# CSE1011 – Cryptography fundamentals

**Final Review Report**

“**Implementing a secured hashing algorithm to authenticate a particular database”**

****

## Department of Computer Science

**School of Computing Science and Engineering**

**Project Members**:

Nimish Shah -18BCI0197

Yashraj Agarwal - 18BCI0183

**COURSE -** Cryptography Fundamentals CSE-1011

**FACULTY** – Prof. AMUTHA PRABAKAR N

**TOPIC -** Implementing a secured hashing algorithm to authenticate a particular database.

# Introduction:

Logging into any particular system requires to be authenticated using an username and password. Passwords are generally stored in tupular format. Retrieval of passwords without being encrypted allows vulnerability and loss of security. Thus hashing and then matching provides more security to any malicious attack which can reveal sensitive information. To obtain above we are using a hashing algorithm to encrypt the password.

# Abstract:

We have created a website that encrypts the password one has entered. Here we use blake2b to hash our password inputted and store it in our mongo DB. To prove this, we also demonstrate it by showing our mongo database. There we see the inputted data is completely hashed and cannot be decrypted by a hacker. Therefore, an SQL injection would be useless.

Here we build our back-end using nodeJS where we use it to host our website, to create and implement blake2b code in JS and connect with our website such that when we enter the data in pssword space the input stored in our mongo dB is hashed.

To implement that we also create a JS file named db.js and log.js which transfers the data entered in the database.

# MOTIVATION:

The requirement for security has been ascent in the present current and creating world. As innovation propels distinctive techniques for hashing algorithms have been produced to counter the issues of unsecured stages and assaults by programmers. Hashing is one of the most secure ways to secure the passwords and nowadays passwords are generally secured by triple des encryption. But the ongoing progress in decryption department will now ensure even its decryption. Therefore, we were motivated to use Blake2b in our encryption website. In today’s world, id and password usage has become a very important .With it comes the need for the professional programmer to protect these assets cryptographically as well as to devise ways for efficient computing . To make our transaction on saving id and password more secure, we need some security protocols. Therefore, in an attempt to create a robust and secure application using Blake2s and Hashing algorithm, we have undertaken this project.

The requirement for security has been ascent in the present current and creating world. As innovation propels distinctive techinques for hashing algorithms have been produced to counter the issues of unsecured stages and assau lts by programmers .

# Literature Survey

Studies on the subject state that within the trendy years the necessity of security has amplified several folds. It explains that the necessity to secure knowledge isn't new and goes means back to the time of first warfare and even any back to the time of solon. This paper provides a history of the info coding techniques employed in the history and the way they need developed from Caesar cipher to DES and from DES to Triple-DES, AES and up to date algorithms.

This study explained the execs and cons of earlier algorithms and why they aren’t used these days. the most focus of this paper is ab initio general knowledge activity strategies so cryptanalytic algorithms. we tend to studied this paper thus examine the development over SHA-512 with planned work BLAKE-512 hash perform. National Institute of normal and technology (NIST) concentrate on the new SHA-3 competition, started by the NIST, that searches for a replacement hash perform in response to authentication considerations concerning the previous hash functions

SHA - 1 and therefore the SHA-2 family.

The paper aims to indicate that since SHA-512 used eighty rounds to calculate 512-bit final hash price with 512-bit initial hash price whereas BLAKE-512 is employed with sixteen rounds to calculate final hash price with same input length and same output length in C programing language.

Every smallest service on the {web/the net} is created on the market through web applications. Services like on-line searching, on-line industry, e-booking system for railways or airlines and lots of a lot of area unit all on the market at the doorstep with the assistance of net. With these various applications comes the majority quantity of information they store on day to day in their backend databases. the difficulty is a way to store the info in associate degree economical and secure manner in order that it mustn't be put-upon. these days most of the applications use Cloud storage for this purpose. however, is that the knowledge extremely secured onCloud?

