PreparedJS: Secure Script-Templates for JavaScript

Martin Johns
SAP Research

Abstract. Content Security Policies (CSP) 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 [18]. Even though the basic problem has been known since at least 2000 [4], XSS still occurs frequently, even on high-profile Web sites and mature applications [24]. The primary defense against XSS is secure coding on the server-side through careful and context-aware sanitization of attacker provided data [19]. 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,

Listing 1 CSP example

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

such as event handlers in HTML attributes, and string-to-code transformations, which are provided by `eval()` and similar functions (see Sec. 2.2 for further details). Unfortunately, as we will discus 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.

1.2 Contribution and paper outline

In this paper, we explore the remaining weaknesses of CSP (see Sec. 3) and examine which steps are necessary to fill the identified gaps for completing CSP's protection capabilities. Based on our results, we propose PreparedJS, an extension of the CSP mechanism (see Sec. 5). PreparedJS is built on two pillars: A templating format for JavaScript which follows SQL's prepared statement model (see Sec. 5.1) and a light-weight script checksumming scheme, which allows fine-grained control over permitted script code (see Sec. 5.2). 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 Sec. 6).

2 Technical background

2.1 Cross-site Scripting (XSS)

The term *Cross-site Scripting (XSS)* [26] 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 [23], and empowers the adversary to conduct a wide range of potential attacks, ranging from session hijacking [17], over stealing of sensitive data [28] and passwords [27], 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 [24] 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 (CSP)

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 (CSP) [25] 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 [10]. 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 [29].

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.

---

[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.

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

```
1 (function() {
2   var ga = document.createElement('script');
3   ga.src = 'http://serv.com/ga.php?source='+document.location;
4   var s = document.getElementsByTagName('script')[0];
5   s.parentNode.insertBefore(ga, s);
6 })();
```

However, as we will show in this section, three potential attack variants remain feasible under the currently standardized version 1.0 of CSP [29]. 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 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.

[2] The depicted code was consciously designed in a naive fashion to make the issue easily understandable. 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.

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 [16]. 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 four further, more subtle attacks which enable the same class of script inclusion attacks and show their practical applicability (see [16] 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 Michael Zalewski [31]: Given an HTML-injection vulnerability, a strict CSP policy will

Listing 4 CSP 1.1 policy requiring script-nonce

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

effectively prevent the direct injection of attacker-provided script code. However, he still is be able to inject HTML markup including further `script`-tags pointing to the whitelisted domains.

This way an attacker is able to control the URLs and order from which the scripts in a Web page 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 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 [30].

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 allow the usage of the directive to reenable inline scripts (as depicted in Lst. 5) without significant security degradation.

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 Goal: 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 clearly define which exact scripts are allowed to be executed. And, as seen in Sec. 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 Sec. 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.

5 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

Listing 6 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]
```

in a fashion that is unsusceptible to code injection vulnerabilities (see Sec. 5.1 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.

5.1 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 6 for a template which represents the dynamic script of Listing 3.

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 [5]) 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 7 for a PHP-script which creates the value list for Lst. 6 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. 8).

Listing 7 Creating value list for Lst. 6 (`ga_values.php`)

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

5.2 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 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 scripts, a policy lists the script's checksums in the policy declaration (see Sec. 5.3). 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, is it 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.

5.3 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. 9 for an example).

Listing 8 PreparedJS HTTP multipart response

