1. This lecture is about key management, covering different approaches for managing secret keys.
Learning Objectives

1. First, we will take a high level look at technologies to protect confidentiality.
2. Shamir Secret Sharing (SSS) is a technique to protect both confidentiality and availability.
3. But SSS has issues in practice, so we will introduce GNU Anastasis as a more practical solution.
4. Then, we will look at threshold signatures as a way to protect highly valuable signing keys.
5. Finally, we’ll look at the different types of keys in GNU Taler as a practical example for how complex applications require various types of keys with different protection strategies.
PKCS#12 is the most common format for software PSEs:
- PKCS#12 is a file container format used for storage and transport of private keys (and possibly certificates).
- Information is protected with a password-based symmetric key (e.g. a password).
- The security of a software PKCS#12 is based on the strength of the password protecting it.

Problem: A PKCS#12 soft-token may be copied unnoticed.

1. Naturally, there are other formats for storing private keys.
2. Password protection is of course already a good idea, but limited as sometimes the password will need to be entered to decrypt the key, and then a compromised host could result in the private key being leaked.
3. Furthermore, while the password should be strong, an adversary getting hold of the PKCS#12 data might still be able to brute-force it in practice. Not everybody uses 58-character passwords.
How can we protect confidentiality of private keys?

1. A general purpose PC or smart phone can run a huge range of applications and has a complex operating system and thus a huge attack surface. The use of dedicated hardware that only runs the cryptographic functions plus some access control logic is thus safer.
Properties of Crypto-tokens/cards

- Crypto-cards have the ability of a secure container for secret data and have an executive platform for cryptographic algorithms.
- A Crypto-card looks like a “Black Box” from the outside, where some operations can only be used over a very restrictive hard- and software interface which is able to enforce specific security policies.
- Access to sensitive data areas (i.e. private keys) is physically “impossible” from the outside.

How can we protect confidentiality of private keys?

1. Depending on the hardware, it is also possible to make it hard to extract the private key. This is of course a double-edged sword, as if the key is generated on the hardware, this would also prevent key backups.
2. Finally, cheaper hardware may also lack good entropy generators, requiring the keys to be provisioned, for example by the manufacturer, creating new risks.
Example: Yubikey and Personal Identity Verification (PIV)

- Yubikey provides Smart Card functionality based on the Personal Identity Verification (PIV) interface specified in NIST SP 800-73.
- Yubikeys perform RSA or ECC sign/decrypt operations using a private key stored on the token, through common interfaces such as PKCS#11.
- Supported key sizes: RSA 2048 or ECC 256/384.
- The “universal smartcard minidriver” provides “standard smart” functionality and additional certificate and PIN management features.
- Special Yubikeys obtained FIPS 140-2 security level certification.

How can we protect confidentiality of private keys?

1. A key issue with such devices is that they usually only support a limited set of cryptographic primitives.
2. The speed of operations is also generally only useful for personal use, like when signing an e-mail or authorizing a login, and not for APIs that need to automatically sign thousands of messages per second.
3. Finally, over 20 CVEs against Yubikey, including some with high or even critical severity (CVE-2015-3298, CVE-2021-43399, CVE-2011-4120) show that certification does not warrant the absence of critical security flaws, and merely shows that the manufacturer followed standard procedures.
Hardware Security Modules (HSM)

Common functionality:
- Secure storing and use of keys
- Random number generator
- Key pair generation
- Digital signing
- Key archiving
- Acceleration for crypto schemes

Should protect keys against:
- Mechanical & chemical attacks
- Temperature attacks
- Manipulation of voltage

How can we protect confidentiality of private keys?

1. For server-side use where hundreds or thousands of public key operations are required per second, you can buy expensive high-end HSMs.
2. These HSMs not only offer higher performance, but also improved lifecycle management (entropy generation, key generation, key rotation and archival) and security.
3. Specifically, such high-end HSMs for servers are expected to have more solid protections against attacks. While consumer-grade devices can often leak private keys when exposed to physical attacks, these HSMs are expected to detect physical attacks and destroy private keys before they get leaked.
4. Plus, they are more unwieldy and thus harder to steal from your data center with cameras watching the racks.
How can we protect confidentiality of private keys?

Availability

1. HSMs may be helpful to protect the confidentiality of keys. However, most are proprietary, which means your control over the keys is worse than with FLOSS on COTS hardware.
2. Confidentiality is most important for signing keys; signing keys are easily replaced when the HSM breaks.
3. But for decryption keys, availability is at least equally critical.
4. Even a high-end HSM doesn’t ensure availability, as the OVH data center fire showed.
Problem 1: Availability

If you give one person (or data center) a secret, it may get lost.

