Project TXA Security Review

Version 1.0

September 4, 2023

Conducted by:
Gogo, Independent Security Researcher
deadrosesxyz, Independent Security Researcher

Table of Contents

1 About Gogo and deadrosesxyz

Gogo and deadrosesxyz are independent security researchers specializing in Solidity smart contracts auditing and bug hunting. Having conducted numerous solo and team smart contract security reviews, they always strive to deliver top-quality security auditing services. For security consulting, you can contact them on Twitter, Telegram, or Discord - *@gogotheauditor* & *@deadrosesxyz*.

2 Disclaimer

Audits are a time, resource and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to find as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

3 Risk classification

Severity	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 significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

Overview

Project Name	Project TXA
Repository	https://gitlab.com/projecttxa/txa-dsl/txa-network-contracts
Commit hash	c4865e55b0522cd058870099427fd5d063e1bc81
Documentation	https://hackmd.io/@adklempner/r1LR6F69n
Methods	Manual review

Scope

src/

Issues Found

Critical risk	7
High risk	0
Medium risk	13
Low risk	11

5 Findings

5.1 Critical severity

5.1.1 Adversary can release locks in wrong order to cause loss of funds

Severity: *Critical Risk*

Context: Staking.sol#L194-L209

Description: When validators use collateral from the Staking contract, the Rollup contract calls the `lock` function which checks the available balance of USDT and TXA tokens deposited by stakers for the following 3 periods of 60 days.

When the corresponding state root for a `lockId` has been confirmed, anyone can call the `unlock` function which will release the locked amount for the lock with `lockId`. Then, the function goes over the 3 periods from which the funds were taken (locked) and release them starting from the nearest one, by using the entire locked amount for each period.

Given this context, consider the following scenario:

There are 5 consecutive tranches (periods) with the following staked and locked amounts in them:

Stake amounts	5	5	10	5	5
Locked amounts	0	0	0	0	0

When the first 3 ones are active, `lock` is called with a value of 15 (returned `lockId` is 1). It locks 5 tokens in each of the first 3 tranches:

Stake amounts	5	5	10	5	5
Locked amounts	5	5	5	0	0

Time goes by (2 periods) and now the last 3 are the active ones. `lock` is once again called with a value of 15 (returned `lockId` is 2):

Stake amounts	5	5	10	5	5
Locked amounts	5	5	10	5	5

Then, when it's time to unlock both `lockId`s, the following code will release the funds to stakers:

```
uint256[3] memory tranches = getActiveTranches(lockRecord.blockNumber);
uint256 amountToUnlock = lockRecord.amountLocked;
// loop through tranches and unlock
for (uint256 t = 0; t < tranches.length; t++) {
    if (amountToUnlock == 0) break;
    uint256 locked = totals[lockRecord.asset][tranches[t]].locked;
    if (locked == 0) continue;
    if (amountToUnlock <= locked) {
```

```
            totals[lockRecord.asset][tranches[t]].locked -= amountToUnlock;
            amountToUnlock = 0;
        } else {
            // set available in tranche to: available - _amountToLock
            totals[lockRecord.asset][tranches[t]].locked = 0;
            amountToUnlock -= locked;
        }
    }
    locks[_lockIds[i]].amountLocked = 0;
```

If a user unlocks the second lock before the first one, the following will happen.

Before unlocking any lock:

Stake amounts	5	5	10	5	5
Locked amounts	5	5	10	5	5
Unlocked amounts	0	0	0	0	0

After unlocking the lock with a `lockId` of 2:

Stake amounts	5	5	10	5	5
Locked amounts	5	5	10	5	5
Unlocked amounts	0	0	10	5	0

After unlocking the lock with a `lockId` of 1:

Stake amounts	5	5	10	5	5
Locked amounts	5	5	10	5	5
Unlocked amounts	5	5	10	5	0

Due to the faulty accounting logic, the last period/tranche ends up with forever locked tokens.

Recommendation: Consider refactoring the way the `unlock` function accounts for the released tokens or allow users to call the function only with `lockId`s in consequent order.

Resolution: Resolved. The recommended fix was implemented.

5.1.2 Lock will be lost when a state root has been replaced with a new one

Severity: *Critical Risk*

Context: Rollup.sol#L88, Rollup.sol#L94-L101

Description: When validators propose a state root for the current epoch in the Rollup contract they have to lock up some amount of TXA tokens from the Staking contract. Then, the `lockId`, `_stateRoot` and `epoch` are saved in the contracts storage in the `lockIdStateRoot` mapping so that the Staking

contract can later check if the `_stateRoot` has been confirmed and not marked as fraudulent. If these two conditions are met, the locked funds are released back to stakers.

