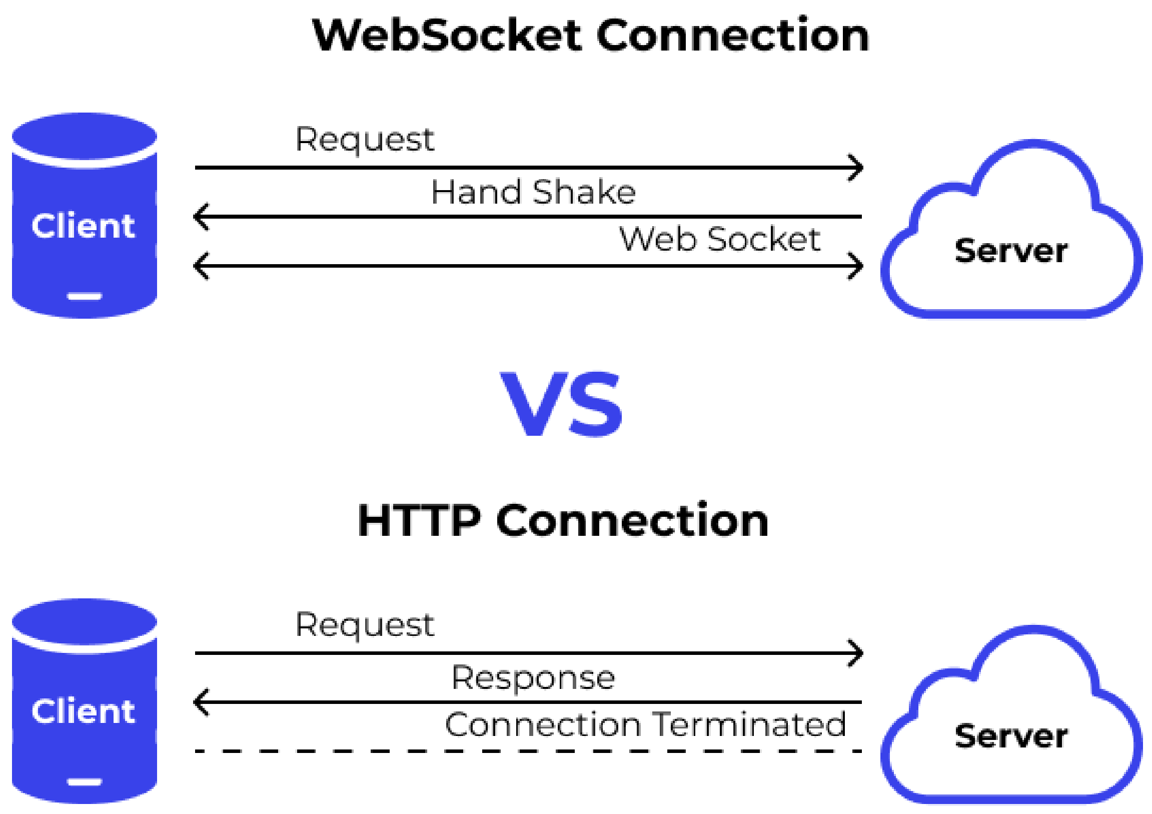
IDATT2104 - Datakom - Arbeidskrav 2

Report

WebSocket: theory and documentation

WebSocket is a duplex protocol which is utilized for client-server communication. It is considered to be bidirectional in nature, meaning that the data communication which occurs will always happen to and from the client and server parts respectively. The WebSocket protocol is an essential tool for server-client communication and is used in many areas of software development due to the benefits which it provides.

