Mittuniversitetet

MID SWEDEN UNIVERSITY

DT167G - Software Security Group Project

MITTUNIVERSITETET

DSV Östersund

Examinator	***Khurram Shahzad***	***raja-khurram.shahzad@miun.se***
Handledare	***Ali hassan Sodhro***	***alihassan.sodhro@miun.se***
Författare	*Marcus Roos* *Einar Sandström* *John Daniel Kyrk*	*maro1904@student.miun.se* *eisa1001@student.miun.se* *joky1900@student.miun.se*
Utbildningsprogram	*Programvaruteknik, 180hp*	
Kurs	***DT167G, Mjukvarusäkerhet***	
Huvudområde	*Datateknik*	
Termin, datum	*VT2021 /* ***2021-05-13***	

Abstract

The number of attacks online made against web applications increases each year and is one of the biggest threats of today's society. There are many different attacks and the biggest three are SQL Injections, Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF). In this experiment three participants set up a web application acting like a guest book. In this guest book users should be able to sign up, sign in, and post messages to their or other users blogs. A functionality to like and dislike posts should be implemented as well as a way to delete your own messages, however, all these features require the user to be logged in, else the website is deemed insecure. Tools used to create the website are Angular, Node.js and MongoDB, as they possess particular properties which makes CSRF, XSS and SQLi attacks unlikely because of already implemented defense mechanisms in place. With the guest book up and running several attacks were conducted to test its strength and try to find a breach, those attacks were unsuccessful and a conclusion can be drawn that the web application is indeed safe against CSRF, XSS and SQLi attacks. This web application is put on a virtual machine with all settings pre-configured allowing the reviewer to boot up the virtual machine, install the image and navigate the website as they please.

1. Introduction

1.1 Background

In web applications being developed there's always a risk for attacks, a vulnerability which an attacker can use for malicious intent. Many applications developed are susceptible to such malicious intent, but the vulnerabilities making the attacks possible can be avoided by proper coding practices [1]. Such attacks can have very serious implications in the end, even if it seems harmless at first sight, such an attack happened to Facebook prior to 2019 when they patched this vulnerability where the data of over 500 million Facebook users were published online as a result of this attack [2].

As web applications are becoming more and more popular, so do the risk of data being leaked, most companies have a website of some sort, whether it be a plain HTML page or a complex website with a lot of underlying functionalities. Attacks against web applications can occur in many different ways but the most common attacks include SQL injection, Cross-Site Scripting and Cross-Site Request Forgery [6]. While the attacks differ in the way they're executed they all have one thing in common, unsanitized user input in one form or another. To execute these attacks the attacker needs to utilize either input fields or sending requests to the server on behalf of the user, in both cases some kind of user input is required.

A Cross-Site Request Forgery attack, or CSRF sends a request to the server on behalf of the legitimate user while it is in fact the malicious user sending the request. Such an attack utilizes parameters in the URL or HTML code and sends a request to the server, in such an attack the attacker can cause harm to the legitimate user by altering their information, such as changing passwords or data, or simply returning data directly to the attacker [6].

Cross-Site Scripting, or XSS, unlike Cross-Site Request Forgery doesn't directly send a request to the server on the attackers behalf, instead XSS attacks focus on unsanitized input fields to execute their code. JavaScript is the language of choice for executing these attacks [6]. On sites which allow different permanent fields, such as social media sites or forums the attacker is able to add text to a forum post, to their profile, or to someone else's profile. In most circumstances the user won't post anything harmful in these fields, if they however want to cause harm they can replace their plain text by adding JavaScript code to the fields and as soon as someone visits the page where this content is posted the JavaScript code added to the field will be executed [4]. By using XSS, famous worms have been spread like wildfire on websites, such as the Samy worm which infected MySpace in 2005, whoever visited an infected profile would get their own profile infected, thus making this worm spread exponentially [12].

