yAudit TLX Review

Review Resources:

- TLX: Internal documentation

Auditors:

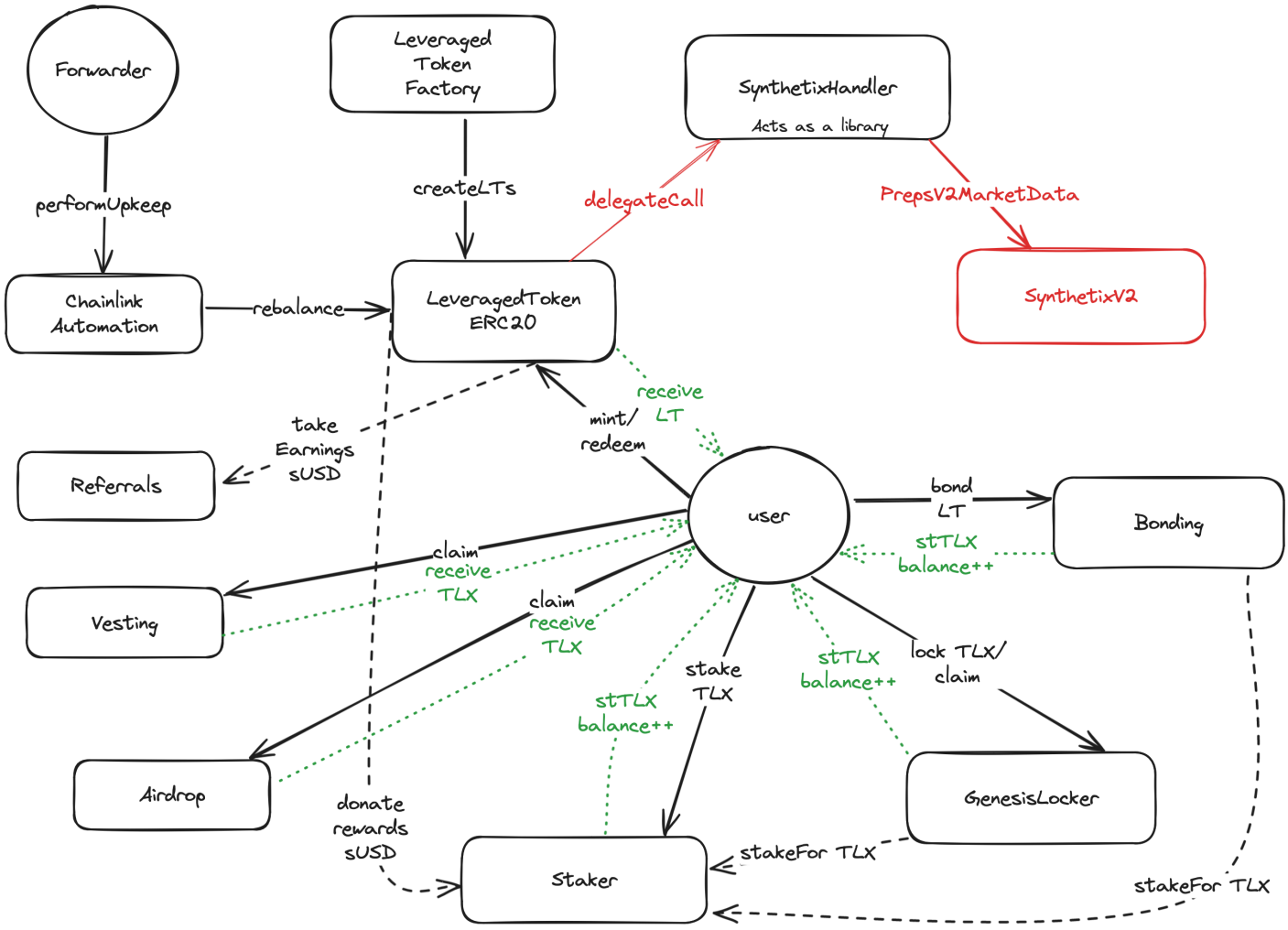
- ljmanini
- spalen

Table of Contents

Review Summary

TLX

TLX provides a permissionless, non-custodial leveraged token platform that enables users to mint and redeem leveraged tokens (LTs), which are backed by Synthetix perpetual futures contracts. The users can pick between multiple tokens that they want to long or short with different leverage levels. The Chainlink automation is used to keep the leverage factor within a targeted range. Alongside LTs, TLX also provides a bonding mechanism for LT holders to bond their tokens, in exchange for TLX token at a discount. TLX token can be staked to earn protocol withdraw fees. Airdrop and referral programs are added to incentivize users to use the protocol. The TLX protocol also enables Zapping for minting LTs with additional tokens, not just base sUSD. Swapping is done using Velodrome V2 and Uniswap V3.



The contracts of the TLX Repo were reviewed over 16 days. The code review was performed by 2 auditors between February 19 and March 5, 2024. The repository was under active development during the review, but the review was limited to the latest commit at the start of the review. This was commit 40b04b9459654ef834b695401832ad611dd41e8d for the TLX repo.

