Locke Security Review

Auditors

Eric Wang, Lead Security Researcher

Harikrishnan Mulackal, Lead Security Researcher

Mukesh Jaiswal, Security Researcher

Report prepared by: Pablo Misirov & Harikrishnan Mulackal

June 16, 2022

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 Introduction

The Locke protocol is designed as a novel way to exchange one token for another. Any user willing to exchange a certain amount of tokens (called reward token in the protocol) for another token (called deposit token in the protocol) can create a stream that happens over a certain duration. The protocol was inspired by Time Weighted Automated Market Makers. Users willing to buy may stake deposit tokens, which get exchanged for the reward token in a mechanism that is similar to liquidity mining contracts (see this article on liqudity mining for an introduction).

Contracts have a special case where the deposit tokens are not exchanged but locked up in the contract for a predefined duration of time.

2.1 Streaming the Deposit Token

For simplicity, let us assume that the stream starts at time 0 and ends at time T.

Suppose S_0 to be the amount of tokens that a user staked at the beginning of the stream, i.e., at 0. Let $S(t)\colon [0, T] \to \mathbb{R}$ be the function that denotes the amount of deposit tokens that are unstreamed. At the end of the stream, all tokens should be streamed.

The stake is exchanged at a rate inversely proportional to the time left and proportional to the current amount of stake[1]. One can model this using the following differential equation, along with the boundary conditions:

$$\begin{aligned} \frac{\mathrm{d}S(t)}{\mathrm{d}t} &= -\frac{S(t)}{T-t} \\ S(0) &= S_0 \\ S(T) &= 0 \end{aligned}$$

The above differential equation has the following solution[2]:

$$S(t) = S_0 \cdot \frac{T-t}{T} \tag{1}$$

From the above equation it is clear that a user's stake decreases as the stream progresses. However, the ratio of the stake for two users who staked at the same duration remains an invariant. Since updating each individual's stake at each discrete time scale (say each block) is impractical in the Ethereum blockchain, one may instead scale the stakes made at a time interval in $(0, T)$. The scaled stake is called **virtual balance**.

Suppose a user stakes the amount S_1 at time t, this is equivalent to staking $S_1 \cdot \frac{T}{T-t}$ at time 0.

Therefore, the *virtual balance* of a stake S_1 at time t is given by $S_1 \cdot \frac{T}{T-t}$.

In the Solidity code this is implemented as the function `dilutedBalance`. Every new stake computes the virtual balance by calling `dilutedBalance`.

[1] Note: the rate here is relative to what is left. The mechanism in the protocol's docs looks at the rate as a whole, instead of the 'instantanoues rate'.

[2] The boundary condition $S(T) = 0$ may be removed and the differential equation would still have the same solution!

```
function dilutedBalance(uint112 amount) internal view returns (uint256) {
    uint32 timeRemaining;
    unchecked { timeRemaining = endStream - uint32(block.timestamp); }
    uint256 diluted = uint256(streamDuration) * amount / timeRemaining;
    return amount < diluted ? diluted : amount;
}
```

The variable `ts.tokens` represents the amount of unstreamed tokens that a user has deposited. At any point of time[3] in the contract, this is supposed to be $S(t)$.

```
function _stake(uint112 amount) internal {
    ...
    ts.tokens += trueDepositAmt;

    uint256 virtualBal = dilutedBalance(trueDepositAmt);
    ts.virtualBalance += virtualBal;
    totalVirtualBalance += virtualBal;
    ...
}
```

The amount of tokens streamed during the period $[t, t + \Delta t)$ is given by $S(t) - S(t + \Delta t)$.

$$
\begin{aligned}
S(t) &= S_0 \cdot \frac{T - t}{T} \\
S(t + \Delta t) &= S_0 \cdot \frac{T - t - \Delta t}{T} && \Rightarrow \\
\frac{S(t + \Delta t)}{S(t)} &= \frac{T - t - \Delta t}{T - t} && \Rightarrow \qquad (2) \\
\frac{S(t) - S(t + \Delta t)}{S(t)} &= \frac{\Delta t}{T - t} && \qquad (3)
\end{aligned}
$$

