CSP: Content Security Policy

The History and the Future of CSP

Takashi Yoneuchi

The University of Tokyo, http://shift-js.info

takashi.yoneuchi@shift-js.info

Outline

- 1: A Brief Description of CSP
- 2: Battle Against Attackers: The History of CSP
- 3: Ways of Bypassing CSP
- 4: Recent Proposals for CSP
- 5: Conclusion

1: A Brief Description of CSP

Content Security Policy

- *"**Content Security Policy** (CSP) is an added layer of security that **helps to detect and mitigate certain types of attacks**, including **Cross Site Scripting** (XSS) and **data injection attacks**. These attacks are used for everything from data theft to site defacement or distribution of malware."*
  (Quoted from: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP)
- CSP orders browsers to load and execute the contents from specified sources.
  - Modern browsers understand Content-Security-Policy header and work with the policies.
  - With good policies, it is strong mitigation of XSS.

2: Battle Against Attackers: The History of CSP

The History of CSP

- 2007: Jim et al. proposed BEEP
- 2008: Oda et al. proposed SOMA.
- 2009: Van Gundy and Chen proposed Noncespaces.
- 2010: Stamm et al. proposed CSP.
- 2012: CSP 1.0 W3C Candidate Recommendation and Group Note was released, and Noncespaces was improved and released again.
- …
- 2016: CSP Level 2 - W3C Recommendation was released.
- 2017: CSP Level 3 - W3C Working Draft is in progress.

Jim et al. proposed BEEP [1]

- BEEP: Browser-Enforced Embedded Policies

  Jim, T., Swamy, N., & Hicks, M. (2007, May). Defeating script injection attacks with browser-enforced embedded policies. In Proceedings of the 16th international conference on World Wide Web (pp. 601-610). ACM.

- **IDEA**: Browsers know perfectly what scripts are loaded and when the scripts will be executed.
- **IDEA**: Web developers know what scripts are needed.
  - Therefore, the server should provide the policy,
  - and the browser should follow it.

Jim et al. proposed BEEP [1]

- ***afterParseHook*** function of JavaScript returns a boolean value whether the given script can be executed.
  - It enables us to define the policies flexibly.
- BEEP can deal with the problem of node-splitting attacks and can be implemented with small changes.
- Limitations
  - BEEP itself doesn't help whether the developer should trust a third-party script or not.
  - It restricts only JavaScript execution.
    - Later BEEP was extended: Mutation-Event Transforms, etc.

      Erlingsson, U., Livshits, B., & Xie, Y. Mutation-Event Transforms: A Flexible Client-side Foundation for End-to-end Web 2.0 Security.

Oda et al. proposed SOMA [1]

- SOMA: Same Origin Mutual Approval

  Oda, T., Wurster, G., van Oorschot, P. C., & Somayaji, A. (2008, October). SOMA: Mutual approval for included content in web pages. In Proceedings of the 15th ACM conference on Computer and communications security (pp. 89-98). ACM.

- It points out that SOP defines only the permissions for R/W/E among the origin and the contents loaded for it.
  - SOMA is tighter than SOP; it defines the permission for fetching (accessing) the contents other domains have.
    - Information stealing with fetching (eg. cookie-stealing) can be prevented.
  - It can restrict **all content types**, including images, styles, …

Oda et al. proposed SOMA [2]

- ***/soma-manifest*** defines a list of domains from which the origin domain allows to include the content.
- ***/soma-approval*** answers whether the content of its domain can be loaded from the origin domain.
- Limitations
  - SOMA doesn't stop attacks to/from mutually allowed origins/sources (**SAME AS CSP**).
  - SOMA doesn't stop the attacks which needs no outside communication; it's meaningless for non-script attacks, SSRF, etc.

