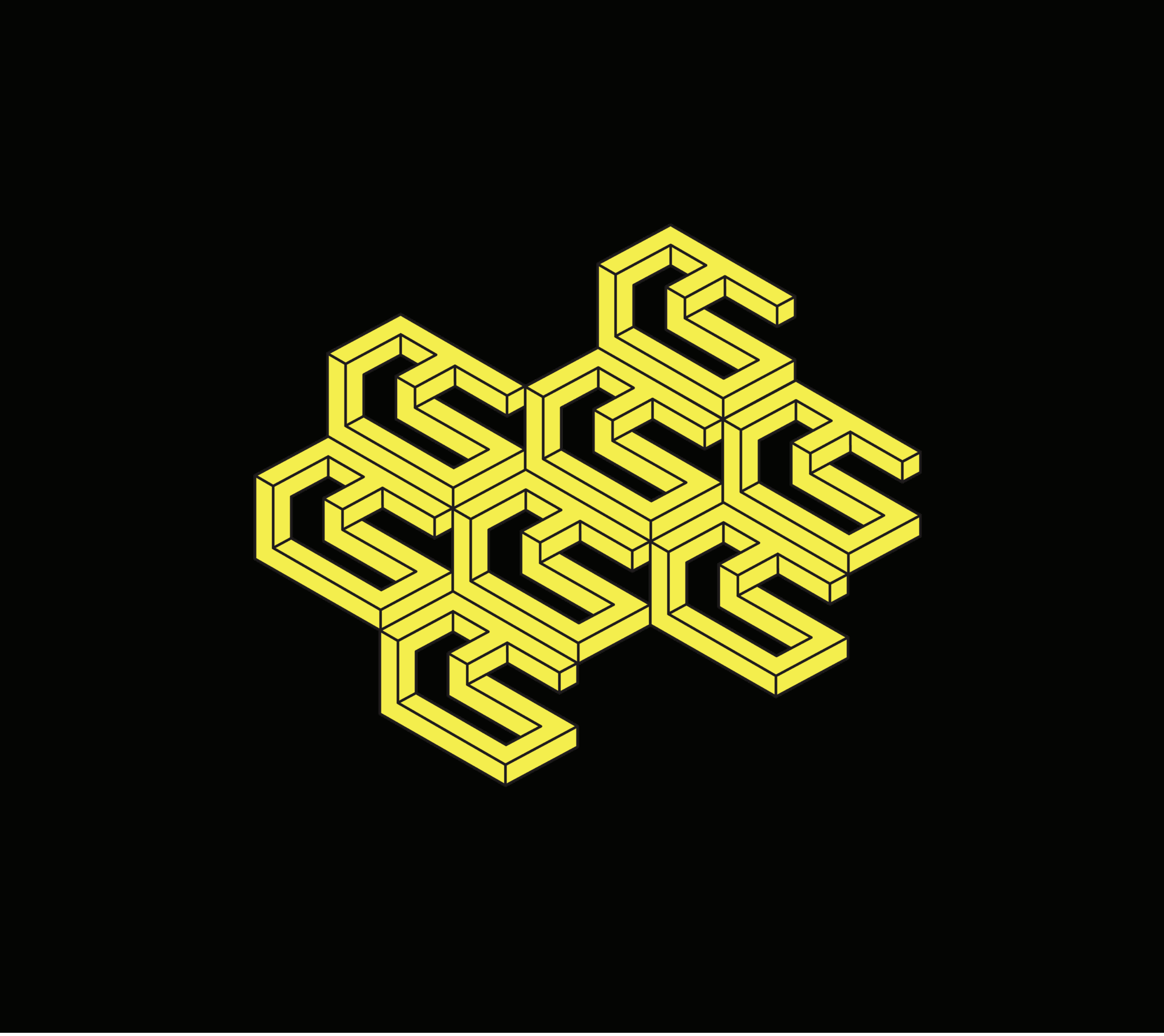

Connext Labs
Vector Smart Contracts Review

by ChainSafe Systems

December 2020

Connext Labs Vector Smart Contracts Review

Auditor: Oleksii Matiiasevych

WARRANTY

1. Introduction

Connext Labs Inc requested ChainSafe Systems to perform a review of the Vector state channel contracts. The contracts in question can be identified by the following git commit hash:

```
f3d72fe0a4f2877bd0453ff8d4161129c459ad83 vector-0.0.17
```

There are 31 contracts/libraries/interfaces in scope.

2. Disclaimer

The review makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts for any specific purpose, or their bug free status. The review documentation below is for internal management discussion purposes only and should not be used or relied upon by external parties without the express written consent of ChainSafe Systems.

3. Executive Summary

There were 0 critical, 0 major, 4 minor, 16 informational/optimizational issues identified in this version of the contracts. There are no known compiler bugs, for the specified compiler version (0.7.1), that might affect the contract's logic. Operation of the Vector protocol heavily depends on the correctness of the off-chain nodes implementation, which was not reviewed during the course of this engagement. I like the quality of the code, which I enjoyed reviewing.

4. Observations and Possible Improvements

4.1 Every new asset on the channel adds 3196 gas to the dispute initiation. It should be possible to fit up to 3700 different assets into the channel state assuming the 12M block gas limit. In case the block gas limit drops to 500K (as it was for a number of days in October 2016), any channel that has more than 110 assets in the state will not be able to initiate a dispute until the block gas limit is eventually raised.

One way to lift this limitation, is to have a separate state for every asset. This will also make the majority of disputes cheaper.

Another way to mitigate this limitation is to have an off-chain circuit breaker linked to the current number of assets in the channel and current block gas limit. In the case that block gas limit comes dangerously close to the one needed for the dispute, then defund the channel immediately.