NoSQL/SQL Injections use the SQL query to send a request to the database and request data in one form or another, the damages caused by SQL injections and NoSQL injections can be severe and thus require proper protection against [7]. When launching a SQL injection the attacker's power is not limited to retrieving data from the database, but may also alter or even delete data, anything that goes within the SQL query language is possible. The favorable way of attack is by utilizing login fields as it's very plausible the login field is in some way sending data to the database, the username and password needs to be stored somewhere to authenticate users and it's more often than not stored in the database, making login fields susceptible to SQL injections. What differs regular SQL injections and NoSQL injections apart is grammar and syntax used to launch the attacks, both injections require sanitization to avoid vulnerabilities [8].

1.2 Purpose

The purpose of the web application that's being built is to showcase the vulnerabilities found in web applications and apply defensive mechanisms to remedy such weaknesses. The web application being built will be a guest book where users can post messages if they're logged on, give each others posts up-vote or down-votes, search on the website for messages by keywords, as well as look up the different users, the owner of a post should also be able to delete their own messages. For all this to be possible defensive mechanisms need to be implemented, the web application will show what vulnerabilities were avoided and how they were avoided, the web application will however use NoSQL due to node.js and MongoDB instead of regular SQL, meaning the protection will be against NoSQL injections instead of regular SQL injections. In the end the web application is to be run on a virtual machine meaning all its files and required connections need to be set up in a virtual box, this creates a closed environment for testing ground.

2. Related Work

2.1 Security flaws in web applications

There are several vulnerabilities to software, web applications in particular which is where the focus of this study will be. The damages those vulnerabilities can cause range all from improper visuals to actual data leaks which can affect millions of users as shown in the Facebook data leak back in 2019 [2]. The common user don't have access to these attacks, as knowledge of the code is required to execute the attack on the platform the web application is running on, the most common attacks are SQL/NoSQL Injections, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF) [6]. All those methods utilize flaws in the code itself as the code isn't sanitizing or validating user input by itself, it's up to the programmer to solve or avoid those vulnerabilities [5].

2.2 NoSQL/SQLInjection

SQL and NoSQL injections are closely related, but the attacks which worked against SQL databases will no longer work against a NoSQL database, however, NoSQL databases are susceptible to different attacks, known as NoSQL injections, the main difference between the two are grammar and syntax used to execute the attacks [7][8]. NoSQL injections are carried out in the same fashion as SQL injections by utilizing some kind of user input, most commonly being the login field to a website. In such a box the attacker can forge an attack which returns, modifies or deletes data found in the databases. This further points towards sanitizing the user input before sending it to the backend to prevent such attacks [8].

2.3 Cross-Site Request Forgery

A Cross-Site Request Forgery, or CSRF attack, requires the attacker to impersonate a user of the website, such as a Facebook or Twitter user and send a request to the client as this user. By launching a CSRF attack it is possible to post messages, alter profile, add friends, anything that the actual user would have been able to accomplish as a logged in and authenticated user [6]. The primary defensive mechanism implemented to avoid CSRF attacks are session authenticated tokens, and timestamps, which in conjunction aim to make CSRF attacks void as we can see on OWASPs website [11].

2.4 Cross-Site Scripting

Cross-Site Scripting (XSS) attacks utilize user input fields or bad JavaScript coding practices to execute malicious JavaScript code. When poorly handled user input is allowed on the web application the attacker may write malicious JavaScript code into different fields, such as a biography field "*about me*", which will execute on any user's client upon rendering that page. XSS is extremely common and is the most commonly found vulnerability in web applications despite being a well-known vulnerability for many years [10].

3. Methodology

In this paper we are going to develop a web application resembling an online guest book where users can log in, post comments, like posts, and it should also be possible to search among those posts. The focus of the web application will be on the security of the website which will require thorough testing to assure there are no security vulnerabilities.

