AN ENHANCED MECHANISM FOR PROTECTING WEB APPLICATIONS FROM CROSS SITE REQUEST FORGERY (CSRF)

Ele. B. I.

Department of Computer Science, University of Calabar, Calabar, Nigeria

mydays2020@gmail.com, +2348064451381

Cite this article:

Ele. B. I. (2024), An Enhanced Mechanism for Protecting Web Applications from Cross Site Request Forgery (CSRF). British Journal of Computer, Networking and Information Technology 6(1), 1-17. DOI: 10.52589/BJCNIT-R5YYKXKA

Manuscript History

Received: 30 Oct 2023

Accepted: 20 Nov 2023

Published: 2 Jan 2024

ABSTRACT: *Cross-Site Request Forgery (CSRF) is considered one of the top vulnerabilities in today's web, where an untrusted website can force the user browser to send an unauthorized valid request to the trusted site. Legitimate users will lose their integrity over the website when the CSRF takes place. So far, many solutions have been proposed for the CSRF attacks such as the referrer HTTP header, custom HTTP header, origin header, client site proxy, a browser plug-in, and random token validation, but in this research, the use of random token validation to solve the problem of CSRF attacks was implemented. The proposed solution in the study used a concept known as Nonce which is a type of random token attached to requests sent over a server. This solution proved to be effective enough to protect and prevent web applications from CSRF attacks. Although no system can be secured against attacks, this study recommends that the random token validation (Nonce) approach of protecting web applications should be adopted by all web applications developers and users to improve on the already established solutions to reduce the chances of attacks on web applications to a very slim percentage, if not eliminated.*

KEYWORDS: Web Applications, Website, Cross-Site Request Forgery, Attacks, vulnerability, Random Token Validation.

INTRODUCTION

The use of the internet is tremendously increasing with the advent of new technology every day. It is now used for a list of diverse activities that can be performed online. Web applications play an important role in providing the appropriate services for these activities; they have become an integral part of human beings in recent times. Some of these are reducing their efforts like (reservation systems, online banking, etc.) and some are entertaining and connecting them socially (Facebook, Instagram, Twitter, etc.). But all these facilities also have their problems, that is, web application attacks. Web application attacks create an insecure environment for web application users which can result in huge losses.

Cross-Site Request Forgery (CSRF) is a kind of a web application vulnerability that allows a malicious website to send unauthorized requests to a vulnerable website using the current active session of the authorized users. It is also known as hostile linking and can also be referred to as a class of attack that affects web-based applications with a predictable structure for invocation (Sentamilsevan & Prasath, 2014).

Nowadays, the Internet plays an important role in business people and commercial use. Everyday life becomes easier for internet users because of the progression in the technologies, but some vulnerability moves the web application to a risky environment. Even though many Internet users get increased, the attackers also get increased in balance. So the security providence becomes a must in the case of a secure organization, defense personnel, and financial bank that interact with the public. The aim of any company is to provide a secure web service for its customers in the case of the web environment and to safeguard the web from threats (Sentamilsevan, Lakshmana, & Ramkumar, 2014).

CSRF attacks occur when a malicious website causes a user's web browser to perform an unwanted action on a trusted site. These attacks have been called the "sleeping giant" of web-based vulnerabilities because many sites on the Internet fail to protect against them and because they have been largely ignored by the web development and security communities, CSRF attacks do not appear in the "Web Security Threat Classification" and are rarely discussed in academic or technical literature (Khurana, & Bindal, 2014).

According to Zeller and Felten (2008), CSRF attacks exist because web developers are uneducated about the cause and seriousness of the attacks. Web developers can also be under the mistaken impression that defenses against the better known Cross-Site Scripting (XSS) problem protect against CSRF attacks.

Problem Definition

A report was submitted by Open Web Applications Security Project (OWASP) in 2013 on vulnerabilities based on critical web applications which can be demoralized. From the survey, Cross Site Request Forgery (CSRF) attack was ranked 5th position, because this attack is harsh against web applications. Most web developers do not know much about CSRF attacks, which is a common vulnerability among various attacks. In this attack, victims are forced to perform unwanted actions on a trusted website, without any user interactions. CSRF is an attack which forces an end user to execute actions on a web application on which he/she is currently authenticated. If the targeted end-user is an administrator account, this can compromise the entire web application. Other issues associated with CSRF include:

a) A request can be issued cross-domain, for example, using HTML forms.

b) An application is vulnerable if it relies solely on HTTP cookies or basic authentication to identify users that issue a request.

c) A request performs some privileged actions within the application, which modify the application state based on the identity of the issuing user.

d) An attacker can determine all the parameters required to construct a request that performs a specific action.

Research Justification

The purpose of this research mainly is to examine the vulnerabilities of web applications exploited by Cross-Site Request Forgery attackers and improve on these vulnerabilities by proposing possible solutions to these problems. This study identifies the various types of CSRF attacks that exist and also the different ways CSRF is used to exploit the vulnerabilities of web applications. By proposing solutions to the problems posed by CSRF on web applications, we hope to make sure that web developers use these methods to secure web applications from being as vulnerable as they have been in the past. Although no system can be secured against threats, we hope to make sure that they can be secured to a very large extent so that users can tread safely without losing valuable information and data to attackers.

This research focuses on describing the implementation of various possible cross-site request forgery methods and describing the pitfalls in the various preventive techniques of cross-site request forgery. It suggests some defense mechanisms to prevent these vulnerabilities.

