QuillAudits

Audit Report October, 2023

For

Liveat verse

Table of Content

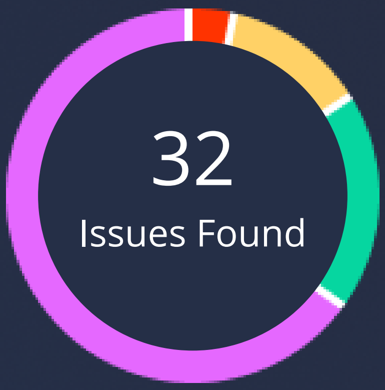
Table of Content

Table of Content

Executive Summary

Project Name	Livaatverse
Overview	LivaatVerse is an innovative 3D ecosystem that seamlessly merges the virtual and physical worlds, providing a platform where users can construct, interact, and transact in expansive interconnected virtual spaces. By transforming traditional 2D live-streaming, it offers an immersive 3D experience that effectively bridges the gap between the digital and physical realms. Catering to a wide range of interests, LivaatVerse features specialized areas such as business expos, universities, NFT galleries, and even virtual tourism. Designed with inclusivity in mind, the platform ensures a low entry barrier while offering cross-platform support, ensuring global accessibility and real-time interactions. A key strength of LivaatVerse lies in its emphasis on live shows, social meetups, and content creation. This vast virtual universe is not only a space for exploration but also a hub for innovation and community-building.
Timeline	19th September 2023 - 13th October 2023
Updated Code Received	10th October 2023 and on 22nd October 2023
Second Review	11th October 2023 - 27th October 2023
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	The scope of this audit was to analyze the Livaatverse codebase for quality, security, and correctness. *https://github.com/Livaat-Organization/blockchain-contract/tree/main/src*
Fixed In	d7a785cbcfab64e8d62690d1c40fcf9dde579634

Number of Security Issues per Severity



🟥 High 🟨 Medium

🟩 Low 🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	1	0	3
Partially Resolved Issues	0	1	0	0
Resolved Issues	1	2	6	18

Checked Vulnerabilities

- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ DoS with Block Gas Limit
- ✓ Transaction-Ordering Dependence
- ✓ Use of tx.origin
- ✓ Exception disorder
- ✓ Gasless send
- ✓ Balance equality
- ✓ Byte array
- ✓ Transfer forwards all gas
- ✓ ERC20 API violation
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Redundant fallback function
- ✓ Send instead of transfer
- ✓ Style guide violation
- ✓ Unchecked external call
- ✓ Unchecked math
- ✓ Unsafe type inference
- ✓ Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Hardhat, Foundry.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

High Severity Issues

1. Potential locking of funds due to non standard ERC20 transfers

Path

LivaatOwnershipDeed.sol

Description

The LivaatOwnershipDeed.sol contract contains methods designed to transfer ERC20 tokens. Notably, it assumes that all ERC20 tokens will adhere to OpenZeppelin's IERC20 interface, expecting a boolean return value for the transfer and transferFrom methods. However, some tokens, notably USDT (Tether), deviate from this standard behavior and do not return a boolean value. Given the documented possibility of using USDT as one of the default-supported currencies during deployment, this discrepancy presents a significant vulnerability. If USDT is selected as cashOutCurrency, any transaction invoking the cashOut function would fail at the required statement, locking funds and prohibiting withdrawals.

Impact

If USDT is employed as the cashOutCurrency, users attempting to execute the cashOut function will encounter a transaction failure. This will not only disrupt platform operations but also lead to locked funds, which can damage user trust and the platform's reputation.

Proof of Concept

```
    function cashOut(uint256 amount) public isPaused isCashOutEnabled {
        ...
        require(LDC.transferFrom(msg.sender, address(this), fee));
        ...
        require(cashOutCurrency.transfer(msg.sender, amountOut));
    }
```

From the deployment documentation

```
    &quot;Deploy LDCToken contract
    Parameters:
    currencies an array of default supported currencies(stablecoins like BUSD, USDT...)
    ...
    currencies an array of default supported currencies(stablecoins like BUSD, USDT...)&quot;
```

Recommendation

To mitigate this critical vulnerability:

Utilize SafeERC20: Leverage OpenZeppelin's SafeERC20 library which offers a set of functions designed to handle tokens that don't return a boolean value. Replace the current transfer and transferFrom methods with safeTransfer and safeTransferFrom, respectively.

Validation During Setup: Implement a check during contract deployment or initialization to

1. Potential locking of funds due to non standard ERC20 transfers

validate that all tokens in the currencies array adhere to the expected ERC20 behavior. Disallow the addition of non-compliant tokens.

Status

Fixed

Medium Severity Issues

1. Unsafe ERC-20 Operations

Path

LivaatOwnershipDeed.sol

Description

The contract LivaatOwnershipDeed.sol utilizes functions to transfer ERC20 tokens. This contract imports the IERC20 interface from OpenZeppelin, which assumes the standard behavior of the transfer and transferFrom methods, i.e., returning a boolean value upon execution. This is not universally adhered to among all ERC20 tokens; some tokens like USDT do not return a boolean. Attempting to transfer these non-standard tokens using these functions can cause the contract to revert, inhibiting the transaction's execution.

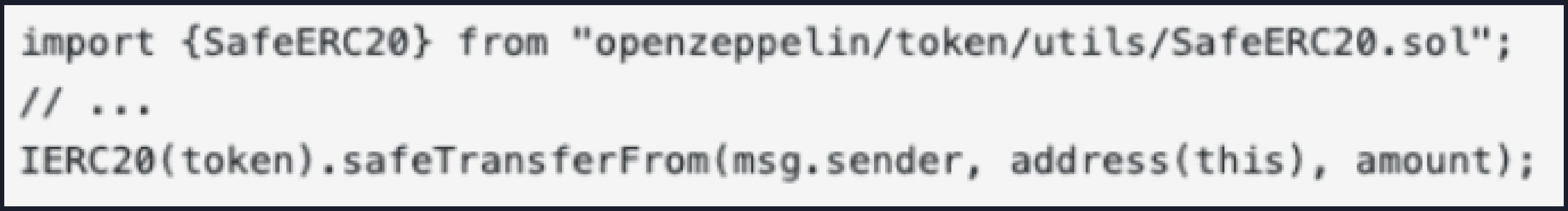
Code

```
withdraw function:
function withdraw(IERC20 currency) external onlyOwner {
    require(_currencies.contains(address(currency)), "invalid currency");
    uint256 balance = currency.balanceOf(address(this));
    require(balance > 0, "insuffecient balance");
    currency.transfer(owner(), balance);
}

cashOut function:
function cashOut(uint256 amount) public isPaused isCashOutEnabled {
    ...
    require(LDC.transferFrom(msg.sender, address(this), fee));
    ...
    require(cashOutCurrency.transfer(msg.sender, amountOut));
}



342: require(LDC.transfer(msg.sender, claimableShare));

522: require(LDC.transfer(owner(), remainingShares)); //donation, award, pinee
r winner, livaatverse
```

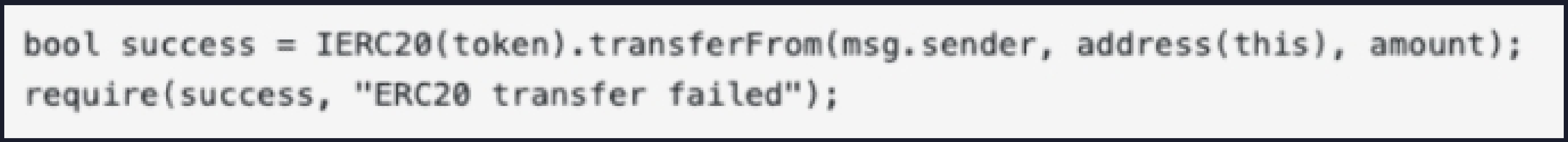
1. Unsafe ERC-20 Operations

Recommendation

To avoid potential issues with non-standard ERC20 tokens, it is advisable to leverage OpenZeppelin's SafeERC20 wrapper along with the safeTransfer and safeTransferFrom methods to execute the token transfers. Or use require on the response from **transfer** and **transferFrom()**.



```
import {SafeERC20} from "openzeppelin/token/utils/SafeERC20.sol";
// ...
IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
```



```
bool success = IERC20(token).transferFrom(msg.sender, address(this), amount);
require(success, "ERC20 transfer failed");
```

Status

Fixed

2. Centralization Risk Due to Excessive Use of onlyOwner Modifier

Description

The contracts in the LivaatVerse protocol have numerous functions protected by the onlyOwner modifier. This level of centralization places a significant amount of trust and power in the owner's hands. If the owner's account becomes compromised, the attacker could:

- Change the base URI, potentially redirecting users or applications to malicious endpoints.
- Add or remove supported currencies, affecting the protocol's interoperability.
- Withdraw funds or assets from the contract.
- Adjust critical parameters like fees, which can financially harm users.
- Toggle functionalities like pausing the protocol or enabling/disabling features.
- Potentially mint or burn tokens, affecting the tokenomics.
- And many more adverse actions based on the function's nature.

