Streamr Network Incentive Layer Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditor

Hans

November 3, 2023

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

The Streamr Network is a peer-to-peer network for publishing and subscribing to data in real-time. Applications use it for decentralized messaging, for example sharing data across applications or broadcasting real-time state changes to large audiences. The decentralized nature of the system makes the data transport scalable, robust, secure, tamper proof, and censorship resistant.

5 Audit Scope

All Solidity source files inside the `network-contracts/contracts/OperatorTokenomics` were in scope except test contracts in `network-contracts/contracts/OperatorTokenomics/testcontracts`.

The initial review took place between October 9th and October 23rd, while the mitigation review occurred between October 30th and November 2nd.

6 Executive Summary

Over the course of 19 days, the Cyfrin team conducted an audit on the Streamr Network Incentive Layer smart contracts provided by Streamr. In this period, a total of 14 issues were found.

Summary

Project Name	Streamr Network Incentive Layer
Repository	network-contracts
Commit	da7a07b1f562...
Audit Timeline	Oct 9th - Nov 2nd
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	4
Medium Risk	6
Low Risk	1
Informational	3
Gas Optimizations	0
Total Issues	14

Summary of Findings

[H 7.1.1] `VoteKickPolicy._endVote()` might revert forever due to underflow	Resolved
[H 7.1.2] Possible overflow in `_payOutFirstInQueue`	Resolved
[H 7.1.3] Wrong validation in `DefaultUndelegationPolicy.onUndelegate()`	Resolved
[H 7.1.4] Malicious target can make `_endVote()` revert forever by forceUnstaking/staking again	Resolved
[M 7.2.1] In VoteKickPolicy.onFlag(), targetStakeAtRiskWei[target] might be greater than stakedWei[target] and _endVote() will revert.	Resolved
[M 7.2.2] Possible front running of `flag()`	Acknowledged
[M 7.2.3] Operators can bypass a `leavePenalty` using `reduceStakeTo()`	Resolved
[M 7.2.4] In `Operator._transfer()`, `onDelegate()` should be called after updating the token balances	Resolved
[M 7.2.5] Centralization risk	Acknowledged
[M 7.2.6] `onTokenTransfer` does not validate if the call is from the DATA token contract	Resolved
[L 7.3.1] Unsafe use of ‘transfer()/transferFrom()‘ with ‘IERC20‘	Acknowledged
[I 7.4.1] Redundant requirement	Acknowledged
[I 7.4.2] Variables need not be initialized to zero	Resolved
[I 7.4.3] Event is not properly ‘indexed‘	Resolved

7 Findings

7.1 High Risk

7.1.1 `VoteKickPolicy._endVote()` might revert forever due to underflow

Severity: High

Description: In `onFlag()`, `targetStakeAtRiskWei[target]` might be less than the total rewards for the flagger/reviewers due to rounding.

```
File: contracts\OperatorTokenomics\StreamrConfig.sol
22:     /**
23:      * Minimum amount to pay reviewers+flagger
24:      * That is: minimumStakeWei >= (flaggerRewardWei + flagReviewerCount * flagReviewerRewardWei) /
↪  slashingFraction
25:      */
26:     function minimumStakeWei() public view returns (uint) {
27:         return (flaggerRewardWei + flagReviewerCount * flagReviewerRewardWei) * 1 ether /
↪  slashingFraction;
28:     }
```

- Let's assume `flaggerRewardWei + flagReviewerCount * flagReviewerRewardWei = 100`, `StreamrConfig.slashingFraction = 0.03e18(3%)`, `minimumStakeWei() = 1000 * 1e18 / 0.03e18 = 10000 / 3 = 3333`.
- If we suppose `stakedWei[target] = streamrConfig.minimumStakeWei()`, then `targetStakeAtRiskWei[target] = 3333 * 0.03e18 / 1e18 = 99.99 = 99`.
- As a result, `targetStakeAtRiskWei[target]` is less than total rewards(=100), and `_endVote()` will revert during the reward distribution due to underflow.

