IERG 4350 PROJECT REPORT

Project Title: Voting System

Group Members:
Name: IP Shing On SID: 1155109011
Name: WONG Shing SID: 1155109027

Motivation:

Nowadays, many online voting websites need many registration processes before really holding a voting event. They are not convenient for the users. Also, they might not provide real-time results for the event holder to view. The event holders might need to wait for the end of the voting to check the results. Therefore, we would like to solve the above problems by focusing on user experience. Moreover, the security of the voting is also our concern, so our online voting website can protect data in transit and secure database data.

Overview:

This voting website is a MERN web application. It is deployed on AWS EC2 and can be visited at http://ec2-52-77-255-166.ap-southeast-1.compute.amazonaws.com/, which is open to the public. Nginx is used to manage the React frontend while Process Manager is used to manage the Node backend. MongoDB is used to store the vote data. Some security tools are implemented, such as ECDH for key exchange between the React Client and the Node Server, AES-256-GCM is used to encrypt the message, HmacSHA256 is used to check the message integrity, and bcrypt to hash the passcode values.

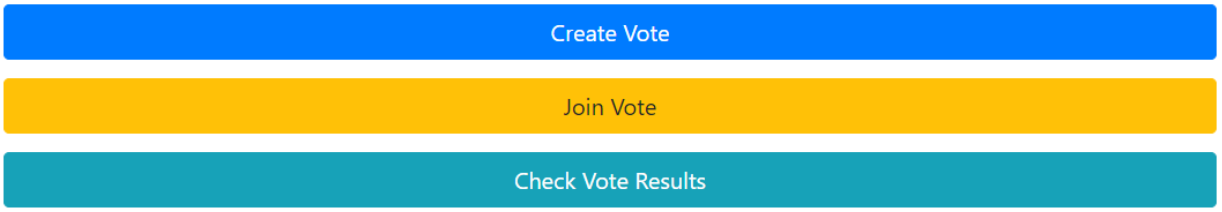
Vote Schema:

title: String,
items: [{ name: String, count: String }],
creatorPasscode: String,
participants: [{ participantPasscode: String, voted: Boolean }]

```
//Vote Schema
const VoteSchema = mongoose.Schema({
    title: {
        type: String,
        default: "",
        required: true
    },
    items: {
        type: [{
            name: {
                type: String,
                default: "",
                //required: true
            },
            count: {
                type: String,
                default: "",
                //required: true
            }
        }],
        default: [],
        required: true
    },
    creatorPasscode: {
        type: String,
        default: "",
        //required: true
    },
    participants: {
        type: [{
            participantPasscode: {
                type: String,
                default: "",
                //required: true
            },
            voted: {
                type: Boolean,
                default: false,
                //required: true
            }
        }],
        default: [],
        required: true
    }
);
```

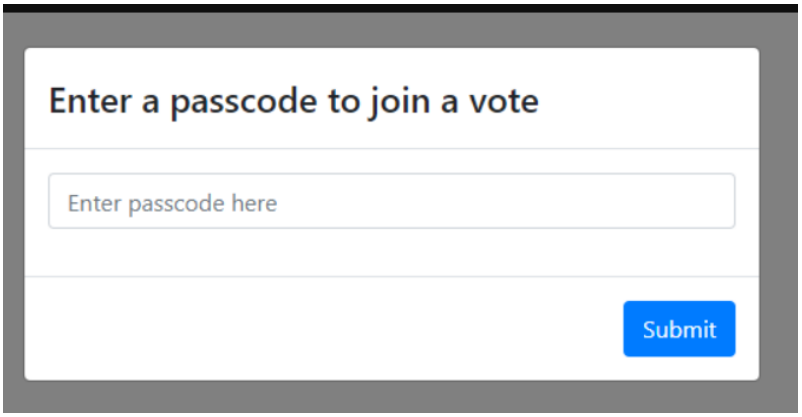
Screenshots of our voting website:

MainPage

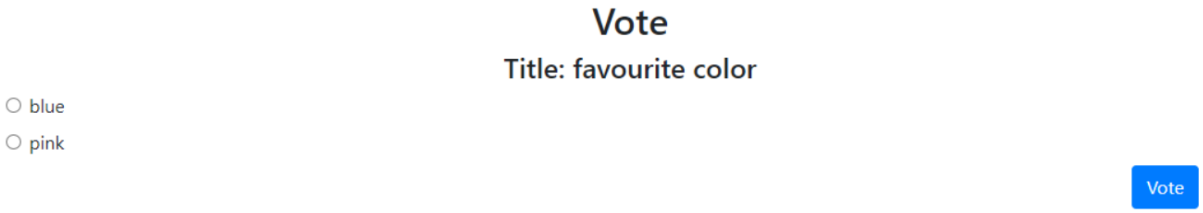


Create Vote

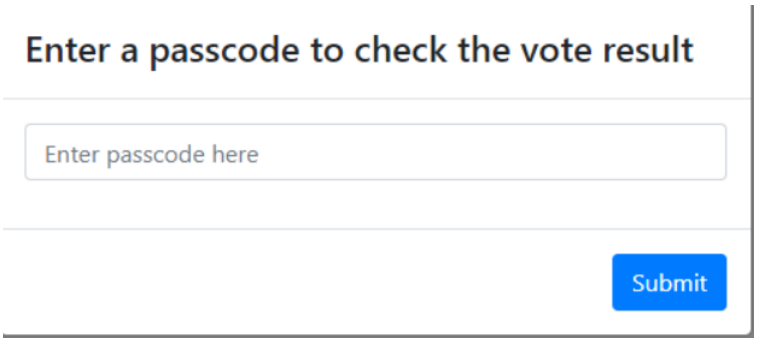
Passcode to join vote



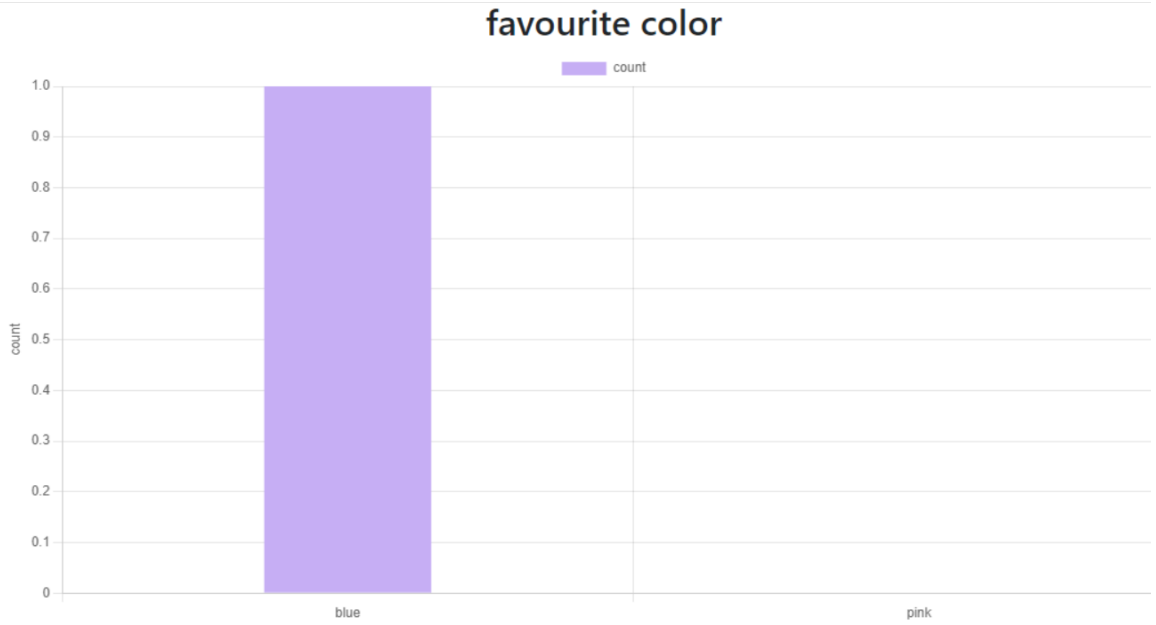
Submit vote



Check vote results



Show vote results



Planning and Product:

Vote validation

We have implemented vote validation as planned, that each participant can vote once only and only validate users can join the vote.