REVIEW OF RELEVANT LITERATURE

In present times, users are increasing in manifolds; at the same time, attackers are also increasing proportionately. So, the necessity of security in accessing the web is a must for secure organizations, defense personnel, financial banks, and those that interact with the public (Sentamilsevan et al., 2013).

In 2013, the Open Web Applications Security Project reported a list of most critical web application security vulnerabilities that are being exploited:

i) Injection

ii) Cross-Site Scripting

iii) Broken Authentication and Session Management

iv) Insecure Direct Object Reference

v) Cross-site Request Forgery

vi) Security Misconfiguration

vii) Insecure Cryptographic Storage

viii) Failure to Restrict URL Access

ix) Insufficient Transport Layer Protection

x) Invalidated Redirects and Forwards.

CSRF ranked 5th position in this list and it is a severe vulnerability in web-based applications. CSRF attacks are typically as powerful as a user, that is, any action that can be performed by a user can also be performed by an attacker using CSRF attacking techniques. Consequently, the more power a site gives a user, the greater chances of more CSRF attacks on that site. For example, if a user account has administrator privileges, this can compromise the entire web application.

CSRF exploits the HTTP protocol's features so that a web page can include HTML elements that will cause the browser to make requests to other websites. Like all HTTP transactions, the requests to the other sites will include the user's session information such as cookies or HTTP integrated authentication if they have an active/established session. Regardless of if the user has a session with the other sites or not, elements of those sites will be loaded in the victim's browser and can appear in the browser's cache and history. A CSRF can occur on an HTTP request using either GET or the POST method. CSRF attacks generally target functions that cause a state change on the server but can also be used to access sensitive data.

A CSRF attack can be carried out in different ways. The attack could be done using an HTML IMG tag or a specially crafted URL embedded into the target application. This works perfectly since the victim will be logged into the system. Another way of doing it is to host a site/blog and influence the victim to visit the site. Many sites can be used to host a CSRF including online forums (which often allow a user to link to an image in an attachment), HTML email, photo galleries, wikis, blogs, online auctions, and E-commerce sites. CSRF attacks using these sites might not always work as users may not be currently logged into the target system when the exploit is tried. CSRF flaws exist in web applications with a predictable action structure and which uses the above credentials to authenticate users. Therefore, if the user is currently authenticated to the site, the site will have no way to distinguish a malicious request from a legitimate user request.

Overview of CSRF

Below, CSRF attacks are described in more details using a specific example.

An Example

Let's consider a hypothetical example of a site vulnerable to a CSRF attack. This site is a web-based email site that allows users to send and receive an email. The site uses implicit authentication (see Section 2.2) to authenticate its users. One page, http://example.com/compose.htm, contains an HTML form allowing a user to enter a recipient's email address, subject, and message as well as a button that says, "Send Email."

```
<form action=”http://example.com/send_email.htm” method="GET">

Recipient’s Email address: <input type="text" name="to">

Subject: <input type="text" name="subject">

Message: <textarea name="msg"></textarea>

<input type="submit" value="Send Email">

</form>
```

When a user of example.com clicks “Send Email,” the data he entered will be sent to http://example.com/send_email.htm as a GET request. Since a GET request simply appends the form data to the URL, the user will be sent to the following URL (assuming he entered “bob@example.com” as the recipient, “hello” as the subject, and “What’s the status of that proposal?” as the message): http://example.com/send_email.htm?to=bob%40example.com&subject=hello&msg=What%27s+the+status+of+that+proposal%3F3. The page send email.htm would take the data it received

and send an email to the recipient from the user. Note that send email.htm simply takes data and performs an action with that data. It does not care where the request originated, only that the request was made. This means that if the user manually typed in the above URL into his browser, example.com would still send an email. For example, if the user typed the following three URLs into his browser, send email.htm would send three emails (one each to Bob, Alice, and Carol):

http://example.com/send_email.htm?to=bob%40example.com&subject=hi+Bob&msg=test

http://example.com/send_email.htm?to=alice%40example.com&subject=hi+Alice&msg=test

http://example.com/send_email.htm?to=carol%40example.com&subject=hi+Carol&msg=test

A CSRF attack is possible here because send email.htm takes any data it receives and sends an email. It does not verify that the data originated from the form on compose.htm. Therefore, if an attacker can cause the user to send a request to send an email.htm, that page will cause example.com to send an email on behalf of the user containing any data of the attacker’s choice and the attacker will have successfully performed a CSRF attack.

To exploit this vulnerability, the attacker needs to force the user’s browser to send a request to send email.htm to perform some nefarious action. (We assume the user visits a site under the attacker’s control and the target site does not defend against CSRF attacks.) Specifically, the attacker needs to forge a cross-site request from his site to example.com. Unfortunately, HTML provides many ways to make such requests. The <img> tag, for example, will cause the browser to load whatever URI is set as the src attribute, even if that URI is not an image (because the browser can only tell that the URI is an image after loading it). The attacker can create a page with the following code:

```
<imgsrc="http://example.com/send_email.htm?

to=mallory%40example.com&subject=Hi&msg=My+

email+address+has+been+stolen">
```

When the user visits that page, a request will be sent to send email.htm, which will then send an email to Mallory from the user. This example is nearly identical to an actual vulnerability we discovered on the New York Times website, which we describe in Section 3.1.

