Klutshnik!!5!

stf

<2023-07-14 Fri>


HELLOWEEN
This is a
DEMONSTRATION
RECORD
If desired, bring to counter
and we will pull a fresh
one from stock.
Keeper Of The Seven Keys
Part I

A detour: Hybrid crypto

> *[..]hybrid cryptosystem is one which combines the convenience of a public-key cryptosystem with the efficiency of a symmetric-key cryptosystem.*[1]

---

[1] `https://en.wikipedia.org/wiki/Hybrid_cryptosystem`

PGP

```
% echo "asdf" | gpg --encrypt --recipient aaaaaaaaaa | pgpdump
Old: Public-Key Encrypted Session Key Packet(tag 1)(524 bytes)
        New version(3)
        Key ID - 0x123456789abcdef0
        Pub alg - RSA Encrypt or Sign(pub 1)
        RSA m^e mod n(4092 bits) - ...
                -> m = sym alg(1 byte) + checksum(2 bytes) + PKCS-1 block type
New: Symmetrically Encrypted and MDC Packet(tag 18)(64 bytes)
        Ver 1
        Encrypted data [sym alg is specified in pub-key encrypted session key]
                (plain text + MDC SHA1(20 bytes))
```

AGE

> *The textual file header wraps the file key for one or more recipients, so that it can be unwrapped by one of the corresponding identities. It starts with a version line, followed by one or more recipient stanzas, and ends with a MAC.*[2]

```
age-encryption.org/v1
-> X25519 XEl0dJ6y3C7KZkgmgWUicg63EyXJiwBJW8PdYJ/cYBE
qRS0AMjdjPvZ/WTO8U2KL4G+PIooA3hy38SvLpvaC1E
--- HK2NmOBN9Dpq0Gw6xMCuhFcQlQLvZ/wQUi/2scLG75s
```

---

[2] `https://github.com/C2SP/C2SP/blob/main/age.md`

Key Management Service (KMS)

Three parties:

- Client
- (remote) Storage
- Key Management Server (KMS)

Traditional KMS encryption

1. KMS has key-encryption-key (KEK)
2. Client chooses data-encryption-key (DEK), $ciphertext = encrypt(DEK, data)$
3. Client Sends (DataID, DEK) to KMS
4. KMS responds to client with $(DataID, wrapped\_dek = Wrap(KEK, DEK))$
5. Client sends $(DataID, wrapped\_dek, ciphertext)$ to Storage

Traditional KMS decryption

1. Client gets $(DataID, wrapped\_dek, ciphertext)$ from Storage
2. Client sends $(DataID, wrapped\_dek)$ to KMS
3. KMS unwraps the key $DEK = Unwrap(KEK, wrapped\_dek)$
4. KMS returns (DataID, DEK)
5. Client decrypts ciphertext: $data = decrypt(DEK, ciphertext)$

Traditional KMS sucks

- KMS knows all DEKs
- DEK depends on security of transport between KMS/Client (TLS)
- Middlebox and endpoints of TLS also see DEK
- KMS can trace usage of DataIDs
- Updating KEK is costly - increases time-to-delete old KEK

Oblivious KMS

Use an OPRF!!5!

$$DEK = OPRF(k_c, DataID)$$

BOOM! Mind blown.

OKMS rulez!

- ~~KMS knows all DEKs~~
- ~~DEK depends on security of transport between KMS/Client (TLS)~~
- ~~Middlebox and endpoints of TLS also see DEK~~
- ~~KMS can trace usage of DataIDs~~
- Updating KEK is costly - increases time-to-delete old KEK

Updateable OKMS

Use an updateable OPRF!!5!*

- ~~KMS knows all DEKs~~
- ~~DEK depends on security of transport between KMS/Client (TLS)~~
- ~~Middlebox and endpoints of TLS also see DEK~~
- ~~KMS can trace usage of DataIDs~~
- ~~Updating KEK is costly - increases time-to-delete old KEK~~

* must have at least $2t + 1$ shareholders!

Threshold UOKMS

Use an updateable threshold OPRF!!5!
BOOM: no more SPoF!

