CSCI-531 Spring 2023 Semester Project Version 5

Design and Implementation of a Simplified Secure Decentralized Audit System (SSDAS)

Nicholas Guerrero and Alan Perdigao

May 1, 2023

[1.0] Introduction

The project/problem domain is Electronic Health Records (EHR) systems have gradually replaced traditional paper-based health record systems in the United States. Audit logs serve multiple functional and regulatory purposes in EHR systems. When patient records are assessed for some reason, the history of all such events must be recorded in a log file for later audit on access histories. The log files are used to reconstructing the past state of medical records, and it can be used as legal evidence in medical malpractice cases.

[1.1] Literature Review:

Over the past four years, EHR Audit Logs have gained much research attention and present claims these logs are a "new goldmine for health services research"—with applications in medical quality domains of: safe, effective, patient-centered, timely, efficient, and equitable outcomes [11]. These logs contain clinician/patient interactions with Protected Health Information and are required under the Health Insurance Portability and Accountability Act and Meaningful Use policies [12]. This past year, researchers claimed the use of EHR Audit Logs "will expand the breadth of research to improve cancer care (and outcomes)" in four domains: (1) diagnostic reasoning and consumption; (2) care team collaboration and communication; (3) patient outcomes and experience; and (4) provider burnout/fatigue [13].

With the security and privacy policy requirements for Protected Health Information, researchers have proposed various prototypes using cryptographic and block chain technologies to meet requirements—such as k-Health, e-Heath, PPAC, open-PHR, MeDShare, MedRec, MediBchain, and PREHEALTH [2-10].

[1.2] Problem Statement: Simplified Secure Decentralized Audit System (SSDAS) Design/Implementation Goals

SSDAS System requirements will focus on the security and privacy of Electronic Health Record Audit Logs prototype design/software artifact with the following goals:

> **[1] Privacy:** Patient privacy should be maintained. Unauthorized entities should not be able to access audit records.

[2] Identification and Authorization: All system users must be identified and authenticated. All requests to access the audit data should be authorized.

[3] Queries: Only authorized entities should be able to query audit records.

[4] Immutability: No one should be able to delete or change EXISTING audit records without detection. Any modifications/deletions of the audit records should be detected and reported. You can focus on attackers who are internal to the system modifying the audit data after it has already been received by the system. Note that you do not need to protect information against modification, it is sufficient to just detect and report any unauthorized modifications to the audit data.

[5] Decentralization: The system should not rely on a single trusted entity to support immutability. This means there is no single entity (organization) that controls the audit logs. Blockchain-based technology works well for such systems.

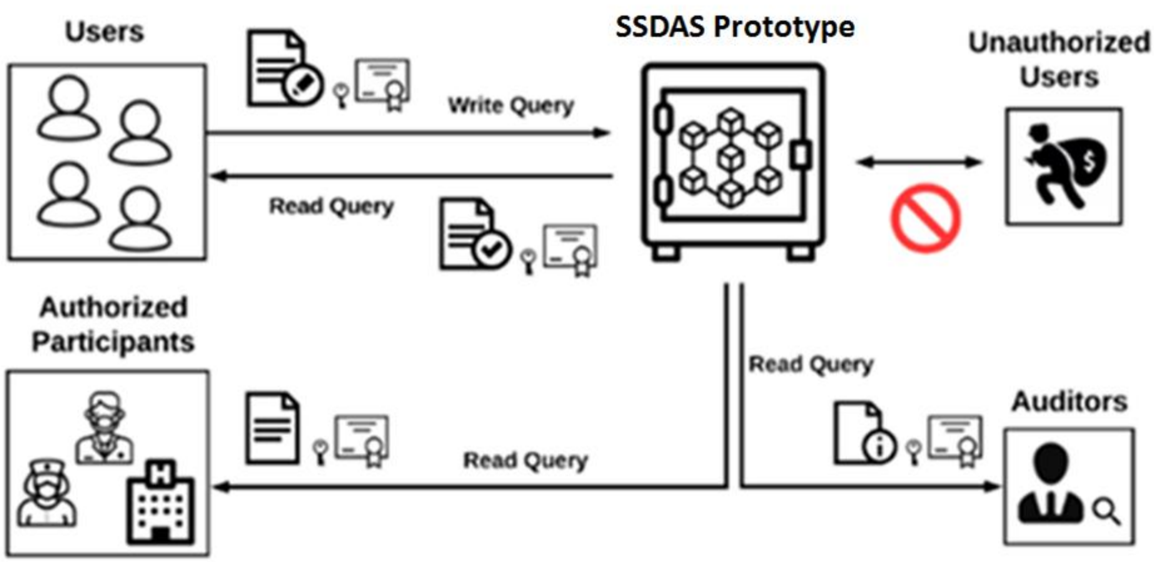
Note: SSDS Key Algorithm Functionality: Supports 10 Patients and three Audit Companies with Audit Records (Spec) that are Scalable/ Distributed—Run over the Network—Web Based Interactions

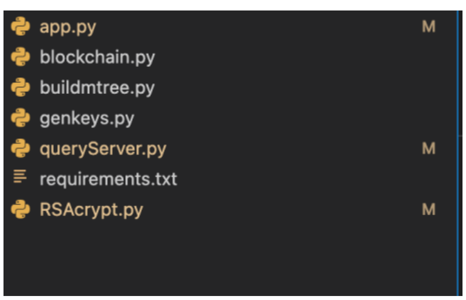
[1.3] Methods: The Lean Startup Method

To address the problem statement and SSDAS design prototype artifact, this research will use Ries's *The Lean Startup Method*—as shown below. Later in Section [5.0], we will evaluate and measure the SSDAS artifact using the Method's Minimal Viable Product (MVP)—"…a version of the product that enables a full turn of the build-measure-learn loop with the minimal amount of effort and the least amount of development time"—against seven scenarios or use cases to prove the MVP meets the five SSDAS goals [14].

[2.0] System Workflow

Describe a general workflow for your system: tasks to be accomplished and steps that are necessary to complete a specific task.





Appy.py - Runs Main Server
Blockchain.py – Contains code to create and run Blockchain
Buildmtree.py – Contains code to create and run MerkelTree
Genkeys.py – Contains code to create public/private keys for User Patient data using RSA
RSAcrypt.py – Contains code to encrypt User Patient data using AES-128 and RSA to protect AES keys
Requirements.txt – Text file containing all libraries used in the project. Can run “pip install requirements.txt” to install on machine

Phase 1: Startup Phase
Run queryServer.py

```python
if __name__ == '__main__':
    with app.app_context():
        blockchain = Blockchain()
    app.run(port=5001)
```

On startup the Query Server will create an empty blockchain and run itself on port 5001

Run app.py

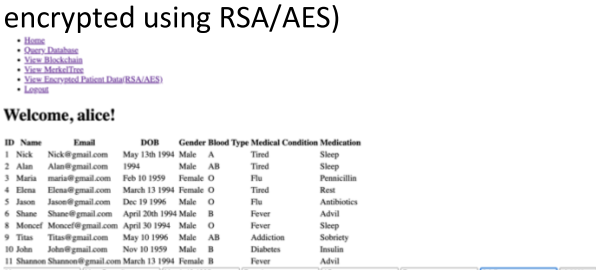
```python
if __name__ == '__main__':
    with app.app_context():
        createMerkelTree()
        createBlockChain()
    app.run(port=5000)
```

On startup the main app will create a Blockchain and Merkel Tree data structures out of the Audit Log records and then post the blockchain to the Query Server and run itself on port 5000 The Blockchain creates a decentralized Audit Log among all clients who access the main server to get HTML and the Query Server itself. The Merkel Tree is used to detect Audit Log tampering via its root hash and making an atomic comparison with mutex locks.

Blockchain: Can be viewed in Section **[5.7]**
Merkel Tree: Can be viewed in Section **[7.2.4]**

Phase 2: Login
A Super User (audit company) or Regular User (patient) can login with their credentials. Super Users have usernames: “alice”, “bob”, “carl” with passwords “password1”, “password2”, “password3” respectively. A Regular User can be given a username and password by adding them as a patient with a GUI on the home page once the Super User logs in, their username is their given name and their password is automatically assigned to be “user{ user_id #}”

Super User alice adds a new patient Mary to the database of patient records (Her data is encrypted using RSA/AES)



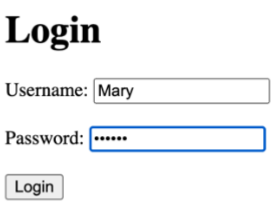
- Home
- Query Database
- View Blockchain
- View MerkelTree
- View Encrypted Patient Data(RSA/AES)
- Logout

Welcome, alice!

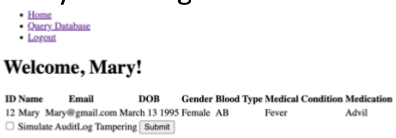
ID	Name	Email	DOB	Gender	Blood Type	Medical Condition	Medication
1	Nick	Nick@gmail.com	May 13th 1994	Male	A	Tired	Sleep
2	Alan	Alan@gmail.com	1994	Male	AB	Tired	Sleep
3	Maria	maria@gmail.com	Feb 10 1959	Female	O	Flu	Pennicillin
4	Elena	Elena@gmail.com	March 13 1994	Female	O	Tired	Rest
5	Jason	Jason@gmail.com	Dec 19 1996	Male	O	Flu	Antibiotics
6	Shane	Shane@gmail.com	April 20th 1994	Male	B	Fever	Advil
8	Moncef	Moncef@gmail.com	April 30 1994	Male	O	Fever	Sleep
9	Titas	Titas@gmail.com	May 10 1996	Male	AB	Addiction	Sobriety
10	John	John@gmail.com	Nov 10 1959	Male	B	Diabetes	Insulin
11	Shannon	Shannon@gmail.com	March 13 1994	Female	B	Fever	Advil

Mary | Mary@gmail.com | March 13 1995 | Female | AB | Fever | Advil | Add User

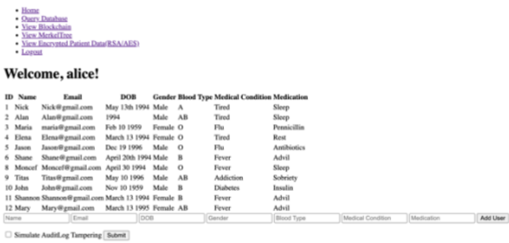
Mary can then login with her generated credentials



Mary has a login screen customized for her authorization level



You can see Mary is now a patient alice can view



Super Users also have the ability to edit user data and delete user data via the "edit_user/user_id" and "delete_user/user_id" routes in addition to the ability to create new Regular Users. An edit and deletion will be recorded by the Audit Log.

