Login System for Web: Session Management using bcryptjs

Pulkit Sharma[1], Namita Goyal[2]

[1]*Student, Department of Information Technology, Maharaja Agrasen Institute of Technology, New Delhi, India*
[2]*Asst. Prof., Department of Information Technology, Maharaja Agrasen Institute of Technology, New Delhi, India*

---***---

Abstract - *Login Systems have become pretty advance with single login using token (Sign in with Google, Facebook have come up and most of the people use it instead of making new passwords for every application because it goes beyond human limits to remember all the passwords. With my project, I researched about how to setup login system using session ID and during my research I found a relatively new way to use bcryptjs – a JavaScript library of famous bcrypt (C++) to generate new type of session IDs, their possible use cases, and how they can also stand along with present authentication systems for which high security is slightly overkill. This research or methodology is not tested for scenarios including e-commerce or any application which requires real money transaction. This is tested with rather simple, beginner-phase applications which don't require world's best security. Also, this paper doesn't account for database security in general which can be attacked outside the scope of application. This is strictly according to login system where user logs in to an application using username password and how his login session can be maintained without a session id in particular, creating a custom session id in particular.*

Key Words: **bcryptjs, hash, customHash, salt, customString**

1. INTRODUCTION

Login Systems, from their very beginning, are intended to provide the rightful users access to their account, usually on the web. This concept is derived from real life scenarios like card-based entry systems in buildings, fetching cash from ATM, etc. Usually what happens is when a user enters his login credentials, majorly username and password, then these are transported to a server which is listening for such logins, captures the credentials, checks it with database match, matching the username and then password, if match found then a session ID is generated by the system and it is stored in cookies, and on every request those cookies are transported in the header so that authentication takes place at all points. While making a login system, and researching, I found out about localStorage and sessionStorage which were introduced in HTML5, it made somethings very clear and easy for the web developers. localStorage and sessionStorage are a part of Web Storage API introduced with HTML5 standards which grants 5MB of storage on client machine for websites to store some data, every domain is provided its own localStorage and sessionStorage so it is not accessible, that is, another domain cannot access different domain's Storage. But it can be accessed via JavaScript so using it with well written frameworks like Angular or libraries like React which have optimization and background checks for memory leakage or external JavaScript injection. If anyone is making their own application from scratch then they have to take care of any potential JavaScript injection avoidance because JavaScript can access Web Storage API. With that out of the way, let's talk about session ids.

1.1 Saving Passwords

Not a long time ago, passwords were saved in the database as plain texts, which means if anyone manages to get access of the database, then there is nothing stopping the hacker to use that information for malicious purposes. Then emerged the need to save passwords in such a way which, if intercepted, cannot be deciphered by end user. That said, hashing was used as method to store passwords in to database. Now before saving passwords to database, first they are hashed using a salt and slow hash method or algorithm, and then after generating hash, that hash is saved in the database so that if anyone intercepts the database all he/she will get is a hash, and just like hashes are built, they can be generated but deciphering them back to plain text is so difficult or unfeasible that it is considered impossible. Hashing algorithms use a salt, which is a string of random numbers. On feeding plain text or message and salt into hashing algorithm, a hash is generated. There are 2 types of hashing algorithms, fast hash and slow hash. Fast hash algorithms are those which can be computed very fast, but because they can be computed very fast, they are vulnerable to rainbow table attacks also. Examples are SHA1, SHA256 and SHA512. They are generally not used for purposes of hashing a password. For hashing a password, slow hashing algorithm is used which is relatively difficult to crack because it is slow to generate the hash. bcrypt and scrypt are good examples of slow hashing algorithms. So saving hashed passwords is now considered a secure way to store passwords, and hashes generated via a slow hash.

1.2 Bcryptjs

bcryptjs is a version of famous C++ library bcrypt purely written in JavaScript which uses bcrypt slow hashing algorithm to generate hash. Great thing about bcryptjs is that it is progressive hashing algorithm, which means when in future we have more computationally powerful computers, then to generate more secure hashes we don't need to generate new hashing algorithms to match them, but just increase the number of rounds to generate salt for it, because higher the number of rounds, more time it takes for

algorithm to generate salt and more secure hash is generated. So bcrypt has become a good standard for moderate security. bcryptjs has 2 major functions that we are going to use, genSalt(rounds) and genHash(message, salt). genSalt takes in number of rounds, an integer and returns asynchronously a salt, and returns a promise, which on fulfillment returns the salt. Default value for rounds is 10. genHash takes in 2 arguments, plaintext or message and salt. It is also asynchronous in nature so it returns a promise which on fulfillment returns hash, which can then be used anywhere.

