Corpus Ventures - Tokenize.it

Date	November 2022
Auditors	David Braun, Martin Ortner

1 Executive Summary

This report presents the results of our engagement with **Corpus Ventures** to review **Tokenize.it**, a platform that enables company shares to be created and traded on an EVM-compatible chain.

The review was conducted over one week, from **November 28, 2022** to **December 2, 2022**. A total of 2 x 5 person-days were spent.

After delivery of the initial report, the client added fixes and improvements based on our report as well as some other changes to the codebase. The fixes have been reviewed in the week of December 12-16, 2022. Most problems have been fixed, but issue 4.1 has only been partially addressed, and the changes in the context of finding 4.4 have introduced new problems. The "Resolution" boxes contain more detailed information.

2 Scope

All and only the files in the `contracts/` directory were in scope; a detailed list with SHA-1 hashes can be found in the Appendix.

The client initially provided v2.2 with commit hash: 7619664f289db76e4c2423e4451ce2b62a934ea3. On the first day of the engagement, it was agreed to switch to the more recent version v2.4 with commit hash e508ab97d8981b47eddba36809431bc74c6592d3, which already addressed some issues.

The version the client sent us to review the fixes is v3.0 with hash becbd359ed0a5284bf908487f2738a6530430a4b. It should be noted that this version contains some changes that are unrelated to our initial report and that those have not been taken into consideration. We have, in particular, not vetted these unrelated changes for compatibility with the fixes we suggested and reviewed, nor did we review them on their own or v3.0 as a whole. Instead, the fixes were treated as if they were the sole changes on top of v2.4.

2.1 Objectives

Together with the client, we identified the following priorities for our review:

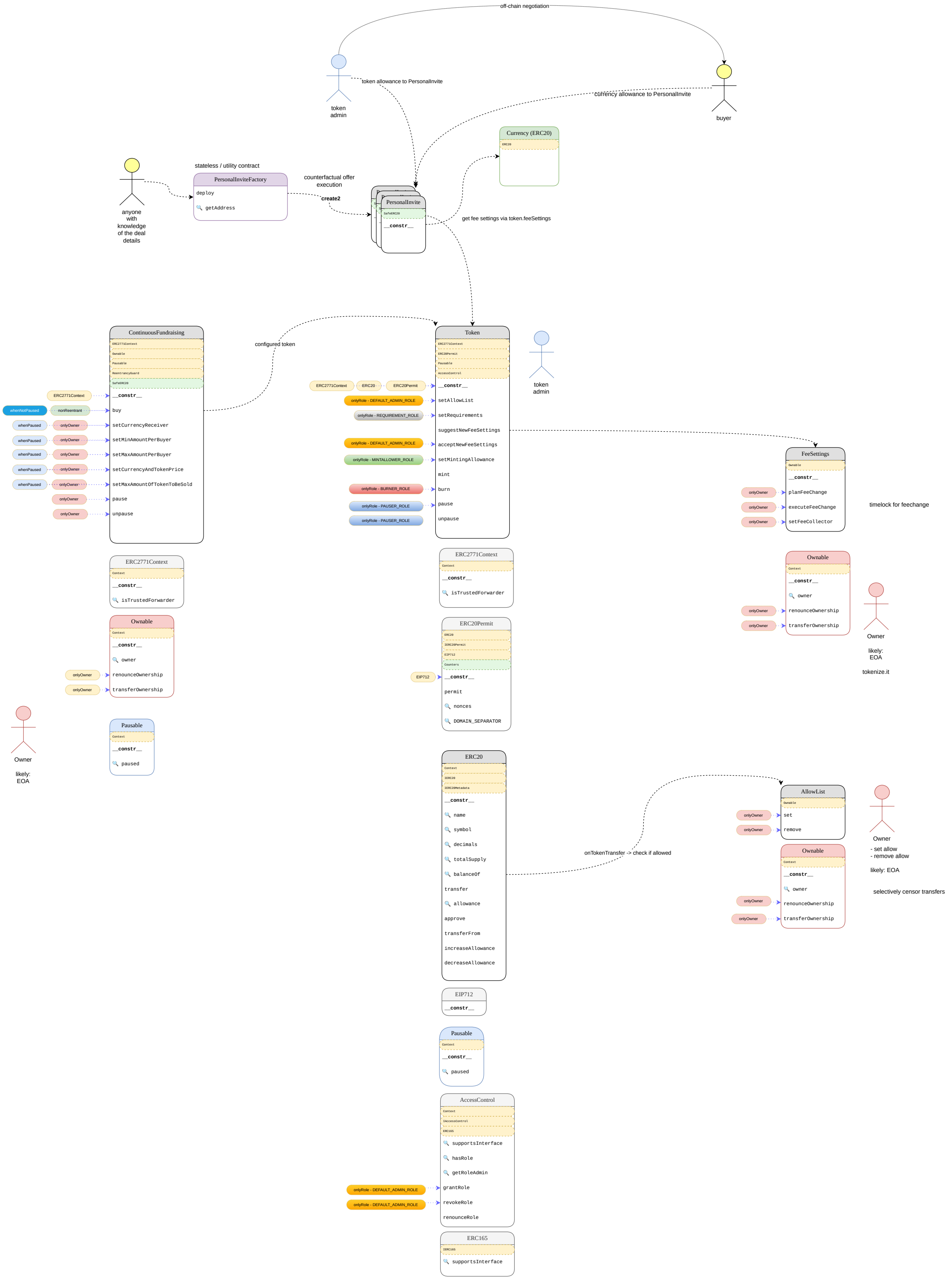
1. Ensure that the system is implemented consistently with the intended functionality, and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.
3. Mitigate the following risks:
   - check if all invariants in docs/specification.md hold
   - loss of funds
   - unintended minting of tokens
   - 3rd parties gaining privileges in any of the contracts
   - frontrunning
   - reentrancy

