True Currencies
Smart Contracts Audit

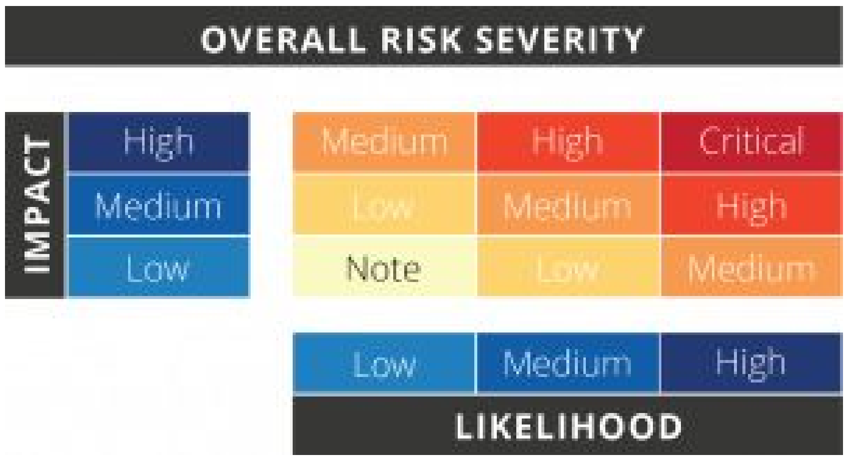
INTRO

The Trust Token team is working on the first regulated stablecoin that is fully backed by the US Dollar. They asked me to audit the new version of TrueCurrencies smart contracts that includes code refactoring, optimisations and a few improvements related to burning, redemptions and blacklisting.

The code which is audited is located in the open-source github repository trusttoken/true-currencies and the scope is limited to contracts located in the true-currencies-new folder.

The version of the code that is being reviewed was published in the PR 274 with the commit `3cf5a98b16df9f37492dd2a1b17c2316296e8335`.

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.



OVERALL RISK SEVERITY

IMPACT \ LIKELIHOOD	Low	Medium	High
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Note	Low	Medium

SUMMARY

The development team demonstrated good engineering skills refactoring the code to be compatible with the standard, open-source ERC-20 token implementation. 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 3 medium and 6 low severity issues that in the edge cases could have severe side-effects. . The issues consist mostly of the possibility of **redemption addresses edge cases, blacklisting procedure, unchecked calculations and missing input values sanitization**. All of the vulnerabilities could be easily fixed with a minimum code change.

ISSUES FOUND

Zero address is incorrectly reported as a redemption address

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

PROBLEM

The `isRedemptionAddress` function wrongly categorises zero address as a correct redemption address. Moreover, both the documentation in the `TrueCurrency` contract and the variable name `REDEMPTION_ADDRESS_COUNT` specify that there are 100,000 valid redemption addresses. It is not true as zero address is a reserved value used to report burning activity in the ERC20 standard and is forbidden to be used in transfer activity.

CONSEQUENCES

Zero address, as a first element from the redemption addresses set, is a natural candidate to be used to redeem tokens. However, when a user wants to redeem tokens by a transfer to a zero address the transaction will fail, raising concerns that the redemption logic is broken.

RECOMMENDATIONS

I recommend updating both the `isRedemptionAddress` and the documentation to specify that zero address is not allowed to be used for the redemption process.

RESOLUTION

The team fixed the issue by extending the redemption address check to accommodate for the special role of the zero address: : `return uint256(account) < REDEMPTION_ADDRESS_COUNT && uint256(account) != 0;` in line 169 of the `TrueCurrency` contract.

Blacklisting redemption addresses should not be allowed

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

PROBLEM

The `setBlacklisted` function from the `TrueCurrency` contract does not validate an account before updating the blacklisted status. It is possible to blacklist the special purpose zero address or one of the redemption addresses.

CONSEQUENCES

Blacklisting any of the addresses that play a special role in the system could break the core feature and should be automatically prevented by arguments validation. It's a good practice to always sanitize input parameters to avoid human errors in passing wrong arguments.

RECOMMENDATIONS

I suggest adding a check in the `line 83`:
`require(account >= REDEMPTION_ADDRESS_COUNT)` to prevent accidentally blocking redemption logic.