With the increasing cases of cyber - attacks one will comprehend the solution to the current question. The attacks like code injection attacks, denial of service attack, spoofing, phishing attack, HTTP flood attack e. area unit a number of the key attacks difficult the protection of those applications. one in all the key harmful attacks is SQL injection attacks (SQLinAs). There are a unit many detection and bar techniques for identical nonetheless the applications area unit extremely susceptible to SQL in As. to raised perceive the attack, all the prevailing varieties, detection and bar techniques for SQL in As area unit analyzed and showcased during this paper.

**AIM** - In this project we exhibit another strategy for encryption before hand existing strategies and adequately consolidating then to build up effective hashing algorithm which is faster than the previous ones.

We will be taking username and password as input from users and then we will encrypt them using the hashing algorithm and store it in database.

**Algorithm in use:**

BLAKE2B : 512 bit encryption algorithm

## Hashing

Hashing is a crucial system that is intended to use a special platform referred to as the Hash platform that is employed to map a given price with a specific key for quicker access of components. The potency of mapping depends of the potency of the hash perform used.

Hash functions kind a crucial class of cryptography, that is wide employed in a good range of protocols and security mechanisms. it's outlined as computationally economical perform, that maps binary strings of absolute length to binary strings of mounted length. The last ones square measure the outputs of a hash computation and that they square measure referred to as hash values. Hash functions square measure applied to support digital signatures, knowledge integrity, random range generators, authentication schemes, and knowledge integrity mechanisms.

### Pseudocode/Algorithm for Blake 2b

**Algorithm BLAKE2b**

**Input:**

**M Message to be hashed**

**cbMessageLen: Number, (0..2128) Length of the message in bytes**

**Key Optional 0..64 byte key**

**cbKeyLen: Number, (0..64) Length of optional key in bytes**

**cbHashLen: Number, (1..64) Desired hash length in bytes**

**Output:**

**Hash Hash of cbHashLen bytes**

**Initialize State vector h with IV**

**h0..7 ← IV0..7**

**Mix key size (cbKeyLen) and desired hash length (cbHashLen) into h0**

**h0 ← h0 xor 0x0101kknn**

**where kk is Key Length (in bytes)**

**nn is Desired Hash Length (in bytes)**

**Each time we Compress we record how many bytes have been compressed**

**cBytesCompressed ← 0**

**cBytesRemaining ← cbMessageLen**

**If there was a key supplied (i.e. cbKeyLen > 0)**

**then pad with trailing zeros to make it 128-bytes (i.e. 16 words)**

**and prepend it to the message M**

**if (cbKeyLen > 0) then**

**M ← Pad(Key, 128) || M**

**cBytesRemaining ← cBytesRemaining + 128**

**end if**

**Compress whole 128-byte chunks of the message, except the last chunk**

**while (cBytesRemaining > 128) do**

**chunk ← get next 128 bytes of message M**

**cBytesCompressed ← cBytesCompressed + 128 increase count of bytes that have been compressed**

**cBytesRemaining ← cBytesRemaining - 128 decrease count of bytes in M remaining to be processed**

**h ← Compress(h, chunk, cBytesCompressed, false) false ⇒ this is not the last chunk**

**end while**

**Compress the final bytes from M**

**chunk ← get next 128 bytes of message M We will get cBytesRemaining bytes (i.e. 0..128 bytes)**

**cBytesCompressed ← cBytesCompressed+cBytesRemaining The actual number of bytes leftover in M**

**chunk ← Pad(chunk, 128) If M was empty, then we will still compress a final chunk of zeros**

**h ← Compress(h, chunk, cBytesCompressed, true) true ⇒ this is the last chunk**

**Result ← first cbHashLen bytes of little endian state vector h**

**End Algorithm BLAKE2b.**

**PROPOSED SYSTEM:**

Back-end of Our WEBSITE:

NODEJS code for :

BLAKE2b js code

Code that hosts local server at http://localhost:3000/

JS code for blake 2b implementation.