3 System Overview

This section describes the top-level/deployable contracts, their inheritance structure, interfaces, actors, permissions, and essential contract interactions of the system under review.

3.1 Architecture Diagram

Contracts are depicted as boxes. Public reachable interface methods are outlined as rows in the box. The 🔍 icon indicates that a method is declared non-state-changing (view/pure) while other methods may change state. A yellow dashed row at the top of the contract shows inherited contracts. A green dashed row at the top of the contract indicates that that contract is used in a usingFor declaration. Modifiers used as ACL are connected as yellow bubbles in front of methods.



Overview

3.2 Roles

AllowList.sol

Role	Related Functions
Owner	`set()` , `remove()`

ContinuousFundraising.sol

Role	Related Functions
Owner	`setCurrencyReceiver()` , `setMinAmountPerBuyer()` , `setMaxAmountPerBuyer()` , `setCurrencyAndTokenPrice()` , `setMaxAmountOfTokenToBeSold()` , `pause()` , `unpause()`

FeeSettings.sol

Role	Related Functions
Owner	`planFeeChange()` , `executeFeeChange()` , `setFeeCollector()`

Token.sol

Role	Description	Related Functions
`BURNER_ROLE`	Burn tokens from anywhere.	`_beforeTokenTransfer()` , `burn()`
`DEFAULT_ADMIN_ROLE`	Default admin role for all roles.	`setAllowList()` , `acceptNewFeeSettings()`
`MINTALLOWER_ROLE`	Grant minting allowances.	`setMintingAllowance()`
`PAUSER_ROLE`	Pause the token.	`pause()` , `unpause()`
`REQUIREMENT_ROLE`	Define which requirements an address must satisfy to receive tokens.	`setRequirements()`
`TRANSFERERADMIN_ROLE`	Grant transfer rights to other addresses.	

Role	Description	Related Functions
`TRANSFERER_ROLE`	Addresses with this role do not need to satisfy any requirements to send or receive tokens.	`_beforeTokenTransfer()`

4 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 `Token._beforeTokenTransfer` allows more than necessary Medium Partially Addressed

Resolution

Our concerns regarding the `BURNER_ROLE` have been met; in v3.0, an account B with the `BURNER_ROLE` is not allowed anymore to move funds from a red account R (unless B is also the fee collector and B = R; see below).

We also pointed out that the fee collector can not only be the recipient of minted *fees* but also the recipient of a regular mint operation. The client has reconsidered the fee collector’s privileges and decided to extend them: Now the fee collector – in addition to their previous privileges – behaves like a green account. Specifically, the fee collector,

even if their account is red, can:
(1) send tokens to a green account (or themself);
(2) receive tokens from a green account;
(3) be the recipient of a fee mint operation;
(4) be the recipient of a regular (i.e., non-fee) mint operation (and receive the fees too).
It should be noted that (1) is a newly introduced privilege in v3.0, whereas (2), (3), and (4) have already been present in v2.4.