In the Solidity code, $S(t) - S(t + \Delta t)$ is represented by the local variable `streamAmt`. From equation 3, we can see that:

$$\texttt{streamAmt} = S(t) - S(t + \Delta t) = S(t) \cdot \frac{\Delta t}{T - t} \qquad (4)$$

The calculation of `streamAmt` and of `ts.tokens` can be seen in the function `updateStreamInternal`.

```
function updateStreamInternal() internal {
    ...
    if (ts.tokens > 0) {
        uint112 streamAmt = uint112(uint256(acctTimeDelta) * ts.tokens / (endStream - ts.lastUpdate));
        if (streamAmt == 0) revert ZeroAmount();
        ts.tokens -= streamAmt;
    }
    ...
}
```

```
if (streamAmt == 0) revert ZeroAmount();
```

The above extra condition is added in the code to prevent an edge case. The amount of streamed deposit tokens during a time interval $[t, t + \Delta t)$ can be zero (due to integer rounding errors), while the amount of reward token

[3]This is only updated when the user interacts with contract, by calling the functions `stake`, `withdraw`, `exit` or `claimReward`.

during the same duration can be non-zero. As an alternative, one may use equation 2 to update the variable `ts.tokens`.

$$S(t + \Delta t) = S(t) \cdot \frac{T - t - \Delta t}{T - t}$$

```
function updateStreamInternal() internal {
    ...
    if (ts.tokens > 0) {
        // a suggested change
        ts.tokens = ts.tokens * (endStream - lastApplicableTime()) / (endStream - ts.lastUpdate);
    }
    ...
}
```

2.2 Streaming the Reward Token

The rewards are distributed in a way similar to liquidity mining contracts. A key difference is the fact that the virtual balances are used instead of staked amounts in the typical share computation in liquidity mining contracts.

That is, given a time t, let $V(t)$ denote the virtual balance of a particular user, and let $W(t)$ represent the total virtual balance. If the total amount of reward tokens for the entire stream is R, the amount of reward token distributed in the time period $[t, t + \Delta t)$ for this user is:

$$\frac{V(t)}{W(t)} \cdot \frac{R\,\Delta t}{T}$$

The total reward distributed for this particular user is:

$$\int_0^T \frac{V(t)}{W(t)} \cdot \frac{R\,\mathrm{d}t}{T}$$

Let us define the cumulative rewards until time t by the function cumulative:

$$\text{cumulative}(t) = \int_0^t \frac{V(t)}{W(t)} \cdot \frac{R\,\mathrm{d}t}{T}$$

Assuming that there were no stakes by anyone in between the time interval $[t, t + \Delta t)$, i.e., the values of $V(t)$ and $W(t)$ remains unchanged in this interval, one can see that:

$$\begin{aligned} \text{cumulative}(t + \Delta t) &= \int_0^{t+\Delta t} \frac{V(t)}{W(t)} \cdot \frac{R\,\mathrm{d}t}{T} \\ &= \int_0^t \frac{V(t)}{W(t)} \cdot \frac{R\,\mathrm{d}t}{T} + \int_t^{t+\Delta t} \frac{V(t)}{W(t)} \cdot \frac{R\,\mathrm{d}t}{T} \\ &= \text{cumulative}(t) + \frac{V(t)}{W(t)} \cdot \frac{R\,\Delta t}{T} \end{aligned} \tag{5}$$

The function $\frac{\text{cumulative}(t)}{V(t)}$ is represented by the variable `currCumRewardPerToken` in the Solidity contract. One can show from 5 that the variable `currCumRewardPerToken` is independent of the function V, and therefore has the same value for all the users, as long as there were no stakes in the time duration $[t, t + \Delta t)$:

$$\text{currCumRewardPerToken}(t) \equiv \frac{\text{cumulative}(t)}{V(t)}$$

$$
\begin{aligned}
\text{currCumRewardPerToken}(t+\Delta t) &= \frac{\text{cumulative}(t+\Delta t)}{V(t+\Delta t)} \\
&= \frac{\text{cumulative}(t+\Delta t)}{V(t)} \\
&= \frac{\text{cumulative}(t)}{V(t)} + \frac{1}{W(t)} \cdot \frac{R\,\Delta t}{T} && \text{(from 5)} \\
&= \text{currCumRewardPerToken}(t) + \frac{1}{W(t)} \cdot \frac{R\,\Delta t}{T} && (6)
\end{aligned}
$$