The above scenario is possible only when there is a rounding during `minimumStakeWei` calculation. So it works properly with the default `slashingFraction = 10%`.

Impact: The `VoteKickPolicy` wouldn't work as expected and malicious operators won't be kicked forever.

Recommended Mitigation: Always round the `minimumStakeWei()` up.

Client: Fixed in commit 615b531.

Cyfrin: Verified.

7.1.2 Possible overflow in `_payOutFirstInQueue`

Severity: High

Description: In `_payOutFirstInQueue()`, possible revert during `operatorTokenToDataInverse()`.

```
uint amountOperatorTokens = moduleCall(address(exchangeRatePolicy),
↪  abi.encodeWithSelector(exchangeRatePolicy.operatorTokenToDataInverse.selector, amountDataWei));
```

If a delegator calls `undelegate()` with `type(uint256).max`, `operatorTokenToDataInverse()` will revert due to uint overflow and the queue logic will be broken forever.

```
    function operatorTokenToDataInverse(uint dataWei) external view returns (uint operatorTokenWei) {
        return dataWei * this.totalSupply() / valueWithoutEarnings();
    }
```

Impact: The queue logic will be broken forever because `_payOutFirstInQueue()` keeps reverting.

Recommended Mitigation: We should cap `amountDataWei` before calling `operatorTokenToDataInverse()`.

Client: Fixed in commit c62e5d9.

Cyfrin: Verified.

7.1.3 Wrong validation in `DefaultUndelegationPolicy.onUndelegate()`

Severity: High

Description: In `onUndelegate()`, it checks if the operator owner still holds at least `minimumSelfDelegationFraction` of total supply.

```
function onUndelegate(address delegator, uint amount) external {
    // limitation only applies to the operator, others can always undelegate
    if (delegator != owner) { return; }

    uint actualAmount = amount < balanceOf(owner) ? amount : balanceOf(owner); //@audit amount:DATA,
    ↪  balanceOf:Operator
    uint balanceAfter = balanceOf(owner) - actualAmount;
    uint totalSupplyAfter = totalSupply() - actualAmount;
    require(1 ether * balanceAfter >= totalSupplyAfter *
    ↪  streamrConfig.minimumSelfDelegationFraction(), "error_selfDelegationTooLow");
}
```

But `amount` means the DATA token amount and `balanceOf(owner)` indicates the `Operator` token balance and it's impossible to compare them directly.

Impact: The operator owner wouldn't be able to undelegate because `onUndelegate()` works unexpectedly.

Recommended Mitigation: `onUndelegate()` should compare amounts after converting to the same token.

Client: Fixed in commit 9b8c65e.

Cyfrin: Verified.

7.1.4 Malicious target can make `_endVote()` revert forever by forceUnstaking/staking again

Severity: High

Description: In `_endVote()`, we update `forfeitedStakeWei` or `lockedStakeWei[target]` according to the `target`'s staking status.

```
File: contracts\OperatorTokenomics\SponsorshipPolicies\VoteKickPolicy.sol
179:     function _endVote(address target) internal {
180:         address flagger = flaggerAddress[target];
181:         bool flaggerIsGone = stakedWei[flagger] == 0;
182:         bool targetIsGone = stakedWei[target] == 0;
183:         uint reviewerCount = reviewers[target].length;
184:
185:         // release stake locks before vote resolution so that slashings and kickings during
↪  resolution aren't affected
186:         // if either the flagger or the target has forceUnstaked or been kicked, the
↪  lockedStakeWei was moved to forfeitedStakeWei
187:         if (flaggerIsGone) {
188:             forfeitedStakeWei -= flagStakeWei[target];
189:         } else {
190:             lockedStakeWei[flagger] -= flagStakeWei[target];
191:         }
192:         if (targetIsGone) {
193:             forfeitedStakeWei -= targetStakeAtRiskWei[target];
194:         } else {
195:             lockedStakeWei[target] -= targetStakeAtRiskWei[target]; //@audit revert after
↪  forceUnstake() => stake() again
196:         }
```

We consider the target is still active if he has a positive staking amount. But we don't know if he has unstaked and staked again, so the below scenario would be possible.

