IFS4205 Final Report - Team 2

SEMESTER I 2022-2023

Name	Matriculation Number
Ng Zhao Zhi Glendon	A0216960A
Deon Chung Hui	A0205227N
Bang Hee Kit	A0217010A
Sim Tian Boon	A0226131W
Lok Ke Wen	A0192344E
Zhou Qi	A0211230H

Table of Contents

1. Project Introduction

Public Hosting : https://ifs4205-gp02-1.comp.nus.edu.sg

Public Github Repo:
https://github.com/IFS4205-Group-2/IFS4205-AY2223-S1-G2-Track2Gather

Our team developed a contact tracing application, Track-2-gather, using an IoT dongle to trace the user's location when it comes near an IoT receiver stationed at a location.

The application features a user view where a notification is sent when in close contact with a possible carrier of the KIL-22 virus. Users can also log in to view if they are a close contact of a KIL-22 patient. Users will also be able to log in to upload their test kit results.

A different view is also available for Health authorities and Contact Tracers to view the traces of the contacts.

The application also hosts a publicly anonymized and curated list of data related to the KIL-22 patients. This can be publicly accessed by researchers for their research purposes.

The application is built based on DevSecOps and CI/CD frameworks - ensuring a security-by-design implementation of the application and its data. The security concerns addressed are primarily related to the security of the database, the provision of anonymised data for research purposes and the data transfer from IoT dongle to the servers along with its systems.

2. Testing Instructions

2.1 Web Application Testing

Roles	**Group A**	**Group B**
Public User	**Username**: PublicUser1 **Password**: REDACTED	**Username**: PublicUser2 **Password**: REDACTED
Contact Tracer	**Username**: ContactTracer0 **Password**: REDACTED	**Username**: ContactTracer3 **Password**: REDACTED

	Username: ContactTracer2 **Password**: REDACTED	**Username**: ContactTracer4 **Password**: REDACTED
Health Authorities	**Username**:HealthAuthority1 **Password**:REDACTED **Username**:HealthAuthority2 **Password**:REDACTED	**Username**:HealthAuthority3 **Password**:REDACTED **Username**:HealthAuthority4 **Password**:REDACTED

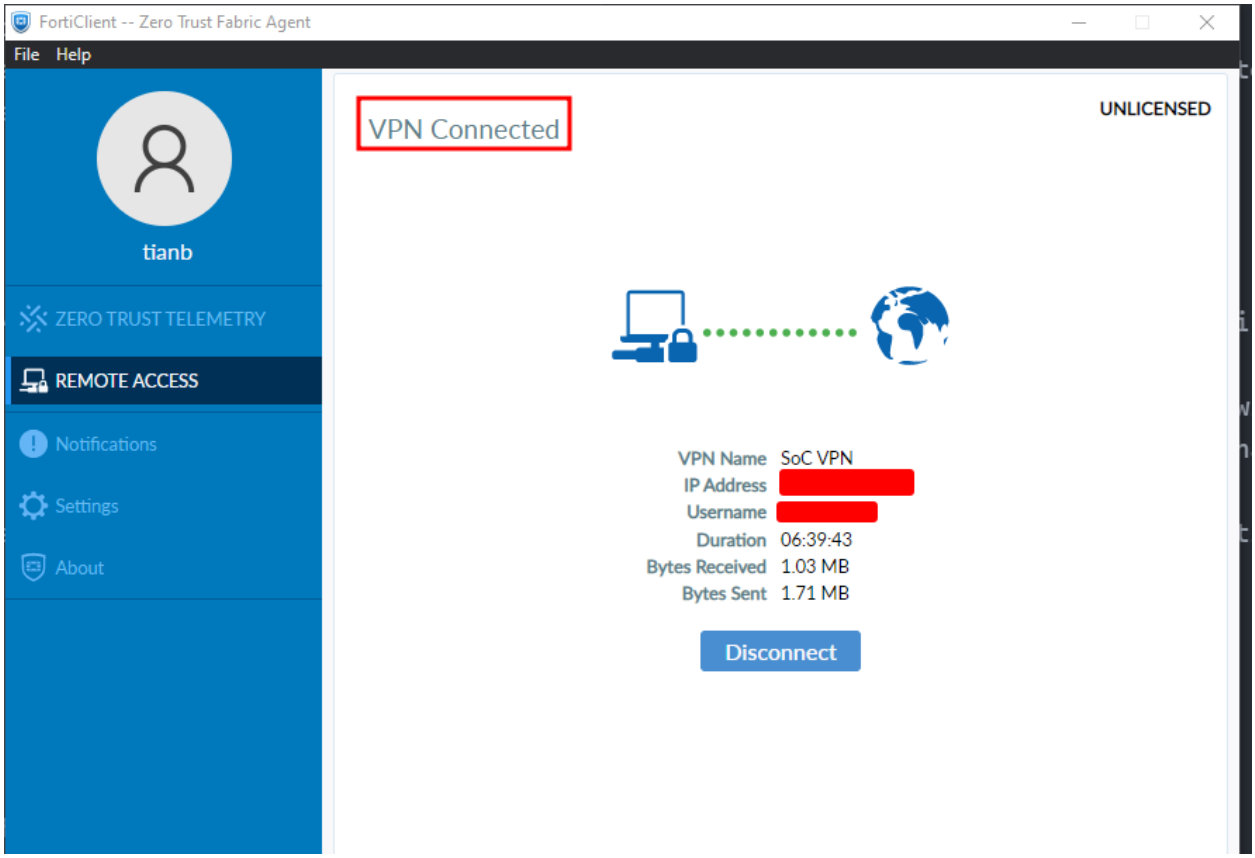
2.2 IoT Testing

2.2.1 Utility Service for PenTesting/Functional Testing

On the IoT server, there is a utility service that is not part of the actual functionality of Track-2-gather, hosted on port 4205. This utility has been provided to help and facilitate the testing of functionalities and the PenTesting process over the next 2 weeks. It provides whitelisting of IPs and is REQUIRED to start on testing the security claims and functional parts of the IoT Components.

Below are the steps needed to whitelist your IP address (on SoC VPN/NUS network).

1. First, connect to the SoC VPN with FortiClient. Make sure it is truly connected, before going to step 2.



2. Next, connect to the port 4205 by running
   ```
   nc ifs4205-gp02-5-i 4205
   ```
3. You should be prompted with the following

```
==========================================================================================================
IMPORTANT - This utility is only meant to facilitate PenTesting.
Do not attack this as this is not in scope.
==========================================================================================================
Please enter your IP address (ON NUS network, e.g. 192.168.232.100):
```

4. Lastly, provide the IP address obtained after connecting in step 1.
5. Whitelisted!

Then, you can now start with testing the functionalities and security claims!

2.2.2 IoT Subsystem Testing

To test the configured IoT Receiver, you can download the zip file here. Further details for testing can be found in the README.
If you are setting up the Receiver on your own, you can replace the `config.py` as instructed in the README with this.

Credentials provided for the IoT Subsystem (username:password):

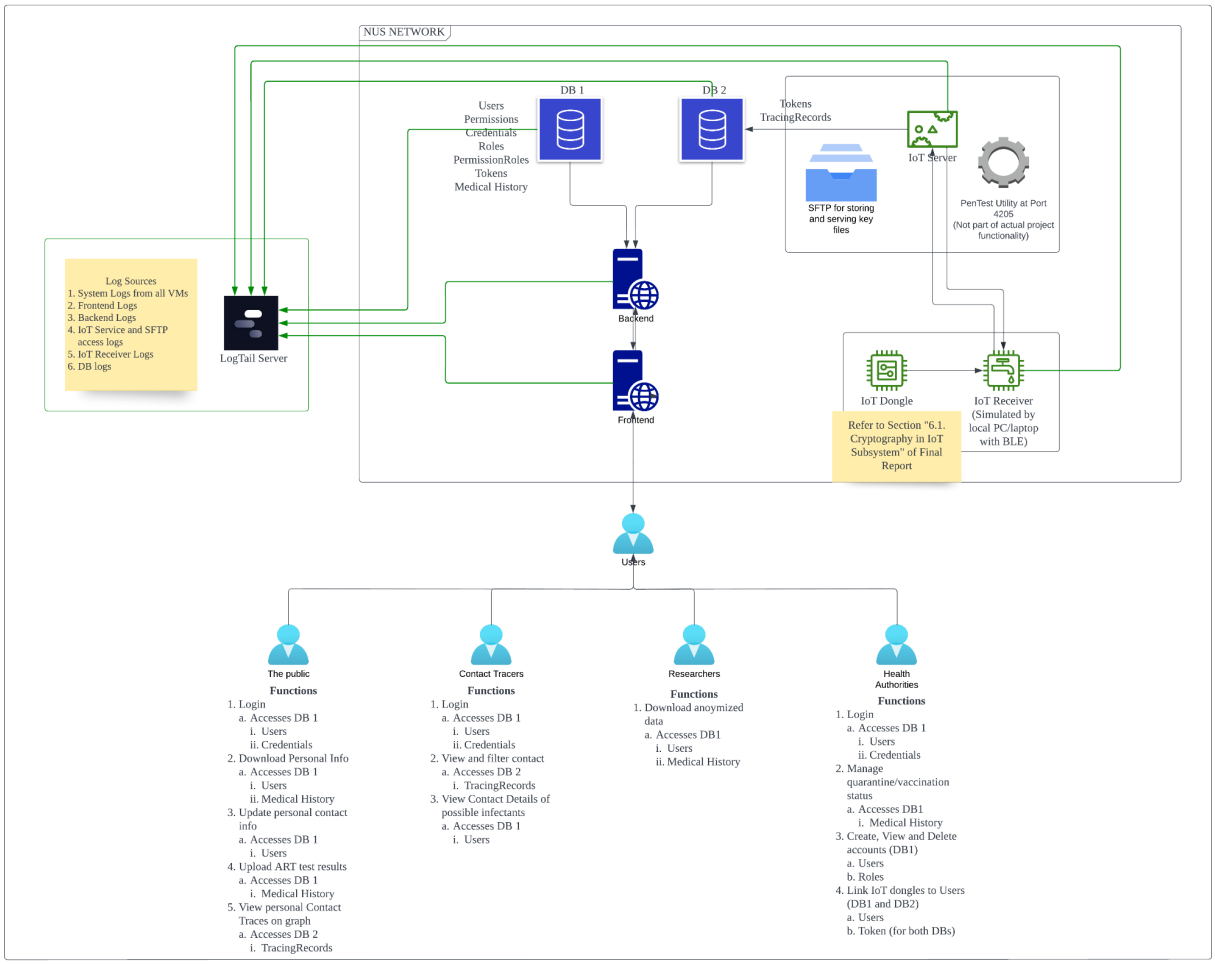
1. `sftp` user
    a. sftp:REDACTED
2. `iotsvc` user
    a. iotsvc:REDACTED
3. VM `Health Authorities` user:
    a. healthauth:REDACTED
4. VM `admin` user:
    a. admin:REDACTED

Each group will be given one Dongle to facilitate testing as well.

