University of Southern California

CSCI531 Applied Cryptogrophy
Final Project

EHR Audit System
System Report

Student: Hallgrimur David Egilsson
USC ID: 6059-2639-79

Professor: Dr. Tatyana Ryutov

May 2, 2022

I have read the Guide to Avoiding Plagiarism published by the student affairs office. I understand what is expected of me with respect to properly citing sources, and how to avoid representing the work of others as my own. The material in this paper was written by me, except for such material that is quoted or indented and properly cited to indicate the sources of the material. I understand that using the words of others, and simply tagging the sentence, paragraph, or section with a tag to the copied source does not constitute proper citation and that if such material is used verbatim or paraphrased it must be specifically conveyed (such as through the use of quotation marks or indentation) together with the citation. I further understand that overuse of properly cited quotations to avoid conveying the information in my own words, while it will not subject me to disciplinary action, does convey to the instructor that I do not understand the material enough to explain it in my own words, and will likely result in a lesser grade on the paper.

Signed: Hallgrimur David Egilsson

1 System architecture

Figure 1 shows a high level overview of the main software components of the system.

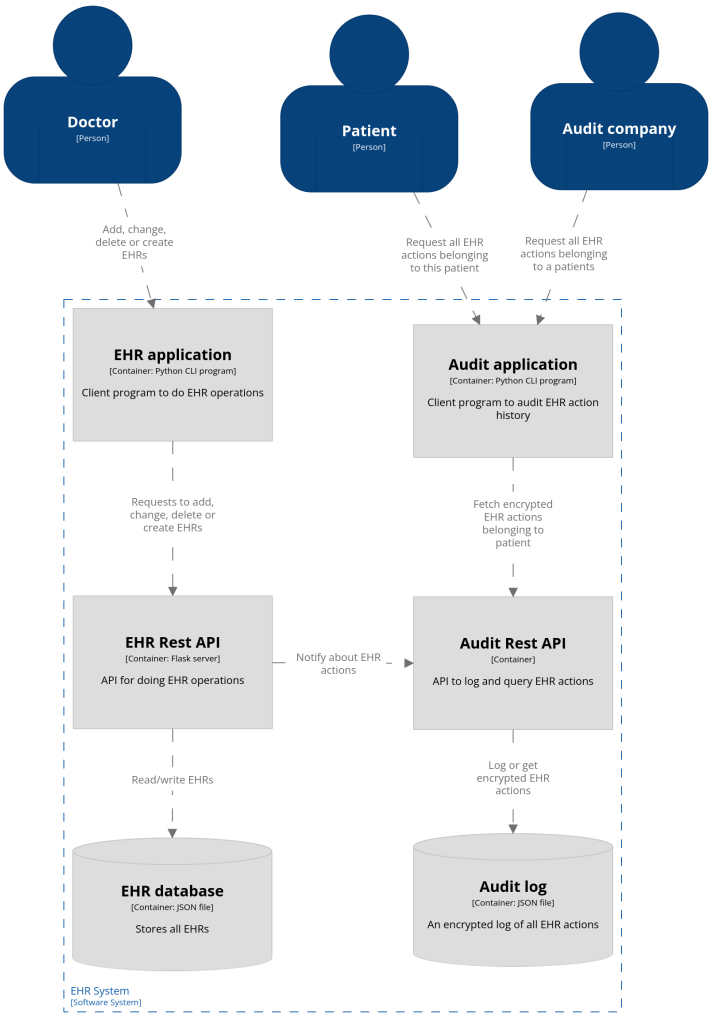

Figure 1: System overview

First lets take a look at the EHR application, this is an application that only doctors should have access to which they can use during patient visits to modify EHRs (Electronic Health Records). The EHR application sends request to the EHR Rest API which handles reading and writing to the EHR database and notifying the Audit Rest API about EHR actions. Seperating the EHR application and the EHR Rest API allows the EHR application to be implemented with least priviledge, it should only be able to request actions on EHRs but not directly access the EHR database or independently call the Audit Rest API.

The Audit application is for patients and audit companies to read the EHR action history about a patient. The EHR action history of all patients is stored encrypted in the Audit log. Patients can only decrypt their own records in the audit log so direct access from the Audit application to the Audit log would not be too bad but still would leak some information, for example number of EHR actions in the log. Preferably the Audit application should only receive the patient's records which is what the Audit Rest API sends to the Audit application. The Audit application then proceeds to decrypt the patient's records and show them to the Audit application user.

Audit companies use the Audit application in the same way as a patient does but are authorized to fetch records for all patients registered under them.

2 Cryptographic system architecture

The overview in Figure 1 is a simplified view of the system because it does not show details like a Certificate Authority, system admin scripts, key storage and how the desired security goals are achieved. This chapter will explain the the cryptographic system architecture of the system.