CSRF attacks are successful when an attacker can cause a user's browser to perform an unwanted action on another site. For this action to be successful, the user must be capable of performing this action. CSRF attacks are typically as powerful as a user, meaning any action the user can perform can also be performed by an attacker using a CSRF attack. Consequently, the more power a site gives a user, the more serious are the possible CSRF attacks.

CSRF attacks can be successful against nearly every site that uses implicit authentication and does not explicitly protect itself from CSRF attacks.

The same-origin policy was designed to prevent an attacker from accessing data on a third-party site. This policy does not prevent requests from being sent; it only prevents an attack from reading the data returned from the third-party server. Since CSRF attacks are the result of the requests sent, the same-origin policy does not protect against CSRF attacks.

Figures 1, 2 and 3 show how CSRF attacks generally work:

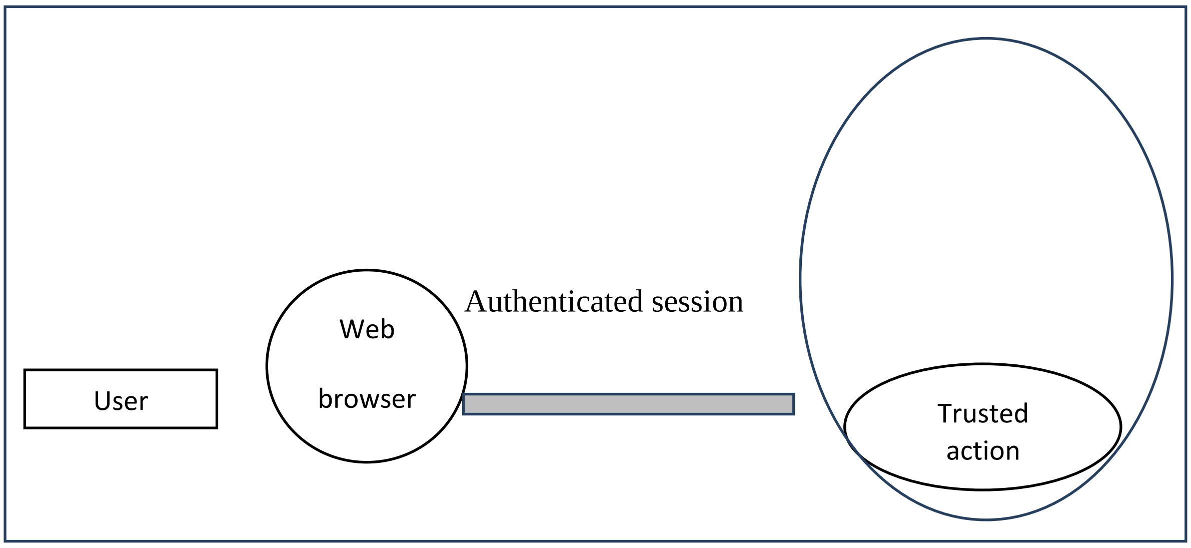


Figure 1: Here, the web browser has established an authenticated session with the trusted site. Trusted action should only be performed when the web browser requests the authenticated session.

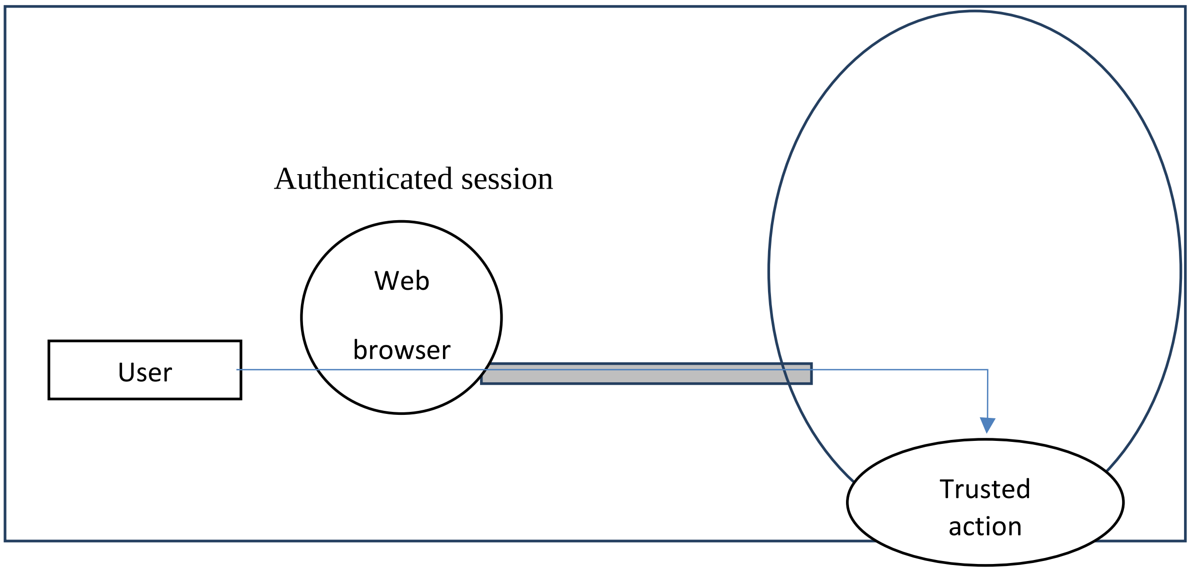


Figure 2: A valid request. The web browser attempts to perform a trusted action. The trusted site confirms that the web browser is authenticated and allows the action to be performed.

