Chronicle

Security Review

Cantina Managed review by:

Christoph Michel, Lead Security Researcher

M4rio.eth, Security Researcher

Shung, Associate Security Researcher

November 27, 2023

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 Critical Risk . . . 4
  - 3.1.1 Valid `opPoke` Schnorr signatures can be challenged by using custom calldata . . . 4
- 3.2 Medium Risk . . . 5
  - 3.2.1 Unbounded `schnorrData` in `opPoke` . . . 5
- 3.3 Low Risk . . . 6
  - 3.3.1 Non-cleaned up variables in inline assembly . . . 6
  - 3.3.2 Newly deployed Scribe instances can receive old data as first value . . . 6
  - 3.3.3 The `age` of the pokeData is updated once accepted . . . 6
- 3.4 Gas Optimization . . . 7
  - 3.4.1 `drop` optimization in `opChallenge` . . . 7
- 3.5 Informational . . . 7
  - 3.5.1 Low level read and write access to storage . . . 7
  - 3.5.2 Unused imports . . . 8

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

Chronicle Protocol is a blockchain-agnostic layer for securely and verifiably transitioning data onchain.

From Oct 27th to Nov 2nd the Cantina team conducted a review of scribe on commit hash c508c23d. The team identified a total of **8** issues in the following risk categories:

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

3 Findings

3.1 Critical Risk

3.1.1 Valid `opPoke` Schnorr signatures can be challenged by using custom calldata

Severity: Critical Risk

Context: LibSchnorrData.sol#L79, ScribeOptimistic.sol#L177

Description: The `LibSchnorrData.loadFeedId(IScribe.SchnorrData calldata schnorrData, uint8 index)` function tries to read `schnorrData.feedIds[index]` using assembly but it uses a hardcoded calldata offset of `0x80` for the `feedIds`. Calling a function on the contract with a custom calldata encoding using a different offset is a valid encoding but will lead to this function reading the wrong `feedIds` according to the calldata spec:

```
assembly ("memory-safe") {
    // Calldata index for schnorrData.feedIds[0] is schnorrData's offset
    // plus 4 words, i.e. 0x80.
    let feedIdsOffset := add(schnorrData, 0x80)

    // Note that reading non-existing calldata returns zero.
    word := calldataload(add(feedIdsOffset, wordIndex))
}
```

The `ScribeOptimistic._opPoke` function uses Solidity to read the `schnorrData.feedIds` and therefore abides by the calldata spec and reads the `feedIds` that the offset specified in the calldata actually points to:

```
_schnorrDataCommitment = uint160(
    uint(
        keccak256(
            abi.encodePacked(
                schnorrData.signature,
                schnorrData.commitment,
                schnorrData.feedIds
            )
        )
    )
);
```

It's possible to create a custom calldata encoding such that `_opPoke` reads one `feedIds` array and `LibSchnorrData.loadFeedId` (used in verifying the Schnorr signature) reads a different `feedIds` array. This allows an attacker to pass all checks including the `_schnorrDataCommitment` check in `opChallenge` but have the verification of the Schnorr signature fail. Optimistic pokes by honest feeds can successfully be challenged, the challenge reward can be earned, and the feed will be removed. This can be repeated to remove honest feeds until there are fewer feeds than `bar`, putting the oracle to a halt and locking in the last seen value.

```
// add to test/IScribeOptimisticTest.sol
function test_calldataEncoding() public {
    LibFeed.Feed[] memory feeds = _liftFeeds(opScribe.bar());

    IScribe.PokeData memory pokeData;
    pokeData.val = 1;
    pokeData.age = 1;
    assertEq(opScribe.feeds().length, 2, "expected 2 feeds to start with");

    IScribe.SchnorrData memory schnorrData;
    schnorrData = feeds.signSchnorr(opScribe.constructPokeMessage(pokeData));

    IScribe.ECDSAData memory ecdsaData;
    ecdsaData = feeds[0].signECDSA(
        opScribe.constructOpPokeMessage(pokeData, schnorrData)
    );

    // Execute valid opPoke.
    opScribe.opPoke(pokeData, schnorrData, ecdsaData);

    // Challenge opPoke.
    // bytes memory defaultCalldata = abi.encodeCall(opScribe.opChallenge, schnorrData);
    // console2.logBytes(defaultCalldata);
    // 8928a1f8 // selector
    // 0000000000000000000000000000000000000000000000000000000000000020 // struct offset
    // fecd661c0731ca99672edb7303a072da3c2b8342e714c2f2a90639b966298958 // signature
    // 000000000000000000000000026428bf84a659a2d371be1e705613d89d93f78f // commitment
    // 0000000000000000000000000000000000000000000000000000000000000060 // offset(feedIds)
    // 0000000000000000000000000000000000000000000000000000000000000002 // length(feedIds)
    // 2b68000000000000000000000000000000000000000000000000000000000000 // feedIds
    bytes memory customCalldata = (
        hex"8928a1f8" // selector
        hex"0000000000000000000000000000000000000000000000000000000000000020" // struct offset
        hex"fecd661c0731ca99672edb7303a072da3c2b8342e714c2f2a90639b966298958" // signature
        hex"000000000000000000000000026428bf84a659a2d371be1e705613d89d93f78f" // commitment
        hex"00000000000000000000000000000000000000000000000000000000000000a0" // offset(feedIds)
        hex"0000000000000000000000000000000000000000000000000000000000000002" // injected(feedIds).length
        hex"2b2b000000000000000000000000000000000000000000000000000000000000" // injected(feedIds) (double
↪   sign)
        hex"0000000000000000000000000000000000000000000000000000000000000002" // length(feedIds)
        hex"2b68000000000000000000000000000000000000000000000000000000000000" // feedIds
    );

    (bool success, bytes memory retData) =
        address(opScribe).call(customCalldata);
    if (!success || retData.length != 32) revert();
    bool opPokeInvalid = abi.decode(retData, (bool));

    assertTrue(opPokeInvalid, "expected opChallenge to return true");
    assertEq(opScribe.feeds().length, 1, "expected feed to be dropped");
}
```

