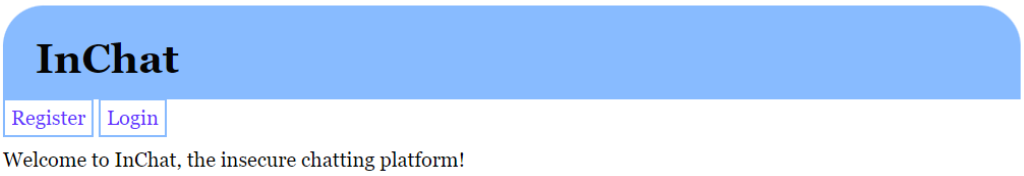
Kasper Melheim

Mandatory 2

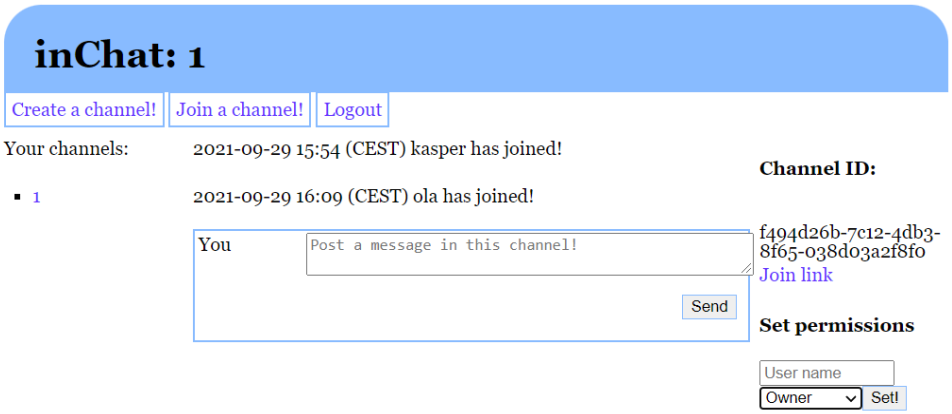
Table of contents:

Introduction

The application we want to analyse is called “InChat”. It is an application where users can communicate with each-other using text messages. This whole thing happens on a website located on localhost:8080 once you have built the program.



The program has a register and login function to get access to the application. Once entered you can either join a channel, create a channel or logout. When you have entered a channel, you are able to communicate with any other potential members of the channel.



When it comes to the code structure the program consists of multiple java classes for different aspects of the application. Some for registering users, some for storage and so on. It also has a test. It also has some static html files, as well as some CSS and JS.

Within this report you can find a description of the threat model and security design of the application, and a short explanation of what security issues and attacks one can encounter,

as well as what we expect the program will protect us from. There is also a short review of the different methods and tools that we will use when trying to breach the program. Towards the end of the report the test results are presented a long with a thorough analysis and explanation of these results.

Threat model and security design

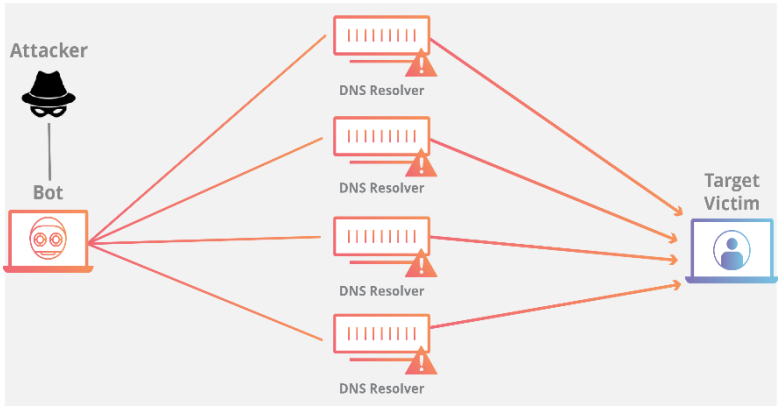
As a user of the system, you will have many expectations for the program. You don't want to use an application that is insecure and could put you and/or others in risk. When using the program, you want to be certain that your sensitive data is secure and can't end up in a data breach.