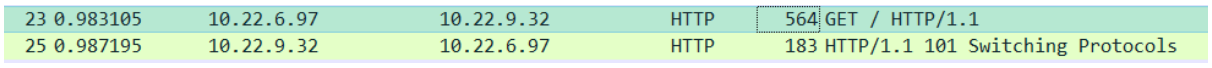
The main difference between WebSocket and HTTP is their directionality. The WebSocket protocol can essentially be considered as an upgrade or alteration of HTTP. While HTTP is primarily a unidirectional protocol functioning above the TCP protocol, WebSocket is bidirectional which means communication will occur from both ends - consequently making it the faster alternative. Additionally, the WebSocket connection will always remain persistent until one side chooses to terminate or cancel the connection whereas the HTTP protocol will sever the connection once a request has been completed. One can visualize the difference between the two like so:



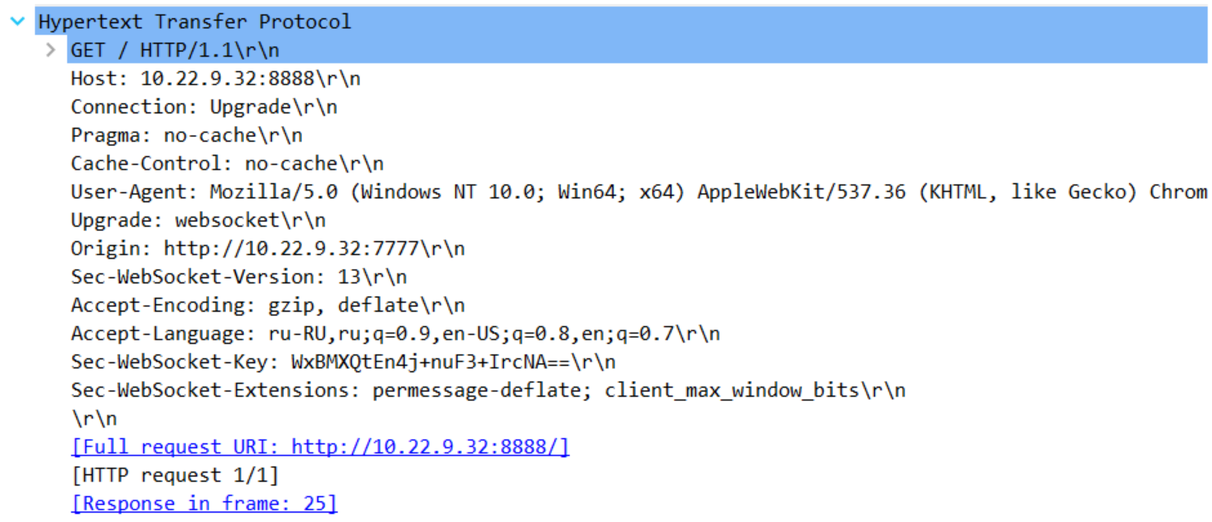
We can observe from the picture that the HTTP connection terminates immediately after the server responds to the request, but the WebSocket connection remains open to receive additional data until one of the sides chooses to disconnect, which in most cases would be the client unless an error occurs within the server. In order to be used, WebSocket actually requires assistance from the HTTP protocol to initiate a connection between a server and a client. Despite its reliance on HTTP, the difference between the two protocols is quite major which makes them most optimal in vastly different scenarios.

Websocket Security

At the beginning of a WebSocket connection, the client initiates a handshake with HTTP. On request there is a special header called "Sec-WebSocket-Key". It contains an encoded random key. This header is used for opening a handshake. Why do we need randomly generated code here? The answer is pretty simple, to prove that I received a valid opening Websocket handshake. Key specification is described in official documentation RFC 6455 11.3.1. Below is a demonstration of the a standard WebSocket client-server handshake which is performed when a client connects to the WebSocket server (10.22.9.32 being the IPv4 address of the server, while 10.22.6.97 is the IPv4 address of the client):


```
23 0.983105     10.22.6.97      10.22.9.32      HTTP     564 GET / HTTP/1.1
25 0.987195     10.22.9.32      10.22.6.97      HTTP     183 HTTP/1.1 101 Switching Protocols
```

