BAIL security

BAILSEC.IO

EMAIL : OFFICE@BAILSEC.IO

TWITTER : @BAILSECURITY

TELEGRAM : @HELLOATBAILSEC

# FINAL REPORT:

## Cairo Finance Vaults

**January 2024**

# Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, 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. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

## 1. Project Details

**Important:**
**Please ensure that the deployed contract matches the source-code of the last commit hash.**

| Project | Amphor |
|---|---|
| Website | cairofinance.app |
| Type | Vaults |
| Language | Solidity |
| Methods | Manual Analysis |
| Github repository | https://github.com/cairofinanceapp/Farm/tree/0c315eba3e2f9eb8c84b87e6e8a880612f0e91ec |
| Resolution 1 | https://github.com/cairofinanceapp/Farm/tree/231ad6a42d1fc68548c1a775aad954564649908c |
| Resolution 2 | https://github.com/cairofinanceapp/Farm/tree/4a291d0f91e03c324706d81213addee2b74f14b4 |
| Resolution 3 | https://github.com/cairofinanceapp/Farm/tree/cff7fe63fdd6f3e148b6505fb87cb852afbee443 |

## 2. Detection Overview

### 1st Audit:

| Severity | Found | Resolved | Partially Resolved | Acknowledged (no change made) |
|---|---|---|---|---|
| High | 25 | 24 | | |
| Medium | 12 | 12 | | |
| Low | 9 | 7 | 1 | |
| Informational | 0 | | | |
| Configurational | 0 | | | |
| Governance | 0 | | | |
| Quality assurance | 0 | | | |
| Total | 46 | 43 | 1 | |

### 2nd Audit:

| Severity | Found | Resolved | Partially Resolved | Acknowledged (no change made) |
|---|---|---|---|---|
| High | 8 | 7 | 1 | |
| Medium | 8 | 7 | 1 | |
| Low | 6 | 4 | | 1 |
| Informational | | | | |
| Configurational | | | | |
| Governance | | | | |
| Quality assurance | | | | |
| Total | 22 | 18 | 2 | 1 |

## 2.1 Detection Definitions

| Severity | Description |
|---|---|
| High | The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users. |
| Medium | While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences. |
| Low | Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately |
| Informational | Effects are small and do not post an immediate danger to the project or users |
| Configurational | Issues which may arise due to different configurational settings |
| Governance | Governance privileges which can directly result in a loss of funds or other potential undesired behavior |
| Quality assurance | Aggregated minor issues, ensuring a high quality codebase. |

# 3.Detection

**Disclaimer:**
**This report is for the second subsequent audit after the first audit round, the first round uncovered the following issues:**

## 1st Audit:

### CairoStaking (Avalanche)

The `CairoStaking` contract is a simple staking contract that allows users to stake their AVAX tokens. These tokens will then be deposited into the underlying ANKR protocol. For each staked AVAX token, the Cairo contract will receive a corresponding Certificate token which represents the underlying AVAX value and will steadily decrease over time, once per day.

The contract implements a 4-tier referral logic which distributes additional Cairo tokens to referrers. The referral amount is based on the overall profit value in AVAX, which is then converted to the Cairo value and distributed to the different referral tiers with 5%, 2%, 2% and 1%.

Users will therefore stake their AVAX and receive more AVAX in return for their position. Moreover, users will receive Cairo tokens for their staked position, based on the amount of AVAX tokens which are initially allocated towards their position, calculated using a modified form of the synthetix reward logic (https://github.com/Synthetixio/synthetix/blob/develop/contracts/StakingRewards.sol).

Users can request a withdrawal of their AVAX tokens at any time which then will calculate the corresponding AVAX amount for the amount of certificate tokens they have received. This will then trigger a withdrawal request within the AvalanchePool contract. After some time, eventually this withdrawal request gets fulfilled and the corresponding AVAX tokens are being transferred to the Cairo contract. However, this solely depends on the ANKR team on when this request is fulfilled. Upon the withdrawal request, a 30% fee is applied on the profit which was generated due to the increase of the AVAX amount.

This fee is used based on two different strategies:

a) The AVAX fee is accumulated, withdrawable by the contract owner
b) The corresponding Cairo value for the fee amount is fetched via a third party oracle and allocated to 4 upstream referrers with the following percentages: [5%, 2%, 2%, 1%], therefore the sum of the referral value is 10% on the total profit made by the staker.

Once a user has successfully requested a withdrawal and the Cairo contract has a valid balance, users can then claim their requested balance, deducting the fee which has been calculated during the withdrawal initialization.

*It is of utmost importance to mention that the external Oracle contract has not been audited by BailSec. We highly recommend getting a thorough audit for this part as oracles are always vulnerable to manipulations, especially an oracle which returns the price of Cairo, since Cairo has no CEX listing nor a large liquidity pool, it can be easily manipulated.*

| Issue | Users can steal ETH via calling claimUnstakeAmount multiple times |
|---|---|
| Severity | High |
| Description | The claimUnstakeAmount function can be called multiple times, allowing users to steal ether from the contract. |
| Recommendations | Consider resetting the fee and amount value. |
| Comments / Resolution | Resolved. |

| Issue | Admin can steal ETH |
|---|---|
| Severity | High |
| Description | Within the withdrawFee function, the _unstakeFeeCollected amount is being transferred out. However, this value is not being reset, allowing the admin to steal all eth in the contract.<br>Moreover, the unsafeTransfer function allows the owner to simply transfer out all eth in the contract |
| Recommendations | Consider resetting _unstakeFeeCollected before the ether transfer, as well as removing the unsafeTransfer function |
| Comments / Resolution | Resolved. |

| Issue | Withdrawals might revert if the ratio is below 1e17 |
|---|---|
| **Severity** | High |
| **Description** | The ratio value is a steadily decreasing value which denotes how much certTokens will be received per AVAX. As this value decreases over time, a withdrawal can revert if a user deposited 1 AVAX whenever the ratio is below 1e17, as this will then revert in the following line:<br>require(underlyingAmount >= 0.1 ether, "Invalid amount Called"); |
| **Recommendations** | Consider removing this check. |
| **Comments / Resolution** | Resolved |

| Issue | Flawed reward logic |
|---|---|
| **Severity** | High |
| **Description** | The contract implements a modified version of the synthetix reward logic, which can be found here:<br>https://github.com/Synthetixio/synthetix/blob/develop/contracts/StakingRewards.sol<br>Whenever clients modify this reward logic, it is often flawed and can result in stolen reward tokens or inflated reward values. This scenario is also present here, illustrated by the following PoC:<br>a) rewardRate = 1e18 (tokens per second)<br>b) Alice stakes 100e18 tokens at timestamp 1000<br>c) Alice deposits 1 wei at timestamp 1100<br>-> at this point, user[Alice].rewards will be 100e18, since 1 token per |

| | |
|---|---|
| | second is distributed and Alice is the only staker in the contract<br>d) Alice deposits another 1 wei at the same timestamp<br>-> Alice should not receive any additional rewards since no time has passed and therefore no tokens will be distributed<br>-> However, Alice's pending now becomes 200e18<br><br>Alice can repeat this scenario to create a large pending value, which can then later be abused to steal tokens from the vault. |
| **Recommendations** | Consider simply switching to the synthetix staking rewards logic while using the allocated underlyingAmount as multiplier for the reward calculation and using the total underlyingAmount as divisor for the rewardPerToken calculation.<br><br>Moreover, the developer can easily select the desired reward rate by simply modifying the notifyRewardAmount. It should be ensured that the contract contains sufficient reward tokens though. |
| **Comments / Resolution** | Failed resolution. |