Password hashing and checking is done with the werkzeug.security Python Library

```
audit_users = {
                'alice': 'pbkdf2:sha256:260000$yMFmeJlSzF661LHr$3fd22051a19db631f4c46bea63d6c43f77c893201
                'bob' : 'pbkdf2:sha256:260000$AQs5TJTsEBVXUk4o$c0067e62b4a9b0bdeec49aadd56e24136583adcbf3
                'carl' : 'pbkdf2:sha256:260000$jevexjzerWaZmi6Z$16e87b52b7b6bd67629d569793f7cad334e318ef8
                }
```

werkzeug.security is a Python library that provides various security-related functions for web applications. Two of the most commonly used functions from this library are heck_password_hash and generate_password_hash.

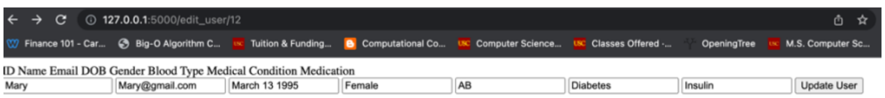
generate_password_hash(password, method='pbkdf2:sha256', salt_length=8) is a function that generates a hash of the input password using a secure one-way hashing algorithm. The password parameter is the input password that you want to hash, while the method parameter specifies the algorithm to use (default is PBKDF2 with SHA-256). The salt_length parameter specifies the length of the salt to use in the hashing process (default is 8). The function returns the hashed password as a string.

check_password_hash(hash, password) is a function that checks if a given password matches a given hash. The hash parameter is the hashed password string, while the password parameter is the password you want to check against the hash. The function returns a boolean value indicating whether or not the password matches the hash.

Phase 3: Querying

A Regular User has their querying abilities limited to just their own data. A patient Mary can only view edits and queries of her own data while a Super User such as alice can view all patient data and all Audit Log data. This is managed by a Query Server that is running on port 5001 that manages what data to send back to the main app server based on login credentials. Note all patient data in the sqlite database is encrypted so anyone listening over the network would not be able to decrypted the ciphertext without the appropriate key.

Alice makes an update to Mary's data



Alice can see all Audit Log Records

Query Results

ID	Datetime	Patient ID	User ID	Action Type
1	2023-04-30 09:54:05.439673	alice	alice	query - SELECT * FROM audit_log;
2	2023-04-30 10:03:48.487917	5	alice	change
3	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
4	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
5	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM user WHERE id=1
6	2023-05-01 09:47:01.833094	1	alice	change
7	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
8	2023-05-01 09:47:01.833094	bob	bob	query - SELECT * FROM audit_log;
9	2023-05-01 09:47:01.833094	10	bob	query - SELECT * FROM user;
10	2023-05-01 09:47:01.833094	11	bob	create
11	2023-05-01 09:47:01.833094	12	bob	create
12	2023-05-01 09:47:01.833094	11	bob	delete
13	2023-05-01 09:47:01.833094	12	bob	delete
14	2023-05-01 09:47:01.833094	bob	bob	query - SELECT * FROM audit_log;
15	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
16	2023-05-01 09:47:01.833094	11	alice	create
17	2023-05-01 09:47:01.833094	11	alice	change
18	2023-05-01 10:01:03.952779	11	alice	change
19	2023-05-01 10:02:00.778160	11	alice	delete
20	2023-05-01 10:02:30.115801	10	alice	delete
21	2023-05-01 10:11:09.611014	7	alice	delete
22	2023-05-01 10:13:46.275916	10	alice	create
23	2023-05-01 10:13:46.275916	11	alice	create
24	2023-05-01 10:13:46.275916	12	alice	create
25	2023-05-01 10:13:46.275916	12	alice	change

On login, Mary can see only Audit Log records pertaining to her health data

Query Results

ID	Datetime	Patient ID	User ID	Action Type
11	2023-05-01 09:47:01.833094	12	bob	create
13	2023-05-01 09:47:01.833094	12	bob	delete
24	2023-05-01 10:13:46.275916	12	alice	create
25	2023-05-01 10:13:46.275916	12	alice	change

While a Regular User "Nick" only sees Audit Log records pertaining to his health data

Query Results

ID	Datetime	Patient ID	User ID	Action Type
3	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
4	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
5	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM user WHERE id=1
6	2023-05-01 09:47:01.833094	1	alice	change
7	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
15	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1

All patient data is encrypted by the main server

12|Mary|7b226362635f63697068657274657874223a2022653237303832623165396162396263633132
65795f7072696d65223a2031343739313738393739343933343939393333313338323434313239373832
343238353135363637303132333730353034353730383935393832353038373234343533323137313630
323735343333935323137373432303931313338363035363739343537363032373934353534363631353 4
353736303534333433373632323834333734383930393234303835313436323937333432333232373332
363033333239383931333932313436353330393832303335353131353933333313937343538323036373 1
3131333832393433343535343130342c20226976223a2022353032666638343063353935306661313033 66
2635f63697068657274657874223a2022373434353663613062383765333566386136313232303561373
5223a2032313032333234383436303237323031313234383238313832383334333630373336373039353
73634303631323931313339323933393538343833343136343832303238303331373131363437343431 3

Phase 4: Simulate Tampering

Checking this GUI box will edit one of the Audit Log records, and attempt at access to another route such as querying the database or editing user name the app will warn the system that tampering has been detected by the following algorithm:

☑ Simulate AuditLog Tampering [Submit]

On generation & insertion of Audit Log record:

1.) Acquire mutex lock
2.) Given current Audit Log records recompute a Merkel Tree
3.) If the previous Merkel Tree's root is equal to the newly computed Merkel Tree's root then no tampering has occurred, else tampering has definitely occurred (First Merkel Tree is computed in Phase 1: Startup)
4.) If no tampering, insert Audit Log record
5.) Set the current Merkel Tree to be the Merkel Tree computed after the addition of the new Audit Log Record
6.) Release mutex lock

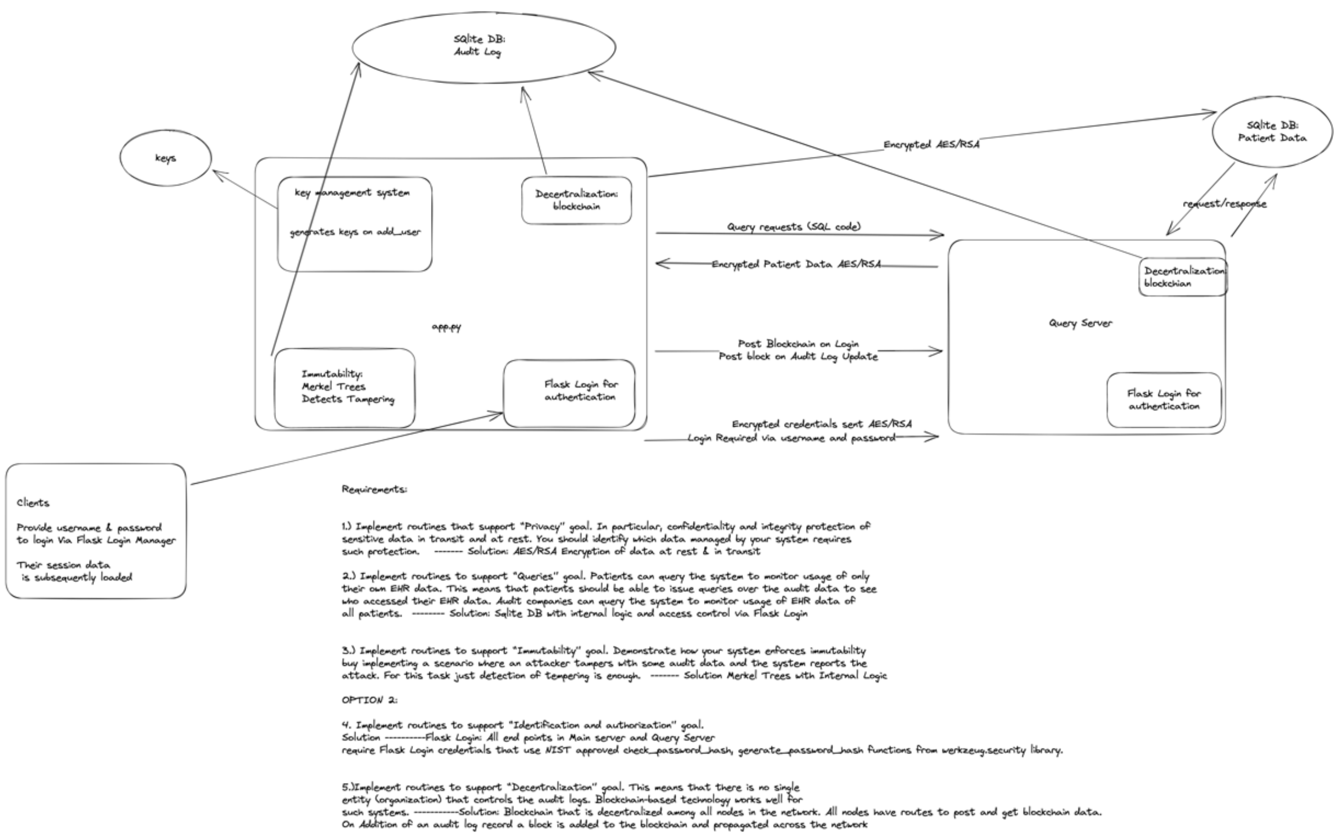
```python
def manageAuditLog(log):
    # create a lock object
    lock = threading.Lock()
    # acquire the lock before executing the code
    lock.acquire()
    try:
        isImmutable = checkLogImmutability()

        if isImmutable == False:
            audit_record = AuditLog.query.get(1)
            audit_record.action_type = "query - SELECT * FROM audit_log;"
            db.session.commit()
            return False
        #  create audit log entry
        db.session.add(log)
        db.session.commit()

        #recreate Merkel Tree
        createMerkelTree()

        # add audit log entry as a block to the blockchain
        blockchain.add_block(log.to_dict())

         # Post blockchain block
        password = session.get('password')
        block = json.dumps(log.to_dict(), sort_keys=True, cls=DateTimeEncoder)
        response = requests.post('http://127.0.0.1:5001/add_block', data={'block' : block}, params={'username': current_user.id, 'password': password, 'block' : block})

    finally:
        # release the lock
        lock.release()
        if isImmutable == False:
            return False
        return True
```

```python
def checkLogImmutability():
    audit_logs = AuditLog.query.all()
    if len(audit_logs) != 0:
        mTreeInput = [f"{audit_record.date_time}|{audit_record.patient_id}|{audit_record.user_id}|{audit_record.action_type}" for audit_re
        mTree = MerkleTree(mTreeInput)

        if mTree.root.hashHex != AuditLog.mTree.root.hashHex:
            return False
    return True
```

