Cryptography and Network Security Lab

Assignment 11
Implementation and Understanding of Diffie-Hellman Key Exchange Algorithm

2019BTECS00058
Devang K

Batch: B2

Title: Implementation and Understanding of Diffie-Hellman Key Exchange Algorithm

Aim: To Study, Implement and Demonstrate the Diffie-Hellman Key Exchange Algorithm

Theory:

Diffie–Hellman key exchange is a mathematical method of securely exchanging cryptographic keys over a public channel and was one of the first public-key protocols as conceived by Ralph Merkle and named after Whitfield Diffie and Martin Hellman. DH is one of the earliest practical examples of public key exchange implemented within the field of cryptography. Published in 1976 by Diffie and Hellman, this is the earliest publicly known work that proposed the idea of a private key and a corresponding public key.

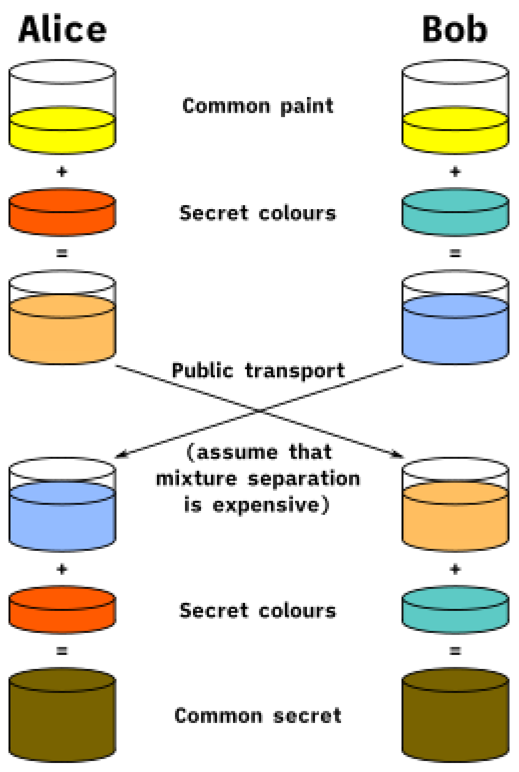
Traditionally, secure encrypted communication between two parties required that they first exchange keys by some secure physical means, such as paper key lists transported by a trusted courier. The Diffie–Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure channel. This key can then be used to encrypt subsequent communications using a symmetric-key cipher.

Diffie–Hellman is used to secure a variety of Internet services. However, research published in October 2015 suggests that the parameters in use for many DH Internet applications at that time are not strong enough to prevent compromise by very well-funded attackers, such as the security services of some countries.

We look at an illustration:

Diffie–Hellman key exchange establishes a shared secret between two parties that can be used for secret communication for exchanging data over a public network. An analogy illustrates the concept of public key exchange by using colors instead of very large numbers:

The process begins by having the two parties, Alice and Bob, publicly agree on an arbitrary starting color that does not need to be kept secret. In this example, the color is yellow. Each person also selects a secret color that they keep to themselves – in this case, red and cyan. The crucial part of the process is that Alice and Bob each mix their own secret color together with their mutually shared color, resulting in orange-tan and light-blue mixtures respectively, and then publicly exchange the two mixed colors. Finally, each of them mixes the color they received from the partner with their own private color. The result is a final color mixture (yellow-brown in this case) that is identical to their partner's final color mixture.



We look at Cryptographic Explanation:

The simplest and the original implementation of the protocol uses the multiplicative group of integers modulo p, where p is prime, and g is a primitive root modulo p. These two values are chosen in this way to ensure that the resulting shared secret can take on any value from 1 to p–1. Here is an example of the protocol, with non-secret values in blue, and secret values in red.

1. Alice and Bob publicly agree to use a modulus p = 23 and base g = 5 (which is a primitive root modulo 23).
2. Alice chooses a secret integer a = 4, then sends Bob A = g^a mod p
    a. A = 5^4 mod 23 = 4 (in this example both A and a have the same value 4, but this is usually not the case)
3. Bob chooses a secret integer b = 3, then sends Alice B = g^b mod p
    a. B = 5^3 mod 23 = 10
4. Alice computes s = B^a mod p
    a. s = 10^4 mod 23 = 18
5. Bob computes s = A^b mod p
    a. s = 4^3 mod 23 = 18
6. Alice and Bob now share a secret (the number 18).