	Group A	**Group B**
Token ID	e9:fc:cd:62:0a:49	f6:4f:42:e6:42:43

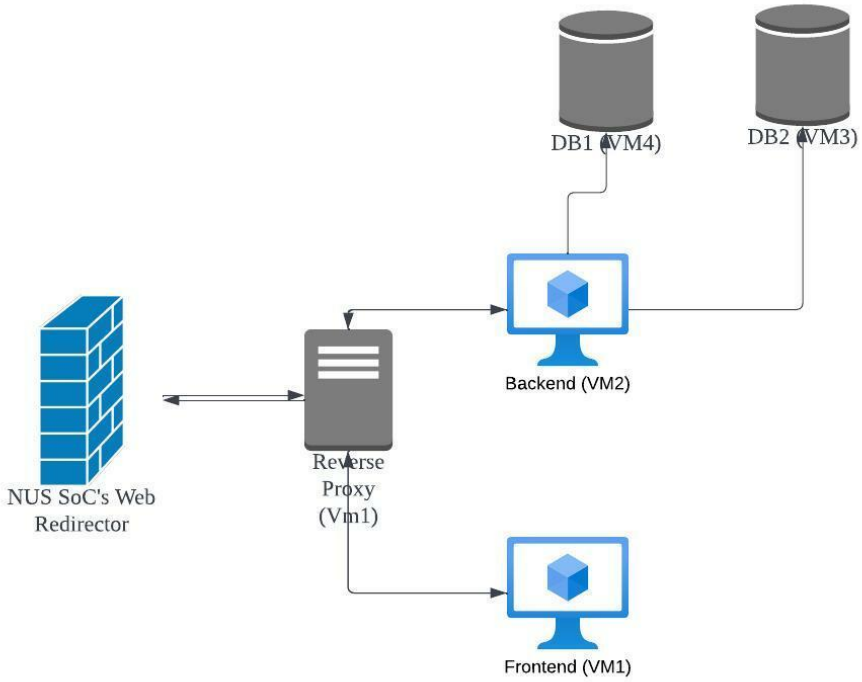
3. Architecture and Tools

3.1. Architecture, Design & User Roles



The architecture consist of 4 main subsystems

1. Frontend
2. Backend
3. Databases
4. IoT



Within the web application's subsystem, there consist of several components:

- Frontend (Hosted on VM1)
- Backend (Hosted on VM2)
- Reverse Proxy (Run from VM1)
- DB1 (Hosted on VM4)
- DB2 (Hosted on VM3)

The reverse proxy, hosted on port 80 of VM1 will be directing the traffic from users to the Frontend and Backend respectively.

The database subsystem consists of 2 databases hosted on 2 different servers. Database 1 stores user information and Database 2 stores data for contact tracing purposes. Users with different roles will have different access to the 2 databases. Refer to section 3.3 and 3.4 for more information.

The IoT Subsystem consists of the IoT dongle, receiver and server. The dongle will be the data source while the receiver will be the middle-man to ensure that all data sent from the dongle is securely and safely transported to the IoT server.

Additionally, the utility service on port 4205 is not part of the actual Track-2-gather, but a utility provided to help with testing over the next 2 weeks. Refer to section 2.2.1. for more information.

In Track-2-gather, there are also 4 different user roles (corresponding to real-world roles)

1. The public
2. Contract Tracers (CT)

3. Researchers
4. Health Authorities (HA)

3.2. Tools

3.2.1. Web

Broadly, the choice of technology for frontend is the ReactJS library while the backend is powered by ExpressJS in a NodeJS environment. ReactJS has great documentation and compliments ES6 node modules.. Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. Specifically, Express enables routing for our application, allowing RESTful interactions with our Frontend server.

Across both frontend and backend, both servers employ Json Web Tokens (JWT), stored on user's local storage to remain logged in after their first username-password authentication. JWTs are an open, industry standard and are chosen over the use of cookies due to cookies' susceptibility to Cross-site request forgery (CSRF) attacks and scaling limitations.

Further analysis and explanation of the security of choice of web tools are elaborated in the Frontend Subsystem and Backend Subsystem section.

3.2.1.1 Frontend (VM1)

The login page makes use of formik, a popular open-source form library for React applications. As Formik has a special configuration option/prop Yup, our login page incorporates Yup module to provide a form-level validation.

Although not a necessity in the scope of this module, our team has also used Chakra UI for the styling of our frontend pages. Chakra UI is a React components library with built-in accessibility and comes with a modern-looking, extendable and configurable design system.

The react-cytoscapejs module is used as a graphing library tool to visualize the network of close contacts for users, contact tracers and health authorities to view.

3.2.1.2 Backend (VM2)

Backend relies on a number of node modules to enable the endpoints for the Frontend server to communicate with. Node-postgres is used to connect from the backend server to the databases (VM 3 & 4).

Helmet middleware is also used alongside ExpressJS to harden our backend server by setting various HTTP headers, while Express Routing is used to prevent directory traversal by clients. The bcrypt module is used for hashing the given password with a salt before comparing it with the salted hash of the password in the *Credentials* database.

Lastly, the backend also uses Redis to cache the number of attempts a user has attempted to log in. The use of Redis eliminates the use of our databases which incurs a relatively expensive retrieval as compared to using Redis.

3.2.2. Database (VM3 & VM4)

Our web application interacts with PostgresSQL (psql) database. Psql is a powerful, open source object-relational database system with a majority of our team members having deep expertise working with the chosen language. Psql also offers comprehensive documentation that greatly assists in the development of our application. Additionally, our team uses pgTAP for unit testing on databases. pgTAP is a suite of database functions that allows developers to write TAP-emitting unit tests in psql scripts or xUnit-style test functions, and it allows us to verify the structure of our database schema, exercising procedures, functions, rules, and triggers that we write.

3.2.3. IoT

The IoT Receiver and Server uses mainly Python as the main language. The reason for choosing Python was that the functionalities such as socket-related functionalities can be easily implemented. Furthermore, Python is also simpler to use and has many available packages online. Thus, Python was chosen to be the main language for the receiver and the server. In particular, Python 3.8.10 is the chosen Python version for the IoT Receiver for compatibility with the Python Bluetooth package.

However, for the Dongle, it uses C/C++ in RFduino. There wasn't much consideration, since there were only limited options available for programming the Dongle.

In terms of the mode of data transmission, as the Dongle is Bluetooth enabled, we decided to use Bluetooth Low Energy (BLE) for communication between the IoT Dongle and Receiver. On the other hand, we decided to use TCP for communication between the IoT Receiver and Server as it provides reliable delivery. In terms of retrieving and storing files, we decided to use SFTP as it allows for authenticity and confidentiality.

3.2.4. Logging

The logs in all of our systems will be using LogTail which is a managed log management application developed by Better Stack. The purpose for choosing LogTail is because first, it provides the ease and convenience of not having to manage a log server which may be storage intensive. Secondly, LogTail provides benefits in the form of operational security as it provides an easy use of log filtering using SQL-like statements and customized alerts sent directly to email e.g. due to a potential DDoS attack or login brute force happening. Lastly, LogTail also provides visualization in the form of grafana which can help to visualize data for analysis on the operations.

3.2.5. DevOps

Use of github actions for DevOps workflow such as source code analysis, secrets management and automating unit testing.

3.2.6. DevSecOps

The first tool we used for DevSecOps was gitleaks, which ensures that confidential data such as passwords and API tokens are not leaked into the repository. The reason for choosing gitleaks was that there is already a substantial amount of rules used by default which is constantly updated. As of writing this, it was updated just 14 days ago.

The second tool we used was semgrep, which is a SAST tool which helps to analyze static code for vulnerabilities. The purpose for using semgrep over other SAST tools was that it was more suited to our project as most of our code base consist of Python and Javascript which is semgrep provides checking of. This also helps to simplify the DevSecOps process of the workflow by having a single tool rather than multiple tools for different language respectively.

The third tool used by our team for DAST is the OWASP ZAP's full scan, or ZAP full scan. We have decided to use this tool as it does not require any obtaining of tokens from the respective provider to use. Other than that, it is also simple and efficient to use. As such, we have deployed the ZAP full scan as our project's DAST which runs daily at 01:00 am (most likely PST, since Github Servers are running in this timezone) and generates a zip file containing the scan report in the respective Github Actions workflow for ZAP (workflow file named as DAST.yml).