No interpretation from organizer

Once the organizer creates a vote, the passcode to join the vote is sent to the participant's emails. Only the participants' themselves can join the vote. Therefore, there is no interpretation from the organizer. Besides, the database only stores the hash value of the participants' passcode instead of their email address, so no one knows the voting status of the participants.

Protecting data in transit / data at rest

We provide protection to data in transit and data at rest. For the data in transit, they are encrypted byAES with ECDH shared secret key between the client and the server as well as using HmacSHA256 to check the message integrity. For the data at rest, they are stored in the database after being encrypted by AES-256-GCM.

Use HTTPS instead HTTP

We did not implement this function because it requires costs to buy a valid SSL certificate from CA authority.

Technology Used:

Development:

1. ReactJS (to be the frontend framework)
2. React-Router (to build an SPA)
3. Bootstrap (to do the layout)
4. NodeJS (to be the development platform)
5. ExpressJS (to do request and response)
6. Mongoose (to work with MongoDB)
7. axios (to connect frontend to backend)
8. nodemailer (to send the emails)
9. bcryptJS (to hash the passcodes)
10. express-async-errors (to handle uncaught exceptions)
11. dotenv (to save the predefined secret key in the environment)

Security:

1. Node Crypto (to protect data at rest in the database)
2. CryptoJS (to protect data in transit)
3. Elliptic (to enable ECDH between client and server)

Technical difficulties:

Data Format

The standardized crypto API has restrictions on the passed-in data type and passed-out data type. On the other hand, communication between the client and the server also has restrictions on the format of the data. For instance, there are conversions between Curve, Hex String and JSON objects. Therefore, there are complicated codes that work on the data format conversion.

getPublic() returns curve.base.basepoint, then encode to hex String in order to response it

```
    //Return the necessary data
    return res.status(200).json({ serverPublicKey: serverKeyPair.getPublic().encode('hex'), serverKey });
})
```

clientKeyPair.derive intakes curve.base.basepoint, so use **ec.keyFromPublic()** & **getPublic()** to convert the hex string back to curve.base.basepoint
shared secret in string format using **toString(16)**

```
//Encrypt the data
const sharedSecret = clientKeyPair.derive(ec.keyFromPublic(serverPublicKey, 'hex').getPublic()).toString(16);
```

AES encrypt for objects using **JSON.stringify()**

```
const encryptedData = CryptoJS.AES.encrypt(JSON.stringify(dataToSend), sharedSecret).toString();
```

AES encrypt for simple passcode strings

```
const encryptedData = CryptoJS.AES.encrypt(this.state.passcode, sharedSecret).toString();
```

Decrypt the data and use **JSON.parse()** to get back the object type.

```
//Decrypt the data
const bytes = CryptoJS.AES.decrypt(encryptedDataReturn, sharedSecret);
const decodedData = bytes.toString(CryptoJS.enc.Utf8)
const decryptedData = JSON.parse(decodedData);

this.setState({
    title: decryptedData.title,
    items: decryptedData.items,
    error: false,
    errorMessage: undefined,
    validPasscode: true,
    voteId: decryptedData.voteId,
    participantId: decryptedData.participantId,
    passcode: undefined,
    spinner: 'invisible'
});
```

Key storage

Since the key exchange process and vote operation is not synchronous, the clients' public key and its corresponding server key pairs need to be stored. The solution is to put them into Javascript.Map. Once the vote operation is finished, the client's public key and server key pair for that particular action is deleted from the Javascipt.Map immediately. The key exchange process and the vote operation are within the same function, so the overall operation finishes in a very short time, though they are not synchronous.

