



Streaming Protocol contest Findings & Analysis Report

2022-02-11

Table of contents

- [Overview](#)
 - [About C4](#)
 - [Wardens](#)
- [Summary](#)
- [Scope](#)
- [Severity Criteria](#)
- [High Risk Findings \(10\)](#)
 - [\[H-01\] Wrong calculation of excess depositToken allows stream creator to retrieve `depositTokenFlashloanFeeAmount` , which may cause fund loss to users](#)
 - [\[H-02\] Tokens can be stolen when `depositToken == rewardToken`](#)
 - [\[H-03\] Reward token not correctly recovered](#)
 - [\[H-04\] Improper implementation of `arbitraryCall\(\)` allows protocol gov to steal funds from users' wallets](#)
 - [\[H-05\] Possible incentive theft through the `arbitraryCall\(\)` function](#)
 - [\[H-06\] Creating rewardTokens without streaming depositTokens](#)
 - [\[H-07\] Business logic bug in `__abdicate\(\)` function - 2 Bugs](#)

- [\[H-08\] ts.tokens sometimes calculated incorrectly](#)
- [\[H-09\] DOS while dealing with erc20 when value\(i.e amount*decimals\) is high but less than type\(uint112\).max](#)
- [\[H-10\] recoverTokens doesn't work when isSale is true](#)
- [Medium Risk Findings \(5\)](#)
 - [\[M-01\] LockeERC20 is vulnerable to frontrun attack](#)
 - [\[M-02\] Any arbitraryCall gathered airdrop can be stolen with recoverTokens](#)
 - [\[M-03\] This protocol doesn't support all fee on transfer tokens](#)
 - [\[M-04\] arbitraryCall\(\) can get blocked by an attacker](#)
 - [\[M-05\] Storage variable unstreamed can be artificially inflated](#)
- [Low Risk Findings \(27\)](#)
- [Non-Critical Findings \(23\)](#)
- [Gas Optimizations \(53\)](#)
- [Disclosures](#)



Overview



About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the code contest outlined in this document, C4 conducted an analysis of Streaming Protocol contest smart contract system written in Solidity. The code contest took place between November 30—December 7 2021.



Wardens

38 Wardens contributed reports to the Streaming Protocol contest:

1. [hack3r-Om](#)
2. bitbopper
3. WatchPug ([jtp](#) and [ming](#))
4. [gpersoon](#)
5. hyh
6. [cyberboy](#)
7. [MetaOxNull](#)
8. [kenzo](#)
9. OxOxOx
10. Jujic
11. pedroais
12. [cmichel](#)
13. [defsec](#)
14. [gzeon](#)
15. [pauliax](#)
16. [GiveMeTestEther](#)
17. GeekyLumberjack
18. ScopeLift ([wildmolasses](#), [bendi](#), and [mds1](#))
19. harleythedog
20. Ox1f8b
21. [Ruhum](#)
22. hubble (ksk2345 and shri4net)
23. [wuwe1](#)
24. [itsmeSTYJ](#)
25. [jonah1005](#)
26. [toastedsteaksandwich](#)
27. [Omik](#)
28. jayjonah8
29. egjlmn1

30. robe

31. [csanuragjain](#)

32. mtz

33. [yeOlde](#)

34. [pmerkleplant](#)

35. [danb](#)

36. pants

This contest was judged by [Oxean](#).

Final report assembled by [itsmetechjay](#) and [CloudEllie](#).



Summary

The C4 analysis yielded an aggregated total of 42 unique vulnerabilities and 118 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 10 received a risk rating in the category of HIGH severity, 5 received a risk rating in the category of MEDIUM severity, and 27 received a risk rating in the category of LOW severity.

C4 analysis also identified 23 non-critical recommendations and 53 gas optimizations.



Scope

The code under review can be found within the [C4 Streaming Protocol contest repository](#), and is composed of 3 smart contracts written in the Solidity programming language and includes ~880 source lines of Solidity code.



Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on [OWASP standards](#).

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](#).



High Risk Findings (10)



[H-01] Wrong calculation of excess depositToken allows stream creator to retrieve

`depositTokenFlashloanFeeAmount` , which may cause fund loss to users

Submitted by WatchPug, also found by 0x0x0x, ScopeLift, gpersoon, harleythedog, hyh, gzeon, jonah1005, and kenzo

<https://github.com/code-423n4/2021-11-streaming/blob/56d81204a00fc949d29ddd277169690318b36821/Streaming/src/Locke.sol#L654-L654>

```
uint256 excess = ERC20(token).balanceOf(address(this)) - (deposi
```

In the current implementation, `depositTokenFlashloanFeeAmount` is not excluded when calculating `excess depositToken`. Therefore, the stream creator can call `recoverTokens(depositToken, recipient)` and retrieve `depositTokenFlashloanFeeAmount` if there are any.

As a result:

- When the protocol `governance` calls `claimFees()` and claim accumulated `depositTokenFlashloanFeeAmount`, it may fail due to insufficient balance of `depositToken`.
- Or, part of users' funds (`depositToken`) will be transferred to the protocol `governance` as fees, causing some users unable to withdraw or can only withdraw part of their deposits.



Proof of Concept

Given:

- `feeEnabled : true`
- `feePercent : 10 (0.1%)`
- Alice deposited `1,000,000 depositToken`;
- Bob called `flashloan()` and borrowed `1,000,000 depositToken`, then repaid `1,001,000` ;
- Charlie deposited `1,000 depositToken`;
- After `endDepositLock`, Alice called `claimDepositTokens()` and withdrawn `1,000,000 depositToken`;
- `streamCreator` called `recoverTokens(depositToken, recipient)` and retrieved `1,000 depositToken (2,000 - (1,001,000 - 1,000,000))` ;
- `governance` called `claimFees()` and retrieved another `1,000 depositToken`;
- Charlie tries to `claimDepositTokens()` but since the current balanceOf `depositToken` is `0`, the transaction always fails, and Charlie loses all the `depositToken`.



Recommendation

Change to:

```
uint256 excess = ERC20(token).balanceOf(address(this)) - (deposi
```

[brockelmore \(Streaming Protocol\) confirmed](#)



[H-02] Tokens can be stolen when `depositToken ==`

`rewardToken`

Submitted by cmichel, also found by 0x0x0x, gzeon, Ruhum, gpersoon, hack3r-0m, and pauliax

The `Streaming` contract allows the `deposit` and `reward` tokens to be the same token.