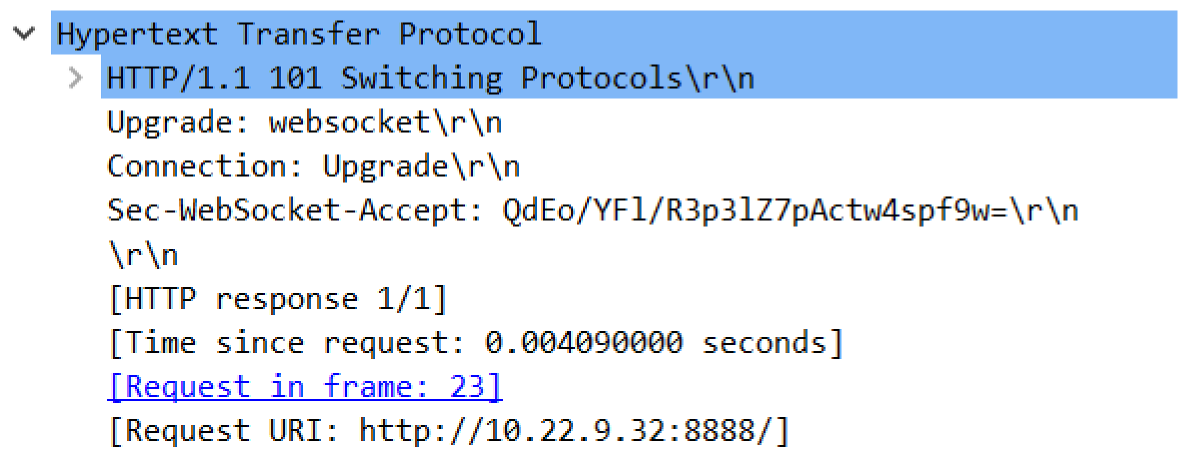
Here we can see the initiation of the handshake, which is distinguished by noticing the Sec-WebSocket-Key header:


```
Hypertext Transfer Protocol
  GET / HTTP/1.1\r\n
  Host: 10.22.9.32:8888\r\n
  Connection: Upgrade\r\n
  Pragma: no-cache\r\n
  Cache-Control: no-cache\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrom
  Upgrade: websocket\r\n
  Origin: http://10.22.9.32:7777\r\n
  Sec-WebSocket-Version: 13\r\n
  Accept-Encoding: gzip, deflate\r\n
  Accept-Language: ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7\r\n
  Sec-WebSocket-Key: WxBMXQtEn4j+nuF3+IrcNA==\r\n
  Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits\r\n
  \r\n
  [Full request URI: http://10.22.9.32:8888/]
  [HTTP request 1/1]
  [Response in frame: 25]
```

And server response with status code 101 Switching protocols.In response status we can see the “Sec-WebSocket-Accept” header. The first server takes the value from the request from the header Sec-WebSocket-Key. After it, the server concatenates the value with the GUID "258EAFA5-E914-47DA-95CA-C5AB0DC85B11" (Globally Unique Identifier which is described in RFC6455). Then it takes the SHA-1 hash of the combination and encodes it in base64.

```
const getAcceptKey = (key) => {
   return
base64encode.stringify((sha1(key.concat('258EAFA5-E914-47DA-95CA-C5AB0DC85B1
1'))));
}
```

Below is a demonstration of the server’s response to the client during a standard WebSocket handshake:



```
Hypertext Transfer Protocol
  HTTP/1.1 101 Switching Protocols\r\n
  Upgrade: websocket\r\n
  Connection: Upgrade\r\n
  Sec-WebSocket-Accept: QdEo/YFl/R3p3lZ7pActw4spf9w=\r\n
  \r\n
  [HTTP response 1/1]
  [Time since request: 0.004090000 seconds]
  [Request in frame: 23]
  [Request URI: http://10.22.9.32:8888/]
```

This header has almost the same mission in connection: confirm that the server is willing to initiate the Websocket connection. Described in RFC 6455 11.3.3