Validators also have the option to replace the state root proposed for a given epoch by calling `replaceStateRoot` and providing a new state root. This can be used if for example the validator made a mistake or the state root has been marked as fraudulent. The function will update the `proposedStateRoot` for the passed `_epoch` as well as the `proposalBlock` of the new state root. However, it will not create a new lock, but instead use the already created one for that epoch:

```
function replaceStateRoot(bytes32 _stateRoot, Id _epoch) external {
    if (!manager.isValidator(msg.sender)) revert CALLER_NOT_VALIDATOR();
    if (lastConfirmedEpoch >= _epoch) revert("Cannot replace state root that's been
        confirmed");
    if(processedSettlements[_epoch][_stateRoot].length() > 0) revert("A settlement
        has already been processed for this state root");

    proposedStateRoot[_epoch] = _stateRoot;
    proposalBlock[_stateRoot] = block.number;
}
```

The problem here is that the previous lock is not updated with the new `_stateRoot`. Therefore, since the previous `_stateRoot` will not be confirmed or could have been marked as fraudulent, the lock will be unable to be released resulting in a permanent loss of funds for the stakers.

Recommendation: Update the lock so that it uses the new `_stateRoot` instead of the old one when it gets replaced.

Resolution: Partially resolved. The lock is updated even if the replaced state root has been marked as fraudulent while it should be slashed instead and a new one should be created.

5.1.3 Staker rewards are lost when a settlement for a confirmed state root is processed

Severity: *Critical Risk*

Context: Rollup.sol#L187-L194

Description: When processing a settlement, an `insuranceFee` and `stakerReward` are calculated and deducted from the trader's balance. The `stakerReward` is used to incentivise stakers in the Staking contract to provide liquidity in TXA and USDT tokens that can be locked when a settlement for an unconfirmed state root is processed.

This `stakerReward` is not paid when the state root of the processed settlement has been confirmed as there's no need to lock any collateral. However, it is still deducted from the trader's balance meaning that it becomes lost forever.

Recommendation: Consider implementing the following change on line 193:

```
state.settlement.balanceBefore.amount - (insuranceFee + requiresCollateral ?
    stakerReward : 0)
```

Resolution: Resolved. The staker rewards are now sent to the insurance fund if no collateral is required.

5.1.4 Permanent DoS if a settlement for a fraudulent state root gets processed

Severity: *Critical Risk*

Context: Rollup.sol#L120, Rollup.sol#L99

Description: If a validator makes a mistake by proposing an invalid state root for the current `epoch` or marking a valid state root as `fraudulent`, they have the option to replace it using the `replaceStateRoot` function in order to not block the next epochs from getting confirmed:

```
function replaceStateRoot(bytes32 _stateRoot, Id _epoch) external {
    if (!manager.isValidator(msg.sender)) revert CALLER_NOT_VALIDATOR();
    if (lastConfirmedEpoch >= _epoch) revert("Cannot replace state root that's been
        confirmed");
    if (processedSettlements[_epoch][_stateRoot].length() > 0) {
        revert("A settlement has already been processed for this state root");
    }

    proposedStateRoot[_epoch] = _stateRoot;
    proposalBlock[_stateRoot] = block.number;
}
```

Validators can also process a signed update corresponding to a state root that has been proposed, but not yet confirmed. This is done through `processSettlements` by using a collateral from the Staking contract.

An edge case occurs when a validator proposes an invalid state root and then executes a signed update corresponding to it, but before it gets confirmed. This would increase the length of `processedSettlements` for this `_epoch` and `_stateRoot`. Then, the `_stateRoot` would get marked as fraudulent using the `markFraudulent` function, which would lead to the `confirmStateRoot` function *not* being able to proceed with confirming the `_stateRoot` for the current `epoch`:

```
function confirmStateRoot() external {
    if (!manager.isValidator(msg.sender)) revert CALLER_NOT_VALIDATOR();
    lastConfirmedEpoch = lastConfirmedEpoch.increment();

    bytes32 stateRoot = proposedStateRoot[lastConfirmedEpoch];
    if (stateRoot == "") revert("Trying to confirm an empty state root");

    uint256 blockNumber = proposalBlock[stateRoot];
    if (block.number < blockNumber + manager.fraudPeriod()) {
        revert("Proposed state root has not passed fraud period");
    }

    if (fraudulent[lastConfirmedEpoch][stateRoot]) revert("Trying to confirm a
        fraudulent state root");

    confirmedStateRoot[lastConfirmedEpoch] = stateRoot;
}
```

The only way to unblock the `confirmStateRoot` function would be by calling `replaceStateRoot` with a valid `_stateRoot` for the blocked `_epoch`. However, this function will also revert because of the following check:

```
if (processedSettlements[_epoch][_stateRoot].length() > 0) {
```

```
    revert("A settlement has already been processed for this state root");
}
```

Therefore, there would be no way to continue updating the project state which means a permanent Denial of Service for the whole project's functionality.