4. Database Subsystem

4.1.Tables

Database 1

Credentials	
PK	**uid**
	password_hash
	username

Permissions	
PK	**pid**
	pname

Roles	
PK	**rid**
	rname

PermissionRoles	
PK	**(pid, rid)**

Tokens	
PK	**tid**
	assignedDate
	status

Users	
PK	**uid**
	nric
	name
	email
	contact_no
	gender
	date_of_birth
	address
	zipcode
	quarantineUntil
	accountStatus
	failedLoginFrequency
	tid
	rid

Medical History	
PK	**uid**
	vaccination_history
	recent_test_result

Database 2

Tokens	
PK	**tid**
	assignedDate
	status

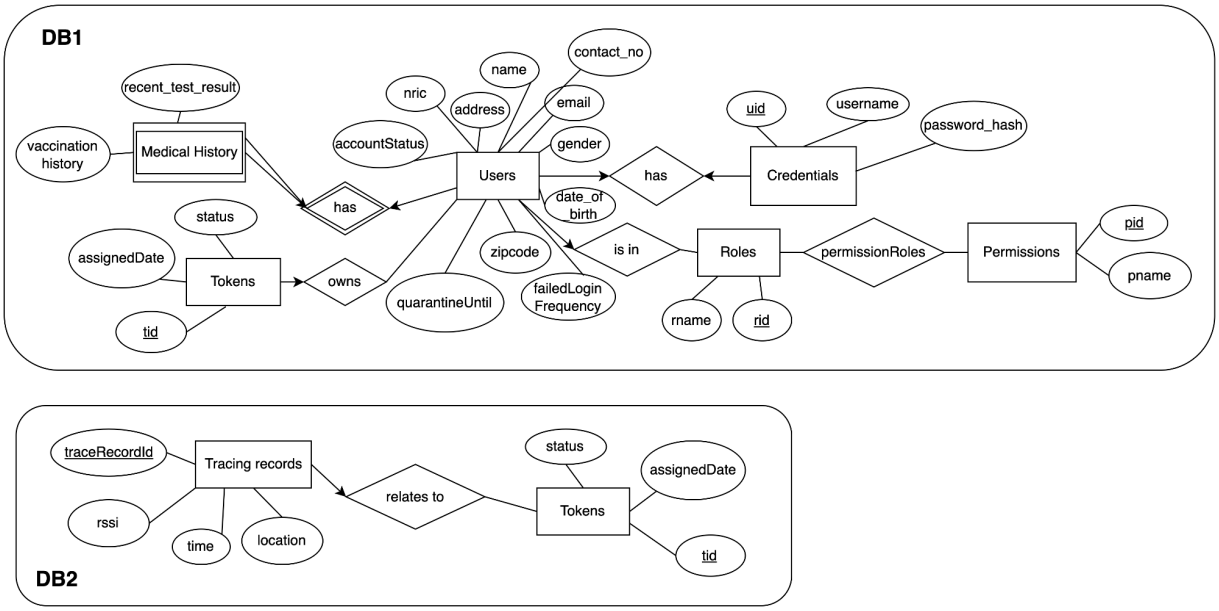
Tracing Records	
PK	**traceRecordId**
	tokenID
	time
	location
	rssi

Assumptions

1. Token can be reused but token (dongle) ID must be changed.
2. All tracing records for a token will be written to Tracing records before the token is deactivated.

Our system uses 2 databases. Database 1 stores user information, roles and their corresponding permissions, user credentials, token information and users' medical histories. This database can be accessed by the backend. Database 2 stores token information and tracing records, and it can be accessed by both backend and IoT server.

4.2. Entity Relationship Diagram



4.3. Core functions

Function	**Usage**
get_all_user_data_admin (int) get_all_role_data_admin (int) get_all_permission_data_admin (int) get_all_rolepermission_data_admin (int) get_all_token_data_admin (int) get_all_credential_data_admin (int) get_all_medical_history_data_admin (int) get_one_user_data_admin (int)	For the *admin user* to retrieve data.
get_one_user_data_tracer (int)	For *contact tracers* to view their own information.
get_all_user_data_tracer (int)	For *contact tracers* to view users' contact information (name, email, contact number and token ID).
get_one_user_data_authority (int)	For *health authorities* to view their own information.
get_health_data_authority (int)	For *health authorities* to view users' medical histories.

get_one_user_data_public (int)	For *registered users* without a specific role (role 'public') to view their own information.
get_all_user_data_researcher () get_all_health_data_researcher ()	For *non-registered users (researchers)* to view anonymized data.

*Basic functions for adding/editing/deleting data are omitted here.

4.4. Testing

For databases, we are using pgTAP for automated testing. Testing is done on database schemas and functions for both databases.

```
CREATE OR REPLACE FUNCTION public.testschema()
RETURNS SETOF TEXT LANGUAGE plpgsql AS $$
BEGIN
  RETURN NEXT has_table('roles');
  RETURN NEXT has_table('tokens');
  RETURN NEXT has_table('credentials');
  RETURN NEXT has_table('users');
  RETURN NEXT has_table('permissions');
  RETURN NEXT has_table('rolepermissions');
  RETURN NEXT has_table('medicalhistories');
END;
$$;

SELECT * FROM runtests('public'::name);
```

Example of schema testing done on DB 1

4.5. Security Considerations

In database 1, we implemented different functions to check users' roles before allowing them to access the data. Different users will have access to different data, by using functions in 4.3 (core functions). For example, a user with role 'contact tracer' will only be able to view users' contact information, but not their NRICs, genders and dates of birth. We limit users' access to raw data as much as possible and anonymize data for researchers by providing users' zip code instead of address, and year of birth instead of date of birth.

For database 2, we created a separate user for our IoT server. This user is only allowed to use functions assigned to it and only has access to data in the TracingRecords table but not the Tokens table. By having a least privileged user for the IoT server, we limit the possible data leakage even when the IoT server is compromised.

5. Frontend Subsystem

5.1. Landing Page of Web Application

```jsx
const Views = () => {
  const { user } = useContext(AccountContext);
  return user.loggedIn === null ? (
    <Text>Loading...</Text>
  ) : (
    <Routes>
      <Route path="/login" element={<Login />} />
      <Route path="/dashboard" element={
        <PrivateRoutes>
          <Dashboard />
        </PrivateRoutes>
      }/>
      <Route path="*" element={<Login />} />
    </Routes>
  );
};

export default Views;
```

Screenshot of Views.jsx

```jsx
return (
  <>
    <NavBar />
    <div style={{ paddingLeft: '32px', paddingRight: '32px' }}>
      { userRole === USER_ROLES.user
        ? <UserDashboard />
        : userRole === USER_ROLES.contactTracer
          ? <ContactTracerDashboard />
          : userRole === USER_ROLES.healthAuthority
            ? <HealthAuthorityDashboard />
            : null
      }
    </div>
  </>
);
```

Screenshot of Dashboard Element.

In the landing page of our website, the logic is driven by the above code, where the loggedIn status of the user is checked. Combined with Express Router, the user is redirected to the login

page if the user is not logged in. Invalid URL paths are covered by the last line where users are redirected to the login page as well. To ensure that only privileged users are able to access their respective dashboards, PrivateRoutes is used to encapsulate the protected routes.

The user's role is checked through retrieving their role ID located in the JWT, and set to the respective role based on setUserRole() function.

There are 2 ways which the user's loggedIn status will be changed:

1. Sending the server a valid JWT. This would mean that the signature contained in the JWT is valid and the token has not expired yet.
2. The user provides a valid username and password credential.

5.2. JWT Implementation

```
const UserContext = ({ children }) => {
  const [user, setUser] = useState({
    loggedIn: null,
    token: localStorage.getItem("token"),
  });
  const navigate = useNavigate();
  useEffect(() => {
    fetch("http://172.25.76.159:4000/auth/login", {
      credentials: "include",
      headers: {
        authorization: `Bearer ${user.token}`
      }
    })
      .catch(err => {
        setUser({ loggedIn: false });
        return;
      })
      .then(r => {
        if (!r || !r.ok || r.status >= 400) {
          setUser({ loggedIn: false });
          return;
        }
        return r.json();
      })
      .then(data => {
        if (!data) {
          setUser({ loggedIn: false });
          return;
        }
```

Screenshot of AccountContext.jsx

In the previous section, the user's loggedIn status was checked before redirecting the user to the appropriate landing page. As per the implementation of JWTs, the token in the user's local storage is retrieved and sent to the server's specified IP address and port number. If the

retrieved token is invalid (contains an invalid signature or the token has reached its expiry), or if no token is present in the client's local storage, the user's loggedIn status is set to false and is denied from accessing any other pages other than the log in page. Further analysis on the JWT generation is elaborated in the following subsystem section.

5.3. Login Page

```
const Login = () => {
  const { setUser } = useContext(AccountContext);
  const [error, setError] = useState(null);
  const navigate = useNavigate();


  return (
      <Formik
        initialValues={{ username: "", password: "" }}
        validationSchema={Yup.object({
          username: Yup.string()
            .required("Username required!")
            .min(6, "Username too short!"),
          password: Yup.string()
            .required("Password required!")
            .min(6, "Password too short!"),
        })}
```

Screenshot of Login.jsx

The login page relies heavily on the Formik module where user's input username and password are stored in a Formik object, with a validationSchema (using Yup) as defined above. The Yup module allows us to define a validationSchema easily and is set to have a requirement of 6 characters for both username and password. Users are also disallowed from submitting a form with either or both fields empty. A similar form validation check is performed on the server side as well, in the event adversaries may choose to target the server's opened IP address and port number.

There is no explicit encryption and message authentication code (MAC) in the frontend's code as the frontend code is made accessible to users via NUS School of Computing's Reverse Web Proxy. This enables the HTTPS protocol, protecting the users from man-in-the-middle eavesdropping and attacks.

5.4. Dashboard Pages

In our web application, we have a total of three different views of the dashboard for different user roles: Health Authority, Contact Tracer and User. There is no dashboard page for the researcher as he/she will only be required to access the publicly accessible medical data page to download anonymized data for research. For the health authority, we display necessary information and management pages for them to manage user accounts, token issuance, overall pandemic conditions and user vaccination status. Meanwhile, for contact tracers, we display contact tracing record information and other information needed for the contact tracers to complete their day-to-day tasks. Lastly, we display the user's COVID-19 contact tracing information, his/her vaccination status and test kit results, and personal information management page.

