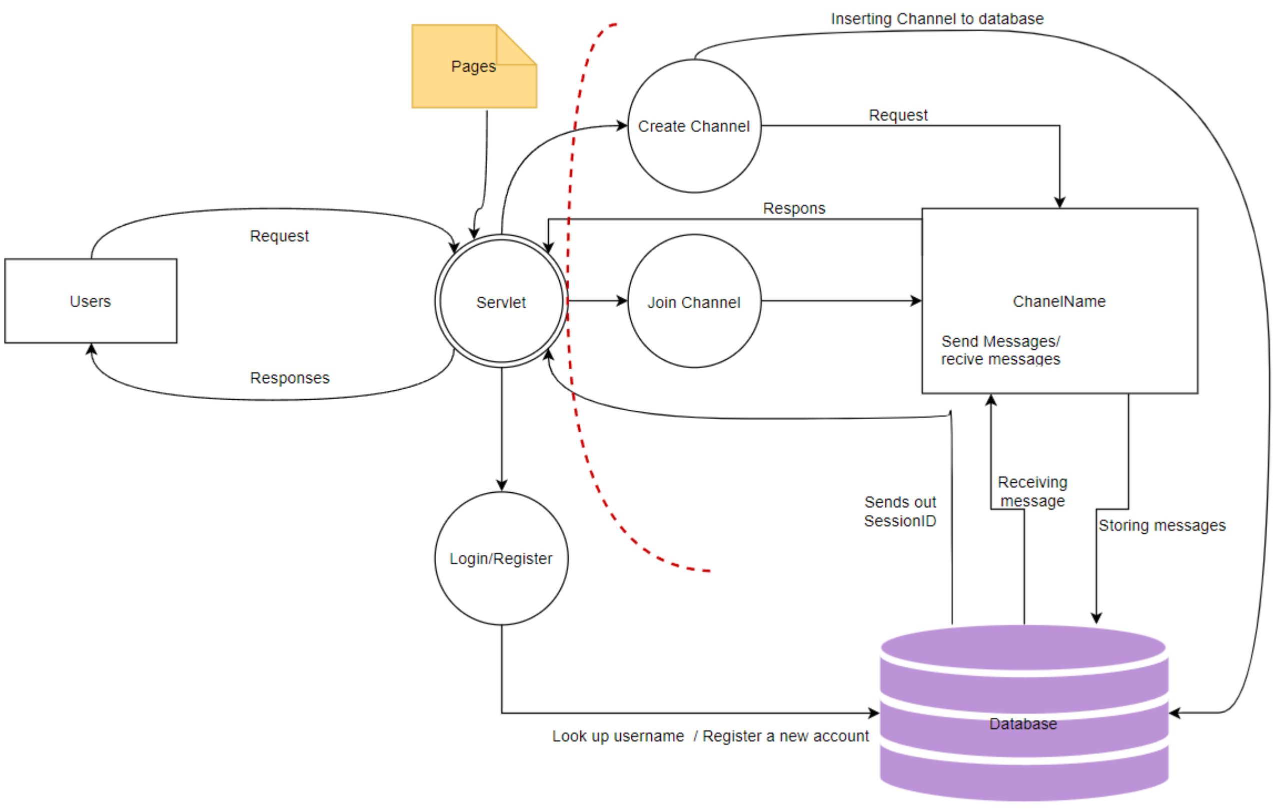
1. Introduction:

Inchat is a web chatting platform that has a classic website structure. When arriving on the website, we have to log in to access more pages, such as to create a channel and one to join a channel. Once a user has access to a channel the website offers a different page for every channel.

A user has access to various functionalities in inchat. The main ones are registering, login, creating a channel, joining a channel, and logout. But the web application provides more features around its main purpose: channels. A channel is a discussion thread. A user who has access can write messages, edit them, and delete them. He can also set permissions for other users.

We will first discuss the threat model and security design of the web application. In a second time, we will present how we tested the web application and show and analyze the results. We will finally conclude and show some points on how to improve the security of the web application.

2. Threat model and security design:



What security guarantees should InChat give to the users?

- Should not leak any sensitive information to other users
- Make share that no malicious code will get executed
- Should hash and salt passwords
- Should make sure that no one can look up their private chat, without getting invited
- No one should be able to change their password

What is the access control model?

The user needs to be logged to accessing channels or creating channels. Each channel got its own unique “Channel ID”, where everyone with the ID could join the private chat. A user who has joined a channel can also invite other people, without permission from the owner.

