Towards stateless, client-side driven Cross-Site Request Forgery protection for Web applications

Sebastian Lekies, Walter Tighzert, and Martin Johns
SAP Research
firstname.lastname@sap.com

Abstract: Cross-site request forgery (CSRF) is one of the dominant threats in the Web application landscape. In this paper, we present a lightweight and stateless protection mechanism that can be added to an existing application without requiring changes to the application's code. The key functionality of the approach, which is based on the double-submit technique, is purely implemented on the client-side. This way full coverage of client-side generation of HTTP requests is provided.

1 Introduction

Cross-site Request Forgery (CSRF) is one of the dominant threats in the Web application landscape. It has been rated by the OWASP on place five in their widely regarded "Top Ten Most Critical Web Application Security Vulnerabilities" [Ope10a]. CSRF exists due to an unfortunate combination of poor design choices which have been made while evolving the Web application paradigm. The fundamental cause being the adding of transparent authentication tracking using HTTP cookies onto a stateless hypertext medium consisting of HTTP and HTML. For this reason, there is no "easy fix" for CSRF. Instead, the current practice is to selectively identify all potential vulnerable interfaces that a Web application exposes and protecting them manually within the application logic.

In this paper, we present a lightweight and stateless protection mechanism which can be added to an application deployment without requiring any changes to the actual application code. Our approach reliably outfits all legitimate HTTP requests with evidence which allows the Web server to process these requests securely.

2 Technical background

In this section, we briefly revisit the technical background of Web authentication tracking and how it can be misused by CSRF, before detailing the mechanism of the double-submit cookie protection.

2.1 Web authentication tracking

The HTTP protocol is stateless. For this reason, the application itself has to provide measures to track sessions of users that span more than one request-response pair. The most common approach to do so relies on HTTP cookies: The first time a client connects to a server, the server generates a random, hard to guess number (the so-called session identifier) and sends it to the client. From now on, the client's Web browser will attach this cookie value to all subsequent requests, allowing the server to keep track of the session.

In case the Web application requires authentication, the user will enter his credentials (usually a combination of a username and a password). After validating the provided information, the server upgrades the user's session context into an authenticated state. In consequence, the session identifier is utilized as the user's authentication token: All further requests which carry the user's session identifier will be regarded by the Web server as being authenticated.

2.2 CSRF

Cross-Site Request Forgery (CSRF) is an attack that consists in tricking the victim to send an authenticated HTTP request to a target Web server, e.g., via visiting a malicious page that creates such a request in the background. In cases that the user is currently in an authenticated state with the targeted Web site, the browser will automatically attach the session cookies to such requests, making the server believe that this request is a legitimate one and, thus, may cause state-changing actions on the server-side.

Take for instance the attack illustrated in Fig. 1: First, the victim logs into his online bank account (www.bank.com) and doesn't log out. Afterwards, he navigates to a site under the control of an attacker (www.attacker.org). Tricking the victim to visit this website can be done, for instance, by sending an email to the victim promising him nice cat pictures. Unfortunately, the website doesn't contain only pictures. A GET cross-domain request to www.bank.com is made using an invisible image tag that carries a src-URL pointing to the bank's server[1]. As the user didn't log out, the request is made in his authentication context as the browser attach the cookies to this request and the money transfer is started. The Same Origin Policy doesn't prevent this request to be made, it only prevent the attacker to read the request's response but the attacker is not interested in the response, only in the action that has been triggered on the server-side.

2.3 CSRF protection with double-submit cookies

The most common protection against CSRF is to restrict the state changing action requests to POST requests and the usage of a nonce. When a form is requested, the application

[1] There are different possibilities to make such cross domain requests: an HTTP GET request can be embedded in an image tag whereas a POST request can be the result of a FORM created using JavaScript.