I believe this is intended, think Sushi reward on Sushi as is the case with `xSushi`.

The reward and deposit balances are also correctly tracked independently in `depositTokenAmount` and `rewardTokenAmount`. However, when recovering tokens this leads to issues as the token is recovered twice, once for deposits and another time for rewards:

```
function recoverTokens(address token, address recipient) public
// NOTE: it is the stream creators responsibility to save
// tokens on behalf of their users.
require(msg.sender == streamCreator, "!creator");
if (token == depositToken) {
    require(block.timestamp > endDepositLock, "time");
    // get the balance of this contract
    // check what isnt claimable by either party
    // @audit-info depositTokenAmount updated on stake/withd
    uint256 excess = ERC20(token).balanceOf(address(this)) -
    // allow saving of the token
    ERC20(token).safeTransfer(recipient, excess);

    emit RecoveredTokens(token, recipient, excess);
    return;
}

if (token == rewardToken) {
    require(block.timestamp > endRewardLock, "time");
    // check current balance vs internal balance
    //
    // NOTE: if a token rebases, i.e. changes balance out fr
    // most of this contract breaks and rugs depositors. thi
    // to this function but this function would in theory al
    // and recover the excess (if it is worth anything)
```

```

        // check what isnt claimable by depositors and governanc
        // @audit-info rewardTokenAmount increased on fundStream
uint256 excess = ERC20(token).balanceOf(address(this)) -
ERC20(token).safeTransfer(recipient, excess);

        emit RecoveredTokens(token, recipient, excess);
        return;
    }
    // ...

```



Proof Of Concept

Given `recoverTokens == depositToken`, Stream creator calls

`recoverTokens(token = depositToken, creator)`.

- The `token` balance is the sum of deposited tokens (minus reclaimed) plus the reward token amount. `ERC20(token).balanceOf(address(this)) >= (depositTokenAmount - redeemedDepositTokens) + (rewardTokenAmount + rewardTokenFeeAmount)`
- if `(token == depositToken)` executes, the `excess` from the deposit amount will be the reward amount (`excess >= rewardTokenAmount + rewardTokenFeeAmount`). This will be transferred.
- if `(token == rewardToken)` executes, the new token balance is just the deposit token amount now (because the reward token amount has been transferred out in the step before). Therefore, `ERC20(token).balanceOf(address(this)) >= depositTokenAmount - redeemedDepositTokens`. If this is non-negative, the transaction does not revert and the creator makes a profit.

Example:

- outstanding redeemable deposit token amount: `depositTokenAmount - redeemedDepositTokens = 1000`
- funded `rewardTokenAmount` (plus `rewardTokenFeeAmount` fees): `rewardTokenAmount + rewardTokenFeeAmount = 500`

Creator receives $1500 - 1000 = 500$ excess deposit and $1000 - 500 = 500$ excess reward.



Impact

When using the same deposit and reward token, the stream creator can steal tokens from the users who will be unable to withdraw their profit or claim their rewards.



Recommended Mitigation Steps

One needs to be careful with using `.balanceOf` in this special case as it includes both deposit and reward balances.

Add a special case for `recoverTokens` when `token == depositToken == rewardToken` and then the excess should be

```
ERC20(token).balanceOf(address(this)) - (depositTokenAmount - redeemedDepositTokens) - (rewardTokenAmount + rewardTokenFeeAmount);
```

[brockelmore \(Streaming Protocol\) confirmed](#)



[H-03] Reward token not correctly recovered

Submitted by cmichel, also found by GeekyLumberjack, kenzo, pedroais, and hyh

The `Streaming` contract allows recovering the reward token by calling `recoverTokens(rewardToken, recipient)`.

However, the excess amount is computed incorrectly as

```
ERC20(token).balanceOf(address(this)) - (rewardTokenAmount + rewardTokenFeeAmount) :
```

```
function recoverTokens(address token, address recipient) public
    if (token == rewardToken) {
        require(block.timestamp > endRewardLock, "time");

        // check what isnt claimable by depositors and governanc
        // @audit-issue rewardTokenAmount increased on fundStrea
        uint256 excess = ERC20(token).balanceOf(address(this)) -
            ERC20(token).safeTransfer(recipient, excess);
```

```

        emit RecoveredTokens(token, recipient, excess);
        return;
    }
    // ...

```

Note that `rewardTokenAmount` only ever *increases* (when calling `fundStream`) but it never decreases when claiming the rewards through `claimReward`. However, `claimReward` transfers out the reward token.

Therefore, the `rewardTokenAmount` never tracks the contract's reward balance and the excess cannot be computed that way.



Proof Of Concept

Assume no reward fees for simplicity and only a single user staking.

- Someone funds 1000 reward tokens through `fundStream(1000)`. Then `rewardTokenAmount = 1000`
- The stream and reward lock period is over, i.e. `block.timestamp > endRewardLock`
- The user claims their full reward and receives 1000 reward tokens by calling `claimReward()`. The reward contract balance is now 0 but `rewardTokenAmount = 1000`
- Some fool sends 1000 reward tokens to the contract by accident. These cannot be recovered as the `excess = balance - rewardTokenAmount = 0`



Impact

Reward token recovery does not work.



Recommended Mitigation Steps

The claimed rewards need to be tracked as well, just like the claimed deposits are tracked. I think you can even decrease `rewardTokenAmount` in `claimReward` because at this point `rewardTokenAmount` is not used to update the `cumulativeRewardPerToken` anymore.



[H-04] Improper implementation of `arbitraryCall()` allows protocol gov to steal funds from users' wallets

Submitted by WatchPug, also found by Jujic and hack3r-0m

<https://github.com/code-423n4/2021-11-streaming/blob/56d81204a00fc949d29ddd277169690318b36821/Streaming/src/Locke.sol#L733-L735>

```
function arbitraryCall(address who, bytes memory data) public {
    // cannot have an active incentive for the callee
    require(incentives[who] == 0, "inc");
    ...
}
```

When an `incentiveToken` is claimed after `endStream`, `incentives[who]` will be 0 for that `incentiveToken`.