**CODE** -

**Back-end JavaScript Code -**

const db = require("./db");

const collection="login";

const path=require('path');

const express = require('express');

const body\_parser=require('body-parser');

const log=express();

const blake=require('./blakejs-master/blake2b');

const blake2b=blake.blake2bHex;

//const blake=require('./blake')

log.use(body\_parser.json());

log.use(express.json());

log.use(express.urlencoded());

log.get('/',(req,res)=>{

res.sendFile(path.join(\_\_dirname,'index.html'));

});

log.post('/',(req,res,next)=>{

const ip=req.body;

console.log(ip.username,ip.password);

db.getDB().collection(collection).find({username:ip.username}).toArray().then((doc)=>{

if(blake2b(ip.password)===doc[0]['password']){

res.sendFile(path.join(\_\_dirname,'login.html'));

}else{

res.send('error');

}

}).catch((err)=>{

console.log(err);

}).finally(()=>{

var a=1;

})

})

// log.get('/:id',(req,res)=>{

// const ip=req.params.id;

// })

log.get('/signup',(req,res)=>{

res.sendFile(path.join(\_\_dirname,'Signup.html'));

});

log.post('/signup',(req,res,next)=>{

const ip=req.body;

console.log(ip);

ip.password=blake2b(ip.password);

db.getDB().collection(collection).insertOne(ip,(err,result)=>{

if(err){

const error=new Error("failed");

error.status=400;

next(error);ca

}

else{

console.log('success');

//next();

res.json({result:result, document:result.ops[0],msg:"success",error:null});

next();

}

})

});

// log.get('/update/:id',(req,res)=>{

// res.sendFile(path.join(\_\_dirname,'Update.html'));

// })

log.put('/update/:id',(req,res)=>{

const id=req.params.id;

const user\_ip=req.body;

db.getDB().collection(collection).findOneAndUpdate({\_id:db.getPrimaryKey(id)},{$set:{usname:user\_ip.usname}},{returnOriginal:false},(err,result)=>{

if(err) console.log(err);

else res.json(result);

})

})

db.connect((err)=>{

if(err){

console.log("can't connect");

process.exit(1);

} else {

log.listen(3000,()=>{

console.log('listening on port xx3000');

// var a=db.getDB().collection(collection).find({}).toArray();

// console.log(a);

});

}

});

**Normalizing Input -**

var ERROR\_MSG\_INPUT = 'Input must be an string, Buffer or Uint8Array'

// For convenience, let people hash a string, not just a Uint8Array

function normalizeInput (input) {

var ret

if (input instanceof Uint8Array) {

ret = input

} else if (input instanceof Buffer) {

ret = new Uint8Array(input)

} else if (typeof (input) === 'string') {

ret = new Uint8Array(Buffer.from(input, 'utf8'))

} else {

throw new Error(ERROR\_MSG\_INPUT)

}

return ret

}

**Blake2b Initializing Input -**

function blake2bInit (outlen, key) {

if (outlen === 0 || outlen > 64) {

throw new Error('Illegal output length, expected 0 < length <= 64')

}

if (key && key.length > 64) {

throw new Error('Illegal key, expected Uint8Array with 0 < length <= 64')

}

// state, 'param block'

var ctx = {

b: new Uint8Array(128),

h: new Uint32Array(16),

t: 0, // input count

c: 0, // pointer within buffer

outlen: outlen // output length in bytes

}

// initialize hash state

for (var i = 0; i < 16; i++) {

ctx.h[i] = BLAKE2B\_IV32[i]

}

var keylen = key ? key.length : 0

ctx.h[0] ^= 0x01010000 ^ (keylen << 8) ^ outlen

// key the hash, if applicable

if (key) {

blake2bUpdate(ctx, key)

// at the end

ctx.c = 128

}

return ctx

}

**Blake2b update:**

function blake2bUpdate (ctx, input) {

for (var i = 0; i < input.length; i++) {

if (ctx.c === 128) { // buffer full ?

ctx.t += ctx.c // add counters

blake2bCompress(ctx, false) // compress (not last)

ctx.c = 0 // counter to zero

}

ctx.b[ctx.c++] = input[i]

