PUBLIC

Code Assessment

of the Scribe Smart Contracts

July 04, 2023

Produced for

by

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 11
4 Terminology 12
5 Findings 13
6 Resolved Findings 16
7 Informational 20
8 Notes 22

1 Executive Summary

Dear all,

Thank you for trusting us to help Chronicle with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Scribe according to Scope to support you in forming an opinion on their security risks.

Chronicle implements Scribe, a Schnorr multi-signature based price oracle. An optimistic extension allows price updates where the signature is only evaluated on-chain if challenged. Reading the pricefeed on-chain is restricted to whitelisted addresses only.

The most critical subjects covered in our review are functional correctness, integration of the signature scheme, and access control. All uncovered issues have been either fixed or acknowledged. Notable findings included: .. [Security regarding all the aforementioned subjects is high.]

- Update using stale pokedata
- Assessement of Finalized after authed action
- unreset oppokedata after unsuccessful challenge

The general subjects covered are code complexity, integration by external systems and the quality of the specification / documentation. The correctness of the signature scheme itself was not in scope of this review.

In summary, we find that the codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	2
• Code Corrected	1
• Risk Accepted	1
Low-Severity Findings	5
• Code Corrected	1
• Specification Changed	1
• Risk Accepted	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Scribe repository based on the documentation files.

The scope consists of the following solidity smart contracts:

1. scribe/src/*
2. scribe/lib/chronicle-std/src/*

The review of LibSecp256k1's implementation was conducted based on the specifications provided in the comments and the formulas outlined on https://www.hyperelliptic.org/EFD/g1p/auto-shortw-jacobian.html#addition-madd-2007-bl. It is presumed that these formulas are correct.

Furthermore, LibSchnorr's implementation was also assessed in comparison with the formulas documented in the comments and in Schnorr.md.

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	29 May 2023	f4566b889509477ce0af94e51c6665d3e0f29ecf	Initial Version
2	26 June 2023	633292333390a6ea630ff35dfa3cfe01c9f25040	After Intermediate Report

For the solidity smart contracts, the compiler version `0.8.16` was chosen.

2.1.1 Excluded from scope

Any other file not explicitly mentioned in the scope section. In particular tests, scripts, external dependencies, and configuration files are not part of the audit scope.

Although thoroughly checked, the cryptographic schemes, notably the custom Schnorr-based signature scheme, is not part of the review.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Scribe is an efficient Schnorr multi-signature based oracle, which aggregates and verifies the signatures of a group of feeds on a `(value, age)` tuple via a custom Schnorr scheme. The oracle only advances to a new `(value, age)` tuple if the verification succeeds.

ScribeOptimistic, an optimistic extension, allows one feed to push an update optimistically: The update is accepted if it has not been challenged successfully within the challenge period.

Specifically, Scribe performs elliptic curve computations on the secp256k1 curve for aggregation and verification. Feeds are ward-lifted identities represented by their distinguished public keys as a point on the curve. The onchain keys aggregation conducts points addition, which is expensive regarding the mod inverse operations. Scribe leverages the conversion between Affine coordinates and Jacobian coordinates to conduct points addition efficiently by reducing the inverse operation to 1.

2.2.1 Customized Schnorr Signature Scheme

The scheme works on elliptic curve secp256k1. Following terminology will be used:

- H() - Keccak256 hash function
- $()_e$ - Ethereum address: right most 20 bytes of Keccak256 hash result
- || - Concatenation operator
- G - Generator of secp256k1
- Q - Order of secp256k1
- x - The signer's private key as type `uint256`
- P - The signer's public key, i.e. [x]G, as type `(uint256, uint256)`
- P_x - P's x coordinate as type `uint256`
- P_p - Parity of P's y coordinate, i.e. 0 if even, 1 if odd, as type `uint8`
- m - Message as type `bytes32`. Note that the message **SHOULD** be a keccak256 digest
- k - Nonce as type `uint256`

2.2.1.1 Single Signing

The public key P signs via the signature s and the *commitment* R_e the message m by the following steps:

1. Select a cryptographically secure nonce k where `1<=k<Q`
2. Compute `R = [k]G`
3. Derive R_e being the Ethereum address of R. Let R_e be the *commitment*
4. Construct the *challenge* as:
$$e = H(P_x \,||\, P_p \,||\, m \,||\, R_e)$$
5. Compute the *signature* as:
$$s = k + (e * x) \bmod Q$$

2.2.1.2 Single Verification

1. Compute the *challenge* as:
$$e = H(P_x \,||\, P_p \,||\, m \,||\, R_e) \bmod Q$$
2. Compute the *commitment* as:
$$[s]G - [e]P = [k + (e * x)]G - [e]P = [k + (e * x)]G - [e * x]G = [k]G$$
3. Verification succeeds iff
$$([s]G - [e]P)_e = = R_e$$

2.2.1.3 Multi Signature

In the multi signature scenario, commitments and signatures are computed in a collective way. Exactly n=IScribe::bar() many feeds need to follow the steps to generate a collective signature:

1. All the parties generate the secure nonce k_i as before.
2. All parties generate and share the individual *commitment* R_i as:

$$R_i = [k_i]G$$

3. The aggregated *commitment* is computed as:

$$R_e = (\sum_{i=0}^{n} R_i)_e$$

4. The *challenge* binds the aggregated public key P, message, and commitment as:

$$e = H(P_x \,||\, P_p \,||\, m \,||\, R_e)$$

5. The *signature* is computed as:

$$s_i = k_i + (e * x_i) \bmod Q$$

After collecting all the signatures, one can verify it by reconstructing the *commitment* as:

$$[\sum_{i=0}^{n} s_i]G - [e]\sum_{i=0}^{n} P_i = [\sum_{i=0}^{n} k_i]G$$

and verification succeeds iff

$$([\sum_{i=0}^{n} k_i]G)_e = = R_e$$

Note that a collective schnorr signature of n parties is constructed via n public keys and n individual signatures, and it can only be verified with all the public keys and signatures together. Any k < n shares of it cannot be verified. In a nutshell, if there is a collective signature signed by n > `bar` parties, this collective signature as well as any bar-subset of it cannot be a valid signature. Only the collective signatures generated with exactly `bar` parties can be accepted and verified by the contract.

2.2.2 Scribe

Scribe is an efficient Schnorr multi-signature based oracle. It stores a tuple `(value, age)` which represents the current value and the timestamp when the value is accepted, and exposes several interfaces for authenticated read of this tuple. The `(value, age)` tuple will only be updated if it is submitted together with a valid schnorr signature.

Wards are roles governing Scribe (initialized with msg.sender) with the privilege to call:

- `rely()` - adds a new address to wards.
- `deny()` - removes an existing address from wards.
- `kiss()` - gives reading access to an address by adding it to _buds.
- `diss()` - revokes reading access of an address by removing it from _buds.
- `lift()` - adds a new address to feeds.
- `drop()` - removes an existing address from feeds.
- `setBar()` - sets the exact amount of signature required for verification.

During `lift()`, a valid signature must be provided and verified to prove the ownership of the corresponding private key. This eliminates key cancellation attacks.

`poke()` is the main permissionless entry point of the contract, where a user can submit a PokeData encoded as `(uint128 val, uint32 age)` with corresponding SchnorrData as its endorsement. `poke_optimized_7136211()` is a gas efficient alias for `poke()`. The following steps are conducted in `_poke()`:

- `pokeData.age` is compared with the current `_pokeData.age` to enforce the age between the last age and the current block.timestamp.
- `constructPokeMessage()` is called, where the poke message is constructed as the hash of the ethereum signed message header with the hashed pokeData.
- `_verifySchnorrSignature()` is called to verify the schnorrData.

SchnorrData encodes the following fields:

- `signature (bytes32)` - aggregated schnorr signature.
- `commitment (address)` - ethereum address of the aggregated committed point.
- `signersBlob (bytes)` - index of signers' public keys that participate in this signature.