Define Maps to store the public keys and EC key pairs

```
//Gloabl Temp Variable
publicKeyMap = new Map(); //Hex strings
ECKeyMap = new Map(); //EC keys
```

During key exchange, store the server EC key pairs and client public key

```
//Save the private key and received public key
publicKeyMap.set(`publicKey_${serverKey}`, req.body.clientPublicKey);
ECKeyMap.set(`ECKey_${serverKey}`, serverKeyPair);
```

Once get the EC key paris and the public key, delete them from the Map

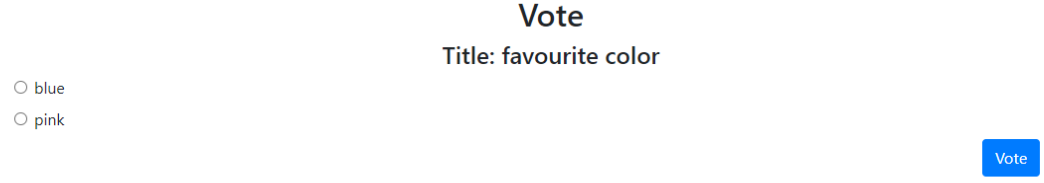
```
//Derive the secret key
const serverKeyPair = ECKeyMap.get(`ECKey_${req.body.serverKey}`);
const clientPublicKey = publicKeyMap.get(`publicKey_${req.body.serverKey}`);
const sharedSecret = serverKeyPair.derive(ec.keyFromPublic(clientPublicKey, 'hex').getPublic()).toString(16);

//Delete the specific item in the Maps
publicKeyMap.delete(`publicKey_${req.body.serverKey}`);
ECKeyMap.delete(`ECKey_${req.body.serverKey}`);
```

Key exchange and create vote are in the same function, the lifetime of the public key and the EC key pairs is about the time needed to run the codes below.

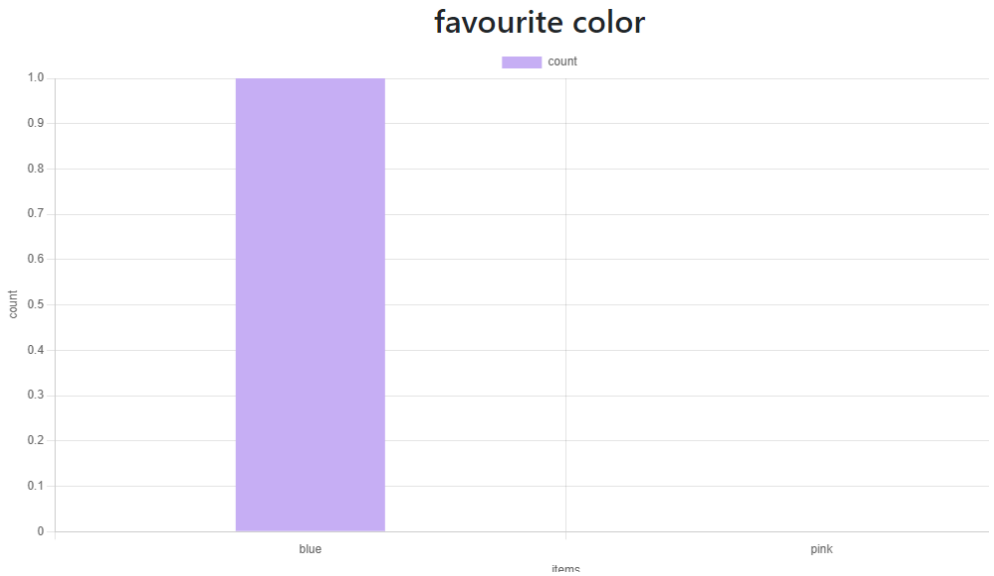
```
//Do key exchange
const resKeyExchange = await API.post('/keyExchange', { clientPublicKey: clientKeyPair.getPublic().encode('hex') });
const serverPublicKey = resKeyExchange.data.serverPublicKey;
const serverKey = resKeyExchange.data.serverKey;

//Encrypt the data
const sharedSecret = clientKeyPair.derive(ec.keyFromPublic(serverPublicKey, 'hex').getPublic()).toString(16);
const encryptedData = CryptoJS.AES.encrypt(JSON.stringify(dataToSend), sharedSecret).toString();
const authTagData = CryptoJS.HmacSHA256(encryptedData, sharedSecret);
const finalData = encryptedData + " " + authTagData;

//Send the data and create the vote via doing post request
await API.post('/createVote', { data: finalData, serverKey }, {});
```

Highlights:

Real-time results:



When the voter has finished his vote, the synchronized vote data will be presented on the check vote result function for the organizer.



