Multiple Bugs in Multi-Party Computation: Breaking Cryptocurrency's Strongest Wallets

JP Aumasson[1] and Omer Shlomovits[2]

[1]Kudelski Security, Taurus group
[2]ZenGo X

Abstract

We describe a new type of logical vulnerabilities, enabled by extra layers of complexity in threshold signatures schemes (TSS) implementations. These vulnerabilities opens up a new attack surface and devastating attacks allowing a malicious participant to sabotage key generation and break TSS's security, allowing an attacker, for example, to empty an organization's cold wallet. In this paper we explore three vulnerabilities instances, found in open source libraries. For each one we discuss technical details, show how to exploit and provide mitigation. We conclude with lessons learned and best practices across the development pipeline of complex cryptographic software, including extensive testing, defense-in-depth controls, how to implement new academic work, and how an audit by specialists should be done to be the most effective.

1 Introduction

Cryptocurrency wallets in exchange platforms or banks require strong security because they protect vast amounts of money. Some solutions rely on advanced cryptographic methods that distribute trust across multiple parties, in the spirit of Shamir's secret-sharing. These include multi-party computation (MPC) and threshold signature schemes (TSS), which are a special case of MPC to sign data in a distributed, yet trustless manner. TSS has notably been tested and deployed in major organizations where secret key generation and digital signing are needed. But these techniques, although powerful and "magic" on paper, can prove fragile in practice.

1.1 Setup and Attacker Model

In all the applications we tested, TSS guarantees no single point of failure. In our attacks we demonstrate how an attacker with full control over a

single party can break the security of the TSS. For Simplicity, we assume that the exchange is compromised, modelling either an outside attacker or some insider threat. In the setup figure (figure 1), starting from the left we

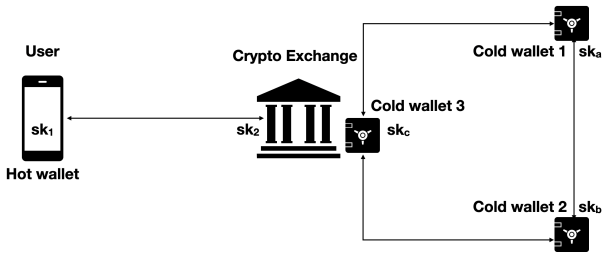

Figure 1: Setup

have a user and the exchange run a two party key generation, resulting with the user getting a secret key share sk_1 and the exchange getting a secret key share sk_2. The user deposit funds to his address and from that point can initiate orders. Each order comes with a signature generated by running a two party signing between the user and the exchange. On the other side, the crypto exchange runs an infrastructure to manage access to its liquidity. as part of this setup there are several sites that together can authorise the transfer of large amounts, per the requirements of the exchange. We are using the terms "hot wallet" and "cold wallet" to express the frequency of the use and mention it since this is terminology used in the industry, but it has no relevance to our attacks.

1.2 Attacks Taxonomy

While found on different libraries, implementing different protocols for TSS, we argue that there are common traits to the issues we identified. At first, TSS protocols are interactive. They progress in rounds of communication between the parties. While in modern cryptography we usually get a single party to generate key and sign locally, here we must have interaction. This is essentially all that was needed to mount the first attack. Second, TSS often makes use in some cryptographic primitives which are not common. This makes them hard to understand or to nail properly how they work. Finally, it is important to note that in the real world, where an attacker can attack any one of the participating parties, a party cannot assume anything about the correctness of the messages it receives. This means that for any message sent, the sender needs to attach a proof that the message was computed according to the protocol. One common way of doing it is using zero knowledge proofs. Some of which are tailored to prove statements on particular messages in the protocol.

2 Attack I: Forget-And-Forgive

In this attack we show how it was possible to erase secret shares for honest parties. The library we attacked implements [GG18] for threshold KeyGen and threshold Signing and in addition implements a protocol for threshold secret key rotation.

Key rotation is a common industry practice. If every safe (figure 1) holds a secret share of a key at time t, we want a protocol for all safes to change their secret key shares at time $t' > t$ but still maintain access to the funds locked under the joint public key. This is a basic requirement for such a system because otherwise a single attacker will be able to systematically compromise site by site until uncovering the full private key

The Reshare protocol has no formal protocol description and is not described in [GG18]. In fact we did not encounter in the literature a practical protocol for Resharing in a malicious dishonest majority setting, which is the adversary model assumed in [GG18].

The Reshare protocol in the code is based on verifiable secret sharing (VSS) [vss]. The protocol steps can be sketched as follows:

1. $t+1$ old parties transforms their linear secret sharing to additive secret sharing $sk = w_1 + ... + w_{t+1}$ where sk is the distributed secret key, unknown to all, and w_i is the secret share of sk that party P_i knows.

2. Each old party Po_i is sampling a random polynomial,taking its zero coefficient to be w_i.

3. Each Po_i broadcasts commitment vector to the coefficients (As required by VSS)

4. Each Po_i sends to each of the n' new parties a secret share sampled as a point on the random polynomial.

5. Each new party Pn_j verifies the secret shares using the VSS commitments and takes the sum of the secret shares to be its new secret.

