Attacks to deployed threshold signatures

Omer Shlomovits
omer@ZenGo.com

JP Aumasson
jp@taurusgroup.ch

NIST MPTS workshop 2020

See results' details in https://eprint.iacr.org/2020/1052

Agenda

- Threshold signing theory and practice
- New attacks on threshold ECDSA production code
  - Forget-and-Forgive: Reshare protocol sabotage
  - Golden Shoe: Leaky share conversion
- Q&A

Threshold signature schemes (TSS)

Probably already clear thanks to previous speakers :)

(t, n) threshold signing, $t < n$

- Signing key represented as n shares
- Distributed key generation (DKG)
- t+1 shares necessary and sufficient to sign
- t or fewer shares “useless"

Components

- **Homomorphic** encryption (often Paillier)
- Verifiable **threshold secret-sharing** (often Shamir/Feldman)
- **Zero-knowledge** proofs (discrete log, range, etc.)
- Multiplicative-to-additive **share conversion** (“MtA”)

Among “real-world” crypto protocols, TSS are some of the most complex and with the **widest attack surface** wrt failures in subcomponents, their code, and in security proofs.

TSS security 🤝

- Standard **EUF-CMA** signature security
- Standard corrupted parties model (**static, malicious, rushing**)
  - Adaptive security usually doable with some overhead
  - Generally captures "real-world" risks
- Pure "network attacker” *mostly* captured by corruption model
- **Majority**: honest vs. dishonest one (security with *n*-1 corruptions)
- **Proofs**: different approaches; UC provides higher guarantees

TSS research 🔥

Tons of papers after Lindell 2017 and GG 2018 ECDSA protocols

Research challenges addressed so far:

- Deal with **ECDSA's *k*** sharing/operation (compared to Schnorr case)
- Minimise **rounds number** and **proofs computations**
- Detect **errors and imposters** ("identifiable aborts")
- Maximise **offline computations** (presigning)

Research driven by applications, mainly blockchain wallet/custody…

In practice 💸

Shared control implementation, as an alternative and complement to TEE-based solutions; a critical part of secure custody solutions

Multiple use cases with different requirements:

- 2-sharing between a service **provider and a client**
- (t, n) **cold wallet** within an exchange, with heterogenous systems, possibly number of shares per party depending on the system's trust
- (t, n) **Hot/warm wallet** within a single organisation

In practice ฿ Ξ

Notes on real-world TSS:

- Safe **reshare** protocols needed for shares update
- **Performance** mainly driven by network latency and processing
- Offline **presigning** not always applicable, but a nice-to-have
- Not a replacement for reliable back-up and recovery processes :)

TSS-friendly signatures gaining adoption: Schnorr signatures now in Bitcoin (BIP 340), BLS signatures in Ethereum 2.0, Celo, etc.

Real-world security ⚠️

Moxie Marlinspike
@moxie

Many trends in modern programming language design seem to focus on developers pressing fewer keys on the keyboard. To me, that's a strange priority.

For large systems where the industry spends most of its time, I think "readability" is much more important than "writability."

Implementation structures and language features can amplify a protocol's complexity

Security proofs are perhaps ~20% of what makes a TSS deployment secure

Examples of non-crypto issues observed:

- Crashes due to unsafe decoding
- Known vulnerabilities in dependencies
- Leverage of lower-level vulnerabilities (OS, runtime, etc.)
- Failures of "trusted" hardware

Real-world security ☣️

Implementing papers can be risky, when

- Developers are not used to the terminology and notations
- Encodings, primitives, etc. are not defined
- Papers sometimes hide/obfuscate critical requirements

Theorem 1 *The non-interactive proof system defined by*

- Common Input: N
- Random Input: $x \in Z_N^*$
- Prover: *compute* $M = N^{-1} \bmod \phi(N)$ *and output* $y = x^M \bmod N$
- Verifier: *accept iff* $y^N = x \bmod N$.

is one-sided error perfect zero-knowledge with soundness error at most $1/d$ *for the language* SF', *where* d *is the smallest factor of* N.

Typical errors: Non-safe primes, commitment hash not covering all values, missing range validation mod q, lack of public keys validation, etc.

⬅️ Broken ZK factorisation proof because "common input" was not defined in the paper ¯_(ツ)_/¯

The attacks 🏴‍☠️

Among our **most impactful** attacks (responsible disclosed and fixed):

- Target TSS software used by major organisations' wallets
- Arguably exploitable under realistic conditions

On an **implementation** of GG18's threshold ECDSA, but our attacks do not invalidate the security claims of the paper

Stress the important of **input validation**, and more generally of **correctness verification** in a protocol's execution

Forget & Forgive

Forget & Forgive Setup

The vulnerability was found the **“Secret Re-sharing”** protocol

Forget & Forgive Setup

The vulnerability was found the **"Secret Re-sharing"** protocol

Input: a committee of parties each holding a secret share of a secret key sk

Output: a new committee, each holding a new secret share of sk

Forget & Forgive Setup

The vulnerability was found the **“Secret Re-sharing”** protocol

Input: a committee of parties each holding a secret share of a secret key sk

Output: a new committee, each holding a new secret share of sk

Protocol:

1) Each old committee member secret-shares their sk share using **Feldman VSS**

2) Each new committee member verifies and sums its received shares

Forget & Forgive Setup

The vulnerability was found the **“Secret Re-sharing”** protocol

Input: a committee of parties each holding a secret share of a secret key sk

Output: a new committee, each holding a new secret share of sk

Protocol:

1) Each old committee member secret-shares their sk share using **Feldman VSS**

2) Each new committee member verifies and sums its received shares

Where is the problem ?

