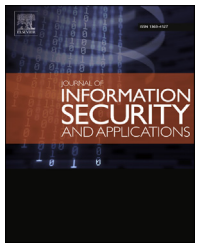
Available online at www.sciencedirect.com

ScienceDirect

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



Script-templates for the Content Security Policy

Martin Johns

SAP Research, Germany

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.

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/

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.

Please cite this article in press as: Johns M, Script-templates for the Content Security Policy, Journal of Information Security and Applications (2014), http://dx.doi.org/10.1016/j.jisa.2014.03.007

dynamic script assembly on the server-side, this directive is powerless.

Server-side generated JavaScript is utilized to fill values in scripts with data that is retrieved at runtime. If such data can be controlled by the attacker, he might be able to inject further JavaScript.

Take for instance the scenario that is outlined in Listings 2 and 3: A script-loader JavaScript (`loader.js`, Lst. 2), is used to dynamically outfit further JavaScript resources with runtime data via URL parameters.[2] The referenced script (`ga.php`, Lst. 3) is assembled dynamically on the server-side, including the dynamic data in the source code without any sanitization.

If the attacker is able to control the `document.location` property, he can break out of the variable assignment in line 5 and inject arbitrary JavaScript code. Thus, he can effectively circumvent CSP's protection features: The attack uses no string-to-code conversion on the client-side. All the browser retrieves is apparently static JavaScript. In addition, the attack does not rely on inline scripts, as the injected script is included externally. Finally, the vulnerable script is part of the actual application and, hence, the script's hosting domain is included in the policy's whitelist.

Listing 2 JavaScript for dynamic script loading (`loader.js`)

```
(function() {
  var ga = document.createElement('script');
  ga.src = 'http://serv.com/ga.php?source='+
document.location;
  var s = document.getElementsByTagName('script')
[0];
  s.parentNode.insertBefore(ga, s);
})();
```

Listing 3 Variable setting script (`ga.php`)

```
// JS code to set a global variable with the
// request's call context
<?php
$s = '$_GET["source"]';
echo "var callSource='".$s."';";
?>
// [...rest of the JavaScript]
```

3.2. Weakness 2: full control over external, whitelisted scripts

It is common practice to include external JavaScript components from third party hosts into Web applications. This is done to consume third party services (such as Web analytics), enhance the Web application with additional functionality (e.g., via integrating external mapping services), or for monetary reasons (i.e, to include advertisements).

Recently Nikiforakis et al. conducted a wide scale analysis on the current state of cross-domain inclusion of third party JavaScripts (Nikiforakis et al., 2012). Their survey showed that 88.45% of the Alexa top 10,000 Web sites included at least one remote JavaScript. If the attacker is able to control the script's content, which is provided by the external provider, he is able to execute JavaScript in the context of the targeted Web application.

A straight forward scenario for such an attack is a full compromise of one of the external script providers for the targeted site. In such a case, the adversary is able to inject and execute arbitrary JavaScript in the context of targeted application. To examine this potential threat, Nikiforakis et al. created a security metric for script providers, which is based on indicators for maintenance quality of the hosts. Subsequently, they compared the security score of the including sites to the score of the consumed script providers: In approximately 25% of all cases, the security score of the script provider was lower than the score of the consumer, suggesting that a compromise of the script provider was more likely than a compromise of the targeted Web application.

As alternatives to a full compromise of the script provider, Nikiforakis et al. list another four, more subtle attacks which enable the same class of script inclusion attacks and show their practical applicability (see (Nikiforakis et al., 2012) for details).

CSP is not able to protect against such cases: To utilize external JavaScript components, a CSP-protected site has to whitelist the script provider's domain in the CSP policy. However, as the adversary is able to control the contents of the whitelisted host, he is able to circumvent CSP's protection mechanism.

3.3. Weakness 3: injection of further script-tags

This class of potential CSP circumvention was first observed by Zalewski (2011): Given an HTML-injection vulnerability, a strict CSP policy will effectively prevent the direct injection of attacker-provided script code. However, he still is able to inject HTML markup including further `script`-tags pointing to the whitelisted domains.

This way an attacker is able to control the URLs of included scripts and the order in which they are retrieved. Thus, he might be able to combine existing scripts in an unforeseen fashion. All scripts in a Web page run in the same execution context. JavaScript provides no native isolation or scoping, e.g., via library specific name-spaces. Hence, all side effects that a script causes on the global state directly affect all scripts that are executed subsequently. Given the growing quantity and complexity of script code hosted by Web sites, a non-trivial site might provide an attacker with a well equipped toolbox for this purpose. Also, the adversary is not restricted to the application's original site. Scripts from all domains that are whitelisted in the CSP-policy can be combined freely.

Only little research has been conducted to validate this class of attacks. Nonetheless, such attacks are theoretically possible. Furthermore, with the ever-growing reliance on client-side functionality and the rising number of available JavaScripts their likelihood can be expected to increase.

3.4. CSP 1.1's script-nonce directive

The 1.0 version of CSP currently holds the status of a W3C "Candidate Recommendation". This means the significant

[2] The depicted code was consciously designed in a naive fashion for better understanding. In more realistic conditions, the attacker controlled data could find its way into the script assembly in more subtle fashions, e.g., through existing data in the user's session.

Please cite this article in press as: Johns M, Script-templates for the Content Security Policy, Journal of Information Security and Applications (2014), http://dx.doi.org/10.1016/j.jisa.2014.03.007

features of the standard are mostly locked and are very unlikely to change in the further standardization process. Hence, major changes and new features of CSP will happen in the subsequent versions of CSP. The next iteration of the standard is CSP version 1.1, which is currently under active discussion (W3C, 2012).

Among other changes, that primarily focus on the data exfiltration aspect of CSP, the next version of the standard introduces a new directive called `script-nonce`. This directive directly relates to a subset of the identified weaknesses of CSP 1.0. In case, that a site's CSP utilizes the `script-nonce` directive (see Lst. 4), the policy specifies a random value that is required to be contained in all `script`-tags of the site. Only JavaScript in the context of a `script`-tag that carries the nonce value as an attribute value is permitted to be executed (see Lst. 5). For apparent reasons, a site is required to renew the value of the nonce for every request. Please note, that the nonce is not a signature or hash of the script nor has it other relations to the actual script content. This characteristic allows the usage of the directive to reenable inline scripts (as depicted in Lst. 5) without significant security degradation.

Listing 4 CSP 1.1 policy requiring script-nonce

```
Content-Security-Policy: script-src 'self';
  script-nonce A3F1G1H41299834E;
```

Listing 5 Exemplified usage of script-nonce

```
<script nonce="A3F1G1H41299834E">
  alert("I execute! Hooray!");
</script>
<script> alert("I don't execute. Boo!"); </script>
```

Effect on the identified weaknesses: The `script-nonce` directive effectively prevents the attacker from injecting additional `script`-tags into a page, as he won't be able to insert the correct nonce value into the tag. In this section, we examine to which degree the directive is able to mitigate the identified weaknesses:

Unsafe script assembly: To exploit this weakness, an attacker is not necessarily required to inject additional `script`-tags into the page. The unsafe script assembly can also happen in legitimate scripts due to attacker controlled data which was transported through session data or query parameters set by the vulnerable application itself.

Adversary controlled scripts: In such cases, the directive has no effect. The script import from the external host is intended from the vulnerable application. Hence, the corresponding `script`-tag will carry the nonce and, thus, is permitted to be executed.

Adversary controlled script tags: This weakness can be successfully mitigated through the directive. As the attacker is not able to guess the correct nonce value, he cannot execute his attack through injecting additional `script`-tags.

Only the third weakness can be fully mitigated through the usage of script-nonces. The reason for the persistance of the other two problems, lies in the missing relationship between the nonce and the script content. A further potential downside of the `script-nonce` directive is that it requires dynamic creation of the CSP policy for each request. Hence, a rollout of a well audited, static policy is not possible.