| Issue | Upstream referral logic is flawed |
|---|---|
| **Severity** | High |
| **Description** | The contract is developed in an effort to support upstream referrals, this involves the reward distribution towards the referrer of the referrer.<br><br>Unfortunately, the implementation is flawed as it will not take the referrer from the referrer but the referrer from the initial caller:<br><br>referrer = user[msg.sender].referralAddress;<br><br>This results in the first-tier referrer getting all referral rewards. |

| | |
|---|---|
| **Recommendations** | Consider caching the referralAddress from the referrer instead. |
| **Comments / Resolution** | Resolved. |

| Issue | Change of rewardRate will change un updated positions |
|---|---|
| **Severity** | High |
| **Description** | Since the rewardRate change directly changes how much tokens will be accumulated, a change of this rate will also change the tokens received as reward from users that do not have updated their position to date.<br>As an example, if a user has staked 100 AVAX 1 year ago but never called getReward during that time, the new rewardRate will now be applied on the whole 1 year period, effectively decreasing/increasing the amount the user will receive. |
| **Recommendations** | Whi Consider simply switching to the synthetix staking rewards logic and then executing updateRewards(address(0)) before the rewardRate change. This will ensure that rewardPerTokenStored is on the updated value:<br>An example can be found here:<br>https://github.com/Synthetixio/synthetix/blob/develop/contracts/StakingRewards.sol#L113 |
| **Comments / Resolution** | Resolved. |

| Issue | Flash-theft possibility due to Ankr architecture |
|---|---|
| **Severity** | High |
| **Description** | The Ankr protocol manually decreases the _ratio parameter once per day. A malicious user can frontrun this change, receive a certToken amount based on the old _ratio parameter and immediately withdraw again with the new _ratio, receiving more ether.<br>This bug is also known to Ankr, but partially prevented since Ankr does not immediately pay out the tokens.<br>However, if the Cairo contract has a pending balance, an attacker can in fact immediately withdraw the desired amount, effectively stealing tokens from the vault.<br>This can even be abused if there are not enough tokens in the contract by artificially increasing the profit, receiving referral rewards on that profit. If the amount of deposited AVAX is large enough, this will have a significant impact which can lead up to stealing all Cairo tokens in the contract. |
| **Recommendations** | This issue cannot be easily fixed since Ankr is responsible for this. Therefore, we simply recommend taking a minor fee during each deposit from the initial msg.value. |
| **Comments / Resolution** | Resolved. |

| Issue | Last withdrawal might revert due to insufficient bond tokens |
|---|---|
| **Severity** | High |
| **Description** | Within the withdrawal logic of the Ankr contract, during the last transaction of the whole flow, the corresponding bondTokens for the AVAX value are being burned:<br>uint256 shares = _bondsToSharesCeil(amount);<br>_burn(from, shares);<br>The slight problem here is that Ankr is rounding against the favor of the user within the _bondsToSharesCeil function, which will round up the amount of bondTokens to be burned, this is considered as a general best-practice which is often applied when it comes to burning receipt tokens.<br>This can result in slightly more bonds to be burned than the user has allocated as balance, potentially DoS'ing the withdrawal due to an insufficient balance of CertTokens in the contract. |
| **Recommendations** | Consider slightly decreasing the underlyingAmount parameter for the claimCert function, to ensure that not more tokens are burned than the user has allocated. |
| **Comments / Resolution** | Resolved, however, it is necessary to provide a proper test for this scenario to ensure the fix works as expected. It is mandatory for the developer to provide such a test. |

| Issue | Using initial ETH balance instead of accumulated value for reward calculation |
|---|---|
| **Severity** | Medium |
| **Description** | Within the reward calculation, user[_account].balance is used to calculate the desired reward.<br>However, this will NOT represent the real accumulated value through the ANKR reward accrue. |
| **Recommendations** | Consider using the underlyingBalance, as this represents the CertToken value, which will be decreased for newer deposits with the same initial value. |
| **Comments / Resolution** | Resolved. |

| Issue | _unsafeTransfer does not force success |
|---|---|
| **Severity** | Medium |
| **Description** | The aforementioned function can silently revert, leaving users without their claimed tokens while deducting their claimable amount and fee (once it is fixed in the updated commit). |
| **Recommendations** | Consider forcing success to be true. |
| **Comments / Resolution** | Resolved. |

| Issue | _treasuryWallet can be set to address(0) |
|---|---|
| Severity | Medium |
| Description | Whenever the aforementioned variable is set to address(0), this will result in a permanent loss of all eth which is claimed. |
| Recommendations | Consider implementing a validation that prevents this scenario. |
| Comments / Resolution | Resolved. |

| Issue | Owner cannot be changed |
|---|---|
| Severity | Medium |
| Description | The owner is set once during contract deployment and can call many privileged functions as well as receiving the tokens which are recovered.<br>However, the owner cannot be changed which can result in undesired side-effects for example in the scenario of compromised keys. |
| Recommendations | Consider simply inheriting the Ownable library or implementing a transferOwnership function. |
| Comments / Resolution | Resolved. |

| Issue | Referrer can be changed |
|---|---|
| Severity | Low |
| Description | Generally speaking, once a referrer is set, it should not be changeable. Moreover, it should not be possible to set the own address as a referrer. |
| Recommendations | Consider only setting the referrer during the very first deposit, moreover a self-address check should be implemented. |
| Comments / Resolution | Resolved. |

| Issue | Architecture: Claiming happens on a first-come-first-serve basis |
|---|---|
| Severity | Low |
| Description | Whenever a user initiates a withdrawal, this will then be scheduled within the Ankr staking contract and the withdrawal will happen at an arbitrary time.<br>This can result in issues for the following example:<br>a) Alice queues a withdrawal of 100 AVAX<br>b) The withdrawal is being executed within the Ankr staking contract, the Cairo contract now has 100 AVAX<br>b) Bob queues a withdrawal of 100 AVAX in the same block as c) and immediately claims the tokens<br>Alice therefore needs to wait for the next withdrawal from ANKR to be executed. |
| Recommendations | This issue is an architectural issue and therefore a fix would involve modifying the whole logic. |

| | |
|---|---|
| **Comments / Resolution** | Partially resolved, a cooldown time has been implemented. |

| Issue | isClaimAvailable is flawed |
|---|---|
| **Severity** | Low |
| **Description** | The aforementioned function will return true as long as the contract has a non-zero balance and the user has a valid request.<br>However, this is incorrect as the claim will only be available when the contract balance covers the request amount. |
| **Recommendations** | Consider implementing such a check. |
| **Comments / Resolution** | Failed resolution, the function contains another critical as mentioned in the second audit round. |

## CairoStaking (Ethereum)

The CairoStaking (Ethereum) contract is similar to the Avalanche version, with the difference that the msg.value to certToken conversion is being directly fetched from the bondsToShares function instead of the ratio being fetched and manually calculated.

Moreover, during the withdrawal, the conversion is directly done by calling sharesToBonds on the CertToken. Lastly, the underlying Ankr source code is slightly different.

