Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Veridise Inc.

August 19, 2022

► **Prepared For:**

AlloyX
https://www.alloyx.xyz/

► **Prepared By:**

Bryan Tan
Xiangan He
Himanshu
Ben Mariano
Hongbo Wen

► **Contact Us:** contact@veridise.com

► **Version History:**

August 19, 2022 V1
August 5, 2022 Draft

Contents

1 Executive Summary

From July 13 to July 29, AlloyX engaged Veridise to review the security of their AlloyX Decentralized Autonomous Organization (DAO), a liquid staking protocol running on top of the Ethereum blockchain. The audit covered the Solidity smart contracts that the protocol consists of, which included implementations of the DURA and CRWN tokens defined by the protocol, "desk" contracts for interacting between the AlloyX DAO and third-party protocols (such as the Goldfinch lending protocol), and a few internal AlloyX DAO contracts for whitelisting users and tracking staking rewards. Veridise conducted this assessment over 10 person-weeks, with 5 engineers working on code from commit `5f8c250` to `7606eac` of the `staging` branch of the `https://github.com/AlloyXChange/alloyx-smart-contracts-v2` repository. The auditing strategy involved tool-assisted analysis of the source code performed by Veridise engineers. The tools used in the audit included a combination of static analysis and formal verification.

Summary of issues detected. The audit uncovered 23 issues. The most severe issues included two business logic attack vectors involving bugs in the staking logic and treasury logic; as well as access control issues in an internal contract used to rank Goldfinch tranches. In addition to these issues, auditors also identified other potential issues regarding the interaction of the contracts with the protocol – in these cases, improper handling of this interaction could lead to protocol disruptions. Additionally, our auditors also discovered multiple gas optimizations and code suggestions which can help improve the efficiency, safety, and maintainability of the code.

Code assessment. The smart contracts being assessed for this audit implement the functionality of the AlloyX DAO as described in the AlloyX whitepaper. In general, our auditors feel that the code quality is above average; the contracts are well-organized into separate modules focusing on specific pieces of functionality, and the contract methods are documented with comments. Furthermore, the contracts are not a fork of an existing protocol but appear to have been written by the developers from scratch. The contracts make good use of the well-known and audited contracts from OpenZeppelin. The code also includes testing of the contracts using the Hardhat framework; however, the tests only provide partial coverage of the contract behaviors.

Recommendations. As two high-severity issues and two medium-severity issues are related to errors in business logic, the AlloyX developers should ensure that all contracts in their protocol are covered by automated testing, and that they test the functionality of the protocol with respect to both intended and unintended behaviors. For example, our auditors uncovered a high-severity issue in the `AlloyxStakeInfo.sol` contract that would have been caught by adequate unit test coverage.

Secondly, the AlloyX developers should remain aware of the issues reported during this audit that have not been fully addressed (e.g., issues that have been marked as "Acknowledged", or issues marked as "Fixed" with an "Update" or "Developer Response"). While these issues are

sufficiently mitigated or of low priority in the short term, they may cause problems in the future as the developers continue to update the AlloyX smart contracts.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions, nor that the findings in this report will remain valid if the system is modified in the future. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
	`5f8c250 - 7606eac`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
July 13 - July 29, 2022	Manual & Tools	5	10 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	3	3
Medium-Severity Issues	4	4
Low-Severity Issues	4	4
Warning-Severity Issues	7	7
Informational-Severity Issues	5	4
TOTAL	23	22

Table 2.4: Category Breakdown.

Name	Number
Access Control	2
Data Validation	3
Denial of Service	1
Gas Optimization	1
Logic Error	5
Maintainability	7
Missing/Incorrect Events	2
Reentrancy	1
Transaction Ordering	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the smart contracts implementing the AlloyX protocol, as well as writing formal specifications that could automatically be checked by tooling provided by Veridise. In our audit, we sought to answer the following questions regarding these contracts:

- Do the contracts have appropriate access control? In particular, are whitelisted users able to access the resources they're authorized to do so? Are calls from unauthorized users reverted? Does whitelisting on Alloy conflict in any way with or depend on Goldfinch whitelists as a potential attack vector?
- Can users abuse token exchange rates and other tokenomics-related functionality with flashloans, frontrunning, or other similar attack vectors?
- Are there any gas-related, protocol breaking issues in updating the configuration or running code in loops? In what ways, if any, can configuration updates cause business logic errors or break user-accessible functions?
- Are inputs properly verified/sanitized? That is, can a malicious user cause unwanted behavior by sending carefully crafted requests?
- Are events appropriately emitted to ensure that the protocol can track relevant actions? Can a malicious user alter the order/timing of events to inappropriately manipulate the behavior of the protocol?
- Are there any current pieces of dead or unused code, and can they potentially serve as entrypoint vulnerabilities? Is it possible to extend code test coverage in any way, and where? Were there any pieces of code in follow-up patches that introduced new issues trying to fix existing ones?
- Are the Alloy contracts correctly interacting with the third-party contracts such as the Goldfinch contracts?
- Are the staking reward calculations performed correctly? In particular, are there ways that malicious users can gain excessive rewards or destroy other users' rewards?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis and testing tools. In particular, we conducted our audit with the aid of the following techniques:

- Static Analysis. We leveraged the open-source tool Slither to find common vulnerabilities in smart contracts, such as reentrancy and uninitialized variables. In addition to the default behavior of Slither, we also extended Slither with custom detectors for more precise and expansive bug detection.

- Formal Verification. We used an internal version of Veridise's open source tool Eurus* which enables symbolic reasoning about smart contracts. Our engineers wrote a set of pre/post conditions (i.e. specifications) for each function of interest in the smart contracts and used Eurus to formally verify that the functions satisfy the specifications.

Scope. To understand the scope of the audit, we first reviewed the documentation shared by the AlloyX developers, including the online documentation and whitepaper. AlloyX's GoldfinchDesk contract interacts heavily with Goldfinch's contracts, so we also looked at some of the relevant Goldfinch contracts and the Goldfinch whitepaper. Afterwards, we carefully reviewed the AlloyX code for bugs and security issues. As AlloyX fixed bugs in response to our findings, we also reviewed any of the fixes they made to check that they did not introduce additional issues.

We also ran some of the tests provided by Alloy in the `v7.0` branch (as none were found in the `staging` branch we were auditing), but the tests only provided partial test coverage. This environment was insufficient for us to prototype proof-of-concept exploits. Thus, we instead set up a custom test environment based on the Foundry tool, which we used to successfully demonstrate several concrete attacks.

Finally, the AlloyX developers asked us to write formal specifications where possible. While performing the code review, we observed that formal verification would be most effectively applied to the `AlloyxStakeInfo` contract; the other contracts either were too simple (low cost-effectiveness of formal verification) or involved many interactions with external contracts (requires too many assumptions that make writing correct formal specifications to be difficult). During review of the `AlloyxStakeInfo` contract, the Veridise auditors noted down any potential contract invariants as well as function preconditions and postconditions. These were then transformed into formal properties, which were then checked by our Eurus formal verification tool. A description of these properties can be found in Chapter 5.