This algorithm guarantees that any tampering will be detected by the system since no operations having to do with creating/checking the Merkel Tree can occur outside the mutex locks.

[3.0] SSDAS System Architecture

As shown below there are multiple functions: **[3.1] Describe the system components and their functionality:** [3.1.1] Authentication Server; [3.1.2] Audit Server; [3.1.3] Query Server [3.1.4] Others, etc.. **[3.2] Describe the communication patterns among the components (requests and response).**



The Main Server on login will allow access to routes that will allow a user to query the audit log and their user (patient) data. Blockchain and Merkel Tree data structured are generated on bootup of the Main Server and Query Server respectively. The blockchain is posted to clients and to the Query Server to maintain decentralization of the Audit Log. A key management system is held within the Main Server that encrypts patient data that is being sent to any other route/port and it only decrypted when HTML needs to be rendered to the appropriate user or edits to user data need to be submitted by a Super User.

Patient data needs to be decrypted to render HTML properly

```
    <td>{{ user.id }}</td>
    <td>{{ user.name }}</td>
    <td>{{ decrypt(user.email, "keys/" + user.name + ".prv") }}</td>
    <td>{{ decrypt(user.dob, "keys/" + user.name + ".prv") }}</td>
    <td>{{ decrypt(user.gender, "keys/" + user.name + ".prv") }}</td>
    <td>{{ decrypt(user.blood_type, "keys/" + user.name + ".prv") }}</td>
    <td>{{ decrypt(user.medical_condition, "keys/" + user.name + ".prv") }}</td>
    <td>{{ decrypt(user.medication, "keys/" + user.name + ".prv") }}</td>

</tr>
```

Patient data needs to be decrypted to submit to the Query Server so the sqlite database can update properly.

```
<form method="POST" action="{{ url_for('edit_user', user_id=user.id) }}">
    <input type="text" name="name" value="{{ user.name }}">
    <input type="email" name="email" value="{{ decrypt(user.email, 'keys/' + user.name + '.prv') }}">
    <input type="text" name="dob" value="{{ decrypt(user.dob, 'keys/' + user.name + '.prv') }}">
    <input type="text" name="gender" value="{{ decrypt(user.gender, 'keys/' + user.name + '.prv') }}">
    <input type="text" name="blood_type" value="{{ decrypt(user.blood_type, 'keys/' + user.name + '.prv') }}">
    <input type="text" name="medical_condition"
        value="{{ decrypt(user.medical_condition, 'keys/' + user.name + '.prv') }}">
    <input type="text" name="medication" value="{{ decrypt(user.medication, 'keys/' + user.name + '.prv') }}">

    <button type="submit">Update User</button>
</form>
```

Patient data is subsequently re-encrypted using the User's public key once the form is submitted on the "edit_user/user_id" route

```
def edit_user(user_id):
    user = User.query.get(user_id)
    if current_user.username not in loginUser.audit_users:
        return render_template('unauthorized.html'), 401
    if request.method == 'POST':
        user.name = request.form['name']
        user.email = encrypt(request.form['email'], "keys/" + user.name + ".pub")
        user.dob = encrypt(request.form['dob'], "keys/" + user.name + ".pub")
        user.gender = encrypt(request.form['gender'], "keys/" + user.name + ".pub")
        user.blood_type = encrypt(request.form['blood_type'], "keys/" + user.name + ".pub")
        user.medical_condition = encrypt(request.form['medical_condition'], "keys/" + user.name + ".pub")
        user.medication = encrypt(request.form['medication'], "keys/" + user.name + ".pub")
```

[4.0] Cryptographic Components

[4.1] Discuss appropriate choice of cryptographic primitives to ensure system supports goals.

Goal/Crypto Primitive	Authentication Server (app.py)	Audit Server (app.py/ sqlite DB)	Query Server (QueryServer.py)	Patient Data (sqlite DB)
Privacy	Confidentiality (Block Cipher) Integrity (Public-Key Cipher)	Confidentiality (Block Cipher) Integrity (Public-Key Cipher)	Confidentiality (Block Cipher) Integrity (Public-Key Cipher)	Confidentiality (Block Cipher) Integrity (Public-Key Cipher)

Identification & Authorization	Authenticity (Hash Functions)	Authenticity (Hash Functions)	Authenticity (Hash Functions)	Authenticity (Hash Functions)
Queries	Authenticity (Hash Functions)	Authenticity (Hash Functions)	Authenticity (Hash Functions)	Authenticity (Hash Functions)
Immutability	Nonrepudiation/ Integrity (Hash Functions/Sha-256)	Nonrepudiation/ Integrity (Hash Functions/ Sha-256)	Nonrepudiation/ Integrity (Hash Functions/ Sha-256)	Nonrepudiation/ Integrity (Hash Functions/ Sha-256)
Decentralization	Nonrepudiation/ Integrity (Hash Functions/ Sha-256)	Nonrepudiation/ Integrity (Hash Functions/ Sha-256	Nonrepudiation/ Integrity (Hash Functions/ Sha-256)	Nonrepudiation/ Integrity (Hash Functions/ Sha-256)

[4.2] Describe concrete encryption schemes and key management approaches used in system.

Encryption Schemes		
Merkel Tree - Tampering Detection		
RSA - Encryption of AES keys		
AES-128 - Encryption of Patient Data		
Blockchain - Decentralization		
Key Management		
Diffie Hellman		

[5.0] Evaluation: SSDAS System Meets Design and Implementation Goals/Requirements

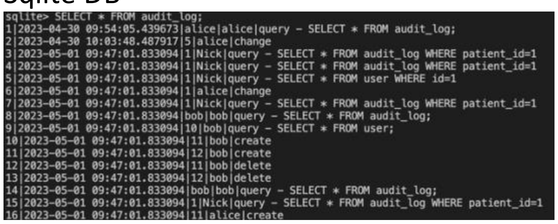
To evaluate and measure the SSDAS artifact using the Method's Minimal Viable Product (MVP) concept—"…a version of the product that enables a full turn of the build-measure-learn loop with the minimal amount of effort and the least amount of development time"—the artifact will be evaluated/measured against seven scenarios (or use cases) listed below to demonstrate the MVP meets the five overall SSDAS goals: (1) Privacy; (2) Queries; (3) Immutability; (4) Identification and Authorization; and (5)Decentralization [14].

For ease of reading, the seven scenarios (or use cases) are summarized below in outline format. Then, the SSDS artifact will demonstrate and measure it against each scenario as proof—using screenshots.

