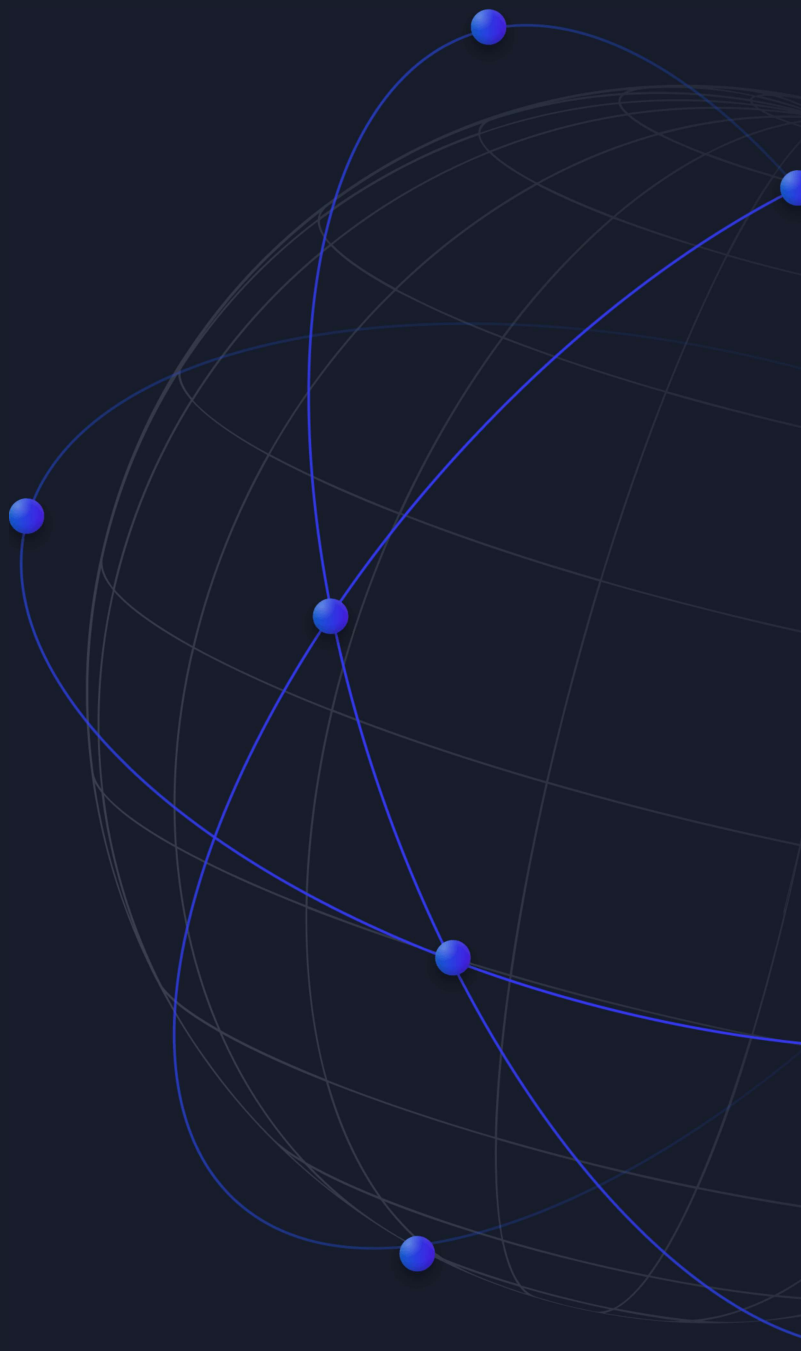
Black paper

Lay3rs

Security audit



2nd review
Date: 16/02/2024

Summary

LOW-1 Simplify URI handling using ERC1155URIStorageUpgradeable

LOW-2 Compliance with ERC1155 standard recommendations Fixed

LOW-3 Redundant check and range clarification in `updateReserveAllocatedPercentage` function Fixed

LOW-4 Inconsistent event argument in `RewardsDistributed` event Fixed

LOW-5 Absence of _disableInitializers() invocation in upgradeable Contracts Fixed

LOW-6 Redundant check in donate function Fixed

LOW-7 ERC165 Implementation for Project Validation Fixed

LOW-8 Lack of Check for Non-Null Merkle Root Fixed

LOW-9 Burn functionality in NFT_DatasetUpgradeable.sol Fixed

LOW-10 Unused _burn function override Acknowledge

INF-1 Remove unnecessary `canClaim` mapping

INF-2 Simplify Emitted Event in `withdraw` Function Fixed

INF-3 Unnecessary `contractAddress` Variable from `ContributorShareUpdate` Event Fixed

INF-4 Useless check in withdraw function Fixed

INF-5 Variable shadowing Fixed

INF-6 Missing check for _uri parameter in createExport function Fixed

INF-7 Redundant check in mint function Fixed

INF-8 Restricting safeBatchTransferFrom function to pure Fixed

INF-9 Restricting safeTransferFrom function to pure Fixed

INF-10 Smart contracts interface inheritance Fixed

INF-11 Redundant Fallback Function Reversion Fixed

INF-12 Useless LAY Token Variable in DAOMaster.sol Fixed

INF-13 Unnecessary nonReentrant Modifier in batchTransfer Function Fixed

I. Introduction

1. About Black Paper

Black Paper has been created to help developer teams. Our goal is to help you to make your smart contract safer.
Cybersecurity requires specific expertise which is very different from smart contract development logic. To ensure everything is well fixed, we stay available to help you.

2. Methodology

a. Preparation

A first technical meeting was held on 15/12/2023. It allows the technical team to explain the contract workflow, to define the exact scope, and start the audit process. Black Paper team has interacted with the Lay3rs for few details to allow the best result in terms of security.

The initial tests were done for the non-upgradeable contracts. Black Paper team made the choice to internally replace the Truffle framework by a Hardhat one to upgrade most of the tests.

b. Review

Before manually auditing, we pass contracts into automatic tools. This allows us to find some easy-to-find vulnerabilities.
Afterward, we manually go deeper. Every variable and function in the scope are analyzed.