- The target staked 100 amount and a flagger reported him.
- In `onFlag()`, `lockedStakeWei[target] = targetStakeAtRiskWei[target] = 100`.
- During the voting period, the target called `forceUnstake()`. Then `lockedStakeWei[target]` was reset to 0 in `Sponsorship._removeOperator()`.
- After that, he stakes again and `_endVote()` will revert forever at L195 due to underflow.

After all, he won't be flagged again because the current flagging won't be finalized.

Furthermore, malicious operators would manipulate the above state by themselves to earn operator rewards without any risks.

Impact: Malicious operators can bypass the flagging system by reverting `_endVote()` forever.

Recommended Mitigation: Perform stake unlocks in `_endVote()` without relying on the current staking amounts.

Client: Fixed in commit 8be1d7e.

Cyfrin: Verified.

7.2 Medium Risk

7.2.1 In `VoteKickPolicy.onFlag()`, `targetStakeAtRiskWei[target]` might be greater than `stakedWei[target]` and `_endVote()` would revert.

Severity: Medium

Description: `targetStakeAtRiskWei[target]` might be greater than `stakedWei[target]` in `onFlag()`.

```
targetStakeAtRiskWei[target] = max(stakedWei[target], streamrConfig.minimumStakeWei()) *
↪  streamrConfig.slashingFraction() / 1 ether;
```

For example,

- At the first time, `streamrConfig.minimumStakeWei() = 100` and an operator(=target) has staked 100.
- `streamrConfig.minimumStakeWei()` was increased to 2000 after a reconfiguration.
- `onFlag()` is called for target and `targetStakeAtRiskWei[target]` will be `max(100, 2000) * 10% = 200`.
- In `_endVote()`, `slashingWei = _kick(target, slashingWei)` will be 100 because target has staked 100 only.
- So it will revert due to underflow during the reward distribution.

Impact: Operators with small staked funds wouldn't be kicked forever.

Recommended Mitigation: `onFlag()` should check if a target has staked enough funds for rewards and handle separately if not.

Client: Fixed in commit 05d9716. Flag targets with not enough stake (to pay for the review) will be kicked out without review. Since this can only happen after the admin changes the minimum stake requirement (e.g. by increasing reviewer rewards), the flag target is not at fault and will not be slashed. They can stake back again with the new minimum stake if they want.

Cyfrin: Verified.

7.2.2 Possible front running of `flag()`

Severity: Medium

Description: The `target` might call `unstake()/forceUnstake()` before a flagger calls `flag()` to avoid a possible fund loss. Also, there would be no slash during the unstaking for `target` when it meets the `penaltyPeriodSeconds` requirement.

```
File: contracts\OperatorTokenomics\SponsorshipPolicies\VoteKickPolicy.sol
65:     function onFlag(address target, address flagger) external {
66:         require(flagger != target, "error_cannotFlagSelf");
67:         require(voteStartTimestamp[target] == 0 && block.timestamp >
↪  protectionEndTimestamp[target], "error_cannotFlagAgain"); // solhint-disable-line not-rely-on-time
68:         require(stakedWei[flagger] >= minimumStakeOf(flagger), "error_notEnoughStake");
69:         require(stakedWei[target] > 0, "error_flagTargetNotStaked"); //@audit possible front run
70:
```

Impact: A malicious target would bypass the kick policy by front running.

Recommended Mitigation: There is no straightforward mitigation but we could implement a kind of `delayed unstaking` logic for some percent of staking funds.