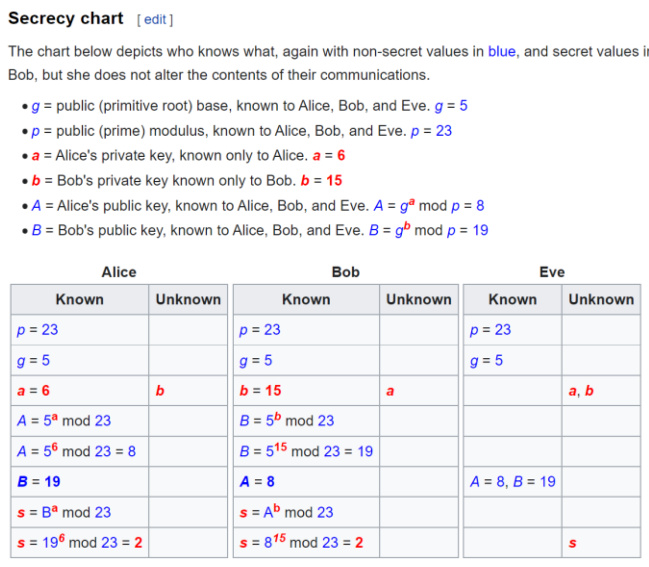
Both Alice and Bob have arrived at the same values because under mod p,

$$A^b \bmod p = g^{ab} \bmod p = g^{ba} \bmod p = B^a \bmod p$$

More specifically,

$$(g^a \bmod p)^b \bmod p = (g^b \bmod p)^a \bmod p$$

Only a and b are kept secret. All the other values – p, g, g^a mod p, and g^b mod p – are sent in the clear. The strength of the scheme comes from the fact that g^{ab} mod p = g^{ba} mod p take extremely long times to compute by any known algorithm just from the knowledge of p, g, g^a mod p, and g^b mod p. Once Alice and Bob compute the shared secret they can use it as an encryption key, known only to them, for sending messages across the same open communications channel.



Secrecy chart [edit]

The chart below depicts who knows what, again with non-secret values in blue, and secret values in **red**. Here Eve is an eavesdropper – she watches what is sent between Alice and Bob, but she does not alter the contents of their communications.

- g = public (primitive root) base, known to Alice, Bob, and Eve. $g = 5$
- p = public (prime) modulus, known to Alice, Bob, and Eve. $p = 23$
- a = Alice's private key, known only to Alice. $a = 6$
- b = Bob's private key known only to Bob. $b = 15$
- A = Alice's public key, known to Alice, Bob, and Eve. $A = g^a \bmod p = 8$
- B = Bob's public key, known to Alice, Bob, and Eve. $B = g^b \bmod p = 19$

Alice		Bob		Eve	
Known	Unknown	Known	Unknown	Known	Unknown
$p = 23$		$p = 23$		$p = 23$	
$g = 5$		$g = 5$		$g = 5$	
$a = 6$	b	$b = 15$	a		a, b
$A = 5^a \bmod 23$		$B = 5^b \bmod 23$			
$A = 5^6 \bmod 23 = 8$		$B = 5^{15} \bmod 23 = 19$			
$B = 19$		$A = 8$		$A = 8$, $B = 19$	
$s = B^a \bmod 23$		$s = A^b \bmod 23$			
$s = 19^6 \bmod 23 = 2$		$s = 8^{15} \bmod 23 = 2$			s

Let’s now look at the program implementation.

Code:

We have implemented the program in React + Node using Socket.io

There’s a server-side Node.js server which acts as a relay for the data passing. In node, we write the script to open the portal for the socket, then broadcast the incoming data:

```
const express = require('express');
const app = express();
const cors = require("cors");
const http = require('http').createServer(app);
const PORT = 8080;
const io = require('socket.io')(http,{cors: {origin: "*"}});

app.use(cors());
app.use(require('express').json());

let p = 0;
let g = 0;

let sharedKeyOfAlice = 0
let sharedKeyOfBob = 0;


io.on("connection", (socket) => {
  console.log("Someone joined!");
  socket.on("hello", (message)=>{
    console.log(message)
  });

  socket.on("publicKeyValues", (keys)=>{
    p = keys['p'];
    g = keys['g'];
    console.log(p)
    console.log(g)
    socket.broadcast.emit("publicKeyValues", keys);
  });

  socket.on("privKeyValueAlice", (value)=>{
    sharedKeyOfAlice = value
    socket.broadcast.emit("privKeyValueAlice", sharedKeyOfAlice);
  });

  socket.on("privKeyValueBob", (value)=>{
    sharedKeyOfBob = value
    socket.broadcast.emit("privKeyValueBob", sharedKeyOfBob);
  });

});

io.listen(PORT);
```

In the React side, we use socket-io-client and build 3 pages for the 3 users – Alice, Bob and Eve.

