EUROPEAN ACADEMIC RESEARCH
Vol. IV, Issue 12/ March 2017

Impact Factor: 3.4546 (UIF)
DRJI Value: 5.9 (B+)

ISSN 2286-4822
www.euacademic.org

Security Evaluation of CSRF Protection Mechanisms

EDNA OGARI
Prof. WAWERU MWANGI
Dr. AGNESS MINDILA
Jomo Kenyatta University of Agriculture and Technology
Nairobi, Kenya

Abstract:

In a cross site request forgery attack, the trust of a web application in its authenticated users is compromised, thereby allowing the intruder to make arbitrary HTTP requests on behalf of a victim user. The challenge has been that web applications classically act upon such requests without verifying that the performed actions are undeniably intended. This means that if the victim is authenticated, a successful cross site request forgery attack effectively circumvents the underlying authentication mechanism. Depending on the web application that is being exploited, the attacker can post messages or send mails in the name of the victim, or even change the victim's login credentials such as name and password. Legitimate users will therefore lose their integrity over the website when the Cross site request forgery takes place. Over the years, researchers have proposed a number of techniques for protection against cross site request forgery. Such methods include the referrer HTTP Header, Custom HTTP header, Origin Header, client site proxy, Browser plug-in and Random Token Validation. This paper sought to investigate the security features of the existing cross site request forgery prevention techniques to determine whether they truly protect the much needed protection. The survey results indicated that these existing solutions are not so immune to various attacks. Therefore, these applications employing these solutions are partially protected. This paper therefore proposes

an application programming interface as the probable solution to these attacks.

Key words: Cross site request forgery, application programming interfaces, CSRF prevention techniques.

I. INTRODUCTION

Cross-Site Request Forgery (CSRF) is an attack where a malicious website sends a request to a web application that a user is already authenticated against from a different website. Employing this attack, an intruder gains access to functionalities in a target web application through the victim's already authenticated browser. According to Sentamilselvan et al.,(2013), the CSRF targets include web applications such as social media, in-browser email clients, online banking and web interfaces for network devices.

In these attacks, an intruder exploits how the target web application manages the authentication process (Ramarao, 2009). For this to be successful, the victim must be authenticated against the target site. For instance, if *edna.com* has an online banking website that is vulnerable to CSRF, then if intruder X visits a page containing a CSRF attack on *edna.com* but X is not currently logged in, nothing occurs. On the other hand, provided X is logged in, the requests in the attack will be executed as if they were actions that X had intended to take.

The impact of successful cross site request forgeries is restricted to the capabilities exposed by the vulnerable application. Rob (2016) explains that these attacks could result in a transfer of funds, changing a password, or purchasing an item in the victim's context. Effectively, using CSRF attacks, an intruder is able to make a target system carry out activities on the target's browser without the knowledge of the victim.

II. CSRF PREVENTION TECHNIQUES AND THEIR WEAKNESSES

A. Client-Side Proxy

Ramarao et al., (2009) presented a client-side proxy solution that could detect and avert CSRF attacks using *img* element or other HTML elements which are employed to access the graphic images for the webpage. This method is able to inspect and modify client requests as well as the application's replies automatically. In so doing, applications with the secret token validation technique could transparently be extended.

The setbacks of this method are that if a proxy is compromised, then all sensitive information will lost as well. Moreover as Sentamilselvan (2013) point out, the technique does not have the ability to detect login CSRF.

B. POST Method

Another common technique for mitigating cross site request forgeries is the utilization of POST form submission method instead of GET parameters. However, as Neil (2015) point out, this approach only raises the bar for the attacker, as it closes certain attack vectors such as the use of image tags, but does not adequately prevent these attacks. Moreover, completely doing away with the use of GET parameters is not always possible as this may result in applications that are more cumbersome for users to navigate and more difficult for developers to implement.

C. Client Side Browser Plug-In

This method, implemented as an extension to the Firefox web browser, can protect users from certain types of CSRF attacks (Dav et al., 2016). This plug-in intercepts every HTTP request and decides whether it should be allowed or not. The decision process is based on the following criteria: any request that is not a POST request is allowed; if the requesting site and target

site fall under the same-origin policy, the request is allowed; if the requesting site is allowed to make a request to the target site using Adobe's cross-domain policy, the request is allowed; if a request is rejected, the user is alerted that the request has been blocked using a familiar interface, such as the one used by Firefox's popup blocker.

Luca et al., (2016) discuss that this gives the user the option of adding the site to a white list. The setback is that users will need to download and install this extension for it to be effective against CSRF attacks.

