Liquid Collective Security Review

Auditors

Optimum, Lead Security Researcher

Emanuele Ricci, Security Researcher

Danyal Ellahi, Junior Security Researcher

Matt Eccentricexit, Junior Security Researcher

Report prepared by: Pablo Misirov

January 24, 2023

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 About Alluvial

Alluvial is a software development company supporting the development of the Liquid Collective protocol. Alluvial is building the industry standard for enterprise-grade liquid staking, combining institutions' technical and security requirements with the web3 ethos of community-driven collaboration. You can learn more here: `https://alluvial.finance/`.

Because Alluvial is conducting protocol development on behalf of Liquid Collective, this report notes actions that Alluvial has taken or should take to remediate findings of the report.

3 Introduction

Liquid Collective is a multichain enterprise-grade liquid staking protocol, launching first on Ethereum. It allows institutional investors to stake and earn staking rewards while evidencing ownership of staked tokens in the form of a liquid receipt token. Liquid Collective offers a solution that caters to the needs of institutions including:

- KYC / AML allowlisting process for all participants (including validators).
- Top performing node operators with multi-cloud, multi-region, and multi-client infrastructure.
- Governance by a broad and dispersed collective of industry participants.

Disclaimer : This security review does not guarantee against a hack. It is a snapshot in time of the Liquid Collective protocol according to the specific commit. Any modifications to the code will require a new security review.

4 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

4.1 Impact

- 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.

4.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

4.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

5 Executive Summary

Over the course of 5 days in total, Liquid Collective engaged with Spearbit to review the Liquid Collective protocol. In this period of time a total of **31** issues were found.

Summary

Project Name	Liquid Collective
Repository	Liquid Collective Protocol
Feature	TLC Token
Feature	Slashing coverage
Feature	Cliff duration
Feature	Contract metadata
Commit	7693929c...59399121
Type of Project	Liquid Staking, DeFi
Audit Timeline	Nov 7 - Nov 11
Two week fix period	Nov 11 - Dec 2

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	1	1	0
High Risk	0	0	0
Medium Risk	2	1	1
Low Risk	2	1	1
Gas Optimizations	1	1	0
Informational	25	23	2
Total	**31**	**27**	**4**

6 Remediation Table

The following table contains all issues found during the audit together with its corresponding severity and fix PR

Number	Issue	Severity	PR
7.1.1	A malicious user could DOS a vesting schedule by sending only 1 wei of TLC to the vesting escrow address	Critical	SPEARBIT/171
7.2.1	Coverage funds might be pulled not only for the purpose of covering slashing losses	Medium	Acknowledged
7.2.2	Consider preventing CoverageFundAddress to be set as `address(0)`	Medium	SPEARBIT/169
7.3.1	`CoverageFund.initCoverageFundV1` might be front-runnable	Low	SPEARBIT/170
7.3.2	Account owner of the minted TLC tokens must call delegate to own vote power of initial minted tokens	Low	Acknowledged
7.4.1	Consider using unchecked block to save some gas	Gas Op	SPEARBIT/172
7.5.1	`_createVestingSchedule` allows the creation of a vesting schedule that could release zero tokens after a period has passed	Informational	SPEARBIT/172
7.5.2	CoverageFund - Checks-Effects-Interactions best practice is violated	Informational	SPEARBIT/168
7.5.3	River contract allows setting an empty metadata URI	Informational	SPEARBIT/167
7.5.4	Consider requiring that the `_cliffDuration` is a multiple of `_period`	Informational	SPEARBIT/172
7.5.5	Add documentation about the scenario where a vesting schedule can be created in the past	Informational	SPEARBIT/172
7.5.6	`_ERC20VestableVotesUpgradeableV1 createVestingSchedule` allows the creation of vesting schedules that have already ended and cannot be revoked	Informational	Acknowledged
7.5.7	`_getVestingSchedule` returns misleading information if the vesting token creator revokes the schedule	Informational	SPEARBIT/172
7.5.8	The `_computeVestingVestedAmount` will return the wrong amount of vested tokens if the creator of the vested schedule revokes the schedule	Informational	SPEARBIT/172
7.5.9	Consider writing clear documentation on how the voting power and delegation works	Informational	SPEARBIT/172
7.5.10	Fix mismatch between revert error message and code behavior	Informational	SPEARBIT/172
7.5.11	Improve documentation and naming of period variable	Informational	SPEARBIT/172
7.5.12	Consider renaming period to `periodDuration` to be more descriptive	Informational	SPEARBIT/172
7.5.13	Coverage funds might be left stuck in the contract	Informational	Acknowledged
7.5.14	Consider removing `coverageFunds` variable and explicitly initialize `executionLayerFees` to zero	Informational	SPEARBIT/168

7.5.15	Consider renaming `IVestingScheduleManagerV1` interface to `IERC20VestableVotesUpgradeableV1`	Informational	SPEARBIT/172
7.5.16	Consider renaming `CoverageFundAddress COVERAGE_FUND_ADDRESS` to be consistent with the current naming convention	Informational	SPEARBIT/168
7.5.17	Consider reverting if the `msg.value` is zero in `CoverageFundV1.donate`	Informational	SPEARBIT/168
7.5.18	Consider having a separate function in River contract that allows `CoverageFundV1` to send funds instead of using the same function used by `ELFeeRecipientV1`	Informational	SPEARBIT/168
7.5.19	Extensively document how the Coverage Funds contract works	Informational	SPEARBIT/168
7.5.20	Missing/wrong natspec comment and typos	Informational	SPEARBIT/172
7.5.21	Different behavior between River `_pullELFees` and `pullCoverageFunds`	Informational	SPEARBIT/168
7.5.22	Move local mask variable from `Allowlist.1.sol` to `LibAllowlistMasks.sol`	Informational	SPEARBIT/166
7.5.23	Consider adding additional parameters to the existing events to improve filtering/monitoring	Informational	SPEARBIT/172
7.5.24	Missing indexed keyword in events parameters	Informational	SPEARBIT/168
7.5.25	Add natspec documentation to the TLC contract	Informational	SPEARBIT/172

7 Findings

7.1 Critical Risk

7.1.1 A malicious user could DOS a vesting schedule by sending only `1 wei` of `TLC` to the vesting escrow address

Severity: Critical Risk

Context:

- ERC20VestableVotesUpgradeable.1.sol#L132-L134
- ERC20VestableVotesUpgradeable.1.sol#L137-L139
- ERC20VestableVotesUpgradeable.1.sol#L86-L97
- ERC20VestableVotesUpgradeable.1.sol#L353

Description: An external user who owns some TLC tokens could DOS the vesting schedule of any user by sending just `1 wei` of TLC to the escrow address related to the vesting schedule.

By doing that:

- The creator of the vesting schedule will not be able to revoke the vesting schedule.
- The beneficiary of the vesting schedule will not be able to release any vested tokens until the end of the vesting schedule.
- Any external contracts or dApps will not be able to call `computeVestingReleasableAmount` .

In practice, all the functions that internally call `_computeVestingReleasableAmount` will revert because of an underflow error when called before the vesting schedule ends.

The underflow error is thrown because, when called before the schedule ends, `_computeVestingReleasableAmount` will enter the `if (_time < _vestingSchedule.end)` branch and will try to compute `uint256 releasedAmount = _computeVestedAmount(_vestingSchedule, _vestingSchedule.end) - balanceOf(_escrow);`

In this case, `_computeVestedAmount(_vestingSchedule, _vestingSchedule.end)` will always be lower than `balanceOf(_escrow)` and the contract will revert with an underflow error.

When the vesting period ends, the contract will not enter the `if (_time < _vestingSchedule.end)` and the user will be able to gain the whole vested amount plus the extra amount of TLC sent to the escrow account by the malicious user.