Concretely, the contracts considered in this audit were the following:

- AdminUpgradeable.sol
- AlloyxConfig.sol
- AlloyxExchange.sol
- AlloyxStakeInfo.sol
- AlloyxTokenCRWN.sol
- AlloyxTokenDURA.sol
- AlloyxTreasury.sol
- AlloyxWhitelist.sol
- ConfigHelper.sol
- ConfigOptions.sol
- GoldfinchDesk.sol
- SortedGoldfinchTranches.sol
- StableCoinDesk.sol
- StakeDesk.sol
- SwapTokens.sol†

* `https://github.com/Veridise/Eurus`
† The developers noted that this contract is to be removed in the future, so our auditors did not look at this contract.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows:

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows:

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-ALL-VUL-001	Business Logic Errors In Tracking Stakeholders	High	Fixed
V-ALL-VUL-002	Treasury Fee Variables Can Only Increase	High	Fixed
V-ALL-VUL-003	Lack of Access Controls in SortedGoldfinchTranches	High	Fixed
V-ALL-VUL-004	claimRewards Can Revert if Config is Wrong	Medium	Fixed
V-ALL-VUL-005	Potential Reentrancy Vulnerability	Medium	Fixed
V-ALL-VUL-006	Potential Stake Reward Inconsistency	Medium	Acknowledged
V-ALL-VUL-007	Bugs in newly added pausing mechanism	Medium	Fixed
V-ALL-VUL-008	Missing Interface Inheritance I	Low	Fixed
V-ALL-VUL-009	Potential Upgrade Race Condition	Low	Fixed
V-ALL-VUL-010	Missing Interface Inheritance II	Low	Fixed
V-ALL-VUL-011	removeWhitelistedUser Cannot Remove Address	Low	Fixed
V-ALL-VUL-012	copyFromOtherConfig is Error-Prone	Warning	Fixed
V-ALL-VUL-013	Lack of Zero Address Checks	Warning	Fixed
V-ALL-VUL-014	High Gas Cost For Goldfinch Token Counts	Warning	Acknowledged
V-ALL-VUL-015	Expensive Gas On Config Copying	Warning	Acknowledged
V-ALL-VUL-016	Invariant Involving stake.since Violated	Warning	Acknowledged
V-ALL-VUL-017	Access Control Pitfalls When Expanding Whitelist	Warning	Acknowledged
V-ALL-VUL-018	Invariant Involving totalPastRedeemableReward	Warning	Acknowledged
V-ALL-VUL-019	Multiple Public Functions Can Be Declared External	Info	Fixed
V-ALL-VUL-020	Explicitly Mark State Visibility With Some Variables	Info	Fixed
V-ALL-VUL-021	Events Should Be Emitted in AlloyxTreasury	Info	Fixed
V-ALL-VUL-022	Public Variables in AlloyxConfig	Info	Fixed
V-ALL-VUL-023	More Missing Events	Info	Open

4.1 Detailed Description of Bugs

In this section, we describe each uncovered vulnerability in more detail.

4.1.1 V-ALL-VUL-001: Business Logic Errors In Tracking stakeholders

Severity	High	Commit	5f8c250
Type	Logic Error	Status	Fixed
Files	AlloyxStakeInfo.sol		
Functions	removeStake, removeStakeholder		

Description `AlloyxStakeInfo` uses two state variables `stakeholderMap` and `stakesMapping` to track information related to stakeholders. However, the logic does not appear to be consistent with the conceptual idea of who a "stakeholder" is.

For example, the `removeStake` method will add the staker as a stakeholder. Consequently, an account can never be marked as "not a stakeholder" in `stakeholderMap` .

```
function removeStake(address _staker, uint256 _stake) public onlyAdmin {
  require(stakeOf(_staker).amount >= _stake, "User has insufficient dura coin staked"
    );
  if (stakesMapping[_staker].amount == 0) addStakeholder(_staker); //@audit-issue
    stakeholder added on remove
  addPastRedeemableReward(_staker, stakesMapping[_staker]);
  stakesMapping[_staker] = StakeInfo(stakesMapping[_staker].amount.sub(_stake), block
    .timestamp);
  updateTotalStakeInfoAndPastRedeemable(0, _stake, 0, 0);
}
```

Snippet 4.1: The definition of `removeStake()`

As another example, the `addStake` will add an account as a new stakeholder even if an amount of zero is staked:

```
function addStake(address _staker, uint256 _stake) public onlyAdmin {
  if (stakesMapping[_staker].amount == 0) addStakeholder(_staker);
  addPastRedeemableReward(_staker, stakesMapping[_staker]);
  stakesMapping[_staker] = StakeInfo(stakesMapping[_staker].amount.add(_stake), block
    .timestamp);
  updateTotalStakeInfoAndPastRedeemable(_stake, 0, 0, 0);
}
```

Snippet 4.2: The definition of `addStake()`

Furthermore, as observed in both examples, the checks for `stakesMapping[_staker].amount == 0` seem to imply that a user is a stakeholder if and only if they have a non-zero amount entry in `stakesMapping` ; this seems to suggest that the `stakeholderMap` variable is redundant.

`removeStakeholder` itself as a function is also never used, nor are any access modifiers implemented in relation to `isStakeholder` .

Impact The stakeholder tracking logic seems to be incorrect, which can lead to problems when trying to use the `isStakeholder()` method. For example, governance is determined by the stakeholders. As another example, someone could become a stakeholder with zero amount staked.

Recommendation Clarify the notion of what a "stakeholder" is, and fix the business logic to be consistent with the notion of "stakeholder".

Add additional checks to validate the method parameters.

4.1.2 V-ALL-VUL-002: Treasury Fee Variables Can Only Increase

Severity	High	**Commit**	5f8c250
Type	Logic error	**Status**	Fixed
Files	AlloyxTreasury.sol		
Functions	See description		

Description The `AlloyxTreasury` provides four methods `addEarningGfiFee` , `addRepaymentFee` , `addRedemptionFee` , and `addDuraToFiduFee` to increase the four "fee" state variables. These fee variables are used in the `transferAllUsdcFees` and `transferAllGfiFee` methods to calculate the amount of ERC20 tokens that should be transferred by an admin to some beneficiary.

Because the fees are not "reset" in those fee transfer methods, this means that any subsequent transfers must transfer the cumulative fee amounts, which does not seem to be a desirable behavior. For example, if the current `earningGfiFee` is 100, and an admin invokes `transferAllGfiFees(...)` , then the current `earningGfiFee` will remain as 100. Even if 200 GFI tokens are added to the Alloy protocol, the `earningGfiFee` will be 300 (not 200), although there are only 200 actual tokens held by Alloy.

Impact Any subsequent calls to the fee transfer methods will either 1) require the admins to supply a large amount of ERC20 tokens before being able to make the fee transfer (due to the fees not resetting); or 2) prevent the admins from transferring the fees altogether.

Recommendation Modify the fee transfer methods so that the appropriate fee variables are reset.