Impact

If the owner's account is compromised, users could lose funds, be exposed to malicious activities, or experience a loss of trust in the platform. Additionally, the platform's reputation could suffer, leading to a potential decrease in usage and trust.

2. Centralization Risk Due to Excessive Use of onlyOwner Modifier

Recommendation

- Decentralized Control: Consider transitioning to a more decentralized governance mechanism, such as a DAO (Decentralized Autonomous Organization) or a multisig wallet. This will distribute control among multiple parties, reducing the risk associated with a single point of failure.
- Time Locks: Implement time locks for critical functions. This gives users a window to observe and react to any suspicious or unintended changes.
- Function Limitation: Evaluate the necessity of each onlyOwner function. If certain functions don't need to be changed frequently, consider removing them or setting them during contract deployment.
- Transparency: Any changes made by the owner should be transparently communicated to the users in a timely manner.

Status

Acknowledged

3. Potential Scenario For Not Able To Claim Land NFTs

Path

LandAndFieldManager.sol

Function

addReservedLands

Description

The contact LandAndFieldManager.sol uses addReservedLands function to reserve lands by Admin. This function takes a lands array and based on its length , it reserves that number of lands. There is no check on the number of lands which an owner can reserve. There is a check on whether the land's row and ray is valid but not on how much an owner can reserve.

When a b2b user comes to claim their land and field **_b2bToClaimOrBuy** is called.

3. Potential Scenario For Not Able To Claim Land NFTs

```
function _b2bToClaimOrBuy(Constants.RingRootTypes ringRoot) internal view returns (uint256 valu
e) {
require(ringRoot >= Constants.RingRootTypes.VIP && ringRoot <= Constants.RingRootTypes.Perceive
r, "invalid ring root");

if(ringRoot == Constants.RingRootTypes.VIP) {
// 9500 - 1000 - 5000
value = Constants.VIP_ROOT_RING_LANDS_COUNT - ringLandMinted[Constants.RingTypes.VIP].reservedT
oMint - Constants.ANGEL_TIER_COUNT;
} else if (ringRoot == Constants.RingRootTypes.Industrial) {
value = Constants.INDUSTRIAL_ROOT_RING_LANDS_COUNT - ringLandMinted[Constants.RingTypes.Industr
ialOne].reservedToMint - ringLandMinted[Constants.RingTypes.IndustrialTwo].reservedToMint - rin
gLandMinted[Constants.RingTypes.IndustrialThree].reservedToMint - Constants.RARE_TIER_COUNT;
} else if (ringRoot == Constants.RingRootTypes.Perceiver) {
value = Constants.PERCEIVER_ROOT_RING_LANDS_COUNT - ringLandMinted[Constants.RingTypes.Perceive
r].reservedToMint - Constants.PERCEIVER_TIER_COUNT;
}
}
```

Lets take the case of 'VIP' , if the reservedToMint > 4500 then the user will not be able to claim the land since a negative number will be returned. Same is the case with other types of Lands i.e Industrial and Perceiver. The issue can arise due to the fact that there is no check for how much an admin can reserve.

Impact

If admin reserve land without keeping in consideration the total lands then the B2B users wont be able to claim lands.

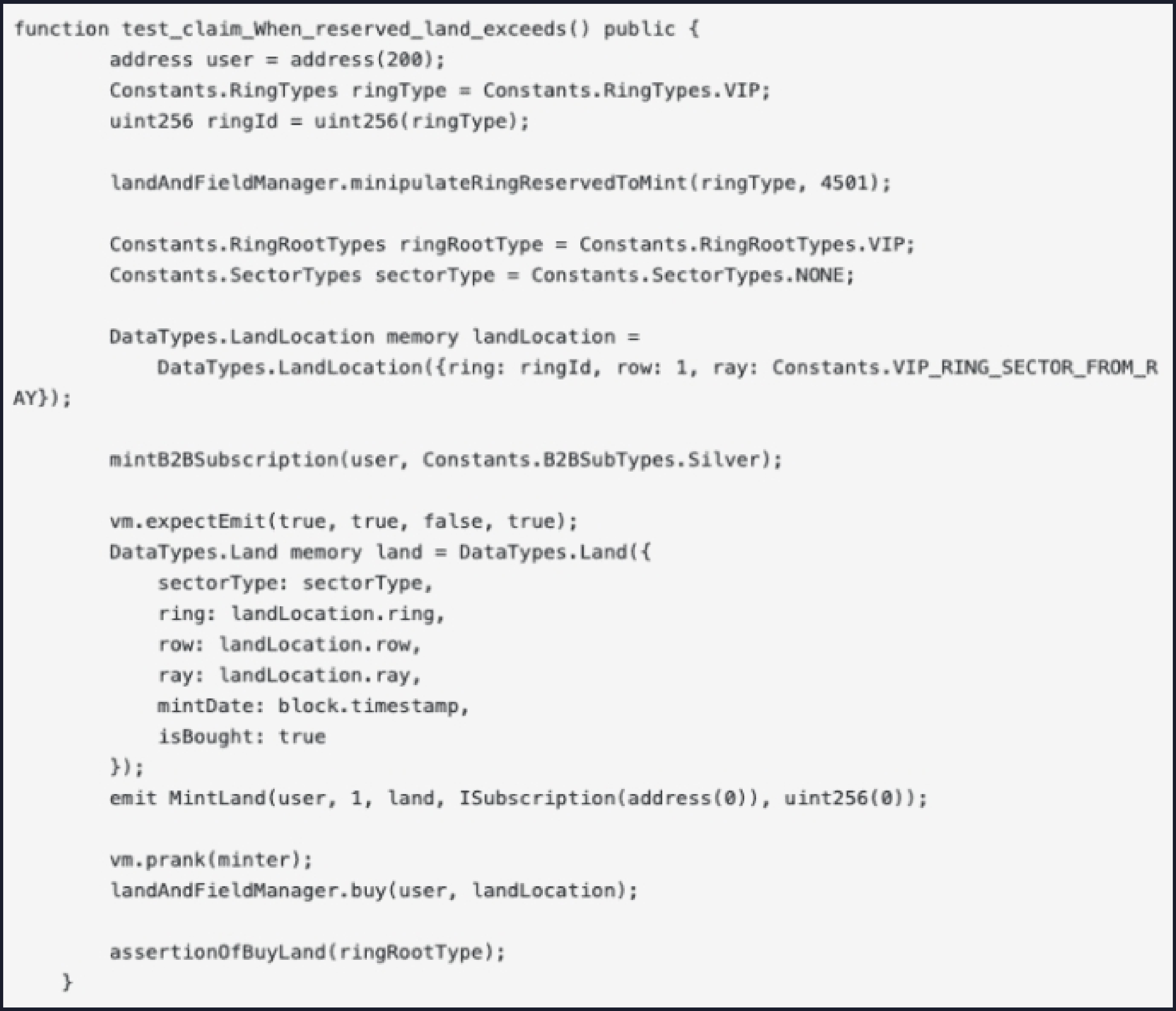
Proof of Concept

Following is a Foundry function , when the reserve land in

landAndFieldManager.minipulateRingReservedToMint(ringType, Reserve_land_amount)

is kept below 4500 , it will pass , for Reserve_land_amount > 4500, it will fail.

3. Potential Scenario For Not Able To Claim Land NFTs



```
function test_claim_When_reserved_land_exceeds() public {
        address user = address(200);
        Constants.RingTypes ringType = Constants.RingTypes.VIP;
        uint256 ringId = uint256(ringType);

        landAndFieldManager.minipulateRingReservedToMint(ringType, 4501);

        Constants.RingRootTypes ringRootType = Constants.RingRootTypes.VIP;
        Constants.SectorTypes sectorType = Constants.SectorTypes.NONE;

        DataTypes.LandLocation memory landLocation =
            DataTypes.LandLocation({ring: ringId, row: 1, ray: Constants.VIP_RING_SECTOR_FROM_R
AY});

        mintB2BSubscription(user, Constants.B2BSubTypes.Silver);

        vm.expectEmit(true, true, false, true);
        DataTypes.Land memory land = DataTypes.Land({
            sectorType: sectorType,
            ring: landLocation.ring,
            row: landLocation.row,
            ray: landLocation.ray,
            mintDate: block.timestamp,
            isBought: true
        });
        emit MintLand(user, 1, land, ISubscription(address(0)), uint256(0));

        vm.prank(minter);
        landAndFieldManager.buy(user, landLocation);

        assertionOfBuyLand(ringRootType);
    }
```

Recommendation

It is recommended that for each land a specific amount of land should be reserved and it should be documented in the tokenomics of Livaat so that there is transparency. Plus while reserving there should be a check for max number of reserve land so that by any chance an admin doesn't reserve more land. It should be kept into consideration that this is an extreme edge case but can happen due to any unknown reasons.