Scenario:

1) Bob owns 1 TLC token.
2) Alluvial creates a vesting schedule for Alice like the following example:

```
createVestingSchedule(
    VestingSchedule({
        start: block.timestamp,
        cliffDuration: 1 days,
        lockDuration: 0,
        duration: 10 days,
        period: 1 days,
        amount: 10,
        beneficiary: alice,
        delegatee: address(0),
        revocable: true
    })
);
```

3) Bob sends 1 TLC token to the vesting schedule escrow account of the Alice vesting schedule.

4) After the cliff period, Alice should be able to release 1 TLC token. Because now `balanceOf(_escrow)` is `11` it will underflow as `_computeVestedAmount(_vestingSchedule, _vestingSchedule.end)` returns `10`.

Find below a test case showing all three different DOS scenarios:

```
//SPDX-License-Identifier: MIT

pragma solidity 0.8.10;

import "forge-std/Test.sol";

import "../src/TLC.1.sol";

contract WrappedTLC is TLCV1 {
    function deterministicVestingEscrow(uint256 _index) external view returns (address escrow) {
        return _deterministicVestingEscrow(_index);
    }
}

contract SpearVestTest is Test {
    WrappedTLC internal tlc;

    address internal escrowImplem;
    address internal initAccount;
    address internal bob;
    address internal alice;
    address internal carl;

    function setUp() public {
        initAccount = makeAddr("init");
        bob = makeAddr("bob");
        alice = makeAddr("alice");
        carl = makeAddr("carl");

        tlc = new WrappedTLC();
        tlc.initTLCV1(initAccount);
    }

    function testDOSReleaseVestingSchedule() public {
        // send Bob 1 vote token
        vm.prank(initAccount);
        tlc.transfer(bob, 1);

        // create a vesting schedule for Alice
        vm.prank(initAccount);
        createVestingSchedule(
            VestingSchedule({
                start: block.timestamp,
                cliffDuration: 1 days,
                lockDuration: 0,
                duration: 10 days,
                period: 1 days,
                amount: 10,
                beneficiary: alice,
                delegatee: address(0),
                revocable: true
            })
        );

        address aliceEscrow = tlc.deterministicVestingEscrow(0);

        // Bob send one token directly to the Escrow contract of alice
```

```
        vm.prank(bob);
        tlc.transfer(aliceEscrow, 1);

        // Cliff period has passed and Alice try to get the first batch of the vested token
        vm.warp(block.timestamp + 1 days);
        vm.prank(alice);
        // The transaction will revert for UNDERFLOW because now the balance of the escrow has been
↪   increased externally
        vm.expectRevert(stdError.arithmeticError);
        tlc.releaseVestingSchedule(0);

        // Warp at the vesting schedule period end
        vm.warp(block.timestamp + 9 days);

        // Alice is able to get the whole vesting schedule amount
        // plus the token sent by the attacker to the escrow contract
        vm.prank(alice);
        tlc.releaseVestingSchedule(0);

        assertEq(tlc.balanceOf(alice), 11);
    }

    function testDOSRevokeVestingSchedule() public {
        // send Bob 1 vote token
        vm.prank(initAccount);
        tlc.transfer(bob, 1);

        // create a vesting schedule for Alice
        vm.prank(initAccount);
        createVestingSchedule(
            VestingSchedule({
                start: block.timestamp,
                cliffDuration: 1 days,
                lockDuration: 0,
                duration: 10 days,
                period: 1 days,
                amount: 10,
                beneficiary: alice,
                delegatee: address(0),
                revocable: true
            })
        );

        address aliceEscrow = tlc.deterministicVestingEscrow(0);

        // Bob send one token directly to the Escrow contract of alice
        vm.prank(bob);
        tlc.transfer(aliceEscrow, 1);

        // The creator decide to revoke the vesting schedule before the end timestamp
        // It will throw an underflow error
        vm.prank(initAccount);
        vm.expectRevert(stdError.arithmeticError);
        tlc.revokeVestingSchedule(0, uint64(block.timestamp + 1));
    }

    function testDOSComputeVestingReleasableAmount() public {
        // send Bob 1 vote token
        vm.prank(initAccount);
        tlc.transfer(bob, 1);

        // create a vesting schedule for Alice
```

```
        vm.prank(initAccount);
        createVestingSchedule(
            VestingSchedule({
                start: block.timestamp,
                cliffDuration: 1 days,
                lockDuration: 0,
                duration: 10 days,
                period: 1 days,
                amount: 10,
                beneficiary: alice,
                delegatee: address(0),
                revocable: true
            })
        );

        address aliceEscrow = tlc.deterministicVestingEscrow(0);

        // Bob send one token directly to the Escrow contract of alice
        vm.prank(bob);
        tlc.transfer(aliceEscrow, 1);

        vm.expectRevert(stdError.arithmeticError);
        uint256 releasableAmount = tlc.computeVestingReleasableAmount(0);

        // Warp to the end of the vesting schedule
        vm.warp(block.timestamp + 10 days);
        releasableAmount = tlc.computeVestingReleasableAmount(0);

        assertEq(releasableAmount, 11);
    }

    struct VestingSchedule {
        uint256 start;
        uint256 cliffDuration;
        uint256 lockDuration;
        uint256 duration;
        uint256 period;
        uint256 amount;
        address beneficiary;
        address delegatee;
        bool revocable;
    }

    function createVestingSchedule(VestingSchedule memory config) internal returns (uint256) {
        return createVestingScheduleStackOptimized(config);
    }

    function createVestingScheduleStackOptimized(VestingSchedule memory config) internal returns
↪   (uint256) {
        return
            tlc.createVestingSchedule(
                uint64(config.start),
                uint32(config.cliffDuration),
                uint32(config.duration),
                uint32(config.period),
                uint32(config.lockDuration),
                config.revocable,
                config.amount,
                config.beneficiary,
                config.delegatee
            );
    }
```

```
}
```

Recommendation: Consider re-implementing how the contract accounts for the amount of released tokens of a vesting schedule to avoid this situation. In case the new implementation does not rely anymore on `balanceOf(_escrow)`, remember that tokens sent directly to the escrow account would be stuck forever.

Alluvial: Fixed in liquid-collective/liquid-collective-protocol@7870787 by introducing a new variable inside the user vesting schedule named `releasedAmount` that tracks the already released amount and can not be manipulated by an external attacker.

Spearbit: Fixed.

7.2 Medium Risk

7.2.1 Coverage funds might be pulled not only for the purpose of covering slashing losses

Severity: Medium Risk

Context: OracleManager.1.sol#L108-L113

Description: The newly introduced coverage fund is a smart contract that holds ETH to cover a potential `lsETH` price decrease due to unexpected slashing events. Funds might be pulled from `CoverageFundV1` to the River contract through `setConsensusLayerData` to cover the losses and keep the share price stable In practice, however, it is possible that these funds will be pulled not only in emergency events. `_maxIncrease` is used as a measure to enforce the maximum difference between `prevTotalEth` and `postTotalEth`, but in practice, it is being used as a mandatory growth factor in the context of coverage funds, which might cause the pulling of funds from the coverage fund to ensure `_maxIncrease` of revenue in case fees are not high enough.