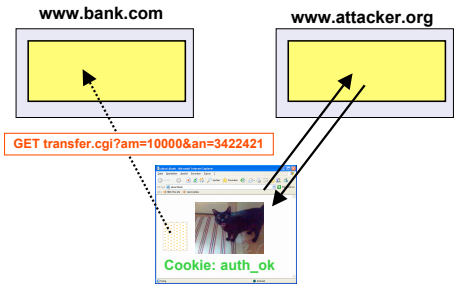

Abbildung 1: CSRF attack

generates a nonce and attach it to the form as a hidden field. When the user submits a form, the nonce is then sent back to the server, which then checks if this nonce is valid. One of the disadvantage of this protection is that the server has to manage all nonces, i.e., the server has to remember each nonce for each form, invalidate the expired one. This can under certain circumstances even lead to a Denial of Service attack, where an attacker successively requests forms, generating for each form a new nonce. With each new nonce the available memory of the server is reduced until nothing is left anymore to store legitimate data. Moreover, this protection can not be easily integrated in a legacy application without having to modify the application's code.

The double-submit cookie method as first described in [ZF08] offers a protection against CSRF that can be implemented without the need for server-side state. Thereby, the CSRF token is submitted twice. Once in the cookie and simultaneously within the actual request. As the cookie is protected by the Same-Origin Policy only same-domain scripts can access the cookie and thus write or receive the respective value. Hence, if an identical value is stored within the cookie and the form, the server side can be sure that the request was conducted by a same-domain resource. As a result, cross-domain resources are not able to create a valid requests and therefore CSRF attacks are rendered void.

Nevertheless, this methods still requires some code modifications, as the token attachment needs to be conducted within the application's code. In the next section we present a mechanism for legacy applications that makes use of the double-submit cookie technique, but without the need for code modifications. In order to do so, we implemented a library that conducts the token attachment completely on the client-side.

3 Protection approach

This section is structured as follows: First we provide an outline of the general architecture of our approach (see Sec. 3.1). Then we explore our client-side framework in depth, covering the aspects of HTML-tag augmentation (Sec. 3.2), dealing with implicit HTTP request generation (Sec. 3.3), handling JavaScript-based HTTP request generation (Sec. 3.4) and covering server-side redirects (Sec. 3.5).

3.1 High-level overview

Our proposed protection mechanism, which is based on the double-submit cookie technique (see Sec. 2.3) consists of a client-side and a server-side component:

On the server-side a proxy (in the following called gatekeeper) tracks any incoming request and validates whether the request carries a valid token according to the double-submit scheme. Additionally, the gatekeeper holds a whitelist of defined entry points for the Web application that a client is allowed to call without a valid token. Any request that does not carry a valid token and is targeted towards a non-whitelisted resource is automatically discarded by the gatekeeper[2]. Besides that, the gatekeeper injects the client-side component in the form of a JavaScript library into any outgoing response by inserting an HTML script element directly into the beginning of the document's head section. Therewith, the JavaScript library is executed before the HTML rendering takes place or before any other script runs. Its duty is to attach the CSRF token according to the double-submit scheme to any outgoing request targeted towards the server. Unlike other approaches, such as [JKK06], which rely on server-side rewriting of the delivered HTML, our rewriting method is implemented completely on the client-side. This way, we overcome server-side completeness problems (see Sec. 4.2 for details).

Within a browser multiple different techniques can be leveraged to create state changing HTTP requests. Each of these mechanisms has to ensure that CSRF tokens are attached to valid requests. Otherwise a valid request would be discarded by the gatekeeper (as it does not carry a valid token). Hence, the client-side component needs to implement such a token attachment mechanism for any of these techniques. In general we are, thereby, able to distinguish between 4 different possibilities to create valid HTTP requests:

1. Requests that are caused by a user interacting with DOM elements (e.g. 'a' tags or 'form' elements)
2. Requests that are implicitly generated by HTML tags (e.g. frames, img, script, link tags)
3. Requests that are created by JavaScript's XMLHttpRequest
4. Redirects triggered by the server

