# Smart Contract Audit Report for Dimo

Testers

1. Or Duan
2. Avigdor Sason Cohen

# Table of Contents

# Management Summary

Dimo contacted Sayfer to perform a security audit on their smart contract.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for Dimo’s smart contract.

Over the research period of 40 research hours, we discovered 14 vulnerabilities in the contract. None of them are critical.

In conclusion, several fixes should be implemented following the report, but the system's security posture is competent.

**After a review by the Sayfer team, we certify that all the security issues mentioned in this report have been addressed by the Dimo team.**

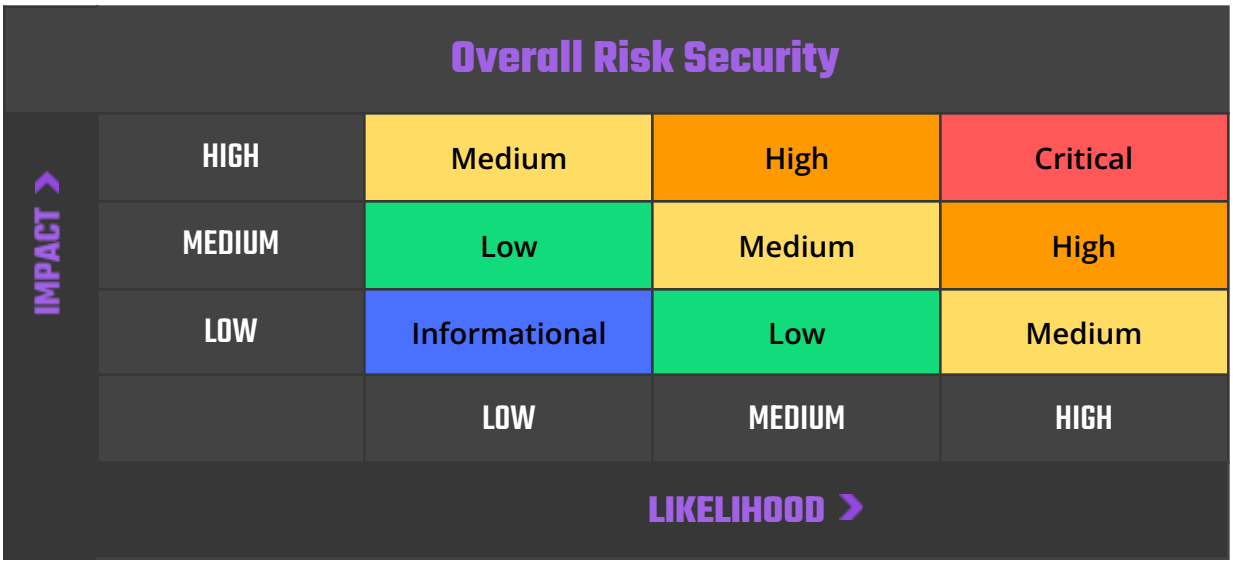
# Risk Methodology

At Sayfer, we are committed to delivering the highest quality smart contract audits to our clients. That's why we have implemented a comprehensive risk assessment model to evaluate the severity of our findings and provide our clients with the best possible recommendations for mitigation.

