Security Review Report
NM-0163 TOKENTABLE

NETHERMIND
SECURITY

(Feb 27, 2024)

Contents

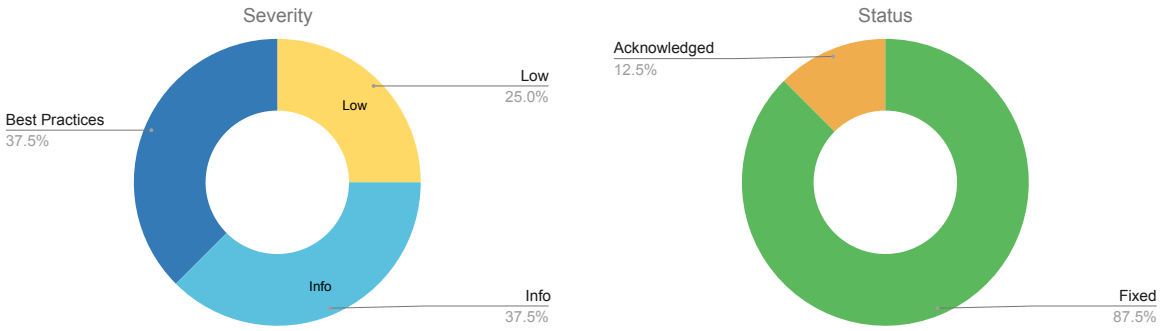
1 Executive Summary

This document outlines the security review conducted by Nethermind Security for the TokenTable. `TokenTable` is a token management platform for web3 founders and token holders. It provides advanced tools that streamline the operations involved in securely unlocking and distributing tokens. The system allows the founder to set up his schedule to distribute the tokens. Every founder can create their token streaming smart contract, unlocker, via deployer and send all project tokens for distribution to the founder's instance of unlocker.

The audited code comprises 1599 lines of code in Cairo. The `TokenTable` team has provided comprehensive documentation that explains TokenTable's architecture and the detailed properties of each component. Furthermore, the documentation specifies the actors interacting with the protocol and standard workflow, which helps understand the protocol.

The audit was performed using: (a) manual analysis of the codebase, (b) simulation of the smart contracts and (c) automation analysis of the smart contracts. **Along this document, we report** 16 points of attention, where four are classified as `Low`, and twelve are classified as `Informational` or `Best Practice`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.



(a) distribution of issues according to the severity

(b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (4), **Undetermined** (0), **Informational** (6), **Best Practices** (6). **(b) Distribution of status: Fixed** (14), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Feb 9, 2024
Final Report	Feb 27, 2024
Methods	Manual Review, Automated analysis
Repository	tokentable-v2-starknet
Commit Hash	f7253f2a9c4f52c76affef54031d3a50672d8e13
Final Commit Hash	95172d6471587abc8c751397b9008cfa480800de
Documentation	TokenTable docs v2.5
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/unlocker.cairo	824	18	2.2%	43	885
2	src/feecollector.cairo	91	12	13.2%	12	115
3	src/deployer.cairo	177	12	6.8%	12	201
4	src/components.cairo	4	0	0.0%	0	4
5	src/lib.cairo	6	0	0.0%	0	6
6	src/futuretoken.cairo	169	24	14.2%	14	207
7	src/components/interfaces.cairo	7	0	0.0%	0	7
8	src/components/structs.cairo	3	0	0.0%	0	3
9	src/components/structs/actual.cairo	6	12	200.0%	1	19
10	src/components/structs/ttsuite.cairo	5	5	100.0%	2	12
11	src/components/structs/preset.cairo	8	15	187.5%	2	25
12	src/components/interfaces/unlocker.cairo	171	212	124.0%	37	420
13	src/components/interfaces/feecollector.cairo	34	36	105.9%	8	78
14	src/components/interfaces/deployer.cairo	52	53	101.9%	9	114
15	src/components/interfaces/hook.cairo	9	12	133.3%	2	23
16	src/components/interfaces/versionable.cairo	3	6	200.0%	1	10
17	src/components/interfaces/futuretoken.cairo	30	46	153.3%	8	84
	Total	**1599**	**463**	**29.0%**	**151**	**2213**