Client: Our current threat model is a staker who doesn't run a Streamr node. They could be a person using Metamask to do all smart contract transactions via our UI, or they could be a complex flashbot MEV searcher. But if they're not running Streamr nodes, they should be found out, flagged, and kicked out by the honest nodes.

While an advanced bot could stake and listen to `Flagged` events, if they're found out and flagged before their minimum stay (`DefaultLeavePolicy.penaltyPeriodSeconds`) is over, their stake would still get slashed even if they front-run the flagging. We aim to select our network parameters so that it will be very likely that someone staking but not actually running a Streamr node would get flagged during those `penaltyPeriodSeconds`. Then front-running the flagging wouldn't save them from slashing.

Cyfrin: Acknowledged.

7.2.3 Operators can bypass a `leavePenalty` using `reduceStakeTo()`

Severity: Medium

Description: Operators should pay a leave penalty when they unstake earlier than expected. But there are no relevant requirements in `reduceStakeTo()` so they can reduce their staking amount to the minimum value.

- An operator staked 100 and he wants to unstake earlier.
- When he calls `forceUnstake()`, he should pay `100 * 10% = 10` as a penalty.
- But if he reduces the staking amount to the minimum(like 10) using `reduceStakeTo()` first and calls `forceUnstake()`, the penalty will be `10 * 10% = 1`.

Impact: Operators will pay a `leavePenalty` for the minimum amount only.

Recommended Mitigation: The penalty should be the same, whether an Operator only calls `forceUnstake`, or first calls `reduceStakeTo`.

Client: Fixed in commit 72323d0.

Cyfrin: Verified

7.2.4 In `Operator._transfer()`, `onDelegate()` should be called after updating the token balances

Severity: Medium

Description: In `_transfer()`, `onDelegate()` is called to validate the owner's `minimumSelfDelegationFraction` requirement.

```
File: contracts\OperatorTokenomics\Operator.sol
324:         // transfer creates a new delegator: check if the delegation policy allows this
↪  "delegation"
325:         if (balanceOf(to) == 0) {
326:             if (address(delegationPolicy) != address(0)) {
327:                 moduleCall(address(delegationPolicy),
↪  abi.encodeWithSelector(delegationPolicy.onDelegate.selector, to)); //@audit
should be called after _transfer()
328:             }
329:         }
330:
331:         super._transfer(from, to, amount);
332:
```

But `onDelegate()` is called before updating the token balances and the below scenario would be possible.

- The operator owner has 100 shares(required minimum fraction). And there are no undelegation policies.
- Logically, the owner shouldn't be able to transfer his 100 shares to a new delegator due to the min fraction requirement in `onDelegate()`.
- But if the owner calls `transfer(owner, to, 100)`, `balanceOf(owner)` will be 100 in `onDelegation()` and it will pass the requirement because it's called before `super._transfer()`.

Impact: The operator owner might transfer his shares to other delegators in anticipation of slashing, to avoid slashing.

Recommended Mitigation: `onDelegate()` should be called after `super._transfer()`.

Client: Fixed in commit 93d6105.

Cyfrin: Verified.

7.2.5 Centralization risk

Severity: Medium

Description: The protocol has a `DEFAULT_ADMIN_ROLE` with privileged rights to perform admin tasks that can affect users. Especially, the owner can change the fee/reward fraction settings and various policies.

Most admin functions don't emit events at the moment.

Impact: While the protocol owner is regarded as a trusted party, the owner can change many settings and policies without logging. This might lead to unexpected results and users might be affected.

Recommended Mitigation: Specify the owner's privileges and responsibilities in the documentation. Add constant state variables that can be used as the minimum and maximum values for the fraction settings. Log the changes in the important state variables via events.

Client: Logging added to StreamrConfig in commit c530ec5. Better documentation and more logging added to other contracts in commit c343850. Those commits partially mitigate risks associated with leaking of the admin key.