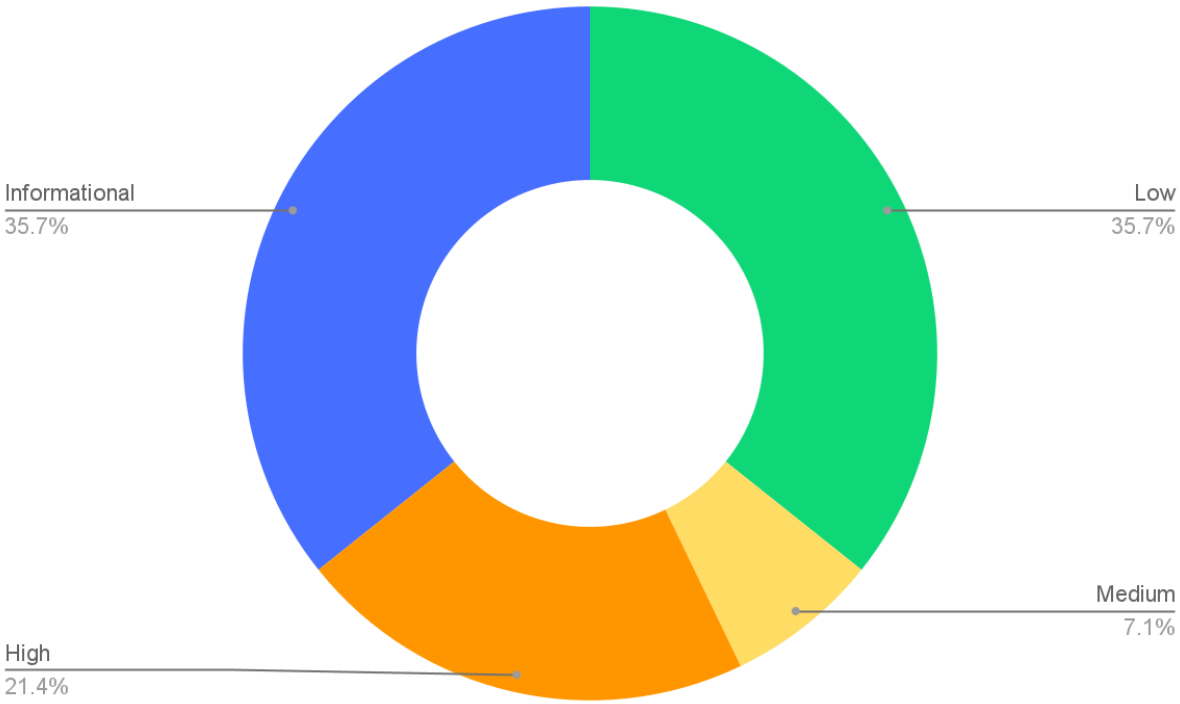
Our risk assessment model is based on two key factors: **IMPACT** and **LIKELIHOOD**. Impact refers to the potential harm that could result from an issue, such as financial loss, reputational damage, or a non-operational system. Likelihood refers to the probability that an issue will occur, taking into account factors such as the complexity of the contract and the number of potential attackers.

By combining these two factors, we can create a comprehensive understanding of the risk posed by a particular issue and provide our clients with a clear and actionable assessment of the severity of the issue. This approach allows us to prioritize our recommendations and ensure that our clients receive the best possible advice on how to protect their smart contracts.

**Risk is defined as follows:**



# Vulnerabilities by Risk




| Risk | Low | Medium | High | Critical | Informational |
|---|---|---|---|---|---|
| # of issues | 5 | 1 | 3 | 0 | 5 |

# Approach

## Introduction

Dimo contacted Sayfer to perform a security audit on their smart contract.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for the aforementioned contracts.

## Scope Overview

Together with the client team we defined the following contract as the scope of the project.
Commit hash: `14ce58aa499e3672fb0b61da13948a6ea51fb879`

| Contract | Sha-256 |
| --- | --- |
| `Reward.sol` | `86cffc0108ebe977ac55650da60cccc5f790013332186a2cb4dab47d7d38ac87` |

Our tests were performed in January 2024.

## Scope Validation

We began by ensuring that the scope defined to us by the client was technically logical. Deciding what scope is right for a given system is part of the initial discussion.

## Threat Model

We defined that the largest current threat to the system is the ability of malicious users to steal funds from the contract.

# Protocol Overview

## Protocol Introduction

DIMO is a web3 IoT company that allows users and developers to tap into the rich stream of data generated by modern vehicles. Its solution is a user-owned ecosystem that allows drivers to reap economic benefits from their data and make possible applications like parametric insurance, peer-to-peer car sharing, and vehicle marketplaces. The decentralized platform also gives developers peace of mind, knowing their access to the data is not subject to the whims of a centralized gatekeeper. This solution is built on Polygon.