As seen from the content above, we can tell that there should be a different view for users with different roles to achieve less privilege principle. To do this, we split the dashboard into three different pages and render each page based on their roles. The main dashboard layout page will first call the backend server to verify the user's role and use the role information to display the respective pages. The extra call to the server is to ensure that the user's role is correct and not tempered, but at the expense of some computation and network overhead.

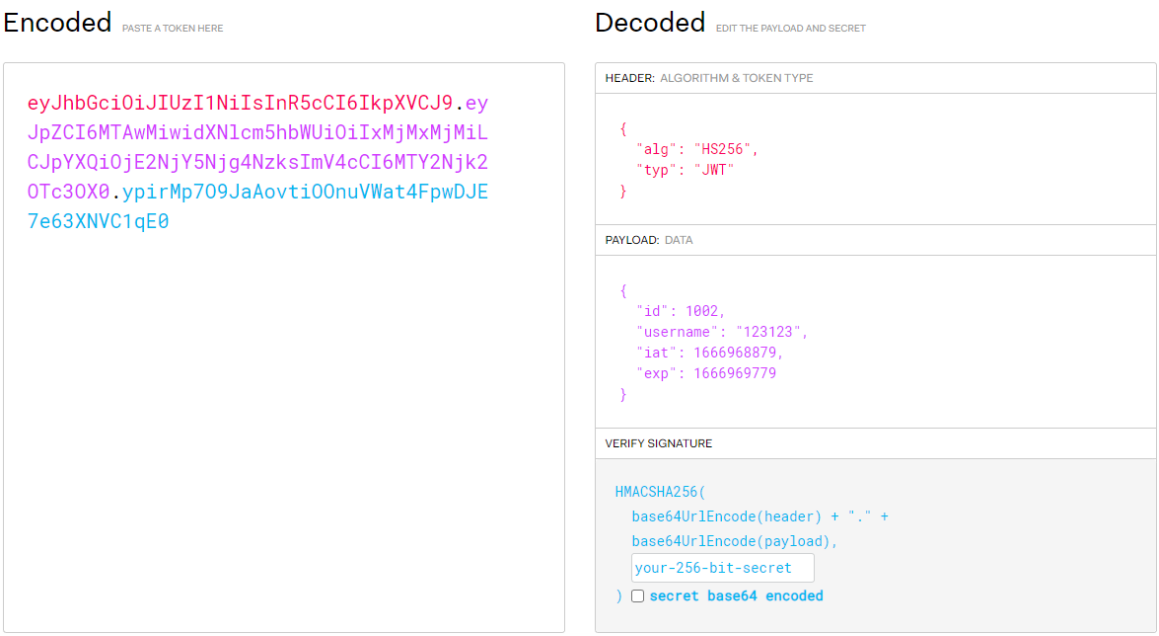
6. Backend Subsystem

6.1 JWT Generation

Snippet of .env file

The secret used in the signing of JWT is generated via NodeJS crypto.randomBytes(), returning a cryptographically secure 64 bytes/512 bits. This will provide 512 entropy bits, which is more than sufficient for a cryptographically strong signature to disable brute forcing.

6.2 JWT Format



Sample JWT from user '123123'

The fields of the JWT includes a unique ID identifier, id, and along with the user's username. The JWT also includes the expiry date which is checked upon for the validity of the token before users are logged in. Users with expired JWT are required to authenticate themselves by inputting a valid username and password in the Login page. The choice of signing algorithm is the industry standard of HS256 (HMAC with SHA-256).

6.3 API User Authentication

In our backend system, we complement the frontend service by providing a series of API services that enables the frontend to fetch information it needs from the database in a centrally managed manner. The backend server is only accessible from the frontend side and the respective database server. As we have multiple roles in the system and each role will have different levels of access to data, we implemented an authentication method on the backend server before processing a privileged API request. Below is a figure showing an example of backend authentication code:

```
const token = getJwt(req);

if (!token) {
  res.json({ loggedIn: false });
  return;
}

jwtVerify(token, process.env.JWT_SECRET)
  .then(async decoded => {
    // If account role is not user
    if (decoded.roleid !== 3) {
      res.status(400);
      res.json({ status: 'Forbidden' });
      return;
    }
```

Screenshot of handleUserInfoUpdate.js

In the sample code above, we can see that a token is retrieved from the user request using the "getJwt" function. This function essentially retrieves the JWT token attached to all request's "Authorization" header. The retrieved token is then verified using "jwtVerify" function with the JWT secret used to generate the token. If the token is verified, further checks such as the user's role is validated to ensure he/she indeed has the permission to call the specified API.

6.4 Secure Storage in Credentials

```
db1=# select * from credentials where uid = 1002;
 uid  |                        password_hash                         | username
------+--------------------------------------------------------------+----------
 1002 | $2b$10$jrUVh8rzQ0Hx99XZU/MdKO9U6X/CHUHeebCS82YiAFAqVKoEylSoO | 123123
(1 row)

db1=# select * from credentials where uid = 1003;
 uid  |                        password_hash                         | username
------+--------------------------------------------------------------+----------
 1003 | $2b$10$nBe4LjBJaIh0u2Zt7S.b5eK8/hCDdSp4b.SMu8F7nGc8zHERgfcQC | 123124
(1 row)
```

Screenshot of Credentials database of 2 users with same password

The Backend subsystem also exercises best practices of storing the salted hash of the user's password before sending it to DB1 for storage and retrieval. Above is an image of 2 users, '123123' and '123124', both registered using the same password but are stored differently due to the concatenation of the salt before hashing. Storing salted hashes of passwords also hardens our database in the event of a data breach where the passwords of users are not compromised.

6.5 Username-Password Authentication

```
const attemptLogin = async (req, res) => {
  const potentialLogin = await pool.query(
    "SELECT uid, password_hash, username FROM credentials c WHERE c.username=$1",
    [req.body.username]
  );
```

When users attempt to log in via username-password authentication, the given username is used in the above query to retrieve matching usernames in the database. In such query, it is prone to injection attacks (i.e. SQL Injection) if the input from users are not sanitized. Therefore, our query is parameterised to prevent susceptibility to SQL Injection Attacks.

```
if (potentialLogin.rowCount > 0) {
  const isSamePass = await bcrypt.compare(
    req.body.password,
    potentialLogin.rows[0].password_hash
  );
  if (isSamePass) {
    const ip = req.headers["x-forwarded-for"] || req.connection.remoteAddress;
    logtail.info("User " + req.body.username + " in via login page.", {
      ipaddress: ip
    });
```

If the query yields a return from the database, bcrypt.compare() is called to compare the hash of the salted password given by the user with the salted hash in the database for the corresponding user. If the correct password is supplied by the user, we log the information that the specific user has logged into the application, accompanied by their IP address. Information and format of the Login logs is shown below.

```
2022-10-29 14:39:49.995 [backend] [INFO] User 123123 in via login page.
▾ context
  › runtime
  › system
"dt": "2022-10-29T06:39:49.995Z"
"ipaddress": "::ffff:192.168.152.48"
"level": "info"
"message": "User 123123 in via login page."
Expand all  Show nulls  Copy JSON  Visualize
```

```
jwtSign(
  {
    id: potentialLogin.rows[0].uid,
    username: req.body.username,
  },
  process.env.JWT_SECRET,
  { expiresIn: "20mins" }
)
```

After logging, the backend server signs a new JWT with the user's unique ID and username with a 512 bits secret, along with the set expiry time. In line with Oracle's guidelines, we have set the given JWT to last for 20 minutes. With the given JWT, users do not have to repeat the username-password authentication as long as they possess the valid JWT in their local storage.

6.6 Server-Side validation

```
const Yup = require("yup");

const formSchema = Yup.object({
  username: Yup.string()
    .required("Username required")
    .min(6, "Username too short"),
  password: Yup.string()
    .required("Password required")
    .min(6, "Password too short"),
});

const validateForm = (req, res, next) => {
  const formData = req.body;
  formSchema
    .validate(formData)
    .catch(() => {
      res.status(422).send();
    })
    .then(valid => {
      if (valid) {
        next();
      } else {
        res.status(422).send();
      }
    });
};
```

Screenshot of validateForm.js

Anticipating adversaries that might try to bypass the frontend's form validation by sending packets straight to the IP address of our backend server, the backend server also implements a validation check for incoming direct login attempts.

6.7 Rate Limiting

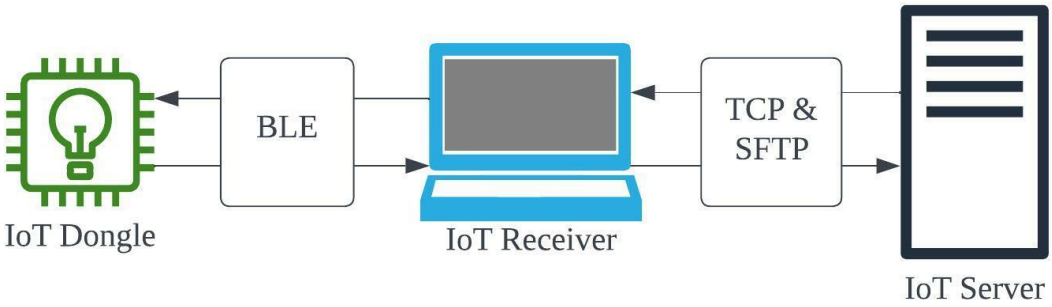
```
module.exports.rateLimiter =
  (secondsLimit, limitAmount) => async (req, res, next) => {
    const ip = req.headers["x-forwarded-for"] || req.connection.remoteAddress;
    [response] = await redisClient
      .multi()
      .incr(ip)
      .expire(ip, secondsLimit)
      .exec();

    if (response[1] > limitAmount) {
      logtail.warn("Bruteforce login attempt detected!", {
        ipaddress: ip
      });

      res.json({
        loggedIn: false,
        status: "Slow down!! Try again in a minute.",
      });
    }
    else next();
  };
```

The backend server also uses Redis to cache the number of login attempts a user has made. This is done through a key-value pair where the key is the user's IP address and the corresponding value starts from 0, incrementing each time a login attempt is made by the user. For our application, we have set the limit to be 10 attempts in 60 seconds.