Recommendation: The fraud system is not yet completely implemented and in scope for this audit. However, consider what should be the intended behaviour when a settlement has been processed for a fraudulent state root. Given the current state of the contract, such scenario could lead to bricking all users' funds stored in the Portal contracts.

Resolution: Resolved. The `processedSettlements` check is not performed if the replaced state root has been marked as fraudulent.

5.1.5 Stakers will receive rewards from tranches that they have no deposit in

Severity: *Critical Risk*

Context: Staking.sol#L246-L250

Description: The current validation in `claim` to prevent a trader from claiming rewards that do no belong to them looks as follows:

```
// Deposit should be eligible for the given lock record
if (
    depositRecord.blockNumber >= lockRecord.blockNumber
        || lockRecord.blockNumber > depositRecord.unlockTime - manager.fraudPeriod()
        || lockRecord.asset != depositRecord.asset
) continue; // @audit If these conditions are not true, the deposit is eligible
```

There's a missing check to prevent users from claiming rewards for tranches that they were not part of. If a trader has staked tokens for e.g. 10 years ahead, they will be eligible to claim rewards for all locks/tranches during this time since there's no check that the `depositRecord.unlockTime` is less than the 3rd active tranche that the lock used tokens from.

Recommendation: Do not let traders claim rewards if their `depositRecord.unlockTime` is greater than the 3rd active tranche at the time the lock was created.

Resolution: Pending.

5.1.6 The total collateral available to lock is incorrectly calculated

Severity: *Critical Risk*

Context: Staking.sol#L156, Staking.sol#L246-L250

Description: The Staking contract allows validators from the Rollup contract to lock certain amounts of USDT and TXA tokens when they want to process an order settlement for a state root that has not been confirmed yet.

Stakers can deposit their token in certain tranches determined by the lockup time duration. Each tranche has a `PERIOD_TIME` of 60 days.

When calculating the available amount to lock, the contract iterates through the 3 nearest tranches and calculates the total deposited amount and then subtracts the total locked amount. This way the Rollup can use tokens from users that have locked their tokens for the next 3 active periods (~180 days).

The problem here is that the for loop contains an early `break` that breaks it if the needed amount to lock has been found in the first or second nearest active tranches:

```
uint256 totalAvailable = 0;
uint256 amountLeft = _amountToLock;
for (uint256 i = 0; i < tranches.length; i++) {
    if (amountLeft == 0) break;
    // get balance of asset in tranche
    uint256 available = totals[_asset][tranches[i]].total - totals[_asset][tranches[
        i]].locked;
    if (available == 0) continue;
    totalAvailable += available;
    ...
}
```

The `totalAvailable` amount therefore does not include the available tokens in the next tranches which then affects the reward distribution as depositors in the other tranches are still eligible to receive rewards for this `lockId`.

```
// Deposit should be eligible for the given lock record
if (
    depositRecord.blockNumber >= lockRecord.blockNumber
        || lockRecord.blockNumber > depositRecord.unlockTime - manager.fraudPeriod()
        || lockRecord.asset != depositRecord.asset
) continue;
```

Consider the following scenario:

There are currently 3 active tranches:

Stake amounts	0	100	1000
Locked amounts	0	0	0

A lock should be created for 100 tokens. We iterate through the first 2 tranches and see that we have the needed tokens available.

Stake amounts	0	100	1000
Locked amounts	0	100	0

Notice how the 1000 tokens in the third tranche were not used and the `totalAvailable` is equal to just the 100 from the second tranche.

However, in the `claim` function the depositors of the uncounted 1000 tokens are still eligible for a reward for this `lockId`:

```
// Deposit should be eligible for the given lock record
if (
    depositRecord.blockNumber >= lockRecord.blockNumber
        || lockRecord.blockNumber > depositRecord.unlockTime - manager.fraudPeriod()
        || lockRecord.asset != depositRecord.asset
) continue; // @audit If these conditions are not true, the deposit is eligible
```

Therefore, let's say that the reward for this lock was 100 USDT tokens, the depositors of the 100 tokens in the 2nd tranche would receive a total of 100 USDT, but the depositors of the tokens in the 3rd tranche would receive 100 * 1000 / 100 = 1000 USDT.