# Security Evaluation

The following test cases were the guideline while auditing the system. This checklist is a modified version of the SCSVS v1.2, with improved grammar, clarity, conciseness, and additional criteria. Where there is a gap in the numbering, an original criterion was removed. Criteria that are marked with an asterisk were added by us.

| Architecture, Design and Threat Modeling | Test Name |
|---|---|
| G1.2 | Every introduced design change is preceded by threat modeling. |
| G1.3 | The documentation clearly and precisely defines all trust boundaries in the contract (trusted relations with other contracts and significant data flows). |
| G1.4 | The SCSVS, security requirements or policy is available to all developers and testers. |
| G1.5 | The events for the (state changing/crucial for business) operations are defined. |
| G1.6 | The project includes a mechanism that can temporarily stop sensitive functionalities in case of an attack. This mechanism should not block users' access to their assets (e.g. tokens). |
| G1.7 | The amount of unused cryptocurrencies kept on the contract is controlled and at the minimum acceptable level so as not to become a potential target of an attack. |
| G1.8 | If the fallback function can be called by anyone, it is included in the threat model. |
| G1.9 | Business logic is consistent. Important changes in the logic should be applied in all contracts. |
| G1.10 | Automatic code analysis tools are employed to detect vulnerabilities. |
| G1.11 | The latest major release of Solidity is used. |
| G1.12 | When using an external implementation of a contract, the most recent version is used. |
| G1.13 | When functions are overridden to extend functionality, the super keyword is used to maintain previous functionality. |
| G1.14 | The order of inheritance is carefully specified. |
| G1.15 | There is a component that monitors contract activity using events. |
| G1.16 | The threat model includes whale transactions. |
| G1.17 | The leakage of one private key does not compromise the security of the entire project. |

| Policies and Procedures | Test Name |
|---|---|

| | |
|---|---|
| G2.2 | The system's security is under constant monitoring (e.g. the expected level of funds). |
| G2.3 | There is a policy to track new security vulnerabilities and to update libraries to the latest secure version. |
| G2.4 | The security department can be publicly contacted and that the procedure for handling reported bugs (e.g., thorough bug bounty) is well-defined. |
| G2.5 | The process of adding new components to the system is well defined. |
| G2.6 | The process of major system changes involves threat modeling by an external company. |
| G2.7 | The process of adding and updating components to the system includes a security audit by an external company. |
| G2.8 | In the event of a hack, there's a clear and well known mitigation procedure in place. |
| G2.9 | The procedure in the event of a hack clearly defines which persons are to execute the required actions. |
| G2.10 | The procedure includes alarming other projects about the hack through trusted channels. |
| G2.11 | A private key leak mitigation procedure is defined. |

| **Upgradability** | **Test Name** |
|---|---|
| G2.2 | Before upgrading, an emulation is made in a fork of the main network and everything works as expected on the local copy. |
| G2.3 | The upgrade process is executed by a multisig contract where more than one person must approve the operation. |
| G2.4 | Timelocks are used for important operations so that the users have time to observe upcoming changes (please note that removing potential vulnerabilities in this case may be more difficult). |
| G2.5 | *initialize()* can only be called once. |
| G2.6 | *initialize()* can only be called by an authorized role through appropriate modifiers (e.g. *initializer*, *onlyOwner*). |
| G2.7 | The update process is done in a single transaction so that no one can front-run it. |
| G2.8 | Upgradeable contracts have reserved gap on slots to prevent overwriting. |
| G2.9 | The number of reserved (as a gap) slots has been reduced appropriately if new variables have been added. |
| G2.10 | There are no changes in the order in which the contract state variables are declared, nor their types. |
| G2.11 | New values returned by the functions are the same as in previous versions of the contract (e.g. *owner()*, *balanceOf(address)*). |
| G2.12 | The implementation is initialized. |
| G2.13 | The implementation can't be destroyed. |

| Business Logic | Test Name |
|---|---|
| G4.2 | The contract logic and protocol parameters implementation corresponds to the documentation. |
| G4.3 | The business logic proceeds in a sequential step order and it is not possible to skip steps or to do it in a different order than designed. |
| G4.4 | The contract has correctly enforced business limits. |
| G4.5 | The business logic does not rely on the values retrieved from untrusted contracts (especially when there are multiple calls to the same contract in a single flow). |
| G4.6 | The business logic does not rely on the contract's balance (e.g., *balance == 0*). |
| G4.7 | Sensitive operations do not depend on block data (e.g., *block hash*, *timestamp*). |
| G4.8 | The contract uses mechanisms that mitigate transaction-ordering (front-running) attacks (e.g. pre-commit schemes). |
| G4.9 | The contract does not send funds automatically, but lets users withdraw funds in separate transactions instead. |

