# EasyCSPeasy: Automatic XSS Prevention (Final Report)

Gianluca Stringhini (PI), Manuel Egele (Co-PI), Beliz Kaleli (RA)

This project funded PhD candidate Beliz Kaleli for two semesters. During this time, Beliz developed a prototype of the EASYCSPEASY system and is in the process of finalizing it. She has also tested EASYCSPEASY on ten popular websites including Facebook and Twitter, showing that our approach works and does not break the website's functionality. In the rest of this document, we first summarize the research conducted during this project. We then discuss our future publication and funding plans.

## I. SUMMARY OF THE COMPLETED RESEARCH

**Introduction.** Web-security is the cornerstone of our online life, and allows us to safely engage in online activities such as shopping, banking, and the management of medical records (e.g., BU's Healthway to curb the spread of COVID-19 on campus[1]). The Content Security Policy (CSP) framework ratified by the World Wide Web Consortium (W3C) has developed into a central pillar to enable a secure and trustworthy Web. Unfortunately, the policy language has become sufficiently expressive and complicated leading to most websites eschewing the use of CSP altogether.[2] As hypothesized by prior work [10], the reason is that defining the policy that guards a given website is a labor-intensive and largely manual task that does not scale well with the ever-changing nature of today's Web. In this project, we developed a novel and automated capability that intelligently builds a security policy for arbitrary websites. Our system first, automatically extracts a fine-grained CSP based on a website's code. Second, it rewrites the source code of the web-application to ensure developer-intended full functionality of the website.

**Background.** The Open Web Application Security Project (OWASP) periodically publishes a list of Top-10 [2] web security threats. Cross-site scripting (XSS) is constantly featured in this list. Despite significant research and engineering efforts to thwart XSS vulnerabilities, the continuous presence on the Top-10 list is testament to the fact that XSS remains a prolific threat to web-security. At its core, XSS is an injection vulnerability where an adversary tricks a victim's browser to interpret the attacker's injected script code to execute in the context of a benign but buggy victim website. The consequences of a XSS attack are dire as the adversary can commonly access the victim's authentication tokens (cookies) and hence gain the capability to impersonate the victim on the website (e.g., to commandeer an online banking account). The most popular countermeasure against XSS is the *Content Security Policy* (CSP) [1] which essentially restricts what script content a user's browser should execute when visiting a website. One of the main challenges for the widespread deployment of CSP is the process of *policy generation*. Modern websites are an amalgamation of content originating from a slew of providers (e.g., social interaction buttons, advertisements, or scripts to monitor visitor statistics), and the web-developer cannot easily know what script content will appear on their site. This makes the development of effective policies particularly challenging. CSP is designed to allow the web server to instruct the web browser to restrict the resources that a page can load such as images, scripts, and objects. Each resource type has a directive in CSP in which the allowed domains or other CSP options are set. The following is an example of a very simple CSP policy which permits the loading of scripts from `example.com` and prevents the loading of images from any domain: `Content-Security-Policy: script-src http://example.com; img-src 'none'`. The responsibility to configure a CSP policy falls on the developer of a Web application. However, defining such a policy is challenging, especially in the context of complicated real-world web applications which use many external resources. Previous research [1] found that the average web developer lacks sufficient understanding of CSP to design a secure policy. As a consequence, developers either implement an overly-permissive (insecure) policy or omit the policy altogether. It is therefore not surprising that the adoption of CSP in the wild is low [10] (around 10% for Alexa top 10K [3]). To help web developers devise secure CSPs, researchers have proposed two automated approaches. Pan et al. [8], introduced CSPAutoGen, a tool that trains so-called templates for each domain, generates CSPs based on the templates, rewrites incoming webpages on the fly to apply those generated CSPs, and then serves those rewritten webpages to client browsers. An issue with this approach is, CSPAutoGen has to run an additional server to do the template-matching and to store scripts. This additional server also needs security measures. Moreover, the whole template-matching and rewriting process should execute for each url fetch. These two phenomena complicate the deployment of CSPAutoGen which adds more confusion to CSP adoption. Prior research has demonstrated that CSP policies that exclusively rely on allowlists are inherently insecure and bypassable [11], for example by exploiting *open redirect vulnerabilities*. Calzavara et al. [6], presented CCSP in which they generate a final CSP which is the lowest bound of the CSPs provided by the web developer and the content provider. This approach however still requires considerable manual effort by web developers as well as content providers, and is still based on allowlists. To prevent allowlist vulnerabilities, CSP Level 2 introduced nonces [5], random strings generated by the server every time a website is requested. When a nonce is specified in the policy, all scripts which carry that nonce as an attribute are allowed to execute.

[1]Note that the PIs identified a number of significant web-security flaws in BU's Healthway portal. After we responsibly disclosed these issues to BU's security team, the underlying problems have been addressed by the vendor, thus leading to an overall more secure Healthway experience.

[2]A non-representative analysis shows that neither the BU, nor the CISE, nor the Healthway websites make use of CSP.

**System Overview.** The EASYCSPEASY system automatically derives CSP policies to prevent XSS while allowing the website to operate as intended by its developer. The CSP specification states that when multiple policies are present, each of them has to be enforced. Our system defines one nonce-based policy to check the script nonce attributes and one allowlist-based policy to check the script sources. This multiple CSP approach can mitigate most types of XSS attacks such as so-called script gadget attacks [7] along with other known attacks such as Open Redirect and JSONP vulnerabilities. To automatically build the two policies, EASYCSPEASY has to tackle two challenges. The first challenge is to identify an exhaustive list of domain names from where the scripts are being loaded. EASYCSPEASY distills that list into the first policy that contains the allowlist. Second, it needs a mechanism to recognize scripts that "belong" on the website. EASYCSPEASY then rewrites such scripts with a nonce and hence allows their execution. EASYCSPEASY system has the following three phases to accomplish these challenges: Crawling, CSP Generation and Rewriting.