If the protocol gov is malicious or compromised, they can call `arbitraryCall()` with the address of the `incentiveToken` as `who` and `transferFrom()` as `calldata` and steal all the `incentiveToken` in the victim's wallet balance up to the allowance amount.



Proof of Concept

1. Alice approved `USDC` to the streaming contract;
2. Alice called `createIncentive()` and added `1,000 USDC` of incentive;
3. After the stream is done, the stream creator called `claimIncentive()` and claimed `1,000 USDC`;

The compromised protocol gov can call `arbitraryCall()` and steal all the `USDC` in Alice's wallet balance.



Recommendation

Consider adding a mapping: `isIncentiveToken` , setting

```
isIncentiveToken[incentiveToken] = true in createIncentive() , and  
require(!isIncentiveToken[who], ...) in arbitraryCall() .
```

[brockelmore \(Streaming Protocol\) confirmed](#)



[H-05] Possible incentive theft through the `arbitraryCall()` function

Submitted by toastedsteaksandwich, also found by Omik, ScopeLift, bitbopper, pedroais, gzeon, MetaOxNull, and wuwe1



Impact

The `Locke.arbitraryCall()` function allows the inherited governance contract to perform arbitrary contract calls within certain constraints. Contract calls to tokens provided as incentives through the `createIncentive()` function are not allowed if there is some still some balance according to the incentives mapping (See line 735 referenced below).

However, the token can still be called prior any user creating an incentive, so it's possible for the `arbitraryCall()` function to be used to set an allowance on an incentive token before the contract has actually received any of the token through `createIncentive()` .

In summary:

1. If some possible incentive tokens are known prior to being provided, the `arbitraryCall()` function can be used to pre-approve a token allowance for a malicious recipient.
2. Once a user calls `createIncentive()` and provides one of the pre-approved tokens, the malicious recipient can call `transferFrom` on the provided incentive token and withdraw the tokens.



Proof of Concept

<https://github.com/code-423n4/2021-11-streaming/blob/main/Streaming/src/Locke.sol#L735>



Recommended Mitigation Steps



Recommendation 1

Limit the types of incentive tokens so it can be checked that it's not the target contract for the `arbitraryCall()`.



Recommendation 2

Validate that the allowance of the target contract (if available) has not changed.

[brockelmore \(Streaming Protocol\) confirmed](#)



[H-06] Creating rewardTokens without streaming depositTokens

Submitted by bitbopper



Impact

`stake` and `withdraws` can generate rewardTokens without streaming depositTokens. It does not matter whether the stream is a sale or not.

The following lines can increase the reward balance on a `withdraw` some time after

`stake` : <https://github.com/code-423n4/2021-11-streaming/blob/main/Streaming/src/Locke.sol#L219:L222>

```
// accumulate reward per token info
cumulativeRewardPerToken = rewardPerToken();

// update user rewards
ts.rewards = earned(ts, cumulativeRewardPerToken);
```

While the following line can be gamed in order to not stream any tokens (same `withdraw tx`).

Specifically an attacker can arrange to create a fraction less than zero thereby subtracting zero.

<https://github.com/code-423n4/2021-11-streaming/blob/56d81204a00fc949d29ddd277169690318b36821/Streaming/src/Locke.sol#L229>

```
ts.tokens -= uint112(acctTimeDelta * ts.tokens / (endStream - ts  
// WARDEN TRANSLATION: (elapsedSecondsSinceStake * stakeAmount)
```

A succesful attack increases the share of rewardTokens of the attacker.

The attack can be repeated every block increasing the share further. The attack could be done from multiple EOA increasing the share further. In short: Attackers can create loss of funds for (honest) stakers.

The economic feasability of the attack depends on:

- staked amount (times number of attacks) vs total staked amount
- relative value of rewardToken to gasprice



Proof of Concept



code

The following was added to `Locke.t.sol` for the `StreamTest` Contract to simulate the attack from one EOA.

```
function test_quickDepositAndWithdraw() public {  
    ///// SETUP  
    // accounting (to proof attack): save the rewardBalance of a  
    uint StartBalanceA = testTokenA.balanceOf(address(alice));  
    uint112 stakeAmount = 10_000;  
  
    // start stream and fill it  
    (  
        uint32 maxDepositLockDuration,  
        uint32 maxRewardLockDuration,  
        uint32 maxStreamDuration,  
        uint32 minStreamDuration  
    ) = defaultStreamFactory.streamParams();
```

```

uint64 nextStream = defaultStreamFactory.currStreamId();
Stream stream = defaultStreamFactory.createStream(
    address(testTokenA),
    address(testTokenB),
    uint32(block.timestamp + 10),
    maxStreamDuration,
    maxDepositLockDuration,
    0,
    false
    // false,
    // bytes32(0)
);

testTokenA.approve(address(stream), type(uint256).max);
stream.fundStream(1_000_000_000);

// wait till the stream starts
hevm.warp(block.timestamp + 16);
hevm.roll(block.number + 1);

// just interact with contract to fill "lastUpdate" and "ts."
// without changing balances inside of Streaming contract
alice.doStake(stream, address(testTokenB), stakeAmount);
alice.doWithdraw(stream, stakeAmount);

///// ATTACK COMES HERE
// stake
alice.doStake(stream, address(testTokenB), stakeAmount);

// wait a block
hevm.roll(block.number + 1);
hevm.warp(block.timestamp + 16);

// withdraw soon thereafter
alice.doWithdraw(stream, stakeAmount);

// finish the stream
hevm.roll(block.number + 9999);
hevm.warp(block.timestamp + maxDepositLockDuration);

// get reward
alice.doClaimReward(stream);

// accounting (to proof attack): save the rewardBalance of a

```

```

uint EndBalanceA = testTokenA.balanceOf(address(alice));
uint EndBalanceB = testTokenB.balanceOf(address(alice));

// Stream returned everything we gave it
// (doStake sets balance of alice out of thin air => we compare
assert(stakeAmount == EndBalanceB);

// we gained reward token without risk
assert(StartBalanceA == 0);
assert(StartBalanceA < EndBalanceA);
emit log_named_uint("alice gained", EndBalanceA);
}

```



commandline

```

dapp test --verbosity=2 --match "test_quickDepositAndWithdraw
Running 1 tests for src/test/Locke.t.sol:StreamTest
[PASS] test_quickDepositAndWithdraw() (gas: 4501209)

