# Attacking Threshold Wallets

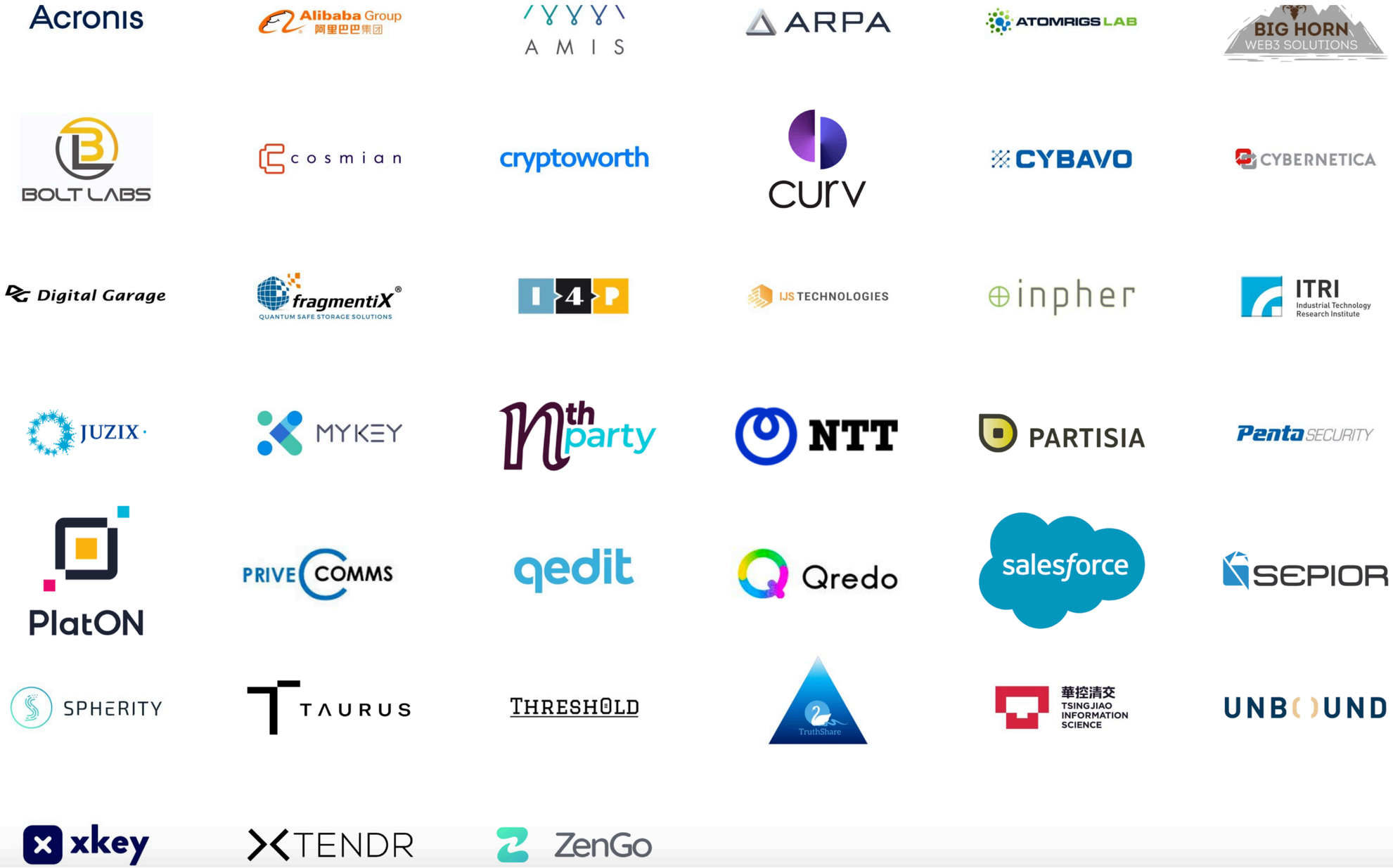
**Omer Shlomovits**

omer@ZenGo.com


JP Aumasson

jp@taurusgroup.ch


**RWC 2021**



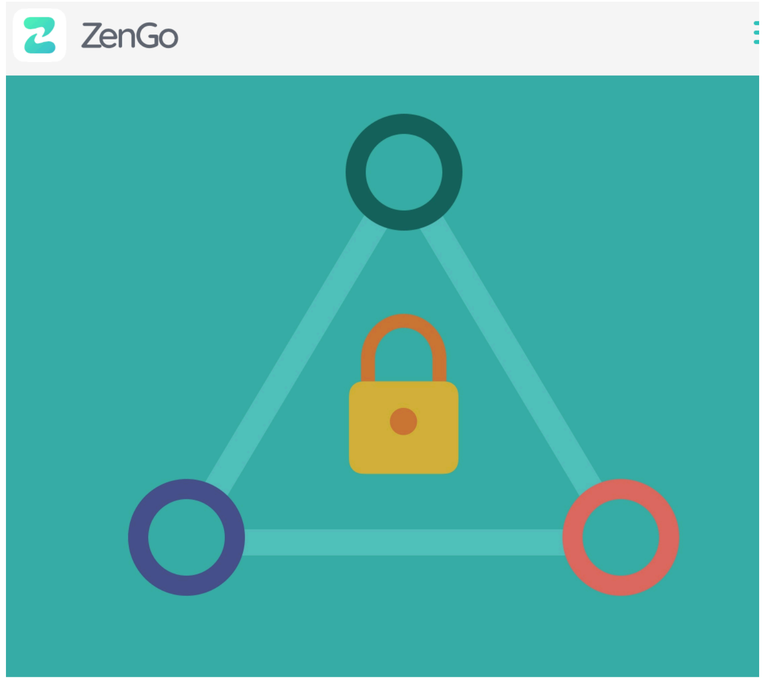

Source :

MPC ALLIANCE

# Unbound Releases Open Source Library for Blockchain Developers, Provides Proven Security for Crypto Assets

The blockchain-crypto-mpc library stands as a game changer for blockchain developers, arming them with bank-grade security technology that far surpasses existing options for securing crypto assets and wallets

UNBOUND
(MATH OVER MATTER)




Crypto

## Introducing Multi-Party ECDSA library

# ING RELEASES MULTIPARTY THRESHOLD SIGNING LIBRARY TO IMPROVE CUSTOMER SECURITY

*Security* *14.12.2020* *06:22 Pm*

ING

## Binance Open-Sources Threshold Signature Scheme Library

**Binance continues to contribute to open-source blockchain development, improving the security of funds and information for Binance Chain, Bitcoin networks and more.**

# Map

- ~~Threshold signing theory and practice~~ (A Survey of ECDSA Threshold Signing)
- New attacks on threshold ECDSA impl. (Attacking Threshold Wallets)
  - Forget-and-Forgive: Re-share protocol sabotage
  - ~~Latter-Rinse-Repeat: secret key oracle~~
  - Golden Shoe: Leaky share conversion
- ~~New attacks on threshold EdDSA impl.~~ (RWdC)
- ~~Learnings & Best Practices~~ (BlackHat USA2020)
- Q&A

# Threshold signature schemes (TSS)

1. Distributed key generation (DKG)
2. ($t$, $n$) threshold signing, $t < n$
    - Signing key represented as $n$ shares