[5.1] Scenario 1: Audit Data Record is Generated. Transmitted, and Stored in the Audit Log

Audit Record is generated and stored in the Sqlite DB, Blockchain, & Merkel Tree

Sqlite DB



Blockchain

Merkel Tree

[5.2] Scenario 2: Patient Wants to Know Who Accessed Their EHR Data

Patients will be sent a response by the Query Server with all Audit Logs pertaining to their user id

Query Results for **User: Nick** with **Patient ID: 1**

- Home
- Query Database
- Logout

Query Results

ID	Datetime	Patient ID	User ID	Action Type
3	2023-04-28 14:21:07.001476	1	Nick	query - SELECT * FROM user WHERE id=1
43	2023-04-29 12:15:45.090511	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
44	2023-04-29 12:15:45.090511	1	alice	change

[5.3] Scenario 3: Auditor Wants to Know Who Accessed Data of All or Particular Patients

Auditors are Super Users and therefore receive a different response from the Query Server with all Audit Log records

INPUT QUERY: SELECT * FROM audit_log;
QUERY RESULT: All audit log records

Query Results

ID	Datetime	Patient ID	User ID	Action Type
1	2023-04-30 09:54:05.439673	alice	alice	query - SELECT * FROM audit_log;
2	2023-04-30 10:03:48.487917	5	alice	change
3	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
4	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
5	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM user WHERE id=1
6	2023-05-01 09:47:01.833094	1	alice	change
7	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
8	2023-05-01 09:47:01.833094	bob	bob	query - SELECT * FROM audit_log;
9	2023-05-01 09:47:01.833094	10	bob	query - SELECT * FROM user;
10	2023-05-01 09:47:01.833094	11	bob	create
11	2023-05-01 09:47:01.833094	12	bob	create
12	2023-05-01 09:47:01.833094	11	bob	delete
13	2023-05-01 09:47:01.833094	12	bob	delete
14	2023-05-01 09:47:01.833094	bob	bob	query - SELECT * FROM audit_log;
15	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
16	2023-05-01 09:47:01.833094	11	alice	create
17	2023-05-01 09:47:01.833094	11	alice	change
18	2023-05-01 10:01:03.952779	11	alice	change
19	2023-05-01 10:02:00.778160	11	alice	delete
20	2023-05-01 10:02:30.115801	10	alice	delete
21	2023-05-01 10:11:09.611014	7	alice	delete
22	2023-05-01 10:13:46.275916	10	alice	create
23	2023-05-01 10:13:46.275916	11	alice	create
24	2023-05-01 10:13:46.275916	12	alice	create
25	2023-05-01 10:13:46.275916	12	alice	change
26	2023-05-01 10:13:46.275916	alice	alice	query - SELECT * FROM audit_log;
27	2023-05-01 10:13:46.275916	12	Mary	query - SELECT * FROM audit_log WHERE patient_id=12

INPUT QUERY: SELECT * FROM audit_log WHERE patient_id=1;
QUERY RESULT: A subset of audit_log records just pertaining to User “Nick”

Query Results

ID	Datetime	Patient ID	User ID	Action Type
3	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
4	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
5	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM user WHERE id=1
6	2023-05-01 09:47:01.833094	1	alice	change
7	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
15	2023-05-01 09:47:01.833094	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
28	2023-05-01 10:13:46.275916	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
30	2023-05-01 10:13:46.275916	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
31	2023-05-01 10:13:46.275916	1	Nick	query - SELECT * FROM user WHERE id=1
38	2023-05-01 10:13:46.275916	1	alice	change

[5.4] Scenario 4: Attacker Corrupts an Existing Audit Record

A GUI is provided on the home screen to simulate Audit Log Tampering

Submitting the button will run the code in the following route:

```
@app.route('/tamper', methods=['POST'])
def tamper_audit_log():
    if request.form.get('tamper'):
        audit_record = AuditLog.query.get(1)
        audit_record.action_type = "Tamperered With"
        db.session.commit()
        return redirect(url_for('index'))
```

As seen the Audit Log "record 1" has been tampered with

```
sqlite> sqlite> SELECT * FROM audit_log;
1|2023-04-28 14:19:06.542923|alice|alice|Tamperered With
2|2023-04-28 14:19:06.542923|4|alice|create
3|2023-04-28 14:21:07.001476|1|Nick|query - SELECT * FROM user WHERE id=1
4|2023-04-28 14:21:07.001476|4|alice|delete
5|2023-04-28 14:22:21.847462|4|alice|create
6|2023-04-28 14:22:21.847462|4|alice|delete
7|2023-04-28 14:22:21.847462|4|alice|create
8|2023-04-28 14:22:21.847462|4|alice|delete
9|2023-04-28 14:22:21.847462|4|alice|create
10|2023-04-28 14:22:21.847462|4|alice|delete
11|2023-04-28 14:22:21.847462|4|alice|create
12|2023-04-28 14:22:21.847462|4|alice|delete
13|2023-04-28 14:22:21.847462|4|alice|create
14|2023-04-28 14:22:21.847462|4|alice|delete
15|2023-04-28 14:22:21.847462|4|alice|create
16|2023-04-28 14:22:21.847462|4|alice|delete
17|2023-04-28 14:22:21.847462|4|alice|create
```

The following "manageAuditLog" function will detect this tampering atomically using mutex locks as described in Section **[2.0] System Workflow Phase 4**

```python
def manageAuditLog(log):
    # create a lock object
    lock = threading.Lock()
    # acquire the lock before executing the code
    lock.acquire()
    try:
        isImmutable = checkLogImmutability()

        if isImmutable == False:
            audit_record = AuditLog.query.get(1)
            audit_record.action_type = "query - SELECT * FROM audit_log;"
            db.session.commit()
            return False
        #  create audit log entry
        db.session.add(log)
        db.session.commit()

        #recreate Merkel Tree
        createMerkelTree()

        # add audit log entry as a block to the blockchain
        blockchain.add_block(log.to_dict())

         # Post blockchain block
        password = session.get('password')
        block = json.dumps(log.to_dict(), sort_keys=True, cls=DateTimeEncoder)
        response = requests.post('http://127.0.0.1:5001/add_block', data={'block' : block}, params={'username': current_user.id, 'password': password, 'block' : block})

    finally:
        # release the lock
        lock.release()
        if isImmutable == False:
            return False
        return True
```

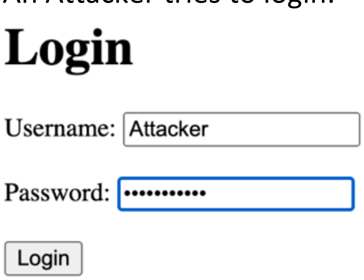
```python
log = AuditLog(patient_id=str(patient_ids), user_id=current_user.id, action_type='query - ' + sql_code)
isSecure = manageAuditLog(log)
if not isSecure:
    return "Warning: Audit Log Tampering Detected!!!"
```

Tampering Detected Screen

Warning: Audit Log Tampering Detected!!!

[5.5] Scenario 5: Patient, Auditor, or Attacker Tries to Log-In to Your System

An Attacker tries to login:



Unless proper credentials are provided the Main Server will not provide rendered HTML for a client and a 401 screen will appear.

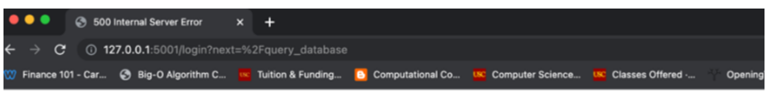
```
Traceback (most recent call last):
  File "/Users/nguerrer/Documents/Graduate_School/CS_531/Final_Prog/FinalProg/lib/python3.10/site-packages/flask/app.py
", line 2528, in wsgi_app
    response = self.full_dispatch_request()
  File "/Users/nguerrer/Documents/Graduate_School/CS_531/Final_Prog/FinalProg/lib/python3.10/site-packages/flask/app.py
", line 1826, in full_dispatch_request
    return self.finalize_request(rv)
  File "/Users/nguerrer/Documents/Graduate_School/CS_531/Final_Prog/FinalProg/lib/python3.10/site-packages/flask/app.py
", line 1845, in finalize_request
    response = self.make_response(rv)
  File "/Users/nguerrer/Documents/Graduate_School/CS_531/Final_Prog/FinalProg/lib/python3.10/site-packages/flask/app.py
", line 2137, in make_response
    raise TypeError(
TypeError: The view function for 'login' did not return a valid response. The function either returned None or ended wi
thout a return statement.
127.0.0.1 - - [29/Apr/2023 12:40:04] "GET /login?next=%2Fget_usernames%3Fusername%3DAttacker%26password%3Dafasfasg HTTP
/1.1" 500 -
```

[5.6] Scenario 6: Patient, Auditor, or Attacker Tries to Access Audit Data

@login_required decorator on Query Server prevents Querying unless authenticated

```python
@app.route('/query_database')
@login_required
def query_database():
    if request.method == 'GET':
        # sql_code = request.form['sql-code']
        sql_code = request.args.get('sql_code')

        # connect to the database
        conn = sqlite3.connect('./instance/sqlite.db')

        # create a cursor object
        cursor = conn.cursor()
```

An attacker fails to access endpoint /query_database unless authenticated



Internal Server Error

The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.

Login credentials required to access endpoint

```
Traceback (most recent call last):
  File "/Users/nguerrer/Documents/Graduate_School/CS_531/Final_Prog/FinalProg/lib/python3.10/site-packages/flask/app.py
", line 2528, in wsgi_app
    response = self.full_dispatch_request()
  File "/Users/nguerrer/Documents/Graduate_School/CS_531/Final_Prog/FinalProg/lib/python3.10/site-packages/flask/app.py
", line 1825, in full_dispatch_request
    rv = self.handle_user_exception(e)
  File "/Users/nguerrer/Documents/Graduate_School/CS_531/Final_Prog/FinalProg/lib/python3.10/site-packages/flask/app.py
", line 1823, in full_dispatch_request
    rv = self.dispatch_request()
  File "/Users/nguerrer/Documents/Graduate_School/CS_531/Final_Prog/FinalProg/lib/python3.10/site-packages/flask/app.py
", line 1799, in dispatch_request
    return self.ensure_sync(self.view_functions[rule.endpoint])(**view_args)
  File "/Users/nguerrer/Documents/Graduate_School/CS_531/Final_Prog/queryServer.py", line 70, in login
    username = parse.get('username')[0]
TypeError: 'NoneType' object is not subscriptable
127.0.0.1 - - [29/Apr/2023 12:43:21] "GET /login?next=%2Fquery_database HTTP/1.1" 500 -
```

Auditor accessing the Audit Data at the same endpoint is authenticated by Flask Login manager and werkzeug.security Python library