}

}

**Blake2b Final:**

function blake2bFinal (ctx) {

ctx.t += ctx.c // mark last block offset

while (ctx.c < 128) { // fill up with zeros

ctx.b[ctx.c++] = 0

}

blake2bCompress(ctx, true) // final block flag = 1

var out = new Uint8Array(ctx.outlen)

for (var i = 0; i < ctx.outlen; i++) {

out[i] = ctx.h[i >> 2] >> (8 \* (i & 3))

}

return out

}

**Blake2B Compressing Function:**

function blake2bCompress (ctx, last) {

var i = 0

// init work variables

for (i = 0; i < 16; i++) {

v[i] = ctx.h[i]

v[i + 16] = BLAKE2B\_IV32[i]

}

// low 64 bits of offset

v[24] = v[24] ^ ctx.t

v[25] = v[25] ^ (ctx.t / 0x100000000)

// high 64 bits not supported, offset may not be higher than 2\*\*53-1

// last block flag set ?

if (last) {

v[28] = ~v[28]

v[29] = ~v[29]

}

// get little-endian words

for (i = 0; i < 32; i++) {

m[i] = B2B\_GET32(ctx.b, 4 \* i)

}

// twelve rounds of mixing

for (i = 0; i < 12; i++) {

B2B\_G(0, 8, 16, 24, SIGMA82[i \* 16 + 0], SIGMA82[i \* 16 + 1])

B2B\_G(2, 10, 18, 26, SIGMA82[i \* 16 + 2], SIGMA82[i \* 16 + 3])

B2B\_G(4, 12, 20, 28, SIGMA82[i \* 16 + 4], SIGMA82[i \* 16 + 5])

B2B\_G(6, 14, 22, 30, SIGMA82[i \* 16 + 6], SIGMA82[i \* 16 + 7])

B2B\_G(0, 10, 20, 30, SIGMA82[i \* 16 + 8], SIGMA82[i \* 16 + 9])

B2B\_G(2, 12, 22, 24, SIGMA82[i \* 16 + 10], SIGMA82[i \* 16 + 11])

B2B\_G(4, 14, 16, 26, SIGMA82[i \* 16 + 12], SIGMA82[i \* 16 + 13])

B2B\_G(6, 8, 18, 28, SIGMA82[i \* 16 + 14], SIGMA82[i \* 16 + 15])

}

for (i = 0; i < 16; i++) {

ctx.h[i] = ctx.h[i] ^ v[i] ^ v[i + 16]

}

}

**Processing Function:**

function blake2b (input, key, outlen) {

// preprocess inputs

outlen = outlen || 64

input = util.normalizeInput(input)

// do the math

var ctx = blake2bInit(outlen, key)

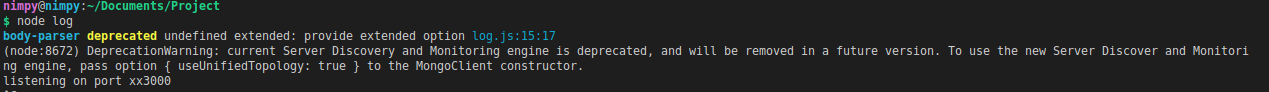
blake2bUpdate(ctx, input)

return blake2bFinal(ctx)