The fixes for presented issues are included in this new commit f0a196e392596c153a133d5ef02b876360338af6, but any changes between these commit hashes that are unrelated to issues found in this report are not in scope.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
src/
├── AddressProvider.sol
├── Airdrop.sol
├── Bonding.sol
├── ChainlinkAutomation.sol
├── GenesisLocker.sol
├── helpers/
│   └── LeveragedTokenHelper.sol
├── LeveragedTokenFactory.sol
├── LeveragedToken.sol
├── libraries/
│   ├── AddressKeys.sol
│   ├── Config.sol
│   ├── Contracts.sol
│   ├── Errors.sol
│   ├── InitialMint.sol
│   ├── LeveragedTokens.sol
│   ├── ParameterKeys.sol
│   ├── ScaledNumber.sol
│   ├── Symbols.sol
│   ├── TimelockDelays.sol
│   ├── Tokens.sol
│   ├── Unstakes.sol
│   ├── Vestings.sol
│   └── ZapAssetRoutes.sol
├── ParameterProvider.sol
├── Referrals.sol
├── RewardsStreaming.sol
├── Staker.sol
├── SynthetixHandler.sol
├── Timelock.sol
├── TlxToken.sol
├── utils/
│   └── TlxOwnable.sol
├── Vesting.sol
```

```
└── zaps/
    └── ZapSwap.sol
```

After the findings were presented to the TLX team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, TLX and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Adequate access control is present in user and admin controlled functionality.
Mathematics	Low	There is no complex math. An internal library is used for scaling numbers to avoid rounding errors. Few issues are raised for internal accounting
Complexity	Average	The protocol integrates closely with Synthetix Perps V2 for its core functionality and the zap functionality integrates with Uniswap and Velodrome. Additional tokenomics in the protocol is built using bonding and staking.
Libraries	Good	The protocol uses the OpenZeppelin and Chainlink base libraries.

Category	Mark	Description
Decentralization	Average	Contracts are non-upgradeable and require minimal intervention from an authorized party. New LTs may be created only a privileged actor and `Timelock` allows for some calls to be made instantly. The Chainlink automation jobs are managed off-chain and are naturally managed in a more centralized way by the owner of the automation jobs.
Code stability	Average	Although the repository was in active development during the review, the changes to the repository were minor.
Documentation	Average	All interface contracts provide NatSpec comments, internal documentation is complete and rather concise when explaining some of the protocol's inner workings.
Monitoring	Average	Events were emitted where applicable but missing in some functions.
Testing and verification	Average	The codebase includes unit and integration tests, though the test suite may be improved with proper e2e and invariant tests. Fuzzing could be added easily.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
  - Findings that can improve the gas efficiency of the contracts.
- Informational
  - Findings including recommendations and best practices.

Critical Findings

1. Critical - Bonding limit is bypassable

The amount of `TLX` meant to be purchasable by users through bonding grows piece-wise linearly: in periods of 20 days, the amount of available `TLX` grows by `20 days * tlxPerSecond`. When a new period begins, a decay factor is applied to `tlxPerSecond`, effectively reducing it. Particularly, a bonding event should cause the amount of available `TLX` to be reduced, as the total amount bonded reaches the current bonding limit.

Technical Details

Because `Bonding.bond()` fails to use its internal method `Bonding.availableTlx()` and instead reads directly from `_availableTlxCache`, to determine the maximum amount of available `TLX` to be sold through a bond, an external user can overcome this limit by repeatedly calling `Bonding.bond()` purchasing an amount below `_availableTlxCache`.

Furthermore, as a side effect, this attack may cause a DOS on `Bonding.availableTlx()` through an overflow panic, as `totalTlxBonded` may be larger than the other two operands seen here.

Following is a PoC that demonstrates this issue : gist The PoC shows how an attacker can receive twice the initial amount of `_availableTlxCache`. The test case should be added to `test/Bonding.t.sol` and executed with `forge t --mt testDoubleBondAllowsToBypassAvailableTlx`.

Impact

Critical. TLX bonding limit can be bypassed, making the contract drainable of all its `TLX`.

Recommendation

Use `uint256 availableTlx_ = _availableTlxCache - totalTlxBonded` to correctly account for `TLX` that has already been sold via bonds:

```
function bond(
    address leveragedToken_,
    uint256 leveragedTokenAmount_,
    uint256 minTlxTokensReceived_
) external override returns (uint256) {

   ...

    // Calculate the amount of TLX tokens to send to the user
-   uint256 availableTlx_ = _availableTlxCache;
+   uint256 availableTlx_ = _availableTlxCache - totalTlxBonded;

   ...
}
```

This approach has the same effect as using `availableTlx()`, as `_updateCache()` is called right before the calculation takes place.

Developer Response

Fixed: d08f3e919ada742cd4c394b8a45f2508ffdfe2ba

High Findings

1. High - Wrong calculation of `totalValue()`

The function `totalValue()` in SynthetixHandler.sol is used to get total value in Synthetix, but the calculation is double counting profit and loss made in Synthetix.

Technical Details

`totalValue()` provides the total value deposited and earned in Synthetix. This value is used to calculate the amount of leveraged token the user will receive on minting by calculating exchange rate. Or the amount of base token the user will get for redeeming.

The function first fetches `remainingMargin_` and adds profit and loss, `pnl_`, to the final calculation. In Synthetix, `remainingMargin()` already contains profit, as specified in NatSpec. By checking Synthetix's internal implementation of `_remainingMargin()`, it is clear that `_profitLoss()` value is accounted in remaining margin value. This means that `totalValue()` will double count profit and loss, first in Synthetix, second time inside a function as `pnl_` value.

Impact

High. `totalValue()` calculates the exchange rate for minting and redeeming. After any profit or loss is made in Synthetix, the user will get the wrong value for his tokens.

Recommendation

Remove additional calculations that add profit and loss. Synthetix function `remainingMargin()` already returns the total value with accounted in profit and loss.

Developer Response

Fixed: 4998a7eaa893c294e4f0573ef6c15e5038dfcee2

Medium Findings

1. Medium - Lost part of the redeem fee

During redemption, the user must pay a fee which is distributed between referrals and stakers. If there is no staked amount, this part of the fee will be lost.

Technical Details

In `redeem()` function, the fee is taken from the user and donated to stakers. The problem is when the condition `staker_.totalStaked() != 0` is false. This means that the fee `amount` is allocated for the stakers, but if there is no staked value, the `amount` value will stay in the contract. There is no option to collect undistributed fees which is a loss for the protocol.

Impact

Medium. A part of the fee will be lost in the contract under certain conditions.

Recommendation

If the staked value is zero, send the fee to the treasury or return it to the user.