3.5. *Analysis*

The discussed CSP weaknesses are caused by two characteristics of the policy mechanism:

1. A site can only specify the origins which are allowed to provide script content, but not the actually allowed scripts.
2. Even if a site would be able to provide more fine-grained policies on a per-script-URL level, at the moment there are no client-side capabilities to reason about the validity of the actual script content.

The first characteristic is most likely a design decision which aims to make CSP more easily accessible and maintainable to site-owners. It could be resolved through making the CSP policy format more expressive. However, the second problem is non-trivial to address, especially in the presence of dynamically assembled scripts.

4. JSONP inclusion

During our examination of the theoretical limitations of CSP and a subsequent study of real-life code, we encountered a widespread coding practice which amplifies the previously identified weaknesses: *JSONP*. In this section, we discuss the special conditions that occur if JSONP and CSP are used on the same document. Furthermore, we report on an empirical study, in which we examined how common this coding practice is.

4.1. *Technical background: JSONP*

The term *JSONP* is short for "JSON with padding" (Oezses and Erguel, 2009), with the abbreviation *JSON* standing for "JavaScript Object Notation" (Crockford, 2006). JSONP is a common method for data sharing among cross-domain Web applications. In general, client-side cross-domain exchange of data is prohibited by the Same-origin Policy (Ruderman, 2001). Hence, exporting cross-domain HTTP interfaces requires explicit whitelisting of the resource using either Flash's policy or via Cross-origin Resource sharing (CORS) (Johns and Lekies, 2011).

JSONP circumvents the limits imposed by the SOP through exploiting the fact that the targets of HTML tags are not subject to origin restrictions. Through pointing `script`-tags to the data-producer, information can be retrieved through client-side JavaScript (see Lst. 6).

Listing 6 Using JSONP for cross-domain information retrieval (Schock, 2013)

```
<script>
function processJSON (json) {
  // Do something with the JSON response
};
</script>
<script     src='http://api.flickr.com/services/
feeds/photos_
public.gne?jsoncallback=
processJSON&tags=monkey&format=json'>
</script>
```

Please cite this article in press as: Johns M, Script-templates for the Content Security Policy, Journal of Information Security and Applications (2014), http://dx.doi.org/10.1016/j.jisa.2014.03.007

Unlike most cross-domain script includes (Nikiforakis et al., 2012), JSONP provides the including site to specify a *callback parameter*. Using this parameter in the script-URL, the including page tells the data producer to call the referenced function, with the requested data as the function's argument. The data itself is then passed in JSON encoded form. For this to function, the callback function has to be previously defined by the site's JavaScript and reside in the global object space.

Example: Please refer to Listing 6. First the integrator page defines the function `processJSON`, which expects its argument to be JSON encoded data. Then the page includes a cross-domain JSONP resource, pointing to the server `flickr.com`. In the `jsoncallback` parameter, the name of the local function is given. Upon receiving the request for the JSONP resource, the data producer compiles the requested information and encodes them in JSON format. Then it creates a JavaScript that at least consists of calling the specified callback function, i.e., `processJSON` in our case, with the JSON as argument.

4.2. JSONP and the CSP

In case a CSP protected site utilizes a JSONP service, the host of this service necessarily has to be whitelisted in the CSP's `script-src` directive, as otherwise, the JSONP script-include is prevented by the policy enforcement mechanism.

Now consider the case, that this page has an HTML injection problem. This results in a combination of weaknesses 1 (insecure server-side assembly of JavaScript code) and weakness 3 (injection of further script-tags). For one, the attacker can inject further `script`-tags pointing to the data provider. Furthermore, if these `script`-tags point to the JSONP endpoint, he is able to influence the script content.

At least, he is able to specify arbitrary JavaScript function as the JSONP callback. This leads to calling these functions upon script inclusion. Through chaining several of such script includes, the attacker could manufacture a series of function calls which ultimately lead to the goal of his attack. For this to be possible depends on the availability of suitable functions in the page's current global scope and the exact data format of the JSON argument. However, the attacker's capabilities to execute a "JavaScript ROP" attack, as outlined in Section 3.3, increase significantly, as his attack no longer depends on implicit side effects but also now can execute functions directly.

A worse scenario occurs in situations, in which the script provider does not conduct validation steps in respect to the callback parameter. In this case, the attacker has a full insecure server-side assembly weakness, as discussed in Section 3.1, to inject his payload into the included script code.

Before the introduction of CSP, a missing validation of the callback parameter is not necessarily a security problem by itself. To inject JavaScript this way, the attacker would need an HTML injection vulnerability, to inject a `script`-tag pointing to the JSONP resource. However, if the attacker can already inject `script`-tags, he can in the absence of a CSP, inject his script payload directly. And also the JSONP provider is secure, as long as the JSONP is delivered with the correct mime-type, such that the browser would not interpret the response body as HTML. Hence, it is questionable that the current JSONP interfaces implement robust sanitization. See Section 4.4 for empirical results that confirm this suspicion.

4.3. Experimental set-up

To examine how common the usage of JSONP is, we conducted an empirical study in July 2013. In this study, we examined the homepages of the top 45,000 sites in the Alexa.com ranking (Alexa Internet and Inc, 2013). Using a crawling engine based on the headless Web browser HTMLUnit,[3] each of these Web pages was retrieved and rendered. During the rendering process, all JavaScript includes were recorded.

Out of the recorded script-includes, JSONP-includes were identified as follows: First, it was verified, that the script's URL carries at least one GET parameter. If this is the case, the GET parameters were extracted from the URL. For each of these parameters it was verified that:

1. The parameter's value can be found as a function name in the current JavaScript global space.
2. The parameter's value is used in the form of a function call in the retrieved script code.
3. If the parameter's value is changed, the corresponding function call in the retrieved script code changes accordingly. This step was conducted out of the context of the crawling step. Instead, the script was requested a second time using a modified GET parameter.

4.4. Results

On the 45.000 examined home pages 588,939 script includes could be found, averaging in approximately 13 script included per page. Out of the 588,939 scripts 180,794 carry at least one GET parameter, hinting that some form of server-side dynamic script assembly is conducted. Of these scripts a total of 12,378 satisfy our criteria for a JSONP call. These scripts are spread over 6586 site, resulting in 14.64% of the examined sites using JSONP.

Furthermore, we checked, which of the found JSONP providers conduct sanitization steps on the callback parameter. For this, we requested the JSONP scripts with a URL that provided an XSS payload as its callback parameter. Out of all 45,000 sites that we examined, 5299 (11,78%) included at least one insecure JSONP script that inserted the callback parameter without validation, and, hence allowed the attacker to inject his XSS payload directly in the JSONP.

For these 5299 sites, a given CSP policy would be incapable to prevent the XSS exploitation, as the JSONP provider necessarily is whitelisted in the policy and the attacker can inject arbitrary JavaScript into the script include via the callback parameter.

5. Objective: stable cryptographic checksums for scripts

As deducted above, all existing loopholes which allow the circumvention of CSP can be reduced to the fact that no reliable link exists between the policy and the actual script code. Hence, a mechanism is needed that allows site owners to

[3] HtmlUnit: http://htmlunit.sourceforge.net.

Please cite this article in press as: Johns M, Script-templates for the Content Security Policy, Journal of Information Security and Applications (2014), http://dx.doi.org/10.1016/j.jisa.2014.03.007

clearly define which exact scripts are allowed to be executed. And, as seen in Section 3.1, this specification mechanism should not only rely on a script's URL. It should also take the script's content into consideration.

A straight forward approach to solve this problem is utilizing script signatures or cryptographic checksums, that are calculated over the scripts' source code: On deploy-time the checksums of all legitimate JavaScripts are generated and are included in an extended CSP policy. At runtime, this policy is communicated to the browser which in turn only allows the execution of scripts with correct checksums. This technique works well as long as only static scripts are utilized.

Unfortunately, this approach is too restrictive. As soon as the need for dynamic data values during script assembly occurs, the mechanism cannot be applied anymore: The source code of the scripts is non-static and, hence, creating source code checksums on deploy-time is infeasible. However, creating these checksums at runtime defeats their purpose, as in such cases in-script injection XSS (see Section 3.1) will be included in the checksum and, thus, the browser will allow the script to be executed.