⇒ So give it to more than one person (or data center)!

1. The usual answer to this is to have multiple backups.
2. This way, if one copy is lost, another is still available.
Problem 1: Availability

If you give one person (or data center) a secret, it may get lost.

⇒ So give it to more than one person (or data center)!
Problem 2: Confidentiality

If you give many entities a secret, it may get disclosed.

How does Shamir Secret Sharing work?

1. Now, we still want to keep the key a secret, and giving copies out to many entities to improve availability makes confidentiality harder.
2. A simple solution here is to not give everyone a working key, but a share of the key.
3. The idea being that then some subset of the shares, often $k$ out of $n$, are enough to reconstruct the secret.
Problem 2: Confidentiality

If you give many entities a secret, it may get disclosed.

⇒ So give them only a key share!
Problem 3: Scalability

If you want $k$ out of $n$ entities to coordinate to recover a secret, there are

\[ \binom{n}{k} = \frac{n!}{k!(n-k)!} \]  

(1)

combinations to consider.

How does Shamir Secret Sharing work?

1. The number of subsets of size $k$ of a set of size $n$ can be large.
2. Polynomial interpolation gives us a way to do this efficiently.
Problem 3: Scalability

If you want $k$ out of $n$ entities to coordinate to recover a secret, there are

\[
\binom{n}{k} = \frac{n!}{k!(n-k)!}
\]  

(1)

combinations to consider.

⇒ Use polynomials!

1. The number of subsets of size $k$ of a set of size $n$ can be large.
2. Polynomial interpolation gives us a way to do this efficiently.
A polynomial of degree $k - 1$ is fully determined by $k$ data points

$$(x_0, y_0), \ldots, (x_j, y_j), \ldots, (x_{k-1}, y_{k-1}),$$

where no two $x_j$ may be identical.

---

How does Shamir Secret Sharing work?

1. The idea is that $L(0)$ is the secret. The other coefficients of the polynomial of degree $k - 1$ are chosen at random.
2. We then give the $n$ entities $y_i = L(i)$ for a polynomial of degree $k - 1$.
3. This way, everyone has to store only $(i, y_i)$ and $k$ entities can reconstruct the polynomial and then compute $L(0)$. 
The interpolation polynomial in the Lagrange form is:

\[ L(x) := \sum_{j=0}^{k} y_j \ell_j(x) \]

where

\[ \ell_j(x) := \prod_{0 \leq m \leq k, m \neq j} \frac{x - x_m}{x_j - x_m} = \frac{(x - x_0)}{(x_j - x_0)} \cdot \frac{(x - x_{j-1})}{(x_j - x_{j-1})} \cdot \frac{(x - x_{j+1})}{(x_j - x_{j+1})} \cdots \frac{(x - x_k)}{(x_j - x_k)} \]  

for \( 0 \leq j \leq k \).

1. This fun equation tells us how to recompute the coefficients of \( L \) given \( k \) points.
2. \( \ell_j \) can be pre-computed from just knowing the subset \( k \) and without knowing the secret shares \( y_i \).
Our secrets will typically be integers. Calculations with floating points are messy. ⇒ Use finite field arithmetic, not \( \mathbb{R} \).

1. If we did the calculation in \( \mathbb{R} \), rounding errors will in most cases make the result non-sensical.
2. So instead, we need to do basically the same math over some finite field \( \mathbb{GF}(q) \).
3. Surprisingly, this is not actually useful in practice for key management.
How does Shamir Secret Sharing work?

Real world scalability

1. Looking at practical values for $k$ and $n$, we first of all find that $\binom{n}{k}$ is usually not very big.
2. $n = 20$ and $k = 10$ is already pretty big for distributing key shares.
3. $n = 30$ and $k = 15$ is where we may start to see a need for scalability, but even that is still doable on modern computers.

Other values:

- $\binom{10}{5} = 252$
- $\binom{20}{10} = 184756$
- $\binom{30}{15} = 155117520$
Do we have a scalability problem?

How many people do you have to share your secrets with?

How many people realistically participate in recovery?

