Noise Email

Project Research Document - X00189661 Grzegorz Maniak

Section 1: Detailed Discussion

My project, Noise.Email aims to create a highly secure, privacy-first usable email platform, it will leverage modern cryptographic techniques to ensure client-to-client end-to-end encryption.

The core functionality of Noise.Email is built around *ZKP's (Zero Knowledge Proofs)* which boils down to *'The server never sees the password in plain text. Therefore, the verification process is done without exposing the actual password'*, which is important for this project, since we will derive a secure master key from the user's password.

Below is a breakdown of the registration process *(The most critical part)*.

User Inputs

- **Username:** The user's plain text username.
- **Password:** The user's chosen password.
- **TOS:** A Boolean to confirm the user accepts our terms of service.

Client Generated Secrets

- **Password-Derived Secret:** Using the *Argon2* hashing algorithm, a symmetric key is derived from the user's password. This key is used to encrypt sensitive information, like the root key.
- **Hashed Username:** The user's username is hashed using SHA256 to ensure it is stored securely and cannot be easily guessed.
- **Private Key:** A randomly generated high-entropy asymmetric key is generated, this key will be used to decrypt emails sent to the user.
- **Root Key:** A randomly generated high-entropy symmetric key. This key is crucial for encrypting the user's private key.

Sent to Server

- **Proof:** The hashed username is signed using the private key as proof that the user has correctly generated their private key.
- **Public Key:** The public key corresponding to the user's private key is sent to the server to verify the user's identity and so that others have access to it.
- **Encrypted Private Key:** The user's private key is encrypted using the root key and sent to the server. This allows the server to store the private key safely without having direct access to the unencrypted key.
- **Encrypted Root Key:** The root key is encrypted using the password-derived secret. This allows the user to change their password without needing to rotate or regenerate all their keys. Only the private key must be re-encrypted if the root key changes, simplifying key management.

Section 2: Existing Applications in this Domain

But first, a little explainer, the realm of secure email communication is not new, rather, it just remains under-explored, with existing platforms often falling way short of expectations. I’ve compiled a list of probably the most notable examples out there.

ProtonMail

Link: https://proton.me/mail

ProtonMail is an *‘End-to-End'* encrypted email service that focuses on usability, ProtonMail is regularly criticized as falling short of their encryption claims.

- **Similarities:** Provides encryption for emails; Partially open source;
- **Differences:** Claims of inadequate encryption; lacks a public API; ownership raises concerns; not compatible with other services.

Skiff

Link: https://skiff.com/mail

Skiff was the first mainstream privacy-focused mail service for regular people, it recently got acquired and is no longer accepting new users.

- **Similarities:** Emphasis on secure communication and privacy; Easy to use; Modern and intuitive user interface; Partially open source; Feature rich;
- **Differences:** Currently unavailable to new users; lacks interoperability with other platforms.

Tutanota

Link: https://tuta.com

Tuta is an end-to-end encrypted email service, it has allegedly been back-doored before, and it's losing trust amongst its user base.

- **Similarities:** Emphasis on secure communication and privacy;
- **Differences:** Has an outdated user interface, circa 2000s; not compatible with other services; fully open source.

Mailfence

Link: https://mailfence.com/en/private-email.jsp

Mailfence is an end-to-end encrypted email service, as with Tuta / Tutanota, the client itself is outdated, and looks like a hobbyist project.

- **Similarities:** Emphasis on secure communication and privacy;
- **Differences:** Features are somewhat limited; closed source; user interface feels like a bad Outlook clone; not interoperable with other email services.

Section 3: Platform, Technology and Library's

Platform

I am currently deliberating between hosting it on a managed platform such as AWS *or* AZURE *or* GCP or just running it on a bare metal machine provided by Hetzner. I am leaning towards bare metal.

