Cross-Site Request Forgery in 2020: Demonstration, Defenses, and Recommendations

Kanoa Haley, *0956712*

Ralph De Castro, *0923223*

***Abstract*— This paper introduces the cross-site request forgery (CSRF) attack. We also provide a demonstration environment and software stack which allows testing and investigation of CSRF attacks. Finally, we provide an updated set of recommendations of defenses against CSRF. These recommendations involve using SameSite cookies in addition to only allowing actionable requests through HTTP POST requests.**

***Index Terms*— Internet, Network security, Web services, Web sites, Browsers, Programming**

I. Introduction

CROSS-Site Request Forgery (CSRF) is a type of attack that occurs when users are being maliciously exploited by an attacker to do an unwanted action on a trusted website.[1] Through the help of social engineering, the users are unaware that they are being tricked into executing an unwanted action while they are currently authenticated on the website. This is particularly interesting since the website does not have a way to tell if the action sent by the victim is legitimate or forged. Since as early as 2001, CSRF has been a known exploit on the web.[2] Because of this, multiple large attacks have been carried out using the exploit on major companies. Some of these include an attack on Netflix 2006, an attack on ING Direct, an online banking app, as well as multiple attacks on YouTube in 2008. There are a lot of ways and security measures to prevent CSRF. However, there is currently no perfect method to solve this issue. Therefore, we propose to investigate more on this issue and find a potential solution to add to the current preventative methods.

This type of attack can be achieved in many ways. One prime example would be forging login requests, more specifically, *login cross site request forgery attacks*.[3] In login CSRF, the attacker forges a request to log the victim in to a vulnerable site using the credentials of the attacker. There are current defense methods that address this issue: validating secret request tokens, validating the HTTP Referer headers, and validating custom headers attached to XMLHttpRequests. However, these methods are not satisfactory to prevent CSRF. Firstly, CSRF can be prevented by using secret tokens. This works by having a token bound to user's session and to include this token in each request. This prevents CSRF since it forces the attacker to guess the victim's session's token. However, in the case of log in CSRFs, this is not enough since log in requests lack sessions where tokens can bind. Another defense mechanism against CSRF attacks is by validating HTTP referer headers. This works by accepting request from trusted sources only. However, this will be ineffective against requests without headers. Lastly, sites can defend against CSRF through the use XMLHttpRequests, which can be implemented in AJAX interfaces. This method works by setting custom headers through XMLHttpRequests and validating this header before processing requests. The downside of XMLHttpRequests is that it requires sites to change their natural site design to accommodate for this method, making it inefficient and time-consuming. In conclusion, these methods prevent CSRF in certain attack situations but does not account to all vulnerabilities. Therefore, we propose to investigate more defense methods and explore other aspects of web security design that can decrease the vulnerability of sites towards CSRF attacks.

II. Objectives

Our objectives for this paper are:

- To detail the modern and applicable preventative methods that may be used to prevent CSRF attacks.
- To demonstrate a simple CSRF attack with configurable defenses and how this demo may be used to investigate CSRF farther.
- To prove the strength of the modern SameSite cookie attribute as a CSRF preventative method.
- To create an up-to-date recommendation of required and suggested practices and methods to provide the best defense against CSRF in a general sense.

III. Design

In this section, the design of the demonstration applications will be investigated and discussed. We will first introduce the overall design of the applications required for this study. Then, we will provide a summary of the technology stack and libraries used for our application services. Finally, once the overall

design is understood and the technologies are known, we will discuss the design decisions and implementation of each web application service one by one.

A. Overall Design

To correctly test and investigate different CSRF preventative measures, we required a simulated test environment. This is needed to simulate different preventative measures used with different attacks without inadvertently attacking a real service.

For the demonstration and investigation software, it was determined that two separate web services were needed. One service which is to be attacked, and another service which an attack may be launched from by the attacker. For these two services, a demo banking server and application would be created, and a demo chat server and page would be created. The banking application would be used as the target of the test CSRF attacks, and the chat app would provide an attack vector through image chats. To illustrate the purpose of these two services in regard to CSRF attacks, please see Fig. 1 which outlines the basic CSRF attack on the demo services.