The user will have to wait for a full 60 seconds without making any login attempts before the user can make another attempt to login. If the user attempts to login before the set time limit of 60 seconds is up, the timer is reset and the user will have to wait another 60 seconds before attempting.

A log of the bruteforce attempt is also made for every attempt above the limit. The log includes the IP address from which the brute force is being attempted by, allowing the blacklisting of IP addresses, if necessary.

7. IoT Subsystem



IoT Architecture Diagram

As mentioned, the IoT Subsystem consists of three main components: IoT Server, IoT Receiver and IoT Dongle. An IoT Dongle is a Bluetooth-enabled device which serves to report to an IoT Receiver when it is within range of it. An IoT Receiver acts as a middleman which will transmit the data to the IoT Server. The IoT Server is responsible for the generation of the keys used in the data transmission and for sending the data to the database upon successful authentication of the data received.

7.1. Cryptography in IoT Subsystem

This section will cover the types of Cryptography used for the components in the IoT Subsystem. Detailed explanation of how the Cryptography is used can be found in the following sections.

Component	Keys
IoT Dongle	Ed25519 default keypair (Server-Generated, 256-bits keypair) Used for signing when syncing
	Ed25519 individual keypair (Dongle-Generated, 256-bits keypair) Used for signing messages
	ECC (SECP160r1) ephemeral keypair (Dongle-Generated, 320-bits for public key, 168-bits for private key) Used for ECDH between Dongle and Receiver
	Symmetric Key (Generated from ECDH, 128-bits key)

	Used for encryption via AES-GCM
IoT Receiver	Ed25519 default keypair (Server-Generated, 256-bits keypair) Used for signing messages
	ECC (SECP160r1) ephemeral keypair (Receiver-Generated, 320-bits for public key, 168-bits for private key) Used for ECDH between Dongle and Receiver
IoT Server	RSA keypair (Server-Generated, 4096-bits keypair) Used for encryption using PKCS#1 OAEP

We chose Ed25519 as a digital signature algorithm which has a 256-bits key length as it provides a 128-bits security which is cryptographically secure. More importantly, Ed25519 is based on Elliptic Curve Cryptography (ECC) which allows for quick computation during signing and verification. This is essential especially for the IoT Dongles as they are lacking in computational power.

Similarly, our ECC ephemeral keypairs are based on the curve SECP160r1. This curve provides 80-bits security as a tradeoff for the efficiency in computation that it provides, which is very necessary for the IoT Dongle.
The symmetric key generated in this key exchange is generated with SHA256, which is a cryptographically secure hash algorithm. The symmetric key is used in AES GCM mode, with a 128-bits security and cryptographically secure, while still being efficient.

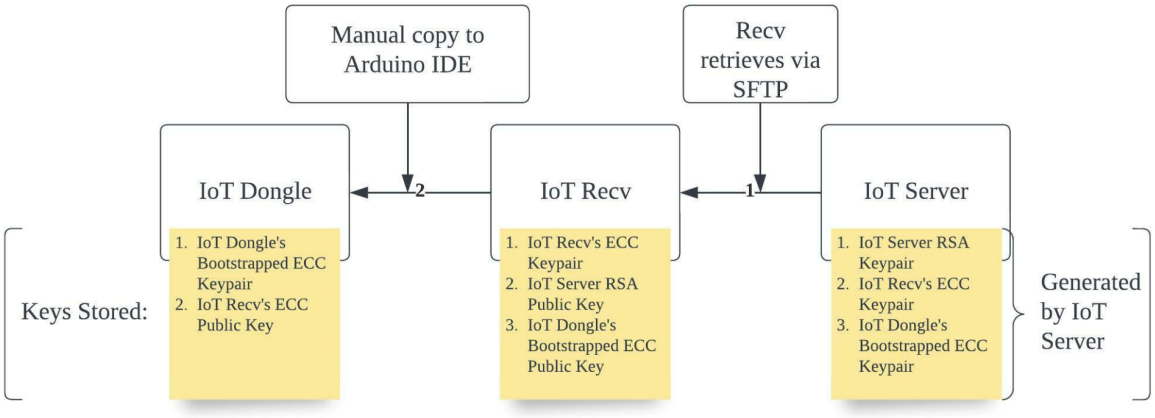
The choice of RSA keypair using 4096 bits helps to mitigate the chances of the encryption from being cracked and revealing the data in transit. Furthermore, it is also worth the tradeoff of efficiency for security by having a longer key length.

7.2. Setup of IoT Receivers and Dongles

An IoT Receiver is simulated using a virtual machine. The virtual machine is password-protected which is only known to the Health Authorities. The Receiver will obtain its default keypair from the Server via SFTP. This keypair will be used to sign the data sent to a Dongle. All Receivers will possess this same keypair generated from the Server. However, each Receiver will have their own individual configuration file, which contains both general details, such as the IP address of the IoT Server, and specific details, such as the location the Receiver is placed in.

The setup of the IoT Receivers and Dongles are based on the following two processes: Bootstrap and Sync.

7.2.1. Bootstrap Process



IoT Bootstrap Process Diagram

The files used for this process are setup.sh and bootstrap.py.

setup.sh essentially calls *bootstrap.py* on top of other Receiver configuration not related to this process.

```
print("Retrieving Keys from Server...")
with pysftp.Connection(sftpServer, username=config.sftp_user, password=config.sftp_password) as sftp:
    # To get files from server.
    with sftp.cd('/Server'):           # Replace with Server Directory
        sftp.get('serverPubKey.pem', f"{cwd}/ServerGenKeys/serverPubKey.pem")         # get a remote file
print("Retrieved serverPubKey.pem")

with pysftp.Connection(sftpServer, username=config.sftp_user, password=config.sftp_password) as sftp:
    with sftp.cd('/ServerGenKeys'):           # Replace with Server Directory
        for filename in ['recvPrivateKey.pem', 'recvPublicKey.pem']:
            sftp.get(filename, f"{cwd}/ServerGenKeys/{filename}")
            print("Retrieved " + filename)
```

bootstrap.py - Retrieving key files

Each Receiver will run a shell script that will automatically retrieve the files needed for the workflow from the Server via SFTP.

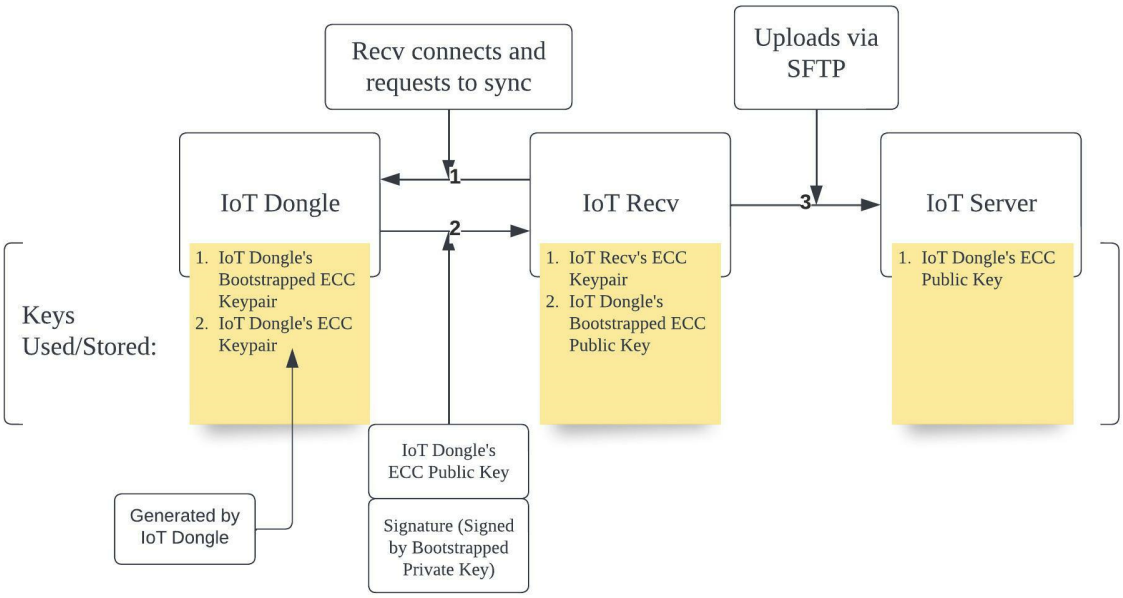
In addition, there will be a dedicated Receiver which will be responsible for setting up the IoT Dongles as well. This is done by specifying a command line argument when running the shell script mentioned (`./setup.sh all`).

```
if len(sys.argv) > 1 and sys.argv[1] == "--all":
    with pysftp.Connection(sftpServer, username=config.sftp_user, password=config.sftp_password) as sftp:
        with sftp.cd('/ServerGenKeys'):           # Replace with Server Directory
            for filename in ['defaultPrivateKey.pem', 'defaultPublicKey.pem']:
                sftp.get(filename, f"{cwd}/ServerGenKeys/{filename}")
                print("Retrieved " + filename)

    print("....................................................................")
    # Implement printing and deleting of default keys
    print("Please copy this private key into the Arduino code, variable def_pri_key:")
    printPrivateKeyAsIntArray('defaultPrivateKey.pem')
    print("....................................................................")

    print("Please copy this public key into the Arduino code, variable def_pub_key:")
    printKeyAsIntArray('defaultPublicKey.pem')
    print("....................................................................")

    print("Please copy this public key into the Arduino code, variable verify_recv_key:")
    printKeyAsIntArray('recvPublicKey.pem')
    print("....................................................................")

    if os.path.isfile(f"{cwd}/ServerGenKeys/defaultPrivateKey.pem"):
        os.remove(f"{cwd}/ServerGenKeys/defaultPrivateKey.pem")
```

bootstrap.py - Printing Dongle Keys to be pasted in Arduino IDE

The setup of an IoT Dongle revolves around uploading the sketch to the Dongle via the Arduino IDE. When running the shell script, the dedicated Receiver will be presented with the keys to be pasted into the sketch in the IDE. Upon proper configuration, the sketch can be uploaded to the Dongle and it will be ready to transmit data.