Status

Fixed

4. Inaccurate Share calculation

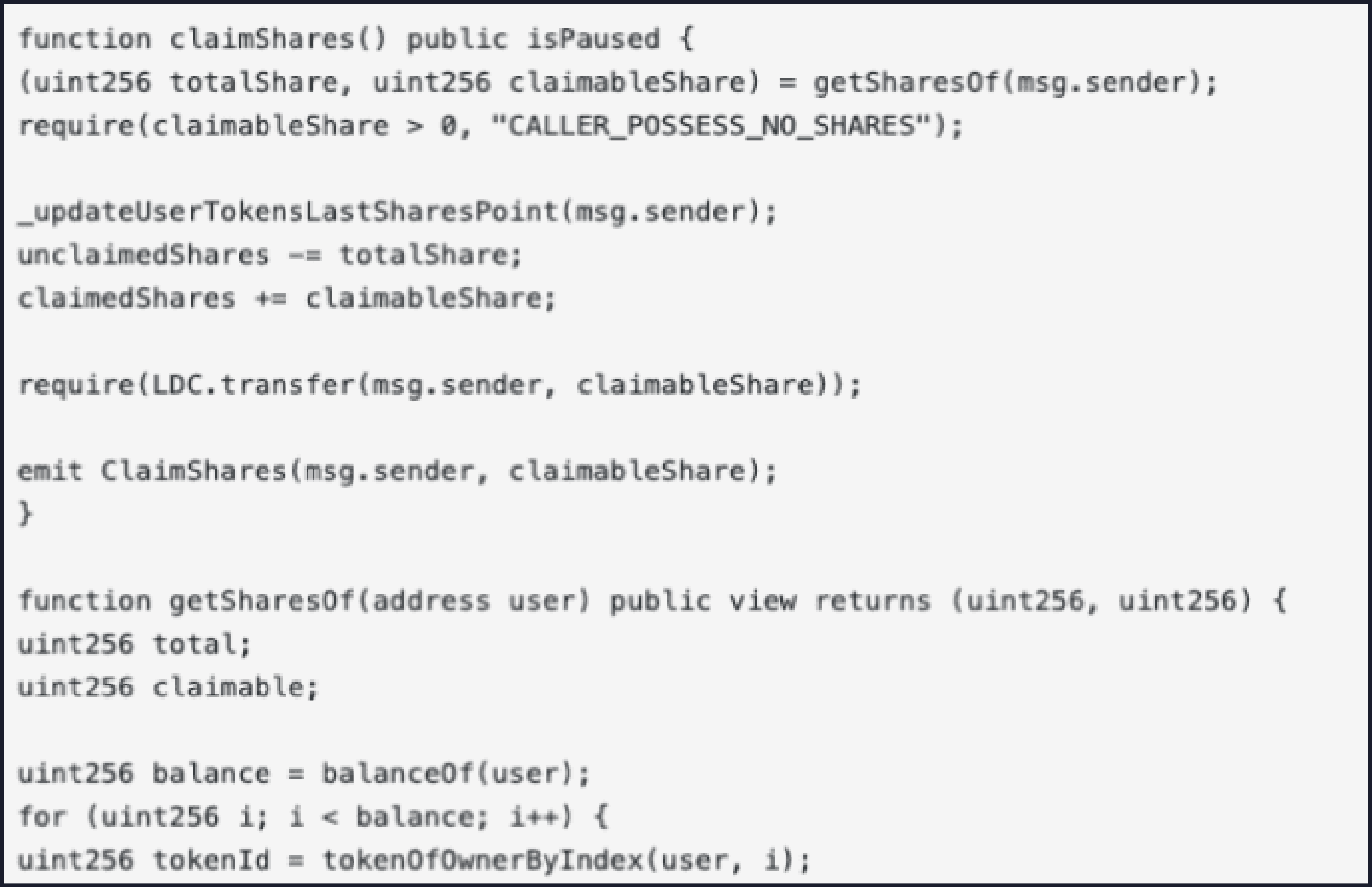
Path

LivaatOwnershipDeed.sol

Description

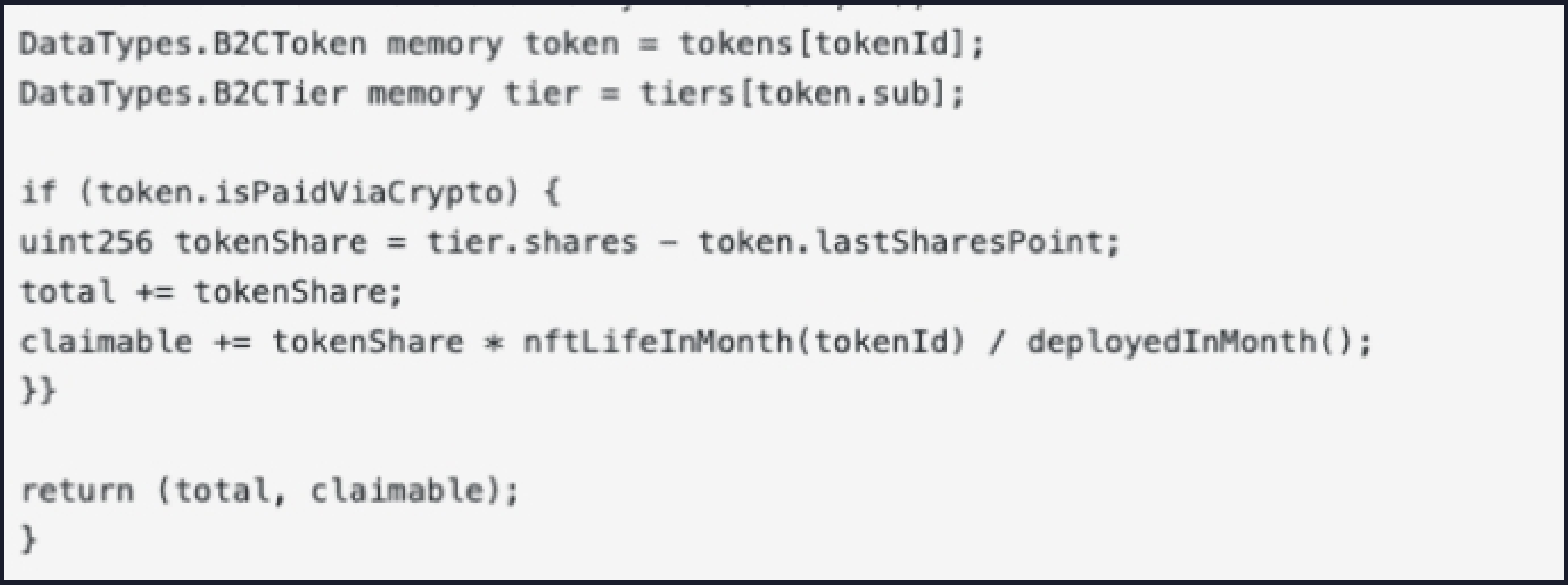
It was identified that when a user buys NFT via web2 , it’s sent as a gift to the user (as per the discussion with the developers). In that case the isPaidViaCrypto field is not set to true hence when it’s time to claim the shares the calculation for shares won’t be accurate. And can result in potential loss to User.

Proof of Concept



```
function claimShares() public isPaused {
(uint256 totalShare, uint256 claimableShare) = getSharesOf(msg.sender);
require(claimableShare > 0, "CALLER_POSSESS_NO_SHARES");

_updateUserTokensLastSharesPoint(msg.sender);
unclaimedShares -= totalShare;
claimedShares += claimableShare;

require(LDC.transfer(msg.sender, claimableShare));

emit ClaimShares(msg.sender, claimableShare);
}

function getSharesOf(address user) public view returns (uint256, uint256) {
uint256 total;
uint256 claimable;

uint256 balance = balanceOf(user);
for (uint256 i; i < balance; i++) {
uint256 tokenId = tokenOfOwnerByIndex(user, i);
```

4. Inaccurate Share calculation



```
DataTypes.B2CToken memory token = tokens[tokenId];
DataTypes.B2CTier memory tier = tiers[token.sub];

if (token.isPaidViaCrypto) {
uint256 tokenShare = tier.shares - token.lastSharesPoint;
total += tokenShare;
claimable += tokenShare * nftLifeInMonth(tokenId) / deployedInMonth();
}}

return (total, claimable);
}
```

When claimshare() is called it further makes a call to getShares(msg.sender) , if the isPaidViaCrypto is true then in that case its share is calculated.