In StreamrConfig, there isn't much difference in the power to change the config values, and in replacing the whole contract (it's upgradeable). Some maximum and minimum limits exist currently, but their main point is to sanity-check new values, especially the initial values. "Binding our hands" with tighter limits wouldn't thus really change anything, at best it would signal an intent.

Before using these admin powers to change config values or amend the contracts using upgrades, wider review (community, auditors) will be needed, to avoid unexpected side-effects that may affect users. The day-to-day is not designed to require any admin intervention. Admin powers are only needed for unforeseen circumstances (e.g. hotfixing bugs) or planned policy changes. There is no foreseeable need for such changes at the moment.

Cyfrin: Acknowledged.

7.2.6 `onTokenTransfer` does not validate if the call is from the DATA token contract

Severity: Medium

Description: `SponsorshipFactory::onTokenTransfer` and `OperatorFactory::onTokenTransfer` are used to handle the token transfer and contract deployment in a single transaction. But there is no validation that the call is from the DATA token contract and anyone can call these functions.

The impact is low for `Sponsorship` deployment, but for `Operator` deployment, `ClonesUpgradeable.cloneDeterministic` is used with a salt based on the operator token name and the operator address. An attacker can abuse this to cause DoS for deployment.

We see that this validation is implemented correctly in other contracts like `Operator`.

```
        if (msg.sender != address(token)) {
            revert AccessDeniedDATATokenOnly();
        }
```

Impact: Attackers can prevent the deployment of `Operator` contracts.

Recommended Mitigation: Add a validation to ensure the caller is the actual DATA contract.

Client: Fixed in commit 8b13df4.

Cyfrin: Verified.

7.3 Low Risk

7.3.1 Unsafe use of `transfer()/transferFrom()` with `IERC20`

Some tokens do not implement the ERC20 standard properly but are still accepted by most code that accepts ERC20 tokens. For example Tether (USDT)'s `transfer()` and `transferFrom()` functions on L1 do not return booleans as the specification requires, and instead have no return value. Consider using OpenZeppelin's `SafeERC20`'s `safeTransfer()/safeTransferFrom()` instead

```
File: contracts\OperatorTokenomics\Operator.sol
264:         token.transferFrom(_msgSender(), address(this), amountWei);
442:         token.transfer(msgSender, rewardDataWei);

File: contracts\OperatorTokenomics\Sponsorship.sol
187:         token.transferFrom(_msgSender(), address(this), amountWei);
215:         token.transferFrom(_msgSender(), address(this), amountWei);
261:         token.transfer(streamrConfig.protocolFeeBeneficiary(), slashedWei);
273:         token.transfer(operator, payoutWei);

File: contracts\OperatorTokenomics\OperatorPolicies\QueueModule.sol
86:          token.transfer(delegator, amountDataWei);

File: contracts\OperatorTokenomics\OperatorPolicies\StakeModule.sol
128:         token.transfer(streamrConfig.protocolFeeBeneficiary(), protocolFee);
```

Client: We will only use DATA token in our system. It doesn't have the above methods. So: `transfer` it is.

Cyfrin: Acknowledged.

7.4 Informational Findings

7.4.1 Redundant requirement

The first requirement is redundant because the second one is enough.

```
File: contracts\OperatorTokenomics\SponsorshipPolicies\VoteKickPolicy.sol
156:         require(reviewerState[target][voter] != Reviewer.NOT_SELECTED, "error_reviewersOnly");
↪  //@audit-issue redundant
157:         require(reviewerState[target][voter] == Reviewer.IS_SELECTED, "error_alreadyVoted");
```

Client: We want to give an informative error message for the case where a non-reviewer tries to vote. So: prefer to keep it.

Cyfrin: Acknowledged.

