Peer Review - Isaac Joynes

October 12, 2024

Overview

The implementation submitted is well documented and contains clear instructions on how to use the chat application. It also has a pleasant user experience which makes it simple to work with and test. However, it contains numerous vulnerabilities which are discussed below.

Testing

To test the implementation, I connected a user to server 1 (preset 1) and a user to server 2 (preset 2) and tested sending messages between them and publically, uploading files and having both users download the files, and performed some packet sniffing using wireshark.

Discovered Vulnerabilities/Bugs

Invalid JSON handling on server

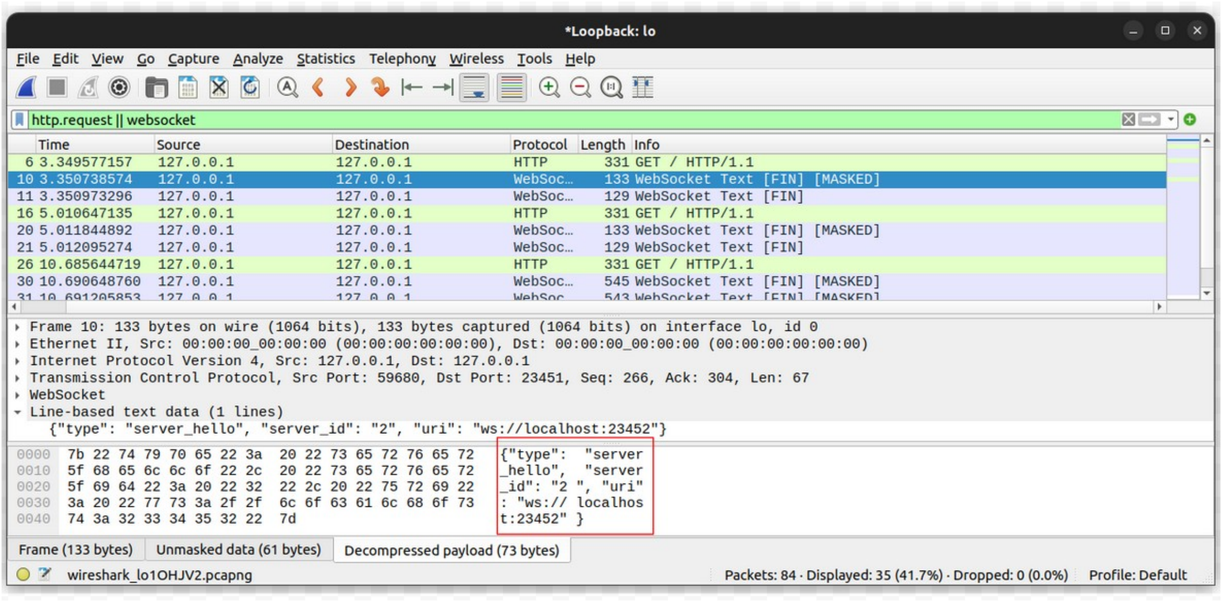
If a malicious user is able to determine that JSON messages are being expected by the server, they can intercept and disform packets to cause the server to crash as it will not be able to parse the malformed JSON message. If the “type” key, which is expected in every message is not present, then the server will also crash as it will be unable to access it. This can be abused to perform DOS attacks and cripple the neighborhood network as it can also affect messages bound for other servers, subsequently crashing them. To patch this vulnerability, try and catch blocks need to be implemented to be able to handle receiving malformed JSON messages or ones that don’t contain the “type” key.

Private key stored unencrypted

If a user’s file system is compromised, their private keys will also be compromised as they are stored in plaintext. They can then be loaded to impersonate another user and send messages on their behalf or upload malicious files using their name. To patch this vulnerability, some form of strong encryption should be used to encrypt the keys when stored.

Insecure Websockets leading to Packet Sniffing Messages

Since the current implementation has not secured websockets using TLS, messages that are sent between servers can be intercepted in plaintext as I have done below.


*Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

http.request || websocket

```
   Time            Source            Destination       Protocol  Length Info
 6 3.349577157     127.0.0.1         127.0.0.1         HTTP         331 GET / HTTP/1.1
10 3.350738574     127.0.0.1         127.0.0.1         WebSoc…      133 WebSocket Text [FIN] [MASKED]
11 3.350973296     127.0.0.1         127.0.0.1         WebSoc…      129 WebSocket Text [FIN]
16 5.010647135     127.0.0.1         127.0.0.1         HTTP         331 GET / HTTP/1.1
20 5.011844892     127.0.0.1         127.0.0.1         WebSoc…      133 WebSocket Text [FIN] [MASKED]
21 5.012095274     127.0.0.1         127.0.0.1         WebSoc…      129 WebSocket Text [FIN]
26 10.685644719    127.0.0.1         127.0.0.1         HTTP         331 GET / HTTP/1.1
30 10.690648760    127.0.0.1         127.0.0.1         WebSoc…      545 WebSocket Text [FIN] [MASKED]
31 10.691205853    127.0.0.1         127.0.0.1         WebSoc…      543 WebSocket Text [FIN] [MASKED]
```

```
› Frame 10: 133 bytes on wire (1064 bits), 133 bytes captured (1064 bits) on interface lo, id 0
› Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
› Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
› Transmission Control Protocol, Src Port: 59680, Dst Port: 23451, Seq: 266, Ack: 304, Len: 67
› WebSocket
⌄ Line-based text data (1 lines)
    {"type": "server_hello", "server_id": "2", "uri": "ws://localhost:23452"}
```

```
0000  7b 22 74 79 70 65 22 3a  20 22 73 65 72 76 65 72   {"type":  "server
0010  5f 68 65 6c 6c 6f 22 2c  20 22 73 65 72 76 65 72   _hello",  "server
0020  5f 69 64 22 3a 20 22 32  22 2c 20 22 75 72 69 22   _id": "2 ", "uri"
0030  3a 20 22 77 73 3a 2f 2f  6c 6f 63 61 6c 68 6f 73   : "ws:// localhos
0040  74 3a 32 33 34 35 32 22  7d                        t:23452" }
```

Frame (133 bytes) | Unmasked data (61 bytes) | Decompressed payload (73 bytes)

wireshark_lo1OHJV2.pcapng   Packets: 84 · Displayed: 35 (41.7%) · Dropped: 0 (0.0%)   Profile: Default

Knowing the structure of these messages and their contents as well as the IP and port of the server can allow an attacker to pretend to be a server (which leads to another vulnerability), or pretend to be another user (by extracting public keys). It can also allow attackers to intercept and modify packets to disrupt communications or modify messages between users. The best way to patch this vulnerability is to secure the websocket connections using TLS.

No authentication of users using RSA signatures

There is no presence of signatures in the implementation and the messages being sent. This means that if anybody has the public key of another user, they can log on as them and send messages on their behalf. When combined with the packet sniffing mentioned above, this allows malicious users to pretend to be servers by sending server_hello messages, where they could then forward fake messages or poison the client lists with fake users. However, patching the packet sniffing vulnerability still means that users can impersonate others using public keys. This vulnerability needs to be patched by generating RSA signatures using private keys, and using public keys to verify users.

Counter not implemented

The counter value which is a part of the Olaf Neighbourhood protocol. This allows attackers to sniff and store sent JSON messages, and replay them at later dates to pretend to send a message as another user. Implementing the signature system alongside this counter will patch this vulnerability as the signature changes depending on the counter and the signature is generated using a user's private key. This means an attacker will not be able to impersonate another user and those sniffed messages will then become invalid.