Since the instructions for this project specified that the project should focus on the security of the audit log there were no security mechansisms designed or implemented for the EHR application, EHR Rest API or EHR database. In this project, these components are simply used to populate data in the audit log and to demonstrate the authorization mechansism in Audit Rest API which only allows the EHR Rest API to notify about EHR

actions.

2.1 Authentication and encryption in transit

To be able to enforce authorization rules, clients connecting to Rest APIs must be authenticated, and to protect against attacks such as MitM or network sniffing, data in transit must be encrypted and integrity preserved.

In the design phase of this project it was explored to authenticate and encrypt messages using a similar design as in PGP [1].

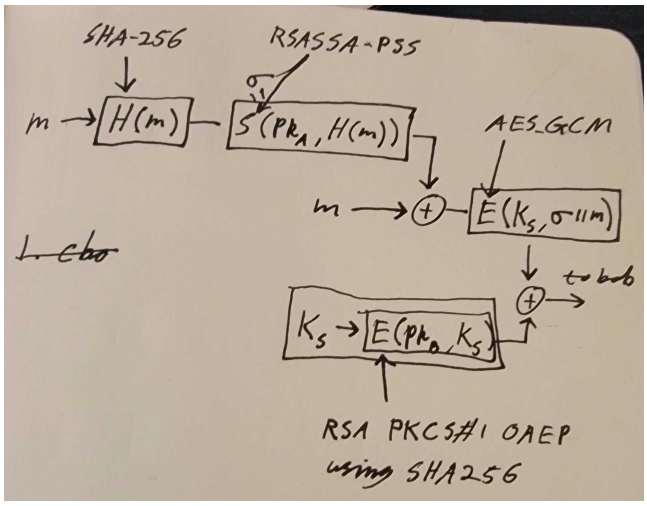

Figure 2: Initial design idea for authentication and encryption in transit

Upon further inspection it was realized that there could be issues with this approach. For example, this design needs to be extended to protect against replay attacks. So at that point it was realized that there is a lot of room for error and mistakes in this design and it needed some rethinking.

Using TLS is a very common approach to solving the encryption-in-transit problem. On the web TLS is usually configured to authenticate only the server and afterwards the client is authenticated using password-based authentication at the application layer. Although not common on the web, the TLS protocol can be configured to do mutual authentication, which requires the client to hold a certificate as well as the server. Additionally

it's designed to protect against attacks such as replay attacks [**?**] which is an improvement from the previous design discussed.

Authenticating clients with TLS removes the need for implementing components needed for password-based authentication such as a database of salted and hashed passwords. However the TLS mutual authentication method requires the client to hold a certificate and a private key so connecting from multiple different devices becomes more difficult. Additionally the TLS solution requires a Certificate Authority to sign user certificates. It was anticipated that the client would need to hold more keys than the TLS keys so TLS with mutual authentication was picked to solve the authentication and encryption in transit problem.

2.2 Encryption and integrity at rest

Records in the audit log are encrypted at rest. The encryption needs to fulfill the following goals:

1. For any given record in the database, only the patient the record belongs to, and the audit company of the patient, should be able to decrypt the EHR action included in the record.
2. For any given record in the database, the Audit Rest API should be able to decrypt the patient's name in the record but not any other parts of it. This allows the Audit Rest API to find records for any given patient.
3. Given access to the database, patients can discover which records belong to them but should not be able to discover the patient of other records, the same applies to Audit companies.
4. If any record is tampered with, it should be detected and reported.
5. If any record is deleted, it should be detected and reported.
6. The database cannot be deleted by any single person or company.

A record in the audit log is split into two parts: the patient's name and the EHR action.

The patient's name and the EHR action are encrypted using a hybrid cryptosystem [**?**]. The hybrid cryptosystem first generates a random 256-bit key and then encrypts the data using AES in Galois Counter Mode (GCM) which is a NIST recommendation for authenticated encryption [**?**]. The GCM mode generates a tag that is stored along with the ciphertext which is used to detect tampering with the ciphertext or tag. The AES key is encrypted using RSA PKCS#1 OAEP which requires a RSA public key. Remember that the database record is split into two parts, to encrypt the AES key to decrypt the patient's name, the Audit Rest API's RSA public key is used. The AES key to decrypt the EHR action is encrypted using two different RSA keys, one is the patient's public key and the other is the audit company's public key.

Although the AES ciphertext and tag are tamper-proof the encrypted RSA key is not, because RSA PKCS#1 OAEP is not an authenticated encryption mechanism. So with the above design there is no protection against the encrypted AES key being tampered with. Therefore there was added one last step to the hybrid encryption mechanism: signing the encrypted AES key using RSASSA-PSS. The encrypted AES key is signed using a public RSA key of the Audit Rest API. To avoid using the same keys for RSA PKCS#1 OAEP and RSASSA-PSS which could possibly decrease the overall security the system, the Audit Rest API holds two pairs of RSA keys, one set for encryption and decryption, the other set for signing and verifying.