You can find many articles on the lesson website. Here is a snippet list of what we test :

- Constructor Mismatch
- Ownership Takeover
- Redundant Fallback Function
- Overflows & Underflows
- Reentrancy
- Money-Giving Bug
- Blackhole
- Unauthorized Self-Destruct
- Revert DoS
- Unchecked External Call
- Gasless Send
- Send Instead Of Transfer
- Costly Loop
- Use Of Untrusted Libraries
- Use Of Predictable Variables
- Transaction Ordering Dependence
- Deprecated Uses

This is not an exhaustive list since we also focus on logic execution exploits, and help optimizing gas price.

c. Reporting

Every point in the code is subject to internal discussions with the team. At this stage, a majority of the probable issues have already been identified and documented.

Post the completion of the code review, analysis, and testing, we prepare a report. For each vulnerability, it contains the following informations.

- Description
- Severity
- Recommendation

Here are severity score definitions.

Critical

A critical vulnerability is a severe issue that can cause significant damage to the contract and its users. These vulnerabilities are easy to exploit and can result in the loss of funds, theft of sensitive data, or other serious consequences. Immediate attention is required to address these vulnerabilities.

Major

A major vulnerability is an issue that can cause significant problems for the contract and its users, but not to the same extent as a critical vulnerability. These vulnerabilities are also easy to exploit and may result in the loss of funds or other negative consequences, but they can be mitigated with timely action.

Medium

A medium vulnerability is an issue that could potentially cause problems for the contract and its users, but the difficulty to exploit is higher than major or critical vulnerabilities. These vulnerabilities may pose a risk to the contract's functionality or security, but they can be addressed without causing significant disruption.

Low

A low vulnerability is a minor issue that does not pose a significant risk to the contract or its users. These vulnerabilities are difficult to exploit and may be cosmetic or technical in nature, but they do not compromise the contract's security or functionality.

Informational

An informational finding is not a vulnerability but rather a suggestion or recommendation for improvement. These findings may include best practices for contract design, suggestions for improving code readability, or other non-critical issues. While not urgent, addressing these findings can help to optimize the contract's performance and reduce the risk of future vulnerabilities.

3. Disclaimer

In this audit, we sent all vulnerabilties found by our team. **We can't guarantee all vulnerabilities have been found.**

4. Scope

Defining the scope of a smart contract audit is crucial for ensuring that all aspects of the
contract are thoroughly reviewed and any potential issues are identified. At Black Paper, we
highlight the importance of defining the scope upfront and ensuring that it is comprehensive
enough to cover all relevant components of the contract. This helps us to provide a comprehensive audit report that accurately identifies any potential vulnerabilities and
provides recommendations for remediation.

The following smart contracts are considered in scope (commit 060990d49bc1f8787bca911d5daa17d24eeb8086):
- NFT_DatasetUpgradeable.sol
- NFT_ExportUpgradeable.sol
- ProjectUpgradeable.sol
- INFTDataset.sol
- IProject.sol
- LAY.sol
- DAOMaster.sol

Other libraries and smart contracts are considered safe.

II. Vulnerabilities

CRIT-1 Lack of Pausing Functionality in Lay.sol ERC20 Contract Fixed

Impact: Critical

Description:

The Lay.sol ERC20 contract uses the OpenZeppelin Pausable functionality but did not include the `whenNotPaused` modifier in critical functions like `transfer` and `transferFrom`. This omission allows transactions to be processed even when the contract is in a paused state.

Recommendation:

Implement the `whenNotPaused` modifier in critical functions of the Lay.sol ERC20 contract to ensure transactions are only processed when the contract is not paused. This can be achieved by either using OpenZeppelin's ERC20Pausable extension or manually checking the paused state in each function.

Status:

A fix is implemented, following recommendations.

CRIT-2 Unsafe initialization for reserveAllocatedPercentage variable Fixed

Impact: Critical

Description:

In the `ProjectUpgradeable.sol` contract, the `reserveAllocatedPercentage` variable is assigned an initial value directly in its declaration. To enhance contract upgradeability and maintain best practices, it is recommended to move the assignment of `reserveAllocatedPercentage` to an initializer function. This ensures proper initialization and aligns with upgrade-safe patterns.

Recommendation:

Refactor the contract as follows:

```
// Contract declaration
contract ProjectUpgradeable is Initializable, UUPSUpgradeable {
    // Percentage allocated for reserve at each sale of export
    uint256 public reserveAllocatedPercentage;

    // Upgradeable constructor
    function initialize() initializer public {
        reserveAllocatedPercentage = 50; // 50 % by default
        __Ownable_init();
        __Pausable_init();
        __ReentrancyGuard_init();

        // Additional initialization logic...
    }
}
```

Status:

A fix is implemented, following recommendations.

CRIT-3 Unsafe initilization for currentTermsOfServiceVersion variable Fixed

Impact: Critical

Description:

In the `ProjectUpgradeable.sol` contract, the `currentTermsOfServiceVersion` variable is assigned an initial value directly in its declaration. To enhance contract upgradeability and maintain best practices, it is recommended to move the assignment of `currentTermsOfServiceVersion` to an initializer function. This ensures proper initialization and aligns with upgrade-safe patterns.

Recommendation:

Refactor the contract as follows:

```
// Contract declaration
contract ProjectUpgradeable is Initializable, UUPSUpgradeable {
    // Could be updatable by the owner
    uint8 public currentTermsOfServiceVersion; // By default (v1 to v255)

    // Upgradeable constructor
    function initialize() initializer public {
        currentTermsOfServiceVersion = 1;
        __Ownable_init();
        __Pausable_init();
        __ReentrancyGuard_init();

        // Additional initialization logic...
    }
}
```

Status:

A fix is implemented, following recommendations.

CRIT-4 Unsafe Initialization for MINIMUM_CONTRIBUTION_AMOUNT variable Fixed

Impact: Critical

Description:

In the `ProjectUpgradeable.sol` contract, the `MINIMUM_CONTRIBUTION_AMOUNT` variable is assigned an initial value directly in its declaration. To enhance contract upgradeability and maintain best practices, it is recommended to move the assignment of `MINIMUM_CONTRIBUTION_AMOUNT` to an initializer function. This ensures proper initialization and aligns with upgrade-safe patterns.

Recommendation:

Refactor the contract as follows:

```
// Contract declaration
contract ProjectUpgradeable is Initializable, UUPSUpgradeable {
    // Define state variables
    uint256 public MINIMUM_CONTRIBUTION_AMOUNT;

    // Upgradeable constructor
    function initialize() initializer public {
        MINIMUM_CONTRIBUTION_AMOUNT = 1000e18; // = 1000 LAY
        __Ownable_init();
        __Pausable_init();
        __ReentrancyGuard_init();

        // Additional initialization logic...
    }
}
```

Status:

A fix is implemented, following recommendations.

