Security Report

inChat web application

University of Bergen
Software Security INF226

Robin Erd
Autumn 2021

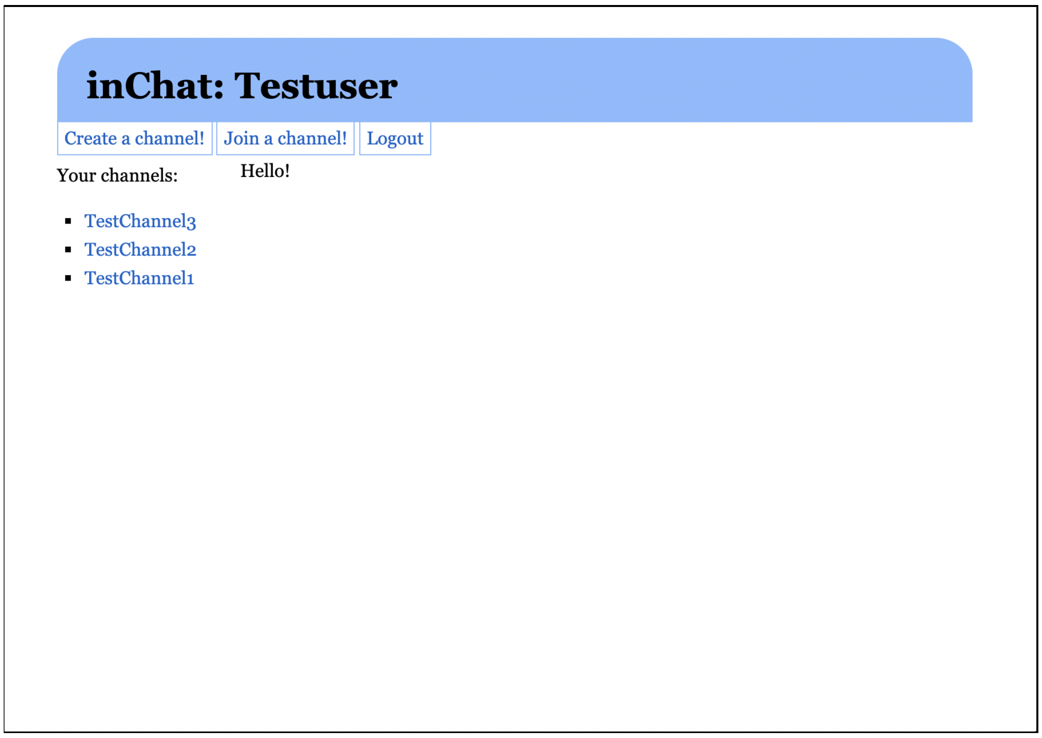
Contents

Introduction

This report is an analysis of the security of the Java-based web application “inChat”. The web application provides users with various functionalities, including but not limited to registering, login, creating channels, joining channels, sending messages in channels, editing messages, and deleting messages. This means that the server must implement session management mechanisms[1], authentication mechanisms, access control mechanisms and a database. These parts have to be taken special care of when developing a secure web server and therefore will also be closely examined in this report. This report will be oriented on the OWASP Web Security Testing Guide[2].

The user interface of inChat is kept very simple. The home page provides the possibility to register or login. After logging in or registering one is redirected to a page which can be considered the dashboard or home (see *Figure 1*). It consists of a header and a list which contains all channels the person is part of. The header provides the user with the possibility to go back to the home page, create a new channel, join another channel by its id or logout. The user can display a channel by clicking on the name of the channel in the channel list. Every channel is structured in the same way, with the header at the top of the page and the previously sent messages listed below, each with a button to edit it or delete it. Below this list there is a form which allows the current user to post a message to the channel. There also is a sidebar next to the messages, displaying the id of the channel, a join link and a (non-functional) interface to change the permissions of the users in the channel.

Figure 1: inChat dashboard



[1] more information: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html
[2] more information: https://owasp.org/www-project-web-security-testing-guide/

The source code of the web application was provided for the purpose of this security analysis and reveals the internal structure of the program, which is composed of the main class “Handler” which handles all incoming requests, an “inChat” class, which models the chat logic (the business logic in this case), and some helper classes (“*Storage”, “Stored”, “*Exception”, util) which facilitate the interaction of the chat logic with the database and classes modeling objects like “Account”, “User”, “Channel” and “Session”.

The following sections cover the threat model[3] of the inChat web application, the methods that will be used to test the security of the web application based on the threat model, the results that these tests yielded as well as an analysis of these results and a conclusion regarding the overall security status of the web application. Finally, some recommendations will be made on how the found vulnerabilities might be closed or mitigated.

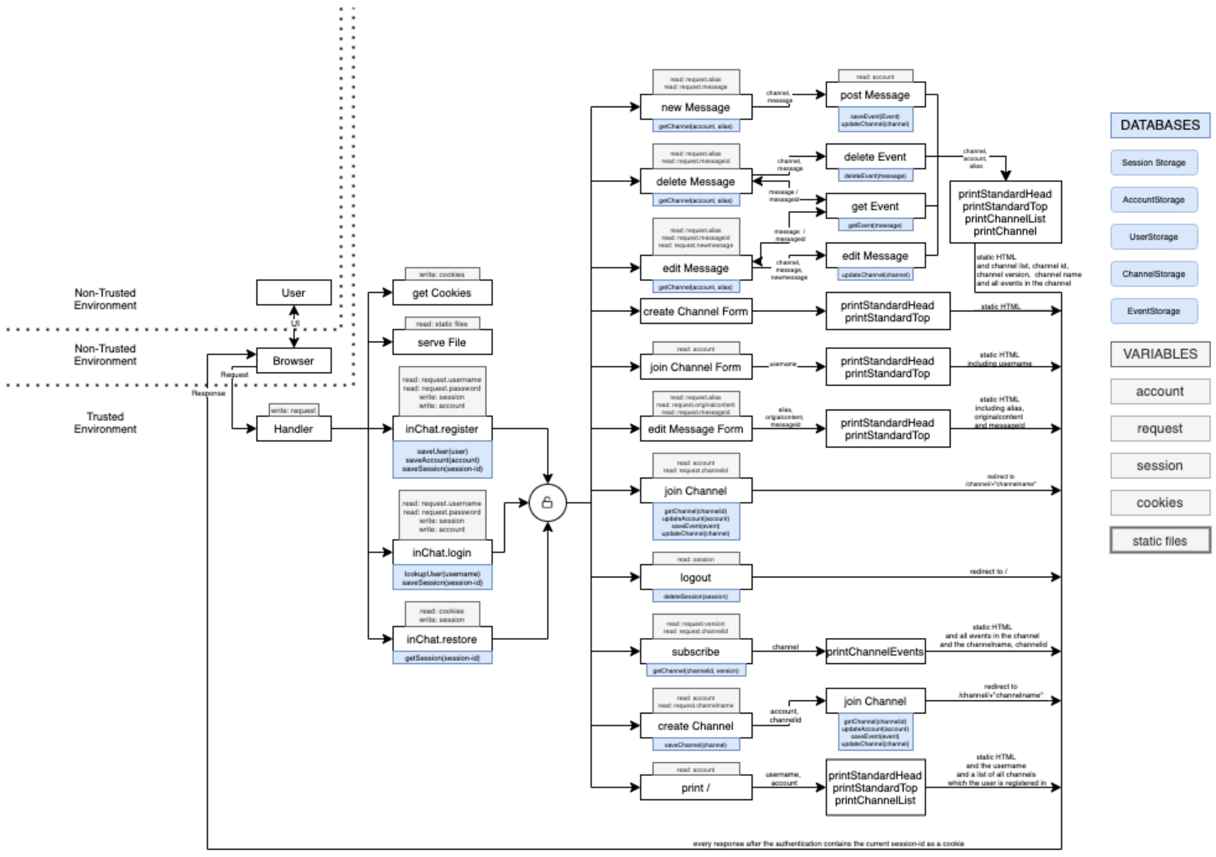
[3] more information: https://owasp.org/www-community/Threat_Modeling

Threat model and security design

Data Flow

During threat modeling, the web application is analyzed with the objective of devising a list of vulnerabilities that have to be addressed during the testing. For this, it is helpful to create a map of the system that captures the dataflows and interactions of different parts of the system. The data flow diagram depicted in *Figure 2* represents the inChat web application. Due to the complexity of the web application the design had to be slightly modified compared to the usual structure of data flow diagrams to improve readability. Parts of the application which are not relevant to the user interaction with the application were excluded.

Figure 2: Data Flow Diagram (attached as pdf)


Non-Trusted Environment
User
UI
Non-Trusted Environment
Browser
Request
Response
Trusted Environment
write: request
Handler
write: cookies
get Cookies
read: static files
serve File
read: request.username read: request.password write: session write: account
inChat.register
saveUser(user) saveAccount(account) saveSession(session-id)
read: request.username read: request.password write: session write: account
inChat.login
lookupUser(username) saveSession(session-id)
read: cookies write: session
inChat.restore
getSession(session-id)
read: request.alias read: request.message
new Message
getChannel(account, alias)
channel, message
read: account
post Message
saveEvent(Event) updateChannel(channel)
read: request.alias read: request.messageid
delete Message
getChannel(account, alias)
channel, message
delete Event
deleteEvent(message)
message / messageid
get Event
getEvent(message)
channel, account, alias
printStandardHead printStandardTop printChannelList printChannel
static HTML and channel list, channel id, channel version, channel name and all events in the channel
read: request.alias read: request.messageid read: request.newmessage
edit Message
getChannel(account, alias)
message / messageid
channel, message, newmessage
edit Message
updateChannel(channel)
create Channel Form
printStandardHead printStandardTop
static HTML
read: account
join Channel Form
username
printStandardHead printStandardTop
static HTML including username
read: request.alias read: request.originalcontent read: request.messageid
edit Message Form
alias, originalcontent, messageid
printStandardHead printStandardTop
static HTML including alias, originalcontent and messageid
read: account read: request.channelid
join Channel
getChannel(channelid) updateAccount(account) saveEvent(event) updateChannel(channel)
redirect to /channel/+"channelname"
read: session
logout
deleteSession(session)
redirect to /
read: request.version read: request.channelid
subscribe
getChannel(channelid, version)
channel
printChannelEvents
static HTML and all events in the channel and the channelname, channelid
read: account read: request.channelname
create Channel
saveChannel(channel)
account, channelid
join Channel
getChannel(channelid) updateAccount(account) saveEvent(event) updateChannel(channel)
redirect to /channel/+"channelname"
read: account
print /
username, account
printStandardHead printStandardTop printChannelList
static HTML and the username and a list of all channels which the user is registered in
every response after the authentication contains the current session-id as a cookie
DATABASES
Session Storage
AccountStorage
UserStorage
ChannelStorage
EventStorage
VARIABLES
account
request
session
cookies
static files

From the data flow diagram one can see that there are various data pathways through the system, related to the different functions supported by the application. All the data paths lead through the "handle" method, which then handles the different requests. Before any business logic is exposed the handle method tries to authenticate the user from which the request originated by trying to read a session-id from a cookie sent with the request. This session-id is generated and sent to the client as a cookie during registration or login. It is included with subsequent HTTP requests for authentication. If the authentication fails, no other action is taken and the user is redirected to the login page. Consequently, all following functionality is protected by this initial authentication mechanism as long as the mechanism itself is not compromised. The data flow diagram also shows that depending on the target path of the request different methods are executed, some of which serve only static HTML pages or HTML pages with very little dynamic content while other routes are mostly used to process data (e.g. HTTP POST requests) and only redirect (dynamically) to another page.