```

```

Success: test_quickDepositAndWithdraw

```

```

    alice gained: 13227

```



Tools Used

dapptools



Recommended Mitigation Steps

Ensure staked tokens can not generate reward tokens without streaming deposit tokens. First idea that comes to mind is making following line

<https://github.com/code-423n4/2021-11-streaming/blob/56d81204a00fc949d29ddd277169690318b36821/Streaming/src/Locke.sol#L220> dependable on a positive amount > 0 of:

<https://github.com/code-423n4/2021-11-streaming/blob/56d81204a00fc949d29ddd277169690318b36821/Streaming/src/Locke.sol#L229>

[brockelmore \(Streaming Protocol\) confirmed](#)



[H-07] Business logic bug in `__abdicate()` function - 2 Bugs

Submitted by cyberboy, also found by MetaOxNull



Impact

The `__abdicate()` function at <https://github.com/code-423n4/2021-11-streaming/blob/main/Streaming/src/Locke.sol#L46-L50> is the logic to remove the governance i.e., to renounce governance. However, the function logic does not consider emergency governor and pending governor, which can be a backdoor as only the “gov” is set to zero address while the emergency and pending gov remains. A pending gov can just claim and become the gov again, replacing the zero address.



Proof of Concept

1. Compile the contract and set the `__GOVERNOR` and `__EMERGENCY_GOVERNOR`.
2. Now set a `pendingGov` but do not call `acceptGov()`

Bug 1

3. Call the `__abdicate()` function and we will notice only “gov” is set to zero address while emergency gov remains.

Bug2

4. Now use the address used in `pendingGov` to call `acceptGov()` function.
5. We will notice the new gov has been updated to the new address from the zero address.

Hence the `__abdicate()` functionality can be used as a backdoor using emergency governor or leaving a pending governor to claim later.



Tools Used

Remix to test the proof of concept.



Recommended Mitigation Steps

The `__abdicate()` function should set `emergency_gov` and `pendingGov` as well to zero address.

[brockelmore \(Streaming Protocol\) confirmed and disagreed with severity:](#)

Yes, the governor can be recovered from abdication if `pendingGov != 0` as well as emergency gov needs to be set to 0 before abdication because it won't be able to abdicate itself.

Would consider it to be medium risk because chances of it ever being called are slim as it literally would cutoff the protocol from being able to capture its fees.

[Oxean \(judge\) commented:](#)

Given that the functionality and vulnerability exists, and the governor does claim fees, this could lead to the loss of funds. Based on the documentation for C4, that would qualify as high severity.

3 – High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).



[H-08] `ts.tokens` sometimes calculated incorrectly

Submitted by gpersoon, also found by WatchPug



Impact

Suppose someone stakes some tokens and then withdraws all of his tokens (he can still withdraw). This will result in `ts.tokens` being 0.

Now after some time he stakes some tokens again. At the second stake `updateStream()` is called and the following if condition is false because

```
ts.tokens==0
```

```
if (acctTimeDelta > 0 && ts.tokens > 0) {
```

Thus `ts.lastUpdate` is not updated and stays at the value from the first withdraw. Now he does a second withdraw. `updateStream()` is called and calculates the updated value of `ts.tokens`. However it uses `ts.lastUpdate`, which is the time

from the first withdraw and not from the second stake. So the value of `ts.token` is calculated incorrectly. Thus more tokens can be withdrawn than you are supposed to be able to withdraw.



Proof of Concept

<https://github.com/code-423n4/2021-11-streaming/blob/56d81204a00fc949d29ddd277169690318b36821/Streaming/src/Locke.sol#L417-L447>

```
function stake(uint112 amount) public lock updateStream(msg.sender, amount) {
    ...
    uint112 trueDepositAmt = uint112(newBal - prevBal);
    ...
    ts.tokens += trueDepositAmt;
}
```

<https://github.com/code-423n4/2021-11-streaming/blob/56d81204a00fc949d29ddd277169690318b36821/Streaming/src/Locke.sol#L455-L479>

```
function withdraw(uint112 amount) public lock updateStream(msg.sender, amount) {
    ...
    ts.tokens -= amount;
}
```

<https://github.com/code-423n4/2021-11-streaming/blob/56d81204a00fc949d29ddd277169690318b36821/Streaming/src/Locke.sol#L203-L250>

```
function updateStreamInternal(address who) internal {
    ...
    uint32 acctTimeDelta = uint32(block.timestamp) - ts.lastUpdate;
    if (acctTimeDelta > 0 && ts.tokens > 0) {
        // some time has passed since this user last interacted
        // update ts not yet streamed
        ts.tokens -= uint112(acctTimeDelta * ts.tokens / (endStreamTime - ts.lastUpdate));
        ts.lastUpdate = uint32(block.timestamp);
    }
}
```



Recommended Mitigation Steps

Change the code in `updateStream()` to:

```

if (acctTimeDelta > 0 ) {
    // some time has passed since this user last interacted
    // update ts not yet streamed
    if (ts.tokens > 0)
        ts.tokens -= uint112(acctTimeDelta * ts.tokens / (er
    ts.lastUpdate = uint32(block.timestamp); // always update t
}

```

Note: the next if statement with `unstreamed` and `lastUpdate` can be changed in a similar way to save some gas

[brockelmore \(Streaming Protocol\) confirmed:](#)



Nice catch :)



[H-09] DOS while dealing with `erc20` when value(i.e `amount*decimals`) is high but less than `type(uint112).max`

Submitted by `hack3r-0m`



Impact

<https://github.com/code-423n4/2021-11-streaming/blob/main/Streaming/src/Locke.sol#L229>

reverts due to overflow for higher values (but strictly less than `type(uint112).max`) and hence when user calls `exit` or `withdraw` function it will revert and that user will not be able to withdraw funds permanently.