| Access Control | Test Name |
|---|---|
| G5.2 | The principle of the least privilege is upheld. Other contracts should only be able to access functions and data for which they possess specific authorization. |
| G5.3 | New contracts with access to the audited contract adhere to the principle of minimum rights by default. Contracts should have a minimal or no permissions until access to the new features is explicitly granted. |
| G5.4 | The creator of the contract complies with the principle of the least privilege and their rights strictly follow those outlined in the documentation. |
| G5.5 | The contract enforces the access control rules specified in a trusted contract, especially if the dApp client-side access control is present and could be bypassed. |
| G5.6 | Calls to external contracts are only allowed if necessary. |
| G5.7 | Modifier code is clear and simple. The logic should not contain external calls to untrusted contracts. |
| G5.8 | All user and data attributes used by access controls are kept in trusted contracts and cannot be manipulated by other contracts unless specifically authorized. |
| G5.9 | the access controls fail securely, including when a revert occurs. |
| G5.10 | If the input (function parameters) is validated, the positive validation approach (whitelisting) is used where possible. |

| Communication | Test Name |
|---|---|
| G6.2 | Libraries that are not part of the application (but the smart contract relies on to operate) are identified. |

| | |
|---|---|
| G6.3 | Delegate call is not used with untrusted contracts. |
| G6.4 | Third party contracts do not shadow special functions (e.g. revert). |
| G6.5 | The contract does not check whether the address is a contract using *extcodesize* opcode. |
| G6.6 | Re-entrancy attacks are mitigated by blocking recursive calls from other contracts and following the Check-Effects-Interactions pattern. Do not use the *send* function unless it is a must. |
| G6.7 | The result of low-level function calls (e.g. *send*, *delegatecall*, *call*) from other contracts is checked. |
| G6.8 | Contract relies on the data provided by the right sender and does not rely on tx.origin value. |

| Arithmetic | Test Name |
|---|---|
| G7.2 | The values and math operations are resistant to integer overflows. Use SafeMath library for arithmetic operations before solidity 0.8.*. |
| G7.3 | the unchecked code snippets from Solidity ≥ 0.8.* do not introduce integer under/overflows. |
| G7.4 | Extreme values (e.g. maximum and minimum values of the variable type) are considered and do not change the logic flow of the contract. |
| G7.5 | Non-strict inequality is used for balance equality. |
| G7.6 | Correct orders of magnitude are used in the calculations. |
| G7.7 | In calculations, multiplication is performed before division for accuracy. |
| G7.8 | The contract does not assume fixed-point precision and uses a multiplier or store both the numerator and denominator. |

| Denial of Service | Test Name |
|---|---|
| G8.2 | The contract does not iterate over unbound loops. |
| G8.3 | Self-destruct functionality is used only if necessary. If it is included in the contract, it should be clearly described in the documentation. |
| G8.4 | The business logic isn't blocked if an actor (e.g. contract, account, oracle) is absent. |
| G8.5 | The business logic does not disincentivize users to use contracts (e.g. the cost of transaction is higher than the profit). |
| G8.6 | Expressions of functions assert or require have a passing variant. |
| G8.7 | If the fallback function is not callable by anyone, it is not blocking contract functionalities. |
| G8.8 | There are no costly operations in a loop. |
| G8.9 | There are no calls to untrusted contracts in a loop. |
| G8.10 | If there is a possibility of suspending the operation of the contract, it is also |

| | |
|---|---|
| | possible to resume it. |
| G8.11 | If whitelists and blacklists are used, they do not interfere with normal operation of the system. |
| G8.12 | There is no DoS caused by overflows and underflows. |

| Blockchain Data | Test Name |
|---|---|
| G9.2 | Any saved data in contracts is not considered secure or private (even private variables). |
| G9.3 | No confidential data is stored in the blockchain (passwords, personal data, token etc.). |
| G9.4 | Contracts do not use string literals as keys for mappings. Global constants are used instead to prevent Homoglyph attack. |
| G9.5 | Contract does not trivially generate pseudorandom numbers based on the information from blockchain (e.g. seeding with the block number). |