6. The new parties check that the old public key equals the new public key.

7. If all checks passed the party rewrites over its old secret share w_i the new secret w_i'.

We note that new parties are also generating new paillier keys and verify them however this is less relevant to the attack.

The attack: Under the adversary model, we are allowed to corrupt t out of the $t + 1$ old parties. A malicious old party will send "bad" secret shares to some parties in n' and "good" secret shares to the rest. "good" means that step 5 above passes, "bad" that it fails. The new parties that

got "good" shares will continue in the protocol and will finish it successfully, therefore rewriting their old secret share with a new one. The new parties that got the "bad" secret shares will abort the protocol and will keep their old secret share. Old secret shares and new secret shares combined are not additive shares of sk anymore. We effectively "deleting" the secret shares for one of the subsets.

Implication: In our model we assume the exchange is controlled by the attacker. We assume a multiparty (> 2) setting. The simple attack is for the exchange to be able to delete the key forever. A smart attacker can leverage this situation to mount a ransom attack: running enough Refresh protocols such that all honest parties cannot reach the required threshold to execute a signature, without the involvement of the attacker. The attacker will require half the locked amount to publish its key.

Fix: In VSS based protocols with malicious adversaries there should be a blame phase. See classical works on Distributed key generation. In the case of dishonest majority there is not much to do besides aborting the protocol *before rewriting the secret* in the case of another party claiming to have been treated bad secret share. Therefore the solution is to simply add another round of communication in which parties can ask the other honest parties to abort the protocol before deletion of the key material.

3 Attack II: Lather, Rinse, Repeat

The library we looked into implements a version of Lindell's two party ECDSA [Lin17]. Here again it is important for the parties to refresh their secret shares every once in a while. It could be after every transaction for example. Conceptually this is the same as what we saw before. Concretely though, we are looking at a different protocol. The bug we found can lead to breaking security of two party key-gen as we show below. The problem starts with key refreshing. This protocol must take care of refreshing both party one Paillier keypair, and party two ciphertext encrypting party one secret share using party one Paillier public key. The relevant part of protocol in the library works as follows:

1. Party one generates a new Paillier keypair
2. Party one sends a new ciphertext under the new Paillier key
3. Party one proves in zero knowledge that there is a linear relation between the value encrypted in the new ciphertext and the value encrypted in the old ciphertext

The refresh protocol therefore runs a sub protocol proving that the same value is encrypted under two different Paillier keys. The specific proof in the code is probably taken from [Bou00] Appendix A, and can work only if

there is one modulo with unknown order (for the prover/party one). In the code however, the prover knows both factorizations of N_{new} and N_{old} and can cheat. The prover can convince the verifier that the new Paillier ciphertext is an encryption of any value x_1 chosen by the prover. Unfortunately, this gives party one the following oracle access:

1. choose N_i ,a random RSA public key
2. check if $x_2 < N_i - f(q)$, where $f(q)$ is some function of q (equals 0 mod q) such that $0 < N_i - f(q) < q$. q is the order of the elliptic curve group

In practice the attacker will do a "key refresh" to a new N_i and take its secret share to be $f(q)$. In a correct execution it is suppose to be $x_1' = x_1 - d$ for some random d but our attack allow this to be $f(q) + x_1 - d$ so that d is canceled with the $+d$ in $c_{key} \oplus Enc(x_2)$ and for simplicity we can take the original x_1 to be equal to 0. If there was a mod N_i operation in $c_{key} \oplus Enc(x_2)$ the signature verification will fail. Otherwise it will pass. Each such test gives the adversary, who is running party one, some extra information on the private key share x_2.

Implications: The exchange hijacks the user secret key which means it can now sign transactions without the user involvement.

Mitigation: We offer to use [CGM16] section 4.2 but with composite groups instead of q, p. This is the case if we insist on using this specific zero knowledge proof. This will probably be less efficient than simply repeating the relevant steps from [Lin17] two party KeyGen.

4 Attack III: Golden Shoe

The library we attacked implements [GG18] threshold ECDSA protocol.

As part of the protocol there is sub-protocol called *share conversion* protocol. This protocol is a two party protocol to transform secret multiplicative inputs to secret additive inputs such that $\alpha + \beta = x = a + b /$ In the paper the protocol is described in section 3 and the required zero knowledge proofs are described in [GG18] Appendix A.

As part of the zero knowledge proofs setup, each party (acting as Verifier) must generate a RSA modulus $\widetilde{N}$ and two random values h_1, h_2. These values must be publicly verifiable and tested. However, in the code they are not tested by the receiving party acting as the Prover. This can be exploited to let a single party extract the secret shares of all other parties and effectively the full private key during a single threshold signature generation.