UOKMS[3]

"Updatable Oblivious Key Management for Storage Systems"
– Stanislaw Jarecki, Hugo Krawczyk, and Jason Resch

---

[3] `https://eprint.iacr.org/2019/1275`

Advantages over KMS

- hides keys and object identifiers from the KMS,
- offers unconditional security for key transport,
- provides key verifiability,
- reduces storage,
- distributed threshold implementation that enhances protection against server compromise & denial.
- updatable encryption capability that supports key updates (aka key rotation),
- very efficient non-interactive update procedure,
- forward and post-compromise security
- public key encryption

Comparison with legacy tools

Pro:

- cheap key-rotation
- forward & post-compromise security,
- threshold operation (t peers need to agree to decrypt)
- better resilience against denial/loss of keys.
- nice for traveling into hostile countries
- KEKs not stored next to ciphertexts

Contra:

- Online
- Strong authentication needed

are these really disadvantages though?

Anyway

threshold construction

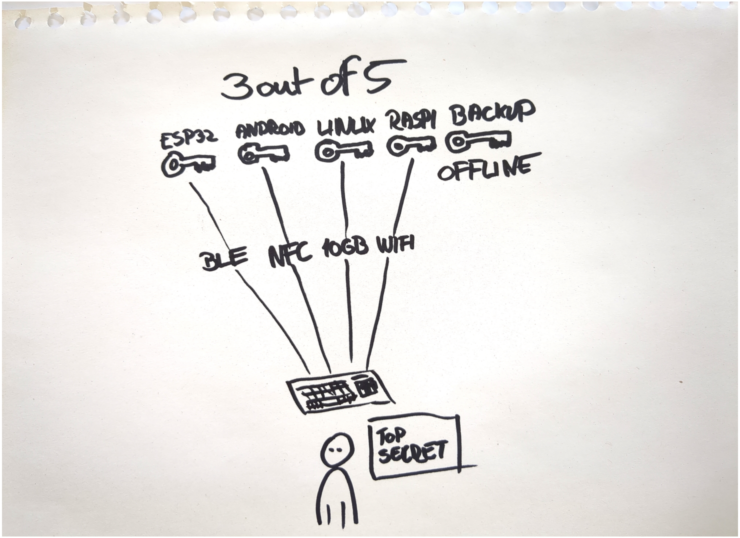
$$+$$

updateable encryption

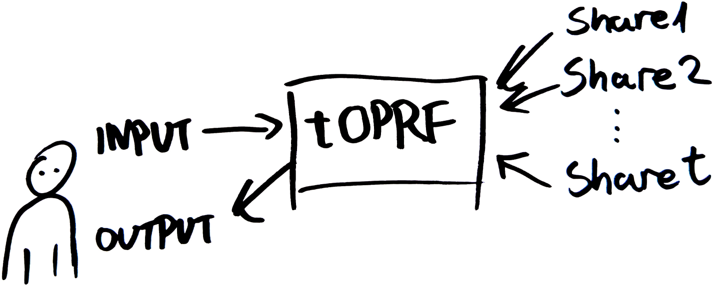
are some really sexy properties, gotta have 'em!
thus...

Klutshnik

Klutshnik Overview



The heart of Klutshnik - tOPRF[4]



the shared secret KMS KEK is **never** reconstructed!

---

[4] TOPPSS: Cost-minimal Password-Protected Secret Sharing based on Threshold OPRF
`https://eprint.iacr.org/2017/363`

Distributed Key Generation (DKG)[5]

The "shareholders" participate in a protocol in which they create a shared secret split among them just like Shamirs Secret Sharing, but do so in a way, that there is no central - trusted - third-party and the shared secret itself is never reconstructed.