Van Gundy and Chen proposed Noncespaces 1

- Noncespaces
  Van Gundy, M., & Chen, H. (2012). Noncespaces: Using randomization to defeat cross-site scripting attacks. *computers & security*, *31*(4), 612-628.
- Clients can't distinguish the difference between trusted or untrusted content.
  - **IDEA:** It is the servers that can tell the difference.
  - **IDEA**: Trusted contents are almost static; Server can give HTML tags a randomized ***nonce*** attackers can't guess.
    - It's effective for node-splitting attack and non-script ones.
    - With flexible policy language, in untrusted content some tags can be used if properly configured: <b>, <s>, etc.

Van Gundy and Chen proposed Noncespaces 2

- ***X-Noncespaces-{Version|Policy|Context}*** tells a browser nonces and the place of a policy file.
  - Policy: URI
  - Context: (TrustClass = Rand)+
- Limitation
  - It can restrict only documents based on XML.
    - Documents would be served as *application/xhtml+xml*.
  - Policies can be complex esp. for large websites.
    - In 2012 training mode was released.

Stamm et al. proposed CSP 1

- CSP: Content Security Policy

  Stamm, S., Sterne, B., & Markham, G. (2010, April). Reining in the web with content security policy. In Proceedings of the 19th international conference on World wide web (pp. 921-930). ACM.

- The way to prevent data leak attacks, execution of arbitrary JS codes and clickjacking integrally is needed.
  - Images, styles and other contents can be restricted with checking its origin (like SOMA).
  - **However** clients can't differentiate the legitimate or injected scripts.
    - **IDEA**: Developers can separate scripts and others!
    - If separated, scripts can be restricted like other contents.

Stamm et al. proposed CSP 2

- Basic restrictions: inline scripts and strings cannot be executed. (eg. *<script>*, *javascript:*, *onclick="…")*
- ***Directives*** state how browser behave: the source of loading contents, executing scripts, etc.
- Limitations
  - For a large and complex website, the manual definition of the policy will be difficult.
  - Web sites will be needed some or drastic changes because of basic restrictions.

Content Security Policy 1.0 [1]

- https://www.w3.org/TR/CSP1/
- ***X-Content-Security-Policy(-Report-Only)*** *header* with *directives* will be delivered.
- There are some special keywords (source expressions):
  - **'self'** represents the set of URIs in the same origin.
  - **'unsafe-inline'** represents the approval for executing inline scripts like *<script>*, *javascript:* and *onclick=""*.
  - **'unsafe-eval'** represents the approval for executing strings.
- ***script-src: (URI)+*** defines the list of allowed sources.
  - Here you can put 'unsafe-(inline|eval)' keywords.

Content Security Policy 1.0 2

- Each of ***object-src***, ***style-src***, ***img-src***, ***font-src***, ***media-src***, ***frame-src*** and ***connect-src*** defines the allowed sources.
- If a directive which matches a processing content (img, styles, …) is not defined, ***default-src*** will be used.
- ***sandbox*** directive forces the browser enable to use sandbox.
- ***report-uri*** defines where to report the policy violation.
  - This enables developers to notice:
    - misconfigurations
    - attacks

CSP Level 2 [1]

- ***Content-Security-Policy(-Report-Only)*** *header* with *directives* will be delivered.
- Some directives were added:
  - **base-uri**: restriction on href of <base>
  - **child-src**: restriction on frame and Workers (frame-src is deprecated)
  - **form-action**: restriction on targets of submit forms
  - **frame-ancestors**: replacement of *X-Frame-Options* Header
  - **plugin-types**: restriction on plugin types
- With nonces or hashes, inline script and styles can be whitelisted individually.

CSP Level 2 2

- Redirection from the trusted source to an untrusted source become to be ignored; it's not a violation of the policy.
  - Using Content-Security-Policy for Evil
    http://homakov.blogspot.jp/2014/01/using-content-security-policy-for-evil.html
- ***SecurityPolicyViolationEvent*** became to be fired upon violations.
  - Developers can handle violations with JS.
- For reporting violations, new fields were added.
  - ***effectiveDirective***, ***status code***, etc.

