

Introduction:

My initial goal for this project was to create a secure way for users to sign up for services that required personal information, without having to enter it in. In addition to not having to worry about prying eyes, users of this system would have access to a complete history of the information that they've released to organizations. If Comcast, for example, reported a data breach, it is unlikely that I would remember the information that I gave them 5 years ago to sign up with their service. Having my own record gives me the ability to immediately assess my worst-case exposure.

My plan was to create a backend server, a frontend user client, and one or two frontend consumer clients which would request data. During my research though, I found the OWASP Secure Coding Guidelines which outlined what a secure backend server looked like in detail. Due to the nature of the information I am storing, I decided that I should try to incorporate as many of the design points as possible.

Unfortunately, that meant taking time away from the other pieces and I had to scrap the frontend consumer clients. This functionality still exists though, it just doesn't have a face. CURL requests or POSTman can be used to simulate consumer clients to act out the original proposal flow. I had also originally wanted my user client to be in Angular, however, after reading about the Yahoo data breach and how it was caused by Spear Phishing, I began to evaluate how I could mitigate that risk.

My original flow had the consumer client acting within the OAuth flow to obtain access to a user's data. The problem with using OAuth safely is that it relies on the user recognizing that a new browser window has opened to accept their credentials. Most users would not notice if a rouge client redirected to a new page within their domain and presented themselves with the same fields that my user client would present. Again, with time constraints and without a better mitigation, I opted to create an Android app with an embed client key. When a consumer wants access to a user's data, they create an AccessRequest object. Only the user, logged in on the Android app, can view and approve those requests.

Design:

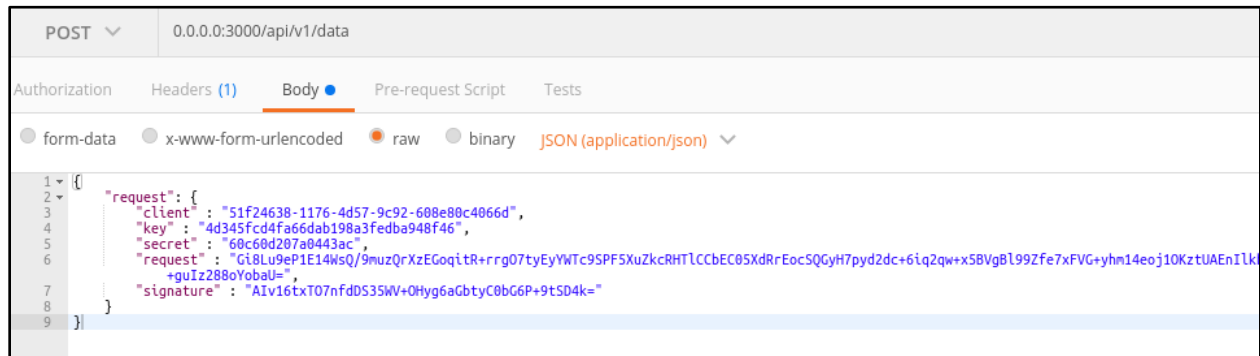
I designed the system to combat several common hacking methods and incorporate as many points from the OWASP Guidelines as possible. My main points of interest were:

- 1) SSL Snooping – Defeated by wrapping the original request and encrypting it at the app level
- 2) Replay Attacks – Defeated by using sequence numbers transmitted with each request
- 3) Sanitized Outputs – Providing as little data as possible while still meeting the client's needs
- 4) Vague Error Responses – Responding with only a status code rather than a specific list of errors for the client to address
- 5) Centralized Authorization Logic – Controllers call this logic rather than implementing their own
- 6) Least to Most Involved Request Checking – Controllers methodically parse requests, trying to identify bad requests as early as possible. First the number of parameters is examined, then the length of the parameters, then the content of the parameters, and finally, the objects that the parameters are referring to are collected and examined
- 7) Securely Generating Keys – Using SecureRandom to generate keys and initialization vectors
- 8) Storing data securely – Datum values are encrypted before being stored in the database

Communication:

One of the most important aspects of the communication between the app and the server is the app level encryption. This prevents bad actors who have inserted themselves in the middle of secure traffic as proxies from gaining any useful information. Any attempts to replay the request would be immediately identified by the server as the sequence number would already be expired. This design makes it easier to determine when the system is being attacked.

Outer Request:

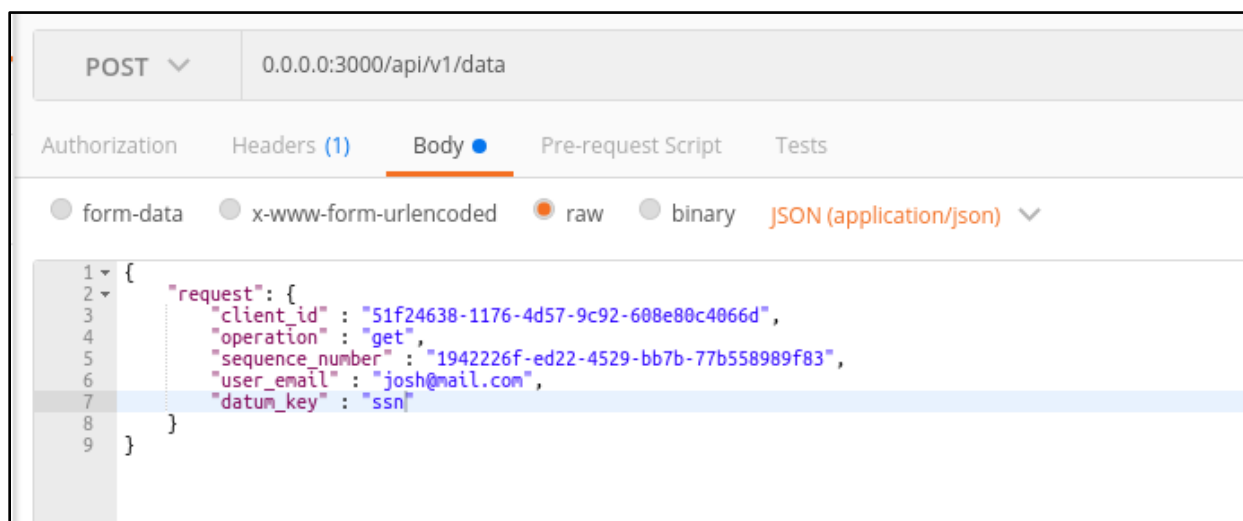


```
POST 0.0.0.0:3000/api/v1/data
Authorization
Headers (1)
Body
Pre-request Script
Tests
form-data x-www-form-urlencoded raw binary JSON (application/json)
1 {
2   "request": {
3     "client": "51f24638-1176-4d57-9c92-608e80c4066d",
4     "key": "4d345fcd4fa66dab198a3fedba948f46",
5     "secret": "60c60d207a0443ac",
6     "request": "Gi8Lu9eP1E14WsQ/9muzQrXzEGoqitR+rrg07tyEYWTc9SPF5XuZkcRHTLCCbEC05XdRrEoc5QGyH7pyd2dc+6iq2qw+x5BVgB199Zfe7xFVG+yhm14eoJ10KztUAEnI1k+guIz288oYobaU=",
7     "signature": "Aiv16txT07nfdD535MW+0Hyg6aGbtYC0bG6P+9tSD4k="
8   }
9 }
```

This is an example of an outer request, what you would see if you were looking at data coming across an unsecured connection. The **client** is the entity sending the request. The **request** is the inner request that has been encrypted with AES256 encryption in CBC mode. The **secret** is the initialization vector used to encrypt the request. The **key** is the key that was used to encrypt the request, but it has been encrypted with the recipient's public key so that only they can decrypt it. The **signature** is a substring of the **key** parameter that the sender has encrypted with their private key.

When the recipient, in this case the server, receives the request, it begins with the **signature**. The server gets the public key associated with the client and validates the signature. Next it uses its own private key to decode the **key**. Finally, the **key** and **secret** are used to decrypt the **request**.

Inner Request:



```
POST 0.0.0.0:3000/api/v1/data
Authorization
Headers (1)
Body
Pre-request Script
Tests
form-data x-www-form-urlencoded raw binary JSON (application/json)
1 {
2   "request": {
3     "client_id": "51f24638-1176-4d57-9c92-608e80c4066d",
4     "operation": "get",
5     "sequence_number": "1942226f-ed22-4529-bb7b-77b558989f83",
6     "user_email": "josh@mail.com",
7     "datum_key": "ssn"
8   }
9 }
```

This is an example of an inner request. The **operation** refers to the REST method. The **sequence number** is the stamp generated by the server for the clients next request. The server will only accept the **sequence number** one time, for a single request. The **user email** is the email of the user that data is being retrieved from. Finally, the **datum key** is the type of information that the consumer wants.

The server takes this request object and validates each of the parameters. In this example, the consumer has asked for the user's social security number. The server will check to see if the consumer has an AccessRequest for that type of data for that user. If the AccessRequest does not exist, the server will respond with a 400. If the AccessRequest has not been approved, the server will respond with a 401. If it has been approved the server will generate an inner response with only the value for the social security number and a new sequence number. It will then encrypt the response and generate the outer response, just like the consumer did.

Endpoints:

There are 6 endpoints provided by the backend, only 4 of which are currently in use. Each endpoint only accepts POST requests with the params shown in the OuterRequest. The endpoints in use are:

sequence_numbers – Used to get the initial sequence number for communication. Will return a 400 if a client has a valid, outstanding sequence number

data – Used to add or get datums for a user

access_requests – Used to add, get, or update consumer requests for user data

alerts – Used to add or get information about consumer data breaches

Current Status:

The project is at the point of a working prototype. Once the backend server is seeded with the client and user information, CURL or POSTman can be used to simulate a frontend consumer's request for data. The consumer obtains an initial SequenceNumber, then they use it to create an AccessRequest for a user's data. The user opens the Android application, enters the IP address of the local server, and pulls all the requests for their account. They can then grant or deny the requests. If a request is granted, the consumer can then get the data that the request was created for.

Because of all the encryption, doing the consumer requests by hand is very painful. To make it easier, the seed file provided starts the server in a state where each consumer has already made requests.

Future Steps:

While the project is a working prototype, it would need several things to advance to a production state. The first is that the backend would need to be hosted with a valid SSL certificate. Next, a frontend consumer example would need to be created to allow better integration testing. Finally, the Android app would need to be made more robust to handle errors and a larger variety of screen sizes.