Recommendation

If there is a script backend maintained for managing users who bought via web2 then it should be audited too. In current implementation according to our findings , users buying via web2 will be at potential loss once shares claiming is turned ON.

Status

Partially Resolved

Low Severity Issues

1. Conversion Fee Can Be Set To 100%

Path

LivaatOwnershipDeed.sol

Function

setConvertFee

Description

The setConvertFee function in the protocol allows the owner to set the conversion fee to any value between 0% and 100% inclusive. If set to 100%, users who attempt to convert their tokens will be charged the entire amount as a fee, rendering the conversion operation meaningless.

```
function setConvertFee(uint256 newFee) external onlyOwner {
        require(newFee <= 10000, "Only between 0% and 100%");
        convertFee = newFee;
    }
```

Impact

Adjust the requirement check in the setConvertFee function to enforce a maximum fee that is strictly less than 100%. For example:

```
require(newFee < 10000, "Fee should be less than 100%");
```

Clearly document in user-facing materials (e.g., a frontend interface or user documentation) the fee structure and any changes made to it.

Status

Fixed

Auditor's Response

Although 100% fee can't be set but 99% can be, if this is the intended functionality of the protocol then it should be specified in the documentation, otherwise a reasonable amount of 1-10% should be set here.

2. Owner Can Renounce Ownership, Potentially Locking Important Functions

Path

LandandFieldManager.sol, LivaatOwnershipDeed.sol, LivaatPartnershipDeed.sol

Function

mintReserved(address, DataTypes.LandLocation[ ]),
addReservedLands(DataTypes.LandLocation[ ]), setMinter(address)

setB2BSubContract(address), declareShares(uint256), addCurrency(address),
removeCurrency(address), withdraw(IERC20), setCashOutCurrency(IERC20),
setBaseURI(string)

setBaseURI(string), togglePause(), setAuthorizedMinter(address), getAuthorizedMinter(),
setKYCApproval(address), getKYCApproval(), setUSDReceiver(address)

Description

Several contracts in the project inherit from OpenZeppelin's Ownable contract. The Ownable contract provides a function, renounceOwnership(), which allows the current owner to relinquish control over the contract. If the owner renounces ownership, certain functions that are restricted to the owner would be rendered non-callable, potentially resulting in a loss of control and functionality.

Impact

If the owner renounces ownership, the above functions will become non-callable. This might lead to loss of control, inability to manage lands and partnerships, change configurations, add/remove supported currencies, withdraw funds, or make other critical modifications.

Recommendation

If the renouncement of ownership is not intended, it's recommended to either:

Override the renounceOwnership() function to revert or make it non-functional.

```
function renounceOwnership() public override onlyOwner {
    revert("Renouncing ownership is disabled");
}
```

Clearly mention in the contract and project documentation that renouncing ownership is possible, and detail the consequences of such an action.

2. Owner Can Renounce Ownership, Potentially Locking Important Functions

Ensure to review all contracts for the potential ramifications of renouncing ownership and apply necessary fixes or documentation clarifications as deemed appropriate.

Status

Fixed

3. Missing two-step transfer ownership Pattern

Path

LandAndFieldManager.sol, LivaatOwnershipDeed.sol, LivaatPartnershipDeed.sol, LandNFT.sol, LDCToken.sol

Description

The below mentioned contracts use OpenZeppelin's Ownable contract, which provides a single-step mechanism to transfer ownership. The new owner isn't required to confirm their new ownership status. If a mistake occurs during transfer (e.g., a wrong address is provided, or the renounceOwnership function is mistakenly called), it might lead to the contract becoming non-functional if essential functionalities are restricted to the owner.

Impact

Mistakenly transferring ownership to an incorrect or malicious address can lead to potential manipulation of the contract, or the contract might become completely non-functional if the new owner doesn't cooperate. It can also lead to potential financial loss if the contract holds any funds or valuable assets.

Recommendation

Implement a two-step transfer ownership mechanism. This would entail:

- A function for the current owner to propose a new owner.
- A function for the proposed new owner to confirm and finalize the ownership transfer.
- Follow the pattern from OpenZeppelin's **Ownable2Step**

This two-step process ensures that the new owner is aware and willing to accept the ownership, reducing the risk of accidental ownership transfers.

Status

Fixed

4. Custom Pausing Mechanism

Path

LivaatOwnershipDeed.sol, LivaatPartnershipDeed.sol

Description

The contracts LivaatOwnershipDeed.sol and LivaatPartnershipDeed.sol implement their own pausing mechanisms rather than utilizing established libraries such as OpenZeppelin's Pausable contract. This custom implementation has been identified to have several issues that could lead to sub-optimal performance or potential vulnerabilities.

Detailed Findings

Visibility of State Variable: The paused state variable's visibility is set to public. This exposes the internal state variable to external views.
Recommendation: Make the paused state variable private and provide a view function, if necessary, to get its status.Naming

Convention: The isPaused modifier's naming is counter-intuitive. Typically, isPaused would indicate checking if the contract is paused, but it checks the opposite.
Recommendation: Rename the modifier to notPaused or whenNotPaused for clarity.

Lack of Events: The function that toggles the pausing (togglePause) does not emit any events to log the change in the contract's state.
Recommendation: Emit events like Paused and Unpaused whenever the state changes to enhance transparency and to aid DApps or off-chain services.

Example

```
event Paused();
event Unpaused();
function togglePause() external onlyOwner {
    paused = !paused;
    if (paused) {
        emit Paused();
    } else {
        emit Unpaused();
    }
}
```

Recommendation

For better optimization and assurance of security, consider using OpenZeppelin's **Pausable** library for both contracts. OpenZeppelin contracts are widely used in the Ethereum

4. Custom Pausing Mechanism

developer community, have been audited multiple times, and are optimized for gas efficiency and security.

By leveraging OpenZeppelin's Pausable library, the contracts would inherit a well-tested pausing mechanism, potentially reducing the risk of errors and vulnerabilities while ensuring efficient operations.

Status

Fixed

5. Potential Locking of Funds Due to Currency Removal

Path

LivaatOwnershipDeed.sol

Description

The contract provides a mechanism for users to cash out their LDC tokens in exchange for a stablecoin (cashOut function). When a user executes this function, a fee is deducted and transferred to the contract. However, the owner has the ability to remove a currency from the list of approved currencies (removeCurrency function). If a currency with a balance inside the contract is removed from the approved list, the funds of that currency become irretrievable since the withdraw function checks for the currency's presence in the approved list before allowing a withdrawal.

Impact

If the owner removes a currency that still holds a balance within the contract, those funds will be locked indefinitely. Users who cashed out and paid fees in that currency would effectively lose access to those funds, which can undermine trust in the platform and lead to financial loss.

Recommendation

To prevent potential loss of funds, consider the following changes:

Modify the withdraw function to allow the owner to retrieve any currency without checking its presence in the approved list. This ensures that even if a currency is removed, any remaining funds can still be retrieved.