CSP Level 3 (Working Draft) 1

- ***worker-src*** was added and ***frame-src*** was undeprecated.
  - If they are not set, ***child-src*** and ***default-src*** will be applied.
- ***manifest-src*** was added.
- ***report-uri*** was deprecated and ***report-to*** was added.
- For ***script-src*** and ***default-src***, ***strict-dynamic*** source expression was added.
- URL matching algorithm was changed.
  - if you set the policy for http, more secure protocols like https will be regarded as a trusted source.

CSP Level 3 (Working Draft) 2

- When an inline script/style or eval is blocked, *blocked-uri* now returns *inline* or *eval*.
- If **strict-dynamic** is set in ***(script|default)-src*** ,
  - The whitelist, ***unsafe-inline*** and ***self*** will be ignored.
  - Nonce-source (nonce-xxxxxx…) and hash-source (sha256-xxxxxxx….) will be applied.
  - The ***parser-inserted* <script>** (inserted by trusted script to the DOM) will be executed.
    - If a trusted script executes *document.write('<script>…')*, it won't be executed (because it's ***non-parser-inserted***).

Summary of present CSP

- Current CSP requires significant efforts because
  - CSP is powerless with ***unsafe-eval*** or ***unsafe-inline***, but in real-world websites they can't be omitted perfectly.
    - Since CSP3, nonces and hashes changed this situation.
  - Defining whitelists are too difficult.
  - It has some problems in harmony with CDNs.
- Is the ***'strict-dynamic'*** keyword a perfect way ? No.
  - **Problem 1:** it's only for JS; other content types will be blocked even if loaded from trusted scripts.
  - **Problem 2:** This is a kind of all-or-nothing mechanisms.

Comparison among studies

	BEEP	SOMA	Noncespaces	CSP 1.0	CSP Level 2	CSP Level 3
Policies	Inside <head>	External Files	External Files &Headers	Headers (+ Ext. Files)	Headers (+ Ext. Files)	Headers (+ Ext. Files)
Contents Types	Only JS	All	XHTML	All	All	All
Outside Origin *1	Strong	Strong	Strong	Strong	Strong	Strong
Inside Origin *2	Strong	Weak	Strong	Strong	Strong	Strong
Additional Req.	0	1 per domain	1 per domain	0 or 1	0 or 1	0 or 1
Difficulty for site owners	Mid	Mid	Mid	High	??	??

3: Ways of Bypassing CSP

Ways of Bypassing CSP

- **Example 1:** Server Misconfigurations
  - An important problem: setting CSP policy for large websites often burdens with developers !
- **Example 2:** Attacks with trusted sources (JSONP Injection,etc)
- **Example 3:** Attacks with polyglot JPEGs
  - Bypassing CSP using polyglot JPEGs
    http://blog.portswigger.net/2016/12/bypassing-csp-using-polyglot-jpegs.html
- **Example 4:** Code-Reuse Attacks
- Of cource, There're many other problems: Mis-implementation on browsers, E4X, etc.

1: Server Misconfigurations 1

- ***'unsafe-inline'*** and ***'unsafe-eval'*** makes CSP almost powerless against XSS.
  - However, if these source expression is not set, many websites running now won't work.
- If ***default-src*** is omitted, other directives which isn't set will be regarded as ***'*'***.
  - When *Content-Security-Policy: script-src: 'self'* is set,
  - ** will be allowed.

1: Server Misconfigurations 2

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

  Weichselbaum, L., Spagnuolo, M., Lekies, S., & Janc, A. (2016). CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16, 1376–1387.

- Google's research above (2016) shows that **94.72%** of all distinct policies in the real-world Web can be bypassed.
  - **84.38%** of them are setting '***unsafe-inline***' keyword !
  - Nonces and hashes are safer than other policies, but dynamic script insertion by JS libraries will be blocked.
    - In this paper, ***strict-dynamic*** keyword was proposed.