There also are a few methods that serve HTML pages with large chunks of dynamic content such as the pages displaying the list of channels available to the currently logged-in user or the page displaying the channel itself. These methods access sensitive information such as the channels the user is part of and the messages of these channels and require a more refined access control model[4] as e.g. not everybody should be able to edit any messages. This is implemented by the methods only having access to the channels (and their messages) which the authenticated user is part of, as they reference the account variable which is set during authentication and initialized as final (for each request).

In conclusion, the data flow diagram reveals that the only way to input data into the web application is via the HTTP requests sent to the web application, all of which are handled by the “handle” function and the only way to receive information from the web application is through the HTTP responses. It also shows that all sensitive methods which deal with the business logic (e.g. access the database) require the user to be authenticated, that some methods require additional authentication, and which parameters of the requests are passed on to the methods that then perform the sensitive operations (e.g. access the database) and return output to the user.

Assumptions

The data flow diagram provides insight into which assumptions regarding the adversarial environment were made during the development. One assumption that was made (that is made most of the time) is that the users cannot be trusted to be who they claim to be (login system). Otherwise one could imagine a chat application where the users can choose their name freely (except for already taken names) every time they connect to the server. This would only work in a reliable, trusted environment like a classroom if the goal is to be able to identify who sent which message. Apart from this it seems the developers assumed that the environment of the application would be secure and friendly as there is e.g. no input sanitization.

Security Guarantees and Access Control

The objective of inChat should be to provide the three basic security guarantees (the CIA triad): confidentiality, integrity and availability. Examples for this would be that: Messages posted to a channel should not be alterable by someone who is not the owner of the message (integrity), no one should be able to join a channel that he was not supposed to join and see its messages (confidentiality) and no one should be able to prevent others from using inChat to communicate (availability). Another security guarantee which should be provided is authenticity, the ability to have confidence in the message originator and it’s content (sometimes considered part of integrity).

[4] more information: https://owasp.org/www-community/Access_Control

Confidentiality, integrity and authenticity can be achieved by making use of an access control model managing the permissions each user has. This can be used to e.g. check if a user has the permission to delete or edit a message or send a message as someone.
InChat implements neither a role-based nor a capability-based access control model but uses a system similar to an access control list (as discussed in the discussion of the data flow diagram). While the user interface of inChat contains elements referring to roles such as “Observer”, “Participant”, “Moderator” and “Banned” they currently do not have any effect.

Availability can be ensured by providing a secure application and can be improved by additionally relying on services such as load-balancing, dynamic scaling of the service with demand, and meticulous maintenance. This however is not part of the scope of this report.

Attackers and Threats

As the web application is accessible from the internet for everybody there is no restriction on the number of potential attackers or who might be an attacker. An attacker could be an unregistered or registered user, as there are no requirements for registering as a user.

A potential attacker can use a browser to explore the web application and use the input forms to send data to the server. Alternatively, an attacker might use the information he was able to extract from the web application to get data from the web server using custom-crafted HTTP requests.

A popular model used to classify the threats an application is facing is the S.T.R.I.D.E. model[5]. It also provides some orientation regarding which kinds of attacks one should be able to defend against. According to the S.T.R.I.D.E model, there are six types of threats:

- Spoofing - pretending to be someone other than who you really are
    - e.g. setting up a similar website to grab the credentials of users
- Tampering - adding, deleting, or changing data (without permission)
    - e.g. edit a message posted by someone else
- Repudiation - denying some (possibly illicit) action
    - e.g. denying having edited or sent a message
- Information Disclosure - exposing confidential data
    - e.g. having access to other channels without permission
- Denial of Service - make services or data unavailable to legitimate users
    - e.g. preventing others from using the inChat to communicate
- Elevation of Privilege - allowing an action that is prohibited by policy
    - e.g. deleting all channels

There are threats that can be excluded from the considerations such as attackers gaining physical access to the system which is running the server, to the configuration of the web server or the system running the web server. InChat will and also cannot protect the users against forms of social engineering used to obtain the credentials of users.

[5] more information: https://owasp.org/www-pdf-archive/STRIDE_Reference_Sheets.pdf

Security Mechanism

The security mechanisms implemented in the inChat web application are very basic. The login system compares the credentials with a database before providing a session-id connected to the account of the authenticated user. This session-id is then part of every following request and acts as an authentication and identification token. With every operation, it is validated against the server-side session-id storage which stores valid session-ids. This serves the identification of users, a key requirement for the access control model.

The implemented access control model achieves confidentiality, integrity, and authenticity if it works as intended. It does not have an impact on availability. For checking if a user has permission to delete or edit a message, inChat compares the identities of the owner of the message and the user trying to modify the message (integrity, authenticity). InChat also verifies that a user is part of the channel before posting a message to the channel and does not display the channel or channel messages if the requesting user is not part of the channel (confidentiality). For this purpose, each account keeps a list of channels he is part of and therefore has access to. There is no access control system in place for joining channels. Channels, therefore, are only secured by the fact that one needs the channel-id (UUID[6]) of a channel to join it.

[6] universally unique identifier, randomly generated 128 bit number

Testing methods

As discussed during threat modeling the inChat web application is facing a wide variety of potential threats and attackers. Due to this, the web application is tested extensively using dynamic analysis[7], static analysis[8] and manual inspection in this order. This order is convenient because it allows one to center the attention on the remaining security aspects and can save some possibly tedious manual work checking for vulnerabilities which tools would have found in a matter of seconds. Special attention will be paid to the most common vulnerabilities found in web applications according to the OWASP Top 10[9].

For testing, two tools, OWASP ZAP for dynamic analysis and SonarQube for static analysis, will be used. They should cover the most common vulnerabilities and threats a web application is faced with.

SonarQube

SonarQube is a platform for the static analysis and the rating of the quality of source code. It runs locally but is accessed through a web interface. Starting the analysis is very straightforward as it only requires executing a Maven[10] command in the project folder of the application to analyze. SonarQube then checks the code for bugs, bad coding practices, and security issues, all of which are reported on. Information is provided on how long it estimates it will take to fix the bugs and the bad coding practice as well as what kind of bugs, bad coding practices, and security issues it found and an approach on how to fix them, accompanied by reasoning on why the concerned code is a security issue.

OWASP ZAP

OWASP ZAP is a web application security scanner suitable for professional use and is developed by the Open Web Application Security Project[11], providing dynamic analysis features. The first step is to manually explore the website through ZAP as a proxy. While doing so ZAP analyzes the traffic and flags potential vulnerabilities. After going through the registration and login process the corresponding POST request can be found in the logs of ZAP and be flagged as Form-based Auth Login Request. One can then stop manual exploration and start a (regular) spider[12] which automatically explores the website. The use of the Ajax spider[13] is not necessary because there is little Javascript used in the inChat web application. Manually exploring the website first and flagging the login request enables the spider to register, log in, and explore the parts of the website requiring a successful login too. After the spider explored the website and discovered all pages an active scan[14] can be started on these discovered pages. The active scan tries to find vulnerabilities and lists the found known vulnerabilities in the alerts panel. Finally ZAP can be used to generate a report on the website, summarizing all the information obtained from the exploring and scanning.

[7] the process of testing a software while it is running
[8] the (mostly automated) process of analyzing the source code
[9] more information: https://owasp.org/Top10/
[10] a build automation tool for Java projects
[11] more information: https://owasp.org/
[12] more information: https://www.zaproxy.org/docs/desktop/start/features/spider/
[13] more information: https://www.zaproxy.org/docs/desktop/addons/ajax-spider/
[14] more information: https://www.zaproxy.org/docs/desktop/start/features/ascan/

Manual Inspection

Of course, one cannot rely on tools to discover all potential vulnerabilities. It is always recommended to take a close look at the web application and source code manually. Some logical flaws or security issues cannot be found by tools as they are part of an insufficient or bad domain model or other design flaws. Finding them requires an understanding of what the business logic does or is supposed to do. Manual inspection includes checking for the most common vulnerabilities found in web applications: Broken Authentication, Broken Access Control, Security Misconfigurations, and Insufficient Logging and Monitoring. There are many sources for what can be considered a "common vulnerability" a web application might contain. As mentioned earlier this analysis relies on the OWASP Top 10 in this matter.

Another thing to do in manual inspection is to just play around with the web application without looking out for something special. This provides an intuitive feeling for how the website works and can sometimes lead to additional ideas on potential vulnerabilities which can then be added to the checklist (see *Table 1*). Some things like e.g. cookie flags or HTTP headers are not part of this checklist because they are reliably checked by the tools.

Table 1: Manual Inspection Checklist

Check	Example(s)	✔
secure web server configuration	- using HTTPS - preventing fingerprinting - Error pages	
sufficient and safe logging and monitoring	- not storing sensitive information - read-only	
sufficient use of authentication	- measures to prevent information disclosure, tampering, elevation of privilege - valid access control model	
safe password storage, transfer	- safely encrypted - salted	
safe password policy, recovery	- minimum length for passwords - security questions for recovery	
two-factor authentication	- SMS - time-based tokens	
session timeouts, session-id rotation	- session-id lifetime	
prevents brute force attacks / credential stuffing[15]	- delay after failed log in attempts - lock-out mechanisms	
denial of service attack mitigations	- CAPTCHA	
protected database access	- exposed network access point	
no debugging leftovers	- login backdoors - comments hinting on security issues	

[15] more information https://owasp.org/www-community/attacks/Credential_stuffing

Test results

SonarQube

The static analysis with SonarQube found 159 Code Smells, 58 Bugs, and 36 Security Hotspots. SonarQube defines “Code Smells” as maintainability-related issues which complicate the maintenance of the software. They are categorized according to their severity (by SonarQube) and listed in *Table 2*. The 58 Bugs are also categorized according to their severity (by SonarQube) and listed in *Table 2*.