4.1.3 V-ALL-VUL-003: Lack of Access Controls in SortedGoldfinchTranches.sol

Severity	High	**Commit**	5f8c250
Type	Access Control	**Status**	Fixed
Files	SortedGoldfinchTranches.sol		
Functions	All		

Description Even though `SortedGoldfinchTranches.sol` inherits from `Ownable`, it is not initialized with an owner, and none of the functions use access control modifiers such as `onlyOwner`. We used a unit test (available upon request) to show that these functions are callable by any arbitrary address, and that any arbitrary payable address can be added as a tranch.

Impact The attacker is able to add arbitrary addresses as tranches in the tranches mapping `_nextTranches` , and modify the score of any tranch via `updateScore` . This breaks the protocol by ruining the tranch rating system, and can potentially cause future issues with anything `SortedGoldFinchTranches`- related functionality that can be manipulated by an attacker.

Recommendation Implement access controls with `require` statements or by using the `onlyOwner` modifier.

4.1.4 V-ALL-VUL-004: claimRewards Can Revert If PercentageCRWNEarning Is Configured Incorrectly

Severity	Medium	**Commit**	5f8c250
Type	Data Validation	**Status**	Fixed
Files	StakeDesk.sol		
Functions	getRewardTokenCount		

Description The method `getRewardTokenCount` appears to return a pair of amount to reward and a fee such that the fee does not exceed the amount of reward. The fee is calculated by taking a linear multiple of the amount to reward:

```
uint256 fee = amountToReward.mul(config.getPercentageCRWNEarning()).div(100);
return (amountToReward, fee);
```

Snippet 4.3: Fee calculation

If the `PercentageCRWNEarning` configuration option is not in the bounds `[0, 100]`, then the `fee` will be greater than `amountToReward`, causing the `claimReward` function to always revert due to subtraction integer overflow:

```
function claimReward(uint256 _amount) external returns (bool) {
  (uint256 amountToReward, uint256 fee) = getRewardTokenCount(_amount);
  config.getTreasury().transferERC20(config.gfiAddress(), msg.sender, amountToReward.
    sub(fee));
  config.getTreasury().addEarningGfiFee(fee);
  config.getCRWN().burn(msg.sender, _amount);
  emit Reward(msg.sender, _amount);
  return true;
}
```

Snippet 4.4: Function `claimReward`

Because the value `PercentageCRWNEarning` is not checked, it is possible for an admin to mistakenly set an invalid value that is out of bounds. For example, consider a scenario where an admin mixes up the order of the configuration values, so that an out-of-bounds value is assigned to `PercentageCRWNEarning`.

Impact If `claimRewards` always reverts, then users will be unable to claim their rewards, leading to financial damage to the users, bad publicity, and a loss of user trust.

Recommendation Validate parameters such as `PercentageCRWNEarning` in `AlloyxConfig.setNumber` or any other appropriate location where parameters are being set.

Update The developers added validation when this config value is used in the `StakeDesk.getRewardTokenCount()` method in commit `f349bf6ec4ed6a6fb78430d5947f0e2178f2e7f6`; however, they did not fix the underlying problem (validation needed when setting the config value). Because the developer's change mitigates this specific problem sufficiently, we have marked this issue as fixed. In the future, the developers should be aware that the underlying problem is still an issue.

4.1.5 V-ALL-VUL-005: Potential Reentrancy Vulnerability In depositDuraForPoolToken

Severity	Medium	Commit	5f8c250
Type	Reentrancy	Status	Fixed
Files	GoldfinchDesk.sol, AlloyxTreasury.sol		
Functions	depositDuraForPoolToken(), transferTokenToDepositor()		

Description The `depositDuraForPoolToken` method invokes the `transferTokenToDepositor` method (of the `AlloyxTreasury` contract), which transfers pool tokens (ERC721) to the sender.

```
function depositDuraForPoolToken(uint256 _tokenId) external isWhitelisted(msg.sender)
    {
  uint256 purchaseAmount = getJuniorTokenValue(_tokenId);
  uint256 withdrawalFee = purchaseAmount.mul(config.getPercentageJuniorRedemption()).
    div(100);
  uint256 duraAmount = config.getExchange().usdcToAlloyxDura(purchaseAmount.add(
    withdrawalFee));
  config.getTreasury().addRedemptionFee(withdrawalFee);
  transferTokenToDepositor(msg.sender, _tokenId);
  config.getDURA().burn(msg.sender, duraAmount);
  emit Burn(msg.sender, duraAmount);
  emit DepositDURA(msg.sender, duraAmount);
  emit WithdrawPoolTokens(msg.sender, _tokenId);
}

```

Snippet 4.5: Definition of `depositDuraForPoolToken` in `GoldfinchDesk.sol`

The `transferTokenToDepositor` method checks whether the `_depositor` (in this case, the sender of `depositDuraForPoolToken`) owns the token. The problem is that `AlloyxTreasury.transferERC721` will invoke `safeTransferFrom` method on the Goldfinch PoolTokens contract. Because PoolTokens is derived from OpenZeppelin's ERC721 implementation, `safeTransferFrom` will call an `onERC721Received` callback on the receiver of the token (the `_depositor`).

```
function transferTokenToDepositor(address _depositor, uint256 _tokenId) internal {
    require(tokenDepositorMap[_tokenId] == _depositor, "The token is not deposited by
     this user");
    config.getTreasury().transferERC721(config.poolTokensAddress(), _depositor,
    _tokenId);
    delete tokenDepositorMap[_tokenId];
}

```

Snippet 4.6: Definition of `transferTokenToDepositor` in `AlloyxTreasury.sol`

Because `tokenDepositorMap` is not updated until after the `transferERC721` call, it is possible for a malicious sender to use `onERC721Received` to perform a reentrant call into `depositDuraForPoolToken` .

Impact Since the sender's DURA tokens must be burned by the `depositDuraForPoolToken` method, and the sender is also required to be whitelisted, it does not seem like a reentrancy-based exploit would be very likely or useful. Nevertheless, there is a possibility that a determined attacker could combine this reentrancy flaw with another exploit to create a damaging attack.

Recommendation

- In `AlloyxTreasury`, move the `delete tokenDepositorMap[_tokenId]` line before the call to `transferERC721`.
- In `GoldfinchDesk`, move the `transferTokenToDepositor(...)` method call to the end of the `depositDuraForPoolToken` method.
- Document the fact that it is possible for the beneficiary in `AlloyxTreasury.transferERC721` to make further calls.

4.1.6 V-ALL-VUL-006: Potential Stake Reward Inconsistency Caused By Config Update

Severity	Medium	Commit	73cbee8
Type	Logic Error	Status	Acknowledged
Files	StakeDesk.sol, AlloyxStakeInfo.sol		
Functions	StakeDesk.claimAllAlloyxCRWN(), StakeDesk.claimAlloyxCRWN()		

Description In `AlloyxStakeInfo` , the total amount of potentially redeemable CRWN token amounts is calculated by summing

1. the function `calculateRewardFromStake(totalStakeInfo)` , which computes the amount of reward that can be claimed since the last update to `totalStakeInfo`;
2. the `totalPastRedeemableReward` state variable, which tracks the cumulative past redeemable reward before the last update to `totalStakeInfo`.