Recommendation: Do not break the for loop before `totalAvailable` is updated.

Resolution: Resolved. The `totalAvailable` amount is now correctly calculated.

5.1.7 Rewards distribution design is flawed

Severity: *Critical Risk*

Context: Staking.sol#L251-L268

Description: The current implementation of the Staking contract is fundamentally flawed as the rewards distribution mechanics do not work as intended.

Consider the scenario where 30 locks are created for the same 3 active tranches each tranche having 1000 staked and available initially.

Stake amounts	1000	1000	1000
Locked amounts	0	0	0

The `totalAvailable` tokens for the first lock would be 3000 and all depositors in the 3 tranches should be eligible to receive rewards from providing liquidity to these tranches.

The `totalAvailable` tokens for the second lock would be 2900. Now, 100 tokens are already locked, so only the other 2900 are considered available and should therefore receive rewards. However, the `claim` function would allow all depositors of these 3000 tokens to receive the reward for the seconds lock even though the divisor is 2900.

After 29 locks of 100 tokens, only 100 tokens should be left available. However, the `claim` function would allow the depositor of all 3000 tokens to claim rewards for this lock and therefore 30x bigger reward will be actually distributed causing direct loss of funds in the corresponding Portal.

Recommendation: The fix is non-trivial with the current implementation. Consider refactoring the whole functionality.

Resolution: Pending.

5.2 High severity

5.3 Medium severity

5.3.1 Off-chain orderbook's signature is not verified when processing signed updates

Severity: *Medium Risk*

Context: Rollup.sol, StateUpdateLibrary.sol#L63-L69

Description: The off-chain orderbook is reffered to as the "participating interface or PI". It detects on-chain contract events created by traders interacting with the Portal contract and off-chain orders submitted to the PI directly. It uses both to generate its own sequence of events that are broadcast to the validator.

When validators process any state updates signed by the PI, they are verified to belong to a specific state root via a merkle proof. However, it is never verified on-chain whether the signed update struct has been actually signed by the PI. The signed update struct is constructed of the following properties:

```
/// @notice A StateUpdate that was signed by the participating interface
struct SignedStateUpdate {
    StateUpdate stateUpdate;
    uint8 v;
    bytes32 r;
    bytes32 s;
}
```

The `v`, `r` and `s` value are never validated in `processSettlements`, `processRejectedDeposits` and `claimTradingFees`. Even though that they are included in the merkle proof verification, it's still recommended to validate that the signature belongs to the participating interface.

Recommendation: Consider validating whether the `v`, `r` and `s` values are valid for the corresponding `stateUpdate` and `participatingInterface`.

Resolution: Resolved. The PI signature is now checked on each signed update.

5.3.2 Validators can replace state roots for epochs that have not been proposed yet

Severity: *Medium Risk*

Context: Rollup.sol#L80-L101

Description: Validators should propose new state roots for the current epoch, wait for the fraud period to pass and then confirm them. If the state root is marked as fraudulent or the validator simply made a mistake, they can call `replaceStateRoot` to replace the `proposedStateRoot` for any previous epoch that has not been confirmed.

However, an input validation is missing to ensure that a validator cannot "replace" a state root for an epoch that is in the future. Due to the lack of this check, a validator can basically create or "propose" a new state root for a future `epoch` without locking any funds as collateral.

Recommendation: Add the following check in `replaceStateRoot`:

```
if (_epoch >= _epoch) revert("Cannot replace state root that has not been proposed")
    ;
```

Resolution: Resolved. The recommended check was implemented.

5.3.3 Staking rewards are split in incorrect ratio between the stablecoin and TXA pool

Severity: *Medium Risk*

Context: Rollup.sol#L198-L208, Rollup.sol#L219-L221, FeeManager.sol#L53-L60

Description: Stakers are incentivised to provide liquidity in USDT and TXA on the processing chain by depositing tokens into the Staking contract that will then be used as collateral when needed by the Rollup contract.

The incentive for stakers is the reward that they would earn from settlement fees when a settlement is processed before its corresponding state root has been confirmed. The fees are split in a 85:15 ratio (between the USDT and TXA depositors) by using the `calculateStakingRewards` function of the processing chain manager.