3.1 Sources & Framework

For creating our website we used Angular, Node.js and MongoDB which utilize the NoSQL database query. We searched different databases to find data relevant to our subject, most commonly used was IEEE. Terms relating to web applications and security were mostly utilized, hence search terms such as "*web application security*" or "*web application NoSQL*" were very frequent. We settled with Angular and Node.js as we are familiar with the frameworks and they both utilize NoSQL, the plan was to split the work into backend and frontend, the backend will serve as an API, a server, while the frontend is sending the requests to the backend. Utilizing this method we need to focus on sanitizing and validating user input on the frontend instead of the backend. If the input is not sanitized or validated a request to the backend won't be sent, by doing this we make sure the user can't send malicious code easily as everything is being reviewed before sent to the server. The sources we found pointed towards NoSQL being less frequent in injection attacks, but SQL-injections as well as cross-site scripting was by far the most frequent attacks, changing to NoSQL requires an attacker to change their attack approach which is beneficial for our website.

We chose Angular because they treat all values as unsafe by default and apply sanitization as well as escape untrusted values to avoid XSS attacks. By using Angular as our framework we shouldn't have any bigger issues with XSS attacks according to their own website. XSS attacks will still be tested on the website despite Angular claiming it should be safe [13]. We also want to add a HTTPS security protocol for transferring encrypted data around as using regular HTTP has proven to be less secure than HTTPS.

For packaging the web application into a virtual machine we chose Oracle VM VirtualBox with an Ubuntu installation because of previous experiences with it. Using VirtualBox we can install the web application and ensure it's running as intended before packaging it together and sending it off for review. VirtualBox allows you to save any of the currently installed VMs in their current state and share it with others, for this VirtualBox seemed like the natural choice for packaging our application.

3.2 Web application vulnerabilities and proposed solutions

As the web application got built we had to take different security flaws into consideration, from the related works we know the most common vulnerabilities and focused our resources on those. The plan to solve those flaws are previously used strategies such as prepared statements for SQL, blocking certain keywords which are

required for malicious content, such characters include but are not limited to: less-than signs, double quotes, single quotes. Where blocking specific characters isn't applicable such as in a post being typed to the board the approach would be to instead embed and alter the characters. Such as a less-than sign would be changed to < instead, as HTML interprets the less-than sign as < instead of '<', while on the other hand JavaScript sees the less-than sign as the start or end of a script this could cause vulnerabilities for the attacker to exploit.

These methods would effectively help against XSS and NoSQL injections, but CSRF requires a different solution, the plan is to go ahead with session token and timestamps, this will prohibit any common CSRF as there's no way for the attacker to retrieve the tokens prior to the request being sent, it will also help against XSS attacks. The tokens required to make these attacks work are sent from the server, to the web client, while a CSRF attack requires creating a link between the user and the web client, if the server receives a request for those tokens without it being sent from the web application no tokens will be returned. Thus meaning a successful CSRF attack would need to be sent from the web client, to the server, and back to the attacker which isn't feasible.

The guest book requires some kind of authentication system for the user to login to their guest book account, requiring a username and a password. Storing a plain password in the database is always bad practice and some kind of encryption is recommended, the proposed solution for the guest book is to use a key encryption method called "AES-256", Advanced Encryption Standard 256 bit. The AES-256 key is known to be a strong and reliable encryption method [9], which is why the solution of choice will be AES-256 encryption hash. With hashed passwords stored in the database a NoSQL attack won't reveal the login details to the attacker, but instead return the plain username but a hashed password to the attacker. To be able to login to the account the attacker will be required to decrypt the password before they're able to login to the website as this user, adding another layer of defense against cyberattacks.

These proposed solutions will guide the guest book application into avoiding the most common web application attacks, XSS, NoSQL injection and CSRF attacks. In the end OWASP ZAP (zaproxy) tool will be used to automate attacks towards the website and try to find potential vulnerabilities in case something was missed throughout the testing process.

3.3 Difficulties and Issues

When creating the web application the focus was to make sure the web application meets all the demands, security was ignored for the moment. This proved problematic once the website was up and running, we used Node.js, Angular and MongoDB when building the website without any defensive mechanisms. As we are using an authentication system for logging in we also store the username and password in the database, albeit the password is encrypted it is still susceptible to SQL attacks which means an attacker can still login as this user by ignoring the password field entirely. No

input goes through any kind of sanitizing either which makes XSS and CSRF attacks possible.