The encryption mechanism described above protects against goals 1.-4. but does not fulfill goals 5. and 6. Although each record is tamper-proof there is no mechanism to detect if a record has been deleted as a whole. Additionally the database is centralized so whoever has access to the database, such as a system admin can delete it. Note that although it's possible to change the order of records in the database without detection it has no real effect since the plaintext of the EHR actions contain timestamps.

To achieve goal 6. the database must be decentralized and to achieve goal 5. there needs to be something that ties each record to the previous records. A blockchain cryptosystem would provide both of these features. For the blockchain to be truly decentralized the blockchain nodes cannot all be controlled by the same person or company. The blockchain will contain the records of the audit log which means that all parties controlling

a blockchain node will learn the contents of the database. Since the records in the database are encrypted storing the database in a decentralized blockchain cryptosystem should not leak information about the contents of the EHR actions or its patients. However, if each record would be a block in the blockchain it would however leak information to the blockchain participants about how many EHR actions there are in the database and such as how fast they are growing. More generally, adversaries could perform traffic analysis on the data.

2.3 Authorization rules

Authorization is enforced in the Audit Rest API. The Audit Rest API requires all connecting clients to authenticate through TLS. For each call to the API endpoint the application layer reads the id of the connecting client from the TLS connection context and then uses the id to enforce authorization rules at the application layer.

The authorization rules enforced are as follows:

1. Only the EHR Rest API is allowed to notify the Audit Rest API about EHR actions.
2. A patient can only fetch encrypted EHR actions belonging to them.
3. An audit company can only fetch encrypted EHR actions belonging to a patient that's registered under the audit company.

3 Implementation

The project was implemented on a Ubuntu 21.04 machine but should run macOS as well. Since the project mostly relies on Python packages, getting it to run on Windows should not require a lot of changes either.

3.1 IDs

The user ID of the five users in the system are simply their names: `alice`, `bob`, `carol`, `david`, `eve`. The server names have similar simple IDs: `ehr_server` and `audit_server` and finally the two audit companies have the IDs `usc` and `ucla`.

3.2 Audit company authorization rules

It was decided that USC should be able to query the EHR action history of alice bob and eve. UCLA should be able to query the EHR action history of carol and david.

3.3 The Network

The project's proof of concept implementation runs on the local loopback interface. Although using networking was not a requirement of the project the benefits of doing it is that it helps keep component code organized, more similar to production systems and it makes using TLS is easier. Both the Rest APIs of the project only accept HTTPS connections and their TLS certificates bind their hostnames to their TLS public keys. Therefore therefor their domain names must be added to the local DNS. In linux this is possible by tying their domain names to the local loopback interface in `/etc/hosts`:

```
$ cat /etc/hosts | grep server
127.0.0.1       ehr_server
127.0.0.1       audit_server
(.venv) david1471@arch:~
```

`ehr_server` was run on port 5000 and `audit_server` was run on port 5001.

3.4 Applications, REST APIs and databases

For simplicity the following technologies were chosen for each type of the software components:

Type	Technology
Application	Python CLI program
REST API	Python Flask server
Database	JSON file

For TLS the Certificate Authority's keys are cert are stored in a `system_admin` folder in the implementation.

3.5 Key generation

The software components and users in the system all hold various private keys. For the project it was decided that they should be generated by a system admin and distributed to users and software components out of bounds.

