INF226 - Assignment 3 - Group 60

Kristian Glendrange & Sigve Bergsagel Hølleland

Table of contents

Introduction

Inchat is a forum website, designed by people at UiB. It is written in java, and builds on maven and SQLite. As a forum there are lots of potential security issues. We will begin by explaining what common issues a website might have with inappropriate security measures, before we go in depth and see what issues are in the web application and what measures you might take to avoid them.

Threat model and security design

An attacker can be anyone with access to the server. Things to look after are gaining control of an account or channel. Send malicious code. Collecting sensitive information in a channel or channels or delete data. Send you to another website or modify the website to gain access to your computer. This is what we know:

A website needs to protect itself from buffer overflow attacks, in this case that is not an issue since JAVA which the code is written in is memory-safe and therefore also safe from buffer overflow.

The site is to be used by people who might not know web security. The website partly fixes this by having access control and forcing the user to have a good password.

SQLite is being used to store all the data. SQLite can be a subject to SQL Injection, which would give the hacker your data. This means that it is very important to take countermeasures against SQL injections.

Interacting with the website is done through user input that is sent as a form, and a lot of what is sent back to the client is a result of what the user/attacker has used as the input. By validating the user input before handling it, we can make sure that what the server sends back is correct.

The website has many open fields, which means they are potentially open for Cross-site scripting and request forgery. This is problematic since the hacker not only can get information about users, but also gain access and even full control of channels.

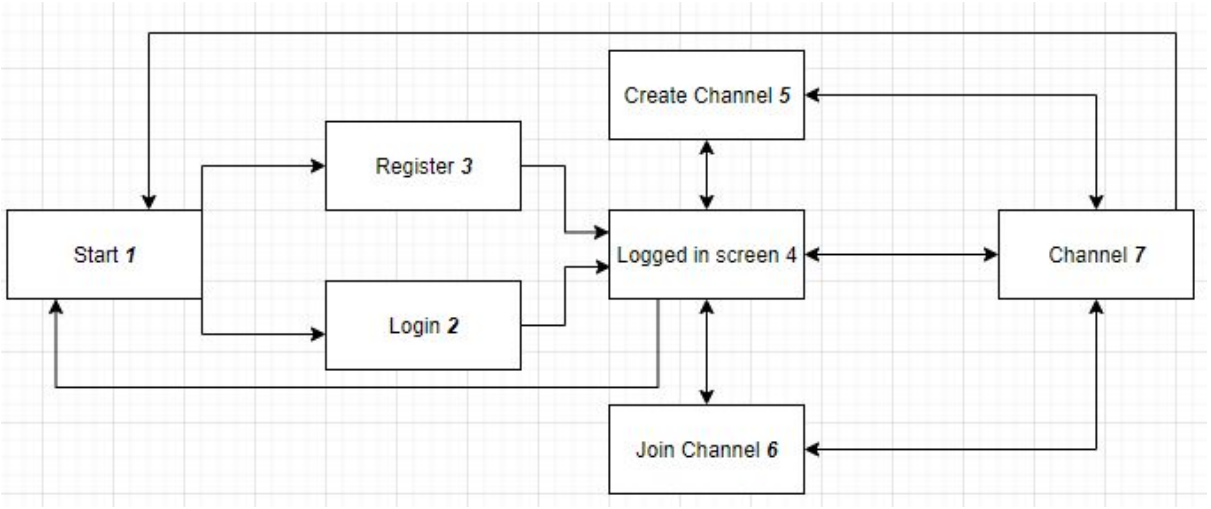
We assume that InChat will not protect against Denial of Service attacks or Harvesting/Scraping data. There will always be a balance between user experience and security. We could force the user to have to do a small amount of computing power for each request, which wouldn't be noticeable to an actual user, but which would add up quickly if an attacker sends tons of requests, but overall this is a low priority issue.
Some user mistakes are not inChats control. For example giving people owner access or sending their password through unsafe channels.

Security measures we found:

- Password requiring a minimum of 8 characters.
- Cookies set with SameSite and HttpOnly flags.
- SQL Injection prevention with the use of prepared statements.
- Cross Site Scripting prevention by sanitizing user input.
- Unimplemented Access Control Panel

Design of the web application

The web application has **7** different html you can be in.



1 The user starts by having two options

2 “Login” and ***3*** “Register”, in login you need to write a username and the associated password in their respective fields. Register makes you enter a username, a password and the password repeated. From there both Login and Register will send you to the

4 logged in screen from there you can see and enter the channels you are in and also create or join a new channel. There is also a Logout button which will take you back to ***1***.