```
HTTP/1.1 200 OK
Date: Thu, 23 Jan 2012 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--
```

5.4 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. 8), 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. 10). 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.

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 6.2 we show, how applications can provide backwards compatibility support for legacy browser automatically.

Listing 9 Extended CSP syntax, whitelisting two script checksums

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

Listing 10 Extended PreparedJS `script`-tag syntax

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

5.5 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. 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. 8). 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.

6 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 Sec. 6.1) and discuss how backwards compatibility can be provided for browsers that are not able to handle PreparedJS's template format natively (see Sec. 6.2).

6.1 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 threat 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 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[3]. 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.

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

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

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

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 a 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 implementations with inferior performance compared to native code. Furthermore, 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 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*[4] extension, using mean values of ten page load iterations over a standard German household DSL line. During the tests, all encountered external JavaScripts were treated, as if they were PreparedJS-templates and, thus, fully parsed and checksummed.

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

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. Here three new steps are introduced, that do currently not exist. The cryptographic checksum has to be calculated, value list has to be parsed, and the obtained values have to be inserted for the placeholders. Non 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-formated value list, and inserting the data values after the script parser's tokenization step is straight foreword 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.

6.2 Transparently providing legacy support

As mentioned in Section 5.4, 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 scripts URL, such as `?pjs-prerender=true`. All requests carrying this parameter automatically go through the composition service.

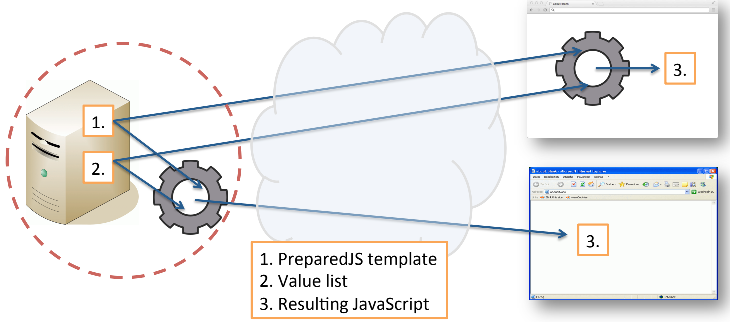

Fig. 1. Native browser support (top), backwards compatibility via server-side composition service (bottom)

7 Discussion

7.1 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 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*[5], 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 Sec. 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.

7.2 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 to be handled scripts 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.

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

8 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 [20, 15, 3], enforcing type safety [21, 8, 9], or providing integrity guarantees over the document structure [11, 14]. 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 out approach is BEEP [7], 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 a 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 [13, 1]. 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 [2], Internet Explorer [22], and the Firefox extension NoScript [12].

Script-less attacks: In [6] Heiderich et al. 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 towards HTML markup or CSS is neither planned nor realistic.

9 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 these two capabilities with the base-line protection provided by CSP, full protection against XSS attacks can be achieved in a robust fashion.

Acknowledgments This work was in parts supported by the EU Projects STREWS (FP7-318097) and WebSand (FP7-256964). Furthermore, we would like to thank anonymous reviewers for their helpful comments.

References

1. Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. WebJail: Least-privilege Integration of Third-party Components in Web Mashups. In *Proceedings of the ACSAC 2011 conference*, 2011.
2. Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side XSS filters. In *WWW*, 2010.
3. Prithvi Bisht and V. N. Venkatakrishnan. Xss-guard: Precise dynamic prevention of cross-site scripting attacks. In *DIMVA*, pages 23–43, 2008.
4. 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` (01/30/06), February 2000.
5. D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, `http://www.ietf.org/rfc/rfc4627.txt`, July 2006.
6. Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. Scriptless attacks: stealing the pie without touching the sill. In *ACM Conference on Computer and Communications Security*, 2012.
7. Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW2007*, May 2007.
8. Martin Johns. *Code Injection Vulnerabilities in Web Applications - Exemplified at Cross-site Scripting*. PhD thesis, University of Passau, 2009.
9. Martin Johns, Christian Beyerlein, Rosemaria Giesecke, and Joachim Posegga. Secure Code Generation for Web Applications. In *2nd International Symposium on Engineering Secure Software and Systems (ESSoS '10)*. Springer, 2010.
10. Amit Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. [online], `http://www.webappsec.org/projects/articles/071105.shtml`, (05/05/07), Sebtember 2005.
11. Mike Ter Louw and V.N. Venkatakrishnan. BluePrint: Robust prevention of Cross-site Scripting Attacks for Existing Browsers. In *IEEE Symposium on Security and Privacy (Oakland'09)*, May 2009.
12. Giorgio Maone. NoScript Firefox Extension. [software], `http://www.noscript.net/whats`, 2006.
13. Leo A. Meyerovich and V. Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *IEEE Symposium on Security and Privacy*, pages 481–496. IEEE Computer Society, 2010.

14. Yacin Nadji, Prateek Saxena, and Dawn Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *NDSS 2009*, 2009.
15. A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, May 2005.
16. Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *CCS 2012*, 2012.
17. Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. SessionShield: Lightweight Protection against Session Hijacking. In *ESSoS 2011*, February 2011.
18. 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.
19. 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`, last accessed 12/03/12, 2012.
20. Tadeusz Pietraszek and Chris Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection (RAID2005)*, 2005.
21. W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium*, Montreal, Canada, August 2009.
22. David Ross. IE 8 XSS Filter Architecture / Implementation. [online], `http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx`, last accessed 05/05/12, August 2008.
23. Jesse Ruderman. The Same Origin Policy. [online], `http://www.mozilla.org/projects/security/components/same-origin.html` (01/10/06), August 2001.
24. Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Have things changed now? an empirical study on input validation vulnerabilities in web applications. *Computers & Security*, 31(3):344–356, 2012.
25. Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *WWW*, 2010.
26. The webappsec mailing list. The Cross Site Scripting (XSS) FAQ. [online], `http://www.cgisecurity.com/articles/xss-faq.shtml`, May 2002.
27. Ben Toews. Abusing Password Managers with XSS. [online], `http://labs.neohapsis.com/2012/04/25/abusing-password-managers-with-xss/`, last accessed 05/05/2012, April 2012.
28. Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS 2007*, 2007.
29. W3C. Content Security Policy 1.0. W3C Candidate Recommendation, `http://www.w3.org/TR/2011/WD-CSP-20111129/`, November 2012.
30. 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.
31. Michal Zalewski. Postcards from the post-XSS world. [online], `http://lcamtuf.coredump.cx/postxss/`, December 2011.