| Gas Usage and Limitations | Test Name |
|---|---|
| G10.2 | Gas usage is anticipated, defined and has clear limitations that cannot be exceeded. Both code structure and malicious input should not cause gas exhaustion. |
| G10.3 | Function execution and functionality does not depend on hard-coded gas fees (they are bound to vary). |

| Clarity and Readability | Test Name |
|---|---|
| G11.2 | The logic is clear and modularized in multiple simple contracts and functions. |
| G11.3 | Each contract has a short 1-2 sentence comment that explains its purpose and functionality. |
| G11.4 | Off-the-shelf implementations are used, this is made clear in comment. If these implementations have been modified, the modifications are noted throughout the contract. |
| G11.5 | The inheritance order is taken into account in contracts that use multiple inheritance and shadow functions. |
| G11.6 | Where possible, contracts use existing tested code (e.g. token contracts or mechanisms like *ownable*) instead of implementing their own. |
| G11.7 | Consistent naming patterns are followed throughout the project. |
| G11.8 | Variables have distinctive names. |
| G11.9 | All storage variables are initialized. |
| G11.10 | Functions with specified return type return a value of that type. |

| | |
|---|---|
| G11.11 | All functions and variables are used. |
| G11.12 | *require* is used instead of *revert* in *if* statements. |
| G11.13 | The *assert* function is used to test for internal errors and the *require* function is used to ensure a valid condition in input from users and external contracts. |
| G11.14 | Assembly code is only used if necessary. |

| Test Coverage | Test Name |
|---|---|
| G12.2 | Abuse narratives detailed in the threat model are covered by unit tests. |
| G12.3 | Sensitive functions in verified contracts are covered with tests in the development phase. |
| G12.4 | Implementation of verified contracts has been checked for security vulnerabilities using both static and dynamic analysis. |
| G12.5 | Contract specification has been formally verified. |
| G12.6 | The specification and results of the formal verification is included in the documentation. |

| Decentralized Finance | Test Name |
|---|---|
| G14.1 | The lender's contract does not assume its balance (used to confirm loan repayment) to be changed only with its own functions. |
| G14.2 | Functions that change lenders' balance and/or lend cryptocurrency are non-re-entrant if the smart contract allows borrowing the main platform's cryptocurrency (e.g. Ethereum). It blocks the attacks that update the borrower's balance during the flash loan execution. |
| G14.3 | Flash loan functions can only call predefined functions on the receiving contract. If it is possible, define a trusted subset of contracts to be called. Usually, the sending (borrowing) contract is the one to be called back. |
| G14.4 | If it includes potentially dangerous operations (e.g. sending back more ETH/tokens than borrowed), the receiver's function that handles borrowed ETH or tokens can be called only by the pool and within a process initiated by the receiving contract's owner or another trusted source (e.g. multisig). |
| G14.5 | Calculations of liquidity pool share are performed with the highest possible precision (e.g. if the contribution is calculated for ETH it should be done with 18 digit precision - for Wei, not Ether). The dividend must be multiplied by the 10 to the power of the number of decimal digits (e.g. dividend * 10^18 / divisor). |
| G14.6 | Rewards cannot be calculated and distributed within the same function call that deposits tokens (it should also be defined as non-re-entrant). This protects from momentary fluctuations in shares. |
| G14.7 | Governance contracts are protected from flash loan attacks. One possible |

| | |
|---|---|
| | mitigation technique is to require the process of depositing governance tokens and proposing a change to be executed in different transactions included in different blocks. |
| G14.8 | When using on-chain oracles, contracts are able to pause operations based on the oracles' result (in case of a compromised oracle). |
| G14.9 | External contracts (even trusted ones) that are allowed to change the attributes of a project contract (e.g. token price) have the following limitations implemented: thresholds for the change (e.g. no more/less than 5%) and a limit of updates (e.g. one update per day). |
| G14.10 | Contract attributes that can be updated by the external contracts (even trusted ones) are monitored (e.g. using events) and an incident response procedure is implemented (e.g. during an ongoing attack). |
| G14.11 | Complex math operations that consist of both multiplication and division operations first perform multiplications and then division. |
| G14.12 | When calculating exchange prices (e.g. ETH to token or vice versa), the numerator and denominator are multiplied by the reserves (see the *getInputPrice* function in the *UniswapExchange* contract). |

# Audit Findings

## Centralization Risk

During our research, we noticed that many aspects of the contracts are highly centralized. Oracles and admins can perform dramatic actions that could possibly break the contract. The following findings demonstrate specific cases. The mitigation section in each finding concerns only that particular case. However, we recommend employing a more general strategy.

