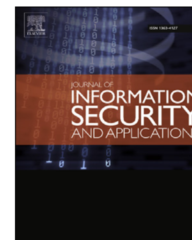
JOURNAL OF INFORMATION SECURITY AND APPLICATIONS 19 (2014) 209–223

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/jisa



Script-templates for the Content Security Policy

Martin Johns

SAP Research, Germany

ARTICLE INFO

Article history:
Available online 20 August 2014

Keywords:
Cross-site Scripting
XSS
Content Security Policy
CSP
Secure coding
Web application security

ABSTRACT

Content Security Policies (CSPs) provide powerful means to mitigate most XSS exploits. However, CSP's protection is incomplete. Insecure server-side JavaScript generation and attacker control over script-sources can lead to XSS conditions which cannot be mitigated by CSP. In this paper we propose PreparedJS, an extension to CSP which takes these weaknesses into account. Through the combination of a safe script templating mechanism with a light-weight script checksumming scheme, PreparedJS is able to fill the identified gaps in CSP's protection capabilities.

© 2014 Elsevier Ltd. All rights reserved.

1. Introduction

1.1. Motivation

Cross-site Scripting (XSS) is one of the most prevalent security problems of the Web. It is listed at the second place in the OWASP Top Ten list of the most critical Web application security vulnerabilities (Open Web Application Project (OWASP), 2010). Even though the basic problem has been known since at least 2000 (CERT/CC, 2000), XSS still occurs frequently, even on high-profile Web sites and mature applications (Scholte et al., 2012). The primary defense against XSS is secure coding on the server-side through careful and context-aware sanitization of attacker provided data (Open Web Application Project (OWASP), 2012). However, the apparent difficulties to master the problem on the server-side have let to investigations of client-side mitigation techniques.

A very promising approach in this area is the Content Security Policy (CSP) mechanism, which is currently under active development and has already been implemented by the Chrome and Firefox Web browsers. CSP provides powerful tools to mitigate the vast majority of XSS exploits.

However, in order to properly benefit from CSP's protection capabilities, site owners are required to conduct significant changes in respect to how JavaScript is used within their Web application, namely getting rid of inline JavaScript, such as event handlers in HTML attributes, and string-to-code transformations, which are provided by `eval()` and similar functions (see Section 2.2 for further details). Unfortunately, as we will discuss in Section 3, all this effort does not result in complete protection against XSS attacks. Some potential loopholes remain, which cannot be closed by the current version of CSP.

Listing 1 CSP example

```
Content-Security-Policy: default-src 'self'; img-src *;
object-src media.example.com;
script-src trusted.example.com;
```

1.2. Contribution and paper outline

In this paper, we explore the remaining weaknesses of CSP (see Section 3) and examine which steps are necessary to fill the identified gaps for completing CSP's protection capabilities. In Section 4, we show that the widespread JSONP coding convention is especially problematic and report on an empirical study that examines how widespread this potential vulnerability is. Based on our results, we propose PreparedJS, an extension of the CSP mechanism (see Section 6). PreparedJS is built on two pillars: A templating format for JavaScript

E-mail address: mj@martinjohns.com.
http://dx.doi.org/10.1016/j.jisa.2014.03.007
2214-2126/© 2014 Elsevier Ltd. All rights reserved.

which follows SQL's prepared statement model (see Section 6.1) and a light-weight script checksumming scheme, which allows fine-grained control over permitted script code (see Section 6.2). In combination with the base-line protection provided by CSP, PreparedJS is able to prevent the full spectrum of potential XSS attacks. We outline how PreparedJS can be realized as a native browser component while providing backwards compatibility with legacy browsers that cannot handle PreparedJS's script format. Furthermore, we report on a prototypical implementation in the form of a browser extension for Google Chrome (see Section 7). In Section 9 we show how the basic mechanism can be extended with a lightweight macro meta syntax to enable flexible script assembly. And finally, in Section 10 we discuss non-security benefits of PreparedJS in the area of network traffic and caching.

2. Technical background

2.1. Cross-site Scripting (XSS)

The term *Cross-site Scripting (XSS)* (The webappsec mailing list, 2002) summarizes a set of attacks on Web applications that allow an adversary to alter the syntactic structure of the application's Web content via code or mark-up injection.