The `updateTotalStakeInfoAndPastRedeemable()` method is used to update the amount and timestamp recorded in `totalActiveStake` :

```
function updateTotalStakeInfoAndPastRedeemable(
  uint256 increaseInStake,
  uint256 decreaseInStake,
  uint256 increaseInPastRedeemable,
  uint256 decreaseInPastRedeemable
) internal {
  uint256 additionalPastRedeemableReward = calculateRewardFromStake(totalActiveStake)
    ;
  totalPastRedeemableReward = totalPastRedeemableReward.add(
    additionalPastRedeemableReward);
  totalPastRedeemableReward = totalPastRedeemableReward.add(increaseInPastRedeemable)
    .sub(
    decreaseInPastRedeemable
  );
  totalActiveStake = StakeInfo(
    totalActiveStake.amount.add(increaseInStake).sub(decreaseInStake),
    block.timestamp
  );
}
```

Snippet 4.7: Definition of `updateTotalStakeInfoAndPastRedeemable()`. Note that the `totalActiveStake` amount is increased by the amount of unclaimed rewards since last update.

When this method is used alongside an update to a given staker's `pastRedeemableReward`, the `additionalPastRedeemableReward` may potentially be inconsistent with the change to the staker's `pastRedeemableReward`.

One specific way to cause such inconsistency is to change the configuration value `PercentageRewardPerYear` . As a concrete example, consider the following scenario:

- At some time $t = 0$ years, user A has 200 DURA staked. Assume the percentage reward per year is 1% at this point.
- At time $t = 5$ years, user B stakes 100 DURA via `StakeDesk.stake()`. This updates the total past redeemable reward (TPRR) to `200*0.01*5` .
- Suppose the admin changes the percentage reward per year to 2%.
- At time $t = 10$ years, the `totalClaimableCRWNToken()` is inconsistent with the sum of the `claimableCRWNToken()` over all users. The former is calculated as `300*0.02*5 + 200*0.01*5 = 40`, while the latter is now calculated as `200*0.02*10 + 100*0.02*5 = 50`.
- This inconsistency is due to the fact that the `pastRedeemableReward` for user A was not updated after the reward rate was changed. In this scenario, the correct calculation for user A should be `200*0.01*5 + 200*0.02*5`.

Impact This may mean that the `totalPastRedeemableReward` is no longer consistent with the values contained in `pastRedeemableReward` (i.e., the former may be greater than the sum of the latter). Consequently, `AlloyxStakeInfo.totalClaimableCRWNToken()` may be higher or lower than expected, causing `StakeDesk.getRewardTokenCount()` (and therefore claimed GFI reward amounts, as calculated in `StakeDesk.claimReward()`) to be higher or lower than expected.

Recommendation Ensure that the total additional past redeemable amount is updated in the appropriate place(s), such as in `AlloyxStakeInfo.resetStakeTimestampWithRewardLeft()`.

The developer should be careful to look for similar problems such as:

- Other scenarios in which `totalPastRedeemableReward` can become inconsistent with the sum of the `pastRedeemableReward` values.
- Whether the way the `pastRedeemableReward` is calculated is correct, esp. with respect to methods or variables that may affect it.

Developer Response The developers observed that changing the `PercentageRewardPerYear` will impact all stakers in the same way regardless if they gain proportionally more or less CRWN, so the developers believe that they will be unlikely to change the `PercentageRewardPerYear`. However, this issue is still unfixed, so the developers should continue to remain aware of this issue.

4.1.7 V-ALL-VUL-007: Bugs in newly added pausing mechanism

Severity	Medium	**Commit**	3a52664
Type	Logic Error	**Status**	Fixed
Files	AlloyxTreasury.sol, AlloyxStakeInfo.sol, StakeDesk.sol, etc.		
Functions	functions in contracts AlloyxTreasury, AlloyxStakeInfo, StakeDesk, etc.		

Description To address the config race condition issue that we reported, the developers added a mechanism that forces the protocol to be "paused" before the config contract can be updated. However, we observed several issues with the newly added mechanism:

- In `StakeDesk`, `updateConfig` has modifier `notPaused` instead of `isPaused`.
- Multiple contracts define a `notPaused` modifier as `require(!config.isPaused(), "the user operation should be paused first");` The error thrown here seems incorrect: the condition `!config.isPaused()` is violated when `isPaused` returns true (i.e. the user operations are paused), and to proceed the contract must be unpaused. The message should be changed to something like `user operations should be unpaused first`.
- The `isPaused` modifier is missing in `updateConfig()` of `AlloyxStakeInfo` and `AlloyxTreasury`.

Impact If the affected interfaces are modified to have more methods, or if the concrete implementations of those methods have their type signatures changed, then calls to any such methods may revert.

Recommendation Fix the issues described, and check that there are no additional missing cases.

4.1.8 V-ALL-VUL-008: Missing Interface Inheritance

Severity	Low	**Commit**	5f8c250
Type	Maintainability	**Status**	Fixed
Files	AlloyxConfig.sol, SortedGoldfinchTranches.sol		
Functions	N/A		

Although `AlloyxConfig` implements methods from `IAlloyxConfig`, it does not actually inherit `IAlloyxConfig`. Given the following comment on lines 54-57, it seems like this missing inheritance is unintended:

```
1 /*
2   Using custom getters in case we want to change underlying implementation later,
3   or add checks or validations later on.
4 */
```

Snippet 4.8: Comment on lines 54-57 of `IAlloyxConfig.sol`

A similar issue affects `SortedGoldfinchTranches`.

Impact

If the affected interfaces are modified to have more methods, or if the concrete implementations of those methods have their type signatures changed, then calls to any such methods may revert.

Recommendation

- Inherit `IAlloyxConfig` in `AlloyxConfig`.
- Inherit `ISortedGoldfinchTranches` in `SortedGoldfinchTranches`

4.1.9 V-ALL-VUL-009: Potential Upgrade Race Condition

Severity	Low	**Commit**	5f8c250
Type	Transaction Ordering	**Status**	Fixed
Files	AlloyxStakeInfo.sol, AlloyxExchange.sol and four others		
Functions	updateConfig()		

Description If the address of the configuration of all of the DAO contracts is not updated within the same transaction (i.e., with `updateConfig()`), then it is possible for some intermediate transaction to be performed (e.g., as a result of a malicious miner or transaction reordering attack) when some of the contracts point to the new config and the rest of the contracts point to the old config.

The four files also affected are:`GoldfinchDesk.sol`, `StakeDesk.sol`,`AlloyxTreasury.sol` and `SortedGoldfinchTranches.sol`.

Impact We believe such a scenario is unlikely, but it could be possible for an attacker to take advantage of this scenario. For example, if two contracts X and Y use a rate parameter, but X is updated to use a new rate while Y still uses the old rate, an attacker could take out a flash loan to exploit differences in these rates.

Recommendation Provide a smart contract, or some other functionality, that will update all of the configurations within the same transaction.

Developer Response The developers acknowledged that they would be careful during config upgrade deployments, and they mitigated the issue by adding a mechanism that forces all contracts to be “paused” during config upgrades.