**Since the AETH contract is unknown, we cannot determine whether following line within the stake function is correct:*

*uint256 unAmount = underlyingToken.bondsToShares(_amount);*

*But we assume that it is incorrect as it rounds up and therefore allocates more certTokens to the user than it should.*

StakingContract:
https://etherscan.io/address/0x1701ad6a252e24dee1d71dc1cad6da5426e0a3f1#code

BondToken:
https://etherscan.io/address/0x5de57c3535e1f840ecb3e2a10c9955387685756d#code

CertToken:
https://etherscan.io/address/0x3ed1dfbccf893b7d2d730ead3e5edbf1f8f95a48#code

***Eventual issues might apply if the proxy upgrades the implementations. This audit is solely focused on the aforementioned implementation contracts.**

| Issue | Deposit value can round down |
|---|---|
| Severity | High |
| Description | Contrary to the Avalanche contract, users can stake all values of ETH, including such as 1.1-1.9 and 0.9.<br>Due to the bondsToShares calculation, these values will round down, resulting in a loss of user tokens. |
| Recommendations | Consider implementing the same modulo and >0 check as in avalanche |
| Comments / Resolution | Fixed, rounding issues can not occur anymore with the following ANKR implementation:<br>https://etherscan.io/address/0xe672e0e0101a7f58d728751e2a5e6da5ff1fda64#code |

| Issue | Users can steal ETH via calling claimVested multiple times |
|---|---|
| Severity | High |
| Description | The claimVested function can be called multiple times, allowing users to steal ether from the contract. |
| Recommendations | Consider resetting the fee and amount value. |
| Comments / Resolution | Fixed. |

| Issue | Admin can steal ETH |
|---|---|
| **Severity** | High |
| **Description** | Within the withdrawFee function, the _unstakeFeeCollected amount is being transferred out. However, this value is not being reset, allowing the admin to steal all eth in the contract.<br>Moreover, the unsafeTransfer function allows the owner to simply transfer out all eth in the contract |
| **Recommendations** | Consider resetting _unstakeFeeCollected before the ether transfer, as well as removing the unsafeTransfer function |
| **Comments / Resolution** | Fixed. |

| Issue | Withdrawals might revert if the ratio is below 1e17 |
|---|---|
| **Severity** | High |
| **Description** | The ratio value is a steadily decreasing value which denotes how much certTokens will be received per AVAX. As this value decreases over time, a withdrawal can revert if a user deposited 1 AVAX whenever the ratio is below 1e17, as this will then revert in the following line:<br>require(underlyingAmount >= 0.1 ether, "Invalid amount Called"); |
| **Recommendations** | Consider removing this check. |
| **Comments / Resolution** | Fixed. |

| Issue | Flawed reward logic |
|---|---|
| Severity | High |
| Description | The contract implements a modified version of the synthetix reward logic, which can be found here:<br>https://github.com/Synthetixio/synthetix/blob/develop/contracts/StakingRewards.sol<br>Whenever clients modify this reward logic, it is often flawed and can result in stolen reward tokens or inflated reward values. This scenario is also present here, illustrated by the following PoC:<br>a) rewardRate = 1e18 (tokens per second)<br>b) Alice stakes 100e18 tokens at timestamp 1000<br>c) Alice deposits 1 wei at timestamp 1100<br>-> at this point, user[Alice].rewards will be 100e18, since 1 token per second is distributed and Alice is the only staker in the contract<br>d) Alice deposits another 1 wei at the same timestamp<br>-> Alice should not receive any additional rewards since no time has passed and therefore no tokens will be distributed<br>-> However, Alice's pending now becomes 200e18<br>Alice can repeat this scenario to create a large pending value, which can then later be abused to steal tokens from the vault. |
| Recommendations | Consider simply switching to the synthetix staking rewards logic while using the allocated underlyingAmount as multiplier for the reward calculation and using the total underlyingAmount as divisor for the rewardPerToken calculation.<br>Moreover, the developer can easily select the desired reward rate by simply modifying the notifyRewardAmount. It should be ensured that the contract contains sufficient reward tokens though. |

|  |  |
|---|---|
| **Comments / Resolution** | Fixed, however, extensive provided testing is necessary. |

| Issue | Upstream referral logic is flawed |
|---|---|
| **Severity** | High |
| **Description** | The contract is developed in an effort to support upstream referrals, this involves the reward distribution towards the referrer of the referrer.<br>Unfortunately, the implementation is flawed as it will not take the referrer from the referrer but the referrer from the initial caller:<br>referrer = user[msg.sender].referralAddress;<br>This results in the first-tier referrer getting all referral rewards. |
| **Recommendations** | Consider caching the referralAddress from the referrer instead. |
| **Comments / Resolution** | Fixed. |

| Issue | Change of rewardRate will change un updated positions |
|---|---|
| **Severity** | High |
| **Description** | Since the rewardRate change directly changes how much tokens will be accumulated, a change of this rate will also change the tokens received as reward from users that do not have updated their position to date.<br>As an example, if a user has staked 100 AVAX 1 year ago but never |

| | |
|---|---|
| | called getReward during that time, the new rewardRate will now be applied on the whole 1 year period, effectively decreasing/increasing the amount the user will receive. |
| **Recommendations** | Consider simply switching to the synthetix staking rewards logic and then executing updateRewards(address(0)) before the rewardRate change. This will ensure that rewardPerTokenStored is on the updated value:<br>An example can be found here:<br>https://github.com/Synthetixio/synthetix/blob/develop/contracts/StakingRewards.sol#L113 |
| **Comments / Resolution** | Fixed. |

| **Issue** | **Flash-theft possibility due to Ankr architecture** |
|---|---|
| **Severity** | **High** |
| **Description** | The Ankr protocol manually decreases the _ratio parameter once per day. A malicious user can frontrun this change, receive a certToken amount based on the old _ratio parameter and immediately withdraw again with the new _ratio, receiving more ether.<br>This bug is also known to Ankr, but partially prevented since Ankr does not immediately pay out the tokens.<br>However, if the Cairo contract has a pending balance, an attacker can in fact immediately withdraw the desired amount, effectively stealing tokens from the vault.<br>This can even be abused if there are not enough tokens in the contract by artificially increasing the profit, receiving referral rewards |

|  |  |
|---|---|
|  | on that profit. If the amount of deposited AVAX is large enough, this will have a significant impact which can lead up to stealing all Cairo tokens in the contract. |
| **Recommendations** | This issue cannot be easily fixed since Ankr is responsible for this. Therefore, we simply recommend taking a minor fee during each deposit from the initial msg.value. |
| **Comments / Resolution** | Fixed, a cooldown period has been implemented. |

| Issue | Using initial ETH balance instead of accumulated value for reward calculation |
|---|---|
| **Severity** | Medium |
| **Description** | Within the reward calculation, user[_account].balance is used to calculate the desired reward.<br>However, this will NOT represent the real accumulated value through the ANKR reward accrue. |
| **Recommendations** | Consider using the underlyingBalance, as this represents the CertToken value, which will be decreased for newer deposits with the same initial value. |
| **Comments / Resolution** | Resolved. |

| Issue | _unsafeTransfer does not force success |
|---|---|
| **Severity** | Medium |
| **Description** | The aforementioned function can silently revert, leaving users without their claimed tokens while deducting their claimable amount and fee (once it is fixed in the updated commit). |
| **Recommendations** | Consider forcing success to be true. |
| **Comments / Resolution** | Resolved. |

| Issue | _treasuryWallet can be set to address(0) |
|---|---|
| **Severity** | Medium |
| **Description** | Whenever the aforementioned variable is set to address(0), this will result in a permanent loss of all eth which is claimed. |
| **Recommendations** | Consider implementing a validation that prevents this scenario. |
| **Comments / Resolution** | Resolved. |

| Issue | Owner cannot be changed |
|---|---|
| **Severity** | Medium |
| **Description** | The owner is set once during contract deployment and can call many privileged functions as well as receiving the tokens which are recovered.<br>However, the owner cannot be changed which can result in undesired side-effects for example in the scenario of compromised |

| | |
|---|---|
| | keys. |
| **Recommendations** | Consider simply inheriting the Ownable library or implementing a transferOwnership function. |
| **Comments / Resolution** | Resolved. |

| Issue | Referrer can be changed |
|---|---|
| **Severity** | Low |
| **Description** | Generally speaking, once a referrer is set, it should not be changeable. Moreover, it should not be possible to set the own address as a referrer. |
| **Recommendations** | Consider only setting the referrer during the very first deposit, moreover a self-address check should be implemented. |
| **Comments / Resolution** | Resolved. |

| Issue | Architecture: Claiming happens on a first-come-first-serve basis |
|---|---|
| **Severity** | Low |
| **Description** | Whenever a user initiates a withdrawal, this will then be scheduled within the Ankr staking contract and the withdrawal will happen at an arbitrary time.<br>This can result in issues for the following example:<br>a) Alice queues a withdrawal of 100 AVAX<br>b) The withdrawal is being executed within the Ankr staking contract, |

|  |  |
|---|---|
|  | the Cairo contract now has 100 AVAX<br>b) Bob queues a withdrawal of 100 AVAX in the same block as c) and immediately claims the tokens<br><br>Alice therefore needs to wait for the next withdrawal from ANKR to be executed. |
| **Recommendations** | This issue is an architectural issue and therefore a fix would involve modifying the whole logic. |
| **Comments / Resolution** | Resolved, a cooldown time has been implemented. |

| Issue | isClaimAvailable is flawed |
|---|---|
| **Severity** | Low |
| **Description** | The aforementioned function will return true as long as the contract has a non-zero balance and the user has a valid request.<br><br>However, this is incorrect as the claim will only be available when the contract balance covers the request amount. |
| **Recommendations** | Consider implementing such a check. |
| **Comments / Resolution** | Resolved, a minor issue is still present. |