File upload vulnerabilities

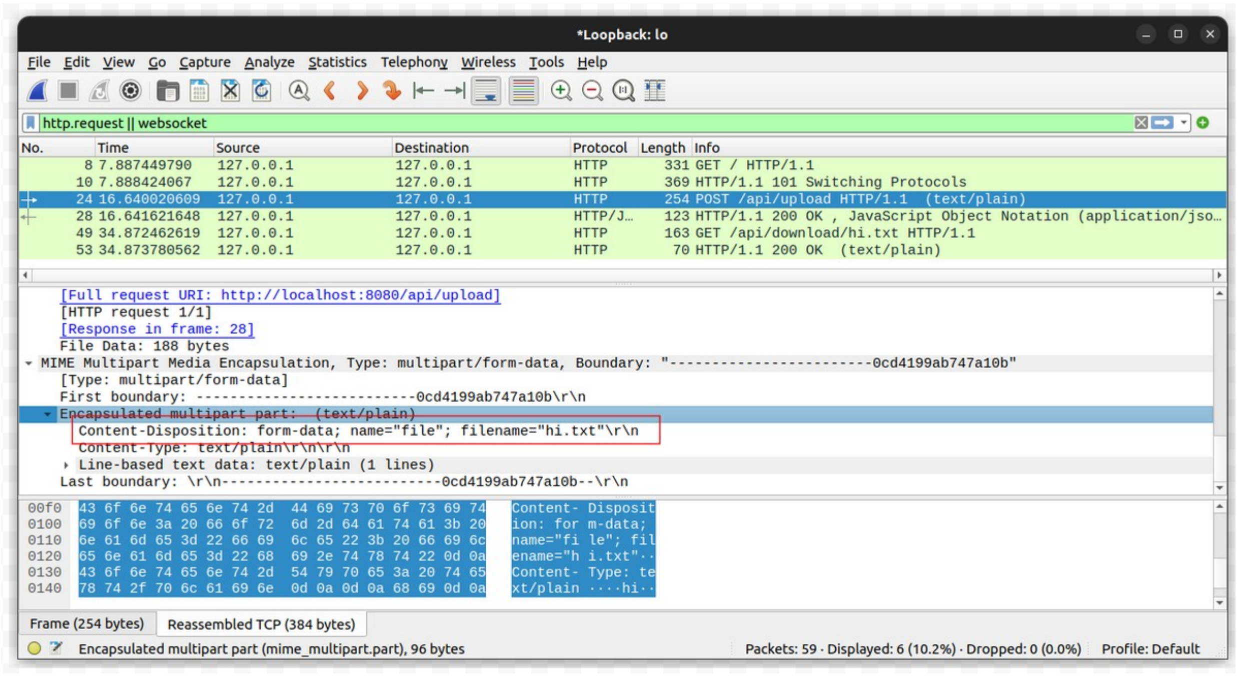
No file restrictions

The file upload feature integrated into the application creates numerous vulnerabilities for user's of the application as well as the server. There are no restrictions on the file types that can be uploaded, meaning executable files or webshells could be uploaded, which if the server is later configured to run executable code or is breached from another vector, then those files can become a vulnerability. As well, if an extremely large file is uploaded many times, it could consume many of the server resources and deny the service for other users. To patch this vulnerability, only certain file types need to be allowed, and files need to be analyzed for malware before hosting them. As well, there needs to be a limit on file size.

Redirection and potential XSS

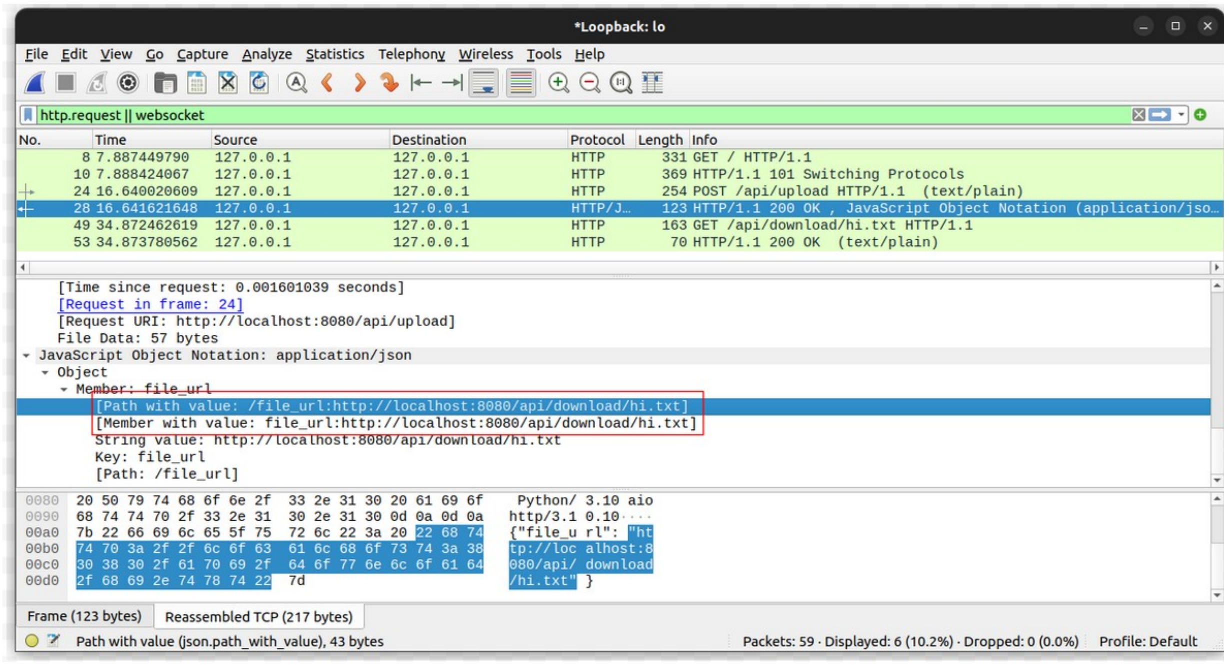
During my testing, I have managed to upload an index.html file which redirects a user to google.com. This proves that if users were to access these malicious files, they could be redirected to a malicious web page and that could lead to session hijacking, information being stolen, scripts being run or malware being distributed. To patch this vulnerability, HTML and executable files could be banned from the file upload system, and HTML content could be sanitized by the server before hosting the file. A content security policy should also be implemented to restrict where scripts can be loaded from and executed.