Who can an attacker be? What can an attacker do? How do they interact with the web application?

The attacker could be anyone who is trying to exploit the user or the web application. An attacker could do a lot of harmful stuff, this includes Cross-site scripting where they insert malicious code into the website which makes it vulnerable, or injection attacks that injects code into the website, so they could get data from the database. The attacker interacts with the web application the same as a normal user.

Describe the security mechanisms in the web application, how are they designed to tender to the requirements?

The server builds a strong UUID for channels, which makes it difficult for an attacker to guess the correct UUID to join a random channel. This UUID is formalized on the server-side, increasing its security. When a user requests a specific channel, it checks whether the user is logged in or not, and from there it authenticates with a SessionId for the rest of the web session. Also, when authenticating, a session variable is created depending on the time the user logs in which is a form of security.

3. Testing methods:

Following the thread model, we first want to find most vulnerabilities through automated tools and then search for more vulnerabilities manually if possible using manual tools and looking at the source code. For example, we want to see which data are accessible using vulnerabilities if we can take control of a user

For the automated tools, we used SonarQube and OWASP ZAP, which provide both insights into critical flaws in the system as well as several tools to test these security flaws in the application. In our case, we first used a “classic spider” which gives a map of the web application. We then use an active scan based on this map to find vulnerabilities. We provided OWASP ZAP information about how to authenticate on the web application and input which technology we want to use to aim at vulnerabilities. SonarQube, the use is mostly information gathering where we will be provided with a list of bugs and security hotspots as well as information on these security flaws. With this information, we can further inspect these errors in the editor and how they affect the rest of the system through unintended interaction.

[Manual inspection using HUD interface]

4. Test results

After executing our tests, we found a lot of vulnerabilities of different types in the web application. These were found through the active scan executed on OWASP ZAP and SonarQube. We can classify these vulnerabilities into 4 different categories.

High-level vulnerabilities:

- Cross site-scripting (DOM-based/reflected/stored): The attacker can execute code (HTML, javascript, …) on a user's browser. The code is just added to the application and executed. The way it's executed, the attacker can read, modify and send any sensitive data that the browser has access to. Thus, a user could be redirected to another website, getting its account hijacked, or having its system compromised. There are different types of Cross site-scripting, but in the case of our web application, we have 2 types: DOM-based and reflected. These types require the user to click on a link that contains code and will send a form. This form can be executed without the user being aware. Once the form is submitted, the attack begins.
- Remote OS command injection: This vulnerability aims at executing operating system commands on a vulnerable application. These commands are executed with the privileges of the application which can lead to high damage directly on the operating system.
- SQL injection: Writing SQL code in a form that is supposed to execute a request to a database. This way, an attacker could execute a query that should never be executed by a user and get access to data, modify data, remove data, … Furthermore, in this web application, SQL injection can be used to bypass authentication on the login page and to modify the database on most of the input forms as they are not sanitized. According to SonarQube, there are 34 of these SQL injection vulnerabilities

Medium level vulnerabilities:

- X-frame-option Header not set: X-frame-option Header is included in the HTTP response and used to avoid click-jacking attacks. Click-jacking attacks are about tricking a user with a link that is not what he expects. In the case of our web application, they are not included and thus do not prevent click-jacking attacks.

Low-level vulnerabilities:

- Cross-Site Request Forgery: a CSRF attack aims at forcing a user to send an HTTP request to a target destination without the user being aware to execute actions as the victim.
- Application error disclosure: One of the pages of the web application contains errors/warnings that may disclose sensitive information. This information could be used for future attacks by an attacker.
- Cookie: The cookies of the web application aren't set correctly. Some of them do not have the "HttpOnly" flag and "SameSite" attributes. In the first case, it means that cookies can be accessed through javascript and transmitted to another website. In the second case, cookies could be a result of a cross-site request which could lead to Cross-site script and cross-site request forgery vulnerability.
- Timestamp Disclosure: The web application gives access to the timestamp which could be used by an attacker to access some information using the timestamp.
- X-Content-Type-Options header: The option "nosniff" hasn't been set to the X-Content-Type-Options. Thus, an old version of a browser could perform MIME-sniffing on the response body.