For the project, we assume that Alice enters the Public Keys (P and G). Then both Alice and Bob would create the private keys and receive their final key. Eve, meanwhile is able to eavesdrop on the data shared by Alice and Bob.

Alice:

```
import React, {useState, useEffect} from 'react'
import socketClient  from "socket.io-client";
const SERVER = "http://localhost:8080";

function gcd(x, y) {
    if ((typeof x !== 'number') || (typeof y !== 'number'))
      return false;
    x = Math.abs(x);
    y = Math.abs(y);
    while(y) {
      var t = y;
      y = x % y;
      x = t;
    }
    return x;
}

const checkIfPrimitiveRoot = (p, g) => {
    if(g<0 || gcd(p,g)!=1 || g>=p){
        alert("g<0 || gcd(p,g)!=1 || g>=p")
        return false;
    }
    let solutionsList = [];
    for(let i=1; i<p; i++){
        let value = Math.pow(g, i);
        value %= p
        if(solutionsList.includes(value)){
            alert(solutionsList)
            alert(value)
            return false;
        }
        solutionsList.push(value);
    }
    return true;
}

export default function Alice() {
    var socket = socketClient (SERVER, {
        rejectUnauthorized: false // WARN: please do not do this in production
    });

    socket.on("connect", () => {
        console.log(socket.id); // x8WIv7-mJelg7on_ALbx
        socket.emit('hello', "Hello from Alice!")
    });

    const [arePublicKeysDeclared, setArePublicKeysDeclared] = useState(false);
    const [areKeysExchanged, setAreKeysExchanged] = useState(false);
    const [hasSharedKey, setHasSharedKey] = useState(false);
    const [hasIncomingKeyReceived, setHasIncomingKeyReceived] =
useState(false)

    const [p, setP] = useState(0);
    const [g, setG] = useState(0);
    const [privKey, setPrivKey] = useState(0);
    const [keyToExchange, setKeyToExchange] = useState(0);
    const [incomingSharedKey, setIncomingSharedKey] = useState(0);
    const [finalKey, setFinalKey] = useState(-1)

    const publicKeysSubmitHandler = (e) => {
        e.preventDefault();
        if(!checkIfPrimitiveRoot(p, g)){
            alert('Invalid Value of Primitive Root.');
            return;
        }
        socket.emit("publicKeyValues", {
            p, g
        });
        setArePublicKeysDeclared(true);
    }

    const keysExchangeSubmitHandler = (e) => {
        e.preventDefault();
        let theValueToExchange = Math.pow(g, privKey);
        theValueToExchange %= p;
        setKeyToExchange(theValueToExchange)
        socket.emit("privKeyValueAlice", theValueToExchange);
        setAreKeysExchanged(true);
    }

    socket.on("privKeyValueBob", (value)=>{
        setIncomingSharedKey(value);
        setHasSharedKey(true);
        setHasIncomingKeyReceived(true)
    });

    const computeFinalKey = () => {
        let finalKeyValue = Math.pow(incomingSharedKey, privKey);
        finalKeyValue %= p;
        setFinalKey(finalKeyValue)
    }

    if(finalKey === -1){
        if(areKeysExchanged&&hasIncomingKeyReceived){
            computeFinalKey()
        }
    }

  return (
    <div className='container container-fluid'>
        <h2 style={{textAlign: 'center', marginBottom: '1rem'}}>Alice 👧</h2>
        {!arePublicKeysDeclared ? <></> : <div className='container container-
fluid'>
          <h6>Public Keys:</h6>
          <h6>P: {p}</h6>
          <h6>G: {g}</h6>
          </div>}
        {!arePublicKeysDeclared ? <form className='PandG'
onSubmit={publicKeysSubmitHandler} >
            <div className="form-group">
                <label htmlFor="p">Enter Public Key P</label>
                <input onChange={(e)=>setP(parseInt(e.target.value))}
type="number" className="form-control" id="p" placeholder="P" required />
            </div>
            <div className="form-group">
                <label htmlFor="g">Enter Public Key G</label>
                <input onChange={(e)=>setG(parseInt(e.target.value))}
type="number" className="form-control" id="g" placeholder="G - Primitive Root
of P" required />
            </div>
            <button style={{margin: '1rem auto'}} type="submit" class="btn
btn-primary">Submit</button>
        </form> : <></>}
        {arePublicKeysDeclared && !areKeysExchanged ? <>
        <div>
          <h6>Selection of Private Key and Exchange</h6>
          <form className='privKey' onSubmit={keysExchangeSubmitHandler} >
            <div className="form-group">
              <label htmlFor="privKey">Enter Private Key</label>
              <input onChange={(e)=>setPrivKey(parseInt(e.target.value))}
type="number" className="form-control" id="privKey" placeholder="Private Key"
required />
            </div>
            <button style={{margin: '1rem auto'}} type="submit" class="btn
btn-primary">Submit</button>
        </form>
        </div>
        </> : <></>}
        {areKeysExchanged && !hasSharedKey ? <>
        <div>
          <h6>Key Shared on Public Channel - {keyToExchange}</h6>
          <h6>Waiting for Key to be shared from Bob...</h6>
        </div>
        </> : <></>}
        {areKeysExchanged&&hasIncomingKeyReceived ? <>
        <div className='container container-fluid'>
            <h6>Key Shared on Public Channel - {keyToExchange}</h6>
            <h6>Incoming Key - {incomingSharedKey}</h6>
        </div>
        <div className='container container-fluid'>
            <h6>We get final common key as: {finalKey}</h6>
        </div>
        </> : <></>}
    </div>
  )
}
```