There are multiple places in the application where data can be transferred. When registering and logging in you enter your username, as well as your password. Once you have logged in you can create a channel or join an already existing one. You here enter the id of the channel you want to join. When you have entered the channel, you can send messages back and forth.

Security guarantees InChat should give to its users when offering such a service are multiple. They should guarantee that their sensitive data, such as registration info, is safely stored and not accessible by other people and used for malicious intent. You don't want your username and password combination leaked anywhere for others to grab that and exploit it. This can go beyond just taking over a InChat account, seeing as many people use the same username and password combinations on other platforms also. InChat should also guarantee that data such as users IP address and similar is not accessible to other users. Skype was vastly used back in the day, and users was able to grab other users IP using special tools. This is not possible on Discord for example.

If a user is in chatroom with exclusive access, InChat should offer guarantee that that chatroom in fact is exclusive, so that users that shouldn't have access to that chatroom doesn't get to read any of the data (messages) transferred between users in that chatroom. The program also has a "set permission" function, where you can add roles or ban users. The users should be assured that the respective permissions are added, and actions are executed when the user click the "Set!" button.

When it comes to the access control system, InChat uses Role-Based Access Control. As mentioned above you can set different roles for users depending on what you want to do. If you want a user to just be able to observe but not type you can give him a fitting role for that. If you want to transfer ownership of the channel to a different user, you can also do that.

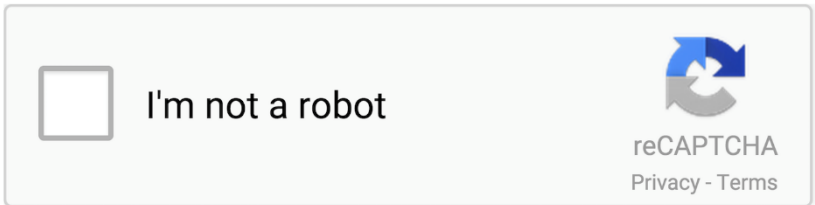


The application can be subject for many different attacks. An attacker can be just a normal user already registered on the site, trying to plant some form of worm or extract sensitive data. An attacker can also be someone on the outside attacking the system with a botted service trying to take the application down (DDOS). An attacker that manages to get his hands on sensitive data and other user's info can not only take over the users account on InChat, but also cross-check the username and password combination on other platforms to take over other account as well.

Other type of attacks can be someone trying to advertise his product, recruit members to other similar applications or for political agenda. If these 3 are done by exploiting bugs in the program, it is looked at as very unethical. Attackers can try to change the profiles of others, to display something in their interests instead, or create popups or fake forms in order to trick users. Some attackers might only be doing it for fun, and you can end up with gifs of cats all over the website just because some guy wanted to have a little laugh.

You also have ethical hackers, or white-hat hackers. These can try to exploit the program in any way possible with the main goal of reporting this to the owners of the application.

Attackers are somewhat limited. They need to abide with the laws and checks in place already, and undiscover the bugs they can exploit by being creative. There are multiple user inputs on the application where they can attempt to insert script in order to exploit the program. When it comes to DDOS attacks, one would assume that a chat platform with a registration form has a proof-of-work implemented (CAPTCHA). InChat does not have this, which makes it vulnerable to denial-of-service attacks.



Testing methods

Based on the threat model, there are multiple things I want to test when it comes to security in this program. The application has many input fields. I want to check if it is possible to paste some code in this field in a attempt to change content on the website or display data to other users in other ways than originally intended (by using the chat function). SQL Injection and cross-side-scripting will be tested. In order to disclose some of weak parts of the program when it comes to security, I will use programs such as

- SonarQube
    - Tool for analysing the source-code without running the application and look for security faults.