Informational alerts:

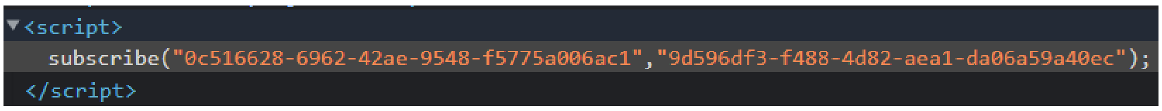
- Charset Mismatch: There is a difference between the charset declared in the HTTP Content-Type header and the one defined in the body. An attacker can exploit this vulnerability by manipulating the content of the page to be interpreted in an encoding of his choice.
- Loosely Scoped Cookie: The cookies of the web application are not scoped correctly which means they are not linked with a strict subdomain. Thus, all subdomains could access every cookie and be exploited.

Looking at test results, there are several significant vulnerabilities at different places of the web application. Some of them are known and could be easily exploitable for an attacker.

5. Analysis of findings

- Cross-site -scripting (DOM)

Cross-site DOM would not work, for the reason that this website is not taking any input from the user, but doing everything on the backend.



```
▼<script>
   subscribe("0c516628-6962-42ae-9548-f5775a006ac1","9d596df3-f488-4d82-aea1-da06a59a40ec");
  </script>
```

All private chats have their GET request waiting for the server to check if someone has replied to our message, such as this.

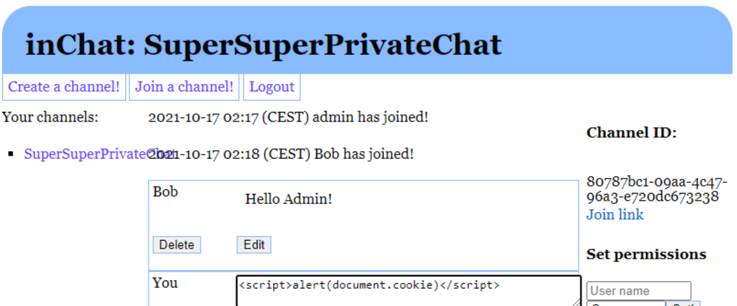
http://localhost:8085/subscribe/0c516628-6962-42ae-9548-f5775a006ac1?version=9d596df3-f488-4d82-aea1-da06a59a40ec

Because the server is expecting the versionID to be a UUID, then writing a malicious javascript code won't work, because the server will just give us error messages.

The ZAP report said that it would be possible to have XSS reflected on the login page and the create channel page. After testing around, it looks like it would be

- Cross site-scripting (Stored/reflected)

Stage 1) Here the user Admin writes a malicious code, which is going to display the "current" user's session cookie. This is vulnerable because the attacker could send this session cookie to his server and log in as the other user who executed this malicious code.



inChat: SuperSuperPrivateChat

Create a channel! | Join a channel! | Logout

Your channels:
- SuperSuperPrivateChat

2021-10-17 02:17 (CEST) admin has joined!
2021-10-17 02:18 (CEST) Bob has joined!

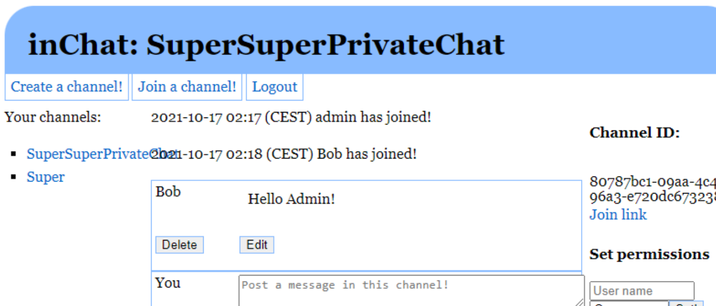
Bob: Hello Admin!
Delete | Edit

You: `<script>alert(document.cookie)</script>`

Channel ID:
80787bc1-09aa-4c47-96a3-e720dc673238
Join link

Set permissions

User name



inChat: SuperSuperPrivateChat

Create a channel! | Join a channel! | Logout

Your channels:
- SuperSuperPrivateChat
- Super

2021-10-17 02:17 (CEST) admin has joined!
2021-10-17 02:18 (CEST) Bob has joined!