---

[5] Secure Distributed Key Generation for Discrete-Log Based Cryptosystems
`https://link.springer.com/content/pdf/10.1007/s00145-006-0347-3.pdf`

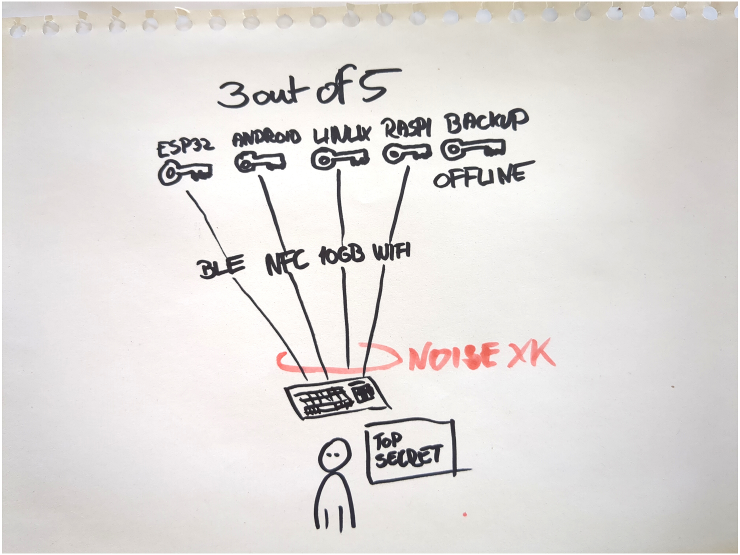
DKG Caveat

- Rabin et al DKG requires **partially synchronous** medium
- Kate&Goldberg DKG[6] - does not but is a "mess"
- In Klutshnik the client is the center of a star topology where messages between shareholders ~~are~~ will be e2e encrypted.
- Cannot change threshold without running a new DKG, but can anytime increase N.

---

[6] `https://crysp.uwaterloo.ca/software/DKG/`

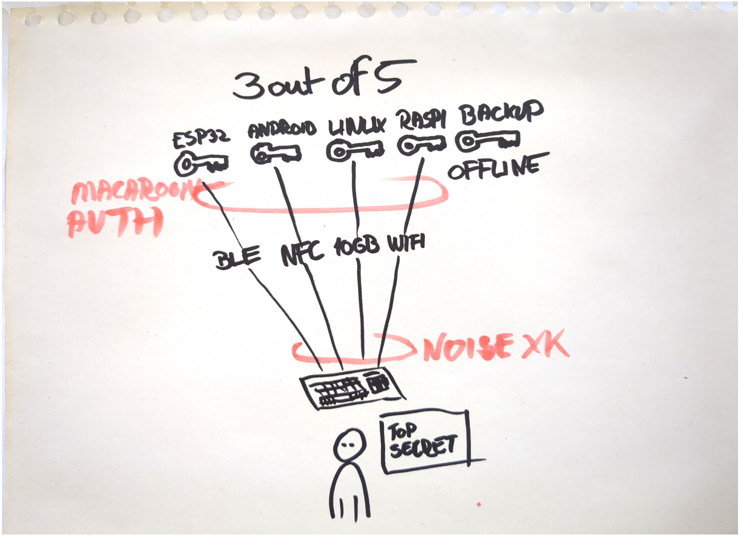
Protected channels[7]



---

[7] https://github.com/Inria-Prosecco/noise-star/

Strong Authorization



Physical control, *authorized_keys* and macaroons

Macaroon[8] authorization tokens

Macaroons are authentication tokens (cookies but better) that

- can be delegated
- can be attenuated
- carry their own proof
- can be extended with 3rd party caveats
- are simple to verify
- decouple authorization logic

---

[8] `https://research.google/pubs/pub41892/`

Macaroons in Klutshnik

- new macaroon minted on DKG and bound to "keyid"
- default TTL 1 year

can be attenuated:

- to a client pubkey
- shorter expiration date
- action: update or decrypt

comes with handy CLI tool to operate on (klutshnik specific) macaroons.

File encryption

Files in Klutshnik are encrypted with the DEK using the STREAM construction from the paper: "Online Authenticated-Encryption and its Nonce-Reuse Misuse-Resistance"[9] using XSalsa20/Poly1305 from libsodium.