```
if (amount_ != 0) {
    if (staker_.totalStaked() != 0) {
        baseAsset_.approve(address(staker_), amount_);
        staker_.donateRewards(amount_);
    } else {
        baseAmountReceived_ += amount_; // return the fee to the user
        // or send to treasury: baseAsset_.transfer(addressProvider_.treasury(),
amount_);
    }
}
```

Developer Response

Fixed: 0d8a034f8daf973dddf94e0dde5d6b6ca08933c4

2. Medium - Incorrect slippage protection in ZapSwap redeem

ZapSwap contract enables users to use the protocol without having to swap to the base asset. Function `mint()` and `redeem()` have an additional parameter for min amount out to offer users slippage protection. In the function `redeem()` slippage protection is incorrectly implemented which could result in lost value for users.

Technical Details

Zap function `redeem()` receives the leveraged token, redeems it for the base asset then swaps the base asset for zap asset. It also has parameter `minZapAssetAmountOut_` which defines the minimum amount of zap token the user will receive. This way the user defines slippage protection when redeeming via ZapSwap.

The parameter `minZapAssetAmountOut_` is incorrectly used when calling the redeem function on the leveraged token. Leveraged token redeem has `minBaseAmountReceived` parameter which defines minimal amount of base token received, but instead of min base amount, min zap asset amount is passed. Because those two tokens are not the same, it could cause leveraged token redeem function to revert on any meaningful slippage protection defined by the user using `minZapAssetAmountOut_`.

Instead of applying slippage protection to the leveraged token redeem function, it should be checked only after swapping is done and the final amount of zap asset is received. This is implemented correctly in ZapSwap here.

Impact

Medium. Incorrect slippage protection could disable users from redeeming with slippage protection. This could force users to use redeem without the slippage protection and incur a big loss for users.

Recommendation

Remove the incorrect slippage parameter when calling redeem on leveraged token and pass zero instead. It is safe to use zero because the slippage protection is handled after swapping from base asset to zap asset.

```
targetLeveragedToken.redeem(
    leveragedTokenAmountIn_,
-    minZapAssetAmountOut_
+   0
);
```

Developer Response

Fixed: 98d21b6725d6c7ab727a2b43edaa0111af4ae8bb

Low Findings

1. Low - `GenesisLocker.totalStaked` will report an incorrect value

`GenesisLocker.migrateFor()` is used to move unlocked `TLX` tokens to the Staker.sol contract.

Technical Details

In the migration process, `GenesisLocker.migrateFor()` correctly deletes `_balances[msg.sender]` and `unlockTime[msg.sender]`, but it fails to decrease the `totalStaked` storage variable. Although `totalStaked` impacts the way rewards are streamed to stakers, as can be seen in `_globalCheckpoint()`, given that stakers can only exit once the entire amount of rewards has been streamed, we haven't identified a way in which this issue affects the rewards distribution process.

Impact

Low. `GenesisLocker.totalStaked` reports an incorrect value once a migration occurs.

Recommendation

Correctly account for the amount of `TLX` unstaked:

```
function migrateFor(address receiver_) public {
    uint256 amount_ = _balances[msg.sender];
    if (amount_ == 0) revert ZeroAmount();
    if (receiver_ == address(0)) revert Errors.ZeroAddress();
    if (!_shutdown && unlockTime[msg.sender] > block.timestamp)
        revert NotUnlocked();

    _checkpoint(msg.sender);

    delete _balances[msg.sender];
    delete unlockTime[msg.sender];
+   totalStaked -= amount_;

    _addressProvider.staker().stakeFor(amount_, receiver_);

    emit Migrated(msg.sender, receiver_, amount_);
}
```

Developer Response

Fixed: 8d13c9f61cabd2caa33c153aae9399b70d787c00

2. Low - `Referrals` insufficient validation during construction

`Referrals` has 2 storage variables, `Referrals.rebatePercent` and `Referrals.earningsPercent`, which are used to calculate how fees earned by the system are distributed between a user and the referrer he selected.

Technical Details

Within the contract's constructor, the values assigned to the 2 mentioned variables aren't validated to be such that `rebatePercent + earningsPercent <= 1e18`. In the case in which the above condition doesn't hold, the contract wouldn't allow its owner to fix the issue, as `Referrals.setRebatePercent()` and `Referrals.setEarningsPercent()` would both revert, calling for a redeployment of the contract to fix the issue. Furthermore, this issue will cause a DoS in `LeveragedToken.redeem()`, as the contract grants an approval for `fee_` here, but `Referrals.takeEarnings()` would try to pull an amount of tokens larger than `fee_`.

Impact

Low. Unlikely deployment error will cause DoS in `LeveragedToken` redeem flow.

Recommendation

Validate the values of parameters passed to `Referrals`'s constructor, e.g.:

```
constructor(
    address addressProvider_,
    uint256 rebatePercent_,
    uint256 earningsPercent_
) TlxOwnable(addressProvider_) {
+   require(rebatePercent_ + earningsPercent_ <= 1e18);
    _addressProvider = IAddressProvider(addressProvider_);
    rebatePercent = rebatePercent_;
    earningsPercent = earningsPercent_;
}
```

Developer Response

Fixed: 3343136ffdd4c8eb435f63c53027b62fb4700e8a

3. Low - `LeveragedToken.redeem()` may revert unexpectedly

Within the `LeveragedToken.redeem()` method, some fees are detracted from the amount the sender is to receive from redeeming his `LeveragedToken`s and are distributed as rewards to users in the referral program and to stakers who have locked funds in `Staker`.

Technical Details

The fee allocated for `TLX` stakers is donated by calling `Staker.donateRewards()`, which occurs within `LeveragedToken.redeem()` at L120. The rewards donation occurs on the condition that there is a non-zero amount of `TLX` to donate to the `Staker` contract, and that such contract has a non zero amount of funds staked: L118. Inspecting `Staker.donateRewards()`, we found that this method enforces a stricter condition than the one mentioned above: the method checks that `Staker.totalStaked - Staker.totalPrepared != 0` at L38, reverting if such condition holds.

As a result, in the unlikely event in which **all** of `TLX` staked in `Staker` enters the unstaking process, every instance of `LeveragedToken` will suffer a DoS on its `redeem()` flow.

Following is a PoC we've developed to confirm our finding: gist. You should add it to `test/Staker.t.sol` and run it with `forge t --mt testDonateRewards_FullUnstakeDOS -vv`

Impact

Low. An unlikely event will cause a short-lived DoS.

Recommendation

Modify the condition in `LeveragedToken.redeem()` to also take into account this situation:

```
...
IStaker staker_ = addressProvider_.staker();
uint256 amount_ = fee_ - referralAmount_;
- if (amount_ != 0 && staker_.totalStaked() != 0) {
+ if (amount_ != 0 && staker_.totalStaked() > staker_.totalPrepared()) {
    baseAsset_.approve(address(staker_), amount_);
    staker_.donateRewards(amount_);
}