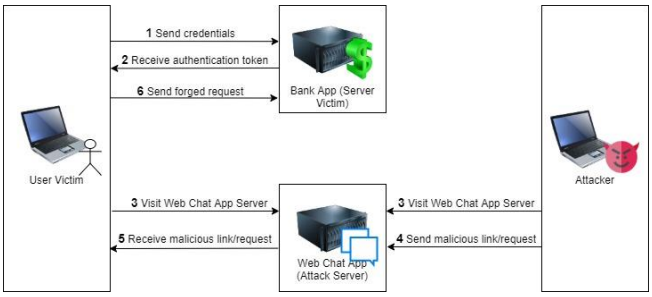

Fig. 1. Overall design plan for CSRF attack testing and demo services including the banking service and the chat service.

B. Technology Stack

For the development of the two demonstration services, the NodeJS environment was used. Both demonstration services are built as independent NPM projects. Both services are implemented within the Express HTTP framework. Both services leverage the "morgan" package for logging purposes and the "nodemon" package for launch and execution control.

The NodeJS environment was chosen for multiple reasons. First, both authors, Kanoa and Ralph, were experienced and familiar with the NodeJS environment. Secondly, NodeJS powered services are becoming one of, if not the, most popular environment used to develop new high-performance web services in the industry.[10] Because of the authors' experience and the current popularity of NodeJS, it was determined to be the ideal environment for our demonstration services.

C. Banking Service Design

The service to be attacked for the demonstration is an online banking service. The online banking service will need to have sufficient functionality to allow proper demonstration of a complete CSRF attack using multiple different preventative measures. The functionality needed for the banking service to be sufficient can be shown as four required components:

1) User Account Login

There must exist functionality to allow a user to log in to a banking account. This component must function as a persistent login. That is, it must store a token on the user's client such that future banking service requests do not require re-authentication while logged in. For our service, browser cookies are used to store the authentication token. In particular, we store a randomly generated token containing no encoded data within the cookie with the key "token". This token value is used as an authenticated session identifier by the server running the bank service. The options of the cookie may be configured through the "cookieConfig" object within the bank service's "index.js" file.

2) User Account Logout

Along with the functionality to allow a user to log in to a banking account, there must exist such functionality which allows the user to log out of any currently logged in bank accounts. For our banking service, this component is implemented as a simple request which deletes the token cookie. This allows a fast implementation and allows a user to project against CSRF attacks by logging out of their banking account when finished.

3) Check Account Balance

For the purposes of investigation and testing CSRF attacks, there must exist some functionality to check the current monetary balance of a banking account. For our banking service, this was implemented as a separate "check balance" web page that requires authentication to access. This page will be used to show the before and after balances of a banking account in the context of a CSRF attack. This page includes an auto reload feature such that the balance displayed will update from the server every ten seconds.

4) Send Money Transfer

For the purpose of creating a reasonably usable banking service and to provide the action that a CSRF attack may take, there must exist functionality to send a monetary transfer through a banking account. This component must include the ability to specify both a transfer amount and the recipient of the transfer. For our banking service, this was implemented through one web page and one HTTP route. A web page was created to act as the UI for the money transfer and a separate HTTP route, "do_transfer.html", was created to be used by the UI. The action route requires the user be authenticated through the token cookie. The action route can be configured to respond to POST requests only, or to respond to both POST and GET requests. This allows the investigation of CSRF attacks and their success when websites mistakenly allow GET requests to have effects.

The action route expects two parameters to be supplied, the "amount" field is a number which identifies the number of dollars to transfer, and the "recipient" is a string that should be populated with the email of the receiver. In our banking service, only the "amount" field has an impact since all transfers simply remove the money being transferred from the user's account and does not add the money to any other balance. Both fields are expected to be present in either the body or the query depending on the

request method. The data supplied is expected to be encoded in the following format: “application/x-www-form-urlencoded”

D. Chat Service Design

The service to be used to launch the CSRF attacks for the demonstration and investigation is an online chat service. The chat service leverages the WebSockets protocol to provide the live multiuser chat communication. The WebSocket connections are managed by the “socket.io” library to simplify implementation and to provide a more realistic use case of WebSockets. The chat service front end is implemented as a single vanilla (i.e. no libraries or frameworks used) HTML web page powered by native browser JavaScript. The web page is designed to prevent XSS attackers by sanitizing and correctly inserting user messages as text nodes. The web page will use the “socket.io” library from the server on initial load. Then, the web page will attempt to ask the user for a “username” that will be used to identity the user on the web chat. Once the web page has determined the user’s intended identifier, it will create a new WebSocket connection to the service’s server and attempt to self-identify the user as soon as connected. The chat service back end is a NodeJS/Express web server which also manages the WebSocket clients and communication channels. The back end provides a single static route to retrieve the generic front-end HTML page and performs all other operations over WebSocket channels. A global channel is used for all communications with chat messages identified as a “message” type communication.