RESOLUTION

The team fixed the issue by adding an extra check:
`require(uint256(account) >= REDEMPTION_ADDRESS_COUNT, "TrueCurrency: blacklisting of redemption address is not allowed");` in line 88 of the `TrueCurrency` contract.

Users cannot withdraw approvals for blacklisted accounts

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

PROBLEM

The `_approve` function from the `TrueCurrency` contract disables any change of the user allowance. Therefore a user cannot cancel the allowance after learning that the funds could be potentially used by a blacklisted account.

CONSEQUENCES

Being blacklisted is a strong indicator of illegal activity and users and services should be able to break any links with the suspected account by removing their allowance. Users may also be concerned about being front-runned when the blockade is released and they fail to cancel their approval before the pre-approved funds are collected.

RECOMMENDATIONS

I suggest allowing users to remove the approval for blacklisted accounts by having a special case for `amount == 0`.

RESOLUTION

The team fixed the issue by allowing to set 0 allowance for a blacklisted spender: `require(!isBlacklisted[spender] || amount == 0, "TrueCurrency: tokens spender is blacklisted");` in line 142 of the `TrueCurrency` contract.

Unvalidated ownership change

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

PROBLEM

The `transferOwnership` function from the `ClamableOwnable` contract does not validate input parameters. Therefore, it's possible to set up `pendingOwner` to the same address as the existing one or to point it to a null address.

CONSEQUENCES

It is always important to sanitize all of the input arguments, especially these controlling the most important admin access. Early catching an error in ownership update could help to discover the problems with an updating script and prevent setting obsolete values.

RECOMMENDATIONS

I suggest validating the input arguments by adding a check: `require(newOwner != owner && newOwner != address(0))`

Blacklisting status is not accessible

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

PROBLEM

The `setBlacklisted` function from the `TrueCurrency` contract updates the blacklisted status of the account but is not properly signalled to the contract clients. As the low-level `isBlacklisted` map is not publicly visible, it is impossible to query the blacklisted status.

CONSEQUENCES

Being blacklisted is an important indicator that there could be illegal activity detected on an account. All of the clients and protocols could use this flag to prevent interacting with blocked accounts instead of learning the status only after a failed transaction.

RECOMMENDATIONS

I suggest emitting an event `Blacklisted(indexed address account, bool isBlacklisted)` to properly communicate the status update.

RESOLUTION

The team fixed the issue adding the `Blacklisted(indexed address account, bool isBlacklisted)` which is emitted in line 90 of the `TrueCurrency` contract.

Potentially unsafe math calculations

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

PROBLEM

The `_transfer` method from the `TrueCurrency` contract performs unchecked math operations in line 117: `amount - (amount % CENT).`

CONSEQUENCES

In this particular situation there is no risk of the under or overflow. However, when certain formulas or parameters are updated it may introduce a potential vulnerability. I strongly recommend always using checked math operations to eliminate any risk of incorrect calculations.

RECOMMENDATIONS

I suggest always using safe arithmetic operators to perform calculation and changing the code to use the `sub` and `mod` method.

RESOLUTION

The team fixed the issue using checked operations and refactoring the calculations to: `amount.sub(amount.mod(CENT))`.

Frozen redemption remainders

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

PROBLEM

The `_transfer` function of the `TrueCurrency` contract burns funds with a one-cent precision. However, the rest of the transferred tokens are forever locked in the redemption address and could not be reclaimed.

CONSEQUENCES

Although the amount that could potentially be frozen could seem insignificant it could easily pile up when the protocol gets more traction. Moreover, frozen funds could be problematic in case of protocol updates and migrations when all of the funds should be relocated.

RECOMMENDATIONS

I suggest having a clear policy of handling redemption remainders. Even if the team decides to leave it frozen, it should be clearly documented and communicated to the users.

RESOLUTION

The team fixed the issue by reducing the amount that is being transferred to the redemption address, so it can always be fully burned:

```
super._transfer(sender, recipient, amount.sub(amount.mod(CENT));
```

Potential reentrancy vulnerability after adding transfer hook

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

PROBLEM

The `transferFrom` transfers the tokens before updating the approved amount. Therefore it is possible to re-enter the function with a stale approved amount from a contract that received transferred funds.

CONSEQUENCES

Although there is currently no vulnerability in the code that is being audited any update of the transfer logic that will include notifying the receiver could open a possibility for an attack that will drain more funds that were approved.