Recommendation: Consider replacing

```
if (((_maxIncrease + previousValidatorTotalBalance) - executionLayerFees) > _validatorTotalBalance) {
    coverageFunds = _pullCoverageFunds(
        ((_maxIncrease + previousValidatorTotalBalance) - executionLayerFees) - _validatorTotalBalance
    );
}
```

with

```
if (previousValidatorTotalBalance > _validatorTotalBalance + executionLayerFees)) {
    coverageFunds = _pullCoverageFunds(
        ((_maxIncrease + previousValidatorTotalBalance) - executionLayerFees) - _validatorTotalBalance
    );
}
```

Alluvial: Trying to clarify the use-case and the sequence of operations here:

- Use case: Liquid Collective partners with Nexus Mutual (NXM) and possibly other actors to cover for slashing losses. Each time Liquid Collective adds a validator key to the system, we will submit the key to NXM so they can monitor it and cover it in case of slashing. In case one of the validator's keys gets slashed (slashing being defined according to NXM policy), NXM will reimburse part or all of the lost ETH. The period between the slashing event occurs and the reimbursement that happens can go from 30 days up to 365 days. The reimbursement will go to the CoverageFund contract and subsequently be pulled into the core system respecting maximum bounds.
- Sequence of Operations:

1. Liquid Collective submits a validator key to NXM to be covered.
2. A slashing event occurs (e.g a validator key gets slashed 1 ETH).
3. NXM monitoring catches the slashing event.
4. 30 days to 365 days later NXM reimburses 1 ETH to the CoverageFund.

5. 1 ETH gets progressively pulled from the CoverageFund into River respecting the bounds.

Spearbit: Acknowledged as discussed with the Alluvial team, the impact of this issue is limited since the coverage fund should hold ETH only in case of a slashing event.

7.2.2 Consider preventing `CoverageFundAddress` to be set as `address(0)`

Severity: Medium Risk

Context:

- River.1.sol#L176
- CoverageFundAddress.sol#L21

Description: In the current implementation of `River.setCoverageFund` and `CoverageFundAddress.set` both function do not revert when the `_newCoverageFund` address parameter is equal to `address(0)`.

If the Coverage Fund address is empty, the `River._pullCoverageFunds` function will return earlier and will not pull any coverage fund.

Recommendation: If having an empty coverage fund address equal to `address(0)` is the intended behavior, we suggest explaining in both the documentation and natspec comments the reason and document in which scenario this could happen.

Otherwise, add inside the `CoverageFundAddress.set` function a sanity check on the new address and revert in case of `_newValue == address(0)`.

Alluvial: Added sanity check inside the `CoverageFundAddress.set` function in PR 169.

Spearbit: Acknowledged.

7.3 Low Risk

7.3.1 `CoverageFund.initCoverageFundV1` might be front-runnable

Severity: Low Risk

Context: CoverageFund.1.sol#L21

Description: Upgradeable contracts are used in the project, mostly relying on a `TUPProxy` contract. Initializing a contract is a 2 phase process where the first call is the actual deployment and the second call is a call to the `init` function itself. From our experience with the repository, the upgradeable contracts deployment scripts are using the `TUPProxy` correctly, however in that case we were not able to find the deployment script for `CoverFund`, so we decided to raise this point to make sure you are following the previous policy also for this contract.

Recommendation: Use the same structure of deployment scripts that are used in other upgradeable contracts also for `CoverFund` to make sure that the initialization process is atomic (i.e - executed in a single transaction).

Alluvial: Recommendation implemented in PR 170.

Spearbit: Acknowledged.

7.3.2 Account owner of the minted TLC tokens must call `delegate` to own vote power of initial minted tokens

Severity: Low Risk

Context: TLC.1.sol#L20

Description: The `_account` owner of the minted TLC tokens must remember to call `tlcToken.delegate(accountOwner)` to auto-delegate to itself, otherwise it will have zero voting power.

Without doing that anyone (even with just 1 voting power) could make any proposal pass and in the future manage the DAO proposing, rejecting or accepting/executing proposals.

As the OpenZeppelin ERC20 documentation says:

> *By default, token balance does not account for voting power. This makes transfers cheaper. The downside is that it requires users to delegate to themselves in order to activate checkpoints and have their voting power tracked.*

Recommendation: Remember to call `tlcToken.delegate(accountOwner)` after the deployment of the TLC token.

7.4 Gas Optimization

7.4.1 Consider using `unchecked` block to save some gas

Severity: Gas Optimization

Context: ERC20VestableVotesUpgradeable.1.sol#L354-L356

Description: Because of the `if` statement, it is impossible for `vestedAmount - releasedAmount` to underflow, thus allowing the usage of the `unchecked` block to save a bit of gas.

Recommendation: Consider implementing the code snippet below:

```
if (vestedAmount > releasedAmount) {
-   return vestedAmount - releasedAmount;
+   unchecked { return vestedAmount - releasedAmount; }
}
```

- Gas diff:

```
testReleaseVestingScheduleAfterLockDuration() (gas: -65 (-0.023%))
testReleaseVestingScheduleAtLockDuration() (gas: -65 (-0.023%))
testRevokeAtCliff() (gas: -65 (-0.025%))
testRevokeDefault() (gas: -65 (-0.025%))
testRevokeTwiceAfterEnd() (gas: -65 (-0.026%))
testReleaseVestingScheduleAfterRevoke() (gas: -130 (-0.044%))
testRevokeTwice() (gas: -130 (-0.048%))
testcomputeVestingAmounts() (gas: -195 (-0.059%))
testVestingScheduleFuzzing(uint24,uint32,uint8,uint8,uint256,uint256,uint256) (gas: -1732 (-0.595%))
Overall gas change: -2512 (-0.869%)
```

Alluvial: Recommendation implemented in PR 172.

Spearbit: Fixed.

7.5 Informational

7.5.1 `createVestingSchedule` allows the creation of a vesting schedule that could release zero tokens after a `period` has passed

Severity: Informational

Context: ERC20VestableVotesUpgradeable.1.sol#L368-L387

Description: Depending on the value of `duration` or `amount` it is possible to create a vesting schedule that would release zero token after a whole `period` has elapsed.

This is an edge case scenario but would still be possible given that `createVestingSchedule` can be called by anyone and not only Alluvial.

See the following test case for an example

```
//SPDX-License-Identifier: MIT

pragma solidity 0.8.10;

import "forge-std/Test.sol";

import "../src/TLC.1.sol";

contract WrappedTLC is TLCV1 {
    function deterministicVestingEscrow(uint256 _index) external view returns (address escrow) {
        return _deterministicVestingEscrow(_index);
    }
}

contract SpearVestTest is Test {
    WrappedTLC internal tlc;

    address internal escrowImplem;
    address internal initAccount;
    address internal bob;
    address internal alice;
    address internal carl;

    function setUp() public {
        initAccount = makeAddr("init");
        bob = makeAddr("bob");
        alice = makeAddr("alice");
        carl = makeAddr("carl");

        tlc = new WrappedTLC();
        tlc.initTLCV1(initAccount);
    }

    function testDistributeZeroPerPeriod() public {
        // create a vesting schedule for Alice
        vm.prank(initAccount);
        createVestingSchedule(
            VestingSchedule({
                start: block.timestamp,
                cliffDuration: 0 days,
                lockDuration: 0,
                duration: 365 days,
                period: 1 days,
                amount: 100,
                beneficiary: alice,
                delegatee: address(0),
```

```
                revocable: true
            })
        );

        // One whole period pass and alice check how many tokens she can release
        vm.warp(block.timestamp + 1 days);

        uint256 releasable = tlc.computeVestingReleasableAmount(0);
        assertEq(releasable, 0);
    }

    struct VestingSchedule {
        uint256 start;
        uint256 cliffDuration;
        uint256 lockDuration;
        uint256 duration;
        uint256 period;
        uint256 amount;
        address beneficiary;
        address delegatee;
        bool revocable;
    }

    function createVestingSchedule(VestingSchedule memory config) internal returns (uint256) {
        return createVestingScheduleStackOptimized(config);
    }

    function createVestingScheduleStackOptimized(VestingSchedule memory config) internal returns
↪   (uint256) {
        return
            tlc.createVestingSchedule(
                uint64(config.start),
                uint32(config.cliffDuration),
                uint32(config.duration),
                uint32(config.period),
                uint32(config.lockDuration),
                config.revocable,
                config.amount,
                config.beneficiary,
                config.delegatee
            );
    }
}
```

Recommendation: Consider preventing TLC owner to create vesting schedules or add another check that would revert the creation of a vesting schedule if the amount of releasable token for each period is equal to zero.

