Thales Airdrop and Staking Smart Contract Audit

1. Introduction

iosiro was commissioned by Thales DAO to conduct a smart contract audit on their Ongoing Airdrop and Staking contracts. The audit was performed by two consultants between 23 August and 1 September 2021, using 10 resource days. A review of changes was performed intermittently between 7 to 10 September 2021.

This report is organized into the following sections.

- **Section 2 - Executive summary:** A high-level description of the findings of the audit.
- **Section 3 - Audit details:** A description of the scope and methodology of the audit.
- **Section 4 - Design specification:** An outline of the intended functionality of the smart contracts.

- **Section 5 - Detailed findings:** Detailed descriptions of the findings of the audit.

The information in this report should be used to better understand the risk exposure of the smart contracts, and as a guide to improving the security posture of the smart contracts by remediating issues identified. The results of this audit are only a reflection of the source code determined to be in scope and available for review at the time of the audit.

The purpose of this audit was to achieve the following:

- Identify potential security flaws.
- Ensure that the smart contracts functioned according to the documentation provided.

Assessing the off-chain functionality associated with the contracts, for example, backend web application code, was out of scope of this audit.

Due to the unregulated nature and ease of transfer of cryptocurrencies, operations that store or interact with these assets are considered very high risk with regards to cyber attacks. As such, the highest level of security should be observed when interacting with these assets. This requires a forward-thinking approach, which takes into account the new and experimental nature of blockchain technologies. Strategies that should be used to encourage secure code development include:

- Security should be integrated into the development lifecycle and the level of perceived security should not be limited to a single code audit.
- Defensive programming should be employed to account for unforeseen circumstances.
- Current best practices should be followed where possible.

2. Executive summary

The sub-sections below present a high-level overview of the audit performed by iosiro on the Thales Token, Ongoing Airdrop, and Staking contracts.

Audit findings

iosiro found three high-risk, two medium-risk, one low-risk and a number of informational issues. The majority of these issues required minor code changes to remediate. iosiro also assisted Thales in simplifying the implementation of their staking escrow logic.

All findings were addressed before the conclusion of the audit.

Recommendations

At a high level, Thales' security posture could be further strengthened by:

- Performing additional audits at regular intervals, as security best practices, tools, and knowledge change over time. Additional audits over the course of the project's lifespan ensure the longevity of the codebase.
- Promote the use of the Thales Bug Bounty program to encourage the responsible disclosure of security issues identified in the system.

3. Audit details

3.1 Scope

The source code considered in-scope for the assessment is described below. Code from all other files was considered to be out-of-scope. Out-of-scope code that interacts with in-scope code was assumed to function as intended and not introduce any functional or security issues.

3.1.1 Smart contracts

- **Project name:** Thales Ongoing Airdrop and Staking
- **Initial audit commit:** a7a15db
- **Second audit commit:** 69f46f7
- **Final audit commit:** 49685c2
- **Files:** Thales.sol, OngoingAirdrop.sol, VestingEscrow.sol, StakingThales.sol, EscrowThales.sol

3.2 Methodology

A variety of techniques, described below, were used to conduct the audit.

3.2.1 Code review

The source code was manually inspected to identify potential security issues. Code review is a useful approach for detecting security flaws, discrepancies between the specification and implementation, design improvements, and high risk areas of the system.

3.2.2 Dynamic analysis

The contracts were compiled, deployed, and tested in a test environment, both manually and through the test suite provided. Manual analysis was used to confirm that the code was functional and to discover whether any potential security issues identified could be exploited.

3.2.3 Automated analysis

Tools were used to automatically detect the presence of several types of security issues, including reentrancy, timestamp dependency bugs, and transaction-ordering dependency bugs. The static analysis results were manually reviewed and any false positives were removed from the results. Any true positive results were included in this report.

Static analysis tools commonly used include Slither, Securify, and MythX. Tools such as the Remix IDE, compilation output, and linters could also be used to identify potential areas of concern.

3.3 Risk ratings

Each issue identified during the audit has been assigned a risk rating. The rating is determined based on the criteria outlined below.

- **High risk** - The issue could result in a loss of funds for the contract owner or system users.
- **Medium risk** - The issue resulted in the code specification being implemented incorrectly.

- **Low risk** - A best practice or design issue that could affect the security of the contract.
- **Informational** - A lapse in best practice or a suboptimal design pattern that has a minimal risk of affecting the security of the contract.
- **Closed** - The issue was identified during the audit and has since been addressed to a satisfactory level to remove the risk that it posed.