Even though, XSS in most cases also enables the attacker to inject HTML or CSS into the vulnerable application, the main concern with this class of attacks is the injection of JavaScript. JavaScript injection actively circumvents all protective isolation measures which are provided by the same-origin policy (Ruderman, 2001), and empowers the adversary to conduct a wide range of potential attacks, ranging from session hijacking (Nikiforakis et al., 2011), over stealing of sensitive data (Vogt et al., 2007) and passwords (Toews, 2012), up to the creation of self-propagating JavaScript worms.

To combat XSS vulnerabilities, it is recommended to implement a careful and robust combination of input validation (only allow data into the application if it matches its specification) and output sanitation (encode all potential syntactic content of untrusted data before inserting it into an HTTP response). However, a recent study (Scholte et al., 2012) has shown, that this protective approach is still error prone and the quantitive occurrence of XSS problems is not declining significantly.

2.2. Content Security Policies (CSPs)

Due to the fact, that even after several years of increased attention to the XSS problem, the number of vulnerabilities remains high, several reactive approaches have been proposed, which mitigate the attacks, even if a potential XSS vulnerability exists in a Web application.

Content Security Policies (CSPs) (Stamm et al., 2010) is such an approach: A Web application can set a policy that specifies the characteristics of JavaScript code which is allowed to be executed.[1] CSP policies are added to a Web document through an HTTP header or a `Meta`-tag (see Lst. 1 for an example). More specifically, a CSP policy can:

1. Disallow the mixing of HTML mark-up and JavaScript syntax in a single document (i.e., forbidding inline JavaScript, such as event handlers in element attributes).
2. Prevent the runtime transformation of string-data into executable JavaScript via functions such as `eval()`.
3. Provide a list of Web hosts, from which script code can be retrieved.

If used in combination, these three capabilities lead to an effective thwarting of the vast majority of XSS attacks: The forbidding of inline scripts renders direct injection of script code into HTML documents impossible. Furthermore, the prevention of interpreting string data as code removes the danger of DOM-based XSS (Klein, 2005). And, finally, only allowing code from whitelisted hosts to run, deprives the adversary from the capability to load attack code from Web locations that are under his control.

In summary, strict CSP policies enforce a simple yet highly effective protection approach: Clean separation of HTML-markup and JavaScript code in connection with forbidding string-to-code transformations via `eval()`. The future of CSP appears to be promising. The mechanism is pushed into major Web browsers, with recent versions of Firefox (since version 4.0) and Chrome (since version 13) already supporting it. Furthermore, CSP is currently under active standardization by the W3C (W3C, 2012).

However, using CSP comes with a price: Most of the current practices in using JavaScript, especially in respect to inline script and using `eval()`, have to be altered. Making an existing site CSP compliant requires significant changes in the codebase, namely getting rid of inline JavaScript, such as event handlers in HTML attributes, and string-to-code transformations, which are provided by `eval()` and similar functions.

3. CSP's remaining weaknesses

In general, CSP is a powerful mitigation for XSS attacks. If a site issues a strong policy, which forbids inline scripts and unsafe string-to-code transforms, the vast majority of all potential exploits will be robustly prevented, even in the presence of HTML injection vulnerabilities.

However, as we will show in this section, three potential attack variants remain feasible under the currently standardized version 1.0 of CSP (W3C, 2012). Furthermore, in Section 3.4, we will discuss to which degree the proposed enhancements of CSP 1.1 affect these identified weaknesses.

3.1. Weakness 1: insecure server-side assembly of JavaScript code

As described above, CSP can effectively prevent the execution of JavaScript which has been dynamically assembled on the client-side. This is done by forbidding all functions that convert string data to JavaScript code, such as `eval()` or `setTimeout()`. However, if a site's operator implements

[1] CSP also provides further features in respect to other HTML elements, such as images or iframe. However, these features do not affect JavaScript execution and, hence, are omitted in the CSP description for brevity reasons.

Download English Version:

https://daneshyari.com/en/article/457075

Download Persian Version:

https://daneshyari.com/article/457075

Daneshyari.com