Bob: Hello Admin!
Delete | Edit

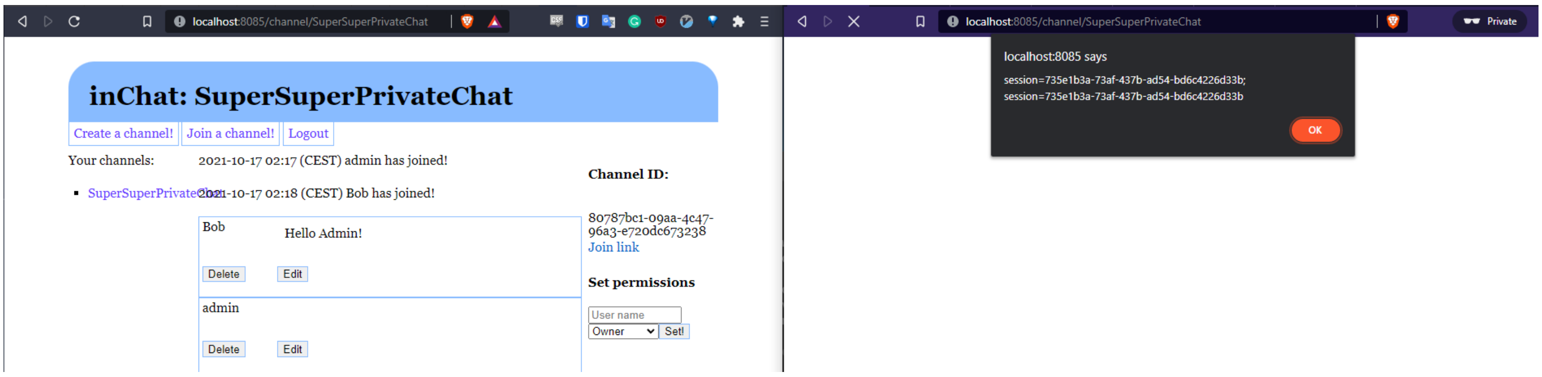
You: Post a message in this channel!

Channel ID:
80787bc1-09aa-4c4 96a3-e720dc67323
Join link

Set permissions

User name

Stage 2) Here you can see on the left that the code was successfully executed on Bob’s computer.



localhost:8085/channel/SuperSuperPrivateChat

inChat: SuperSuperPrivateChat

Create a channel! | Join a channel! | Logout

Your channels:

- SuperSuperPrivateChat

2021-10-17 02:17 (CEST) admin has joined!

2021-10-17 02:18 (CEST) Bob has joined!

Bob — Hello Admin!

admin

Channel ID:

80787bc1-09aa-4c47-96a3-e720dc673238

Join link

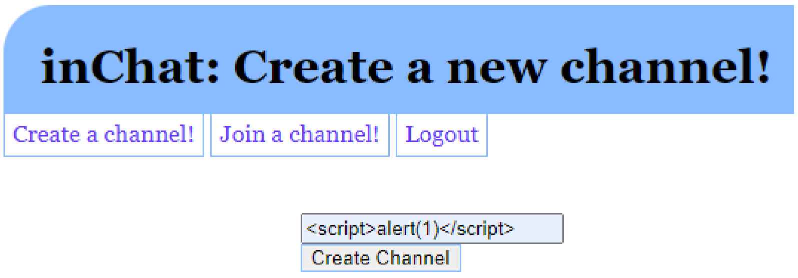
Set permissions

User name / Owner / Set!

localhost:8085/channel/SuperSuperPrivateChat

```
localhost:8085 says
session=735e1b3a-73af-437b-ad54-bd6c4226d33b;
session=735e1b3a-73af-437b-ad54-bd6c4226d33b
```

When creating a channel, you could also do XSS(stored), this is critical because when a user joins the attacker’s channel, then the code would be executed. This is more vulnerable than the first XSS example, because when users have already joined the channel, then the code would always execute when a user logs in. Meaning that they don't even need to go to the channel, and the attacker always gets their sessionID.



inChat: Create a new channel!

Create a channel! | Join a channel! | Logout

`<script>alert(1)</script>`

In the registration field for username, you can also have your username as a malicious code, such as “<script>alert(1)</script>”. Now you can just join random private channels, or invite users to your chat, and then your username with the malicious code would get executed and the attacker would get users sessionID.

XSS reflected won’t work on the username field as what ZAP report told us, because when we write malicious code into the username and try to log in, it automatically redirects us to the homepage, which makes it impossible to send the link to a user.