7.4.2 Variables need not be initialized to zero

The default value for variables is zero, so initializing them to zero is redundant.

```
File: contracts\OperatorTokenomics\SponsorshipPolicies\DefaultLeavePolicy.sol
10:           uint public penaltyPeriodSeconds = 0;

File: contracts\OperatorTokenomics\SponsorshipPolicies\VoteKickPolicy.sol
100:         uint sameSponsorshipPeerCount = 0;
226:         uint rewardsWei = 0;
```

Client: Fixed in commit c847fab.

Cyfrin: Verified.

7.4.3 Event is not properly `indexed`

Index event fields make the field more quickly accessible to off-chain tools that parse events. This is especially useful when it comes to filtering based on an address. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Where applicable, each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three applicable fields, all of the applicable fields should be indexed.

Client: Fixed in commit 92d4145.

Cyfrin: Verified.

8 Appendix

8.1 Reported Issues

The Streamr team discovered some vulnerabilities after the audit had started. The Cyfrin team has included the self-reported issues in the review.

8.1.1 No validation in `Operator.updateOperatorsCutFraction()`

Description: `Operator.updateOperatorsCutFraction()` doesn't check the argument (expected to be less than "1 ether" or 1e18).

```
File: contracts\OperatorTokenomics\Operator.sol
417:     function updateOperatorsCutFraction(uint newOperatorsCutFraction) external onlyOperator {
418:         if (totalStakedIntoSponsorshipsWei > 0) {
419:             revert StakedInSponsorships();
420:         }
421:
422:         operatorsCutFraction = newOperatorsCutFraction;
423:         emit MetadataUpdated(metadata, _msgSender(), newOperatorsCutFraction);
424:     }
```

Client: Fixed in commit d6ae2a6.

Cyfrin: Verified.

8.1.2 No minimum delegation amount in `_delegate()`

Description: We don't check for minimum delegation amount in `_delegate`. We do check it in `_transfer`, also when processing the undelegation queue.

```
File: contracts\OperatorTokenomics\Operator.sol
274:     function _delegate(address delegator, uint amountDataWei) internal {
275:         uint amountOperatorToken = moduleCall(address(exchangeRatePolicy),
↪  abi.encodeWithSelector(exchangeRatePolicy.dataToOperatorToken.selector, amountDataWei,
↪  amountDataWei));
276:         _mint(delegator, amountOperatorToken);
277:
278:         // check if the delegation policy allows this delegation
279:         if (address(delegationPolicy) != address(0)) {
280:             moduleCall(address(delegationPolicy),
↪  abi.encodeWithSelector(delegationPolicy.onDelegate.selector, delegator));
281:         }
282:
283:         emit Delegated(delegator, amountDataWei);
284:         emit BalanceUpdate(delegator, balanceOf(delegator), totalSupply());
285:         emit OperatorValueUpdate(totalStakedIntoSponsorshipsWei - totalSlashedInSponsorshipsWei,
↪  token.balanceOf(address(this)));
286:     }
```

Client: Fixed in commit b1b9d19.

Cyfrin: Verified.

8.1.3 Possible division by zero

Description: It's possible to trigger division by zero in `operatorTokenToData` like this:

- Create a new Operator
- Send in tokens using ERC-20 (NOT transferAndCall)

- Go to the undelegation queue (we don't check if you're a delegator; which is fine, all the action happens in the front of the queue during payout only)
- Activate the queue payment

Fix by adding back the guard that we removed during branch coverage work, because we thought the condition was impossible to trigger.

Client: Fixed in commit 5740ee5.

Cyfrin: Verified.

8.1.4 Lack of validation

Description: `SponsorshipFactory` only checks that a given (sponsored) stream exists in one of the two code paths when deploying sponsorships. Going to fix that by putting the check on the common code path.

Client: Fixed in commit db76b0c.

Cyfrin: Verified.

8.1.5 Redundant `payable` modifier

Description: There was a random payable left in one of the functions (`Operator.delegate()`). It will be removed, there's no reason for any of our functions to be "payable" since we aren't dealing with native tokens.