4.1.10 V-ALL-VUL-010: Missing Interface Inheritance II

Severity	Low	**Commit**	5f8c250
Type	Maintainabilty	**Status**	Fixed
Files	AlloyxStakeInfo.sol		
Functions	N/A		

Description `AlloyxStakeInfo` implements all the methods of the interface `IAlloyxStakeInfo` but does not explicitly inherit it.

Impact If the affected interfaces are modified to have more methods, or if the concrete implementations of those methods have their type signatures changed, then calls to any such methods may revert.

Recommendation Inherit `IAlloyxStakeInfo` in `AlloyxStakeInfo` .

4.1.11 V-ALL-VUL-011: removeWhitelistedUser Cannot Remove Address whitelisted by Goldfinch KYC

Severity	Low	Commit	7606eac
Type	Logic error	Status	Fixed
Files	AlloyxWhitelist.sol		
Functions	removeWhitelistedUser()		

Description Various contracts used in the Alloy protocol use the `isUserWhitelisted` method to check authorization.

```
function isUserWhitelisted(address _whitelistedAddress) public view override returns
    (bool) {
  return whitelistedAddresses[_whitelistedAddress] || hasWhitelistedUID(
    _whitelistedAddress);
}
```

Snippet 4.9: Function `isUserWhitelisted`

Based on the above code, a user is considered to be whitelisted if they satisfy any of the following conditions:

- The value of the given address in `whitelistedAddresses` is `true` (i.e., they are whitelisted internally by the Alloy protocol).
- They are whitelisted by Goldfinch (e.g., by owning at least one NFT issued by Goldfinch's `UniqueIdentity` token contract).

However, the `removeWhitelistedUser` method in `AlloyxWhitelist` only sets the `whitelistedAddress` entry of the given address to `false` , which can be confusing or misleading. This leads to a confusing behavior where calling `removeWhitelistedUser` on an address whitelisted by Goldfinch has no effect.

```
function removeWhitelistedUser(address _addressToDeWhitelist)
  external
  onlyOwner
  isWhitelisted(_addressToDeWhitelist)
{
  whitelistedAddresses[_addressToDeWhitelist] = false;
}
```

Snippet 4.10: Function `removeWhitelistedUser`

Impact In a situation where a user's whitelist status needs to be revoked, this behavior can cause confusion if the user is actually whitelisted by Goldfinch. For example, admins might believe that a user is no longer whitelisted after calling `removeWhitelistedUser` (which executes successfully), but the user might still be whitelisted by Goldfinch and will still have authorization to access Alloy protocol functions.

Because the whitelist is used for checking authorization, a flaw in the whitelist has negative implications for the security of the protocol.

Recommendation

- Add a `require` statement in `removeWhitelistedUser` to explicitly check that the user is contained in `whitelistedAddresses`.
- Consider renaming `removeWhitelistedUser` to clearly indicate that this only affects the Alloywhitelist.

Developer Response

The developers renamed `removeWhitelistedUser` to `removeAlloyxWhitelistedUser`.

4.1.12 V-ALL-VUL-012: copyFromOtherConfig Is Error-Prone

Severity	Warning	**Commit**	5f8c250
Type	Data Validation	**Status**	Fixed
Files	AlloyxConfig.sol		
Functions	copyFromOtherConfig		

Description The `copyFromOtherConfig` requires an admin to specify the `numbersLength` and the `addressesLength` variables, corresponding to the number of entries in `ConfigOptions.sol` . However, this since it is possible to add entries to the `ConfigOptions`.Numbers and `ConfigOptions` .Addresses enums, an admin may be able to enter incorrect values of `numbersLength` and `addressesLength` during an upgrade or deployment.

```
function copyFromOtherConfig(
address _initialConfig,
uint256 numbersLength,
uint256 addressesLength
) public onlyAdmin
```

Snippet 4.11: Function copyFromOtherConfig

Recommendation

- The developers should carefully think about the scenarios in which they will be using this function and document them in a comment. Furthermore, they should ensure that this method is tested for each of those scenarios.
- The range of values of the enums in `ConfigOptions` can stored by adding a dummy `Last` entry to each enum. The `AlloyxConfig` contract can then provide methods to query the `Last` entries (returning it as the length), so that a newer config is able to automatically determine the correct number of entries to copy from an older config.

4.1.13 V-ALL-VUL-013: High Gas Cost When Computing Goldfinch Token Counts

Severity	Warning	**Commit**	5f8c250
Type	Denial of Service	**Status**	Acknowledged
Files	GoldfinchDesk.sol		
Functions	getGoldFinchPoolTokenBalanceInUsdc() and two other functions		

Description The `GoldfinchDesk` contract provides various methods to compute token balances based on the number of ERC721 tokens that the treasury contract has in the Goldfinch `PoolTokens` contract. However, such total balances are computed by looping over the total number of ERC721 tokens in the following methods:

- getFoldFinchPoolTokenBalanceInUsdc()
- getTokensAvailableForWithdrawal()
- getTokensAvailableCountForWithdrawal()

```
function getGoldFinchPoolTokenBalanceInUsdc() public view override returns (uint256)
    {
    uint256 total = 0;
    uint256 balance = config.getPoolTokens().balanceOf(config.treasuryAddress());
    for (uint256 i = 0; i < balance; i++) {
        total = total.add(
            getJuniorTokenValue(config.getPoolTokens().tokenOfOwnerByIndex(config.
    treasuryAddress(), i))
        );
    }
    return total;
}
```

Snippet 4.12: Example of a loop that may consume a large amount of gas

Since contract calls are being made within the loop body, this is liable to cost large amounts of gas, especially as the treasury acquires more ERC721 tokens.

Additionally, since there are no limitations on which tokens are whitelisted for deposit, a malicious user could gift a large number of pool tokens to the treasury contract, causing the array to grow ever larger.

Impact The gas costs may grow to the point where every transaction starts to revert, resulting in a denial-of-service issue.

4.1.14 V-ALL-VUL-014: Lack of zero address checks

Severity	Warning	Commit	5f8c250
Type	Data Validation	Status	Fixed
Files	AlloyxConfig.sol, AlloyxWhitelist.sol, GoldfinchDesk.sol		
Functions	See description		

Some functions (listed below) that change `address` type state variables do not validate their parameters with statements like `require(myAddressParameter != address(0))`.

```
function purchasePoolToken(
  uint256 _amount,
  address _poolAddress,
  uint256 _tranche
) public onlyAdmin {
  ITranchedPool juniorPool = ITranchedPool(_poolAddress);
  // ...
}

function withdrawFromJuniorToken(
  uint256 _tokenID,
  uint256 _amount,
  address _poolAddress
) external onlyAdmin {
  ITranchedPool juniorPool = ITranchedPool(_poolAddress);
  // ...
}

function getTokensAvailableForWithdrawal(address _depositor)
  external
  view
  returns (uint256[] memory)
{
  // ...
}

function getTokensAvailableCountForWithdrawal(address _depositor) public view returns
    (uint256) { //@audit-issue is each user supposed to only have 1 NFT per pool?
  // ...
}
```

Snippet 4.13: Affected methods in `GoldfinchDesk.sol`

```
setAddress(uint256 addressIndex, address newAddress) public override onlyAdmin {
  emit AddressUpdated(msg.sender, addressIndex, addresses[addressIndex], newAddress);
  addresses[addressIndex] = newAddress;
}
```

Snippet 4.14: Affected methods in `AlloyxConfig.sol`

```
function changeUIDAddress(address _uidAddress) external onlyOwner {
  uidToken = IERC1155(_uidAddress);
  emit ChangeAddress("uidToken", _uidAddress);
}

function addWhitelistedUser(address _addressToWhitelist)
  external
  onlyOwner
  notWhitelisted(_addressToWhitelist)
{
  whitelistedAddresses[_addressToWhitelist] = true;
}
```

Snippet 4.15: Affected methods in `AlloyxWhitelist.sol`

Impact Adding zero address checks can help minimize mistakes/errors made during contract deployment.