1. The real problem we have is usability, not scalability.
2. Here, \( k \) out of \( n \) is not flexible enough in practice. If Alice is the boss, we may want Alice or Bob and Alice or Dave, but not Bob and Dave without Alice to be able to reconstruct the key!
3. Usability includes how we authenticate during recovery. Here also, it may make sense to have a policy like password plus SMS-TAN or password plus Email-TAN, but not SMS-TAN plus E-mail TAN as both of those factors might just need control over the same smartphone!
4. Finally, we want to apply data minimization also to the key escrow services that hold the shares to minimize risk.
What is GNU Anastasis? [2]

- Distributed key escrow and recovery service
- Users split their secret keys and distribute shares across multiple service providers
- Only the authorized user can recover the key by following standard authentication procedures
- Service providers learn nothing about the user, except possibly some details about how to authenticate the user

1. GNU Anastasis addresses these challenges, allowing flexible recovery policies (not just $k$-out-of-$n$) to be freely configured.
2. Moreover, the system focuses on allowing different methods to authorize key recovery, and considers the economic aspect of operating such a service by integrating payment.
3. Finally, it minimizes data exposed to the providers and addresses the issue of users forgetting their credentials.
Overview

1. At a high level, the user shares their secret shares with the various providers together with information on the authentication method chosen for each share.
2. During recovery, the user authenticates to a subset of the providers to recover the shares.
3. Given enough shares, they can then reconstruct the secret.
4. Next, we’ll look at a much simplified version of the protocol in a bit more detail.
Step 1: Enter secret information

1. The cryptographic process starts with the master secret being provided to the application.
2. In practice, this is actually almost the last step in the user-story, but it is the first step for the cryptography.
Step 2: Split information

1. First, the master secret is “split” into various shares. The master secret is encrypted with multiple random symmetric keys, each of which is derived from the respective set of nonces used as key shares.
2. Regardless, we can view this as splitting up the original secret into various pieces that allow recovery.
1. This is a highly unusual step. The user is first asked to provide various highly personal details about themselves. The design calls for attributes to be provided that ideally never change, cannot be forgotten, and at the same time are ideally not well-known.

2. The user can actually provide false data here, but what is critical is that they remember the data that they provide! So in practice, it is recommended to be perfectly honest, and to provide as much data as possible.

3. All provided data is then run through an expensive cryptographic hash function to create the user ID, a unique identifier for the user.

4. Here, we thus have a privacy paradox: the more private data is provided, the less predictable or invertible the user ID will be, and thus the more privacy and confidentiality the system will achieve.
Step 4: Key derivation

1. From the user ID, two keys are derived using different HKDF functions.
Step 5: Encrypt parts

1. The first key is then used to encrypt the key shares.
2. This ensures that the providers do not even learn the key shares that they are storing, except if they somehow guess or know their user’s identity with all of the attributes and are thus able to mount a confirmation attack to invert the hash function.
1. Anastasis calls the authentication data of the user the **truth**. This can be the (salted hash of the) answer to a security question, an image of the user for video identification, their phone number or e-mail address to send a TAN, or basically any other data that would be required to perform some authorization check.

2. Each key share is combined with the respective truth that defines the authorization check that the user will need to pass to recover the encrypted key share.
1. The truth (and the associated encrypted key share) are then encrypted with the second key that was derived from the user’s identity.
2. This ensures that the providers do not learn the truth during backup.
1. The last step of the backup process is to then upload these encrypted truth and double-encrypted key shares to the Anastasis providers.
2. The providers store this information under provider-specific unlinkable identifiers. Thus, they only learn the time and the size of the data (and possibly the IP address of the client, which they should immediately discard).
3. Note that this is actually only part of the story, as independently more encrypted information is stored to enable recovery, but for the simplified protocol description we will leave it at this.
1. To recover their secret, a user first must again enter all of their personal information. It is again hashed to compute the unique user ID.
Step 10: Key derivation

1. The client can then again derive the same two symmetric encryption keys.
Step 11: Provide key

1. The encryption keys are sent to the respective providers.
2. Note that in practice, each provider will be given a different key, but this is again a detail.
1. The providers can then decrypt the truth and challenge the user to pass the authorization check.
2. This can be quite different depending on the check, but we can imagine providers sending TAN codes to the e-mail address or mobile phone number they decrypted.
1. Next, the user must pass the authorization challenge, for example by sending back the TAN.
1. In response, the client is given the encrypted key shares for the challenges that they have passed.
Step 15: Decrypt parts

1. The client can then decrypt the key shares using the first key derived from the user ID.
Step 16: Reassembly

1. Given enough shares, the original master secret can be recovered.
1. Policy documents are created and stored by providers under another hash derived from the user ID to remember which combinations of secret shares at which providers would allow recovery of the master secret.