MAJ-1 Unfair rewards distribution Fixed

Impact: Major

Description:

The `distributeRewardsByBatch` function uses the `updateLAYRewardsAmount` function to update the `currentRewardsAmount`. However, if the `updateLAYRewardsAmount` is called between two reward distributions batches, it may result in an unfair distribution. The second part of the reward can be higher due to the increased `currentRewardsAmount`.

Recommendation:

To address the issue and ensure fair reward distribution, it is recommended to add a check within the `updateLAYRewardsAmount` function to prevent updates to `currentRewardsAmount` during the distribution process. If `isDistributingRewards` is set to `true`, indicating that the rewards distribution is in progress, the function should revert to prevent any modifications to the rewards amount during the distribution.

Replace the current implementation of `getClaimableAmount` with the following modification:

```
function updateLAYRewardsAmount(
        uint256 _amount
    ) external onlyExportContract whenNotDistributing {
        // First, we calculate the reserve amount
        uint256 reserveLAYAmount = FullMath.mulDiv(_amount, reserveAllocatedPercentage, 100);
        // We update the balance of reserve accordingly
        currentReserveAmount += reserveLAYAmount;
        // We update the rewards balance with the rest
        currentRewardsAmount += (_amount - reserveLAYAmount);
        // Finally, we emit the event to notify about this update of rewards balance
        emit UpdateLAYRewardsAmount(_amount);
    }
```

This modification ensures that the `updateLAYRewardsAmount` function reverts if rewards distribution is in progress, preventing any interference with the reward calculation during the distribution process. The downsize of this modification is that it involves an impossibility to mint exports during the reward distribution.

Status:

A fix is implemented, following recommendations.

MAJ-2 Front-running vulnerability in `invest` function

Impact: Major

Description:

The `invest` function in the smart contract is susceptible to front-running, allowing an attacker to manipulate the invested amount before reaching the hard cap. The recalculation of the investment amount, when exceeding the hard cap, creates an opportunity for front-running users to force the victim to invest less than intended.

Recommendation:

To mitigate the front-running vulnerability, it is recommended to revert the transaction if the calculated investment amount exceeds the hard cap. Remove the recalculation logic and instead ensure that the entire desired investment amount is transferred, preventing front-running users from manipulating the invested amount.

Replace the following code block:

```
// In the case the contributor has invested an amount that exceeds the hard cap
if (
    milestone.layHardcap <
    (milestone.currentAmountLAY +
        milestone.autofinancingAmountLAY +
        _amount)
) {
    // In this case, we only transfer the amount required
    _amount =
        milestone.layHardcap -
        milestone.currentAmountLAY -
        milestone.autofinancingAmountLAY;
}
```

With the following code:

```
// Revert if the investment amount exceeds the hard cap
if (
    milestone.layHardcap <
    (milestone.currentAmountLAY +
        milestone.autofinancingAmountLAY +
        _amount)
) {
    revert HardcapExceeded();
}
```

This modification ensures that the entire desired investment amount is transferred, and the transaction is reverted if it exceeds the hard cap.

Moreover, this change breaks the logic so we recommend to add a `layLowcap`.

Status:

16/02: A different fix was implemented to allow the user reaching the hardcap to invest less than the `MINIMUM_CONTRIBUTION_AMOUNT` . This fix is not safe since it allows any user to call the invest function with the `_amount` equal to 0 when the hardcap is reached. This involves a lot of critical behaviors such as receiving rewards more than one time. We recommend to add a check on `_amount`.

MAJ-3 Private Visibility for State Variables in ProjectUpgradeable.sol Fixed

Impact: Major

Description:

In the upgradeable version of `Project.sol` (named `ProjectUpgradeable.sol`), the state variables `tokenLAY`, `daoMaster`, and `nftExport` are declared as private. This differs from the original `Project.sol` contract, where these variables are declared as public. The change in visibility lead to breaking changes in specs. Moreover, it breaks tests that assume these variables are public.

Recommendation:

It is recommended to align the visibility of the state variables in `ProjectUpgradeable.sol` with the original `Project.sol` contract to avoid breaking changes and maintain consistency. Make the variables `tokenLAY`, `daoMaster`, and `nftExport` public in `ProjectUpgradeable.sol`. Update the contract as follows:

```
// ProjectUpgradeable.sol

// LAY Token smart contract
IERC20 public immutable tokenLAY;

// DAO Master smart contract
address public immutable daoMaster;

// NFT Export smart contract
address public immutable nftExport;
```

Status:

A fix is implemented, following recommendations.

MED-1 Move `verifyVote` function off-chain Acknowledge

Impact: Medium

Description:

The `verifyVote` function in `DAOMaster.sol` is designed to check the validity of a vote using a Merkle proof. However, since the contract owner is the only entity capable of creating and committing proposals, and other users do not have any power over the functions, this check does not provide any additional trust than off-chain information.

Recommendation:

Consider moving the `verifyVote` functionality and the `merkleRoot` information off-chain, as it currently does not enhance the trust or security of on-chain operations.

Status:

Lay3rs team prefer to keep merkle root storage on-chain as a unique point of trust for everyone. They are aware this is not a complete DAO since there are no on-chain votes.

MED-2 Lack of Maximum Deadline Range for Milestones Fixed

Impact: Medium

Description:

In the contract ProjectUpgradeable, there is a potential issue where the contract might get stuck in the `RUNNING` state if the set deadline is unreasonably high and the milestone is not reached. The function to postpone the deadline currently checks if the deadline has already been reached using `if (milestone.deadline <= block.timestamp) revert MilestoneDeadlineReached();`.

To prevent the contract from being stuck in the `RUNNING` state for an extended period, making it useless, it is recommended to implement a maximum range for the milestone deadline.

Recommendation:

Add a check to ensure that the deadline does not exceed a reasonable maximum duration. This prevents scenarios where a very high deadline might cause the contract to be stuck in the `RUNNING` state for an extended period.

1. Add

```
uint256 public constant MAX_MILESTONE_DEADLINE_DURATION = 30 days; // Set your desired maximum duration
```

1. In `createMilestone` function, replace

```
        // TODO: Should we specify a specific range of deadline?
        if (block.timestamp > _deadline) revert InvalidDeadline();
```

With

```
        if (deadline > block.timestamp + MAX_MILESTONE_DEADLINE_DURATION) revert InvalidDeadline();
        if (block.timestamp > _deadline) revert InvalidDeadline();
```

Adjust the `MAX_MILESTONE_DEADLINE_DURATION` constant according to your project's requirements.