## CairoStaking (ETHMatic)

The CairoStaking (ETHMatic) contract is similar to the Avalanche version with the difference that users can stake WMatic instead of the gas token.

StakingContract:
https://etherscan.io/address/0xcb6805e51ea42741d17d1c1f59e01fbe80aba389#code

BondToken:
https://etherscan.io/address/0xd4502103dd36c5595dccedf33e7308c61428ce3b#code

CertToken:
https://etherscan.io/address/0x50be7ae35c5bf838d060045f33f93449f9aff49c

***Eventual issues might apply if the proxy upgrades the implementations. This audit is solely focused on the aforementioned implementation contracts.**

| Issue | recoverERC20 allows admin to steal wMatic token |
|---|---|
| **Severity** | High |
| **Description** | The aforementioned function allows the admin to steal any wMatic token within the contract. |
| **Recommendations** | Consider removing this function. |
| **Comments / Resolution** | Resolved. |

| Issue | No unstake fee forwarded as msg.value |
|---|---|
| Severity | High |
| Description | During the withdraw function, the unstakeCerts function within the StakingContract is being called. This function expects a msg.value forwarded with the call:<br>function unstakeCerts(uint256 shares, uint256 fee, uint256 useBeforeBlock, bytes memory signature) override external payable nonReentrant {<br>_collectFee(msg.sender, fee, useBeforeBlock, signature); uint256 amount = IinternetBond_R2(_bondContract).sharesToBalance(shares); IinternetBond_R2(_bondContract).lockSharesFor(msg.sender, shares);<br>_unstake(msg.sender, amount, shares, fee, false);<br>}<br>function _collectFee(address staker, uint256 fee, uint256 useBeforeBlock, bytes memory signature) internal {<br>if (fee == 0 && useBeforeBlock == 0) { require(msg.value >= ethUnstakeFee, "PolygonPool: not enough ETH to pay fee");<br>address payable wallet = payable(_feeCollector);<br>require(wallet.send(msg.value), "PolygonPool: could not transfer unstake fee");<br>emit EthFeeCollected(staker, msg.value);<br>} else {<br>However, there is no value attached to the call, hence any withdrawal will always revert if ANKR decides to implement this fee. |

| | |
|---|---|
| **Recommendations** | Consider attaching the necessary value to this call. |
| **Comments / Resolution** | Resolved. |

| Issue | Deposit value can round down |
|---|---|
| **Severity** | High |
| **Description** | Contrary to the Avalanche contract, users can stake all values of WMatic, including such as 1.1-1.9 and 0.9.<br>Due to the bondsToShares calculation, these values will round down, resulting in a loss of user tokens. |
| **Recommendations** | Consider implementing the same modulo and >0 check as in avalanche |
| **Comments / Resolution** | Resolved |

| Issue | Users can steal ETH via calling claimUnstakeAmount multiple times |
|---|---|
| **Severity** | High |
| **Description** | The claimVested function can be called multiple times, allowing users to steal ether from the contract. |
| **Recommendations** | Consider resetting the fee and amount value. |
| **Comments / Resolution** | Resolved |

| Issue | Flawed reward logic |
|---|---|
| Severity | High |
| Description | The contract implements a modified version of the synthetix reward logic, which can be found here:<br>https://github.com/Synthetixio/synthetix/blob/develop/contracts/StakingRewards.sol<br><br>Whenever clients modify this reward logic, it is often flawed and can result in stolen reward tokens or inflated reward values. This scenario is also present here, illustrated by the following PoC:<br><br>a) rewardRate = 1e18 (tokens per second)<br>b) Alice stakes 100e18 tokens at timestamp 1000<br>c) Alice deposits 1 wei at timestamp 1100<br>-> at this point, user[Alice].rewards will be 100e18, since 1 token per second is distributed and Alice is the only staker in the contract<br>d) Alice deposits another 1 wei at the same timestamp<br>-> Alice should not receive any additional rewards since no time has passed and therefore no tokens will be distributed<br>-> However, Alice's pending now becomes 200e18<br><br>Alice can repeat this scenario to create a large pending value, which can then later be abused to steal tokens from the vault. |
| Recommendations | Consider simply switching to the synthetix staking rewards logic while using the allocated underlyingAmount as multiplier for the reward calculation and using the total underlyingAmount as divisor for the rewardPerToken calculation.<br><br>Moreover, the developer can easily select the desired reward rate by simply modifying the notifyRewardAmount. It should be ensured that the contract contains sufficient reward tokens though. |

| | |
|---|---|
| **Comments / Resolution** | Resolved |

| **Issue** | **Upstream referral logic is flawed** |
|---|---|
| **Severity** | High |
| **Description** | The contract is developed in an effort to support upstream referrals, this involves the reward distribution towards the referrer of the referrer.<br>Unfortunately, the implementation is flawed as it will not take the referrer from the referrer but the referrer from the initial caller:<br>referrer = user[msg.sender].referralAddress;<br>This results in the first-tier referrer getting all referral rewards. |
| **Recommendations** | Consider caching the referralAddress from the referrer instead. |
| **Comments / Resolution** | Resolved |

| **Issue** | **Change of rewardRate will change un updated positions** |
|---|---|
| **Severity** | High |
| **Description** | Since the rewardRate change directly changes how much tokens will be accumulated, a change of this rate will also change the tokens received as reward from users that do not have updated their position to date.<br>As an example, if a user has staked 100 AVAX 1 year ago but never |

| | |
|---|---|
| | called getReward during that time, the new rewardRate will now be applied on the whole 1 year period, effectively decreasing/increasing the amount the user will receive. |
| **Recommendations** | Consider simply switching to the synthetix staking rewards logic and then executing updateRewards(address(0)) before the rewardRate change. This will ensure that rewardPerTokenStored is on the updated value:<br>An example can be found here:<br>https://github.com/Synthetixio/synthetix/blob/develop/contracts/StakingRewards.sol#L113 |
| **Comments / Resolution** | Resolved. |

| **Issue** | **Flash-theft possibility due to Ankr architecture** |
|---|---|
| **Severity** | **High** |
| **Description** | The Ankr protocol manually decreases the _ratio parameter once per day. A malicious user can frontrun this change, receive a certToken amount based on the old _ratio parameter and immediately withdraw again with the new _ratio, receiving more ether.<br>This bug is also known to Ankr, but partially prevented since Ankr does not immediately pay out the tokens.<br>However, if the Cairo contract has a pending balance, an attacker can in fact immediately withdraw the desired amount, effectively stealing tokens from the vault.<br>This can even be abused if there are not enough tokens in the contract by artificially increasing the profit, receiving referral rewards |