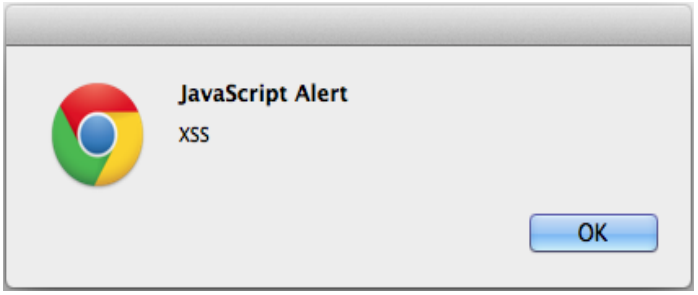
- OWASP Zed Attack Proxy
    - A scanner used to check web applications when they are running for potential security faults. This is a dynamic analysis tool.

I will also play around with the various input fields and try to run some cross-side-scripting. The code itself can also be informative so I will also look at the source code and see what I can find. There might be some comments or functions in the source code which can be useful when trying to disclose security issues.

Test results

Running and using the application:

After logging into the application and entering a chat room, an input field for chatting will be displayed. By inserting: **<script>alert(123)</script>**, an alert box pops up on the screen. This means that running scripts in the input field is possible, and we can now display data on the website in a different way than intended.



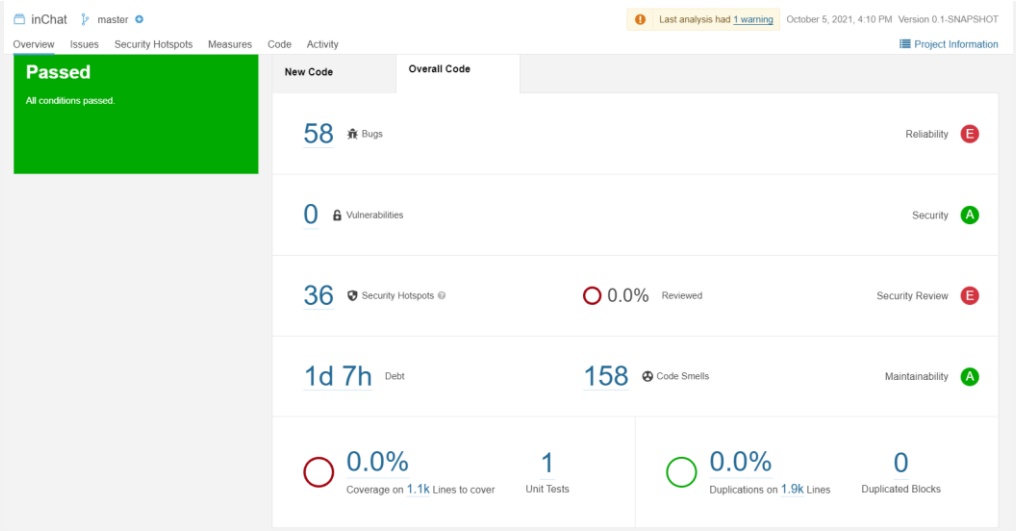
We then try something more fancy: **"><script>alert(document.cookie)</script>**. Now, another alert box pops up, this time containing this: "session=4616fe18-b07f-4575-ab96-27273c9175f8; session=4616fe18-b07f-4575-ab96-27273c9175f8", in other words the session ID for cookies.

By testing with various inputs in the create channel input form, I also found that the program will acknowledge the input as SQL code: **');DROP TABLE Channel;'a');.** Instead of interpreting the program as normal text/string, InChat interpreted it as SQL code. This indicates that there are nothing in the code that prevents user from performing SQL Injection to harm the application.

