LMMT-SecureDove

CPTS 428 Fall 2023
Professor Haipeng Cai

Logan Kloft
Manjesh Puram
Matthew Gerola
Taylor West

Table of Contents

Introduction

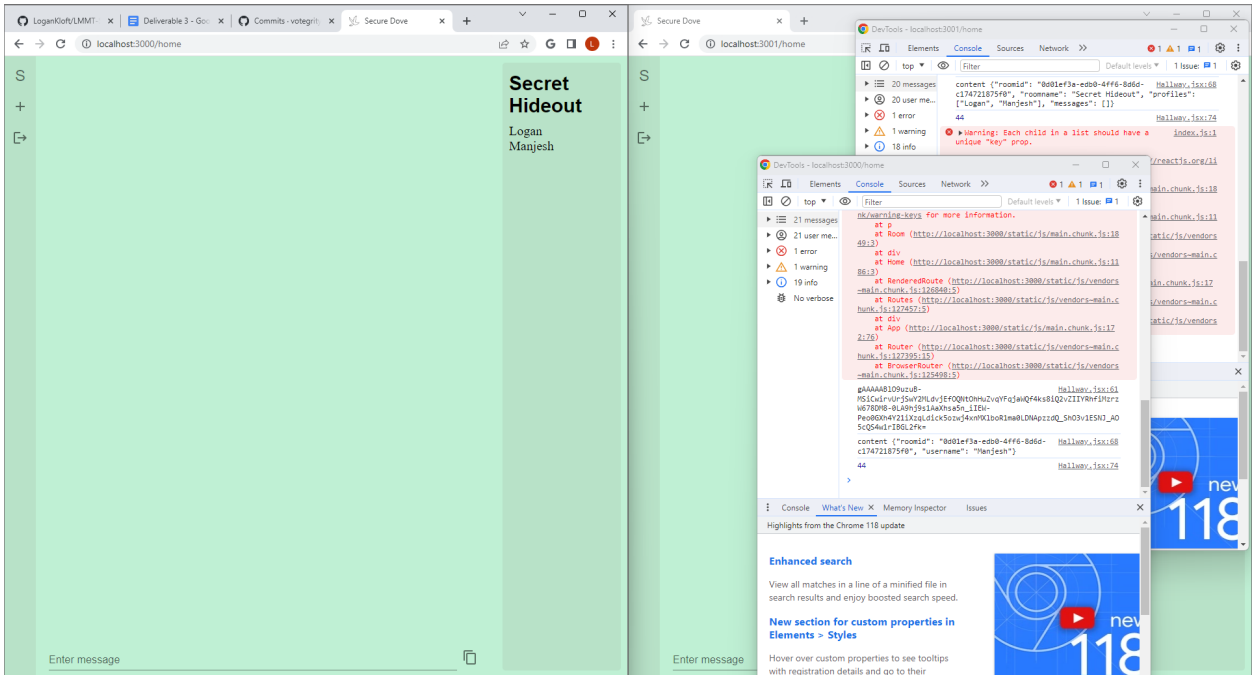
In the remainder of the report, you will see security tests targeting the confidentiality, integrity, and availability of our application. These tests target our original security goals, in addition to other security vulnerabilities that we have thought of since the security goal / measurement report. We signify failed tests using ❌ which means there exists a vulnerability. We indicate successful tests using ✅ which means our project implements security measurements that guard against the targeted vulnerability.

After these tests, we summarize the findings of the break / attack part and enhance the security plan to reflect any vulnerabilities discovered during the test. You may find code used to test security requirements in our repository linked here: https://github.com/LoganKloft/LMMT-SecureDove. Also, for each test that has code associated with it, we link to the code for convenience.

Confidentiality

Goal: At every step of message transferral, the message is encrypted ✅
How: Trace message path from creation in a client to receipt in another client

We will trace the route of one client sending a message to a chat room and then checking that it is encrypted when received by the server and also encrypted when broadcasted from the server (received on the client). To do this, we have embedded print statements in both the backend and frontend. Both the backend and frontend only print the “content” of the messages they receive. The content carries the actual payload of information that we wish to keep confidential. We use the developer tools to monitor print statements on the client:



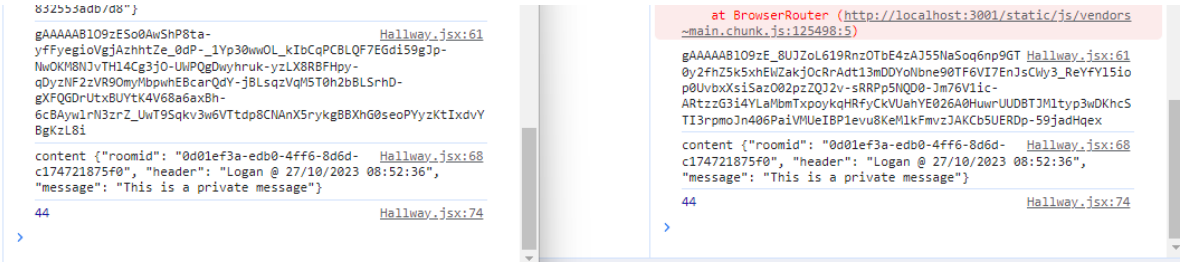
We use a terminal to view the backend’s output:

```
who:  Manjesh
{'type': 'room', 'verb': 'get', 'id': '84721251-b651-4def-bed1-7e3b9ebe4c80', 'content': {'roomid': '0d01ef3a-edb0-4ff6-8d6d-c174721875f0'}}

===== SINGLE REPONSE =====
{'type': 'room', 'verb': 'get', 'content': {'roomid': '0d01ef3a-edb0-4ff6-8d6d-c174721875f0', 'roomname': 'Secret Hideout', 'profiles': ['Logan'
, 'Manjesh'], 'messages': []}}

IN PARSER
gAAAAABlO9u1nINIF2-srqFeMTveGwmI-YrxJJ44dnyMVUpNn90PvRdBinFoEv_MsyC_NDhX4LFx8bdj1B44K7kHcqgXVF2__wFC6nlhtj-YUVrfXBKteK6RwXNGWUFeTnP1tCf9_Go7
FINISH DECODE
{'type': 'ACK', 'verb': 'post', 'content': {'id': '9c485ae1-cdc1-4717-b82b-aa9d03b2a5f7'}}
```

We start by entering: “This is a private message” on the left client (Logan). We will observe what the server receives and verify that the “content” is encrypted.

```
IN PARSER
gAAAAABlO9zFXTANTrVVrPkRH-CtP7KHSmCOlIwFOf0RTrgv_0C2qBET1MwiTDHb5P9KrXLWO6-Og4V5A1gWxxSfDllv3e2kiyKLGfs6F8prZeTCwZhRqk3Jcu01MnuvAjh3kdlksuOyykxfPT
mlo5aOPjmFosEglgFss-ov87UgoB5EVZHSwws5b_TDt8ZHYJiNC_yYbmBw
FINISH DECODE
{'type': 'message', 'verb': 'post', 'id': '07cc8101-c292-430a-a8c0-832553adb7d8', 'content': {'roomid': '0d01ef3a-edb0-4ff6-8d6d-c174721875f0', 'm
essage': 'This is a private message'}}
===== SEND ACK ENCRYPTED =====
{'type': 'ACK', 'verb': 'post', 'content': {'id': '07cc8101-c292-430a-a8c0-832553adb7d8'}}

===== REQUEST =====
who:  Logan
{'type': 'message', 'verb': 'post', 'id': '07cc8101-c292-430a-a8c0-832553adb7d8', 'content': {'roomid': '0d01ef3a-edb0-4ff6-8d6d-c174721875f0', 'm
essage': 'This is a private message'}}
```

In the above image we see that the content looks like a bunch of random characters (the stuff under “IN PARSER”). But once decrypting the content, then printing the entire request, we see that the message was successfully encrypted and transported to the server (the “FINISH DECODE” and “== REQUEST ==”). Now we need to confirm that the message is still encrypted when received by clients in the chatroom.