Proof of Concept

Attaching diff to modify tests to reproduce behaviour:

```

diff --git a/Streaming/src/test/Locke.t.sol b/Streaming/src/test
index 2be8db0..aba19ce 100644

```

```

--- a/Streaming/src/test/Locke.t.sol
+++ b/Streaming/src/test/Locke.t.sol
@@ -166,14 +166,14 @@ contract StreamTest is LockeTest {
    );

    testTokenA.approve(address(stream), type(uint256).max);
-   stream.fundStream((10**14)*10**18);
+   stream.fundStream(1000);

-   alice.doStake(stream, address(testTokenB), (10**13)*10**18);
+   alice.doStake(stream, address(testTokenB), 100);

    hevm.warp(startTime + minStreamDuration / 2); // move t

-   bob.doStake(stream, address(testTokenB), (10**13)*10**18);
+   bob.doStake(stream, address(testTokenB), 100);

    hevm.warp(startTime + minStreamDuration / 2 + minStreamDuration);

@@ -182,10 +182,10 @@ contract StreamTest is LockeTest {
    hevm.warp(startTime + minStreamDuration + 1); // warp t

-   // alice.doClaimReward(stream);
-   // assertEq(testTokenA.balanceOf(address(alice)), 533*(10**18));
-   // bob.doClaimReward(stream);
-   // assertEq(testTokenA.balanceOf(address(bob)), 466*(10**18));
+   alice.doClaimReward(stream);
+   assertEq(testTokenA.balanceOf(address(alice)), 533);
+   bob.doClaimReward(stream);
+   assertEq(testTokenA.balanceOf(address(bob)), 466);
}

function test_stake() public {
diff --git a/Streaming/src/test/Utils/LockeTest.sol b/Streaming/src/test/Utils/LockeTest.sol
index eb38060..a479875 100644
--- a/Streaming/src/test/Utils/LockeTest.sol
+++ b/Streaming/src/test/Utils/LockeTest.sol
@@ -90,11 +90,11 @@ abstract contract LockeTest is TestHelpers {
    testTokenA = ERC20(address(new TestToken("Test Token A")));
    testTokenB = ERC20(address(new TestToken("Test Token B")));
    testTokenC = ERC20(address(new TestToken("Test Token C")));
-   write_balanceOf_ts(address(testTokenA), address(this), 1000);
-   write_balanceOf_ts(address(testTokenB), address(this), 1000);
-   write_balanceOf_ts(address(testTokenC), address(this), 1000);

```

```

-         assertEquals(testTokenA.balanceOf(address(this)), (10**14)')
-         assertEquals(testTokenB.balanceOf(address(this)), (10**14)')
+         write_balanceOf_ts(address(testTokenA), address(this),
+         write_balanceOf_ts(address(testTokenB), address(this),
+         write_balanceOf_ts(address(testTokenC), address(this),
+         assertEquals(testTokenA.balanceOf(address(this)), 100*10**1
+         assertEquals(testTokenB.balanceOf(address(this)), 100*10**1

        defaultStreamFactory = new StreamFactory(address(this),

```



Tools Used

Manual Review



Recommended Mitigation Steps

Consider doing arithmetic operations in two steps or upcasting to u256 and then downcasting. Alternatively, find a threshold where it breaks and add require condition to not allow total stake per user greater than threshold.

[brockelmore \(Streaming Protocol\) confirmed](#)



[H-10] recoverTokens doesn't work when isSale is true

Submitted by harleythedog, also found by kenzo, pedroais, hyh, and pauliax



Impact

In `recoverTokens`, the logic to calculate the excess number of deposit tokens in the contract is:

```
uint256 excess = ERC20(token).balanceOf(address(this)) - (deposi
```

This breaks in the case where `isSale` is true and the deposit tokens have already been claimed through the use of `creatorClaimSoldTokens`. In this case,

`redemeedDepositTokens` will be zero, and `depositTokenAmount` will still be at its original value when the streaming ended. As a result, any attempts to recover deposit tokens from the contract would either revert or send less tokens than should

be sent, since the logic above would still think that there are the full amount of deposit tokens in the contract. This breaks the functionality of the function completely in this case.



Proof of Concept

See the excess calculation here: <https://github.com/code-423n4/2021-11-streaming/blob/56d81204a00fc949d29ddd277169690318b36821/Streaming/src/Locke.sol#L654>

See `creatorClaimSoldTokens` here: <https://github.com/code-423n4/2021-11-streaming/blob/56d81204a00fc949d29ddd277169690318b36821/Streaming/src/Locke.sol#L583>

Notice that `creatorClaimSoldTokens` does not change `depositTokenAmount` or `redeemedDepositTokens`, so the excess calculation will be incorrect in the case of sales.



Tools Used

Inspection



Recommended Mitigation Steps

I would recommend setting `redeemedDepositTokens` to be `depositTokenAmount` in the function `creatorClaimSoldTokens`, since claiming the sold tokens is like “redeeming” them in a sense. This would fix the logic issue in `recoverTokens`.

[brockelmore \(Streaming Protocol\) commented](#)

[Oxean \(judge\) commented:](#)

upgrading to High as assets would be lost in the case outlined by the warden

3 – High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).



Medium Risk Findings (5)



[M-01] LockeERC20 is vulnerable to frontrun attack

Submitted by egjlmn1, also found by itsmeSTYJ, toastedsteaksandwich, and WatchPug



Impact

A user can steal another user's tokens if he frontrun before he changes the allowance.

The `approve()` function receives an amount to change to. Lets say user A approved user B to take N tokens, and now he wants to change from N to M, if he calls `approve(M)` the attacker can frontrun, take the N tokens, wait until after the approve transaction, and take another M tokens. And taking N tokens more than the user wanted.



Tools Used

Manual code review



Recommended Mitigation Steps

Change the approve function to either accept the old amount of allowance and require the current allowance to be equal to that, or change to two different functions that increase and decrease the allowance instead of straight on changing it.

[brockelmore \(Streaming Protocol\) acknowledged and disagreed with severity](#)

[Oxean \(judge\) commented:](#)

Front running of the `approve` ERC20 function is pretty well documented and this point and there are some good ways to mitigate this risk. I am going to downgrade to Medium since there are some other requirements for this to actual mean that assets have been lost 2 – Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted,

or leak value with a hypothetical attack path with stated assumptions, but external requirements.



[M-02] Any arbitraryCall gathered airdrop can be stolen with recoverTokens

Submitted by hyh



Impact

Any airdrop gathered with `arbitraryCall` will be immediately lost as an attacker can track `arbitraryCall` transactions and back run them with calls to `recoverTokens`, which doesn't track any tokens besides reward, deposit and incentive tokens, and will give the airdrop away.