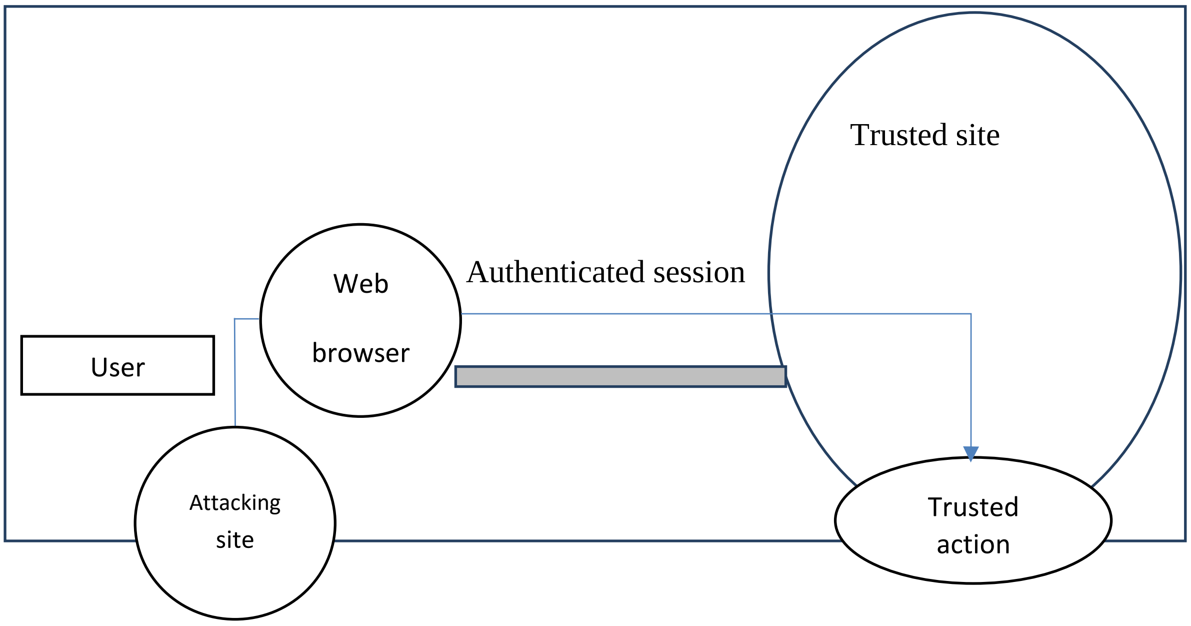


Figure 3: A CSRF attack. The attacking site causes the browser to send a request to the requested site. The trusted site sees a valid, authenticated request from the web browser and performs the trusted action. CSRF attacks are possible because websites authenticate the web browser, not the user.

Classifications of CSRF

CSRF is one of the powerful vulnerabilities where unauthorized activities can be carried out by an attacker without a user's knowledge. This vulnerability can be classified as follows:

a. Stored CSRF

A stored CSRF vulnerability is one where the attacker can use the application itself to provide the victim the exploit link or other contents that direct the victim's browser back into the application and cause the attacker's controlled actions to be executed as though it were the victim. Any application that uses HTML is vulnerable to a CSRF attack where the attacker stores the malicious code using the IMG, IFRAME tag, or XSS. Examples of these tags are shown below:

- <img> tag

  <imgsrc=”http://127.0.0.1/klog.js”>

- <iframe> tag

  <iframe src=”http://127.0.0.1” width=100% height=100%></iframe>

- <link> tag

  <a href=”http://127.0.0.1/klog.js”>click here</a>

- <form> tag

  <form action=http://127.0.0.1/klog.js method=“get”></form>

- <script> tag

  <script src=”http://127.0.0.1/klog.js”></script>

The asperity of the site will be high if the attacker can be able to store the CSRF attack on that site. The likelihood is increased because the victim is more likely to view the page containing the attack than some random page on the internet. Stored CSRF vulnerabilities are more likely to succeed since the user who receives the exploits content is almost certainly authenticated to perform actions. Stored CSRF vulnerabilities also have a more obvious trail that may lead back to the attacker.

Reflected CSRF

In this attack, the attacker performs the attack outside the system application by exposing the victim to links or contents that are malicious. This can be done using a blog, an email message, an instant message, a message board posting, or even an advertisement posted in a public place with a URL that a victim types in. Reflected CSRF attacks frequently fail as users may not be logged into the target system at the time when the attack is exploited. The trail from a reflected CSRF attack may be under the control of the attacker, however, and could be deleted once the exploitation process is completed. Here, the page can be submitted automatically; an example of that tag is shown below:

- Auto-posting forms

  <body onload=“document.forms[0].submit()">

  <form method="POST" action=“https://abc.com/fn”>

  <input type="hidden" name="sp" value="8109"/>

  </form>

c. Login CSRF

By using an adversary’s identities, this attack can be implemented. Making a forgery request with the identity of the victim here, personal data of the victim are collected in the process, the main objective of this type of attack is to make both the attacker and the users authorized users. But the attacker will send his identity to the user utilizing some arbitrary link; thus, the user will now enter using the attacker’s identity. Then all the information related to the user will get stored in the user browser but actually in the attacker’s identity. This will allow the attacker to be able to access all the user information. Therefore, they input personal data to the web server, such as credit card numbers or search keywords. Moreover, by using Gadget, adversaries can execute malicious gadgets registered to the adversaries' accounts within the victim's local machine. Both CSRF and login CSRF are subtle exploitations based on the common belief that every web request made by a browser is initiated under users' supervision. However, in the real world, there are many ways for malicious adversaries to initiate page requests from the victim's browser. Here is an example script for a login CSRF:

```
<form action=https://www.google.com/login method=POST target=invisibleframe>

<input name=username value=attacker>

<input name=password value=xyz>

</form>

<script>document.forms[0].submit()

</submit>
```