| | |
|---|---|
| | on that profit. If the amount of deposited wMatic is large enough, this will have a significant impact which can lead up to stealing all Cairo tokens in the contract. |
| **Recommendations** | This issue cannot be easily fixed since Ankr is responsible for this. Therefore, we simply recommend taking a minor fee during each deposit from the initial amount. |
| **Comments / Resolution** | Resolved. |

| Issue | Last withdrawal might revert due to insufficient bond tokens |
|---|---|
| **Severity** | High |
| **Description** | Within the withdrawal logic of the Ankr contract, during the last transaction of the whole flow, the corresponding bondTokens for the AVAX value are being burned:<br>uint256 shares = _bondsToSharesCeil(amount);<br>_burn(from, shares);<br>The slight problem here is that Ankr is rounding against the favor of the user within the _bondsToShares function, which will round up the amount of bondTokens to be burned, this is considered as a general best-practice which is often applied when it comes to burning receipt tokens.<br>This can result in slightly more bonds to be burned than the user has allocated as balance, potentially DoS'ing the withdrawal due to an insufficient balance of CertTokens in the contract. |
| **Recommendations** | Consider slightly decreasing the underlyingAmount parameter for the claimCert function, to ensure that not more tokens are burned than the user has allocated. |

| | |
|---|---|
| **Comments / Resolution** | Resolved. |

| **Issue** | **Using initial ETH balance instead of accumulated value for reward calculation** |
|---|---|
| **Severity** | **Medium** |
| **Description** | Within the reward calculation, user[_account].balance is used to calculate the desired reward.<br>However, this will NOT represent the real accumulated value through the ANKR reward accrue. |
| **Recommendations** | Consider using the underlyingBalance, as this represents the CertToken value, which will be decreased for newer deposits with the same initial value. |
| **Comments / Resolution** | Resolved |

| **Issue** | **_unsafeTransfer does not force success** |
|---|---|
| **Severity** | **Medium** |
| **Description** | The aforementioned function can silently revert, leaving users without their claimed tokens while deducting their claimable amount and fee (once it is fixed in the updated commit). |
| **Recommendations** | Consider forcing success to be true. |
| **Comments / Resolution** | Resolved. |

| Issue | _treasuryWallet can be set to address(0) |
|---|---|
| **Severity** | Medium |
| **Description** | Whenever the aforementioned variable is set to address(0), this will result in a permanent loss of all eth which is claimed. |
| **Recommendations** | Consider implementing a validation that prevents this scenario. |
| **Comments / Resolution** | Resolved. |

| Issue | Owner cannot be changed |
|---|---|
| **Severity** | Medium |
| **Description** | The owner is set once during contract deployment and can call many privileged functions as well as receiving the tokens which are recovered.<br>However, the owner cannot be changed which can result in undesired side-effects for example in the scenario of compromised keys. |
| **Recommendations** | Consider simply inheriting the Ownable library or implementing a transferOwnership function. |
| **Comments / Resolution** | Resolved |

| Issue | Referrer can be changed |
|---|---|
| Severity | Low |
| Description | Generally speaking, once a referrer is set, it should not be changeable. Moreover, it should not be possible to set the own address as a referrer. |
| Recommendations | Consider only setting the referrer during the very first deposit, moreover a self-address check should be implemented. |
| Comments / Resolution | Resolved |

| Issue | Architecture: Claiming happens on a first-come-first-serve basis |
|---|---|
| Severity | Low |
| Description | Whenever a user initiates a withdrawal, this will then be scheduled within the Ankr staking contract and the withdrawal will happen at an arbitrary time.<br>This can result in issues for the following example:<br>a) Alice queues a withdrawal of 100 AVAX<br>b) The withdrawal is being executed within the Ankr staking contract, the Cairo contract now has 100 AVAX<br>b) Bob queues a withdrawal of 100 AVAX in the same block as c) and immediately claims the tokens<br>Alice therefore needs to wait for the next withdrawal from ANKR to be executed. |
| Recommendations | Whi This issue is an architectural issue and therefore a fix would involve modifying the whole logic. |

|  |  |
|---|---|
| **Comments / Resolution** | Resolved. |

| Issue | isClaimAvailable is flawed |
|---|---|
| **Severity** | Low |
| **Description** | The aforementioned function will return true as long as the contract has a non-zero balance and the user has a valid request.<br>However, this is incorrect as the claim will only be available when the contract balance covers the request amount. |
| **Recommendations** | Consider implementing such a check. |
| **Comments / Resolution** | Resolved. |

## 2nd Audit:

Following our initial audit, which revealed several high-risk issues, it became evident that a simple resolution round would not suffice to meet the rigorous standards required for a safe and robust system. Acknowledging the critical nature of these findings, we embarked on a comprehensive revision, leading to the introduction of a new audit round with a significantly refactored codebase.

To ensure the highest level of scrutiny and readiness, a full testing suit is required by the development team, which was not yet provided. This is mandatory before mainnet deployment.

**Disclaimer:**
Bailsec is not responsible for any rounding errors which occur due to different mathematical operations within ANKR and the CairoStaking contract. Sufficient tests should be executed in an effort to ensure the correctness. Moreover, only the Avalanche, Ethereum and ETHMatic contracts have been audited in this round.

Since the post-audit contract contains several vulnerabilities, it goes without saying that a validation of the fixes is necessary. However, for this round, a simple diff-checker comparison might be sufficient, Bailsec can offer this service for a small nominal fee.

Note that this audit is solely for the Avalanche, Ethereum and ETHMatic contracts in the following repository:

https://github.com/cairofinanceapp/Farm/tree/cff7fe63fdd6f3e148b6505fb87cb852afbee443

Requested tests have not yet been provided by the customer and should be provided before mainnet deployment.

### CairoStaking (Avalanche)

The business logic for this contract stays mainly the same as with the current implementation, therefore we refer to the aforementioned description.

The following ANKR smart contract implementations are currently used:

AvalanchePool_R6:
https://snowtrace.io/address/0xfad2d6dc790b6a12f7117b2965755cfabbb45874#code

FutureBondAVAX_R8:
https://snowtrace.io/address/0x0f74bfd1a33471641d86d9c7d468694e0b0a9be3#code

FutureCertAVAX_R3:
https://snowtrace.io/address/0xfc8d81a01ded207ad3deb4fe91437cae52ded0b5#code

***Eventual issues might apply if the proxy upgrades the implementations. This audit is solely focused on the aforementioned implementation contracts.**

| Issue | Oracle is out of scope |
|---|---|
| **Severity** | High |
| **Description** | The used oracle is out of scope, the likelihood of an exploit for these auxiliaries is high, which can result in a loss of funds. |
| **Recommendations** | We recommend audit coverage for the oracle as well. |
| **Comments / Resolution** | Resolved, this has been changed and instead of Cairo, AVAX is now used as referral reward.<br>**High risk 1:**<br>**However**, when the claimReferRewards function is invoked, the user will lose all his rewards in the scenario that the contract does not hold sufficient tokens:<br>function claimReferRewards() external {<br>uint256 rew = rewards[msg.sender].available;<br>require(rew != 0, "No rewards available");<br>require(rewardsToken.balanceOf(address(this)) >= rew, "Wait for rewards to be available");<br>rewards[msg.sender].available = 0;<br>_unsafeTransfer(msg.sender, rew, true); |

```
        rewards[msg.sender].claimed += rew;
        emit ReferRewardsClaimed(msg.sender, rew);
    }
```

As one can see, the highlighted line still checks if the contract has sufficient Cairo tokens, while in fact it should check for a sufficient AVAX balance. The reason why the user will lose all his rewards is that there is no success validation of the _unsafeTransfer call, since the assembly low-level call be executed:

```
    function _unsafeTransfer(address receiverAddress, uint256
amount, bool limit) internal virtual returns (bool) {
        address payable wallet = payable(receiverAddress);
        bool success;
        if (limit) {
            assembly {
                success := call(27000, wallet, amount, 0, 0, 0, 0)
            }
            return success;
        }
        (success,) = wallet.call{value: amount}("");
        require(success == true, "Send Eth reverted");
        return success;
    }
```