D. HTTP Referrer Checking

According to Neil (2015), this is an effective countermeasure in circumstances where the web application relies on its correctness. It works by maintaining a white list of accepted referrers, thereby enabling applications to figure out requests initiated due to cross site request attacks, and therefore not to carry out the requested transactions. Unfortunately, configurations can be carried out on modern browsers, permitting the sending of empty or even arbitrary values for this header (Lance, 2016). Moreover, sending the referrer header is dejected since during this sending process, sensitive information may be leaked to third parties.

Another challenge is that when classifying requests with an empty referrer header as valid, it would become impracticable to detect attacks against users who follow the recommendation and disable the transmission of the referrer header. On the flip side, when treating such requests as cross site request forgery attacks, then all requests of the concerned users would be rejected. This problem is further aggravated by the fact that an attacker can make use of several browser-specific tricks to trigger a cross site request forgery request with an empty referrer.

E. Dynamic Token Generation

The basic goal of this technique is to prevent cross site request forgeries by adding a fresh token to every web request whose target page should be protected one way. This method, according to Dave et al., (2016), efficiently prevents CSRF attacks toward PHP web applications. It provides an automatic robust solution again cross site request forgeries by employing a CSRF token. It used to verify whether the token has been previously issued from servers, utilizing the property of cryptographically secure hash function. The demerit of this is that it requires frequent dynamic generation of tokens.

F. Shared Secret

This method is employed between the client and the server to identify the authentic origin of a request. For instance, a web based banking application could be adapted such that the form contains an additional, hidden token field (Acunetix, 2017). This token must be generated by the application. This is meant to ensure that the token is not easily predicted by an attacker and associated with the current session. Ultimately, this guarantees that requests for financial transactions are processed only if they contain the correct token.

However, as Rob (2016) explains, the drawback of this approach is that it requires a considerable amount of manual work. Since majority of the current web applications have evolved into large and complex systems, retrofitting them with the mechanisms necessary for token management would require detailed application-specific knowledge and considerable modifications to the application source code. Importantly, there is no guarantee that the modified code is indeed free of cross site request forgery vulnerabilities, as developers tend to make errors and omissions.

G. Server-Side Proxy

This is a mitigation mechanism for cross site request forgery that provides only partial protection by replacing GET requests by POST requests. It can also rely on the information in the Referer header of HTTP requests (Dingjie, 2017). This approach is based on a server-side proxy that detects and prevents CSRF attacks in a way that is transparent to users as well as to the web application itself. The setback is that it only provides partial protection for the underlying web applications.

H. Cryptographic Tokens

These tokens are employed to prove that the Action Formulator knows a session specific secret. To achieve this, it utilizes secret tokens to prove the Action Formulator knew an Action and user specific secret. An optional HTTP referrer header is used to verify Action Formulators (Matthew and Myers, 2016). This requires changes to application state so that it is done only with HTTP POST operations. This is facilitated by use of simplified cross site prevention token.

Acunetix (2017) explain that the side effects of this is that the attackers can modify their attacks to be form based CSRF, submitting forms automatically or though tricking users by making huge, mislabeled submit buttons. In this technique, the header is optional and may not be present. In addition, some browsers deactivate this header, making it unavailable when interactions occur between HTTPS and HTTP served pages. This increases the risk of header spoofing, and tracking the valid sources of invocations may be difficult in some applications.

I. Anti-CSRF Tokens

According to Abdalla (2015), the Synchronizer Token Pattern is the recommended method and the most widely used prevention technique. This method finds applications in many search engines such as Google, social media applications such as

Facebook and Twitter, and popular open source web applications such as WordPress and Joomla. The synchronizer token pattern requires the generation of random challenge tokens, referred to as anti-CSRF tokens, which are associated with the user's current session. These challenge tokens are then inserted within the HTML forms and links associated with sensitive server-side operations. Whenever a user submits a form or makes a request to the links, the anti-CSRF token is included in the request.

Thereafter, the server application verifies the existence and correctness of this token before processing the request. If the token is missing or incorrect, the request will be rejected. The problem is that this methods requires secure socket layer o be implemented in all applications. Additionally, it would not detect login cross site request forgeries.

J. Limiting The Lifetime Of Authentication Cookies

Limiting the lifetime of cookies to a short period of time ensures that if users were going on to other websites, then the cookies should expire after a short period of time. This means that if an intruder was trying to send any HTTP request to the users which he was able to know, then the attacker would not fill the password again (Luca et al., 2016). In this way, CSRF attacks are often reduced to a brief period of the users' time. On the flip side, this may necessitate frequent logins on the side of legitimate users.

K. Double Submitting Cookie

According to Telikicherla et al.,(2014), storing the cross site request forgery token in session can prove problematic. , an alternative defense is use of a double submit cookie. A double submit cookie is defined as ending a random value in both a cookie and as a request parameter, with the server verifying if the cookie value and request value match.