Benefits and Cons of using Public Cloud Infrastructure

- **Benefits:** Faster time to deployment, I mean, they manage everything for you; Load balancing, e.g. AWS Cloud Front, makes it very easy to scale things up and down;
- **Cons:** Security & Price implications, it's not an unknown fact that the big three are very bad with protecting your data, so why would a platform focused on privacy choose them; The costs can be very high, notably managed RDBs;

Benefits and Cons of using a Bare Metal Machine

- **Benfits:** It's located in the EU which is notably privacy positive (*GDPR*); We can closely monitor our machines; No unexpected charges; We can divide our system resources whichever way we desire; Ultimate control; Wayyyy cheaper;
- **Cons:** You must do everything;

Back-end Technologies

Back-end

The backend API will be written entirely in Go. For routing and HTTP handling, I am using the Gin web framework, it is a minimalistic and high-performance web framework. Authentication will be handled using a custom package I developed called GOWL, which implements OWL aPAKE. All cryptographic functions are provided by Go's internal cryptography libraries.

SMTP (Mail Server)

As for the SMTP server, I am still deliberating between a GO-based or TypeScript-based solution, regardless of the language, I will use a well-tested open-source project as the base for this project. Again, regardless of language, the SMTP server will communicate with our backend API using gRPC, due to its high performance. In the end, my modifications to SMTP are going to be very minimal, adding only two / three commands.

Front-end Technologies

The front end will be written in TypeScript using Svelte. I won’t be using Svelte’s server-side rendering (SSR) / server-side features, I just really like Svelte’s syntax. For client-side crypto, I will use a custom authentication library within GOWL, built around Noble Curves. For UI components, I will rely on ShadCN, which offers a comprehensive set of reusable Svelte components. For styling the site, I will use Tailwind CSS.

Database and Libraries

My goal with this project is to learn something new, be it large or small, therefore I am trying to keep dependencies to a minimum *(Of course to a reasonable extent)*.

Databases

Oracle MySQL

Link: https://www.oracle.com/mysql/what-is-mysql

Our *Primary* Database, nice balance of read and write performance, good feature set.

Memcached

Link: https://github.com/eko/gocache

Our *Cache* Database, It is in memory and it'll be used to cache keys.

Libraries

GOWL

Link: https://github.com/GrzegorzManiak/GOWL

This is the ZKP authentication framework that I will be using, it is written in GO by me, the original paper is linked below.

Nobel Curves

Link: https://www.npmjs.com/package/@noble/curves

This is the ECC library that I will be using on the client side to facilitate all cryptographic functions.

GORM

Link: https://gorm.io

This is the ORM that I will be using, I choose to use an ORM as it can make life a lot easier if you use it correctly.

GIN

Link: https://gin-gonic.com/

Gin will be my routing framework of choice, it is quite simple yet robust, it is also unopiniated which allows for a lot of flexibility.

Frameworks

Svelte

Link: https://svelte.dev/

Svelte is a *relativly* mature front end framework, out of all the frameworks that I have used, Svelte is the only one that I like.

ShadCN

Link: https://www.shadcn-svelte.com/

ShadCN is a fantastic front end kit, it is sleek and modern.

Section 4: The Risks

Critical Library Dependencies

Problem: The project relies on several third-party libraries (e.g., GORM, Noble Curves, Gin). Any security vulnerabilities, bugs, or deprecation in these libraries could significantly impact the platform's functionality, security, and potentially corrupt user data.

Proposed Solution: We could perform regular security audits and implement automated code scanners to flag potential issues and alert us to any relevant CVE's; Locking libraries to specific vetted versions can help prevent supply chain attacks; Performing frequent full backups will ensure that user data is protected.

Key Management Complexity

Problem: The security model introduces key management complexities. In cases of lost passwords or corrupted encrypted keys, recovering data without compromising security could be highly challenging, if not outright impossible; Since user passwords serve as the root for all security, weak passwords would make brute force attacks easier.

Proposed Solution: Whenever a new public / private key pair is created, we will only accept the users keys if they provide *proof* of key ownership by using their private key to sign a piece of data, allowing the server to verify it using their public key. We will also offer constant data-backups and store user data locally; Do not allow the user to choose a weak password, since we never see the password in plain text, we will have to do this verification client-side, we could also prompt users to use a secure password manager.

Operational Overhead and Security

Problem: The complexity of the overall architecture and implementation can lead to extended development times and potential delays in delivering my project.

Proposed Solution: Plan, Plan, Plan, Before writing any substantial amount of code, I will research and plan out each part of the project; Initially only focus on the user side of the application, once that is successfully implemented, we can shift our focus to organizational needs; I have the added benefit that the only technology that I have not worked with so far is GO Lang, I have experience with SMTP, Svelte, Typescript, DNS management and Docker.