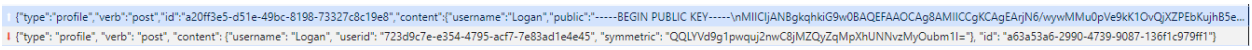


Here we can see that both clients receive encrypted content. Then upon decrypting the content and printing the entire response object, we see that it is indeed the original message that was sent from ‘Logan’. This demonstration shows that the message is encrypted during transport, so an attacker would have to either gain access to the server or a client to view unencrypted message content.

Goal: The message is undecipherable while encrypted ❌
How: Analyze complexity of the algorithm

We use symmetric encryption on two separate branches. On the main branch we use fernet symmetric encryption. On the asymmetric branch, we use `aes-256-ctr`. This algorithm is equivalent to AES with a 256-bit key except that it also includes an “iv” vector for additional security. This makes it safe against CPAs (chosen-plaintext attacks). Work in the future might include picking the iv vector more carefully; currently it’s chosen with random bytes and it’s important to avoid using the same iv for multiple messages. It could be that the crypto library we use has a cryptographically secure `randomBytes` function, but we would have to do more research.

For the main branch, after a user is created, all “content” is encrypted. However, there is an inherent vulnerability of someone stealing the symmetric key during the initial client-server transaction that shares the symmetric key. The following is a screenshot of the symmetric key with public visibility:



The security improvements suggest a solution to prevent the symmetric key from being public observable. Assuming we are able to secure the symmetric key, let's analyze the complexity of the algorithm to decide whether Fernet is susceptible to brute-force attacks. Fernet uses AES in CBC mode with a 128-bit key for encryption with PKCS7 padding[1]. Estimates for brute forcing AES-128-CBC are anywhere from ~3400 years to 2.158*10^9 years[2]. Clearly, brute forcing AES is not feasible.

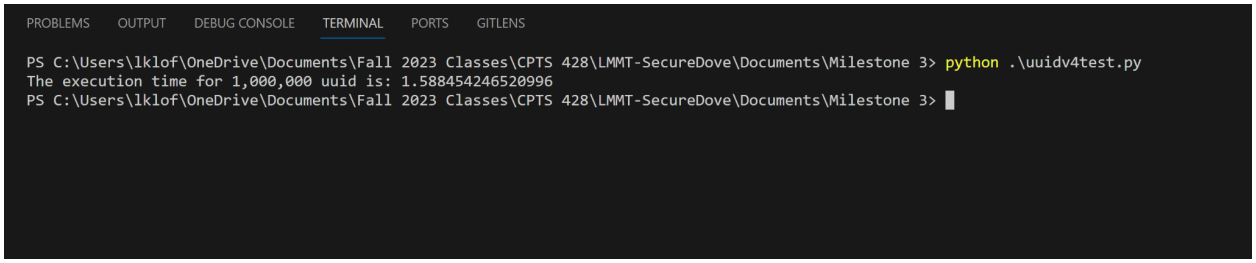
1. https://cryptography.io/en/latest/fernet/#limitations
2. Estimating aes brute force

Goal: Chat rooms are not able to be easily joined ✅
How: Calculate probability of guessing a chatroom id

Chat Room ids are used to join a chat room. They are generated using python's uuid library. Version 4 UUIDs are 128 bits long and reserve 2 bits for the UUID variant and 4 for the UUID version[1]. This leaves 122 bits to be randomly generated. Thus there are 2^122 or 5.3 undecillion (5.3 * 10^36) unique version 4 UUIDs[2].

Let's see how long it would take to have a 50% chance of guessing the uuid. One would need to generate 2^121 unique uuids (half of the number of unique version 4 UUIDs). If we assume that a user may generate 1 billion uuids per second, then it would take them 2^121 / 1 billion = 2.658456e+27 seconds. In 365 day years that is 2.658456e+27 / 31,536,000 = 8.4299086e+19 years. Not to mention the time of checking whether the uuid is correct will take longer than actually generating the uuid. It becomes clear that it is unreasonable to guess a room's uuid.

As an extension, we tested generating uuids on the following processor: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz. Here is an example output from the program:



The program linked is here:
https://github.com/LoganKloft/LMMT-SecureDove/blob/main/Documents/Milestone%203/uuidv4test.py

We can see that it took about 1.588 seconds to generate 1 million version 4 UUIDs. Now multiply the time by 1000 to estimate how long it would take to generate 1 billion UUIDs. It would take about 26.467 minutes. Clearly the idea of generating 1 billion UUIDs per second is

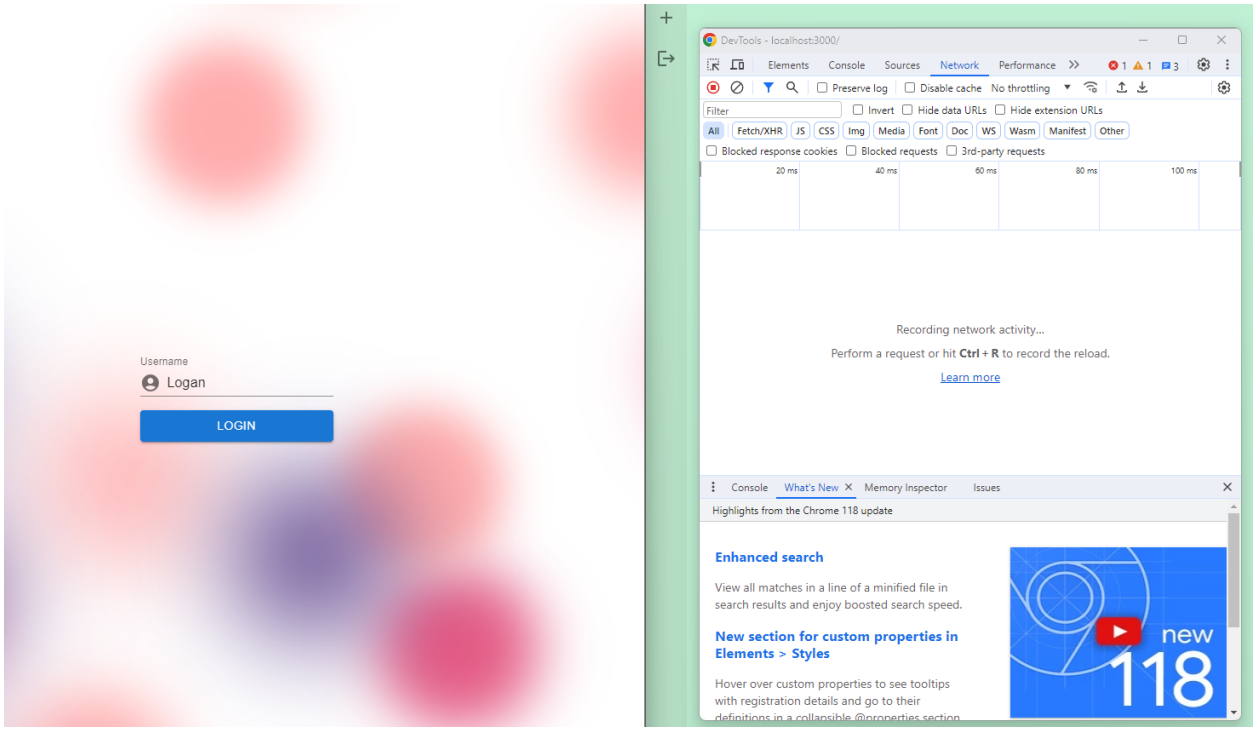
not feasible for a common computer, so to attempt the feat of guessing a UUID, one would need vast compute resources.