When a user authenticates to a site, the site should generate a (cryptographically strong) pseudorandom value and set it as a cookie on the user's machine separate from the session id. The site does not have to save this value in any way, thus avoiding server side state. The site then requires that every transaction request include this random value as a hidden form value (or other request parameter). According to Matthew and Myers (2016), a cross origin attacker cannot read any data sent from the server or modify cookie values, per the same-origin policy. This means that while an attacker can force a victim to send any value he wants with a malicious CSRF request, the attacker will be unable to modify or read the value stored in the cookie. Since the cookie value and the request parameter or form value must be the same, the attacker will be unable to successfully force the submission of a request with the random CSRF value.

L. Encrypted Token Pattern

The Encrypted Token Pattern leverages an encryption, rather than comparison, method of Token-validation. After successful authentication, the server generates a unique Token comprised of the user's ID, a timestamp value and a nonce, using a unique key available only on the server. This Token is returned to the client and embedded in a hidden field. Subsequent AJAX requests include this Token in the request-header, in a similar manner to the Double-Submit pattern. As Singh et al.,(2014) point out, Non-AJAX form-based requests will implicitly persist the Token in its hidden field. On receipt of this request, the server reads and decrypts the Token value with the same key used to create the Token. Inability to correctly decrypt suggest an intrusion attempt. Once decrypted, the UserId and timestamp contained within the token are validated to ensure validity; the UserId is compared against the currently logged in user, and the timestamp is compared against the current time.

On successful Token-decryption, the server has access to parsed values, ideally in the form of claims. These claims are processed by comparing the UserId claim to any potentially stored UserId (in a Cookie or Session variable, if the site already contains a means of authentication). The Timestamp as Abdalla (2016) discusses, is validated against the current time, preventing replay attacks. Alternatively, in the case of a CSRF attack, the server will be unable to decrypt the poisoned Token, and can block and log the attack.

This pattern exists primarily to allow developers and architects protect against CSRF without session-dependency. It also addresses some of the shortfalls in other stateless approaches, such as the need to store data in a Cookie, circumnavigating the Cookie sub-domain and HTTP ONLY issues (Luca et al., 2016). However, this requires dynamic generation and requires a small amount of system resources to check tokens and big database tables to manage tokens and sessions.

M. Custom Request Headers

Adding CSRF tokens, a double submit cookie and value, encrypted token or other defense that involves changing the user interface can frequently be complex or otherwise problematic. An alternate defense which is particularly well suited for AJAX endpoints is the use of a custom request header (Batarfi et al., 2014). This defense relies on the same-origin policy restriction that only JavaScript can be used to add a custom header, and only within its origin. By default, browsers do not allow JavaScript to make cross origin requests.

A particularly attractive custom header and value to use is: X-Requested-With: MLHttpRequest because most JavaScript libraries already add this header to requests they generate by default. Some of them do not though. For example, AngularJS used to, but does not anymore. According to Nenad et al., (2015), if this is the case for a given system, one can simply

verify the presence of this header and value on all the server side AJAX endpoints in order to protect against CSRF attacks. This approach has the double advantage of typically requiring no user interface changes and not introducing any server side state, which is particularly attractive to REST services.

However, bypasses of this defense using Flash were documented as early as 2008 and again as recently as 2015 to exploit a CSRF flaw in Vimeo (Kavitha et al., 2016). On the other hand, it is believed that the Flash attack cannot spoof the Origin or Referer headers. Hence, by checking both of them, it is anticipated that this combination of checks should prevent Flash bypass CSRF attacks. However, many browsers disable this Header.

N. Orthogonal Proxy-Based Solution

An orthogonal proxy-based solution on the client side builds upon the token approach, and additionally proposes the use of an outside entity for detecting IP-based authentication (Kadambari and Manisha, 2016). For cases in which JavaScript code initiates HTTP requests, this code is altered automatically to contain the token. According to Jaya and Suneeta (2016), without evaluation, the reliability of this technique, which requires a certain extent of program understanding, is difficult to assess. Also, it is believed that a manual treatment of these rare cases on the server side provides a more stable and efficient solution. Besides, due to the usual difficulties with client-side proxies, this implementation does not support secure socket layer connections yet.

III. DISCUSSIONS

The detection of web-based attacks has received considerable attention because of the increasingly critical role that web-based services are playing on the Internet. This includes web application firewalls to protect applications from malicious

requests as well as intrusion detection systems that attempt to identify attacks against web servers and their applications. Also, code analysis tools were proposed that check applications for the existence of bugs that can lead to security vulnerabilities. In particular, cross site scripting attacks have received much interest, and both server-side and client-side solutions were proposed. For example, the use of a variety of software-testing techniques, including dynamic analysis, black-box testing, fault injection and behavior monitoring, are suggested to identify cross site scripting vulnerabilities. Alternatively, dynamic techniques on the server side can be used to track non-validated user input while it is processed by the application. This can help to detect and mitigate cross site scripting flaws. Client-side solution to protect users from cross site scripting attempts cannot be applied to the problem of cross site request forgery. This is because cross sire request forgery attacks are not due to input validation problems.