Table 2: SonarQube - Code Smells

Severity	#	Issue
Blocker	1	“Add at least one assertion to this test case.” (1)
Critical	19	“Define a constant instead of duplicating this literal … “ (14) “Rename this constant name to match the regular expression …” (2) “Refactor this method to reduce its Cognitive Complexity from ... to 15 allowed” (2) “Make ‘newObject’ transient or serializable” (1)
Major	36	“Replace this use of System.out or System.err by a logger” (18) “Remove this expression which always evaluates to ‘false’” (1) “Extract this nested try block into a separate method” (4) “Add the @Override annotation above this method signature” (1) “Remove this unused … .” (3) “Either remove or fill this block of code” (3) “Provide the parameterized type for this generic” (2”) “Add a private constructor to hide the implicit public one” (1) “Q is not used in the method” (2) “Make this anonymous inner class a lambda” (1)
Minor	101	“Rename this … to match the regular expression …”, “Remove this unused import … “
Info	2	“Complete the task associated to this TODO comment” (1), “Remove this ‘public’ modifier” (1)

Table 3: SonarQube - Bugs

Severity	#	Issue
Blocker	46	“Use try-with-resources or close this ‘...’ in a “finally” clause.”, (41) “Add an end condition to this loop” (5)
Critical	0	None (0)
Major	6	“Make this method ‘synchronized’ to match the parent class implementation”, (3) “Remove this call to ‘equals()’; comparisons between unrelated types always return false” (1) “Either re-interrupt this method or rethrow the ‘InterruptedException’ caught here.”, (1) “‘w’ is a method parameter and should not be used for synchronization.” (1)
Minor	6	“Cast one of the operands of this multiplication operation to a ‘long’ ” (2) “This class overrides ‘equals()’ and should therefore also override ‘hashCode()’” (4)
Info	0	None (0)

Of the 36 Security Hotspots found 34 are related to SQL Injection[16] (Review priority: High), one is related to Cross-Site Scripting[17] (Review priority: High) and one is related to an Insecure Configuration (Review priority: Low).

The SQL Injection Security Hotspots found all refer to the construction of SQL queries, places where SonarQube recommends to “make sure using dynamically formatted SQL query is safe”. All of the referenced passages in code are using String concatenation (with variables) to build a String which is then executed as a SQL query. They are located in *Storage.java files which are used for interaction with the database and cover SQL Select (Data Retrieval[18]), Delete and Insert as well as Update statements (Data Manipulation). SonarQube also explains what risks there are, how one can determine if those risks apply in this case and how to fix the issue if necessary (see analysis).

As listing all the found passages containing this same threat would go beyond the scope of this report but a selection of them is listed below.

Table 4: SQL Injection Security Hotspot Examples

```
UserStorage.java line 71

String sql =  "DELETE FROM User WHERE id ='" + user.identity + "'";
```

```
AccountStorage.java lines 51-55

String sql = "INSERT INTO Account VALUES('" + stored.identity + "','"
                                           + stored.version  + "','"
                                           + account.user.identity + "','"
                                           + account.password + "')";
```

```
AccountStorage.java line 176

final String sql = "SELECT Account.id from Account INNER JOIN User ON user=User.id where
User.name='" + username + "'";
```

```
AccountStorage.java lines 87-91

String sql = "UPDATE Account SET"
                             + " (version,user) =('"
                                           + updated.version  + "','"
                                           + new_account.user.identity
                                           + "') WHERE id='"+ updated.identity + "'";
```

The Cross-Site Scripting Security Hotspot refers to the cookie which is created to save the session-id. It is used to restore the session and keep the user logged in, once authenticated. The cookie is created without the “HttpOnly” flag and SonarQube recommends to “make sure creating this cookie without the ‘HttpOnly’ flag is safe” and also explains what the involved risk is, how one can determine if those risks apply in this case and how to fix the issue if necessary (see analysis).

[16] more information: https://owasp.org/www-community/attacks/SQL_Injection
[17] more information: https://owasp.org/www-community/attacks/xss/
[18] more information: https://docs.oracle.com/cd/B14117_01/server.101/b10759/statements_1001.htm

The Insecure Configuration Security Hotspot refers to the same line creating the cookie for saving the session-id. The cookie is also created without the “secure” flag. SonarQube recommends to “make sure creating this cookie without the ‘secure’ flag is safe here” and also explains what the involved risk is, how one can determine if those risks apply in this case, and how to fix the issue if necessary (see analysis).

OWASP ZAP

OWASP ZAP found twelve different vulnerabilities, some of which were also found by SonarQube. They are listed (categorized and sorted according to severity by OWASP) in *Table 5*.

The full report generated by ZAP can be found attached and includes references to every single occurrence of the vulnerability with information regarding a potential solution to the issue and details on the attack including a description and other information such as the corresponding CWEs[19].

Table 5: OWASP ZAP Result

Severity	#	Issue
High	4	Cross-Site Scripting (Reflected)
High	1	Path Traversal
High	1	SQL Injection
High	1	SQL Injection - Authentication Bypass
Medium	1	Buffer Overflow
Medium	9	X-Frame-Options Header Not Set
Medium	15	Absence of Anti-CSRF Tokens
Low	11	Cookie No HttpOnly Flag
Low	11	Cookie without SameSite Attribute
Low	11	X-Content-Type-Options Header Missing
Info	3	Charset Mismatch (HTTP Header vs. Meta Content-Type Charset)
Info	21	Loosely Scoped Cookie

Manual Inspection

Testing concludes with the manual inspection of the web application and source code. During manual inspection checking again for vulnerabilities that SonarQube and ZAP already discovered, like SQL Injections, Cross-Site Scripting, or missing Cross-Site Request Forgery tokens is skipped.

[19] more information: https://cwe.mitre.org/

Web server

Trying to access a page that does not exist or providing invalid input redirects users to an Error 404 page which reveals the jetty version the web server is running (allows Web Server Fingerprinting[20]). The web application is only using HTTP and not supporting HTTPS (and therefore not making use of the Strict-Transport-Security header. The cookie containing the session-id is not set with the HttpOnly flag, the secure flag, or the SameSite attribute. The domain attribute is not set either. HTTP Responses do not have the X-Content-Options or the X-Frame-Options header set and the provided Content-Type header declares a different charset from the charset defined by the body of the HTML.

Logging and Monitoring

The web application does not implement any form of persistent logs, but prints certain actions such as received requests and SQL queries without further contextual information to the standard output. It also prints the username and password (plaintext) of every successful registration.

Business Logic

Every action that should only be performed after authentication is protected by the authentication mechanism and the access control, although ZAP and SonarQube already found vulnerabilities in these mechanisms. In addition to that there is a problem with the access control linked to the confidentiality of messages in a channel. Knowing the UUID of a channel (in combination with the possession of any valid session-id) allows users to send subscribe requests for this channel. The response contains all the messages of this channel and provides the user with read-access without the other channel participants being aware. The user experience could also be improved by requiring confirmation by the user before joining a channel by link, adding a hint that a message has been edited or deleted after it was edited or deleted, and providing users with a way of leaving a channel.

Authentication

The session-ids or username-password pairs used in this authentication mechanism are transmitted to the server in plaintext and stored in the database in plaintext. During registration, there is no password policy. The server only checks that the password is not too long (fewer than 1000 characters). There also is no way of recovering a password once forgotten and no possibility of using two-factor authentication. There is no password change or reset functionality implemented and there are no weaker alternative authentication channels such as a mobile app or customer support.

There are no restrictions placed on the number of login tries (and failures) and there are no lock-out or delay mechanisms in place to prevent brute-forcing or credential stuffing. There also is no check in place for verifying that the user is human (CAPTCHA[21]). The login name is the same as the name which is displayed in the forum (weak username policy). Accounts cannot be closed, deactivated, or resumed. There is no “remember my password” or “keep me logged in” functionality implemented.

[20] identifying the version and type of web server that a target is running

[21] a computer program or system intended to distinguish human from machine input, typically as a way of thwarting spam and automated extraction of data from websites (Definition from Oxford Languages)

Session Management

The session-id that is created during registration is not changed after registration (the user stays logged in with the same session-id assigned during registration). The session-id is not changed when the session is restored (session fixation[22]). A new session-id is only provided on login. One session-id is valid for as long as the cookie's lifetime, which is defined as 60*60*24 seconds, which equals 24 hours or one day (if the session is not invalidated before).

Debugging Remnants

The source code also reveals that there is an admin account that has the username "admin" and the password "pa$$w0rd". This account is created on web application initialization and, as neither passwords nor username can be changed once registered, will always be accessible on every instance of this web application. The account is a member of a channel called "debug" and cannot be disabled without changing the source code itself.

The table below summarizes the findings made during the manual inspection.

Table 6: Manual Inspection Summary

Check	Result	Comment
web server configuration	ISSUE	fingerprinting is possible, no HTTPS
sufficient and safe logging, monitoring	ISSUE	No logging of user actions, transient logging of parts of the chat logic and requests
sufficient use of authentication	PASS	UUID is used as access control for channels
safe password storage, transfer	ISSUE	No, passwords are stored and transmitted in plaintext
safe password policy & recovery	PASS	No password policy, no possibility to recover a password.
two-factor authentication	N/A	No use or possibility of use of two-factor authentication
session timeouts, session-id rotation after login	ISSUE	Session timeout is set to 24 hours, which is too long.
preventive measures brute-forcing, credential stuffing	ISSUE	No delay or limited login attempts or other measures.
denial of service attack mitigations	ISSUE	None
database access	PASS	SQLite does not expose a network access point
vulnerable debugging leftovers	ISSUE	hardcoded admin credentials / account and channel

[22] more information: https://owasp.org/www-community/attacks/Session_fixation

Analysis

SonarQube

SonarQube found Code Smells, Bugs, and Security Hotspots. The Code Smells themselves do not pose weaknesses and the bugs do neither. Still, they should be fixed in order to ensure that during further development no new security issues are introduced into the application. The fix of one of the Code Smells which recommends replacing the use of System.err.out with a logger is part of a logging and monitoring issue discussed at a later point. There were three different categories of Security Hotspots found by SonarQube: SQL Injection, Cross-Site Scripting, and Insecure Configuration, which this analysis will go through one by one.

SQL Injection

As mentioned earlier all of the SQL Injection Security Hotspots refer to the use of dynamically formatted SQL queries. Those are difficult to maintain and debug and can lead to untrusted values ending up in the query. In most of the found Hotspots the concatenated values are generated UUIDs or other values previously retrieved from the database itself. Some SQL queries also concatenate variables like the session or user-id of the user who is currently logged in. The use of session-ids that have been verified can generally be considered safe to concatenate because they have been generated by the server. The same can be said about channel-ids, user-ids (also referred to as UUIDs), and message ids (also referred to as UUIDs) originating from within the server.

In these cases, it is safe to use this way of constructing a SQL Injection, although not recommended.