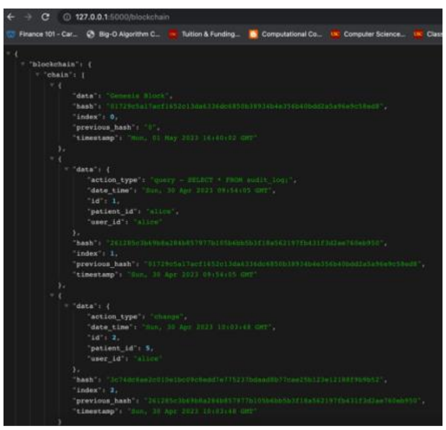
- Home
- Query Database
- Logout

Query Results

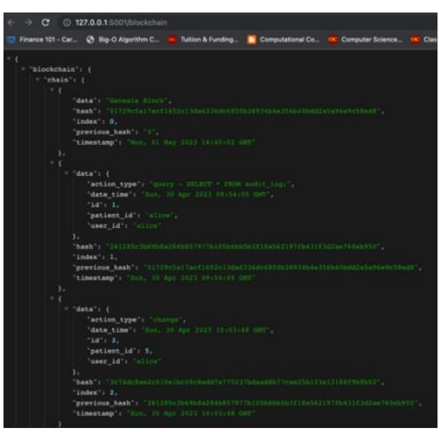
ID	Datetime	Patient ID	User ID	Action Type
1	2023-04-28 14:19:06.542923	alice	alice	query - SELECT * FROM audit_log;
2	2023-04-28 14:19:06.542923	4	alice	create
3	2023-04-28 14:21:07.001476	1	Nick	query - SELECT * FROM user WHERE id=1
4	2023-04-28 14:21:07.001476	4	alice	delete
5	2023-04-28 14:22:21.847462	4	alice	create
6	2023-04-28 14:22:21.847462	4	alice	delete
7	2023-04-28 14:22:21.847462	4	alice	create
8	2023-04-28 14:22:21.847462	4	alice	delete
9	2023-04-28 14:22:21.847462	4	alice	create
10	2023-04-28 14:22:21.847462	4	alice	delete
11	2023-04-28 14:22:21.847462	4	alice	create
12	2023-04-28 14:22:21.847462	4	alice	delete
13	2023-04-28 14:22:21.847462	4	alice	create
14	2023-04-28 14:22:21.847462	4	alice	delete
15	2023-04-28 14:22:21.847462	4	alice	create
16	2023-04-28 14:22:21.847462	4	alice	delete
17	2023-04-28 14:22:21.847462	4	alice	create

[5.7] Scenario 7: Multiple Servers Keep an Updated Version of the Ledger

On Port 5000 we have a blockchain of the Audit Log



On Port 5001 we have an exact blockchain Copy of the Audit Log



Notice every block has the same hash as the correspond block on the different port. A true exact copy. Every client who is a Super User will also have a copy of the Blockchain sent to them.

[6.0] System Assumptions and Limitation

[6.1] Prototype Assumptions—assignment requirements,

Assumption:

Assumption 1: The only threat model considered in this prototype is an insider to the system who was able to get Super User login credentials. We recognize that in a production environment cryptosystem requires precisely specifying formal methods. Specifically, we would need to consider network attacks such as Brute force attack, known plaintext dictionary, Replay attack, Man in the middle, Password sniffing, IP spoofing, Connection hijacking, SYN flooding.

Assumption 2: In a true production environment cryptosystem we would need to consider the organizational, legal, and regulatory policies however for this prototype we only consider 5 goals specified in **[5.0] Evaluation: SSDAS System Meets Design and Implementation Goals/Requirements**

Assumption3: The instructions asked for a PDF format with a font size of 12 points, single-spaced, single column. Not less than 10 and no more than 20 pages. Figures, tables, screenshots and the like are not included in the 20-page maximum page count. You will not be penalized for exceeding the page limit, but text beyond the 20-page limit will not be considered in grading. Therefore, please consider that if we removed this documents screenshots/figures/tables the document would actually be less than 20-pages. (Presently at 30)

Limitations:

Limitation 1: Since on bootup of the Main Server sends a copy of the blockchain to the Query Server the servers must be started in the order
1.) Query Server
2.) Main Server

Limitation 2: The Audit Log does not exceed 500 records
It should be noted that a substantially large Audit record will not be sent over HTTP to the Query Server or clients because the message would be too long. The message needs to be cut up into chunks and sent over the network but did not have time to implement this case handling.

Limitation 3: Main Server always logs in a User or Super User as the first action taken
On Login is when the blockchain is sent to the Query Server and client nodes.

Limitation 4: We are not using an actual networking to simulate packet flow among nodes.

Our implementation runs on a single machine with different ports being used as a simulation of an entirely different machine. Message exchange is done with the Python requests library and called from a client or a server. We explored using DETER (Cyber Defense Technology Experimental Research) for a realistic implementation that is scalable, distributed, and decentralized. Our key management system just uses a basic implementation that is easily substitutable for an actual key management implementation such as Diffie-Hellman key exchange for extra network security TLS 1.3

[6.2] Take MVP baseline and compare to EHR Blockchain References, Privacy Class, legal(weaknesses)—free flow like assignment big picture and actual prototype.

[7.0] Implementation:

[7.1] SSDAS Programs Design Written Description

In the Main Server (app.py) Flask is used to render HTML, redirect URLS, create session variables, create requests to the Query Server and serve json and appropriate endpoints

Flask Login is used to create a Login Manager that will authenticate users with the werkzeug.security libeary

SQLAlchemy is used to create SQL models that define the scheme of the User and Audit Log tables.

Buildmtree, genkeys, RSAcrypt, and blockchain are all custom code to meet the requirements of the assignment

All other libraries are to do basic datetime, and string manipulation

```
app.py > ...
from flask import Flask, render_template, redirect, url_for, request, session, jsonify
from flask_login import LoginManager, login_user, logout_user, login_required, UserMixin, current_user
from flask_sqlalchemy import SQLAlchemy
from werkzeug.security import check_password_hash, generate_password_hash
import sqlite3
import datetime
import re
import pytz
import requests
import json
from buildmtree import MerkleTree
import threading
from genkeys import get_keys
from RSAcrypt import encrypt, decrypt
from blockchain import Blockchain, DateTimeEncoder
```

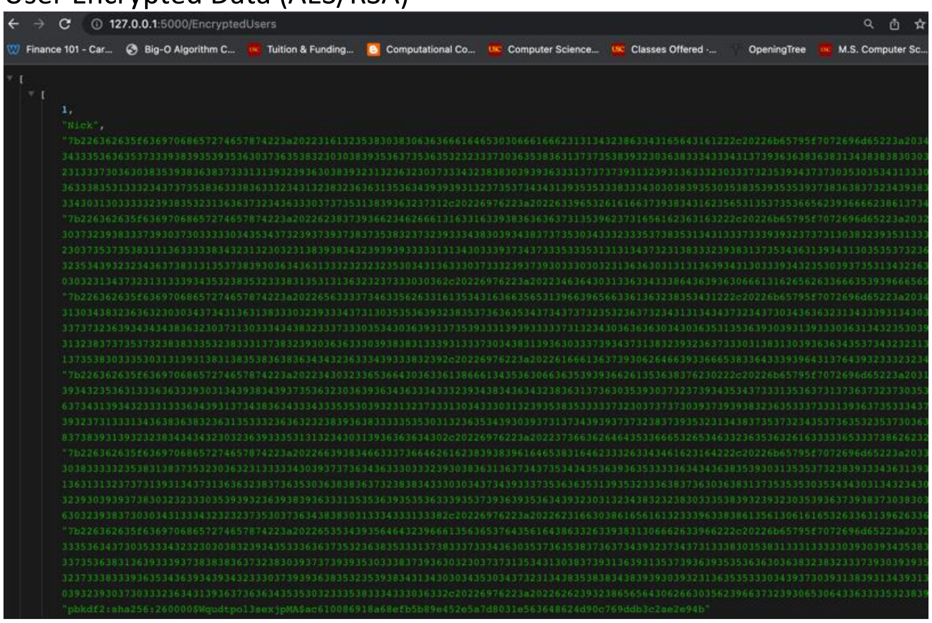
```python
from flask import Flask, request, redirect, url_for, jsonify
from flask_login import login_required, LoginManager, UserMixin, login_user, current_user
from werkzeug.security import check_password_hash
import sqlite3
from urllib.parse import urlparse, parse_qs
from blockchain import Blockchain
import json

```

[7.2] Demonstrate SSDAS Programs Work by Goals with Screen Capture

[7.2.1] Privacy: Patient privacy should be maintained. Unauthorized entities should not be able to access audit records.

Audit Records are protected by Flask Login Manager and additionally Patient User data is encrypted using (AES/RSA) and only decrypted by the Main Server when HTML is rendered at the appropriate time using a User's private key.

