Security and Cryptographic Architecture Review

Pushkar Jaltare

Disclaimer

- These views and opinions are my own and do not represent any of my employers or Ente
- This talk is created for beginners. DO NOT USE THIS ARCHITECTURE AS IT IS
- Get a consulting company to review your design and implementation by reputed pentesting vendor

About Me

- Security Architect at Fastly

- Previous experience working for a cloud company, pentesting company

- Crypto enthusiast who has solved CryptoHack, Cryptopals, and has been exposed to real world cryptographic architectures

Cryptographic Background

- This is not a crypto talk

- Some familiarity with cryptographic primitives is useful
  - Encryption, Hashing, Symmetric Key Encryption, Asymmetric Key Encryption
  - Password based Key Derivation Function (KDF)
  - Envelope Encryption

Ente

- I am not associated with Ente in any capacity and do not speak on behalf of Ente

- We can think of Ente as a open source alternative to Google Photos which you can host and maintain

- For a monthly payment, Ente will host your media such as photos and videos for you

Ente Resources

- Architecture details are available online
- Source code is available online
- Security audit was performed by Cure53 and report is available here
- Uses libsodium library for cryptographic operation which is industry standard

Security Properties We Want

- Privacy - Encryption key should be inaccessible to us (Ente)

- Protect Confidentiality as well as Integrity of the uploaded data

- Secure Cryptography : Only use industry standard crypto primitives but supported by wide range of platforms

Security Properties - Continued

- Isolation (Reduce Blast Radius): Utilize unique per customer keys
  - Domain Separation

- Rotation : It should be possible to rotate user password, encryption keys, in case of a compromise

- Performant when encrypting large amounts of data
  - Symmetric vs Asymmetric

Encryption Keys

- Goal : Make encryption keys inaccessible to our operators
  - We shouldn’t store plaintext keys on our servers
- Goal : Unique encryption keys per customer
  - How do we generate and manage potentially 100s of thousands of keys
- We need a cryptographic primitive to solve this challenge

Key Derivation Function

- Construction which allows us to generation encryption key
  - Input can be a master key, password, passphrase, or some other source with enough randomness
- Derive a key encryption key (KEK) from user password
- Use Argon2 to derive a Key Encryption Key (KEK)
  - We don’t need to store this KEK on our server infrastructure
  - We can always derive it when a user is logging in with their credentials
  - Solves our issue of privacy and storage of encryption keys
- We can encrypt anything we want with this KEK

Lab 1: Derive Encryption Key

- Sample Code

- Python sample code to derive encryption key
  - Note that Salt should be unique

- We will review Ente code later

Lab 1: Bonus

- What happens if we change the memory and CPU limits
    - The memory and ops limit changes the derived key

- Makes brute force harder as attacker will have to utilize more memory and CPU

- Find the right balance for your application

Key Encryption Key

- Derived per user from the password

- Can we directly use this key to encrypt customer data?
  - This would make rotation of password incredibly hard