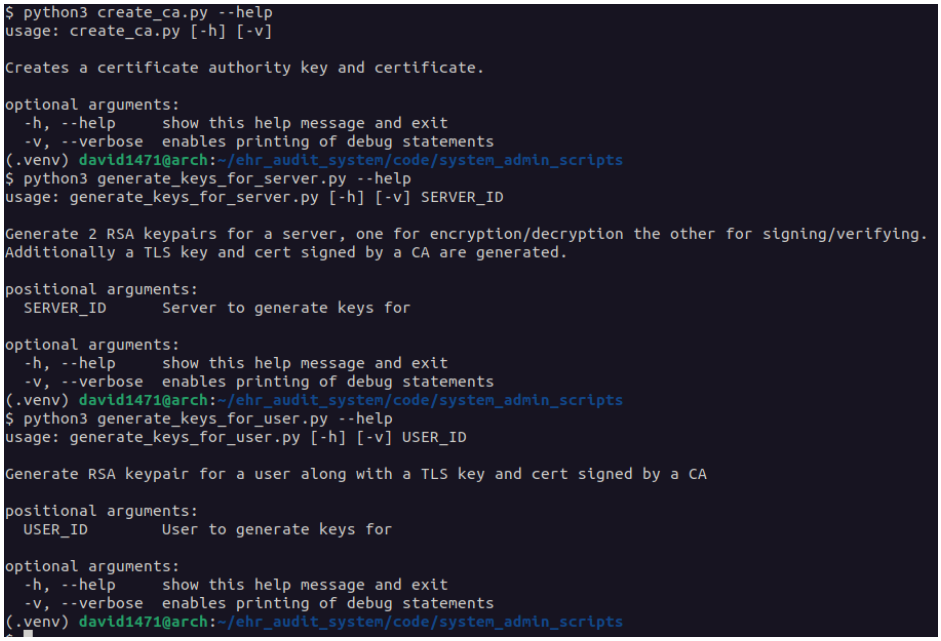

Figure 3: Usage instructions of 3 system admin scripts implemented

The key generation scripts depend on `openssl` being installed and on path. OpenSSL 'Version 1.1.1j Feb 16 2021' was used in this project.

3.6 TLS

Audit server's Flask application was configured to use TLS and require mutual authentication:

```
42  # Configure TLS for client authentication
43  ssl_context = ssl.create_default_context(
44      purpose=ssl.Purpose.CLIENT_AUTH, cafile=ca_cert
45  )
46
47  # Load server TLS key and certificate
48  ssl_context.load_cert_chain(
49      certfile=server_cert, keyfile=server_key, password=server_key_password
50  )
51
52  # Configure TLS to require client certificate signed by CA
53  ssl_context.verify_mode = ssl.CERT_REQUIRED
```

```
112 def run():
113     audit_db.initialize()
114     log.setLevel(logging.DEBUG)
115     app.run(
116         debug=True,
117         port=5001,
118         ssl_context=ssl_context,
119         request_handler=PeerCertWSGIRequestHandler,
120     )
```

Figure 4: The TLS configuration for audit_server

To evade old and vulnerable versions of TLS it only accepts TLS 1.2 and TLS 1.3 connections. Preferably it should only accept TLS 1.3 connection but the easiest way to fix that would be to upgrade Python to version 3.10 which was decided to be out of scope for this project.

```
(.venv) david1471@arch:~/ehr_audit_system/code/ehr_server
$ openssl s_client -CAfile ../ca/keys/ca.crt -cert keys/ehr_server_tls.crt -key keys/ehr_server_tls.key -connect audit_server:5001 -tls1_3 >/dev/null 2>&1 0>&1; echo $?;
openssl s_client -CAfile ../ca/keys/ca.crt -cert keys/ehr_server_tls.crt -key keys/ehr_server_tls.key -connect audit_server:5001 -tls1_2 >/dev/null 2>&1 0>&1; echo $?;
openssl s_client -CAfile ../ca/keys/ca.crt -cert keys/ehr_server_tls.crt -key keys/ehr_server_tls.key -connect audit_server:5001 -tls1_1 >/dev/null 2>&1 0>&1; echo $?;
openssl s_client -CAfile ../ca/keys/ca.crt -cert keys/ehr_server_tls.crt -key keys/ehr_server_tls.key -connect audit_server:5001 -tls1 >/dev/null 2>&1 0>&1; echo $?;
0
0
1
1
(.venv) david1471@arch:~/ehr_audit_system/code/ehr_server
```

Figure 5: Verifying that the server only accepts TLS versions >= 1.2

The TLS communications were also monitored through Wireshark as an exercise.

3.7 The EHR application - generating data

.

