Cyberscope

Audit Report

GemPad Lockers

January 2024

Network	BSC_TESTNET
Address	0x0bf94d51c7b525604d9e076bd875ca3468a75a6a
Audited by	© cyberscope

Table of Contents

Review

Explorer	https://testnet.bscscan.com/address/0x0bf94d51c7b525604d9e076bd875ca3468a75a6a

Audit Updates

Initial Audit	30 Jan 2024

Source Files

Filename	SHA256
LockV2.sol	e1d87db983cd4f5442e160f920494652e58c5d1db21aea9aa977d036c000aa8c
FullMath.sol	cf33688bcc87ed97503be8c33c733686b15469e83b15e4d9969f9f0ed4f04fcc
interfaces/IUniswapV3Pair.sol	d8314d7f1e0ace4292a06daa32bccd07ae4af97558da6d484e24830a9c7266a9
interfaces/IUniswapV3Factory.sol	cd9959c50c8bc592d729c6458bfb1d471e0ff1a799de609f54f57946da855dcf
interfaces/IUniswapV2Pair.sol	d2a719db1ef447e334a57cba344e05ea85ec3fb6766ef717b4448fc4a5032634
interfaces/IUniswapV2Factory.sol	a63a844ad84f9df76c9822e73adf2f66b6e0c100eece0c4af5103734cb3f4698
interfaces/IPoolInitializer.sol	1c6c3661807129156f46ac0e3a8a582a2600cbfe983751b79844981b573ac33a
interfaces/IPeripheryPayments.sol	ccecfe115faf38c5aa2f805f9b11826b8d0b906334fd8782c6e27e8fef43860e

interfaces/IPeripheryImmutableState.sol	87a6be0e845e5fef99ea73f2503ff2e9ab42 44f6e87fd1f82416d93d152082eb
interfaces/INonfungiblePositionManager.sol	313e6595a8a74ffcddbf3cd385eb60d4af5 240a5b279d71ba5bf9add360627dd
interfaces/ILockV2.sol	f053c5c524e980ebbbdd096a2d5bc40d1c 6625eaabec6b8d011ce4de82f93d05
interfaces/IERC721Permit.sol	d6e1c8868b63e77027761be60b3c7e9724 416c4cea8b148889b9066458a144c7
contracts/interfaces/IUniswapV3Pair.sol	d8314d7f1e0ace4292a06daa32bccd07ae 4af97558da6d484e24830a9c7266a9
contracts/interfaces/IUniswapV3Factory.sol	cd9959c50c8bc592d729c6458bfb1d471e 0ff1a799de609f54f57946da855dcf
contracts/interfaces/IUniswapV2Pair.sol	d2a719db1ef447e334a57cba344e05ea85 ec3fb6766ef717b4448fc4a5032634
contracts/interfaces/IUniswapV2Factory.sol	a63a844ad84f9df76c9822e73adf2f66b6e0 c100eece0c4af5103734cb3f4698
contracts/interfaces/IPoolInitializer.sol	1c6c3661807129156f46ac0e3a8a582a26 00cbfe983751b79844981b573ac33a
contracts/interfaces/IPeripheryPayments.sol	ccecfe115faf38c5aa2f805f9b11826b8d0b 906334fd8782c6e27e8fef43860e
contracts/interfaces/IPeripheryImmutableState.sol	87a6be0e845e5fef99ea73f2503ff2e9ab42 44f6e87fd1f82416d93d152082eb
contracts/interfaces/INonfungiblePositionManager.sol	313e6595a8a74ffcddbf3cd385eb60d4af5 240a5b279d71ba5bf9add360627dd
contracts/interfaces/ILockV2.sol	f053c5c524e980ebbbdd096a2d5bc40d1c 6625eaabec6b8d011ce4de82f93d05
contracts/interfaces/IERC721Permit.sol	d6e1c8868b63e77027761be60b3c7e9724 416c4cea8b148889b9066458a144c7

contracts/Lock/LockV2.sol	e1d87db983cd4f5442e160f920494652e5 8c5d1db21aea9aa977d036c000aa8c
contracts/Lock/FullMath.sol	cf33688bcc87ed97503be8c33c733686b1 5469e83b15e4d9969f9f0ed4f04fcc

Overview

The GemPad `LockV2` contract offers a versatile locking mechanism for both v2 and v3 liquidity pool tokens, as well as normal tokens, providing users with a wide array of functionalities to lock, unlock, and manage their assets. This comprehensive audit evaluates the contract's security, examines its business logic, and identifies potential improvements, ensuring the contract's robustness, efficiency, and the safety of its users in handling various types of token locks.

Function multipleLock

This function enables multiple users to lock tokens in the contract. Users specify the token to be locked, whether it's a liquidity pool token, the amounts to be locked, and the unlock date. The function also handles fee deductions for users who are not excluded from fees, with the fee amount varying based on the type of token being locked. It ensures that the number of owners matches the number of amounts specified and that the unlock date is set in the future. The function ultimately calls an internal function to process the multiple locks, handling token transfers and lock registrations.