The reason this ratio is chosen is because the amount of tokens that should be locked when processing a settlement is 100% of the balance value in USDT and 15% more in TXA.

The problem here is that the ratio of the locked tokens is actually 100:15 while the rewards are split in 85:15 which means USDT stakers would receive bigger portion of the rewards than what's intended.

Recommendation: Consider using the same ratio either by changing the `stablePoolPortion` variable or requiring more than 15% of the USDT value of trader's balance to be locked in TXA tokens.

Resolution: Pending.

5.3.4 Settlements would not work if there are no staked tokens to use as collateral

Severity: *Medium Risk*

Context: Rollup.sol#L84, ProcessingChainManager.sol#L30-L31

Description: Every time a validator wants to propose a new state root, they have to lock 10000 TXA tokens (~$3400 at current market price or $40,000 in 2021) from the Staking contract until the state root is confirmed. Stakers are incentivised to provide liquidity in both USDT and TXA tokens on the processing chain by accrueing fees from trades processed before a state root has been confirmed.

A potential problem would be if the Staking contract does not have enough free balance of TXA tokens to execute the `proposeStateRoot` function. This could happen if stakers are not incentivised enough to provide liquidity especially for TXA tokens as they do not benefit anything from the locks that are created when proposing a new state root.

Therefore, if this happens, the on-chain settlement mechanism will be blocked as no new root would be able to get proposed.

Recommendation: Consider whether the incentive structure is good enough for stakers to keep providing TXA liquidity. Consider reducing the `rootProposalLockAmount` that has to be locked each time a state root is proposed.

Resolution: Resolved. A method has been implemented to allow the ProcessingChainManager contract's admin to update the `rootProposalLockAmount`.

5.3.5 Reporter can update the price of an asset before initializing it

Severity: *Medium Risk*

Context: Oracle.sol#L72-L110

Description: Oracle price reporter must first initialize a supported asset using the `initializePrice` function and then regularly update the price of this asset using the `report` function.

The problem is that the reporter can directly `report` a price of an asset before it has been initialized:

```
function report(uint256 _chainId, address _asset, uint256 _price, bool _modulo)
    external {
    ...
    if (block.number < lastReport[_chainId][_asset] + PRICE_COOLDOWN) {
        revert("Price cooldown period has not passed");
    }
    // @audit If the price has not been initialized ‘lastReport‘ will be 0 and the
        check will pass
    ...
}
```

That way the `lastReport` can be set to `block.number` before the price has actually been initialized. The impact is that now instead of reverting when `getStablecoinValue` and `stablecoinToProtocol` are called, those function will return 0 which could result in unexpected behaviour such as locking 0 collateral when processing unconfirmed state root settlements.

Recommendation: Add the following check in `report`:

```
if (lastReport[_chainId][_asset] == 0) revert("Asset price not initialized");
```

Resolution: Resolved. The recommended check was implemented.

5.3.6 Validators can propose the same state root more than once

Severity: *Medium Risk*

Context: Rollup.sol#L87

Description: When proposing a `_stateRoot`, there is no check to see if the `_stateRoot` has already been proposed for a previous `epoch`. Considering there could be numerous validators, it is realistic to believe 2 or more may accidentally call `proposeStateRoot` with the same `_stateRoot`. By doing so they will not only lock more tokens than needed, but also the second call will change the value of `proposalBlock[_stateRoot]` and will “restart” the timer.

```
    function proposeStateRoot(bytes32 _stateRoot) external {
        if (!manager.isValidator(msg.sender)) revert CALLER_NOT_VALIDATOR();
        if (_stateRoot == "") revert("Proposed empty state root");
        IStaking staking = IStaking(manager.staking());
        uint256 lockId = staking.lock(staking.protocolToken(), manager.
            rootProposalLockAmount());

        proposedStateRoot[epoch] = _stateRoot;
        proposalBlock[_stateRoot] = block.number;
        lockIdStateRoot[lockId] = StateRootRecord(_stateRoot, epoch);
        epoch = epoch.increment();
    }
```

Recommendation: Revert if `proposalBlock[_stateRoot]` is not `0`.

Resolution: Resolved. The validators should now pass the value of the previous proposed state root as an argument to the `proposeStateRoot`. This effectively mitigates the above issue.

