

Security Audit

of TENX's Smart Contracts

February 15, 2019

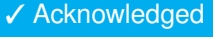
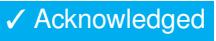
Produced for



by



Table Of Content

Foreword	1
Executive Summary	1
Audit Overview	2
1. Scope of the Audit	2
2. Depth of Audit	3
3. Terminology	3
Limitations	4
System Overview	5
1. Overview	5
2. Extra Token Features	5
Best Practices in TENX's project	6
1. Hard Requirements	6
2. Soft Requirements	6
Security Issues	7
1. Dependence on block information  	7
2. Front running possible  	7
3. Possibility to withdraw additional reward tokens  	7
4. Possibility to get double rewards  	7
Trust Issues	9
1. Inconsistent way of issuing tokens  	9
2. Inconsistent PAY token balance in the Rewards contract  	9
3. PAY token to TENX token conversion happens off-chain  	9

4.	Redundant issuer scheme			10
5.	The Controller and Regulator responsibilities are partially conflicting			10
6.	It is possible to mint (in total) too many TENX token			10
7.	Input to reward scheme			10
Design Issues				11
1.	PauserRole.sol not used			11
2.	Inconsistent reward scheme			11
3.	ERC-1644 and ERC-1594 standard functions have public visibility			11
4.	Inefficient struct			12
5.	Issuer allowed to update claim amount			12
6.	Omitted function return			12
7.	Unnecessary function call			12
8.	No sanity check for new regulator			13
9.	SignedSafeMath should be updated			13
10.	Possibility of unclaimable tokens			13
11.	Locked PAY Tokens			13
Recommendations / Suggestions				15
Disclaimer				16

Foreword

We first and foremost thank TENX for giving us the opportunity to audit their smart contracts. This documents outlines our methodology, limitations, and results.

– ChainSecurity

Executive Summary

The TENX smart contracts have been analyzed under different aspects, with a variety of tools for automated security analysis of Ethereum smart contracts.

Overall, CHAINSECURITY found that TENX employs good coding practices. Anyway, CHAINSECURITY found one higher security issue and some flaws. But the majority of these are trust and design issues that could be avoided and were addressed or fixed by TENX in a professional manner. TENX uses a new token standard (ERC-1400) that is currently still in draft status and thereby could still be subject to changes. Therefore, CHAINSECURITY recommends TENX to keep an eye on the latest developments regarding this standard.

Audit Overview

Scope of the Audit

The scope of the audit is limited to the following source code files. All of these source code files were received on January 23, 2019. The latest update has been received on February 12, 2019. The corresponding Git commit is: 10e8ef4de4d426e5a6ec1553268cfd040c7f07f1.