Forget & Forgive Vulnerability

For simplicity, wlog, assume the new committee is the same as the old committee

Protocol:

1) Each old committee member secret-shares their sk share using **Feldman VSS**

2) Each new committee member verifies and sums its received shares.

3) Each committee member overwrites the old secret share with the new share

Forget & Forgive Vulnerability

For simplicity, wlog, assume the new committee is the same as the old committee

<u>Protocol</u>:

1) Each old committee member secret-shares their sk share using **Feldman VSS**

2) Each new committee member verifies and sums its received shares.
   If at least one share is invalid, then **return**

3) Each committee member overwrites the old secret share with the new share

Forget & Forgive Vulnerability

For simplicity, wlog, assume the new committee is the same as the old committee

Protocol:

1) Each old committee member secret-shares their sk share using **Feldman VSS**

2) Each new committee member verifies and sums its received shares.
   If at least one share is invalid, then **return**

3) Each committee member overwrites the old secret share with the new share

- A party receiving an invalid share —> will abort the protocol, keeping its old share
- A party receiving valid shares —> will finish the protocol, overwriting the old share

Forget & Forgive Vulnerability

An attacker will divide the committee by sending valid shares to a subset, and invalid shares to the other subset.

Forget & Forgive Vulnerability

An attacker will divide the committee by sending valid shares to a subset, and invalid shares to the other subset.

From the security release:

It allows for a malicious actor to cause a new committee member to abort the protocol, unable to write a valid share to disk. The other participants would continue as normal and overwrite their share data

Forget & Forgive Exploit

- The adversary model allows for t corrupted parties, however the attack can be mounted **by a single party**

Forget & Forgive Exploit

- The adversary model allows for t corrupted parties, however the attack can be mounted **by a single party**
- In some cases, even a **network adversary** that corrupts selected messages can mount such attack

Forget & Forgive Exploit

- The adversary model allows for t corrupted parties, however the attack can be mounted **by a single party**
- In some cases, even a **network adversary** that corrupts selected messages can mount such attack
- Example exploitation scenarios:
  - Money lock
  - Money loss (in case the key is not backed up)
  - Money extortion (if attacker gets enough reshare iterations)

Forget & Forgive Mitigation

From the security release:

a final round has been added to the re-sharing protocol where the new committee members send ACK messages to members of both the old and new committees. Each participant must receive ACK messages from n members of the new committee (excluding themselves) before they save any data to disk.

Forget & Forgive Mitigation

From the security release:

a final round has been added to the re-sharing protocol where the new committee members send ACK messages to members of both the old and new committees. Each participant must receive ACK messages from n members of the new committee (excluding themselves) before they save any data to disk.

- The requirement of a “**Blame phase”** was observed in classical works on DKG
- The [GG18] protocol assumes a dishonest majority, therefore, a single party can abort the resharing protocol (no robustness)

Golden Shoe

Golden Shoe Setup

- [GG18] MtA 2-party share conversion

Fast Multiparty Threshold ECDSA with Fast Trustless Setup

Rosario Gennaro[1] and Steven Goldfeder[2]

3 A share conversion protocol

Golden Shoe Setup

- [GG18] MtA 2-party share conversion

- **Input**: Alice and Bob hold multiplicative secret shares a, b
- **Output**: additive secret shares α, β such that $\alpha + \beta = a \cdot b \bmod q$

Golden Shoe Setup

- [GG18] MtA 2-party share conversion

- **Input**: Alice and Bob hold multiplicative secret shares a, b
- **Output**: additive secret shares α, β such that $\alpha + \beta = a \cdot b \mod q$

Protocol:

- Paillier cryptosystem

- For security against malicious adversaries, need for ZK proofs.

- In all zk proofs, the prover must use an RSA group (modulus N), not knowing the group order, as well as two group elements, h_1, h_2, not knowing the relation between them

Golden Shoe Vulnerability

Protocol:

- Paillier cryptosystem

- For security against malicious adversary use ZK proofs.

- In all proofs the prover must use an RSA group (modulus N), not knowing the group order, as well as two group elements, h_1, h_2, not knowing the relation between them

- N, h_1, h_2 must be publicly verifiable and tested

Golden Shoe Vulnerability

- N, h_1, h_2 must be publicly verifiable and tested
- The two popular methods in the literature are:

1) A trusted party generates N, h_1, h_2

2) The verifier generates N, h_1, h_2 and proves their validity in ZK

Golden Shoe Vulnerability

- N, h_1, h_2 must be publicly verifiable and tested
- The two popular methods in the literature are:

  1) A trusted party generates N, h_1, h_2

  2) The verifier generates N, h_1, h_2 and proves their validity in ZK

- In the library attacked, the two methods got mixed: The verifier generates the parameters and sends them to the prover, however, **the prover does not check them**!