- We did not question (3).
- A convincing case can be made for (1), but we'd like to remark that this also means that the company is no longer entirely in control of freezing funds, and it extends the potential targets of bribery (for the movement of frozen funds) to the `FeeSettings` owner, who belongs to the platform. Assuming the motivation for this change is to ensure that the company cannot freeze the collected fees, note that the `BURNER_ROLE` could still even burn them. So more restrictions might be necessary, and they will be complicated by the fact that the fee collector address can change.
- We still believe that privilege (2) is not necessary and should be removed.
- We also maintain that (4) should be removed. To emphasize this point, note that the company cannot keep the fee collector from calling `ContinuousFundraising.buy()` – a privilege that is not necessary for the collection of fees.

Description

OpenZeppelin's `ERC20` contract has a `_beforeTokenTransfer` hook that is frequently used to formulate (additional) requirements that must be fulfilled for a transfer (or minting/burning) to succeed. Tokenize.it's `Token` contract utilizes this hook too:

code/contracts/Token.sol:L244-L269

```
/**
@notice aborts transfer if the sender or receiver is neither transferer nor fu
@dev this hook is executed before the transfer function itself
 */
function _beforeTokenTransfer(
    address _from,
    address _to,
    uint256 _amount
) internal virtual override {
    super._beforeTokenTransfer(_from, _to, _amount);
    _requireNotPaused();
    require(
        allowList.map(_from) & requirements == requirements ||
            _from == address(0) ||
            hasRole(BURNER_ROLE, _msgSender()) ||
            hasRole(TRANSFERER_ROLE, _from),
        "Sender is not allowed to transact. Either locally issue the role as
    ); // address(0), because this is the _from address in case of minting ne
    require(
        allowList.map(_to) & requirements == requirements ||
            _to == address(0) ||
            _to == feeSettings.feeCollector() ||
            hasRole(TRANSFERER_ROLE, _to),
        "Receiver is not allowed to transact. Either locally issue the role
    ); // address(0), because this is the _to address in case of burning toke
}
```

For the sake of brevity, let us call a non-zero account “green” if it has the `TRANSFERER_ROLE` or its entry in the `allowList` satisfies the contract’s `requirements`. We call a non-zero account “red” if it is not green.

Roughly speaking, the intention is to allow only green accounts to send and receive tokens. Two special cases have to be considered, though:

- Accounts with the `BURNER_ROLE` should be able to burn tokens from any account.
- The code above suggests that the fee collector should be able to receive fees even if the account is not green.

The implementation of `_beforeTokenTransfer` allows these exceptions – and is even a bit lenient:

1. An account with the `BURNER_ROLE` can also move its own funds – or funds it has been approved to move – to a different (green) account. It is difficult to assess the implications of this. A conceivable exploit scenario is that the owner of a red account R bribes the burner to move R's (supposedly) frozen tokens somewhere else. On the other hand, even if this option is removed, R could still try to bribe someone who can make their account green and then move the tokens themself. But in any case, there's no reason to grant an already very powerful role more privileges than deemed necessary. It might also be nice to have the guarantee that tokens cannot be moved from red accounts (only burnt).
2. The fee collector – even if their account is red – can not only receive fees but also be the recipient of a regular transfer or a regular mint operation. The latter means, for example, that the fee collector could just buy shares like a normal investor – and get the fees too... Again, it would be cleaner to only grant the privileges that are considered necessary, so the fee collector can only receive fees (unless their account is green).

Recommendation

We suggest modifying the `_beforeTokenTransfer` function such that it implements these restrictions. It is probably helpful to structure the function as follows:

```
// general requirements
...

if (_from == address(0)) {
    // requirements for minting
    ...
} else if (_to == address(0)) {
    // requirements for burning
    ...
} else {
    // requirements for regular transfers
    ...
}
```

The function's arguments are not sufficient to determine if we're in a regular minting or a fee-minting process, so fixing 2 is not as straightforward as fixing 1. An ugly workaround is to set a boolean state variable `feeMintingInProgress` to `true` immediately before the fee minting and back to `false` immediately after.

4.2 Negative implications of insisting on absence of rounding errors Medium ✓ Fixed

Resolution

The client has adopted the first solution outlined below, i.e., the currency amount that has to be paid for a requested amount of shares is obtained by rounding up; with this approach, the investor's loss is at most 1 currency bit. Candidate currencies must be vetted to make sure the loss of 1 bit is negligible or at least acceptable.

Our recommendation to implement `view` functions that determine (1) the amount of share bits that a certain number of currency bits will buy and (2) the number of currency bits that have to be paid for a requested number of share bits has not been followed. We'd like to point out that the latter could be useful for the on-chain computation of necessary approval amounts.