1. RFC 4122
2. Universally Unique Identifier

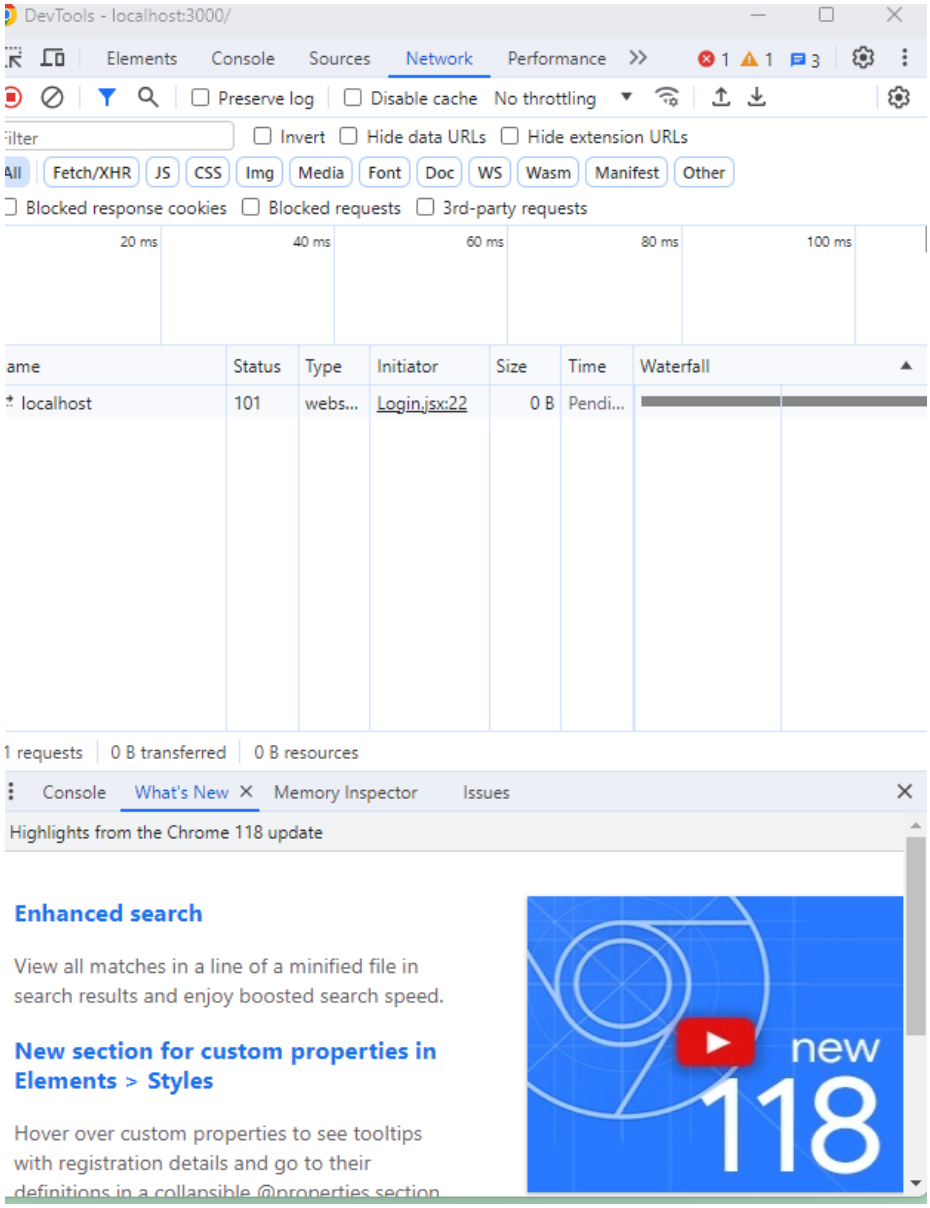
How: Intercept a chatroom creation / join request using developer tools

Now, instead of trying to guess a uuid, how about intercepting it? In the following we use the network developer tools to observe outbound requests by one user to see whether the room id is publicly visible.