Resolved, a balance check is now enforced as well as a success requirement

| Issue | Malicious user can steal all Cairo tokens |
|---|---|
| Severity | High |
| Description | This sophisticated exploit lies within a logical flaw within the _payRefs call. Within a withdrawal, the _payRefs function is called with the whole withdrawable amount:<br>_payRefs(_unAmount); |

| | |
|---|---|
| | This means that 10% of the whole withdrawable amount will be distributed to referrers, as Cairo token, while the unstake fee is only calculated on the profit which was made.<br>A user can for example deposit 100_000 AVAX, wait until the price appreciation of the certificate token becomes 100_000 AVAX + 1 WEI. Therefore, a user will only pay 30% unstakeFee on 1 WEI (profit), effectively zero, while receiving 10% of 100_000 AVAX as Cairo tokens. |
| **Recommendations** | The _unAmount calculation should be based on the real unstake fee instead of the whole profit. |
| **Comments / Resolution** | Resolved, first of all, the _payRefs function is now invoked within the claimUnstakeAmount function, previously it was invoked within the _checkReferReward function, during a withdrawal. Moreover, the correct value, aka the unstakeFee is now used to determine how much rewards should be distributed. |

| **Issue** | **Flaw in synthetix reward logic** |
|---|---|
| **Severity** | High |
| **Description** | Within the first audit, we have recommended the client to switch to the synthetix staking rewards logic, using the certificate tokens as basic.<br>Unfortunately, this logic was wrongly implemented:<br>modifier updateReward(address account) {<br>    lastUpdateTime = block.timestamp;<br>    rewardPerTokenStored = rewardPerToken();<br>    if (account != address(0)) {<br>        user[account].rewards = earned(account);<br>        user[account].userRewardPerTokenPaid = |

| | |
|---|---|
| | rewardPerTokenStored;<br>}<br><br>_;<br>}<br><br>The lastUpdateTime is always updated before the rewards are being calculated, thus the contract will never accumulate rewards. |
| **Recommendations** | We recommend following 1:1 the synthetix staking rewards logic:<br><br>https://github.com/Synthetixio/synthetix/blob/develop/contracts/StakingRewards.sol<br><br>Moreover, the lastUpdateTime should be set within the constructor.<br><br>A fix for this issue will only be accepted, if proper testing documentation either via hardhat or foundry was provided, with different time-sensitive tests.<br><br>Once this issue is fixed and extensive tests have been provided, an additional peer review for this logic is necessary by Bailsec. |
| **Comments / Resolution** | Partially resolved, **extensive tests are yet to be expected in the corresponding github repository.** |

| **Issue** | **Claim might revert due to flawed check in isClaimAvailable** |
|---|---|
| **Severity** | **High** |
| **Description** | The isClaimAvailable is executing the following check:<br><br>address(this).balance - feeCollected) >= requests[_user].amount<br><br>However, this check is using the full amount instead of the amount deducted the fee. |

| | |
|---|---|
| | Consider the following PoC to illustrate this issue:<br>1. Alice deposits 100 AVAX<br>2. The deposit generates 10 AVAX in fees<br>3. Alice withdraws 110 AVAX<br>a. requests[msg.sender].amount += 110<br>b. requests[msg.sender].fee += 3<br>4. The contract now has 110 AVAX as balance<br>5. User calls claimUnstakeAmount, the following check will revert<br>(address(this).balance - feeCollected) >= requests[_user].amount<br>110 - 3 >= 110<br>The claim is effectively locked, while the user should pratically be able to execute the claim. |
| **Recommendations** | We recommend using the amount, deducted the fee, instead of the amount. |
| **Comments / Resolution** | Resolved, since the feeCollected is now only increased after the isClaimed check, this issue cannot occur anymore. |

| **Issue** | **Inconsistency for msg.value conversion** |
|---|---|
| **Severity** | **Medium** |
| **Description** | Whenever a user stakes AVAX, the Cairo Staking contract will receive the corresponding amount of certificate tokens.<br>However, the user[msg.sender].underlyingBalance is increased by unAmount (deducted the fee), which might expose a problem due to a potential incorrect mathematical expression.<br>The Cairo contract calculates the received amount of certificate |

| | |
|---|---|
| | token as follows:<br><br>unAmount = (unAmount * msg.value) / 1e18;<br><br>while the underlying balancer contract uses the following expression:<br><br>function safeMultiplyAndDivide(uint256 a, uint256 b, uint256 c) internal pure returns (uint256) {<br><br>uint256 reminder = a.mod(c);<br>uint256 result = a.div(c);<br>bool safe;<br>(safe, result) = result.tryMul(b);<br>if (!safe) {<br>return 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff;<br>}<br>(safe, result) = result.tryAdd(reminder.mul(b).div(c));<br>if (!safe) {<br>return 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff;<br>}<br>return result;<br>}<br><br>While both calculations do the exact same thing, there still might be discrepancies, which can result in issues, as example if the unAmount calculation is slightly rounded up against the real received amount of certificate tokens, this will result in a potential DoS within withdraw, since the user attempts to withdraw slightly more tokens than the contract has received (we assume that fees already have been withdrawn, otherwise the withdrawal with succeed but the fees cannot be withdrawn). |
| **Recommendations** | We recommend applying the same mathematical calculation as used within the underlying contract. A simple look into the ANKR contract indicates that the _bondsToShares function is handling this logic. |

| | |
|---|---|
| | Moreover, corresponding tests should be applied to ensure the correctness.<br>A fix for this issue will only be accepted with sufficient tests, using multiple different values. |
| **Comments / Resolution** | Partially resolved, **corresponding tests are yet to be expected in the github repository.** |

| Issue | User might receive more AVAX during withdraw than actually queued |
|---|---|
| **Severity** | Medium |
| **Description** | Whenever a user attempts to withdraw tokens, the underlying amount of certificate tokens is transferred to ANKR and the corresponding AVAX amount will be eventually transferred to Cairo.<br>The developer implemented decreasing the underlyingAmount due to our recommendation. However, the request[msg.sender].amount variable is still increased using the un-decreased underlyingAmount.<br>Thus a user will receive slightly more AVAX as requested balance than the contract received.<br>This will result in a potential DoS for the last user that withdraws since the contract does not have enough AVAX to honor this withdrawal. |
| **Recommendations** | We recommend using the decreased underlyingAmount for the currentAmount conversion. |
| **Comments / Resolution** | Resolved. However, potential side-effects might occur due to the adjusted decrease of totalUnderlyingToken. Tests should be executed for that scenario |

| Issue | Lack of validation for different settings |
|---|---|
| **Severity** | Medium |
| **Description** | Multiple variables influence the contract's business logic, if these are ever setted wrongly, it can result in a loss of funds or denial of service, the following variables are affected:<br>a) minimumUnstake<br>b) rewardRate<br>c) depositFee<br>e) cooldownTime |
| **Recommendations** | We recommend setting an appropriate limit for these variables. |
| **Comments / Resolution** | Failed resolution, the check for the depositFee is totally flawed as the feeMax is 100_000 and the depositFee is always a partial from it. Assuming it is desired to have a 1% depositFee, this would mean a depositFee of 1_000. Given the current validation:<br>function setDepositFee(uint256 _fee) external onlyOwner {<br>    require(_fee/FEE_MAX <= 30,"Invalid Fee");<br>    depositFee = _fee;<br>}<br>it is required that the depositFee can be up to 3_000_000, which then would be a real depositFee of 3000%, totally pointless.<br>Resolved, a maximum deposit fee of 3% is now enforced within the setDepositFee function. |

| Issue | Unstake might revert due to fee deduction within deposit |
|---|---|
| **Severity** | Low |
| **Description** | Whenever a user deposits, he will receive less certificate tokens due to the applied fee. This *can* potentially result in a revert within the withdraw function due to the minimum unstake check, since underlyingAmount is lower than the corresponding provided msg.value. |
| **Recommendations** | We recommend preventing this edge-case by setting a reasonable unstake limit, which is smaller than the corresponding stake limit. |
| **Comments / Resolution** | Resolved, it is expected that the developer sets a reasonable unstake limit. |

| Issue | RewardToken should be 1e18 |
|---|---|
| **Severity** | Low |
| **Description** | Whenever the reward token is for example USDT with 6 decimals, this will break the referral logic, since a way larger $ value is being allocated than expected. |
| **Recommendations** | This issue was only rated as low since we assume that the Cairo token has 18 decimals. However, we highly recommend keeping this issue in mind, whenever this contract is used with another reward token. |
| **Comments / Resolution** | Resolved. |