Additionally, there must exist some functionality to allow a GET-based CSRF attack to be carried out. For this, the web chat includes a “Image” message feature. This feature allows a use to choose, instead of the default “Text” message, a “Picture” message when composing a chat message. Instead of the text entry, a “Picture” message will be sent alongside a specified “URL” value. This feature allows a much more diverse multimedia chat environment as well as including the functionality for a CSRF attack to be carried out.

The web chat service is intended to be used only for demonstration and testing purposes but is implemented in a method similar to that of a real web chat service. This is to ensure that our investigations are as applicable to industry as possible.

IV. Attacking with CSRF (Demonstration)

In this section, the default configuration of our software stack will be used to demonstrate a real CSRF attack on a bank account from the web chat service. The perspective of the attacker and the user will be shown. Additionally, comments from the perspective of defending against CSRF will be given where applicable. The demonstration is split into three separate sections so that comments on the demonstration may have a more relevant context.

A. Part One: User Checks Bank Account Balance

To start, our user is in their web browser and navigates to the online banking app. Once on the banking app, the user wishes to view their account bank account balance. To do this, the user navigates to the banking app’s login page (through the menu shown in Fig. 2) and enters their credentials. The user now sees the screen shown in Fig. 3 containing their entered credentials.

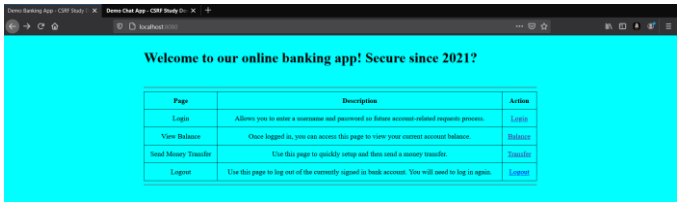

Fig. 2. The main menu of the banking service as seen by the user's web browser.

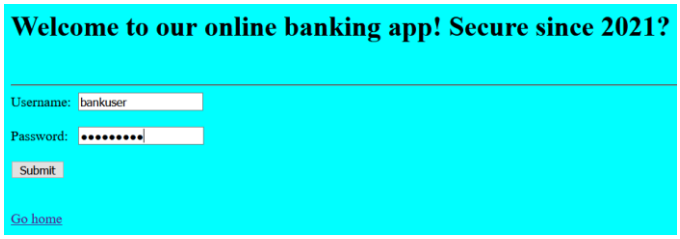

Fig. 3. The screen shown to the user while they enter their login credentials. Username: bankuser. Password: csrfstudy.

At this point in the demonstration, the first key action occurs when the user clicks “Submit” in Fig. 3. When the user submits their credentials, their browser makes a POST request to the banking service with a body containing the user’s entered credentials. The banking service reponds, assuming the credentials are correct, with a page redirecting the user back to the banking service’s home page. But, in addition to this redirect page, the server’s HTTP response also includes a “Set-Cookie” header with a key of “token” and a value representing the user’s authentication token. This response causes the user’s browser to save the token as a cookie for the banking service’s domain.

Because the user’s browser now has a valid authentication token saved for the banking service, all HTTP requests sent to the banking service will include that cookie and thus include the authentication token.

The user, now that they are authenticated, navigates to the account balance page (through the menu shown in Fig. 2.) The user is then shown their current account balance as seen below in Fig. 4.

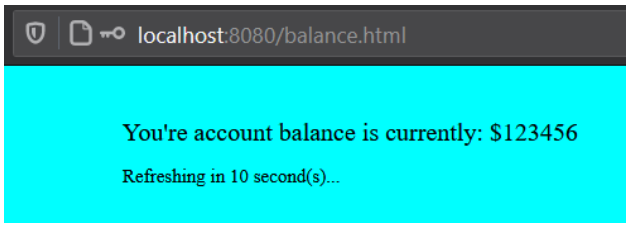

Fig. 4. The account balance page showing that the user's current account balance is $123,456.

