Architecture, Cryptography and Gaining User Trust

Kaj Botty, Lee Coppens and Mathias Meeus

March 25, 2026

Contents

1 Introduction

This article was written as a part of the PoC-project requested by KdG Research (Research Centre Sustainable Industries). It describes our technical choices behind our GDPR-compliant platform called Endurance that aims to help veterans get the help they need.

Not every project begins with a business case. This project began with a question: how do you build something for people who, by definition, distrust your application?

Commissioned by KdG Research [4], we are developing Endurance, a platform that connects war veterans (including those from the war in Ukraine) with their families and healthcare providers.

Research among Ukrainian healthcare professionals shows that soldiers and military personnel are widely affected by complex trauma, PTSD, depression, and sleep disorders [5].

Existing systems are too impersonal, therapists lack the data needed to deliver truly personalized care.

Endurance is designed to collect both **quantitative data** (heart rate, blood pressure via wearables) and **qualitative parameters** (journal entries, anxiety levels). And that is precisely where the challenge begins.

In a wartime context, fear of cyber espionage is not paranoia, it is realism.

Recent discoveries of zero-day exploits targeting smartphones, such as the DarkSword exploit chain on iOS 18, show that personal devices are active targets [9].

Attacks on Ukrainian government institutions also demonstrate that hostile actors are continuously searching for vulnerabilities [1].

Without a demonstrably secure foundation, no veteran will use Endurance.

Endurance only needs to appear in the news **once** in a negative way to lose the trust of all veterans and all users. We handle sensitive data and must treat it responsibly.

2 From Simple to Secure

2.1 Our Options

Our first step was to analyze the most common encryption strategies.

Single Global Encryption Key

This option was immediately rejected. The blast radius is unacceptable. If that one key is compromised, the data of all users is exposed.

Key Derived From Password

This initially seemed attractive, since the key is unique per user. However, it has a critical drawback. If a user forgets their password, the derived encryption key becomes unusable. All data is then permanently lost.

2.2 Our Struggle With End to End Encryption (E2EE)

During our research, we explored end-to-end encryption (E2EE) based on the proven Signal Protocol, which is often considered the theoretical gold standard.

With E2EE, the backend acts as a relay that doesn't have access to any personal data. Private keys remain on the user's device.

For a proof of concept, full E2EE proved too complex to implement. The main technical reason for not implementing E2EE was a fundamental architectural conflict.

The project requirements call for server-side processing. Data needs to be processed to generate insights for therapists and the support group around veterans.

E2EE remains a prominent part of our roadmap. For now, we have chosen a secure middle ground.

2.3 Our Solution: Envelope Encryption

We designed a hierarchical envelope encryption architecture, a recognized best practice for securing large volumes of sensitive data [3].

Each veteran is assigned a randomly generated, unique 32-byte key that is independent of their password. That key is itself encrypted using a higher-level master key, which is managed outside the database [8]. If a password is forgotten, it is not a problem. The data remains intact.

3 Why We Avoided External Cloud Key Management

In a commercial setting, we would have quickly opted for a managed cloud KMS, such as AWS KMS or GCP KMS. For this user group, however, we deliberately chose not to.

Digital Sovereignty

Veterans in a conflict zone have little trust in Western tech giants. Handing over ultimate control of encryption keys is not an option in this context.

Supply Chain Attacks

State actors actively exploit vulnerabilities in external dependencies [1]. By minimizing the number of external libraries and using cryptography primitives (from the standard library), we retain full control.

4 Our Implementation

We chose Golang as our implementation language for several reasons.

- Unparalleled performance in API calls
- Extremely fast startup times
- Robust standard crypto library that requires no external dependencies [10].

Below is the core of our encryption service. It has been intentionally kept compact and deliberately avoids black-box libraries.

```
package encryption

import (
    "crypto/rand"
    "fmt"
    "io"
)

func (s *service) GenerateUserEncryptionKey() ([]byte, error) {
    key := make([]byte, 32)

    if _, err := io.ReadFull(rand.Reader, key); err != nil {
        return nil, fmt.Errorf("%w: %v", EncryptionFailed, err)
    }

    return key, nil
}
```

The crucial detail here is that we use the `crypto/rand` package, Go's cryptographically secure pseudo-random number generator. `math/rand` may look similar but is predictable, which is a fatal flaw in a security context.