Client: Fixed in commit abc8a8b.

Cyfrin: Verified.

8.1.6 First depositor issue in `Operator`

Description: The operator owner can steal DATA by evading minimum delegation limits and exploiting rounding errors

- Create new Operator.
- Send in tokens using ERC-20 (e.g. 1000 DATA).
- Create a Sponsorship that pays 1 wei / second.
- Stake to that sponsorship.
- Withdraw very soon, receive ~10 wei.
- Initial exchange rate is 1:1, so totalSupply becomes 10 wei.
- `valueWithoutEarnings()` is still ~1000 DATA.
- Exchange rate is now ~10 : 1000e18
- New delegators send DATA to the contract. Here could also be some rounding errors problem: if anyone delegates < ~100 DATA, they won't get ANY operator tokens.
- Owner can undelegate < ~100 DATA. Corresponds to 0 operator tokens, so 0 get burned, but 100 DATA gets paid out. Repeat.

Will fix by burning AT LEAST so many operator tokens that the amount of DATA is covered (i.e. rounding UP instead of down). Also staking will be disabled when there's no self-delegation. This prevents operating the contract before there are operator tokens in existence, blocking this path to rounding error exploits via absurd exchange rates.

Client: Fixed in commit 8dc015f.

Cyfrin: Verified.

8.2 Mitigation Review

There were some new risks introduced during the mitigation process. The Streamr team mitigated the newly introduced issues.

8.2.1 In `_handleNoKick()`, it slashes the flagger although he has forceUnstaked already.

Description: In `_handleNoKick()`, it shouldn't slash the flagger if he has forceUnstaked already. It's because the flagger has paid the penalty while forceUnstaking.

```
File: contracts\OperatorTokenomics\SponsorshipPolicies\VoteKickPolicy.sol
330:         // Unlock the flagger's stake. Slash just enough to cover the rewards, the rest will be
↪  unlocked = released
331:         uint flagStake = flagStakeWei[target];
332:         if (lockedStakeWei[flagger] >= flagStake) {
333:             lockedStakeWei[flagger] -= flagStake;
334:         } else {
335:             //...unless flagger has forceUnstaked or been kicked, in which case the locked
↪  flag-stake was moved into forfeited stake
336:             forfeitedStakeWei -= flagStake - lockedStakeWei[flagger];
337:             lockedStakeWei[flagger] = 0;
338:             leftoverWei += flagStake - rewardsWei;
339:         }
340:         if (stakedWei[flagger] > 0) {
341:             _slash(flagger, rewardsWei); //@audit shouldn't slash if forceUnstaked
342:             emit StakeLockUpdate(flagger, lockedStakeWei[flagger], getMinimumStakeOf(flagger));
343:         }
```

We should slash inside the `if (lockedStakeWei[flagger] >= flagStake)` block only.

Client: Fixed in commit 8fb3ac0.

Cyfrin: Verified.

8.2.2 `OperatorFactory.voters` might contain stale voters.

Description: `OperatorFactory.updateStake()` is called only when the operator changes the staked amount. So if operators don't change the staked amount for a while, voters won't be updated promptly.