Cross-site request forgery focuses on requests that mutate server-side state, either by leveraging the browser's network connectivity or by leveraging the browser's state. CSRF attacks that mutate browser states have received less attention even though these attacks also disrupt the integrity of the user's session with honest sites. In a login CSRF attack, the attacker forges a login request to an honest site using the attacker's username and password at that site. If the forgery succeeds, the honest server responds with a set-cookie header that instructs the browser to mutate its state by storing a session cookie, logging the user into the honest site as the attacker. This session cookie is used to bind the subsequent requests to the user's session and hence to the attacker's authentication credentials. Login CSRF attacks can have serious consequences, depending on other site behavior.

Authentication and CSRF

CSRF attacks often exploit the authentication mechanisms of targeted sites. The root of the problem is that web authentication normally assures a site that a request came from a certain user's browser, but it does not ensure that the user requested or authorized the request.

For example, suppose that Alice visits a target site T. T gives Alice's browser a cookie containing a pseudorandom session identifier sid, to track her session. Alice is asked to log in to the site, and upon entry of her valid username and password, the site records the fact that Alice is logged in to session sid. When Alice sends a request to T, her browser automatically sends the session cookie containing sid. T then uses its record to identify the session as coming from Alice.

Now suppose Alice visits a malicious site M. Content supplied by M contains JavaScript code or an image tag that causes Alice's browser to send an HTTP request to T. Because the request is going to T, Alice's browser "helpfully" appends the session cookie sid to the request. On seeing the request, T infers from the cookie's presence that the request came from Alice, so T performs the requested operation on Alice's account. This is a successful CSRF attack.

Most of the other web authentication mechanisms suffer the same problem. For example, the HTTP BasicAuth mechanism would have Alice tell her browser her username and password for T's site, and then the browser would "helpfully" attach the username and password to future requests sent to T. Alternatively, T might use client-side SSL certificates, but the same problem would persist because the browser would still "helpfully" use the certificate to carry out requests to T's site. Similarly, if T authenticates Alice by her IP address, CSRF attacks would still be possible.

In general, whenever authentication happens implicitly, because of which site a request is being sent to and which browser it is coming from, there is a danger of CSRF attacks. In principle, this danger could be eliminated by requiring the user to take an explicit, unspoofable action (such as re-entering usernames and passwords) for each request sent to the site, but in practice, this would cause major usability problems. The most standard and widely used authentication mechanisms fail to prevent CSRF attacks, so a practical solution must be sought elsewhere.

Capabilities of CSRF Attacks

As has been noted, CSRF attacks allow an attacker to send HTTP requests to a third-party site using the victim's web browser. That is a pretty wide net. Listed below are some actions a CSRF is capable of carrying out:

1. **Simulate Valid Reports**: When CSRF is being discussed, this is what it typically means. Actions might include:

- Changing a user's email address and then performing a "forgot your password" operation to gain access to the user's account
- Adding an account for a user's blog or other systems
- Transferring funds from a user's bank account
- Putting in buy orders for a stock as part of a "pump and dump" scheme
- Checking for the existence of a file. This can be used to find fingerprint web servers on an intranet.

Activate XSS, SQL Injection: Beyond normal operations, CSRF can be used to exploit vulnerabilities on the target website. When prioritizing remediation, site owners often de-prioritize cross-site scripting on administrative and intranet systems, XSS attacks that can only be exploited via HTTP POST and SQL injection in parts of the site only accessible to trusted users. With CSRF, all of these are essentially vulnerable to authenticated attackers.

Call Web Services: In some cases, CSRF attacks can call web services. By using an "enctype" attribute of an HTML form, attackers can pass semi-valid XML to a web service endpoint, for example:

```
<form action=”http://target.example.com/webservice” method=”POST” enctype=”text/plain”>

<input name=”<msg><attr><name>a</name><value>b</value></attr></msg>” value=””>

<input type=”submit”>

</form>
```

The one caveat to this is the attacker cannot set request headers, so requests to SOAP web services (which requires a "SOAPAction" header) will not work.

RECENT WORKS ON CSRF

Several researchers have worked on CSRF and how it can be prevented from being exploited on web applications. This study will refer to a few of the works of others on the subject.

Ramarao et al. (2014) presented a client-side proxy solution that detects and prevents CSRF attacks using IMG elements or other HTML elements, which are used to access the graphic images for the web page. This proxy can inspect and modify client requests as well as the application's replies (output) automatically and transparently extend applications with the secret token validation technique.

Zeller and Felten (2008) implemented a client-side browser plug-in that can protect users from certain types of CSRF attacks. They implemented their tool as an extension to the Firefox web browser. Users will need to download and install this extension for it to be effective against CSRF attacks. Their extension works by intercepting every HTTP request and deciding whether it should be allowed. This decision is made using the following rules. First, any request that is not a POST request is allowed. Second, if the requesting site and target site fall under the same-origin policy, the request is allowed. Third, 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 their extension rejects a request, the extension alerts the user that the request has been blocked using a familiar interface (the same one used by Firefox's popup blocker) and gives the user the option of adding the site to a white list.