File	SHA-256 checksum
./contracts/lib/CanReclaimPAY.sol	e4002dbee523eba1487b949a2e56042718ae006e69c8c9a667ae7a73f62f0c1d
./contracts/lib/Blacklistable.sol	50cf57835181dba3c17192f80da1460b3d228398bf43f8e31aa3a83159e29af8
./contracts/lib/Whitelistable.sol	08fada5ff2dc2456c8ff5c2588b2313a696711177fe7418761cdccdc46957d85
./contracts/token/RewardableToken.sol	824266cd13156668d82293eb81e8a1112fd921aaf7cb671fcfe57cef4ca59131
./contracts/token/PayToken.sol	eae172df1f9acae985b17f38f778b5fe724eb902d7b0fdb800c68f47b8912fb0
./contracts/token/ERC1644.sol	ada77bc1565f96aa5892cb74bfb235bee8394817d89f3996e481ebf1a608c13
./contracts/token/ERC1594.sol	289d5a2d4a214fc78c705a5afc15a1be4c1e65167681932d66b8e2be14953dc8
./contracts/token/ERC20Redeemable.sol	6f4d39968306148bd76435a598ab90bc863644b7e15fd7ede36c01d38065c080
./contracts/token/ERC20Capped.sol	e82755100d18b813d54e43b21d41a60d8df5b36746299d20237d6b30d0aaadd
./contracts/token/ERC1400.sol	97b59831b66aeeb69deaaec87bfe1b6bdabf0a605c24c9c94ad6c94b812115a
./contracts/token/TENXToken.sol	5869cb009c78fd3612bc3470fc79f0d8148ede228a861cf90b56460be7d82696
./contracts/rewards/Rewards.sol	463e7aadee5a3a808f7b9f6cbc25488b66b4598285ff024537c44ae176636fc9
./contracts/rewards/Rewardable.sol	ad63efd0668e9b94b1ccbc81277581e8d6cc54eddf76861e66ea39eac462560a
./contracts/compliance/Regulated.sol	a364948b9264f1320317aa4a77a05098caf9ad85410c2eeb92dfb685d7c23683
./contracts/compliance/PermissionedRegulator.sol	ea5eb6ce87d016e57ab3acddcfed87895531db68db3482b0cc9e2a70f5fd93c8
./contracts/compliance/BasicRegulator.sol	18927177789edbb9825f7c2677cdf5e61917b50cad170f3addf55fab7b6dac3
./contracts/compliance/BlacklistRegulator.sol	89a1d72e851e564e152c4f940f297c7d73c0eb0e847497e2808a78e785e9ffb3
./contracts/issuance/Issuer.sol	07f42866efb8329fffc9d775bc55cf8f2a09eb337d2bd8447a72b39a6ca9ce47
./contracts/roles/RegulatorRole.sol	0597b32d8c914f2c02063ae2c0ac1f6dc4198aabb8af7e681eef02d2c680a303
./contracts/roles/IssuerStaffRole.sol	46fa4ff63868618556a2c904db11005ff6a7003404456ee4142f05273b9e5dc2
./contracts/roles/RewarderRole.sol	174eb75abcd0bd6d7ed4a59df8a0010bc3e6e548ad766fa1c979d7602abe3abf
./contracts/roles/IssuerRole.sol	2921c9ccc56856c8ac118bcaf9cc2e40673be587951c694d47de8124da3e09af
./contracts/roles/ControllerRole.sol	dd6fcfd9e7bd643b9003dbd348ce215125fcef7d01cbf0447147a1b400dba446
./contracts/interfaces/IERC1644.sol	6154670109e0bd1919d617c5934453bd61465dbdb518caaa47542d3a59957d01
./contracts/interfaces/IIssuer.sol	397f1e3ea0a26768dbaa9704f10dbec185e03653d401e41c1a1ca90f4681bbd5
./contracts/interfaces/IPAYToken.sol	362faeca6ae56ce997587e998a6607dac9a0013e77b79737db8247a3fbb5e3b9
./contracts/interfaces/IERC1594Capped.sol	e46f212b80fc5e972ef1c0fd86c7f3d388153a5cb30a5a8effab0af293cc2203
./contracts/interfaces/IERC20Capped.sol	1c80f4a3e1d167df6a8b6812ef4677487d35a36d3b79afa4fae469059110e88e
./contracts/interfaces/IRegulator.sol	7d3e24a63a1dada3a9ce2212727a9df21dca2165e6e113d07eac981ccdc03120
./contracts/interfaces/IHasIssuership.sol	2e6fa4c89fff6d74fa65905f67ef7e1180224d32b9dd013d0702d81c2b727e30
./contracts/interfaces/IERC1594.sol	aac02004f5d001a51625f47b7d159713ef73832080c0def95005b869e509d7e9
./contracts/interfaces/IRewards.sol	5d61aef76cfb8764717725f7b2b4d55b9e3ba2ffe5af3f9f018628d7f25e5ffe
./contracts/interfaces/IRewardable.sol	bb8ce4f75f0fa91baf19c5337115efeb67d2ceaf92a138e7f3be50c8a492ece8

Depth of Audit

The scope of the security audit conducted by CHAINSECURITY was restricted to:

- Scan the contracts listed above for generic security issues using automated systems and manually inspect the results.
- Manual audit of the contracts listed above for security issues.