- Cross-Site Request Forgery

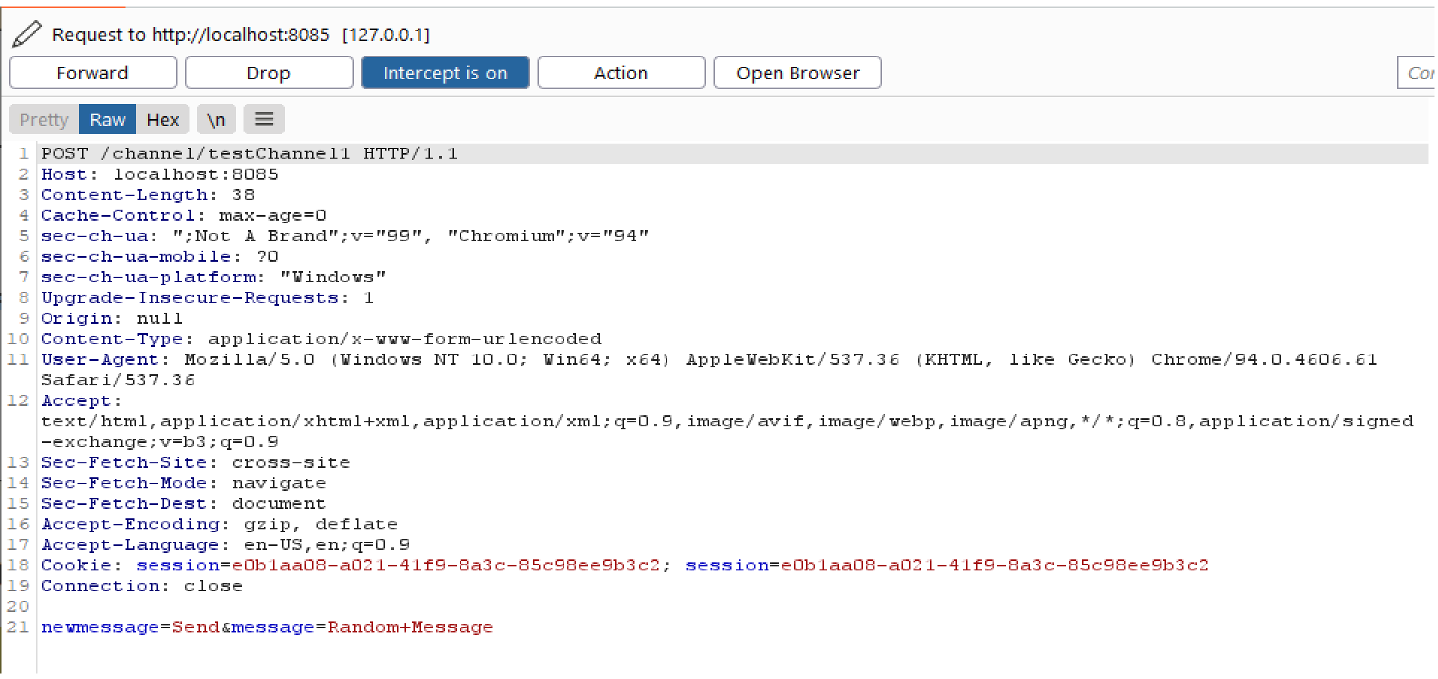
The policy for the cookie is set to “SameSite; None” which means that there are no limitations for where the request with a cookie can come from, meaning that if I would send a request from a third-party website with a “form”, then I would authenticate with the user’s cookie.

By baiting the victim to press the attacker’s link, then the attacker could execute the form from another third-party website, and write a message on behalf of the user.

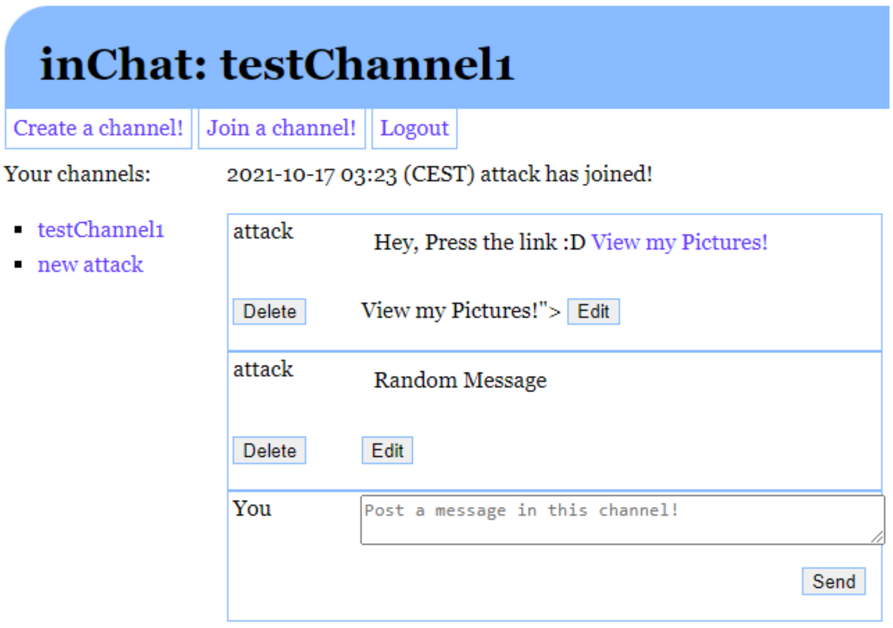
attacker website with the form

```html
<html>
  <body>
<form name ="new" action="http://localhost:8085/channel/testChannel1" method="POST">

<input type="hidden" name="newmessage" value="Send">
<input type="hidden" name="message" value="Random Message">

</form>
<script>document.new.submit();</script>
  </body>
</html>
```