3 Summary of Issues

	Finding	Severity	Update
1	ERC777 reentrancy allows the founder to claim the same actual twice	Low	Fixed
2	Future token is starting with `token_id = 2`	Low	Fixed
3	Rounding to zero when calculating the linear interval for each unlock could cause a DOS and lock the funds indefinitely	Low	Fixed
4	Rounding when calculating the linear interval for each unlock can allow a recipient to claim more than intended	Low	Fixed
5	Calling hook during hook disabling	Info	Fixed
6	Duplicate URI variable in FutureToken contract	Info	Fixed
7	Risk of DOS in the function `deploy_ttsuite(...)` due to front-running with the same `project_id`	Info	Acknowledged
8	Stream mode calculation could be simplified and made more accurate	Info	Fixed
9	Users can input random `recipient_id` values, which could potentially disrupt the off-chain front-end	Info	Acknowledged
10	`_charge_fees(...)` should not revert when the fee collector is not set	Info	Fixed
11	Documentation lacking for setting `fee_bips = 0` when it equals `BIPS_PRECISION`	Best Practices	Fixed
12	Duplication of code	Best Practices	Fixed
13	No check of the return value from `transfer(...)` function	Best Practices	Fixed
14	No limit when setting fees	Best Practices	Fixed
15	Unused code	Best Practices	Fixed
16	Use `_safe_mint(...)` instead of `mint(...)` to make sure that the receiver can obtain NFT.	Best Practices	Fixed

4 System Overview

TokenTable is a token management platform for web3 founders and token holders. It enables users to streamline operations involving the secure unlocking and distribution of tokens, such as token vesting to investors, monthly payroll to employees, and airdrop distribution to participants. **TokenTable** provides a trustless execution environment for founders to programmatically execute token unlocks and claims, mitigating human error and third-party custodial risks. Furthermore, the **TokenTable** smart contracts allow investors and token holders access to real-time and publicly verifiable information about the cap table. Token claimers can also easily track and export data about unlock schedules, claims, and secondary sale information, etc.

Fig. 2 presents the interaction diagram of the contracts.

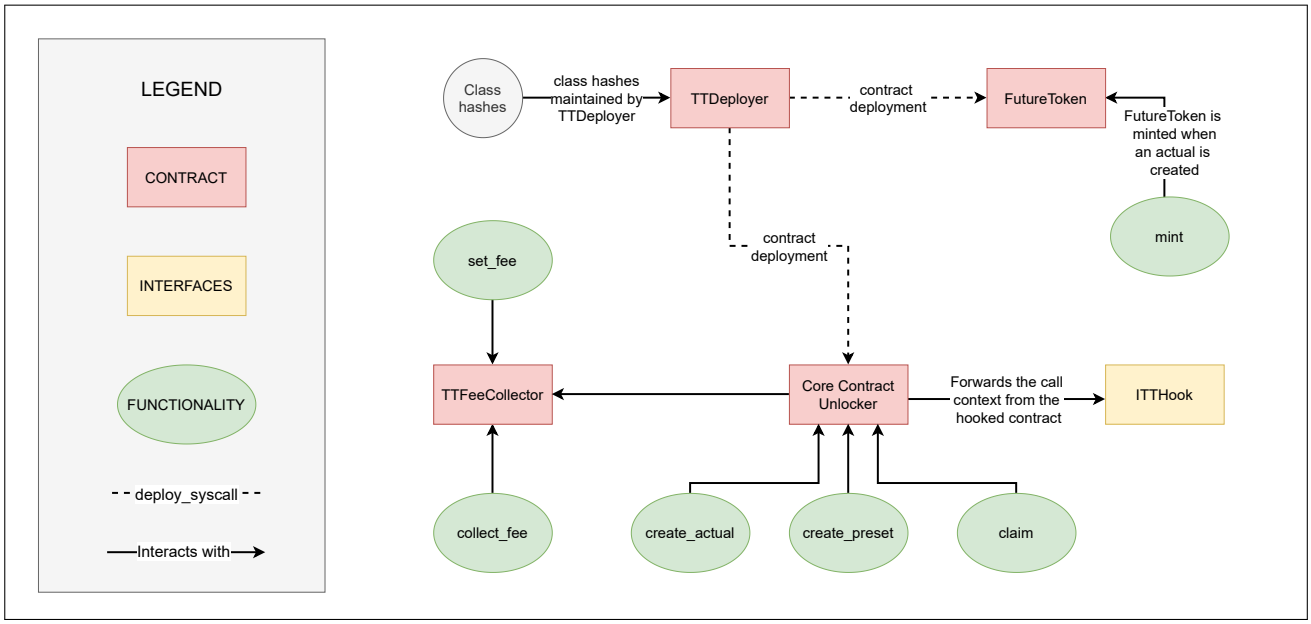


Fig. 2: Interaction Diagram of Contracts.