Status:

A fix is implemented, following recommendations. MAX_MILESTONE_DEADLINE_DURATION is set to 60 days.

MED-3 Improved rewards distribution for dataset operators Fixed

Impact: Medium

Description:

In the `_distributeLAYRewards` function, during the distribution of rewards to dataset operators, the use of Euclidean division in each iteration may result in a loss of LAY tokens due to rounding. To address this potential loss, it is recommended to introduce a separate variable, `datasetOffset`, to accumulate the dataset operator rewards in each iteration. This way, the total distributed amount can be checked against the originally calculated `datasetLAYAmount`, and any remaining tokens (the "lost" amount) can be transferred to a designated address.

Recommendation:

Refactor the relevant section of the code as follows:

```
// Initialize datasetOffset to keep track of accumulated rewards for dataset operators
uint256 datasetOffset = 0;

// Iterate across the DatasetShare array to distribute rewards
for (uint256 i = 0; i < datasetsLength; i++) {
    // Retrieve the dataset
    DatasetShare storage datasetShare = export.datasets[i];

    // Retrieve the operator address
    address datasetOperatorAddress = nftDataset.getOperatorAddress(datasetShare.datasetId);

    // Calculate the amount of LAY to transfer
    uint256 datasetOperatorLAYAmount = FullMath.mulDiv(
        datasetLAYAmount,
        datasetShare.scarcityPondered,
        export.datasetTotalScarcityWeight
    );

    // Transfer the allocated dataset operator amount to the appropriate address
    offsetLAY += _safeRewardsTransfer(_exportId, datasetOperatorAddress,
datasetOperatorLAYAmount);

    // Accumulate the dataset operator rewards in datasetOffset
    datasetOffset += datasetOperatorLAYAmount;
}

// Calculate the "lost" amount that can be sent to any address
uint256 lostLAYAmount = datasetLAYAmount - datasetOffset;

// Transfer the remaining "lost" amount to a designated address
if (lostLAYAmount > 0) {
    // Specify the address to receive the "lost" amount
    address designatedAddress = // Set the desired address;
    _safeRewardsTransfer(_exportId, designatedAddress, lostLAYAmount);
}
```

Status:

A fix was implemented. It is a bit different from the snippet but it is valid too.

MED-4 Redundant checks in _safeRewardsTransfer function Acknowledge

Impact: Medium

Description:

In the `_safeRewardsTransfer` function, unnecessary checks are implemented to verify the recipient address and the transferred amount. Here is why this function should be removed :

1. If a role, such as the association or the owner, has a reward percentage set to 0, it results in a scenario where it reverts. In such cases, this configuration make the minting of tokens impossible.
2. The Lay ERC20 transfer function already reverts if the recipient address is zero.
3. The actual implementation of the LAY token's transfer function will revert if the amount is zero.

Recommendation:

Remove each usage of the `_safeRewardsTransfer` function and replace it by:

```
tokenLAY.transfer(_recipient, _amount);
```

Status:

It is not considered as a vulnerability since it prevents from Denial of Service due to any transfer revert.

MED-5 Unsafe sequential minting order Fixed

Impact: Medium

Description:

In the `NFT_DatasetUpgradeable.sol` contract, the `safeMint` function uses the `_safeMint` function to mint a new NFT. However, the current order of operations within the function can lead to undesirable behavior, where the token is minted before the dataset is fully set. While there is no direct vulnerability within the contract, this sequencing could potentially create vulnerabilities in external contracts interacting with this one.

```
/**
 * @dev Mint a new NFT to the given address
 * @notice Counter is auto-incremented using Counters library
 * @notice Only a project owner address can call this function
 * @notice The contract must not be paused
 * @param _to {{address}} - New minted NFT owner address
 * @param _uri {{string}} - URI of the minted NFT
 * @param _projectAddress {{address}} - Project owner address related to this new minted NFT
 * @param _koSize {{uint256}} - Size in Ko of the minted NFT
 * @param _scarcityLevel {{ScarcityLevel}} - Scarcity level of the minted NFT (from 0 to 9)
 */
function safeMint(
    address _to,
    string memory _uri,
    address _projectAddress,
    uint256 _koSize,
    ScarcityLevel _scarcityLevel
) public onlyProjectOwnerByAddress(_projectAddress) whenNotPaused {
    // First we increment the counter to avoid duplicate NFT Id
    uint256 tokenId = _tokenIdCounter.current();
    _tokenIdCounter.increment();

    // Then we mint the NFT
    _safeMint(_to, tokenId);
    // Then we set the URI
    _setTokenURI(tokenId, _uri);
    // Finally we set the dataset data to the new minted NFT
    _setDataset(tokenId, _to, _projectAddress, _koSize, _scarcityLevel);
}
```

Recommendation:

It is recommended to adjust the order of operations in the `safeMint` function to ensure that the dataset is fully set before minting the NFT. This change aims to prevent any undesired side effects or vulnerabilities that may arise from external contracts relying on the correct sequential order.

Replace the existing `safeMint` function with the following:

```
/**
 * @dev Mint a new NFT to the given address
 * @notice Counter is auto-incremented using Counters library
 * @notice Only a project owner address can call this function
 * @notice The contract must not be paused
 * @param _to {{address}} - New minted NFT owner address
 * @param _uri {{string}} - URI of the minted NFT
 * @param _projectAddress {{address}} - Project owner address related to this new minted NFT
 * @param _koSize {{uint256}} - Size in Ko of the minted NFT
 * @param _scarcityLevel {{ScarcityLevel}} - Scarcity level of the minted NFT (from 0 to 9)
 */
function safeMint(
    address _to,
    string memory _uri,
    address _projectAddress,
    uint256 _koSize,
    ScarcityLevel _scarcityLevel
) public onlyProjectOwnerByAddress(_projectAddress) whenNotPaused {
    // First we increment the counter to avoid duplicate NFT Id
    uint256 tokenId = _tokenIdCounter.current();
    _tokenIdCounter.increment();

    // We set the dataset data to the new minted NFT
    _setDataset(tokenId, _to, _projectAddress, _koSize, _scarcityLevel);
    // Then we mint the NFT
    _safeMint(_to, tokenId);
    // Finally, we set the URI
    _setTokenURI(tokenId, _uri);
}
```

This modification ensures that the dataset is fully set before minting the NFT, providing a more secure and predictable behavior for external contracts interacting with this one.