B. Part Two: User Visits Web Chat Service

After our fictional user has visited their banking service and checked their balance, they decide they want to socialize a bit and navigate to our web chat service. The user is then asked for a username and choses the value “bankuser” as their username. At this point, the web chat service has connected to the

communications server and the user has effectively joined chat. In our demonstration scenario, there is already a user with the username “Attacker” connected to the web chat service. Once our user joins, the attacker known as “Attacker” decides to act. The attacker knows that our user is a client of our banking service and configures their attack accordingly. The attacker selects an “Image” message and enters the following URL: “http://localhost:8080/do_transfer.html?recipient=Attacker%40mail.com&amount=99999”

The attacker is now ready to act with their CSRF message filled in as can been seen in Fig. 5.

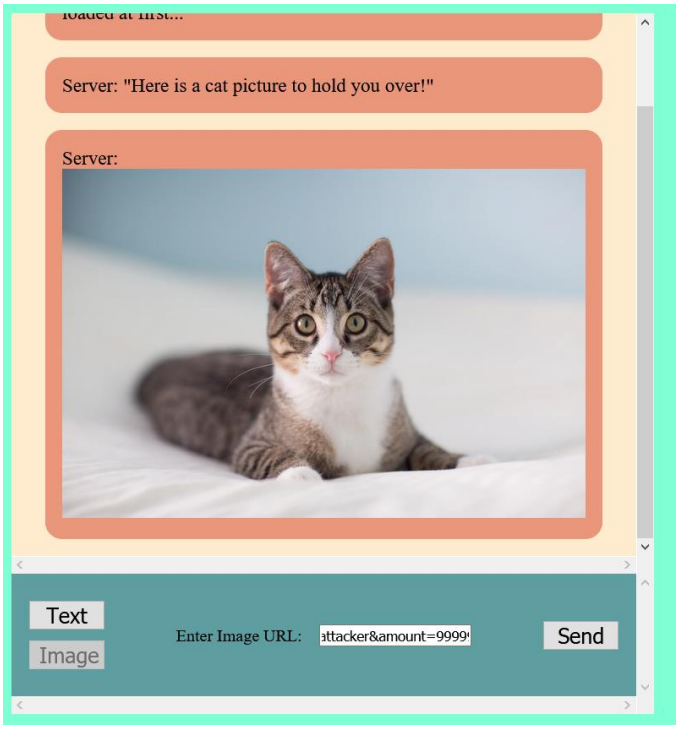

Fig. 5. The web chat as seen by the attacker moments before they carry out their CSRF attack on our unsuspecting user.

The attacker now sends their “Image” message to the web chat service. The web chat service performs as expected and forwards the attackers message to all current chat users. Our bank service user now sees the attacker’s message on their screen. Since no image is shown, our user is unaware of anything occurring and continues on as if nothing has occurred as can be seen in Fig. 6.

Though our user is still unaware, the CSRF attack has already been carried out. When our user received the message sent by the attacker, their browser saw an HTML “img” element with a “src” attribute value where that value is the one specified by the attacker in Fig. 5. This causes the user’s browser to make a GET request to the given URL since the browser assumes an image exists there. This request to the banking service will include the “token” cookie since the domain of the requested URL is the banking service’s domain. As a result, the banking service sees a valid money transfer request with valid authentication and performs the money transfer request as normal.

Despite the user having been attacked and losing money as a result, the user remains unaware of anything out of the ordinary occurring. On the other hand, our attacker has now received an illegitimate money transfer from the user and has thus succeeded in their CSRF attack.