Terminology

For the purpose of this audit, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology¹).

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical and business related consequences of an exploit.

Severity is derived based on the likelihood and the impact calculated previously.

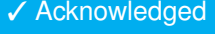
We categorize the findings into 4 distinct categories, depending on their severities:

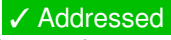
-  Low: can be considered as less important
-  Medium: should be fixed
-  High: we strongly suggest to fix it before release
-  Critical: needs to be fixed before release

These severities are derived from the likelihood and the impact using the following table, following a standard approach in risk assessment.

LIKELIHOOD	IMPACT		
	High	Medium	Low
High			
Medium			
Low			

During the audit concerns might arise or tools might flag certain security issues. After careful inspection of the potential security impact, we assign the following labels:

-  No Issue : no security impact
-  Fixed : during the course of the audit process, the issue has been addressed technically
-  Addressed : issue addressed otherwise by improving documentation or further specification
-  Acknowledged : issue is meant to be fixed in the future without immediate changes to the code

Findings that are labelled as either  Fixed or  Addressed are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview what kind of issues were found during the audit.

¹https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing allows to discover vulnerabilities that were overlooked during development and areas where additional security measures are necessary.

In most cases, applications are either fully protected against a certain type of attack, or they lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. We therefore carry out a source code review trying to determine all locations that need to be fixed. Within the customer-determined timeframe, CHAINSECURITY has performed auditing in order to discover as many vulnerabilities as possible.

System Overview

Token Name & Symbol	TENX TOKEN, TENX
Decimals	18
Issuing Rate	1 PAY: 1 TENX
Hard Cap	205,218,255.948577763364408207
Token Type	ERC-20, ERC-1400 (ERC-1594, ERC-1644)
Token Generation	Issuable, Burnable, Capped, Redeemable
Vesting	No
Pausable	Yes
KYC	Can be added
Owner Rewards	Yes

Table 1: Facts about the TENX token and the Token Sale.

Overview

TENX introduces a security token. The token implements the ERC-20 and ERC-1400 (ERC-1644 and ERC-1594) specifications. It grants the right to receive $Y\%$ ($Y = \text{amount TENX tokens owned by account} / \text{total amount of TENX tokens}$) of the PAY tokens which are deposited in the rewards contract by TENX. An eligible user can withdraw his share of PAY token from the reward contract. A user is eligible if he passed KYC and held TENX token when the deposit was made.

The TENX token is issued to all PAY token holders (based on a snapshot from end of 2018). The ratio PAY token/TENX token is 1:1. To receive PAY token the account needs to go through KYC and maybe also claim the token (depending if the token is issued or airdropped by the issuer). The token is transferable only between KYC approved accounts. If TENX tokens are transferred, past rewards still belong to the account which had the tokens when the reward was deposited. For security reasons it is possible to pause and unpause token and reward transfers.

Extra Token Features

Emergency Drain The Owner of the contract can drain any amount of funds from the Rewards contract.

Rewards TENX tokens holders are entitled to receive a share of the PAY tokens that are deposited into the Rewards contract by TENX.

Damping TENX keeps track of the rewards. A TENX token transfer does only transfer the tokens but not past unclaimed rewards.

ERC-1400 (ERC-1644 and ERC-1594) The combination of ERC-1644 and ERC-1594 introduces the possibility for a regulator role to verify and interfere with token transfers (ERC-1594). Furthermore, a controller role has the power to redeem or transfer tokens from and to any account (ERC-1644).

Best Practices in TENX's project

Projects of good quality follow best practices. In doing so, they make audits more meaningful, by allowing efforts to be focused on subtle and project-specific issues rather than the fulfillment of general guidelines.

Avoiding code duplication is a good example of a good engineering practice which increases the potential of any security audit.

We now list a few points that should be enforced in any good project that aims to be deployed on the Ethereum blockchain. The corresponding box is ticked when TENX's project fitted the criterion when the audit started.