The additional term in equation 6 is $\frac{1}{W(t)} \cdot \frac{R\,\Delta t}{T}$. In the Solidity code, this is represented by the variable `rewards`. To avoid potential rounding issues, this number is scaled by a constant 10^{decimals} (represented in the code by `depositDecimalOne`).

The cumulative reward calculation (currCumRewardPerToken) happens in the function `rewardPerToken`:

```
function rewardPerToken() public override view returns (uint256) {
    if (totalVirtualBalance == 0) {
        return cumulativeRewardPerToken;
    } else {
        uint256 rewards;
        unchecked {
          rewards = (uint256(lastApplicableTime() - lastUpdate) * rewardTokenAmount *
↪   depositDecimalsOne) / streamDuration / totalVirtualBalance;
        }
        return cumulativeRewardPerToken + rewards;
    }
}
```

Now, assume that a user staked at time t, and that the stake is unchanged[4] during the time interval $[t, t+\Delta t)$, i.e., $V(t)$ for this user remains the same in this interval. Assume that there were n different actions (stakes and withdraws) performed by other users at times $t_1, \cdots, t_n$ all in the interval $[t, t+\Delta t)$ which changes the value of the total virtual balance $(W(t))$ for t in $\{t_1, \cdots, t_n\}$.

This means that reward earned by the user in this period is:

$$
\begin{aligned}
\Delta\text{Reward} =& V(t)\left(\int_t^{t_1} \frac{1}{W(t)} \cdot \frac{R\,\mathrm{d}t}{T} + \int_{t_1}^{t_2} \frac{1}{W(t)} \cdot \frac{R\,\mathrm{d}t}{T} + \cdots + \int_{t_n}^{t+\Delta t} \frac{1}{W(t)} \cdot \frac{R\,\mathrm{d}t}{T}\right) \\
=& V(t)\left(C(t_1) - C(t) + C(t_2) - C(t_1) + \cdots + C(t+\Delta t) - C(t_n)\right) \\
=& V(t)(C(t+\Delta t) - C(t)) && (7)
\end{aligned}
$$

Here $C(t) \equiv$ currCumRewardPerToken(t).

The Solidity variable `ts.lastCumulativeRewardPerToken` stores the value of currCumRewardPerToken(t). The variable `ts.rewards` is updated using equation 7 in the function `updateStreamInternal` and `earned`. Notice how in the function `earned`, the calculation is scaled down by `depositDecimalOne`. This is to invert the scaling up done in the function `rewardPerToken`.

[4] It's not necessary that a user staked at time t. Any operation that lead to the parameters corresponding to the user getting updated / changed at time t would be sufficient.

```
  function earned(TokenStream storage ts, uint256 currCumRewardPerToken) internal
    view returns (uint112) {
    uint256 rewardDeltaPerToken;
    unchecked {
        rewardDeltaPerToken = currCumRewardPerToken - ts.lastCumulativeRewardPerToken;
    }
    uint112 rewardDelta =
        uint112(ts.virtualBalance * rewardDeltaPerToken / depositDecimalsOne);
    return rewardDelta + ts.rewards;
}
```

```
function updateStreamInternal() internal {
    ...
    cumulativeRewardPerToken = rewardPerToken();
    ts.rewards = earned(ts, cumulativeRewardPerToken);
    ts.lastCumulativeRewardPerToken = cumulativeRewardPerToken;
    ...
}
```

2.2.1 Some notes on streaming auctions

A typical staking rewards contract is designed in a way so that, all things remaining the same, the user gets the same reward when staking at any time period Δt in the duration of the stream. However, this is not entirely true in the Locke protocol. Since the Locke protocol streams the deposit tokens according to equation 4, one can see that more deposit tokens are streamed near the end of the stream than towards the beginning of the stream.