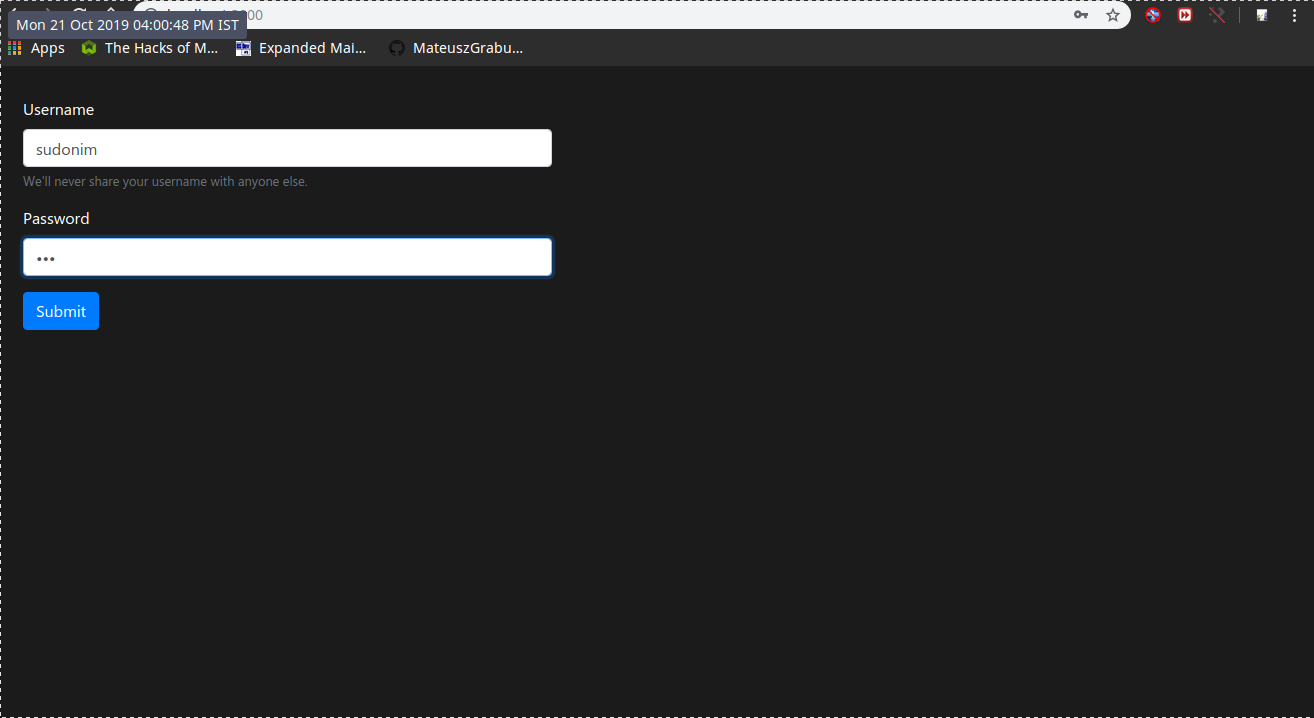
}

**Procedure:**

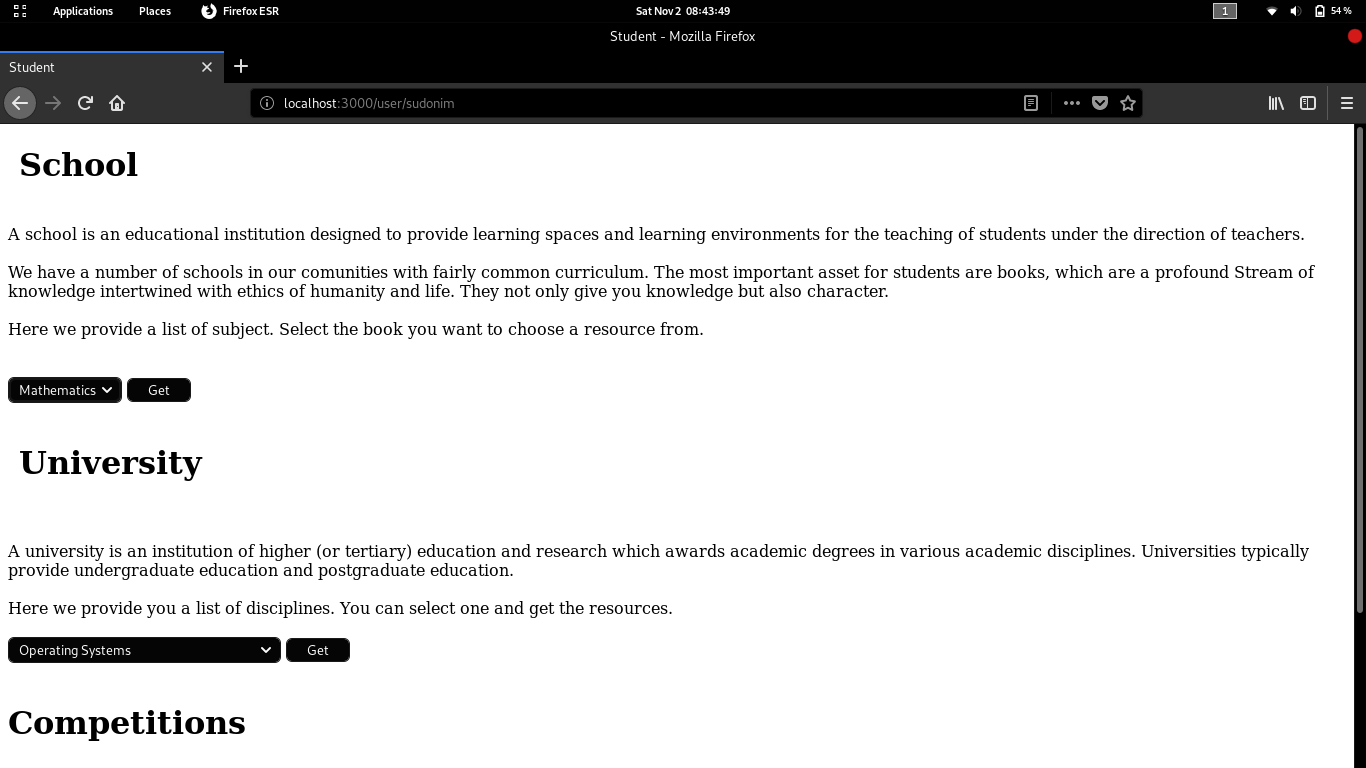