Email:

Passcode to check the vote result 收件箱 ×

shingshing4350@gmail.com
收件者：我

You just created a vote.
Vote title: favourite color
Time Created: Fri May 14 2021 08:56:31 GMT+0000 (Coordinated Universal Time)
Here is the verfification code to check the vote results.
Verification Code: 609e3b3ed638d51aac14ab335PODUh'383
Check the vote results at http://ec2-52-77-255-166.ap-southeast-1.compute.amazonaws.com

When the vote event is created, the organizer will receive an email containing a unique passcode that can enter the function of the check vote result.

Your are invited to join a vote 收件箱 ×

shingshing4350@gmail.com
收件者：我

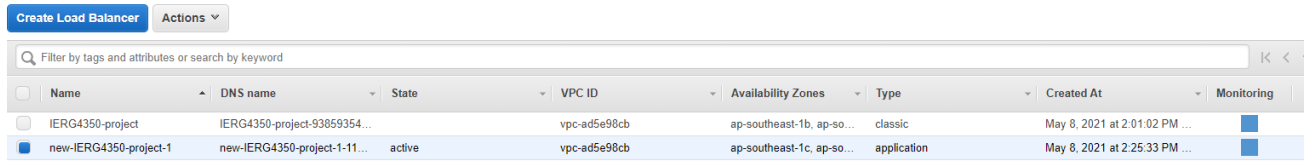
英文 › 中文（繁體） 翻譯郵件

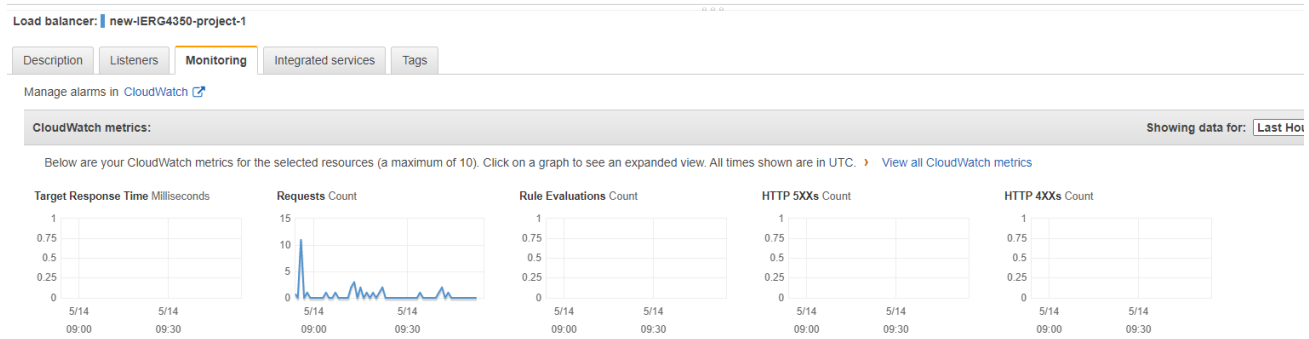
You are invited to join a vote.
Vote title: game
Here is the verfification code to join the vote results.
Verification Code: 609769dd50aa1766a6a38946)8p#Sq92q9
Join the vote at http://ec2-52-77-255-166.ap-southeast-1.compute.amazonaws.com