Bob:

```
import React, {useState, useEffect} from 'react'
import socketClient  from "socket.io-client";
const SERVER = "http://localhost:8080";

export default function Bob() {

  var socket = socketClient (SERVER, {
    rejectUnauthorized: false // WARN: please do not do this in production
  });

  socket.on("connect", () => {
      console.log(socket.id); // x8WIv7-mJelg7on_ALbx
      socket.emit('hello', "Hello from Bob!")
  });

  const [arePublicKeysDeclared, setArePublicKeysDeclared] = useState(false);
  const [areKeysExchanged, setAreKeysExchanged] = useState(false);
  const [hasSharedKey, setHasSharedKey] = useState(false)
  const [hasIncomingKeyReceived, setHasIncomingKeyReceived] = useState(false)

  const [p, setP] = useState(0);
  const [g, setG] = useState(0);
  const [privKey, setPrivKey] = useState(0);
  const [keyToExchange, setKeyToExchange] = useState(0);
  const [incomingSharedKey, setIncomingSharedKey] = useState(0)
  const [finalKey, setFinalKey] = useState(-1)

  socket.on("publicKeyValues", (keys)=>{
    setP(keys['p'])
    setG(keys['g'])
    setArePublicKeysDeclared(true);
  });

  const keysExchangeSubmitHandler = (e) => {
    e.preventDefault();
    let theValueToExchange = Math.pow(g, privKey);
    console.log(theValueToExchange);
    console.log("Aaaaaaa")
    theValueToExchange %= p;
    setKeyToExchange(theValueToExchange)
    socket.emit("privKeyValueBob", theValueToExchange);
    setAreKeysExchanged(true);
  }

  socket.on("privKeyValueAlice", (value)=>{
    setIncomingSharedKey(value);
    setHasSharedKey(true);
    setHasIncomingKeyReceived(true)
  })

  const computeFinalKey = () => {
    let finalKeyValue = Math.pow(incomingSharedKey, privKey);
    finalKeyValue %= p;
    setFinalKey(finalKeyValue)
  }

  if(finalKey === -1){
    if(areKeysExchanged&&hasIncomingKeyReceived){
        computeFinalKey()
    }
  }

  return (
    <div className='container container-fluid'>
        <h2 style={{textAlign: 'center', marginBottom: '1rem'}}>Bob 👦</h2>
        {!arePublicKeysDeclared ? <p>Waiting for declaration of Public
Keys...</p> : <div className='container container-fluid'>
          <h6>Public Keys:</h6>
          <h6>P: {p}</h6>
          <h6>G: {g}</h6>
          </div>}
        {arePublicKeysDeclared && !areKeysExchanged ? <>
        <div>
          <h6>Selection of Private Key and Exchange</h6>
          <form className='privKey' onSubmit={keysExchangeSubmitHandler} >
            <div className="form-group">
              <label htmlFor="privKey">Enter Private Key</label>
              <input onChange={(e)=>setPrivKey(parseInt(e.target.value))}
type="number" className="form-control" id="privKey" placeholder="Private Key"
required />
            </div>
            <button style={{margin: '1rem auto'}} type="submit" class="btn
btn-primary">Submit</button>
        </form>
        </div>
        </> : <></>}
        {areKeysExchanged && !hasSharedKey ? <>
        <div>
```

```
          <h6>Key Shared on Public Channel - {keyToExchange}</h6>
          <h6>Waiting for Key to be shared from Alice...</h6>
        </div>
        </> : <></>}
        {areKeysExchanged&&hasIncomingKeyReceived ? <>
        <div className='container container-fluid'>
            <h6>Key Shared on Public Channel - {keyToExchange}</h6>
            <h6>Incoming Key - {incomingSharedKey}</h6>
        </div>
        <div className='container container-fluid'>
            <h6>We get final common key as: {finalKey}</h6>
        </div>
        </> : <></>}
    </div>
  )
}
```