- We need to learn about another construct

Envelope Encryption

- Generate a new data encryption key and encrypt it with Master key

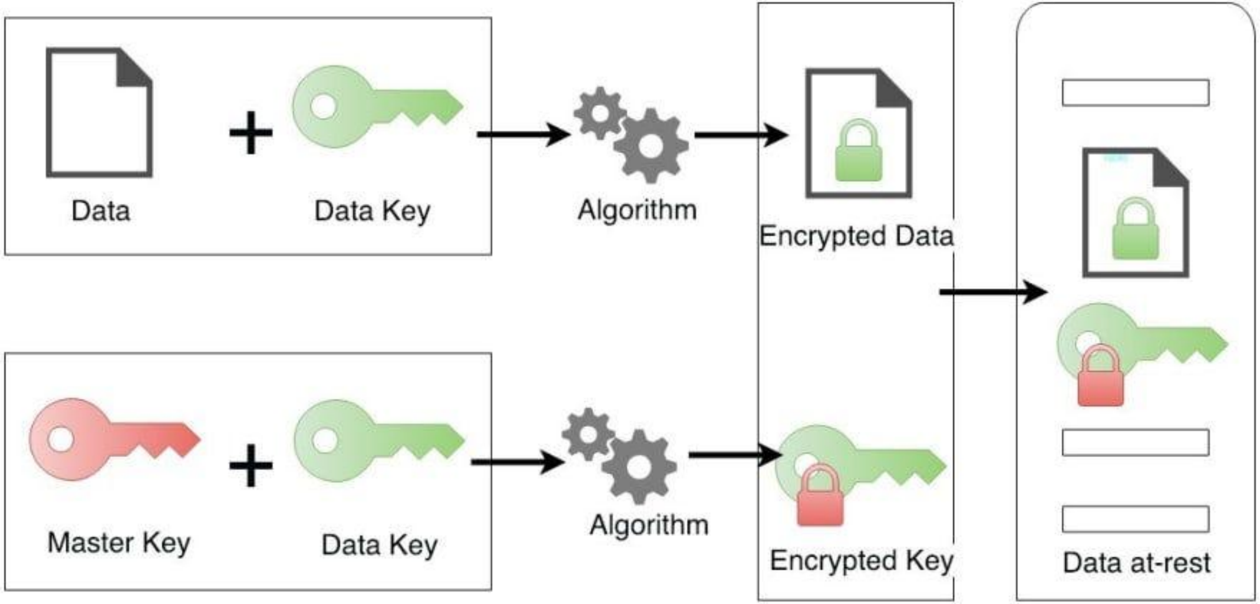


Figure 1: Envelope Encryption

Master Key

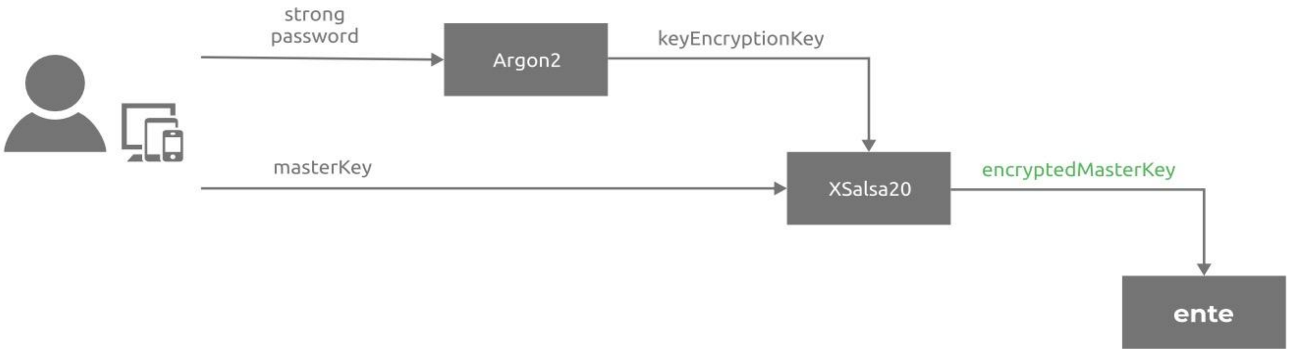
- Securely generate a master key per user during registration

- Encrypt the master key with KEK and store the encrypted master key on the server