Johns and Winter (2014) introduced RequestRodeo, a client-side solution to counter this threat. Except for client-side SSL, RequestRodeo implements protection against the exploitation of implicit authentication mechanisms. This protection is achieved by removing authentication information from suspicious requests. They proposed a client-side solution to enable security-conscious users to protect themselves against CSRF attacks. Their solution works as a local proxy on the user's computer.

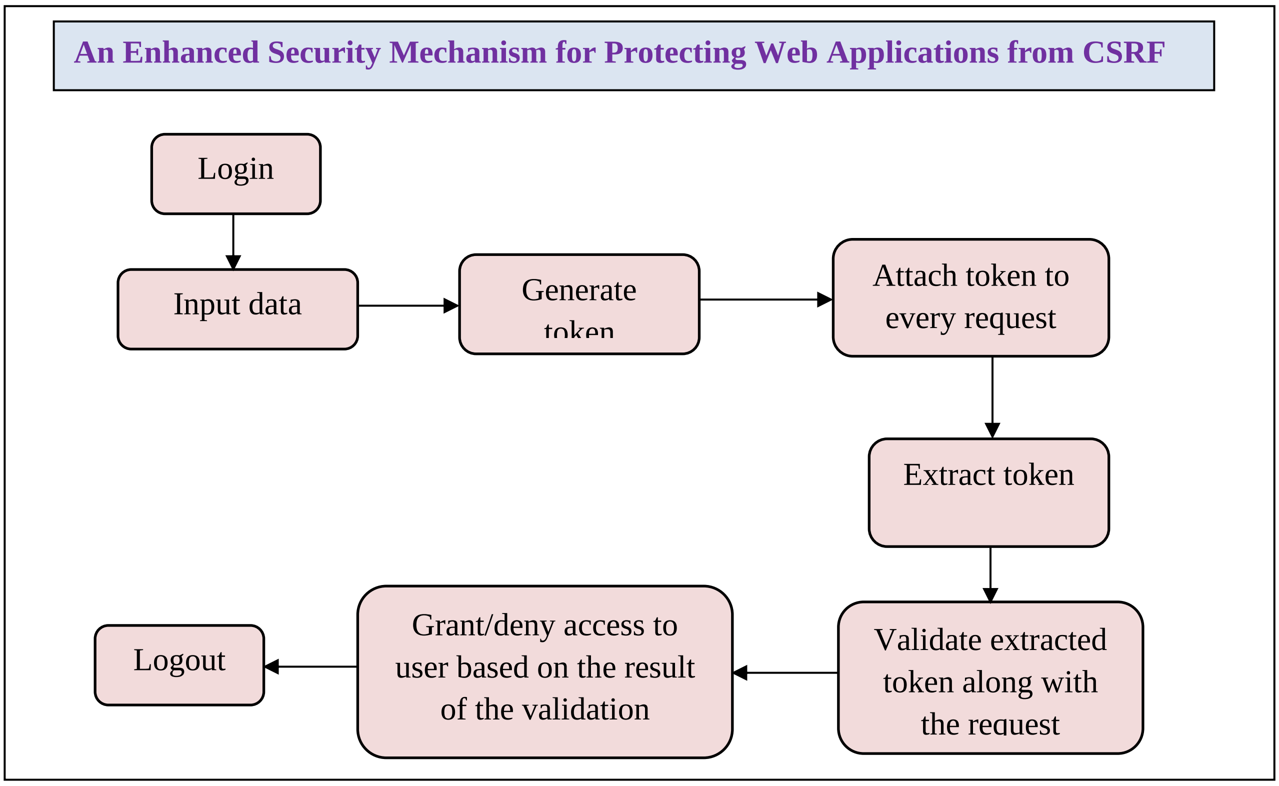
RESEARCH METHODOLOGY

The methodology employed in the course of this study is the agile software development methodology. This methodology is adopted because the requirements of the scope of the study involve the collaborative effort of the researcher and users of systems that have been affected by CSRF to come up with a best-fit solution that will prevent and protect such systems and their users from CSRF attacks.

The method employed to collect data in this research involves studying the works of other researchers about the topic at hand and also using publications about the topic by standard bodies like OWASP and iSEC. The internet which is home to tons of research materials and information on the topic at hand played a vital role in collecting data for the study.