- An operator staked enough amounts but couldn't be a voter due to `minEligibleVoterAge`. He won't be a voter until he changes the staked amount.
- With an increased `totalStakedWei`, a voter is not eligible anymore due to the `minEligibleVoterFractionOfAllStake`, but he will be a voter anyway.

```
File: contracts\OperatorTokenomics\OperatorFactory.sol
236:     function updateStake(uint newStakeWei) external { //@audit might be stale
237:         address operator = msg.sender;
238:         if (deploymentTimestamp[operator] == 0) { revert OnlyOperators(); }
239:
240:         totalStakedWei = totalStakedWei + newStakeWei - stakedWei[operator];
241:         stakedWei[operator] = newStakeWei;
242:
243:         uint voterThreshold = totalStakedWei * streamrConfig.minEligibleVoterFractionOfAllStake()
↪  / 1 ether;
244:         bool isEligible = newStakeWei >= voterThreshold && deploymentTimestamp[operator] +
↪  streamrConfig.minEligibleVoterAge() < block.timestamp; // solhint-disable-line not-rely-on-time
245:
246:         if (isEligible && votersIndex[operator] == 0) {
247:             voters.push(operator);
248:             votersIndex[operator] = voters.length; // real index + 1
249:             emit VoterUpdate(operator, true);
250:         }
251:
252:         if (!isEligible && votersIndex[operator] > 0) {
253:             uint index = votersIndex[operator] - 1; // real index = votersIndex - 1
254:             address lastOperator = voters[voters.length - 1];
255:             voters[index] = lastOperator;
256:             voters.pop();
257:             votersIndex[lastOperator] = index + 1; // real index + 1
258:             delete votersIndex[operator];
259:             emit VoterUpdate(operator, false);
260:         }
261:     }
```

Client: Fixed in commit 68dea4d.

Cyfrin: Verified.

8.2.3 In `_handleKick()`, a flagger wouldn't receive a flagger reward if he has forceUnstaked/staked

Description: `_handleKick()` checks flaggers with `lockedStakeWei` mapping and the below scenario would be possible.

- Alice got flagged(e.g. Flag1) by other one and `lockedStakeWei[Alice] = targetStakeAtRiskWei[Alice] = 100`.
- For some reason, Alice has forceUnstaked and staked again.
- After that, Alice noticed a bad operator and flagged(e.g. Flag2) him. So `lockedStakeWei[Alice] = 100` again.
- In `_handleKick()` for Flag1, Alice will be kicked at `L260` and `lockedStakeWei[Alice] = 0`.
- In `_handleKick()` for Flag2, it will go to `L277`(else block) and she will be considered as forceUnstaked(so no rewards) although she's eligible to receive a reward.

```
File: contracts\OperatorTokenomics\SponsorshipPolicies\VoteKickPolicy.sol
256:         // Take the slashing from target's locked stake...
257:         uint slashingWei = targetStakeAtRiskWei[target];
258:         if (lockedStakeWei[target] >= slashingWei) {
259:             lockedStakeWei[target] -= slashingWei;
260:             _kick(target, slashingWei); // ignore return value, there should be enough (now
↪  unlocked) stake to slash
261:         } else {
262:             //...unless target has forceUnstaked, in which case the locked stake was moved into
↪  forfeited stake, and they already paid for the KICK
263:             forfeitedStakeWei -= slashingWei - lockedStakeWei[target];
264:             lockedStakeWei[target] = 0;
265:             if (stakedWei[target] > 0) {
266:                 emit StakeLockUpdate(target, lockedStakeWei[target], getMinimumStakeOf(target));
267:             }
268:         }
269:
270:         // Unlock the flagger's stake and pay them from the slashed stake
271:         uint flagStake = flagStakeWei[target];
272:         if (lockedStakeWei[flagger] >= flagStake) {
273:             lockedStakeWei[flagger] -= flagStake;
274:             slashingWei -= safeSendRewards(flagger, flaggerRewardWei[target]);
275:         } else {
276:             //...unless flagger has forceUnstaked or been kicked; so unlock the remaining part
↪  from forfeitedStake
277:             forfeitedStakeWei -= flagStake - lockedStakeWei[flagger];
278:             lockedStakeWei[flagger] = 0;
279:             leftoverWei += flagStake;
280:         }
281:         if (stakedWei[flagger] > 0) {
282:             emit StakeLockUpdate(flagger, lockedStakeWei[flagger], getMinimumStakeOf(flagger));
283:         }
```

Client: This is by design: if flagger has been removed by the time the flag ends, they've lost their right to the flagger rewards.

Cyfrin: Acknowledged.