Here we can see that the POST request contains the cookie and a message which is going to get executed.



```
POST /channel/testChannel1 HTTP/1.1
Host: localhost:8085
Content-Length: 38
Cache-Control: max-age=0
sec-ch-ua: ";Not A Brand";v="99", "Chromium";v="94"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
Origin: null
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.61
Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed
-exchange;v=b3;q=0.9
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: navigate
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: session=e0b1aa08-a021-41f9-8a3c-85c98ee9b3c2; session=e0b1aa08-a021-41f9-8a3c-85c98ee9b3c2
Connection: close

newmessage=Send&message=Random+Message
```


inChat: testChannel1

Create a channel! | Join a channel! | Logout

Your channels:

- testChannel1
- new attack

2021-10-17 03:23 (CEST) attack has joined!

attack — Hey, Press the link :D View my Pictures!

Delete — View my Pictures!"> Edit

attack — Random Message

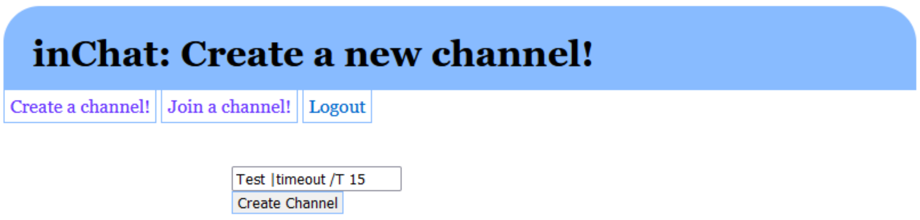
Delete — Edit

You — Post a message in this channel!

Send

- <u>Remote OS command injection:</u>

Remote OS command injection is a false positive as long as we are not executing any OS commands in our code. Looking at ZAP report, the problem occurred when creating a channel name. With this specific string “Test |timeout /T 15”


inChat: Create a new channel!

Create a channel! | Join a channel! | Logout

Test |timeout /T 15

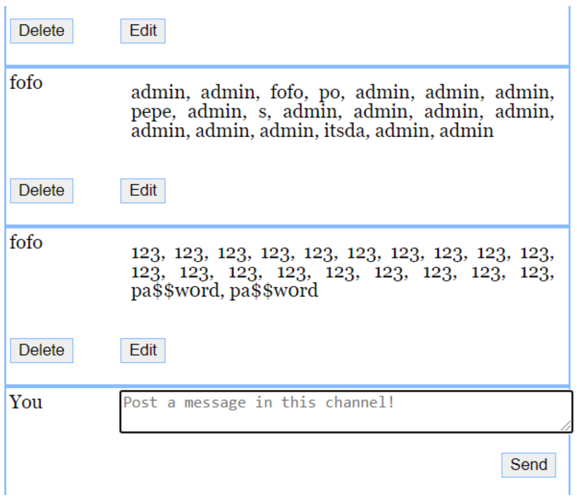
Create Channel

Here we are trying to delay the execution by 15 seconds, and if it successfully waits 15 seconds before creating a channel, then we can confirm that OS injections work for this web application. After I executed the injection myself, it created the web page instantly. This means that when ZAP was doing its injection, the server may have had a bit of overload, and loaded the webpage with a bit of delay, which caused an alarm for ZAP. Meaning that this result is a false positive.