- Password rotation only requires re-encryption of master key

Ente encryption key generation flow

- From the architecture design document



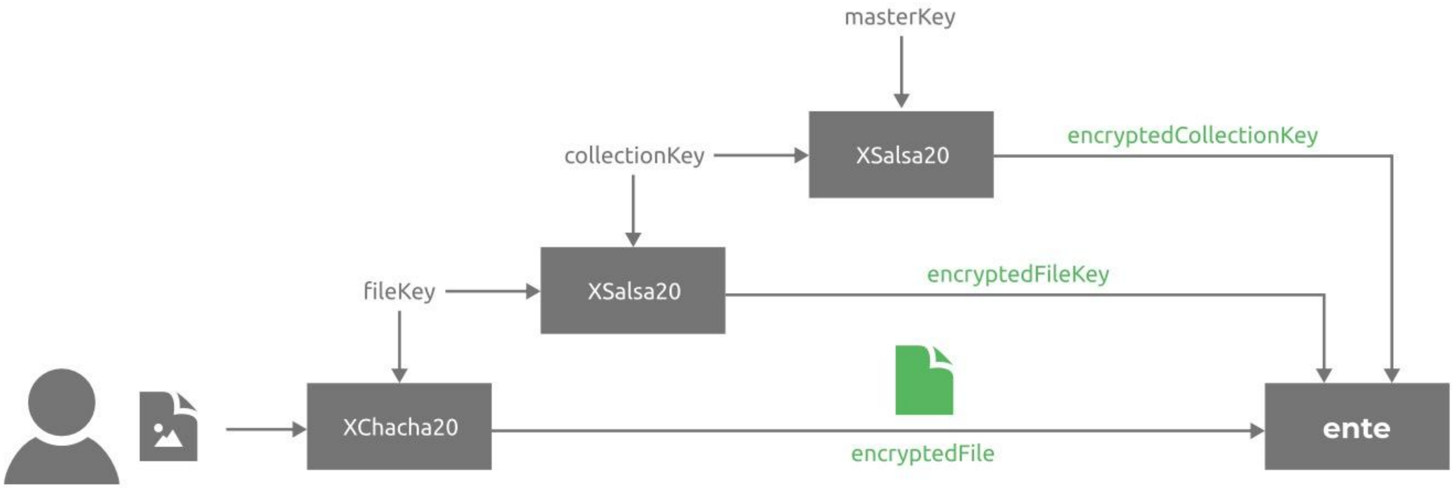
Should we directly use master key?

- Use different keys to encrypt different files
  - Easy to rotate encryption key if compromised and re-encrypt the relevant file
  - Also makes it easier to share a file with someone else
    - Only share the relevant encryption key for the relevant file

- Isolation and key separation can and will prevent you from unforeseen changes to product features, architectural change, and any regulatory requirement

Uploading a file flow

- Flow when uploading a file



Download flow

- Download encrypted file, encrypted collection key, encrypted file key
- Only user knows the password which can be used to derive the KEK
- Since only the user can decrypt master key, only the user can decrypt the subsequent decryption keys

Lab 2

- Sample Python code

- Derive an encryption key and encrypt user file

- Store the ciphertext, nonce, salt together

Lab 2 : Shortcuts

- We are encrypting 1 file

- We are storing the ciphertext,nonce, salt together
    - We are not storing version
    - We are not storing any other metadata such as owner ID
    - Ente header

- There are dragons

Lab 2 : Bonus

- What happens if we enter wrong passphrase at decryption

- Which error do we get
  - Why do we get this error
  - We could authenticate owner ID as one of the fields
  - Check if the same owner ID is requesting the decrypted files
  - DB Cryptography blog post

Lab 2 Bonus : Authenticated Encryption

- Authenticated Encryption provides us confidentiality and integrity

- Always prefer using AEAD modes
  - Unless you know better
  - CBC bitflipping attack, padding oracle attack in Cryptopals

- Hard drive encryption XTS mode

Lab 2 : Privacy

- Filename encryption
    - Length-preserving encryption such as HCTR2

- Metadata
    - Extract metadata and encrypt it
    - Where to store it?
    - Associate metadata with a file

File Sharing

- We need public and private key pair for sharing

- Biggest downside of symmetric key encryption is sharing, you can’t realistically share encryption keys for 100K + other users

- Bootstrapping itself is also an issue

Key Generation

- Generate public key and private key when a user onboards

- Encrypt private key with the master key and store encrypted private key on the servers