One such mitigation strategy would be to rethink the overall architecture of the project such that no one account can make platform-changing decisions. A less drastic mitigation method would be to establish multisig accounts for these roles.

### [H] Changing Genesis Time Can Potentially Disrupt Reward Distribution

| | |
|---|---|
| ID | SAY-01 |
| Status | Fixed |
| Risk | High |
| Business Impact | `rewardsGenesisTime` is a critical variable that tracks the time when the rewards were started and thus governs reward distribution.<br><br>The rate of rewards continues to decrease over time slowly reducing to 85% of the previous cycle at some point in future. Therefore, resetting genesis time after beginning reward distribution may disrupt the whole scheme. |
| Location | - `Reward.sol; resetRewardsGenesisTime()`<br>- `Reward.sol; manuallySetRewardsGenesisTime(uint256)` |
| Description | Changing `rewardsGenesisTime` later in the cycle later will impact the crucial `currentWeek` and `weekLimit` variables and could potentially mess up the whole reward distribution scheme and its projected schedule. |
| Mitigation | One possible mitigation is to revise these functions such that resetting genesis time is only possible before reward distribution has started: |

```
function resetRewardsGenesisTime() external onlyRole(ADMIN_ROLE) {
        require(dimoTotalSentOutByContract==0,"Reward Distribution
already started");
        rewardsGenesisTime = block.timestamp;
}
```

If resetting or changing the variable after the fact is necessary, perhaps it can be handled in a controlled and predictable manner on-chain, rather than allowing admins arbitrary control.

## [H] Unchecked Admin Withdrawal

| | |
|---|---|
| ID | SAY-02 |
| Status | Fixed |
| Risk | High |
| Business Impact | While admin accounts by definition are given a measure of trust, if they are compromised, they can be used to drain the contract of its funds using this function. |
| Location | - `Reward.sol; adminWithdraw(address, uint256)` |
| Description | The function `adminWithdraw(address, uint256)` allows admins to send contract funds to users without limitation. According to the documentation, this is used if users send DIMO to the contract without staking. However, we believe this function may be liable to abuse, as explained in the business impact section. |
| Mitigation | One conceivable way of increasing security is to submit a request on behalf of a user, perhaps using an account with the oracle role. An admin then has to approve the transaction. This at least requires two accounts to approve such transactions, providing another layer of security. |

### [H] Admins Can Reset the Registry at Will

| | |
|---|---|
| ID | SAY-03 |
| Status | Acknowledged |
| Risk | High |
| Business Impact | Registry is where the user data is stored and validations are performed. If the registry is reset, existing users will stop receiving rewards unless the data is migrated to the new registry. |
| Location | - `Reward.sol; setRegistryContractAddress(address)` |
| Description | `setRegistryContractAddress(address)` allows admins to reset the registry address at will. |
| Mitigation | One solution is to make sure that this function reverts unless the registry is not set in the first place.<br>`function setRegistryContractAddress(`<br>`  address registryContractAddress`<br>`) external onlyRole(ADMIN_ROLE) {`<br>`  require(`<br>`  registryContractAddress != address(0),`<br>`  "registryContractAddress is an invalid zero address"`<br>`  );`<br>`  require(address(registry)==address(0)),"already set, cannot be set again")`<br>`  registry = IRegistry(registryContractAddress);`<br>`}`<br><br>Another solution is to set up a data migration procedure. If such a procedure is already in place, then this finding can be safely ignored. |