Description

In the following discussion, we will adopt the terminology used by Tokenize.it in their code and the accompanying documentation. To briefly summarize, "token" or "shares" is used for the asset that is minted to investors, and "currency" refers to the asset that is used to pay for these shares. (Technically, both are ERC-20 tokens, but we'll avoid the term "token" for the currency, obviously.) For an asset A, the term "Abit" refers to the smallest subunit of that asset. So 1 ETHbit is synonymous to 1 Wei; if T is a deployment of the `Token` contract, then T has 18 decimals, and therefore 1 T = 10^18 Tbit. Similarly, as USDC has 6 decimals, we have 1 USDC = 10^6 USDCbit. Prices are denominated as number of currency bits per token T (not token *bits*). Consequently, if `amount` token bits are to be minted and the price is `price`, then the amount of currency bits to pay is `amount * price / 10^d`, where `d` is the number of token decimals (typically 18). But what if this number is not an integer, i.e., `amount * price` is not a multiple of `10^d`? In order to avoid rounding errors, the development team has chosen to revert the minting/buying process if the remainder of this division is not zero. In this issue, we will discuss the implications of this choice and compare it to alternative ways of dealing with this situation. We'll start with an analysis of the rounding error.

Many projects tolerate rounding errors – and that is perfectly fine as long as the error is negligible. Rounding should always happen in the direction that benefits the contract. So if we're computing an amount that the contract should transfer to the user (or another contract), we round down; if we want to calculate an amount that the user (or another contract) has to *pay*, we're rounding up. That avoids any incentive to "exploit" the rounding error. It does, however, incur a loss for the user, but if this loss is negligible, that is typically acceptable to users. Let's look at the example discussed in the documentation, i.e., we're dealing with a token T with 18 decimals, and the currency is USDC. Assume Alice wants to buy `amount` token bits. We calculate the amount of currency bits Alice has to pay as `ceil(amount * price / 10^18)`. (Note that we round up, as Alice has to pay the contract for the shares minted.) Analyzing the rounding error is easy: Alice's loss is less than 1 USDCbit – which is 0.000001 USDC because USDC has 6 decimals. That's a tiny fraction of a cent, and we may safely assume that Alice doesn't care. In general, Alice cannot lose more with this approach than 1 bit of the currency, so as long as that doesn't have significant value, the rounding-up solution works fine. However, for a currency where 1 bit does have significant value (more than 1 cent, say, or perhaps even a dollar), Alice may suffer a loss that can be considered relevant.

We now compare that solution to the revert-if-non-zero-remainder approach the project adopted: Let us look at a concrete example and assume the price is 123.456789 USDC per share, i.e., 123456789. Bob wants to invest 200 dollars. How many token bits can he buy? As `amount * price` must be a multiple of 10^18 and `price` and 10^18 are relatively prime, `amount` must be a multiple of 10^18. That means only *entire* shares can be bought. Either Bob has to spend considerably more than he wanted to – almost 250 dollars – and buy 2 shares, or he has to settle for 1 share and a smaller investment than he intended – roughly 123 dollars. Both options might very well inconvenience Bob more than losing 1/1000 of a cent due to a rounding error. Admittedly, the price was deliberately chosen to exhibit worst-case behavior, but the example shows that this solution – despite the guarantee that the investor will not suffer any losses due to rounding errors – is not entirely without problems and can, at least for “bad” prices, frustrate investors and cause more harm than a tiny rounding error would.

Recommendation

We have demonstrated that for currencies where 1 bit has only insignificant value (for example, 1 cent or less), rounding in the direction that favors the contract is a feasible solution. If you can commit to only accepting such currencies – for the sake of brevity, let’s call them high-granularity currencies – we recommend this easy and pragmatic solution. Note that low-granularity currencies – let’s say 1 bit is worth 100 dollars – are not well-suited for the payment of shares anyway because the price can only be defined in 100-dollar steps.

If, however, you cannot make the commitment to high-granularity currencies and would like to keep the current solution, the effect discussed above should be noted and clearly communicated to any affected parties. Specifically, the smallest fraction of a share that can be bought is `1 / gcd(price, 10^d)`, where `d` is the number of decimals of the token. If possible, the price should be adjusted to mitigate the impact of this effect. In the example above, pricing one share at 123.45 or 123.46 USDT would not make a big difference overall but guarantee that shares can be bought with a precision of 4 decimals. Such price adjustments are not always feasible, though, and we recommend abandoning this approach.