No access controls on HTTP server


*Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
http.request || websocket

```
No.  Time          Source     Destination  Protocol Length Info
   8 7.887449790   127.0.0.1  127.0.0.1    HTTP     331 GET / HTTP/1.1
  10 7.888424067   127.0.0.1  127.0.0.1    HTTP     369 HTTP/1.1 101 Switching Protocols
  24 16.640020609  127.0.0.1  127.0.0.1    HTTP     254 POST /api/upload HTTP/1.1  (text/plain)
  28 16.641621648  127.0.0.1  127.0.0.1    HTTP/J…  123 HTTP/1.1 200 OK , JavaScript Object Notation (application/jso…
  49 34.872462619  127.0.0.1  127.0.0.1    HTTP     163 GET /api/download/hi.txt HTTP/1.1
  53 34.873780562  127.0.0.1  127.0.0.1    HTTP      70 HTTP/1.1 200 OK  (text/plain)

    [Full request URI: http://localhost:8080/api/upload]
    [HTTP request 1/1]
    [Response in frame: 28]
    File Data: 188 bytes
  MIME Multipart Media Encapsulation, Type: multipart/form-data, Boundary: "------------------------0cd4199ab747a10b"
    [Type: multipart/form-data]
    First boundary: --------------------------0cd4199ab747a10b\r\n
    Encapsulated multipart part:  (text/plain)
      Content-Disposition: form-data; name="file"; filename="hi.txt"\r\n
      Content-Type: text/plain\r\n\r\n
      Line-based text data: text/plain (1 lines)
    Last boundary: \r\n--------------------------0cd4199ab747a10b--\r\n

00f0  43 6f 6e 74 65 6e 74 2d  44 69 73 70 6f 73 69 74   Content- Disposit
0100  69 6f 6e 3a 20 66 6f 72  6d 2d 64 61 74 61 3b 20   ion: for m-data;
0110  6e 61 6d 65 3d 22 66 69  6c 65 22 3b 20 66 69 6c   name="fi le"; fil
0120  65 6e 61 6d 65 3d 22 68  69 2e 74 78 74 22 0d 0a   ename="h i.txt"··
0130  43 6f 6e 74 65 6e 74 2d  54 79 70 65 3a 20 74 65   Content- Type: te
0140  78 74 2f 70 6c 61 69 6e  0d 0a 0d 0a 68 69 0d 0a   xt/plain ····hi··

Frame (254 bytes)  Reassembled TCP (384 bytes)
Encapsulated multipart part (mime_multipart.part), 96 bytes    Packets: 59 · Displayed: 6 (10.2%) · Dropped: 0 (0.0%)    Profile: Default
```

With minimal difficulty, I was able to sniff an HTTP POST request and determine the name of the file being uploaded. If a user uploads this file with the intention of it only being for some users, this file is vulnerable to being downloaded by other users.