Therefore, a secure mechanism is needed which allows the creation of stable cryptographic checksums of script code while still allowing a certain degree of flexibility in respect to run-time script creation.

6. PreparedJS

In this section, we present PreparedJS – our approach to fill the identified weaknesses of CSP. PreparedJS is built on two pillars:

- A *templating mechanism*, that enables developers to separate dynamic data values from script code, thus, allowing the usage of purely static scripts without losing needed flexibility,
- and a *script checksumming scheme*, that allows the server to non-ambiguously communicate to the browser which scripts are allowed to run.

As the name of our mechanism suggests, the templating mechanism is inspired by SQL's prepared statements: In a prepared statement, the query syntax is separated from the data values, using placeholders. At runtime, this statement is passed to the database together with a set of values which are to be used within the query at the placeholders' position. This way, the statement can be outfitted with dynamic values. As the syntactic structure of the statement has already been processed by the database engine, before the placeholders are exchanged with the data values, code injection attacks are impossible.

Following the prepared statement's model, PreparedJS defines a JavaScript variant which allows placeholder for data values, which will be filled at runtime in a fashion that is unsusceptible to code injection vulnerabilities (see Section 6.1 for details). This way, developers can create completely static script source code, for which the calculation of stable cryptographic checksums on deploy-time is feasible. While the Web application is accessed, only scripts which have a valid checksum are allowed to run: If the checksum checking terminates successfully, the data values, which are retrieved along with the script code, are inserted into the respective placeholders, thus, creating a valid JavaScript, that can be executed by the Web browser.

6.1. JavaScript templates for static server-side scripts

In this section, we give details on the PreparedJS templating mechanism. The mechanism consists of two components: The *script template* and the *value list*.

The PreparedJS *script template* format supports using insertion marks in place of data values. These placeholders are named using the syntactic convention of framing the placeholders identifier with question marks, e.g., `?name?` Such placeholders can be utilized in the script code, wherever the JavaScript grammar allows the injection of data values. See Listing 7 for a template which represents the dynamic script of Listing 3.

Listing 7 PreparedJS variable setting script (`ga.js`)

```
// JS code to set a global variable with the
// request's call context
var callSource= ?source?;
// [...rest of the JavaScript]
```

The PreparedJS *value list* contains the data values, which are to be applied during script execution in the browser. The list consists of identifier/value pairs, in which the identifier links the value to the respective placeholder within the script template. The values can be either basic datatypes, i.e., strings, booleans, numbers, or JSON (JavaScript Object Notation (Crockford, 2006)) formated complex JavaScript data objects. The latter option allows the insertion of non-trivial data values, such as arrays or dictionaries.

Also, the value list itself follows the JSON format, which is very well suited for this purpose: The top level structure represents a key/value dictionary. By using the placeholder identifiers as the keys in the dictionary, a straight forward mapping of the values to insertion points is given. Furthermore, JSON is a well established format with good tool, language, and library support for creation and verification of JSON syntax. See Listing 8 for a PHP-script which creates the value list for Lst. 7 according to the dynamic JavaScript assembly in Lst. 3.

In the communication with the Web browser, the script template and the value list are sent in the same HTTP response, using an HTTP multipart response (see Lst. 9).

Listing 8 Creating value list for Lst. 7 (`ga_values.php`)

```
<?php
$source = $_GET["source"];
$vals = array('callSource' => $source);
echo json_encode($vals);
?>
```

6.2. Code legitimacy checking via script checksums

As discussed in Section 3, parts of the existing shortcomings of CSP result from the mechanism's inability to specify which

Please cite this article in press as: Johns M, Script-templates for the Content Security Policy, Journal of Information Security and Applications (2014), http://dx.doi.org/10.1016/j.jisa.2014.03.007

exact scripts are allowed to run in the context of a given Web page. Within PreparedJS we fill the gap by unambiguously identifying whitelisted scripts through their *script checksums*.

A script's PreparedJS-checksum is a cryptographic hash calculated over the corresponding PreparedJS script template. The script's value list is not included in the calculation. This allows a script's values to change on run time without affecting the checksum.

To whitelist a specific script, a policy lists the script's checksums in the policy declaration (see Section 6.3). For each script that is received by the browser, the browser calculates the checksum of the corresponding script template and verifies that it indeed is contained in the policy's set of allowed script checksums. If this is the case, the script is permitted to execute. If not, the script is discarded.

This approach is well aligned with the applicable attacker type. The sole capability of the XSS Web attacker consists of altering the syntactic structure of the application's HTML content. The XSS attacker is not able to alter the application's CSP policy, which is generally transported via HTTP header (if the attacker is able to compromise the site's CSP itself, all provided protection is void anyway). Hence, if the application's server-side can unambiguously communicate to the browser which exact scripts are whitelisted, altering the syntactic structure of the document has no effect.

For this purpose, cryptographic checksums are well suited: The checksum is sufficient to robustly identify the script, as long as a strong cryptographic hash function algorithm, such as SHA256, was used. Due to the algorithm's security properties, it is a reasonable assumption that the attacker is not able to produce a second script which both carries his malicious intend and produces the same checksum.

6.3. *Extended CSP syntax*

For the PreparedJS scheme to function, we require a simple extension of the CSP syntax. In addition to the list of allowed script hosts, also the list of allowed script checksums has to be included in a policy. This can be achieved, for instance, using a comma delimited list of script checksums following directly a whitelisted script host (see Lst. 10 for an example).

Listing 9 PreparedJS HTTP multipart response

```
HTTP/1.1 200 OK
Date: Thu, 23 Jan 2014 10:03:25 GMT
Server: Foo/1.0
Content-Type: multipart/form-data;boundary=xYzZY
-xYzZY
Content-Type: application/pjavascript;
charset=UTF-8
Content-Disposition: form-data;name="preparedJS"
// JS code to set a global variable with the
// request's call context
var callSource = ?callSource?;
-xYzZY
Content-Type: application/json
Content-Disposition: form-data;name="valueList"
{"callSource": "http://serv.com?this=that#
attackerData"}
-xYzZY-
```

6.4. *PreparedJS-aware script tags*

CSP was carefully designed with backward compatibility in mind: If a legacy browser, that does not yet implement CSP, renders a CSP-enabled Web page, the CSP header is simply ignored and the page's functionality is unaffected.

We intend to follow this example as closely as possible. However, as the PreparedJS-format differs from the regular JavaScript syntax (see Lst. 9), the server-side explicitly has to provide backwards compatible versions of the script code. A PreparedJS-aware HTML document utilizes a slightly extended syntax for the `script`-tag. The reference to the PreparedJS-script is given in a dedicated `pjs-src`-attribute. If an application also wants to provide a standard JavaScript for legacy fallback, this script can be referenced in the same tag using the standard `src`-attribute (see Lst. 11). This approach provides transparent backwards compatibility on the client-side: PreparedJS-aware browsers only consider the `pjs-src`-attribute and handle it according to the process outlined above. The legacy script is never touched by such browsers. Older browsers ignore the `pjs-src`-attribute, as it is unknown to them, and retrieve the fallback script referenced by `src`-attribute.

Listing 10 Extended CSP syntax, whitelisting two script checksums

```
X-Content-Security-Policy: script-src 'self'
  (135c1ac6fa6194bab8e6c5d1e7e98cd9,
  2de1cd339756e131e873f3114d807e83)
```

Listing 11 Extended PreparedJS `script`-tag syntax

```
<script src="[path to legacy script]"
  pjs-src="[path to preparedJS script]">
```

Please note: If naively implemented, this approach causes additional implementation effort on the server-side, as all scripts have to be maintained in two versions. However, in Section 7.2 we show, how applications can provide backwards compatibility support for legacy browser automatically.