## CairoStaking (Ethereum)

The business logic for this contract stays mainly the same as with the current implementation, therefore we refer to the aforementioned description.

The following ANKR smart contract implementations are currently used:

GlobalPool_R46:
https://etherscan.io/address/0xecce8778214fd9fe37c141a00cff19853ef5bc4a#code

AETH_R21:
https://etherscan.io/address/0xe672e0e0101a7f58d728751e2a5e6da5ff1fda64#code

It is important to note that these implementations can change with time, which can result in integration issues.

The Ethereum implementation has slight differences compared to the Avalanche implementation such as that the deposit fee is taken directly in ETH and the minimum stake and unstake parameters have been removed.

***Eventual issues might apply if the proxy upgrades the implementations. This audit is solely focused on the aforementioned implementation contracts.**

| Issue | Oracle is out of scope |
|---|---|
| **Severity** | High |
| **Description** | The used oracle is out of scope, the likelihood of an exploit for these auxiliaries is high, which can result in a loss of funds. |
| **Recommendations** | We recommend audit coverage for the oracle as well. |
| **Comments / Resolution** | Resolved, this has been changed and instead of Cairo, AVAX is now used as referral reward.<br>**High risk 1:**<br>**However**, when the claimReferRewards function is invoked, the user will lose all his rewards in the scenario that the contract does not |

hold sufficient tokens:

```
    function claimReferRewards() external {
        uint256 rew = rewards[msg.sender].available;
        require(rew != 0, "No rewards available");
        require(rewardsToken.balanceOf(address(this)) >= rew, "Wait for rewards to be available");
        rewards[msg.sender].available = 0;
        _unsafeTransfer(msg.sender, rew, true);
        rewards[msg.sender].claimed += rew;
        emit ReferRewardsClaimed(msg.sender, rew);
    }
```

As one can see, the highlighted line still checks if the contract has sufficient Cairo tokens, while in fact it should check for a sufficient ETH balance. The reason why the user will lose all his rewards is that there is no success validation of the _unsafeTransfer call, since the assembly low-level call is executed:

```
    function _unsafeTransfer(address receiverAddress, uint256 amount, bool limit) internal virtual returns (bool) {
        address payable wallet = payable(receiverAddress);
        bool success;
        if (limit) {
            assembly {
                success := call(27000, wallet, amount, 0, 0, 0, 0)
            }
            return success;
        }
        (success,) = wallet.call{value: amount}("");
        require(success == true, "Send Eth reverted");
        return success;
    }
```

Resolved.

| Issue | Malicious user can steal all Cairo tokens |
|---|---|
| **Severity** | High |
| **Description** | This sophisticated exploit lies within a logical flaw within the _payRefs call. Within a withdrawal, the _payRefs function is called with the whole withdrawable amount:<br>_payRefs(_unAmount);<br>This means that 10% of the whole withdrawable amount will be distributed to referrers, as Cairo token, while the unstake fee is only calculated on the profit which was made.<br>A user can for example deposit 100 ETHER, wait until the price appreciation of the certificate token becomes 100 ETHER + 1 WEI.<br>Therefore, a user will only pay 30% unstakeFee on 1 WEI (profit), effectively zero, while receiving 10% of 100 ETHER as Cairo tokens. |
| **Recommendations** | The _unAmount calculation should be based on the real unstake fee instead of the whole profit. |
| **Comments / Resolution** | Resolved, first of all, the _payRefs function is now invoked within the claimUnstakeAmount function, previously it was invoked within the _checkReferReward function, during a withdrawal. Moreover, the correct value, aka the unstakeFee is now used to determine how much rewards should be distributed. |

| Issue | User might receive more ETHER during withdraw than actually queued |
|---|---|
| **Severity** | Medium |
| **Description** | Whenever a user attempts to withdraw tokens, the underlying amount of certificate tokens is transferred to ANKR and the corresponding ETHER amount will be eventually transferred to Cairo.<br>The developer implemented decreasing the underlyingAmount due to our recommendation. However, the request[msg.sender].amount variable is still increased using the un-decreased underlyingAmount.<br>Thus a user will receive slightly more ETHER as requested balance than the contract received.<br>This will result in a potential DoS for the last user that withdraws since the contract does not have enough ETHER to honor this withdrawal. |
| **Recommendations** | We recommend using the decreased underlyingAmount for the currentAmount conversion. |
| **Comments / Resolution** | Resolved. However, potential side-effects might occur due to the adjusted decrease of totalUnderlyingToken. **Tests should be executed for that scenario** |

| Issue | Lack of validation for different settings |
|---|---|
| **Severity** | Medium |
| **Description** | Multiple variables influence the contract’s business logic, if these are ever setted wrongly, it can result in a loss of funds or denial of service, the following variables are affected:<br>a) rewardRate |

| | |
|---|---|
| | b) depositFee<br>c) cooldownTime |
| **Recommendations** | We recommend setting an appropriate limit for these variables. |
| **Comments / Resolution** | Failed resolution, the check for the depositFee is totally flawed as the feeMax is 100_000 and the depositFee is always a partial from it. Assuming it is desired to have a 1% depositFee, this would mean a depositFee of 1_000. Given the current validation:<br><br>`function setDepositFee(uint256 _fee) external onlyOwner {`<br>`    require(_fee/FEE_MAX <= 30,"Invalid Fee");`<br>`    depositFee = _fee;`<br>`}`<br><br>it is required that the depositFee can be up to 3_000_000, which then would be a real depositFee of 3000%, totally pointless.<br><br>Resolved. |

| Issue | RewardToken should be 1e18 |
|---|---|
| **Severity** | Low |
| **Description** | Whenever the reward token is for example USDT with 6 decimals, this will break the referral logic, since a way larger $ value is being allocated than expected. |
| **Recommendations** | This issue was only rated as low since we assume that the Cairo token has 18 decimals. However, we highly recommend keeping this issue in mind, whenever this contract is used with another reward token. |
| **Comments / Resolution** | Resolved |

| Issue | Claiming does not check for outstanding fee |
|---|---|
| Severity | Low |
| Description | Whenever the claimUnstakeAmount function is called, there is no check to ensure that the contract has sufficient protocol fees after the amount has been withdrawn. |
| Recommendations | We recommend a check for the contract balance + protocol fees against the _totalAmount to be withdrawn. |
| Comments / Resolution | Failed resolution. While in fact the recommended fix was implemented, it now needs to be checked for the outstanding referrer rewards as well, due to the logical change.<br><br>However, since this issue is just a low severity and this codebase underwent already multiple resolution rounds, this can be simply kept as-is. |

| Issue | Profit is applied without accounting deposit fee in |
|---|---|
| Severity | Low |
| Description | Whenever a user deposits ETHER, the balance is increased as follows:<br><br>user[msg.sender].balance += _amount;<br><br>This will result in the position becoming instantly profitable, after the first ratio decrease, applying a fee on a profit which is not reflecting the real profit, since a fee has been paid.<br><br>The issue lies within the balance increase using the amount minus the fee. The real behavior should be to increase the balance by the real |

| | |
|---|---|
| | deposited msg.value such that a profit is only calculated once the withdrawn amount in fact exceeds the initially deposited value. |
| **Recommendations** | This can be fixed easily by increasing the amount by msg.value. However, since this does not pose large harm to users, it can be acknowledged as well. |
| **Comments / Resolution** | Acknowledged. |

## CairoStaking (ETHMatic)

The business logic for this contract stays mainly the same as with the current implementation, therefore we refer to the aforementioned description.