2. ALGORITHM

This Algorithm works in applications that make use of a Frontend (Client-Side Application) working in collaboration of Backend (Server-Side Application). Although, this application works and runs on Backend (Server-Side Application).

Also, This Algorithm presumes that there is a working logic in place to match username and password from one provided by user to one saved at the time of registration. This strategy/ algorithm starts working as soon as a user is authenticated by Server-Side Application after matching credentials.

When a particular user is authenticated, a customHash will be generated that will be unique to that user. This customHash will be generated using user's string type information like name, email, and concatenating them into a single long string of length less than or equal to 72 characters. Characters in the string can be shuffled also to make it even more secure so that it is difficult to replicate the customHash.

Although without shuffling this system works due to progressive nature of bcrypt hashing system. While hashing, message string and salt is provided. Salt generation is random and is done by genSalt(rounds) function of bcryptjs. So even if we use same string/message to generate hash, it will still be unique due to uniqueness of salt+message.

To generate customHash, customString is created by using user's string type information like name, email, address, even recordId, concatenate it and one can even randomize the character placing but that is optional. Generate Salt using genSalt(rounds). This returns a promise which on resolution gives salt, so in callback function use salt and customString to call bcryptjs' hash(message,salt) function which returns a promise which on resolving returns a 60 character long hash. We will call this hash customHash.

After customHash is generated, save it in database in customHash field (you must create a customHash field while making database schema) and send response back to Client-Side Application with customHash.

On Client-Side Application, save this customHash and other data in window.localStorage or window.sessionStorage. Decision depends on whether your application contains any "Remember Me" prompt which asks user consent to keep their session alive until they logout. If their consent is yes, save it in window.localStorage, and if their consent was no, save it in window.sessionStorage.

Navigation to new authenticated routes will either include route guards (like Angular, ReactJS) or you will have to code your own logic to implement auth guard. Auth guard is a piece of code which is run every time one visits a route. That piece of code or logic is usually to check with server-side application or third party whether this session is authenticated.

On every navigation, client-side application will first check whether a user's session is present by checking window.localStorage's length, and window.sessionStorage's length. If both are 0, then auth guard will stop them from visiting that link, and redirect them to other link wherever necessary. But if there are some values, length is not 0, then send userID, customHash back to server-side application to find record by userID and then match customHash.

If customHash matches, then server-side gives a green signal to client-side and navigation is completed, if there is some discrepancy, then customHash is deleted from Server-Side application and server-side sends a red signal to client side and navigation will be redirected to login, window.localStorage and window.sessionStorage will be wiped off, and user will be prompted to login.

3. PSEUDO-CODE

Pseudo code is in broken JavaScript to make it easy to understand how to use bcryptjs functions and where to include application specific logic.

```
// means comment

// Information that will be sent back to Client Side

userInfo = Array

customString = userStringDetail1 + userStringDetail2 + userStringDetail3 ...

customHash = null;

//Custom Hash generation at the time of login successful.

bcryptjs.genSalt(10).then(function(salt){

bcryptjs.hash(customString,salt).then(function(hash){

customHash = hash;

// customHash Generated

userInfo.push(customHash);
```

sendToFrontend(userInfo);

});

});

// Authentication Logic

requestData from Client-Side Application

// UNIQUE ID of user generated by database at the time of registration

requestData.id

// customHash returned by Client side for authentication

requestData.customHash

userDataFromDatabase = find-in-database(id == requestData.id)

if(userDataFromDatabase){

if(userDataFromDatabase.customHash === requestData.customHash){

sendToFrontend(user authenticated. allow navigation)

} else {

// If customHash does not match then wipe it off from database from that user

update-in-database(id == requestData.id, customHash = null)

sendToFrontend(not authenticated. redirect to login and wipe off localStorage and sessionStorage)