As a user, they want to maximize the amount of reward token they receive for each deposit token staked. They get a better rate for their deposit tokens if they stake as early as possible, assuming that everything else remains the same. This may be used a strategy for users optimizing the rates.

To see this, assume that the user stakes S amount of deposit tokens at time t. And from time t to $t + \Delta t$, there were no stakes or withdrawals in the stream. The amount of tokens deposit streamed (ΔS) during this period is

$$\texttt{streamAmt} = \Delta S = S \cdot \frac{\Delta t}{T - t}$$

The amount of reward token streamed in the same duration for the deposit S is given by

$$\Delta R = \frac{V}{V + W} \cdot \frac{R \cdot \Delta t}{T}$$

Where V is the virtual balance of S, i.e., $V = S \cdot \frac{T}{T-t}$ and W is the total virtual balance of the stream at time t.

The rate of reward to deposit tokens (exchange rate) in this duration is

$$\begin{aligned}\frac{\Delta R}{\Delta S} &= \frac{V}{V + W} \cdot \frac{R \cdot \Delta t}{T} \cdot \frac{T - t}{S \cdot \Delta t} \\ &= \frac{S \cdot \left(\frac{T}{T-t}\right)}{S\left(\frac{T}{T-t}\right) + W} \cdot \frac{T - t}{T} \cdot \frac{R}{S} \\ &= \frac{R \cdot (T - t)}{S \cdot T + W(T - t)}\end{aligned}$$

An illustration can be found in a desmos graph.

Users participating in the auction will need to continuously monitor it until it is finished and may have to withdraw if the exchange rate for the stake gets lower than their personal threshold. For example, a user's stake may be followed by a very large stake which decreases the exchange price. Therefore, users may consider streams after this time to be unfavorable. The protocol allows withdrawals of unstreamed stakes to accommodate such cases. However, this requires the participant to play an active role throughout the process.

We recommend building automation that allows users to monitor auctions easily. For example: a user would be able to set a price for the exchange, and a bot can withdraw the unstreamed stake if the exchange price falls below a threshold. However, such automation can introduce nobel issues, for example, if a malicious entity knows that a user will withdraw their stake if the exchange rate is below a certain threshold, they may stake a large amount to decrease the rate, wait for automation to withdraw and follow the withdrawal by withdrawing their own stake to get a better rate.

3 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

3.1 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.

3.2 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

3.3 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

4 Executive Summary

Over the course of 8 days in total, Locke engaged with Spearbit to review Locke Protocol. In this period of time a total of 20 issues were found.

Summary

Project Name	Locke
Repository	Locke
Commit	596de43e6320eccbf1d274a...
Type of Project	Streaming Auctions, DeFi
Audit Timeline	Feb 7st - Feb 16th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	3
Medium Risk	3
Low Risk	4
Gas Optimizations	6
Informational	4
Total Issues	20

5 Findings

5.1 High Risk

5.1.1 `UnaccruedSeconds` do not increase even if nobody is actively staking

Severity: *High Risk*

Context: Lock.sol.sol#L180

Description: The `unstreamed` variable tracks whether someone is staking in the contract or not. However, because of the division precision loss at Locke.sol#L164-L166 and Locke.sol#L187, `unstreamed > 0` may happen even when everyone has already withdrawn all deposited tokens from the contract, i.e. `ts.token = 0` for everyone.

Consider the following proof of concept with only two users, Alice and Bob:

- `streamDuration = 8888`
- At `t = startTime`, Alice stakes `1052 wei` of deposit tokens.
- At `t = startTime + 99`, Bob stakes `6733 wei` of deposit tokens.
- At `t = startTime + 36`, both Alice and Bob exits from the contract.