There’s a third option: When the investor specifies the amount `a` of token bits they want to buy, the contract first determines – rounding up – the amount of currency bits that have to be paid for `a` token bits, i.e., `c := ceil(a * price / 10^18)`. (We assume the token has 18 decimals.) That’s what the investor has to pay. However, the actual amount of token bits that the investor receives is not `a` but the maximum number of token bits they could *still* get for `c` currency bits. Formally, the investor receives `t := max { x | x * price / 10^18 <= c } = floor(c * 10^18 / price)` token bits. Note that the definition of `t` implies that `t + 1` token bits cost more than `c` currency bits, since `(t + 1) * price / 10^18 > c`. Hence, if the investor gets `t` token bits for `c` currency bits, the loss through rounding errors is less than the value of 1 token bit. For a token with 18 decimals and under the reasonable assumption that 1 share is worth less than a trillion (10^12) dollars, the investor's loss is less than 1/10000 cent. This solution works well with currencies of any granularity. The downside is that for the typical case of high-granularity currencies, the gas costs for the additional calculation will likely outweigh the small gain in token bits. Another potential problem with this solution is confusion on the investor's part because they might receive more token bits than they actually requested. Of course, it is also possible to change the interface and have the investor specify their investment directly in currency bits.

In summary, we have shown that it is possible to control the rounding errors. We believe insisting on preventing them altogether is not necessary and causes avoidable problems, so we recommend choosing an alternative solution based on the discussion above.

It might also be beneficial to have `view` functions that (1) return the amount of token bits that can be bought with a given amount of currency bits and (2) return the amount of currency bits necessary to buy a given amount of token bits (and perhaps the greatest amount of token bits that can be bought with the same amount of currency bits, cf. option three above).

4.3 Currency Tokens with fee on transfer are not supported

Medium ✓ Fixed

Resolution

The client decided explicitly against handling such tokens in the code and added detailed information about supported payment currencies to the documentation. In addition, we would recommend adding NatSpec comments in `ContinousFundraising.sol` and `PersonalInvite.sol` in case users deploy the contracts on their own.

Description

There are ERC20 tokens that charge fees for every `transfer()` or `transferFrom()`
.

In the current implementation, `PersonalInvite` and `ContinuousFundraising` assume that the received amount is the same as the transfer amount. However, this may not always be the case.

For example, as a result, `ContinuousFundraising.buy()` calculates a `currencyAmount` that does not take that fee into account leading to the company receiving less “currency” than expected for the requested amount of company token.

Examples

code/contracts/PersonalInvite.sol:L73-L82

```
    fee =
        currencyAmount /
        _token.feeSettings().personalInviteFeeDenominator();
    _currency.safeTransferFrom(
        _buyer,
        _token.feeSettings().feeCollector(),
        fee
    );
}
_currency.safeTransferFrom(_buyer, _receiver, (currencyAmount - fee));
```

Recommendation

Give guidance (Documentation) on how to use the system with deflationary/inflationary tokens. Consider comparing the before and after balance to get the actual transferred amount. Ensure that the method is reentrancy protected.

4.4 FeeSettings - fee change can be executed on a zero-init fee struct immediately Medium

Resolution

The client acknowledged this behavior and came to the conclusion that they actually appreciate it. In fact, they decided to implement a more comprehensive mechanism that allows fee *reductions* without a required minimum delay.

The new logic is neither correct nor complete, though. First of all, a change from 0 to a different value is not recognized as fee increase, making it possible to introduce fees without delay. Moreover, not every fee reduction is correctly detected: When at least one fee is changed to 0, but not all of them are 0 after the change, then the 12-week delay is always imposed, even if no fee increases.

Finally, we'd like to point out that an "accidental" call to `executeFeeChange`, when no fee change has been planned before, will set all fees to 0 – and raising them again takes 12 weeks (assuming the flaw above has been fixed). Setting `proposedFees.time` to a high value such as `type(uint256).max` – both in the constructor and at the end of `executeFeeChange` – will make such accidental calls revert and might therefore be a bit safer. An alternative approach is to leave the `proposedFees` struct unchanged at the end of `executeFeeChange` and initialize it in the constructor to the given fee values. That will make such accidental calls not revert, but they will have no effect other than emitting the event and wasting gas.

Description

`FeeSettings` promises that changes to the fees require a 12-week notification window via `planFeeChange`. However, `proposedFees` is zero-initialized which means that the timestamp that `executeFeeChange` compares against is zero as well which allows the owner to immediately commit a fee change (zero fees only) on a zero-initialized `proposedFees` struct.

(a) Right after the first deployment and (b) after every fee change execution the owner can reset fees to zero without having to plan a fee change. This may violate the guarantee the contract ought to give that an owner can only change fees after the notification period passed.