But there also are SQL queries where values obtained from untrusted and or unsanitized user input are used by concatenating them to the query such as channel names, usernames, or passwords. This is insecure and provides the user with the ability to use carefully crafted inputs to perform unintended operations on the database. One example is the query below which is used to create new channels. Creating a channel with the name `foobar' OR '1'='1` results in the created channel being called `1`.

```
String sql =  "INSERT INTO Channel VALUES('" + stored.identity + "','"
                                             + stored.version  + "','"
                                             + channel.name  + "')";
```

While in this case there is no possibility for the user to misuse the statement to disclose information it might be possible to find other queries where it is possible to update all values in the table by changing the condition to a tautology and thereby severely impacting availability and integrity of the web application. Such queries where an untrusted variable is included are used in many places, e.g. during registration or channel creation and joining:

```
String sql =  "INSERT INTO User VALUES('" + stored.identity + "','"
                                          + stored.version  + "','"
                                          + user.name  + "','"
                                          + user.joined.toString() + "')";
```

```
String sql =  "INSERT INTO Account VALUES('" + stored.identity + "','"
                                             + stored.version  + "','"
                                             + account.user.identity + "','"
                                             + account.password + "')";
```

There also are instances where other untrusted values (message ids, channel-ids) acquired from HTTP requests are used in SQL Select queries.

```
final String asql = "SELECT sender FROM Joined WHERE id = '" + id.toString()
```

Extensive testing revealed no possibility for the attacker to somehow make use of the returned values and get them to be output by an HTTP Response. In addition to that, fortunately, the default configuration of JDBC prevents the simple attachment of another query by making use of the `;` and `--` operators. Such statements would require `allowMultiQueries=true` or the use of the addBatch and executeBatch methods.

In conclusion, there are instances where SQL Injection can be performed in such a way that it alters the behavior of the web application in an unintended way and the possibility that there are complex constructs capable of making use of possibly a combination of several such injection vulnerabilities to disclose information or modify tables resulting in a possible loss of confidentiality, integrity, authenticity, and availability cannot be excluded. Therefore this must be considered a high-impact vulnerability.

Cross-Site Scripting (XSS)

With regard to Cross-Site Scripting SonarQube only found one hotspot. It refers to the cookie which saves the session-id and indicates that this cookie is not making use of the “HttpOnly” flag. This flag can be used to enable a browser feature that guarantees that no client-side Javascript can read the value of the cookie (the session-id) and therefore helps prevent the theft of session cookies through Cross-Site-Scripting vulnerabilities (whose presence will be discussed at a later point). It has to be noted that this flag by itself is not enough to prevent the misuse of the session-id cookie because even if the HttpOnly flag was set, the cookie could still be sent as part of an HTTP request made by Javascript from a malicious web page opened in the same browser or malicious Javascript contained in inChat itself.

Insecure Configuration

The same cookie is also referred to in the Insecure Configuration Hotspot, which is about the cookie not making use of the “secure” flag. This flag can be used to prevent the browser from sending the cookie over an unencrypted HTTP connection. The secure flag does not prevent all access to sensitive information in cookies as someone with access to the clients’ storage can still read and modify the stored information. This could for example be done through the use of Javascript. It also does not prevent other websites from sending this cookie as part of their requests (see Cross-Site-Request Forgery).

Note:
As the inChat web server does not support HTTPS, enabling this feature without enabling HTTPS would render the application unusable because Users would never receive their session-ids contained in the cookie because the cookie could not be transferred over the unencrypted connection.

OWASP ZAP

SQL Injection

Similar to SonarQube, ZAP found SQL Injection vulnerabilities. While SonarQube found 34 potentially vulnerable SQL queries, ZAP only detected two instances of possible SQL Injection which both refer to the same target URL and parameters and therefore form.

The found vulnerability “SQL Injection - Authentication Bypass” states that it might be possible to use SQL Injection on a login page which could potentially allow the authentication mechanism to be bypassed. The corresponding report of ZAP references the root URL (/) with the parameter channelname which indicates that this is a misclassified vulnerability not specifically related to the authentication mechanism. The misclassification probably occurred due to the same route being used for authentication-related requests (flagged during testing as Form-based Auth Login Request). The related SQL query is :

```
String sql =  "INSERT INTO Channel VALUES('" + stored.identity + "','"
                                             + stored.version  + "','"
                                             + channel.name  + "')";
```

General SQL Injections which of course might also impact the authentication mechanism were discussed extensively during the analysis of the SQL Injection vulnerabilities found by SonarQube and therefore here are not.

Cross-Site Scripting (XSS)

The report of ZAP also states that there is a Cross-Site Scripting (Reflected) vulnerability in the inChat web application which allows attackers to insert Javascript code into another user's browser. This vulnerability was found in four places and none of the target URLs in the report actually refer to a situation in which there is a possibility to reflect a script but rather situations where a custom string (potential script) is stored on the server via a POST request and the user is immediately redirected to the page containing the stored string (potential script). Therefore it probably was misclassified as Reflected Cross-Site Scripting while actually being a Stored Cross-Site Scripting vulnerability (worse). One instance of this issue is the form for sending messages and the associated server-side logic. It allows the user to enter any message and post it to the channel, including messages which contain Javascript. Because the input is not sanitized the message containing the code is stored on the server and served to every unsuspecting user visiting the channel. When a web browser is served Javascript it is executed immediately. This is a major security issue as it allows remote code execution and invalidates the security guarantees confidentiality, integrity, availability, and authenticity. In combination with the vulnerable storage of the session-id in a cookie without the HttpOnly flag set an attacker could easily hijack the session of another user by reading the session-id from the cookie and sending it over the network (violating confidentiality, integrity, authenticity). Another potential attack vector could be spoofing the login screen trick a user to enter his password again which could then be sent to the attacker. Availability could be targeted by writing a script that (client-side) deletes the content of the whole webpage (Document Object Model) and posting it to every known channel.

The other instances of this issue involve the username provided during registration, the channel name provided during channel creation, and the form allowing users to edit their messages.

Cross-Site-Request Forgery (CSRF)

Another vulnerability found is the “Absence of Anti-CSRF Tokens”. This refers to the HTML submission forms present in the inChat web application not containing any form of Anti-CSRF Token. A cross-site request forgery attack involves forcing a user to unknowingly send an HTTP request to a target. This is possible due to the submission forms being the same for every visitor and allows the attacker to exploit the authentication status a user has with a website and can be used to perform actions against the target with the user’s privileges. An example for this is a user of inChat surfing on another web page which contains a malicious script that sends a form submission (including the session-id cookie) to the inChat server acting as if it was the user. This could e.g. be deleting, posting, or editing a message.

All the forms (POST requests) of the inChat web application are vulnerable to this type of attack as they do not contain a CSRF Token. CSRF Tokens are random, secret, and unique values that are generated by the server and sent to the client which then sends them back when submitting a form so that the server can validate the token. In combination with the Cross-Site Scripting (XSS) vulnerability, this vulnerability is easily exploitable due to XSS enabling the circumvention of the same-origin policy.

Cookies

In addition to the missing secure and HttpOnly flag found by SonarQube regarding the session-id cookie, ZAP finds one more vulnerability - the cookie is lacking the SameSite attribute. SameSite specifies under which circumstances cookies are sent with cross-site requests and has three possible values: Strict, Lax, and None. The default value is None[23] and specifies that cookies are sent on all requests. This is another issue related to the theft of the session-id because it allows other currently opened web pages in the browser to make requests containing the session-id cookie of the unsuspecting user.

Furthermore, ZAP finds that the cookie is a “Loosely Scoped Cookie” which refers to the cookie not being restricted to a domain or path. How much of an impact this vulnerability has depends on the environment the web application is run in. If the domain used by the web application has sub-domains or a parent domain the lack of scoping would allow all the other domains access to the cookie and to transmit it. The defined domain in this attribute controls what the SameSite attribute considers the same site.

Path Traversal

This vulnerability was most probably falsely classified by ZAP. By looking at the target URL provided in the report one can see that the request is calling the subscribe method and providing identical channel-identity and version parameters. The subscribe method simply checks whether the provided version is the same as the most recent version. If that is the

[23] is currently being changed to “Lax” in some browsers but cannot be relied on

case it loops until there is a newer version and returns it. If the provided version is not equal to the most recent version it immediately returns the newest version. In this case, the channel-id is passed as the version and therefore is not equal to the most recent version. This leads to the newer version being sent as a response.

During further investigation, it turned out that this is not a path traversal vulnerability but rather a trick that allows the content of a channel to be requested (and received) without having to be a member of the channel. The only requirement for the subscribe path to work is that the request contains a valid session-id and that the channel-id which one wants to spy on is known. The subscribe method then calls the printChannelEvents function, which sends a response containing all the events from this channel. This is not a confidentiality issue as the channel-id also allows a user to join the channel-identified by the channel-id but this trick allows reading a channel without others knowing and can therefore be considered a bug or a minor vulnerability (especially due to guessing the channel-id of a channel being near impossible).

HTTP Response Header

The vulnerability "X-Frame-Options Header Not Set" found by ZAP refers to just this header not being included in the HTTP responses originating from the web application. This header protects web applications against "ClickJacking" attacks. Clickjacking describes an attack in which users are tricked into clicking on actionable content on an invisible website (which in this case would be inChat) by clicking on other content on a so-called decoy web page whose UI is overlaid on the target web page (e.g. inChat) and buttons are (involuntarily) clicked. An attacker could, with significant expenditure, trick a user into sending messages he did not want to send or involuntarily perform other actions such as sharing the channel invite link in another channel. Due to the relatively insensitive actions which are performed on inChat and the effort required to exploit this vulnerability it may be considered of low impact.

The finding "X-Content-Type-Options Header Missing" indicates that the referred header was not set. The X-Content-Type-Options header is used to prevent MIME-Sniffing by older versions of Internet Explorer and Chrome. This could lead to the response body being displayed as a content type other than the declared content type and may allow XSS attacks. This is not an exceedingly relevant discovery either because inChat does not provide users with the ability to upload any content which in turn prevents the inChat web application from misinterpreting user-provided files and inadvertently executing Javascript.

Another issue identified is the Content-Type-Options header ("Charset Mismatch"). The HTTP header used by inChat declares that the iso-8859-1 encoding was used while the HTML META tag content-type declares that utf-8 is used. This inconsistency can potentially be made use of by an attacker to inject script into the web page due to the sanitization failing to detect the Javascript encoded in a different way than expected. The exploitability of this vulnerability also is rather limited and an exploit would require extensive effort, but naturally it is better to be safe than sorry, especially if it is just one header to set.

Buffer Overflow

This issue seems to be an issue itself. The information provided by ZAP did not allow replaying the attack. As the target URL and parameters are similar to the “Path Traversal” vulnerability found by ZAP one might assume that this is another classification error.

Although ZAP reports that this request caused the connection to close and resulted in a 500 Internal Server Error it is highly unlikely that the Java-based inChat web application contains a buffer overflow vulnerability. This issue is probably rather related to performance issues of the application after having endured ~ 4000 requests by ZAP. Another possibility is that the Internal Server Error was caused by a bug in the implementation of any of the dependencies of inChat or ZAP (all the applications have been run on an ARM-based machine where unfixed bugs or issues with the Rosetta[24] emulation currently are frequently encountered). Therefore this finding will not be considered a vulnerability of inChat.