Status:

A fix is implemented, following recommendations.

MED-6 Ineffective Protection Against MATIC Ownership in Smart Contracts Fixed

Impact: Medium

Description:

The contracts `DAOMaster.sol`, `Lay.sol`, `NFT_DatasetUpgradeable.sol`, `NFT_ExportUpgradeable.sol`, and `ProjectUpgradeable.sol` include logic to prevent the ownership of MATIC tokens by smart contracts. Two different methods have been implemented for this purpose:

```
/**
 * @dev Receive function to handle transfers of MATIC (Ether on Ethereum)
 */
receive() external payable {
    // Redirect received MATIC to the owner's address
    payable(owner()).transfer(msg.value);
}

/**
 * @dev Fallback that reverts any method not referenced in the contract
 */
fallback() external payable {
    revert FunctionNotImplemented();
}
```

```
/**
 * @dev Receive function to handle transfers of MATIC (Ether on Ethereum)
 */
receive() external payable {
    // Redirect received MATIC to the owner's address
    payable(owner()).transfer(msg.value);
}

/**
 * @dev Fallback function to handle transfers of ERC20 tokens
 */
fallback() external payable {
    if (msg.sender == address(tokenLAY)) {
        // Transfer received LAY tokens to the owner's address
        uint256 balance = tokenLAY.balanceOf(address(this));
        tokenLAY.transfer(owner(), balance);
    }
}
```

However, both methods are not entirely effective in preventing the receipt of MATIC, as a malicious user can exploit the `selfdestruct` function to force MATIC ownership by the smart contract.

Recommendation:

It is recommended to remove the existing functionality, as it cannot fully prevent the receipt of MATIC. Instead, a more secure approach is advised, such as implementing a dedicated function to withdraw MATIC. The following is a suggested modification:

```
/**
 * @dev Function to withdraw MATIC from the contract
 */
function withdrawMATIC() external {
    require(msg.sender == owner(), "Only the owner can withdraw MATIC");
    uint256 balance = address(this).balance;
    payable(owner()).transfer(balance);
}
```

This withdrawal function ensures that only the owner can initiate the withdrawal of MATIC from the smart contract. It provides a more controlled and secure mechanism for handling MATIC funds.

Status:

A fix is implemented, following recommendations except for the `withdrawMATIC` function. No user will have interest to send tokens wihtout calling a function, so it is better to revert in this case.

LOW-1 Simplify URI handling using ERC1155URIStorageUpgradeable

Impact: Low

Description:

The current implementation of handling URIs in the `NFT_ExportUpgradeable.sol` smart contract involves the use of a separate `baseUri` variable, and the logic is not aligned with the standard URI handling provided by OpenZeppelin's ERC1155URIStorageUpgradeable extension. It leads to useless variables and a complicated logic.

Recommendation:

1. Remove the `baseUri` variable and the `setURI` function.
2. Update the contract to inherit from ERC1155URIStorageUpgradeable.
3. Replace the current `uri` function with the standard implementation provided by the extension.

By using the ERC1155URIStorageUpgradeable extension, the contract follows a more standardized approach for handling URIs, making the code cleaner and easier to understand.

Status:

A fix is implemented, following recommendations. To allow owner to change the base URI, a new setBaseUri function should be added.

LOW-2 Compliance with ERC1155 standard recommendations Fixed

Impact: Low

Description:

The `uri` function in the smart contract should be updated to align more closely with the ERC1155 standard specifications. According to the ERC1155 standard :

> *The uri function MUST NOT be used to check for the existence of a token as it is possible for an implementation to return a valid string even if the token does not exist.*

Recommendation:

Modify the `uri` function to remove the `isExportCreated(_exportId)` modifier and ensure that it returns an empty string for non-existing tokens.

This change ensures compliance with the ERC1155 standard and provides a more accurate implementation of the `uri` function.

Status:

A fix is implemented, following recommendations.

LOW-3 Redundant check and range clarification in `updateReserveAllocatedPercentage` function Fixed

Impact: Low

Description:

The `updateReserveAllocatedPercentage` function checks for negative values of `_reserveAllocatedPercentage`, which is unnecessary since `_reserveAllocatedPercentage` is of type `uint256` and cannot be negative. Additionally, if this variable can be equal to 0 or 100, it might be clearer to only revert for values strictly greater than 100, clarifying the valid range and saving gas.

Recommendation:

Remove the check for negative values, and update the condition to only revert for values strictly greater than 100 if needed.

Status:

A fix is implemented, following recommendations.

LOW-4 Inconsistent event argument in `RewardsDistributed` event Fixed

Impact: Low

Description:

The `RewardsDistributed` event has an inconsistency in the argument name between its definition and its usage in the `distributeRewardsByBatch` function. The event is defined with `batchSize` as the argument, but in the function, it uses `endIndex`.

Recommendation:

We recommend one of the following solutions:

1. Change the event argument name from `batchSize` to `endIndex` to align with its usage in the `distributeRewardsByBatch` function.
2. Use `batchSize` instead of `endIndex` when emitting the event to match the event's definition. You should be careful for the case when batchSize is recompute when higher than the contributor number.

Status:

A fix is implemented, following recommendations.

LOW-5 Absence of _disableInitializers() invocation in upgradeable Contracts Fixed

Impact: Low

Description:

In the upgradeable contracts within the project, the `_disableInitializers()` function, designed to prevent the execution of any initialize functions in the implementation contract, is not being invoked in the constructors. Failing to call `_disableInitializers()` leaves the implementation contract vulnerable to potential attacks, as an attacker could potentially trigger uninitialized functions in the implentation contract.

Recommendation:

In each upgradeable contract, add the `_disableInitializers()` function invocation within the constructor to enforce that no initialize functions in the implementation contract can be executed. This aligns with security best practices and helps mitigate the risk of potential vulnerabilities.

Modify each contract's constructor as follows:

```
constructor() {
    _disableInitializers();
}
```

Status:

A fix is implemented, following recommendations.