5.3.7 Stakers can bypass the minimum lockup time check

Severity: *Medium Risk*

Context: Staking.sol#L101-L102

Description: A comment above the `PERIOD_LENGTH` constant states that this is the "Minimum number of blocks for which funds must be locked". However, users are allowed to stake for as low as `fraudPeriod()` blocks because of the following check on line 102:

```
if (block.number >= _unlockTime - manager.fraudPeriod()) revert("Can no longer stake
    into this tranche");
```

Recommendation: Consider changing the check on line 102 to the following:

```
if (block.number >= _unlockTime - PERIOD_LENGTH) revert("Can no longer stake into
   this tranche");
```

Resolution: Acknowledged. The mentioned comment has been corrected to "Stakers can enter a tranche from its beginning up to the end (minus the fraud period). No matter what time a staker enters, the funds will be locked until the end of the period.".

5.3.8 Missing input validation for minimum stake amount

Severity: *Medium Risk*

Context: Staking.sol#L94-L95

Description: The following two variables are defined in the Staking contract but are never actually used:

```
uint256 public constant minimumStablecoinStake = 200e6;
uint256 public constant minimumProtocolStake = 200e18;
```

Recommendation: Add the following check in Staking.stake:

```
uint256 minStake = _asset == stablecoin ? minimumStablecoinStake :
   minimumProtocolStake;
if (_amount < minStake) revert("Amount too low");
```

Also, consider whether hardcoding these values won't cause the `minimumProtocolStake` to become too high or too low at some point in time. The TXA protocol token has changed from ~4$ in 2021 to ~0.4$ in 2022 which means that the `minimumProtocolStake` could have ranged between 800$ and 80$ if the contracts were used at that time period.

Resolution: Resolved. The missing validation was added.

5.3.9 Less collateral could be locked than needed

Severity: *Medium Risk*

Context: Oracle.sol

Description: The system makes use of a custom price oracle to report assets price denominated in the stablecoin chosen by the team on the processing chain.

The way the Oracle contract works is by allowing reporters to first initialize an asset and then update its price on 15 minutes to 6 hours intervals. The price change can be at most 15% at each 15 minutes.

This means that if a price has not been updated for too long due to inactive reporter, or the actual price of the asset changed with more than 15% in those 15 minutes, the price will not be completely correct.

The oracle is currently used in the Rollup contract when processing a settlement that needs collateral. Therefore, if the reported price is lower or higher than the actual one, wrong amount of collateral will be locked.

Recommendation: Since the fraud engine is not developed yet, the amount of collateral locked is not really used as compensation in the beta version of the app. Consider whether this could become a problem in the future and implement an oracle that updates the price regularly based on a deviation threshold as Chainlink does. Alternatively, see if another oracle can be used as a solution for the project's needs.

Resolution: Acknowledged.

5.3.10 Admin won't be able to revoke a validator's privileges

Severity: *Medium Risk*

Context: ProcessingChainManager.sol#L114

Description: The admin of the ProcessingChainManager contract should be able to give and remove `validator` rights to chosen accounts. However, the `revokeValidator` incorrectly sets the `validators[_validator]` mapping value to **`true`** instead of **`false`**.

Recommendation: Set `validators[_validator]` to **`true`** instead of **`false`** in `revokeValidator`. Consider executing further tests as these types of issues should be caught pre-audit.

Resolution: Resolved. The recommendation was implemented.

5.3.11 Rejecting a deposit doesn't update the collateralized mapping value

Severity: *Medium Risk*

Context: Portal.sol#L180-L194

Description: When a deposit is rejected, funds that a trader has deposited to the Portal contract are released by the `rejectDeposits` function so they can then withdraw them using the `withdrawRejected` function.

These funds have been accounted for in the `collateralized` mapping when deposited to track the balance of the contract of the given `_token`:

```
unchecked {
    collateralized[_token] += _amount;
}
```

The problem is that this value is not decreased by the corresponding amount in the `rejectDeposits` function nor in `withdrawRejected`.

Recommendation: Consider reducing `collateralized[deposit.asset]` by `deposit.amount` in `rejectDeposits`.

Resolution: Resolved. The recommended fix was implemented.

5.3.12 User can accidentally stake their tokens too long into the future

Severity: *Medium Risk*

Context: Staking.sol#L102

Description: When a user stakes in `Staking`, they have to choose and manually input the `_unlockTime` .

The problem is that as long as it passes the following two requirements, it is considered a valid lockup period:

```
if (_unlockTime % PERIOD_LENGTH != 0) revert("Invalid unlock time");
if (block.number >= _unlockTime - manager.fraudPeriod()) revert("Can no longer stake
    into this tranche");
```

A user can accidentally set their `_unlockTime` too far into the future (e.g. pass it in miliseconds instead of seconds) and forever lock their tokens.