4. Design specification

The following section outlines the intended functionality of the system at a high level. The specification is based on the implementation in the codebase and any perceived points of conflict should be highlighted with the auditing team to determine the source of the discrepancy.

4.1 Thales Token

Thales.sol

The Thales Token is an ERC-20 token with the following properties:

Field	Value
Symbol	THALES
Name	Thales Token
Decimals	18
Initial supply	100 million

The token implementation uses OpenZeppelin's `ERC20` contract. The token supply is fixed at the initial supply, which is minted to the deployer upon deployment.

Plans for the token's initial distribution and use are provided in the Thales Tokenomics Medium post.

4.2 Token distribution

Airdrop.sol, OngoingAirdrop.sol, VestingEscrow.sol

The airdrop contracts can be used to distribute THALES tokens to users. After depositing tokens into the contracts, the owner could set the relevant Merkle root to allow users to claim tokens from the contract. The `Airdrop` contract allowed users to claim tokens directly, while the `OngoingAirdrop` contract added the tokens to the `EscrowThales` contract, as detailed below .

`VestingEscrow` can be used to distribute THALES tokens to users, subject to linear vesting over a set period.

Owner functionality

The `Airdrop` contract owner can call `_selfDestruct(...)` once 365 days have passed since the contract's deployment. This will destroy the contract using Solidity's `selfdestruct(...)` and transfer all Ether and THALES to a specified beneficiary address.

The `OngoingAirdrop` contract owner can take the following privileged actions:

- Change the `EscrowThales` contract address.
- Set the root of the Merkle tree.
- Destroy the contract, under the same conditions and with the same outcomes as described above for the `Airdrop` contract.

The `VestingEscrow` contract owner can take the following privileged actions:

- Add tokens to the contract's unallocated supply.
- Allocate additional tokens to individual addresses.
- Destroy the contract once 365 days have passed since the end of the vesting period. This will destroy the contract using Solidity's `selfdestruct(...)` and transfer all Ether and THALES to a specified beneficiary address.

4.3 Staking and escrow

StakingThales.sol, EscrowThales.sol

Users can stake their Thales Token to acquire rewards, paid in the same token. Stakers may also be able to earn a share of sUSD fees, should this functionality be enabled by Thales DAO governance in the future.

At the end of every staking period, stakers can call `StakingThales.claimReward()` to place their staking rewards in escrow and claim their fees. Staking rewards and claimable fees are calculated based on the staker's share of the total staked and escrow balances. Per the Tokenomics post, the `OngoingAirdrop` contract will be used to offset the gas costs of regular reward claiming.

Staking rewards are placed in escrow for a vesting period of 10 staking periods. Once this vesting period has passed, stakers can vest these rewards.

To begin the unstaking process, a user must call `StakingThales.startUnstake(...)` with the amount of tokens they would like to unstake, which can be some or all of their staked amount. This initiates a staking 7 day cooldown period, during which their tokens are still held `StakingThales` but do not produce rewards. Once this period has elapsed, the user can call `StakingThales.unstake()` to complete the unstaking process and retrieve their previously staked tokens.

The `closePeriod()` function must be called at the end of every staking period to add the period's rewards for all stakers and move to the next period. This function can be called by any user after a full staking period has passed since the start of the previous staking period.

By default, the staking period and unstaking cooldown periods are both 7 days, but these durations can be changed by the contract owner.

Owner functionality

The `EscrowThales` contract owner can take the following privileged actions:

- Change the `StakingThales` and `Airdrop` contract addresses.
- Destroy the contract through `selfdestruct()` and transfer its Ether and THALES to a specified beneficiary address.

The `StakingThales` contract owner can take the following privileged actions:

- Change the `EscrowThales` contract address.
- Enable and disable fee distribution to stakers.
- Change the reward multiplier used to calculate rewards for each period.
- Enable and disable claiming.
- Change the length of the staking period.
- Change the length of the unstaking cooldown period.

- Start the first staking period (this can only be done once).
- Destroy the contract through `selfdestruct()` and transfer its Ether, THALES and sUSD to a specified beneficiary address.

5. Detailed findings

The following section details the findings of the audit.

5.1 High risk

No high-risk issues were open at the conclusion of the audit.