Proof of Concept

`arbitraryCall` requires that tokens to be gathered shouldn't be reward, deposit or incentive tokens: <https://github.com/code-423n4/2021-11-streaming/blob/main/Streaming/src/Locke.sol#L735>

Also, the function doesn't mark gathered tokens in any way. Thus, the airdrop is freely accessible for anyone to be withdrawn with `recoverTokens`:

<https://github.com/code-423n4/2021-11-streaming/blob/main/Streaming/src/Locke.sol#L687>



Recommended Mitigation Steps

Add airdrop tokens balance mapping, record what is gathered in `arbitraryCall` and prohibit their free withdrawal in `recoverTokens` similarly to incentives[].

Now:

```
mapping (address => uint112) public incentives;
...
function recoverTokens(address token, address recipient) public
...
    if (incentives[token] > 0) {
        ...
    }
```

```

        uint256 excess = ERC20(token).balanceOf(
        ...
    }

```

To be:

```

mapping (address => uint112) public incentives;
mapping (address => uint112) public airdrops;
...
function recoverTokens(address token, address recipient) public
...
    if (incentives[token] > 0) {
        ...
        uint256 excess = ERC20(token).balanceOf(address(this)) -
        ...
    }
    if (airdrops[token] > 0) {
        ...
        uint256 excess = ERC20(token).balanceOf(address(this)) -
        ...
    }
    ...
// we do know what airdrop token will be gathered
function arbitraryCall(address who, bytes memory data, address t
    ...

    // get token balances
    uint256 preDepositTokenBalance = ERC20(depositToken).balance
    uint256 preRewardTokenBalance = ERC20(rewardToken).balanceOf
    uint256 preAirdropBalance = ERC20(token).balanceOf(address(t

    (bool success, bytes memory _ret) = who.call(data);
    require(success);

    uint256 postAirdropBalance = ERC20(token).balanceOf(address(
    require(postAirdropBalance <= type(uint112).max, "air_112");
    uint112 amt = uint112(postAirdropBalance - preAirdropBalance
    require(amt > 0, "air");
    airdrops[token] += amt;

```

[brockelmore \(Streaming Protocol\) disputed:](#)

The intention is that the claim airdrop + transfer is done atomically. Compound-style governance contracts come with this ability out of the box.

[Oxean \(judge\) commented:](#)

Going to agree with the warden that as the code is written this is an appropriate risk to call out and be aware of. Downgrading in severity because it relies on external factors but there is no on chain enforcement that this call will be operated correctly and therefore believe it represent a valid concern even if the Sponsor has a mitigation plan in place.



[M-03] This protocol doesn't support all fee on transfer tokens

Submitted by 0x0x0x

Some fee on transfer tokens, do not reduce the fee directly from the transferred amount, but subtracts it from remaining balance of sender. Some tokens prefer this approach, to make the amount received by the recipient an exact amount. Therefore, after funds are send to users, balance becomes less than it should be. So this contract does not fully support fee on transfer tokens. With such tokens, user funds can get lost after transfers.



Mitigation step

I don't recommend directly claiming to support fee on transfer tokens. Current contract only supports them, if they reduce the fee from the transfer amount.

[brockelmore \(Streaming Protocol\) acknowledged:](#)

We will make this clear for stream creators



[M-04] arbitraryCall() can get blocked by an attacker

Submitted by GiveMeTestEther, also found by ScopeLift



Impact

`arbitraryCall()` 's (L733) use case is to claim airdrops by “gov”. If the address “who” is a token that could be sent as an incentive by an attacker via `createIncentive()` then such claim can be made unusable, because on L735 there is a `require(incentives\[who] == 0, "inc");` that reverts if a “who” token was received as an incentive.

In this case the `incentives\[who]` can be set to 0 by the stream creator by calling `claimIncentive()` but only after the stream has ended according to `require(block.timestamp >= endStream, "stream");` (L520)

If the airdrop is only claimable before the end of the stream, then the airdrop can never be claimed.

If “gov” is not the stream creator then the stream creator must become also the “gov” because `claimIncentive()` only can be called by the stream creator and the `arbitraryCall()` only by “gov”. If resetting `incentives\[who]` to 0 by calling `claimIncentive()` and `arbitraryCall()` for the “who” address doesn’t happen atomic, an attacker can send between those two calls again a “who” token.



Proof of Concept

- <https://github.com/code-423n4/2021-11-streaming/blob/56d81204a00fc949d29ddd277169690318b36821/Streaming/src/Locke.sol#L733>
- <https://github.com/code-423n4/2021-11-streaming/blob/56d81204a00fc949d29ddd277169690318b36821/Streaming/src/Locke.sol#L500>



Recommended Mitigation Steps

- Best option at the moment I can think of is to accept the risk but clearly communicate to users that this can happen

[brockelmore \(Streaming Protocol\) acknowledged:](#)

Yep this is the tradeoff being made. To maintain trustlessness, we cannot remove the `incentives\[who] == 0` check. Additionally, governance shouldn’t be in charge of an arbitrary stream’s `recoverTokens` function.

The upshot of this is most `MerkleDrop` contracts are generally external of the token itself and not baked into the ERC20 itself. If a user wants to grief governance, they could continuously `createIncentive` after the stream creator claims the previous. But it does cost the user.



[M-05] Storage variable unstreamed can be artificially inflated

Submitted by harleythedog, also found by csanuragjain, gpersoon, hubble, and WatchPug



Impact

The storage variable `unstreamed` keeps track of the global amount of deposit token in the contract that have not been streamed yet. This variable is a public variable, and users that read this variable likely want to use its value to determine whether or not they want to stake in the stream.

The issue here is that `unstreamed` is incremented on calls to `stake`, but it is not being decremented on calls to `withdraw`. As a result, a malicious user could simply stake, immediately withdraw their staked amount, and they will have increased `unstreamed`. They could do this repeatedly or with large amounts to intentionally inflate `unstreamed` to be as large as they want.

Other users would see this large amount and be deterred to stake in the stream, since they would get very little reward relative to the large amount of unstreamed deposit tokens that *appear* to be in the contract. This benefits the attacker as less users will want to stake in the stream, which leaves more rewards for them.