2: With trusted sources

- See: H5SC Minichallenge 3: "Sh*t, it's CSP!”
  https://github.com/cure53/XSSChallengeWiki/wiki/H5SC-Minichallenge-3:-%22Sh*t,-it%27s-CSP!%22
- Attacks using features of frameworks like AngularJS
  - In the Cure53’s challanges (the link above), ***ng-app*** and ***ng-csp*** are used.
- JSONP Injection
  - <script src = “http://trusted.source/jsonp**?callback=**alert(0)//“>
- Redirection from the trusted sources to untrusted ones
  - This is not a violation since CSP Level 2.0.

3: With polyglot JPEGs

- Bypassing CSP using polyglot JPEGs
  http://blog.portswigger.net/2016/12/bypassing-csp-using-polyglot-jpegs.html
- Recent web services often hosts user-upload images.
  - Besides, the images are often placed inside ***'self'***.
  - If the policy is like ***script-src: 'self'***, the image can be set in *src* of <script>.
- With JPEG format, valid JavaScript code can be made!
  - FF D8 FF E0 2F 2A …. 2A 2F (JS Here) 2F 2A ….FF D9

    File Signature (non-ASCII, ignored) | File Length /* | JPEG Comment */ (JS Here) /* | End

4: Code-Reuse Attacks

- Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets

  Lekies, S., Kotowicz, K., Groß, S., Nava, E. A. V., & Johns, M. (2017). Code-Reuse Attacks for the Web : Breaking Cross-Site Scripting Mitigations via Script Gadgets. ACM SIGSAC Conference on Computer and Communications Security (CCS), 1709–1723.

- ***'strict-dynamic'*** keyword enables *parser-inserted* <script> to be executed.
  - Therefore, when benign HTMLs is inserted and transformed to <script> by legitimate scripts, it can run.
  - **Example:** RequireJS (when trusted) and *<script data-main='data:1,alert(1)'></script>* will show us an alert dialogue.

4: Recent Proposals & Discussion on CSP

Recent Proposals & Discussion on CSP

- CCSP: Controlled Relaxation of Content Security Policies by Runtime Policy Composition
- CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-worldWebsites
- Content Security Policy: Embedded Enforcement

Controlled Relaxation of Content Security Policies 1

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

  Calzavara, S., Rabitti, A., Bugliesi, M., Ca, U., & Venezia, F. (2017). CCSP : Controlled Relaxation of Content Security Policies by Runtime Policy Composition. USENIX Security 2017 (26th USENIX Security Symposium).

- CCSP composes CSP rules dynamically with 3 HTTP headers.
  - ***CSP-Compose*** is used by protected resource and states initial CSP rules (ex. sources of JS libraries, URLs of API, etc).
  - ***CSP-Intersect*** states the trusted sources from which each loaded script can deal.
  - ***CSP-Union*** is used by content providers and states what is needed to run successfully (img-src, etc).

Controlled Relaxation of Content Security Policies 2

- CSP depends on website developer's hard work, but CCSP distributes the burden into not only them but **third-parties**.
- Limitations
  - The adoption of CCSP depends mainly on third-parties.
  - CCSP is more complex than CSP although it is more expressive.
  - Because policies will be changed dynamically, debugging of policies is difficult.
    - Robust reporting system should be created.

CSPAutoGen 1

- CSPAutoGen

  Pan, X., Liu, S., Zhou, Y., Chen, Y., Zhou, T., & Cao, Y. (2016). CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites. *Ccs*, 653–665.

- (TBA)

Content Security Policy: Embedded Enforcement 1

- Content Security Policy: Embedded Enforcement W3C Working Draft
  https://www.w3.org/TR/csp-embedded-enforcement/
- (TBA)

5: Conclusion

Conclusion

- **Current Content Security Policy is not a perfect**; it burdens with web developers.
- The features of CSP 1.0 and Level 2 is almost powerless due to misconfigurations.
- However, CSP is effective with good policies.
  - Use ***'strict-dynamic'*** keywords.
  - Use **nonces** and **hashses**.
  - Avoid using ***'unsafe-inline'*** and ***'unsafe-eval'*** keywords.