Alluvial: Recommendation has been implemented in PR 172.

Spearbit: Fixed.

7.5.2 `CoverageFund` - Checks-Effects-Interactions best practice is violated

Severity: Informational

Context: CoverageFund.1.sol#L35 CoverageFund.1.sol#L43

Description: We were not able to find any concrete instances of harmful reentrancy attack vectors in this contract, but it's recommended to follow the Checks-effects-interactions pattern anyway.

Recommendation: Consider moving the "effects" (code lines that modify the storage) right before the "interactions" (external calls)

Alluvial: Fixed in PR 168 by implementing auditor's recommendation.

Spearbit: Fixed.

7.5.3 River contract allows setting an empty metadata URI

Severity: Informational

Context: River.1.sol#L181-L184, MetadataURI.sol#L33-L44

Description: The current implementation of `River.setMetadataURI` and `MetadataURI.set` both allow the current value of the metadata URI to be updated to an empty string.

Recommendation: Consider adding a check inside `MetadataURI.set` (to follow the current project style) to revert in case `_newValue` is an empty string.

Alluvial: Recommendation implemented in PR 167.

Spearbit: Fixed.

7.5.4 Consider requiring that the `_cliffDuration` is a multiple of `_period`

Severity: Informational

Context: ERC20VestableVotesUpgradeable.1.sol#L158-L236

Description: When a vesting schedule is created via `_createVestingSchedule`, the only check made on `_period` parameter (other than being greater than zero) is that the `_duration` must be a multiple of `_period`.

If after the `_cliffDuration` the user can already release the matured vested tokens, it could make sense to also require that `_cliffDuration % _period == 0`

Recommendation: Consider requiring that `_cliffDuration % _period == 0` when a vesting schedule is created.

Alluvial: Recommendation has been implemented in PR 172.

Spearbit: Fixed.

7.5.5 Add documentation about the scenario where a vesting schedule can be created in the past

Severity: Informational

Context: ERC20VestableVotesUpgradeable.1.sol#L200-L202

Description: In the current implementation of `ERC20VestableVotesUpgradeable` `_createVestingSchedule` function, there is no check for the `_start` value.

This means that the creator of a vesting schedule could create a schedule that starts in the past. Allowing the creation of a vesting schedule with a past `_start` also influences the behavior of `_revokeVestingSchedule` (see *`ERC20VestableVotesUpgradeableV1` `createVestingSchedule` allows the creation of vesting schedules that have already ended and cannot be revoked*).

Recommendation: Consider documenting this behavior and the reason to allow vesting schedules with `_start < block.timestamp`.

Alluvial: The behavior has been documented in the PR 172.

Spearbit: Fixed.

7.5.6 `ERC20VestableVotesUpgradeableV1` `createVestingSchedule` allows the creation of vesting schedules that have already ended and cannot be revoked

Severity: Informational

Context: ERC20VestableVotesUpgradeable.1.sol#L158-L236, ERC20VestableVotesUpgradeable.1.sol#L243-L258

Description: The current implementation of `_createVestingSchedule` allows the creation of vesting schedules that

- Start in the past: `_start < block.timestamp`.
- Have already ended: `_start + _duration < block.timestamp`.

Because of this behavior, in case of the creation of a past vesting schedule that has already ended

- The `_beneficiary` can instantly call (if there's no lock period) `releaseVestingSchedule` to release the whole amount of tokens.
- The `creator` of the vesting schedule cannot call `revokeVestingSchedule` because the new `end` would be in the past and the transaction would revert with an `InvalidRevokedVestingScheduleEnd` error.

The second scenario is particularly important because it does not allow the creator to reduce the length or remove the schedule entirely in case the schedule has been created mistakenly or with a misconfiguration (too many token vested, lock period too long, etc...).

Recommendation: Consider changing the behavior of `_createVestingSchedule` to at least prevent the creation of vesting schedules that are already ended because `_start + _duration < block.timestamp`.

Spearbit: Alluvial acknowledges the behavior with PR 172.

7.5.7 `getVestingSchedule` returns misleading information if the vesting token `creator` revokes the schedule

Severity: Informational

Context: ERC20VestableVotesUpgradeable.1.sol#L71-L73

Description: The `getVestingSchedule` function returns the information about the created vesting schedule. The `duration` represents the number of seconds of the vesting period and the `amount` represents the number of tokens that have been scheduled to be released after the `period` end (or after `lockDuration` if it has been configured to be greater than `end`).

If the `creator` of the vesting schedule calls `revokeVestingSchedule`, only the `end` of the vesting schedule struct will be updated.

If external contracts or dApps rely only on the `getVestingSchedule` information there could be scenarios where they display or base their logic on wrong information.

Consider the following example. Alluvial creates a vesting schedule for `alice` with the following config

```
{
    "start": block.timestamp,
    "cliffDuration": 1 days,
    "lockDuration": 0,
    "duration": 10 days,
    "period": 1 days,
    "amount": 10,
    "beneficiary": alice,
    "delegatee": alice,
    "revocable": true
}
```

This means that after 10 days, Alice would own in her balance 10 TLC tokens.

If Alluvial calls `revokeVestingSchedule` before the cliff period ends, all of the tokens will be returned to Alluvial but the `getVestingSchedule` function would still display the same information with just the `end` attribute updated.

An external dApp or contract that does not check the new `end` and compares it to `cliffDuration`, `lockDuration`, and `period` but only uses the `amount` would display the wrong number of vested tokens for Alice at a given timestamp.

Recommendation: Consider documenting this behavior and explain how to display the correct information in all the scenarios, or update how `getVestingSchedule` returns the vesting schedule information.

Another possible solution is to be very explicit on the meaning of each attribute, declaring that those are not real-time values but just the configuration used at the creation of the vesting schedule and that only the `end` attribute can change when `revokeVestingSchedule` is called.

Alluvial should anyway take care to extensively document which is the best practice for a user, external contract or dApps to query the TLC contracts to gather the correct and up-to-date information relative to a vesting schedule.

Spearbit: Alluvial has extended the `natspec` documentation of `getVestingSchedule` in PR 172 explaining that only the `end` field is updating when a schedule is revoked.

No changes have been made to the `getVestingSchedule` code.

7.5.8 The `computeVestingVestedAmount` will return the wrong amount of vested tokens if the `creator` of the vested schedule revokes the schedule

Severity: Informational

Context:

- ERC20VestableVotesUpgradeable.1.sol#L100-L103
- ERC20VestableVotesUpgradeable.1.sol#L368-L387

Description: The `computeVestingVestedAmount` will return the wrong amount of vested tokens if the `creator` of the vested schedule revokes the schedule.

This function returns the value returned by `_computeVestedAmount` that relies on `duration` and `amount` while the only attribute changed by `revokeVestingSchedule` is the `end`.

```
function _computeVestedAmount(VestingSchedules.VestingSchedule memory _vestingSchedule, uint256 _time)
    internal
    pure
    returns (uint256)
{
    if (_time < _vestingSchedule.start + _vestingSchedule.cliffDuration) {
        // pre-cliff no tokens have been vested
        return 0;
    } else if (_time >= _vestingSchedule.start + _vestingSchedule.duration) {
        // post vesting all tokens have been vested
        return _vestingSchedule.amount;
    } else {
        uint256 timeFromStart = _time - _vestingSchedule.start;

        // compute tokens vested for completly elapsed periods
        uint256 vestedDuration = timeFromStart - (timeFromStart % _vestingSchedule.period);

        return (vestedDuration * _vestingSchedule.amount) / _vestingSchedule.duration;
    }
}
```

If the creator revokes the schedule, the `computeVestingVestedAmount` would return more tokens compared to the amount that the user has vested in reality.