3.2 Requests caused by a user interacting with DOM elements

Traditionally, requests within a Web browser are caused by user interaction, for example, by clicking on a link or submitting a form. Incorporating tokens into such a request is straight forward. For the case of 'a' tags our library simply rewrites the 'href' attribute whenever such a tag is inserted into the document[3]. For form tags we can app-

[2] Note: Pictures, JS and CSS files are whitelisted by default

[3] Note: Tags are only rewritten if the URLs point to the webpage's own server, so that no foreign website is accidentally able to receive the CSRF token.

ly a similar technique, but instead of rewriting the location we register an `onSubmit` handler that takes care of attaching the token when a form is submitted. This technique of rewriting attributes or registering handler function is executed multiple times during the runtime of a document. Once, when the page has finished loading[4] and once whenever the DOM is manipulated in some way. Such a manipulation can be caused by several different JavaScript functions and DOM attributes such as `document.write`, `document.createElement` or `.innerHTML`. Therefore, our library wraps all those functions and attributes by utilizing the function wrapping techniques described by Magazinius et al. [MPS10] . The wrapper functions simply call our rewrite engine whenever they are invoked i.e. whenever the DOM is manipulated. In that way the rewriting functionality is applied to the plain HTML elements that were already inserted into the page.

3.3 Implicitly generated Requests caused by HTML tags

Besides requests that are generated by the user, HTML tags are also able to create new requests implicitly. For example, if a new img, iframe, link or script tag is inserted into a document the browser immediately creates a new request. As a CSRF protection only has to protect state changing actions it is not necessary to incorporate protection tokens into img, script and link tags as these tags are not supposed to be used for any state changing activity. Therefore, the gatekeeper on the server-side won't block any requests targeted towards pictures, script- or CSS-files not carrying a protection token. Frames, however, could potentially be used to trigger actions in the name of the user, hence, CSRF tokens have to be incorporated into it somehow. But as the browser fires the corresponding requests implicitly our JavaScript library is not able to do a rewriting of the location as described in the previous section (the request is already sent out when our rewriting engine is triggered). One way of overcoming this problem would be to parse the content given to DOM manipulating functions and rewriting iframe tags before inserting them into the page. This, however, has two major drawbacks. On one hand parsing is prone to errors and implies a lot of performance overhead and on the other hand we cannot parse the content during the initial loading of the Web page as this is completely done by the browser and not by JavaScript functions. Therefore, we would not be able to incorporate CSRF tokens into iframes that are initially available within an HTML document. Hence we need to utilize a different method to overcome this problem.

Figure 2 shows the general setup of our framing approach. Whenever a request towards a protected resource arrives without a valid token (1), the gatekeeper responds with a 200 HTTP status code and delivers a piece of JavaScript code back to the client (2). The code which is seen in Listing 1 exploits the properties guaranteed by the Same-Origin Policy in order to detect whether a request was valid or not. The Same-Origin Policy allows scripting access from one frame to another frame only if both frames run on the same domain. So if the framed page has access to DOM properties of its parent window it can ensure that the website in the parent window runs on the same domain. As a CSRF attack is always conducted across domain boundaries, a request that is conducted cross-domain

[4] at the onload event

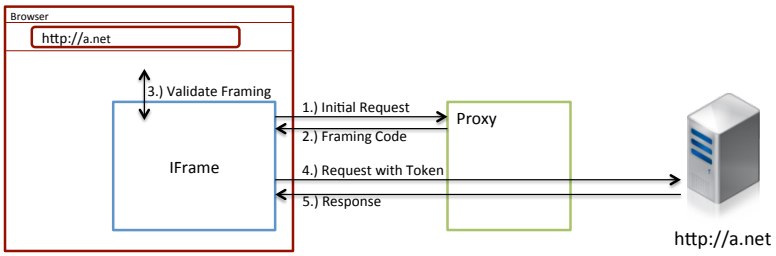

Abbildung 2: Framing Solution

is very suspicious while a request from the same domain is not suspicious at all. A valid request is thus caused by a frame that is contained in a page that relies on the same domain while an illegitimate request is caused by a frame that is contained on a page running on a different domain. Therewith our Script is able to validate the legitimacy of the request by accessing the `location.host` DOM property of its parent window (3). If it is able to access it (and thus it is running on the same domain), it receives the CSRF token from the cookie and triggers a reload of the page, but this time with the token incorporated into the request. The gatekeeper will recognize the valid token and instead of sending out our script again it will forward the request to the protected resource (4) that answers with the desired response (5).

