# Trust Token
# Smart Contracts Audit

## INTRO

The Trust Token team is working on the first regulated stablecoin that is fully backed by the US Dollar. They asked me to audit a new feature that will allow periodic unlocking of the TRU tokens that are distributed to investors.

The code which is audited is located in the open-source github repository trusttoken/true-currencies and is consisted of two smart-contracts: TimeLockedToken.sol and TimeLockRegistry.sol

The version of the code that is being reviewed was published in the PR 254 with the commit `d3e39aaf50b9481d81c950f339cc0bbbac6e8c06`.

All of the issues are classified using the popular OWASP risk rating model. It estimates the severity taking into account both the likelihood of occurrence and the impact of consequences.


# SUMMARY

The development team demonstrated solid engineering skills adding a new feature to a complex project. The design is clean and the responsibilities are well divided between the two smart contracts. The code is readable and well-documented.

There have been **no critical** and **high** severity issues found. There are medium and low severity issues that in the edge cases could have non-negligible consequences. The issues consist mostly of the possibility of **front-running, reentrancy attack, lack of compatibility with ethereum standards, unchecked calculations and missing input values sanitization**. All of the vulnerabilities could be easily fixed with a minimum code change.

# ISSUES FOUND

## Registry cancellation could be avoided

Severity: **MEDIUM** (Impact: HIGH, Likelihood: MEDIUM)

### PROBLEM

When the `cancel` method call is broadcasted to the network the beneficiary may try to front-run the transaction by calling the `claim` method with a higher gas value and having it processed before the cancellation is executed.

### CONSEQUENCES

A sophisticated distribution beneficiary could cause any cancellation to fail by either front running the cancel method or simply claiming the tokens as soon as the `register` call is processed. This could deprive contract owners the ability to cancel distribution and make updating allocations impossible,

### RECOMMENDATIONS

I suggest rethinking the cancellation logic as in the current form it could be easily escaped by a skilful user. One way to implement that is to introduce a delay period between a time when tokens could be registered and the time when it's possible to claim them. The cancellation may be effective during such a period allowing to correct any mistakes made during the registration.

## Incorrect transfer signalling with duplicated events

Severity: **MEDIUM** (Impact: LOW, Likelihood: HIGH)

### PROBLEM

The `registerLockup` method emits a `Transfer` event as the last action in the function body. However, in the line before, it invokes the `_transferAllArgs` function, which also emits the same event.

### CONSEQUENCES

`Transfer` event is a key part of the `ERC-20` token standard and it's widely adopted to show the flow of funds by many of popular blockchain explorers like Etherscan. Duplicating the event may lead to problems with monitoring token flow by users who will be presented with corrupted data.

### RECOMMENDATIONS

I suggest removing the `line 109` which emits the duplicated event and use the mechanism implemented in the base contract.

## Error in epoch calculations before locking period starts

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

### PROBLEM

The `epochsPassed` function should return the number of epochs that passed since the beginning of the unlocking. However, when it is called before the unlocking has started it will throw an exception breaking the execution of any dependent methods.

### CONSEQUENCES

Calculation of passed epochs is a crucial part of checking balance and verifying transfers. Therefore, if the method breaks, it is impossible to make any transfer and funds are frozen. It might be impossible to distribute the funds if locking is set to begin after the tokens should be successfully registered as SAFTs.

### RECOMMENDATIONS

I suggest fixing the logic of the `epochsPassed` function to take into account the scenario when `block.timestamp<LOCK_START` and return `0` in such a case.

## Reentrant vulnerability during distribution cancelling

Severity: **MEDIUM** (Impact: MEDIUM, Likelihood: LOW)

### PROBLEM

The `cancel` function from the `TimeLockRegistry` contract transfers tokens before updating the state by removing an entry in the `registeredDistributions` mapping. It is possible to invoke the function again from the contract that is a target for token transfer and pull more tokens before updating the state.

### CONSEQUENCES

All of the funds in the registry contract could be removed in a single transaction that will exploit the reentrancy vulnerability. Moreover, the action will not be noticed by the investors as they will still have the tokens registered on their accounts and no events will be published.