Consider the following example. Alluvial creates a vesting schedule with the following config

```
{
    "start": block.timestamp,
    "cliffDuration": 1 days,
    "lockDuration": 0,
    "duration": 10 days,
    "period": 1 days,
    "amount": 10,
    "beneficiary": alice,
    "delegatee": alice,
    "revocable": true
}
```

Alluvial then calls `revokeVestingSchedule(0, uint64(block.timestamp + 5 days));`. The effect of this transaction would return `5` tokens to Alluvial and set the new `end` to `block.timestamp + 5 days`.

If `alice` calls `computeVestingVestedAmount(0)` at the time `uint64(block.timestamp + 7 days)`, it would return `7` because `_computeVestedAmount` would execute the code in the `else` branch. But `alice` cannot have more than `5` vested tokens because of the previous revoke.

If `alice` calls `computeVestingVestedAmount(0)` at the time `uint64(block.timestamp + duration)` it would return `10` because `_computeVestedAmount` would execute the code in the `else if (_time >= _vestingSchedule.start + _vestingSchedule.duration)` branch. But `alice` cannot have more than `5` vested tokens because of the previous revoke.

Attached test below to reproduce it:

```
//SPDX-License-Identifier: MIT

pragma solidity 0.8.10;

import "forge-std/Test.sol";

import "../src/TLC.1.sol";

contract WrappedTLC is TLCV1 {
```

```
    function __computeVestingReleasableAmount(uint256 vestingID, uint256 _time) external view returns
↪   (uint256) {
        return
            _computeVestingReleasableAmount(
                VestingSchedules.get(vestingID),
                _deterministicVestingEscrow(vestingID),
                _time
            );
    }
}

contract SpearTLCTest is Test {
    WrappedTLC internal tlc;

    address internal escrowImplem;
    address internal initAccount;
    address internal creator;
    address internal bob;
    address internal alice;
    address internal carl;

    function setUp() public {
        initAccount = makeAddr("init");
        creator = makeAddr("creator");
        bob = makeAddr("bob");
        alice = makeAddr("alice");
        carl = makeAddr("carl");

        tlc = new WrappedTLC();
        tlc.initTLCV1(initAccount);
    }

    function testIncorrectComputeVestingVestedAmount() public {
        vm.prank(initAccount);
        tlc.transfer(creator, 10);

        // create a vesting schedule for Alice
        vm.prank(creator);
        createVestingSchedule(
            VestingSchedule({
                start: block.timestamp,
                cliffDuration: 0 days,
                lockDuration: 0, // no lock
                duration: 10 days,
                period: 1 days,
                amount: 10,
                beneficiary: alice,
                delegatee: address(0),
                revocable: true
            })
        );

        // creator call revokeVestingSchedule revoking the vested schedule setting the new end as half
↪   of the duration
        // 5 tokens are returned to the creator and `end` is updated to the new value
        // this means also that at max alice will have 5 token vested (and releasable)
        vm.prank(creator);
        tlc.revokeVestingSchedule(0, uint64(block.timestamp + 5 days));

        // We warp at day 7 of the schedule
        vm.warp(block.timestamp + 7 days);
```

```
        // This should fail because alice at max have only 5 token vested because of the revoke
        assertEq(tlc.computeVestingVestedAmount(0), 7);

        // We warp at day 10 (we reached the total duration of the vesting)
        vm.warp(block.timestamp + 3 days);

        // This should fail because alice at max have only 5 token vested because of the revoke
        assertEq(tlc.computeVestingVestedAmount(0), 10);
    }

    struct VestingSchedule {
        uint256 start;
        uint256 cliffDuration;
        uint256 lockDuration;
        uint256 duration;
        uint256 period;
        uint256 amount;
        address beneficiary;
        address delegatee;
        bool revocable;
    }

    function createVestingSchedule(VestingSchedule memory config) internal returns (uint256) {
        return createVestingScheduleStackOptimized(config);
    }

    function createVestingScheduleStackOptimized(VestingSchedule memory config) internal returns
↪   (uint256) {
        return
            tlc.createVestingSchedule(
                uint64(config.start),
                uint32(config.cliffDuration),
                uint32(config.duration),
                uint32(config.period),
                uint32(config.lockDuration),
                config.revocable,
                config.amount,
                config.beneficiary,
                config.delegatee
            );
    }
}
```

Recommendation: Consider refactoring the code inside `computeVestingVestedAmount` to correctly handle the scenario when the vesting schedule has been revoked.

Alluvial: Recommendation has been implemented in PR 172.

Spearbit: Acknowledged.

7.5.9 Consider writing clear documentation on how the voting power and delegation works

Severity: Informational

Context: ERC20VestableVotesUpgradeable.1.sol

Description: The `ERC20VestableVotesUpgradeableV1` is an extension of the OpenZeppelin `ERC20VotesUpgradeable` contract. As the official OpenZeppelin documentation says (also reported in the Alluvial's natspec contract):

> *By default, token balance does not account for voting power. This makes transfers cheaper. The downside is that it requires users to delegate to themselves in order to activate checkpoints and have their voting power tracked.*

Because of how `ERC20VotesUpgradeable` behaves on voting power and delegation of voting power could be counterintuitive for normal users who are not aware of it, Alluvial should be very explicit on how users should act when a vesting schedule is created for them.

When a Vote Token is transferred, `ERC20VotesUpgradeable` calls the hook `_afterTokenTransfer`

```
function _afterTokenTransfer(
    address from,
    address to,
    uint256 amount
) internal virtual override {
    super._afterTokenTransfer(from, to, amount);

    _moveVotingPower(delegates(from), delegates(to), amount);
}
```

In this case, `_moveVotingPower(delegates(from), delegates(to), amount);` will decrease the voting power of `delegates(from)` by `amount` and will increase the voting power of `delegates(to)` by `amount`. This applies if some conditions are true, but you can see them here

```
function _moveVotingPower(
    address src,
    address dst,
    uint256 amount
) private {
    if (src != dst && amount > 0) {
        if (src != address(0)) {
            (uint256 oldWeight, uint256 newWeight) = _writeCheckpoint(_checkpoints[src], _subtract,
↪   amount);
            emit DelegateVotesChanged(src, oldWeight, newWeight);
        }

        if (dst != address(0)) {
            (uint256 oldWeight, uint256 newWeight) = _writeCheckpoint(_checkpoints[dst], _add, amount);
            emit DelegateVotesChanged(dst, oldWeight, newWeight);
        }
    }
}
```

When a vesting schedule is created, the `creator` has two options:

1) Specify a custom `delegatee` different from the `beneficiary` (or equal to it, but it's the same as option 2).
2) Leave the `delegatee` empty (equal to `address(0)`).

- Scenario 1) empty `delegatee` OR `delegatee === beneficiary` (same thing)

After creating the vesting schedule, the voting power of the `beneficiary` will be equal to the amount of tokens vested. If the `beneficiary` **did not** call `tlc.delegate(beneficiary)` previously, after releasing some tokens, **its voting power will be decreased by the amount of released tokens.**

- Scenario 2) `delegatee !== beneficiary && delegatee !== address(0)`

Same thing as before, but now we have two different actors, one is the `beneficiary` and another one is the `delegatee` of the voting power of the vested tokens.