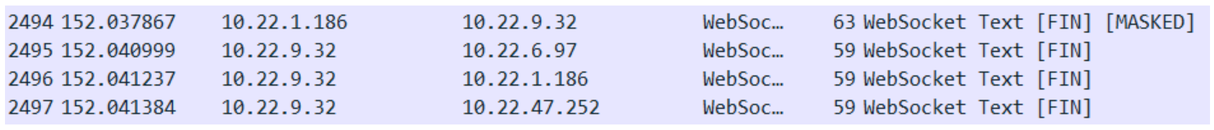
This header looks like it helps to achieve security on our server, but they are used only for establishing communication. Improper WebSocket establishing creates vulnerabilities that should be fixed.

It is also important to remember that the server is bidirectional and this can create a big danger for our site user. For example, we have a chat echo site that uses WebSocket and if a hacker attacks, all clients will be in danger. It can be different attacks for example : injections attacks. Here is an example of such an attack that can only send messages and with help of this message create many problems: `{"user":"attacker","message":"<img src=x onerror='alert(1337)'"}.` And that is not the only problem of improper

implementation. In the next topic we will discuss how to secure WebSocket server:

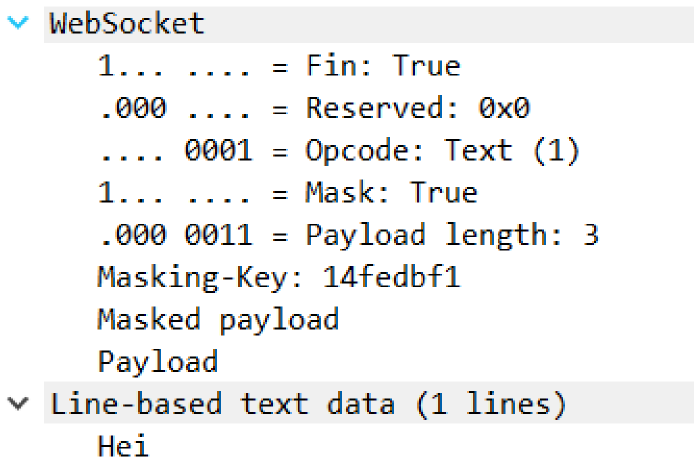
1. The Origin request header indicates the origin that caused the request. And we should verify this origin, and if it can not be trusted, we should reject the request.
2. It is always better to use CSRF protection on the server side. CSRF protection uses header `X-CSRF-Token`, a random token which is generated on the server-side. CSRF is validating user requests.
3. Using a WebSocket Secure. Protocol wss:// over normal ws:// is using SSL/TLS (special security technologies to protect the connection between server and client)
4. It is always reasonable to validate client input before sending it to other clients. It will help us to protect servers from running dangerous code.

Once the clients are connected to the server and have successfully upgraded their HTTP connection to the WebSocket protocol, every client is able to send messages to the server which the server will later on distribute to every other connected client. The initial message from the client is masked due to the protocol's requirement to prevent malicious scripts from the client which may attempt to perform a cache poisoning attack against intermediaries. It is also a way for us to distinguish between messages sent from the client to the server and from the server to the client. We can observe such a scenario in Wireshark using a WebSocket chat application:



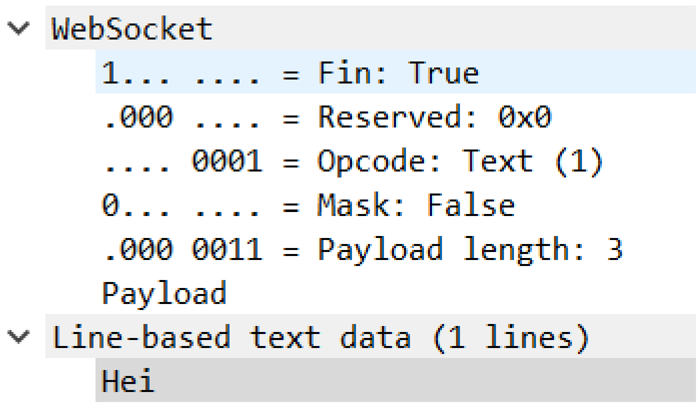
```
2494 152.037867   10.22.1.186    10.22.9.32     WebSoc…   63 WebSocket Text [FIN] [MASKED]
2495 152.040999   10.22.9.32     10.22.6.97     WebSoc…   59 WebSocket Text [FIN]
2496 152.041237   10.22.9.32     10.22.1.186    WebSoc…   59 WebSocket Text [FIN]
2497 152.041384   10.22.9.32     10.22.47.252   WebSoc…   59 WebSocket Text [FIN]
```

Taking a closer look at the client's message to the server, we can see the transmitted message and confirm that it is masked:



```
WebSocket
    1... .... = Fin: True
    .000 .... = Reserved: 0x0
    .... 0001 = Opcode: Text (1)
    1... .... = Mask: True
    .000 0011 = Payload length: 3
    Masking-Key: 14fedbf1
    Masked payload
    Payload
Line-based text data (1 lines)
    Hei
```

In our case, one of the clients chooses to send a specific message to all other clients. When analyzing the connection, we can see that the client's masked message is sent to the server first. After the server receives the message, it iterates through its list of clients and sends the received message to them. We can confirm that the message successfully goes through by analyzing the line-based text data which can be found inside of the WebSocket packet:



```
WebSocket
    1... .... = Fin: True
    .000 .... = Reserved: 0x0
    .... 0001 = Opcode: Text (1)
    0... .... = Mask: False
    .000 0011 = Payload length: 3
    Payload
Line-based text data (1 lines)
    Hei
```

This time, we can see that the masking is off since the message is being sent from the server to a client. It is no longer required as it is assumed that the server's messages will not be read or acted upon by intermediaries such as proxies and routers. While cache poisoning is rarely an issue nowadays, some older intermediaries can still be subject to a malicious attack due to being unaware of the WebSocket protocol and its inner workings. This may cause clear text which may resemble a HTTP request to be performed. It is therefore required that WebSocket utilizes masking when transmitting client message data.

Continuous Integration & Deployment

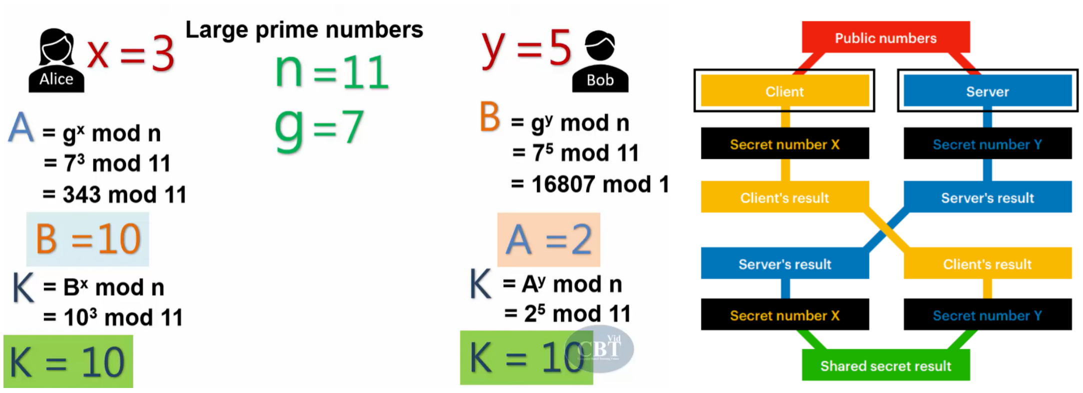
SSH (Secure SHell) is a network protocol that allows you to connect to a remote server and execute commands on it, upload files. A key feature of this protocol is encryption of transmitted information. It helps to ensure that hackers cannot catch the traffic between two connected devices. Then the client connects to the server using ssh. He can use a server like a local computer.