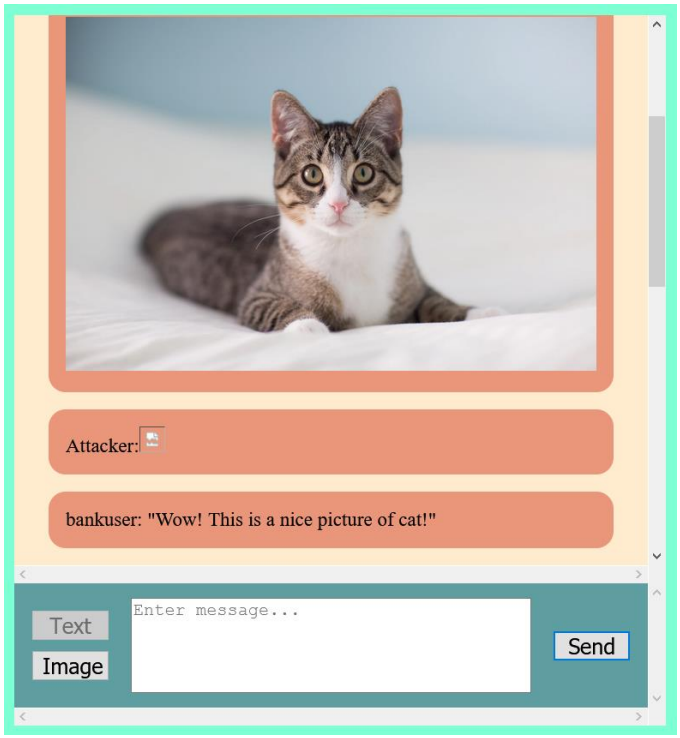

Fig. 6. The web chat as seen by the user moments after they fell victim to a CSRF attack.

C. *Part Three: User Becomes Aware of the Attack*

Once our fictional user has finished chatting, they decide to double check their banking service before they leave their computer for the day. To do this, our user navigates to the main menu of the banking service as shown in Fig. 2. The user then navigates to the banking service's account balance page through the main menu. The banking service then responds with the page shown in Fig. 7 below. The user now sees that their balance is nearly $100,000 less than when they last checked. It is at this point that the user becomes aware of the attack, or at least that an attack has occurred. The user may still not be aware of how the attack happened or even that the web chat service was involved.

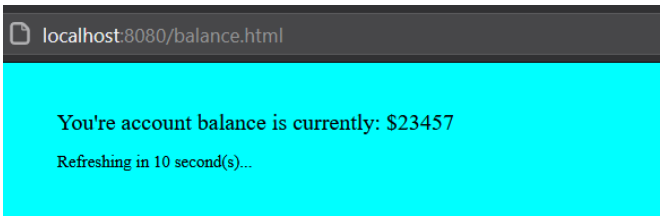

Fig. 7. The account balance page showing that the user's current account balance is $23,457.

In the end, the attacker successfully carried out a CSRF attack against our user which resulted in a monetary loss of $99,999 for our user. The HTTP and WebSocket communications made as part of the attack are shown in Fig. 8 as a sequence diagram.

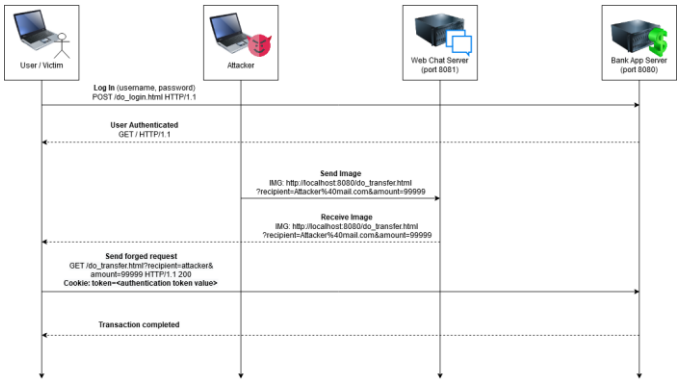

Fig. 8. Sequence diagram of HTTP and WebSocket communications during the CSRF attack demonstrated.

V. Defending against CSRF

In this section, we will discuss the majority of currently popular CSRF preventative methods and include a brief description of how each work. For more information on a specific preventative method, please see our references for relating literature. This is the list that we analyzed to reach our final result given in the Results & Recommendations section.

A. *HTTPS vs HTTP*

Using HTTPS instead of HTTP is always a great practice when sending requests. HTTPS is more secure since it requires SSL certificates, which is signed by CAs. It does nothing to prevent CSRF, however, it is always a good preventative measure against other security attacks.

B. *Post requests only*

Accepting POST requests only is also a great preventative measure since attackers won't be able to construct a malicious link. However, there are methods in which attackers can trick the victims into sending a forged POST request. This can be completed by a simple form on an attacker's website with hidden values, which can be triggered automatically by JavaScript.