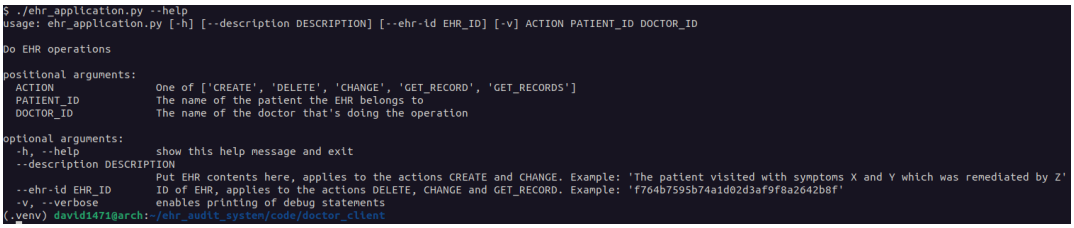

Figure 6: Usage instructions for the EHR application

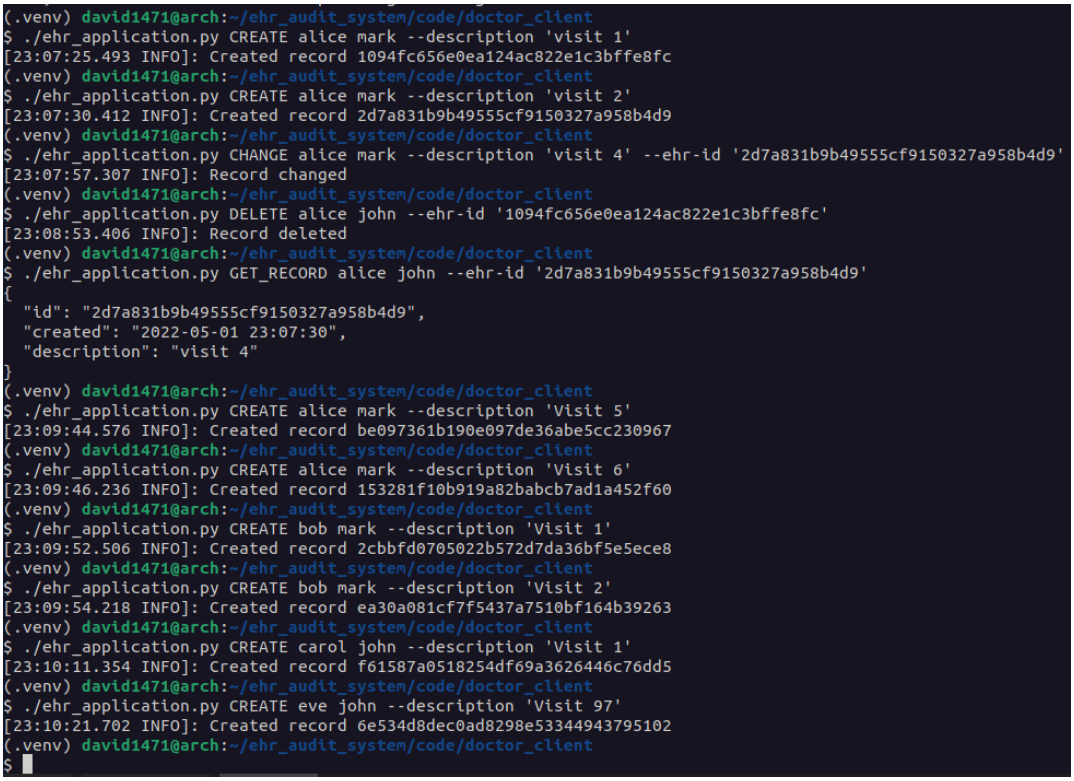

Figure 7: Generating data using the EHR application

.

Figure 8: The generated data in the EHR database.

.

3.8 The Audit log application - querying EHR action history

3.8.1 Fetching a record that has been tampered with

Now one of the patient's name in the audit log database has been tampered with. It's detected once a user tries to fetch records:

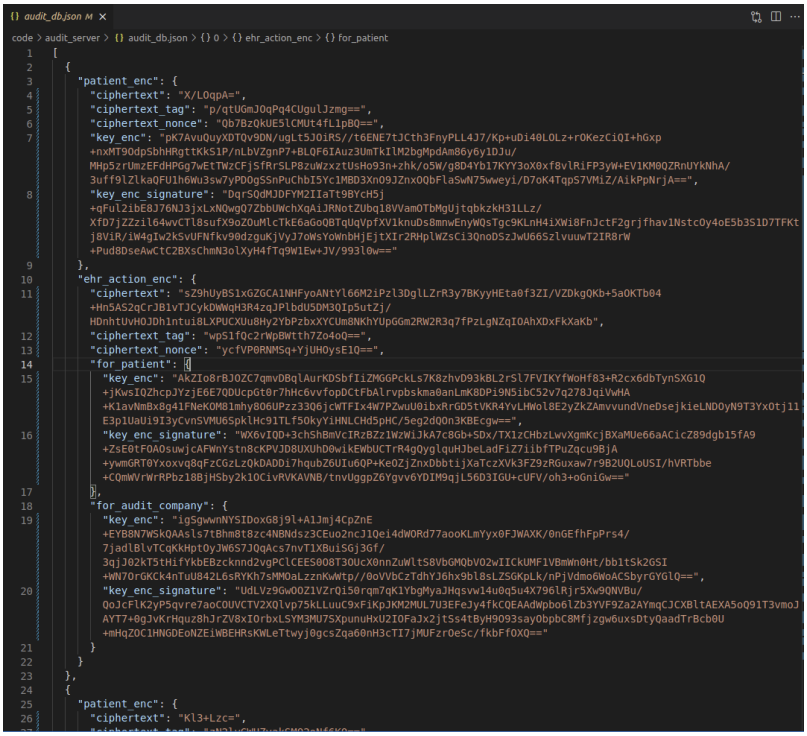

Figure 9: The first few lines of the audit log database

```
$ ./audit_application.py --help
usage: audit_application.py [-h] [-v] PATIENT USER_KEY_DIR

Query EHR action history for patient

positional arguments:
  PATIENT        Patient to fetch EHR usage about
  USER_KEY_DIR   A directory with the keys user should have received from a system admin.

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  enables printing of debug statements
(.venv) david1471@arch:~/ehr_audit_system/code/audit_client
```