7.2.2. Sync Process



IoT Sync Process Diagram

The file used in this process is sync.py.

Once all of the IoT Dongles have been set up, the dedicated Receiver can sync up the Dongles with the Server, thus completing the overall setup, allowing the normal workflow to begin.

When the IoT Dongles have been set up, they will generate their own ECC keypair (Ed25519) to be used for signing messages.

```python
def startScan(verifier):
    print("Scanning for 30 seconds...")
    for adv in ble.start_scan(ProvideServicesAdvertisement, minimum_rssi=-95, timeout=30):
        if GMS in adv.services:
            mac = adv.address.string
            try:
                if found[mac] == False:
                    print(mac)
                    found[mac] = True
                    try:
                        GMS_connection = ble.connect(adv)
                        connectToDongle(GMS_connection, mac, verifier)
                    except KeyboardInterrupt:
                        if GMS_connection and GMS_connection.connected:
                            GMS_connection.disconnect()
                        ble.stop_scan()
                        exit()
                    except:
                        print("Failed to retrieve key, retrying...")
                        found[mac] = False
                        if GMS_connection and GMS_connection.connected:
                            GMS_connection.disconnect()
            except:
                pass
        if all(value == True for value in found.values()):
            break
    if any(value == False for value in found.values()):
        for mac in found.keys():
            if found[mac] == False:
                print(f"{mac} was not found in the scan. If this mac address is to be synced, please try again.")
    ble.stop_scan()
```

sync.py - Searching for Dongles to sync with

This syncing process is done by first searching for and connecting with the IoT Dongles present via BLE.

```python
"""
Use the bootstrapped public key to verify the signature
Retrieve the generated public key of the IoT Dongle in clear
"""
def receiveAndVerifyPublicKey(GMS_connection, verifier):
    GMS_service = GMS_connection[GMS]
    GMS_service.write(b"sync")
    try:
        msg = GMS_service.read(256).rstrip(b'\x00')
        signature = msg[:SIG_BYTES]
        publicKey = msg[SIG_BYTES:]
        verifier.verify(publicKey, signature)
    except (ValueError, AttributeError) as e:
        raise Exception("Packet Error")
    return publicKey
```

sync.py - Receiver request to sync and verifying signature

Upon connection to a Dongle, the Receiver will send a message to request for syncing. The Dongle will then send its generated public key, along with the signature signed with the default private key to the Receiver.

```
"""
Save the generated public key of the IoT Dongle in PEM format
"""
def writeToPEM(publicKey, mac):
    donglePublicKey = eddsa.import_public_key(publicKey)
    pubKeyPem = donglePublicKey.export_key(format='PEM')
    filename = mac.replace(":", "") + ".pem"
    with open(serverDirectory + filename, 'w') as f:
        f.write(pubKeyPem)

    with pysftp.Connection(sftpServer, username=config.sftp_user, password=config.sftp_password) as sftp:
        # To store/upload files to server.
        with sftp.cd('/Dongles'):
            sftp.put(serverDirectory + filename)
            print("Sent to IoT Server")

    if os.path.isfile(serverDirectory + filename):
        os.remove(serverDirectory + filename)

def connectToDongle(GMS_connection, mac, verifier):
    if GMS_connection and GMS_connection.connected:
        publicKey = receiveAndVerifyPublicKey(GMS_connection, verifier)
        writeToPEM(publicKey, mac)
    GMS_connection.disconnect()
```

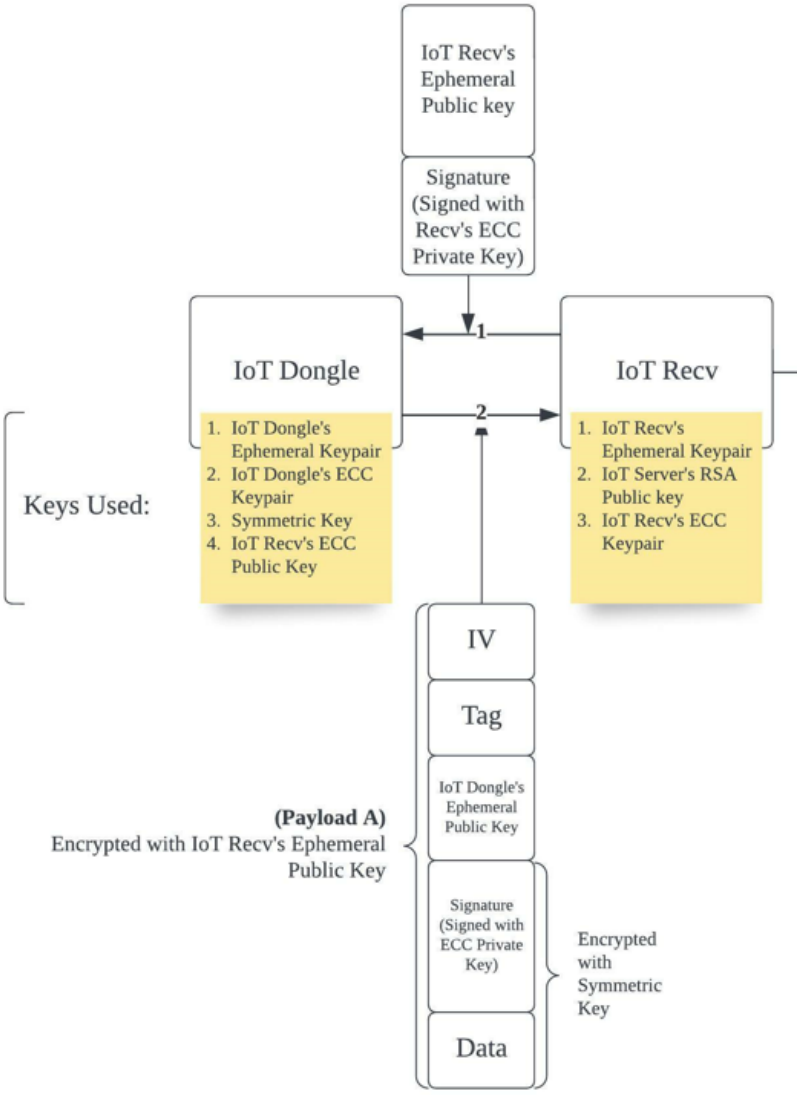
sync.py - Uploading via SFTP

The Receiver will then verify the signature and send the public key file to the IoT Server via SFTP if verified. This process ensures that the IoT Server receives the most updated public key files of the IoT Dongles.

7.3. Data Workflow

The files used in this process are IoTRecv.py and IoTDongle.ino. The Python file is meant to be run as a background process. All logging will be sent to LogTail under “IoT Receiver”.

7.3.1. Data collection from IoT Dongles



IoT Receiver - Dongle Workflow Diagram

When an individual carrying an IoT Dongle walks near an IoT Receiver (maximum distance depends on the strength of the Bluetooth capabilities and the environment), the Receiver will attempt to establish a connection with the Dongle. The time needed for a Receiver to detect and attempt a connection to the Dongle might take up to thirty seconds, which means a transient contact is likely to be insufficient to register a connection.

```
def connectToDongle(GMS_connection, mac, serverPubKey, signer):
    if GMS_connection and GMS_connection.connected:
        logDongleConnection(mac)
        privKey, pubKey = generateEphemeralKeys()
```

```
"""
Generate Ephemeral Keys on the IoT Recv to perform key exchange
"""
def generateEphemeralKeys():
    privKey = SigningKey.generate(curves.SECP160r1)
    pubKey = privKey.verifying_key
    return (privKey, pubKey)
```

IoTRecv.py - Generating Ephemeral Keypairs for ECDH

Once the connection is established, the Receiver will generate an ephemeral keypair using ECC with the curve "SECP160r1". It will then continue to perform an ECDH key exchange with the Dongle to ensure that both parties will have a shared symmetric secret key, which will be described below.

```
"""
Send to Dongle the Ephemeral public key along with the signature
Receive the encrypted payload from the Dongle (This payload could have some errors a
"""
def writeReadToDongle(GMS_connection, pubKey, signer):
    GMS_service = GMS_connection[GMS]
    pubKeyBytes = pubKey.to_string()
    sig = signer.sign(pubKeyBytes)
    GMS_service.write(pubKeyBytes + sig)
    time.sleep(5)
    msg = GMS_service.read(256).rstrip(b'\x00')
    return msg
```

IoTRecv.py - Sending (Ephemeral Public Key + Signature) to Dongle

In Process 1 as shown in the diagram above, the Receiver will use its default private key (generated by the Server) to sign the generated ephemeral public key. It will then send the public key and signature to the Dongle via BLE.

```
263     memcpy(recv_msg + public_count, data, len);
264     public_count += len;
265     if (public_count < PUB_KEY_BYTES + SIG_BYTES) {
266       return;
267     } else {
268       public_count = 0;
269     }
270     recv_msg[PUB_KEY_BYTES + SIG_BYTES] = '\0';
271     uint8_t recv_sig[SIG_BYTES];
272     memcpy(pub_recv, (uint8_t *)recv_msg, PUB_KEY_BYTES);
273     memcpy(recv_sig, (uint8_t *)(recv_msg + PUB_KEY_BYTES), SIG_BYTES);
274     if (!Ed25519::verify(recv_sig, verify_recv_key, pub_recv, PUB_KEY_BYTES)) {
275       Serial.println("Invalid Signature");
276       return;
277     }
```