6.5. *Summary: the three stages of PreparedJS*

PreparedJS affects three stages in an application's lifecycle: The development phase, the deployment phase, and the execution phase:

During development: If the Web application requires JavaScript, with dynamic, run-time generated data values, PreparedJS templates are created for these scripts and methods are implemented to generate matching value lists.

On deployment: For all JavaScripts and PreparedJS templates, which are authorized to run in the context of the Web application, cryptographic checksums are calculated. On application deployment these checksums are added to the site's extended CSP policy.

During execution: Before the execution of regular script code, the CSP policy is checked, if the script's host is whitelisted in the policy and if for this host a list of allowed script checksums is given. If both is the case, the cryptographic checksum for the received script code is calculated and compared with the policy's whitelisted script checksums.

Please cite this article in press as: Johns M, Script-templates for the Content Security Policy, Journal of Information Security and Applications (2014), http://dx.doi.org/10.1016/j.jisa.2014.03.007

Only if the calculated checksum can be found in the policy, the script is allowed to execute.

For scripts in the PreparedJS format, first the script template is retrieved from the multipart response (see Lst. 9). Then, the checksum is calculated over the template. If the checksum test succeeds, the value list is retrieved from the HTTP response and the placeholders in the script are substituted with the actual values. After this step, the script is executed.

7. Implementation and enforcement

In this section, we show how the PreparedJS scheme can be practically realized. In this context, we propose a native, browser-based implementation (see Section 7.1) and discuss how backwards compatibility can be provided for browsers that are not able to handle PreparedJS's template format natively (see Section 7.2).

7.1. *Native, browser-based implementation*

As mentioned earlier, the main motivation behind PreparedJS is to fill the last loopholes that the current CSP approach still leaves for adversaries to inject JavaScript into vulnerable Web applications. For this reason, we envision a native, browser-based implementation of PreparedJS as an extension of CSP.

To execute JavaScript and enforce standard CSP, a Web browser already implements the vast majority of processes which are needed to realize our scheme, namely HTML/script parsing and checking CSP compliance of the encountered scripts. Hence, an extension to support our scheme is straight forward:

Whenever during the parsing process a `script`-tag is encountered, the script's URL is tested, if it complies with the site's CSP policy. Furthermore, if the policy contains script checksums for the URL's host, the checksum for the script's source code is calculated and it is verified, that the checksum is included in the list of legitimate scripts.

In case of PreparedJS templates, first the template code is parsed by the browser's JavaScript parser, treating the placeholders as regular data tokens. Only after the parse tree of the script is established, the placeholders are exchanged with the actual data values contained in the value list. This way, regardless of their content, these values are unable to alter the script's syntactic structure, hence, no code injection attacks are possible.

Prototypical implementation for Google Chrome. To gain insight in practically using PreparedJS's protection mechanism and experiment with the templating format, we conducted a prototypical implementation of the approach in the form of a browser extension for Google Chrome.

Chrome's extension model does not allow direct altering of the browser's HTML parsing or JavaScript execution behavior. Hence, to implement PreparedJS we utilized two capabilities that are offered by the extension model: The network request interception API, to examine all incoming external JavaScripts, and the extension's interface to Chrome's JavaScript debugger, to insert the compiled PreparedJS-code into the respective `script`-tags.

When active, the extension monitors all incoming HTTP responses for CSP headers. If such a header is identified, the extension extracts all contained PreparedJS-checksums and intercepts all further network requests that are initiated because of `src`-attribute in `script` tags in the corresponding HTML document. Whenever such a request is encountered, the extension conducts two actions: First, the actual request is redirected to a specific JavaScript, that causes the corresponding JavaScript thread to trap into Chromes's JavaScript debugger via the `debugger` statement, causing the JavaScript execution to briefly pause until the script legitimacy checking has concluded. Furthermore, the request's original URL is used to retrieve the external JavaScript's source code, or, in the presence of a `pjs-src`-attribute, the PreparedJS-template and value list the extension.

For the retrieved source code or the PreparedJS-template the cryptographic checksum is calculated using the SHA256 implementation of the Stanford JavaScript Crypto Library.[4] If the resulting checksum was not contained in the site's CSP policy, the process is terminated and the script's source code is blanked out. If the checksum was found in the policy, the script is allowed to be executed. In case of a PreparedJS-template, the template is parsed and the items of the value-list are inserted in the marked positions. To re-insert the resulting script code into the Web page, the extension uses Chrome's JavaScript debugger and the Javascript execution is resumed.

Performance measurements: Using our prototypical implementation, we conducted measurements to gain first insight into the runtime characteristics of the proposed mechanism. For several reasons, the obtained results can be regarded as an absolut worst case measurement: For one, the full implementation, including the template parsing and the checksum calculation, is done in JavaScript instead of native code, resulting in an implementations with inferior performance compared to native code. Furthermore, due to the clean separation between the extension and the browser's native code, every script had to be parsed twice, once by the extension and once by the browser's parser. In a native implementation, the checksumming and value insertion steps would be tightly integrated in the default parsing process. And finally, the Chrome extension model made it necessary to repeatedly conduct costly context-switches into Chrome's debugger.

As it can be seen in Table 1, we conducted three separate measurements of page load times: Without the extension, with the PreparedJS extension, and with an "empty" extension that neither processes the script code nor calculates checksums but traps into the debugger and conducts the network interception steps. This was done to be able to distinguish between the performance cost that is caused by the limitations of Chrome's extension model, i.e., the script redirection and context-switches into the debugger, and the effort that is caused by the actual PreparedJS functionality, namely the calculation of the script checksum and the parsing of the JavaScript code. As the former only occurs because of the implementation method's limitiations and won't occur in a native integration in the browser's CSP implementation, only

[4] Stanford JavaScript Crypto Library: http://crypto.stanford.edu/sjcl/.

Please cite this article in press as: Johns M, Script-templates for the Content Security Policy, Journal of Information Security and Applications (2014), http://dx.doi.org/10.1016/j.jisa.2014.03.007

Table 1 – Performance of the browser extension, mean values over 10 iterations.

Site	Scripts[a]	LoC[b]	Default[c]	Debugger[d]	PJS[e]	Overhead
Local testpage[f]	2	3624	67.9 ms	230.6 ms	309.8 ms	79 ms
Mail.google.com	5	16132	2184.5 ms	2542.8 ms	2691.4 ms	148.6 ms
twitter.com	2	9195	1686.0 ms	2058.8 ms	2112.8 ms	54 ms
Facebook.com	18	31701	2583.8 ms	4067.5 ms	4189.0 ms	121.5 ms

[a] Number of external scripts contained in the page.
[b] Total lines of JS code after de-minimizing.
[c] Loadtime without extension.
[d] Loadtime with extension (debugger only, no script processing).
[e] Load time with full PreparedJS functionality on all external scripts.
[f] Testpage with PreparedJS template, served from the same machine as the test browser.

the additional performance overhead of the latter measurement is relevant in estimating PreparedJS's actual cost (as reflected in the table). To conduct the actual measurements we utilized the *Page Benchmarker*[5] extension, using mean values of ten page load iterations over a standard German household DSL line. Please note: During the tests, *all* encountered external JavaScripts were treated, as if they were PreparedJS-templates and, thus, fully parsed and checksummed. Under realistic circumstances, dynamic server-side script assembly is the exception, and only utilized in specific cases, as it is with JSONP (see Section 4).

In general, we do not expect the PreparedJS approach to cause noticeable performance overhead (an estimate that is backed by the performance evaluation): PreparedJS only takes effect during the initial script parsing steps, in which three new steps are introduced: The cryptographic checksum has to be calculated, value list has to be parsed, and the obtained values have to be inserted for the placeholders. None of these steps requires considerable computing effort: Modern hash-functions are highly optimized to perform very well, the browser's JavaScript engine has already native capabilities for parsing the JSON-formatted value list, and inserting the data values after the script parser's tokenization step is straight forward and does not require sophisticated implementation logic. From here on, the browser's actual JavaScript execution functionality remains unchanged. After script parsing, a PreparedJS script is indistinguishable from a regular JavaScript and all recent performance increases of modern JavaScript engines apply unmodified.