To use **TokenTable**, the founder will use the **TTDeployer** to deploy an instance of a `TTSuite`. Each suite comprises an **Unlocker** contract and a corresponding **FutureToken** contract. The founder's address will be registered as the owner of the **Unlocker** contract. Using this address, the founder can execute owner-restricted functions such as:

a) Creating **Presets** and **Actuals**

b) Depositing and withdrawing tokens

c) Canceling an existing schedule

A **Preset** is a set of configurations providing vesting details to one or more **Actuals**. It contains information such as the number of unlocks, the timestamp of each unlock, the number of tokens per unlock, and the end timestamp.

An **Actual** is a set of configurations specific to each recipient. It contains information such as the preset ID to use, the start timestamp, how many tokens have been claimed by the recipient, and the total amount distributed.

To set up a token vesting schedule for a user, the founder will use the **Unlocker** contract to create a **Preset**, each uniquely identified by a `preset_id`. This ID is required for creating an **Actual**.

Next, the founder will create an **Actual** for the recipient. The **Unlocker** contract will invoke the `mint(...)` function on the **FutureToken** contract to mint a **FutureToken** NFT to the recipient. This NFT represents ownership of the **Actual** and can be transferred to a new recipient if required.

Lastly, the founder will fund the **Actual** by transferring the tokens into the **Unlocker** contract. Once some time has passed, the owner of the NFT is authorized to call the `claim(...)` function to claim any vested tokens.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely an attacker will uncover and exploit the finding. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Low] ERC777 reentrancy allows the founder to claim the same actual twice

File(s): unlocker.cairo

Description: The function `_update_actual_and_send(...)` transfers tokens and handles accounting when users claim an asset. However, it doesn't follow the CEI pattern, as it executes the token transfer (an external call) before updating the storage. If the token is ERC777, which includes a call hook on the receiver's address, it may create a reentrancy vulnerability. Here's a potential scenario:

1. The token used in this case is ERC777, which makes a call hook to the receiver after a transfer;
2. When claiming an actual A, the `recipient` is set to the founder address. After receiving the call hook from the token, it calls back to the function `cancel(...)` with `wipe_claimable_balance = false`;
3. Because actual A does not get updated during the claim process (as this happens after the token transfer), the claimable amount is the same as before claiming and is written to `pending_amount_claimable_for_cancelled_actuals`;
4. Claim actual A again. As a result, the founder successfully claims the same actual twice;

```
fn cancel(
    ...
) -> u256 {
    ...
    let (pending_amount_claimable, _) =
        self.calculate_amount_claimable(actual_id);
    ...
    if !wipe_claimable_balance {
        self.pending_amount_claimable_for_cancelled_actuals.write(
            actual_id,
            pending_amount_claimable
        );
    }
    ...
}

...
fn _update_actual_and_send(
    ...
) -> u256 {
    ...
    // @audit Token transfer is done before writing to storage.
    //        If the token is ERC777 with a hook to the receiver, the founder can reentrant to cancel the actual.
    self._send(recipient, delta_amount_claimable);
    actual.amount_claimed = updated_amount_claimed;
    self.actuals.write(actual_id, actual);
    delta_amount_claimable
}
```

Recommendation(s): Consider following the CEI (Checks-Effects-Interactions) pattern by updating the storage before making any external calls.

Status: Fixed

Update from the client: Resolved with 84e03e4074688651e604abf19914df8fca0845a7 and 95092b9545d61707685b39b6992420a08216851c.

6.2 [Low] Future token is starting with `token_id = 2`

File(s): `futuretoken.cairo`

Description: The minting function for future tokens only requires the `recipient` of the NFT as a parameter. The `token_id` is generated based on the `token_counter` value.

The internal function `_increment_token_counter_and_return_new_value` is used to obtain `token_id`. This function increases the `token_counter` value by one and assigns this new value as `token_id`. However, when the future token contract is deployed, `token_counter` is set to one. This will cause the first minted token to have a `token_id` of 2.

Recommendation(s): Avoid setting the counter to one when deploying the future token.