Establishing an SSH connection:

To initiate the connection you should run the following command from your client side: `ssh [username]@[server_ip_or_hostname]`.This command creates requests to the server and a session of encryption starts.

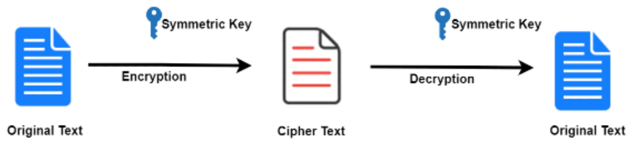
Right after sending a request to the server, the server in response sends a set of supported encryption protocols. After receiving a set of different protocols, the

client should compare them with his own available protocols and if there are matching protocols machines agree to use it to establish connection.
On the first connection attempt, the client compares his own private key to this server's public key. If the keys match, the client and the server agree to use symmetric encryption to communicate during SSH connection. To find secret key clients and servers use an asymmetrically encrypted process that is done by the Diffie-Hellman key exchange algorithm.
This picture below describes DH algorithm very good:

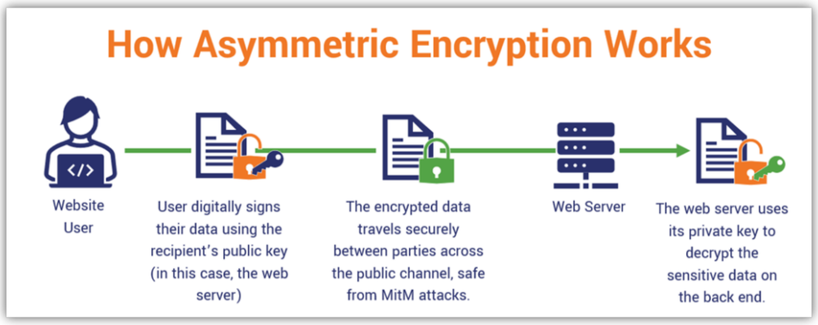


How we can see in our example secret key here is K=10. It is the same value for both machines, their secret key.
There is three different encryption types during the machines` communication:

1. Symmetric encryption : Generating a single key for encryption and decryption that both machines can use.



2. Asymmetric encryption: Generating two different keys public and private. These keys are related mathematically between each other.

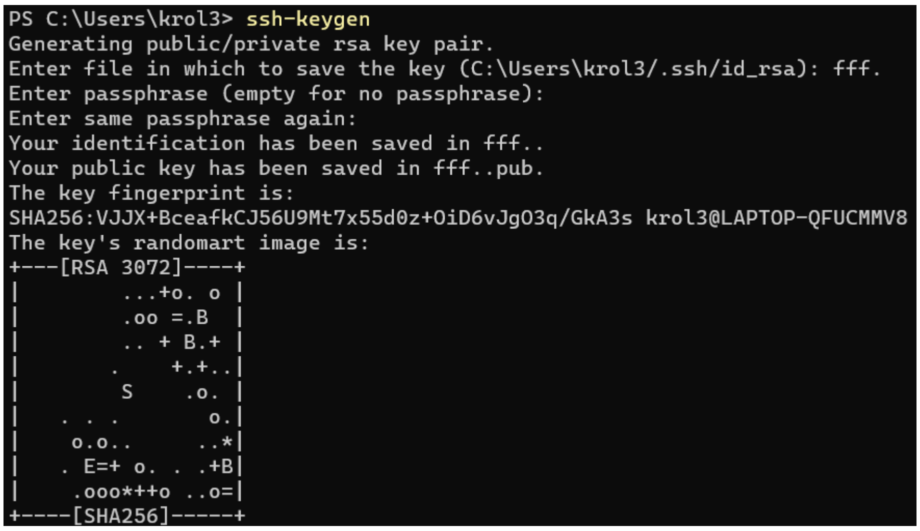

How Asymmetric Encryption Works
Website User
User digitally signs their data using the recipient's public key (in this case, the web server)
The encrypted data travels securely between parties across the public channel, safe from MitM attacks.
Web Server
The web server uses its private key to decrypt the sensitive data on the back end.