4.2 When signing a new `transferId` make sure it is not already disputed. Not doing so will result in the transfer being undisputable, potentially resulting in loss of funds.

4.3 In the current implementation, once a dispute is started the channel cannot proceed until the dispute is expired, even if both parties already defunded every asset/transfer they intended to. It is possible to add a function that can explicitly end the dispute by mutual agreement, without waiting for the expiration. This will make channel recovery much faster. Implementing this improvement will allow defund phases to last for an infinite period of time which will make it less likely for parties to run out of time to get their funds back.

4.4 Protocol users have to be soundly informed that the protocol only accepts ERC20 or native assets. Sending any other value to the channel will result in permanent loss of that value. It is possible to implement a general withdrawal function that will allow recovering anything sent to the channel.

4.5 Channel allows Bob to send deposits directly from any source, while Alice is required to call a special function in order to do so. It is possible to let Alice do deposits in the same way as Bob, by deploying an additional deposit contract for Alice as part of channel creation.

4.6 All functions utilize an `onlyViaProxy` modifier to ensure that the `ChannelMastercopy` is not called directly. This adds a small gas overhead to every call, which can end up being substantial in the long run given a high number of channels. This will not restrict someone from sending tokens directly and getting them locked forever.

One way to remove this overhead is to extend the `CMCCore` constructor to set the `alice` variable to a non zero address, while leaving `ReentrancyGuard` uninitialized. This will effectively lock the `ChannelMastercopy` from being used or setup and at the same time will deny any direct calls, because the `nonReentrant` modifier condition `require(lock == OPEN)` will always fail due to the `lock` being 0.

In order to make any assets sent to the `ChannelMastercopy` recoverable, it can instead be initialized with the deployer keys as Alice and Bob. The deployer can then be contacted in order to recover the funds by signing a withdrawal with those keys.

4.7 All view functions utilize the `nonReentrantView` modifier to ensure that there are no calls made to the channel during the state transition. This is meant to protect the caller from making incorrect assumptions on the channel state, while not protecting the channel state in any way. It should be safe for the channel to remove it.

5. Critical Bugs and Vulnerabilities

No critical issues were identified during the course of review.

6. Line by line Review

6.1 ITransferDefinition, line 19. Note, `encodedBalance` could utilize `Balance` type, as it is the only one allowed by `CMCAdjudicator`.

6.2 LibChannelCrypto, line 38. Note, the comment is outdated.

6.3 LibIterableMapping, line 58. Optimization, `isEmptyString(name)` check can be removed because the same check happens in the `nameExists()`. Will save gas.

6.4 LibIterableMapping, line 90. Typo, `NAME_NOT_FOUND` error is incorrect. It should be something like `NAME_ALREADY_ADDED`.

6.5 LibIterableMapping, line 102. Optimization, `isEmptyString(name)` check can be removed because the same check happens in the `nameExists()`. Will save gas.

6.6 HashlockTransfer, line 41. Note, `balance.amount[0]` is still allowed to be zero, even though there is no reason to allow that. Consider adding a requirement for sender balance to be positive.

6.7 HashlockTransfer, line 68. Minor, the "If timelock is nonzero and has expired, payment is canceled" comment is misleading. `Resolve` will be reverted in this case, forcing the caller to supply zero `preImage`. And if zero `preImage` is supplied then expiration will not be checked.

6.8 HashlockTransfer, line 71. Note, if the expiration part of the condition didn't pass, then it will always fail with `NONZERO_LOCKHASH`.

6.9 HashlockTransfer, line 74. Optimization, `keccak256` is cheaper than `sha256`. Consider updating off-chain code to use `keccak256`.

6.10 Withdraw, line 34. Note, the `callTo` and `callData` parameters are not used, and can be removed from the definition or can be verified, i.e. `callTo` can be verified to not be zero in case `callData` is not empty.

6.11 Withdraw, line 56. Note, `state.fee == balance.amount[0]` will result in zero value withdrawal as everything will go to fee.

6.12 Withdraw, line 60. Minor, `state.data` could be constructed here if all the `WithdrawData` parameters are passed along. In this way verification will make sure that the actual withdrawal will be possible. Currently it is up to off-chain code to make sure the data hash is correct, even though it could be totally unrelated.

6.13 Withdraw, line 93. Note, other transfer definition, `HashlockTransfer`, doesn't have `.resolve` namespace for errors in `resolve()` function. Consider removing the namespace here, or adding it to `HashlockTransfer`, to have a unified style.

6.14 ChannelFactory, line 29. Note, consider deploying the `ChannelMastercopy` here inline to make sure that the correct `mastercopy` address is used. This will make the deployment process atomic and will exclude the possibility of misconfiguration.

6.15 ChannelFactory, line 48. Optimization, in this code path, `SLOAD(chainId_slot)` happens twice. Consider putting the `chainId` into a local var in the beginning of the function to save gas.

6.16 ChannelFactory, line 84. Optimization, `keccak256(getProxyCreationCode())` can be stored in an immutable var in the constructor to make `getChannelAddress()` cheaper.

6.17 CMCAdjudicator, line 183. Note, make sure off-chain code never sets a defund nonce to `type(uint256).max`, which will block any further refunds of the asset.

6.18 CMCDeposit, line 21. Minor, Bob will not be able to deposit Ether with `address.transfer(value)` from another contract, because there is only 2300 gas available, and it will not be enough for the `nonReentrant` modifier.

6.19 CMCDeposit, line 74. Minor, Ether can be deposited, along with another asset, and will end up credited to Bob instead of Alice. Consider requiring `msg.value == 0` in the `else` clause.

6.20 CMCWithdraw, line 77. Note, can only withdraw if asset supports `transfer(address, uint256)`;

[illegible]

Oleksii Matiiasevych