Status: Fixed

Update from the client: Resolved with `632c13f770d11c959e3aaa5a77f27ba790625e69`

6.3 [Low] Rounding to zero when calculating the linear interval for each unlock could cause a DOS and lock the funds indefinitely

File(s): `unlocker.cairo`

Description: The function `simulate_amount_claimable(...)` calculates the claimable amount at the current timestamp. For the latest incomplete segment, it first calculates the interval length of each unlock and then uses this to determine how many unlocks can be claimed.

```
// @audit Might round down to 0 and cause revert later
let latest_incomplete_linear_interval_for_each_unlock =
    latest_incomplete_linear_duration /
        *preset_num_of_unlocks_for_each_linear.at(
            latest_incomplete_linear_index
        );
...

// @audit Might revert here because of divided by zero
let num_of_claimable_unlocks_in_incomplete_linear =
    latest_incomplete_linear_claimable_timestamp_relative *
    time_precision_decimals /
        latest_incomplete_linear_interval_for_each_unlock;
```

As illustrated in the code snippet, if `latest_incomplete_linear_duration < num_of_unlocks`, the result `latest_incomplete_linear_interval_for_each_unlock` will round down to `0`. Since this variable is used as a divisor later, it could cause the function to revert due to division by zero. It's also important to note that if the latest incomplete segment is the last segment, this could potentially result in a permanent DOS. Furthermore, the actual cannot be canceled because the function `cancel(...)` also uses `simulate_amount_claimable(...)`.

Recommendation(s): Consider adding validation when creating the actual to prevent this scenario.

Status: Fixed

Update from the client: Resolved with `4b175626d1c8e8d889752bf4d0cc0ebf16840e1a`.

6.4 [Low] Rounding when calculating the linear interval for each unlock can allow a recipient to claim more than intended

File(s): unlocker.cairo

Description: The function `simulate_amount_claimable(...)` calculates the claimable amount at the current timestamp. For the latest incomplete segment, it first calculates the interval length of each unlock and then uses this to determine how many unlocks can be claimed.

```
// @audit Rounding down causes interval for each unlock to be shorter than intended.
let latest_incomplete_linear_interval_for_each_unlock =
    latest_incomplete_linear_duration /
        *preset_num_of_unlocks_for_each_linear.at(
            latest_incomplete_linear_index
        );
...

// @audit Num of claimable unlocks is higher than intended because of rounding.
let num_of_claimable_unlocks_in_incomplete_linear =
    latest_incomplete_linear_claimable_timestamp_relative *
    time_precision_decimals /
        latest_incomplete_linear_interval_for_each_unlock;

// @audit As a result, updated amount claimed is also higher than expected.
updated_amount_claimed +=
    (*preset_linear_bips.at(latest_incomplete_linear_index)).into()
        *
        TOKEN_PRECISION_DECIMALS *
        num_of_claimable_unlocks_in_incomplete_linear.into() /
    (*preset_num_of_unlocks_for_each_linear.at(
        latest_incomplete_linear_index
    )).into() / time_precision_decimals.into();
```

As highlighted in the code snippet above, `num_of_claimable_unlocks_in_incomplete_linear` will be higher than intended because of the rounding in its calculation, which will in turn skew the value of `updated_amount_claimed`. Let's go through a numerical example to illustrate this. Suppose we have the following:

```
linearStartTimestampRelative: [0, 100]
linearEndtimeStampRelative: 200
linearBps: [0, 5000, 5000]
numOfUnlocksForEachLinear: [51, 51]
currentTimestamp = 99
```

The correct value for `num_of_claimable_unlocks_in_incomplete_linear` should be `100 / 51 = 1.96`, but because the division rounds down, `num_of_claimable_unlocks_in_incomplete_linear` will round down to 1. As such, `num_of_claimable_unlocks_in_incomplete_linear` has a much higher value of `99 * time_precision_decimals`. Finally, when `updated_amount_claimed` is calculated, it will have the value `5000 * token_precision_decimals * 99 / 51`, i.e. almost all tokens are unlocked at the first claim instead of half.

It is important to note that these values are specifically chosen to illustrate the worst-case scenario of the impact of the rounding. In the event that we have more linears e.g. 20% unlocked each year for five years or if the rounding has a smaller impact e.g. rounding down to 99 instead of 100, the impact will be significantly reduced.

Recommendation(s): Multiple `latest_incomplete_linear_duration` by a constant factor (similar to `token_precision_decimals` and `time_precision_decimals`) and divide it at the end in order to handle the rounding.