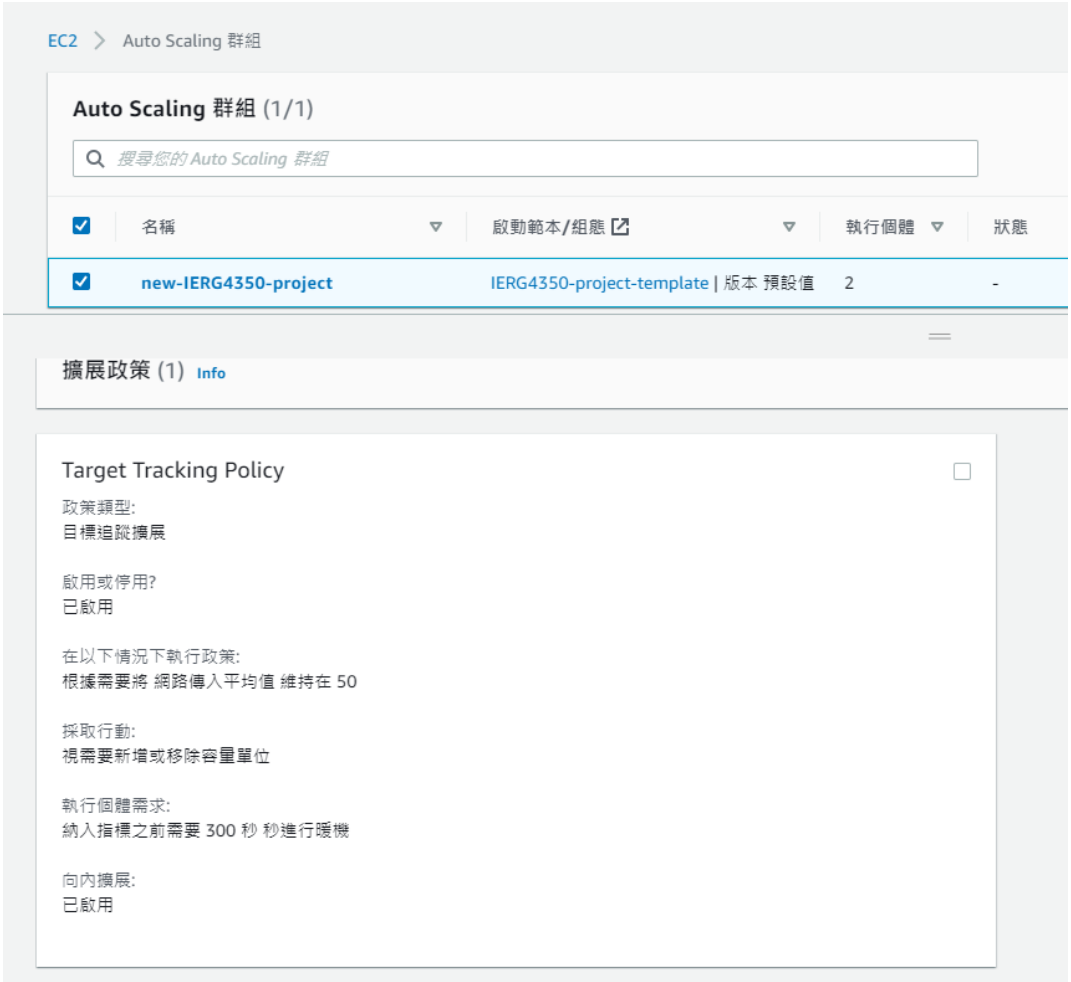
The voters will also receive an email containing a unique passcode that can enter the function of the join vote.

The email verification first helps our online voting website to do the identity checking. Without the email passcode, the voter and the organizer will be considered to be invalid users. Second, the email verification would not border the voters for registering private information.

Load Balancer and auto scaling group







The load balancer can distribute the incoming network to different servers. Therefore, if the average network incoming is higher than 50, it would be automatically scaled up.

Security considerations

Protecting data in transit

All the data is encrypted with AES cipher before sending out. For every single action, such as posting a passcode to check validation and getting the vote data from the server, the client and the server will first do ECDH key exchange and then do AES encryption or decryption. The data in transit is protected.

Client-side EC keys and key exchange and sent encrypted data

```
//Setting data to send
const dataToSend = { title, items: itemsToSend, emailOfCreator, participantEmails: participantEmailsToSend };

try {
    //Generate own EC key pair
    const clientKeyPair = ec.genKeyPair();

    //Do key exchange
    const resKeyExchange = await API.post('/keyExchange', { clientPublicKey: clientKeyPair.getPublic().encode('hex') });
    const serverPublicKey = resKeyExchange.data.serverPublicKey;
    const serverKey = resKeyExchange.data.serverKey;

    //Encrypt the data
    const sharedSecret = clientKeyPair.derive(ec.keyFromPublic(serverPublicKey, 'hex').getPublic()).toString(16);
    const encryptedData = CryptoJS.AES.encrypt(JSON.stringify(dataToSend), sharedSecret).toString();
    const authTagData = CryptoJS.HmacSHA256(encryptedData, sharedSecret);
    const finalData = encryptedData + " " + authTagData;

    //Send the data and create the vote via doing post request
    await API.post('/createVote', { data: finalData, serverKey }, {});

    //Set the spinner
    this.setState({ spinner: 'invisible' }, () => {
        alert('Created Successfully! Please check your email for the verification code to check the vote results.');
        this.setState({ created: true })
    });
```