Hard Requirements

These requirements ensure that the TENX's project can be audited by CHAINSECURITY.

- ☒ The code is provided as a Git repository to allow the review of future code changes.
- ☒ Code duplication is minimal, or justified and documented.
- ☒ Libraries are properly referred to as package dependencies, including the specific version(s) that are compatible with TENX's project. No library file is mixed with TENX's own files.
- ☒ The code compiles with the latest Solidity compiler version. If TENX uses an older version, the reasons are documented.
- ☒ There are no compiler warnings, or warnings are documented.

Soft Requirements

Although these requirements are not as important as the previous ones, they still help to make the audit more valuable to TENX.

- ☒ There are migration scripts.
- ☒ There are tests.
- ☒ The tests are related to the migration scripts and a clear separation is made between the two.
- ☒ The tests are easy to run for CHAINSECURITY, using the documentation provided by TENX.
- ☐ The test coverage is available or can be obtained easily.
- ☒ The output of the build process (including possible flattened files) is not committed to the Git repository.
- ☒ The project only contains audit-related files, or, if not possible, a meaningful separation is made between modules that have to be audited and modules that CHAINSECURITY should assume correct and out of scope.
- ☒ There is no dead code.
- ☒ The code is well documented.
- ☐ The high-level specification is thorough and allow a quick understanding of the project without looking at the code.
- ☒ Both the code documentation and the high-level specification are up to date with respect to the code version CHAINSECURITY audits.
- ☒ There are no getter functions for public variables, or the reason why these getters are in the code is given.
- ☒ Function are grouped together according either to the Solidity guidelines², or to their functionality.

²<https://solidity.readthedocs.io/en/latest/style-guide.html#order-of-functions>

Security Issues

In the following, we discuss our investigation into security issues. Therefore, we highlight whenever we found specific issues but also mention what vulnerability classes do not appear, if relevant.

Dependence on block information



✓ Acknowledged

The functions `canSend()`, `canReceive()`, `isTimelocked()` and `setPermission()`, which are located inside the `PermissionedRegulator` contract makes use of the special `block.timestamp` field. Although block manipulation is considered hard to perform, a malicious miner is able to move forward block timestamps by up to 900 seconds (15 min) compared to the actual time. CHAINSECURITY notes that TENX and its users should be aware of this and adhere to the 15 seconds rule³

Likelihood: Low

Impact: Low

Acknowledged: TENX has acknowledged that they are aware of this and will adhere to the 15 seconds rule.

Front running possible



✓ Acknowledged

The issuer can issue a new claim, containing an amount and a payee. Such a claim can then be claimed by the payee. In case, the issuer update the amount of a previously created claim, this update is vulnerable to front running by the payee. If the claimable amount is lowered in the new claim, the payee can front-run the update by calling the `claim` function to receive the higher amount.

Likelihood: Low

Impact: Medium

Acknowledged: TENX has acknowledged that they are aware about this issue. TENX added this as a functionality to re-issue an unclaimed claim in case one was made with a wrong amount. Although this is super unlikely to happen TENX has decided to keep the functionality as is.

However, CHAINSECURITY recommend to document this functionality.

Possibility to withdraw additional reward tokens



✓ Fixed

The way the damping value is recalculated after the redemption of TENX TOKENS allows redeeming users to claim a higher amount of reward tokens than they would normally be entitled to. If a user makes such an additional claim, then the additionally withdrawn tokens are being stolen from other honest users.

Likelihood: Medium

Impact: High

Fixed: TENX changed the damping calculation after redemption. Now a redemption also leads to an update of the `totalRewards` variable which therefore, makes sure that already withdrawn rewards cannot be withdrawn again.

Possibility to get double rewards



✓ Acknowledged

In theory, a token holder can claim a double reward by performing many transfers with tiny token amounts. Token transfers are supposed to update the `_dampings` which is updated as follows: `_totalRewards * _sharesChange / _totalShares`.