`_verifySchnorrSignature()` first checks `signersBlob.length` is exactly equal to the bar. Then, the public key (a point on elliptic curve with Affine coordinates) of each signersBlob index will be loaded. The points must be non-zero and their corresponding ethereum addresses must be strictly monotonically increasing to avoid double signing. Afterwards, the Affine coordinates will be converted to Jacobian coordinates and aggregated. And the aggregated public key will be used to verify the signature following the schnorr signature verification.

Compared to the points addition in Affine coordinates where mod inverse is computed in every iteration, points addition in Jacobian coordinates only requires one mod inverse for the final aggregation, which reduces the gas cost.

When a pokeData is successfully accepted, its age is set to current block.timestamp. This necessarily limits the maximum update frequency to be once per block, given the freshness check of pokeData. In addition, as the number of feeds can be larger than the bar, feeds can form small subsets that compete with each other to take the only one chance of `poke()` in each block.

The following view functions are provided in Scribe for authenticated reading of the value and age:

- `read()` - returns `_pokeData.val`, and reverts if it is zero.
- `tryRead()` - returns `_pokeData.val` and a boolean flag indicating if it is zero.
- `read()` - returns `_pokeData.val` and `_pokeData.age`, and reverts if val is zero.
- `tryReadWithAge()` - returns `_pokeData.val` and `_pokeData.age`, together with a flag indicating if val is zero.
- `peek()` - same as `tryRead()`, whereas return values are in an reverse order.
- `latestRoundData()` - reading entrypoint for Chainlink compatibility, where `roundId` and `answeredInRound` are always 1. `startedAt` is always 0. `answer` and `updatedAt` are `int(uint(_pokeData.val))` and `_pokeData.age` respectively.

The following view functions are also provided to retrieve the feeds, wards and buds:

- `feeds(address)` - returns the index of an address and a boolean flag indicating if it is 0.
- `feeds(uint)` - returns the address of a public key at a given index and if it is zero.
- `feeds()` - returns two arrays of the existing feeds addresses and indices.
- `authed(address)` - returns if the address is a ward.
- `authed()` - returns an array of existing wards. Note that there could be duplicates address in the returned array if one ward is removed and added back later.
- `wards(address)` - returns an uint indicating if the address is a ward.
- `tolled(address)` - returns wether the address is in buds.
- `tolled()` - returns an array of existing buds, which could include duplicates as `authed()`.
- `bud(address)` - returns an uint indicating if the address is in buds.

2.2.3 Scribe Optimistic

ScribeOptimistic is an optimistic extension of Scribe which, in addition to the original `poke` function to push an update, features the option to optimistically poke an update without performing the gas-intensive signature verification immediately. Within the following challenge period anyone can come to challenge the optimistic poke.

`opPoke()` is the main permissionless entry point of the contract, where a user can submit a PokeData, a SchnorrData for verification, and an ECDSAData showing the endorsement of a feed. `opPoke_optimized_397084999()` is a gas efficient alias for `opPoke()`. The following steps are conducted in `_opPoke()`:

- The last opPoke data is checked, and a new opPoke can only happen after an old opPokeData has been finalized.
- `opPokeData.age` is compared with the `_pokeData.age` or the last finalized `_opPokeData.age` to enforce the age between the last age and the current block.timestamp.
- The ECDSA signature is verified, and this endorsement should come from an existing feed, whose index will be stored for the challenge.
- The submitted schnorrData is hashed and stored for the challenge.
- The new opPokeData is stored for the challenge.

During the challenge period, anyone can challenge a opPokeData by `opChallenge()` with the same schnorrData:

- The current block.timestamp is checked to ensure that the challenge period has not elapsed.
- The submitted schnorrData is hashed and compared with the stored hash.
- `_verifySchnorrSignature()` is called to verify the signature.
- In case the signature verification succeeds, the opPokeData becomes valid.
- In case the signature verification fails, the feed who endorses the opPoke will be dropped and a reward (in ether) will be paid to the challenger, which is bounded by the `maxChallengeReward`.

The functions of reading val and age are overridden to return the opPokeData in case it is finalized and its age is larger than `pokeData.age`.

