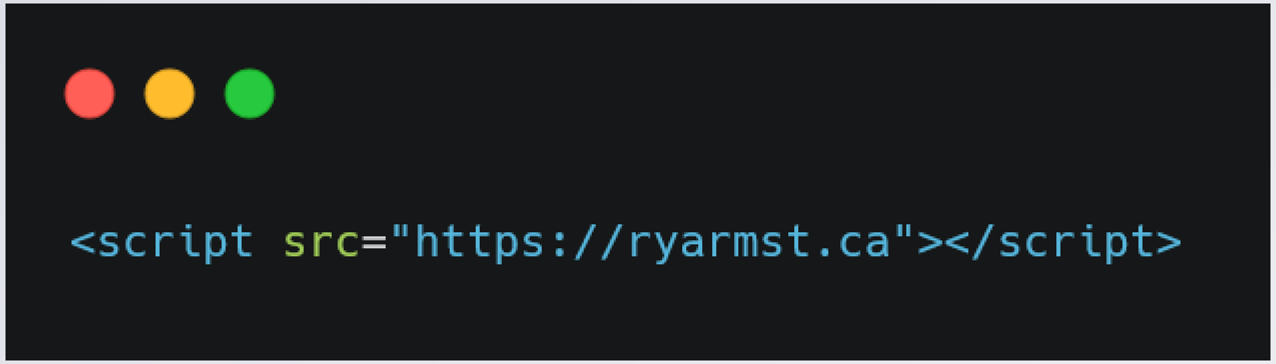
Original String:	{"	aa	a"	:"	00	0"	...	22	2"	}
HEX:	7B22	6161	6122	3A22	3030	3022	...	3232	3222	7d--
Decoded as UTF-16:	笢	慡	愢	㨢	〰	〢	...	㈲	㈢	(Ignored)

For example, the first two bytes of the JSON string is `'{"'`, 0x7B22 in hex. When browsers decode these bytes as UTF-16, they turn into a character U+7B22 (笢). Obviously, you can regain the original JSON string from the garbled string, just by reversing the process.

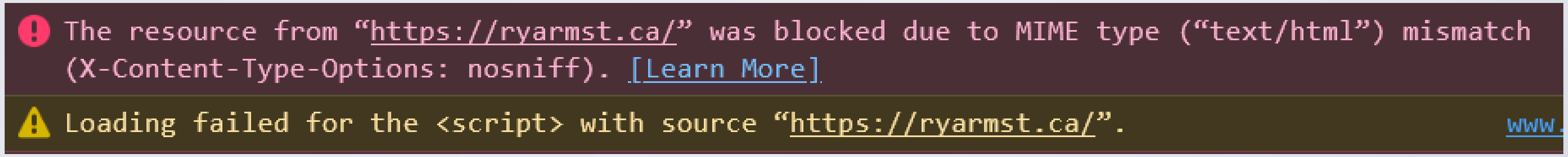
From: Takeshi Terada's Identifier based XSSI Attacks (2015): https://www.mbsd.jp/Whitepaper/xssi.pdf

XSSI: Non-JavaScript Blocked!

Embedded resource on different Origin



```
<script src="https://ryarmst.ca"></script>
```



```
The resource from “https://ryarmst.ca/” was blocked due to MIME type (“text/html”) mismatch
(X-Content-Type-Options: nosniff). [Learn More]
Loading failed for the <script> with source “https://ryarmst.ca/”.                    www.
```

XSSI: Non-JavaScript Script Execution

- A *Content-Type* of *application/javascript* is not required!
- Behavior ultimately depends on *Cross-Origin Read Blocking* (CORB)
- If the *Content-Type* is "protected" (JSON, HTML, XML) and the initial bytes match ("confirmation sniffing"), script becomes empty string
    - Example with *Content-Type: text/html* and a response body beginning <*html*, then the response cannot be embedded cross-origin
- The *Content-Type* header is strictly enforced with *X-Content-Type-Options: nosniff* even if body content matches JS

Cross-Origin Read Blocking (CORB)

- An algorithm to block dubious cross-origin resource loads
- CORB is like applying SOP to the original exceptions (<img>, <script>)
- Blocks before the response is rendered in any way by the browser
- Blocks non-JS XSSI
- When CORB acts, response body is made empty and headers removed
- Exempt: navigation requests or requests with destination of "object" or "embed" (such as <iframe>)
- Protections enhanced with *X-Content-Type-Options: nosniff*
- Worth reading:
    - https://chromium.googlesource.com/chromium/src/+/91216c0f/services/network/cross_origin_read_blocking_explainer.md

Cross-Origin Resource Protection (CORP)

- Set per-resource, allowing application to selectively opt-in to protect resources from Cross-Origin *no-cors* requests
- HTTP Header *Cross-Origin-Resource-Policy*, with values:
  - *same-site*: Only requests from the same Site can read/embed resource
  - *same-origin*: Only requests from the same Origin can read/embed resource
  - *cross-origin*: Requests from any Origin can read/embed resource
- Can mitigate XSSI and Spectre-like attacks
- Does NOT prevent request; applied by browser to protect response

XSSI: Mitigations

- Ensure proper *Content-Type* is set
- Return well-formed content and be mindful of injection
- Enforce strict CORB via *X-Content-Type-Options: nosniff*
- Restrict resource read/embed capabilities with CORP
- Do not embed sensitive content in JS

Cross-Origin Embedder Policy (COEP)

- Document-level HTTP response header to mandate that embedded Cross-Origin subresources are CORP or CORS permissive
- Ensures that subresources were intended to be embedded
- Primarily enforces thoughtful inclusion of embedded resources

Site Isolation

- In the age of Spectre-like vulnerabilities, browsers take additional precautions to prevent exploitation of hardware through shared processes
- *Site Isolation* is a mechanism that restores dangerous features when certain conditions are met
- Requirements to achieve Site Isolation:
  - COOP set to *same-origin*
  - COEP set to *require-corp* or *credentialless*
  - Due to COEP, embedded resources naturally CORS or CORP compliant

Testing for XSSI

- A resource returns potentially sensitive information (typically dynamic, and tied to user's session)
- Resource must be accessible via Cross-Origin embed requests
  - Likely need permissive *SameSite* cookies OR perhaps internal resource
- Resource must be interpreted by the browser as valid JS content
  - Consider: *Content-Type*, *CORB/nosniff*, encoding quirks, *CORP*
- There must be some mechanism to leak data
  - Global exposure, function calls, DOM manipulation, timing…
- Good luck!

OWASP ASVS 5.0

V1 Encoding and Sanitization: V1.3 Sanitization
- **1.3.3: Verify data passed to dangerous contexts is sanitized**

V3 Web Frontend Security: V3.2 Unintended Content Interpretation
- **3.2.1: Prevent rendering of response content in unintended context**

V3 Web Frontend Security: V3.4 Browser Security Mechanism Headers
- **3.4.4: Implement *X-Content-Type-Options: no-sniff***
- **3.4.7: Implement CSP reporting**

V3 Web Frontend Security: V3.5 Browser Origin Separation
- **3.5.7: Do not include authorized data in script content**
- **3.5.8: Permit loading of authenticated resources only when intended**

V3 Web Frontend Security: V3.6 External Resource Integrity
- **3.6.1: Implement SRI where applicable**

https://github.com/OWASP/ASVS/blob/master/5.0/en/