Recommendation Add `require` statements in the places noted above.

4.1.15 V-ALL-VUL-015: Expensive Gas On Config Copying

Severity	Warning	Commit	5f8c250
Type	Gas Optimization	Status	Acknowledged
Files	AlloyxConfig.sol		
Functions	copyFromOtherConfig()		

Description The function `copyFromOtherConfig` uses `for` loops bounded by `numbersLength` and `addressesLength` that each corresponds to lengths of the enums `Numbers` and `Addresses` given in the contract library `ConfigOptions`.

```
function copyFromOtherConfig(
address _initialConfig,
uint256 numbersLength,
uint256 addressesLength
) public onlyAdmin {
    IAlloyxConfig initialConfig = IAlloyxConfig(_initialConfig);
    for (uint256 i = 0; i < numbersLength; i++) {
        setNumber(i, initialConfig.getNumber(i));
    }
    for (uint256 i = 0; i < addressesLength; i++) {
        if (getAddress(i) == address(0)) {
        setAddress(i, initialConfig.getAddress(i));
        }
    }
}
```

Snippet 4.16: Function `copyFromOtherConfig`

Due to the fact that these enums may grow significantly in the future, the gas costs may grow significantly.

```
enum Addresses {
    Treasury, Exchange, Config, GoldfinchDesk, StableCoinDesk,
    StakeDesk, Whitelist, AlloyxStakeInfo, PoolTokens, SeniorPool,
    SortedGoldfinchTranches, FIDU, GFI, USDC, DURA, CRWN, BackerRewards
}
```

Snippet 4.17: Addresses enum from `ConfigOptions`

Impact Increased gas cost when migrating configuration.

4.1.16 V-ALL-VUL-016: Invariant Involving stake.since Violated in resetStakeTimestamp

Severity	Warning	**Commit**	6356a9b
Type	Maintainability	**Status**	Acknowledged
Files	AlloyxStakeInfo.sol		
Functions	resetStakeTimestamp()		

Description In the `addStake` and `removeStake` functions, the timestamp of `totalActiveStake` is updated by a call to `updateTotalStakeInfoAndPastRedeemable`, maintaining a property that the maximum value of all of the `stakesMapping[*].since` times should all be no later than the `totalActiveStake.since`. Specifically:

$$\max(\text{stakesMapping}[*].\text{since}) \leq \text{totalActiveStake.since}$$

However, the function `resetStakeTimestamp` updates the time stamp of stake holder's `StakeInfo` , but does not do so for `totalActiveStake` , meaning that the above property may no longer hold after a call to `resetStakeTimestamp` .

```
function resetStakeTimestamp(address _stakeholder) internal {
    addPastRedeemableReward(_stakeholder, stakesMapping[_stakeholder]);
    stakesMapping[_stakeholder] = StakeInfo(stakesMapping[_stakeholder].amount, block
    .timestamp);
}
```

Snippet 4.18: Implementation of `resetStakeTimestamp`

Impact It may be possible for the developers to forget to update the `totalActiveStake.since` after calling `resetStakeTimestamp` , especially if they use `resetStakeTimestamp` in more public methods.

Currently, this is not a problem; `resetStakeTimestamp()` is an internal method that is only used in by the external method `resetStakeTimestampWithRewardLeft()`, and the latter will update the `totalActiveStake.since` with a call to `adjustTotalStakeWithRewardLeft()`.

```
function resetStakeTimestampWithRewardLeft(address _staker, uint256 _reward)
external
override
onlyAdmin
{
    resetStakeTimestamp(_staker);
    adjustTotalStakeWithRewardLeft(_staker, _reward);
    pastRedeemableReward[_staker] = _reward;
}
```

Snippet 4.19: Definition of `resetStakeTimestampWithRewardLeft()`. According to our formal verification tool Eurus, the invariant is temporarily violated in between the calls to `resetStakeTimestamp` and `adjustTotalStakeWithRewardLeft`.

Recommendation Refactor the functions so that both `stakesMapping[*].since` and `totalActiveStake.since` are updated in the same function, making it more difficult to violate this invariant.

Developer Response While the developers acknowledged that this could be a possible maintainability issue, they also pointed out that this invariant is not that important to them as long as the contract still maintains the other invariant `sum(stakesMapping[*].amount) == totalActiveStake.amount`.

4.1.17 V-ALL-VUL-017: Access Control Pitfalls when Expanding Whitelist to Include More 3rd Party Protocols

Severity	Warning	**Commit**	73cbee8
Type	Access Control	**Status**	Acknowledged
Files	AlloyxWhitelist.sol, GoldfinchDesk.sol, etc.		
Functions	See description		

Description The whitelisting logic in `AlloyxWhitelist` considers a user to be "whitelisted" whenever one of the following is true:

- Goldfinch whitelists the user through their KYC mechanism (e.g., via ownership of a Goldfinch `UniqueIdentity` NFT).
- The user is explicitly whitelisted by Alloy in the `AlloyxWhitelist` contract.

The `AlloyxWhitelist` contract provides a `isWhitelisted()` method that can be used to check when a user is whitelisted according to the definition above. In the current code, this is sufficient because Goldfinch is the only third-party protocol supported by the Alloy contracts.

However, the developers indicated that they would also like to support other third-party protocols, in which case the above definition would be too coarse-grained.

For example, if the developers want to support a new third-party "Example Protocol" that also has some whitelist/KYC mechanism, then the developers may want to refine their access control model to deal with requirements like "only users whitelisted by Alloy or users whitelisted by Example Protocol may access Example Protocol functionalities" and "users whitelisted by Example Protocol but not by Goldfinch may not access Goldfinch functionalities".

4.1.18 V-ALL-VUL-018: Invariant involving totalPastRedeemableReward violated in addPastRedeemableReward and resetStakeTimestamp

Severity	Warning	**Commit**	6356a9b
Type	Maintainability	**Status**	Acknowledged
Files	AlloyxStakeInfo.sol		
Functions	addPastRedeemableReward(), resetStakeTimestamp()		

Description The invariant

$$\text{sum}(\text{pastRedeemableReward}[*]) \leq \text{totalPastRedeemableReward}$$

is temporarily violated in `addPastRedeemableReward()` and `resetStakeTimestamp()`. This is because in each of these two functions, an entry in `pastRedeemableReward` is updated without also updating `totalPastRedeemableReward`.