At this point Alice's and Bob's `ts.tokens` are both 0 but `unstreamed = 1 wei`. The abovementined numbers are the resault of a fuzzing campaign and were not carefully crafted, therefore this issue can also occur under normal circumstances.

```
function updateStreamInternal() internal {
    ...
    uint256 tdelta = timestamp - lastUpdate;
    if (tdelta > 0) {
        if (unstreamed == 0) {
            unaccruedSeconds += uint32(tdelta);
        } else {
            unstreamed -= uint112(tdelta * unstreamed / (endStream - lastUpdate));
        }
    }
    ...
}
```

Recommendation: Consider using `totalVirtualBalance == 0` instead of `unstreamed == 0`

5.1.2 Old governor can call `acceptGov()` after renouncing its role through `_abdicate()`

Severity: *High Risk*

Context: Gov.sol#L30

Description: The `__abdicate` function does not reset `pendingGov` value to `0`. Therefore, if a pending governor is set the user can become a governor by calling `acceptGov`.

Recommendation: Consider setting `pendinGov` to `address(0)` inside the `__abdicate` function.

```
    function __abdicate() governed external override {
        address old = gov;
        gov = address(0);
+       pendingGov = address(0);
        emit NewGov(old, address(0));
    }
```

5.1.3 User can lose their reward due truncated division

Severity: *High Risk*

Context: Locke.sol#L321

Description: The truncated division can cause users to lose rewards in this update round which may happen when any of the following conditions are true:

1. `RewardToken.decimals()` is too low.
2. Reward is updated too frequently.
3. `StreamDuration` is too large.
4. `TotalVirtualBalance` is too large (e.g., stake near the end of stream).

This could potentially happen especially when the 1st case is true. Consider the following scenario:

- `rewardToken.decimals() = 6.`
- `depositToken.decimals()` can be any (assume it's 18).
- `rewardTokenAmount = 1K * 10**6.`
- `streamDuration = 1209600` (two weeks).
- `totalVirtualBalance = streamDuration` * `depositTokenAmount` / `timeRemaining` where `depositToken-Amount = 100K 10**18` and `timeRemaining` = `streamDuration` (a user stakes 100K at the beginning of the stream) `lastApplicableTime() - lastUpdate = 100` (about 7 block-time).

Then rewards `= 100 * 1000 * 10**6 * 10**18 / 1209600 / (1209600 * 100000 * 10**18 / 1209600) = 0.8267 < 1.` User wants to buy the reward token at the price of 100K/1K = 100 deposit token but does not get any because of the truncated division.

```
function rewardPerToken() public override view returns (uint256) {
    if (totalVirtualBalance == 0) {
        return cumulativeRewardPerToken;
    } else {
        // time*rewardTokensPerSecond*oneDepositToken / totalVirtualBalance
        uint256 rewards;

        unchecked {
            rewards = (uint256(lastApplicableTime() - lastUpdate) * rewardTokenAmount *
↪   depositDecimalsOne) / streamDuration / totalVirtualBalance;
        }
        return cumulativeRewardPerToken + rewards;
    }
}
```

Recommendation: Consider scaling up `cumulativeRewardPerToken` and users reward.

5.2 Medium Risk

5.2.1 The `streamAmt` check may prolong a user in the stream

Severity: *Medium Risk*

Context: Locke.sol#L165

Description: Assume that the amount of tokens staked by a user (`ts.tokens`) is low. This check allows another person to deposit a large stake in order to prolong the user in a stream (until`streamAmt` for the user becomes non-zero). For this duration the user would be receiving a bad rate or 0 altogether for the reward token while being unable to exit from the pool.

```
if (streamAmt == 0) revert ZeroAmount();
```

Therefore, if Alice stakes a small amount of deposit token and Bob comes along and deposits a very large amount of deposit token, tt's in Alice's interest to exit the pool as early as possible especially when this is an indefinite stream. Otherwise the user would be receiving a bad rate for their deposit token.

Recommendation: The ideal scenario is if `streamAmt` ends up being zero for a certain `accTimeDelta`, the user should be able to exit the pool with `ts.tokens` as long as they don't receive rewards for the same duration. However, in practice, implementing this may create issues related to `unaccured` seconds.