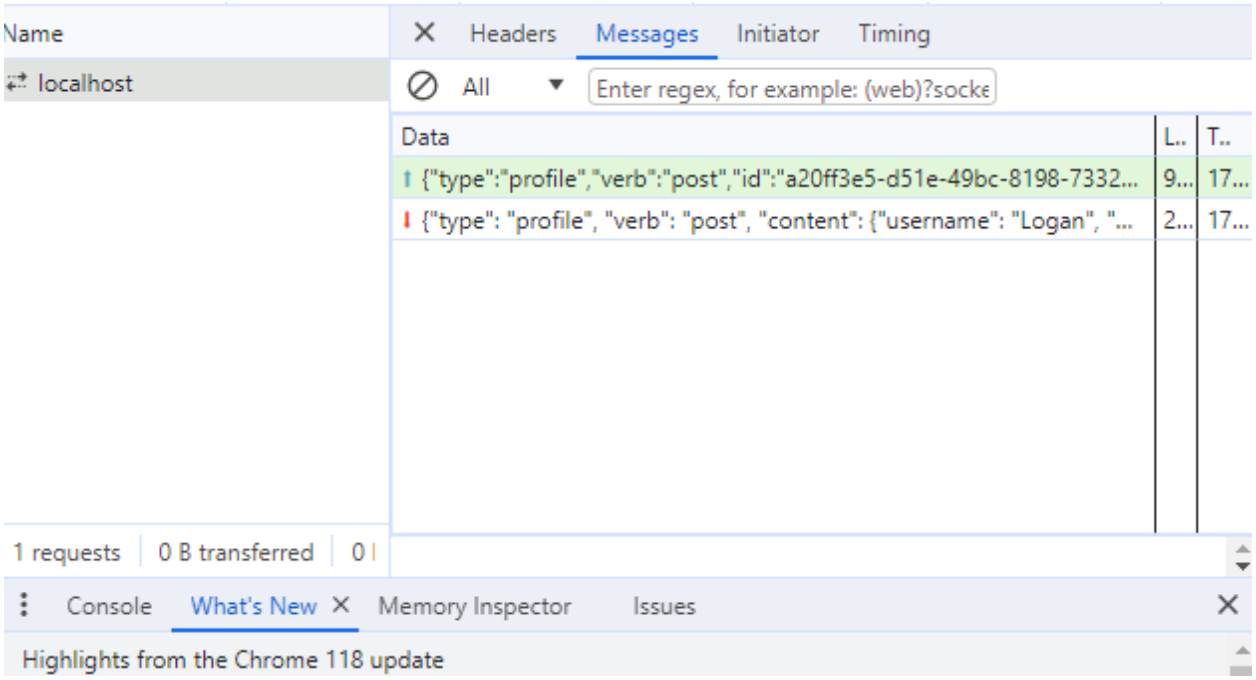
This image shows the network tab



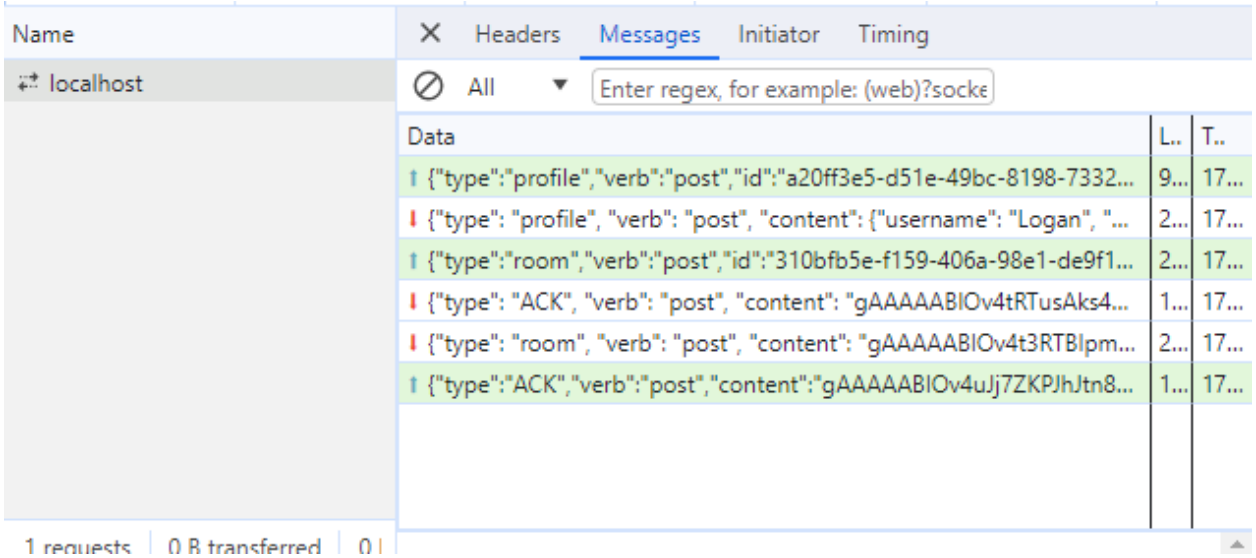
After logging in, we see the websocket connection:



We click the request and view the messages pane:



You can see that we have an original message from the client to the server (green arrow) that establishes a new user profile. Then, a second message from the server to the client (red arrow) that lets us know that profile was successfully created. Now let's create a room and observe what happens:



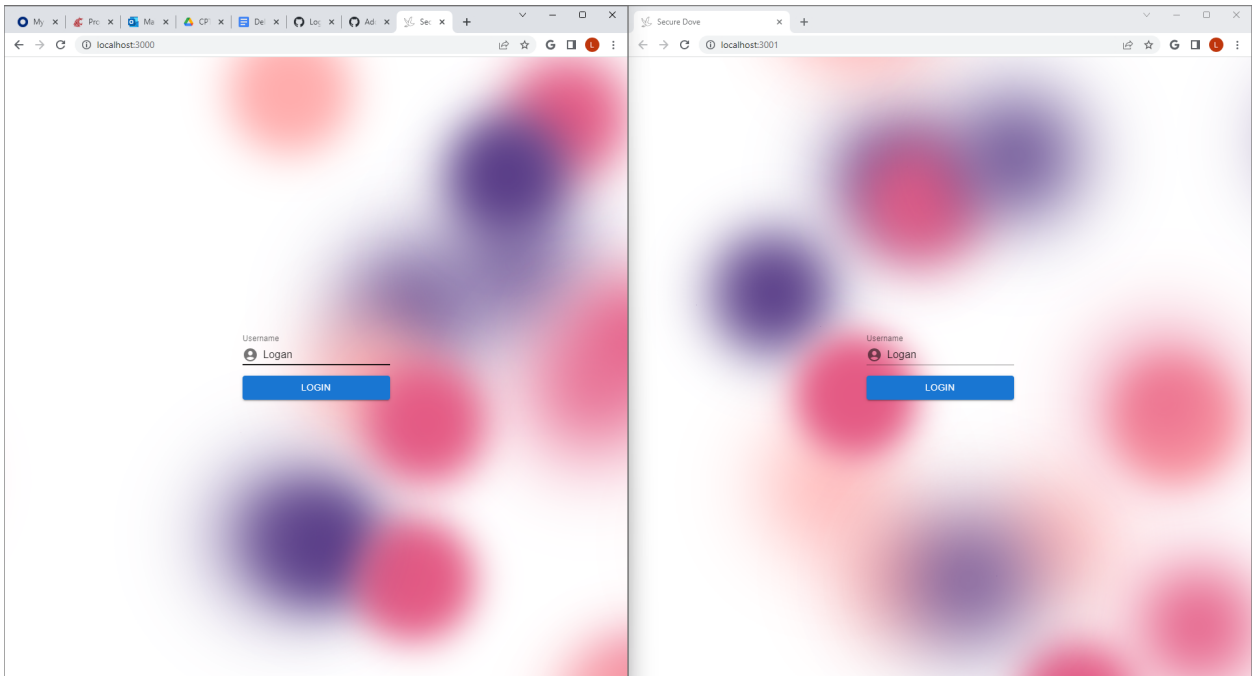
That looks odd, what is that id field for the "type": room message from the client? Let's copy that and try entering it on another client. The id is: '310bfb5e-f159-406a-98e1-de9f1ef70c00'. We try entering that into the other client but it does not work. Let's copy the actual room id from the first client: f5ecf61b-5bf1-468d-aaa4-7b2ba5d1d499. Obviously the code we copied was not actually the roomid. Additionally, no other message looks suspect for containing the roomid. This is

because our application encrypts the room id so that if it’s intercepted, it can’t be reasonably used by an attacker.

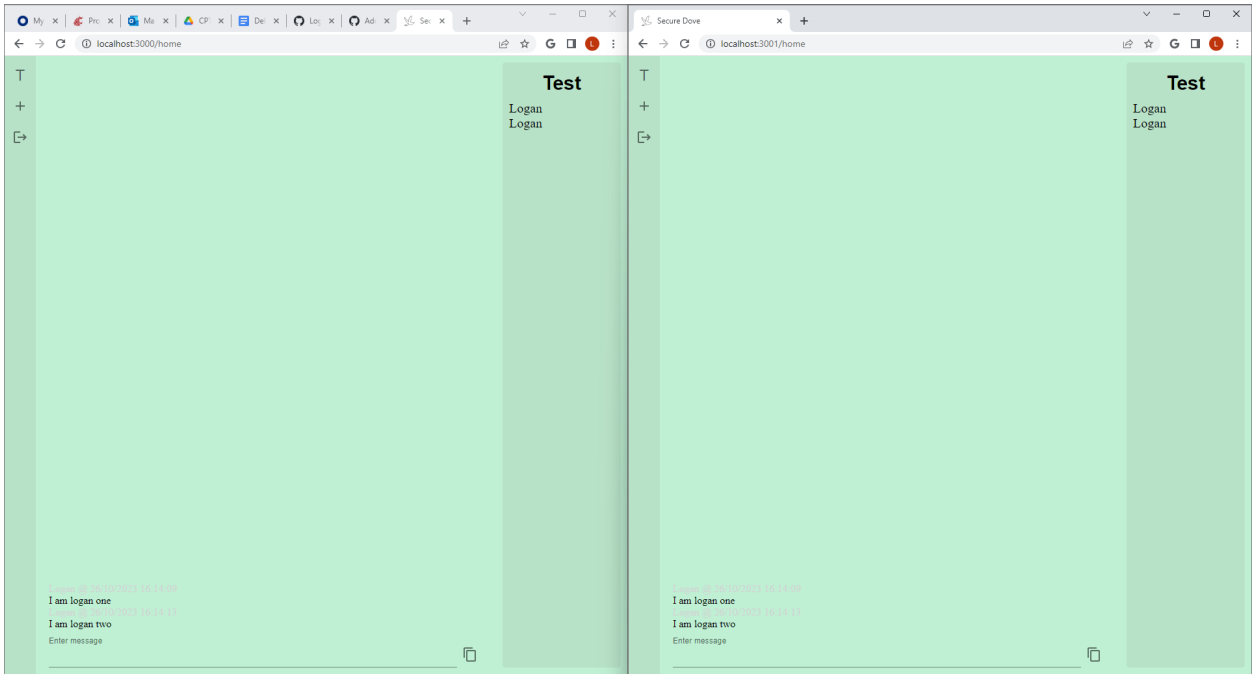
Integrity

Goal: Prevent attacker from mimicking somebody else ❌
How: Attempt to log in with the same username on two separate clients. Join the same room and send messages.

We have two clients running, both sign in as Logan.