7.2. *Transparently providing legacy support*

As mentioned in Section 6.4, providing a second, backwards compatible version of all scripts can cause considerable additional development and maintenance effort. This in turn might hinder developer acceptance of the measure.

However, providing a backwards compatible version of scripts that only exist in the PreparedJS format can be conveniently achieved with a server-side composition service: Such a service compiles the script-template together with the value list on the fly, before sending the resulting JavaScript to the browser. For this purpose, the service conducts the exact same steps as the browser in the native case (see Fig. 1): It retrieves the template, the value-list, and the list of whitelisted checksums from the Web server. Then it calculates the templates checksum and verifies that the script is indeed in the whitelist. Then it parses the value list and inserts the resulting values into the template in place of the corresponding value identifiers.

Please note: The actual script compiling process has to be carefully implemented to avoid the reintroduction of injection vulnerabilities. For this, the data values have to be properly sanitized, such that they don't carry syntactic content which could alter the semantics of the resulting JavaScript.

Taking advantage of the composition service, the `script`-tags of the application can reference the script in its PreparedJS form directly (via the `pjs-src`-attribute) and utilize a specific URL-format for the legacy `src`-attribute, which causes the server-side to route request through the composition service. For instance, this can be achieved through a reserved URL-parameter which is added to the script's URL, such as `?pjs-prerender=true`. All requests carrying this parameter automatically go through the composition service.

8. Discussion

8.1. *Security evaluation*

In this section, we verify that PreparedJS indeed closes CSP's existing protection gaps, as identified in Section 3.

(1) **Insecure server-side assembly of JavaScript code:** Vulnerabilities, such as discussed in Section 3.1 and shown in Lst. 2 and 3, cannot occur if PreparedJS is in use. The cryptographic checksum of dynamically assembled scripts vary for every iteration, hence, the checksumming validation step will fail, as the script's checksum won't be included in the site's CSP policy (see below for a potential limitation, in case the scheme is used wrongly).

The introduction of the PreparedJS templates offers a reliably secure alternative to insecure server-side script assembly via string concatenation. As the script's syntactic structure is robustly maintained through preparsing in the browser, before the potentially untrusted data values are inserted, XSS

[5] Page Benchmarker: https://chrome.google.com/webstore/detail/page-benchmarker/channimfdomahekjcahlbpccbgaopjll.

Please cite this article in press as: Johns M, Script-templates for the Content Security Policy, Journal of Information Security and Applications (2014), http://dx.doi.org/10.1016/j.jisa.2014.03.007

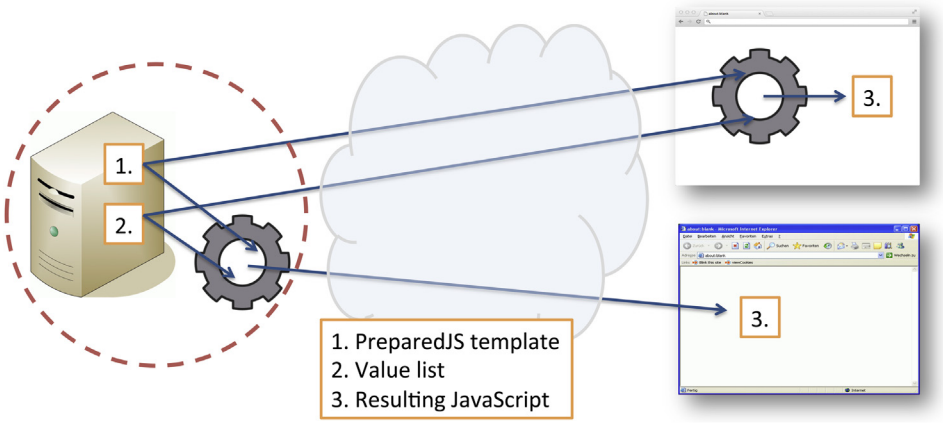


Fig. 1 – Native browser support (top), backwards compatibility via server-side composition service (bottom).

vulnerabilities are rendered impossible, even in cases in which the attacker controls the dynamic values.

(2) **Full control over external, whitelisted script-sources:** The mechanism's fine-grained checksum whitelisting reliably prevents this attack. Due to the checksum checking step, the attacker cannot leverage a compromised external host or related weaknesses. If he attempts to serve altered script code from the compromised origin this code's checksum won't appear in the policy's list of permitted scripts. Hence, the browser will refuse to execute the adversary's attack attempt.

(3) **Attacker provided src-attributes in script-tags:** Our proposed CSP syntax allows for finer-grained control, which scripts are allowed to run in the context of a given Web page. Hence, each page can exactly specify which scripts it really requires, leaving the adversary only minimal opportunities to combine script side effects to his liking. This is especially powerful, when it comes to script inclusion from large scale external service providers, such as Facebook or Google, from which, in most cases, only dedicated scripts are needed for the site to function. Take for example analytics services: If a site utilizes the product *Google Analytics*,[6] currently *all* scripts hosted on Google's domain have to be allowed by the CSP policy. This provides the attacker with a lot of potential options under the scenario outlined in Section 3.3. Using our extended policy mechanism, it is ensured that only the required analytics script will be executed by the browser.

Limitation – Checksumming of insecurely assembled code: Apparently, if a developer creates an application which first insecurely creates dynamic script code and only after this step creates the checksums and CSP policies, the introduced protection measure can be circumvented. However, it is easy to enforce development and deployment processes that prevent such a scenario: The CSP policy generation (which requires a full set of calculated checksums) has to be decoupled from the parts of the application that handles potentially untrusted data. For instance, a requirement that decrees that all script checksums are calculated on deploy-time of the application and remain stable during execution would resolve the issue.

8.2. *Cost of adoption*

Before the introduction of CSP, a mechanism like PreparedJS would have been infeasible, due to the highly flexible nature of the Web: JavaScript can be inserted on many places within a Web page's markup, e.g., through numerous inline event handlers or `JavaScript`-URLs. Creating templates and code checksums for each of these mini-scripts would cause very high development and maintenance overhead, which in turn would hinder the mechanisms acceptance.

However, CSP policies already impose considerable restrictions on how JavaScript is used within Web applications. Thus, to adopt the PreparedJS mechanism on top, is only a small further step and the needed effort appears to be manageable: Strong CSP policies requires all JavaScript to be delivered by dedicated HTTP responses. Hence, script code is already cleanly separated from HTML markup. In result, the total number of scripts to be handled for CSP-enabled sites will be much smaller. Also, this clean separation of the script-code from the markup eases the task of identifying the to-be signed code and creating the actual code checksums considerably. We expect for a sanely designed Web site that the majority of its JavaScript sources are contained in a limited number of dedicated places within the application structure (such as a `/js`-path).

Starting with an enumerable set of dedicated paths in which the scripts reside, the task to separate the script's dynamic code insertion routines from the main static script content is straight forward.

9. Macro-assembly of script templates

9.1. *Dynamic scripts with varying code blocks*

The PreparedJS template format described in Section 6 primarily supports the separation of dynamic data, which is isolated in a template's value list, and its stable code.

However, as we documented in (Johns et al., 2008), other variations of dynamic script assembly can be witnessed in real

[6] Google Analytics: http://www.google.com/intl/de/analytics/.

Please cite this article in press as: Johns M, Script-templates for the Content Security Policy, Journal of Information Security and Applications (2014), http://dx.doi.org/10.1016/j.jisa.2014.03.007

life code. More precisely, we identified on the two following patterns of dynamic JavaScript assembly that exceed the usage of dynamic data values:

- *Code repetition*: These JavaScripts included blocks of repeated code which apparently were created by loops in the generating server-side code (see Listing 12).
- *Selective code omission:* In some cases, a given JavaScript can contain or omit blocks of script code. E.g., according to the authorizations of the logged-in user, a dynamic JavaScript-driven dialog might contain varying options.