Status: Fixed

Update from the client: Replicated and resolved with `7f73762f0b99b57406786a8e4a362996176396da`

6.5 [Info] Calling hook during hook disabling

File(s): unlocker.cairo

Description: In the unlocker, a hook can be defined and called after every user's action. It enables external logic like verification or tracking on the hook contract. As it is invoked in every function, any issue with the hook contract could impact all unlocker functionalities. For instance, if the `did_call(...)` function always reverts, all functions in the unlocker will also revert. Therefore, the owner can disable the hook by calling `disable_hook(...)`. However, this disable function still calls the hook. So, if the function `did_call(...)` reverts, then `disable_hook(...)` will also revert.

```
fn disable_hook(
    ref self: ContractState
) {
    self.ownable.assert_only_owner();
    self.is_hookable.write(false);
    self._call_hook_if_defined(
        'disable_hook',
        array![
            get_caller_address().into()
        ].span()
    );
    self.hook.write(ITTHookDispatcher {
        contract_address: Zeroable::zero()
    });
    self.emit(Event::HookDisabled(TTUnlockerEvents::HookDisabled{}));
}
```

Recommendation(s): Consider eliminating the hook call in the function `disable_hook(...)`.

Status: Fixed

Update from the client: Resolved with d71b1c33dac56ee343ddc2800d6ca65b52fffac6.

6.6 [Info] Duplicate URI variable in FutureToken contract

File(s): futuretoken.cairo, custom_erc721.cairo

Description: In the `futuretoken` contract, the authorized minter can call `set_uri()` to update the `base_uri` storage variable. However, since the `futuretoken` contract also uses the `custom_erc721` component, the `base_uri` variable is unnecessary because the `custom_erc721` component also contains a `ERC721_token_base_uri` variable.

Recommendation(s): Consider removing the `base_uri` variable and `get_base_uri()` function and updating the `set_uri()` functions to call the internal `_set_token_uri()` function.

Status: Fixed

Update from the client: Resolved with 004238ef4ba04c647fda87497384a4fdd63e1734.

6.7 [Info] Risk of DOS in the function `deploy_ttsuite(...)` due to front-running with the same `project_id`

File(s): futuretoken.cairo

Description: In the function `deploy_ttsuite(...)`, the founder must input a `project_id`. This `project_id` is used to check whether the project already exists before proceeding with the deployment.

However, since the `project_id` value can be arbitrary, there is a risk of DOS when an attacker repeatedly calls `deploy_ttsuite(...)` using the same `project_id` as a legitimate founder's transaction. This could cause the legitimate founder's transaction to revert because the `project_id` already existed when it was executed.

Recommendation(s): Consider tracking the project's existence using both the `project_id` and the sender's address.

Status: Acknowledged

Update from the client: We will keep the original implementation since we would like to keep all IDs in the same namespace to enforce uniqueness and decrease possible ambiguity.

6.8 [Info] Stream mode calculation could be simplified and made more accurate

File(s): unlocker.cairo

Description: The current calculation for stream mode sets `time_precision_decimals = 1e5`. However, accuracy could be improved by setting the number of unlocks to the segment's duration in seconds. This change eliminates the need for a new `time_precision_decimals` variable and achieves 100% precision, as every second becomes an unlock.

```
if preset_stream {
    time_precision_decimals = 100000;
}
...
let num_of_claimable_unlocks_in_incomplete_linear =
    latest_incomplete_linear_claimable_timestamp_relative *
    time_precision_decimals /
        latest_incomplete_linear_interval_for_each_unlock;
```

If `latest_incomplete_linear_claimable_timestamp_relative * time_precision_decimals < latest_incomplete_linear_interval_for_each_unlock`, the result will round down to 0.

Recommendation(s): Consider changing the calculation to use the duration in seconds as the number of unlocks when `preset_stream = true`.

Status: Fixed

Update from the client: Resolved with 6b544491bd49cf25df6f38ee6eb5efb0f0e32df9.

Update from Nethermind: It is necessary to replace the use of `preset_num_of_unlocks_for_each_linear` with the calculated `num_of_unlocks` in the whole function(mainly when calculating `updated_amount_claimed`).

Update from the client: Resolved with 12e739cc676c0eff008a12df8d60858eea2c164f

6.9 [Info] Users can input random `recipient_id` values, which could potentially disrupt the off-chain front-end

File(s): unlocker.cairo

Description: All external user functions in `unlocker.cairo` accept an extra `recipient_id` parameter. This parameter isn't used in any logic and is only intended to emit an event for off-chain tracking. However, because callers set the value, users can input any `recipient_id` value, which can potentially cause disruptions off-chain. Nonetheless, it doesn't impact the contract level since the functionality remains operational.