host our website locally



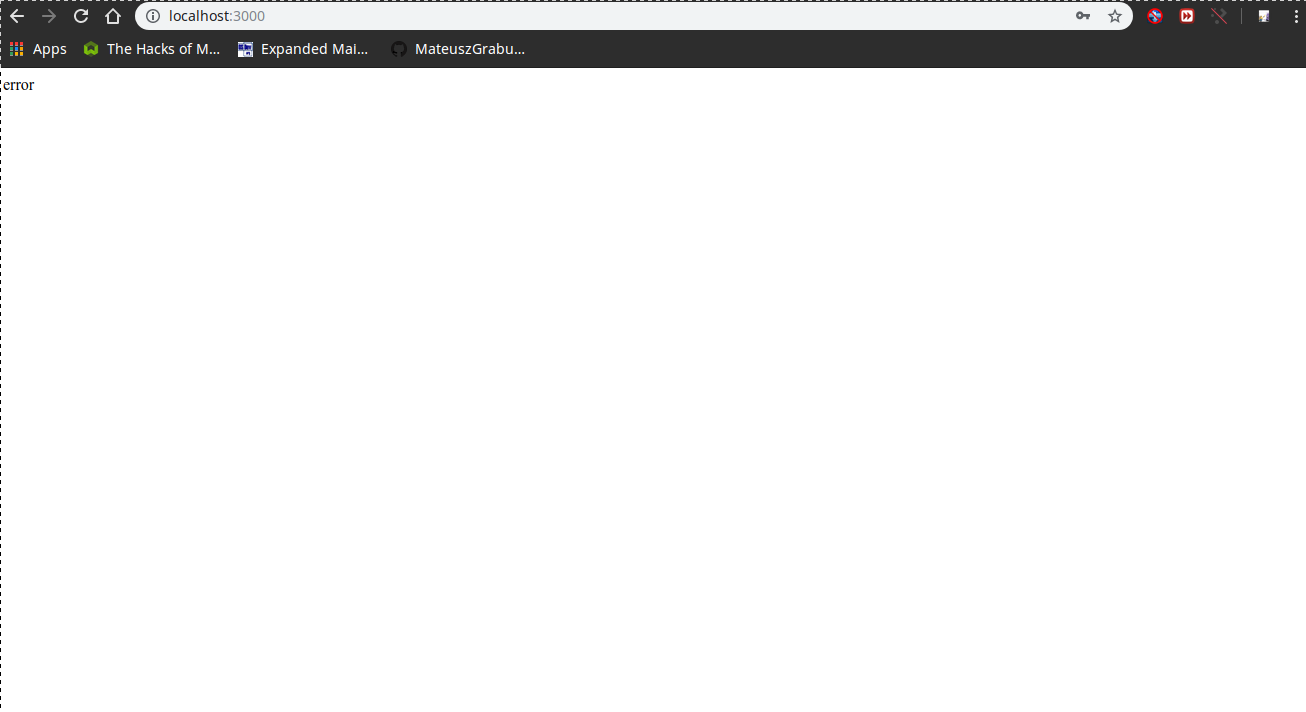
goto <http://localhost:3000/>