Status

Fixed

6. Missing Check on Minting Reserve Land

Path

LandAndFieldManager.sol

Description

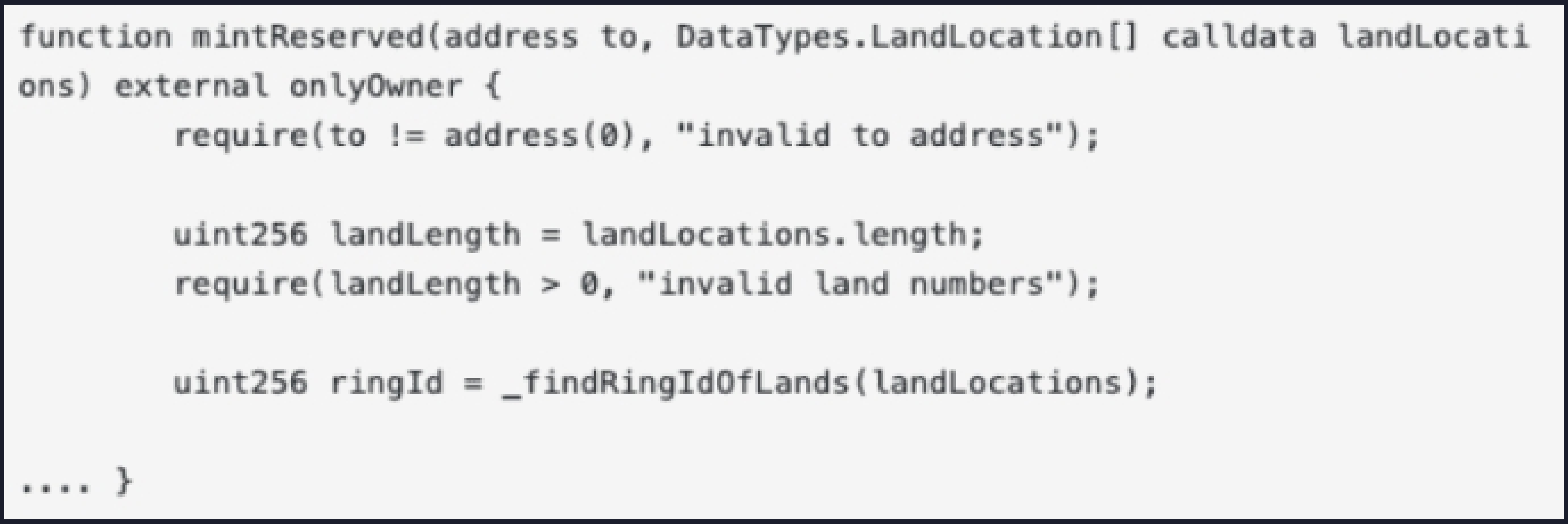
It was identified that when a reserve land is minted by the owner there is no check applied to check whether the **to** address is an owner of B2B or B2C NFT Owner.

Impact

To be a participant of Livaatverse owning B2B or B2C NFT is mandatory. This invariant can be bypassed if the check is missing from the mintReserved function.

Proof of Concept

There is check for a zero address but missing if the **to** address is an owner of B2C or B2B NFT.



```
function mintReserved(address to, DataTypes.LandLocation[] calldata landLocati
ons) external onlyOwner {
        require(to != address(0), "invalid to address");

        uint256 landLength = landLocations.length;
        require(landLength > 0, "invalid land numbers");

        uint256 ringId = _findRingIdOfLands(landLocations);

.... }
```

Recommendation

It is recommended to add the check so the invariant holds properly.

Status

Fixed

Informational Issues

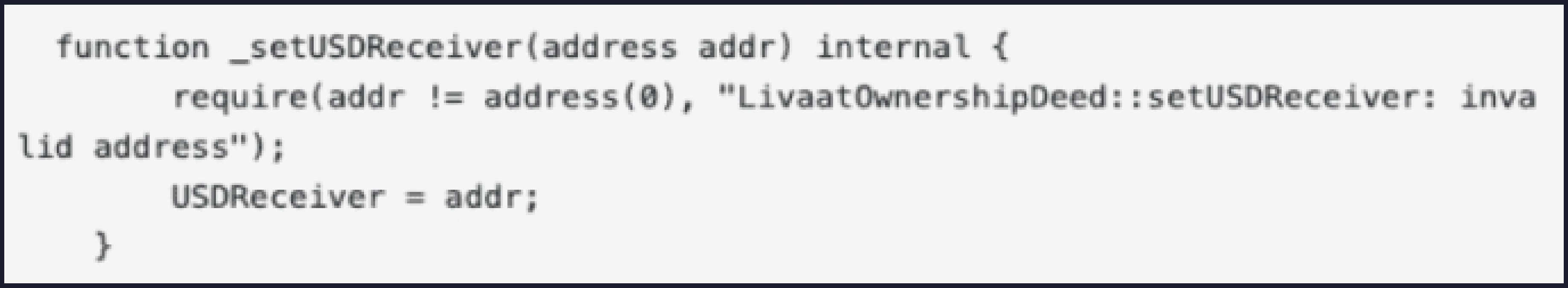
1. Use Custom Errors

Path

All Contracts

Description

Custom errors from 0.8.4, leads to cheaper deploy- and run-time costs. Note: the run time cost is only relevant when the revert condition is met. In short, replace revert strings with custom errors.



```
  function _setUSDReceiver(address addr) internal {
        require(addr != address(0), "LivaatOwnershipDeed::setUSDReceiver: inva
lid address");
        USDReceiver = addr;
    }
```

Recommendation

To save gas custom errors should be used.

Status

Fixed

2. Cache array length outside of loop

Description

When iterating through an array in Solidity, accessing its length at each loop iteration incurs a gas cost. Specifically, it takes 6 gas units—3 for mload and 3 for placing the memory_offset on the stack. To optimize, it's advisable to cache the array length in a variable before entering the loop.

Proof of Concept

Here, the compiler constantly checks the array's length for each iteration. Depending on the type of the array:

- For storage arrays, it adds an sload operation, adding 100 extra gas (as per EIP-2929) for every iteration after the first.
- For memory arrays, an additional mload operation costs 3 gas for each iteration after the first.

2. Cache array length outside of loop

- For calldata arrays, it's an extra calldataload operation costing 3 gas per subsequent iteration.

Recommendation

To mitigate this, the array's length can be cached:

```
uint256 length = currencies.length;
for (uint256 i = 0; i < length; i++) {
    addCurrency(currencies[i]);
}
```

With this approach, sload, mload, or calldataload operations are executed just once, later using a cheaper dupN instruction for subsequent iterations. While mload, calldataload, and dupN share similar gas costs, the first two operations need an additional dupN to position the offset on the stack, resulting in an extra gas cost of 3 units.

Status

Fixed

3. Use of unchecked for Loop Increment Optimization

Description

In Solidity 0.8+, the default overflow checks on unsigned integers can add unnecessary gas overhead, especially when used inside loops. For loop indices that purely traverse fixed-sized data structures without modifying the index value within the loop, the overflow checks are redundant. We can safely employ unchecked blocks to optimize gas usage.

Proof of Concept

Example:

```
Traditional loop with checked arithmetic:
for (uint256 i = 0; i < dataArray.length; i++) {
    // loop body
}
Optimized loop with unchecked arithmetic:
for (uint256 i = 0; i < dataArray.length;) {
    // loop body
    unchecked { i++; }
}
```

3. Use of unchecked for Loop Increment Optimization

Recommendation

For loops that adhere to the aforementioned conditions, consider using unchecked blocks for increment operations to optimize gas consumption. Always ensure thorough testing after integrating such changes.

Roughly speaking this can save 30–40 gas per loop iteration. For lengthy loops, this can be significant! (This is only relevant if you are using the default solidity-checked arithmetic).

Status

Fixed

Auditor Response

Fixed changed all i++; to ++i; in loops.

4. Gas Efficiency: Prefer Pre-increment (++i) over Post-increment (i++)

Description

When incrementing an unsigned integer within Solidity, using pre-increment (i) is more gas-efficient than post-increment (i) or the addition assignment (i += 1). This efficiency gain stems from the inherent workings of these operations. With pre-increment, the value is incremented first and then returned, eliminating the need for an intermediary variable. On the other hand, post-increment increments the value but returns its initial state, thus necessitating a temporary variable when utilized.

Proof of Concept

Example Illustration:

```
Using post-increment:
uint256 i = 1;
i++;  // Result is 1, but `i` gets updated to 2 subsequently

Using pre-increment:
uint256 i = 1;
++i;  // Directly results in 2, and `i` is also updated to 2 without any inter
mediary steps.
```

4. Gas Efficiency: Prefer Pre-increment (++i) over Post-increment (i++)

Recommendation

To optimize gas usage in loops or recurring operations, consider adopting pre-increment over its alternatives. Ensuring these best practices can lead to tangible gas savings over numerous iterations, especially in contracts with high transaction frequency.

Status

Fixed

5. Redundant Initialization of Variables with Default Values

Path

LDCToken.sol 25:
for(uint256 i = 0; i < currencies.length; i++) {

LandAndFieldManager.sol
695: for (uint256 i = 0; i < length; i++) {

LivaatOwnershipDeed.sol
69: for (uint256 i = 0; i < currencies.length; i++) {
695: for (uint256 i = 0; i < length; i++) {

LivaatPartnershipDeed.sol
62: for (uint256 i = 0; i < currencies.length; i++) {
457: for (uint256 i = 0; i < length; i++) {

Description

In Solidity, variables are implicitly initialized with default values, such as 0 for uint, false for bool, and address(0) for address types. It is unnecessary and inefficient to explicitly set these variables to their default values. Doing so results in superfluous gas consumption and can be regarded as a programming anti-pattern.