C. *SameSite cookies*

A type of cookie attribute that can help users defend against CSRF are SameSite cookies. These cookies can have a value of Strict, Lax, or None. When the SameSite cookie attribute is set to Strict or Lax, the browser will only send these cookies with requests that originated from the same site as the target domain. This effectively prevents CSRF since the attacker needs to be in the same site as the victim to perform the attack. In our simulation, the link from the web application attack would not simply work since it does not originate from the same site as the victim's site.

D. *JavaScript only cookies*

Another viable method to defend against CSRF is isolating your tokens to the JavaScript environment only. That means using localStorage or sessionStorage instead of a cookie. This defends against CSRF since the browser does not automatically include data from the JavaScript environment and thus does not automatically send authentication tokens in a forged request. This method has the downside of preventing the use of HttpOnly cookies and because of that, can potentially introduce an XSS attack vector.

E. *Referer & Origin header*

An early preventative method for CSRF attacks involves the HTTP Referer header. The Referer header will contain the URL of the page which initiated (or "referred") the request. This can be used to defend against CSRF since the server may compare the value of the Referer header against its own known domain. If the value belongs to a separate domain, the server can simply reject the request thus preventing CSRF. The reason this method is no longer used is related to the privacy issues of the Referer header. Since the Referer header includes the complete URL of the site that initiated the request, it leaks far too much private information to the server.

An alternative header, known as the Origin header, was introduced to help mitigate the privacy concerns of the Referer header. Similar to the Referer header, the Origin header will contain the location where the request was initiated from. But unlike the Referer header, the Origin header only contains the domain and not the entire URL. This prevents the privacy concerns of the Referer header.

Despite the new Origin header, this method is rarely used in newer systems since most networks and user agents strip the Referer header and some networks also strip the Origin header.

F. *Session tokens*

Session tokens provide a good defense against CSRF attacks. We will not discuss session tokens since they are not a generalized solution to CSRF in the majority of cases. Additionally, session tokens do not prevent CSRF on their own and require a much more complex server component to exist.

G. *Always log out*

Another preventative method would be to ask and remind the user to always log out when the site is not being used. This social engineering technique decreases the likelihood of CSRF attacks since it prevents the users from being inactive and vulnerable. The attacker can only successfully execute a CSRF attack when the user is authenticated to the target site.

VI. Results & Recommendations

Based on our simulation and analysis of defense mechanisms, we conclude and recommend that setting all authentication cookies to SameSite attribute with values Secure or Lax is the most effective defense mechanism against CSRF We also recommend to set requests to POST only and prohibit effects to be executed in GET routes. Therefore, combining these two defense mechanisms would be very effective and robust against CSRF attacks. The other methods such as using HTTPS instead of HTTP does not fully prevent CSRF, however, they are always a good security practice to implement.

Overall, combining all these methods would further decrease the vulnerability of the site against CSRF attacks.

REFERENCES

[1] Kirsten S. Cross Site Request Forgery (CSRF) [Online]. Available: https://owasp.org/www-community/attacks/csrf. [Accessed: 01-Dec-2020].

[2] Burns, Jesse (2005). "Cross Site Request Forgery: An Introduction To A Common Web Weakness" (PDF). Information Security Partners, LLC. Retrieved 2011-12-12.

[3] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," *Proceedings of the 15th ACM conference on Computer and communications security - CCS '08*, 2008.

[4] Z. Mao, N. Li, and I. Molloy, "Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection," *Financial Cryptography and Data Security Lecture Notes in Computer Science*, pp. 238–255, 2009.

[5] Node.js, "Node.js." [Online]. Available: https://nodejs.org/en/. [Accessed: 04-Dec-2020].

[6] Express, "Node.js web application framework." [Online]. Available: https://expressjs.com/. [Accessed: 04-Dec-2020].

[7] Express, "morgan - npm," Apr-2020. [Online]. Available: https://www.npmjs.com/package/morgan. [Accessed: 04-Dec-2020].

[8] remy, "nodemon - npm," npm, Nov-2020. [Online]. Available: https://www.npmjs.com/package/nodemon. [Accessed: 04-Dec-2020].

[9] D. Arrachequesne, Socket.IO, 28-Nov-2020. [Online]. Available: https://socket.io/. [Accessed: 04-Dec-2020].

[10] W3Techs, "Usage statistics of Node.js," W3Techs, 04-Dec-2020. [Online]. Available: https://w3techs.com/technologies/details/ws-nodejs. [Accessed: 04-Dec-2020].