// Redeeming
_burn(msg.sender, leveragedTokenAmount_);
...
```

Developer Response

Fixed: efcd00332427766cea0209a2e7605e1de00a182f

4. Low - Missing validation before setting `baseForAllTlx`

Setting the variable `baseForAllTlx` to zero will disable bonding.

Technical Details

The bonding contract has the function `setBaseForAllTlx()` without the validation that the value is not zero. If the value is set to zero, calling the function `_exchangeRate()` will revert every time because of dividing with zero. This will disable any further usage of bonding.

Token allocation for boding is 42%. Because bond value grows over time, it is recommended to disable any pausing which would happen by setting `baseForAllTlx` to zero.

Impact

Low. Unintended or malicious setting `baseForAllTlx` to zero will disable bonding.

Recommendation

Verify that `baseForAllTlx` is not zero before setting the new value in the setter function `setBaseForAllTlx()`.

Developer Response

Fixed: 8cc419d8ee324db88fcc058723458e5c099488857

5. Low - Inconsistent value for `GenesisLocker.lockTime`

There is a discrepancy between docs and code config for value `GenesisLocker.lockTime`.

Technical Details

The docs define the genesis locking period value as 26 weeks which is equal to 182 days. But in the config, this variable is defined as 180 days.

Impact

Low.

Recommendation

Define the same value for genesis lock time in both the docs and the config.

Developer Response

Fixed: d07675bf96e5db1f6cd8531eb71b120416d4a88a

6. Low - `ChainlinkAutomation` doesn't adhere to Chainlink's recommendations

`ChainlinkAutomation` implements Chainlink's `AutomationCompatibleInterface`, which serves to specify custom logic triggers for Chainlink keepers to upkeep the protocol.

Technical Details

`ChainlinkAutomation` fails to respect the recommendation provided by `AutomationCompatibleInterface` here, as `ChainlinkAutomation.performUpkeep()` fails to validate the `performData_` argument it's passed. In particular, the method doesn't validate that every entry in the `rebalanceTokens_` array is a legitimate instance of `LeveragedToken`, registered within `LeveragedTokenFactory`.

Impact

Low. Not following Chainlink's guidelines may lead to unexpected issues in the future.

Recommendation

Within `ChainlinkAutomation.performUpkeep()`, correctly validate the dynamic array of addresses passed as an argument. In particular, ensure that:

1. `token_` is always a `LeveragedToken` registered within `LeveragedTokenFactory`.
2. Validate that `block.timestamp >= _nextAttempt[token_]` for every `token_`.

Developer Response

Fixed: dc587e862a14053607d8a2d5cbaf1714ecb0b3ba

7. Low - Referrals system is unfair for referrers

Users can change the referral code used to earn part of the redeem fee at any time. This can lead to an unfair system for referrers because they could lose fees from the users they have brought to the TLX protocol.

Technical Details

The referral program is intended to reward referrals that have brought new users to the protocol. They usually provide their referral code to use for registration, the first interaction with the new protocol. As the user brings value to the protocol, the referrer is eligible to get part of the value. In the TLX, it is done in Referrals contract and referral fee is earned by getting part of the redeeming fee. This is a fair system for both the user and the referrer because they both earn part of the fee.

The user has the option to update his referral code at any time. By changing the referral code, part of the fee will be directed to another referrer which can be unfair to the first referrer because he brought the user to the protocol. This leaves an opportunity for referrers to basically buy users by offering them some reward if they change referral code.

Impact

Low. Referrers could lose their part of the fees if the user changes the referral code.

Recommendation

Disable the option to update the referral code after it is set.

An additional constraint could be set so that only new users can update the referral code, depending on the protocol preferences. This would limit the rewards only to referees who have brought new users to the protocol.

Developer Response

Fixed: c374a6c9ff083ff9ea355c33818e83c2fdd31b6d

8. Low - `LeveragedToken.targetLeverage` may exceed perpetual's maximum leverage

Each instance of `LeveragedToken` has an immutable `targetLeverage`, which defines the leverage level the underlying perpetual position should aim towards at all times.

Technical Details

`Config` defines the MAX_LEVERAGE constant as `50e18`, specifying that instances of `LeveragedToken` are allowed to have a maximum of `50x` target leverage. Analyzing the markets intended to be utilized for the initial `LeveragedToken`s, it was found that Synthetix Perpetuals for `OP` and `LINK` allow for a maximum `27.5x` leverage, while for other assets like `BTC` and `ETH` a maximum of `55x` is allowed.

Impact

Low. Insufficient validation during `LeveragedToken` construction allows deploying tokens that will never reach their target leverage.

Recommendation

Within `LeveragedTokenFactory.createLeveragedTokens()`, also validate that `targetLeverage_` is smaller than the `targetAsset_` market's max leverage.

Developer Response

Fixed: ecda7b4af10b0b9077263035a00cfeed994ea384

9. Low - The same rebalance threshold is not optimal for all leveraged values

The rebalance threshold value is defined only per token. Token can have different leveraged value, from 1x to 50x. Having the same rebalance threshold for all leverage targets could be suboptimal.

Technical Details

The rebalance threshold value is used to trigger the rebalancing of LeveragedToken. Maximum leverage value is set to 50x. If the same threshold value is used for leverage values, it will have to be configured to cover the worst-case scenario, in this case, 50x. This would result in too often calls to leveraged tokens where the target leverage is set to 1x.

The users pay the price for too often calling rebalance, once in higher gas paid for a mint and redeem call, if the rebalance is called. The second time when rebalance is triggered by the Chainlink automator. Also, additional fees are paid to Synthetix when submitting the rebalance off-chain order.

Enabling a more fine-grained setting of the threshold value, by splitting the threshold value per token and the target value will lead to more efficient rebalance calls.

Impact

Low. Calling rebalance too often is costly for the users.

Recommendation