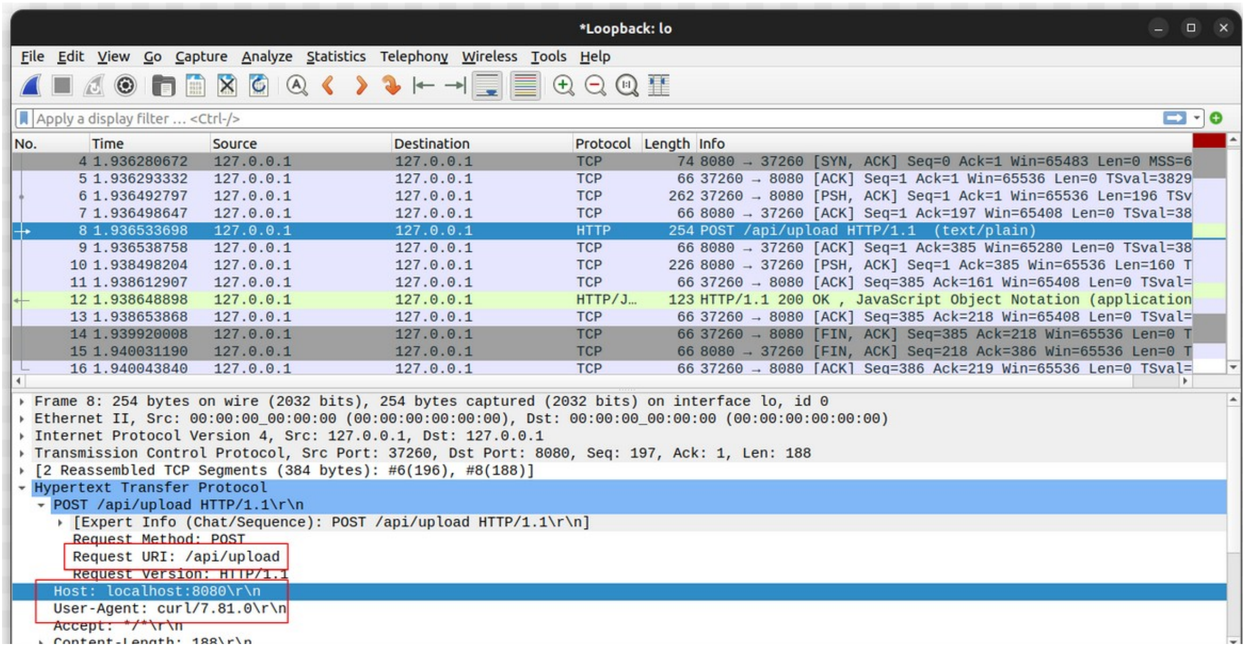

*Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
http.request || websocket

```
No.  Time          Source     Destination  Protocol Length Info
   8 7.887449790   127.0.0.1  127.0.0.1    HTTP     331 GET / HTTP/1.1
  10 7.888424067   127.0.0.1  127.0.0.1    HTTP     369 HTTP/1.1 101 Switching Protocols
  24 16.640020609  127.0.0.1  127.0.0.1    HTTP     254 POST /api/upload HTTP/1.1  (text/plain)
  28 16.641621648  127.0.0.1  127.0.0.1    HTTP/J…  123 HTTP/1.1 200 OK , JavaScript Object Notation (application/jso…
  49 34.872462619  127.0.0.1  127.0.0.1    HTTP     163 GET /api/download/hi.txt HTTP/1.1
  53 34.873780562  127.0.0.1  127.0.0.1    HTTP      70 HTTP/1.1 200 OK  (text/plain)

    [Time since request: 0.001601039 seconds]
    [Request in frame: 24]
    [Request URI: http://localhost:8080/api/upload]
    File Data: 57 bytes
  JavaScript Object Notation: application/json
    Object
      Member: file_url
        [Path with value: /file_url:http://localhost:8080/api/download/hi.txt]
        [Member with value: file_url:http://localhost:8080/api/download/hi.txt]
        String value: http://localhost:8080/api/download/hi.txt
        Key: file_url
        [Path: /file_url]

0080  20 50 79 74 68 6f 6e 2f  33 2e 31 30 20 61 69 6f    Python/ 3.10 aio
0090  68 74 74 70 2f 33 2e 31  30 2e 31 30 0d 0a 0d 0a   http/3.1 0.10····
00a0  7b 22 66 69 6c 65 5f 75  72 6c 22 3a 20 22 68 74   {"file_u rl": "ht
00b0  74 70 3a 2f 2f 6c 6f 63  61 6c 68 6f 73 74 3a 38   tp://loc alhost:8
00c0  30 38 30 2f 61 70 69 2f  64 6f 77 6e 6c 6f 61 64   080/api/ download
00d0  2f 68 69 2e 74 78 74 22  7d                        /hi.txt" }

Frame (123 bytes)  Reassembled TCP (217 bytes)
Path with value (json.path_with_value), 43 bytes    Packets: 59 · Displayed: 6 (10.2%) · Dropped: 0 (0.0%)    Profile: Default
```

I was also able to determine the url that the server is hosting the file on by sniffing the server’s packets, making it even easier to download the file. If the intention is for files to be accessible by all users then this is not an issue, but if the intention is for it to be accessible for some users, then some access controls need to be implemented to prevent unwanted users from having access to the file. As well, some user education into how to safely use the application would prevent users from having files unintentionally accessed, and upgrading to TLS will prevent information being uncovered from sniffing packets, but does not solve the core vulnerability being mentioned here.