*Crawling:* We leverage *python-sitemap*, a python library to generate a sitemap for a given seed url. We modified the source code of this library to output a file called *urls.txt* that contains all public urls of the given website. This file will later on be used by the CSP Generator as an input. CSP Generator will visit each url inside *urls.txt* to collect script sources. Our crawler is also capable of taking cookies as input and crawl behind-the-login pages.

*CSP Generation:* The goal of CSP generator is to collect script resource urls given a clean target url to devise the allowlist. CSP Generator takes *urls.txt* as an input and visits all urls with a headless Chrome instance. We use the *seleniumwire* python library as a man-in-the-middle proxy to catch the requests and responses made while visiting the urls. The request interceptor of CSP Generator, intercepts the request to add the cookies and records the requests that are made to same-party or third-party javascript resources.

*Rewriting:* In the rewriting phase, we have the following two main goals: modify the source code of a web-application to add nonces to scripts so that CSP will recognize them as benign and allow execution and set the proper CSP header for the webpage.

A web-application can be based on different backend languages such as PHP and Ruby. It can also be built on a content management system (CMS) such as Wordpress and Drupal. According to W3C [9], 36% of websites do not use any CMS and 65% of websites that use one, use Wordpress which is based on PHP so we decided to build our system for PHP. Our python-based program goes over all PHP files in the source code and replaces script tags with a script tag concatenated with the nonce attribute.

One of the ways to set a CSP header is to use PHP `header()` function. The proper location to add this function can change according to the used CMS. For example, by default, Wordpress has a file called *header.php* where we can add the desired CSP configuration. The location of this *header.php* file is always the same by default and should not be altered. Hence, to set the CSP pairs, we locate this file and add our generated CSP pair by using two `header()` functions.

**Evaluation of Effectiveness.** The source code of real world popular websites is not publicly available. Testing our approach on these websites is however important, since it would be an indication that EASYCSPEASY can protect them from XSS without breaking their functionality. To addres this issue we developed a proxy testing system, and tested it on popular websites obtained from the Tranco [4] ranking. Our proxy-system outputs a similar HTML code given a URL to our EASYCSPEASY system. The difference between them is that the proxy-system modifies the response from the server on the fly when a webpage is requested by the client whereas EASYCSPEASY runs on the server-side once per web-application. We implemented the proxy-system in python using the *mitmproxy* library to intercept the requests and the responses. We used the *BeautifulSoup* package to parse the server responses. The process of the proxy-system is similar to EASYCSPEASY. We assume the given url is clean (does not contain attacker injected code) since EASYCSPEASY runs on the server-side, urls are guaranteed to be clean. We visit each input url and collect script resources then we modify the server response to add nonces to scripts and add the curated CSP to the HTML response. We tested 10 pages each for 10 popular websites including *twitter.com, facebook.com and reddit.com*. By manual analysis, we observed that 9 out of 10 websites were still fully-functional when modified with our proxy-system, indicating that EASYCSPEASY could be deployed on those websites.

We are now in the process of implementing a version of EASYCSPEASY that automatically re-writes the source code of popular content management system applications like Wordpress and Drupal to set up our CSP policies.

## II. PUBLICATION AND FUTURE FUNDING PLANS

**Publication plans.** We plan to wrap up the work on EASYCSPEASY by the end of the Fall semester. We will then submit a paper on the system and its results to a computer security conference early 2022. Potential venues include the ACM Conference on Computer and Communications Security (CCS) and the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA). This work will also become part of Beliz's PhD dissertation.

**Funding plans.** As discussed in the original proposal, now that we have a working prototype of our anti-XSS solution we plan to write a grant proposal to get additional funding and continue working in this space. More precisely, we plan to send a whitepaper to our contacts at ONR, ARL, and AFOSR to gauge their interest in the project. We also plan to write an NSF proposal and submit it to the Secure and Trustworthy Cyberspace (SaTC) program in Spring 2022.

## REFERENCES

[1] "Content security policy level 3," https://w3c.github.io/webappsec-csp/#multiple-policies.
[2] "Owasp top ten," https://owasp.org/www-project-top-ten/.
[3] "The top 500 sites on the web." https://www.alexa.com/topsites.
[4] "Tranco," https://tranco-list.eu/.
[5] "W3c," https://www.w3.org/.
[6] S. Calzavara, A. Rabitti, and M. Bugliesi, "CCSP: Controlled relaxation of content security policies by runtime policy composition," in *Proceedings of USENIX Security Symposium*, Vancouver, BC, August 2017.
[7] S. Lekies, K. Kotowicz, S. Groß, E. A. Vela Nava, and M. Johns, "Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Texas, USA, October 2017.

[8] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou, “Cspautogen: Black-box enforcement of content security policy upon real-world websites,” in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[9] Q-Success, “Usage Statistics and Market Share of Content Management Systems for Websites,” https://w3techs.com/technologies/overview/content_management/all, Aug. 2019.

[10] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, “Complex security policy? a longitudinal analysis of deployed content security policies,” in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, California, USA, February 2020.

[11] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, “Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy,” in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.