SonarQube:

When SonarQube is installed and docker-compose is properly set up, we initiate a scan of the source-code. This is the result we get:

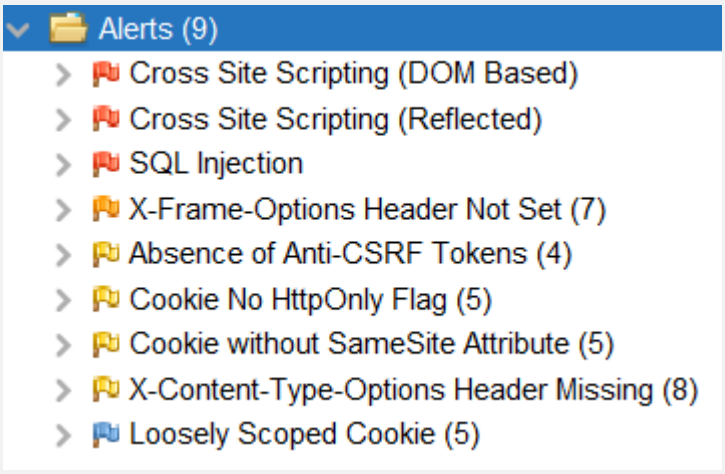
Kasper Melheim



It is detecting 58 bugs and 158 code smells, as well as 36 security hotspots.

OWASP Zed Attack Proxy:

When the OWASP program has been set up we run it on the InChat application, we get this result:



OWASP is detecting multiple security issues with the application, whereas Cross Site Scripting and SQL Injection are the two most fatal issues.

Looking at source code:

After reviewing the source-code we can see multiple potential security faults. There is no code for checking the user input before altering the tables in the database, so pretty much anything can be sent in. There is also no code that checks if code/scripts have been sent in the input field, and no code for stopping the execution of these scripts. There are also no banned characters.

Analysis

After looking at the results of our testing we can see that there are multiple security issues with this application, many who breaks the expectancies we talked about in the threat model. By both using automated tools and by manually using the application we found out that **cross-site-scripting** is possible. I inserted very simple alert scripts into the input field, and the application ran it like normal code. This is a very serious issue with an application and there are many types of attacks that can happen here.

Being able to alter data and run your own code on a website can lead to spreading of worms if used correctly and can affect a lot of users. It can used to create fake forms and input fields in a attempt to extract sensitive data from others. Malicious files can also end up on your computer if you somehow manage to click a button that starts a download on your side. The site could be filled with data and info the creators did not intend to be there, and a lot of users can be tricked into doing something very serious. As shown in the test results an attacker can also be able to get the session ID for the cookie, which connects a session of a user to the application.

Another big problem with the application is **SQL Injection.** Currently there are no backend code that checks the user input before sending it to the database, and the user is communicating directly with the database. Some would say this is an even bigger security fault than cross-site-scripting seeing as, if used correctly, SQL Injection can completely delete and alter huge amounts of data, and tables can be removed. One can also potentially extract sensitive data from the database, data that was never intended to be public to anyone.

SonarQube found 58 bugs in the code, many that were fatal. It is obvious that the application is missing many key points for having a secure program.

OWASP found some other issues with the program as well, issues the program deemed to be less dangerous than the ones stated above. These are:

- X-Frame-Options Header Not Set
    - X-Frame-Options header is not included in the HTTP response to protect against 'ClickJacking' attacks. ClickJacking is a trick when you're making a user accidently click a element disguised as a different element.

- Absence of Anti-CSRF Tokens:
    - No Anti-CSRF tokens were found in a HTML submission form. A cross-site request forgery is an attack that involves forcing a victim to send an HTTP request to a target destination without their knowledge or intent in order to perform an action as the victim.

- Cookie No HttpOnly Flag:
    - A cookie has been set without the HttpOnly flag, which means that the cookie can be accessed by JavaScript. If a malicious script can be run on this page, then the cookie will be accessible and can be transmitted to

another site. If this is a session cookie then session hijacking may be possible.