As we were able to witness both patterns multiple times, it appears, that in selected scenarios it is preferable for programmers to resort to this coding style.

Listing 12 Repeated code in dependence on the number of values (Johns et al., 2008)

```
<script>
var c = initComputer("superfy");
a[0] = c.compute("value 1");
a[1] = c.compute("value 2");
...
a[9] = c.compute("value 10");
...
</script>
```

9.2. *Extension of PreparedJS to enable code macros*

To accommodate the requirements of more freedom in assembling dynamic JavaScripts, the basic PreparedJS mechanism can be extended. In this section, we present the basic functionally of the extended mechanism, through simple meta macros, and show how Merkle trees can be used to fulfill the stable checksum requirement.

9.2.1. *High level overview*

To allow the script assembly usecases identified above, a PreparedJS template can optionally consist of a set of individual *script building blocks*, which are combined on the client-side into the final JavaScript. Each of these building blocks is in the form of the basic PreparedJS format and carries its individual checksum. A given script building block can appear once or multiple times at a given location of the template. Also, it is permitted to omit a building block.

Through the means of Merkle Hashtrees (Merkle and Brassard, 1989) the individual checksums are combined into the final script checksum, which can be whitelisted in the CSP-policy (see Section 9.2.4).

9.2.2. *Script building blocks*

A script building block is always enclosed in PreparedJS macro instructions, which mark the beginning and end of a building block, as well as its unique identifier (see Lst. 13). The block's code itself is plain JavaScript with PreparedJS placeholders for dynamic values, pointing to the value list.

Listing 13 Meta macro syntax for script building blocks

```
?StartPJSBuildingBlock id="ID1337"?
...PreparedJS template code...
?EndPJSBuildingBlock id="ID1337"?
```

9.2.3. *Script blueprint*

A *script blueprint* specifies how the final JavaScript is composed out of the script building blocks. A script blueprint is a tree structure in JSON format (see Lst. 14).

A node in the tree can either be the root of a subtree or a leaf which points to a script building block via the block's ID. A top to bottom walk through the tree results in the sequence of allowed script blocks.

In the tree, each node carries one of the following four attributes:

- *"once"*: This building block **has** to appear at this place in the script and can only appear once.
- *"multiple"*: This building block can appear at this place in the script multiple times. At least one occurrence is required.
- *"onceOptional"*: This building block **may** appear at this place in the script and can only appear once.
- *"multipleOptional"*: This building block can appear at this place in the script multiple times.

In case one of the *optional* attributes is given, a second attribute is expected, which contains the scripts checksum. In case that the optional block is omitted in the generated instance of the template (see Lst. 15), this checksum is used in hashtree calculation (see Section 9.2.4).

Through this structure a simple macro grammar for the template can be specified.

Listing 14 PreparedJS script blueprint

```
{
  "ID1": "once",
  "ID2": "multiple",
  "ID3": {"onceOptional",
  "ae45fde..."},
  "PJSSubtree": {"once",
  {
  "ID4": "once",
  "ID5": {"multipleOptional",
  "bb89ade..."}
  }
  }
}
```

9.2.4. *Stable checksums through hash trees*

Using the script blueprint and building blocks, a single stable checksum for the template can be generated using Merkle hash trees (Merkle and Brassard, 1989):

The template's checksum is calculated through generating the hash over the top level nodes' checksums. If a node represents a subtree, its checksum is calculated via hashing the checksums of its children. For omitted nodes, the checksum given in the blueprint is used. Finally, for leaf nodes, the checksum is calculated over the code contained in the building block. Each building block is in the PreparedJS

Please cite this article in press as: Johns M, Script-templates for the Content Security Policy, Journal of Information Security and Applications (2014), http://dx.doi.org/10.1016/j.jisa.2014.03.007

format, hence, its checksum does not change during processing.

9.2.5. *Value list and response format*

The format or functionality of the value list does not change, in cases that the macro mechanism is utilized. All script building blocks share the same value list. Also the HTTP response format stays the same, with the exception that a third multipart segment, containing the script blueprint, is added. The extended CSP syntax (see Section 6.3) remains unchanged as well.

Please refer to Listing 15 for a complete HTTP response representing the JavaScript of Listing 12.

Listing 15 PreparedJS macro format

```
HTTP/1.1 200 OK
Date: Thu, 23 Jan 2014 10:03:25 GMT
Server: Foo/1.0
Content-Type: multipart/form-data;boundary=xYzZY
-xYzZY
Content-Type: application/json;
Content-Disposition: form-data;name="PJSBlueprint"
{
  "ID1": "once",
  "ID2": "multiple",
  "ID3": {"onceOptional",
  "ae45fde..."}
}
-xYzZY
Content-Type: application/pjavascript;
  charset=UTF-8
Content-Disposition: form-data;name="preparedJS"
?StartPJSBuildingBlock id="ID1"?
// Single occurrence
var c = initComputer(?initvalue?);
?EndPJSBuildingBlock id="ID1"?
?StartPJSBuildingBlock id="ID2"?
a[?idx[0]?] = c.compute(?value[1]?);
?EndPJSBuildingBlock id="ID2"?
?StartPJSBuildingBlock id="ID2"?
a[?idx[1]?] = c.compute(?value[2]?);
?EndPJSBuildingBlock id="ID2"?
...
?OmitPJSBuildingBlock id="ID3"?
-xYzZY
Content-Type: application/json
Content-Disposition: form-data;name="valueList"
{
  "initvalue": "superfy",
  "idx": [0,1,2,3,4,5,6,7,8,9],
  "value": [ "value 1",
  "value 2",
  ... ]
}
-xYzZY-
```

9.3. *Implementation and practical experiments*

To verify the feasibility of the proposed script macros, we implemented the mechanism in our Chrome extension-based prototype (see Section 9.3.1). Furthermore, to exemplify the usage of the macro mechanism, we ported three generic JSONP libraries to use PreparedJS's template format (see Section 9.3.2).

9.3.1. *Integration of the macro processor into the prototype*

Compared to the processing of plain PreparedJS templates, as described in Section 6.3, introducing script macros only requires additional steps in the checksum verification. The main script parsing process and the final substitution of the placeholders with the actual values remain unchanged. Hence, the integration of the macro mechanism into the prototype solely consisted of adding capabilities to parse the macro blueprint and execute the checksumming steps. More precisely, the checksum verification and template assembly processes were extended as follows:

1. The script blueprint's JSON is parsed and the corresponding tree structure is built.
2. Then, the PreparedJS part of the response is split up sequentially in its separate building blocks. For each of the blocks, the checksum is calculated, and inserted into the blueprint's tree structure at the corresponding position.
   - In case of multiple occurrences of a *"multiple"* building block, it is immediately verified, that the checksums for all encountered instances match. If this is not the case, the checksumming process terminates unsuccessful.
   - Also the checksumming process terminates unsuccessful, in case a building block is found, that doesn't satisfy the blueprint, such as, blocks that carry IDs which are not found in the blueprint, or multiple occurrences of block IDs which are not labeled with *"multiple"*.
3. After this sequential walk through the script template has finished, it is verified, that all elements in the blueprint tree structure (with the exception of the *optional* blocks) now carry a checksum. If this is not the case, the checksumming process terminates unsuccessful.
4. Finally, the overall checksum is calculated, using the collected building block checksums (and potentially placeholders for omitted *optional* blocks) as described in Section 9.2.4.

In case that the checksumming process terminates successfully, the macro syntax is removed from the template completely. This results in a script template, which is now in plain PreparedJS format. This template is subsequently handed to the script parser for further processing and value substitution, precisely as it is in the case of fully static PreparedJS templates.

9.3.2. *Porting of existing, generic JSONP services to PreparedJS*

As explored in Section 4, scripts that follow the JSONP convention pose a considerable problem with today's CSP implementation. Furthermore, it is not straight forward to port generic JSONP scripts into the PreparedJS format, as they rely on dynamic server-side script assembly.