Impact Currently, this is not a problem; both methods are only used in external methods that also update `totalPastRedeemableRreward`.

However, it may be possible for the developers to forget to update the `totalPastRedeemableReward` if they use the methods mentioned in this issue in more external methods in the future.

Recommendation To avoid introducing bugs in the future, consider placing the updates to `pastRedeemableReward` and `totalPastRedeemableReward` in the same function so that the invariant is harder to violate.

4.1.19 V-ALL-VUL-019: Multiple Public Functions Can Be Declared As External

Severity	Informational	**Commit**	5f8c250
Type	Maintainability	**Status**	Fixed
Files	See description		
Functions	See description		

Description Some methods in various contract functions (which are currently public) can be listed as external. This can result in possible gas savings. Solidity immediately copies array arguments to memory, while external functions can read directly from calldata. Memory allocation is expensive, whereas reading from calldata is cheap.

Below are methods, listed by contract, that may be marked as external:

```
function addAdmin(address account) public virtual onlyAdmin { ... }
function renounceAdmin() public virtual { ... }
```

Snippet 4.20: Functions which may be listed as external in AdminUpgradable.sol

```
function initialize() public initializer { ... }
function copyFromOtherConfig(
address _initialConfig,
uint256 numbersLength,
uint256 addressesLength
) public onlyAdmin { ... }
function getNumber(uint256 index) public view returns (uint256) { ... }
```

Snippet 4.21: Functions which may be listed as external in AlloyxConfig.sol

```
function initialize(address _configAddress) public initializer { ... }
function alloyxDuraToUsdc(uint256 _amount) public view override returns (uint256) {
    ... }
function usdcToAlloyxDura(uint256 _amount) public view override returns (uint256) {
    ... }
```

Snippet 4.22: Functions which may be listed as external in AlloyxExchange.sol

```
function initialize(address _configAddress) public initializer { ... }
function isStakeholder(address _address) public view returns (bool) { ... }
function addStake(address _staker, uint256 _stake) public onlyAdmin { ... }
function removeStake(address _staker, uint256 _stake) public onlyAdmin { ... }
function resetStakeTimestampWithRewardLeft(address _staker, uint256 _reward) public
    onlyAdmin { ... }
function claimableCRWNToken(address _receiver) public view returns (uint256) { ... }
function totalClaimableCRWNToken() public view returns (uint256) { ... }
```

Snippet 4.23: Functions which may be listed as external in AlloyxStakeInfo.sol

```
function initialize() public initializer { ... }
```

Snippet 4.24: Function which may be listed as external in AlloyxTokenCRWN.sol

```
function initialize() public initializer { ... }
```

Snippet 4.25: Function which may be listed as external in AlloyxTokenDURA.sol

```
function initialize(address _configAddress) public initializer { ... }
function getAllGfiFees() public view override returns (uint256) { ... }
```

Snippet 4.26: Functions which may be listed as external in AlloyxTreasury.sol

```
function addWhitelistedUser(address _addressToWhitelist)
  public
  onlyOwner
  notWhitelisted(_addressToWhitelist)
  { ... }
function removeWhitelistedUser(address _addressToDeWhitelist)
  public
  onlyOwner
  isWhitelisted(_addressToDeWhitelist)
  { ... }
```

Snippet 4.27: Functions which may be listed as external in AlloyxWhitelist.sol

```
function initialize(address _configAddress) public initializer { ... }
function getGoldFinchPoolTokenBalanceInUsdc() public view override returns (uint256)
    { ... }
```

Snippet 4.28: Functions which may be listed as external in GoldfinchDesk.sol

```
function increaseScore(address tranch, uint256 score) public { ... }
function reduceScore(address tranch, uint256 score) public { ... }
function getTop(uint256 k) public view returns (address[] memory) { ... }
```

Snippet 4.29: Functions which may be listed as external in SortedGoldfinchTranches.sol

```
function initialize(address _configAddress) public initializer { ... }
```

Snippet 4.30: Function which may be listed as external in StableCoinDesk.sol

```
function initialize(address _configAddress) public initializer { ... }
```

Snippet 4.31: Function which may be listed as external in StakeDesk.sol

Recommendation To save gas, change the visibility of these functions from `public` to `external`
.

4.1.20 V-ALL-VUL-020: Explicitly Mark State Visibility With Some Variables

Severity	Informational	**Commit**	5f8c250
Type	Maintainability	**Status**	Fixed
Files	AlloyxStakeInfo.sol		
Functions	See description		

Description Some variables in various contracts do not have state mutability marked. for example:

```
address vaultAddress;
StakeInfo totalActiveStake;
```

Snippet 4.32: State variables in `AlloyxStakeInfo.sol` with unmarked visibility

Impact Marking state visibility can increase code clarity and potentially improve maintainability.

Recommendation Mark code visibility.

4.1.21 V-ALL-VUL-021: Events Should Be Emitted In AlloyxTreasury

Severity	Informational	**Commit**	5f8c250
Type	Missing/Incorrect Events	**Status**	Fixed
Files	AlloyxTreasury.sol		
Functions	In contract AlloyxTreasury		

Description To provide better monitoring of critical parameters, the following state-modifying functions should emit events.

```
function addEarningGfiFee(uint256 _amount) external override onlyAdmin {
  earningGfiFee += _amount;
}
function addRepaymentFee(uint256 _amount) external override onlyAdmin {
  repaymentFee += _amount;
}
function addRedemptionFee(uint256 _amount) external override onlyAdmin {
  redemptionFee += _amount;
}
function addDuraToFiduFee(uint256 _amount) external override onlyAdmin {
  duraToFiduFee += _amount;
}
```

Snippet 4.33: Functions that should emit events

Impact Events can be used to trace who is changing the parameters and by what. For example, this is important for security monitoring post-deployment.

Recommendation Emit events logging the changes to these parameters.

4.1.22 V-ALL-VUL-022: Public Variables In AlloyxConfig

Severity	Informational	**Commit**	5f8c250
Type	Maintainability	**Status**	Fixed
Files	AlloyxConfig.sol		
Functions			

Description In the contract `AlloyxConfig`, the variables `addresses` and `numbers` have been declared public while the getter functions `getAddress` and `getNumber` have also been implemented.

Recommendation Declare `addresses` and `numbers` as private.

4.1.23 V-ALL-VUL-023: More Missing Events

Severity	Informational	**Commit**	73cbee8
Type	Missing/Incorrect Events	**Status**	Open
Files	AlloyxStakeInfo.sol, AlloyxWhitelist.sol		
Functions	AlloyxStakeInfo.addStake, AlloyxStakeInfo.removeStake and three others		

Description Below is a list of functions that should emit events, but currently don't.