- Public key is sent to server will be used to lookup other users

Sender

- Each file and metadata is encrypted with random file key

- File key is encrypted with collection key

- Collection key is encrypted with the public key of the receiver

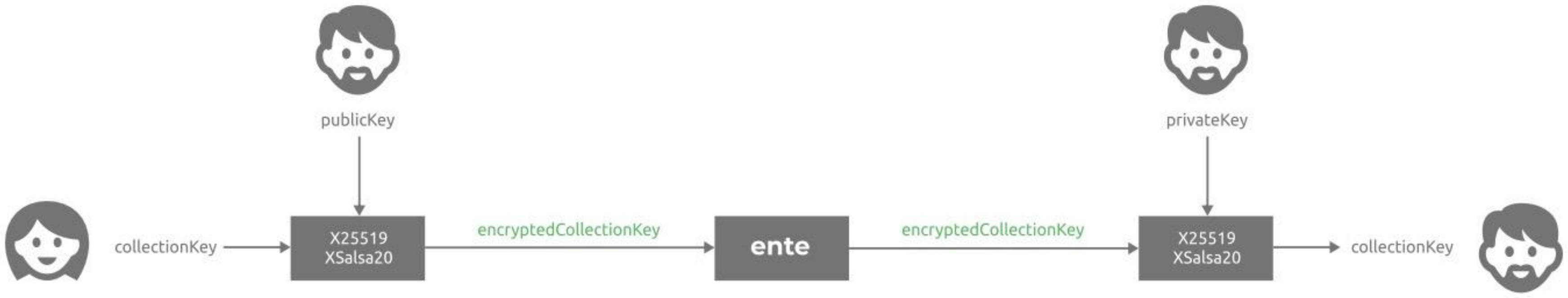
Receiver

- Receiver pulls the encrypted data from server

- Receiving client decrypts collection key with private key

- Decrypt collection key, file key, and the actual file

Flow


publicKey
privateKey
collectionKey
X25519
XSalsa20
encryptedCollectionKey
ente
encryptedCollectionKey
X25519
XSalsa20
collectionKey

Sharing Security

- What if someone malicious changes the public key stored on the server

- Sender can view Verification ID (human readable representation) of the public key of receiver

- This ensures the sender can verify the public key of receiver independently

Verify before sharing

- Verification ID generation
  - Generate a verification ID by converting sha256 value of public key
  - Generate a human readable mnemonic phrase with BIP39
  - Example 12 word phrase
    - glory remain shrug expand feed they notice similar diagram acquire hour razor

Lab 3 : RSA Keys

- Shortcut : We assume we know the public key
  - We need to verify it out of band with BIP39

- Notice we use RSA OAEP mode
  - This is the mode you should use

- We are going to ignore decryption

Recovery

- A well designed application needs a recovery key
- If user forgot their password, all the uploaded data will become useless
- We need a recovery mechanism in consumer facing application

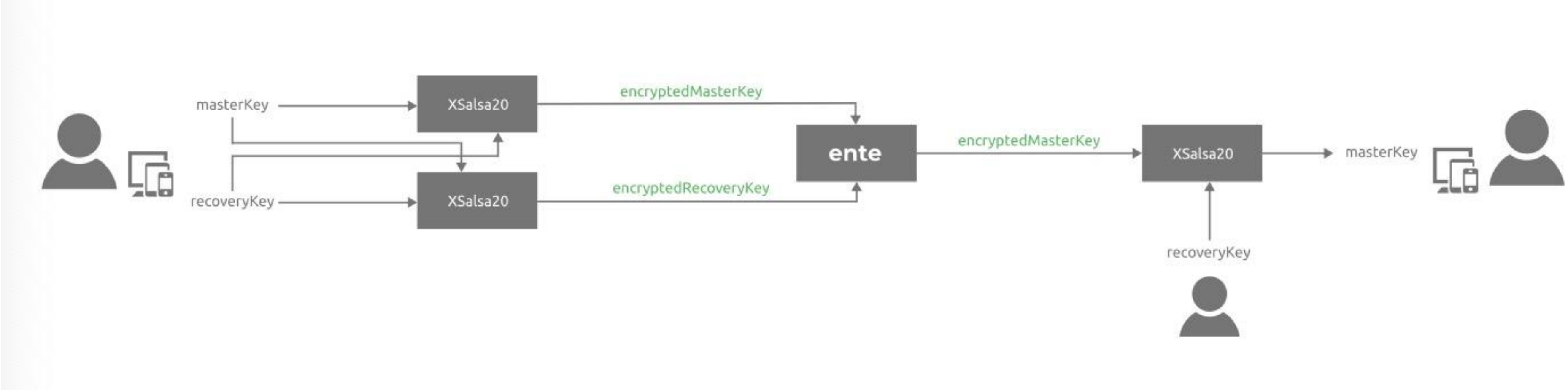
Ente Recovery

- Generate a recovery key during user registration

- Encrypt master key with recovery key as well and encrypt recovery key with master key