As discussed before, general JSONP services accept arbitrary function names that will be used as callback function, to pass the JSON data to the requesting script. Usage of arbitrary

Please cite this article in press as: Johns M, Script-templates for the Content Security Policy, Journal of Information Security and Applications (2014), http://dx.doi.org/10.1016/j.jisa.2014.03.007

function names cannot be realized with PreparedJS, as changing a function's identifier leads to a different checksum for the containing script building block. Furthermore, as discussed in Section 4, enabling the adversary to call arbitrary functions in a page could potentially lead to successful exploits.

A straight forward approach to realize a JSONP service with PreparedJS would be to remove the integrator's freedom to chose the receiving function freely, and instead hardcode the name of the callback:

```
jsonp_callback(?json_data?);
```

This works sufficiently well for internal, special purpose services. However, with external services hardcoded callback function names can lead to potential problems:

- If the same JSONP service is used twice on the same page within two different code contexts, only one of these code contexts can register the callback function and receive the data.
- Similar complications arise, in case that two different, external JSONP services from two different providers are consumed, and both services utilize an identical callback function name.

Both of these potential problems are likely to occur as soon as *generic JSONP proxies* are part of the application's architecture. A generic JSONP proxy is a small-scale programming library or component, which transforms internal, server-side data structures (such as multidimensional PHP arrays) into JSONP responses and export these as an HTTP service. Examples for such proxies include PHP_JSONP_Response (Janjic, 2013), JSONPness (Lafferty, 2010), and JSONPProxy (Chloride Cull, 2013).

As explained above, neither hardcoded callback names, due to the reasons given above, nor referencing arbitrary callbacks, due to their incompatibility with PreparedJS, are an option. However, the middle-ground, namely selecting a callback function out of a set of predefined function names, can be easily realized using PreparedJS script macros.

Please refer to the example in Listing 16 for an example: The JSONP provider offers a set of predefined callback names – in the example `jsonp_callback_1()` to `jsonp_callback_3()`. Each of them is listed in the script's blueprint as *onceOptional*. Furthermore, the blueprint's final building block contains the JavaScript code necessary to pass the JSON data to the chosen callback. Thus, instead of directly passing the callback function's name, the `script` tag's URL specifies one of the available callbacks via it's corresponding ID, which then results in the PreparedJS response given in Listing 16:

```
<script src="[path]/JSONPproxy?cbID=2[&parameters]">
</script>
```

If a second script in the same page utilizes the same JSONP service or the same proxy library from a different provider, it can route the response to a different callback receiver via simply changing the `cbID` parameter.

As part of our practical evaluation, we ported the mentioned three PHP JSONP proxies (Janjic, 2013; Lafferty, 2010; Chloride Cull, 2013) to use our PreparedJS script macro scheme in the outlined fashion. For most parts, the required changes only consisted in adding the PreparedJS macro boilerplate to the response and in replacing the server-side code, which blindly mirrors the callback parameter, with the ID lookup step. Only in the case of JSONPness also the client-side receiver JavaScript library had to be adapted to correctly process the JSONP response. In total, it took a single programmer less than a day to port these three libraries.

Listing 16 Example: Script macro response for a JSONP address data service

```
HTTP/1.1 200 OK
Date: Wed, 14 Jan 2014 23:12:39 GMT
Server: Foo/1.0
Content-Type: multipart/form-data;boundary=xYzZY
-xYzZY
Content-Type: application/json;
Content-Disposition: form-data;name="PJSBlueprint"
{
  "callback_1": {"onceOptional", "bfe291d..."},
  "callback_2": {"onceOptional", "114aedf..."},
  "callback_3": {"onceOptional", "98aa12b..."},
  "values": "once"
}
-xYzZY
Content-Type: application/pjavascript;
  charset=UTF-8
Content-Disposition: form-data;name="preparedJS"
?StartPJSBuildingBlock id="callback_2"?
jsonp_callback_2
?EndPJSBuildingBlock id="callback_2"?
?StartPJSBuildingBlock id="values"?
(?json_data?);
?EndPJSBuildingBlock id="values"?
-xYzZY
Content-Type: application/json
Content-Disposition: form-data;name="valueList"
{
  "json_data": {
  "name": "Springfield",
  "zipcode": "77612",
  ...
  }
}
-xYzZY-
```

10. Additional benefits of PreparedJS

Besides the security improvements, PreparedJS also provides benefits in the area of network traffic and caching:

For one, as long as the checksum given in the CSP does not change, the template source code of the script can be cached by the browser indefinitely. This does not only apply to static scripts but also to scripts with dynamic values, which up to this point, could not be cached at all. With PreparedJS, the static and dynamic parts of a script are cleanly separated. Hence, the significantly larger static part can be cached and only the freshly generated value list can be retrieved from the

Please cite this article in press as: Johns M, Script-templates for the Content Security Policy, Journal of Information Security and Applications (2014), http://dx.doi.org/10.1016/j.jisa.2014.03.007

server (e.g., through setting a corresponding HTTP request header signifying that only the value list should be responded).

Furthermore, PreparedJS checksums allow *cross-domain* caching of static JavaScripts. The Web has entered a time in which large JavaScript libraries are omnipresent. A recent study has shown that more than 25% of the popular WebSites utilize the JQuery library. The code of such libraries is static. For this reason, their checksum would be the same regardless of the hosting origin. Thus, the JQuery cached from site `a.net` could be retrieved from the browser cache of `b.net`, as long as the cached version carries the specified checksum.

11. Related work

Server-side XSS prevention: Preventing and mitigating Cross-site Scripting attacks has received considerable attention. Most documented methods aim to fight XSS through preventing the actual code injection. They approach the problem, for instance, via tracking untrusted data during execution (Pietraszek and Berghe, 2005; Nguyen-Tuong et al., 2005; Bisht and Venkatakrishnan, 2008), enforcing type safety (Robertson and Vigna, 2009; Johns, 2009; Johns et al., 2010), or providing integrity guarantees over the document structure (Ter Louw and Venkatakrishnan, 2009; Nadji et al., 2009). As a general observation, it can be stated, that these approaches have to address a wide range of potential attack variants and injection vectors, thus, requiring extensive browser/server infrastructure or significant changes on the server-side. In comparison, the scope of PreparedJS's templating mechanism is much more focussed on one specific problem, hence, allowing for a concise solution that effectively can leverage the existing CSP infrastructure.

Client-side techniques: Furthermore, conceptional close to our approach is BEEP (Jim et al., 2007), which proposes whitelisting of static scripts using cryptographic checksums. Similar to our approach, a JavaScript's checksum is calculated and verified, before the script is executed. In comparison to our approach, BEEP does not consider server-side script assembly. Instead, they propose runtime calculation of the server-side checksums. Hence, the protection characteristics of BEEP do not significantly surpass CSP's capabilities while requiring a considerably different enforcement architecture. Our approach only requires an extension to the browser's CSP handling.

Furthermore, several approaches exist that aim to restrict JavaScript execution in general, through applying fine-grained security policies that enforce *least privilege* measures on script code (Meyerovich and Livshits, 2010; Van Acker et al., 2011). In certain cases, such techniques can be utilized to soften the effect of successful XSS attacks. However, their primary focus is at runtime control over third party JavaScript components. Due to this focus, the provided means of these techniques are not sufficient to reach the protection coverage of CSP (and, thus, of PreparedJS).

Finally, more techniques exist, that explicitly aim to prevent the execution of XSS payloads. Most prominent in this area are browser-based XSS filters, which are currently provided by Webkit-based browsers (Bates et al., 2010), Internet Explorer (Ross, 2008), and the Firefox extension NoScript (Maone, 2006).

Script-less attacks: Heiderich et al. (2012) discuss XSS payloads that do not rely on JavaScript execution. Instead, the presented attacks function via the injection of HTML markup and CSS. The primary goal of these attacks is data exfiltration, i.e., transmitting sensitive information, such as credit card numbers or passwords, to the adversary. While CSP's `unsafe-inline` also restricts inline CSS declarations, such attacks are generally out of reach for our proposed technique. PreparedJS sole focus is on secure JavaScript generation and tight control over which scripts are allowed to be executed. A generalization toward HTML markup or CSS is neither planned nor realistic.