2. Per-provider salts are used to ensure that key material differs between providers.

3. Request limits and payments are used to protect Anastasis providers against attacks, and key shares against brute-force attacks.

4. A core secret is used for the actual “larger” data the user wants to back up, while the master secret is a symmetric key used to encrypt the core secret. Versioning support ensures that the same user can upload and recover more than one core secret.

5. Providers can also set liability limits, so that users that pay for the service know the amount of legal liability the providers assume for keeping the secret shares.
Alice wants to create a cryptographic signature, but:

- No single piece of hardware is trusted
- No single service provider is trusted

But: Using $t$ independent signature service providers might be ok!

If we need $t$ providers, we probably should initially sign up with $n$ providers so that we can still create signatures if only $t/n$ are available...

1. GNU Anastasis still requires the user to have a secure client as the core secret originates from the user’s computer and is fully restored on a computer presumed to be under the user’s control. For applications involving encryption, this is probably the best practical solution today.

2. However, in practice, end-user devices may be compromised. For encryption keys, there is little we can do as the user will need to access their data on their system in the end.

3. But, for signatures, we can actually achieve security even if we cannot trust any individual piece of client hardware and also no single provider!

4. The solution here is to use threshold signatures, where $t/n$ providers are required to contribute to create a signature.
Flexible Round-Optimized Schnorr Threshold (FROST) is a $t$-out-of-$n$ threshold signature scheme:

- Distributed key generation protocol can be used to ensure private key is never stored on a single device
- $t$ providers required to collaborate to create digital signature

1. FROST is a modern threshold signature protocol that generates Schnorr signatures.
2. If we do not want to trust the client or any individual provider, distributed key generation must be used to collaboratively create private key shares and a master public key without ever having the private key available on any system involved.
3. We can create $n$ private key shares, and then use $t \leq n$ key shares to create a signature that verifies against the master public key.
4. FROST only defines the cryptographic primitives, but is not a usable system by itself.
FROSIX

Free Software implementation for threshold signatures using FROST with:
- RESTful API to interact between signer and signing services
- Configurable authentication methods to authorize creation of signature
- Client should still use multiple devices (for authorization and to check distributed key generation) to avoid single point of failure
- Command-line tool to interact with FROSIX service providers

What are threshold signatures?

FROSIX is an effort to add proper authorization procedures (in the same style as Anastasis) and a REST protocol to orchestrate the process from a client.

1. Given that we do not trust the client, a user would need to use multiple client devices, after all a compromised client might display that they have set up a distributed key signing system but only use a single compromised provider (or even only local computation!) behind the scenes/screens.

2. Thus, FROSIX has provisions to check which providers were involved in the key generation on other devices.
1. This is an overview of the FROSIX architecture, consisting of various supporting libraries to implement the REST service and payments below, the core cryptographic FROST implementation, and then the FROSIX protocol implementation.

2. Various authentication methods are implemented, but the system is extensible.

3. What is missing today is any kind of nice user interface, as only a command-line tool exists.
Open issues:
- Support additional signature schemes beyond EdDSA
- Pay signature service providers for their service
- Graphical user interfaces (Gtk+, WebUI,...)
GNU Taler has many types of keys:
- Coin keys (EdDSA + ECDHE)
- Denomination keys (blind)
- Online message signing keys
- Offline key signing keys
- Merchant keys
- Auditor key
- Security module keys
- Transfer keys (ECDHE)
- Wallet keys
- TLS keys, DNSSEC keys

What does key management look like in practice?
1. This is an overview of the different types of keys used in GNU Taler.
2. The diagram on the right gives an overview of which keys are used to sign what, and quite often keys are used to basically sign over other keys.
3. Most of the keys are signing keys using EdDSA.
4. But some keys are different also because the cryptography they used with is different, especially the transfer keys (ECDHE), the denomination keys (blind signatures) and the coin keys (EdDSA + ECDHE).
5. An actual Taler deployment will also have some key material outside of the core scope of GNU Taler, such as those for securing TLS and DNS.
6. The keys also differ by the party that owns them, and in some cases the protection level applied (at a high level: online vs. offline).
**Offline keys**

Both exchange and auditor use offline keys.

- Those keys must be backed up and remain highly confidential!
- We recommend that computers that have ever had access to those keys to NEVER again go online.
- We recommend using a Raspberry Pi for offline key operations. Store it in a safe under multiple locks and keys.
- Apply full-disk encryption on offline-key signing systems.
- Have 3–5 full-disk backups of offline-key signing systems.