User Encrypted Data (AES/RSA)



Data is Encrypted in user table in Sqlite DB

```
sqlite> SELECT * FROM user;
1|Nick|7b226362635f63697068657274657874223a20223161323538303830636366616465303066616662313134323863343165643161222c2022
6b65795f7072696d65223a20343538323230363138383131383834363435343335363635373339383935393536303736353832303038393536373
63535323233373036353836313737353839323036383334333431373936363836383134383838303031383134353737393837373230343436383231
3373036303835393836383733313139323936303839323132363230373334323838303939363331373737393132393136333230333732353934373
7303535343133303931303830303236363930303131353331363338353133323437373538363338363332343132383236363135363439393931323
73537343431393535333833343030383935303538353935353937383638373234393834383936353136373731393234303731313334303130333
3323938353231363637323436333037373531383936323731 2c20226976223a20226339653261616637393834316235653135373536656239366662386
137346266227d|7b226362635f63697068657274657874223a2022623837393662346266613163316339383636363731353962373165616236316322
2c20226b65795f7072696d65223a20323339373234383638313035313532363338303732393833373930373033333034353437323937393738373
5383237323933343830393438373735303433323335373835313431333733393932373731303832393531333232353036363638363533383936393
1393230373537353831313633333834323132303231383938343239393933333131343033393734373335333531313134373231383332393831373
5343631393431303535373236373431393435383238393439363530383132353439323234363738313135373839303634363133323232323530343
1363330373332393739303330303231363630313131363934313033393432353039373531343236333638353539323539303431363832323130303
2313437323131333934353238353233383135313136323237333030362c20226976223a2022346364303133363343338643639363066613162656
2633666353939666565323222 7d|7b226362635f63697068657274657874223a2022656333373463356263316135343163663565313966396566336
1363238353431222c20226b65795f7072696d65223a203433303737383638333031363433303036333130343832363632303034373431363138333
0323933343
```


Data encrypted on /add_user route

```python
@app.route('/add_user', methods=['POST'])
@login_required
def add_user():
    name = request.form['name']
    get_keys(name)
    email = encrypt(request.form['email'], f'keys/{name}.pub')
    dob = encrypt(request.form['dob'], f'keys/{name}.pub')
    gender = encrypt(request.form['gender'], f'keys/{name}.pub')
    blood_type = encrypt(request.form['blood_type'], f'keys/{name}.pub')
    medical_condition = encrypt(request.form['medical_condition'], f'keys/{name}.pub')
    medication = encrypt(request.form['medication'], f'keys/{name}.pub')
```

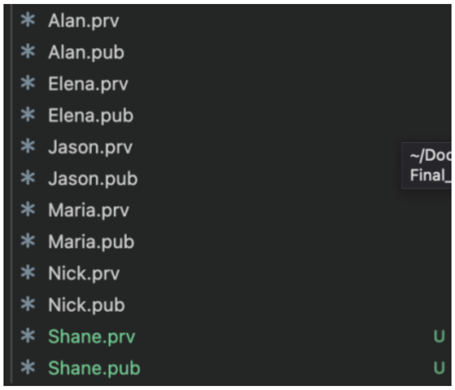
Data encrypted on /edit_user route

```python
@app.route('/edit_user/<int:user_id>', methods=['GET', 'POST'])
@login_required
def edit_user(user_id):
    user = User.query.get(user_id)
    if current_user.username not in loginUser.audit_users:
        return render_template('unauthorized.html'), 401
    if request.method == 'POST':
        user.name = request.form['name']
        user.email = encrypt(request.form['email'], "keys/" + user.name + ".pub")
        user.dob = encrypt(request.form['dob'], "keys/" + user.name + ".pub")
        user.gender = encrypt(request.form['gender'], "keys/" + user.name + ".pub")
        user.blood_type = encrypt(request.form['blood_type'], "keys/" + user.name + ".pub")
        user.medical_condition = encrypt(request.form['medical_condition'], "keys/" + user.name + ".
        user.medication = encrypt(request.form['medication'], "keys/" + user.name + ".pub")
```

On /index html render the data is decrypted using the private key of a user

```html
{% for user in users %}
<tr>

    <td>{{ user.id }}</td>
    <td>{{ user.name }}</td>
    <td>{{ decrypt(user.email, "keys/" + user.name + ".prv") }}</td>
    <td>{{ decrypt(user.dob, "keys/" + user.name + ".prv") }}</td>
    <td>{{ decrypt(user.gender, "keys/" + user.name + ".prv") }}</td>
    <td>{{ decrypt(user.blood_type, "keys/" + user.name + ".prv") }}</td>
    <td>{{ decrypt(user.medical_condition, "keys/" + user.name + ".prv") }}</td>
    <td>{{ decrypt(user.medication, "keys/" + user.name + ".prv") }}</td>
```

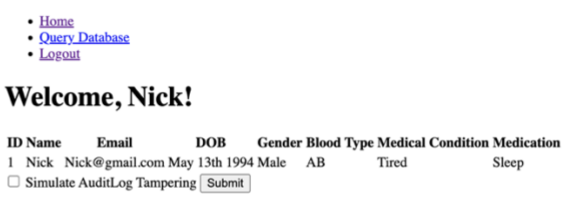
Keys Stored in app.py and generated on /add_user



[7.2.2] Identification and Authorization: All system users must be identified and authenticated. All requests to access the audit data should be authorized.

Super Users (audit companies) and Users (Regular Patients) are given login credentials

Patient Log-In Screen (Only the Regular User Record)



Audit User Login Screen (All Records)

- Home
- Query Database
- View Blockchain
- View MerkelTree
- View Encrypted Patient Data(RSA/AES)
- Logout

Welcome, alice!

ID	Name	Email	DOB	Gender	Blood Type	Medical Condition	Medication
1	Nick	Nick@gmail.com	May 13th 1994	Male	AB	Tired	Sleep
2	Alan	Alan@gmail.com	1994	male	AB	Tired	Sleep
3	Maria	maria@gmail.com	Feb 10 1959	Female	O	Flu	Pennicillin
4	Elena	Elena@gmail.com	March 13 1994	Female	O	Tired	Rest
5	Jason	Jason@gmail.com	Dec 19 1996	Male	O	Flu	antibiotics

Name | Email | DOB | Gender | Blood Type | Medical Condition | Medication | Add User

☐ Simulate AuditLog Tampering Submit

[7.2.3] Queries: Only authorized entities should be able to query audit records.

To be able to Query the SQL database through the Query Server each endpoint needs to be authenticated through the Flask Login manager and the password hashing and checking functions of the werkzeug.security Python Library. The Query Server will send back an appropriate response for the logged in User based on whether they are a Super User or a Regular User.

Query Database Screen

- Home
- Query Database
- Logout

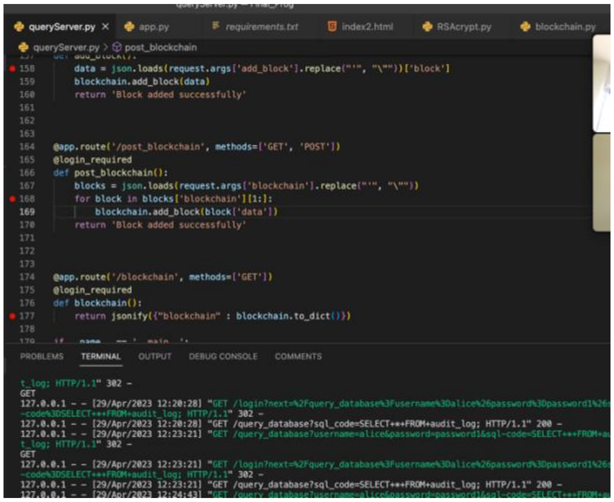
Query the Database

Enter your SQL code:

```
SELECT * FROM
audit_log;
```

Execute

Query Server Running on Port 5001



Audit Superuser Query Result for Audit Log (All Records shown)

- Home
- Query Database
- Logout

Query Results

ID	Datetime	Patient ID	User ID	Action Type
1	2023-04-28 14:19:06.542923	alice	alice	query - SELECT * FROM audit_log;
2	2023-04-28 14:19:06.542923	4	alice	create
3	2023-04-28 14:21:07.001476	1	Nick	query - SELECT * FROM user WHERE id=1
4	2023-04-28 14:21:07.001476	4	alice	delete
5	2023-04-28 14:22:21.847462	4	alice	create
6	2023-04-28 14:22:21.847462	4	alice	delete
7	2023-04-28 14:22:21.847462	4	alice	create
8	2023-04-28 14:22:21.847462	4	alice	delete
9	2023-04-28 14:22:21.847462	4	alice	create
10	2023-04-28 14:22:21.847462	4	alice	delete
11	2023-04-28 14:22:21.847462	4	alice	create
12	2023-04-28 14:22:21.847462	4	alice	delete
13	2023-04-28 14:22:21.847462	4	alice	create
14	2023-04-28 14:22:21.847462	4	alice	delete
15	2023-04-28 14:22:21.847462	4	alice	create
16	2023-04-28 14:22:21.847462	4	alice	delete
17	2023-04-28 14:22:21.847462	4	alice	create

Audit Superuser Query Result for Patient Data (All Patient Data shown)

- Home
- Query Database
- Logout

Query Results

ID	Name	Email	DOB	Gender	Blood Type	Medical Condition	Medication
1	Nick	Nick@gmail.com	May 13th 1994	Male	AB	Tired	Sleep
2	Alan	Alan@gmail.com	1994	male	AB	Tired	Sleep
3	Maria	maria@gmail.com	Feb 10 1959	Female	O	Flu	Pennicillin
4	Elena	Elena@gmail.com	March 13 1994	Female	O	Tired	Rest
5	Jason	Jason@gmail.com	Dec 19 1996	Male	O	Flu	antibiotics

Patient User Query Result for Audit Log (Only Audit Log records shown pertaining to User: Nick)

- Home
- Query Database
- Logout

Query Results

ID	Datetime	Patient ID	User ID	Action Type
3	2023-04-28 14:21:07.001476	1	Nick	query - SELECT * FROM user WHERE id=1
43	2023-04-29 12:15:45.090511	1	Nick	query - SELECT * FROM audit_log WHERE patient_id=1
44	2023-04-29 12:15:45.090511	1	alice	change

Patient User Query Result for Patient Data (Only records shown pertaining to User: Nick)

- Home
- Query Database
- Logout

Query Results

ID	Name	Email	DOB	Gender	Blood Type	Medical Condition	Medication
1	Nick	Nick@gmail.com	May 13th 1994	Male	O	Tired	Sleep

[7.2.4] Immutability: No one should be able to delete or change EXISTING audit records without detection. Any modifications/deletions of the audit records should be detected and reported. You can focus on attackers who are internal to the system modifying the audit data after it has already been received by the system. Note that you do not need to protect information against modification, it is sufficient to just detect and report any unauthorized modifications to the audit data.