12. Conclusion

The Content Security Policy mechanism is a big step forward to mitigate XSS attacks on the client-side. Unfortunately, CSP is not bulletproof. In this paper, we identified three distinct scenarios in which a successful XSS attack can occur even in the presence of a strong CSP. Based on this motivation, we presented PreparedJS, an extension to CSP which addresses the identified weaknesses: Through safe script templates, PreparedJS removes the requirement of unsafe server-side JavaScript assembly. Furthermore, using script checksums, PreparedJS allows fine grained control via whitelisting specific scripts. The combination of CSP's base-line protection with these two capabilities, provides full mitigation of XSS attacks in a robust fashion.

Acknowledgments

This work was in parts supported by the EU Projects STREWS (FP7-318097) and WebSand (FP7-256964). The empirical study of Section 4 was conducted by Martin Wentzel.

REFERENCES

Van Acker Steven, De Ryck Philippe, Desmet Lieven, Piessens Frank, Joosen Wouter. WebJail: least-privilege integration of third-party components in web Mashups. In: Proceedings of the ACSAC 2011 conference; 2011.

Alexa Internet, Inc. Alexa top global sites. Website, http://www.alexa.com/ [accessed August 2013].

Bates Daniel, Barth Adam, Jackson Collin. Regular expressions considered harmful in client-side XSS filters. In: WWW; 2010.

Bisht Prithvi, Venkatakrishnan VN. Xss-guard: precise dynamic prevention of cross-site scripting attacks. In: DIMVA; 2008. pp. 23–43.

CERT/CC. CERT advisory CA-2000–02 malicious HTML tags embedded in client web requests [online], http://www.cert.org/advisories/CA-2000-02.html; February 2000 [accessed 30.01./06].

Crockford D. The application/json Media type for JavaScript object notation (JSON). RFC 4627, http://www.ietf.org/rfc/rfc4627.txt; July 2006.

Chloride Cull. JSONPProxy [software], https://github.com/iuyes/JSONPProxy; 2013.
Heiderich Mario, Niemietz Marcus, Schuster Felix, Holz Thorsten, Schwenk Jörg. Scriptless attacks: stealing the pie without touching the sill. In: ACM conference on computer and communications security; 2012.
Janjic Radovan. PHP_JSONP_Response [software], https://github.com/uzi88/PHP_JSONP_Response; 2013.
Jim Trevor, Swamy Nikhil, Hicks Michael. Defeating script injection attacks with browser-enforced embedded policies. In: WWW2007; May 2007.
Johns Martin. Code injection vulnerabilities in web applications – exemplified at cross-site scripting. University of Passau; 2009. PhD thesis.
Johns Martin, Beyerlein Christian, Giesecke Rosemaria, Posegga Joachim. Secure code generation for web applications. In: 2nd International symposium on engineering secure software and systems (ESSoS '10). Springer; 2010.
Johns Martin, Engelmann Bjoern, Posegga Joachim. XSSDS: server-side detection of cross-site scripting attacks. In: Annual computer security applications conference (ACSAC '08). IEEE Computer Society; December 2008. pp. 335–44.
Johns Martin, Lekies Sebastian. Biting the hand that serves you: a closer look at client-side flash proxies for cross-domain requests. In: Detection of intrusions and malware & vulnerability assessment (DIMVA 2011), LNCS; 2011.
Klein Amit. DOM based cross site scripting or XSS of the Third Kind [online], http://www.webappsec.org/projects/articles/071105.shtml; September 2005 [accessed 05.05.07].
Lafferty James. JSONPness [software], https://github.com/kalchas/JSONPness; 2010.
Ter Louw Mike, Venkatakrishnan VN. BluePrint: robust prevention of cross-site scripting attacks for existing browsers. In: IEEE symposium on security and privacy (Oakland '09); May 2009.
Maone Giorgio. NoScript firefox extension [software], http://www.noscript.net/whats; 2006.
Merkle Ralph C. A certified digital signature. In: Brassard Gilles, editor. CRYPTO. Lecture Notes in Computer Science, vol. 435. Springer; 1989. pp. 218–38.
Meyerovich Leo A, Benjamin Livshits V. Conscript: specifying and enforcing fine-grained security policies for javascript in the browser. In: IEEE symposium on security and privacy. IEEE Computer Society; 2010. pp. 481–96.
Nadji Yacin, Saxena Prateek, Song Dawn. Document structure integrity: a robust basis for cross-site scripting defense. In: NDSS 2009; 2009.
Nguyen-Tuong A, Guarnieri S, Greene D, Shirley J, Evans D. Automatically hardening web applications using precise tainting. In: 20th IFIP international information security conference; May 2005.
Nikiforakis Nick, Invernizzi Luca, Kapravelos Alexandros, Van Acker Steven, Joosen Wouter, Kruegel Christopher, Piessens Frank, Vigna Giovanni. You are what you include: large-scale evaluation of remote JavaScript inclusions. In: CCS 2012; 2012.
Nikiforakis Nick, Meert Wannes, Younan Yves, Johns Martin, Joosen Wouter. SessionShield: lightweight Protection against session hijacking. In: ESSoS 2011; February 2011.
Oezses Seda, Erguel Salih. Cross-domain communications with JSONP. Whitepaper, IBM DeveloperWorks; February 2009. http://www.ibm.com/developerworks/library/wa-aj-jsonp1/wa-aj-jsonp1-pdf.pdf.
Open Web Application Project (OWASP). OWASP top 10 for 2010 (The top ten most critical web application security vulnerabilities) [online], http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project; 2010.
Open Web Application Project (OWASP). XSS (cross site scripting) prevention cheat sheet [online], https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet; 2012 [accessed 12.03.12].
Pietraszek Tadeusz, Berghe Chris Vanden. Defending against injection attacks through context-sensitive string evaluation. In: Recent Advances in Intrusion Detection (RAID2005); 2005.
Robertson W, Vigna G. Static enforcement of web application integrity through strong typing. In: Proceedings of the USENIX security symposium, Montreal, Canada; August 2009.
Ross David. IE 8 XSS filter architecture/implementation [online], http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx; August 2008 [accessed 05.05.12].
Ruderman Jesse. The same origin policy [online], http://www.mozilla.org/projects/security/components/same-origin.html; August 2001 [accessed 01.10.06].
Schock Jason. So how does JSONP really work? [online], http://schock.net/articles/2013/02/05/how-jsonp-really-works-examples/; February 2013.
Scholte Theodoor, Balzarotti Davide, Kirda Engin. Have things changed now? An empirical study on input validation vulnerabilities in web applications. Comput. Secur. 2012;31(3):344–56.
Stamm Sid, Sterne Brandon, Markham Gervase. Reining in the web with content security policy. In: WWW; 2010.
The webappsec mailing list. The cross site scripting (XSS) FAQ [online], http://www.cgisecurity.com/articles/xss-faq.shtml; May 2002.
Toews Ben. Abusing password managers with XSS [online], http://labs.neohapsis.com/2012/04/25/abusing-password-managers-with-xss/; April 2012 [accessed 05.05.12].
Vogt Philipp, Nentwich Florian, Jovanovic Nenad, Kruegel Christopher, Kirda Engin, Vigna Giovanni. Cross site scripting prevention with dynamic data tainting and static analysis. In: NDSS 2007; 2007.
W3C. Content security policy 1.0. W3C candidate recommendation, http://www.w3.org/TR/2011/WD-CSP-20111129/; November 2012.
W3C. Content security policy 1.1. W3C editor's draft 02, https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html; December 2012.
Zalewski Michal. Postcards from the post-XSS world [online], http://lcamtuf.coredump.cx/postxss/; December 2011.

Please cite this article in press as: Johns M, Script-templates for the Content Security Policy, Journal of Information Security and Applications (2014), http://dx.doi.org/10.1016/j.jisa.2014.03.007