Proof of Concept

Optimized usage:

```
for (uint256 i; i < currencies.length; ++i) {
```

5. Redundant Initialization of Variables with Default Values

Recommendation

We advise a thorough review of the contract to identify and remove any explicit initializations with default values. Simplifying the code in this manner will optimize gas consumption and align with best practices in Solidity development.

Status

Fixed

6. Suboptimal Use of memory Data Location for External Function Arguments

Description

In Solidity, using calldata instead of memory for function arguments in external functions can be more gas-efficient when these arguments are read-only. This is because the ABI decoding process involves copying data from calldata to memory, leading to increased gas costs. Accessing the argument directly from calldata eliminates the need for this extra step, saving gas.

Proof of Concept

Current Usage:

```
function addReservedLands(DataTypes.LandLocation[] memory lands) public onlyOw
ner { ... }
```

Optimized Usage:

```
function addReservedLands(DataTypes.LandLocation[] calldata lands) external on
lyOwner { ... }
```

Recommendation

It's recommended to switch the function's visibility to external and use calldata instead of memory for the lands parameter. This change would make the function more gas-efficient when dealing with sizable input arrays.

Status

Fixed

7. Potential Gas Optimization Using Double require Statements

Description

The current function declareShares has a single require statement that evaluates two separate conditions using the logical AND operator (&&). Although this approach is clear and concise, it might not be the most gas-efficient. Splitting this single require statement into two separate require statements can lead to a minor gas savings when the optimizer is enabled. The optimizer is more effective at optimizing individual require conditions as opposed to combined conditions. By splitting the conditions, the EVM can potentially exit early if the first condition fails, thus saving gas. With the optimizer enabled, this change can save around 10 gas per transaction. This issue is identified in various functions throughout the project repository.

Recommendation

Consider refactoring the code as follows:

```
require(currentMonth > 12, "invalid month");
require(currentMonth % 3 == 0, "invalid date");
```

This separation provides a clearer distinction between the two conditions and potentially results in a gas savings. Even though the savings might be minor on an individual transaction level, it could accumulate over numerous transactions, especially in contracts with high usage.

Status

Fixed

8. Optimizing Gas Usage by Refactoring Modifiers

Description

In Solidity, directly embedding require statements within modifiers can result in increased contract bytecode size, especially when these modifiers are repeatedly used throughout the contract. By refactoring these require statements into separate internal virtual functions, the size of the compiled contracts that utilize these modifiers can potentially be reduced. This optimization is particularly pertinent in contracts where the modifiers are extensively inherited and applied, as the savings accumulate with frequent use.

8. Optimizing Gas Usage by Refactoring Modifiers

Proof of Concept

In the current codebase, several modifiers contain direct require statements. For example:

```
modifier onlyContract(address contractAddr) {
    require(msg.sender == contractAddr, "invalid caller");
    _;
}

modifier onlyMinter() {
    require(msg.sender == minter, "only minter");
    _;
}

modifier isPaused() {
    require(!paused, "CONTRACT_PAUSED");
    _;
}
```

Recommendation

Refactor the require checks into individual internal virtual functions:

```
modifier onlyContract(address contractAddr) {
    _checkOnlyContract(contractAddr);
    _;
}

modifier onlyMinter() {
    _checkOnlyMinter();
    _;
}

modifier isPaused() {
    _checkIsPaused();
    _;
}
```

By employing this structure, the contract can potentially realize optimal gas deployment costs, especially as the number of functions employing the modifiers grows.

Status

Fixed

9. Use Assemble to check for address(0)

Description

Overview:

Multiple instances in the code utilize the require function to check for the zero address (address(0)). While this method is straightforward and widely recognized by the Ethereum development community, there is a gas-optimized approach available by using assembly to check for the zero address.

Details:

Gas savings of 6 gas per instance can be achieved if using assembly to check for address(0) as opposed to the traditional require method.

Impact

While the gas savings per instance is relatively small, given the frequent usage of zero address checks throughout the contract, adopting the assembly-based method can cumulatively lead to a noticeable reduction in gas consumption.

Proof of Concept

For example, in the mintReserved function:

require(to != address(0), “invalid to address”);

The gas-optimized assembly approach would look something like:

```
assembly {
    if iszero(to) {
        mstore(0x00, "zero address")
        revert(0x00, 0x20)
    }
}
```

Recommendation

Refactor the zero address checks in the mentioned and all the functions in the code to use the assembly-based method. This will provide gas savings and optimize the contract’s overall efficiency.

Status

Fixed

10. Use hardcoded address instead of address(this)

Path

LivaatOwnershipDeed.sol

Function

totalShares, circulatingTokens, cashOut, withdraw

Description

Overview:
Several functions within the LivaatOwnershipDeed.sol contract make use of the address(this) expression to reference the contract's own address. While this method is straightforward and prevalent, there's a more gas-efficient approach available by pre-calculating and using a hardcoded address instead.

Details:
Using address(this) requires an additional EXTCODESIZE operation to fetch the contract's address from its bytecode, leading to an increased gas cost. When the same address is used repeatedly, this added operation can significantly impact the cumulative gas expenditure.

Foundry's script.sol and solmate's LibRlp.sol contracts provide tools to pre-calculate and hardcode a contract's address, offering a more gas-efficient alternative.

Impact

While the gas overhead of address(this) might seem minimal on a per-use basis, the accumulation of these additional costs, especially in frequently-called functions, can result in a noticeable increase in total gas consumption.

Proof of Concept

For example, in the totalShares function:

return LDC.balanceOf(address(this)) - unclaimedShares;

A more gas-efficient approach would replace the address(this) expression with the pre-calculated hardcoded address.

Recommendation

- Use tools like Foundry's script.sol or solmate's LibRlp.sol to pre-calculate the contract's address.
- Replace all instances of address(this) in the mentioned functions with the pre-calculated hardcoded address.

10. Use hardcoded address instead of address(this)

- By making these adjustments, the contract can achieve gas savings and become more efficient in its operations.

Reference

Foundry's Guide on Address Calculation

Status

Acknowledged

Developer Response

I think it's not safe because we don't know what number the nonce will be at deployment time.

11. Inefficient Struct Layout - Violation of Tight Variable Packing Standards

Path

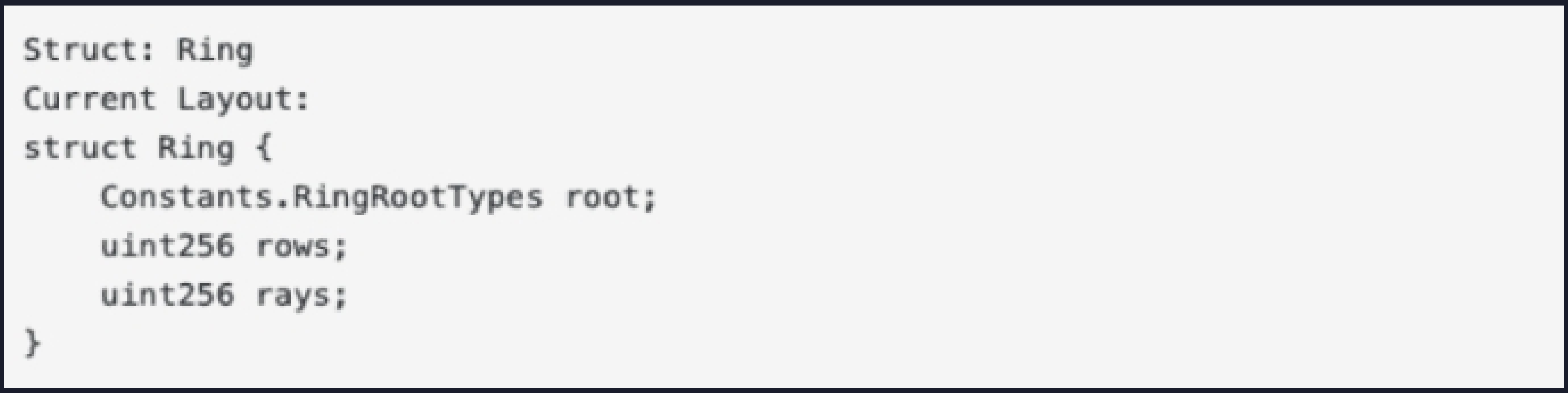
DataTypes.sol

Description

In Solidity smart contracts, one significant way to optimize gas costs is by properly ordering variables within structs. This ensures efficient storage utilization on the Ethereum network, a practice commonly referred to as 'Tight Variable Packing'. Ethereum's storage model allots 32 bytes for each storage slot. By efficiently organizing data types within a struct, multiple smaller-sized variables can share a single storage slot, reducing overall storage costs. Improperly ordered structs could lead to underutilized storage slots and unnecessarily increased gas costs for operations involving these structs.