Recommendation(s): Consider deriving `recipient_id` from the caller address.

Status: Acknowledged

Update from the client: This is not a concern as only backend-initiated transactions will be monitored for their `recipient_id`.

6.10 [Info] `_charge_fees(...)` should not revert when the fee collector is not set

File(s): unlocker.cairo

Description: During a call of function `_claim(...)`, the fees are calculated based on the number of tokens claimed by the user.

The fees are calculated at `_charge_fees(...)` by asking for fees from the fee collector. The Fee collector's address is obtained from the deployer. However, the code does not handle a case when the fee collector is not set in the deployer. The `claim(...)` function will revert if the fee collector is not set.

Note: This check is present in the solidity version of the code.

Recommendation(s): Consider additional checks that the fee collector is not zero when asking for fees.

Status: Fixed

Update from the client: Resolved with 627c9a222a6a98f17157b7a53732eca369b50b44.

6.11 [Best Practices] Documentation lacking for setting `fee_bips = 0` when it equals `BIPS_PRECISION`

File(s): `feecollector.cairo`

Description: In the `get_fee(...)` function, there's a unique logic for a custom fee for an unlocker instance. If `fee_bips = BIPS_PRECISION`, then `fee_bips = 0`. This is because the function treats `fee_bips = 0` as an unset custom fee, and so uses the default fee. However, this is confusing and lacks explanatory documentation.

```
if fee_bips == 0 {
    fee_bips = self.default_fee_bips.read();
} else if fee_bips == BIPS_PRECISION { // @audit No documentation
    fee_bips = 0;
}
```

Recommendation(s): Consider adding documentation to explain this behavior in the `get_fee(...)` function.

Status: Fixed

Update from the client: Resolved with `b826b85af292055b4f594441964da1314ae57cb2`.

6.12 [Best Practices] Duplication of code

File(s): `unlocker.cairo`

Description: In the code, occurrences of duplicated functionality exist, which can be replaced with already created functions.

That is the case of `_charge_fees(...)` function and the following code block:

```
let result = self.project_token.read().transfer(
    fee_collector_address,
    fees_collected
);
assert(
    result,
    TTUnlockerErrors::GENERIC_ERC20_TRANSFER_ERROR
);
```

Another example is in the `withdraw_deposit(...)` function:

```
let result = self.project_token.read().transfer(get_caller_address(), amount);
assert(result, TTUnlockerErrors::GENERIC_ERC20_TRANSFER_ERROR);
```

This code can be replaced by the internal function `_send(...)`, which sends project tokens to a given recipient.

Recommendation(s): Consider replacing duplicated code with defined functions.

Status: Fixed

Update from the client: Resolved with `8d8866cdfde968a8ce8a7d07406b44ba87195dc0`.

6.13 [Best Practices] No check of the return value from `transfer(...)` function

File(s): `feecollector.cairo`

Description: When performing an erc20 **transfer**, it is good to practice checking the boolean value that this function returns.

```
fn withdraw_fee(ref self: ContractState, token: ContractAddress, amount: u256) {
    self.ownable.assert_only_owner();
    IERC20Dispatcher { contract_address: token }.transfer(self.ownable.owner(), amount);
}
```

If it does not return true, the transfer can indicate a failure.

Recommendation(s): Consider implement check for boolean return value of `transfer(...)` function.

Status: Fixed

Update from the client: Resolved with `7e94180d9433a1e5c9149cb19d1334b97c0d0674`.

6.14 [Best Practices] No limit when setting fees

File(s): `feecollector.cairo`

Description: Fees can be set in two ways: a default value for every unlocker using the `set_default_fee(...)` function or a custom fee for a specific unlocker using `set_custom_fee(...)`. Both functions are implemented in `feecollector.cairo`.

However, these functions do not have a fee limit. They can set the fee higher than `BIPS_PRECISION`, which would exceed 100%. Establishing specific limits assures the customer they won't be charged an exorbitant fee if an attacker compromises the fee collector.

Recommendation(s): Consider implementing an upper limit for the fees.

Status: Fixed

Update from the client: Resolved with `a1b85e9fb39fe02dc0db56f218371fb808905dd0`.