Recommendation: Consider using Solidity to read the `schnorrData.feedIds[index]` and `schnorrData.feedIds.length` in `LibSchnorrData.loadFeedId/numberFeeds` which automatically resolves the calldata offsets to the correct ones. Note that it's important that `_verifySchnorrSignature` should still not revert such that all `schnorrData` can be challenged.

Chronicle: Fixed in commit f074096.

Scribe's `_verifySignature()` function now reads the `calldata schnorrData` via Solidity's calldata read functionality. The calldata array access via `schnorrData.feedIds[i]` is guaranteed to not revert as its length is being checked to be sufficient beforehand. Any manipulation of the `schnorrData.feedIds.length` value is caught beforehand via the `schnorrDataCommitment` check.

This fix allowed the removal of the `LibSchnorrData` and `LibBytes` libraries.

3.2 Medium Risk

3.2.1 Unbounded `schnorrData` in `opPoke`

Severity: Medium Risk

Context: ScribeOptimistic.sol#L115

Description: Invalid optimistic pokes should always be challengeable and it is important that the `opChallenge` call challenging a previous `schnorrData` submitted through `opPoke` does not revert and the gas cost do not exceed the block gas limit. Otherwise, the invalid optimistic poke data will automatically be finalized after the challenge period.

The `opChallenge` function must copy the `schnorrData` to memory and hash it to compare it against the committed Schnorr hash. The `schnorrData` submitted via `opPoke` is not bounded in size as its `feedIds` array length can be arbitrary. A malicious feed can create an optimistic poke with a very large `feedIds` array that would cost more to challenge than the challenge reward, making it economically unprofitable for keepers to challenge it. In the worst case, challenging it might even exceed the block gas limit making it impossible to challenge even given enough incentive. (Note that `opPoke` might cost more gas than the `opChallenge` for large `schnorrData` as the former hashes `schnorrData.feedIds` twice, making the block gas limit attack vector potentially impossible to execute. We still recommend restricting the size.)

Recommendation: Consider restricting the size of `schnorrData.feedIds` in `opPoke`. It can already be checked that `schnorrData.feedIds.length == bar` at this point as all valid Schnorr signatures for the current configuration must have been signed by exactly `bar` feeds.

Chronicle: Fixed in commit 2bd72c2a.

The opPoke() function now checks that schnorrData.feedIds's length not greater then bar's max value. Note that this approach was chosen instead of checking equality with bar to prevent another cold storage read.

3.3 Low Risk

3.3.1 Non-cleaned up variables in inline assembly

Severity: Low Risk

Context: Scribe.sol#L475, Scribe.sol#L501

Description: index, a uint8 variable, is used in _sloadPubKey() and _sstorePubKey() functions inside inline assembly blocks. This variable is used without any clean up. Although there is no issues in the current versions of the code, if these functions are used in other places without care, bugs might be introduced.

Recommendation: Always clean up non-full words used in inline assembly. In this case, instances of index in inline assembly can be replaced with and(index, 0xff).

Chronicle: Issue resolved via commit d8ac6732.