Audit Log Immutability is guaranteed by the steps:

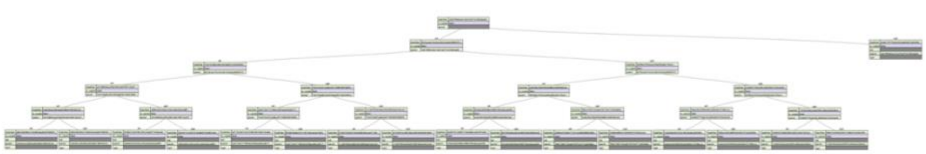
On generation & insertion of Audit Log record:

1.) Acquire mutex lock
2.) Given current Audit Log records recompute a Merkel Tree
3.) If the previous Merkel Tree's root is equal to the newly computed Merkel Tree's root then no tampering has occurred, else tampering has definitely occurred (First Merkel Tree is computed in Phase 1: Startup)
4.) If no tampering, insert Audit Log record
5.) Set the current Merkel Tree to be the Merkel Tree computed after the addition of the new Audit Log Record
6.) Release mutex lock

```
def checkLogImmutability():
    audit_logs = AuditLog.query.all()
    if len(audit_logs) != 0:
        mTreeInput = [f"{audit_record.date_time}|{audit_record.patient_id}|{audit_record.user_id}|{audit_record.action_type}" for audit_re
        mTree = MerkleTree(mTreeInput)

        if mTree.root.hashHex != AuditLog.mTree.root.hashHex:
            return False
    return True
```

This algorithm guarantees that any tampering will be detected by the system since no operations having to do with creating/checking the Merkel Tree can occur outside the mutex locks.

Merkel Tree Visualized



JSON Merkel Tree


```
{
  "content": "2023-04-28 14:19:06.542923|alice|alice|query - SELECT * FROM audit_log;+2023-04-28 14:19:06.542923|4|alice|c
  FROM user WHERE id=1+2023-04-28 14:21:07.001476|4|alice|delete+2023-04-28 14:22:21.847462|4|alice|create+2023-04-28 14:22
  14:22:21.847462|4|alice|create+2023-04-28 14:22:21.847462|4|alice|delete+2023-04-28 14:22:21.847462|4|alice|create+2023-
  14:22:21.847462|4|alice|create+2023-04-28 14:22:21.847462|4|alice|delete+2023-04-28 14:22:21.847462|4|alice|create+2023-
  14:22:21.847462|4|alice|create+2023-04-28 14:22:21.847462|4|alice|delete+2023-04-28 14:22:21.847462|4|alice|create+2023-
  14:22:21.847462|4|alice|create+2023-04-28 14:22:21.847462|4|alice|delete+2023-04-28 14:22:21.847462|4|alice|create+2023-
  14:22:21.847462|4|alice|delete+2023-04-28 14:22:21.847462|alice|alice|query - SELECT * FROM audit_log;+2023-04-28 14:31:2
  14:31:23.105559|4|alice|create+2023-04-28 14:38:36.862724|4|alice|delete+2023-04-28 14:38:36.862724|4|alice|create+2023-
  14:38:36.862724|5|alice|delete+2023-04-28 14:38:36.862724|4|alice|change+2023-04-28 14:38:36.862724|5|alice|create+2023-
  14:38:36.862724|5|alice|delete+2023-04-28 14:41:01.797816|5|alice|create+2023-04-28 14:54:56.724964|5|alice|change+2023-
  audit_log;+2023-04-28 14:54:56.724964|5|alice|query - SELECT * FROM user;+2023-04-28 14:54:56.724964|4|alice|query - SEL
  "hashHex": "74d9c0a957fbbb79b98dd8f894b84b5a2c8c3b3e30ad42ed52fa9ef8c04f8202",
  "is_copied": false,
  "left": {
    "content": "2023-04-28 14:19:06.542923|alice|alice|query - SELECT * FROM audit_log;+2023-04-28 14:19:06.542923|4|alic
    SELECT * FROM user WHERE id=1+2023-04-28 14:21:07.001476|4|alice|delete+2023-04-28 14:22:21.847462|4|alice|create+202
    14:22:21.847462|4|alice|create+2023-04-28 14:22:21.847462|4|alice|delete+2023-04-28 14:22:21.847462|4|alice|create+20
    14:22:21.847462|4|alice|create+2023-04-28 14:22:21.847462|4|alice|delete+2023-04-28 14:22:21.847462|4|alice|create+20
    14:22:21.847462|4|alice|create+2023-04-28 14:22:21.847462|4|alice|delete+2023-04-28 14:22:21.847462|4|alice|create+20
    14:22:21.847462|4|alice|create+2023-04-28 14:22:21.847462|4|alice|delete+2023-04-28 14:22:21.847462|4|alice|create+20
    14:22:21.847462|4|alice|delete+2023-04-28 14:22:21.847462|alice|alice|query - SELECT * FROM audit_log;+2023-04-28 14:
    04-28 14:31:23.105559|4|alice|create+2023-04-28 14:38:36.862724|4|alice|delete+2023-04-28 14:38:36.862724|4|alice|cre
    14:38:36.862724|5|alice|delete+2023-04-28 14:38:36.862724|4|alice|change+2023-04-28 14:38:36.862724|5|alice|create",
    "hashHex": "60dd0f66c47592ac1936c727fd61cb67acc2fb88e49c602df52ebb0754216a8d",
    "is_copied": false,
    "left": {
      "content": "2023-04-28 14:19:06.542923|alice|alice|query - SELECT * FROM audit_log;+2023-04-28 14:19:06.542923|4|
      SELECT * FROM user WHERE id=1+2023-04-28 14:21:07.001476|4|alice|delete+2023-04-28 14:22:21.847462|4|alice|create
      14:22:21.847462|4|alice|create+2023-04-28 14:22:21.847462|4|alice|delete+2023-04-28 14:22:21.847462|4|alice|create
      14:22:21.847462|4|alice|create+2023-04-28 14:22:21.847462|4|alice|delete+2023-04-28 14:22:21.847462|4|alice|create
      14:22:21.847462|4|alice|create+2023-04-28 14:22:21.847462|4|alice|delete",
      "hashHex": "ddb77b5f8106510b08f3327a812de5e4f231806ce7e074d249147d63c8298a07",
      "is_copied": false,
      "left": {
```

Button on the Home Screen to tamper with an Audit Log Record

☑ Simulate AuditLog Tampering [Submit]

Detection Screen Warning

Warning: Audit Log Tampering Detected!!!

Example of Tampering in Sqlite DB

```
sqlite> sqlite> SELECT * FROM audit_log;
1|2023-04-28 14:19:06.542923|alice|alice|Tamperered With
2|2023-04-28 14:19:06.542923|4|alice|create
3|2023-04-28 14:21:07.001476|1|Nick|query - SELECT * FROM user WHERE id=1
4|2023-04-28 14:21:07.001476|4|alice|delete
5|2023-04-28 14:22:21.847462|4|alice|create
6|2023-04-28 14:22:21.847462|4|alice|delete
7|2023-04-28 14:22:21.847462|4|alice|create
8|2023-04-28 14:22:21.847462|4|alice|delete
9|2023-04-28 14:22:21.847462|4|alice|create
10|2023-04-28 14:22:21.847462|4|alice|delete
11|2023-04-28 14:22:21.847462|4|alice|create
12|2023-04-28 14:22:21.847462|4|alice|delete
13|2023-04-28 14:22:21.847462|4|alice|create
14|2023-04-28 14:22:21.847462|4|alice|delete
15|2023-04-28 14:22:21.847462|4|alice|create
16|2023-04-28 14:22:21.847462|4|alice|delete
17|2023-04-28 14:22:21.847462|4|alice|create
```

Main Function to deal with the Immutability requirement

```
def manageAuditLog(log):
    # create a lock object
    lock = threading.Lock()
    # acquire the lock before executing the code
    lock.acquire()
    try:
        isImmutable = checkLogImmutability()

        if isImmutable == False:
            audit_record = AuditLog.query.get(1)
            audit_record.action_type = "query - SELECT * FROM audit_log;"
            db.session.commit()
            return False
        #  create audit log entry
        db.session.add(log)
        db.session.commit()

        #recreate Merkel Tree
        createMerkelTree()

        # add audit log entry as a block to the blockchain
        blockchain.add_block(log.to_dict())

         # Post blockchain block
        password = session.get('password')
        block = json.dumps(log.to_dict(), sort_keys=True, cls=DateTimeEncoder)
        response = requests.post('http://127.0.0.1:5001/add_block', data={'block' : block}, params={'username': current_user.id, 'password': password, 'block' : block})

    finally:
        # release the lock
        lock.release()
        if isImmutable == False:
            return False
        return True
```

[7.2.6] Identification and authorization**:** Identification refers to the process of verifying the identity of an individual or entity. Authorization refers to the process of granting access to a particular resource or service based on the verified identity of the user or system.

