# CSRFing the Web

*Dimokritos Stamatakis, Dimitrios Chasapis*

*Department of Computer Science*
*University of Crete*
*P.O. Box 2208, Heraklion, GR 71409, Greece*
*{dstamat,hassapis}@csd.uoc.gr*

## Abstract

In this report we study Cross-Site Request Forgery (CSRF), a common exploit used to attack websites. We present the most well known defenses against this attack and explain how each one works. Moreover, We conducted an analysis on the top 100,000 websites, as rated by alexa.com, as to which defense they implement to counter any CSRF attacks and present the results. To meet this end we implemented a tool that crawls the web, parses HTTP headers and HTML body of each site to deduct which defense is employed.

## 1 Introduction

Cross Site Request Forgery (CSRF) is an attack which maliciously forces the user's web application to sent a request to a website, on which the user is authenticated, thus fooling the server that he, the attacker is actually an authenticated user. The severity of a CSRF attack varies as it depends upon the level of access the victim user has. For instance a simple user can only compromise his data, while an administrator user can compromise the entire site infrastructure. A common way to launch a CSRF attack is to fool the user, through means of social engineering, to click on a link from a fraud website/email, made or sent by the attacker, which redirects to a website that the user has an open session with, thus hijacking that session. Due to the difficulty of achieving these prerequisites (user has a running session with target, social engineering), many regard a CSRF attack as an unlikely scenario, undermining its importance. This misconception could not be further from the truth. CSRF is ranked as the 909th most dangerous software bug ever found [2] and it is among the twenty most exploited security-vulnerabilities in 2007 [10].

The most common CSRF countermeasures are the following:

1. **Secret Validation Token**
   The idea in this defense is to send an additional information in each HTTP request. This information should be hard to guess and it should be used to detect whether a request comes from the authorized user, or an attacker. The server should reject requests with different token, or with no token at all. A common practice is to store the token in a *hidden* input field in every form on the website, that should be protected.

2. **Referer Header**
   Referer is a HTTP Header field that indicates the URL that initiated the HTTP request. The server can detect requests that originated from an unauthorized website, if they do not send a Referer header with a whitelisted URL as a value. However, the field is optional, thus the server has to deal with cases where there is no Referrer in the request, by either accepting or denying the request.

3. **Custom XMLHtmlHeader**
   Site's that implement AJAX interfaces have the option to require custom XMLHtmlHeaders to all state modifying actions. Web browsers only allow XMLHtmlHeaders to be sent from a website to itself, thus a scenario where a fraud website tries to sent a custom XMLHtmlHeader to another website is not possible.

4. **Origin Header**
   Improves on the Referer header scheme, enabling the client to only send information needed by the server to identify the origin of the request. Contrary to the Referer header, it includes no path information that could leak sensitive data in URL parameters.

5. **X-Frame Header**
   It's an HTTP response header that instructs the web browser to refuse display the content of the website

in a frame. It can be set to either allow framing if the website is the same one that delivered the web page, or it can deny any framing at all. All major web browsers support it.

6. **Content-Security-Policy Header**
   Content Security Policy is an added layer of security and can be used as a defense for CSRF attacks. Using this header a server can instruct a web browser what content, and from which origin he can load, when displaying the web page.

Given the number of defenses again CSRF, we are interested to know which one the majority of the websites deploy. For this purpose we implemented a web crawler that automatically parses HTTP headers and the corresponding HTML bodies, using pattern matching to deduct, to its best of abilities, which of the above defense mechanisms are employed. Our experiment indicates that the most common defense is the secret validation token, a defense that indeed offers a high level of security, if impelemented properly.

The following of this report is organized as follows: In section 1 we explain how a CSRF attack can be launched. In section 2.2 we present the existing CSRF defenses and discuss each's advantages and disadvantages. In section 3.6 we discuss how we implemented a web crawler that analyzes websites and decides which CSRF defense is deployed. In section 4.3 we present our findings, regarding the defense of choice of the most visited websites and of some of the major Content Management frameworks. We also evaluate our crawler's performance by comparing its findings with our manual examination of some of the most visited websites. Finally, we conclude in section 5.3.