Eve:

```
import React, {useState, useEffect} from 'react'
import socketClient  from "socket.io-client";
const SERVER = "http://localhost:8080";

export default function Eve() {

  var socket = socketClient (SERVER, {
    rejectUnauthorized: false // WARN: please do not do this in production
  });

  const [arePublicKeysReceived, setArePublicKeysReceived] = useState(false);
  const [p, setP] = useState(0)
  const [g, setG] = useState(0)
  const [isKeySharedByAlice, setIsKeySharedByAlice] = useState(false)
  const [keySharedByAlice, setKeySharedByAlice] = useState(0)
  const [isKeySharedByBob, setIsKeySharedByBob] = useState(false)
  const [keySharedByBob, setKeySharedByBob] = useState(0)


  socket.on("connect", () => {
      console.log(socket.id); // x8WIv7-mJelg7on_ALbx
      socket.emit('hello', "Hello from Eve!")
  });

  socket.on("publicKeyValues", (keys)=>{
    setP(keys['p']);
    setG(keys['g']);
    setArePublicKeysReceived(true);
  });

  socket.on("privKeyValueAlice", (value)=>{
    setKeySharedByAlice(value);
    setIsKeySharedByAlice(true);
  });

  socket.on("privKeyValueBob", (value)=>{
    setKeySharedByBob(value);
    setIsKeySharedByBob(true);
  });

  return (
    <div className='container container-fluid'>
        <h2 style={{textAlign: 'center', marginBottom: '2rem'}}>Eve 😼</h2>
        <div className='container container-fluid'>
          <h6>Public Key P: {arePublicKeysReceived ? p : 'Waiting for
it...'}</h6>
          <h6>Public Key G: {arePublicKeysReceived ? g : 'Waiting for
it...'}</h6>
          <h6>Key Shared By Alice: {isKeySharedByAlice ? keySharedByAlice :
'Waiting for it...'}</h6>
          <h6>Key Shared By Bob: {isKeySharedByBob ? keySharedByBob : 'Waiting
for it...'}</h6>
        </div>
    </div>
  )
}
```

We now illustrate with examples:

Say we wish to demonstrate the example above.

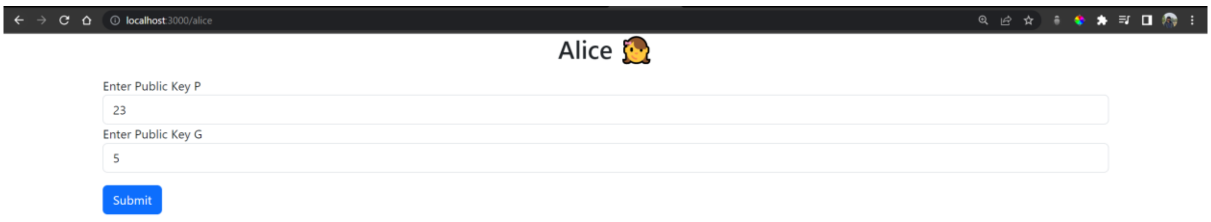
P – 23

G – 5

a – 6 (Private key chosen by Alice)

b – 15 (Private key chosen by Bob)