1. Offline keys are the most protected keys in the system.
2. They are used to sign or revoke online keys, and the public keys are distributed with the software to other users in the system.
3. Because they are offline, signing with them is extremely expensive, and may require waking up people in the middle of the night to get the signature. So they should be used rarely in operation, as otherwise their deliberately low availability might threaten the availability of the overall system.
4. HSMs, physical security (a physical vault), passwords and possibly threshold signatures are techniques to provide the highest possible level of protection for such offline keys.
Online keys

The exchange needs RSA and EdDSA keys to be available for online signing.

- Knowledge of these private keys will allow an adversary to mint digital cash, possibly resulting in huge financial losses.
- The corresponding public keys are certified using Taler’s public key infrastructure (which uses offline-only keys).

taler-exchange-offline can be used to revoke the online signing keys, if we find they have been compromised.

1. Some keys inherently need to be online, say because users frequently interact with them. For these online signing keys, we actually expect that a Taler operator at scale may use them tens of thousands of times per second.
2. Thus, the primary protection has to be that they are still used on a machine with tight logical and physical access control.
3. An HSM is theoretically an option, but today HSMs on the market rarely support blind signatures and are very expensive if you need them to do tens of thousands of signatures per seconds.
Protecting online keys

The exchange needs RSA and EdDSA keys to be available for online signing.

- `taler-exchange-secmod-*` are the only processes that must have access to the private keys. These `secmod` processes should run under a different UID, but share the same GID with the exchange.
- The `secmods` generate the keys, allow `taler-exchange-httpd` to sign with them, and eventually delete the private keys.
- Communication between `secmods` and `taler-exchange-httpd` is via a UNIX domain socket.
- Online private keys are stored on disk (not in database!) and should NOT be backed up (RAID should suffice). If disk is lost, we can always create fresh replacement keys!

### What does key management look like in practice?

1. GNU Taler today thus implements a different protection strategy, that is minimizing the amount of code that has the right to interact with the key, minimizing the surface to interact with that code, and running that code under a specific UID.
2. The entire key lifecycle management is implemented in these well-isolated and easy to audit helper processes: key creation, key use and key deletion.
3. Furthermore, while the private keys are stored on disk, it is conceivable to make that disk a RAM disk — assuming sufficient availability can be assured by someone signing replacement keys with the offline tool in a timely fashion after a loss of power!
What happens if private keys are disclosed or lost?

1. Especially for such online signing keys, we always have to do a threat model as to what could happen if control over these keys is lost.
2. For signing keys, the case where they are lost can imply downtime or (in the case of keys representing financial assets) a (hopefully small) financial loss to the owner.
3. What if signing keys used to create digital cash are disclosed to an adversary?
Denomination key \((d, n)\) disclosure

- Auditor and exchange can detect this once the total number of deposits exceeds the number of legitimate coins.
- At this point, \((e, n)\) is revoked. Users of unspent legitimate coins reveal \(b\) from their withdrawal operation and obtain a refund.
- The financial loss of the exchange is bounded by the number of legitimate coins signed with \(d\).
- Taler frequently rotates denomination signing keys and deletes \(d\) after the signing period of the respective key expires.

Rightarrow: Taler frequently rotates denomination signing keys and deletes \(d\) after the signing period of the respective key expires.

What does key management look like in practice?

1. An attacker who learns \(d\) can sign an arbitrary number of illicit coins into existence and deposit them.
2. Detecting that signatures were created outside of the control of the legitimate system is key. Consider also Certificate Transparency for X.509: we cannot prevent bad certificates from being signed, but at least we learn about it!
3. The next step is to assess the damage, and to stop further losses. Like in X.509, this involves revoking the compromised keys and possibly issuing new signatures to legitimate users.
4. Finally, we can try to limit the extent of the damage: the private key is deleted after the “withdraw” period expires, and signatures with it are only accepted up to a certain “deposit” time. This way, an adversary has limited windows of opportunity.
Deirdre Connolly, Chelsea Komlo, Ian Goldberg, and Christopher A. Wood.
Two-round threshold Schnorr signatures with frost.

Dominik Samuel Meister and Dennis Neufeld.
Anastasis: Password-less key recovery via multi-factor multi-party authentication.
Acknowledgements

Co-funded by the European Union (Project 101135475).

Co-funded by SERI (HEU-Projekt 101135475-TALER).

Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.