High-Level Model of the Proposed System

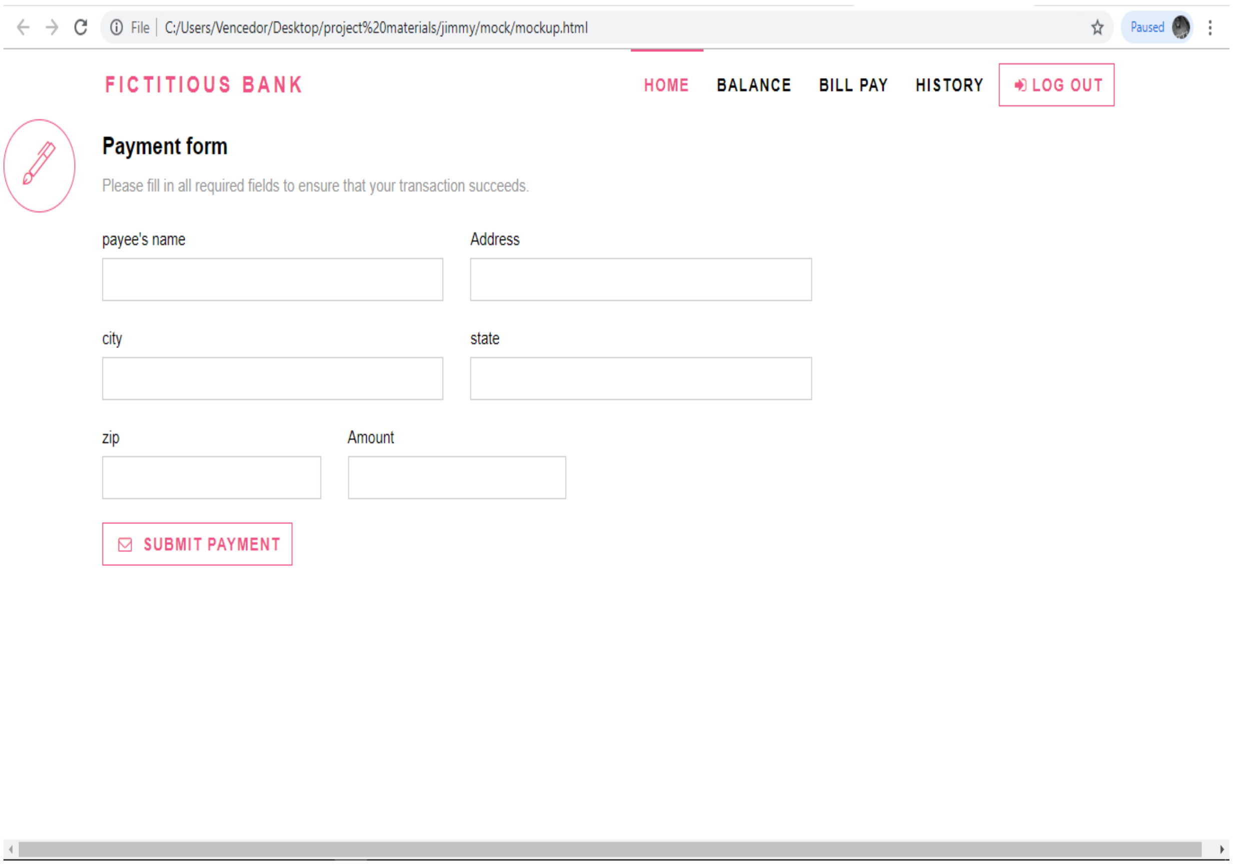
This section illustrates the high-level design of the proposed system. This process gives a clear overview of the procedure of developing the proposed scheme in preventing CSRF attacks.


An Enhanced Security Mechanism for Protecting Web Applications from CSRF
Login
Input data
Generate token
Attach token to every request
Extract token
Validate extracted token along with the request
Grant/deny access to user based on the result of the validation
Logout

The proposed system is a very straight-forward approach which can be very effective in carrying out its function. This approach is likely one of the strongest mechanisms for protecting web applications from CSRF attacks. Although no system can be secured, this approach promises security and protection of web applications to a large extent. It takes care of protecting user data and information by providing extra credentials for data sent to and fro in a system or web application.

Main Menu Design

The main menu used in demonstrating the concept in this study will be a web form for money transactions on a bank's website. The form will look like this:


File | C:/Users/Vencedor/Desktop/project%20materials/jimmy/mock/mockup.html
Paused
FICTITIOUS BANK
HOME
BALANCE
BILL PAY
HISTORY
LOG OUT
SUBMIT PAYMENT

Payment form

Please fill in all required fields to ensure that your transaction succeeds.

payee's name	Address
city	state
zip	Amount

Figure 4: Main menu design for testing CSRF in this study

RESULTS AND DISCUSSION

In the proposed solution to preventing and protecting web applications against CSRF, there has to be a clear process or pattern flow that the web application has to follow (some sort of protocol). On sending a POST request from a web form, the server performs the following processes:

i) A unique identifier, preferably called a CSRFToken is created in the form of randomized characters; the generate_token() module will take care of generating this unique identifier that will be sent to the server.

ii) The validate_token() module will take care of comparing the token generated from the form against the one on the server.

iii) If the token does not match, the log_CSRF_attack() module is fired and then the error_and_exit() module is also fired respectively.

iv) Otherwise, if the generated token matches on the server, the user is granted access to carry out whatever activity they wish to perform.

CONCLUSION

In this study, the researcher discussed CSRF vulnerabilities which will help to understand CSRF attack scenarios and causes behind it, and also mechanisms that can be used to protect web applications against CSRF attacks. In this study, the researcher proposed a concept known as Nonces as a mechanism for protecting web applications against CSRF; we also highlighted the different types of CSRF that exist, how they work, and their capabilities. The researcher also discussed the works of others related to this study.

Cross-site request forgery is a subtle attack technique that can be extremely powerful. In some cases (such as attacks against a user management system that allow an attacker to create administrative users), it can lead to the complete compromise of a web-based system. In other cases, the impact is minimal.

Website owners can defend their sites against CSRF by validating the origin of requests or by requiring secret information to accompany requests. Users may protect themselves by limiting their exposure to potential attacks.

As per the analysis, it is found that Nonces are very powerful techniques but still cannot provide full protection; they can only minimize the CSRF attacks. Hence, complete protection against CSRF is not available and our discussed techniques need more improvement so that they can completely protect the application. Robust and strong protection mechanism against CSRF is needed to protect web applications.