## 2 CSRF Attack

Cross Site Request Forgery (CSRF or XSRF) works by injecting network queries to the victim's web browser. By doings so the attacker is able to hijack the user's session, exploiting the server's trust to that user. The server would be fooled that the attacker is a legitimate user and allow him to change the session state or give him access to sensitive data. The CSRF could also be used to exploit the user's trust to a server, by logging the victim under the attacker's, without the first noticing, and logging hist activity or stealing sensitive data the victim may try to send through queries. In this section we will see a possible scenarios of how such an attack could be launched, in order to better understand how it works, and define the prerequisites needed to successfully deliver the attack.

### 2.1 Example

A user usually interacts with a website by sending requests to a server in a form of a URL including parameter values. The attacker simply needs to fool the victim into sending a request URL with his parameter values. For example, consider the following GET request:

```
GET
HTTP://bank.com/transfer.do?\
acct=Bob&amount=1000\
HTTP/1.1
```

Sending this GET request a user can transfer money from his account to another user's account (in our example Bob). In order to launch a CSRF attack, an attacker needs to generate his own URL in the exact fashion a legitimate URL would be generated. For example a forged URL could be the following [1], where our attacker's account is Alice:

```
HTTP://bank.com/transfer.do?\
acct=Alice&amount=1000\
HTTP/1.1
```

All the attacker need to do now is make the user sent a GET request using his URL. A simple scenario would be to the link through an email or a fraud website, where the user is tricked to click on the link. Of course, in order to successfully complete the transaction the victim should already be logged in, so that the browser will include the cookies that manage the state of the session between the server and the victim. There are even more sophisticated ways to force the user's web browser to send the request, without requiring any action from the victim other than opening a website or an email. This is accomplished by putting the URL in an HTML IMAGE, SCRIPT or IFRAME tag src attribute, in which case the web browser will automatically load the elements, and make an GET request for the resource, where in fact he will be sending the request URL. Note that the POST request method is not CSRF-proof, since an attacker could include a form with hidden values to generate a POST request with all the necessary parameters.

### 2.2 Attack Prerequisites

Understanding the above example, we can deduct that following prerequisites are needed for a CSRF attack to be successful:

1. The attacker knows how a valid request URL is formed
2. The attacker can provide correct values for all the parameters

[1]The example was taken from [3]

3. The attacker need to lure the victim into a fraud website or email, while the victim is logged in the target website.

By observing the above we can see how such an attack could be stopped. Either the server or the browser needs to know where did the request originated from, or make it impossible for the attacker to guess the correct values for all the parameters, by using some secret between user and server.

## 3 CSRF Defenses

As discussed in section 1 a CSRF attack can be countered by either enabling the server to identify the origin of the request or use a secret session token with the user, to make not feasible for the attacker to send a forged request. All the defense mechanism discussed in the this section employ one of these two methods, some of them with the slight variation that the web browser might be responsible to block request, or content from unauthorized sources, as instructed by the server.

### 3.1 Secret Validation Token

A common and very secure method employed to counter CSRF attacks is to include a secret token, that is session depended in every form that needs to securely submitted to the server. This token should be hard for the attacker to guess and should vary between different users and sessions. The secret validation token can be generated from a Session Identifier which is unique for each user, and hard to guess. A disadvantage of this technique is that if the session identifier leaks to a website, then anyone that visits that website can see that session identifier, and impersonate its owner, until the session cookie expires. Another method is to use a random generated nonce, store it in a cookie associated with the user's Session Identifier and check it on every request. The problem here is that the server needs to maintain a large state table. A way to bind the secret token with a user's id, that does not require a large state table, is to use the HMAC of the token and the user's session Identifier (see Ruby on Rails [7]).