Proof of Concept

See `stake` here: <https://github.com/code-423n4/2021-11-streaming/blob/56d81204a00fc949d29ddd277169690318b36821/Streaming/src/Locke.sol#L417>

See `withdraw` here: <https://github.com/code-423n4/2021-11-streaming/blob/56d81204a00fc949d29ddd277169690318b36821/Streaming/src/Locke.sol#L455>

Notice that `stake` increments `unstreamed` but `withdraw` does not affect `unstreamed` at all, even though `withdraw` is indeed removing unstreamed deposit tokens from the contract.



Tools Used

Inspection



Recommended Mitigation Steps

Add the following line to `withdraw` to fix this issue:

```
unstreamed -= amount;
```

[brockelmore \(Streaming Protocol\) confirmed](#)



Low Risk Findings (27)

- [\[L-01\] Avoid fee](#) Submitted by Jujic
- [\[L-02\] Loss of precision causing incorrect flashloan & creator fee calculation](#) Submitted by *hack3r-Om*, also found by Jujic and *toastedsteaksandwich*
- [\[L-03\] Missing `address\(0\)` check can, lead to user transferring token to the burn address, and doesnt reduce the total supply](#) Submitted by Omik, also found by *pauliax*
- [\[L-04\] Missing zero Address check](#) Submitted by *cyberboy*
- [\[L-05\] Token owner cannot claim `rewardToken` if they are not the original depositor](#) Submitted by *gzeon*
- [\[L-06\] `Stream.sol`: possible `tx.origin` attack vector via `recoverTokens\(\)`](#) Submitted by *itsmeSTYJ*
- [\[L-07\] constructor should guard against zero addresses](#) Submitted by *jayjonah8*
- [\[L-08\] Incorrect Validation of `feePercent`](#) Submitted by *mtz*, also found by *hubble* and *kenzo*
- [\[L-09\] Free flashloan for governance](#) Submitted by *Ox1f8b*
- [\[L-10\] Inaccuate comment about `claimFees\(\)`](#) Submitted by *GeekyLumberjack*

- [\[L-11\] depositTokens need to have a decimals\(\) function](#) Submitted by GiveMeTestEther
- [\[L-12\] TODOs List May Leak Important Info & Errors](#) Submitted by MetaOxNull, also found by robee and pauliax
- [\[L-13\] Governance has the ability to withdraw tokens the stream doesn't know about](#) Submitted by Ruhum
- [\[L-14\] Inaccurate comment in recoverTokens](#) Submitted by cmichel
- [\[L-15\] Division before multiple can lead to precision errors](#) Submitted by cyberboy
- [\[L-16\] flashLoan does not have a return statement](#) Submitted by defsec
- [\[L-17\] Use of ecrecover is susceptible to signature malleability](#) Submitted by defsec
- [\[L-18\] Incompatibility With Rebasing/Deflationary/Inflationary tokens](#) Submitted by defsec
- [\[L-19\] prevent rounding error](#) Submitted by gpersoon
- [\[L-20\] balance\(dust\) rewardsTokens may be unclaimable after endRewardLock](#) Submitted by hubble
- [\[L-21\] Floating Pragma is set.](#) Submitted by cyberboy, also found by Jujic, defsec, hyh, robee, and mtz
- [\[L-22\] Missing zero-address checks on LockeERC20 and Stream construction](#) Submitted by hyh, also found by MetaOxNull and Ox1f8b
- [\[L-23\] rewardPerToken\(\) reverts before start time.](#) Submitted by jonah1005
- [\[L-24\] Wrong comment in claimReward](#) Submitted by kenzo
- [\[L-25\] Global unstreamed variable not kept up to date](#) Submitted by kenzo
- [\[L-26\] parameter "who" not used](#) Submitted by gpersoon, also found by GiveMeTestEther, pauliax, pedroais, MetaOxNull, bitbopper, hack3r-0m, and wuwe1
- [\[L-27\] LockeERC20 name is not implemented as comment imply](#) Submitted by wuwe1



Non-Critical Findings (23)

- [\[N-23\] Deny of service because integer overflow](#) Submitted by Ox1f8b

- [\[N-01\] Missing contract check on `rewardtoken`](#) Submitted by Omik
- [\[N-02\] `creatorClaimSoldTokens\(\)` Does Not Check Destination Address](#)
Submitted by MetaOxNull
- [\[N-03\] Incentives paid to creator instead of depositor](#) Submitted by gzeon
- [\[N-04\] `Governed.sol: setPendingGov\(\)` should use the `emergency_governed` modifier.](#) Submitted by itsmeSTYJ
- [\[N-05\] Use `_notSameBlock`](#) Submitted by Ox1f8b
- [\[N-06\] Missing `address\(0\)` check, can crippled the governed functions](#)
Submitted by Omik
- [\[N-07\] Flash loan mechanics do not implement any standard](#) Submitted by hyh
- [\[N-08\] `LockeERC20.transfer\(\)` and `LockeERC20.transferFrom\(\)` emit `Transfer` events when the transferred amount is zero](#) Submitted by pants
- [\[N-09\] `LockeERC20.transferFrom\(\)` emits `Transfer` events when `from` equals `to`](#) Submitted by pants
- [\[N-10\] `LockeERC20.approve\(\)` and `LockeERC20.permit\(\)` emit `Approval` events when the allowance hasn't changed](#) Submitted by pants
- [\[N-11\] `Governed.setPendingGov\(\)` emits `NewPendingGov` events when the pending governor hasn't changed](#) Submitted by pants
- [\[N-12\] `Governed.acceptGov\(\)` emits `NewGov` events when the governor hasn't changed](#) Submitted by pants
- [\[N-13\] `Governed`'s constructor doesn't emit an initial `NewGov` event](#) Submitted by pants
- [\[N-14\] `Governed` doesn't implement the `IGoverned` interface](#) Submitted by pants
- [\[N-15\] Implementations should inherit their interface](#) Submitted by WatchPug
- [\[N-16\] Constructors should not have visibility](#) Submitted by WatchPug
- [\[N-17\] Insufficient input validation](#) Submitted by WatchPug
- [\[N-18\] Inconsistent check of token balance](#) Submitted by WatchPug
- [\[N-19\] Emergency gov is never used](#) Submitted by csanuragjain, also found by kenzo and wuwe1

- [\[N-20\] Missing NatSpec comments](#) *Submitted by cyberboy*
- [\[N-21\] Missing Emit in critical function](#) *Submitted by cyberboy*
- [\[N-22\] Typos](#) *Submitted by wuwe1*



Gas Optimizations (53)

