[VSA-2022-120] Multichain: Key Extraction Vulnerability in fastMPC's Secure Multi-Party Client (smpc)*

By Verichains

December, 2022

1 Summary

We discovered a private key extraction vulnerability that breaks completely the security of the Threshold Signature Scheme (TSS) implementation by Multichain's Secure Multi-Party Client (`smpc`) at https://github.com/anyswap/FastMulThreshold-DSA. As a result, a malicious party can recover the TSS private key of a TSS group, reducing a t/n threshold scheme to 1/n. The attacker only needs to participate in 1 signing ceremony to do so. The vulnerability affects any `smpc` version from 7.1.0 up to the latest. All Multichain mainnet and testnet `smpc` nodes, using version 7.2.5 and 7.2.6 respectively, are at risk.

2 Background

Multichain's MPC uses an implementation of multi-party threshold ECDSA (elliptic curve digital signature algorithm) based on GG20: One Round Threshold ECDSA with Identifiable Abort[1] and eddsa (Edwards curve digital signature algorithm), including the implementation of approval list connected with upper layer business logic and channel broadcasting based on P2P protocol.

A `NtildeProof` (`smpc-lib/crypto/ec2/ntildeZK.go`) is used to verify that a prover knows $\log_g h$ modulo a composite number $\tilde{N}$. In the implemented TSS protocol, at key generation ceremony, each party is required to generate and broadcast a triple $\tilde{N}, h_1, h_2$ (for later use in range proofs of the MtA sub-protocol) together with 2 `NtildeProof`s for $\log_{h_1} h_2$ and $\log_{h_2} h_1$ modulo $\tilde{N}$.

Here's the interactive version of `NtildeProof`:

1. Peggy (the prover) commits to a random value $r \in \mathbb{Z}_n$ (n is the order of g known only to Peggy) by sending $\alpha = g^r$ modulo N to Victor (the verifier).

2. Victor chooses and sends Peggy a random bit $c \in \{0, 1\}$.

---

*https://multichain.org/

[1] https://eprint.iacr.org/2020/540.pdfA

3. Peggy computes and sends back $t = r + c \log_g h$.

4. Victor accepts if and only if $g^t = \alpha h^c$ modulo N.

This protocol is soundness since given a successful prover, one can apply the rewind technique to extract the discrete logarithm value similar to proving soundness of Schnorr protocol[2]. Note that knowing c in advance allows an attacker Eve, who does not have the knowledge of $\log_g h$, to trick Victor into accepting by sending him $\alpha = g^t h^{-c}$ for an arbitrary t of Eve's choice at round 1.

The interactive proof above is converted to the non-interactive `NtildeProof` by applying Fiat-Shamir transformation using the SHA512/256 hash function as a random oracle for the challenge bit c. The proof is also repeated 128 times to reduce the soundness error from $\frac{1}{2}$ (the chance of making a correct guess for c at the beginning of the protocol) to $\frac{1}{2^{128}}$. Specifically, $\overline{c_{127}c_{126}...c_0} = H(g, h, n, \alpha_0, \alpha_1, ..., \alpha_{127}) \bmod 2^{128}$, where c_i, α_i is c, α at the i-th iteration starting from 0, and H is the hash function.

2.1 Forging `NtildeProof`

A commit[3] on Jun 7, 2022, has reduced the `Iterations` parameter from 128 to 1, thus making `NtildeProof` easy to forge by trial-and-error.

Given $\tilde{N}, g, h$, the following pseudocode outputs a valid `NtildeProof` which is essentially an α, t pair:

1. Assume the challenge bit $c_0 = 0$ (can also be 1 as well).
2. Pick a random t and compute $\alpha = g^t h^{-c_0} \bmod \tilde{N}$.
3. If $H(g, h, \tilde{N}, \alpha) = c_0 \bmod 2$ (probability $\frac{1}{2}$), return α, t. Otherwise, go back to step 2.

Note that, $\log_g h \bmod \tilde{N}$ does not necessarily exist. For example, one can forge a proof for $log_1 2$ even though it does not exist.