- SQL injection

These SQL injection results are very relevant as they serve a serious risk to integrity, confidentiality, and availability. The possibility of leaking personal data such as passwords, having messages changed without your knowing, or even having your password changed is a serious threat that could be easily performed. The malicious user could do this easily by inputting a single SQL statement to get every single password and username in the database as shown in the picture below using the following SQL code in the chat:

```
' || (SELECT GROUP_CONCAT(name, ", ") FROM User) || '
' || (SELECT GROUP_CONCAT(password, ", ") FROM account) || '
```



fofo
admin, admin, fofo, po, admin, admin, admin, pepe, admin, s, admin, admin, admin, admin, admin, admin, admin, itsda, admin, admin

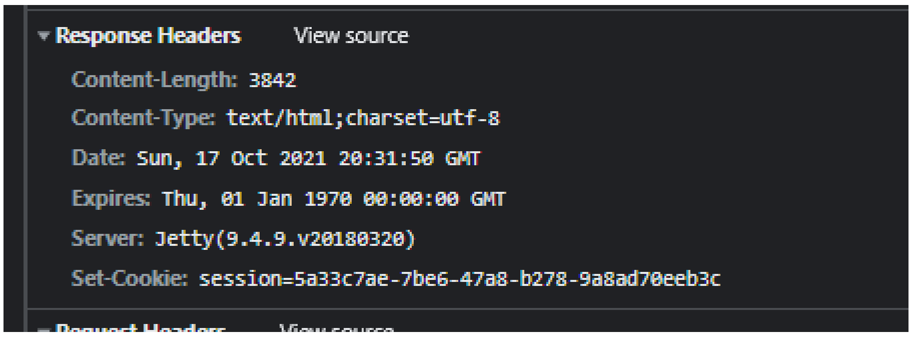
fofo
123, 123, 123, 123, 123, 123, 123, 123, 123, 123, 123, 123, 123, 123, 123, 123, 123, 123, 123, pa$$word, pa$$word

you could even perform an SQL injection without making an account but instead, insert it into the login form itself to manipulate the current data to match your benefit. An example could be the registration web page where you could write this SQL injection into the password field and change all of the passwords

```
'); UPDATE Account SET password='123'--
```

- X-frame-option Header not set :

X-frame option header is not set for InChat, as we can see in this picture.

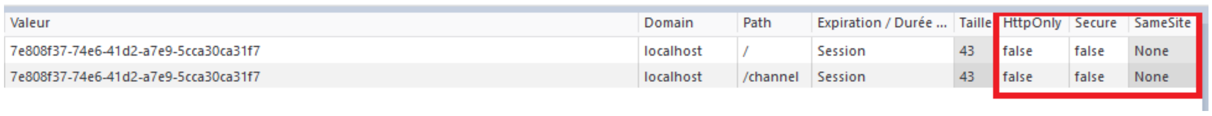


```
Response Headers      View source
    Content-Length: 3842
    Content-Type: text/html;charset=utf-8
    Date: Sun, 17 Oct 2021 20:31:50 GMT
    Expires: Thu, 01 Jan 1970 00:00:00 GMT
    Server: Jetty(9.4.9.v20180320)
    Set-Cookie: session=5a33c7ae-7be6-47a8-b278-9a8ad70eeb3c
Request Headers      View source
```

Meaning that if the attacker manages to get access to an IFRAME element, then they can render their hidden website in front of the IFRAME, and hijack the information of a user. This result would count as a false positive, as there are no iframe elements. This might be vulnerable if the owner of the website implements iframe functions.

- Cookies:

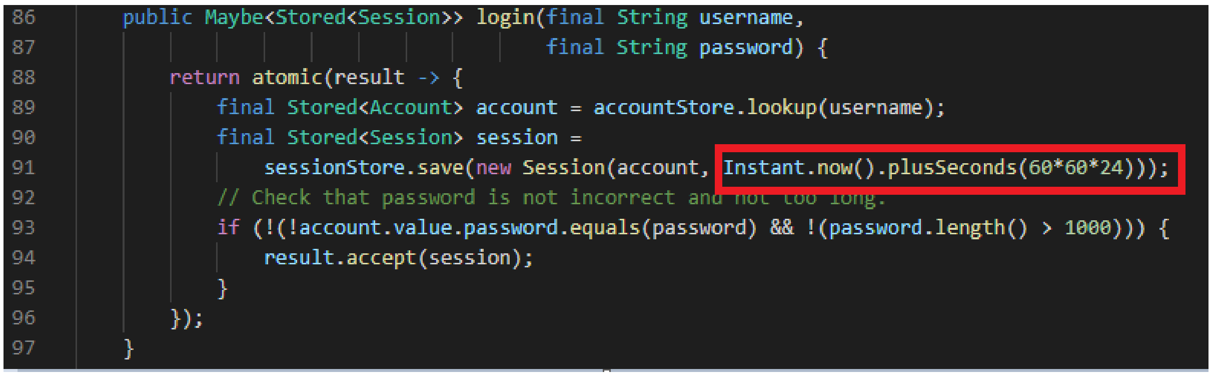
Cookies present a lot of vulnerabilities because they haven't been set correctly. Thus, an attacker could transmit information present in the cookies to another website, and exploit this information through the other vulnerabilities presented. Cookie’s vulnerability mainly comes from the HttpOnly flag and SameSite attribute not being set.