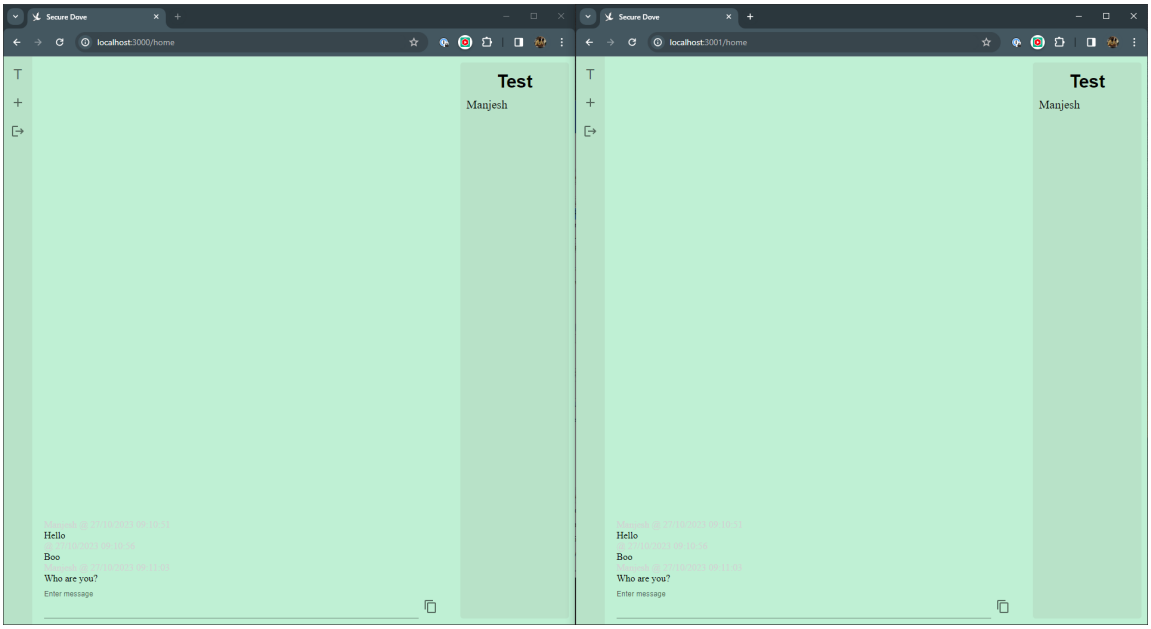


The left client creates a room which the right client joins. On the left client we also join the room, and then continue to enter the message “I am logan one”. Similarly, on the right client we enter, “I am logan two”.



You will notice that the message uses the same name and differs only in timestamp. Also, the user list contains two identical usernames. We can obviously see that there are two users with the same name, but to a third party we would be unable to tell which Logan had sent which message. This test proves that we need to protect against impersonation.

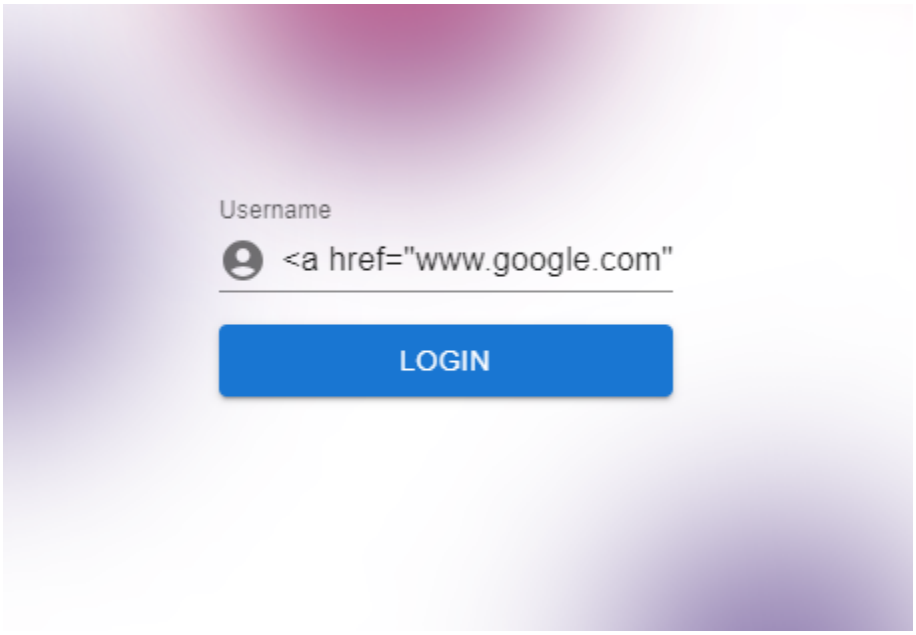
Goal: Transparency of who is able to see messages in a chat room ❌
How: Create a user with a blank name, join a chat room, can users see a visual indication that another user has joined the room?



Here in this image we can see that there are two users. The one on the right is signed in with the user “Manjesh” and the one on the left is the user signed in with the user “ “. This poses an issue because in the list of users, we aren’t able to clearly see that another user has joined the room. Another issue is that since there is no way to tell if a user with a blank name has joined, is that you could be typing in passwords or other secure messages all while the blank user is observing everything that unfolds in front of them.

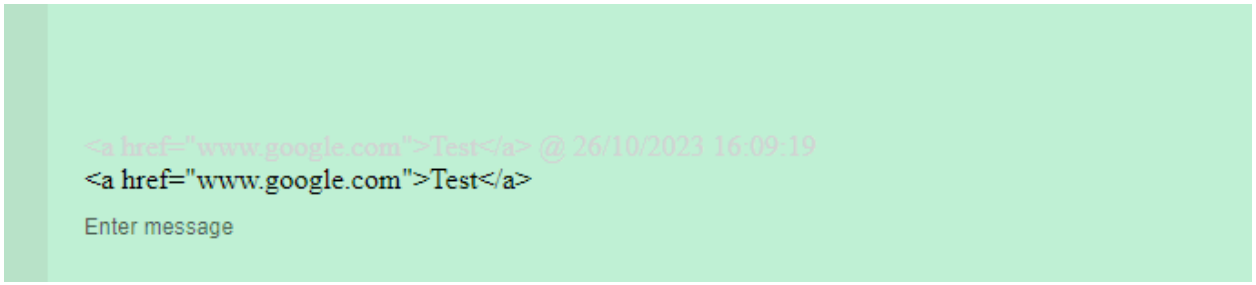
Goal: Input is not vulnerable to injected html ✅
How: In input elements, enter valid html to see whether a change is caused