### [I] Deployer has Both Admin and Oracle Roles

| | |
|---|---|
| ID | SAY-04 |
| Status | Acknowledged |
| Risk | Informational |
| Business Impact | We decided to rate this finding as informational because it is mainly a corollary to the main centralization concerns detailed above. With the current way the system is built, it is necessary for someone to assign and manage the Oracle and Admin roles. However, it is not necessary for that account to hold these roles itself. |
| Location | - `Reward.sol:108-111; initialize(address, address, address)` |
| Description | During initialization, the deployer receives both admin, default admin and oracle roles:<br>`_setupRole(DEFAULT_ADMIN_ROLE, msg.sender);`<br><br>`_setupRole(ORACLE_ROLE, msg.sender);`<br>`_setupRole(ADMIN_ROLE, msg.sender);` |
| Mitigation | Consider only giving the default admin role, inherited from OA's `AccessControlUpgradeable`, to the deployer, allowing it to manage roles for other users. |

## [M] Use `AccessControlDefaultAdminRulesUpgradeable`

| | |
|---|---|
| ID | SAY-05 |
| Status | Acknowledged |
| Risk | Medium |
| Business Impact | Given the inherent centralization of the contract and the importance of admin roles (and therefore their management), we decided to rate this finding as medium risk. |
| Location | - |
| Description | `Reward.sol` currently uses `AccessControlUpgradable` to grant and manage access to critical functions. In OZ's access control scheme, the account possessing `DEFAULT_ADMIN_ROLE` is capable of granting and withdrawing all other roles. In the current state of the contract it is given to the deployer upon initialization.<br><br>`AccessControlDefaultAdminRulesUpgradeable` extends `AccessControlUpgradable` with two crucial security features:<br>• Only one account can hold `DEFAULT_ADMIN_ROLE`.<br>• `DEFAULT_ADMIN_ROLE` can only be transferred using a two-step process. A configurable delay between the two steps, `changeDefaultAdminDelay`, is enforced. |
| Mitigation | Switch from `AccessControlUpgradable` to `AccessControlDefaultAdminRulesUpgradeable`. |

## [L] Insufficient Input Validation in `manuallySetRewardsGenesisTime(uint256)`

| | |
|---|---|
| ID | SAY-06 |
| Status | Fixed |
| Risk | Low |
| Business Impact | We rate this issue as low, since a mistaken value could be rectified by the admin just as easily as it was given. |
| Location | - `Reward.sol; manuallySetRewardsGenesisTime(uint256)` |
| Description | Beside posing a centralization risk, as detailed above, `manuallySetRewardsGenesisTime(uint256)` also fails to validate the timestamp given by the admin. It would happily accept a future time, leading to `_getNumberOfWeeksSinceGenesis()` to revert:<br>`function _getNumberOfWeeksSinceGenesis()`<br>`  private`<br>`  view`<br>`  returns (uint256 unixTimeDiff)`<br>`{`<br>`  unixTimeDiff = (block.timestamp - rewardsGenesisTime) / 7 days;`<br>`}` |
| Mitigation | Make sure that the input is equal or smaller than the current timestamp. |

## [L] Insufficient Input Validation in `setMinimumTimeForRewards(uint256)`

| | |
|---|---|
| ID | SAY-07 |
| Status | Fixed |
| Risk | Low |
| Business Impact | Just like the above finding, we decided to rate this finding as low risk because it can easily be rectified by calling the function again. |
| Location | - `Reward.sol; setMinimumTimeForRewards(uint256)` |
| Description | This finding is very similar in substance to the one directly above it. Too high of a value will render users unable to claim their rewards. This, in tandem with the fact that rewards are diminished on a weekly basis by `_limitForWeek(uint256)`, will neuter the advantage conferred by early adoption. |
| Mitigation | Define an acceptable range of values for `minimumTimeForRewards` and enforce it through the function. Because of the direct influence this value has on the platform's entire reward structure, managing it carefully is paramount. |

## [L] Unused Error

| | |
|---|---|
| ID | SAY-08 |
| Status | Fixed |
| Risk | Low |
| Business Impact | We decided to rate this issue as low rather than informational, this seemingly accidental omission does have some impact on the contract's logic. |
| Location | - `Reward.sol:12` |
| Description | The error `InvalidArrayLegnth()` is declared on line 12, but never used. |
| Mitigation | We hypothesize that this error was supposed to be thrown in `batchTransfer(TransferInfo[])` if the supplied array does not have eight members:<br>`if(transferInfos.length ≠ 8) revert InvalidArrayLength();` |