The following ANKR smart contract implementations are currently used:

PolygonPool_R6:
https://etherscan.io/address/0xcb6805e51ea42741d17d1c1f59e01fbe80aba389#code
aMATICb_R6 (Bond):
https://etherscan.io/address/0xd4502103dd36c5595dccedf33e7308c61428ce3b#code

aMATICc_R3 (Cert):
https://etherscan.io/address/0x50be7ae35c5bf838d060045f33f93449f9aff49c#code

It is important to note that these implementations can change with time, which can result in integration issues.

The ETHMatic implementation has slight differences compared to the Avalanche implementation such as that a wrapped token (WMatic) is used for the deposit, a minimumStake value which is fetched from the ANKR contract, is applied and an unstake fee is necessary for unstaking tokens, which is payable towards the ANKR protocol.

***Eventual issues might apply if the proxy upgrades the implementations. This audit is solely focused on the aforementioned implementation contracts.**

<table>
<tr><th>Issue</th><th>Oracle is out of scope</th></tr>
<tr><td><b>Severity</b></td><td>High</td></tr>
<tr><td><b>Description</b></td><td>The used oracle is out of scope, the likelihood of an exploit for these auxiliaries is high, which can result in a loss of funds.</td></tr>
<tr><td><b>Recommendations</b></td><td>We recommend audit coverage for the oracle as well.</td></tr>
<tr><td><b>Comments / Resolution</b></td><td>Resolved, instead of the Cairo token, WMATIC is now used as referral reward, this allowed for the removal of the oracle logic.

However, the shift resulted in some issues:

<b>Informational Issue 1:</b>

<pre>
    function claimReferRewards() external  nonReentrant{
        uint256 rew = rewards[msg.sender].available;
        require(rew != 0, "No rewards available");
        require(rewardsToken.balanceOf(address(this)) >= rew, "Wait for rewards to be available");
        rewards[msg.sender].available = 0;
        IERC20Extented(WMATIC).safeTransfer(msg.sender, rew);
        rewards[msg.sender].claimed += rew;
        emit ReferRewardsClaimed(msg.sender, rew);
    }
</pre>

As one can see in the highlighted section, the check is still existent for the Cairo token, however, it should in fact be for the WMATIC token. Contrary to the first contract, this transfer will just revert instead of tokens being lost here. However, we still recommend checking for the WMATIC balance instead of the Cairo balance.

Resolved.</td></tr>
</table>

| Issue | Malicious user can steal all Cairo tokens |
|---|---|
| Severity | High |
| Description | This sophisticated exploit lies within a logical flaw within the _payRefs call. Within a withdrawal, the _payRefs function is called with the whole withdrawable amount:<br>_payRefs(_unAmount);<br>This means that 10% of the whole withdrawable amount will be distributed to referrers, as Cairo token, while the unstake fee is only calculated on the profit which was made.<br>A user can for example deposit 100_000 WMATIC, wait until the price appreciation of the certificate token becomes 100_000 AVAX + 1 WEI. Therefore, a user will only pay 30% unstakeFee on 1 WEI (profit), effectively zero, while receiving 10% of 100_000 WMATIC as Cairo tokens. |
| Recommendations | The _unAmount calculation should be based on the real unstake fee instead of the whole profit. |
| Comments / Resolution | Resolved, a withdrawal now calls the _checkReferRewards which keeps track of the requests[msg.sender].fee variable and increases it by a percentage of the profit. This variable exactly determines how much fee was taken during a withdrawal. This variable is then used to allocate the referral reward within the claimUnstakeAmount function. |

| Issue | Insonsistency in WMATIV -> Certificate token conversion |
|---|---|
| Severity | Medium |
| Description | Whenever a user deposits WMATIC, the corresponding underlying amount of certificate tokens is calculated as follows: |

| | |
|---|---|
| | uint256 unAmount = getRatioConversion(_amount);<br>which follows this expression:<br>function getRatioConversion(uint256 _amount) public view returns (uint256) {<br>uint256 rate = underlyingToken.ratio();<br>uint256 ratio = (rate * _amount) / 1e18;<br>return ratio;<br>}<br>However, this is not exactly the way the ANKR contract calculates the to be minted Certificate tokens, as this is done using the following expression:<br>function _bondsToShares(uint256 amount) internal view returns (uint256) {<br>return safeCeilMultiplyAndDivide(amount, _ratio, 1e18);<br>}<br>This can result in discrepancies where users might get more certificate tokens accounted, than they in fact receive. |
| **Recommendations** | We recommend using the same expression as within ANKR to calculate the unAmount. |
| **Comments / Resolution** | Resolved |

| Issue | User might receive more WMATIC during withdraw than actually queued |
|---|---|
| **Severity** | Medium |
| **Description** | Whenever a user attempts to withdraw tokens, the underlying amount of certificate tokens is transferred to ANKR and the corresponding WMATIC amount will be eventually transferred to Cairo.<br>The developer implemented decreasing the underlyingAmount due to our recommendation. However, the request[msg.sender].amount variable is still increased using the un-decreased underlyingAmount.<br>Thus a user will receive slightly more WMATIC as requested balance than the contract received.<br>This will result in a potential DoS for the last user that withdraws since the contract does not have enough WMATIC to honor this withdrawal. |
| **Recommendations** | We recommend using the decreased underlyingAmount for the currentAmount conversion. |
| **Comments / Resolution** | Resolved, the requests[msg.sender].amount variable is now decreased using the update underlyingAmount value.<br>However, potential side-effects might occur due to the adjusted decrease of totalUnderlyingToken. **Tests should be executed for that scenario.** |

| Issue | Lack of validation for different settings |
|---|---|
| **Severity** | Medium |
| **Description** | Multiple variables influence the contract’s business logic, if these are ever setted wrongly, it can result in a loss of funds or denial of service, the following variables are affected:<br><br>a) rewardRate<br>b) depositFee<br>e) cooldownTime |
| **Recommendations** | We recommend setting an appropriate limit for these variables. |
| **Comments / Resolution** | Failed resolution, the check for the depositFee is totally flawed as the feeMax is 100_000 and the depositFee is always a partial from it. Assuming it is desired to have a 1% depositFee, this would mean a depositFee of 1_000. Given the current validation:<br><br>`function setDepositFee(uint256 _fee) external onlyOwner {`<br>`    require(_fee/FEE_MAX <= 30,"Invalid Fee");`<br>`    depositFee = _fee;`<br>`}`<br><br>it is required that the depositFee can be up to 3_000_000, which then would be a real depositFee of 3000%, totally pointless.<br><br>Resolved. |

| Issue | RewardToken should be 1e18 |
|---|---|
| **Severity** | Low |
| **Description** | Whenever the reward token is for example USDT with 6 decimals, this will break the referral logic, since a way larger $ value is being allocated than expected |
| **Recommendations** | This issue was only rated as low since we assume that the Cairo token has 18 decimals. However, we highly recommend keeping this issue in mind, whenever this contract is used with another reward token. |
| **Comments / Resolution** | Resolved |