- Cookie without SameSite Attribute:
  - A cookie has been set without the SameSite attribute, which means that the cookie can be sent as a result of a 'cross-site' request. The SameSite attribute is an effective counter measure to cross-site request forgery, cross-site script inclusion, and timing attacks

- X-Content-Type-Options Header Missing:
  - The Anti-MIME-Sniffing header X-Content-Type-Options was not set to 'nosniff'. This allows older versions of Internet Explorer and Chrome to perform MIME-sniffing on the response body, potentially causing the response body to be interpreted and displayed as a content type other than the declared content type. Current (early 2014) and legacy versions of Firefox will use the declared content type (if one is set), rather than performing MIME-sniffing

The" X-Content-Type-Options Header Missing" problem will be of less relevance than the other problems found, seeing as cross-site-scripting and SQL Injection can be done by sending text/code into a input on the application no matter what browser or system you're using. The "X-Content-Type-Options Header Missing" is only a problem when older version of Explorer and Chrome try to exploit this. That can and will most likely happen but preventing cross-site-scripting and SQL Injection would be of higher priority in this case. With that being said, sniffing is a serious issue and should be dealt with.

Seeing as we have a Cookie without SameSite Attribute and Absence of Anti-CSRF Tokens, the application can be subject to CSRF or Cross Site Request Forgery. This is a attack where an attacker using a web site, email, message or similar of malicious intent, to perform unwanted actions on a site on the behalf of the user, when the user has been authenticated through Cookies. If an attacker can get their hands on a user's cookies, it will be hard for the website to distinguish legit requests from the actual user and forged requests from the attacker. The damages of CSRF can be great, and a lot of sensitive info can be leaked and used for evil.

Conclusion

After performing multiple tests on the application, and having disclosed many security issues, it is safe to say that this program lacks the bare minimum of security for their users. Using automated tools, but also just by browsing and testing the various input fields, we can see that there are multiple ways of altering data and performing actions not originally intended by the creators. Both the website itself and the users using it is in danger of attacks. Cross-site-scripting, SQL Injection and Cross-Site-Request-Forgery are the three

major problems with the application, and the results could be fatal if not dealt with. Offering a chat service such as this need to have those two in check in order to be successful.

Recommended response

This website needs an upgrade in order to be secure and safe for its users. XSS, SQL Injection and CSRF are three very dangerous and common attacking methods, which has also led to a lot of research on the subjects in order to find possible preventions. There are multiple responses one could have to these attacks:

Cross-Site Scripting:

- Encode user data before it is displayed on the website. By doing this you can prevent the program from interpreting the data as active content or code. Scripts wouldn't be able to execute by doing this.
- Filter the user data on input. Prevent users from sending in some symbols or combinations of symbols to prevent executing scripts from being sent to the program. There are many premade filter collections out there already made, and these could be implemented on the site as well.
- Make sure your headers are properly set up. As OWASP told us earlier the X-Content-Type-Options Header is missing on the website. By adding this you can make sure the browser interprets the responses in a way as originally intended by the creators.

SQL Injection:

- Use prepared statements with parameterized queries when entering data into your database. By doing this the creator of the code would define all SQL code and pass in the parameters to the query later. This makes sure the program distinguishes between code and data, and a user sending SQL code wouldn't be able to execute it in any way.
- Stored procedures created in the database. This is very similar to prepared statements and practically does the same thing, the only difference is that instead of writing this in the code that checks the user input, it is already defined within the database itself.
- Add allow-list input validation. This would prevent the user from sending in some words or names, the name of a table in the database for example and doing malicious stuff with it. Adding this will make sure unvalidated input does not end up in the query sent to the database.

Cross-Site Request Forgery:

- Add Cookie with SameSite attribute. This helps the browser decide if it should send cookies alongside cross-site requests. Setting this up to be strict would prevent users session ID to be sent using cross-site requests in some cases, and a user would no longer be authenticated and be able to perform unwanted actions.

- Add CRSF tokens to all HTML code that deals with state changing requests. This would then be validated in the backend and all requests that cause actions on the site will be checked.