### 3.2 Referer Header

Most web browsers have the ability to sent a Referer request header to the server, identifying the origin of the request. The server can then determine if the source is a legitimate URL, possibly from the website's domain, or a fraud. Although this solution to the CSRF thread problem seems appealing, the Referer header is optional for the browser, since due to the nature of the HTTP protocol, the header will leak sensitive information, such as the content of the query, which rises serious privacy concerns. For this reason, the Referer header is often suppressed by web browsers. In the absence of the header, the server can either allows access, which deems the defense ineffective or deny access, in which scenario it is possible that a legitimate user will be denies access to the server's content. According to [9], the number of users that would be denied access under this scenario is very high.

### 3.3 Custom XMLHtmlHeader

websites that use AJAX to implement their interface need to make use of the XMLHtmlHeader. An attractive property of using the custom XMLHtmlHeader is that web browsers do not allow websites to send such headers to URLs outside of their domain. XMLHtmlHeaders can only be sent from a website to itself, thus the attackers effort to send such a header from his website will be blocked by the victim's browser. In order for the defense to be effective, site administrators must take care to deny all state modifying requests that are made without an XMLHtmlHeader.

### 3.4 Origin Header

The Origin header, as proposed in [9], works in similar fashion to the Referer header with main difference that it leaks no information about the query that is being made. The information included are only the required ones from the server to identify if the origin of the request is a legitimate source. If the browser cannot determine the source of the request, then the origin value is set to null, and it's up to the server to decide what action to take. Moreover, the Origin header is sent only for POST requests, to avoid leaking any sensitive information over the network. Another sublime difference between the Origin and Referer header is that, since the privacy issue has been addressed in the first, supporting browsers will always provide the Origin header, thus the attacker cannot trick the server to believe that the browser does not provide the header, where such a scenario is possible when using the Referer header. The Origin header is also proposed as a standard by W3C [1].

### 3.5 X-Frame Header

Originally implemented in IE8, it has now been adopted by all major web browsers. If the header is provided, it can instruct the web browser not to display the content of the website in a frame. The header can hold two distinct value, DENY and SAMEORIGIN. The first one force the browser no to display the content of the web page in any frame, while the second allows frames to display its content, if the the page using the frames, is the someone that delivered the content in the first place. The browser however needs to support this function, so it cannot be regarded as the first line of defense by website administrators.

## 3.6 Content-Security-Policy Header

Content-Security-Policy (CSP) header has been recently introduced in order to address the various content injection vulnerabilities present in most websites. CSP header issues directives to the web browser on how to handle certain content, when loading a web page. For instance a server could instruct the web browser not to load scripts, images or other elements from specific sources and only. Consider the following example:

```
Content-Security-Policy:/
img-src  'self'
```

This directive instructs the web browser to only load images from the server's own origin. The CSP header directive have to bee enforced by the browser, however if the browser does not support them, they will be ignored, making this kind of defense ineffective. CSP header should provide an additional layer of defense, and website administrators should provide other CSRF countermeasures as well.

## 4 Implementation

In order to conduct our experiment, we implemented a web crawler in C++ with the ability to parse HTTP header and the corresponding HTML bodies in order to find patterns that attempting classify a website as to what CSRF counter measure it employs. The tool consists of three main components, the *Network* component which handles the network trafficking, the *HTTP Header Parser* which parses the HTTP headers and pattern matches the header fields with the headers discussed in section 2.2 and the *HTML Parser* who parses the body of the web pages seeking patterns in HTML forms that are candidates for a secret token value. In the rest of this section we will discuss the design of each of the distinct parts of the web crawler and their interaction with one another.