LOW-6 Redundant check in donate function Fixed

Impact: Low

Description:

In the `ProjectUpgradeable.sol` contract, the `donate` function includes a redundant check that verifies if `_amount` is zero and also checks if the balance of the sender is less than `_amount`. Since the `transferFrom` function after already ensures the second condition, the sender's balance check is redundant.

Recommendation:

Replace the redundant check in the `donate` function with a simplified condition:

Replace the existing line:

```
if (_amount == 0 || tokenLAY.balanceOf(_msgSender()) < _amount) revert InvalidSentAmount();
```

With:

```
if (_amount == 0) revert InvalidSentAmount();
```

This modification simplifies the condition and maintains the intended functionality.

Status:

A fix is implemented, following recommendations.

LOW-7 ERC165 Implementation for Project Validation Fixed

Impact: Low

Description:

In the `NFT_Export.sol` contract, the `createExport` function checks that the `_projectAddress` is not equal to zero but does not validate whether it adheres to the `IProject` interface. Even if the `_projectAddress` need to be validate by the owner of the DAO contract, a miss can occur. To enhance security and prevent the accidental creation of exports for contracts that do not implement the required functions, it is recommended to use ERC165 to verify that the `_projectAddress` contract supports the `IProject` interface.

Recommendation:

Update the `createExport` function in the `NFT_Export.sol` contract as follows:

```
function createExport(
    address _projectAddress,
    string memory _uri,
    uint256 _coefficientSize,
    uint256[] calldata _datasetIds,
    uint256[] calldata _percentageWeights
) public onlyProjectOwnerByAddress(_projectAddress) whenNotPaused {
    // There should be at least one dataset
    if (_datasetIds.length == 0) revert DatasetIdListEmpty();
    // There should be as much datasets Ids as percentage weights related to them
    if (_datasetIds.length != _percentageWeights.length) revert DatasetPercentageMismatch();
    // TODO: Should we specify a limit size of array here?

    IProject project = IProject(_projectAddress);

    // Existing logic for createExport function...
    // ...

    // Validate that _projectAddress implements IProject interface
    if (!project.supportsInterface(type(IProject).interfaceId)) {
        revert InvalidProjectAddress();
    }
}
```

In the `IProject.sol` interface, add the `supportsInterface` function:

```
// IProject.sol
interface IProject {
    // Existing interface functions...

    /**
     * @dev Checks whether the contract implements a certain interface.
     * @param interfaceID The interface identifier, as specified in ERC-165.
     * @return `true` if the contract implements `interfaceID`, `false` otherwise.
     */
    function supportsInterface(bytes4 interfaceID) external view returns (bool);
}
```

In the `Project.sol` contract, implement the `supportsInterface` function following the ERC165 recommendations:

```
// Project.sol
contract Project is IProject, ERC165 {
    // Existing contract logic...

    /**
     * @dev Checks whether the contract implements a certain interface.
     * @param interfaceID The interface identifier, as specified in ERC-165.
     * @return `true` if the contract implements `interfaceID`, `false` otherwise.
     */
    function supportsInterface(bytes4 interfaceID) public view override returns (bool) {
        return interfaceID == type(IProject).interfaceId || super.supportsInterface(interfaceID);
    }

    // Existing contract logic...
    // ...
}
```

Status:

A fix is implemented, following recommendations.

LOW-8 Lack of Check for Non-Null Merkle Root Fixed

Impact: Low

Description:

In the `commitProposal` function of the DAOMaster.sol contract, there is no check to ensure that the `_merkleRoot` parameter is non-null. If `_merkleRoot` is inadvertently set to null, it may result in users being unable to retrieve their votes in the `verifyVote` function. To prevent this scenario, it is recommended to add a check at the beginning of the `commitProposal` function to ensure that `_merkleRoot` is not equal to zero.

Recommendation:

Add a check at the beginning of the `commitProposal` function to ensure that `_merkleRoot` is not null. Update the function as follows:

```
function commitProposal(
    uint256 _proposalId,
    bytes32 _merkleRoot,
    uint256 _functionIndex,
    bool _executed
) public onlyOwner whenNotPaused {
    // Ensure that _merkleRoot is not null
    require(_merkleRoot != 0, "Merkle root can't be equal to zero");

    // Existing logic for commitProposal function...
    // ...
}
```

This modification introduces a check to ensure that `_merkleRoot` is not null, preventing potential issues with users retrieving their votes in the `verifyVote` function.

Status:

A fix is implemented, following recommendations.

LOW-9 Burn functionality in NFT_DatasetUpgradeable.sol Fixed

Impact: Low

Description:

In the `NFT_DatasetUpgradeable.sol` contract, the `_burn` function is implemented to revert with the message "BurnDisabled". While this effectively disables the burn functionality, it introduces unnecessary gas consumption. Considering that the result of the function is always the same, it is recommended to make the `_burn` function pure to optimize gas usage.

Recommendation:

Refactor the `_burn` function as follows:

```
/**
 * @dev Disable burn by returning a constant value.
 */
function _burn(uint256 /*_tokenId*/) internal pure override(ERC721Upgradeable,
ERC721URIStorageUpgradeable) {
    revert BurnDisabled();
}
```

However we recommended not to block the burn function completely, as it is useful for maintaining an accurate total supply. Moreover, users can still "burn" their tokens by sending them to any random address.

Status:

A fix is implemented, following recommendations.

LOW-10 Unused _burn function override Acknowledge

Impact: Low

Description:

In the `NFT_DatasetUpgradeable.sol` contract, there exists an internal `_burn` function intended to override the default ERC721 behavior. However, this function is not called anywhere within the contract, rendering it unused and ineffective.

Recommendation:

It is recommended to either remove the unused `_burn` function or implement its usage within the contract. If there is no intention to use custom logic for burning tokens in this contract, simply removing the override would suffice. Otherwise, if a custom `_burn` function is needed, ensure it is properly integrated into the contract logic.

Option 1: Remove the unused `_burn` function:

```
// Remove the following code block
function _burn(uint256 _tokenId) internal override(ERC721Upgradeable,
ERC721URIStorageUpgradeable) {
    super._burn(_tokenId);
}
```

Option 2: Implement custom logic in the `_burn` function:

```
// Implement custom logic in the _burn function if needed
function _burn(uint256 _tokenId) internal override(ERC721Upgradeable,
ERC721URIStorageUpgradeable) {
    // Custom logic for burning tokens
    // ...

    super._burn(_tokenId);
}
```

Status:

The Lay3rs team want to make sure no token can be burn. This improvment is rejected.