We enter our username as an anchor tag that links to www.google.com

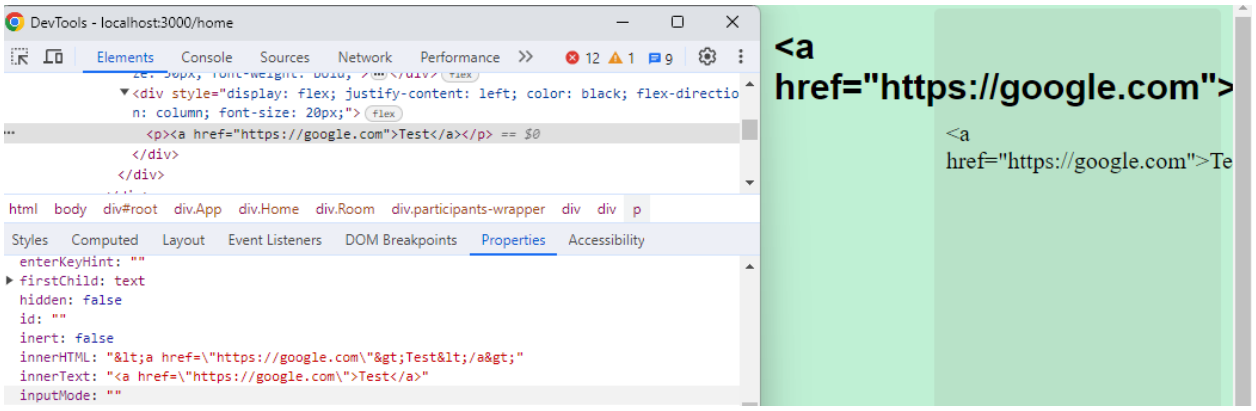


We create a room using the same anchor tag

We enter a message using the same anchor tag



We view the user as an example to see how the content is being rendered

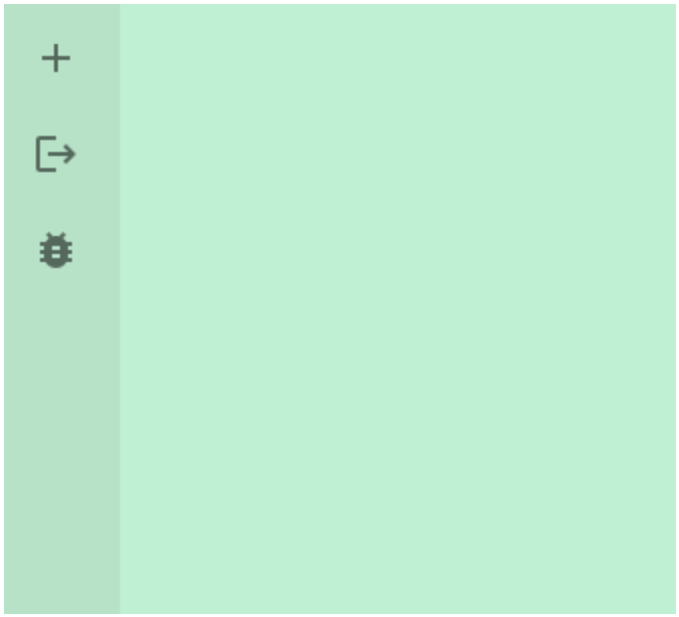


Look at the innerHTML property. You can see that special characters are correctly escaped. In this case our application correctly functions and passes this minimal assessment for vulnerability to XSS.

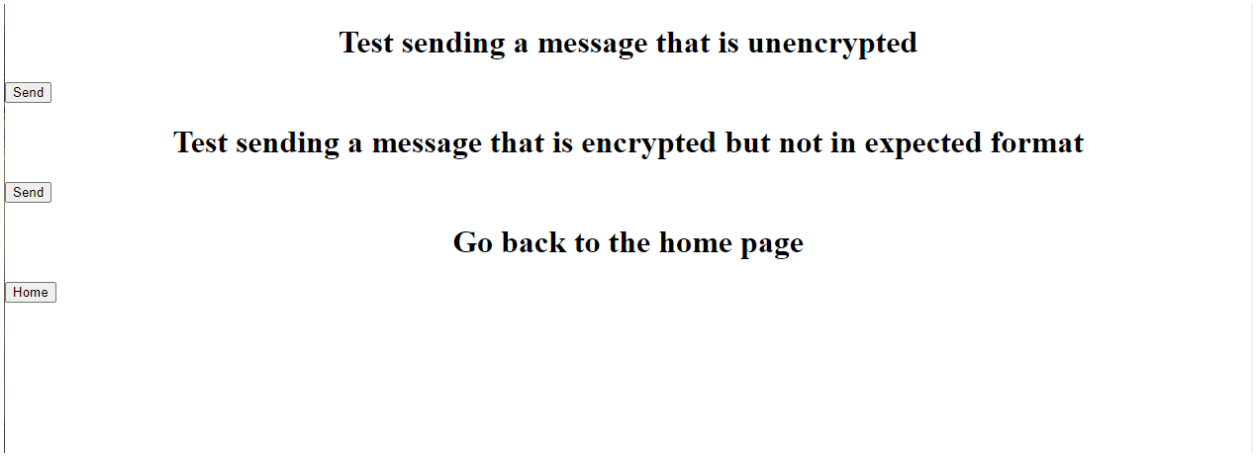
Availability

Goal: Malformed data in websocket does not crash server ✅
How: Send a request that the server is not able to decrypt

To test this we use a test page that can be accessed by clicking the bug icon:



Test Page:
https://github.com/LoganKloft/LMMT-SecureDove/blob/main/client/src/components/TestClient.jsx



Then we send a message that is not encrypted using the first ‘send’ button. The server expects that messages are encrypted once a user is logged in. The image below shows the server output when we click ‘send’. Notice that the server detects an error and logs ‘content’. In this case, the server continues to function even if the content it receives is not encrypted.

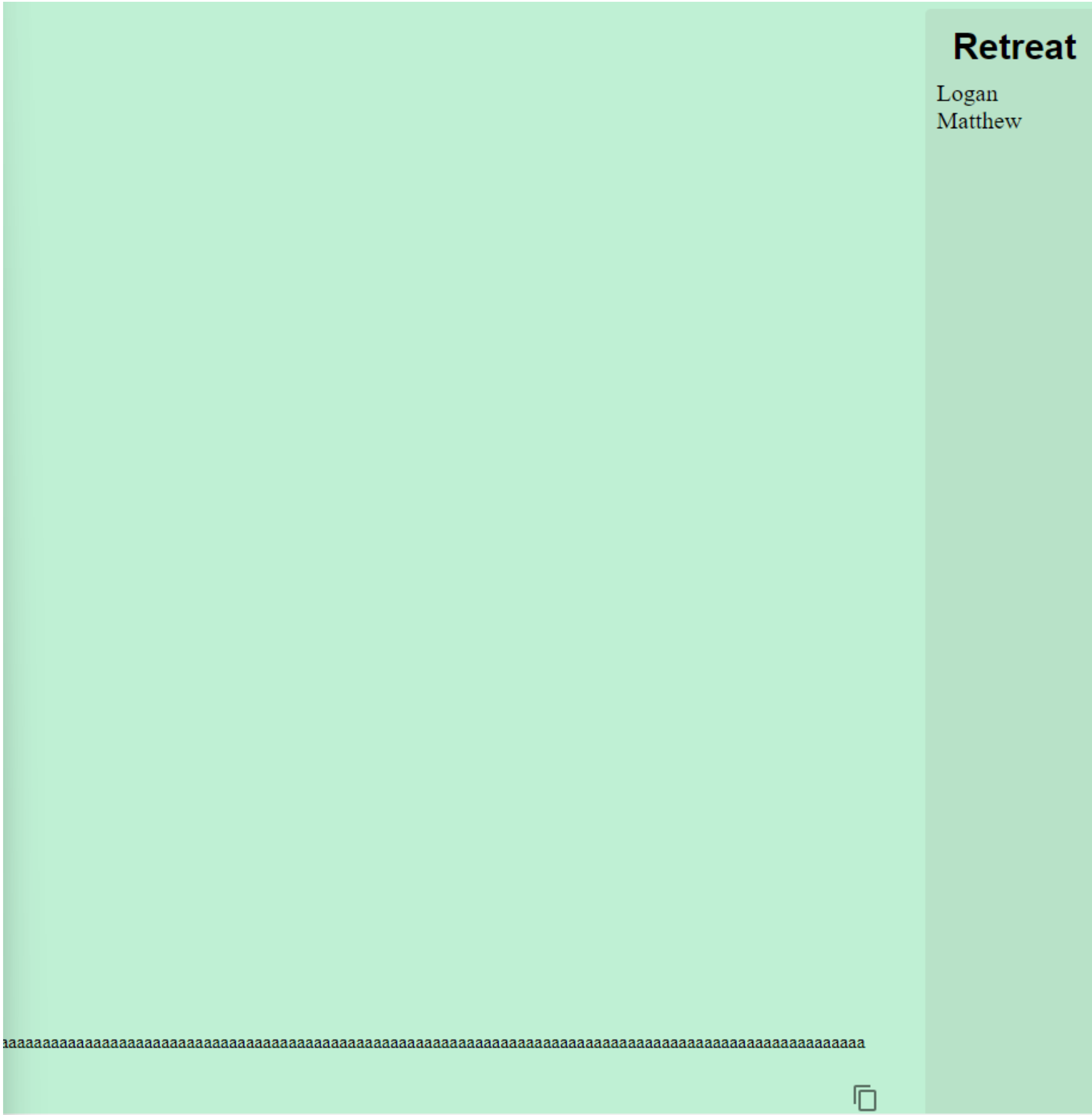
How: Send a request that the server does not understand (but is able to decrypt)

For this method, we click the second ‘send’ button and observe the server behavior.