If `_sharesChange`, so the amount of transferred tokens, is smaller than `_totalShares / _totalRewards`, then no `_dampings` change will occur. This allows an attacker to withdraw rewards multiple times. However, this is very unlikely, as it occurs high gas costs.

We provide an example:

³<https://consensys.github.io/smart-contract-best-practices/recommendations/#the-15-second-rule>

- `_totalShares` = 200 million TENX
- `_totalRewards` = 10,000 PAY
- Maximum transfer amount without `_dampings` update $< 200,000,000 / 10,000 = 20,000$ TENX Wei
- Hence to transfer 1 TENX token using such small amounts more than 50,000,000,000,000 transfers are needed
- Each of these transfers has a gas cost of at least 10,000 gas resulting in gas cost of $5 * 10^{17}$ gas
- Given a low gas price of 1 GigaWei, these transfers cost at least 500,000,000 ETH
- If the attacker manages to use a gas price of 1 Wei, e.g. because it is also mining, these transfers cost at least 0.5 ETH.
- As a reward, the attacker receives $1 * 10,000 / 200 \text{ million} = 5 * 10^{-5}$ PAY tokens

Therefore, the attack is unlikely to be performed. It becomes impossible once so many rewards have accumulated that `_totalShares` $<=$ `_totalRewards`.

Acknowledged: TENX has acknowledged this issue and is aware of it.

Trust Issues

The issues described in this section are not security issues but describe functionality which is not fixed inside the smart contract and hence requires additional trust into TENX, including in TENX's ability to deal with such powers appropriately.

Inconsistent way of issuing tokens

There exist two different ways of issuing tokens. The expected way is through the `Issuer` contract. However, the `Issuer` or `Owner` role of the `TENXToken` contract could directly call `TENXToken.issue()` and thereby mint new token. This bypasses the checks done in the `Issuer` contract:

```
require(_payee != address(0), "Payee must not be a zero address.");
require(_payee != msg.sender, "Issuers cannot airdrop for themselves");
require(_amount > 0, "Claim amount must be positive.");
claims[_payee] = Claim({
    status: ClaimState.CLAIMED,
    amount: _amount,
    issuer: msg.sender
});
```

Thus, the payee could be the `msg.sender` and the tokens are not tagged as claimed.

Fixed: TENX solved the problem by separating the roles into two different roles. Furthermore, TENX makes sure that only one issuer can be active at a time. The owner has the power to transfer this issuer role.

Inconsistent PAY token balance in the Rewards contract

The `Rewards` contract manages the `PAY` tokens paid out. However, it contains the function `reclaimTokens()`. Calling the `reclaimTokens()` function drains all funds but the `totalRewards` variable stays untouched.

This creates an inconsistency between the perceived balance (represented by `totalRewards`, which is updated after each `deposit()` call) and the real balance and breaks the contract's functionality. Consequently, deposits will be computed incorrectly.

Furthermore, TENX might consider, if it shall be allowed for the owner to withdraw all funds from the contract at all. The funds are deposited by the `Rewarder` role. Allowing the owner to drain the funds might be an inconsistent use of the defined roles.

Fixed: TENX uses the `reclaimTokens()` only in emergency cases when there is the need to migrate to a new contract. Therefore, the `totalRewards` variable needs to stay unchanged. TENX changed the permission to call this emergency functions. Now, only the owner is allowed to call it instead of the `rewarder`. After calling `reclaimTokens()` the `Reward` contract gets unusable.

PAY token to TENX token conversion happens off-chain

TENX is using a `.csv` file in a private repository as the `PAY` token balance snapshot taken earlier. This file is the base for paying out the `TENX` tokens to `PAY` token holders. Even though all information could be checked, this check is very hard to perform for a usual user. A user needs to check all accounts' balances at the time the snapshot was made and compare them to all the tokens that were issued. Additionally, the user needs to take into account all burn, redeem and similar events.

Acknowledged: TENX acknowledged that they will open source the script on their GitHub⁴, that generates the token snapshot, so that anyone can audit and run the script themselves.