Manual Inspection

Web server

Manual inspection shows that the web application has several additional security issues. The web server is configured in a way that reveals its version and variety. Web Server Fingerprinting allows the attacker to check for known vulnerabilities of this version and make use of them. Another major security issue, possibly one of the most severe, is that the web application is using HTTP. HTTP allows attackers to read all network traffic to and from the server and also prevents the meaningful use of features such as the cookie “secure” flag which would help prevent session-id theft and creates an opportunity for a man-in-the-middle attack by e.g. being redirected to a malicious site.

Logging and Monitoring

There also are issues regarding the logging and monitoring of the application. Nothing is logged to persistent storage. This is bad practice and can significantly complicate the forensic work after an attack and also doesn’t provide the users with repudiation protection. Other users can delete messages and none can prove that they were ever sent. The precise requirements regarding logging depend on the way the application is intended to be used (e.g. services with the objective of providing an anonymous way of communication should keep logging minimal). Without proper logging there also is no way for the owner of the web application to monitor it’s behaviour and performance and detect potential attacks or irregularities and outages. This is rather a lack of preventive measures than a vulnerability. The logging of the credentials of users registering in plaintext to the standard output on the other hand is a serious security issue and should be stopped. Even the owner of the web application should not have access to the account of every person and should not have insight into their passwords. The plaintext passwords will be discussed further in the authentication section.

Business Logic

The business logic contains a security issue concerning channels connected to the design flaw that channels only require their id to be joined. As mentioned in the results the issue

[24] more information: https://en.wikipedia.org/wiki/Rosetta_(software)

allows anyone who knows the id of a channel to spy on the messages posted in the channel without joining and thereby revealing himself. This can be interpreted as violating the confidentiality security guarantee of the application, although anyone in possession of the channel-id could also just create a throw-aways account to join the channel and then read the messages.

Authentication

The most pressing problem is the authentication mechanism. The valid business logic and appropriate use of authentication and sessions is useless if the authentication mechanism itself is insecure. There are several issues related to the authentication mechanism, starting with passwords being transmitted to the server in plaintext (along with the username) and then being stored in the database in plaintext. This is irresponsible, especially as the server is only using HTTP. A simple tool to monitor the network traffic allows anyone who is able to capture the packets containing the login or register request to hijack the corresponding account.

In addition to that, there are no measures in place to prevent brute-forcing or credential stuffing. This allows attackers to try as many username and password combinations as they want. Making this worse is the lack of a password policy. Users can choose any password they desire, even just one letter or an empty string, completely omitting the password. In combination with the lack of verification that the visitor is human (no CAPTCHA) this renders the authentication useless as, with enough time, any password can be brute-forced. The username of the target would not have to be found out as the usernames visible in channels is the same as the username for logging in, which is another weakness. The missing CAPTCHA also increases the risk of denial of service attacks succeeding and absent support for two-factor authentication prevents users from taking measures to improve the security of their account themselves.

Session Management

The session which is generated during registration or login also has some issues that could be improved. The session-id should be rotated after successful registration and the session should also be rotated after restoring a session. Although this is not linked to a direct vulnerability this is a best practice to prevent situations in which users keep using the same session-id for extended periods. The cookies containing the session-ids have a too long lifetime (24 hours) and the expired session-ids are not deleted from the database after their client-side lifetime expired if the user does not manually log out. This is insecure as eventually (depending on the number of users), after a lot of expired sessions the database contains a huge number of valid session-ids associated with access to accounts. Those session-ids might be brute-forced by an attacker as there is no restricting policy in place for attempts at restoring sessions (authentication mechanism fault). Sessions being valid for 24 hours also pose a vulnerability because often computers are shared between people and users regularly exit the browser without logging out. This grants any other user that opens the inChat webpage in the same browser within 24 hours full access to the account.

Debugging Remnants

Finally, testing also revealed that there is an account with the username “admin” and the password “pa$$w0rd” initialized during the startup of the application and that this account is a member of a channel called “debug”. This turns out not to be a vulnerability as this account has no special privileges. One thing which might happen is that a person or a group of people could discover this channel and consequently act towards other users as if they were the administrator (an authority). This could lead to phishing attacks which in turn could easily be prevented by removing this account.

Table 7 lists all the found vulnerabilities during testing and assigns them a damage and exploitability score (higher is easier to exploit) as well as the security guarantees which are affected.

Table 7: Findings and impact

Issue	Damage (1 to 10)	Affected Security Guarantee[25]	Exploitability (1 to 10)
SQL Injection	8	CIAG	7
missing cookie flag: HttpOnly	7	CIG	10
missing cookie flag: secure	7	CIG	10
Cross-Site Scripting (Stored)	10	CIAG	10
Path Traversal	0	-	0
Buffer Overflow	0	-	0
X-Frame-Options in HTTP responses header not set	4	CIG	3
X-Content-Type-Options header in HTTP responses not set	0	-	0
Strict-Transport-Security header in HTTP responses not set	7	CIAG	7
Absence of Anti-CSRF Tokens	7	CIG	8
missing cookie attribute: SameSite	8	CIG	8
missing cookie attribute: domain	4	CIG	6
charset mismatch of HTTP response header and HTML meta tag	4	CIG	4
Web Server Fingerprinting	3	C	9
HTTP instead of HTTPS	9	CIAG	9
no persistent logging	2	AG	3
no monitoring	3	A	3
transient logging of user credentials during registration	9	C	8
transfer of username-password pairs in plaintext over HTTP	9	C	9
storing of the passwords in plaintext in the database	9	C	9
UUID of channel disclosing the contents (subscribe-route)	2	(C)	8
no password policy	7	CIG	10
no password recovery	1	A	0
no two-factor authentication	5	CIG	4
no lockout or delay mechanisms (invalid credentials)	8	CIG	10
public username is the same as login name	4	CIG	10
no CAPTCHA	8	A	10
session-id is not changed after registration, login, or restore	6	CIG	7
session cookie lifetime is 24 hours	7	CIG	5
admin credentials are hardcoded	1	G	5
no hint for edited or deleted messages	1	G	3
no possibility of leaving channels	1	-	3

[25] C - Confidentiality, I - Integrity, A - Availability, G - Authenticity (Genuineness)

Conclusion

The security status of the web application inChat is critical. The security guarantees of confidentiality, integrity, availability, and authenticity are endangered and there are numerous components that have to be changed to improve the situation. Not only are there single vulnerabilities that provide attackers with the challenge of engineering a complex working attack but there are very easily exploitable vulnerabilities that occur in extremely convenient ensembles.

The authentication mechanism of the web application is fundamentally flawed as the password is transmitted and saved to a database in plaintext, there are design flaws and missing features or policies impacting application security, there is no meaningful logging or monitoring taking place and where there is logging it violates the security goals. The web application fails to secure the traffic it receives and sends by using HTTP over HTTPS and does not make use of the opportunities provided by HTTP(S) to improve the security of requests and cookies and also doesn't make use of Anti-CSRF tokens. Together with the severe Cross-Site-Scripting vulnerability in the main functionality of the web application, the messaging, the use of this platform is highly insecure and could even be considered dangerous.

Recommended response

The further operation of the web application in this state cannot be accepted responsibility for and it should therefore be taken offline until there are security fixes in place for all the aforementioned issues.

The developers should switch to HTTPS and reroute all HTTP requests to HTTPS with the Strict-Transport-Security header set. This header prevents the connection from being downgraded to HTTP and also prevents the client from accepting bad certificates in the future. They should also make use of the “secure” and “httpOnly" flags and the “domain” and “SameSite" attributes of the cookies as well as the “X-Content-Type-Options” and “X-Frame-Options" header of HTTP Responses. The charset mismatch between the HTML meta tag and the HTTP header should also be resolved.

They should set up a password policy following NIST-800-63B[26] and start using safe cryptographic tools such as bcrypt to encrypt the passwords before transmitting them to the server. In addition to that, they should add a delay or lock-out mechanism to the authentication procedure which is triggered when too many failed attempts from the same origin were made. Login and registration forms should be extended by a CAPTCHA.

Cookie (session-id) lifetime should be set to 10 minutes and the session-id itself should be renewed after each restoration of a session and selected other actions on the page such as message posting. Users should not be automatically logged in after completing registration. The server should also keep track of session-id expiration. This could be implemented as a

[26] more information: https://pages.nist.gov/800-63-3/sp800-63b.html

timed procedure that deletes all stored session-ids which were created more than 10 minutes ago (and not renewed).

All forms should include an Anti-CSRF-Token and every user input which makes it to the web application should be sanitized (especially from SQL, HTML, Javascript) to prevent Cross-Site-Scripting, SQL Injection, or other injection-based attacks. This should not be implemented manually, but a well-tested package such as the OWASP Java HTML Sanitizer[27] should be used. The issues regarding the confidentiality of channel content should be approached by either declaring all channels public or password-protecting channels and adding the authentication mechanism to the subscribe method. Every instance of SQL formatting in the application by string concatenation should be replaced by making use of prepared statements in an effective manner.

Finally, aspects like the current number of users, messages, and channels should be monitored to detect suspicious activities. Server activity should be logged to a read-only database. With the necessary disclaimers, this might also include information about channels being created, users logging in, registering or logging out and messages being sent, edited, or deleted.