    - $t$+1 shares necessary and sufficient to sign
    - $t$ or fewer shares “useless”
3. Secret key must re-shared from time to time

# Forget & Forgive

Re-share protocol sabotage

# Forget & Forgive Setup

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

# Forget & Forgive Setup

The vulnerability was found in 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 in 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 in 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

**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

# 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 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

Leaky share conversion

# 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  $\alpha, \beta$ such that $\alpha + \beta = a \cdot b \mod q$

# Golden Shoe Setup

- [GG18] MtA  2-party share conversion

- **Input**:  Alice and Bob hold multiplicative secret shares $a, b$

- **Output**: additive secret shares  $\alpha, \beta$ 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 verifiable and tested

# Golden Shoe Vulnerability

- $N, h_1, h_2$ must be 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

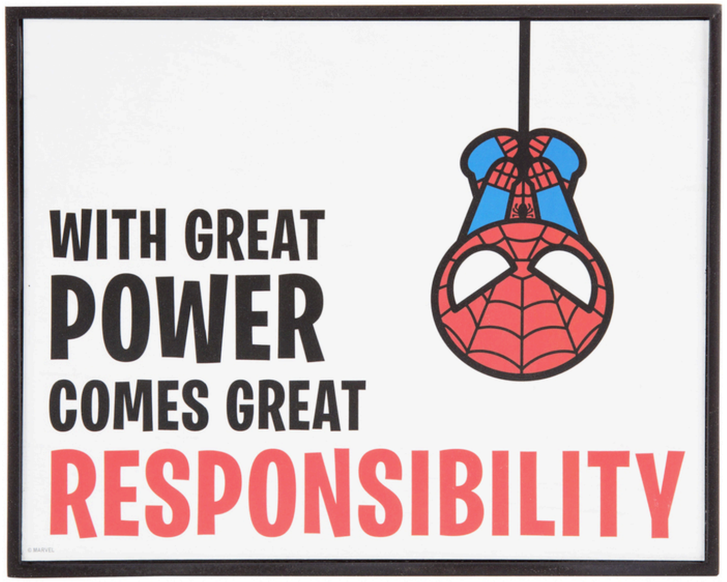
# Golden Shoe Mitigation

- The verifier must prove correctness of $N, h_1, h_2$

# Takeaways

- MPC and TSS offer high assurance on paper thanks to math proofs, but remain susceptible to misimplementations or overlooked threat vectors

- Should I use TSS ?




@OmerShlomovits, omer@ZenGo.com