⁴<https://github.com/tenx-tech/TENX/tree/master/snapshot>

Redundant issuer scheme

There are two issuer roles which are redundant. First, we have the issuers which are added to the Issuer contract. They are supposed to issue new TENX TOKENS. Secondly, the TENXToken contract itself has an Issuer role assigned. In the current migration scripts, the Issuer contract and the deployer of the TENXToken are assigned this second role.

This setup leads to two different and redundant issuer roles, which do have different power. For the current setup to work correctly, both roles are needed. The second role is required to call the `finishIssuance` function, while the first role is required to perform regular issuances.

Individual accounts assigned the second role, can also directly issue tokens. This leads to the issue described as “Inconsistent way of issuing tokens”.

Fixed: TENX solved the problem by adding a new role called `IssuerStaff`. This role now controls the Issuer contract.

The Controller and Regulator responsibilities are partially conflicting

ERC-1644 and ERC-1594 are partially conflicting. While ERC-1594 enforces transfer restrictions through the `canTransfer` function, ERC-1644 allows forced transfers between any addresses using the `controllerTransfer`. Thus, the Controller can bypass the Regulator's checks. The ERC-1644 standard says that a `controllerTransfer` may “potentially also need to respect other transfer restrictions”. Therefore, TENX should clearly state how this conflict should be handled as it represents the power balance between the different roles.

Addressed: TENX addressed the issue and added the functions `verifyControllerTransfer` and `verifyControllerRedeem` to check for this conflict. But at present the functions do not enforce anything in the regulator contracts.

It is possible to mint (in total) too many TENX token

The TENXToken contract is also an ERC20Capped contract which maintains the maximum cap of the total Supply of the TENX tokens. However, while redeeming or burning the TENX tokens, the `totalSupply` is also reduced. Therefore, the total number of minted tokens might be larger than the cap.

CHAINSECURITY notes that this can only be done until `finishIssuance` is called.

Fixed: TENX solved the problem by introducing an additional variable `totalMinted` which only increments and keeps track of the total amount of minted TENX TOKENS.

Input to reward scheme

The funding of the reward scheme is not guaranteed on-chain. The user needs to trust TENX, that the correct reward amount is deposited into the Rewards contract by TENX.

Acknowledged: TENX acknowledged that there is no change for this. Users need to trust TENX for fair deposit of rewards amount into Rewards contract.

Design Issues

The points listed here are general recommendations about the design and style of TENX's project. They highlight possible ways for TENX to further improve the code.

PauserRole.sol not used

TENX implements a Pauser role in the `PauserRole.sol` contract. This role is not being used in any contract. Yet, the functionality is implemented because TENX uses the OpenZeppelin `Pausable` contract. CHAINSECURITY recommends removing their own implementation of the `PauserRole` if it is not used.

Fixed: TENX removed the contract `PauserRole.sol`.

Inconsistent reward scheme

The reward scheme does not efficiently allocate funds in case the `totalSupply` reduces over time. The reward calculation is based on the amount given in the constructor in `Rewards.sol` (`totalShares = _cap`). Thus, the amount of total TENX TOKENS is assumed to be constant.

But this value is not really constant. TENX TOKEN can be burned or redeemed by the Controller. This will change the total supply of TENX TOKEN. Hence, the rewards calculation is not correct any more.

For example:

Let's assume two accounts (A and B)

The total supply (all TENX TOKEN) is 20 (instead of ~200m)

A owns 10 TENX TOKEN and B owns 10 TENX TOKEN

TENX deposits 10 PAY token in the reward contract. Thus:

Reward A = 5 PAY = $10 * 10 / 20 = \text{userShares.mul}(\text{totalRewards}).\text{div}(\text{totalShares})$

Reward B = 5 PAY = $10 * 10 / 20 = \text{userShares.mul}(\text{totalRewards}).\text{div}(\text{totalShares})$

A now burns his 10 TENX TOKEN intentionally or unintentionally (or it is redeemed by the Controller because he is not allowed to own them or any other reason). The total supply is changed!