We can see that just like last time, the server catches the error and logs a message. However, the server does not crash and continues to listen for incoming requests. In fact, you can click the send buttons multiple times and then click the ‘home’ button and continue on to create your own chat rooms. This demonstrates that the server continues to work even under unexpected circumstances.

How: Send a very long request (can encryption handle long messages appropriately)

For this test we create a room called “test”. We have two clients in the room “Logan” and “Matthew”. “Logan” will send a message that contains 1000 characters which is considered a long message. We will observe the behavior on the server and also whether the message is broadcasted. If the server goes down, this is an availability vulnerability, if the message is not transmitted, it doesn’t indicate an availability vulnerability because the vulnerability only affects the message sender and is triggered by the message sender.



The message is displayed on the client. Let's check the server.

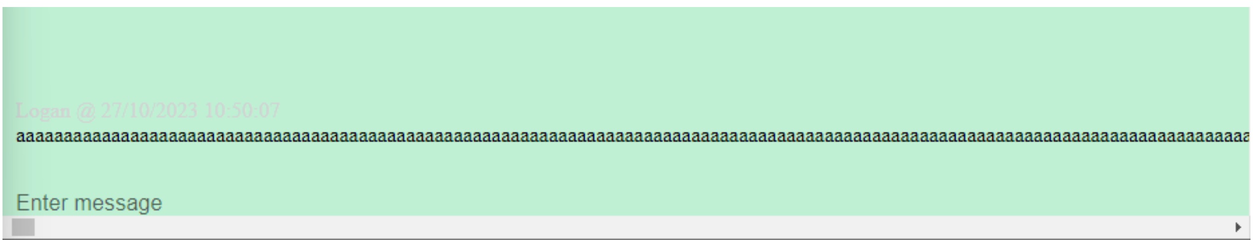

```
===== BROADCAST ENCRYPTED RESPONSE =====
{'type': 'message', 'verb': 'get', 'content': {'roomid': '38de4280-f2b3-4fb5-88cf-3a033915b24b', 'header': 'Logan @ 27/10/2023 10:47:24', 'message
': 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'}}

[{'username': 'Logan', 'socket': <websockets.legacy.server.WebSocketServerProtocol object at 0x000001E76EEC4A10>, 'rooms': [], 'userid': '70d3b7ce
-e360-4441-a5a9-eff99330da90', 'symmetric': b'1tHL_N2DmbO9wt-h9LVY6dV37xrVEcNkxY53gA3mB4Y='}, {'username': 'Matthew', 'socket': <websockets.legacy
.server.WebSocketServerProtocol object at 0x000001E76EC6F890>, 'rooms': [], 'userid': '713ef9f5-b99b-4a38-8c74-9696dcfcf317', 'symmetric': b'5HT5e
GT_Rf5tuYwATiZ6o83aG9V1WaXiX99TvE78lM4='}]
```

The server was able to parse and transmit the message to “Logan” and “Matthew”. Let’s test 10,000 characters.

Example of encrypted message on server:

pMp4iv79H-WH1yxFRpa2gyiCUY9SYHCwuoMkZuG7VHhITgHS9nGc3ZvaJSxFKLmtZycIWsL2YW1CaI-myOOcCGVnLdjoaVq9PWwLfmzdFBEGBxw547tM9m1YXI5_cRkWXa9_PWg5yzDx5O0nge
jnz5svbI3dPSD4JJYCm4nhN04kl6nmV_pRZ6F8ZUehfkq9T5qhoenxI_SNvmcwQK2UExUIPqnHtCn6maTXCfXeuzC5k_9evovahlvMord6ziTQWGXHMP1k8gYqsJDMQtEKQJsK3FUE-26xtizP
L5KqkB5WvfHCE_NIPJPgl-3aVANmSsuhi8U1CBkqLEWJLNjqYbU11_lwnG9bNrGV2614D7WjhqEp3EPtSRuCwJeUD0lD_720gDe_6lNPiAvuAIXu9J04c_Uwfye0rzA1JfdSOMpHvWEEdWSflz
0OULIR76Ry1VM2LuLS6Y3IhVU6rN6Hk3-bmRNkpnbH2xaYkuEJdkcdMXBhDYZ75ZFTyXVvUhyQw9elaNSpm82eIxK1Leox6jFt6ec_WhRQq3_jzPyh6U9D4SxtEPMZvCeUq3--At5opwqV05cO
EO4tQD6OctVGrA3GSrwqWOK66kXM-OflpU8v32lBlZwjp49wlBygK76OtGqI8kT9Z8vF3_rH3v0qgpf-KMmCuv5zGjCM34xloujSHnAoagcRp49N1umqBmmmbIRkGSP9WQNmkJTWuZITClLncA
zBMA-NtRZbsQPUhQ_C0-5JdOjArNxw_c_VT7SmZg0cAVRq7IdNA6r47tDmhHW-SfjKFyCtisFa-KyxmGXUSR9T97qKo2dVEkT9P-V0vr-ZlhZy3qahcW7LBJVWBd7cq0Z0BfBLkr0dVlSxnNPs
zHLsBKSY2DsvlmsFtaI66yW_e0My7_84fH1jeFVsISPVIwQuCg1uTQZEvRaOFmITmCLMeYMcL_fejkaW2QQO9ApKGzTU99RT1eoi4_OnW7ohOpHtMMzX5_oY3rUyB3u5dFSlccQkk97hKUapg4
Wq6SmWNr1SvvBByxHK6EeCLkWRR_6vQzS2_SIkDhqIqvqAPRVU2RnPCqf6CfOomkChv7QQDqLYQj1K8AKRmg7LPbnWc32LmIT41fO-haSUJ9knqbcbMnckGMgxRwXKQp4CR0TTJs253DlcSNTc
X4QmPMm7aYIG5U9Lcl92yvmHRumrF-tTgoYMCud_sT8FCdbLh8R6pH42-TtzoiqnC5UMJ8v-3owBsesz247mhx9aeKwqLlzjJHi35rz2mBLKtWlMyMM1_OET3DCcnQOmR_T4WbatB5baz87R67
vjCaTvIQUN3CeD_wOR7n50mguZgsFwQi_fapUXuGD0UkE3BVYmWoG2isZt50xU_rs984eq2jZzD8yP4Pe1AqXuq9o0xuCY941tGFzqV5Do-IxC5jIy30aFl8QY1J5vNwlaTFValh24AwWB_rGL
jV64PsVGErekTPDZph6_u2rAbf5VaTGdPW0wOpGCQavdm0hzkIPSWw3pEPE3bJxsrDiFgbAHivYJwfaSk5pvreLhFChZGDIuPqH9ZA0RgR4MqBLqQ2FpEMnYZUSIPAUzGC491XeA5i-mrgNDBK
gPnIH7qLLu1RuHKUkvCPUdqJboWqlkDiM_5tOgjn28e1hPZWGDq5zOc_6peBDmdb23PYKV28JOqExAZHUz43PI_fLtB--qgCWYLEM-EEPcA7Ed4_C8-7jw_bhANkH1W4dGl1DRQrUrVP3fm_xy
U7IsFRNTi9mU0xJzQlx2v-LculLQ1EwvowRaLT-cEOUjY5ZX_0q87UldHFAhzpzVsk59_mmU-iV5v2yBWjSveq7-V5zxuNoeRyoi3ZFDZLvCjC_u98uNPfVqjkoxqGSVE_Bh5LucM1rKt_ZwBA
-8t7dvwtM0tPwonw4m7B-Sb6Ox8Cxn0d3EwtR3PNp4GZd5M8vLWhOJ9Akr3dBpdJZagJkbClrG5YfQaUAcfabwIGo0Yx6gdy9mnGghiv2fBdx6XVxxIa2rjjYsapyOeGnvI2JUfFSnhxwJDlpy
rij8PTQxv02iqR2ixJUTg6K-HAJSlcK0b-3ASKNc0-qUjZdhGOebEPXFhltLd4h_Qs1lv1WC0VbVaQV0hYUHWwByf-vIYPTof6N5GijhyXd_uKMfgj7vXWVfQHgqi37Dqt98Abyl4djVI4MDa_
bzyCmoDLPbI0BQ2aEWA_5GALqDhSsLbwhlym01CIwWIJMfUFRvKExyFwVuHjIin04sYFSAJJqDylqAmErshZ_cMQ2BlEyAqVsS_CL_8CnHUDjA6YI1N4SwCcotzKoHp-Ml20KcfUtwHtxVfd1k
QLQzrK4DUmEL3-8leThstZYYuRTrEWTt2o7zKhnDrPntt2te6xF7M7Ud2Djzuy5igkuP2cBzJpBuSCnwZq1GVaz3e6gk-7zqBzHINeEjJEOgq0s6xBxm_AeRoVt7g_ATyhLL6cDoTU4irrE6y0
ShK7oPqlP_yR8kV-57C8liMByhR87rthrXkmCx3tew2uEiAmdTp7wMGXTnKE0k0CQ1CFwmlN3Qlfr1-1gYFM5VkfwrPVAy1IeJJTpBijn_FdM36rUymF89hfjVQgUtnIs47fWF-wNzBtE9eDtq