3. Hashing: SSH uses hashing to validate if data packets came from the right place. Hash algorithms which SSH is using are Message

Authentication Code (MAC) and Hashed Message Authentication Code (HMAC). The sending machine uses a data packet to create a unique hash string and send it with a data packet. And the receiving machine will verify it using the same algorithm and take data or corrupt them.

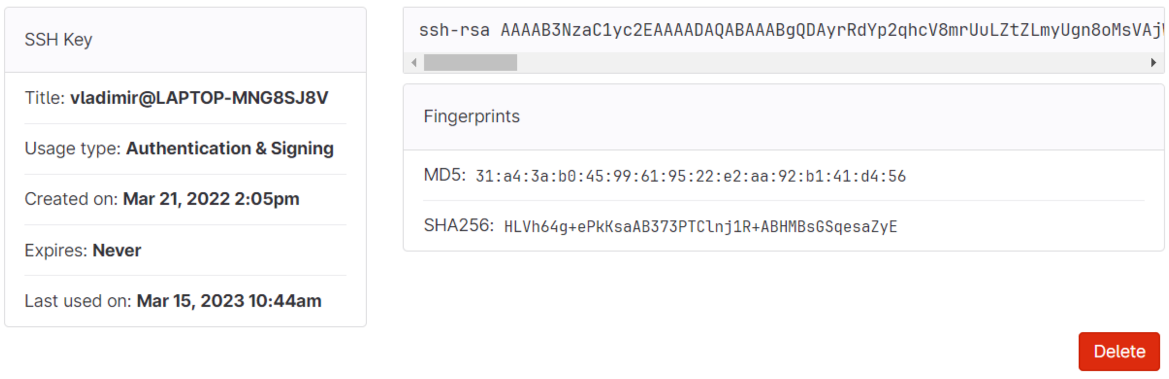
User Authentication

The two main methods to authenticate users are password and SSH keys. Each method has its own advantages and disadvantages. It is easy to handle a client's password, but each password's difficulty depends on people. On the other hand, have we asymmetrically encrypted SSH public-private key pairs. It should be generated, but not using people's imagination. It is very easy to generate such a key, since it simply requires you to type `ssh-keygen` in your Terminal.
Here is the example that i get as response:



```
PS C:\Users\krol3> ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (C:\Users\krol3/.ssh/id_rsa): fff.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in fff..
Your public key has been saved in fff..pub.
The key fingerprint is:
SHA256:VJJX+BceafkCJ56U9Mt7x55d0z+OiD6vJgO3q/GkA3s krol3@LAPTOP-QFUCMMV8
The key's randomart image is:
+---[RSA 3072]----+
|        ...+o. o |
|        .oo =.B  |
|        .. + B.+ |
|       .    +.+..|
|        S     .o. |
|    . . .       o.|
|     o.o..      ..*|
|    . E=+ o. . .+B|
|     .ooo*++o ..o=|
+----[SHA256]-----+
```

This function has many uses, such as being able to connect yourself to GitLab to perform continuous integration or deployment and to keep repositories up to date. In order to connect, we can simply register the key which we generated inside of GitLab which will consequently allow us to operations and work related to SSH:



SSH Key

Title: **vladimir@LAPTOP-MNG8SJ8V**

Usage type: **Authentication & Signing**

Created on: **Mar 21, 2022 2:05pm**

Expires: **Never**

Last used on: **Mar 15, 2023 10:44am**

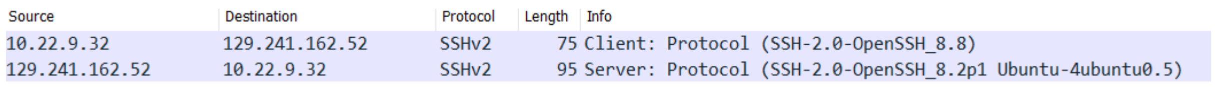
```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDAyrRdYp2qhcV8mrUuLZtZLmyUgn8oMsVAj
```