Now, we start with Alice:


Alice

Enter Public Key P

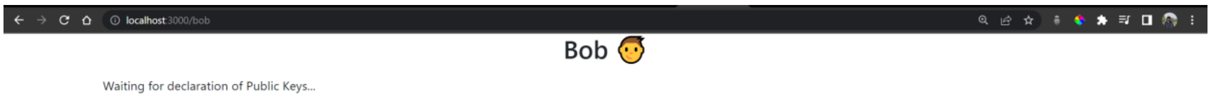
23

Enter Public Key G

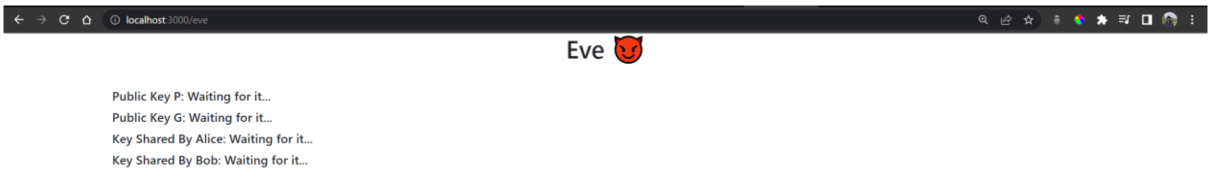
5

Submit

Meanwhile, Bob and Eve are idle and waiting.

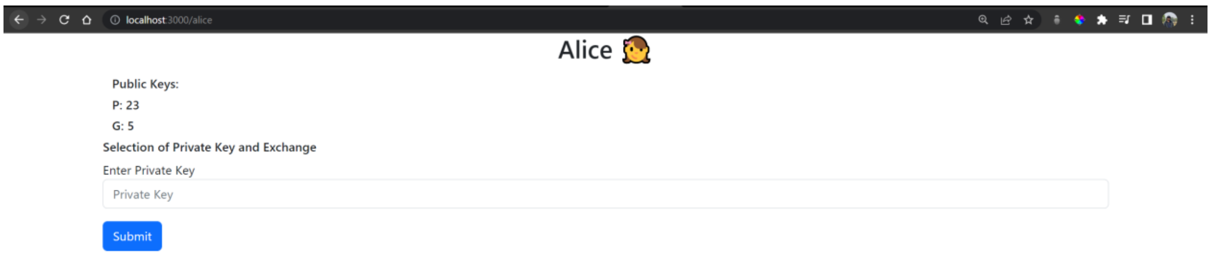

Bob

Waiting for declaration of Public Keys...


Eve

Public Key P: Waiting for it...
Public Key G: Waiting for it...
Key Shared By Alice: Waiting for it...
Key Shared By Bob: Waiting for it...

After Alice submits the key:


Alice

Public Keys:
P: 23
G: 5

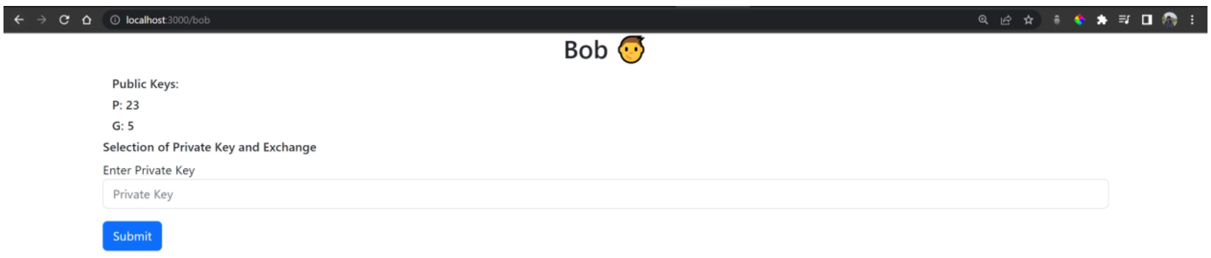
Selection of Private Key and Exchange

Enter Private Key

Private Key

Submit

Same with Bob:


Bob

Public Keys:
P: 23
G: 5

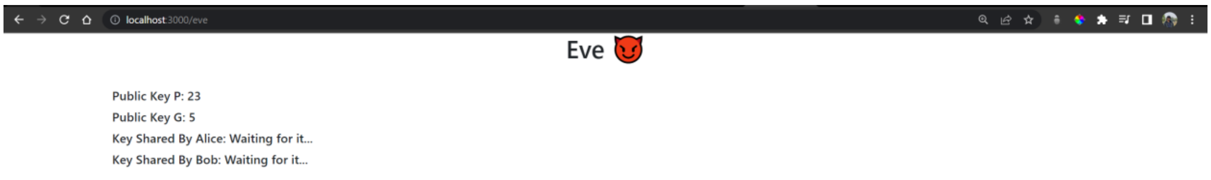
Selection of Private Key and Exchange

Enter Private Key

Private Key

Submit

And with Eve:

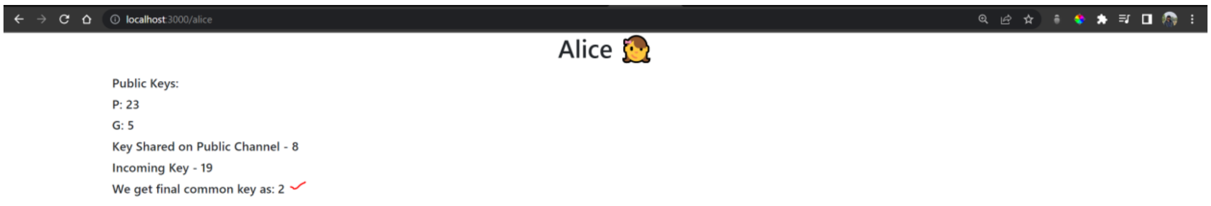


Eve 😈

```
Public Key P: 23
Public Key G: 5
Key Shared By Alice: Waiting for it...
Key Shared By Bob: Waiting for it...
```

When Alice and Bob enter 6 and 15 respectively:

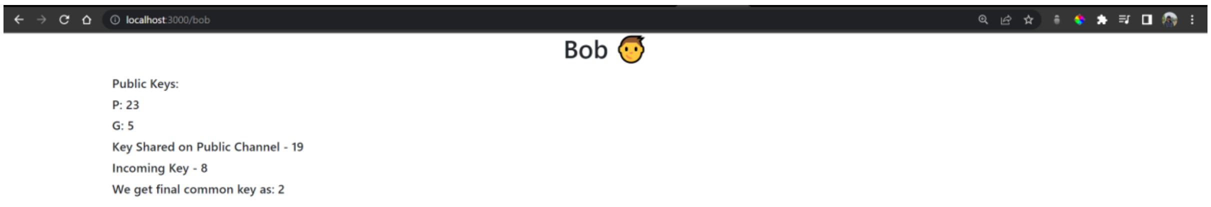
Alice:



Alice 🙋

```
Public Keys:
P: 23
G: 5
Key Shared on Public Channel - 8
Incoming Key - 19
We get final common key as: 2 ✓
```

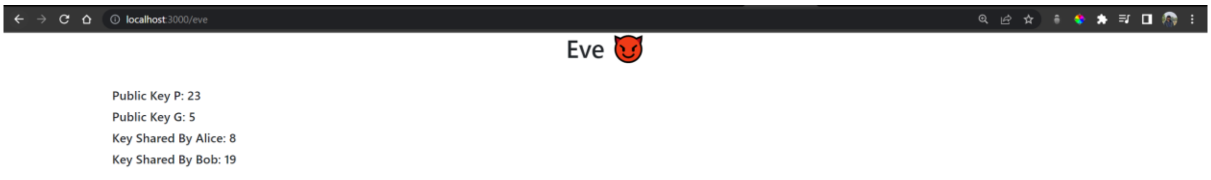
Bob:



Bob 🙂

```
Public Keys:
P: 23
G: 5
Key Shared on Public Channel - 19
Incoming Key - 8
We get final common key as: 2
```

Eve:



Eve 😈

```
Public Key P: 23
Public Key G: 5
Key Shared By Alice: 8
Key Shared By Bob: 19
```

Thus, we see that Alice and Bob compute the required common final key on their end, meanwhile Eve does not know the a and b to do so.

Discrete logarithm would be used to hack Diffie Hellman. Therefore, for very large numbers, this algorithm would be secure.

Conclusion:

Thus, the Diffie-Hellman Key Exchange algorithm was studied and demonstrated with the code.