IoTDongle.ino - Verifying IoT Receiver's Signature

When the Dongle receives the public key and signature, it will use the Receiver's default public key to verify the signature.

```
/**
 * Generates a random iv and the symmetric key using ECDH after generating a ephemeral keypair.
 *
 * @param pub_key The integer array where the ephemeral public key will be stored in
 * @param pub_recv The ephermeral public key received from the Receiver
 * @param key The integer array where the generated symmetric key will be stored in
 * @param iv The integer array where the generated iv will be stored in
 *
 * @return 1 if no errors occurred, 0 if there is an error generating either the keypair or the symmetric key
 */
int generateKeys(uint8_t *pub_key, uint8_t *pub_recv, uint8_t *key, uint8_t *iv) {
  uint8_t priv_key[PRI_KEY_BYTES];
  uint8_t secret[SEC_KEY_BYTES];
  int error = uECC_make_key(pub_key, priv_key, curve);
  if (!error) {
    Serial.print("make key() failed (1)\n");
    return 0;
  }

  int r = uECC_shared_secret(pub_recv, priv_key, secret, curve);
  if (!r) {
    Serial.print("shared_secret() failed (1)\n");
    return 0;
  }

  SHA256 hash = SHA256();
  hash.update(secret, SEC_KEY_BYTES);
  hash.finalize(key, KEY_BYTES);

  for (int i = 0; i < KEY_BYTES; i++) {
    uint8_t val;
    RNG.rand(&val, 1);
    iv[i] = val;
  }

  return 1;
}
```

IoTDongle.ino - Generating Keys

If verified, the Dongle will then generate another ephemeral keypair under the same curve. The Dongle will then take its ephemeral private key and the Receiver's ephemeral public key to produce a shared secret, which will then be hashed using SHA256 to obtain a 128-bit symmetric key.

```
288     char char_arr_rssi[MSG_BYTES];
289     uint8_t int_arr_rssi[MSG_BYTES];
290     int actual_msg_bytes = convertToString(char_arr_rssi, int_arr_rssi);
291
292     uint8_t sig[SIG_BYTES];
293     Ed25519::sign(sig, pri_sign_key, pub_sign_key, char_arr_rssi, actual_msg_bytes);
294
295     int pt_bytes = SIG_BYTES + actual_msg_bytes;
296     uint8_t combined_pt[pt_bytes];
297     uint8_t ct[pt_bytes];
298     uint8_t tag[KEY_BYTES];
299
300     memcpy(combined_pt, sig, SIG_BYTES);
301     memcpy(combined_pt + SIG_BYTES, int_arr_rssi, actual_msg_bytes);
302
303     encryptAndTag(key, iv, ct, combined_pt, tag, pt_bytes);
```

```
/**
 * Encrypts using AES GCM mode.
 *
 * @param key The symmetric key generated
 * @param iv The random iv generated
 * @param ct The integer array where the ciphertext will be stored
 * @param pt The integer array of the plaintext to be encrypted
 * @param tag The integer array where the tag will be stored
 * @param pt_bytes The length of the plaintext
 */
void encryptAndTag(uint8_t *key, uint8_t *iv, uint8_t *ct, uint8_t *pt, uint8_t *tag, int pt_bytes) {
  GCM<AES128> gcm;
  gcm.setKey(key, KEY_BYTES);
  gcm.setIV(iv, KEY_BYTES);
  gcm.encrypt(ct, pt, pt_bytes);
  gcm.computeTag(tag, KEY_BYTES);
}
```

IoTDongle.ino - Encrypting the data

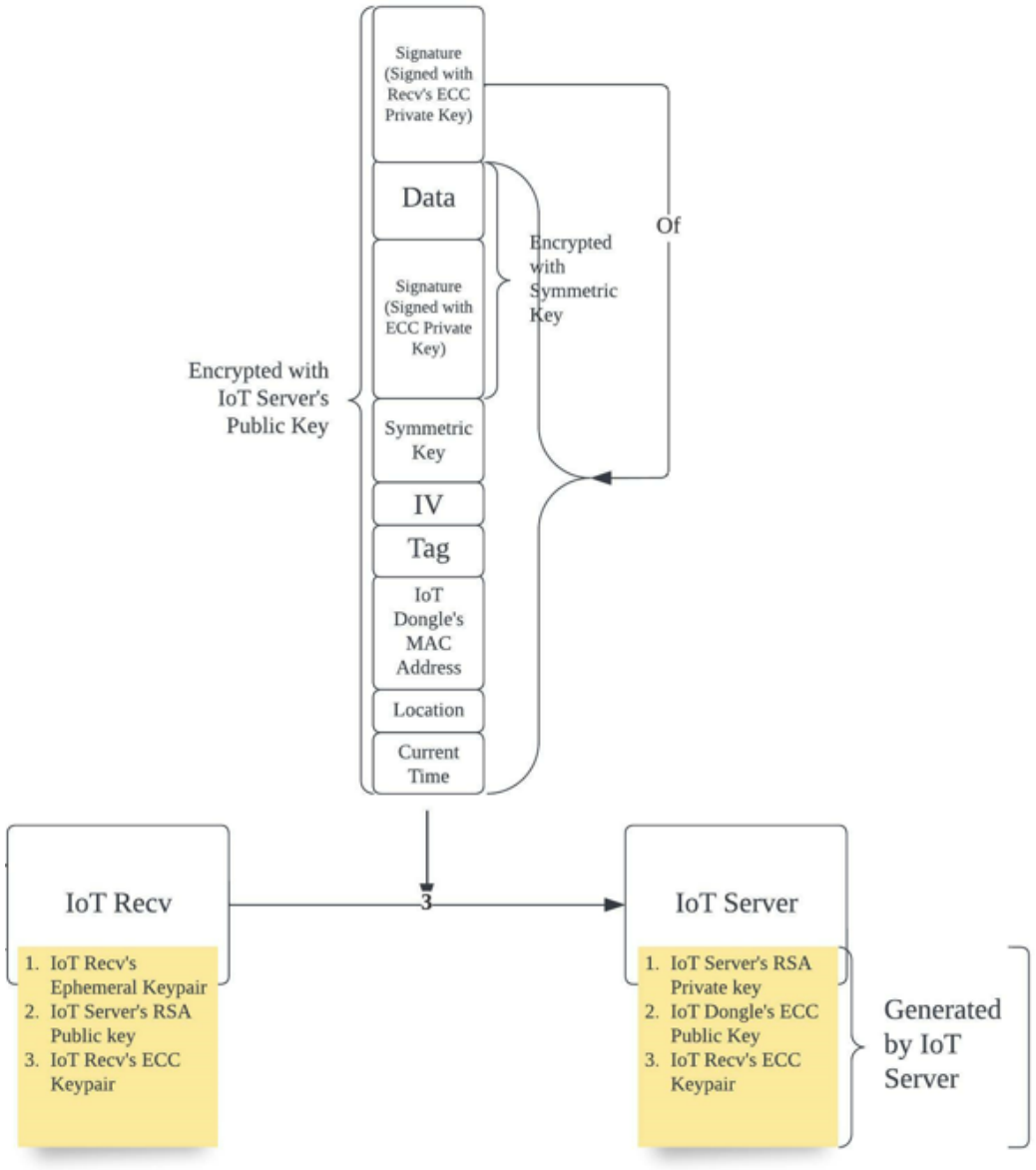
Using the Dongle-generated ECC private key, it will sign the RSSI data, forming the plaintext of (Signature + RSSI). Next, the Dongle will use the symmetric key to encrypt the plaintext using AES GCM mode, producing the encrypted ciphertext and tag as output.

```
/**
 * Constructs the payload to be sent to the Receiver.
 *
 * @param iv The random iv generated
 * @param ct The ciphertext generated from the encryption
 * @param tag The tag generated from the encryption
 * @param pub_key The ephemeral public key generated by the IoT Dongle
 * @param msg The char array which the message/payload will be stored in
 * @param msg_length The total length of the message
 * @param pt_bytes The length of the plaintext
 */
void constructMsg(uint8_t *iv, uint8_t *ct, uint8_t *tag, uint8_t *pub_key, char *msg, int msg_length, int pt_bytes) {
  memcpy(msg, (char *)iv, KEY_BYTES);
  memcpy(msg + KEY_BYTES, (char *)tag, KEY_BYTES);
  memcpy(msg + 2 * KEY_BYTES, (char *)pub_key, PUB_KEY_BYTES);
  memcpy(msg + 2 * KEY_BYTES + PUB_KEY_BYTES, (char *)ct, pt_bytes);
  msg[msg_length - 1] = '\0';
}

/**
 * Sends the message/payload to the Receiver
 *
 * @param msg The constructed message to be sent
 * @param msg_length The total length of the message
 */
void sendMsg(char *msg, int msg_length) {
  int curr = 0;
  int bytes = 20;
  while (RFduinoBLE.send(msg + curr, bytes)) {
    curr += bytes;
    if (msg_length - curr < 20) {
      bytes = msg_length - curr;
    }
  }
}
```

IoTDongle.ino - Constructing and sending payload

Finally, the Dongle will construct Payload A, consisting of (in order) iv, tag, Dongle ephemeral public key and ciphertext. Payload A will hence be sent to the Receiver via BLE as shown in Process 2.