`constructOpPokeMessage()` is provided to bind the opPokeData with a schnorrData for generation and verification of the endorsement ECDSA signature.

The following permissioned functions are provided in addition to the ones in Scribe, which can only be called by the wards:

- `setOpChallengePeriod()` - sets the optimistic challenge period.
- `setMaxChallengeReward()` - sets the max challenge reward.

In case there is an updates of bar or challenge period or dropping a feed, `_afterAuthedAction()` will be called to drop the existing un-finalized opPokeData. Besides, the age of the fresher one between `_opPokeData` and `_pokeData` will be updated to the current block.timestamp. `_setBar()` is overridden to include this feature.

Consequently, the wards indirectly obtain the privilege to:

- Delay or advance the opPoke finality time by `setOpChallengePeriod()`.
- Drop a opPokeData (i.e. by setting the challenge period back and forth).

2.2.4 Libs

The following libs are implemented for the schnorr signature scheme.

2.2.4.1 LibBytes

`getByteAtIndex()` is provided to retrieve one byte at a certain index inside of a `uint256`. This will be used to unpack the indices in the `schnorrData.signersBlob`.

2.2.4.2 LibSchnorrData

`LibSchnorrData` performs efficient loading of an index at `schnorrData.signersBlob`.

2.2.4.3 LibSecp256k1

`LibSecp256k1` contains the constant parameters that define the elliptic curve secp256k1: the coefficients of a and b, the finite field p, the group generator `G`, and the group order `Q`.

It defines the an Affine point as `(uint x,uint y)`, a Jacobian point as `(uint X,uint Y,uint Z)`, and the conversions between them where the following equations hold. The mod inverse operation is implemented as per extended Euclidean algorithm

$$x = \frac{X}{Z^2} \quad , \quad y = \frac{Y}{Z^3}$$

It also implements an optimized elliptic curve points addition on the Jacobian coordinate, which is more efficient compared to a naive approach as no mod inverse is introduced.

2.2.4.4 LibSchnorr

`LibSchnorr` implements the functionalities of `verifySignature()` by leveraging the ecrecover pre-compile to save gas.

2.2.5 Roles and Trust Model

Wards are the privileged roles governing the contracts. They are fully trusted to set all parameters and feeds honestly and correctly, and never interfere with the legitimate updates of the price.

Feeds are authenticated identities that can sign price in a collective way. We assume:

- Only the legitimate and honest feeds will be used. Otherwise, `bar` out of n feeds can generate valid signature and update the price at their discretion.
- The feeds actively sign and update the prices in an honest way without collusion.
- The `bar` is set properly. A small `bar` may decrease the difficulty of collusion between feeds, while a large `bar` may induce excessively high gas cost at verification thus breaks the liveness of Scribe and optimistic Scribe.
- During the life span of a scribe oracle in use, the total lifted feeds should not be below `bar`. Otherwise, no valid signatures can be generated and the oracle's liveness is broken.
- The challenge period is expected to be set in a way that inaccurate updates can always be challenged in time.
- Feeds' secret keys are not leaked and feeds generate nonces correctly and securely. Otherwise, any bias in nonce generation may result to the compromise of secret keys.

We also assume the correctness of the points addition formula in Jacobian coordinates, and there is no efficient way to solve discrete logarithm on secp256k1.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Assessement of Finalized After Authed Action Risk Accepted

Low-Severity Findings	3

- Drop of opPoke if Authed Action Does Not Update Anything Risk Accepted
- Lift Does Not Drop Unfinalized opPoke Risk Accepted
- Race of Feeds Risk Accepted

5.1 Assessement of Finalized After Authed Action

Correctness Medium Version 1 Risk Accepted

CS-CSC-001

In ScribeOptimistic, `_afterAuthedAction()` will be called after every change to the parameters to drop `opPokeData` if it's not yet finalized. The evaluation whether the optimistic poke is finalized is done using the possibly new value for `opChallengePeriod` however.