Those issues need to be dealt with, to deal with the SQL injection vulnerability we want to add a prepared statement to our frontend which would nullify any SQL injection attack towards the website as this would place the queries into predetermined variables. To deal with CSRF we want to add authentication tokens, sent by the server which cause CSRF attacks to require this token before sending requests. XSS is also possible as the token is available on the website and code can be executed and include those tokens directly in the code. Input needs to be sanitized and embedded to make execution of JavaScript code fail, a simple solution would be to change “<” to “&lt” and “>” to “&gt” instead, making the <script></script> tags useless as it would translate the < sign to &lt and > to &gt.

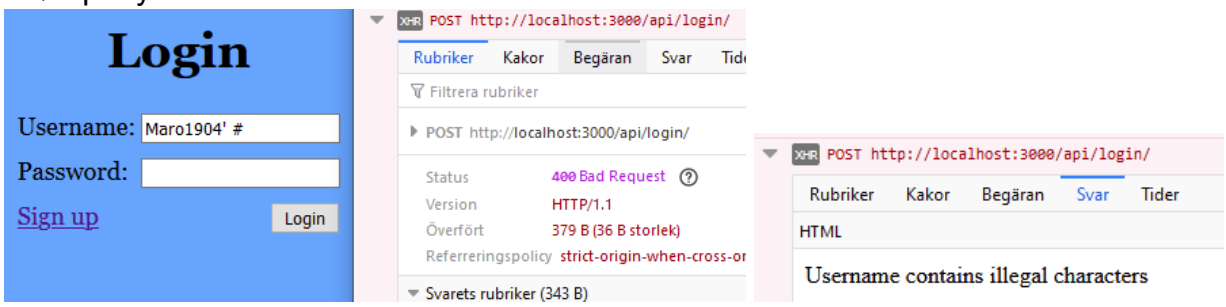
This would effectively turn `<script>alert('XSS');</script>` into `<script>alert('XSS');</script>` which shouldn’t be enough to execute JavaScript code. Porting the whole solution to a VirtualMachine took more effort than anticipated, issues arose with having a communication going between the backend and the cloud database, as well as between the frontend and backend, proposed solutions to those issues are usage of Apache2 to host our web application.

4. Results and Discussion

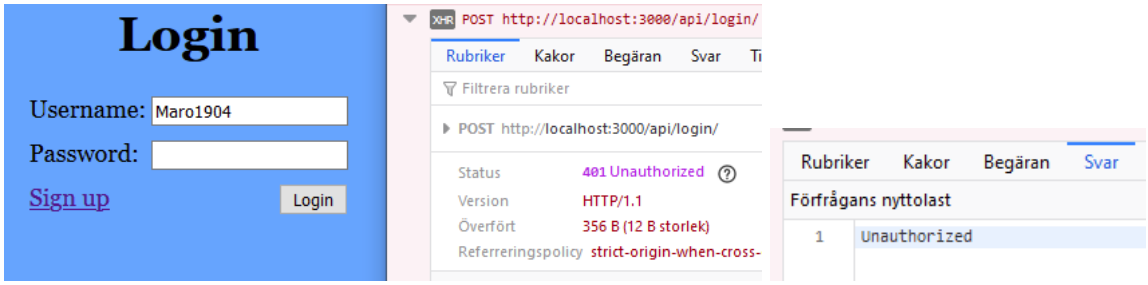
In the methodology we settled for node.js, angular and mongodb frameworks to build our guest book. Once we had the web application up and running a whole slew of attacks towards the site was launched to test the security of the web site.

We aimed primarily to defend against the three vulnerabilities we have experienced in practice, being CSRF, XSS and SQLi, in our case we used NoSQL which makes SQLi irrelevant, thus we focused on NoSQLi.