Golden Shoe Vulnerability

- N, h_1, h_2 must be publicly verifiable and tested
- The two popular methods in the literature are:

  1) A trusted party generates N, h_1, h_2

  2) The verifier generates N, h_1, h_2 and proves their validity in ZK

- In the library attacked, the two methods got mixed: The verifier generates the parameters and sends them to the prover, however, **the prover does not check them**!
- Classical case of missing input sanitisation, as in web applications

Golden Shoe Exploit

- N, h_1, h_2 are crucial to the proof. Specifically "*Zero knowledge requires that discrete logs of* h_1, h_2*, relative to each other modulo* N exist (i.e*. that* h_1, h_2*, generate the same group)*

Golden Shoe Exploit

- N, h_1, h_2 are crucial to the proof. Specifically "*Zero knowledge requires that discrete logs of* h_1, h_2, *relative to each other modulo* N exist (i.e. *that* h_1, h_2, *generate the same group)*

- During KeyGen, a malicious verifier can pick ANY N, h_1, h_2 and send them to all $n - 1$ parties.

Golden Shoe Exploit

- N, h_1, h_2 are crucial to the proof. Specifically "*Zero knowledge requires that discrete logs of* h_1, h_2*, relative to each other modulo* N exist (i.*e. that* h_1, h_2*, generate the same group)*

- During KeyGen, a malicious verifier can pick ANY N, h_1, h_2 and send them to all $n - 1$ parties.

- We focus on a **range proof** (due to its relative simplicity). Proving that a Paillier ciphertext encrypts a bound secret $x_i < B$.

Golden Shoe Exploit

- In the first step the prover uses the parameters N, h_1, h_2 to produce a Pedersen commitment in a group of unknown order : $z = h_1^{x_i} h_2^{\rho} \ mod \ N$ and send z to the verifier.

Golden Shoe Exploit

- In the first step the prover uses the parameters N, h_1, h_2 to produce a Pedersen commitment in a group of unknown order : $z = h_1^{x_i} h_2^{\rho} \ mod \ N$ and send z to the verifier.

- Assume the verifier picks $h_2 = 1$: we are left with $z = h_1^{x_i} \ mod \ N$

Golden Shoe Exploit

- In the first step the prover uses the parameters N, h_1, h_2 to produce a Pedersen commitment in a group of unknown order : $z = h_1^{x_i} h_2^{\rho} \ mod \ N$ and send z to the verifier.

- Assume the verifier picks $h_2 = 1$: we are left with $z = h_1^{x_i} \ mod \ N$

  - {Option 1}: pick $h_1 = 2$ and pick very large N such that $h_1^{x_i}$ is computed over the integers => solve for x_i by trial and error

  - {Option 2}: Choose N to be a composite with small prime factors => use Polling Hellman and field seive on each factor

Golden Shoe Exploit

- The attack can be mounted by a single party given persistence during KeyGen and at least one Signing :

Golden Shoe Exploit

- The attack can be mounted by a single party given persistence during KeyGen and at least one Signing :

1. During DKG: Attacker broadcasts N, h_1, h_2 to all parties

Golden Shoe Exploit

- The attack can be mounted by a single party given persistence during KeyGen and at least one Signing :

1. During DKG: Attacker broadcasts N, h_1, h_2 to all parties

2. During a single signature all t parties send corrupted range proofs to the attacker as part of MtA sub protocol

Golden Shoe Exploit

- The attack can be mounted by a single party given persistence during KeyGen and at least one Signing :

1. During DKG: Attacker broadcasts N, h_1, h_2 to all parties

2. During a single signature all t parties send corrupted range proofs to the attacker as part of MtA sub protocol.

3. The attacker will **learn all secret key shares**

Golden Shoe Exploit

- The attack can be mounted by a single party given persistence during KeyGen and at least one Signing :

1. During DKG: Attacker broadcasts N, h_1, h_2 to all parties

2. During a single signature all t parties send corrupted range proofs to the attacker as part of MtA sub protocol.

3. The attacker will **learn all secret key shares**

4. Signature will pass verification

Golden Shoe Exploit

- The attack can be mounted by a single party given persistence during KeyGen and at least one Signing :

1. During DKG: Attacker broadcasts N, h_1, h_2 to all parties

2. During a single signature all t parties send corrupted range proofs to the attacker as part of MtA sub protocol.

3. The attacker will **learn all secret key shares**

4. Signature will pass verification

Wait, there's more!

See results' details in https://eprint.iacr.org/2020/1052

- ***Lather, Rinse, Repeat*** - in the paper
- ***Baby Shark*** - threshold EdDSA, currently in responsible disclosure

Thank you! Questions?

omer@zengo.com jp@taurusgroup.ch