Figure 10: Usage instructions for the Audit application

Lets suppose the database has been restored again to its previous healthy state but now the EHR action part of one of alice's record has been tampered with, in this case the tamper detection happens client side (in the Audit application):

```
(.venv) david1471@arch:~/ehr_audit_system/code/audit_client
$ ls keys/alice/
audit_server_rsa_verify.pem  rsa_decrypt.pem  rsa_encrypt.pem  tls.crt  tls.key
```

Figure 11: Alice's keys

```
$ ./audit_application.py alice keys/alice
[
  {
    "time": "2022-05-01 23:07:25",
    "patient": "alice",
    "requested_by": "mark",
    "action": "CREATE",
    "ehr_id": "1094fc656e0ea124ac822e1c3bffe8fc"
  },
  {
    "time": "2022-05-01 23:07:30",
    "patient": "alice",
    "requested_by": "mark",
    "action": "CREATE",
    "ehr_id": "2d7a831b9b49555cf9150327a958b4d9"
  },
  {
    "time": "2022-05-01 23:07:57",
    "patient": "alice",
    "requested_by": "mark",
    "action": "CHANGE",
    "ehr_id": "2d7a831b9b49555cf9150327a958b4d9"
  },
  {
    "time": "2022-05-01 23:08:53",
    "patient": "alice",
    "requested_by": "john",
    "action": "DELETE",
    "ehr_id": "1094fc656e0ea124ac822e1c3bffe8fc"
  },
  {
    "time": "2022-05-01 23:09:20",
    "patient": "alice",
    "requested_by": "john",
    "action": "GET_RECORD",
    "ehr_id": "2d7a831b9b49555cf9150327a958b4d9"
  },
  {
    "time": "2022-05-01 23:09:44",
    "patient": "alice",
    "requested_by": "mark",
    "action": "CREATE",
    "ehr_id": "be097361b190e097de36abe5cc230967"
  },
  {
    "time": "2022-05-01 23:09:46",
    "patient": "alice",
    "requested_by": "mark",
    "action": "CREATE",
    "ehr_id": "153281f10b919a82babcb7ad1a452f60"
  }
]
(.venv) david1471@arch:~/ehr_audit_system/code/audit_client
```

Figure 12: Alice sucessfully fetching her EHR action history

```
$ ./audit_application.py bob keys/alice
[23:23:50.795 ERROR]: Failed to get records, audit_server responded with HTTP returncode 403 and the messa
ge: 'alice' is not authorized to get EHR actions of 'bob'
```

Figure 13: Alice trying to fetch bob's history

```
$ ./audit_application.py alice keys/usc | head -6
[
  {
    "time": "2022-05-01 23:07:25",
    "patient": "alice",
    "requested_by": "mark",
    "action": "CREATE",
(.venv) david1471@arch:~/ehr_audit_system/code/audit_client
$ ./audit_application.py bob keys/usc
[
  {
    "time": "2022-05-01 23:09:52",
    "patient": "bob",
    "requested_by": "mark",
    "action": "CREATE",
    "ehr_id": "2cbbfd0705022b572d7da36bf5e5ece8"
  },
  {
    "time": "2022-05-01 23:09:54",
    "patient": "bob",
    "requested_by": "mark",
    "action": "CREATE",
    "ehr_id": "ea30a081cf7f5437a7510bf164b39263"
  }
]
(.venv) david1471@arch:~/ehr_audit_system/code/audit_client
$ ./audit_application.py carol keys/usc
[23:27:28.572 ERROR]: Failed to get records, audit_server responded with HTTP returncode 403 and the messa
ge: 'usc' is not authorized to get EHR actions of 'carol'
(.venv) david1471@arch:~/ehr_audit_system/code/audit_client
```

Figure 14: USC trying to fetch alice's, bob's and carol's EHR action history

3.9 Encrypting database records

To see the implementation of how the records are encrypted reading the code is the best way, but the idea was explained in the cryptographic system architecture chapter. The following image shows a record with comments explaining it parts.