If the `beneficiary` **did not** call `tlc.delegate(vestingScheduleDelegatee)` previously, after releasing some tokens, **the voting power of the current vested schedule's** `delegatee` **will be decreased by the amount of released tokens.**

- Related test for scenario 1

```
//SPDX-License-Identifier: MIT

pragma solidity 0.8.10;

import "forge-std/Test.sol";

import "../src/TLC.1.sol";

contract WrappedTLC is TLCV1 {
    function deterministicVestingEscrow(uint256 _index) external view returns (address escrow) {
        return _deterministicVestingEscrow(_index);
    }
}

contract SpearTLCTest is Test {
    WrappedTLC internal tlc;

    address internal escrowImplem;
    address internal initAccount;
    address internal bob;
    address internal alice;
    address internal carl;

    function setUp() public {
        initAccount = makeAddr("init");
        bob = makeAddr("bob");
        alice = makeAddr("alice");
        carl = makeAddr("carl");

        tlc = new WrappedTLC();
        tlc.initTLCV1(initAccount);
    }

    function testLosingPowerAfterRelease() public {
        // create a vesting schedule for Alice
        vm.prank(initAccount);
        createVestingSchedule(
            VestingSchedule({
                start: block.timestamp,
                cliffDuration: 1 days,
                lockDuration: 0, // no lock
                duration: 10 days,
                period: 1 days,
                amount: 10,
                beneficiary: alice,
                delegatee: address(0),
                revocable: false
            })
        );
        address aliceEscrow = tlc.deterministicVestingEscrow(0);

        assertEq(tlc.getVotes(alice), 10);
```

```
        assertEq(tlc.balanceOf(alice), 0);

        // Cliff period has passed and Alice try to get the first batch of the vested token
        vm.warp(block.timestamp + 1 days);
        vm.prank(alice);
        tlc.releaseVestingSchedule(0);

        // Alice now owns the vested tokens just released but her voting power has decreased by the
↪   amount released
        assertEq(tlc.getVotes(alice), 9);
        assertEq(tlc.balanceOf(alice), 1);
    }

    struct VestingSchedule {
        uint256 start;
        uint256 cliffDuration;
        uint256 lockDuration;
        uint256 duration;
        uint256 period;
        uint256 amount;
        address beneficiary;
        address delegatee;
        bool revocable;
    }

    function createVestingSchedule(VestingSchedule memory config) internal returns (uint256) {
        return createVestingScheduleStackOptimized(config);
    }

    function createVestingScheduleStackOptimized(VestingSchedule memory config) internal returns
↪   (uint256) {
        return
            tlc.createVestingSchedule(
                uint64(config.start),
                uint32(config.cliffDuration),
                uint32(config.duration),
                uint32(config.period),
                uint32(config.lockDuration),
                config.revocable,
                config.amount,
                config.beneficiary,
                config.delegatee
            );
    }
}
```

Recommendation: Consider writing clear documentation on how the voting power and delegation works and explains how both the `beneficiary` and `delegatee` of the vested schedule should act to prevent decreasing their voting power when vested tokens are released or transferred.

Alluvial: Recommendation has been implemented in PR 172.

Spearbit: Fixed.

7.5.10 Fix mismatch between revert error message and code behavior

Severity: Informational

Context: ERC20VestableVotesUpgradeable.1.sol#L196

Description: The error message requires the schedule duration to be greater than the cliff duration, but the code allows it to be greater than *or equal* to the cliff duration.

Recommendation: Update the condition to match the error or vice versa:

```
- if (_cliffDuration > _duration) {
+ if (_cliffDuration >= _duration) {
```

or

```
- revert InvalidVestingScheduleParameter("Vesting schedule duration must be greater than the cliff
↪  duration");
+ revert InvalidVestingScheduleParameter("Vesting schedule duration must be greater than or equal to
↪  the cliff duration");
```

Alluvial: Code behavior is correct, updated the error message in PR 172.

Spearbit: Fixed.

7.5.11 Improve documentation and naming of `period` variable

Severity: Informational

Context: VestingSchedules.sol#L24

Description: Similar to *Consider renaming `period` to `periodDuration` to be more descriptive*, the variable name and documentation are ambiguous. We can give a more descriptive name to the variable and fix the documentation.

Recommendation: Change the variable to `periodDuration` and improve the documentation on both:

```
struct VestingSchedule {
     ...
-    // duration of the vesting period in seconds
-    uint32 duration;
-    // duration of a vesting period in seconds
-    uint32 period;
-    // amount of tokens granted by the vesting schedule
+    // duration of the entire vesting (sum of all vesting period durations)
+    uint32 duration;
+    // duration of a single period of vesting
+    uint32 periodDuration;
     ...
}
```

Alluvial: Recommendation implemented in PR 172.

Spearbit: Fixed.

7.5.12 Consider renaming `period` to `periodDuration` to be more descriptive

Severity: Informational

Context: ERC20VestableVotesUpgradeable.1.sol#L153

Description: `period` can be confused as (for example) a counter or an id.

Recommendation: Rename it to something more descriptive like `periodDuration`.

Alluvial: Recommendation implemented in PR 172.

Spearbit: Fixed.

7.5.13 Coverage funds might be left stuck in the contract

Severity: Informational

Context: OracleManager.1.sol#L79 Oracle.1.sol#L416

Description: The newly introduced coverage fund is a smart contract that holds ETH to cover a potential `lsETH` price decrease due to unexpected slashing events. Funds might be pulled from `CoverageFundV1` to the River contract through `setConsensusLayerData` to cover the losses and keep the share price stable. `_sanityChecks` will revert if a major loss is reported in a single transaction, since the absolute difference between `prevTotalEth` and `postTotalEth` may be greater than the allowed value. Therefore, oracles will have to report this loss gradually using multiple calls to `reportConsensusLayerData`, which will potentially cover a portion of the loss in each transaction by pulling that portion from the coverage fund. The coverage fund is an on-demand source of liquidity. Funds will be deposited to the `CoverageFundV1` (and will be claimable by the River contract) only after a scrutiny process that makes sure the loss event matches the insurance policy. Thus, it may take a while for funds to be deposited to the `CoverageFundV1` contract after a slashing event had occurred. The call to `_pullCoverageFunds` will not revert for the case where the required amount was not eventually pulled, rather, the execution will continue and by the end of the transaction, `CLValidatorTotalBalance` will hold the value of `_validatorTotalBalance` that reflects the already decreased value. The issue arises in case the stream of loss transactions (calls to `reportConsensusLayerData`) will be processed and executed before the coverage fund is loaded with ETH. Since the total loss is not accumulated, it may lead to funds that are left stuck in the `CoverageFundV1` contract which can not be claimed by the River contract, leaving the `lsETH` price low.

Consider the following example

`DepositedValidatorCount = 2` (2 active validators) `CLValidatorTotalBalance = 64` `relativeLowerBound = annualAprUpperBound = 500` (5%) `ELFees = 0` (just for simplicity) No funds yet in the coverage fund Assuming no new deposits to the river contract

- On `T0` a slashing of 14 ETH occurred, leaving only 50 ETH in CL. Now oracles can not report the entire loss, due to the way `_sanityChecks` works. Oracles can only report a loss of `64*0.05 = 3.2` ETH in the first `_pushToRiver` transaction.
- On `T1` `_pushToRiver` is called with `_totalBalance = 64 - 3.2 = 60.8 ETH`. Assuming `executionLayerFees = 0`, `setConsensusLayerData` will try to pull `_maxIncrease + 3.2` from the coverage fund, but since there are no funds there, the transaction will end up with `CLValidatorTotalBalance = 60.8`.
- On `T2` `_pushToRiver` is called again, this time with `_totalBalance = 60.8 - 3.04 = 57.76 ETH`. Assuming `executionLayerFees = 0`, `setConsensusLayerData` will try to pull `_maxIncrease + 3.04` from the coverage fund, but since there are no funds there, the transaction will end up with `CLValidatorTotalBalance = 57.76 ETH`.
- On `T3` 14 ETH are transferred to the coverage fund contract eventually.
- On `T4` `_pushToRiver` is called again, this time with `_totalBalance = 57.76 * 0.95 = 54.872 ETH`. Assuming `T4-T2 = 384 seconds`, `maxIncrease = 57.76 * 0.05 * 384 / 31536000 ~ 0`, `executionLayerFees = 0`, `setConsensusLayerData` will try to pull `_maxIncrease + 2.888 ~ 2.88`, this time successfully. The transaction will end up with `CLValidatorTotalBalance = 57.76 ETH`, `CoverageFundV1.balance = 14 - 2.88 = 11.12 ETH`.