2.2 Choosing `N~`, `h1`, `h2`

The triple $\tilde{N}, h_1, h_2$ is used in MtA range proofs. MtA stands for multiplicative-to-additive, which is a sub-protocol involving two parties Alice and Bob holding secret values a, b respectively. At the end of MtA, Alice obtains α and Bob obtains β such that $ab = \alpha + \beta \bmod q$, the *secp256k1* curve order. During a TSS signing ceremony, for a pair of 2 different parties P_i, P_j, MtA is run four times:

Iteration	Alice	a	Bob	b
1	P_i	k_i	P_j	γ_j
2	P_j	k_j	P_i	γ_i
3	P_i	k_i	P_j	w_j
4	P_j	k_j	P_i	w_i

---

[2] https://en.wikipedia.org/wiki/Proof_of_knowledge

[3] https://github.com/anyswap/FastMulThreshold-DSA/commit/4e543437c632e6ca709260d911c038f15e7663fc

The values of interest are:

- k_i: the private nonce share of P_i. If all k_i are leaked, the private nonce k could be reconstructed ($k = \sum k_i$) and combined with the message and the signature (both are public information) to recover the private key of the TSS group.
- w_i: a value depends on P_i's private key share x_i and which members of the TSS group are currently running the signing ceremony (requiring t/n members). If all w_i is leaked, the TSS private key d can be reconstructed by $d = \sum w_i$.

As a result, we conclude that leaking MtA input a or b both lead to TSS private key recovery.

Next, we need to investigate how $\tilde{N}, h_1, h_2$ is used in MtA. Let $\tilde{N}_A, h_{1A}, h_{2A}$ and $\tilde{N}_B, h_{1B}, h_{2B}$ denote Alice and Bob's $\tilde{N}, h_1, h_2$ respectively. It turns out that the following values are revealed by the range proofs used in MtA:

- $z_A = h_{1B}^a h_{2B}^{\rho_A} \bmod \tilde{N}_B$ via Alice range proof for a. ρ_A is a random value in $\mathbb{Z}_{q\tilde{N}_B}$.
- $z_B = h_{1A}^b h_{2A}^{\rho_B} \bmod \tilde{N}_A$ via Bob respondent range proof for b. ρ_B is a random value in $\mathbb{Z}_{q\tilde{N}_A}$.

It can be seen that if Bob is somehow able to eliminate $h_{2B}^{\rho_A}$ and compute a discrete log to base h_{1B} modulo $\tilde{N}_B$, then he could learn Alice's private input a. A similar result can be obtained when the attacker playing on the side of Alice.

To do so, one can choose $\tilde{N} = \tilde{p}\tilde{q}$ ($\tilde{p}, \tilde{q}$ are both prime) such that:

- $\tilde{p}$ is just around 257-bit long. Computing discrete logarithm over $\mathbb{Z}_{\tilde{p}}$ instead of $\mathbb{Z}_{\tilde{N}}$ drastically reduces its difficulty. Also, $\tilde{p} > q$, the `secp256k1` curve order.
- $\tilde{p} - 1$ is smooth to make computing discrete logarithm over $\mathbb{Z}_{\tilde{p}}$ more efficient.

Since `NtildeProof` can be forged, h_1, h_2 can be freely chosen such that:

- h_1 has large multiplicative order in $\mathbb{Z}_{\tilde{p}}$ (at least q, the curve order). Any random value is likely to satisfy this condition.
- $h_2 = 1 \bmod \tilde{p}$.

For example, let $h_1 = 2$ and $h_2 = \tilde{p} + 1$ (note that $h_2 = 1$ is rejected by the TSS implementation), given $z = h_1^x h_2^r \bmod \tilde{N}$, we have $z = h_1^x \bmod \tilde{p}$. Computing a discrete log to base h_1 should give us x, since x is either a or b (MtA inputs) bounded by the `secp256k1` curve order.

2.3 Recovering the private key

Assuming that Eve, the malicious party, has successfully generated and broadcasted, during a key generation ceremony, a triple $\tilde{N}, h_1, h_2$ together with 2 forged `NtildeProofs` for $\log_{h_1} h_2$ and $\log_{h_2} h_1$ following the above approach, upon participating in the first signing ceremony, Eve can recover the generated TSS private key by:

1. Collect all z_A in Alice range proofs received from other parties when playing as Bob in the 1st/2nd or 3rd/4th iteration of the MtA sub-protocol. Compute $k_i = \log_{h_1} z_{A,i} \bmod \tilde{p}$ for each $z_{A,i}$ received from party i.