5 If you press “Create a channel!” You get sent to a new page where you can enter a channel name. You will then be sent to the channel(***7***).

6 if you press “Join a channel!”, You need to write a channel ID number. Then you get sent to channel(***7***). If you press an already joined channel you enter that channel(***7***).

7 in a channel you have a field where you can enter text, if there are any messages in the channel already you can delete or edit those. You have a “Create a channel”, “Join a channel” and “Logout” which takes you back to ***5***,***6*** and ***1*** respectively.

There is also a field for setting a user's access in a channel and the user can go to another channel(***7***) by clicking on the channel's name. At any point when logged in you can go back to ***4*** by clicking on “inChat *username*”.

2 3 5 6 7 These pages take user input and therefore are potential subjects for SQL injection, CSRF and XSS attacks.

2 Should not be able to log into a user account without the right password. Could have a cooldown if too many wrong attempts to stop brute force.

3 Should not allow username if already taken. Both password fields must be the same. The password should satisfy the requirements from NIST, The National Institute of Standards and Technology such as password length, but also not allow simple words from a dictionary.

4 Show only the channels that you have joined and are not banned in.

5 Check if channel name is taken

6 Should only work if you have the correct channel id

7 An observer should only see what’s in the chat, Banned should not be allowed in at all. A participant should be able to send messages, edit and delete(?) their own messages. Moderators are the same as participants, but can also delete others messages and set others access to “Participant”, “Observer” or “Banned”. As long as the user is not an owner or moderator.
An owner is the same as a moderator, except he can set anyones access. Only exception is if he is the only owner then he can’t remove himself before appointing someone else.

Testing methods

SonarCube gives a quick overview of the application and where there might be vulnerabilities. It can quickly highlight potential risks and bad quality in the code. However with our main threats being malicious user input, we expect to find the most value from a dynamic tool that can really test the input fields.

OWASP has a dynamic scanning tool for identifying security threats called ZAP. We plan to run penetration tests for the webapp to really see where there are security holes, that static tools might not catch.

Zed Attack Proxy(ZAP) - Setup to recreate our results:
VSCode:

- Change the port in the source code to run on 8081, as port 8080 is already used by ZAP.

Command Line:

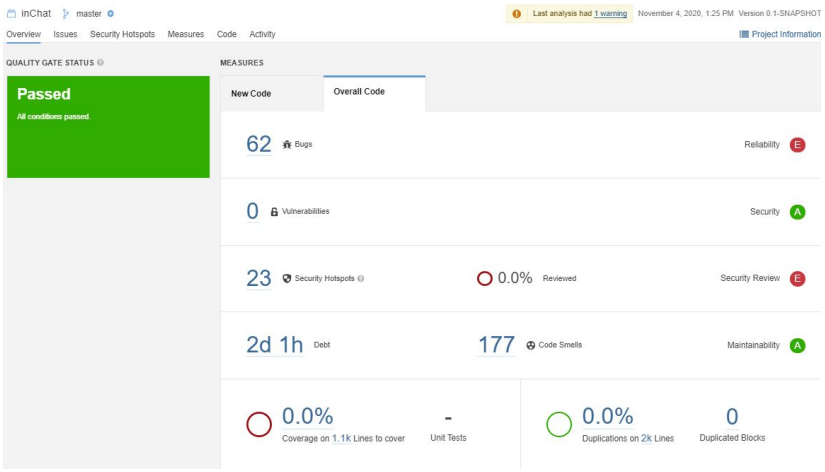
- Compile the changes with *mvn compile*
- Start inChat with *mvn exec:java*

ZAP:

- Use *Manual Explore* to register test users with passwords following the NIST requirements. Add the users to each other's channels.
- Make a new context named *inChat* with top node *localhost:8081*
- Flag the folder *localhost:8081* with the new context *inChat - Form Based Authentication Login Request.*
- *Change the following ZAP session properties:*
    - *Authentication: Change the Login Request Data and include username and password parameters of the user.*
    - *Users: Add the users you created*
    - *Technology: Select the database SQLite and the languages JSP,Java,JavaScript. We ran tests on both Linux and Windows.*
- Spider the web page to make sure ZAP has full logged in access.
- *Run an active scan logged in as a user.*