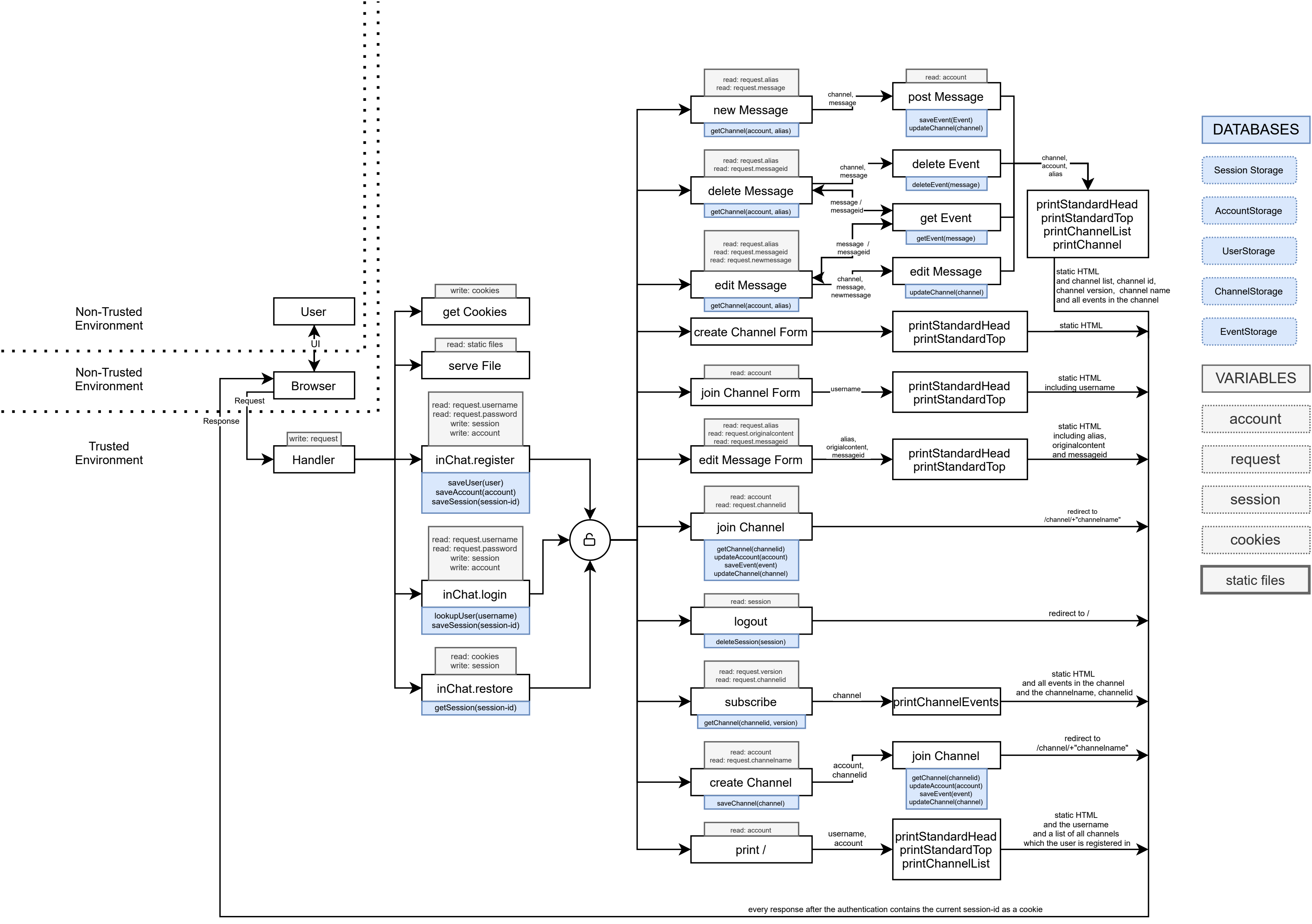
These measures are collected in the form of a checklist in *Table 8*. After implementation of these fixes, the web application should go through another security analysis and can then, if the requirements are met, be cleared for safe deployment.

[27] more information: https://owasp.org/www-project-java-html-sanitizer/

Table 8: Response checklist

Fix	✔
prevent web server fingerprinting by changing the configuration	
enable HTTPS, reroute HTTP to HTTPS	
set “Strict-Transport-Security” header for HTTP Responses	
set cookie flags “secure”, “HttpOnly”	
set “domain” attribute for cookies to the applying (sub)domain	
set “SameSite” attribute for cookies to “Lax” [28]	
set “X-Frame-Options” header for HTTP Responses to “DENY”	
set “X-Content-Type-Options” header for HTTP Responses to “nosniff”	
fix charset mismatch in HTTP Response header [recommended: UTF-8 everywhere]	
setup password policy in accordance with NIST-800-63B	
add cryptographic mechanisms [recommended: bcrypt]	
add delay or lock-out mechanism to login	
add CAPTCHA to login and registration	
set cookie lifetime time to 10 minutes	
rotate session-id on restoration and other selected actions	
change user being logged in after registration to a login-redirect	
add regular removal of expired session-ids from the database	
use Anti-CSRF tokens in forms	
sanitize all user input	
require a password for joining a channel or set all channels to public	
fix UUID disclosing channel content (subscribe-route)	
add required confirmation before joining channels by link	
add the ability to leave channels for users	
add “edited”/”deleted” hint for edited/deleted messages	
use preparedStatements for SQL formatting	
add persistent logging	
add monitoring tools for administrators	

[28] Lax is sufficient if the web application consistently uses POST requests to submit forms - this is the case



Non-Trusted Environment
User
UI
Non-Trusted Environment
Browser
Request
Response
Trusted Environment
write: request
Handler
write: cookies
get Cookies
read: static files
serve File
read: request.username read: request.password write: session write: account
inChat.register
saveUser(user) saveAccount(account) saveSession(session-id)
read: request.username read: request.password write: session write: account
inChat.login
lookupUser(username) saveSession(session-id)
read: cookies write: session
inChat.restore
getSession(session-id)
read: request.alias read: request.message
new Message
getChannel(account, alias)
channel, message
read: account
post Message
saveEvent(Event) updateChannel(channel)
read: request.alias read: request.messageid
delete Message
getChannel(account, alias)
channel, message
delete Event
deleteEvent(message)
message / messageid
get Event
getEvent(message)
channel, account, alias
printStandardHead printStandardTop printChannelList printChannel
static HTML and channel list, channel id, channel version, channel name and all events in the channel
read: request.alias read: request.messageid read: request.newmessage
edit Message
getChannel(account, alias)
message / messageid
channel, message, newmessage
edit Message
updateChannel(channel)
create Channel Form
printStandardHead printStandardTop
static HTML
read: account
join Channel Form
username
printStandardHead printStandardTop
static HTML including username
read: request.alias read: request.originalcontent read: request.messageid
edit Message Form
alias, origialcontent, messageid
printStandardHead printStandardTop
static HTML including alias, originalcontent and messageid
read: account read: request.channelid
join Channel
getChannel(channelid) updateAccount(account) saveEvent(event) updateChannel(channel)
redirect to /channel/+"channelname"
read: session
logout
deleteSession(session)
redirect to /
read: request.version read: request.channelid
subscribe
getChannel(channelid, version)
channel
printChannelEvents
static HTML and all events in the channel and the channelname, channelid
read: account read: request.channelname
create Channel
saveChannel(channel)
account, channelid
join Channel
getChannel(channelid) updateAccount(account) saveEvent(event) updateChannel(channel)
redirect to /channel/+"channelname"
read: account
print /
username, account
printStandardHead printStandardTop printChannelList
static HTML and the username and a list of all channels which the user is registered in
every response after the authentication contains the current session-id as a cookie
DATABASES
Session Storage
AccountStorage
UserStorage
ChannelStorage
EventStorage
VARIABLES
account
request
session
cookies
static files

ZAP Scanning Report

Summary of Alerts

Generated on Mon, 4 Oct 2021 18:41:11

Risk Level	Number of Alerts
High	4
Medium	2
Low	4
Informational	2

Alerts

Name	Risk Level	Number of Instances
Cross Site Scripting (Reflected)	High	4
Path Traversal	High	1
SQL Injection	High	1
SQL Injection - Authentication Bypass	High	1
Buffer Overflow	Medium	1
X-Frame-Options Header Not Set	Medium	9
Absence of Anti-CSRF Tokens	Low	15
Cookie No HttpOnly Flag	Low	11
Cookie without SameSite Attribute	Low	11
X-Content-Type-Options Header Missing	Low	11
Charset Mismatch (Header Versus Meta Content-Type Charset)	Informational	3
Loosely Scoped Cookie	Informational	21

Alert Detail

High (Medium)	Cross Site Scripting (Reflected)
Description	Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology. When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise. There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based. Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash. Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.
URL	http://localhost:8080/editMessage
Method	POST
Parameter	channelname

Attack	javascript:alert(1);
Evidence	javascript:alert(1);
URL	http://localhost:8080/editMessage
Method	POST
Paramet er	message
Attack	"><script>alert(1);</script>
Evidence	"><script>alert(1);</script>
URL	http://localhost:8080/editMessage
Method	POST
Paramet er	originalcontent
Attack	</textarea><script>alert(1);</script><textarea>
Evidence	</textarea><script>alert(1);</script><textarea>
URL	http://localhost:8080/
Method	POST
Paramet er	username
Attack	</title><script>alert(1);</script><title>
Evidence	</title><script>alert(1);</script><title>
Instances	4
Solution	Phase: Architecture and Design Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket. Phases: Implementation; Architecture and Design Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies. For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters. Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed. Phase: Architecture and Design For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server. If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated. Phase: Implementation For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping. To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set. Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of

	consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue." Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.
Reference	http://projects.webappsec.org/Cross-Site-Scripting http://cwe.mitre.org/data/definitions/79.html
CWE Id	79
WASC Id	8
Source ID	1

High (Low)	**Path Traversal**
Description	The Path Traversal attack technique allows an attacker access to files, directories, and commands that potentially reside outside the web document root directory. An attacker may manipulate a URL in such a way that the web site will execute or reveal the contents of arbitrary files anywhere on the web server. Any device that exposes an HTTP-based interface is potentially vulnerable to Path Traversal. Most web sites restrict user access to a specific portion of the file-system, typically called the "web document root" or "CGI root" directory. These directories contain the files intended for user access and the executable necessary to drive web application functionality. To access files or execute commands anywhere on the file-system, Path Traversal attacks will utilize the ability of special-characters sequences. The most basic Path Traversal attack uses the "../" special-character sequence to alter the resource location requested in the URL. Although most popular web servers will prevent this technique from escaping the web document root, alternate encodings of the "../" sequence may help bypass the security filters. These method variations include valid and invalid Unicode-encoding ("..%u2216" or "..%c0%af") of the forward slash character, backslash characters ("..\") on Windows-based servers, URL encoded characters "%2e%2e%2f"), and double URL encoding ("..%255c") of the backslash character. Even if the web server properly restricts Path Traversal attempts in the URL path, a web application itself may still be vulnerable due to improper handling of user-supplied input. This is a common problem of web applications that use template mechanisms or load static text from files. In variations of the attack, the original URL parameter value is substituted with the file name of one of the web application's dynamic scripts. Consequently, the results can reveal source code because the file is interpreted as text instead of an executable script. These techniques often employ additional special characters such as the dot (".") to reveal the listing of the current working directory, or "%00" NULL characters in order to bypass rudimentary file extension checks.

URL	http://localhost:8080/subscribe/4a504623-9f7b-4043-82d2-8fb1fbceb835?version=4a504623-9f7b-4043-82d2-8fb1fbceb835
Method	GET
Parameter	version
Attack	4a504623-9f7b-4043-82d2-8fb1fbceb835
Instances	1
Solution	Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue." For filenames, use stringent allow lists that limit the character set to be used. If feasible, only allow a single "." character in the filename to avoid weaknesses, and exclude directory separators such as "/". Use an allow list of allowable file extensions. Warning: if you attempt to cleanse your data, then do so that the end result is not in the form that can be dangerous. A sanitizing mechanism can remove characters such as '.' and ';' which may be required for some exploits. An attacker can try to fool the sanitizing mechanism into "cleaning" data into a dangerous form. Suppose the attacker injects a '.' inside a filename (e.g. "sensi.tiveFile") and the sanitizing mechanism removes the character resulting in the valid filename, "sensitiveFile". If the input data are now assumed to be safe, then the file may be compromised. Inputs should be decoded and canonicalized to the application's current internal representation before being validated. Make sure that your application does not decode the same input twice. Such errors could be used to bypass allow list schemes by introducing dangerous inputs after they have been checked. Use a built-in path canonicalization function (such as realpath() in C) that produces the canonical version of the pathname, which effectively removes ".." sequences and symbolic links. Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a

	single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations. When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs. Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by your software. OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows you to specify restrictions on file operations. This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.
Other information	Check 5

Reference	http://projects.webappsec.org/Path-Traversal http://cwe.mitre.org/data/definitions/22.html
CWE Id	22
WASC Id	33
Source ID	1

High (Medium)	**SQL Injection**
Description	SQL injection may be possible.