Packet Sniffing Curl Command

If the user does not have access to the source code of the program, they will not know that files are uploaded using a curl command (though they can likely guess that since the server is a HTTP server). Since the file sharing server is an HTTP server, it means communication between the file sharing server and the client is unencrypted and can be sniffed with wireshark.


*Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
No.   Time         Source      Destination  Protocol  Length Info
    4 1.936280672  127.0.0.1   127.0.0.1    TCP         74 8080 → 37260 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=6
    5 1.936293332  127.0.0.1   127.0.0.1    TCP         66 37260 → 8080 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=3829
    6 1.936492797  127.0.0.1   127.0.0.1    TCP        262 37260 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=196 TSv
    7 1.936498647  127.0.0.1   127.0.0.1    TCP         66 8080 → 37260 [ACK] Seq=1 Ack=197 Win=65408 Len=0 TSval=38
    8 1.936533698  127.0.0.1   127.0.0.1    HTTP       254 POST /api/upload HTTP/1.1  (text/plain)
    9 1.936538758  127.0.0.1   127.0.0.1    TCP         66 8080 → 37260 [ACK] Seq=1 Ack=385 Win=65280 Len=0 TSval=38
   10 1.938498204  127.0.0.1   127.0.0.1    TCP        226 8080 → 37260 [PSH, ACK] Seq=1 Ack=385 Win=65536 Len=160 T
   11 1.938612907  127.0.0.1   127.0.0.1    TCP         66 37260 → 8080 [ACK] Seq=385 Ack=161 Win=65408 Len=0 TSval=
   12 1.938648898  127.0.0.1   127.0.0.1    HTTP/J…    123 HTTP/1.1 200 OK , JavaScript Object Notation (application
   13 1.938653868  127.0.0.1   127.0.0.1    TCP         66 37260 → 8080 [ACK] Seq=385 Ack=218 Win=65408 Len=0 TSval=
   14 1.939920008  127.0.0.1   127.0.0.1    TCP         66 37260 → 8080 [FIN, ACK] Seq=385 Ack=218 Win=65536 Len=0 T
   15 1.940031190  127.0.0.1   127.0.0.1    TCP         66 8080 → 37260 [FIN, ACK] Seq=218 Ack=386 Win=65536 Len=0 T
   16 1.940043840  127.0.0.1   127.0.0.1    TCP         66 37260 → 8080 [ACK] Seq=386 Ack=219 Win=65536 Len=0 TSval=

▸ Frame 8: 254 bytes on wire (2032 bits), 254 bytes captured (2032 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 37260, Dst Port: 8080, Seq: 197, Ack: 1, Len: 188
▸ [2 Reassembled TCP Segments (384 bytes): #6(196), #8(188)]
▾ Hypertext Transfer Protocol
  ▾ POST /api/upload HTTP/1.1\r\n
    ▸ [Expert Info (Chat/Sequence): POST /api/upload HTTP/1.1\r\n]
      Request Method: POST
      Request URI: /api/upload
      Request Version: HTTP/1.1
    Host: localhost:8080\r\n
    User-Agent: curl/7.81.0\r\n
    Accept: */*\r\n
  ▸ Content-Length: 188\r\n
```

I was able to sniff the HTTP packet that sent the file where it can be seen the command that was used under the User-Agent section, the IP and Port of the file sharing server and the requested URI. If the file sharing server were made public, any file can be uploaded to the server through a curl command from another user, even if they are not logged in. This allows malicious files to be uploaded to the server or if checks were implemented on the client for the type of files being uploaded and their content, they would be entirely bypassed. To solve this vulnerability, both upgrading the file sharing server to HTTPS to prevent the packet sniffing and more importantly implementing a file processing mechanism that can reject certain file types and process file content to determine if it is malicious or not would patch this vulnerability.

Conclusion

This implementation is well documented and contains many of the features required for a chat messaging system, however, the vulnerabilities mentioned above prevent it from being a secure chat messaging system, and must be addressed before releasing a production build.