In detail: The values $(\widetilde{N}, h_1, h_2)$ must adhere to certain properties, as described in [FO97] section 3 under setup procedure and [MR01] subsection

6.2 under strong RSA. Some of the properties are to make sure the zero knowledge proof is sound (Verifier is motivated to choose them correctly) however there is one condition which is crucial for zero knowledge. Quoting [MR01]: "Zero knowledge requires that discrete logs of h_1, h_2 relative to each other modulo $\widetilde{N}$ exist (i.e. that h_1, h_2 generate the same group)"

Moreover, [GG18] explicitly say that $\widetilde{N}$, h_1, h_2 must be proven correct to the receiver. However, when during KeyGen a party receives these values from other parties - there is not a single test done. The receiving party simply "trusts" the tests done by the sender and saves the values in memory. A malicious sender can therefore send ANY $\widetilde{N}$, h_1, h_2 she wants. Note that previous papers describe the tests/proof to be made or assume these values are generated by a trusted third party.

Let us describe the exploit. At later point in time, during signing, the values are being used several times in the context of *share conversion*. We first note that in our attack the signing will not fail. In fact, it is extremely hard (impossible) to choose them in a way that will fail signing. We will focus on the first range proof (subsection A.1 in the proof [GG18]) because this is the simplest. In this range proof, the receiver that got the spoofed $\widetilde{N}$, h_1, h_2 is now the Prover and the sender is now the Verifier. This proof is done for each pair of parties. Therefore a single Verifier can repeat the attack with all other signing parties. In the first step of the proof the Prover computes $z = h_1^m h_2^\rho \bmod \widetilde{N}$ where m is the secret share of the Prover and ρ is random number the Prover generates. z is sent to the Verifier. From cryptographic perspective the Prover basically send a Pederson commitment in a group of unknown order. (See [LN18] 6.2.4 which uses almost identical proof but describes it in terms of Pederson commitments.). If the values are proven to be chosen according to the protocol this is indeed a correct commitment and breaking it is hard as breaking Discrete log. However, if we are allowed to choose $\widetilde{N}$, h_1, h_2 as we wish, we can make the commitment not secure. To simplify, we first set $h_2 = 1$. Now we are left with the following problem: You give an oracle the values a, n, which correspond to $h_1, \widetilde{N}$, and get a value b, corresponding to z, such that $b = a^x \bmod n$. We need to find x (which is the secret share of the proving party). The easiest option is to choose n to be very large such that a^x is computed over the integers. In that case by trial and error (checking a^y smaller than or larger than the given b for different values of y) we can recover the prover secret share in 256 trials. A somewhat more complex solution (however, completely achievable) is to choose n to be a composite with small prime factors: Then we can use Polling Hellman [poh] and field sieve algorithm on each factor (see [KL14] Chapter 9). This problem is extremely parallelizable. In fact, The breaking of Moscow voting system [mos] follows the same type of attack - computing Dlog for small primes.

Implications: We start with a key distributed between the different parties. We assume the attacker participated in the distributed key generation protocol. After running a single threshold signature, all key shares will be copied into a single location, Which means that the attacker will be able to sign transactions, ignoring all other parties.

Fix: The fix is simple. $\widetilde{N}$, h_1, h_2 must be tested on the receiving side. For $\widetilde{N}$ - the sender must attach a proof that $\widetilde{N}$ is a valid RSA modulus from two safe primes. There are various ways to prove it, the latest is from [GRSB19] (see [LN18] on the recommended parameters to use). For h_1, h_2 there is a nice trick in [FO97]: pick h_1 at random and $h_2 = h_1^{\alpha}$ and prove to the receiver the knowledge of α with respect to h_1, h_2.

References

[Bou00] Fabrice Boudot. Efficient proofs that a committed number lies in an interval. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 431–444. Springer, 2000.

[CGM16] Melissa Chase, Chaya Ganesh, and Payman Mohassel. Efficient zero-knowledge proof of algebraic and non-algebraic statements with applications to privacy preserving credentials. In *Annual International Cryptology Conference*, pages 499–530. Springer, 2016.

[FO97] Eiichiro Fujisaki and Tatsuaki Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *Annual International Cryptology Conference*, pages 16–30. Springer, 1997.

[GG18] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ecdsa with fast trustless setup. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1179–1194, 2018.

[GRSB19] Sharon Goldberg, Leonid Reyzin, Omar Sagga, and Foteini Baldimtsi. Efficient noninteractive certification of rsa moduli and beyond. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 700–727. Springer, 2019.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2014.

[Lin17] Yehuda Lindell. Fast secure two-party ecdsa signing. In *Annual International Cryptology Conference*, pages 613–644. Springer, 2017.

[LN18] Yehuda Lindell and Ariel Nof. Fast secure multiparty ecdsa with practical distributed key generation and applications to cryptocurrency custody. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1837–1854, 2018.

[mos] https://securityboulevard.com/2019/08/moscows-blockchain-based-internet-voting-system-uses-an-encryption-scheme-that-can-be-easily-broken. Accessed: 2020-06-06.

[MR01] Philip MacKenzie and Michael K Reiter. Two-party generation of dsa signatures. In *Annual International Cryptology Conference*, pages 137–154. Springer, 2001.

[poh] https://en.wikipedia.org/wiki/Pohlig–Hellman_algorithm. Accessed: 2020-06-06.

[vss] https://en.wikipedia.org/wiki/Verifiable_secret_sharing. Accessed: 2020-06-06.