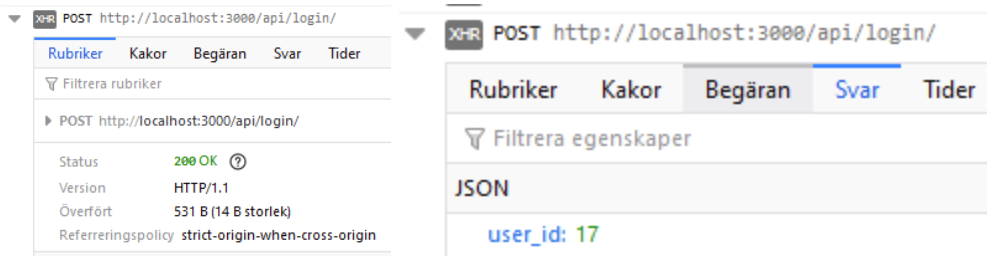
Upon visiting the website the user is greeted with a frontpage requiring authentication or registration to continue to the guestbook, if no authentication is made the user won't be able to visit the guest book and see the contents of the page. Using traditional SQLi to break into the website won't work as it's running on a NoSQL database, as we can demonstrate using a known user's username "Maro1904" and trying to bypass it using SQL query.



This simple injection is failing because the database is running on NoSQL instead of SQL, but we're also receiving a response which indicates the username contains illegal characters. We have implemented defensive measurements to only allow a-z as well as 0-9 when registering an account, as well as when logging in to prevent any kind of injections being made to bypass authentication, as seen when we remove characters the response changes.



Adding a matching username and password our response changes to okay, and redirects us to the website, the user ID is given as a response to this request.



We noticed that our website had a CSRF vulnerability as someone could trick an user into logging out, this proves a CSRF vulnerability being present at a GET request. Using a short code snippet we could logout the user, while this causes no harm it proves a vulnerability is in place which can have severe consequences down the road.

```
</body>
</html>
<html>
<body>
Logged out
<img src="http://localhost:3000/api/logout" alt="LogOut", width="1" height="1">
</body>
</html>
```

We were however unsuccessful in posting a message without being authenticated. The code snippet below executed as intended if we were logged in, it would post a message with the provided text, but if the user is not logged in no message will be posted to the database.

```
<html>
<body>
<script type="text/javascript">
function postMessage()
{
let formfield;

formfield += "<input type='hidden' name='guestbook_user_id' value='17'>";
formfield += "<input type='hidden' name='likes' value='0'>";
formfield += "<input type='hidden' name='message' value='TestMe'>";
formfield += "<input type='hidden' name='message_id' value='0'>";
formfield += "<input type='hidden' name='username' value='Maro1905'>";

let p = document.createElement("form");
p.action = "http://localhost:3000/api/messages/add";
p.innerHTML = formfield;
p.method = "post";
document.body.appendChild(p);
p.submit();
}
window.onload = function() { postMessage();}
</script>
```

Trying to delete with the code snippet failed as well, if we are logged in on the correct user we can delete the post we specified as long as we're the author, but if we are logged in as someone else or not at all, nothing will be deleted.

```
</body>
</html>
<html>
<body>
Deleted message ID 56 on profile John containing text "hello"
<img src="http://localhost:3000/api/messages/56" alt="DeleteMessage", width="1" height="1">
</body>
</html>
```

When the web application were made necessary precautions were taken to make sure there are no CSRF vulnerabilities, there are no hard links for actions (besides the login screen which requires proper authentication) such as "localhost:4200/guestbook/delete" which make CSRF attacks harder as there's no clear link to forge, instead, information is put into a container and the data from within is sent to the backend. This ensures CSRF attacks are improbable from the get go, but having a proper CSRF-Token is an extra step towards a secure application.

After proper implementations of XSRF-token we managed to get rid of the logout vulnerability, we did so by implementing the XSRF-token provided by the Angular framework in our backend. This requires a token to be sent when posting requests, otherwise the request will fail.

No XSS-attacks were successful against the website, the simplest of all XSS attacks were attempted by printing an alert on the website, a short script was added as a post and as soon as the post was made it appears as it should on the website, no execution of the script were made.

We notice that the text is appearing exactly as we typed it in, this is because of the built in sanitation and escaping Angular does whenever untrusted text enters the DOM, thus we don't get what we enter in the text field to display in the guestbook, by entering <> we can see that both brackets appear as they should. This means the sanitation is working as it should and it instead uses the safe html code &lt and &gt to display those characters, making XSS improbable as there's no obvious way to trick script into the DOM.