`AlloyxStakeInfo` has multiple staking-related functions that are external but do not have events, namely:

```
function addStake(address _staker, uint256 _stake) external override onlyAdmin {
  addPastRedeemableReward(_staker, stakesMapping[_staker]);
  stakesMapping[_staker] = StakeInfo(stakesMapping[_staker].amount.add(_stake), block
    .timestamp);
  updateTotalStakeInfoAndPastRedeemable(_stake, 0, 0, 0);
}

/**
 * @notice Remove stake for a staker
 * @param _staker The person intending to remove stake
 * @param _stake The size of the stake to be removed.
 */
function removeStake(address _staker, uint256 _stake) external override onlyAdmin {
  require(stakeOf(_staker).amount >= _stake, "User has insufficient dura coin staked"
    );
  addPastRedeemableReward(_staker, stakesMapping[_staker]);
  stakesMapping[_staker] = StakeInfo(stakesMapping[_staker].amount.sub(_stake), block
    .timestamp);
  updateTotalStakeInfoAndPastRedeemable(0, _stake, 0, 0);
}

function resetStakeTimestampWithRewardLeft(address _staker, uint256 _reward)
    external
    override
    onlyAdmin
{
  resetStakeTimestamp(_staker);
  adjustTotalStakeWithRewardLeft(_staker, _reward);
  pastRedeemableReward[_staker] = _reward;
}
```

Snippet 4.34: Methods in `AlloyxStakeInfo` that should emit events

`AlloyxWhitelist` should have events emitted for the following:


```
function addWhitelistedUser(address _addressToWhitelist)
    external
    onlyOwner
    notWhitelisted(_addressToWhitelist)
{
  whitelistedAddresses[_addressToWhitelist] = true;
}

function removeWhitelistedUser(address _addressToDeWhitelist)
    external
    onlyOwner
    isWhitelisted(_addressToDeWhitelist)
{
  whitelistedAddresses[_addressToDeWhitelist] = false;
}
```

Snippet 4.35: Methods in `AlloyxWhitelist` that should emit events.

Impact This would allow for more code clarity, and prepare contracts for communicating with a frontend.

Recommendation Add `emit(...)` and `event(...)` as needed for the listed functions.

5 Formal Verification Results

5.1 Description

In this section, we detail all of the specifications we formally checked with our formal verification tool Eurus. For each specification, we give its formal representation as well as a textual description of the behavior it is intended to capture. A $\surd$ indicates the specification was verified, while a $\times$ indicates the specification was falsified. We have considered two kinds of formal specifications,

1. Contract Invariants
2. Function Preconditions and Postconditions

Each of these types of formal specifications are described in the next section. As described in Section 3.2, the scope of the formal verification is limited to the `AlloyxStakeInfo` contract, as we found it to be the contract where formal verification has been most cost effective.

5.2 Specification Types

5.2.1 Contract Invariants

A contract invariant is a property of a smart contract's state variables which holds between all invocations of the public methods of the smart contract. This means such a property must hold at program locations that include (but are not limited to): exit from the constructor, entry to a public function of the contract, exit from a public function of the contract, method invocations of other smart contracts that could make a reentrant call back to the original contract, etc.

5.2.2 Function Preconditions and Postconditions

We express a function precondition and postcondition using the following syntax.

$$\{P\}\ f(\bar{A})\ \{Q\}$$

where P is a logical formula representing the precondition, $f(\bar{A})$ is the function being verified (where $\bar{A}$ are the names of the arguments of f), and Q is the postcondition. Informally, one can think of this spec as saying "if P holds before f executes, then after f executes, Q must hold".

In our context, $f(\bar{A})$ is a function in a smart contract (let's call the contract C) and P and Q are first-order logical formulas over the state variables of C, arguments $\bar{A}$ of f, and pure function calls from C. As an example, consider the following specification for a mint function in a simple token:

$$\{msg.sender = owner\}$$

$$\text{mint(address } a\text{, uint256 } amt)$$

$$\{\text{balanceOf}(a) = \text{old}(\text{balanceOf}(a)) + amt\}$$

This specification states that a call to mint with target address a and amount amt will add the desired amount amt to the balance of address a, presuming the function is called by the owner of the token, denoted $owner$. Note that the expression old(balanceOf(a)) uses the $old(e)$ expression, which indicates the value of e before the execution of the transaction.

5.3 Results

In what follows, we give formal specifications related to the smart contract `AlloyxStakeInfo`. For each specification, we provide a textual description of the specification and the result of that (√ if it is verified and × if it is falsified) checking the specification with Eurus.

5.3.1 Contract Invariants

√ **Specification 1** The total active stake is consistent:

{ sum(stakesMapping[⋆].amount) == totalActiveStake.amount }

× **Specification 2** The total past redeemable reward bounds the individual amounts.

{ sum(pastRedeemableReward[⋆]) ≤ totalPastRedeemableReward }

As demonstrated by the counterexample described in Subsection 4.1.6, it is possible for a change to the reward rate to cause the left-hand side of this inequality to be greater than the right-hand side. The invariant otherwise holds in the case of a constant reward rate.

√ **Specification 3** The total active stake timestamp is always up-to-date.

{ max(stakesMapping[⋆].since) ≤ totalActiveStake.since }

While this invariant holds for public methods, we have observed that the invariant may not hold if internal methods are considered (see Subsection 4.1.16).

5.3.2 Function Preconditions and Postconditions

√ **Specification 4** Three properties for `resetStakeTimestamp`: (1) the staked amount remains unchanged; (2) the last reward update time of the stake is updated; (3) the reward may be distributed to the user. The property can be expressed formally as the following:

{ }

`resetStakeTimestamp(_stakeholder)`

{ stakesMapping[_stakeholder].amount == old(stakesMapping[_stakeholder].amount) ∧ stakesMapping[_stakeholder].since ≥ old(stakesMapping[_stakeholder].since ∧ pastRedeemableReward[_stakeholder] ≥ old(pastRedeemableReward[_stakeholder]) }

√ **Specification 5** `addStake` correctly increases the stake amount.

{ isAdmin(`msg.sender`) }

`addStake( _staker, _stake )`

{ stakesMapping[_stakeholder].amount == old(stakesMapping[_stakeholder].amount) + _stake }

√ **Specification 6** If a user has not staked before, and then they stake, then they should have no rewards.

{ isAdmin(`msg.sender`) ∧ stakesMapping[_staker].since == 0 }

`addStake( _staker, _stake )`

{ pastRedeemableReward[_staker] == 0 }

√ **Specification 7** `removeStake` correctly decreases the stake amount, assuming the user has enough staked.

{ isAdmin(`msg.sender`) ∧ stakeOf(_staker).amount ≥ _stake }

`removeStake( _staker, _stake )`

{ stakesMapping[_stakeholder].amount == old(stakesMapping[_stakeholder].amount) - _stake }

√ **Specification 8** Reward of a staker does not decrease when they call `addPastRedeemableReward`.

{}

`addPastRedeemableReward( _staker, _stake )`

{ pastRedeemableReward[_staker] ≥ old(pastRedeemableReward[_staker]) }

√ **Specification 9** Resetting a user's reward update timestamp should never revert if the caller is an admin.

{ isAdmin(`msg.sender`) }

`resetStakeTimestampWithRewardLeft(_staker, _reward)`

{ }