BH20 Supplementary Material

Omer Shlomovits, JP

This white paper is a supplement to the Black Hat USA 2020 talk titled *Multiple Bugs in Multi-Party Computation: Breaking Cryptocurrency's Strongest Wallets* [1]. The talk is about logical bugs found in implementations of "heavy" cryptography. To match the time limit we were forced to abstract many of the concrete details of the attacks. In this paper we provide technical details for each one of the attacks we describe in the talk.

1 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 [8] for threshold KeyGen and threshold Signing and in addition implements a protocol for threshold secret key rotation. This Reshare protocol has no formal protocol description and is not described in [8]. 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 [8].

The Reshare protocol in the code is based on verifiable secret sharing (VSS) [2]. 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.

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 matarial.

2 Attack II: Lather, Rinse, Repeat

The library implements a version of Lindell's two party ECDSA [11]. 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 party one Paillier keypair and party two ciphertext encrypting party one secret share. 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 [5] which as can be seen in Appendix A, 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 $x_1 < N_i - f(q)$, 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 running party one some extra information on the private key share x_2.

Mitigation: We offer to use [6] section 4.2 but with composite groups instead of q,p. This is the case if we insist on using this specific zero knowledge. This will probably be less efficient than simply repeating the relevant steps from [11] two party KeyGen.

3 Attack III: Golden Shoe

The library we attacked implements GG19 protocol for threshold ECDSA [8].

As part of the protocol there is sub-protocol called "share protocol": In the paper the *share protocol* is described in section 3 with zero knowledge proofs described in Appendix A.

As part of the zero knowledge proofs included in *share protocol*, each party must generate 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 a 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 [7] section 3 under setup procedure and [13] 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 :

150 P. MacKenzie and M.K. Reiter

primes and for which the factorization is unknown to the prover, and that the discrete logs of h_1 and h_2 relative to each other modulo $\tilde{N}$ are unknown to the prover. Zero knowledge requires that discrete logs of h_1 and h_2 relative to each other modulo $\tilde{N}$ exist (i.e., that h_1 and h_2 generate the same group). As in Section 4.1, here we assume that these parameters are distributed to alice and bob by a trusted third party. In the full paper, we will describe how this assumption can be eliminated.

[8] explicitly say that $\widetilde{N}$, h_1, h_2 must be proven correct to the receiver:

A The ZK Proofs for the MtA protocol

In this section we describe the ZK proofs that are needed in the MtA protocol (see Section 3). The proofs are based on similar ones from [29]: specifically we prove statements that are simpler than the ones needed in [29].

In these proofs the Verifier uses an auxiliary RSA modulus $\tilde{N}$ which is the product of two safe primes $\tilde{P} = 2\tilde{p} + 1$ and $\tilde{Q} = 2\tilde{q} + 1$ with $\tilde{p}, \tilde{q}$ primes. The Verifier also uses two values $h_1, h_2 \in Z^*_{\tilde{N}}$ according to the commitment scheme in [14]. Security is based on the assumption that the Prover cannot solve the Strong RSA problem over $\tilde{N}$.

Therefore our initialization protocol must be augmented with each player P_i generating an additional RSA modulus $\tilde{N}_i$, and values h_{1i}, h_{2i}, together with a proof that they are of the correct form (see [14]).

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 protocol*. 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 [8]) 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 [12] 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 we 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 [3] and field sieve algorithm on each factor (see [10] Chapter 9). This problem is extremely parallelizable. In fact, The breaking of Moscow voting system [4] follows the same type of attack - computing Dlog for small primes.

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 [9] (see [12] on the recommended parameters to use). For h_1, h_2 there is a nice trick in [7]: 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

[1] https://www.blackhat.com/us-20/briefings/schedule/multiple-bugs-in-multi-party-computation-breaking-cryptocurrencys-strongest-wallets-19763. Accessed: 2020-06-06.

[2] https://en.wikipedia.org/wiki/Verifiable$_s ecret_s haring$. *Accessed* : $2020 - 06 - 06$.

[3] https://en.wikipedia.org/wiki/Pohlig–Hellman$_a lgorithm$. *Accessed* : $2020 - 06 - 06$.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

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

[12] 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.

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