CSRF-attacks were made to try and post messages, as well as like/dislike a post without being logged in. XSS-attacks were attempted but were shot down in the initial stage. NoSQL injections were attempted but were denied as well thanks to the mongo-sanitize library.

As per the purpose of this study we can determine that all the requirements have been fulfilled and the vulnerabilities we set out to protect against have been protected. Users can only post messages if they are logged in, users can only up/down vote messages if they are logged in. It is possible to search and filter messages by keyword so that all messages containing that word are shown. It's possible to search and filter by user so that all messages posted by this user are shown. Users are also able to delete messages they themselves posted, but not messages posted by other users. We can hence draw the conclusion that all requirements have been met and this study proves that with proper defensive mechanisms a web application can be made secure.

The web application can be accessed as intended through the VirtualBox and it is within the VirtualBox that all the security testing took place. The VirtualBox can correctly be packaged and shared with others and the end user will have the same experience using it as the developers of the web application guest book did.

We installed a HTTPS security protocol on our web application which protects encrypted data being transported across the Internet. We had first started using regular HTTP but noticed encrypted data moving between entities which is a security flaw, that's when we added the HTTPS protocol running on port 443 connected to the backend at port 8000, this adds yet another layer of security for our web application.

5. Summary

We set out to build a web application containing a guest book where users will be able to create accounts, sign in, post messages, like and dislike others posts, as well as delete their own messages and edit their profiles. This naturally creates security vulnerabilities and we identified the biggest ones as XSS, CSRF and NoSQLi, we implemented security measurements to counter these three vulnerabilities along with a new HTTPS protocol and hashed passwords to add yet another layer of protection to our website.

After several failed attempts of trying to breach the web application we conclude that it is relatively safe and achieves the requirements we set out to accomplish. No XSS, CSRF or NoSQLi attack were successful, after careful security defensive mechanisms we failed to attain data from the database, post as another user, post while not being authenticated, we couldn't like or dislike posts if we weren't logged on and we couldn't delete someone else's post. Our web application meets the demands we had set for the project and is running as intended within a VirtualMachine.

References

[1] Web Application Vulnerabilities - The Hacker's Treasure, 2018
K Nirmal B. Janet R. Kumar

[2] 533 million Facebook users' phone numbers and personal data have been leaked online, 2021
Holmes A

[3] [PDF] Security Vulnerabilities of NoSQL and SQL Databases for MOOC Applications, 2017
Hossain Shahriar, Misham M. Haddad

[4] Vulnerability Coverage Criteria for Security Testing of Web Applications, 2018
P.V.R. Murthy, R.G. Shilpa

[5] [PDF] Security strategy for vulnerabilities prevention in the development of web applications, 2019
S Vargas, M Vera, J Rodriguez

[6] Web Application Vulnerabilities: Exploitation and Prevention, 2020
Ajjarapu Kusuma Priyanka, Siddemsetty Sai Smruthi

[7] [PDF] No SQL, No Injection? Examining NoSQL Security, 2015
Aviv Ron, Alexandra Shulman-Peleg, Emanuel Bronshtein

[8] Basic NOSQL Injection Analysis And Detection On MongoDB, 2018
Sachdeva, Vrinda, Gupta, Sachin

[9] Revisiting AES related-key differential attacks with constraint programming, 2018
David Géraulta, Pascal Lafourcadea, Marine Minierb, Christine Solnonc

[10] [PDF] Plague of cross-site scripting on web applications: a review, taxonomy and challenges, 2018
Pooja Chaudhary, B.B. Gupta

[11] Cross-Site Request Forgery Prevention Cheat Sheet (n.d.)
Owasp

[12] [PDF] Social Networks' XSS worms, 2009
Mohammad Reza Faghani, Hossein Saidi

[13] Angular - Security, 2021
Angular

[14] [PDF] Network Security Analysis Using HTTPS with SSL On General Election Quick Count Website, 2020
Faiq Wibowo, Hilal Hudan Nuha, Sidik Wibowo