## [L] Unchecked Return Value

| | |
|---|---|
| ID | SAY-09 |
| Status | Fixed |
| Risk | Low |
| Business Impact | The transfer function of ERC20 tokens returns the success or failure of the transfer as a boolean. It is considered good practice to check this value and only proceed if it succeeds. However, because the documentation implies that `batchTransfer(TransferInfo[])` is only used to transfer Dimo tokens, we rate this finding as low. |
| Location | - `Reward.sol:232; batchTransfer(TransferInfo[])` |
| Description | On the indicated line, a `dimoToken` transfer is made, but the return value is left unchecked. |
| Mitigation | Revert with an error if the transfer fails:<br>`if (!dimoToken.transfer(user, amount)) revert TokenTransferFailed();` |

## [L] Deprecated API Call

| | |
|---|---|
| ID | SAY-10 |
| Status | Acknowledged |
| Risk | Low |
| Business Impact | Unlike `grantRole(bytes32, address)`, `_setupRole(bytes32, address)` does not perform any checks on the calling account. It has therefore been deprecated. |
| Location | - `Reward.sol:108-111; initialize(address, address, address)` |
| Description | The function `_setupRole(bytes32, address)`, used in the specified location, has been deprecated in OpenZepplin 5.0. Calling `grantRole(bytes32, address)` instead is now recommended. |
| Mitigation | Replace calls to `_setupRole(bytes32, address)` with `grantRole(bytes32, address)`. |

## [I] Insufficient Event Emission

| | |
|---|---|
| ID | SAY-11 |
| Status | Fixed |
| Risk | Informational |
| Business Impact | Many monitoring tools, frontends, off-chain toolings and reporting services rely on events to capture real time activities of contracts. Moreover, protocols can react quickly to suspicious event emissions. |
| Location | - `Reward.sol; setRegistryContractAddress(address)`<br>- `Reward.sol; setMinimumTimeForRewards(uint256)`<br>- `Reward.sol; setSyntheticProxyAddress(address)`<br>- `Reward.sol; resetRewardsGenesisTime()`<br>- `Reward.sol; manuallySetRewardsGenesisTime()`<br>- `Reward.sol:267-268; batchTransfer(TransferInfo[])` |
| Description | The aforementioned functions do not emit events for important state updates. |
| Mitigation | Add event emissions. |

## [I] Solidity Versioning

| | |
|---|---|
| ID | SAY-12 |
| Status | Acknowledged |
| Risk | Informational |
| Business Impact | The contract can be compiled and tested with different compiler versions during development and review and another different compiler version during deployment to mainnet. This can lead to unexpected results. |
| Location | `Reward.sol:2` |
| Description | The contract specifies its pragma as `pragma solidity ^0.8.13;`. This allows usage of any version of solidity starting from 0.8.13. Moreover, 0.8.13 is not the most recent version. |
| Mitigation | Decide on a single version of solidity to use. The latest stable (and therefore recommended) release is 0.8.19. |

## [I] Index the `week` Parameter in `TokensTransferredForConnectionStreak()`

| | |
|---|---|
| ID | SAY-13 |
| Status | Fixed |
| Risk | Informational |
| Business Impact | Indexing is useful for filtering events in Ethereum logs. Each indexed parameter in an event adds a topic to the event log, making it easier to search for specific events using those parameters. |
| Location | - `Reward.sol:81` |
| Description | The `week` parameter represents the `currentWeek` on which the airdrop was distributed. Since indexing makes it easier for frontend applications to filter specific events, indexing this parameter will make it easy to filter for users that received airdrops at a given week. |
| Mitigation | Consider indexing the specified parameter. |

## [I] Adherence to the Solidity Style Guide

| | |
|---|---|
| ID | SAY-14 |
| Status | Fixed |
| Risk | Informational |
| Business Impact | This issue is purely informational. There is no impact on the security posture of the contract or otherwise. |
| Location | - `Reward.sol:36` |
| Description | `TransferInfo[]`, a struct, is defined on line 36, directly after the state variables. The Solidity style guide recommends placing struct declarations before. |
| Mitigation | Move the declaration of the struct directly to the beginning of the contract, before state variables. |