This has the following consequences:

- In case the new challenge period is bigger, `_afterAuthedAction` will evaluate the finality based on the increased challenge period. Consequently an already finalized opPokeData becomes challengeable again and may be dropped.
- In case the new challenge period is smaller, `_afterAuthedAction` will evaluate the finality based on the decreased challenge period. Consequently an previously un-finalized opPokeData will be regarded finalized immediately.

Risk accepted:

Chronicle has accepted the risk of finality reevaluation and states:

```
The "reevaluation of finality" based on a possibly updated opChallengePeriod
is accepted. We plan to update the challenge period in the beginning a few times
to find "the best" reasonable value. Afterwards, we don't intend to update the
challenge period anymore following a "never stop a running system" approach.
```

Furthermore, the code has been adjusted that in case a finalized `opPokeData` is more recent than the `_pokeData`, it will also be pushed into `_pokeData` and deleted. This avoids resetting the challenge period for a finalized opPokeData. Chronicle states:

```
_afterAuthedAction has been updated to ensure the challenge period of an opPoke
is not reset, which would decrease the possible update frequency via opPoke ().
This is achieved via either moving _opPokeData to _pokeData storage if _opPokeData
is finalized and newer than _pokeData, or deleting opPokeData otherwise.
```

5.2 Drop of opPoke if Authed Action Does Not Update Anything

Design Low Version 1 Risk Accepted

CS-CSC-002

`setOpChallengePeriod()`, `setMaxChallengeReward()`, and `setBar()` will always call `_afterAuthedAction()` to drop unfinalized opPokeData, even if the updated parameter is the same as the old parameter.

Risk accepted:

Chronicle states:

```
Skipping the afterAuthedAction in special cases changes the definition of the
action, which is defined to be called after every authe'd configuration change.
```

5.3 Lift Does Not Drop Unfinalized opPoke

Correctness Low Version 1 Risk Accepted

CS-CSC-003

ScribeOptimistic generally drops unfinalized optimistic poke data after the update of parameters to avoid any issues connected to an unexpected change of the verification result.

`_lift()` is not overridden in ScribeOptimistic to call `_afterAuthedAction()` which drops the unfinalized opPokeData. This may allow not yet but soon to be feeds to sign the price update.

Assume Alice is not a member of the current feeds at t_0 and $t_0 < t_1 < t_2$.

- At t_0, Alice signs a price with other `bar-1` feeds, and `opPoke()` it.
- At t_1, wards add Alice to the feeds.
- At t_2, one comes to challenge the opPokeData, the challenge fails (verification succeeds) and the pokeData becomes valid.

In this example, Alice's signed data successfully passes the verification, though Alice has not been authorized at t_0, the time the price data was aggregated.

Risk accepted:

Chronicle states:

```
This is a valid issue from a theoretical point of view. However, practically we
don't see any problems arising through this.
```

5.4 Race of Feeds

Security Low Version 1 Risk Accepted

CS-CSC-004

Given the gas and runtime limitation, `bar` shouldn't be too large, whereas there could be 254 (`maxFeeds`) feeds at most. In case feeds' amount is larger than `bar`, feeds may form different subsets and sign with different views of the current price. As the price update frequency is at most once per block (limited by the freshness check of the `pokeData.age`), there could be a race case among the feeds.

Risk accepted:

Chronicle states:

```
The bar-to-feed ratio will be set conservatively, i.e. "far more" feeds than bar.
While the relationship will always be that bar > #feeds/2 to ensure that only a
consensus of >50% can advance the oracle to a new price, we want to have more
feeds than bar to not risk downtime due to feeds being dropped.

The current configuration is: 22 feeds with a bar of 13.
```

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	1
Medium-Severity Findings	1
Low-Severity Findings	2

- Update Using Stale PokeData Code Corrected
- Unreset opPokeData After Unsuccessful Challenge Code Corrected
- Incorrect Formula in Docs Specification Changed
- Indexed Fields of Event Poked Code Corrected