URL	http://localhost:8080/
Method	POST
Parameter	channelname
Attack	ZAP AND 1=1 --
Instances	1
Solution	Do not trust client side input, even if there is client side validation in place. In general, type check all data on the server side. If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?' If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries. If database Stored Procedures can be used, use them. Do *not* concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality! Do not create dynamic SQL queries using simple string concatenation. Escape all data received from the client. Apply an 'allow list' of allowed characters, or a 'deny list' of disallowed characters in user input. Apply the principle of least privilege by using the least privileged database user possible. In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact. Grant the minimum database access that is necessary for the application.
Other information	The page results were successfully manipulated using the boolean conditions [ZAP AND 1=1 --] and [ZAP AND 1=2 --] The parameter value being modified was NOT stripped from the HTML output for the purposes of the comparison Data was returned for the original parameter. The vulnerability was detected by successfully restricting the data originally returned, by manipulating the parameter

Reference	https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
CWE Id	89
WASC Id	19
Source ID	1

High (Medium)	**SQL Injection - Authentication Bypass**
Description	SQL injection may be possible on a login page, potentially allowing the application's authentication mechanism to be bypassed

URL	http://localhost:8080/
Method	POST
Parameter	channelname
Attack	ZAP AND 1=1 --
Instances	1
Solution	Do not trust client side input, even if there is client side validation in place. In general, type check all data on the server side. If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?' If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries. If database Stored Procedures can be used, use them. Do *not* concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality! Do not create dynamic SQL queries using simple string concatenation. Escape all data received from the client. Apply an 'allow list' of allowed characters, or a 'deny list' of disallowed characters in user input. Apply the principle of least privilege by using the least privileged database user possible. In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact. Grant the minimum database access that is necessary for the application.
Reference	https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
CWE Id	89
WASC Id	19
Source ID	1

Medium (Medium)	Buffer Overflow
Description	Buffer overflow errors are characterized by the overwriting of memory spaces of the background web process, which should have never been modified intentionally or unintentionally. Overwriting values of the IP (Instruction Pointer), BP (Base Pointer) and other registers causes exceptions, segmentation faults, and other process errors to occur. Usually these errors end execution of the application in an unexpected way.

URL	http://localhost:8080/subscribe/4a504623-9f7b-4043-82d2-8fb1fbceb835?version=558c11a5-5ea4-46c8-bba5-a010dd0a84e7
Method	GET
Parameter	version
Attack	DUJpqgAMmytYejtugnymqQYPtvWGpEOLhCjkqBXfWGaROtSUefSWSIdNWjprSxNnhfpuD LRehHHnkPPkWafWiopdxnVWudPMwxMTtCobDQEofUXdAAaDHHbsarOAURvNCjbGcOs TmPjjadoFFFChParcpCTXNMoPhUAOBhlJvbaKERxErDbtZfjQsvxlwjHNUZcmZntPEBHbsm hXrSCghjNnhSjXigKvNDprINUgEdibmopXcfQdqtxnuwNAfHHDutagifUyjWjhMdSweHKcqLZ jWNYicZRuqhpoFraqyTDgscxtZNjZjtfMxaToTbgcCEneTbidxJlTmCKqfNkOcttXXHoPixwRJe EkZKFYVTFkkFxHOVNBpeQBRBECRgTqkXidMpcgICSQdXegpKCJEwehJlRWcgSLZSSXU taWgwWAGINVsMuPwOxAQAHYEFyWAfnVAcJYDFvhXeWwgcUjnnxLWxKYPnGXDmuQck PMgmwqdKbirsOjRiFmumepbXOqigFrSVfIsHyfHCPQQZouhxNXIyMbYNFTxMRNxMJPXnp GoSirMYhWYKfbJitogBRlVDkKFNNlQSHwDxZSVOOiMbrosbCrayjforeinIgYAgFeLdjELnlsD KNTDuuCUBYfkVUraNIROLYgKTDXDimtqIHMEPNjgCRKYevhgRivtdJpBpsAhZpKTCYXXU mXsLJrKrHLfWdCMrvbtyAnTxOhgPLBweRciTloPmaDIQamReFhqRgaoVMwEaxyFQwtPsac xaDXVBDPKABFVCDIxRCKrKjaxJDFrIpEdsXuIpqibwaMjEHkGartctDISDmjVhAKtcthdwowk CbFSNhpBXcPJbHSXBjVXWJcHkHPAKFYUNRsVgQIybSiOshJyrRagUPntghQetwiHTAYVV ycCwfUVGaTeeElhMkavPSmNNOdkRxMMFiKmxMASmgPmAwbDAppoaFgXGTDKefuWVK EqvEIZpiqDnVPORYFkuKaJWTwmnJVXpiRlTljmrjkZvcNYxlAERrNSUcNJVRibZprOgpJKyIM cdvfCWqfbsNgMyanQVwNIMLnyxdyWbcparcqTqhtgSgjFQlLqPDbZJiCfKBRElxWuRpLStGi ySSaEacILwnBntaOLClCExkoPyIxYydcLYIAEMiKcBgywjpXXTWGmcegXZNwFlVnpWrVVuK rgvGJxrNnUKbhkZXmKPdvKsLercpnaSoSaFXFahDNLrOcIqxinlZFrkYLTIQHtyMBsHGeCGk NKlSWCOQeYdWSBhmNlsOaHglikycRQBkPAFUKkBBhbWwbAGqcnjEaRAIsCjWbbDILRm IGAFuNXTakLXglRLdXJenqDADlIMTEvRmrLHQLkTacCMKPinWiGkXGFJrYSEIMtDvTBRLLF vHsvwQpdqyMKyvTfmkQMRYIqXZBAybRXgeLFwEMrHcIyvWapAxgJCybbGBoGhJpTdPfaf wkyNnfKgtmfLgEuehCbHQPWvBvBGXFTPUQlXVqpXxFWbEaouYrmNjcFTOqqAHXmlNMD BZMyAhxDhBmgosPsTLoCketTnKRQPmyhooGReOknCDZRUjQvcYcQNRYqvIIeUwQpjOR wSFBlXnOiYdwwSbyBxPraLclBOjEfQHrkVFVxYyOsRPuQXeYThSUXrXaQdYOEjUdQvhOQ ORwSWXDBXbgqHUKvkIOdEaIFrVAJydYpOJbWMQjbmmDRJUWqIxdhQftnTUtUXuRUPH bqxbqCUNyMHrIJZhRQDwNIwqkCGUoxgNbuMEuFTmFiYrNuXdlJGTUZrUTgTfJwjmLZbs WDOEFFJlWWvMxvHEyHpQygkrRIuAAxmCbVaRqhObSAMUwEnyTxBvAyOCKEGvNggwH VILFOLgAlAeqeixsQnEudZZaFsKOaDuPpqjhtSLXicrklRxcCcGFpbxlNbVTPAsUGZYQcXphc wINqRvSVHRaGGGfHVaclabsSMTqOCCOFIrEftAjElsGsGwEvOyQlXKweGpOYKscieXUQKn xVhfWgIBcoZAAJvMrHbdJWSCqDsyBWcHqCfbnIevwHHFOFmMFrfYsVfFsuiTbYpwvsskXi HnkdhgwOxhwJbYj
Evidence	Connection: close

Instances	1
Solution	Rewrite the background program using proper return length checking. This will require a recompile of the background executable.
Other information	Potential Buffer Overflow. The script closed the connection and threw a 500 Internal Server Error

Reference	https://owasp.org/www-community/attacks/Buffer_overflow_attack
CWE Id	120
WASC Id	7
Source ID	1

Medium (Medium)	**X-Frame-Options Header Not Set**
Description	X-Frame-Options header is not included in the HTTP response to protect against 'ClickJacking' attacks.

URL	http://localhost:8080
Method	GET
Parameter	X-Frame-Options
URL	http://localhost:8080/
Method	GET
Parameter	X-Frame-Options
URL	http://localhost:8080/channel/TestChannel
Method	POST
Parameter	X-Frame-Options
URL	http://localhost:8080/create
Method	GET
Parameter	X-Frame-Options
URL	http://localhost:8080/register
Method	GET
Parameter	X-Frame-Options
URL	http://localhost:8080/
Method	POST
Parameter	X-Frame-Options
URL	http://localhost:8080/channel/TestChannel
Method	GET
Parameter	X-Frame-Options
URL	http://localhost:8080/subscribe/4a504623-9f7b-4043-82d2-8fb1fbceb835?version=b63debb6-1fce-4c7b-8d27-29c0c869294d
Method	GET
Parameter	X-Frame-Options
URL	http://localhost:8080/login
Method	GET
Parameter	X-Frame-Options
Instances	9
Solution	Most modern Web browsers support the X-Frame-Options HTTP header. Ensure it's set on all web pages returned by your site (if you expect the page to be framed only by pages on your server (e.g. it's part of a FRAMESET) then you'll want to use SAMEORIGIN, otherwise if you never expect the page to be framed, you should use DENY. Alternatively consider implementing Content Security Policy's "frame-ancestors" directive.
Reference	https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options
CWE Id	1021
WASC Id	15
Source ID	3

Low (Medium)	**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

Description	request to a target destination without their knowledge or intent in order to perform an action as the victim. The underlying cause is application functionality using predictable URL/form actions in a repeatable way. The nature of the attack is that CSRF exploits the trust that a web site has for a user. By contrast, cross-site scripting (XSS) exploits the trust that a user has for a web site. Like XSS, CSRF attacks are not necessarily cross-site, but they can be. Cross-site request forgery is also known as CSRF, XSRF, one-click attack, session riding, confused deputy, and sea surf. CSRF attacks are effective in a number of situations, including: * The victim has an active session on the target site. * The victim is authenticated via HTTP auth on the target site. * The victim is on the same local network as the target site. CSRF has primarily been used to perform an action against a target site using the victim's privileges, but recent techniques have been discovered to disclose information by gaining access to the response. The risk of information disclosure is dramatically increased when the target site is vulnerable to XSS, because XSS can be used as a platform for CSRF, allowing the attack to operate within the bounds of the same-origin policy.