Fingerprints

MD5: `31:a4:3a:b0:45:99:61:95:22:e2:aa:92:b1:41:d4:56`

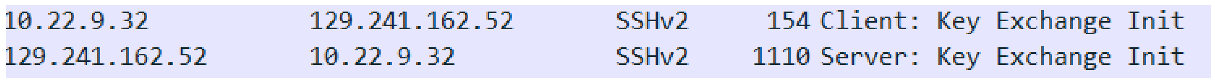
SHA256: `HLVh64g+ePkKsaAB373PTClnj1R+ABHMBsGSqesaZyE`

In order to put all of this theory about SSH into practice, we can observe how GitLab uses SSH in order to connect with us when we are working on a repository:



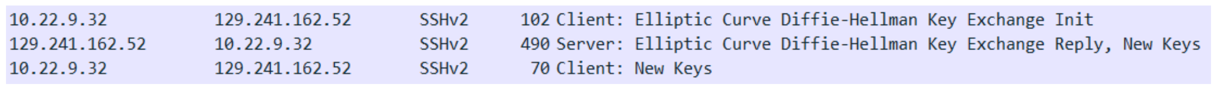
```
Source             Destination        Protocol  Length Info
10.22.9.32         129.241.162.52     SSHv2         75 Client: Protocol (SSH-2.0-OpenSSH_8.8)
129.241.162.52     10.22.9.32         SSHv2         95 Server: Protocol (SSH-2.0-OpenSSH_8.2p1 Ubuntu-4ubuntu0.5)
```

Here we can see the beginning of the SSH connection where the client and server provide protocols which they both support, and afterwards agree on using these supported protocols to continue communication on to the next step:



```
10.22.9.32         129.241.162.52     SSHv2        154 Client: Key Exchange Init
129.241.162.52     10.22.9.32         SSHv2       1110 Server: Key Exchange Init
```

This is where the key exchange begins, meaning that the client will now compare its own key with the server's key in order to see if the keys match, and consequently determine whether the client and server will use symmetric encryption to communicate with one another to establish the SSH connection. Once the communication via symmetric encryption has been agreed upon, the next step is to initiate the Diffie-Hellman key exchange algorithm which the client and the server will use to find the required secret key:



```
10.22.9.32         129.241.162.52     SSHv2        102 Client: Elliptic Curve Diffie-Hellman Key Exchange Init
129.241.162.52     10.22.9.32         SSHv2        490 Server: Elliptic Curve Diffie-Hellman Key Exchange Reply, New Keys
10.22.9.32         129.241.162.52     SSHv2         70 Client: New Keys
```

One can observe that the client initiates the Diffie-Hellman key exchange with the server, and they proceed to exchange calculated values and perform calculations using the received results. Once the key exchange is successfully performed and the secret key has been discovered by both sides, the server may finally attempt to authenticate the client and grant them their requested access.

Sources:
1. WebSockets Security Explained For Security Enthusiasts | 14.03.2022 by Manash Saikia. https://payatu.com/blog/websocketsecurity/

2. RFC 6455. Websocket protocol | December 2011 by Alexey Melnikov and Ian Fette. https://datatracker.ietf.org/doc/html/rfc6455
3. How Does SSH Work? | 17.12.2020 by Marko Aleksic. https://phoenixnap.com/kb/how-does-ssh-work
4. 7 - Cryptography Basics - Diffie-Hellman Key Exchange | 19.01.2021 by CBTVid. https://www.youtube.com/watch?v=d1KXDGgwIpA&ab_channel=CBTVid