4.1 Encryption Using AES-256-GCM

```
package encryption

import (
    "crypto/aes"
    "crypto/cipher"
    "crypto/rand"
    "fmt"
    "io"
)

func (s *service) Encrypt(plaintext []byte, key []byte) ([]byte, error) {
    block, err := aes.NewCipher(key)
    if err != nil {
        return nil, fmt.Errorf("%w: %v", EncryptionFailed, err)
    }

    gcm, err := cipher.NewGCM(block)
    if err != nil {
        return nil, fmt.Errorf("%w: %v", EncryptionFailed, err)
    }

    nonce := make([]byte, gcm.NonceSize())
    if _, err := io.ReadFull(rand.Reader, nonce); err != nil {
        return nil, fmt.Errorf("%w: %v", EncryptionFailed, err)
    }

    return gcm.Seal(nonce, nonce, plaintext, nil), nil
}
```

We specifically chose `AES-256-GCM` (Galois / Counter Mode) [6]. GCM adds an invisible cryptographic authentication tag to every ciphertext.

In practice, this means that if an attacker breaches the database and alters even a single bit of a heart rate measurement, the decryption process fails immediately. Silent data corruption is not possible.

4.2 But How Do We Search for Data?

`AES-GCM` with a unique nonce per encryption traditionally makes SQL queries impossible. Every time the same email is encrypted, it looks completely different.

Identifiable fields, such as email addresses, are normalized and hashed using `SHA-256` before storage. This allows efficient SQL searches without ever exposing plaintext data to the client via our API. The hash is a one-way function, it cannot be used to decrypt the original value.

```
package hashing

import (
    "crypto/sha256"
    "encoding/hex"
    "strings"
)

func (s *service) Hash(value string) string {
    normalized := strings.ToLower(strings.TrimSpace(value))
    hash := sha256.Sum256([]byte(normalized))
    return hex.EncodeToString(hash[:])
}
```

```

func (r *repository) ReadFromEmail(email string) UserEntity {
    hashedEmail := r.enc.Hash(email)
    ...
}
```

5 Granular Permission System

Encryption is worthless if users have no control over who sees their data. Our system defines three clearly separated roles, each with its own fixed permissions.

Veteran

Ability to manage their own data, privacy settings, and network. A veteran can also be part of the support network of other veterans.

Therapist

Access insights and dashboards, but only if the veteran explicitly allows it. This applies to professional healthcare providers.

Support

Sees the same as a therapist, but only if the veteran explicitly allows it. This applies to friends or family members, not professional healthcare providers.

The difference between a therapist and a support member is therefore not technical. Both roles can only see what the veteran allows. The distinction lies in who the person is: a recognized healthcare provider or someone from the veteran's personal network.

To enforce this, we implemented an Attribute-Based Access Control (ABAC) system on top of the roles.

The roles themselves do not grant access to personal data. It is the veteran who determines who can see what via ABAC policies.

ABAC enables this dynamic, fine-grained access control by evaluating user attributes in addition to the static role [7]. This allows a veteran to control access at the resource level.

```
{
    "viewerId": "2ae314ec-c630-4d52-914a-117f15f5c84e",
    "resource": "userProfile",
    "effect": "allow"
}
```

Our system works with strongly typed resources, such as `ResourceStressScores` and `ResourceMoodEntries`. We also support wildcards.

```
{
    "viewerId": "908b0873-ffc8-4d4f-b19b-fdca4fa18547",
    "resource": "*",
    "effect": "allow"
}
```

The critical security detail is that we implemented a strict deny-override principle.

An explicit deny always takes precedence over an allow, regardless of the order of policies [2].