Extended the mapping `_rebalanceThresholds` to enable storing rebalance threshold value per token and target leverage value.

Developer Response

We disagree this is an issue. We don't have any intention of having the same rebalance thresholds for all leveraged tokens. And the codebase already supports having a different rebalance threshold per leveraged token, and the setting of those different rebalanced thresholds. The _rebalanceThresholds mapping already stores the rebalance threshold values per leveraged token.

10. Low - ZapSwap routes can be optimized

Routes for ZapSwap can be changed to be more optimal for the end user.

Technical Details

ZapSwap routes are defined in the library `ZapAssetRoutes`. Routes that can be changed to more optimal are:

- DAI can be swapped with better rates to USDC.e on Uniswap V3.
- USDT can be swapped with better rates to USDC.e on Uniswap V3.

Swap data on March 4, 2024:

DAI	USDC.e on Uniswap V3	USDC.e on Velodrome V2
100000	99979.9	99724.11
10000	9998.62	9995.69
1000	999.89	999.64

USDT	USDC.e on Uniswap V3	USDC.e on Velodrome V2
100000	100052	99059.80
10000	10006.2	9995.37
1000	1000.66	999.67

Impact

Low. Use more optimal swapping routes to provide better value for the end user.

Recommendation

Change swapping routes for DAI and USDT to USDC.e. Swap them on Uniswap V3 instead of Velodrome V2 if the rates stay better on Uniswap V3.

Developer Response

Fixed: 42671703ff5034d92113ac3fd25e7e420578f0f7

11. Low - Incorrect market for getting min keeper fee

Variable `_futuresMarketSettings` is defined as interface `IFuturesMarketSettings` is used to get min keeper fee for Futures Market. Instead, Perps V2 Market should be used because all Synthetix integration is done in Perps V2 Market, the fee should be fetched for this market.

Technical Details

`_futuresMarketSettings` is used to get `_minKeeperFee()`. TLX protocol is depositing only to Perps V2 Market, it should use `IPerpsV2MarketSettings` interface to get `_minKeeperFee`.

In the test `SynthetixHandler` is deployed using `PERPS_V2_MARKET_SETTINGS` which address on OP is `PerpsV2MarketSettings` contract. This means that the address used in the tests is correct, but incorrect naming and interface are defined in the `SynthetixHandler` contract. The tests have passed because both interfaces have the function `_minKeeperFee()`.

Incorrect naming and interface could lead to defining the Synthetix market when deploying `SynthetixHandler`.

Impact

Low. Using the incorrect market to get min keeper fee value will result in incorrect calculations.

Recommendation

Use the correct market to get min keeper fee. Rename `_futuresMarketSettings` to `_perpsMarketSettings` and define it as interface `IPerpsV2MarketSettings`.

Developer Response

Fixed: dd27a14df5a1a45931cfa7bc53f181f0a8e76ac3

Gas Saving Findings

1. Gas - Inherit variable instead of defining it again multiple times

Variable `_addressProvider` is defined as private in abstract contract `TlxOwnable`. Change the visibility to `internal` so other contracts can access it instead of defining the same value again.

Technical Details

Other contracts that extend `TlxOwnable` contract but also define `_addressProvider`:

- Airdrop.sol
- Bonding.sol
- ChainlinkAutomation.sol
- LeveragedToken.sol
- LeveragedTokenFactory.sol
- ParameterProvider.sol

- Referrals.sol
- RewardsStreaming.sol
- ZapSwap.sol

Impact

Gas savings.

Recommendation

Set variable `_addressProvider` visibility to `internal` and use it instead of defining a new variable with the same name and value in the defined contracts above. The variable is immutable in all cases, so it can be safely inherited.

Developer Response

We will not implement this change. We think it is not worth the gas savings in this case.

2. Gas - Remove unused code

Some code is never used and can be removed to save gas during deployment.

Technical Details

Functions that are declared but never used include:

- `_orderFee()`
- `scaleFrom()`
- `scaleTo()`
- `min()`
- `tryGet()`
- `get()`
- `cancelLeverageUpdate()`

Impact

Gas savings.

Recommendation

Remove or comment out the unused functions.

Developer Response

Fixed: bc55169a36132628cbf25eb738212630dbb18f6b

3. Gas - Use more gas efficient merkle proof validation method

Merkle proof validation uses suboptimal method to check proof's validity.

Technical Details

Using `MerkleProof.verify()` within `Airdrop._isValid()` bares the cost of copying the `merkleProof_` array from calldata to memory. For cases in which the proof array is passed as calldata, OpenZeppelin's library offers a better alternative: `verifyCalldata()`

Impact

Gas savings.

Recommendation

Use `MerkleProof.verifyCalldata()`.

Gas savings data from provided tests:

`testClaim() (gas: -413 (-0.358%))`

`testRevertsWhenAlreadyClaimed() (gas: -438 (-0.426%))`

`testRevertsForInvalidProof() (gas: -411 (-2.409%))`

Developer Response

We will not implement this change. We think it is not worth the gas savings in this case.

4. Gas - Less strict validation check can be removed

`Referrals.setRebatePercent()` is used by the contract owner to set the `Referrals.rebatePercent` storage variable.

Technical Details

The checks at L101 and L102 are equivalent when `Referrals.earningsPercent == 0` and the latter is stricter when `Referrals.earningsPercent > 0`.

Impact

Gas savings.

Recommendation

Remove the less strict check.

Developer Response

Fixed: 44befafe284a94850f0ac637c09b174692f8538d

5. Gas - Declare variables immutable when possible

Using immutable variables can provide gas savings compared to non-immutable variables if the variables only need to be set once.

Technical Details

The variable `deadline` can be set as immutable. It is only set in the constructor and is not changed after that.

Impact

Gas savings.

Recommendation

Declare the `deadline` variable as immutable for gas savings.

Developer Response

Fixed: 1543ec767361eadbe333d124039742160dd01ae1

6. Gas - Optimize Airdrop contract

`Airdrop` can be optimized by removing variable `totalClaimed` and changing function `claim()`.

Technical Details

Removing the storage variable `totalClaimed` and access to it can save gas costs. This value can be replaced by checking TLX balance: `_addressProvider.tlx().balanceOf()`. Gas savings on `claim()` function could encourage more users to claim the token and, more importantly, save the gas costs for the end users.

Impact

Gas savings.

Recommendation

Remove variable `totalClaimed` and change function `claim()` to following:

```
function claim(
    uint256 amount_,
    bytes32[] calldata merkleProof_
) external override {
    // Checking claim is valid
    if (block.timestamp > deadline) revert ClaimPeriodOver();
    if (hasClaimed[msg.sender]) revert AlreadyClaimed();
    if (!_isValid(msg.sender, amount_, merkleProof_)) {
        revert InvalidMerkleProof();
    }
    uint256 balance = _addressProvider.tlx().balanceOf(address(this));
    uint256 airdropAmount_ = _airdropAmount;
    bool completed_ = balance == 0;
    if (completed_) revert AirdropCompleted();

    if (amount_ > balance) {
        amount_ = balance;
    }

    // Updating state
    hasClaimed[msg.sender] = true;
    _addressProvider.tlx().transfer(msg.sender, amount_);
    emit Claimed(msg.sender, amount_);
}
```

Change in function `mintUnclaimed()` line L75:

```
- uint256 unclaimed_ = _airdropAmount - totalClaimed;
+ uint256 unclaimed_ = addressProvider_.tlx().balanceOf(address(this));
```

Gas savings data from provided tests:

`testUpdateMerkleRootRevertsForNonOwner() (gas: -6 (-0.035%))`

`testMintUnclaimedFailsForNonOwner() (gas: -11 (-0.064%))`

`testMintUnclaimedFailsWhenStillOngoing() (gas: -11 (-0.065%))`

`testClaimingAfterDeadlineReverts() (gas: -13 (-0.114%))`

`testMintUnclaimed() (gas: -410 (-0.695%))`

`testRevertsForInvalidProof() (gas: -416 (-2.438%))`

`testInit() (gas: -2982 (-8.453%))`

`testClaim() (gas: -22013 (-19.071%))`

`testRevertsWhenAlreadyClaimed() (gas: -21000 (-20.410%))`

Developer Response

We will not implement this change. We think it is not worth the gas savings in this case.

7. Gas - Remove unneeded approval

SynthetixHandler approves base asset to Synthetix market which is not needed for a transfer margin call.

Technical Details

Approval can be safely removed because the PerpsV2Market calls burn on sUSD and that call doesn't need approval from the SynthetixHandler.

Impact

Gas savings.

Recommendation

Remove unneeded approval call.

Developer Response

Fixed: ddd1b3d5fdac181d3429f563cba70447df0f3580

Informational Findings

1. Informational - False events can be emitted

There is a possibility of emitting events for actions that didn't happen, like adding and removing rebalancer addresses from the set.

Technical Details

The function `addRebalancer()` adds provided address to a set of rebalancers and emits `RebalancerAdded` event. If the provided address was already in the set, it wouldn't be added again, so the emitted event would be false. OZ function to add the item to a set returns true when the item is added. This value can be checked before emitting the event.

The function `removeRebalancer()` will emit `RebalancerRemoved` event if the address wasn't removed from set, in the case it didn't exist in the set. OZ function to remove the item from a set returns true when the item is removed.

Impact

Informational.

Recommendation

Emit events only for actions that happened.

Developer Response

Fixed: 84d3ccd6583862e6de73ebad7c5467e9adca4b31

2. Informational - Missing event emits

Some functions that transfer values or modify state variables do not have events. Events can assist with analyzing the on-chain history of contracts and are therefore beneficial to add in important functions.

Technical Details

Functions that could have events added include:

- `setBaseForAllTlx()`
- `launch()`
- `setIsPaused()`
- `resetFailedCounter()`
- `setAssetSwapData()`
- `removeAssetSwapData()`

Impact

Informational.

Recommendation

Add events to the functions listed above.

Developer Response

Fixed: becfd9568db7f3dcce2ce8878a963e678eec9f7d

3. Informational - Typos

Some comments have typos.

Technical Details

- A comment in Timelock has a typo `fro`.

Impact

Informational.

Recommendation

Fix the typos.

Developer Response

Fixed: 1eecec69df35e55fe592f657f4fbc22863296108

4. Informational - Remove unused imports

Remove unneeded imports from contracts for cleaner code.

Technical Details

RewardsStreaming.sol has unused import `Errors`.

Impact

Informational.

Recommendation

Remove specified import or move errors from interfaces to Errors.sol.

Developer Response

Fixed: 14a2b479784e31de703d834d29a99bacd5be8961

5. Informational - Possible lost value if the token will have a high supply

In Unstakes library uses unsafe casting from `uint256` to `uint192` which can lead to loss value for the user.

Technical Details

For unstake flow, user can provide values in `uint256` precision which is then cast to smaller precision `uint192` without any checks. If the user provides the amount above `uint192`, this amount will be transferred from him to Staker contract but only `type(uint192).max` value will be stored in unstake queue. The user will be able to unstake the amount stored in the queue which is lower than the amount sent for unstaking which is a direct loss for the user.

TLX total supply is defined in config to a value way below `type(uint192).max` meaning that there is no possibility to lose value in unstake with the current config values.

Impact

Informational.

Recommendation

Use higher precision for unstake queue amounts if the TLX total supply will be changed to value above `type(uint192).max`.

Developer Response

We disagree this is an issue. We don't have plans of modifying the total supply from what is currently in the Config

6. Informational - Improve events

Values in some events could be changed to provide a more accurate value for off-chain analysis.

Technical Details

Function `takeEarnings()` emits event `EarningsTaken` with `fees_` as second parameter. This value could be incorrect because the token transferred value is done using variable `totalAmount_`. Additionally, two events could be emitted for better analysis, one for `referrer_` and one for `user_`.

In the Staker contract, event `PreparedUnstake` contains only address of message sender, adding amount value could be useful.

Impact

Informational.

Recommendation

Change events mentioned above to improve off-chain analysis.

Developer Response

Fixed: 081c1023f19c6a476cb131d5b8ed9d17b58cb3c6

7. Informational - Misleading function naming

The function name `Airdrop.mintUnclaimed()` is misleading because the function implementation doesn't mint any token.

Technical Details

Even the function NatSpec states that the function is minting unclaimed tokens. In the function implementation, it is clear that the function is not minting any token but just sending unclaimed TLX tokens to the treasury.

Impact

Informational.

Recommendation

Rename the function to `transferUnclaimed()`, rename the event `UnclaimedMinted` to `UnclaimedTransferred` and change the function NatSpec. Remove incorrect comment.

Developer Response

Fixed: bfc56e3424d78335afae041da1eb97409bdf78c1

8. Informational - Inheritance improvement

`RewardsStreaming` is an abstract contract which is inherited by two contracts. Inheritance and abstract contracts can eliminate duplicated code, resulting in smaller code and less room for error.

Technical Details

Some abstract functions are overwritten in only one contract, leading to difficult code reading. `_latestIntegral()` has a default implementation, but it is also overridden in `GenesisLocker`. Another contract `Staker` that implements `RewardsStreaming`, doesn't override this function which leads to a bit of confusion in the abstract contract. For example, the abstract contract is using `_latestIntegral()` which has default implementation making it hard to track if it is overridden later.

Because there are only two implementations of the abstract `RewardsStreaming` contract, the function `_latestIntegral()` could be defined as virtual. This way, each contract extension would have its own implementation of this function resulting in clear code in the abstract contract.

Impact

Informational.

Recommendation