2. Compute the nonce $k = \sum k_i$.
3. Let z and r, s denote the message hash and result signature respectively, the TSS private key can be derived from the ECDSA relation: $s = k(z + rd) \bmod q$.

3 Exploitation

The vulnerability was exploited successfully in a local testnet to recover the private key. In short, the steps to exploit are:

1. Forging `NtildeProofs` for a malicious triple $\tilde{N}, h_1, h_2$.
2. Broadcast the malicious params to other parties during a key generation ceremony.
3. Recover the generated TSS private key via MtA range proofs during the first involving signing ceremony.

Below is the output of our exploit PoC running from a malicious node on local testnet:

```
2022-12-06T12:01:02.523906525Z
2022-12-06T12:01:02.523984059Z UDP listener up, self enode://4af6cc6e7b48f2e86
2549b01d44bca404e07f3f32463c7140bef55d7f84a852525c2e31c9e26e5e07f4257187bc0563
53fcbce993e73c53edfddaf5415f80b21@[::]:4441
2022-12-06T12:01:51.770533418Z Success Generate Safe Random Prime.
2022-12-06T12:02:18.212509408Z Success Generate Safe Random Prime.
2022-12-06T12:02:42.055397130Z Success Generate Safe Random Prime.
2022-12-06T12:03:10.531709969Z Success Generate Safe Random Prime.
2022-12-06T12:03:43.133373466Z ECDSA KEY GENERATING ROUND 4 - ATTACKING MODE E
NABLED!
2022-12-06T12:03:43.133423397Z Generating malicious params...
2022-12-06T12:03:43.699562846Z Done.
2022-12-06T12:03:43.699595341Z  p=13979886554274980336075043922369961436715399
95276788072432319907517515872774167
2022-12-06T12:03:43.699600102Z  h1=1337
2022-12-06T12:03:43.699603072Z  h2=1397988655427498033607504392236996143671539
995276788072432319907517515872774168
2022-12-06T12:03:45.115345246Z ECDSA KEY GENERATING FINISHED - ATTACKING MODE
ENABLED!
2022-12-06T12:03:45.115375695Z Generated public key:  04639c0c4ec4fe0bfcf2f311
8295224b61922e912ef388fefeea3f785eec6990663ae0539820f3909912492413c16de1245392
2c2b57b849b3e15d739fde1a5498
2022-12-06T12:03:56.406497309Z ECDSA SIGNING ROUND 4 - ATTACKING MODE ENABLED!
2022-12-06T12:03:56.406544791Z Recovering nonce...
2022-12-06T12:03:56.456981979Z Nonce recovered: {"nonce": "604b612378afa1bb06d
68030af7a827a6891f923a00859807eea17c7a6ff0096"}
2022-12-06T12:03:58.878456928Z ECDSA SIGNING ROUND 4 - ATTACKING MODE ENABLED!
2022-12-06T12:03:58.878901782Z Recovering nonce...
2022-12-06T12:03:58.925025669Z Nonce recovered: {"nonce": "f40e1eeeb9068928d67
48f5fab42243909deda3628ce8e2905bf33044fb24d22"}
```

```
2022-12-06T12:04:02.072509295Z ECDSA SIGNING ROUND 4 - ATTACKING MODE ENABLED!
2022-12-06T12:04:02.072551014Z Recovering nonce...
2022-12-06T12:04:02.115203957Z Nonce recovered: {"nonce": "42dcb5f00f897648cce
ede5e22c6ae1a87db8e19f31b2c8fe48ef7cde3e1f874"}
2022-12-06T12:04:04.526795209Z ECDSA SIGNING ROUND 4 - ATTACKING MODE ENABLED!
2022-12-06T12:04:04.526844608Z Recovering nonce...
2022-12-06T12:04:04.583039899Z Nonce recovered: {"nonce": "27f01891515bc6693b7
a6e97dd19857f9456b43003728a54ceb1437c2b120504"}