Assuming the rest of the `_pushToRiver` transactions succeed, eventually, only `57.76 - 50 = 7.76 ETH` will be claimed from the by the river contract, leaving the rest `14 - 7.76 = 6.24 ETH` stuck in the coverage fund contract.

Spearbit: Acknowledged, as communicated with the Alluvial team, this issue is less probable since the system is not going to be used in the way described above, i.e., reporting a slashing loss is not intended to be a gradual process, rather, the loss should be reported once coverage funds are ready to be pulled. However, we still want to emphasize that the issue is still possible in certain edge-case scenarios.

7.5.14 Consider removing `coverageFunds` variable and explicitly initialize `executionLayerFees` to zero

Severity: Informational

Context: OracleManager.1.sol#L100-L101

Description: Inside the `OracleManager.setConsensusLayerData` the `coverageFunds` variable is declared but never used. Consider cleaning the code by removing the unused variable.

The `executionLayerFees` variable instead should be explicitly initialized to zero to not rely on compiler assumptions.

Recommendation: Consider removing `coverageFunds` variable and explicitly initialize `executionLayerFees` to zero.

Alluvial: Recommendation has been implemented in PR 168.

Spearbit: Fixed.

7.5.15 Consider renaming `IVestingScheduleManagerV1` interface to `IERC20VestableVotesUpgradeableV1`

Severity: Informational

Context: IVestingScheduleManager.1.sol

Description: The `IVestingScheduleManager` interface contains all the events, errors, and functions that `ERC20VestableVotesUpgradeableV1` needs to implement and use.

Because there's no corresponding `VestingScheduleManager` contract implementation, it would make sense to rename the interface to `IERC20VestableVotesUpgradeableV1`.

Recommendation: Consider renaming `IVestingScheduleManagerV1` interface to `IERC20VestableVotesUpgradeableV1`.

Alluvial: Recommendation has been implemented in PR 172.

Spearbit: Fixed.

7.5.16 Consider renaming `CoverageFundAddress` `COVERAGE_FUND_ADDRESS` to be consistent with the current naming convention

Severity: Informational

Context: CoverageFundAddress.sol#L10

Description: Consider renaming the constant used to access the unstructured storage slot `COVERAGE_FUND_ADDRESS`. To follow the naming convention already adopted across all the contracts, the variable should be renamed to `COVERAGE_FUND_ADDRESS_SLOT`.

Recommendation: Consider renaming `COVERAGE_FUND_ADDRESS` in `CoverageFundAddress` to `COVERAGE_FUND_ADDRESS_SLOT` to be consistent with the already adopted naming convention.

Alluvial: Recommendation has been implemented in PR 168.

Spearbit: Fixed.

7.5.17 Consider reverting if the `msg.value` is zero in `CoverageFundV1.donate`

Severity: Informational

Context: CoverageFund.1.sol#L41-L46

Description: In the current implementation of `CoverageFundV1.donate` there is no check on the `msg.value` value. Because of this, the sender can "spam" the function and emit multiple useless `Donate` events.

Recommendation: Consider reverting early, at the beginning of the function, if `msg.value` is equal to zero.

Alluvial: Recommendation has been implemented in PR 168.

Spearbit: Fixed.

7.5.18 Consider having a separate function in `River` contract that allows `CoverageFundV1` to send funds instead of using the same function used by `ELFeeRecipientV1`

Severity: Informational

Context: CoverageFund.1.sol#L35, River.1.sol#L192-L196

Description: When the `River` contract calls the `CoverageFundV1` contract to pull funds, the `CoverageFundV1` sends funds to `River` by calling `IRiverV1(payable(river)).sendELFees{value: amount}();`.

`sendELFees` is a function that is currently used by both `CoverageFundV1` and `ELFeeRecipientV1`.

```
function sendELFees() external payable {
    if (msg.sender != ELFeeRecipientAddress.get() && msg.sender != CoverageFundAddress.get()) {
        revert LibErrors.Unauthorized(msg.sender);
    }
}
```

It would be cleaner to have a separate function callable **only** by the `CoverageFundV1` contract.

Recommendation: Consider adding to the `River` contract a separate function that allows the `CoverageFundV1` to send ETH. If that function is implemented, remember to also remove the `msg.sender != CoverageFundAddress.get()` from the `sendELFees` implementation.

Alluvial: Recommendation implemented in PR 168.

Spearbit: Fixed.

7.5.19 Extensively document how the Coverage Funds contract works

Severity: Informational

Context: CoverageFund.1.sol

Description: The Coverage Fund contract has a crucial role inside the Protocol, and the current contract's documentation does not properly cover all the needed aspects.

Consider documenting the following aspects:

- General explanation of the Coverage Funds and it's purpose.
- Will donations happen only after a slash/penalty event? Or is there a "budget" that will be dumped on the contract regardless of any slashing events?
- If a donation of `XXX` ETH is made, how is it handled? In a single transaction or distributed over a period of time?
- Explain carefully that when ETH is donated, no shares are minted.
- Explain all the possible market repercussions of the integration of Coverage Funds.
- Is there any off-chain validation process before donating?

- Who are the entities that are enabled to donate to the fund?
- How is the Coverage Funds integrated inside the current Alluvial protocol?
- Any additional information useful for the users, investors, and other actors that interact with the protocol.

Recommendation: Consider extending the current documentation of the `CoverageFund` contract to deeply explain how the coverage funds works and how it interacts with the whole Protocol.

Alluvial: Natspec extended in PR 168.

Spearbit: Fixed.

7.5.20 Missing/wrong natspec comment and typos

Severity: Informational

Context:

- IVestingScheduleManager.1.sol#L48
- IVestingScheduleManager.1.sol#L56
- IVestingScheduleManager.1.sol#L77-L98
- IVestingScheduleManager.1.sol#L111-L114
- VestingSchedules.sol#L37
- VestingSchedules.sol#L82
- ERC20VestableVotesUpgradeable.1.sol#L313-L315
- ERC20VestableVotesUpgradeable.1.sol#L334-L335
- ERC20VestableVotesUpgradeable.1.sol#L389
- Oracle.1.sol#L410-L411
- ICoverageFund.1.sol#L17-L18
- VestingSchedules.sol#L19-L20
- VestingSchedules.sol#L11-L33
- ERC20VestableVotesUpgradeable.1.sol#L41-L42
- ERC20VestableVotesUpgradeable.1.sol#L59-L61
- ERC20VestableVotesUpgradeable.1.sol#L147
- ERC20VestableVotesUpgradeable.1.sol#L156
- ERC20VestableVotesUpgradeable.1.sol#L36-L45

Description:

- **Natspec**
- Missing part of the natspec comment for `/// @notice Attempt to revoke at a` relative to `InvalidRevokedVestingScheduleEnd` in `IVestingScheduleManager`
- Natspec missing the `@return` part for `getVestingSchedule` in `IVestingScheduleManager`.
- Wrong order of natspec `@param` for `createVestingSchedule` in `IVestingScheduleManager`. The `@param _beneficiary` should be placed before `@param _delegatee` to follow the function signature order.
- Natspec missing the `@return` part for `delegateVestingEscrow` in `IVestingScheduleManager`.
- Wrong natspec comment, `operators` should be replaced with `vesting schedules` for `@custom:attribute` of `struct SlotVestingSchedule` in `VestingSchedules`.