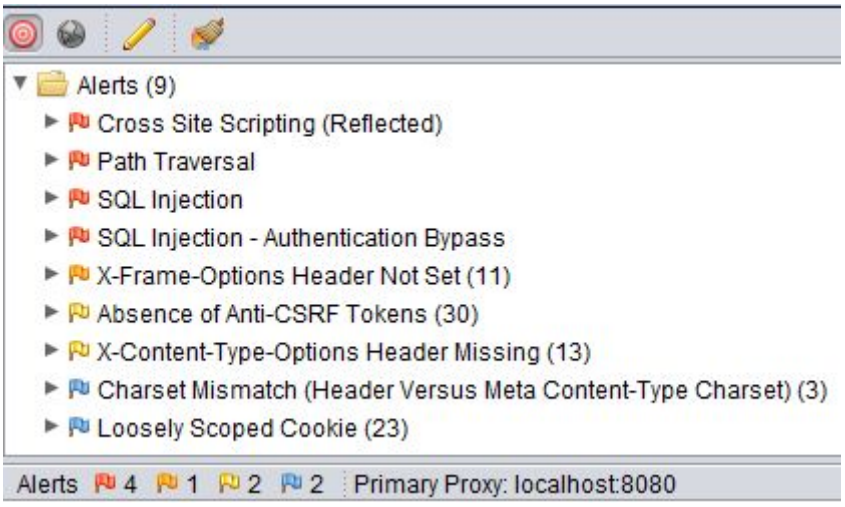
Test results

SonarCube:



- Found 0 vulnerabilities, but with 23 potential SQL Injection vulnerabilities to manually review which. These are all SQL Queries made up from string concatenation.
- Found 62 bugs, mostly concerning missing try-catch blocks around the connection and queries sent to the SQLite database.
- Found 177 code smells, some examples are unused imports, preferring System.out instead of System.err on the console error messages, if-statements instead of switch-statements to increase readability, empty catch blocks and other small improvements it could find.

OWASP ZAP returned with 9 alerts:



High Risk:	**Cross Site Scripting Reflected - CWE ID: 73**
Results:	User input able to make it to the form action URL.
Significance:	Cross Site Scripting Reflected: User input is able to make it to the form action URL from a HTML form. The attacker can then redirect to their own server.
Solution:	Sanitize user input.

High Risk:	**Path Traversal - CWE ID: 22**
Results:	Path Traversal possible.
Significance:	Path Traversal attack technique allows an attacker access to files, directories and commands outside the web document root directory by manipulating the URL.
Solution:	Sanitize user input that will go in the URL.

High Risk:	**SQL Injection - CWE ID: 89**
Results:	SQL Injection Attacks might be possible.
Significance:	SQL Injection vulnerabilities would allow an attacker access to the database. This could either be them able to get privileged data or in the worst case gain full control.
Solution:	Use prepared statements to sanitize input queries.

Medium Risk:	**X-Frame-Options Header Not Set - CWE ID: 16**
Results:	Webpage might not be protected against ClickJacking.
Significance:	A ClickJacking attack is when an attacker tricks the user into clicking a button on a layer they put on top of the original webpage, instead of what they intended to click.
Solution:	Enable X-Frame-Option Header and SameSite flag.

Low Risk:	**X-Content-Type-Options Header - CWE ID: 16**
Results:	The Anti-MIME-Sniffing header is not set to no-sniff.
Significance:	There might be MIME-sniffing related vulnerabilities.
Solution:	Enable no-sniff.

Low Risk:	**Absence of ANTI-CSRF Tokens - CWE ID: 352**
Results:	Missing Anti-CSRF Tokens on HTML forms.
Significance:	Webpage might be vulnerable to cross site request forgery attacks which would allow an attacker the privileges of the victim, and can perform actions on the site using the victims identification.
Solution:	Create Anti-CSRF Tokens on HTML forms.

Informational:	**Charset Mismatch - CWE ID: 73**
Results:	HTTP Content-Type header declares a charset different from the body of the HTML.
Significance:	Potentially opens up for cross site scripting vulnerabilities where non-standard encodings can be used to bypass defensive filters.
Solution:	Be consistent with the charset encoder.

Informational:	**Loosely Scoped Cookie - CWE: 565**
Results:	Cookies can be scoped by domain or path.
Significance:	If a cookie is scoped to a parent domain, it can also be accessed by less secure subdomains. The sensitive information in the cookie may be compromised.
Solution:	Scope all cookies to a fully qualified domain name.