URL	http://localhost:8080/channel/TestChannel
Method	GET
Evidence	<form class="entry" action="/channel/TestChannel" method="post">
URL	http://localhost:8080/login
Method	GET
Evidence	<form class="login" action="/" method="post">
URL	http://localhost:8080/channel/TestChannel
Method	POST
Evidence	<form class="entry" action="/channel/TestChannel" method="post">
URL	http://localhost:8080/create
Method	GET
Evidence	<form class="login" action="/" method="POST">
URL	http://localhost:8080/channel/TestChannel
Method	GET
Evidence	<form action="/channel/TestChannel" method="post">
URL	http://localhost:8080/subscribe/4a504623-9f7b-4043-82d2-8fb1fbceb835?version=b63debb6-1fce-4c7b-8d27-29c0c869294d
Method	GET
Evidence	<form style="grid-area: edit;" action="/editMessage" method="POST">
URL	http://localhost:8080/subscribe/4a504623-9f7b-4043-82d2-8fb1fbceb835?version=b63debb6-1fce-4c7b-8d27-29c0c869294d
Method	GET
Evidence	<form style="grid-area: delete;" action="/channel/TestChannel" method="POST">
URL	http://localhost:8080/channel/TestChannel
Method	GET
Evidence	<form style="grid-area: edit;" action="/editMessage" method="POST">
URL	http://localhost:8080/channel/TestChannel
Method	GET
Evidence	<form style="grid-area: delete;" action="/channel/TestChannel" method="POST">
URL	http://localhost:8080/channel/TestChannel
Method	GET
Evidence	<form class="entry" action="/channel/TestChannel" method="post">
URL	http://localhost:8080/channel/TestChannel
Method	GET
Evidence	<form action="/channel/TestChannel" method="post">
URL	http://localhost:8080/channel/TestChannel
Method	POST
Evidence	<form style="grid-area: edit;" action="/editMessage" method="POST">
URL	http://localhost:8080/register
Method	GET
Evidence	<form class="register" action="/" method="post">
URL	http://localhost:8080/channel/TestChannel
Method	POST
Evidence	<form action="/channel/TestChannel" method="post">

URL	http://localhost:8080/channel/TestChannel
Method	POST
Evidence	<form style="grid-area: delete;" action="/channel/TestChannel" method="POST">
Instances	15
Solution	Phase: Architecture and Design Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, use anti-CSRF packages such as the OWASP CSRFGuard. Phase: Implementation Ensure that your application is free of cross-site scripting issues, because most CSRF defenses can be bypassed using attacker-controlled script. Phase: Architecture and Design Generate a unique nonce for each form, place the nonce into the form, and verify the nonce upon receipt of the form. Be sure that the nonce is not predictable (CWE-330). Note that this can be bypassed using XSS. Identify especially dangerous operations. When the user performs a dangerous operation, send a separate confirmation request to ensure that the user intended to perform that operation. Note that this can be bypassed using XSS. Use the ESAPI Session Management control. This control includes a component for CSRF. Do not use the GET method for any request that triggers a state change. Phase: Implementation Check the HTTP Referer header to see if the request originated from an expected page. This could break legitimate functionality, because users or proxies may have disabled sending the Referer for privacy reasons.
Other information	No known Anti-CSRF token [anticsrf, CSRFToken, __RequestVerificationToken, csrfmiddlewaretoken, authenticity_token, OWASP_CSRFTOKEN, anoncsrf, csrf_token, _csrf, _csrfSecret, __csrf_magic, CSRF] was found in the following HTML form: [Form 3: "newmessage" "send"].

Reference	http://projects.webappsec.org/Cross-Site-Request-Forgery http://cwe.mitre.org/data/definitions/352.html
CWE Id	352
WASC Id	9
Source ID	3

Low (Medium)	**Cookie No HttpOnly Flag**
Description	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.

URL	http://localhost:8080/sitemap.xml
Method	GET
URL	http://localhost:8080/channel/TestChannel
Method	GET
URL	http://localhost:8080/
Method	GET
URL	http://localhost:8080/channel/TestChannel
Method	POST
URL	http://localhost:8080
Method	GET
URL	http://localhost:8080/create
Method	GET
URL	http://localhost:8080/robots.txt
Method	GET
URL	http://localhost:8080/channel
Method	GET
URL	http://localhost:8080/logout
Method	GET
URL	http://localhost:8080/

Method	POST
URL	http://localhost:8080/subscribe/4a504623-9f7b-4043-82d2-8fb1fbceb835?version=b63debb6-1fce-4c7b-8d27-29c0c869294d
Method	GET
Instances	11
Solution	Ensure that the HttpOnly flag is set for all cookies.
Reference	https://owasp.org/www-community/HttpOnly
CWE Id	1004
WASC Id	13
Source ID	3

Low (Medium)	**Cookie without SameSite Attribute**
Description	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.
URL	http://localhost:8080/
Method	GET
URL	http://localhost:8080/logout
Method	GET
URL	http://localhost:8080/create
Method	GET
URL	http://localhost:8080/sitemap.xml
Method	GET
URL	http://localhost:8080/subscribe/4a504623-9f7b-4043-82d2-8fb1fbceb835?version=b63debb6-1fce-4c7b-8d27-29c0c869294d
Method	GET
URL	http://localhost:8080/
Method	POST
URL	http://localhost:8080/robots.txt
Method	GET
URL	http://localhost:8080
Method	GET
URL	http://localhost:8080/channel/TestChannel
Method	POST
URL	http://localhost:8080/channel
Method	GET
URL	http://localhost:8080/channel/TestChannel
Method	GET
Instances	11
Solution	Ensure that the SameSite attribute is set to either 'lax' or ideally 'strict' for all cookies.
Reference	https://tools.ietf.org/html/draft-ietf-httpbis-cookie-same-site
CWE Id	1275
WASC Id	13
Source ID	3

Low (Medium)	**X-Content-Type-Options Header Missing**
Description	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.
URL	http://localhost:8080
Method	GET
Parameter	X-Content-Type-Options
URL	http://localhost:8080/channel/TestChannel
Method	POST
Parameter	X-Content-Type-Options
URL	http://localhost:8080/login

Method	GET
Parameter	X-Content-Type-Options
URL	http://localhost:8080/style.css
Method	GET
Parameter	X-Content-Type-Options
URL	http://localhost:8080/register
Method	GET
Parameter	X-Content-Type-Options
URL	http://localhost:8080/
Method	GET
Parameter	X-Content-Type-Options
URL	http://localhost:8080/create
Method	GET
Parameter	X-Content-Type-Options
URL	http://localhost:8080/
Method	POST
Parameter	X-Content-Type-Options
URL	http://localhost:8080/subscribe/4a504623-9f7b-4043-82d2-8fb1fbceb835?version=b63debb6-1fce-4c7b-8d27-29c0c869294d
Method	GET
Parameter	X-Content-Type-Options
URL	http://localhost:8080/script.js
Method	GET
Parameter	X-Content-Type-Options
URL	http://localhost:8080/channel/TestChannel
Method	GET
Parameter	X-Content-Type-Options
Instances	11
Solution	Ensure that the application/web server sets the Content-Type header appropriately, and that it sets the X-Content-Type-Options header to 'nosniff' for all web pages. If possible, ensure that the end user uses a standards-compliant and modern web browser that does not perform MIME-sniffing at all, or that can be directed by the web application/web server to not perform MIME-sniffing.
Other information	This issue still applies to error type pages (401, 403, 500, etc.) as those pages are often still affected by injection issues, in which case there is still concern for browsers sniffing pages away from their actual content type. At "High" threshold this scan rule will not alert on client or server error responses.

Reference	http://msdn.microsoft.com/en-us/library/ie/gg622941%28v=vs.85%29.aspx https://owasp.org/www-community/Security_Headers
CWE Id	693
WASC Id	15
Source ID	3

Informational (Low)	**Charset Mismatch (Header Versus Meta Content-Type Charset)**
Description	This check identifies responses where the HTTP Content-Type header declares a charset different from the charset defined by the body of the HTML or XML. When there's a charset mismatch between the HTTP header and content body Web browsers can be forced into an undesirable content-sniffing mode to determine the content's correct character set. An attacker could manipulate content on the page to be interpreted in an encoding of their choice. For example, if an attacker can control content at the beginning of the page, they could inject script using UTF-7 encoded text and manipulate some browsers into interpreting that text.

URL	http://localhost:8080/channel
Method	GET

URL	http://localhost:8080/sitemap.xml
Method	GET
URL	http://localhost:8080/robots.txt
Method	GET
Instances	3
Solution	Force UTF-8 for all text content in both the HTTP header and meta tags in HTML or encoding declarations in XML.
Other information	There was a charset mismatch between the HTTP Header and the META content-type encoding declarations: [iso-8859-1] and [utf-8] do not match.

Reference	http://code.google.com/p/browsersec/wiki/Part2#Character_set_handling_and_detection
CWE Id	436
WASC Id	15
Source ID	3

Informational (Low)	**Loosely Scoped Cookie**
Description	Cookies can be scoped by domain or path. This check is only concerned with domain scope.The domain scope applied to a cookie determines which domains can access it. For example, a cookie can be scoped strictly to a subdomain e.g. www.nottrusted.com, or loosely scoped to a parent domain e.g. nottrusted.com. In the latter case, any subdomain of nottrusted.com can access the cookie. Loosely scoped cookies are common in mega-applications like google.com and live.com. Cookies set from a subdomain like app.foo.bar are transmitted only to that domain by the browser. However, cookies scoped to a parent-level domain may be transmitted to the parent, or any subdomain of the parent.

URL	http://localhost:8080/logout
Method	GET
URL	http://localhost:8080/create
Method	GET
URL	http://localhost:8080/subscribe/4a504623-9f7b-4043-82d2-8fb1fbceb835?version=b63debb6-1fce-4c7b-8d27-29c0c869294d
Method	GET
URL	http://localhost:8080/channel/TestChannel
Method	GET
URL	http://localhost:8080/
Method	GET
URL	http://localhost:8080/sitemap.xml
Method	GET
URL	http://localhost:8080/channel/TestChannel
Method	POST
URL	http://localhost:8080/channel/TestChannel
Method	GET
URL	http://localhost:8080/
Method	POST
URL	http://localhost:8080/logout
Method	GET
URL	http://localhost:8080/
Method	POST
URL	http://localhost:8080/
Method	POST
URL	http://localhost:8080/robots.txt
Method	GET
URL	http://localhost:8080/
Method	POST
URL	http://localhost:8080/logout
Method	GET
URL	http://localhost:8080/logout
Method	GET
URL	http://localhost:8080
Method	GET
URL	http://localhost:8080/channel
Method	GET
URL	http://localhost:8080/create

Method	GET
URL	http://localhost:8080/
Method	POST
Instances	21
Solution	Always scope cookies to a FQDN (Fully Qualified Domain Name).
Other information	The origin domain used for comparison was: localhost session=6efc4d01-d244-43de-86cf-fff13ed133fc
Reference	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
CWE Id	565
WASC Id	15
Source ID	3