Listing 1 Frameing code

```
(function () {
    "use strict";

    try {
        if (parent.location.host === window.location.host) {
            var token = getCookie("CSRFToken");
            var redirectLoc = attachToken(window.location, token);
            window.location = redirectLoc;
        }
    } catch (err) {
        console.log(err);
        //parent.location.host was not accessible
    }

})();
```

3.4 Requests created by JavaScript's XMLHttpRequest

Besides generating requests by inserting HTML elements, JavaScript is also able to initiate HTTP requests directly by utilizing the `XMLHttpRequest` object. Most of the modern JavaScript-based application, such as GMail, make use of this technique in order

to communicate asynchronously with a server. Incorporating tokens into this techniques is very easy. Our library simply wraps the XMLHttpRequest Object and is thus able to fully control any request send via this object (See Listing 2 for a code excerpt). Whenever the wrapper is supposed to send a requests towards the protected server the token is inserted as a GET parameter for GET requests or as a POST parameter for post requests. Therewith, protecting modern AJAX applications is very easy.

Listing 2 XMLHttpWrapper (Excerpt)

```
(function () {
    "use strict";
    var OriginalXMLHttpRequest = window.XMLHttpRequest;

    function XHR() {
        this.originalXHR = new OriginalXMLHttpRequest();

        //define getter and setter for variables
        Object.defineProperty(this, "readyState", {
            get: function () { return this.originalXHR.readyState; },
            set: function (val) { this.originalXHR.readyState = val; }
        });
        [...]
    }

    XHR.prototype.open = function (method, url, async, user, password) {
        var token = getCookie("CSRFToken");
        this.attachToken(method, url, token);
        this.originalXHR.open(method, url, async, user, password);
    };
    [...]

    function NewXMLHttpRequest() {
        return new XHR();
    }

    window.XMLHttpRequest = NewXMLHttpRequest;
})();
```

3.5 Redirects triggered by the server

As the application is not aware of the CSRF protection mechanism, it is possible that the application triggers a redirect from one protected resource towards another via 3xx HTTP redirects. In some cases this will cut of the CSRF tokens as the application will not generically attach each received parameter to the new redirect location. Hence, the server-side proxy has to take over this task. Whenever a redirect takes place a server responds with a 3xx status code and the `location` response header that defines the location to which the client is supposed to redirect. Therefore, the gatekeeper monitors the HTTP status codes for responses to requests that carry valid tokens. If the response contains a

3XX status code, the gatekeeper rewrites the `location` header field and includes the token into it.

4 Discussion

4.1 Assessing security coverage and functional aspects

To assess the protection coverage of the approach, we have to verify if the proposed approach indeed provides the targeted security properties. For this purpose, we rely on previous work: Our protection approach enforces the model introduced by Kerschbaum in [Ker07]: Only clearly white-listed URLs are allowed to pass the gatekeeper without providing proof that the corresponding HTTP request was generated in the context of interacting with the Web application (for this proof we rely on the double-submit value, while [Ker07] utilizes the `referer`-header, which has been shown by [BJM09] to be insufficient). Using the model checker Alloy, [Ker07] provides a formal verification of the security properties of the general approach. In consequence, this reasoning also applies for our technique.