In the `RewardsStreaming` contract define the function `_latestIntegral()` as virtual. Each contract that extends it should define its implementation. Try not to override functions from the abstract contract, that are not virtual, especially if there are only two contracts that extend it. Each contract can have its own implementation if needed. Follow the implementation of `activeBalanceOf()`.

Developer Response

Fixed: 0bb8458c59fd9c3ad799d4be2e2017dc1da0666e

9. Informational - Upgrade OpenZeppelin dependency

The OpenZeppelin contracts dependency is version 4.9.1, which is outdated. Consider updating to a newer version.

Technical Details

OZ library v4.9.1 is not the latest version available. Consider upgrading to v4.9.6 which fixes some minor issues in v4.9.1.

Impact

Informational.

Recommendation

Upgrade OZ dependency to a newer version.

Developer Response

Fixed: aff09dd6956c6f572f425bf235c14a232c8d4031

10. Informational - Split ChainlinkAutomation into multiple upkeep tasks

`ChainlinkAutomation` defines the maximum size of the array it can check in `checkUpkeep()`. As the number of LeveragedToken grows, ChainlinkAutomation could end up favouring tokens at the start of the array.

Technical Details

The function `checkUpkeep()` has input parameter `checkData` which is defined during the upkeep task registration process. This parameter can be utilized to define the start index of the array containing all leveraged tokens. It enables deployment of new upkeep tasks when the size of the leveraged tokens grows above `_maxRebalances` value. Each task will only check the maximum number of leveraged tokens that can be rebalanced. This will cost more gas because sometimes it will send an array to rebalance just one leveraged token, but it will not favour leveraged tokens at the start of the array.

There is no information in Chainlink Automation docs on how often function `checkUpkeep()` is checked or if there is any delay after `performUpkeep()` has been triggered. It is recommended to verify there are no delays for tokens specified at the end of the leverage token list. Use a minimal array size of 30, 5 tokens with 3 different leverage values for long and short, where all tokens need a rebalance call.

Impact

Informational.

Recommendation

Define Chainlink Automation the start array index in static data `checkData`. Register multiple upkeep tasks as the number leveraged tokens grow, each with new start index.

Developer Response

We are aware of the restriction and accept that risk. The amount of leveraged tokens currently handled by the ChainlinkAutomation contract is within an acceptable range. If we have more leveraged tokens later on that we think may require additional automation. Then we will build new automation contracts to handle this. The current automation contract is not designed to scale infinitely.

It's worth noting that the proposed fix would also not scale indefinitely. As the `leveragedTokenFactory().allTokens()` call would eventually reach the gas limit of the checkUpkeep function, even with the start index passed through.

11. Informational - `Timelock` allows instant arbitrary calls

`Timelock` allows its owner to propose, execute and cancel a set of external calls the contract will make. The execution of a proposal may only occur after a certain time delay has passed since the moment it was proposed.

Technical Details

`Timelock`'s owner can create a proposal by calling `Timelock.createProposal()`, passing it an array of `(address target, bytes data)`. This method sanitizes the `calls_` array by feeding it into `Timelock._validateCallsAndGetMaxDelay()`, which checks that `target` is never `address(0)` and returns the longest delay, saved in storage, for the selectors specified.

In the case in which the `calls_` array contains a set of selectors which have never been assigned a delay, `Timelock._validateCallsAndGetMaxDelay()` will return `0` as the delay for the proposal, which in turn will make it immediately executable within `Timelock.executeProposal()`.

Impact

Informational.

Recommendation

There are several ways to fix this issue:

1. Set a default delay for calls linked to function selectors that haven't been assigned a delay.
2. Set a minimum delay that all proposals must respect, which OpenZeppelin's `TimelockController` implements here.
3. During creation, force proposals to contain calls exclusively linked to function selectors that have a non-zero delay.

`Timelock` implies standard implementation of `TimelockController` from OpenZeppelin. Using a different name would be more appropriate to avoid confusion with the current implementation.

Developer Response

Renamed `Timelock` to `ProxyOwner`: f0a196e392596c153a133d5ef02b876360338af6

12. Informational - Protocol owned liquidity could lose a lot of value

TLX accepts all Leveraged Tokens for bonding. If the Leveraged Token has high volatility and a high-leverage factor, TLX could accumulate risky tokens in protocol owned liquidity (POL). If these tokens lose a lot of value or even get liquidated before being swapped for other tokens, the protocol will lose money.

Technical Details

The attacker, or even regular users, could bond a lot of risky Leveraged Tokens, with high volatility and a high leverage factor. Because the bonding mechanism can't limit or decline risky tokens or even pause bonding, this could lead to POL accumulating only high-leveraged tokens. This is risky because there is no automated mechanism to swap those tokens. It is planned to do it weekly which could lead to a lot of loss value for TLX. The docs specify: "This liquidity, while initially being acquired in the form of leveraged tokens, undergoes a weekly rebalance".

Impact

Informational.

Recommendation

In the bonding mechanism, implement granularity for accepted Leveraged Tokens. Accept only

Leveraged Tokens that can be handled safely in a weekly rebalance schedule. Another option is to implement an automated flow for redeeming Leveraged Tokens for a base asset. Additionally, swap it for ETH because POL will go to ETH pools.

Developer Response

We disagree that this is an issue and consider this out of scope of the audit. That tokens are send to the POL is the intended protocol design. It is known and intentional that there are risks in the POL holding volatile assets. Just as there is a chance that the POL could lose value from this, there is an equal chance that it can gain value. Sure it's true, but it's not a vulnerability or bug.

Final Remarks

TLX enables the users to enter leveraged positions in Synthetix which are constantly kept at the same leveraged level by rebalancing positions in Synthetix. The users pay the fees for redeeming leveraged tokens. Even though the position is kept at the same leveraged level, the position can get liquidated meaning the leveraged tokens are worth zero. Users should be aware of the risks and fees when using the TLX protocol and high-leveraged positions. Chainlink Automation handles rebalancing, which is off-chain and could not be fully reviewed. Additional tokenomics is implemented in the protocol, like airdrop, bonding, and referrals. The bonding mechanism accepts all leveraged tokens which are exchanged for TLX tokens and sent to the treasury. By accepting and keeping high-leverage tokens, the treasury will be exposed to a high risk of price movements.

The system is designed to be immutable with some parameters and contract addresses that the owner can change. The owner is intended to be a Timelock contract which will delay the execution of the owner's actions. This will enable users to have time to react if the owner decides to change the parameters. There are no upgradeable contracts, only delegate calls to the SynthetixHandler contract which acts as a library to save gas costs.