5.2.2 User can stake before the stream creator produced a funding stream

Severity: *Medium Risk*

Context: Locke.sol#410

Description: Consider the following scenario:

1. Alice stakes in a stream before the stream starts.
2. Nobody funds the stream,.
3. In case of an `indefinite` stream Alice loses some of her deposit depending on when she exits the stream. For a usual stream Alice will have her deposit tokens locked until `endDepositLock`.

Recommendation: Two mitigatations are possible:

1. A frontend check warning the user if a stream does not have any reward tokens.
2. A check in the `stake` function which would revert when `rewardTokenAmount == 0`.

5.2.3 Potential funds locked due low token decimal and long stream duration

Severity: *Medium Risk*

Context: Locke.sol#L166

Description: In case where the deposit token decimal is too low (4 or less) or when the remaining stream duration is too long, checking `streamAmt > 0` may affect regular users. They could be temporarily blocked by the contract, i.e. they cannot stake, withdraw, or get rewards, and should wait until `streamAmt > 0` or the stream ends. Altough unlikely to happen it still is a potential lock of funds issue.

```
function updateStreamInternal() internal {
    ...
    if (acctTimeDelta > 0) {
        if (ts.tokens > 0) {
            uint112 streamAmt = uint112(uint256(acctTimeDelta) * ts.tokens / (endStream -
↪   ts.lastUpdate));
            if (streamAmt == 0) revert ZeroAmount();
            ts.tokens -= streamAmt;
        }
    ...
}
```

Recommendation: Consider scaling `ts.tokens` to 18 decimals of precision for internal accounting.

5.3 Low Risk

5.3.1 Sanity check on the reward token's decimals

Severity: *Low Risk*

Context: Locke.sol#L262

Description: Add sanity check on the reward token's decimals, which shouldn't exceed 33 because `TokenStream.rewards` has a `uint112` type.

```
constructor(
        uint64 _streamId,
        address creator,
        bool _isIndefinite,
        address _rewardToken,
        address _depositToken,
        uint32 _startTime,
        uint32 _streamDuration,
        uint32 _depositLockDuration,
        uint32 _rewardLockDuration,
        uint16 _feePercent,
        bool _feeEnabled
    )
        LockeERC20(
            _depositToken,
            _streamId,
            _startTime + _streamDuration + _depositLockDuration,
            _startTime + _streamDuration,
            _isIndefinite
        )
        MinimallyExternallyGoverned(msg.sender) // inherit factory governance
    {
        // No error code or msg to reduce bytecode size
        require(_rewardToken != _depositToken);
        // set fee info
        feePercent = _feePercent;
        feeEnabled = _feeEnabled;

        // limit feePercent
        require(feePercent < 10000);

        // store streamParams
        startTime = _startTime;
        streamDuration = _streamDuration;

        // set in shared state
```

```
        endStream = startTime + streamDuration;
        endDepositLock = endStream + _depositLockDuration;

        endRewardLock = startTime + _rewardLockDuration;

        // set tokens
        depositToken = _depositToken;
        rewardToken = _rewardToken;

        // set streamId
        streamId = _streamId;

        // set indefinite info
        isIndefinite = _isIndefinite;

        streamCreator = creator;

        uint256 one = ERC20(depositToken).decimals();
        if (one > 33) revert BadERC20Interaction();
        depositDecimalsOne = uint112(10**one);

        // set lastUpdate to startTime to reduce codesize and first users gas
        lastUpdate = startTime;
    }
```

Recommendation: Consider adding sanity checks on the reward token's decimals.