### 4.1 Network component

The *Network* component handles all request and responses to and from a web server. It uses the curl library [4] which offers an comprehensive API to buffer HTTP request Headers and HTML bodies, in addition to modify the HTTP response header. Our implementation uses the pthread library to allow concurrent request to be served simultaneously. The parellelization policy we used allows for concurrent requests to be made and server for every distinct home URL we have in the queue to be visited. However, links in a website are server by the same thread that served the home URL. Our implementation is lock-free and thread-safe with the exception of the underlying openssl library that is used by the curl library, where we needed to define the locking functions as described in [5]. The curl library offered the option to automatically visit websites that offered a Location header, thus we take no special actions for such cases. The *Network* component feeds each new header to the *HTTP Header parser* and each new body to the *HTML Parser*. The headers are served by the same thread that serves their corresponding bodies. Interaction between the *Network* component and the *HTML Parser* is also possible when the second needs to visit a new link, which then goes through the whole a process a queued link would (new header and bodies are parsed). Moreover, the *Network* component is also responsible to sent request to the server using a fake Referer header, if such a request is denied, then we add the website to the list of website's that employ the Referer Header defense. Also note, that we set up the curl library to set our User Agent header value to *Mozilla/4.73 [en] (X11; U; Linux 2.2.15 i686)* so that we get the same website a user would.

### 4.2 HTTP Header Parser component

The *HTTP Header Parser* parses every header it is given by the *Network* component. Each thread can initiate an HTTP parser, independently from the other threads. The *HTTP Header Parser* drops any headers with return code other than 200. If any header is dropped, the *Network* component does not continue to call the *HTML Parser* and drops the link entirely. In section 2.2 we presented a number of defenses based on headers, we have examined a series of response headers manually and have defined a set of fields found in these headers that help us determine if any those defenses are present. Each of the following fields classify to one of the aforementioned defenses:

- X-Requested-(By—With) - Custom XMLHtml-Header
- X-Frame-Options - X-Frame header
- Access-Control-Allow-Origin - Origin Header (this header is intended to restrict the loading of content outside the website's own domain)
- Content-Security-Policy - Content-Security-Policy header

Note that the above is a rough subset of the possible variations the different headers can appear. To match the larger set of all possible variations we use a pattern matching library PCRE [6], which allows us to use Perl like regular expressions in the C++ language. Every time we get a match, we put the home URL to the corresponding list of URLs each defense class maintains.

### 4.3 HTML Parser component

The HTML parsing is done using the libxml2 library [8]. The *HTML Parser* component serves a dual purpose. It parses the HTML body in order to find links that lead to login forms and feed them to the *Network* component to process the new URL. We care especially for login or register forms because they are sensitive, state modifying submit forms that are more likely to be protected by a secret token. Again we use the PCRE library to pattern match links that could lead to such submit forms. The second function it performs is to parse all the Input fields in any Forms found in the web page. If an Input field has an attribute named type with value equal to ”hidden”, then we check the value attribute, to see if it could be a secret token value. Due to the unpredictable nature of the value of the toke, it is very hard to distinct between true secret tokens and false positives. We use the pattern matching functionality of the PCRE library once more, to distinct between hash values, that are more likely to be secret tokens and purge characters that are unlikely to be found in a token. If a value is believed to be a valid secret token, then it is added to the Secret Validation Token defense list of URLs. Furthermore, we noticed that on many occasions, websites used a hidden Input field in forms that contained no value at all. The interesting thing was that many of these fields had names such as ”referer” or ”id” etc. We believe that such fields could be hold secret validation values after the user get authorized, to protect the forms from CSRF attacks. For this reason, we have another class of defense called Possible Secret Validation Token. Again, in order for a website to get that classification, the Input fields name should pass the pattern matching test (referer, id, etc). In section 4.3 we discuss the limitations and evaluate our method in more detail.

## 5 Experimental Results - Evaluation

We run our web crawler described in section 3.6 over the 100,000 most visited web sites as rated by alexa.com. The crawler uses heuristics in order to determine the type of defense a web site employs and we provide no formal proof that our results are correct. In this section we present our findings, regarding the number of web sites that choose to use each defense. In the rest of this section we evaluate the method we used to conduct the experiment (our web crawler), compared to a very small number of web sites we examined manually.