6.1 Update Using Stale PokeData

Correctness High Version 1 Code Corrected

CS-CSC-011

In ScribeOptimistic an optimistic poke is considered finalized after the challenge period elapsed. If the optimistic poke (`opPokeData.age`) is finalized and more recent than the last stored poke data (`pokeData.age`), the pricefeed returns the optimistic value:

```
function _currentPokeData() internal view returns (PokeData memory) {
    // Load pokeData slots from storage.
    PokeData memory pokeData = _pokeData;
    PokeData memory opPokeData = _opPokeData;

    // Decide whether _opPokeData is finalized.
    bool opPokeDataFinalized =
        opPokeData.age + opChallengePeriod <= uint32(block.timestamp);

    // Decide and return current pokeData.
    if (opPokeDataFinalized && opPokeData.age > pokeData.age) {
        return opPokeData;
    } else {
        return pokeData;
    }
}
```

`Scribe.poke()`, the function to update the pricefeeds `_pokeData`, only checks whether the new value is more recent than the stored data. It does not check whether there is a more recent finalized optimistic poke:

```
// Revert if pokeData stale.
 if (pokeData.age <= _pokeData.age) {
```

```
        revert StaleMessage(pokeData.age, _pokeData.age);
    }
```

This update is then stored with the current `block.timestamp` as age:

```
// Store pokeData's val in _pokeData storage and set its age to now.
_pokeData.val = pokeData.val;
_pokeData.age = uint32(block.timestamp);
```

Resulting a stale `pokeData` could be used to update the pricefeed.

This issue arises in the state (_opPokeDate: finalized, _pokeData: set-older or uninitialized)

Code corrected:

`_poke()` has been marked as virtual in Scribe and overridden in ScribeOptimistic, where it checks the stored `_pokeData` and the recent finalized optimistic poke to determine the most recent age. Thus it prevents a stale `pokeData` to update the pricefeed.

6.2 Unreset opPokeData After Unsuccessful Challenge

Design | Medium | Version 1 | Code Corrected

CS-CSC-009

In ScribeOptimistic, in an unsuccessful challenge (successful signature verification), `opPokeData` is finalized and pushed to `_pokeData`. `_opPokeData` remains unchanged however. This will block `opPoke()` even though the current opPokeData has been finalized until the challenge period is over.

```
if (ok) {
        // Decide whether _opPokeData stale already.
        bool opPokeDataStale = opPokeData.age <= _pokeData.age;

        // If _opPokeData not stale, finalize it by moving it to the
        // _pokeData storage.
        if (!opPokeDataStale) {
            _pokeData = _opPokeData;
        }

        emit OpPokeChallengedUnsuccessfully(msg.sender);
}
```

Code corrected:

`_opPokeData` will be deleted in case it is verified successfully and is more fresh than the `_pokeData`. Thus, this finalized `_opPokeData` will not block a new `opPoke()` anymore.

6.3 Incorrect Formula in Docs

Correctness | Low | Version 1 | Specification Changed

CS-CSC-012

In docs/Schnorr.md, the formula of re-computing challenge in signature verification is incorrect compared to the ones in signing and in code implementation. The order of the last two parameters is wrong.

$$e = H(P_x \,||\, P_p \,||\, R_e \,||\, m) \bmod Q$$

Specification changed:

The signature verification formula in docs/Schnorr.md has been corrected to align with the signing formula and the code implementation.

6.4 Indexed Fields of Event Poked

Design | Low | Version 1 | Code Corrected

CS-CSC-008

```
/// @notice Emitted when oracle was successfully poked.
/// @param caller The caller's address.
/// @param val The value poked.
/// @param age The age of the value poked.
event Poked(address indexed caller, uint128 val, uint32 age);
```

Indexing fields in events allows to easily search for certain events. The `val` and `age` of the event above are not indexed. Indexing e.g. the age field would allow off chain observers to easily search for prices in the past.

Code corrected:

The `val` and `age` of the event have been marked as indexed.