The function is protected by the `onlyOwner` modifier, therefore the risk of a trusted party executing such a scenario should be classified as low but it shouldn't be neglected.

### RECOMMENDATIONS

I strongly recommend following the Check Effects Interaction pattern by updating the internal state before calling external contracts. It could be implemented by moving the line 64 above the line 59.

## TimeLockRegistry is not validated

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

### PROBLEM

The `setTimeLockRegistry` function from the `TimeLockedToken` contract does not perform any validation of the `newTimeLockRegistry` parameter. It is possible to set the value to a zero address or invoke the function with a new address that is equal to the old value.

It is also possible to set the `newTimeLockRegistry` in an uninitialized state or with a token that is different from the `TimeLockedToken` which invokes the function.

### CONSEQUENCES

It's a good practice to perform a thorough validation of all of the parameters. Setting the argument to a zero or old address is a signal of an error in the transaction and should be caught and communicated to the user. Moreover, setting a registry that is linked to a wrong token may cause break the lockup registration logic.

### RECOMMENDATIONS

I recommend validating the arguments to check if the address is not 0x0 or equal to the current value of the `timeLockRegistry`. I also recommend emitting an event when the value of `TimeLockRegistry` is set to notify contract users about an important update.

## Reentrant vulnerability during distribution claiming

Severity: **LOW** (Impact: LOW, Likelihood: LOW)

### PROBLEM

The `claim` function from the `TimeLockRegistry` contract moves tokens from the registry to a locked user account before updating the internal state. Therefore, it is possible to invoke the function again, in the same transaction, to move more tokens before the original balance is cleared.

### CONSEQUENCES

A malicious user may participate in the distribution and provide a dedicated smart contract as his address. Such a contract could exploit the reentrancy vulnerability by calling the `claim` function again after tokens are moved to his account and the notification hook is invoked. This will allow the contract to remove registered tokens a few times before the internal balance is updated.

Fortunately, the exploit is currently prevented by an additional check in the `registerLockup` function that prevents multiple distributions for the same account. However, there is still a structural problem in the `claim` that may be manifested if the registry is connected to another contract or the `TimeLockedToken` contract is updated.

### RECOMMENDATIONS

I strongly recommend following the Check Effects Interaction pattern by updating the internal state before calling external contracts. It could be implemented by moving the line 78 above the line 73.

## Unsafe math calculations

Severity: **LOW** (Impact: MEDIUM, Likelihood: LOW)

### PROBLEM

When the `finalEpoch` method uses basic arithmetic operators to perform calculations. It may bring a risk of an overflow if parameters are updated.

### CONSEQUENCES

Last moment parameters update may cause hard to catch errors when the code is not protected against calculation overflows.

### RECOMMENDATIONS

I suggest always using safe arithmetic operators to perform calculation and changing the code to use the `add` and `mul` methods.

## Incorrect value of nextEpoch when lock period is finalised

Severity: **LOW** (Impact: MEDIUM, Likelihood: LOW)

### PROBLEM

The `nextEpoch` method should return the time when the next round of token unlocking begins. However, if it's called after the final 8th round has finished it returns an incorrect value.

### CONSEQUENCES

Returning a start time of an unexisting method could mislead users that there is another round of token distribution that is going to happen in the future.

### RECOMMENDATIONS

I suggest throwing an error if there is no next epoch or return a special, well-documented, value to indicate that the unlocking process is over.

## Token burning may break the distribution logic

Severity: **LOW** (Impact: MEDIUM, Likelihood: LOW)

### PROBLEM

The `unlockedBalance` function relies on the assumption that the current balance must always be greater than the locked balance. However, if any of the token held by the user is burned, this assumption may no longer be true and cause an underflow error in calculation.

### CONSEQUENCES

The `TimeLockToken` contains the code for token minting and burning that could be utilised by any derived contract. However, if a contract uses the `burn` method it may accidentally break `unlockedBalance` logic which will result in breaking any transfer method and freezing the funds.