### 5.1 Platform

We run our experiments on two Intel(R) Xeon(R) CPU E5405 with 8 cores @2.00GHz machines each with 12GB of RAM. One machine was running Red Hat Enterprise Linux Server 5.2 and the other CentOS 5.6. On each machine we raised 64 threads, since the greater portion of the time was spent on waiting for the requests to be served, the 8 cores could efficiently serve this number of threads. Each machine used a list of 50,000 URLs we got from Alexa's Top 100,000 website list.

### 5.2 Results

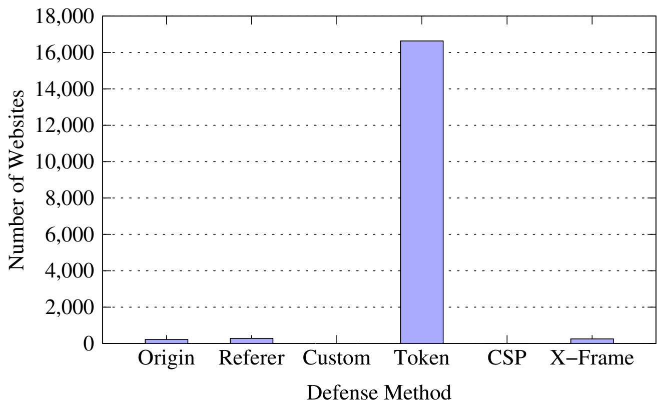

Figure 1: Prefered CSRF Defense Method for the 100,000 most visited websites

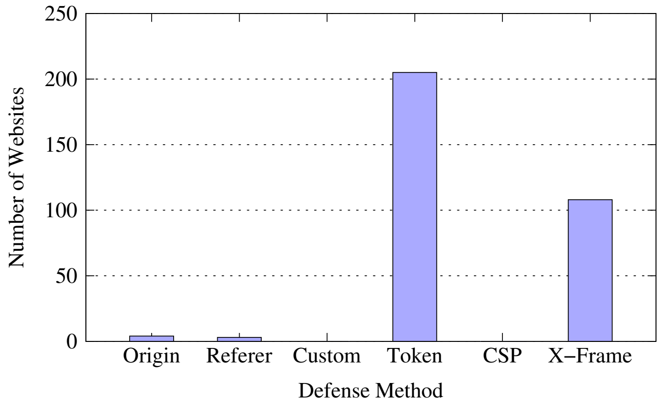

Figure 2: Prefered CSRF Defense Method for the 1,000 most visited websites

Figure 1 shows the total number of websites that use each defense discussed in section 2.2. Origin bar refers to Origin header, Referer bar to Referer header, Custom to Custom XMLHtmlHeader, Token to Secret Validation

| Top 100,000 Websites | |
|---|---|
| Origin header | 221 |
| Referer header | 285 |
| Custom XMLHtmlHeader | 0 |
| Secret Token | 16633 |
| Content Security Policy header | 5 |
| X-Frame header | 260 |

Figure 3: Prefered CSRF Defense Method for the 100,000 most visited websites in absolute values

| Top 1,000 Websites | |
|---|---|
| Origin header | 4 |
| Referer header | 3 |
| Custom XMLHtmlHeader | 0 |
| Secret Token | 205 |
| Content Security Policy header | 0 |
| X-Frame header | 108 |

Figure 4: Prefered CSRF Defense Method for the 1,000 most visited websites in absolute values

Token, X-Frame to X-Frame header and CSP to Content Security Policy header. We see that the preferred method by far, according to our tool's judgement, is the use of secret validation tokens in submit Forms. We also notice that the Referer and Origin Headers appear to be used in equal frequency. Another point of interest is that we did not detect any custom XMLHtmlHeaders, which is an unexpected result. We did not manage to see whether this is accurate or an implementation bug, note that we did, indeed, not find any such headers in the websites we manually examined. Moreover, the Contet-Security-Policy appears in some rare cases and only in websites that hold high places in alexa's list, while the X-Frame header is used much more frequently. Although, the X-Frame offers only a subset of the Content Security Policy's functionalities (namely restricting content to be loaded in frames), the later is a relative new header and has just been implemented in major web browsers, while the X-Frame header has been around for a much longer period. We expect that more websites will start using the Content Security Policy header in the near future.

