Security Handin 2

Albert Ross Johannessen

October 22, 2024

Contents

1 Reflect on the Scenario

Let's start by defining the type of data we're dealing with. Since we have access to medical records, we need to take extra care when processing the data as it is **sensitive** personal data[1]. As stated in GDPR, we can process sensitive personal data if we are given explicit consent by the individual, GDPR Article 9[1]. As we do not know the nature of the medical experiment, it is reasonable to assume that it is not a vital interest of the individual and that they are able to give consent, which makes the explicit consent lawful under Aricle 9[1]. While GDPR allows the storage of plaintext data under certain conditions, encryption, especially regarding sensitive data, is recommended. It is up to the data protection officer who is required in hospitals, GDPR Article 37[1] to document and en sure that the data is handled responsibly. Although they will likely suggest that the data should be encrypted, so that if an attacker gains access to it, a key or certificate would be required to access the data.
The *Integrity and Confidentiality* principle only states that:

> Processing must be done in such a way as to ensure appropriate security, integrity, and confidentiality.

Therefore, if the processes around handling the plaintext data are appropriate and can be documented, no regulations are broken.

The hospital's security vulnerabilities suggest that neither data transmission nor storage is sufficiently protected. This suggests that plaintext data should not be processed by the current systems, even if names are removed, as suggested. This is because, even though names might not be directly included, personal data might quickly lead to identifying individuals.

In line with the data minimization principle only necessary data should be collected, and irrelevant details such as names should be excluded.
There are also risks when using federated learning and secure aggregation, although these may be more subtle.

Federated Learning: Despite its benefits, Federated Learning introduces certain risks, it increases the attack surface, as some servers might be more vulnerable than others. This also raises the risk of a poisoning attack, where an adversary who has compromised one of the servers introduces malicious data that corrupts the global model.

Secure Aggregation: Secure aggregation faces similar issues. Many clients must work together to aggregate data, and if one client is lost, the data may be incorrect or even unusable, depending on the fault tolerance of the secure aggregation algorithm.

2 Threat Model

Our implementation is designed to withstand a passive static adversary [2] with an honest majority[3], as the protocol used for secret sharing is not fault tolerant. This means that an attacker learns nothing about other peers if a single peer is compromised. However, if we were confronted with an active adversary, the risks would increase significantly, as a single compromised peer could potentially poison the aggregate as it relies on all parties following protocol.

[1] Medical data is classified as sensitive personal data under GDPR Article 9[1]
[2] An attacker who listens but does not actively manipulate, and chooses which parties to corrupt before the protocol.
[3] An adversary may corrupt a minority of the parties.

3 Why my solution works

In my solution, I use a simple n-party computation, in which no clients or server learns individual values, only an aggregate.
My solution works by splitting each secret value into shares which is defined the following way for a secret s

$$\text{Pick } s_1, \ldots, s_{n-1} \text{ at random}$$
$$s_n = s - \sum_{i=1}^{n-1} s_i$$

This means that s can be computed in the following way.

$$s = \sum_{i=1}^{n} s_i$$

We can now use this to share shares with other parties, I will demonstrate this with three parties; Alice A, Bob B and Mallory M.
Now Alice, Bob and Mallory each creates three shares to be shared among them in accordance to the system above, this will be sent in the following way.

$$A_{out} = a_1 + b_1 + m_1$$
$$B_{out} = a_2 + b_2 + m_2$$
$$M_{out} = a_3 + b_3 + m_3$$

Now each of these smaller aggregates will now be sent to our hospital server.

$$\begin{aligned} out &= A_{out} + B_{out} + M_{out} \\ &= a_1 + b_1 + m_1 + a_2 + b_2 + m_2 + a_3 + b_3 + m_3 \\ &= \sum_{i=1}^{3} a_i + \sum_{i=1}^{3} b_i + \sum_{i=1}^{3} m_i \\ &= a + b + m \end{aligned}$$

This way no party learns about their peers' private values, and the hospital only knows an aggregate, even if a single peer, or more[4], is corrupted the adversary learns nothing about the other peers.

Now we also need some way of guaranteeing the following.

1. Confidentiality - Because we do not want an attacker to read the information sent between peers as they would be able to deduce their secret value.
2. Authentication - Because we want to verify that the message was actually sent by the correct party.
3. Integrity - Because we do not want any adversary to tamper with any messages.

To be more concise, we want to protect against a Dolev-Yao[2] attacker.
Luckily, there is a standard cryptographic protocol which ensures exactly that, TLS[3]. TLS

[4]Remember honest majority.

guarantees confidentiality, authentication, and integrity[5] and is the standard for securing network traffic, making it an obvious choice for our implementation.
Usually, TLS is used in a "one-way" manner because it is often used by webpages, where the client cares about whether the certificate, sent from the server has been signed by a Certificate Authority (CA), but not the other way around. This means that the server has to prove that it is actually the server, but the client does not really have to prove anything, as anyone is able to connect to the server.
In our case, it is a bit different since each client acts as both a client and a server, which means two things:

1. We need to generate two certificates.
2. We need mutual verification.

Here, mutual TLS (mTLS) is the most important part. This means that the server will verify the client certificate, and the client will verify[6] the server certificate[7]. Because the certificates are required to be self-signed by my generated CA certificate, using the X.509 standard[4], an adversary would not be able to present any certificate issued by any other authority, making external calls impossible[8]. Even though TLS is now implemented, there is still one issue that needs handling, which is duplicate messages.

Even though an adversary might not be able to tamper with the messages, there is nothing stopping them from replaying messages. This can quickly be solved by adding idempotency, which can be done in multiple ways, and nonces.
The way I chose to implement it was that we only want one message from each client on every server. While this is not really good design in a production system, it serves our purposes fine in this toy example. For the nonces, I just accepted a Universally Unique Identifier (UUID)[5] per request; if the UUID has been seen before, we do not perform any action.

A Appendix

References

[1] The European Parliament (2016) General Data Protection Regulation.

[2] Danny Dolev & Andrew C. Yao (1983) On the Security of Public Key Protocols

[3] Dierks & Allen (1999) The TLS Protocol Version 1.0 — RFC 2246.[9]

[4] ITU-T (2012) X.509: Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks.

[5] Leach, et al. (2005) A Universally Unique IDentifier (UUID) URN Namespace — RFC 4122.

[5] If configured correctly
[6] By verifying, of course, I mean verifying if they're signed by a CA
[7] The client will also verify the server name, but the server won't verify the client name for obvious reasons.
[8] They're already impossible, as the Docker network does not expose any ports, but we need to assume an open network.
[9] Here I am referring to TLS 1.0 but the newest version specified in RFC 8446, is TLS 1.3