6.15 [Best Practices] Unused code

File(s): `unlocker.cairo`

Description: In the code are occurrences of unused code:

- `preset_bips_precision` argument of `simulate_amount_claimable(...)` function;

Removing unused variables and functions to keep code clean is good practice.

Recommendation(s): Consider removal of unused code.

Status: Fixed

Update from the client: Resolved with `e5ea072d8e6923c0b58ea037e9423bbfbe0370a5`.

6.16 [Best Practices] Use `_safe_mint(...)` instead of `mint(...)` to make sure that the receiver can obtain NFT.

File(s): `futuretoken.cairo`

Description: The future token NFT implements public mint functionality, which can be called just by the authorized minter. In the protocol case, the authorized minter is just the Unlocker.

The receiver of NFT can be anyone, depending on the recipient address, in the `create_actual(...)` function in `unlocker.cairo`. However, there is no check if the recipient is aware of receiving NFTs, and it can be a problem if the recipient is a custom contract. This can be ensured by using the `_safe_mint(...)` internal functionality of ERC721, as it calls callback to check that the contract is ready for receiving NFTs.

Recommendation(s): Consider replacing `self.erc721._mint(...)` function with `self.erc721._safe_mint(...)` in the `mint(...)` function at `futuretoken.cairo`.

Status: Fixed

Update from the client: Resolved with `36bc4a3808f2b170503fba3083d48bd1ee971d12`.
In production, we came across some older multisig and 2/3FA accounts that do not support SRC-6, which fails `safe_mint(...)`. However, after communicating with the respective vendors, they have assured us despite incompatibility with SRC-6, their wallets do support handling NFTs. Thus, we have added an extra parameter to `create_actual(...)` named `unsafe_mint: bool` that uses `mint(...)` instead of `safe_mint(...)` when set to true. Commit: `95172d6471587abc8c751397b9008cfa480800de`.

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about TokenTable documentation

The `TokenTable` team has provided documentation about their protocol in the codebase comments and through their official documentation - TokenTable docs v2.5. In particular, the token streaming example helped to understand the overall concept. Moreover, the TokenTable team was available to address any inquiries or concerns raised by the Nethermind auditors.

8 Test Suite Evaluation

8.1 Compilation Output

```
> scarb build
Compiling tokentable_v2 v2.5.0 (/home/audits/tokentable-v2-starknet/Scarb.toml)
    Finished release target(s) in 6 seconds
```

8.2 Tests Output

```
> snforge test
   Compiling tokentable_v2 v2.5.0 (/home/audits/tokentable-v2-starknet/Scarb.toml)
    Finished release target(s) in 6 seconds


Collected 11 test(s) from tokentable_v2 package
Running 0 test(s) from src/
Running 11 test(s) from tests/
original value: [4901]
original value: [0]
[PASS] tests::integration_tests::unlocker_simulate_test_edgecase_1 (gas: ~38638)
[PASS] tests::integration_tests::deployer_test (gas: ~36167)
[PASS] tests::integration_tests::unlocker_cancel_test_1 (gas: ~73648)
[PASS] tests::integration_tests::unlocker_withdraw_test (gas: ~74807)
[PASS] tests::integration_tests::unlocker_cancelable_test (gas: ~73579)
[PASS] tests::integration_tests::unlocker_cancel_test_0 (gas: ~72401)
[PASS] tests::integration_tests::unlocker_claimable_calculation_unit_test (gas: ~38695)
[PASS] tests::integration_tests::unlocker_delegate_claim_test (gas: ~77327)
[PASS] tests::integration_tests::unlocker_create_preset_test (gas: ~64452)
[PASS] tests::integration_tests::unlocker_create_actual_test (gas: ~82185)
[PASS] tests::integration_tests::unlocker_claim_test (gas: ~76521)
Tests: 11 passed, 0 failed, 0 skipped, 0 ignored, 0 filtered out
```

Remarks about TokenTable test suite

The `TokenTable` team has created the test suits covering a proper workflow of the interaction with the protocol. They also conducted tests regarding a few edge cases, which can happen when calculating the claimable amount. However, to increase the quality of the tests suite, Nethermind Security recommends implementing fuzz testing for the function `simulate_amount_claimable(...)` to cover more unpredictable scenarios and prevent improper behaviour of this function.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development procehttps://www.overleaf.com/project/65c0e737f41a29601bda5c48ss, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology 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 the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.