5.3.2 Use a stricter bound for transferability delay

Severity: *Low Risk*

Context: LockeErc20.sol#L177

Description:

```
modifier transferabilityDelay {
    // ensure the time is after end stream
    if (block.timestamp < endStream) revert NotTransferableYet();
    _;
}
```

Recommendation: Consider using <= insted of < so that it can cover time up to `endStream`. `block.timestamp <= endStream`

5.3.3 Potential issue with malicious stream creator

Severity: *Low Risk*

Context: Locke.sol#L307

Description: Assume that users staked tokens at the beginning. The malicious stream creator could come and stake an extremely large amount of tokens thus driving up the value of `totalVirtualBalance`. This means that users will barely receive rewards while giving away deposit tokens at the same rate. Users can exit the pool in this case to save their `unstreamed tokens`.

```
function rewardPerToken() public override view returns (uint256) {
    if (totalVirtualBalance == 0) {
        return cumulativeRewardPerToken;
    } else {

        unchecked {
            rewards = (uint256(lastApplicableTime() - lastUpdate) * rewardTokenAmount *
↪   depositDecimalsOne) / streamDuration / totalVirtualBalance;
        }
        return cumulativeRewardPerToken + rewards;
    }
}
```

Recommendation: Consider constant stream monitoring.

5.3.4 Users may exit the pool at a small duration before `endStream` to save tokens

Severity: *Low Risk*

Context: Locke.sol#L164

Description: Given a time `t`, that is very close to `endStream` and a time period Δt, assume that rewards for a particular user during this period is `0`. Since the amount of deposit token streamed is inversely proportional to the time that is left, i.e. more deposit tokens are exchanged during a time period that is closer to the end of the stream rather than at the beginning of the stream, it is likely that the deposit token sold during Δt is non-zero.

A user can use this information to exit the stream just before `endStream`. In this process the user prevents some of their tokens from streaming while receiving no rewards for this duration!

Suppose that there is a stream and two tokens `testTokenB` and `testTokenA`, where `testTokenA` is the reward token and `testTokenB` is the deposit token. Assume that the following conditions are true:

```
assert(testTokenB.balanceOf(alice) == 200);
assert(testTokenB.balanceOf(bob) == 100);
assert(testTokenA.balanceOf(alice) == 0);
assert(testTokenA.balanceOf(bob) == 0);
```

Using the testing conventions from the repository, consider the following stream instance:

```
function test_equality_withdraw() public {
    testTokenA.approve(address(stream), 1000);
    stream.fundStream(1000);

    vm.startPrank(alice);
    testTokenB.approve(address(stream), 200);
    stream.stake(200);
    vm.stopPrank();

    vm.startPrank(bob);
    testTokenB.approve(address(stream), 100);
    stream.stake(100);
    vm.warp(endStream - 1);
    stream.exit();
    vm.stopPrank();

    vm.warp(endStream + 1);

    vm.prank(alice);
    stream.claimReward();

    vm.prank(bob);
    stream.claimReward();
}
```

The above code results in the following assertions to be true after the test:

```
assertEq(testTokenA.balanceOf(alice), 666);
assertEq(testTokenA.balanceOf(bob), 333);
assertEq(testTokenB.balanceOf(alice), 0);
assertEq(testTokenB.balanceOf(bob), 1);
```

The same assertions are true even if Bob does not exit from the stream. In case of an `indefinite` stream Bob would be able save some of his tokens from being sold. In the desired case Bob prevents some of his tokens from being locked up until `endDepositLock` is finished, both work in Bob's favor.

5.4 Gas Optimization

5.4.1 Moving check `require(feePercent < 10000)` in `updateFeeParams` to save gas

Severity: *Gas Optimization*

Context: Locke.sol#L237

Description: `feePercent` comes directly from LockeFactory's feeParams.feePercent, which is configured in the updateFeeParams function and used across all Stream contracts. Moving this check into the `updateFeeParams` function can avoid checking in every contract and thus save gas.

Recommendation: Consider moving the check in the `updateFeeParams` function.

5.4.2 Use `calldata` instead of `memory` for some function parameters

Severity: *Gas Optimization*

Context: MerkleLocke.sol#L10, MerkleLocke.sol#L17 and MerkleLocke.sol#L75

Description: Having function arguments in `calldata` instead of memory is more optimal in the aforementioned cases. See the following reference.

Recommendation: Consider using `calldata` instead of `memory`.

5.4.3 Update `cumulativeRewardPerToken` only once after stream ends

Severity: *Gas Optimization*

Context: Locke.sol#597

Description: Since `cumulativeRewardPerToken` does not change once it is updated after the stream ends, it has to be updated only once.