Examples

- `proposedFees` is a zero initialized struct, hence, `proposedFees.time` is zero too.

code/contracts/FeeSettings.sol:L26

```
Fees public proposedFees;
```

- `planFeeChange` is the only function that sets `proposedFees.time` to a non-zero value.

- `executeFeeChange` checks if `currentTime >= proposedFees.time`. This is **always true** if `proposedFees` is zero!

code/contracts/FeeSettings.sol:L56-L73

```
function executeFeeChange() external onlyOwner {
    require(
        block.timestamp >= proposedFees.time,
        "Fee change must be executed after the change time"
    );
    tokenFeeDenominator = proposedFees.tokenFeeDenominator;
    continuousFundraisingFeeDenominator = proposedFees
        .continuousFundraisingFeeDenominator;
    personalInviteFeeDenominator = proposedFees
        .personalInviteFeeDenominator;
    emit SetFeeDenominators(
        tokenFeeDenominator,
        continuousFundraisingFeeDenominator,
        personalInviteFeeDenominator
    );
    delete proposedFees;
}
```

Recommendation

Either clearly state that fee reductions to zero can happen at any time or add the missing check to `executeFeeChange()` that requires `proposedFees.time > 0`.

4.5 `ContinuousFundraising` – Accidental use of `msg.sender` instead of `_msgSender()` Medium ✓ Fixed

Resolution

This has been fixed in v2.4 by replacing `msg.sender` with `_msgSender()`.

Description

In order to be compatible with ERC-2771 meta transactions, the `Token` and `ContinuousFundraising` contracts inherit from OpenZeppelin's `ERC2771Context` and use `_msgSender()` to obtain the logical sender of the transaction. In one place, however, `msg.sender` is used accidentally instead of `_msgSender()`:

code/contracts/ContinuousFundraising.sol:L115-L118

```
require(
    tokensBought[msg.sender] + _amount >= minAmountPerBuyer,
    "Buyer needs to buy at least minAmount"
);
```

While the two are the same for regular transactions, in a meta transaction context, `msg.sender` is the trusted forwarder and not the logical sender of the transaction. The mistake interferes with the intended business logic of the `ContinousFundraising` contract. More specifically, it affects the requirement that a user, when they first buy tokens, must purchase a certain minimum amount:

- In the weird scenario that the trusted forwarder contract itself has bought tokens, other users would be able to purchase less than the required minimum amount.
- In the more typical case that the trusted forwarder does not engage in buying activities, users would be forced to buy the minimum amount not only the first time, but *every time* they buy with a meta transaction.

Recommendation

Replace `msg.sender` with `_msgSender()`. It might also be beneficial to set up some mechanism (e.g., a git hook) that automatically checks for occurrences of `msg.sender` and `msg.data` in contracts that inherit from `ERC2771Context`. That helps to catch this kind of bug early in the development process.

4.6 Inconsistent declaration of return values and missing event Minor ✓ Fixed

Resolution

The client followed our recommendation and adopted a consistent interface, dropping the boolean return values and reverting in case of error. The missing event was also added.

Description

`ContinuousFundraising.buy()` returns `bool` while other functions in the contract don’t and the contract reverts on errors or else always returns `true` which suggests that the `bool` return value can be omitted. `buy()` is not called from within the contract system. Also, `buy()` does not emit any event in case of success while `PersonalInvite` emits a `Deal` event on success. It could be useful to emit an event to keep a log of the tokens sold.

code/contracts/ContinuousFundraising.sol:L161-L164

```
        require(token.mint(_msgSender(), _amount), "Minting new tokens failed");
        return true;
    }
```

`Token.mint()` returns `bool true` on success and reverts on error. The subcall to `_mint()` does not return a value but reverts or error. `mint()` is the only function that returns a `bool` on success.

code/contracts/Token.sol:L232-L235

```
            );
        }
        return true;
    }
```

`mint()` is called from within the contract system, however, if `token.mint()` fails the following `require(..)` is never evaluated because the revert in `token.mint()` reverts the parent call too.

code/contracts/ContinuousFundraising.sol:L162

```
require(token.mint(_msgSender(), _amount), "Minting new tokens failed");
```

Recommendation

Consider declaring a consistent interface that either returns `bool true|false` or reverts on error. Consider emitting an event in case `ContinousFundraising.buy()` returns successfully.

4.7 Token - consider implementing `increase/decreaseMintingAllowance` instead of the `safeApprove` pattern to prevent frontrunning Minor ✓ Fixed

Resolution

The functions `increaseMintingAllowance` and `decreaseMintingAllowance` have been implemented as recommended.