RECOMMENDATIONS

I recommend documenting the potential vulnerability, so any update will consider switching the order of transferFrom logic.

It’s possible to update burning bonds with old values

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

PROBLEM

The `setBurnBounds` function could be successfully invoked with arguments that are equal to the old values.

CONSEQUENCES

Validating if update arguments are truly changing the state is a valuable way of checking if there is no mistake in the update script and parameters are passed as intended.

RECOMMENDATIONS

It is a good practice to always sanitize the update arguments and validate them against the current ones checking `require(_min != burnMin || _max != burnMax)`.

Notes

Issues with a very low impact

CONSIDER EXTRACTING ROUNDING FUNCTION TO BASE CONTRACT

All of the `TrueCurrency` contracts implement their own, independent `rounding` function. It will make it impossible to refer to multiple contracts with a single interface and over-complicate the on and off-chain integrations. I recommend extracting the method interface to the `TrueCurrency` contract and override it in all of the implementations.

CONTRACT NAME CLAMABLEOWNABLE IS MISSPELLED

According to popular dictionaries(1, 2, 3) the claimable is a correct passive form rather than clamable, which is being used.

Fixed by the team by renaming all the "Clamable" occurrences.

APPENDIX

On 23rd September 2020 the True Currencies team asked me to review an upgrade of their contracts allowing legacy TrueUSD contract to be compatible with the new True Currencies codebase.. The code is located in the GitHub branch delegate token with the commit id: 197369840ec42e35b7a2dd374b2994150364e230.

There have been **no critical** and **high** severity issues found. There is one medium and one low severity issue identified. Both of the vulnerabilities could only be triggered by a careless admin behaviour, therefore could be mitigated by enforcing proper procedures and admin policies. However, I recommend programmatically preventing wrongful actions by adding adequate validations.

TrueUSD breaks while acting both as delegate and delegator

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

PROBLEM

The `TrueUSD` contract deployed at `0x8dd5fbCe2F6a956C3022bA3663759011Dd51e73E` extends both the `StandardDelegate` and the `CanDelegate` contracts. Therefore, it can act both as a delegate for an invoking contract and a delegator to another contract. However, if both features are used together it leads to breaking the contract logic.

CONSEQUENCES

When the `TrueUSD` is defined to act both as a delegator and a delegate some of the functions (`delegateTotalSupply`, `delegateBalanceOf`, `delegateAllowance`) will forward calls to another contract while the others will modify the internal state. This may lead to unpredicted behaviour including lost funds because the data will be read and written to different locations.

RECOMMENDATIONS

I recommend programmatically disabling the possibility of the contract to be a delegate and delegator at the same time by adding extra verification in the setters. However, as the values could be updated only from an owner account it is also possible to have a clear off-chain policy of updating the contract.

Hardcoded delegator could be overridden

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

PROBLEM

The `onlyDelegateFrom` modifier from the `DelegateERC20` is marked with a `virtual` keyword. This will allow any extending contract to modify the logic and set a different delegator address.

CONSEQUENCES

The delegator contract has very vast privileges and may freely move funds. The `DelegateERC20` hard-codes the delegator address to a constant address `DELEGATE_FROM = 0x8dd5fbCe2F6a956C3022bA3663759011Dd51e73E` to give users a strong guarantee that the value and cannot be updated to a different address. However, the `virtual` keyword leaves a path to modify the value, hence reduce the trust and confidence that the contract will always work correctly.

RECOMMENDATIONS

I recommend removing the `virtual` keyword. It may involve refactoring the derived mock contract, but the testing logic should be considered secondary to the security and readability of the production code.

Notes

Issues with a very low impact

SPELLING ERROR IN METHOD DOCUMENTATION

There is a typo in the documentation of the `delegateAllowance` method. Where it says `acconut` , it should say `account` .

INCONSISTENT NAMING

The `DelegateERC20` contract uses `delegateIncreaseApproval` and `delegateDecreaseApproval` methods to implement the logic named `increaseAllowance` and `decreaseAllowance` in the `ERC20` contract. It will be advised to unify the naming conventions or annotate that both methods implement the same functionality.

WRONG DESCRIPTION OF THE RETURN PARAMETER

The `delegateAllowance` method is described to return the `success` value while it doesn't perform any updates and returns the current value of user allowance.

Jakub Wojciechowski
Smart contract auditor