Proof of Concept

Affected Structs:



```
Struct: Ring
Current Layout:
struct Ring {
    Constants.RingRootTypes root;
    uint256 rows;
    uint256 rays;
}
```

11. Inefficient Struct Layout - Violation of Tight Variable Packing Standards

Recommended Layout:

```
struct Ring {
    uint256 rows;
    uint256 rays;
    Constants.RingRootTypes root;
}
```

Reason:
The RingRootTypes enum potentially uses fewer than 32 bytes. By placing uint256 variables at the top, it optimally uses the storage slot, reducing wasted space.

```
Struct: Sector
Current Layout:
struct Sector {
    Constants.SectorTypes sectorType;
    uint256 ring;
    uint256 fromRay;
    uint256 toRay;
}
```

Recommended Layout:

```
struct Sector {
    uint256 ring;
    uint256 fromRay;
    uint256 toRay;
    Constants.SectorTypes sectorType;
}
```

11. Inefficient Struct Layout - Violation of Tight Variable Packing Standards

Reason:
Similar to Ring, the SectorTypes enum should be placed at the end for optimal storage utilization.

```
Struct: Land
Current Layout:
struct Land {
    Constants.SectorTypes sectorType;
    uint256 ring;
    uint256 row;
    uint256 ray;
    uint256 mintDate;
    bool isBought;
}
```

Recommended Layout:

```
struct Land {
    uint256 ring;
    uint256 row;
    uint256 ray;
    uint256 mintDate;
    Constants.SectorTypes sectorType;
    bool isBought;
}
```

Reason:
Grouping larger uint256 types together and placing the smaller sectorType enum and bool type at the end ensures better storage usage.

```
Struct: ToCallback
Current Layout:
struct ToCallback {
    address contractAddress;
    bytes4 funcSig;
    bool exists;
    uint256 index;
}
```

11. Inefficient Struct Layout - Violation of Tight Variable Packing Standards

Recommended Layout:

```
struct ToCallback {
    uint256 index;
    address contractAddress;
    bytes4 funcSig;
    bool exists;
}
```

Reason:
Larger data types, like uint256, should precede smaller ones for optimal packing. The address type uses 20 bytes, bytes4 uses 4 bytes, and bool uses just 1 byte.

Recommendation

Despite these benefits, packing variables in structs requires caution and should be performed understanding its implications:

- Ordering Matters: The variables should be ordered from largest to smallest to utilize the storage space optimally. Incorrect ordering could lead to unused space within the storage slots.
- Explicit Type Sizes: It's recommended to use explicit type sizes (like uint8, uint16, uint32) over int256 and uint256.
- Potential Overflow: Be aware of potential overflows when using smaller integer types.

To optimize gas consumption and smart contract efficiency, it is recommended to pack smaller data types into structs, keeping in mind to order variables from largest to smallest, use explicit type sizes, and be wary of potential overflows when using smaller integer types.

Reference

Tight variable Packing

Status

Fixed

12. Unused Imports

Path

LandNFT.sol

Description

DataTypes is imported but not used.

Recommendation

Consider removing it for better code readability.

Status

Fixed

13. Centralization in Shares declaration

Path

LivaatOwnershipDeed.sol

Description

Shares distribution is one of the prime feature of livaat. After the unlocking period the holders will be able to claim shares , for this the admin will first declareShare() , this funciton takes an input parameter backendTokens , If there is any miscalculation in this the shares will be wrongly calculated. It is advised to pass in an address of the contract which will be holding the backend tokens and fetch balance from that address.

Status

Acknowledged

14. Missing Check on amount while converting

Description

While converting the amount there should be a check whether the account have the desired amount or not. Then the burn function should be called.

Proof of Concept

```
function convert(uint256 amount) public {
LDC.burn(msg.sender, amount);

uint256 fee = (amount * convertFee) / 10000;

emit Convert(msg.sender, amount, amount - fee);
}
```

Status

Fixed

15. Missing Event Emission

Path

LivaatPartnershipDeed.sol

Description

When a setKYCApproval is called an event should be emitted to keep in check which account is set as KYCApproval.

Proof of Concept

```
   function setKYCApproval(address newAddr) external onlyOwner {
         require(newAddr != address(0), "invalid address");
         KYCApproval = newAddr;
    }
```

Status

Fixed

16. Using bools for storage incurs overhead

Path

Instances (7):

LDCToken.sol
14: mapping(address => bool) public allowedToMint;
16: bool public transfersEnabled;

LandAndFieldManager.sol
77: mapping(uint256 => mapping(uint256 => mapping(uint256 => bool))) public _isLandMinted;

LivaatOwnershipDeed.sol
34: bool public paused;
38: bool public cashOutAllowed;

LivaatPartnershipDeed.sol
37: bool public paused;
51: mapping(address => bool) public approvedToMint;

Description

There were 7 instances indentified where using bools for storage incurs overhead.

Booleans are more expensive than uint256 or any type that takes up a full word because each write operation emits an extra SLOAD to first read the slot's contents, replace the bits taken up by the boolean, and then write back.

Recommendation

Use uint256(1) and uint256(2) for true/false to avoid a Gwarmaccess (100 gas) for the extra SLOAD, and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past.

Status

Fixed

17. Short require strings save gas

Path

Instances (4):

LandAndFieldManager.sol
266: require(ring.root == Constants.RingRootTypes.Industrial, “only industrial ring available to claim for b2b”);

LivaatOwnershipDeed.sol
750: require(addr != address(0), “LivaatOwnershipDeed::setAuthorizedMinter: invalid address”);
755: require(addr != address(0), “LivaatOwnershipDeed::setUSDReceiver: invalid address”);

LivaatPartnershipDeed.sol
504: require(addr != address(0), “LivaatOwnershipDeed::setUSDReceiver: invalid address”);

Description

Strings in solidity are handled in 32 byte chunks. A require string longer than 32 bytes uses more gas. Shortening these strings will save gas.

Recommendation

It is recommended to keep the string size under 32 bytes to save gas.

Status

Fixed

18. Redundant msg.sender Check for Manager Role

Path

FieldNFT.sol

Function

safeMint

Description

The safeMint function, which is only callable by the manager, internally calls the _mint function. This _mint function further invokes _beforeTokenTransfer that contains a redundant check to ensure that msg.sender is the manager.

18. Redundant msg.sender Check for Manager Role

Proof of Concept

```
function safeMint(address to) external onlyManager returns (uint256) {
        uint256 tokenId = totalSupply() + 1;

        _safeMint(to, tokenId);

        return tokenId;
    }
```

```
function _beforeTokenTransfer(
        address from,
        address to,
        uint256 firstTokenId,
        uint256 batchSize
    ) internal override(ERC721, ERC721Enumerable) {
        super._beforeTokenTransfer(from, to, firstTokenId, batchSize);

        if(from == address(0) || to == address(0)) {
            return;
        }

        require(msg.sender == address(manager), "only manager can transfer");
    }
```

Recommendation

If _mint (or any other function calling _beforeTokenTransfer) is never used in contexts outside of the manager's control, consider removing the msg.sender check in _beforeTokenTransfer to optimize gas costs.