Description

`Token.setMintingAllowance` implements the `SafeERC.safeApprove` pattern. This pattern mitigates the frontrunning vector if the approver ...

- resets the approval by changing it from `N -> 0` first
- **checks that the approval went through without the approved account front-running the reset**
- sets the new approval amount from `0 -> N` verifying that the current approval is indeed zero

and everything is done in separate transactions that are not included in the same block (actually within a reorg safe distance; note with MEV you can theoretically buy multiple blocks).

The problem with this pattern is that it is not implicitly secure and always requires two individual transactions and a check if the reset approval went through. In practice, it is likely that it does not even prevent the frontrunning vector because the approver fails to check or wait for the new approval.

This is also addressed in https://github.com/OpenZeppelin/openzeppelin-contracts/issues/2219 where it is suggested to use the atomic `safe[Decrease|Increase]Allowance` pattern instead.

> A token owner just needs to make sure that the first transaction actually changed allowance from N to 0, i.e. that the spender didn't manage to transfer some of N allowed tokens before the first transaction was mined.

original description of the frontrunning vector

Examples

- original `safeApprove` pattern

contracts/token/ERC20/utils/SafeERC20.sol:L55-L58

```
    (value == 0) || (token.allowance(address(this), spender) == 0),
    "SafeERC20: approve from non-zero to non-zero allowance"
);
_callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, sp
```

- current implementation

code/contracts/Token.sol:L208-L219

```
function setMintingAllowance(
    address _minter,
    uint256 _allowance
) external onlyRole(MINTALLOWER_ROLE) {
    require(
        mintingAllowance[_minter] == 0 || _allowance == 0,
        "Set up minter can only be called if the remaining allowance is 0 or
    ); // to prevent frontrunning when setting a new allowance, see https://w
    mintingAllowance[_minter] = _allowance;
    emit MintingAllowanceChanged(_minter, _allowance);
}
```

Recommendation

Consider implementing the `safeIncreaseAllowance` and `safeDecreaseAllowance` pattern or give specific guidance on how to safely use `setMintingAllowance`.

4.8 Lock Solidity Version in Pragma Minor ✓ Fixed

Resolution

All contracts now use the version pragma `pragma solidity 0.8.17;`.

Description

Contracts should be deployed with the same compiler version they have been tested with. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs. Contracts may also be deployed by others and the pragma indicates the compiler version intended by the original authors.

See Locking Pragmas in Ethereum Smart Contract Best Practices.

Examples

code/contracts/AllowList.sol:L2

```
pragma solidity ^0.8.0;
```

code/contracts/ContinuousFundraising.sol:L2

```
pragma solidity ^0.8.0;
```

code/contracts/FeeSettings.sol:L2

```
pragma solidity ^0.8.0;
```

Recommendation

Lock the Solidity version to the latest version before deploying the contracts to production.

```
pragma solidity 0.8.17;
```

4.9 Transferring Ownership is Risky with OpenZeppelin’s `Ownable` Library Minor ✓ Fixed

Resolution

The client followed the recommendation and adopted `Ownable2Step`.

In the NatSpec comments for `_msgSender()` and `_msgData()` in `ContinuousFundraising.sol`, `Ownable` should be replaced with `Context`.

Description

OpenZeppelin’s `Ownable` library uses a single-step process when transferring ownership. This is risky because if there is a mistake in the new owner address there is no way to recover from a state in which the contract has no functional owner or a malicious one.

Examples

code/contracts/AllowList.sol:L11

```
contract AllowList is Ownable {
```

code/contracts/ContinuousFundraising.sol:L24-L26

```
contract ContinuousFundraising is
    ERC2771Context,
    Ownable,
```

code/contracts/FeeSettings.sol:L16

```
contract FeeSettings is Ownable {
```

Recommendation

We recommend using a two-step pattern (such as that implemented by OpenZeppelin's Ownable2Step) in which a new owner is proposed by the current one and the pending owner is required to claim the new ownership.

4.10 Mistakes in the high-level documentation Partially Addressed

Resolution

The client fixed the issues. Note: As the pricing calculation changed, the related section in `using_the_contracts.md` still needs to be updated accordingly. Similarly, passages dealing with minting allowance should be updated to reflect recent changes.