- Download the encrypted recovery key each time a new client is used

Recovery Flow



Libsodium

- Modern software library for encryption, decryption, signatures, password hashing, and more

- Cross-platform and cross-language

- Emphasizes security and ease of use

Ente Key Derivation

- Key [Derivation]
- Decryption uses similar [function]

```go
func DeriveArgonKey(password, salt string, memLimit, opsLimit int) ([]byte, error) {
	if memLimit < 1024 || opsLimit < 1 {
		return nil, fmt.Errorf("invalid memory or operation limits")
	}

	// Decode salt from base64
	saltBytes, err := base64.StdEncoding.DecodeString(salt)
	if err != nil {
		return nil, fmt.Errorf("invalid salt: %v", err)
	}

	// Generate key using Argon2id
	// Note: We're assuming a fixed key length of 32 bytes and changing the threads
	key := argon2.IDKey([]byte(password), saltBytes, uint32(opsLimit), uint32(memLimit/1024), 1, 32)

	return key, nil
}
```

Ente Encryption

- [Encryption]()

```go
// EncryptChaCha20poly1305 encrypts the given data using the ChaCha20-Poly1305 algorithm.
// Parameters:
//   - data: The plaintext data as a byte slice.
//   - key: The key for encryption as a byte slice.
//
// Returns:
//   - A byte slice representing the encrypted data.
//   - A byte slice representing the header of the encrypted data.
//   - An error object, which is nil if no error occurs.
func EncryptChaCha20poly1305(data []byte, key []byte) ([]byte, []byte, error) {
	encryptor, header, err := NewEncryptor(key)
	if err != nil {
		return nil, nil, err
	}
	encoded, err := encryptor.Push(data, TagFinal)
	if err != nil {
		return nil, nil, err
	}
	return encoded, header, nil
}
```

Ente Key Pairs

- Generation

```
/**
 * Generate a new public/private keypair for use with public-key encryption
 * functions, and return their base64 string representations.
 *
 * These keys are suitable for being used with the {@link boxSeal} and
 * {@link boxSealOpen} functions.
 */
export const generateKeyPair = async () => {
    await sodium.ready;
    const keyPair = sodium.crypto_box_keypair();
    return {
        publicKey: await toB64(keyPair.publicKey),
        privateKey: await toB64(keyPair.privateKey),
    };
};
```

Bonus Lab

- Trace the decryption code

- Get used to reviewing large codebases
    - Review error handling
    - Review complexity in implementing product features

- When attacking a system start with decryption code as it parses your input
    - Review this post

KDF concern

- User password is weak
  - LastPass!

- 1 Password
  - Generate additional random data and mix with password when deriving the encryption key
  - User needs to keep track of this random data
  - https://blog.1password.com/not-in-a-million-years/

Security Validation

- Security Audit

- Full report worth a read
  - Revocation of shared file is not possible
  - Leaked data encryption key can be a challenge

Possible Concerns

- Trust is hard
    - You have to put some trust in the organization if you are not hosting the code yourself

- End-to-End Encrypted Cloud Storage in the [Wild]
    - Malicious link sharing
    - Malicious file injection
    - Malicious folder injection
    - Malicious filename changes, metadata changes , privacy issues

Other concerns

- Security is hard
    - Security of authentication systems
    - Security of the client applications
    - Security of the server infrastructure
    - Security of the source code
    - And much more…

Other concerns continued

- Loss of data
  - Ente should create backups which increases the attack surface

- Parsing untrusted data such as photos, audio, video
  - Potential RCE attack vectors through these codecs

Further Case Studies

- Other Case Studies
  - 1 Password
  - WhatsApp Encryption Overview
  - AWS KMS Whitepaper
  - Messaging Layer Security MLS Overview
  - End-to-End Encrypted Cloud Storage in the Wild

Questions?

Find me @

- Twitter [pushkar2911]