- [\[G-01\] Use immutable keyword](#) *Submitted by 0x1f8b*
- [\[G-02\] Code Style: public functions not used by current contract should be external](#) *Submitted by WatchPug, also found by Jujic, cyberboy, pedroais, robee, and defsec*
- [\[G-03\] "> 0" is less efficient than "!= 0" for unsigned integers](#) *Submitted by yeOlde, also found by 0x0x0x, Jujic, pedroais, and pmerkleplant*
- [\[G-04\] No need to check fee inside factories constructor](#) *Submitted by 0x0x0x, also found by csanuragjain*
- [\[G-05\] fundStream can be implemented more efficiently](#) *Submitted by 0x0x0x*
- [\[G-06\] When exit is called, updateStream is called twice](#) *Submitted by 0x0x0x, also found by WatchPug, kenzo, and pauliax*
- [\[G-07\] Directly calculate fee in flash loan](#) *Submitted by 0x0x0x, also found by 0x1f8b, GeekyLumberjack, WatchPug, cmichel, danb, and pauliax*
- [\[G-08\] Not needed lastApplicableTime call in claimReward](#) *Submitted by 0x0x0x*
- [\[G-09\] In claimReward, reward can be cached more efficiently.](#) *Submitted by 0x0x0x*
- [\[G-10\] Use const instead of storage](#) *Submitted by 0x1f8b*
- [\[G-11\] Dead code](#) *Submitted by 0x1f8b*
- [\[G-12\] Avoid multiple cast](#) *Submitted by 0x1f8b*
- [\[G-13\] Remove dead code](#) *Submitted by 0x1f8b*
- [\[G-14\] Delete unnecessary variable](#) *Submitted by 0x1f8b*
- [\[G-15\] Flashloan is given for 1 token but checks balances for both reward and deposit token](#) *Submitted by pedroais, also found by 0x1f8b*
- [\[G-16\] Remove redundant math to save gas in dilutedBalance\(\)](#) *Submitted by GeekyLumberjack*

- [\[G-17\] Remove unneeded variable in creatorClaimSoldTokens\(\) to save gas](#) Submitted by *GeekyLumberjack*
- [\[G-18\] Struct TokenStream remove unused variable merkleAccess](#) Submitted by *GiveMeTestEther*, also found by *Jujic* and *mtz*
- [\[G-19\] Cache the return value from rewardPerToken\(\)](#) Submitted by *GiveMeTestEther*
- [\[G-20\] Stream constructor reuse the function arguments instead storage variables](#) Submitted by *GiveMeTestEther*
- [\[G-21\] Subtraction can be done unchecked because the require statement checks for underflow](#) Submitted by *GiveMeTestEther*
- [\[G-22\] Caching variables](#) Submitted by *Jujic*
- [\[G-23\] Use one require instead of several](#) Submitted by *Jujic*
- [\[G-24\] \[Gas optimization\] remove command less else in an if else](#) Submitted by *Omik*, also found by *gzeon*, and *pauliax*
- [\[G-25\] Use immutable variables can save gas](#) Submitted by *WatchPug*, also found by *pauliax*, *pedroais*, and *robee*
- [\[G-26\] Cache and read storage variables from the stack can save gas](#) Submitted by *WatchPug*
- [\[G-27\] Remove unnecessary variables can make the code simpler and save some gas](#) Submitted by *WatchPug*
- [\[G-28\] Slot packing increases runtime gas consumption due to masking](#) Submitted by *WatchPug*
- [\[G-29\] Adding unchecked directive can save gas](#) Submitted by *WatchPug*, also found by *defsec*, *hyh*, and *pauliax*
- [\[G-30\] LockeERC20.sol#toString\(\) Implementation can be simpler and save some gas](#) Submitted by *WatchPug*, also found by *OxOxOx*
- [\[G-31\] Avoid unnecessary storage reads can save gas](#) Submitted by *WatchPug*
- [\[G-32\] 10**6 can be changed to 1e6 and save some gas](#) Submitted by *WatchPug*
- [\[G-33\] Redundant code](#) Submitted by *WatchPug*
- [\[G-34\] Stream#claimReward\(\) storage writes and reads of ts.rewards can be combined into one](#) Submitted by *WatchPug*

- [\[G-35\] Avoid unnecessary external calls can save gas](#) Submitted by WatchPug, also found by gzeon and toastedsteaksandwich
- [\[G-36\] ++currStreamId is more gas efficient than currStreamId += 1](#) Submitted by WatchPug, also found by cmichel
- [\[G-37\] Remove unnecessary function can make the code simpler and save some gas](#) Submitted by WatchPug
- [\[G-38\] arbitraryCall does not need to check returned byte](#) Submitted by bitbopper, also found by yeOlde
- [\[G-39\] Gas: unstreamed not needed](#) Submitted by cmichel
- [\[G-40\] Gas: Check _feePercent instead](#) Submitted by cmichel
- [\[G-52\] Structs can be rearranged to save gas](#) Submitted by cyberboy
- [\[G-53\] Gas Optimization On The 2²⁵⁶-1](#) Submitted by defsec
- [\[G-41\] Use local variable in fundStream\(\)](#) Submitted by gpersoon
- [\[G-42\] Gas Optimization: Move common logic out of if block](#) Submitted by gzeon
- [\[G-43\] Gas Optimization: Use minimal proxy](#) Submitted by gzeon
- [\[G-44\] claimReward unnecessary logic](#) Submitted by harleythedog
- [\[G-45\] Stream.updateStreamInternal performs extra storage reads](#) Submitted by hyh
- [\[G-46\] Stream.claimReward can be simplified](#) Submitted by hyh
- [\[G-47\] Unnecessary call to lastApplicableTime\(\) in claimReward\(\)](#) Submitted by kenzo
- [\[G-48\] No need to temporarily save old values when updating settings](#) Submitted by kenzo
- [\[G-49\] Eliminate amt in fundStream](#) Submitted by pauliax
- [\[G-50\] Internal functions to private](#) Submitted by robee
- [\[G-51\] Use existing memory version of state variables \(Locke.sol\)](#) Submitted by yeOlde



Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top

An open organization | [Twitter](#) | [Discord](#) | [GitHub](#) | [Medium](#) | [Newsletter](#) | [Media kit](#) | [Careers](#) | [code4rena.eth](#)