4.2 The advantage of client-side augmentation

As mentioned before, in our approach the altering of the site's HTML elements is done purely on client-side within the browser. This has significant advantages over doing so on the server-side both in respect to completeness as well as to performance. To do this step on the server-side, the filter of the outgoing HTTP responses would have to completely parse the response's HTML content to find all hyperlinks, forms, and other relevant HTML elements. Already this step is in danger of being incorrect, as the various browser engines might interpret the encountered HTML differently from the utilized server-side parsing engine. In addition, and much more serious, are the shortcomings of the server-side when it comes to JavaScript: In modern Web applications large parts of the displayed HTML UI are generated dynamically. Some Web2.0 applications do not transmit HTML at all. Such HTML elements are out of reach for server-side code and cannot be outfitted with the protection value. In contrast, on the client-side the JavaScript component already works directly on the DOM object, hence, no separate parsing step is necessary. And, as described in Section 3, our technique is capable to handle dynamic HTML generation via function wrapping.

4.3 Open issues

Direct assignment of window.location via JavaScript: Besides the techniques presented in Section 3, there is one additional way to create HTTP requests within the browser.

By assigning a URI to either `window.location`, `document.location` or `self.location` a piece of JavaScript can cause the browser to redirect to the assigned URI. To control these requests it would again be possible to wrap these three location attributes with function wrappers in order to attach the token whenever something is assigned to the `.location` attribute. Due to security reasons, however, some browsers do not allow the manipulation of the underlying getters and setters of `window.location` or `document.location`[5]. Hence, the function wrapping technique cannot be applied to those attributes. Although, there are some ways to overcome this obstacle, we are not aware of an approach that covers the whole browser spectrum. Therefore, our library is not able to cover redirects via `window.location/document.location`. As a result these requests will not carry a CSRF token and thus they will be discarded by the gatekeeper. Hence, applications that are using `.location` for redirects are not yet supported by our library. In a later version we plan to include a work around for this problem.

RIA elements: Plugin-based RIA applets such as Flash, Silverlight or Java applets are also able to create HTTP requests within the user's browser. In order to function with our CSRF protection these applets need to attach the CSRF token to their requests. Basically, this can be achieved in two different ways. On the one hand these applets could make use of JavaScript's XMLHttpRequest object. As this object is wrapped by our function wrapper, tokens will automatically be attached to legitimate requests. However, if an applet uses a native way to create HTTP requests, our JavaScript approach is not able to augment the request with our token. Therefore, such RIA elements have to build a custom mechanism to attach our token.

4.4 Server-side requirements and supporting legacy applications

The proposed approach has only very limited footprint on the server-side. It can be implemented via a lightweight HTTP request filtering mechanism, which is supported by all major Web technologies, such as J2EE or PHP. If no such mechanism is offerd by the utilized framework, the component can also be implemented in the form of a self-standing server-side HTTP proxy.

As explained in Section 3, the component itself needs no back channel into the application logic and is completely stateless, due to choosing the double-submit cookie approach.

In consequence, the proposed approach is well suited to add full CSRF protection to existing, potentially vulnerable applications retroactively. To do so, only two application specific steps have to be taken: For one, potential direct access to the JavaScript `location` object have to be identified and directed to the proxy object (see Sec. 4.3). Secondly, fitting configuration files have to provided, to whitelist the well-defined entry points into the application and the URL paths for the static image and JavaScript data.

[5] Firefox still allows the manipulation of window.location, while other browsers do not allow it. On the other hand Safari allows the overwriting of document.location, while other browsers don't

5 Related work

In the past, the field of CSRF-protection has been addressed from two distinct angles: The application (i.e. the server-side) and the user (i.e., the client-side).

Server-side: Server-side protection measures, such as this paper's approach, aim to secure vulnerable applications directly. The most common defense against CSRF attacks is manual protection via using request nonces, as mentioned in Sec. 2.2. [Ope10b] provides a good overview on the general approach and implementation strategies. In [JKK06] a server-side mechanism is proposed, that automatically outfits the delivered HTML content with protection nonces via server-side filtering. Due to the purely server-side technique, the approach is unable to handle dynamically generated HTML content and direct JavaScript-based HTTP requests.