6 Practical Use

We focused primarily on sectors such as the police, fire department, ambulance services, and the military, as individuals in these professions are more likely to experience conditions like PTSD, which often go unnoticed for extended periods.

6.1 Fire Department

To gather feedback on our application and its overall vision, we reached out to the Antwerp Fire Department. BZA has an internal support unit, the "Steun Team", which provides both proactive and reactive counseling services to firefighters.

The feedback we received was extremely valuable. The main takeaway was that the system and its design should be as straightforward and simple as possible [11].

6.2 Military

We also consulted a friend working in the Belgian military and received similar feedback. The application should be foolproof, enabling even users with limited technical experience to navigate it easily [12].

7 Our Conclusion

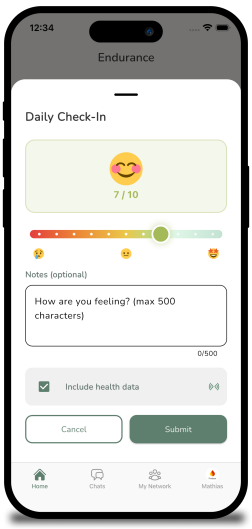
The development of Endurance constantly confronted us with a paradox: how do you build a data-intensive healthcare platform for people who, by definition, distrust any digital solution?

Our solution is a custom envelope encryption architecture in Go, free from black-box cloud solutions, combined with a granular deny-override ABAC permissions system. No reliance on external key managers, no hidden dependencies, full control.

E2EE via the Signal Protocol remains our vision for future versions, as soon as the server-side processing requirements allow for a hybrid implementation.

For now, we have laid a foundation that is secure by design and enables us to protect the privacy of an extremely vulnerable user group.

We learned a ton from this project, it was awesome to build a platform that has the chance to have a **real impact** on so many people.



8 References

[1] BleepingComputer. (2026, March). *Russian APT28 military hackers exploit Zimbra flaw in Ukrainian govt attacks*. Retrieved from https://www.bleepingcomputer.com/news/security/russian-apt28-military-hackers-exploit-zimbra-flaw-in-ukrainian-govt-attacks/

[2] Genesys. (n.d.). *Attribute-based access control overview*. Retrieved from https://help.mypurecloud.com/articles/attribute-based-access-control-overview/

[3] Gunduz, I. (2025, December). *Protecting sensitive data using envelope encryption*. Retrieved from https://dev.to/ibrahimgunduz34/protecting-sensitive-data-using-envelope-encryption-4o3c

[4] KdG Research — Research Centre Sustainable Industries. (n.d.). *Mental health support for war veterans* [Project briefing].

[5] Lawry, L. L., Korona-Bailey, J., Schoenfeld, A. J., et al. (2026). A qualitative preliminary study of Ukrainian healthcare providers' perspectives on service members' mental health since the Russian invasion. *Military Medicine*. https://doi.org/10.1093/milmed/usag092

[6] National Institute of Standards and Technology. (2001). *Advanced Encryption Standard (AES)* (FIPS PUB 197). U.S. Department of Commerce. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[7] Netwrix. (2025, June). *Complete guide to Attribute-Based Access Control (ABAC)*. Retrieved from https://netwrix.com/en/resources/blog/attribute-based-access-control-abac/

[8] OneUptime. (2026, February 17). *How to implement envelope encryption with Cloud KMS in GCP*. Retrieved from https://oneuptime.com/blog/post/2026-02-17-how-to-implement-envelope-encryption-with-cloud-kms-in-gcp/view

[9] SiliconANGLE. (2026, March 18). *Researchers discover zero-day DarkSword exploit chain in iOS 18*. Retrieved from https://siliconangle.com/2026/03/18/researchers-discover-zero-day-darksword-exploit-chain-ios-18/

[10] The Go Authors. (n.d.). *The Go programming language* [Documentation]. Retrieved from https://go.dev/

[11] Bart Botty via BZA Steun Team. (2026, March 22). *UI and UX testing with immediate user feedback.*

[12] Staf Bleuzé via Belgian military. (2026, March 22). *User feedback and consultation.*