7.3.2. Data processing in IoT Receiver



IoT Receiver - Server Workflow Diagram

```python
"""
Split the message to the iv, tag, dongle's ephemeral public key and ciphertext
"""
def splitMsg(msg):
    iv = msg[:KEY_BYTES]
    tag = msg[KEY_BYTES: 2 * KEY_BYTES]
    ctPublicKey = VerifyingKey.from_string(msg[2 * KEY_BYTES:2 * KEY_BYTES + EPHEMERAL_KEY_BYTES], curves.SECP160r1)
    ct = msg[2 * KEY_BYTES + EPHEMERAL_KEY_BYTES:]
    return (iv, tag, ctPublicKey, ct)

"""
Generate the symmetric key from ECDH key exchange
"""
def generateSymKey(ctPublicKey, privKey):
    ecdh = ECDH(curves.SECP160r1, privKey, ctPublicKey)
    sharedSecret = ecdh.generate_sharedsecret_bytes()
    h = hashlib.sha256(sharedSecret)
    symKey = h.digest()[:KEY_BYTES]
    return symKey
```

IoTRecv.py - Splitting payload and generating symmetric key

Upon receiving Payload A, the Receiver will split the message and obtain the Dongle's ephemeral public key and use its ephemeral private key to generate the shared secret, thus completing the ECDH exchange. The shared secret is again hashed using SHA256 to generate the same 128-bit symmetric key.

```python
"""
Create the new payload by adding tokens for splitting and adding the MAC and location to the dongle's payload, before encrypting with the server's public key
"""
def constructPayload(ct, symKey, iv, tag, mac, serverPubKey, signer):
    gcm = AES.new(symKey, AES.MODE_GCM, nonce=iv)
    gcm.decrypt_and_verify(ct, tag)
    current_time = datetime.datetime.now().isoformat(timespec='seconds').replace('T', ' ')
    message = ct + TOKEN + symKey + TOKEN + iv + TOKEN + tag + TOKEN + mac.encode() + TOKEN + LOCATION + TOKEN + current_time.encode()
    signature = signer.sign(message)
    message = signature + message
    cipher = PKCS1_OAEP.new(serverPubKey)
    payload = cipher.encrypt(message)
    return payload

"""
Send the encrypted payload to the server
"""
def sendPayload(payload):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((LHOST, LPORT))
        s.sendall(payload)
```

Using the symmetric key, the Receiver will verify the tag. The Receiver will construct a new message, consisting of (in order) Dongle's ciphertext, symmetric key, iv, tag, Dongle's MAC address, Receiver's location and current time, separated by a token "<TRACK2GATHER>". The Receiver will then sign the new message using its default private key and construct Payload B of (Signature + Constructed Message). Payload B is then sent to the IoT Server via TCP as shown in Process 3.

7.4. Decryption and Storage of Data

The file used for this section is IoTServer.py.

```
'''
Decrypts the second layer of encryption
using the symmetric key obtained from first decryption.
Returns the mac address of the dongle and the decrypted payload (data + signature).
'''
def decryptWithSymmetricKey(layerTwoPayload):
    split = layerTwoPayload.split(b"<TRACK2GATHER>")
    encryptedData = split[0]
    symmetricKey = split[1]
    nonce = split[2]
    authTag = split[3]
    macAddress = split[4]
    location = split[5]
    currTime = split[6]
    return macAddress, decrypt_AES_GCM(encryptedData, nonce, authTag, symmetricKey), location, currTime

'''
Decrypts the first outer layer of encryption
using Server's private key.
Returns the second layer payload.
'''
def decryptWithServerPrivateKey(encryptedPayload):
    privKeyFile = "serverPrivKey.pem"
    f = open(f'{secretDirectory}{privKeyFile}','rb')
    key = RSA.import_key(f.read())
    f.close()
    cipher = PKCS1_OAEP.new(key)
    layerTwoPayload = cipher.decrypt(encryptedPayload)
    if verifySignature(layerTwoPayload, serverGenDirectory, "recvPublicKey.pem"):
        return getData(layerTwoPayload)
    raise Exception
```

IoTServer.py - 2-Layer Decryption to get the Dongle Data

Once the encrypted payload arrives at the IoT Server, it performs the respective decryptions with the respective keys - the server private key and the embedded symmetric key.

```
'''
Verifies the signature of the data.
Returns whether the signature is valid. (Ensure no changes in data and verified signature)
'''
def verifySignature(payload, directory, filename):
    f = open(f'{directory}{filename}','rb')
    donglePublicKey = ECC.import_key(f.read())
    f.close()
    signature = payload[:SIG_BYTES]
    data = payload[SIG_BYTES:]
    verifier = eddsa.new(donglePublicKey, mode='rfc8032')
    try:
        verifier.verify(data, signature)
        return True
    except ValueError:
        return False
```

IoTServer.py - Signature Verification

Then after decrypting, the Server verifies the signatures using the keys stored in the SFTP directory.

```
'''
Sends the record ID, token ID (MAC address) and location. (And also time and RSSI soon)
'''
def sendToDB2(macAddress, location, current_time, payload):
    # Establishes connection to DB 2.
    dbConn = psycopg2.connect(
        database="db2",
        user="iot",
        host="ifs4205-gp02-3-i",
        port="5432",
        password=config.psql_password
        )
    dbConn.autocommit = True
    cursor = dbConn.cursor()
    query = "CALL add_tracingRecord(%s::macaddr, %s, %s, %s);"
    RSSI = getData(payload)
    parameters = (macAddress.decode(), current_time.decode(), location.decode(), RSSI.decode())
    cursor.execute(query, parameters)
    dbConn.close()
```

IoTServer.py - Sending to DB2

The data, if verified to be unmodified, is then sent and stored in the database, specifically DB2.

7.5. Unit Testing

For the IoT Receiver and IoT Server, we are using the Python `unittest` package to automate testing of the methods used in the respective main codes. These unit tests include testing both valid and invalid inputs, along with its respective output and exceptions.

As for `iptables`, we have done unit testing during CI phase, where the expected output of the `iptables` is compared with a mock copy of the actual `iptables` setup.

7.6. Security Considerations

First off, we wanted to ensure that the data from the dongles are not modified and not seen in plaintext. As such, we implemented the use of ephemeral keys and symmetric key generation to encrypt and sign the data sent to the IoT Receiver. Besides that, we wanted to provide authentication of the IoT Dongles, which led to the use of ECC keypairs for signing and verifying messages.

Secondly, to encrypt the newly appended data, and to provide another layer of protection, we made use of the IoT server's public key to ensure that the newly appended data is encrypted and not seen in plain data before sending from receiver to server.

Lastly, the use of SFTP to store the key files allows for authenticity and confidentiality as key files exchanged (uploaded or retrieved) will not be visible to others and only authenticated users can gain access to this channel.

7.7. Other Security Implementations

As the IoT Receivers are simulated as a virtual machine, it requires credentials that only authorized users would know. This prevents unauthorized users from accessing the virtual machine. In addition, authorized users are given the least privileges needed for their functions, which means they do not have any root privileges. Lastly, they can only connect to whitelisted IPs as configured using `iptables`. For configuration purposes, there is also an admin account, which cannot be accessed by any other users. This admin account should not be needed once the virtual machine has been configured, but it exists with root privileges.

For the IoT Server, since it functions to only receive data from the IoT Receivers, the Server is configured to only receive data and be accessed by whitelisted IPs. This is done through `iptables` configurations.

Other than `iptables`, the IoT Server is also configured to restrict SSH shell access for the `sftp` user, while still allowing SFTP access. This restricts and reduces the attack surface in the event that `sftp` user is compromised.

Last of all, for the IoT Server, it is running as a low privileged user -`iotsvc` with no sudo privileges. This is to ensure that in the event that the IoT service is indeed exploited through e.g. a zero day vulnerability of a python package being used, it is not possible to be exploited to gain higher privileges.

7.8. Limitations of the IoT Dongle and Receiver

Given the devices that we have chosen to use for the implementation of the IoT Subsystem, there are some limitations that might require mitigations.

1. Limited Capabilities of the IoT Dongle
    a. The dongle is unable to store files or perform complex computations in a reasonable time. This means that the dongle can be held hostage by a Bluetooth connection and it is assumed that this is not possible.
2. Strength of BLE
    a. As the strength of the BLE connection is dependent on the environment, there are instances where the Receiver is unable to establish a connection to the Dongle. This means that the Receiver might have to retry the connection again. In the Sync process, the script might have to be run several times for all Dongles to be synced correctly.
    b. Due to the time constraint we have in place for a BLE connection, there are instances where the Dongle does not write to the Receiver in time. This is an accepted issue as the Receiver can simply connect to the Dongle again, adding to the fact that transient contact with the Receiver should not be registered.

8. Security Claims

8.1. Security Claims for Web Application

1. Given a username and password, the user is unable to retrieve any information about other usernames and password
2. Given a valid JWT, the attacker is unable to discover the secret key used by the server to sign the JWT or forge another valid JWT
3. Given no username and password, an attacker is unable to discover a valid pair of username-password through brute force
4. Users of our web application is not susceptible to CSRF
5. A normal user without Contact Tracer or Health Authority status is unable to evoke privilege escalation and perform privileged actions
6. Web Application is not susceptible to any forms of injection attack

8.2. Security Claims for Database

1. Any unauthorized users (including IoT server) should not be able to create new dongles in Database.

8.3. Security Claims for IoT Components

1. Given the credentials of HA, you should not be able to perform any privilege escalation in the Receiver.
2. The communication between an IoT Receiver and an IoT Dongle should be secured and encrypted in the sense that when the IoT Dongle connects to an unknown source, it will not transmit any data, and thus cannot be decrypted.
    a. Note that any Bluetooth Device can establish a connection to the IoT Dongle. This prevents the Dongle from connecting to a legitimate Receiver, which is not a security claim here.
3. The identity of the IoT Dongle cannot be spoofed, i.e. if Dongle A is registered, another device is unable to act as Dongle A.
4. Given credentials of an SFTP user, you should not be able to run any commands other than SFTP commands on IoT Server.
    a. Note that you should not be making any changes/modifications on the files used by IoT components in the SFTP directories.
    b. Credential for `sftp`
        i. `sftp:REDACTED`
5. The communication channel between IoT Server and IoT Receiver should be secure and encrypted. The data sent should not be viewable or decrypted by unauthorized individuals.
6. Given access as a low privileged service user, `iotsvc` user on the IoT Server, you should not be able to escalate privileges to root via non-password methods.
    a. Note that leveraging on this access to decrypt encrypted data is not considered a compromise of IoT security claim 5.
    b. Credentials for `iotsvc`
        i. `iotsvc:REDACTED`
7. Any unauthorized (not whitelisted) IPs should not be able to communicate with the important services of IoT Server.
    a. IMPORTANT - To help with facilitating PenTesting, I have added a utility service at port 4205 to whitelist your IP.