INF-1 Remove unnecessary `canClaim` mapping

Impact: Informational

Description:

The `canClaim` mapping is unnecessary and consumes gas without providing additional functionality in the `distributeRewardsByBatch` function. The condition `if (claimableAmount > 0 && canClaim[contributorAddress])` ensures that if `canClaim` is true, `claimableAmount` is zero, making the check redundant.

Recommendation:

Remove the `canClaim` mapping as it is not required for the intended functionality. This change will reduce gas consumption and enhance code simplicity.

Status:

A fix is implemented, following recommendations. However, a new vulnerability is then created due to MAJ-2 fix.

INF-2 Simplify Emitted Event in `withdraw` Function Fixed

Impact: Informational

Description:

The logic in the `withdraw` function contains a redundant check for the `fullyWithdrawn` variable, resulting in two separate emit statements for the `ContributorShareUpdate` event. However, when `fullyWithdrawn` is false, the `contributors[_msgSender()].amount` is already zero, making the separation unnecessary.

Recommendation:

Simplify the emitted event in the `withdraw` function by using a single emit statement:

Replace the existing logic:

```
if (fullyWithdrawn) {
    emit ContributorShareUpdate(
        address(this),
        _msgSender(),
        0,
        totalInvestedAmount
    );
} else {
    emit ContributorShareUpdate(
        address(this),
        _msgSender(),
        contributors[_msgSender()].amount,
        totalInvestedAmount
    );
}
```

With the simplified emit statement:

```
emit ContributorShareUpdate(
    address(this),
    _msgSender(),
    contributors[_msgSender()].amount,
    totalInvestedAmount
);
```

Moreover, the `fullyWithdrawn` variable can be deleted.

Status:

A fix is implemented, following recommendations.

INF-3 Unnecessary `contractAddress` Variable from `ContributorShareUpdate` Event Fixed

Impact: Informational

Description:

The `ContributorShareUpdate` event includes a `contractAddress` variable that is always equal to `address(this)`. This variable appears to be unnecessary, as the event is emitted within the context of the smart contract itself. Removing this variable can simplify the event structure without losing any meaningful information.

Recommendation:

It is recommended to remove the `contractAddress` variable from the `ContributorShareUpdate` event.

Replace the existing event definition:

```
/**
 * @dev Event triggered at each contributor share update
 */
event ContributorShareUpdate(
    address contractAddress,
    address contributor,
    uint256 amount,
    uint256 totalInvestedAmount
);
```

With the modified event definition:

```
/**
 * @dev Event triggered at each contributor share update
 */
event ContributorShareUpdate(
    address contributor,
    uint256 amount,
    uint256 totalInvestedAmount
);
```

Of course, all the event emitters should be modified to follow this new definition.

Status:

A fix is implemented, following recommendations.

INF-4 Useless check in withdraw function Fixed

Impact: Informational

Description:

In the `withdraw` function of the `ProjectUpgradeable` smart contract, there is a useless check.

`(index <= contributorAddressList.length - 1)` should be always be true in any state of the smart contract. It can be replaced by:

1. `(index < contributorAddressList.length - 1)` to save gas during execution
2. Or delete this condition in order to reduce bytecode size.

Recommendation:

Simplify the logic for updating the `contributorAddressList` and improve gas efficiency by removing unnecessary checks. Replace this:

```
        if (totalWithdrawableLAYAmount == _amount) {
            if (index <= contributorAddressList.length - 1) {
                // In the case the index is not the last one
                contributorAddressList[index] = contributorAddressList[
                    contributorAddressList.length - 1
                ];
            }
            // We remove the last address
            contributorAddressList.pop();
```

With:

```
        if (totalWithdrawableLAYAmount == _amount) {
                contributorAddressList[index] = contributorAddressList[
                    contributorAddressList.length - 1
                ];

            // We remove the last address
            contributorAddressList.pop();
```

This modification streamlines the code, removing unnecessary conditions and optimizing gas consumption without altering the intended functionality of the `withdraw` function.

Status:

A fix is implemented, following recommendations.

INF-5 Variable shadowing Fixed

Impact: Informational

Description:

In the `createExport` function, the parameter `_uri` is named the same as the `ERC1155` variable `_uri`. While this does not impact the functionality of the smart contract, it can lead to confusion and decrease code readability. To enhance clarity, it is recommended to rename the parameter to avoid shadowing the existing `_uri` variable.

Recommendation:

Refactor the `createExport` function to rename the `_uri` parameter:

```
function createExport(
    address _projectAddress,
    string memory _exportUri, // Rename the parameter for clarity
    uint256 _coefficientSize,
    uint256[] calldata _datasetIds,
    uint256[] calldata _percentageWeights
) public onlyProjectOwnerByAddress(_projectAddress) whenNotPaused {
    // Existing function logic...
}
```

Status:

A fix is implemented, following recommendations.

INF-6 Missing check for _uri parameter in createExport function Fixed

Impact: Informational

Description:

In the `createExport` function of the smart contract, there is no validation check on the `_uri` parameter to ensure it is not null. This omission may lead to undesired behavior.

Recommendation:

Add a validation check for the `_uri` parameter to ensure it is not null in the `createExport` function:

```
require(bytes(_uri).length != 0, "URI cannot be empty");
```

Status:

A fix is implemented, following recommendations with a custom error.

INF-7 Redundant check in mint function Fixed

Impact: Informational

Description:

In the `mint` function of the smart contract, the `hasEnoughLAY` modifier is applied, checking if the caller has enough LAY tokens. However, this check is already performed within the `transferFrom` function of the LAY token contract during the token transfer. The redundant use of the `hasEnoughLAY` modifier in this context does not provide additional security or functionality.

Recommendation:

Remove the redundant `hasEnoughLAY` modifier from the `mint` function.

Remove the following line:

```
hasEnoughLAY(_layAmount) // TODO: Could be useless to check it here
```

The modified function will be:

```
function mint(
    uint256 _exportId,
    address _to,
    bytes memory _data,
    uint256 _layAmount
)
    public
    // nonReentrant // TODO: check if we face any issues with these multiples transfers -> MAJ:
Yes apparently
    whenNotPaused
    isExportCreated(_exportId)
    // Remove the redundant hasEnoughLAY modifier
    isLAYAmountValid(_exportId, _layAmount)
{
    // First we check if the recipient address is not address 0
    if (_to == address(0)) revert InvalidRecipientAddress();
    // Transfer LAY tokens from the caller to the contract
    (bool sent) = tokenLAY.transferFrom(_msgSender(), address(this), _layAmount);
    if (!sent) revert LAYTransferFailed();

    // Distributes rewards to all stakeholders
    _distributeLAYRewards(_exportId, _layAmount);

    // Mint the token
    _mint(_to, _exportId, 1, _data); // Tokens are minted one by one
}
```

In the same time the `hasEnoughLAY` modifier can be deleted since it is only used here.