Status

Acknowledged

19. Functions guaranteed to revert when called by normal users can be marked payable

Path

Instances (52):

FieldNFT.sol

30: function safeMint(address to) external onlyManager returns (uint256) {
71: function setBaseURI(string memory newBaseURI) public onlyOwner {
80: function setManager(address _manager) external onlyOwner {

LDCToken.sol

47: function mint(address to, uint256 amount) public onlyMinter {
54: function burn(address account, uint256 amount) public onlyMinter {
78: function setUSDReceiver(address addr) external onlyOwner {
88: function getUSDReceiver() external onlyOwner view returns (address) {
96: function setBuyCurrency(IERC20 currency) external onlyOwner {
105: function addCurrency(address currency) public onlyOwner {
114: function removeCurrency(address currency) external onlyOwner {
121: function updateMinter(address addr, bool enabled) external onlyOwner {
130: function enableTransfers() external onlyOwner {

LandNFT.sol

74: function setBaseURI(string memory newBaseURI) public onlyOwner {
83: function setManager(address _manager) external onlyOwner {

LivaatOwnershipDeed.sol

488: function setB2BSubContract(address B2BSubAddr) external onlyOwner {
500: function declareShares(uint256 backendTokens) external onlyOwner {
531: function addCurrency(address currency) public onlyOwner {
540: function removeCurrency(address currency) external onlyOwner {
548: function withdraw(IERC20 currency) external onlyOwner {
562: function setCashOutCurrency(IERC20 currency) external onlyOwner {
570: function setBaseURI(string memory newURI) public onlyOwner {
578: function togglePause() external onlyOwner {
585: function toggleCashOutAllowed() external onlyOwner {
592: function setCashOutFee(uint256 newFee) external onlyOwner {
600: function setConvertFee(uint256 newFee) external onlyOwner {
608: function setAuthorizedMinter(address newAddress) external onlyOwner {
617: function getAuthorizedMinter() external onlyOwner view returns (address) {
624: function setUSDReceiver(address newAddress) external onlyOwner {
633: function getUSDReceiver() external onlyOwner view returns (address) {

19. Functions guaranteed to revert when called by normal users can be marked payable

640: function registerTransferCallback(address contractAddr, bytes4 funcSig) external onlyOwner {
650: function unregisterTransferCallback(address contractAddr, bytes4 funcSig) external onlyOwner {

LivaatPartnershipDeed.sol
308: function approveToMint(address addr, bool approved) external onlyKYCApproval {
317: function setB2CSubContract(address B2CSubAddr) external onlyOwner {
328: function addCurrency(address currency) public onlyOwner {
337: function removeCurrency(address currency) external onlyOwner {
344: function setBaseURI(string memory newURI) public onlyOwner {
352: function togglePause() external onlyOwner {
359: function setAuthorizedMinter(address newAddress) external onlyOwner {
368: function getAuthorizedMinter() external onlyOwner view returns (address) {
379: function setKYCApproval(address newAddr) external onlyOwner {
389: function getKYCApproval() external onlyOwner view returns (address) {
396: function setUSDReceiver(address newAddress) external onlyOwner {
405: function getUSDReceiver() external onlyOwner view returns (address) {
412: function registerTransferCallback(address contractAddr, bytes4 funcSig) external onlyOwner {
422: function unregisterTransferCallback(address contractAddr, bytes4 funcSig) external onlyOwner {

Description

If a function modifier such as onlyOwner is used, the function will revert if a normal user tries to pay the function. Marking the function as payable will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided.

Status

Fixed

20. Use !=0 instead of >0 for unsigned integer comparison

Path

Instances (26):

FieldNFT.sol
54: return bytes(baseURI_).length &gt; 0 ? string(abi.encodePacked(baseURI_, uint256(sectorType).toString(), &quot;.json&quot;)) : &quot;&quot;;

LandAndFieldManager.sol
125: require(landLength &gt; 0, &quot;invalid land numbers&quot;);
187: if(ringReservedCount[i] &gt; 0) ringLandMinted[Constants.RingTypes(i)].reservedToMint += ringReservedCount[i];
225: if(ringReservedCount[i] &gt; 0) ringLandMinted[Constants.RingTypes(i)].reservedToMint -= ringReservedCount[i];
252: require(landLength &gt; 0, &quot;invalid land numbers&quot;);
386: require(B2BSub.subscriptionOf(to).length &gt; 0 || B2CSub.subscriptionOf(to).length &gt; 0, &quot;user is not subscribed&quot;);
386: require(B2BSub.subscriptionOf(to).length &gt; 0 || B2CSub.subscriptionOf(to).length &gt; 0, &quot;user is not subscribed&quot;); 559: if(fieldTokenId &gt; 0) {
634: return land.row &gt; 0 &amp;&amp; land.row &lt;= ring.rows;
639: return land.ray &gt; 0 &amp;&amp; land.ray &lt;= ring.rays;

LandNFT.sol
57: return bytes(baseURI_).length &gt; 0 ? string(abi.encodePacked(baseURI_, rootRingId.toString(), &quot;.json&quot;)) : &quot;&quot;;

LivaatOwnershipDeed.sol
322: bytes(base).length &gt; 0 ? string(abi.encodePacked(base, tokenSubscription(tokenId).toString(), &quot;.json&quot;)) : &quot;&quot;;
330: require(claimableShare &gt; 0, &quot;CALLER_POSSESS_NO_SHARES&quot;);
351: require(amount &gt; 0, &quot;invalid amount&quot;);
362: require(amountOut &gt; 0, &quot;ZERO_OUTPUT&quot;);
401: if (balanceOf(to) &gt; 0) { 407: if (balanceOf(to) &gt; 0) {
430: if (balanceOf(to) &gt; 0) { 434: if (balanceOf(to) &gt; 0) {
553: require(balance &gt; 0, &quot;insuffecient balance&quot;);
683: if (balanceOf(to) &gt; 0) {
687: if (balanceOf(to) &gt; 0) {

LivaatPartnershipDeed.sol
if (b2bBalance &gt; 0) {

20. Use !=0 instead of >0 for unsigned integer comparison

132: if (b2bBalance &gt; 0) {
290: bytes(base).length &gt; 0 ? string(abi.encodePacked(base, tokenSubscription(tokenId).toString(), &quot;.json&quot;)) : &quot;&quot;; 445: if (b2bBalance &gt; 0) {

Recommendation

It's recommended to replace all instances of > 0 checks on unsigned integers with != 0 for clarity and to adhere to best coding practices. This change will make the code's intent more transparent, ensuring that other developers can more easily understand the code in the future.

Status

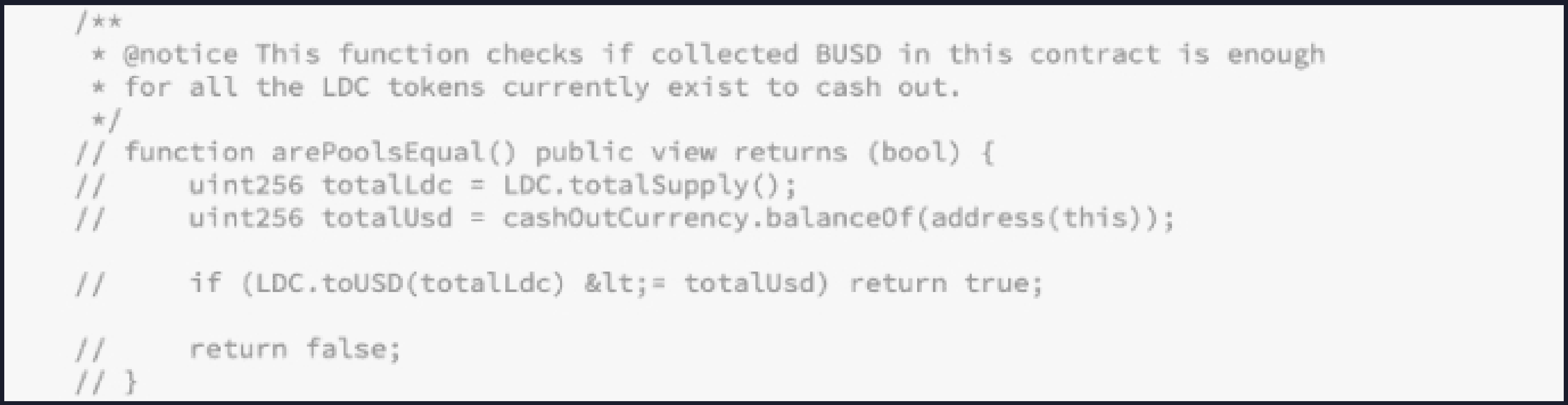
Fixed

21. Commented-Out Function in Contract

Path

LivaatOwnershipDeed.sol

Proof of Concept



```
    /**
     * @notice This function checks if collected BUSD in this contract is enough
     * for all the LDC tokens currently exist to cash out.
     */
    // function arePoolsEqual() public view returns (bool) {
    //     uint256 totalLdc = LDC.totalSupply();
    //     uint256 totalUsd = cashOutCurrency.balanceOf(address(this));

    //     if (LDC.toUSD(totalLdc) &lt;= totalUsd) return true;

    //     return false;
    // }
```

Recommendation

If this function is not meant to be a part of the production code, consider removing it entirely. In case this function is a placeholder or meant to be used in future versions of the contract, it's better to provide a descriptive comment explaining the reason for its current commented status and its potential future use. This approach ensures clarity and understanding for everyone who reviews the codebase.

Status

Fixed

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Livaatverse codebase. We performed our audit according to the procedure described above.

Some issues of Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Livaatverse smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Livaatverse smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the Livaatverse to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

850+
Audits Completed

$30B
Secured

$30B
Lines of Code Audited

Follow Our Journey

Audit Report
October, 2023

For

Livent
verse

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com

QuillAudits