Recommendation: Set a max value limit for `_unlockTime` (e.g. block.number + 5 * PERIOD_LENGTH).

Resolution: Resolved. The recommended fix was implemented by adding and checking against the following constant:

```
uint256 public constant MAX_PERIOD = PERIOD_LENGTH * 5;
```

5.3.13 Non-standard ERC20 tokens such as USDT are not supported

Severity: *Medium Risk*

Context: AssetChainManager.sol, Portal.sol

Description: A well-known issue regarding non-standard ERC20 tokens such as USDT that don't implement the EIP20 interface correctly because of missing return boolean variables on methods like `transfer`, `transferFrom` and `approve`.

Project TXA is expected to be deployed on Ethereum and also to support USDT. However, since USDT does not return a boolean when `transferFrom` and `transfer` are called, but the ERC20 interface defines that there will be a boolean variable returned, the compiler will check whether the `returndatasize()` is exactly 32 bytes (one word size) and revert automatically if this is not the case.

Recommendation: Consider using the SafeERC20 library from OpenZeppelin when interacting with arbitrary ERC20 tokens.

Resolution: Resolved. The recommended fix was implemented.

5.4 Low severity

5.4.1 Incorrect token balance update when a staker withdraws their deposit

Severity: *Low Risk*

Context: Staking.sol#L142-L143

Description: When a staker withdraws their deposit of TXA tokens from the Staking contract, the `totalStaked` and `individualStaked` mappings are updated. The problem is that instead of using the `protocolTokenAmount` withdrawn, the contract uses the `stablecoinAmount` which means that mapping is incorrectly updated.

The worse impact of this issue is that the `withdraw` function may revert in some cases when `totalStaked[protocolToken]` is less than the `stablecoinAmount`. However, this can be easily mitigated by simply withdrawing deposits of TXA tokens separately from stablecoin deposits.

Recommendation: Use the correct amount of tokens when updating the `totalStaked` and `individualStaked` mappings.

Resolution: Resolved.

5.4.2 Missing check for empty state root when replacing a proposed one

Severity: *Low Risk*

Context: Rollup.sol#L82, Rollup.sol#L94

Description: The `proposeStateRoot` function in the Rollup contract checks whether the validator has mistakenly or on purpose tried to propose an empty state root for a given epoch:

```
if (_stateRoot == "") revert("Proposed empty state root");
```

However, this check is missing in the `replaceStateRoot` function which means a validator can still propose an empty state root.

Recommendation: Consider checking for empty state root in the `replaceStateRoot` function.

Resolution: Resolved.

5.4.3 Validators should not be able to replace a state root with a fraudulent one

Severity: *Low Risk*

Context: Rollup.sol#L94-L101

Description: Any proposed state root for a given epoch can be marked as fraudulent by the `fraudEngine` within the first 4 days of its proposal. A validator can replace such a state root with a new one using the `replaceStateRoot` function. Validators can also replace state roots if they have made a mistake with the one already proposed.

However, a validator should not be able to replace a proposed state root with a fraudulent one for the given epoch.

Recommendation: Consider checking if the new `_stateRoot` in `replaceStateRoot` is marked as fraudulent for the passed `_epoch`.

Resolution: Resolved.

5.4.4 Validators should not be able to process settlements for a fraudulent state root

Severity: *Low Risk*

Context: Rollup.sol#L143

Description: A validator is currently able to process a settlement for a state root that has not been confirmed yet, but only proposed. This happens by locking a collateral from the Staking contract.

However, if the state root has already been marked as fraudulent, the validator should not be able to process any corresponding settlements.

Recommendation: Consider checking if the proposed state root used in `processSettlements` is a fraudulent one.

Resolution: Resolved.

5.4.5 Insurance fund fees are lost

Severity: *Low Risk*

Context: Rollup.sol#L235-L240, Rollup.sol#L213-L218, Staking.sol#L219-L223

Description: A user is charged a 0.1% settlement fee every time a settlement request is processed. This fee is split as follows:

- 50% goes to stakers in the Staking contract for providing collateral used when processing settlements for unconfirmed state roots.
- 50% goes to an insurance fund. If the state root of the processed settlement is confirmed, the entire 100% of the settlement fee is directed to the insurance fund.