Main Server Flask Login route that is used to Identify and authenticate session of a User on access to every endpoint

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        session['password'] = password
        user = loginUser(username)

        if not user.is_allowed(data={'username': username, 'password': password}):
            return render_template('unauthorized.html'), 401
        if username in loginUser.audit_users:
            if not check_password_hash(loginUser.audit_users[username], password):
                return render_template('unauthorized.html'), 401
        else:
            response = requests.get('http://127.0.0.1:5001/get_hash', params={'username': use
            payload = json.loads(response.text)
            password_hash = payload['password_hash']

            if not check_password_hash(password_hash, password):
                return render_template('unauthorized.html'), 401
        login_user(user)

        # Post blockchain data
        password = session.get('password')
        json_blockchain = json.dumps(blockchain.to_dict()['chain'], sort_keys=True, cls=DateT
        response = requests.post('http://127.0.0.1:5001/post_blockchain', data={'blockchain'

        return redirect(url_for('index'))
    return render_template('login.html')
```

Query Server Flask Login route that is used to Identify and authenticate session of a User on access to every endpoint

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    print(request.method)
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        user = loginUser(username)
        if not user.is_allowed():
            return
        if username in loginUser.audit_users:
            if not check_password_hash(loginUser.audit_users[username], password):
                return
        else:

            conn = sqlite3.connect('instance/sqlite.db')
            cursor = conn.cursor()
            cursor.execute(f"SELECT password_hash FROM user where name='{username}'")
            password_hash = cursor.fetchall()[0][0]

            if not check_password_hash(password_hash, password):
                return
        login_user(user)
        return "login success"
    parse = parse_qs(urlparse(request.args['next']).query)
    username = parse.get('username')[0]
    password = parse.get('password')[0]

    user = loginUser(username)
    if not user.is_allowed():
        return
    if username in loginUser.audit_users:
        if not check_password_hash(loginUser.audit_users[username], password):
            return
    else:

        conn = sqlite3.connect('instance/sqlite.db')
        cursor = conn.cursor()
        cursor.execute(f"SELECT password_hash FROM user where name='{username}'")
        password_hash = cursor.fetchall()[0][0]

        if not check_password_hash(password_hash, password):
            return
    login_user(user)
    path = urlparse(request.args['next']).path[1:]
    if path == 'query_database':
        return redirect(url_for(path, sql_code=parse.get('sql-code')[0]))
    elif path == 'post_blockchain':
        return redirect(url_for(path, blockchain={"blockchain" : json.loads(parse.get('blockchain')[0])}))
    elif path == 'add_block':
        return redirect(url_for(path, add_block={"block" : json.loads(parse.get('block')[0])}))
    return redirect(url_for(path, username=username))
```

[7.2.6] Decentralization: The system should not rely on a single trusted entity to support immutability. This means there is no single entity (organization) that controls the audit logs. Blockchain-based technology works well for such systems.

Every node in the network is posted a copy of the blockchain from the Main Server. On an Audit Log record creation, a new block is added to the Audit Log, the SQL DB is updated, and the new block is posted to each node on the network. The hashes of each block preserve the consistency

Blockchain code:

```
blockchain.py > Blockchain > add_block
import hashlib
import json
from datetime import datetime
import pytz

class DateTimeEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, datetime):
            return obj.isoformat()
        return json.JSONEncoder.default(self, obj)

# Define the block class
class Block:
    def __init__(self, index, timestamp, data, previous_hash):
        self.index = index
        self.timestamp = timestamp
        self.data = data
        self.previous_hash = previous_hash
        self.hash = self.calculate_hash()

    def calculate_hash(self):
        sha = hashlib.sha256()
        block_string = json.dumps(self.__dict__, sort_keys=True, cls=DateTimeEncoder)
        sha.update(block_string.encode('utf-8'))
        return sha.hexdigest()

# Define the blockchain class
class Blockchain:
    def __init__(self):
        self.chain = [self.create_genesis_block()]

    def create_genesis_block(self):
        return Block(0, datetime.now(pytz.timezone('US/Pacific')), "Genesis Block", "0")

    def add_block(self, data):
        # block = Block(len(self.chain), datetime.now(pytz.timezone('US/Pacific')), data, self.chain[-1].hash)
        block = Block(len(self.chain), data['date_time'], data, self.chain[-1].hash)

        self.chain.append(block)

    def to_dict(self):
```

Query Server route's to POST blockchain and new blocks to

```
@app.route('/add_block', methods=['GET', 'POST'])
@login_required
def add_block():
    data = json.loads(request.args['add_block'].replace("'", "\""))['block']
    data['date_time'] = datetime.strptime(data['date_time'], '%Y-%m-%dT%H:%M:%S.%f')
    blockchain.add_block(data)
    return 'Block added successfully'


@app.route('/post_blockchain', methods=['GET', 'POST'])
@login_required
def post_blockchain():
    blocks = json.loads(request.args['blockchain'].replace("'", "\""))
    blocks['blockchain'][0]['timestamp'] = datetime.fromisoformat(blocks['blockchain'][0]['timestamp'].replace('Z', '+00:00'))
    blockchain.chain[0] = Block(blocks['blockchain'][0]['index'],
                                blocks['blockchain'][0]['timestamp'],
                                blocks['blockchain'][0]['data'],
                                blocks['blockchain'][0]['previous_hash'])
    for block in blocks['blockchain'][1:]:
        block['data']['date_time'] = datetime.strptime(block['data']['date_time'], '%Y-%m-%dT%H:%M:%S.%f')
        blockchain.add_block(block['data'])
    return 'Block added successfully'


@app.route('/blockchain', methods=['GET'])
@login_required
def blockchain():
    return jsonify({"blockchain" : blockchain.to_dict()})
```

[7.3] Explanation of SSDAS Programs Execution and Inputs/Outputs

- Home
- Query Database
- View Blockchain
- View MerkelTree
- View Encrypted Patient Data(RSA/AES)
- Logout

Welcome, alice!

ID	Name	Email	DOB	Gender	Blood Type	Medical Condition	Medication
1	Nick	Nick@gmail.com	May 13th 1994	Male	AB	Tired	Sleep
2	Alan	Alan@gmail.com	1994	male	AB	Tired	Sleep
3	Maria	maria@gmail.com	Feb 10 1959	Female	O	Flu	Pennicillin
4	Elena	Elena@gmail.com	March 13 1994	Female	O	Tired	Rest
5	Jason	Jason@gmail.com	Dec 19 1996	Male	O	Flu	antibiotics
6	Shane	Shane@gmail.com	April 20th 1994	Male	B	Sober	Booze

Name | Email | DOB | Gender | Blood Type | Medical Condition | Medication | Add User

☐ Simulate AuditLog Tampering Submit

[7.4] Submit SSDAS Code (submitted separately)

[8.0] Demo Recording: Demonstrate how SSDAS System Works (submitted separately)

References

[1] Mihailescu, M. I., & Nita, S. L. (2021). *Cryptography and Cryptanalysis in MATLAB*. Apress.

[2] Stamatellis, C., Papadopoulos, P., Pitropakis, N., Katsikas, S., & Buchanan, W. J. (2020). A privacy-preserving healthcare framework using hyperledger fabric. *Sensors*, *20*(22), 6587.

[3] Tith, D., Lee, J. S., Suzuki, H., Wijesundara, W. M. A. B., Taira, N., Obi, T., & Ohyama, N. (2020). Application of blockchain to maintaining patient records in electronic health record for enhanced privacy, scalability, and availability. *Healthcare informatics research*, *26*(1), 3-12.

[4] Madine, M. M., Battah, A. A., Yaqoob, I., Salah, K., Jayaraman, R., Al-Hammadi, Y., ... & Ellahham, S. (2020). Blockchain for giving patients control over their medical records. *IEEE Access*, *8*, 193102-193115.

[5] Dubovitskaya, A., Xu, Z., Ryu, S., Schumacher, M., & Wang, F. (2017). Secure and trustable electronic medical records sharing using blockchain. In *AMIA annual symposium proceedings* (Vol. 2017, p. 650). American Medical Informatics Association.

[6] Xia, Q. I., Sifah, E. B., Asamoah, K. O., Gao, J., Du, X., & Guizani, M. (2017). MeDShare: Trust-less medical data sharing among cloud service providers via blockchain. *IEEE access*, *5*, 14757-14767.

[7] Shi, S., He, D., Li, L., Kumar, N., Khan, M. K., & Choo, K. K. R. (2020). Applications of blockchain in ensuring the security and privacy of electronic health record systems: A survey. *Computers & security*, *97*, 101966.

[8] Ekblaw, A., Azaria, A., Halamka, J. D., & Lippman, A. (2016, August). A Case Study for Blockchain in Healthcare:"MedRec" prototype for electronic health records and medical research data. In *Proceedings of IEEE open & big data conference* (Vol. 13, p. 13).

[9] Rezaeibagha F, Win KT, Susilo W. A systematic literature review on security and privacy of electronic health record systems: Technical perspectives. *Health Information Management Journal*. 2015;44(3):23-38. doi:10.1177/183335831504400304

[10] Sheikh Mohammad Idrees PA. *Blockchain for Healthcare Systems: Challenges, Privacy, and Securing of Data*. CRC Press; 2021. doi:10.1201/9

[11] Adler-Milstein J, Adelman JS, Tai-Seale M, Patel VL, Dymek C. EHR audit logs: A new goldmine for health services research? *Journal of biomedical informatics*. 2020;101:103343-103343. doi:10.1016/j.jbi.2019.103343

[12] Kannampallil T, Adler-Milstein J. Using electronic health record audit log data for research: insights from early efforts. Journal of the American Medical Informatics Association : JAMIA. 2022;30(1):167-171. doi:10.1093/jamia/ocac173

[13] Huilgol YS, Adler-Milstein J, Ivey SL, Hong JC. Opportunities to use electronic health record audit logs to improve cancer care. Cancer medicine (Malden, MA). 2022;11(17):3296-3303. doi:10.1002/cam4.4690

[14] Reis, E. (2011). The lean startup. *New York: Crown Business*, *27*, 2016-2020.