What the client sees:



In both cases, we see that the message was successfully transmitted. This test passes - the encryption algorithm is able to handle large messages.

Goal: Server can detect and respond to when a message has failed to reach the client ❌
METRIC: ACK ratio
How: modify client to arbitrarily ACK a message from the server or not

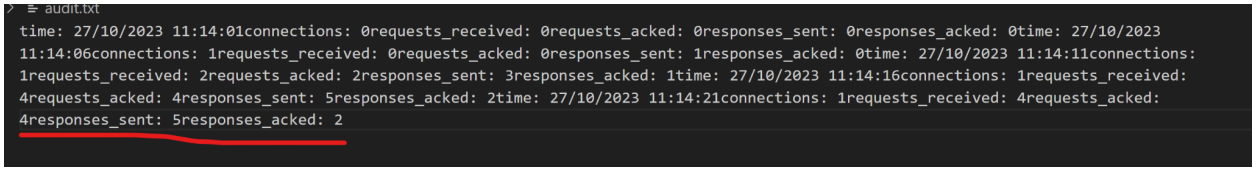
In the normal scenario we expect the ACK ratio for server responses to be 100%. To test this, we login, create a room, and send two messages. Then we compare the responses_sent to the responses_acked for the final time stamp in the audit.txt file.

```
time: 27/10/2023 10:56:24connections: 0requests_received: 0requests_acked: 0responses_sent: 0responses_acked: 0time: 27/10/2023
10:56:29connections: 0requests_received: 0requests_acked: 0responses_sent: 0responses_acked: 0time: 27/10/2023 10:56:34connections:
0requests_received: 0requests_acked: 0responses_sent: 0responses_acked: 0time: 27/10/2023 10:56:39connections: 1requests_received:
0requests_acked: 0responses_sent: 1responses_acked: 0time: 27/10/2023 10:56:44connections: 1requests_received: 0requests_acked:
0responses_sent: 1responses_acked: 0time: 27/10/2023 10:56:49connections: 1requests_received: 0requests_acked: 0responses_sent:
1responses_acked: 0time: 27/10/2023 10:56:54connections: 1requests_received: 2requests_acked: 2responses_sent: 3responses_acked: 2time: 27/10/
2023 10:56:59connections: 1requests_received: 4requests_acked: 4responses_sent: 5responses_acked: 4time: 27/10/2023 10:57:04connections:
1requests_received: 4requests_acked: 4responses_sent: 5responses_acked: 4
```

Notice that the ratio is 80%. This is because the client does not acknowledge the first response by the server (which is the response that contains the symmetric key the client should use when talking to the server). So in reality, for the messages that are supposed to be ACKed we have 100% ACK rate. Next, we drop every other ACK that the client is supposed to send. Ideally, if

there is a resolution technique, the ACKs would be 4 like before. However, if not then they will be 2. This scenario symbolizes an attacker sitting in between the client and server selectively dropping messages.

Code for dropping messages found in:
https://github.com/LoganKloft/LMMT-SecureDove/blob/main/client/src/components/Hallway.jsx



We can see that the responses are half the amount as before. This test fails, the server is not able to respond to missing messages.

Goal: Server maintains stability when receiving too many websocket requests
How: connect a bunch of websockets to the server

To test the server for stability it needs to have a lot of clients connected or connecting, so for this test we connected 22 clients to start and placed them all in a room and sent a few messages to ensure everything was working.

This is a good amount of sockets for the server to handle so we next added client “test23” to see if the server would accept this client. It did and we added this client to the room to see if the server would also place the client in the room, which it did.

When a program was written to automate the process an error of “timed out during handshake” occurred meaning the process would have to be done manually. Which as stated above still wouldn’t have errors handling the incoming traffic.

Improved Security Plan

During the break process we observed that many of our goals withstood the tests. However, we also saw that not all security goals were met. There were three main issues. The first is that there is a window of opportunity during the fernet symmetric key transfer where the key is public. The second is that there are no defenses against users abusing the name system. A user can have the same name as another user or even have no name at all, making them silent observers in a room. The final issue is the lack of robustness for retransmitting messages that were never acknowledged by either the server or client.

Due to the present vulnerabilities in our application, we need to discuss new security plan goals and how we will address them. The following details three goals that are either new, or are an improvement to a previous goal.

Goal: Key sharing is performed such that all keys involved are not publicly discoverable. Note that this only applies to the main branch that uses fernet encryption. The asymmetric branch, which encrypts chat messages, does not share this problem.

Solution: Use private key encryption to share the symmetric key. This can be overlaid over the current process. The asymmetric branch contains a solution for key sharing that can be reused on the main branch.

Goal: Users are not able to impersonate another user or covertly observe a chatroom while inside of the chatroom.

Solution: Usernames will be limited to a minimum and maximum size. Usernames will be limited to the available character set. Usernames will be unique. These requirements must be verified on the server at the least, and preferably also on the client side for instant feedback. Creating a new user must be an atomic operation on the PROFILES object to prevent race conditions.

Goal: Messages that are ‘dropped’ will have a resolution mechanism for attempting to ensure messages are reliably transmitted.

Solution: The client and server will store a list of messages they send. Each message will contain a unique identifier. When the client or server receives a message, they will send an ACK for that message in addition to storing the message id in a container of received messages. When the client or server receives an ACK they will remove the stored message from the message container. At a particular interval, the client and server will check messages that they have not received ACKs for and rebroadcast the message. If a client or server receives a duplicate message (contains the same id) they will ACK the message but otherwise not act on it (since it is assumed that the client or server had previously acted on it). There is an opportunity to configure how many times a message should be re-broadcasted before discarding it.