The insurance fees are subtracted from the amount of tokens the trader will receive as an obligation in the corresponding Portal contract:

```
// Calculate settlement fee
(uint256 insuranceFee, uint256 stakerReward) =
    IFeeManager(address(manager)).calculateSettlementFees(state.settlement.
        balanceBefore.amount);
// create an obligation to be relayed
obligations[i] = IPortal.Obligation(
    state.settlement.balanceBefore.trader,
    state.settlement.balanceBefore.asset,
    state.settlement.balanceBefore.amount - (insuranceFee + stakerReward)
);
```

However, there is no logic implemented at the time of this audit to handle the insurance fund fees:

```
// Called by rollup contract to delegate a portion of settlement fee to the
    insurance fund
function payInsurance(uint256 _chainId, address _asset, uint256 _amount) external {
```

```
    if (msg.sender != manager.rollup()) revert();
    insuranceFees[_chainId][_asset] += _amount;
} // @audit The ‘insuranceFees‘ mapping is never used after that.
```

Therefore, the fees taken from the trader balance become lost.

Recommendation: Consider either removing the insurance fee or implementing the necessary logic before deploying the contract in production.

Resolution: Resolved.

5.4.6 Check-Effects-Interactions pattern is not always followed

Severity: *Low Risk*

Context: Rollup.sol#L243-L246, Staking.sol#L135-L144

Description: There are several places in the code where the CEI pattern is not strictly followed.

Recommendation: Consider consistently executing external calls after state changes have been made to limit the risk of potential attack vectors.

Resolution: Resolved.

5.4.7 Fee-on-transfer token transfers are not handled correctly

Severity: *Low Risk*

Context: src

Description: The protocol currently does not handle fee-on-transfer tokens which could lead to unexpected behavior like withdrawing more tokens than actually deposited in a Portal contract if such tokens are used.

Recommendation: Consider either never adding a fee-on-transfer token as supported to the protocol or adjusting all ERC20 transfers and balance accounting so that they handle this specific logic.

Resolution: Acknowledged.

5.4.8 Project may fail to be deployed to chains not compatible with Shanghai hardfork

Severity: *Low Risk*

Context: src

Description: All of the contracts in scope specify a Solidity compiler version of `^0.8.19`. The versions from `0.8.20` and above of the compiler use the new PUSH0 opcode introduced in the Shanghai hard fork.

This means that the produced bytecode for the different contracts may not be compatible with the chains that Project TXA aims to support that don’t yet support the Shanghai hard fork.

Recommendation: Consider locking the Solidity compiler version to `0.8.19`.

Resolution: Resolved.

5.4.9 isConfirmedLockId will return true for invalid lock IDs

Severity: *Low Risk*

Context: L390-L393

Description: The `isConfirmedLockId` function of the Rollup contract returns whether the state root corresponding to a lockId has been confirmed or not.

```
function isConfirmedLockId(uint256 _lockId) external view returns (bool) {
    StateRootRecord memory r = lockIdStateRoot[_lockId];
    return confirmedStateRoot[r.epoch] == r.stateRoot;
}
```

The problem is that if `_lockId` is invalid (higher than or equal to `Staking.currentLockId()`) the default mapping values will be read from the Rollup contract's storage which means that `true` will be returned as two empty state roots will be compared.

At the same time, `isFraudulentLockId` will also return a default storage value of `false` which means that if those two function are used for input validation, they can be easily passed with an invalid `lockId`. However, no serious vulnerability is currently present in the contracts in scope as the Staking contract performs additional validation.

Recommendation: Consider revering or returning an appropriate variable if `r.stateRoot` is an empty root.

Resolution: Resolved.

5.4.10 Default minimum deposit amount is returned even for unsupported assets

Severity: *Low Risk*

Context: AssetChainManager.sol#L73

Description: The `getMinimumDeposit` function of the AssetChainManager contract does not check whether the passed `_asset` address is supported or not. Therefore, the `defaultMinimumDeposit` will be returned, while a revert might be more appropriate.

Recommendation: Consider reverting in `getMinimumDeposit` if the passed asset is not supported.

Resolution: Resolved.

5.4.11 Fee update proposal time is not set back to default value

Severity: *Low Risk*

Context: FeeManager.sol#L80-L92

Description: The default value of the `proposalTime` variable in the FeeManager contract is `type(uint256).max`. This is to prevent from calling the `updateFees` function if no fees have been proposed after the last update.

However, when `updateFees` fees are called after the first proposal, the `proposalTime` variable is not reset to `type(uint256).max` which means that the Participating Interface can call it again with the same proposed fees which will unnecessary update the `feeHistory`.

Recommendation: Consider setting the `proposalTime` variable back to `type(uint256).max` in `_updateFees`.

Resolution: Resolved.