Valeur	Domain	Path	Expiration / Durée ...	Taille	HttpOnly	Secure	SameSite
7e808f37-74e6-41d2-a7e9-5cca30ca31f7	localhost	/	Session	43	false	false	None
7e808f37-74e6-41d2-a7e9-5cca30ca31f7	localhost	/channel	Session	43	false	false	None

- Timestamp disclosure:

Timestamp disclosure vulnerability could be relevant. When a user logs in, the session value is created with a key depending on the current time. If an attacker writes a script and manages to recreate the same key value using the timestamp and the username of a user, he could access its account by recreating the session cookie.



```
    public Maybe<Stored<Session>> login(final String username,
                                        final String password) {
        return atomic(result -> {
            final Stored<Account> account = accountStore.lookup(username);
            final Stored<Session> session =
                sessionStore.save(new Session(account, Instant.now().plusSeconds(60*60*24)));
            // Check that password is not incorrect and not too long.
            if (!(!account.value.password.equals(password) && !(password.length() > 1000))) {
                result.accept(session);
            }
        });
    }
```

Some of these results are actual vulnerabilities. The main security issue is the data of users' accounts. An attacker could get this information, username, and password, especially through SQL injections and Cross-site scripting, and use it for malicious purposes. These vulnerabilities are critical which leads to a high-risk web application.

Looking at the result, many of the requirements could be violated by an attacker. examples could be, InChat doesn't ensure that the data of users is protected and that passwords are hashed, nor does it protect its website for XSS or SQL injections. And using HTTP and not HTTPS makes it possible for a MITM attack. But on the other side, InChat ensures that channels of users are private and not accessible without getting invited because it would be probably too hard for an attacker to guess the Id of a channel

6. Conclusion

We find out several vulnerabilities from different types. These vulnerabilities go from high-level vulnerabilities to informational alerts. They include some known ones like SQL injections, Cross-site scripting, CSRF, and cookies vulnerabilities. A lot of these vulnerabilities are relevant and exploitable by an attacker for malicious purposes. Thus the security of this web application is more than insecure and users cannot have trust in it.

No	Vulnerabilities	Severity	Status
1	Cross-Site Scripting	Critical	Open
2	SQL Injection	Critical	Open
3	X-frame-option Header not set	Low	Open
4	Timestamp disclosure	Low	Open
5	Cookies	Low	Open

7. Recommended response

To prevent Cross-site scripting, use a library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid, for example: using libraries and frameworks that make it easier to generate properly encoded output like Microsoft's Anti-XSS library. We could also add some checks on both client and server-side to prevent any attack even if the attacker changes some stuff on the client-side. It is also important to understand how the data will be useful and which encoding it will use.

to prevent the cookie from getting stolen by an attacker, we could implement a Cookie policy (httpOnly) and a same-origin policy. This means that cookie policy and same-origin would not leak any data to other origins. such as a document.cookie.

To prevent SQL injections, the query in the code should be secured. These can be made through a prepared statement, parameterized queries. A list of allowed and disallowed characters could also be applied to the user’s input to prevent query as input.

Cookies should be set up correctly, with all attributes and flags needed to avoid their malicious use by an attacker. For example, using the HttpOnly flag to prevent an attacker from transmitting the cookies to another website using javascript.

A lot of other vulnerabilities come from some missing settings. We could go through all of them and make sure everything is set up correctly, i.e, header correctly set, tokens needed setup, X-Frame-option HTTP response header set, making sure the HTTP header and HTML meta tags are UTF-8, and having the Content-Type-header be set up so that it sets the X-Content-Type-Options header to “nosniff” for all web pages.

Bugs in the program:

Users can have the same username, when registering for the website you can write your password as one of the inputs. When deleting a message, you can see that there are hidden values inside the messages, which makes it possible that an attacker could delete someone else's message by copy-pasting the other message value and putting it inside theirs, then will result in deleting the other users’ message.