Looking at the encrypted patient names it was discovered that their length can leak information about who the patient is. Short names have short ciphertext entries and long names have long ciphertext entries. To fix this issue the ciphertext encrypt a SHA256 hash of the patient names instead of the names themselves. When the Audit server is looking up record it doesn't need to reverse the hash, it would compare the SHA256 hash of the

```
(.venv) david1471@arch:~/ehr_audit_system/code/audit_client
$ ./audit_application.py carol keys/ucla
[
  {
    "time": "2022-05-01 23:10:11",
    "patient": "carol",
    "requested_by": "john",
    "action": "CREATE",
    "ehr_id": "f61587a0518254df69a3626446c76dd5"
  }
]
(.venv) david1471@arch:~/ehr_audit_system/code/audit_client
$ ./audit_application.py david keys/ucla
[]
(.venv) david1471@arch:~/ehr_audit_system/code/audit_client
$ ./audit_application.py eve keys/ucla
[23:28:57.579 ERROR]: Failed to get records, audit_server responded with HTTP returncode 403 and the messa
ge: 'ucla' is not authorized to get EHR actions of 'eve'
(.venv) david1471@arch:~/ehr_audit_system/code/audit_client
```

Figure 15: UCLA trying to fetch carol's, david's and eve's EHR action history

```
$ ./audit_application.py alice keys/alice
[23:33:35.319 ERROR]: Failed to get records, audit_server responded with HTTP returncode 500 and the messa
ge: It was detected that a record has been tampered with, relevant authorities have been notified
```

Figure 16: Tamper detection server side

```
(.venv) david1471@arch:~/ehr_audit_system/code/audit_client
$ ./audit_application.py alice keys/alice
Traceback (most recent call last):
  File "/home/david1471/ehr_audit_system/code/audit_client/./audit_application.py", line 133, in decrypt_record
    record_str = cipher.decrypt_and_verify(ciphertext, tag).decode()
  File "/home/david1471/ehr_audit_system/code/.venv/lib/python3.9/site-packages/Crypto/Cipher/_mode_gcm.py", line
567, in decrypt_and_verify
    self.verify(received_mac_tag)
  File "/home/david1471/ehr_audit_system/code/.venv/lib/python3.9/site-packages/Crypto/Cipher/_mode_gcm.py", line
508, in verify
    raise ValueError("MAC check failed")
ValueError: MAC check failed

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/david1471/ehr_audit_system/code/audit_client/./audit_application.py", line 206, in <module>
    main()
  File "/home/david1471/ehr_audit_system/code/audit_client/./audit_application.py", line 31, in main
    query_usage(args.PATIENT, Path(args.USER_KEY_DIR))
  File "/home/david1471/ehr_audit_system/code/audit_client/./audit_application.py", line 44, in query_usage
    records = decrypt_records(encrypted_records, is_audit_company, user_key_dir)
  File "/home/david1471/ehr_audit_system/code/audit_client/./audit_application.py", line 75, in decrypt_records
    return [
  File "/home/david1471/ehr_audit_system/code/audit_client/./audit_application.py", line 76, in <listcomp>
    decrypt_record(record, is_audit_company, user_key_dir)
  File "/home/david1471/ehr_audit_system/code/audit_client/./audit_application.py", line 136, in decrypt_record
    raise RuntimeError(
RuntimeError: It was detected that record has been tampered with, relevant authorities have been notified
```

Figure 17: Tamper detection client side

```
{
  "patient_enc": {
    "ciphertext": "X/L0qpA=", // AES GCM ciphertext
    "ciphertext_tag": "p/qtUGmJOqPq4CUgulJzmg==", // AES GCM tag
    "ciphertext_nonce": "Qb7BzQkUE5lCMUt4fL1pBQ==", // AES GCM nonce
    "key_enc": "pK7AvuQuy...", // RSA PKCS#1 OAEP encrypted using audit_server's public key
    "key_enc_signature": "DqrSQdMJ..." // RSASSA-RSS signed using audit_server's private key
  },
  "ehr_action_enc": {
    "ciphertext": "aZ9hUy...", // AES GCM ciphertext
    "ciphertext_tag": "wpS1fQc2rWpBWtth7Zo4oQ==", // AES GCM tag
    "ciphertext_nonce": "ycfVP0RNMSq+YjUHOysE1Q==", // AES GCM nonce
    "for_patient": {
      "key_enc": "AkZIo...", // RSA PKCS#1 OAEP encrypted using patient's public key
      "key_enc_signature": "WX6vIQD+..." // RSASSA-RSS signed using audit_server's private key
    },
    "for_audit_company": {
      "key_enc": "igSgwwnN...", // RSA PKCS#1 OAEP encrypted using patient's audit company's public key
      "key_enc_signature": "UdLVz9GwOOZ1..." // RSASSA-RSS signed using audit_server's private key
    }
  }
}
```

Figure 18: The parts of an encrypted record in the audit log database, some values shortened for the purpose of brevity.

patient its looking for with the results when the ciphertext is decrypted.