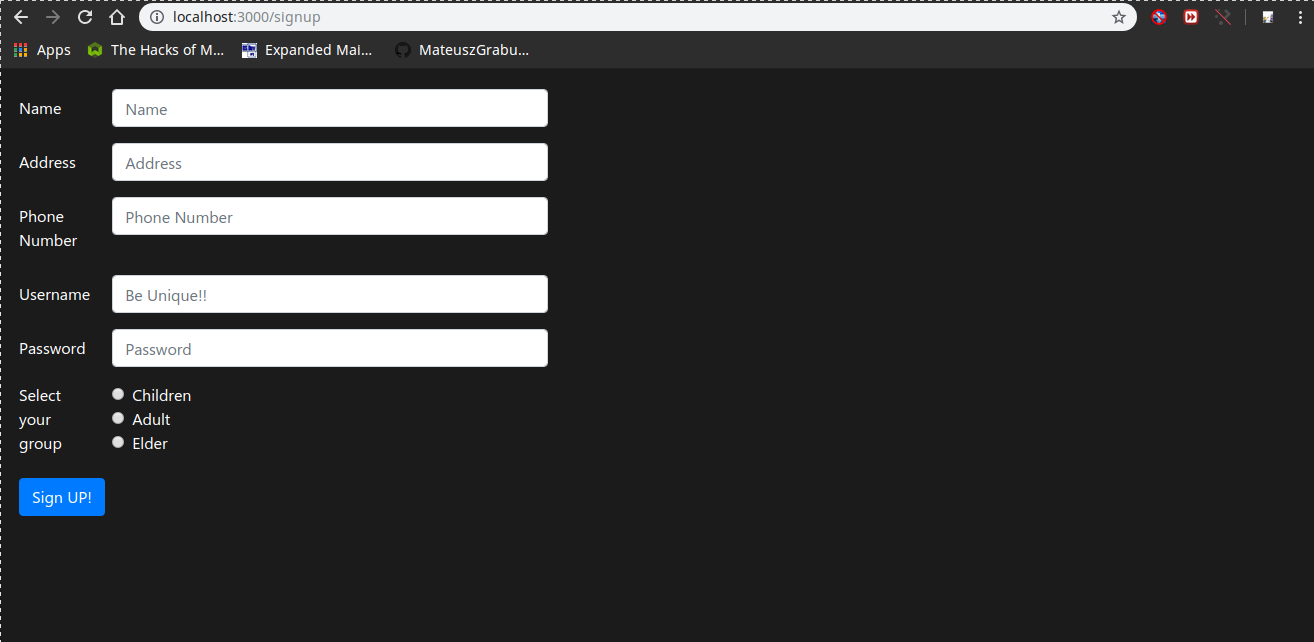
**If password is right , then next screen will be displayed**