Figures 2 and 4 show the preferred defense method for the first 1,000 most visited websites, as found in alexa.com index. The results draw a similar picture to those in shown in 1 and 3.

### 5.3 Method Evaluation

Our method use heuristics in order to associate a defense mechanism with a web site. This leaves a big margin for error which renders it difficult to prove the correctness of our results. Parsing the header to determine the different header fields is trivial, but determining if a web site uses secret tokens is not, for two main reasons.

1. The vast set of possible token values, makes it hard, if not unfeasible to formalize a way to determine is an arbitrary string value is a token value. On many cases, the token is a hash value, but this not the necessarily true for all tokens

2. Using secret tokens to protect forms, is crucial when the user has already been authorized. It is unrealistic for our crawler to get authorized access to all web sites, and then parse its content

We present two cases, where our method failed or was partially correct. The first case is paypal, which protects none of its forms with any token, even the login form when an unauthorized user is viewing the website. When a user is granted authorization, we found out that the forms contained a hidden Input field with a secret token. The second case, is facebook, where the login form is originally protected by a token, that is generated using the current timestamp. This does not offer the desired security for an authorized user, thus when a user logs in, facebook generates a second token, which is session depended. Furthermore, we provide the number of false positives we found by manually examining the first 100 websites on alexa's list. Out of 22 possible tokens found only 7 were actually token values, which gives us a false possitive percentage of 68%. Unfortunatelly, we are not able to provide a less intuitive metric regarding our tool's performance ability to find secret token values.

## 6 Conclusion and Future Work

Our evaluation showed that the preferred method to defend against a CSRF attack is the Secret Validation Token. If correctly implemented, this method provides excellent security. However, we believe that website administrators can benefit from the Content Security Policy and X-Frame header, which provide an extra layer of security. The Origin header appears to be a solid solution to the CSRF problem, but web browsers that will not support it can be the reason for a hole in the security of the web page. A very secure option is to implement the website's interface using AJAX and require a custom XMLHtmlHeader from the web browser. However, this solutions require considerable effort to redesign and rewrite the whole web application.

Our method lacks the sophistication to deal efficiently with secret token value, in order to determine if a website employs this kind of defense. In the future, a better heuristic mechanism could be conceived to deal with this problem. Furthermore, to increase our tool's efficiency, we require to access the HTML code that an authorized user would view. It is unfeasible for an automated tool to get this kind of access, unless accounts are provided for the corresponding websites. Even then, the arbitrary design of websites and error prone html syntax make it difficult to efficently parse the vast ammount of websites. After all, spider traps lurk in every corner of the internet.

## References

[1] Cross-origin resource sharing. `http://dvcs.w3.org/hg/cors/raw-file/tip/Overview.html`.

[2] Cross site request forgery. `http://en.wikipedia.org/wiki/Cross-site_request_forgery#Severity`.

[3] Cross site request forgery (csrf). `https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)`.

[4] Curl. `http://curl.haxx.se/`.

[5] Openssl threads. `http://www.openssl.org/docs/crypto/threads.html`.

[6] Perl like regular expressions. `http://www.pcre.org/`.

[7] Ruby on rails. `http://www.rubyonrails.org/`.

[8] The xml parser toolkit for gnome. `http://xmlsoft.org/`.

[9] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 75–88, New York, NY, USA, 2008. ACM.

[10] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, pages 581–590, New York, NY, USA, 2006. ACM.