3.9.1 Notifying Audit Rest API about EHR actions

.

3.9.2 Fetching encrypted EHR actions

For simplicity of this PoC (Proof of Concept) the system will only support 5 hard-coded patients and 2 hard-coded audit companies. The patients in the systems will be:

Alice, Bob, Carol, David, Eve

The first audit company is USC and should be able to audit the EHR records of Alice, Bob and Eve.

The second audit company is UCLA and should be able to audit the EHR records of Carol and David.

Python pajsonckage setup follow recommendations at: https://packaging.python.org/en/latest/tutorials/packaging-projects/

EHR id generation: * Explain the chance of generating two ids that are the same.

Database file: * current: linux/ MAC OS lockin with /tmp/ehr_db.json

EHR REST API: * POST data should always be in application/json

User IDs are just names for simplicity, we assume everyone has a uniqe name.

System architecture visualization: https://structurizr.com/dsl

Rest API specification: https://editor.swagger.io/

RSA PKCS#1 OEAP max message length: * Changed underlying hashing algorithm from SHA1 to SHA256 so max keysize is 190 bytes which is plenty enough (we just need 32 bytes for the 256 bit AES key)

Using different keys for encryption & signing, probably more secure?

PEM key format: *'PEM'*. (*Default*) Text encoding, done according to 'RFC1421'_/'RFC1423'_.

The reason for timestamp is to make sure adversary can't do replay attack

constantly and see how ciphertext grows (they might learn a visit to the doctor happened).

Instead of doing authentication inside the TLS channel, the TLS channel can be configured to use client keys as well, that would have been another option.

Authenticating clients: TLS client cert vs application level authentication * TLS: * Built-in replay attack protection * Needs generation of certificates & ngnix config * Application level * Need to implement replay attack protection somehow (some timestamp shenanigans?) * More complex code

Better programming to do the TLS stuff in Ngnix and connect to Flask through Gunicorn or similar WSGI.

Name of doctor can be anything and is not verified, because it's not the focus of this project.

doctor client examples:

“‘ python3 ehr.py CREATE alice mark -d 'Visit 8' python3 ehr.py DELETE alice john –ehr-id 73a6259260e4237019e1422c7b31adb0 python3 ehr.py CHANGE alice mark –description 'Visit 8' –ehr-id dee291db65e4cf73ca02c152468cc674 python3 ehr.py GET_RECORD alice john –ehr-id dee291db65e4cf73ca02c152468cc674 python3 ehr.py GET_RECORDS bob john “‘

There is authorization check at https://audit_server:5001/log_action, only ehr_server can call it.

```
‘‘‘
curl -X POST --cacert ca/keys/ca.crt --cert audit_client/keys/bob/tls.crt --key audit_cl
bob is not authorized to log EHR actions(.venv) david1471@arch:~/ehr_audit_system/code
curl -X POST --cacert ca/keys/ca.crt --cert audit_client/keys/bob/tls.crt --key audit_cl
bob is not authorized to log EHR actions
(.venv) david1471@arch:~/ehr_audit_system/code
curl -X POST --cacert ca/keys/ca.crt --cert ehr_server/keys/ehr_server_tls.crt --key ehr
<!doctype html>
<html lang=en>
<title>400 Bad Request</title>
<h1>Bad Request</h1>
<p>Did not attempt to load JSON data because the request Content-Type was not &#x27;appl
‘‘‘
```

Username can't be to long, otherwise RSA PKCS#1 OAEP breaks because encrypted message will become to large

System admin commands:

```
```
for user in alice bob carol david eve usc ucla; do cp audit_client/keys/${user}/rsa_encr
```
```

Possible vulnerability: determine patient name length on length of ciphertext

Possible vulnerability: there is no protection against swapping ehr_action_enc in the database, impact not great because of encryption.

4 Introduction

5 System architecture

Describe the system components (e.g., authentication server, audit server, etc.), their functionality, and communication patterns. Clearly describe how your system meets the five goals discussed above.

6 Cryptographic components

discuss appropriate choice of specific cryptographic primitives to ensure the system supports the goals outlined above.

Describe the concrete encryption schemes and key management approaches to be used in your system

7 Limitations of the system

Which challenges were not addressed?

8 Example section

Example citation [2]

8.1 Example subsection 1

8.2 Example subsection 2

References

[1] "Pretty good privacy design."
`https://en.wikipedia.org/wiki/Pretty_Good_Privacy#Design`.
Accessed: 2022-05-01.

[2] C. Neuman, "Challenges in security for cyber-physical systems," in *DHS workshop on future directions in cyber-physical systems security*, pp. 22–24, Citeseer, 2009.