6.5 Gas Optimizations

Informational | Version 1 | Code Corrected

CS-CSC-010

- `getSignerIndexLength()` could load the length by assembly.
- In `_verifySchnorrSignature()`, `lift()`, and `drop()` the counter `i` inside of the for loop can be increased in an unchecked scope, as it is always bounded.
- There is no amount limitation of inputs for `abi.encodePacked()`, thus one invocation should suffice to pack all the parameters in `constructPokeMessage()` and `_constructOpPokeMessage()`.
- In `_verifySchnorrSignature()`, loading a public key at an index can be abstracted into another internal function to decrease code duplication.
- In `_lift()`, the require statement `index <= maxFeeds` can be moved into the if branch, as we only need to check the number of feeds when a new public key is added.

- The first condition check (`opPokeDataFinalized`) can be removed in `_opPoke()`, as the function would already revert if it is false.

```
if (!opPokeDataFinalized) {
    revert InChallengePeriod();
}

uint32 age = opPokeDataFinalized && opPokeData.age > _pokeData.age
        ? opPokeData.age : _pokeData.age;
```

- In LibSchnorr, the following line can be wrapped in an unchecked scope.

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

- In `addAffinePoint()`, some intermediate results can be cached to avoid computing repeatedly. For example:

```
uint left = mulmod(addmod(z1, h, _P), addmod(z1, h, _P), _P);

uint v = mulmod(x1, mulmod(4, mulmod(h, h, _P), _P), _P);

uint j = mulmod(4, mulmod(h, mulmod(h, h, _P), _P), _P);
```

In addition, the following optimizations only work if the external view functions are called by a smart contract.

- In `feeds()`, the for loop counter `i` can start from 1 as the public key at index 0 is an zero point. And `i` can be increased in an unchecked scope.
- In `feeds(uint index)`, the input index can be checked towards 0 for early revert. And the public key at a specific index is not loaded by assembly as before.

Code corrected:

Code has been corrected to adopt some of the optimizations.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Authed and Tolled May Return Array With Duplicates

Informational **Version 1**

CS-CSC-005

`authed()` will return the existing wards addresses as an array. Upon `rely()` the address is inserted into the mapping and the array. Upon `deny()` the address is only reset in the mapping and not removed from the array. Consequently, in case an address is added, then removed, and later added back, `authed()` will return an array that contains a duplicate of this address. The same applies to `tolled()`.

7.2 Timestamp of PokeData Aggregation Equal to Block.Timestamp

Informational **Version 1** **Acknowledged**

CS-CSC-006

`Scribe._poke()` rejects PokeData with timestamps in the future but accepts PokeData with `block.timestamp`.

```
// Revert if pokeData from the future.
if (pokeData.age > uint32(block.timestamp)) {
    revert FutureMessage(pokeData.age, uint32(block.timestamp));
}
```

It's a theoretical observation only with no impact in practice, but aggregating the price data and updating the data on chain seems infeasible. In practice when `_poke()` is executed it should hold `block.timestamp > pokeData.age` or the aggregation of the price data off chain likely happened for a timestamp (slightly) in the future.

Acknowledged:

Chronicle has acknowledged this theoretical observation with no impact in practice.

7.3 Undropped schnorrData Commitment and opFeedIndex

Informational **Version 1** **Acknowledged**

CS-CSC-007

`_schnorrDataCommitment` stores the schnorr data digest of an optimistically poked data, ensuring it will be challenged by the same schnorr signature later. `opFeedIndex` stores the feed who signs to endorse the opPokeData and the schnorr signature. Upon a successful challenge, these variables are not deleted consistently with the `_opPokeData`. However, as opPokeDataFinalized is computed by the following statement and `opChallengePeriod` is at most `max(uint16)`, it would always be true after `_opPokeData` is deleted. Consequently double challenging an already dropped `_opPokeData` is not possible.

```
bool opPokeDataFinalized =
        opPokeData.age + opChallengePeriod <= uint32(block.timestamp);
```

Acknowledged:

Chronicle states:

> You are correct in that we could drop the schnorrDataCommitment and opFeed Index. However, doing so will increase the costs of the subsequent opPoke as writing to zero-storage is more expensive than overwriting non-zero storage. Note furthermore, that the gas-stipend for emptying the storage is attributed to the opChallenge caller, i.e. a searcher.
>
> Therefore, cleaning the storage would practically give external entities a gas stipend that relays have to pay during the next opPoke().