Analysis of threats with explanations and solutions

Cross Site Scripting (XSS) is a big security issue when not considered properly. This happens when user input can be interpreted as code, we fix it by “Sanitizing” what you write so that it is only text. It takes just one error to forget to sanitize it when displaying it on the webpage, so we suggest you sanitize it right away before it is stored in the server.
The main issue was for a non-persistent XSS also known as Reflected XSS, and the vulnerability lies in injecting javascript in the URL which the website will reflect back by running it. It is not as severe as other a stored XSS attack where the webpage always presents it to the victim by for example entering a channel where the script is injected as a message, because in the reflected XSS case you need the user to click on the link which then runs the script from the url. Newer browsers might catch the javascript in the url for you, but the user input can easily be encoded to remove tags, but also make sure to remove the keyword *javascript:* which also works.

Path Traversal would allow attackers the ability to view restricted files and provide information on the infrastructure which could further compromise the system. This is a vulnerability in inChat because the attacker can manipulate the URL from user input of the channel name. Since the SQLite database is a simple file hosted on the server they could potentially get all users and their hashed passwords. Malicious characters like “../” and others should be escaped to prevent directory traversal attacks.

SQL Injection can be prevented with prepared statements. AccountStorage has a function *lookup(username)* that is a simple concatenation of strings. This can be manipulated by an attacker to run any SQL command they want as long as it is valid. They cannot bypass authentication as the password check isn’t done in the database but rather after fetching the user from the database, but they can still do a lot of damage. A lot of other queries have prepared statements, but all it takes is missing it in one time which here opens up security holes.

Anti-CSRF Tokens are lacking on all forms. It is especially important to have tokens on forms like newmessage, createchannel and setpermissions. The other forms contain unguessable id. The login and register form lacking tokens wouldn't create security risks as they are not logged in with your credentials, but it could still create unwanted issues. Without tokens an attacker could perform actions using your credentials and your authorization.

Role-based access control in inChat is non-existent. Manual testing shows that all users have the same access level. This might not be a security vulnerability in the normal sense where you get access to privileged information like someone's password, but in practise it lets you do all the same things as if you were logged in as another user. You can edit their pre-existing comments as if it was something they said, or simply delete all comments in a channel. This heavily limits inChat to work as a trust based application where you can only invite people to your channel if you fully trust them.

Charset Mismatch isn't an huge security error as having different charset encoding like UTF-8 vs iso-8859-1 isn't a problem as they encode ASCI equally, and weird characters outside of asci should be handled by the OWASP Encoder.

Conclusions

We have found some major issues with the website, we will give a short description of how you should approach fixing the issues we found, for more in depth information please read the analysis. The URL has not been "sanitized" which means that clicking on a link could run javascript. This does not take time to fix so we would do this first. Then we urge you to fix the SQL injection problem, since an attacker can potentially do anything with your database where all your information is stored. Then you should make a functional access control, so that you have control of who can do changes in a channel.
The rest of the security holes are difficult to exploit and/or expose very limited information, still in a talented hacker's hands these can create real trouble. Overall we encourage you to solve all security vulnerabilities, especially the three mentioned above, and eventually all.

References:

SQL Injection:
https://cwe.mitre.org/data/definitions/89.html

https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

http://projects.webappsec.org/w/page/13246963/SQL%20Injection

Cross site Scripting reflected:
https://cwe.mitre.org/data/definitions/73.html

https://owasp.org/www-community/xss-filter-evasion-cheatsheet

Path Traversal:
https://cwe.mitre.org/data/definitions/22.html

http://projects.webappsec.org/Path-Traversal

https://www.troyhunt.com/browser-url-encoding-and-website/

Configuration: X-Frame & X-Content
https://cwe.mitre.org/data/definitions/16.html

http://msdn.microsoft.com/en-us/library/ie/gg622941%28v=vs.85%29.aspx

https://owasp.org/www-community/Security_Headers

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Content-Type-Options

Cross-site Request Forgery:
https://cwe.mitre.org/data/definitions/352.html

https://owasp.org/www-community/attacks/csrf

http://projects.webappsec.org/Cross-Site-Request-Forgery

Cookie:
https://cwe.mitre.org/data/definitions/565.html

https://tools.ietf.org/html/rfc6265#section-4.1

https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/06-Session_Management_Testing/02-Testing_for_Cookies_Attributes.html

http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy_for_cookies

Charset mismatch:
http://code.google.com/p/browsersec/wiki/Part2#Character_set_handling_and_detection