2022-12-06T12:04:06.983777405Z ECDSA SIGNING ROUND 4 - ATTACKING MODE ENABLED!
2022-12-06T12:04:06.984247483Z Recovering nonce...
2022-12-06T12:04:07.030597588Z Nonce recovered: {"nonce": "f327acf129d27d2f513
2cbc717d2612075080c3fb3ee6dd7e09f06a0fa63cd0e"}
2022-12-06T12:04:09.416853311Z ECDSA SIGNING ROUND 4 - ATTACKING MODE ENABLED!
2022-12-06T12:04:09.416902408Z Recovering nonce...
2022-12-06T12:04:09.470678907Z Nonce recovered: {"nonce": "619d2994bdf53c43394
8a196ba369680610957418b55aa76cfd05d0cee98cb78"}
2022-12-06T12:04:11.886891397Z ECDSA SIGNING ROUND 4 - ATTACKING MODE ENABLED!
2022-12-06T12:04:11.886941238Z Recovering nonce...
2022-12-06T12:04:11.934133644Z Nonce recovered: {"nonce": "cfb246ad4af06b1f90c
7a65a8f8ac41357ada212fece475374e857aaa24638a5"}
2022-12-06T12:04:15.050925140Z ECDSA SIGNING ROUND 4 - ATTACKING MODE ENABLED!
2022-12-06T12:04:15.051012779Z Recovering nonce...
2022-12-06T12:04:15.100951769Z Nonce recovered: {"nonce": "e998f91af3eecea735d
1e5ac216986618f1ae5e0d0bbd148cbd2b78e31370bb7"}
2022-12-06T12:04:17.283668877Z ECDSA SIGNING ROUND 4 - ATTACKING MODE ENABLED!
2022-12-06T12:04:17.283766892Z Recovering nonce...
2022-12-06T12:04:17.338165881Z Nonce recovered: {"nonce": "1eecfe9f5dd8331ab7e
9eb7e9ffce584bdfac8c9bde39aae2faa88217dc63bfa"}
2022-12-06T12:04:19.616866563Z ECDSA SIGNING ROUND 4 - ATTACKING MODE ENABLED!
2022-12-06T12:04:19.616926849Z Recovering nonce...
2022-12-06T12:04:19.664266938Z Nonce recovered: {"nonce": "852a6bda7c29f6ca3eb
938c99fd3ebd94bc8a9155724370c2dd4fbdbb5cf6a3e"}
2022-12-06T12:04:27.289974470Z ECDSA SIGNING ROUND 4 - ATTACKING MODE ENABLED!
2022-12-06T12:04:27.290025287Z Recovering nonce...
2022-12-06T12:04:27.339265414Z Nonce recovered: {"nonce": "f7a7ae7b8b99dd4c6c6
b91c603549d1cb562ee67b85d4369b70a2e363e68a4df"}
2022-12-06T12:04:40.795847323Z  ECDSA Signature Verify Passed! (r,s) is a Vali
d Signature
2022-12-06T12:04:40.795894177Z ECDSA SIGNING FINISHED - ATTACKING MODE ENABLED
!
2022-12-06T12:04:40.795897680Z Recovering private key...
2022-12-06T12:04:40.807612263Z Private key recovered: {"private_key": "344baeb
4f4b07b782a60e5bad1a9e9786ad28b3ab5d21d405c909a6d2cd57042", "public_key": "046
39c0c4ec4fe0bfcf2f3118295224b61922e912ef388fefeea3f785eec6990663ae0539820f3909
912492413c16de12453922c2b57b849b3e15d739fde1a5498"}
2022-12-06T12:04:40.807679740Z  r = 106902443427639498321207611372712013732475
172300043096027809929046979058428310
```

```
2022-12-06T12:04:40.807688421Z  s = 535377015853193739489643408127267164275156
919047993851091258776682381044120
2022-12-06T12:04:40.807692735Z SignEC3,verify (r,s) pass,rsv str = EC58A386DA6
7BB7927D465CADAC3CD6E39E36667A4F70CBD292ABE2AEC516996012F033D338E852178B755BE7
D9CE821BC2D7DE23AD7107EF91069C4ECE5E99801, key = 0x28795447d8f6976fedad13fe671
05973804f6de1ee7fae39dab5be6bc0818394
```

4 Suggested Fix

Just revert the commit[4] to increase Iterations parameter to 128.

---

[4] https://github.com/anyswap/FastMulThreshold-DSA/commit/4e543437c632e6ca709260d911c038f15e7663fc