Server key exchange request and response

```
//routes
app.post('/keyExchange', (req, res) => {
    //Generate server key pairs
    const serverKeyPair = ec.genKeyPair();

    //Set key to recognize the private key and public stored
    const serverKey = process.hrtime.bigint().toString();

    //Save the private key and received public key
    publicKeyMap.set(`publicKey_${serverKey}`, req.body.clientPublicKey);
    ECKeyMap.set(`ECKey_${serverKey}`, serverKeyPair);

    //Debugging for the Maps
    console.log(`publicKeyMap is ${[...publicKeyMap.keys()]}`);
    console.log(`EC Key Map is ${[...ECKeyMap.keys()]}`);

    //Return the necessary data
    return res.status(200).json({ serverPublicKey: serverKeyPair.getPublic().encode('hex'), serverKey });
})
```

Server get shared secret key

```
    //Derive the secret key
    const serverKeyPair = ECKeyMap.get(`ECKey_${req.body.serverKey}`);
    const clientPublicKey = publicKeyMap.get(`publicKey_${req.body.serverKey}`);
    const sharedSecret = serverKeyPair.derive(ec.keyFromPublic(clientPublicKey, 'hex').getPublic()).toString(16);
```

Check the auth tag first, then decrypt the data

```
  //Check the auth tag
  const encryptedData = req.body.data.split(' ')[0];
  const authTagReceived = req.body.data.split(' ')[1];

  const authTagCalculated = CryptoJS.HmacSHA256(encryptedData, sharedSecret);
  if(authTagCalculated.toString() !== authTagReceived.toString()) return res.status(400).send({ message: "Bad Auth Tag." })

  //Decrypt the data
  const bytes = CryptoJS.AES.decrypt(encryptedData, sharedSecret);
  const decodedData = bytes.toString(CryptoJS.enc.Utf8)
  const decryptedData = JSON.parse(decodedData);
```

Encrypt the data before returning to the client

```
    //Encrypt the data
    const encryptedDataToReturn = CryptoJS.AES.encrypt(JSON.stringify(itemToReturn), sharedSecret).toString();
    const authTagDataToReturn = CryptoJS.HmacSHA256(encryptedDataToReturn, sharedSecret);
    const finalDataToReturn = encryptedDataToReturn + " " + authTagDataToReturn;

    //Return the item
    return res.status(200).json(finalDataToReturn);
});
```

<u>Protecting data at rest</u>

For the data stored in the database, they are either encrypted or hashed. For the passcodes that used to join or check the vote, their hash value is stored in the database. For the vote details, they are encrypted with AES-256-GCM and store in the database. In other words, when looking at the database, we can know nothing of the vote details. To get the data maliciously, the attackers have to get the secret key stored on the server and also the data stored in the database which is under the protection of MongoDB.

Encrypting the data before storing it into the database

```
//Set the new vote title with encryption
const newTitle = encrypt(decryptedData.title, process.env.VOTE_TITLE_SECRET_KEY);

//Set the new vote items with encryption
const newItems = decryptedData.items.map(item => {
    return {
        name: encrypt(item, process.env.VOTE_ITEM_NAME_SECRET_KEY),
        count: encrypt("0", process.env.VOTE_ITEM_COUNT_SECRET_KEY)
    }
})
```

Database

Access Validation

Passcode to check the vote results and passcode to join the vote are respectively sent to the organizer's and participants' emails. The hash value of the passcode is stored in the database. If the hash value of the input passcode is different from the value stored in the database, the access operation is failed. The passcode is made up of 10 random bytes and the vote ObjectId. The only way for the attackers to get the access passcode is to access the participants' email server while this is nearly impossible because they cannot get the participants list and have to hack the participants email server.