}

}

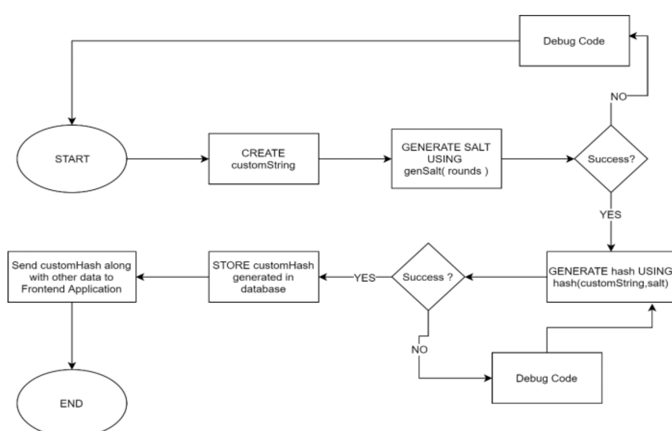

Figure 1 : Flowchart explaining generation of customHash at the time of Login

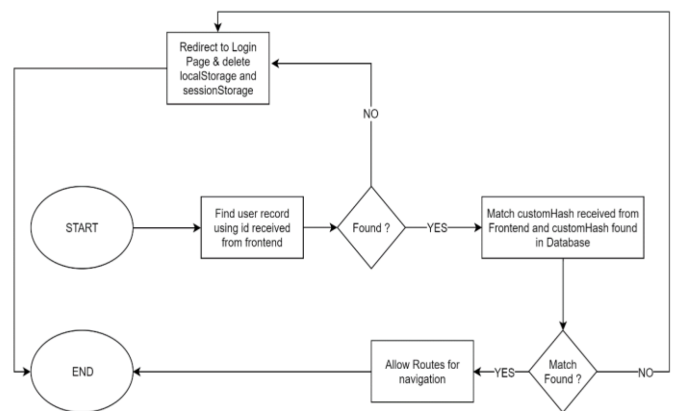

Figure 2 : Flowchart explaining authentication before navigation

4. SECURITY

This Technique is strictly not suitable nor tested for purposes/application which include e-commerce, or where one has no control over security assurance/ quality where application is vulnerable to JavaScript injection. Database security is pre requisite because this strategy doesn't involve relying on trusted third party for storing or doing cryptographic actions.

This strategy is useful for purposes/applications where sensitive information like credit card, banking details are not stored. Some useful examples can be Blogging website, CMS without e-commerce, etc.

This strategy is vulnerable to JavaScript Injection because although Storage API has maintained standards when it comes to safety because a domain can only access its 5MB part of localStorage and sessionStorage so there are no overlaps. But HTML5 Storage API is accessed by JavaScript, so if application is prone to JavaScript injection attack then they will be able to access the storage item, and copying them into their localStorage or sessionStorage, they will be able to hijack the session. So application needs to take care and either use frameworks who implement JavaScript part by themselves like Angular, avoid using functions that can run JavaScript code without any restrictions like JavaScript's evaluate function; although can be very useful in many cases but it is prone to JavaScript Injection.

Hosting the application on https domain also counts because https uses AES encryption and TLS handshake for transferring data on the network which makes it safe to use with e-commerce applications. Then with slight robustness modifications this strategy can be used with e-commerce platform, but because this has not been tested, this is just theoretical and paper does not recommend to use this strategy to be implemented for any application that deals with sensitive information like credit card number, bank details, etc.

5. CONCLUSION

This type of system of manually generating a session id gives a developer more control to make process more secure because password is used just at the point of logging in, after login everything is handled by the id of the user and customHash. This system although moderately secure, is not suitable to use in systems involving sensitive information like a credit card, or any payments system because it has not been tested for it. Also, Web Storage API is secure but it is vulnerable to JavaScript injection. JavaScript injection can be avoided by various checks and methods but still it is bit risky to use Web storage API with sensitive information.

REFERENCES

[1] Girish and Shane, BCRYPTJS API documentation, npm, 2017.

[2] Girish and Shane, BCRYPTJS Github, 2017

[3] Independent Blogs like Stack Overflow and Auth0.