Recommendation: Consider changing the code as follows:

```
- cumulativeRewardPerToken = rewardPerToken();
+ if (lastUpdate < endStream) {
+     cumulativeRewardPerToken = rewardPerToken();
+ }
```

5.4.4 Expression `10**one` can be unchecked

Severity: *Gas Optimization*

Context: Locke.sol#263

Description:

```
        uint256 one = ERC20(depositToken).decimals();
        if (one > 33) revert BadERC20Interaction();
        depositDecimalsOne = uint112(10**one)
```

Recommendation: The following recommendation would perform checked exponentiation which is not that bad and just checks if `one > 77` and the revert, otherwise it uses `exp(10, one)`.

```
unchecked { depositDecimalsOne = uint112(10**one) };
```

5.4.5 Calculation of `amt` can be unchecked

Severity: *Gas Optimization*

Context: Locke.sol#L536

Description: The value `newBal` in this context is always greater than `prevBal` because of the check located at Locke.sol#534. Therefore, we can use unchecked subtraction.

Recommendation:

```
-    uint112 amt = uint112(newBal - prevBal);
+    uint112 amt;
+    unchecked { amt = uint112(newBal - prevBal); }
```

5.4.6 Change `lastApplicableTime()` to `endStream`

Severity: *Gas Optimization*

Context: Locke.sol#L610, Locke.sol#L615, Locke.sol#L642, and Locke.sol#L639.

Description: Since `block.timestamp >= endStream` in the abovementioned cases the `lastApplicableTime` function will always return `endStream`.

Recommendation: Change `lastApplicableTime()` to `endStream` to save gas.

5.5 Informational

5.5.1 Unused variable `_ret`

Severity: *Informational*

Context: Locke.sol#L812

Recommendation: Unused variable `_ret` can be either removed or forwarded in case of an error.

5.5.2 Simplifying code logic

Severity: *Informational*

Context: LockeLens.sol#L26-L37

Description:

```
if (timestamp < lastUpdate) {
    return tokens;
}
uint32 acctTimeDelta = timestamp - lastUpdate;

if (acctTimeDelta > 0) {
    uint256 streamAmt = uint256(acctTimeDelta) * tokens / (endStream - lastUpdate);
    return tokens - uint112(streamAmt);
} else {
    return tokens;
}
```

```
function currDepositTokensNotYetStreamed(IStream stream, address who) external view returns (uint256) {
    unchecked {
        uint32 timestamp = uint32(block.timestamp);
        (uint32 startTime, uint32 endStream, ,) = stream.streamParams();
        if (block.timestamp >= endStream) return 0;

        (
            uint256 lastCumulativeRewardPerToken,
            uint256 virtualBalance,
            uint112 rewards,
            uint112 tokens,
            uint32 lastUpdate,
            bool merkleAccess
        ) = stream.tokenStreamForAccount(address(who));

        if (timestamp < lastUpdate) {
            return tokens;
        }

        uint32 acctTimeDelta = timestamp - lastUpdate;

        if (acctTimeDelta > 0) {
            uint256 streamAmt = uint256(acctTimeDelta) * tokens / (endStream - lastUpdate);
            return tokens - uint112(streamAmt);
        } else {
            return tokens;
        }
    }
}
```

Recommendation: The above code can be optimized as follows:

```
if (timestamp <= lastUpdate) {
    return tokens;
}
uint32 acctTimeDelta = timestamp - lastUpdate;
uint256 streamAmt = uint256(acctTimeDelta) * tokens / (endStream - lastUpdate);
return tokens - uint112(streamAmt);
```

5.5.3 Unsused parameter `_emergency_governor` in constructor

Severity: *Informational*

Context: LockeFactory.sol#L29

Recommendation: Remove the unused parameter `_emergency_governor` from the constructor argument.

5.5.4 Unused Imports

Severity: *Informational*

Context: Locke.sol#L6 and SharedState.sol#L3

Recommendation: Remove the aforementioned imports.