REFERENCES

Sentamilsevan, K, & Prasath, T. (2014). A Comprehensive Study of Cross-Site Request Forgery with Comprehensive Scrutiny. *International Journal of Suitable Science and Engineering*, 2(1).

Sentamilsevan K, Lakshmana S. P, & Ramkumar, N. (2014). Cross-Site Request Forgery: Preventive Measures. *International Journal of Computer Applications,* 106(11).

Burns, J. (2007). Cross-Site Request Forgery: An introduction to a common web application weakness. https://www.isecpartners.com Retrieved: 29/10/2019

Rupali D. K., & Meshram, B. B. (2012). CSRF Vulnerabilities and Defensive Techniques. *International Journal of Computer Network and Information Security,* 1, 31-37.

Sentamilsevan K, Lakshmana S. P, & Sathiyamurthy, K. (2013). Survey on Cross-Site Request Forgery (An overview of CSRF). *IEEE – International Conference on Research and Development Prospects in Engineering and Technology,* 5.

Deepa, G., & Thilagam, P. S. (2016). Securing Web Applications from Injection and Logic Vulnerabilities: Approaches and Challenges. *Journal of Information and Software Technology*, 74, 160 – 180.

Garcia-Teodoro, P., Diaz-Verdejo, J., Tapiador, J., and Salazar-Hernandez, R. (2015). Automatic Generation of HTTP Intrusion Signatures by Selective Identification of Anomalies. *Journal of Computers and Security*, 55, 159 – 174.

Jazi, H. H., Gonzalez, H., Stakhanova, N. A., and Ghorbani, A. (2017). Detecting HTTP-based Application Layer DoS Attacks on Web Servers in the Presence of Sampling. *Journal of Computer Networks,* 121, 25 – 36.

Kar, D., Panigrahi, S., and Sundararajan, S. (2016). Sqligot: Detecting SQL Injection Attacks using the Graph of Tokens and SVM. *Journal of Computers and Security,* 60, 206 – 225.

Mazur, K., Ksiezopolski, B., and Nielek, R. (2016). Multilevel Modeling of Distributed Denial of Service Attacks in Wireless Sensor Networks. *Journal of Sensors,* 13(2), 35-46.

Razzaq, A., Anwar, Z., Ahmad, H.F., Latif, K., and Munir, F. (2014). Ontology for attack detection: An intelligent approach to web application security. *Journal of Computers and Security,* 45, 124 – 146.

Singh, K., Singh, P., and Kumar, K. (2017). Application layer HTTP-get Flood (DDoS) attacks: Research landscape and challenges. *Journal of Computers and Security,* 65, 344 – 372.

Wichers, D. (2013). OWASP Top Ten Project. https://www.owasp.org/ Retrieved 12-03-2020.

Webappsec (2017). Webappsec resources. https://danielmiessler.com/projects/webappsec_testing_resources

Caviglione, L., Merlo, A., & Migliardi, M. (2012). Green-aware security: Towards a new research field. *Journal of Information Assurance and Security*, 7(6), 338-346.

Vala, R., & Jasek, R. (2011). Security testing of web applications. *Proceedings of the Annals of DAAAM and Proceedings*, 1533-1535.

Grossman, J. (2007). Whitehat website security statistics report. http://hhs.janlo.nl/articles/Whitehatstat.pdf. Retrieved 02/02/2020

Ahmed, N., & Abraham, A. (2013). Modeling security risk factors in a cloud computing environment. *Journal of Information Assurance and Security*, 8, 279-289.

Kafer, K. (2008). Cross-site request forgery. Technical report, Hasso-Plattner-Institute.

OWASP. (2017). CSRF prevention cheat sheet. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet. Retrieved: 20/02/2020

Akanbi, O., Abunadi, A., & Zainal, A. (2014). Phishing website classification: A machine learning approach. *Journal of Information Assurance and Security*, 9(5), 222-234.

Khurana, P., & Bindal, P. (2014). Vulnerabilities and defensive mechanism of CSRF. *International Journal of Computer Trends and Technology*, 13(4), 2231-2803.

Jovanovic, N., Kirda, E., & Kruegel, C. (2006). Preventing cross-site request forgery attacks. *Proceedings of the IEEE Securecomm and Workshops*, 2006, 1-10.

Zeller, W., & Felten, E. W. (2008). Cross-site request forgeries: Exploitation and prevention. *The New York Times*, 1-13.

Mansfield-Devine, S. (2008). Anti-social networking: Exploiting the trusting environment of Web 2.0. *Network Security,* 2008(11), 4-7.

Menzel, M., Wolter, C., & Meinel, C. (2007). Access control for cross-organizational web service composition. Journal of Information Assurance and Security, 2(3), 155-160.

Bojinov, H., Bursztein, E., & Boneh, D. (2010). The emergence of cross channel scripting. *Communications of the ACM,* 53(8), 105-113.

Shaikh, R. (2013). Defending cross-site reference forgery (CSRF) attacks on contemporary web applications using a Bayesian predictive model. https://sci-hub.tw/https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2226954.

Barth, A., Jackson, C., & Mitchell, J. C. (2008). Robust defenses for cross-site request forgery. *Proceedings of the ACM 15th ACM Conference on Computer and Communications Security*, 75-88.

Jurcenoks, J. (2013). Owasp top wasc to cwe mapping correlating different industry taxonomy. *Critical Watch*, 7-11.