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Considerations of pokeData.age

Note Version 1

`pokeData.age` is interpreted differently at different stages.

- User submitted pokeData.age: it refers to the time it is generated offchain for freshness check and signature verification.
- Onchain stored pokeData.age: it refers to the time the offchain generated data is poked or opPoked onchain.

In ScribeOptimistic, there could be a potential delay to read the most recent valid opPokeData.age, as it only takes effect after being finalized.

In addition, systems integrate the ScribeOptimistic should aware that an old pokeData could be carried over to the current time (by updating the age to current block.timestamp) in following cases:

- A successful challenge (a failed verification) will drop the current opPokeData and advance the existing `_pokeData.age` to current block.timestamp.
- An update of the oracle parameters (`setBar()`, `setOpChallengePeriod()`, `drop()`) by the wards will always advance the valid most fresh `pokeData.age` to current block.timestamp.

8.2 Decimals of Price Feed

Note Version 1

Scribe oracles use 18 decimals for price values. Besides implementing own interfaces to read the price, the Chainlink interface is implemented to serve potential customer already integrating with Chainlink. Note that Chainlink usually uses 18 decimals for ETH denominated assets but 8 decimals for USD denominated assets. Hence projects need to be careful especially when switching from Chainlink USD based pricefeeds to Scribe.

8.3 Implications Regarding the Value of Bar

Note Version 1

`setBar()` incorporates a check to restrict the `bar` value from being set to zero. However, the upper boundary is only limited by the uint8 datatype, which is 255.

- The system can accommodate a maximum of 254 feeds (assuming no feed is ever removed). Therefore, if the bar is set to 255, the processing of any poke becomes unfeasible.
- Once the `bar` value crosses a certain threshold, verifying the aggregated signature may surpass the block gas limit, thereby rendering on-chain verification impossible.
- Dropping feeds (e.g. directly or in ScribeOptimistic after a successful challenge) may result in the number of feeds remaining being insufficient to cover bar.

The privileged role is expected to set the value for `bar` correctly.

8.4 Max 254 Feeds Over the Contracts Lifetime

Note Version 1

By design, a Scribe Pricefeed can have a maximum of 254 feeds added: Adding a new pricefeed pushes the public key into the `_pubKeys` array. Removing a price feed resets the pricefeeds public key entry to the zero point, however this does not free up the space. Feeds IDs are hence never reused but the tradeoff is the maximum number of possible feeds that can be added.

8.5 Penalty to Rogue Feeds

Note Version 1

If a feed endorses an invalid schnorrData / pokeData combination in ScribeOptimistic, the feed will be dropped when challenged. In addition to dropping the feed, there is no more penalty to the feeds on the contract level.

Chronicle states:

```
Any kind of penalty regarding misbehaving feeds will be handled on the social
layer.
```

8.6 Price Change

Note Version 1

Systems that integrate with Scribe or ScribeOptimistic should be aware of the possible ways that can change the price (`pokeData.val`).

Below we list occasions where the price can change immediately as well as some potential ways how this could be leveraged (i.e. bundled):

- Authenticated (in Scribe and ScribeOptimistic): `_poke()` invoked with a valid new `pokeData` and schnorr signature from feeds. The current price will advance to a new price.
- Wards (only in ScribeOptimistic): `_afterAuthedAction()` invoked by the wards which drops a previously finalized `opPokeData` according to the new challenge period. In case this dropped `opPokeData` is the freshest one, the current price will rollback to the `_pokeData`.
- Permissionless (only in ScribeOptimistic): `onChallenge()` which finalizes a valid fresh `opPokeData` and pushes it to `_pokeData`. The current price will advance to the new price (`opPokeData`). Similarly, after an optimistic poke, anyone could execute `poke()` using this signed data to advance the price immediately.

When integrating with ScribeOptimistic projects must be aware that an optimistic price update is generally but not always subject to the challenge period delay: anyone may finalize it at any point during the challenge period.