Description

The high-level documentation contains small mistakes in a few places:

- `_amount` does not represent the total amount of shares in existence. It should maybe be `_allowance` from `setMintingAllowance(address minter, uint _allowance)` ?

code/docs/using_the_contracts.md?plain=1:L34-L35

```
To create the initial cap table, `_amount` should be the total amount of sha
The minter can then create new shares for each shareholder, by calling `mint
```

- The code extract showing the constructor of `PersonalInvite` does not reflect its actual constructor.

code/docs/using_the_contracts.md?plain=1:L63-L65

~~~
```solidity
constructor(address payable _buyer, address payable _receiver, uint _minAmou
~~~

~~~
- Similarly, the code extract showing the constructor of `ContinousFundraising.sol` omits

**code/docs/using_the_contracts.md?plain=1:L98**
```solidity
Constructor: `constructor(address payable _currencyReceiver, uint _minAmountPerBuyer, uint
~~~

- There is a mistake in the documentation explaining the price computation: `price = PaymentTokenBits/Token = PaymentTokenBits/TokenBits * Token.decimals()` should be `price = PaymentTokenBits/Token = PaymentTokenBits/TokenBits * 10**Token.decimals()`

code/docs/price.md?plain=1:L24-L26

~~~
```solidity
price = PaymentTokenBits/Token = PaymentTokenBits/TokenBits * Token.decimals
~~~

```
#### Recommendation

Fix the documentation to reflect the code.
```

4.11 Gas - Cache results of static external calls locally instead of performing the same external call multiple times

✓ Fixed

Resolution

This has been improved as recommended.

Description

Consider caching return values for external calls for static getters instead of calling the external entity multiple times to save gas.

- `token.feeSettings().continuousFundraisingFeeDenominator()` - 2 external calls in `if` condition and the same 2 external calls in the `else` branch`.

code/contracts/ContinuousFundraising.sol:L143

```
if (token.feeSettings().continuousFundraisingFeeDenominator() == 0) {
```

This can be optimized to cache the result of `token.feeSettings` and/or `token.feeSettings().continuousFundraisingFeeDenominator()` to a local stack variable instead of performing multiple external calls.

- similar case with `decimals()`

code/contracts/PersonalInvite.sol:L62-L68

```
require(
    (_amount * _tokenPrice) % (10 ** _token.decimals()) == 0,
    "Amount * tokenprice needs to be a multiple of 10**token.decimals()"
);
uint256 currencyAmount = (_amount * _tokenPrice) /
    (10 ** _token.decimals());
```

There may be more instances of this pattern.

4.12 use `_grantRole` instead of deprecated `_setupRole` in `Token.sol` ✓ Fixed

Resolution

This has been fixed as recommended.

Description

The `Token.sol` constructor uses the function `_setupRole(DEFAULT_ADMIN_ROLE, _admin)` to grant the `DEFAULT_ADMIN_ROLE` permissions to `_admin`. However, this function was deprecated by OpenZeppelin in favor of `_grantRole`.

code/contracts/Token.sol:L114

```
_setupRole(DEFAULT_ADMIN_ROLE, _admin); // except for the Transferer role, t
```

Recommendation

We recommend using `_grantRole` instead of `_setupRole` for coherence and forward compatibility of the codebase.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash v2.4	SHA-1 hash v3.0
code/contracts/AllowList.sol	`8f6bfc840b41cca2cfbb19a828bd253b6daa1e84`	`ff848099a65e78ff70e6f339c6ec679d613da53d`
code/contracts/ContinuousFundraising.sol	`87833840250123f5e13ab77dd35f6c62dfca8b85`	`364c09e02727ea40256836e126977b8a64be28ca`
code/contracts/FeeSettings.sol	`2dd924248ee79c95ee3c94ceb10936f1733cc9ce`	`2fcda503e52945847952acba827e252069521d47`
code/contracts/PersonalInvite.sol	`945e711f22ac990edfedff936a4c31ac49022937`	`d0fa1db779dfc7c99b29c2fcbb753144b9be5a12`
code/contracts/PersonalInviteFactory.sol	`8aec0f9d09f89d4f7a90be0b27c4cc880f3140a8`	`12d8a039ad8d2c6181c9e65547e52a792663635e`
code/contracts/Token.sol	`f55d0385b0f49641366f287057962a26f02dae54`	`2a89c470ee43862938281ab790c1384ce6e24ff8`

Appendix 2 - Document Change Log

Version	Date	Description
1.0	2022-12-06	Initial report
2.0	2022-12-16	Review of fixes

Appendix 3 - Disclosure

ConsenSys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.

CONTACT US

AUDITS

FUZZING

SCRIBBLE

BLOG

TOOLS

RESEARCH

ABOUT

CONTACT

CAREERS

PRIVACY POLICY

Subscribe to Our Newsletter

Stay up-to-date on our latest offerings, tools, and the world of blockchain security.

POWERED BY CONSENSYS