TENX now deposits another 10 PAY in the reward contract. Hence:

Reward A = 0 PAY = $0 * 10 / 20 = \text{userShares.mul}(\text{totalRewards}).\text{div}(\text{totalShares})$

Reward B = 5 PAY = $10 * 10 / 20 = \text{userShares.mul}(\text{totalRewards}).\text{div}(\text{totalShares})$

We end up with a leftover of 5 unallocated PAY tokens.

After issuing has been finished the reward calculation could be more accurate by using up-to-date `totalSupply`.

Fixed: TENX solved the problem by keeping track of the maximum amount of non-redeemed TENX TOKENS which is then used for the reward calculation. Hence, no rewards will be allocated for burnt tokens. These rewards will be distributed to active token holders proportionally.

ERC-1644 and ERC-1594 standard functions have public visibility

The ERC-1644 and IERC-1594.sol standard interfaces are defined in `IERC-1644.sol` and `ERC-1594`. The implemented functions have the visibility `public`.

However, the actually implemented ERC-1644⁵ & ERC-1594⁶ standard functions have their visibility defined as `external`. CHAINSECURITY recommends to correct the visibility of these functions.

Fixed: TENX solved the problem by correcting the visibility of the functions.

⁵<https://github.com/ethereum/EIPs/issues/1644>

⁶<https://github.com/ethereum/EIPs/issues/1594>

Inefficient struct

In the Issuer contract there is a **struct** called `Claim`, to store the status of the token claims. The **struct** is not tightly packed and would consume more gas in storage.

```
struct Claim {
    ClaimState status;
    uint amount;
    address issuer;
}
```

This struct can be optimized to consume less gas during the call to `issue()` function.

```
struct Claim {
    address issuer;
    ClaimState status;
    uint amount;
}
```

Using the above **struct** the function call to `issue()` would reduce the gas cost per function call.

Fixed: TENX solved the problem by rearranging the **struct** as proposed.

Issuer allowed to update claim amount

An Issuer is able to update the claim amount of a payee. To do so, they perform the following operations:

- An Issuer calls the `issue(payeeX, amountX)` function of the Issuer contract.
- An Issuer calls the `issue(payeeX, amountY)` function again with the updated amount `amountY` for the same address. This has to be done before the payee calls the `claim()` function.

As pointed out as a security issue, this can be vulnerable to front running attacks.

Acknowledged: TENX acknowledged that they are aware about this as this is a functionality.

Omitted function return

When calling ERC-20 standard functions like `transfer()`, they return the result as boolean to let the caller know about the execution. The function `transferWithData()` in the ERC-1594 contract is making a call to `super.transfer(_to, _value)`, however it is not checking the returned result.

CHAINSECURITY recommends checking the returned result, by enclosing the statement with an **require()**.

Fixed: TENX fixed the problem by wrapping the function calls with **require()**.

Unnecessary function call

The Rewards contract defines `_claimedRewards` as private state variable.

```
mapping(address => uint) private _claimedRewards;
```

The getter function for this is defined like:

```
function claimedRewards(address _payee) public view returns(uint) {
    return _claimedRewards[_payee];
}
```

Also, to access the values from the `_claimedRewards` the contract uses function `claimedRewards()` like below:


```
_claimedRewards[payee] = claimedRewards(payee).add(_amount);
```

Same is the case with `_dampings` state variable.

CHAINSECURITY recommends changing the visibility of the `_claimedRewards` variable to `public`, the compiler will autogenerate getter function. Then the explicitly defined function can be removed and the code can directly access the mapping instead of calling the function.

Fixed: TENX removed the custom getter for the `_claimedRewards` variable and set it to `public`. But did not do it for the `_dampings` variable.

No sanity check for new regulator

The `setRegulator()` function of the ERC-1594 contract takes the regulator contract's address and directly updates the `regulator` state variable.