- **online** authenticated stream cipher
- no reordering of "segments"
- no truncation
- no release of ciphertext if mac is invalid
- etc

---

[9] `https://eprint.iacr.org/2015/189`

Key-updates

For each client KMS has secret key $\mathrm{k_c}$, and storage has public key $y_c = g^{k_c}$

- Encryption:
  1. $\mathrm{r} = \mathrm{random}()$, $\omega = \mathrm{g^r}$, $dek = H(y_c^r)$, and,
  2. store $(ObjId, \omega, ciphertext = Enc_{dek}(Obj))$
- Decryption:
  1. retrieve (ObjId, ω, ciphertext),
  2. $dek = OPRF(k_c, \omega)$,
  3. return plaintext = $\mathrm{Dec_{dek}}$(ciphertext)
- Rotation and Update:
  1. KMS generates k', and sends $\Delta = \frac{k'_c}{k_c}, y'_c = g^{k'_c}$ to storage
  2. Storage replaces y_c with y'_c, and each (ObjId, ω, ct), with (Obj, ω^{Δ}, ct)

tOPRF

- Key k is Shamir secret shared among n shareholders with threshold t: shareholders S_i holds share k_i, where i<=n.
- It is very simple, just do a Lagrange interpolation in the exponent.
- λ_i is the Lagrange coefficient for index i:
- $\prod_{j=1,ES_j\neq i}^{t} \dfrac{ES_j}{i - ES_j}$
- where ES is vector holding the indexes of the shareholders participating in this protocol run.

tOPRF - two variants

1. Client blinds input x: $r = \mathit{random}(), \alpha = H'(x)^r$, sends α to t shareholders

If you know before step 2. the indexes of all t shareholders participating in this run:

2. shareholder $\mathrm{S_i}$ responds with $\beta_i = \alpha^{\lambda_i \cdot k_i}$,
3. Client: $f_k(x) = H(x, (\prod_{j=1}^{t} \beta_i)^{1/r})$

If you don't know the set shareholders before step 2:

2. shareholder $\mathrm{S_i}$ responds with $\beta_i = \alpha^{k_i}$,
3. Client: $f_k(x) = H(x, (\prod_{j=1}^{t} \beta_i^{\lambda}{}_i)^{1/r})$

Threshold updateable OKMS

1. DKG a new shared value p, each shareholder holds p_i
2. Multiparty computation of k * p
3. send p_i to storage, which can reconstruct p, which is equal to Δ in the non-threshold version.

Note: needs 2t+1 shareholders for the multiplication!
Note2: all shares must participate in an update, no post-factum update possible without violating security guarantees. Also your backup shares. . .

Klutshnik ops client inputs

- Encryption: the storage only needs the pubkey of the key it encrypts to.
- Decryption: the client needs a Noise keypair that is permitted by the KMS and an authorization macaroon for the keyid necessary for the decryption.
- Key update: same as decryption.

It is possible to store the noise keypair and the macaroon in a config file. But why, when we have an tOPRF at our disposal?

tOPAQUE for storage of arbitrary blobs

- OPAQUE can store arbitrary e2e encrypted blobs on a server (that's also what Whatsapp is doing for backups!) and all the user needs is a password - preferably from SPHINX ;)
- Using our existing KMS infra, which does already tOPRF and also DKG it is trivial to build a **threshold OPAQUE** server that stores encrypted data (keys & macaroons) unlocked by a password.
- Check out opaque-store[10] if you want this, it works with klutshnik KMS servers. Perfect companion to SPHINX.

---

[10] `https://github.com/stef/opaque-store`

Use-cases

- encrypted archives
- passing border controls without being able to decrypt
- temporal sharing of files
- better security if you don't have a hw token for your KEK
- if you need a committee to decrypt

Future

- raspi image
- port to ZephyrOS (bluetooth support)
- FUSE frontend
- threshold signature support
- threshold sphinx support
- generic OPRF/DKG server.

Poke at it

- try it out with docker `https://github.com/v-p-b/klutshnik/tree/docker/docker`
- stare at code: `https://github.com/stef/klutshnik`

?

Questions? Comments!