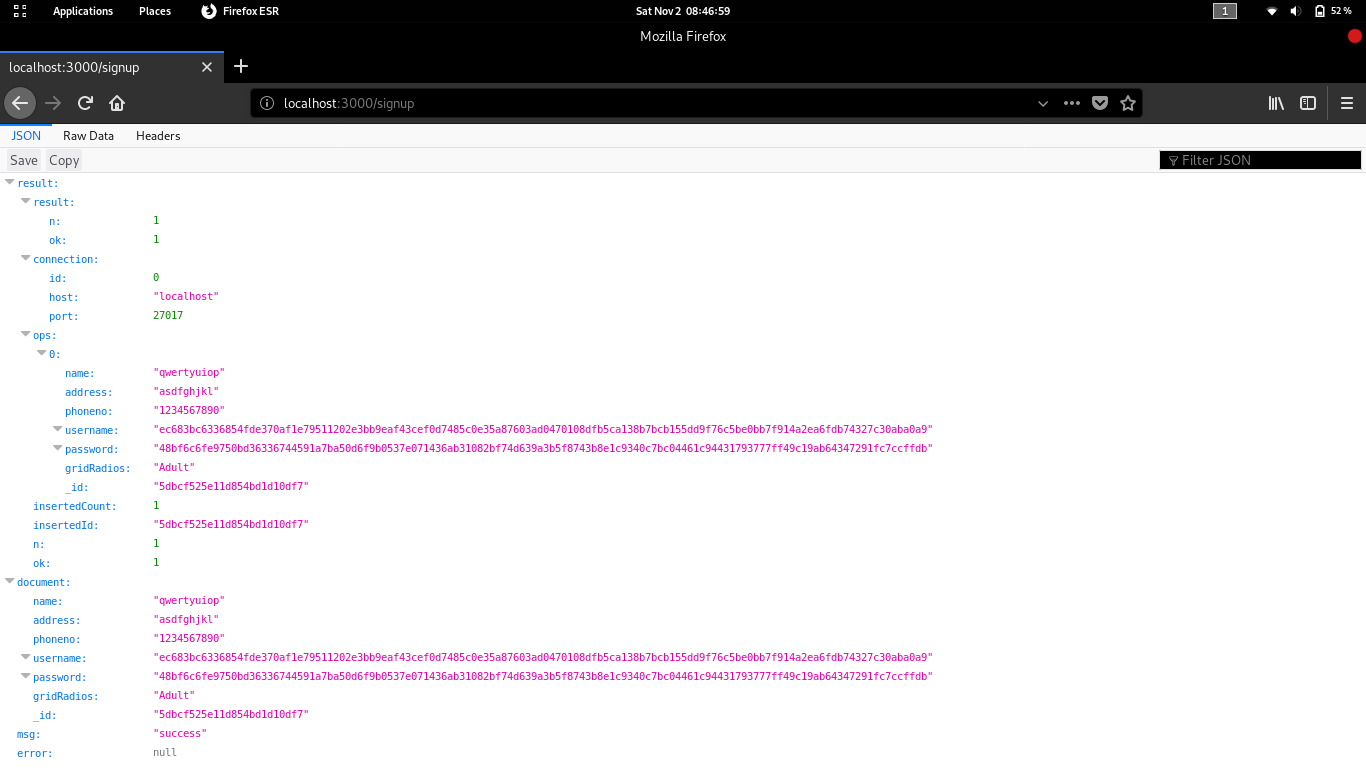
**If password is wrong, then next screen will be displayed**

****

**For signup the next screen will be displayed**

****

**For Successful signup next page will be displayed**

****

### References:

1. Abhishek Bhardwaj et al, “Study ofDifferent Cryptographic Technique and Challenges in Future”, 20161stInternational Conference on Innovation and Challenges inCyber Security (ICICCS2016)
2. Cryptanalytic jh and BLAKE HASH function for authentication and proposed work over BLAKE-512 on CLanguage

[3]<https://link.springer.com/chapter/10.1007/978-3-642-38980-1_8>

1. <https://blake2.net/>
2. The hash function BLAKE by Jean-Philippe Aumasson ,Samuel Neves Zooko Wilcox- O’Hearn ,ChristianWinnerlein.