5.2 Medium risk

No medium-risk issues were open at the conclusion of the audit.

5.3 Low risk

No low-risk issues were open at the conclusion of the audit.

5.4 Informational

No informational issues were open at the conclusion of the audit.

5.5 Closed

5.5.1 Custom `transferFrom` does not remove amount from allowance (high risk)

Thales.sol

Description

The Thales token included a custom `transferFrom(...)` function which did not subtract the transferred amount from the user's `allowance`. This is not compliant with the ERC-20 standard and would allow arbitrary users and contracts to transfer funds between other addresses without authorization.

Recommendation

The custom `transferFrom(...)` function should be removed to ensure that the Thales token will use the secure, compliant `transferFrom(...)` function from OpenZeppelin's `ERC20` contract.

Update

Fixed in 519aa66.

5.5.2 Underflow in `fund` function (high risk)

VestingEscrow.sol

Description

An underflow of the `unallocatedSupply` variable was identified in the `fund()` function. As a result, it would be possible for an owner to drain the funds from the vesting contract.

Recommendation

SafeMath should be used to perform the subtraction.

Update

SafeMath was used in 77b29c8.

5.5.3 Unstaking does not account for cooldown period (high risk)

StakingThales.sol

Description

As per the Thales Tokenomics documentation, unstaking was subject to a cooldown period of 7 days (configurable by the contract owner). Stakers could call `startUnstaking()` to begin the cooldown period, during which no rewards would be received, and should only call `unstake()` after the cooldown period had elapsed to complete unstaking. However, `unstake()` did not include a check that the cooldown period had elapsed, and could be called immediately after `startUnstaking()`, rendering the cooldown period irrelevant.

Recommendation

A check should be included to prevent users from calling `unstake()` until the cooldown period has elapsed.

Update

Fixed in e03c2de.

5.5.4 Comparison between weeks and seconds (medium risk)

EscrowThales.sol

Description

In `getVestingNotAvailable(...)` a comparison was made between `vestingEntry[account][i].vesting_week` (denominated in seconds) and `periodsOfVesting` (denominated in weeks). This prevented the comparison from working as intended.

Recommendation

`vesting_week` should be denominated in weeks to match other values in the contract.

Update

In 16326ec, `vesting_week` 's assignment in `addToEscrow(...)` was changed to denominate the value in weeks.

5.5.5 Reentrancy guard prevents additional staking under certain conditions (medium risk)

StakingThales.sol

Description

If a user has any claimable rewards and tries to stake additional tokens, the transaction would reverts since both the claim function and the stake function have a reentrancy guard. In this scenario, the reentrancy guard is locked on the first call to the `stake` function, which prevents any calls to other `nonReentrant` functions until the `stake` function has finished executing. A user can continue staking if the `claimEnabled` flag is disabled by the contract admin.

Recommendation

An unprotected internal `_claimReward()` function should be created to house the logic of the external `claimReward()` function. The reentrancy guards on `claimReward()` and `stake()` should remain as-is, and the unprotected internal `_claimReward()` function should be called from the external `stake()` and `claim()` functions.

Update

Fixed in 524283b

5.5.6 Restaking fails under certain conditions (low risk)

StakingThales.sol

Description

If a user unstaked while reward claiming was disabled, they would not be able to restake in a later period while reward claiming was enabled. This is because `stake(...)` would call `claimReward()`, which would revert as the user would have no staked balance.

Recommendation

To prevent this denial of service, include a check to ensure that the staking user has a staked balance above zero before calling `claimReward()`.

Update

Fixed in aa165d1

5.5.7 Complex vesting implementation (informational)

EscrowThales.sol

Description

The implementation of the 10 week vesting period for staking rewards included a large number of conditional statements and edge cases, which could be simplified to reduce contract length and gas costs. Such simplification would also make the code simpler to understand and thus less likely to contain bugs.

Recommendation

Instead of manually accounting for different 10-period cycles and maintaining the `stakerSilo` of vestable funds, the contract could store the vesting period with each vesting entry. Vestable funds would then be determined by which entries had completed vesting.

Update

Thales implemented the recommended pattern in aa165d1 and eab27ba.