- Wrong natspec parameter, replace `operator` with `vesting schedule` in the `VestingSchedules.push` function.
- Missing `@return` natspec for `_delegateVestingEscrow` in `ERC20VestableVotesUpgradeable`.
- Missing `@return` natspec for `_deterministicVestingEscrow` in `ERC20VestableVotesUpgradeable`.
- Missing `@return` natspec for `_getCurrentTime` in `ERC20VestableVotesUpgradeable`.
- Add the Coverage Funds as a source of "extra funds" in the `Oracle._pushToRiver` natspec documentation in `Oracle`.
- Update the `InvalidCall` natspec in `ICoverageFundV1` given that the error is thrown also in the `receive() external payable` function of `CoverageFundV1`.
- Update the natspec of `struct VestingSchedule lockDuration` attribute in `VestingSchedules` by explaining that the lock duration of a vesting schedule could possibly exceed the overall `duration` of the vesting.
- Update the natspec of `lockDuration` in `ERC20VestableVotesUpgradeable` by explaining that the lock duration of a vesting schedule could possibly exceed the overall `duration` of the vesting.
- Consider making the natspec documentation of `struct VestingSchedule` in `VestingSchedules` and the natspec in `ERC20VestableVotesUpgradeable` be in sync.
- Add more examples (variations) to the natspec documentation of the vesting schedules example in `ERC20VestableVotesUpgradeable` to explain all the possible combination of scenarios.
- Make the ERC20VestableVotesUpgradeable natspec documentation about the vesting schedule consistent with the natspec documentation of `_createVestingSchedule` and `VestingSchedules struct VestingSchedule`.
- **Typos**
- Replace all `Overriden` instances with `Overridden` in `River`.
- Replace `transfer` with `transfers` in `ERC20VestableVotesUpgradeable.1.sol#L147`.
- Replace `token` with `tokens` in `ERC20VestableVotesUpgradeable.1.sol#L156`.

Recommendation: Consider adding or updating the relative natspec where needed, and fix the word typos.

Alluvial: Recommendation has been implemented in PR 172.

Spearbit: Fixed.

7.5.21 Different behavior between `River _pullELFees` and `_pullCoverageFunds`

Severity: Informational

Context:

- River.1.sol#L254-L265
- River.1.sol#L270-L283

Description: Both `_pullELFees` and `_pullCoverageFunds` implement the same functionality:

- Pull funds from a contract address.
- Update the balance storage variable.
- Emit an event.
- Return the amount of balance collected from the contract.

The `_pullCoverageFunds` differs from the `_pullELFees` implementation by avoiding both updating the `BalanceToDeposit` when `collectedCoverageFunds == 0` and emitting the `PulledCoverageFunds` event.

Because they are implementing the same functionality, they should follow the same behavior if there is not an explicit reason to not do so.

Recommendation: Consider applying the same behavior to both `_pullELFees` and `_pullCoverageFunds` or explain which is the reason why those should be different.

Consider emitting the `PulledCoverageFunds` event even in case `collectedCoverageFunds == 0` if you think that it should be an event to be monitored even in case no funds were pulled from the contract.

Spearbit: With PR 168 Alluvial has changed `_pullELFees` to follow the same behavior of `_pullCoverageFunds`. With the new implementation, both functions do not update the unstructured storage variable value and fire the event if the collected funds are equal to zero.

Alluvial has acknowledged that with the current implementation, they are not emitting and monitoring those cases where the `OracleManager` requests non-zero funds to be pulled but in EL Fees or Coverage Funds are no funds to be pulled.

7.5.22 Move local mask variable from `Allowlist.1.sol` to `LibAllowlistMasks.sol`

Severity: Informational

Context: Allowlist.1.sol#L21, LibAllowlistMasks.sol

Description: `LibAllowlistMasks.sol` is meant to contain all mask values, but `DENY_MASK` is a local variable in the `Allowlist.1.sol` contract.

Recommendation: Move `DENY_MASK` variable to `LibAllowMasks.sol` and make necessary changes to `Allowlist.1.sol`.

Alluvial: Very good point and clearly a miss from our end. Fixed in PR 166.

Spearbit: Fixed.

7.5.23 Consider adding additional parameters to the existing events to improve filtering/monitoring

Severity: Informational

Context:

- IVestingScheduleManager.1.sol#L15
- IVestingScheduleManager.1.sol#L20
- IVestingScheduleManager.1.sol#L25
- IVestingScheduleManager.1.sol#L31

Description: Some already defined events could be improved by adding more parameters to better track those events in dApps or monitoring tools.

- Consider adding `address indexed delegatee` as an event's parameter to `event CreatedVestingSchedule`. While it's true that after the vest/lock period the `beneficiary` will be the owner of those tokens, in the meanwhile (if `_delegatee != address(0)`) the voting power of all those vested tokens are delegated to the `_delegatee`.
- Consider adding `address indexed beneficiary` to `event ReleasedVestingSchedule`.
- Consider adding `uint256 newEnd` to `event RevokedVestingSchedule` to track the updated `end` of the vesting schedule.
- Consider adding `address indexed beneficiary` to `event DelegatedVestingEscrow`.

If those events parameters are added to the events, the Alluvial team should also remember to update the relative natspec documentation.

Recommendation: Consider adding the suggested parameters to the relative events and updated the natspec documentation where needed.

Spearbit: In PR 172, Alluvial has implemented part of the recommendations:

- Added additional parameter `uint256 newEnd` to the `event RevokedVestingSchedule`.
- Added additional parameter `address indexed beneficiary` to the `event DelegatedVestingEscrow`.

7.5.24 Missing `indexed` keyword in events parameters

Severity: Informational

Context:

- IRiver.1.sol#L27
- IVestingScheduleManager.1.sol#L31
- ICoverageFund.1.sol#L15

Description: Some events parameters are missing the `indexed` keyword. Indexing specific parameters is particularly important to later be able to filter those events both in dApps or monitoring tools.

- `coverageFund` event parameter should be declared as `indexed` in `event SetCoverageFund`.
- Both `oldDelegatee` and `newDelegatee` should be `indexed` in `event DelegatedVestingEscrow`.
- `donator` should be declared as `indexed` in `event Donate`.

Recommendation: Declare the specified event parameters as `indexed` where needed.

Alluvial: Recommendation has been implemented in PR 168 and PR 172.

Spearbit: Fixed.

7.5.25 Add natspec documentation to the `TLC` contract

Severity: Informational

Context: TLC.1.sol

Description: The current implementation of `TLC` contract is missing natspec at the root level to explain the contract. The natspec should cover the basic explanation of the contract (like it has already been done in other contracts like River.sol) but also illustrate

- TLC token has a fixed max supply that is minted at deploy time.
- All the minted tokens are sent to a single account at deploy time.
- How TLC token will be distributed.
- How voting power works (you have to delegate to yourself to gain voting power).
- How the vesting process works.
- Other general information useful for the user/investor that receives the TLC token directly or vested.

Recommendation: Add natspec documentation to the `TLC` contract to explain what the contract does and how TLC tokens are minted, distributed, and used.

Alluvial: Recommendation has been implemented in PR 172.

Spearbit: Fixed.