Status:

A fix is implemented, following recommendations.

INF-8 Restricting safeBatchTransferFrom function to pure Fixed

Impact: Informational

Description:

In the `NFT_ExportUpgradeable.sol` contract, the `safeBatchTransferFrom` function is currently declared as public. Similar to the previous recommendation, since it does not interact with the contract state and only reverts with a custom error, it can be restricted to the `pure` function type to save gas.

Recommendation:

Update the `safeBatchTransferFrom` function in the `NFT_ExportUpgradeable.sol` contract to be restricted to the `pure` function type:

Replace the existing function declaration:

```
function safeBatchTransferFrom(
    address /*_from*/,
    address /*_to*/,
    uint256[] memory /*_ids*/,
    uint256[] memory /*_values*/,
    bytes memory /*_data*/
) public override(ERC1155Upgradeable) {
    revert SafeBatchTransferDisabled();
}
```

With:

```
function safeBatchTransferFrom(
    address /*_from*/,
    address /*_to*/,
    uint256[] memory /*_ids*/,
    uint256[] memory /*_values*/,
    bytes memory /*_data*/
) public pure override(ERC1155Upgradeable) {
    revert SafeBatchTransferDisabled();
}
```

Status:

A fix is implemented, following recommendations.

INF-9 Restricting safeTransferFrom function to pure Fixed

Impact: Informational

Description:

In the `NFT_ExportUpgradeable.sol` contract, the `safeTransferFrom` function is currently declared as public. Since it does not interact with the contract state and only reverts with a custom error, it can be restricted to the `pure` function type to save gas.

Recommendation:

Update the `safeTransferFrom` function in the `NFT_ExportUpgradeable.sol` contract to be restricted to the `pure` function type:

Replace the existing function declaration:

```
function safeTransferFrom(
    address /*_from*/,
    address /*_to*/,
    uint256 /*_id*/,
    uint256 /*_amount*/,
    bytes memory /*_data*/
) public override(ERC1155Upgradeable) {
    revert SafeTransferDisabled();
}
```

With:

```
function safeTransferFrom(
    address /*_from*/,
    address /*_to*/,
    uint256 /*_id*/,
    uint256 /*_amount*/,
    bytes memory /*_data*/
) public pure override(ERC1155Upgradeable) {
    revert SafeTransferDisabled();
}
```

Status:

A fix is implemented, following recommendations.

Black paper

INF-10 Smart contracts interface inheritance Fixed

Impact: Informational

Description:

In the `Project.sol` smart contract, it does not explicitly inherit from the `IProject` interface. Inheriting from the interface ensures that the contract implements all the required functions, reducing the risk of missing or mismatched function signatures.

There is the same issue for NFT_DatasetUpgradeable and NFT_ExportUpgradeable.

Recommendation:

It is recommended to explicitly inherit from the `IProject` interface in the `Project.sol` smart contract. This helps ensure that the contract adheres to the defined interface, preventing the accidental omission of required functions and enhancing consistency.

```
// Project.sol
contract Project is IProject {
    // Existing contract logic...
    // ...
}
```

By explicitly inheriting from the interface, you confirm that all the required functions are implemented, improving the overall security and consistency of the smart contract.

We recommend to do the same for NFT_DatasetUpgradeable and NFT_ExportUpgradeable.

Status:

A fix is implemented, following recommendations.

INF-11 Redundant Fallback Function Reversion Fixed

Impact: Informational

Description:

In the contract's code, the following fallback function is implemented:

```
/**
 * @dev Fallback that reverts any method not referenced in the contract
 */
fallback() external payable {
    revert FunctionNotImplemented();
}
```

This fallback function is redundant because, by default, if there is no fallback function implemented in the contract, any call to an undefined or non-existent function will automatically fail and revert. Therefore, implementing an additional fallback function to explicitly revert with `FunctionNotImplemented()` is unnecessary and does not provide any additional security.

Recommendation:

We recommend to remove the fallback function from the contract.

Status:

MED-6 fix also deletes this part of the code.

INF-12 Useless LAY Token Variable in DAOMaster.sol Fixed

Impact: Informational

Description:

In the `DAOMaster.sol` contract, a variable named `tokenLAY` is declared to store the LAY token smart contract address. However, this variable is only used in the constructor to set its value and is not utilized elsewhere in the contract. This redundancy introduces unnecessary complexity to the code and consumes extra storage.

Recommendation:

It is recommended to remove the `tokenLAY` variable from the contract. This action will enhance code readability and optimize gas consumption during deployment. The modified constructor should look like this:

```
constructor() {}
```

Status:

A fix is implemented, following recommendations.

INF-13 Unnecessary nonReentrant Modifier in batchTransfer Function Fixed

Impact: Informational

Description:

In the `batchTransfer` function within `Lay.sol`, the `nonReentrant` modifier is applied. However, upon closer inspection, it appears to be unnecessary. The function does not involve any external calls, and as a result, there is no risk of reentrancy attacks in this context.

Recommendation:

It is recommended to remove the `nonReentrant` modifier from the `batchTransfer` function for the sake of code readability and to save gas costs. The modifier does not provide any additional security benefits in this specific function, given that there are no external calls that could potentially introduce reentrancy vulnerabilities.

The modified function without the `nonReentrant` modifier would look like this:

```
/**
 * @dev Execute a batch of different transfers to a list of recipients' addresses
 * @notice Arrays of recipients and amounts must have the same size
 * @notice There should be at least one element in the array of recipients
 * @notice The sender must hold at least the sum of the array of amounts
 * @param _recipients {{address[]}} - List of recipients we should distribute the amount of LAY
 * @param _amounts {{uint256[]}} - List of amounts of LAY to distribute to recipients
 */
function batchTransfer(
    address[] calldata _recipients,
    uint256[] calldata _amounts
) public whenNotPaused {
    // Remaining function code...
}
```

Status:

A fix is implemented, following recommendations.