5.5.8 Missing events (informational)

EscrowThales.sol, StakingThales.sol

Description

The following state-changing functions did not emit events:

- `EscrowThales.setStakingThalesContract(...)`
- `EscrowThales.setAirdropContract(...)`
- `StakingThales.setClaimEnabled(...)`
- `StakingThales.setDistributeFeesEnabled(...)`
- `StakingThales.setFixedWeeklyReward(...)`
- `StakingThales.setDurationPeriod(...)`
- `StakingThales.setUnstakeDurationPeriod(...)`
- `StakingThales.setEscrow(...)`

- `StakingThales.startStakingPeriod()`

Events aid in the visibility of contract state changes. This information can be used on the dApp frontend, and could also be useful for users.

Recommendation

Events should be added to the affected functions to emit the state changes that are made.

Update

Events added in 69f46f7.

5.5.9 Use of unsafe ERC-20 transfer functions (informational)

General

Description

The contracts did not make consistent use of the `safeTransfer(...)` and `safeTransferFrom(...)` functions from the OpenZeppelin SafeERC20 library. `safeTransfer(...)` and `safeTransferFrom(...)` abstract the standard ERC-20 `transfer(...)` and `transferFrom(...)` functions and throws an exception if the transfer returns `false`. This adds an additional layer of validation in case the ERC-20 token returns `false` on failed transfers instead of reverting.

Recommendation

The OpenZeppelin SafeERC20 library should be used to wrap all important ERC-20 functions as a defense-in-depth measure. Alternatively, transfers could be wrapped in `require` statements, which achieves a similar effect.

Update

Safe transfer functions were implemented for all token transfers in 69f46f7.

5.5.10 Design comments (informational)

Actions to improve the functionality and readability of the codebase are outlined below.

Refactoring suggestions

General

Recommendations for improving code clarity, consistency and concision are provided below.

1. As the staking duration period can be altered by the owner, using a term such as "staking periods" rather than weeks would add to code clarity and future-proofing.
2. `EscrowThales.sol` : The `getCurrentWeek()` function is unnecessary if `_weeksOfVesting` is intended to be public, as Solidity will create a getter for it automatically.
3. `EscrowThales.sol` : The `_StakingThalesContract` address does not need to be stored, as it can be retrieved by calling `address(iStakingThales)` .
4. `EscrowThales.sol#L83` : In the second require statement in `claimable()` , "WeeksOfStaking = 0" should be "WeeksOfVesting = 0".
5. `StakingThales.sol` : The `msg.sender != address(0)` checks in `startUnstaking()` and `unstake()` are unnecessary and can be removed.
6. `StakingThales.sol#L218` : The third require statement in `startUnstake()` checks the same condition as the fourth require statement and can be removed.
7. `StakingThales.sol` : `calculateFeesForWeek()` could store its `feeToken` balance in a variable to avoid making two external calls.
8. `Airdrop.sol` : The comment does not match the code and should be updated or removed.
9. `StakingThales.sol` : The `_lastStakingPeriod` mapping is no longer used, and should be removed.

Update

1. Implemented in 69f46f7.
2. Implemented in 69f46f7-L136.
3. Implemented in 69f46f7-L93.
4. Fixed in 69f46f7-L83.

5. Implemented in 69f46f7.
6. Implemented in 69f46f7-L218.
7. Implemented in 69f46f7-L331.
8. Implemented in af1672f-L66
9. Removed in 47d3625

Fix spelling, grammar and naming convention errors

General

Spelling and grammar mistakes and contraventions of Solidity naming conventions were identified in the codebase. Fixing these mistakes can help improve the end-user experience by providing clear information on errors encountered, and improve the maintainability and auditability of the codebase.

1. `EscrowThales.sol`, `StakingThales.sol`: Internal functions were not named with a preceding underscore (_) as per Solidity convention.
2. `EscrowThales.sol`: State variables with `public` visibility were named with a preceding underscore (_). These should be renamed or made private depending on their intended visibility.
3. `StakingThales.sol`: The state variable `_totalUnlcaimedFees` should be `_totalUnclaimedFees`.
4. `StakingThales`: "Account has not performed triggered unstake cooldown" – "performed" can be removed.
5. `StakingThales`: The message in the second require statement in `closePeriod()` does not take into account the mutability of the duration period. It could be changed from "7 days has not passed since the last closed period" to "A full period has not passed since the last closed period".

Update

1. Fixed in 69f46f7.
2. Fixed in 69f46f7.
3. Fixed in 69f46f7-L166.
4. Fixed in 69f46f7-L240.
5. Fixed in 69f46f7-L181.

Secure your system.

Request a service

START NOW →