It performs no sanity checks, such as checking for an accidentally supplied `address(0)` or checking for the existence of a contract at the provided address.

Fixed: TENX solved the problem by checking if the address is a zero address and if the address is a contract. It therefore uses the OpenZeppelin implementation to check this which uses `extcodesize` to do so. This should be fine in this case but CHAINSECURITY wants to generally highlight that `extcodesize` is not a bulletproof way to check if the address is a contract. As mentioned in the OpenZeppelin `Address.sol` contract's comments, this check will be incorrect if invoked during the constructor of a contract, as the code is not actually created until after the constructor finishes.

SignedSafeMath should be updated

The `SignedSafeMath` library was taken from a gnosis repository. However, the development on it has considerably progressed. Hence, TENX should update their version of the `SignedSafeMath`⁷. Among other things, `asserts` have been replaced with `requires`.

Fixed: TENX solved the issue by replacing the library code with the suggested OpenZeppelin library.

Possibility of unclaimable tokens

This is not a major issue but by design there will be PAY token which are not withdrawable for some users in the Rewards contract. The reason is that rounding errors inevitable occur in some calculations.

The snapshot lists some user with a very low PAY token balance. Thus, resulting in a low TENX TOKEN balance. The reward calculation is `_userShares.mul(_totalRewards).div(_totalShares)`. Hence, to receive any reward at all (more than one PAY token) this equation needs to hold:

$$\frac{_userShares * _totalRewards}{_totalShares} > 1 \quad (1)$$

This will not be possible if either the `_totalRewards` or the `_userShares` are too low. But as the `_totalRewards` will sum up, at some future point the possibility to withdraw the rewards increases (but still depends on the amount of `_userShares`). CHAINSECURITY checked this and this occurs only if the amounts are super low. Furthermore, there is an inevitable rounding error of up to 1 TENX wei for each reward calculation. However, this value is very small and does not accumulate over time.

Acknowledged: TENX has acknowledged this issue and is aware of it.

Locked PAY Tokens

Given the close connection between PAY tokens and TENX tokens, PAY tokens might accidentally be sent to a wrong contract, e.g. the TENX token contract. In such a case, they would be locked in that contract. TENX could add a functionality to recover PAY tokens from these contracts.

Note, that TENX token can be recovered using the `controllerTransfer`.

⁷Example: <https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/drafts/SignedSafeMath.sol>

Fixed: TENX solved the problem by creating a `CanReclaimPAY` contract with a function `reclaimPAY` to reclaim tokens send to the contract address. TENX inherited this functionality to the `TENXToken` and `Issuer` contract.

Recommendations / Suggestions

- ☒ TENX claims, that the TENX token can only be transferred between KYC-approved accounts. The `BasicRegulator` contract does simply approve all transactions. TENX already pointed out that the `BasicRegulator` contract is fine according to their legal advisors. Anyway, this is a clear mismatch between TENX's claims and what is implemented and hence, needs to be mentioned.

CHAINSECURITY therefore, suggests to properly update the documentation and mention both contracts and also how they will be used.

- ☐ TENX needs to remember to upgrade the migration scripts which currently still hold some TODOs.
- ☐ The constructor of the `Rewards.sol` contract sets the PAY token address. This address is known because the contract is already deployed. To avoid a trust issue, for the live system, it would make sense to hardcode this address, to avoid any mistakes/misuse and simultaneously build trust.

Post-audit comment: TENX has fixed some of the issues above and is aware of all the implications of those points which were not addressed. Given this awareness, TENX has to perform no more code changes with regards to these recommendations.

Disclaimer

UPON REQUEST BY TENX, CHAINSECURITY LTD. AGREES MAKING THIS AUDIT REPORT PUBLIC. THE CONTENT OF THIS AUDIT REPORT IS PROVIDED "AS IS", WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND, AND CHAINSECURITY LTD. DISCLAIMS ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT. COPYRIGHT OF THIS REPORT REMAINS WITH CHAINSECURITY LTD..