The cross site request forgery attacks appear to be a little known problem in the academic community and, as a result, have only received little attention. The mitigation mechanisms for cross site that have been proposed so far either provide only partial protection, such as replacing GET requests by POST requests, or relying on the information in the Referer header of HTTP requests or require significant modifications to each individual web application that should be protected. This is normally the case when embedding shared secrets into the application's output.

These attacks are still relatively unknown to web developers and attackers. Even so, it is anticipated that the attention paid to this class of attacks will reach that of more traditional cross site scripting attacks in the near future as the attack becomes better known and understood. Unfortunately, current mitigation techniques have shortcomings that limit their general applicability.

IV. CONCLUSIONS

In a cross site request forgery attack, the trust of a web application in its authenticated users is exploited, allowing an attacker to make arbitrary hypertext transfer protocol requests in the victim's name. Unfortunately, current cross site request forgery mitigation techniques have shortcomings that limit their general applicability. Cross Site Request Forgery is one of the top vulnerabilities in the internet. It remains challenging for the researchers to provide a better solution for mitigating this attack. There are many organizations affected by this cross site request forgery attack. Defense mechanisms and existing solutions for cross site request forgery are working in some extend only. The existing mitigation strategies can be extended to provide suitable solutions for the cross site request forgery attack. This may involve applying parsing techniques to identify the attacking spots before the intruders attack. Some pattern for img, script, form, iframe tags can be designed to identify the attack.

REFERENCES

1. Ramarao R., (2009), *"Preventing Image Based CSRF Attacks"*.
2. Sentamilselvan K., Lakshamana K., & Sathiyamurthy K. (2013), *"Survey on Cross Site Request Forgery (An Overview of CSRF"*, Kongu Engineering College
3. Rob S. (2016), *"Cross-site request forgery: Lessons from a CSRF attack example"*.
4. Neil D. (2015), *"Cross-Site Request Forgery Guide: Learn All About CSRF Attacks and CSRF Protection"*.
5. Dave W., Paul P., & Eric S. (2016), *"Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet"*.

6. Luca G., Sara G., & Davide P. (2016), *"Cross-Site Scripting and Cross-Site Request Forgery attacks"*, Network Security.
7. Lance B. (2016), *"Cross Site Scripting and Cross Site Request Forgery"*.
8. Acunetix (2017), *"CSRF Attacks, XSRF or Sea-Surf – What They Are and How to Defend Against Them"*.
9. Dingjie Y. (2017), *"Do Your Anti-CSRF Tokens Really Protect Your Web Apps from CSRF Attacks?"*.
10. Matthew P. & Myers R. (2016), *"Mitigating Cross-Site Request Forgery (CSRF) Attacks"*.
11. Abdalla A. (2015), *" Building a Robust Client-Side Protection Against Cross Site Request Forgery",* International Journal of Advanced Computer Science and Applications.
12. Telikicherla, Krishna C., Venkatesh C., and Bruhadeshwar B. (2014), *"CORP: A Browser Policy to Mitigate Web Infiltration Attacks",* Information Systems Security, Springer International Publishing, pp. 277-297.
13. Singh, Nanhay, Achin J., Ram S., and Rahul R. (2014), *"Detection of Web-Based Attacks by Analyzing Web Server Log Files",* In Intelligent Computing, Networking, and Informatics, Springer India, pp. 101-109
14. Batarfi, Omar A., Aisha M. Alshiky, Alaa A. Almarzuki, and Nora A.(2014), *"CSRFDtool: Automated Detection and Prevention of a Reflected Cross-Site Request Forgery."*
15. Nenad J., Engin K., and Christopher K. (2015), *"Preventing Cross Site Request Forgery Attacks"*.
16. Kavitha D., Akshaya M., Karthick M., Baghya K., Gomathi E. (2016), *"Prevention of CSRF and XSS Security Attacks in Web Based Applications",* International Journal of Innovative Research in Science, Engineering and Technology.

17. Kadambari C., & Manisha T. (2016), *" Prevention of CSRF Attack using STG pattern and JSED*", International Journal of Applied Engineering Research.
18. Kombade, Rupali D., and Meshram B. (2012), *"CSRF Vulnerabilities and Defensive Techniques",* International Journal of Computer Network and Information Security (IJCNIS*)* 4.1 (2012): 31.
19. Sentamilselvan K, Lakshamana S., Pandian and Sathiyamurthy K. (2013), *"Survey on Cross Site Request Forgery"*.
20. Jaya G. and Suneeta G. (2016), "*Server Side Protection against Cross Site Request Forgery using CSRF Gateway",* Journal of Information Technology & Software Engineering.