3.3.2 Newly deployed Scribe instances can receive old data as first value

Severity: Low Risk

Context: Scribe.sol#L90-L120

Description: When a new instance is deployed, arbitrarily old data for the same wat (for example from a different chain) can be submitted and will be the oracle's value until a new one is produced and submitted. From the functionality point of view, the old data is valid as the signature is valid and it is not time-dependent.

Recommendation: Consider having a check that the data submitted is not too old. As this is a very opinionated issue, we can not recommend what too old might mean. It will really depend on the type of data.

Chronicle: Acknowledged. While we agree with the validity of the issue we don't see practical relevance. Internal processes ensure that users/integrators are only directed to newly deployed Scribe(Optimistic) instances once they got poked for a sufficient amount of time. The timeframe here is of at least multiple days which translates to many pokes.

3.3.3 The age of the pokeData is updated once accepted

Severity: Low Risk

Context: Scribe.sol#L117

Description: When data is created offchain, an age is defined which represents the timestamp of the data creation. Once that data reaches the chain via the poke function, the age is updated to block.timestamp, invalidating the initial value.

This can cause issues in a scenario of high volatility markets and high onchain usage. In such cases, it can happen that data is waiting in the mempool enough time so a second data arrives. If the initial data reaches the block first, then the second value, which should be the most updated value will be discarded due to the fact that on Scribe.sol#L117 the update on age is done:

```
_pokeData.age = uint32(block.timestamp);
```

and for the second call the check on Scribe.sol#L95 will make the transaction to revert:

```
if (pokeData.age <= _pokeData.age) {
    revert StaleMessage(pokeData.age, _pokeData.age);
}
```

The same scenario happens on the ScribeOptimisc contract, with the optimistic pokes.

Recommendation: Consider keeping the integrity of the data that is submitted onchain by not updating the `age` with the current timestamp.

Chronicle: Acknowledged. While we think there are valid arguments for both types of definitions for a value's age, the simple reason we go with this one is due to backwards compatibility reasons with regards to MakerDAO's `Median` contract. However, we will review our documentation to ensure this behavior is sufficiently pointed out to users/integrators.

3.4 Gas Optimization

3.4.1 `drop` optimization in `opChallenge`

Severity: Gas Optimization

Context: ScribeOptimistic.sol#L262, ScribeOptimistic.sol#L476

Description: The overridden `drop` function is called in `opChallenge` which executes the generic `_afterAuthedAction` hook to reset some storage fields. However, in `opChallenge` one already knows that the `opPoke` is not finalized yet and some optimizations can be performed compared to the generic case of a configuration parameter changing.

Recommendation: Consider changing:

```
// Drop opFeed and delete invalid _opPokeData.
// Note to use address(this) as caller to indicate self-governed
// drop of feed.
- _drop(address(this), opFeedId);
+ super._drop(address(this), opFeedId);
+ emit OpPokeDataDropped(msg.sender, _opPokeData);
+ delete _opPokeData;
```

Chronicle: Acknowledged. Keeping as is in favor of code clarity. Note that gas consumption of the `opChallenge()` function is of no big concern.

3.5 Informational

3.5.1 Low level read and write access to storage

Severity: Informational

Context: Scribe.sol#L468-L509

Description: In `Scribe.sol`, the `sloadPubkey()` and `sstorePubKey()` functions are used to read and write to storage respectively using inline assembly. It is not advisable to access storage variables in this low-level manner.

Recommendation: Consider using `_pubKeys[n]` to read and write to `_pubKeys` array.

Chronicle: Prior to using the statically allocated array the read function "had" to be low-level in order to circumvent index-out-of-bounds reverts while saving the bounds check (if out-of-bounds `pubKey` is zero and we would catch it during aggregation).

Fixed in commit d8ac6732, the `_pubKeys` array is now accessed (and written to) via standard Solidity semantics instead of assembly. Index out of bounds reverts are impossible as every access happens via an index of type `uint8`. Note that `_pubKeys` has a statically allocated max index of `type(uint8).max`.

This fix allowed the removal of the two assembly-based functions `_sloadPubKey()` and `_sstorePubKey()`.

3.5.2 Unused imports

Severity: Informational

Context: Scribe.sol#L4, Scribe.sol#L6, ScribeOptimistic.sol#L4, ScribeOptimistic.sol#L8

Description: There are unused imports in `Scribe.sol` and `ScribeOptimistic.sol`. These imports are already inherited through other contracts or exist for testing purposes.

Recommendation: Consider removing unused imports.

Chronicle: Fixed in commit 16055bf2.

Note that we keep the redundant `IChronicle` and `IScribe` imports to allow using Solidity's NatSpec (`@inheritdoc <interface>`) for these interfaces. This helps understanding in which interface from the inheritance tree the respective function is defined in.