All the errors will return the same error message as 'Invalid passcode'.

Unique passcode for every person (_id for us to recognize the target vote event)

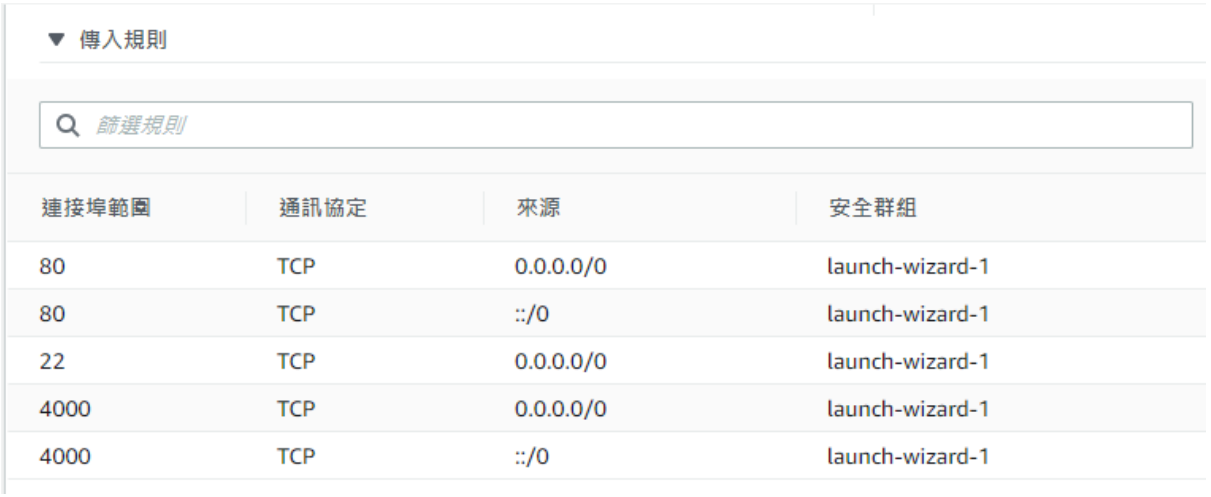
```
//Setting creator passcode
const creatorPasscode = savedVote._id +  randomize('*', 10);
console.log('original passcode creator: ', creatorPasscode);

//Hash the creator passcode
const salt = await bcrypt.genSalt(10);
const hashedCreatorPasscode = await bcrypt.hash(creatorPasscode, salt);

//Setting participants passcode and send emails to participants
const doHash = savedVote.participants.map(async (participant, index) => {
    //Setting participant passcode
    const participantPasscode = savedVote._id +  randomize('*', 10);
    console.log('Original passcode participant: ', participantPasscode);

    //Send email to the participant
    sendEmailToParticipant(decryptedData.participantEmails[index], savedVote.title, participantPasscode, website);

    //Hashing the participant passcode
    const salt = await bcrypt.genSalt(10);
    const hashedPasscode = await bcrypt.hash(participantPasscode, salt);

    //Setting the object to return
    const object = {
        participantPasscode: hashedPasscode,
        voted: participant.voted
    }
    return object;
});
const hashResult = await Promise.all(doHash);
```

Check if the passcode is valid using “bcrypt.compare”.
Return ‘Invalid passcode’ when errors occur.

```
//Check if the passcode is valid
const validPasscode = await bcrypt.compare(passcode, vote.creatorPasscode);
//Final return statement
// if(!validPasscode) return res.status(400).json({message: 'Invalid passcode.'});
//Debugging return statement
if(!validPasscode) return res.status(400).json({message: 'passcode not valid.'});
```

security group

We only allow port 22 for TCP connection, 80 for HTTP, and 4000 for server requests. Other ports are shut down to prevent anonymous access.



▼ 傳入規則

篩選規則

連接埠範圍	通訊協定	來源	安全群組
80	TCP	0.0.0.0/0	launch-wizard-1
80	TCP	::/0	launch-wizard-1
22	TCP	0.0.0.0/0	launch-wizard-1
4000	TCP	0.0.0.0/0	launch-wizard-1
4000	TCP	::/0	launch-wizard-1