As an alternative to checking protection nonces, [BJM09] proposes the introduction of an `origin`-HTTP header. This header carries the domain-value of the URL of the Web side from which the request was created. Through checking the value the application can ensure, that the received request was created in a legitimate context and not through an attacker controlled Web site. Up to this date, browser support of the `origin`-header is incomplete, hence, rendering this mechanism unsuitable for general purpose Web sites.

Client-side: In addition to the server-centric protection approach, several client-side techniques have been proposed, which provide protection to the user's of Web applications, even in cases in which the operator of the vulnerable Web application fails to identify or correct potential CSRF vulnerabilities. RequestRodeo [JW06] achieves this protection via identifying cross-domain requests and subsequent removal of cookie values from these requests. This way, it is ensured that such request are not interpreted on the server-side in an authenticated context. CSFire [RDJP11] improves on RequestRodeo's technique on several levels: For one the mechanism is integrated directly into the browser in form of a browser extension. Furthermore, CSFire utilizes a sophisticated heuristic to identify legitimate cross-domain requests which are allowed to carry authentication credentials. This way support for federated identity management or cross-domain payment processes is granted. Furthermore, related client-side protection mechanism were proposed by [ZF08] and [Sam11]. Finally, the ABE component of the Firefox extension NoScript [Mao06] can be configured on a per-site basis to provide RequestRodeo-like protection.

6 Conclusion

In this paper we presented a light-weight transparent CSRF-protection mechanism. Our approach can be introduced without requiring any changes of the application code, as the server-side component is implemented as a reverse HTTP proxy and the client-side component consists of a single static JavaScript file, which is added to outgoing HTML-content by the proxy. The proposed technique provides full protection for all incoming HTTP requests and is well suited to handle sophisticated client-side functionality, such as AJAX-calls or dynamic DOM manipulation.

Literatur

[BJM09] Adam Barth, Collin Jackson und John C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *CCS'09*, 2009.

[JKK06] Nenad Jovanovic, Christopher Kruegel und Engin Kirda. Preventing cross site request forgery attacks. In *Proceedings of the IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm 2006)*, 2006.

[JW06] Martin Johns und Justus Winter. RequestRodeo: Client Side Protection against Session Riding. In Frank Piessens, Hrsg., *Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*, Seiten 5 – 17. Departement Computerwetenschappen, Katholieke Universiteit Leuven, May 2006.

[Ker07] Florian Kerschbaum. Simple Cross-Site Attack Prevention. In *Proceedings of the 3rd International Conference on Security and Privacy in Communication Networks (SecureComm'07)*, 2007.

[Mao06] Giorgio Maone. NoScript Firefox Extension. [software], `http://www.noscript.net/whats`, 2006.

[MPS10] Jonas Magazinius, Phu H. Phung und David Sands. Safe Wrappers and Sane Policies for Self-Protecting JavaScript. In Tuomas Aura, Hrsg., *The 15th Nordic Conference in Secure IT Systems*, LNCS. Springer Verlag, October 2010. (Selected papers from AppSec 2010).

[Ope10a] 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.

[Ope10b] Open Web Application Security Project. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. [online], `https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet`, accessed November 2011, 2010.

[RDJP11] Philippe De Ryck, Lieven Desmet, Wouter Joosen und Frank Piessens. Automatic and precise client-side protection against CSRF attacks. In *European Symposium on Research in Computer Security (ESORICS 2011)*, Jgg. 6879 of *LNCS*, Seiten 100–116, September 2011.

[Sam11] Justin Samuel. Requestpolicy 0.5.20. [software], `http://www.requestpolicy.com`, 2011.

[ZF08] William Zeller und Edward W. Felten. Cross-Site Request Forgeries: Exploitation and Prevention. Bericht, Princeton University, 2008.