Acknowledgements

- (TBA)

References 1

1. Barth, A., Veditz, D., & West, M. (2016). *Content Security Policy Level 2 W3C Recommendation*. Retrieved December 3, 2017, from https://www.w3.org/TR/CSP2/
2. Calzavara, S., Rabitti, A., Bugliesi, M., Ca, U., & Venezia, F. (2017). *CCSP : Controlled Relaxation of Content Security Policies by Runtime Policy Composition*. USENIX Security 2017 (26th USENIX Security Symposium).
3. Cure53. (2015). *H5SC Minichallenge 3: "Sh*t, it's CSP!"*. Retrieved December 3, 2017, from https://github.com/cure53/XSSChallengeWiki/wiki/H5SC-Minichallenge-3:-%22Sh*t,-it%27s-CSP!%22
4. Erlingsson, U., Livshits, B., & Xie, Y. *Mutation-Event Transforms: A Flexible Client-side Foundation for End-to-end Web 2.0 Security*.
5. Heyes, G. (2016). *Bypassing CSP using polyglot JPEGs*. Retrieved December 3, 2017, from http://blog.portswigger.net/2016/12/bypassing-csp-using-polyglot-jpegs.html
6. Homakov, E. (2014). *Using Content-Security-Policy for Evil*. Retrieved December 3, 2017, from http://homakov.blogspot.jp/2014/01/using-content-security-policy-for-evil.html
7. Jim, T., Swamy, N., & Hicks, M. (2007, May). *Defeating script injection attacks with browser-enforced embedded policies*. In Proceedings of the 16th international conference on World Wide Web (pp. 601-610). ACM.
8. Karlsson, M. (2016). *CSP: bypassing form-action with reflected XSS*. Retrieved December 3, 2017, from https://labs.detectify.com/2016/04/04/csp-bypassing-form-action-with-reflected-xss/
9. Lekies, S., Kotowicz, K., Groß, S., Nava, E. A. V., & Johns, M. (2017). Code-Reuse Attacks for the Web : Breaking Cross-Site Scripting Mitigations via Script Gadgets. ACM SIGSAC Conference on Computer and Communications Security (CCS), 1709–1723.
10. Oda, T., Wurster, G., van Oorschot, P. C., & Somayaji, A. (2008, October). *SOMA: Mutual approval for included content in web pages*. In Proceedings of the 15th ACM conference on Computer and communications security (pp. 89-98). ACM.
11. Pan, X., Liu, S., Zhou, Y., Chen, Y., Zhou, T., & Cao, Y. (2016). CSPAutoGen: *Black-box Enforcement of Content Security Policy upon Real-world Websites*. Ccs, 653–665.
12. Stamm, S., Sterne, B., & Markham, G. (2010, April). *Reining in the web with content security policy*. In Proceedings of the 19th international conference on World wide web (pp. 921-930). ACM.
13. Scalzi, G. (2016). *Content-Security-Policy: misconfigurations and bypasses*. Retrieved December 3, 2017, from https://blog.compass-security.com/2016/06/content-security-policy-misconfigurations-and-bypasses/
14. Sterne, B., & Barth, A. (2015) *Content Security Policy 1.0 W3C Working Group Note*. Retrieved December 3, 2017, from https://www.w3.org/TR/CSP1/
15. Van Gundy, M., & Chen, H. (2012). *Noncespaces: Using randomization to defeat cross-site scripting attacks*. computers & security, 31(4), 612-628.
16. Weichselbaum, L., Spagnuolo, M., Lekies, S., & Janc, A. (2016). *CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy*. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16, 1376–1387.
17. West, M. (2016). *Content Security Policy Level 3 W3C Working Draft*. Retrieved December 3, 2017, from https://www.w3.org/TR/CSP3/
18. West, M. (2016). *Content Security Policy: Embedded Enforcement W3C Working Draft.* Retrieved December 3, 2017, from https://www.w3.org/TR/csp-embedded-enforcement/

Thanks :-)

Please feel free to contact me:
takashi.yoneuchi@shift-js.info