### RECOMMENDATIONS

I suggest fixing the code to check if the current balance is greater or equal to the locked balance before calculating the unlocked tokens. This will allow any inheriting contract to support the burning logic.

# Notes

Issues with a very low impact

### REDUNDANT CHECKS OF BASIC ERC-20 LOGIC

The `register` function of the `TimeLockRegistry` checks if the message sender has got the necessary allowance before making an attempt to transfer funds. Allowance checking is a well-documented and standard feature of ERC-20 token standard and should be implemented on the token, not the client-side.

### MISSING FUNCTION DOCUMENTATION

The `initialize` function from the `TimeLockRegistry` contract lacks documentation. Please try to maintain a consistent practice of documenting every function and its parameters, especially as the function is external.

### INCONSISTENT COMMENTS FOR THE SAME CODE

Removing an entry from the `registeredDistributions` mapping is done twice in the `TimeLockRegistry` contract. However, every time it is commented with a different description despite having exactly the same effect. Please make sure to comment on equal behaviour with an equal description not to mislead contract readers.

### TYPO IN COMMENTS

There is a typo in the function comments. When it says: “set distribution mappig to 0” it should say: “set distribution mapping to 0”.

# APPENDIX

The team asked me to review two additional commits:

1) Refactoring TrustToken to be standard ERC20 implementation

   (b4c6847e11092c438c8a4fbc508a600d9a48fb32)

2) Rejecting transfer to autosweep addresses

   (7118a4c5aaca905a4f62256e189802dda43eaab7)

Below there are my comments regarding proposed changes validating correctness and suggesting potential improvements:

### PUBLIC BURN FUNCTION LOOKS GOOD

The team has fixed the issue adding an extra check: `require(unlockedBalance(_from) >= _value, "attempting to burn locked funds");` thus restricting the burn function only to unlocked tokens. In the current shape, the public burn function is compatible with the ERC20 specification and should work correctly.

### REDUNDANT BALANCE CHECK IN BURN LOGIC

The validation in the burn function:

```
require(balanceOf[_from] >= _value, "insufficient balance");
```

duplicates the logic from the base `ERC20.sol` contract:

```
_balances[account] = _balances[account].sub(amount, "ERC20: burn amount exceeds balance");
```

I recommend removing the redundant validation as it doesn't increase the security while adding code and making it less readable and harder to maintain.

**REDUNDANT BALANCE CHECK IN TRANSFER LOGIC**

The validation in the _transfer function:

```
require(balanceOf[_from] >= _value, "insufficient balance");
```

duplicates the logic from the base `ERC20.sol` contract:

```
_balances[sender] = _balances[sender].sub(amount, "ERC20:
transfer amount exceeds balance");
```

Removing an extra check will make the code easier to read and maintain. Moreover, the duplicated check may break the behaviour of the base contract, where the `_beforeTokenTransfer` function is invoked before checking the balance. If the token implemented any logic of accumulating rewards or bonuses before the transfer, the additional balance wouldn't be included preventing a legitimate transfer from happening.

**STAKING DEPOSIT COULD BE DOUBLE-REGISTERED**

The commit added a manual invocation of the _deposit function:

```
_deposit(msg.sender, _amount);
```

However, this function is still being called from the `tokenFallback` method:

```
_deposit(_originalSender, _amount);
```

I recommend checking if the stakeAsset() doesn't implement the hook invoking token fallback function to prevent potential double-registration.

### STAKING DEPOSITOR IS NOT FULLY VALIDATED

In the updated code there was a check verifying if the depositor was not a liquidator:

```
if (_originalSender == liquidator()) {
   // do not credit the liquidator
   return;
}
```

I recommend double checking if the modified `deposit` function cannot be invoked by the `liquidator` to remain consistent with the previous restrictions.

### IMPOSSIBLE TO LOCATE THE FINAL AUTO-SWEEPING DEPLOYMENT

The implemented logic looks fine and I don't see any potential vulnerabilities. However, the code is implemented only at an `abstract` level and in the `mock` contract raising a question whether this is still a feature under development waiting to be fully rolled out at a later point in time.

*Jakub Wojciechowski*
*Smart contract auditor*