Function multipleVestingLock

Similar to multipleLock, this function allows multiple users to lock tokens, but with additional vesting parameters. It includes `tgeDate`, `tgeBps`, `cycle`, and `cycleBps` to define the vesting schedule. The function also incorporates a fee mechanism for non-exempt users and performs checks to ensure the validity of the token, the vesting cycle, and the percentage of tokens released at the Token Generation Event (TGE) and each cycle. The function concludes by calling an internal function to execute the vesting locks, ensuring the correct handling of token transfers and lock registration with vesting conditions.

Function lockLpV3

This function is specifically designed for locking v3 liquidity pool positions. It allows a user to lock an NFT representing a liquidity position in a Uniswap v3 pool. The function checks for fee requirements, the validity of the NFT manager, and the future unlock date. It also verifies the project token's involvement in the liquidity pool and ensures the pool's

existence. Upon successful validation, the function locks the NFT position, updates cumulative lock information, and transfers the NFT from the user to the contract. This lock is particularly tailored for DeFi scenarios involving liquidity positions in Uniswap v3 pools.

Function unlock

This function allows users to unlock their locked tokens by specifying the lock ID. It first verifies that the caller is the owner of the lock and then determines the unlocking mechanism based on the `tgeBps` value of the lock. If `tgeBps` is greater than zero, indicating a vesting schedule, the function calls `_vestingUnlock` to handle the unlock process. Otherwise, it calls `_normalUnlock` for standard unlocks without vesting. The function ensures that the unlock conditions are met, such as the current time being past the unlock date, and then proceeds to transfer the tokens back to the user, updating the lock's state accordingly.

Function unlockAllAvailable

This function is designed for users who wish to unlock all their available locks in one transaction. It iterates through all the locks associated with the caller's address, both liquidity pool (LP) and normal locks. For each lock, it checks the `tgeBps` value to determine the appropriate unlocking mechanism, similar to the unlock function. The function then either calls `_vestingUnlock` for vesting locks or `_normalUnlock` for standard locks. This batch process allows users to efficiently unlock multiple locks without the need to individually specify each lock ID.

Function editLock

The editLock function provides users with the flexibility to modify certain aspects of their existing locks. It allows the lock owner to increase the amount locked or extend the unlock date. The function first ensures that the caller is the owner of the lock and that the lock has not been unlocked yet. If a new unlock date is provided, the function checks that it is both after the current time and the original unlock date, ensuring the lock's time extension is valid. For normal locks (non-NFT based), the function permits the addition of more tokens to the lock. This is achieved by updating the lock's amount and the cumulative lock information, followed by transferring the additional tokens from the user to the contract. The function ensures the exact amount of tokens is transferred and updates the lock's details

accordingly. This feature is particularly useful for users who wish to increase their stake or extend their commitment period in the locking contract.

Function increaseLiquidityCurrentRange

This function is designed for users who wish to increase the liquidity of their existing v3 liquidity pool (LP) locks. It allows the lock owner to add more liquidity to their current position by specifying the lock ID and the amounts of the two tokens to be added. The function first ensures that the lock is a v3 LP lock and that the caller is the owner of the lock. It also checks that the lock has not been unlocked yet. The function then transfers the specified amounts of the two tokens from the user to the contract and approves the NFT manager to use these tokens. It calls the `increaseLiquidity` function of the NFT manager to add liquidity to the position, updating the lock's amount and the cumulative lock information accordingly. This function is essential for users looking to enhance their position in a liquidity pool while maintaining the lock.

Function decreaseLiquidityCurrentRange

This function enables users to decrease the liquidity of their v3 LP locks. Users specify the lock ID and the amount of liquidity they wish to remove. The function verifies that the lock is a v3 LP lock, the caller is the owner, and the lock's unlock date has passed. It also checks that the lock's amount is sufficient for the requested liquidity decrease and that the lock has not been unlocked yet. The function then calls the decreaseLiquidity function of the NFT manager to remove the specified liquidity, transferring the corresponding amounts of the two tokens back to the user. It updates the lock's amount and the cumulative lock information, reflecting the decreased position. This function is crucial for users who need to partially withdraw their stake from a liquidity pool while keeping the remaining position locked.

Function collectFees

This function allows users to collect accumulated fees from their v3 liquidity pool (LP) locks without unlocking the position. Users specify the lock ID, and the function ensures that the lock is a v3 LP lock and that the caller is the owner of the lock. It then sets the maximum amounts for both tokens to collect all fees and calls the collect function of the NFT manager. The collected fees are transferred to the contract and then safely sent to the lock

owner. This function is particularly useful for LP lock owners who want to harvest their earned fees while keeping their liquidity position intact in the pool.

Function editLockDescription

This function provides the ability for users to update the description of their existing locks. Users specify the lock ID and the new description. The function verifies that the caller is the owner of the lock and then updates the lock's description with the provided new description. This feature is beneficial for users who need to modify the details or notes associated with their locks, allowing for better management and organization of their locked assets.

Function editProjectTokenMetaData

This function enables project owners to update the metadata associated with their project tokens. Users specify the token address and the new metadata. The function checks that the caller is the owner of the project token and then updates the project's metadata with the new information. This functionality is essential for project owners who need to keep their token information up-to-date. It helps maintain accurate and current information about the project tokens within the contract ecosystem.

Function transferProjectOwnerShip

This function allows the current owner of a project to transfer ownership to a new owner. The user specifies the token associated with the project and the address of the new owner. The function first verifies that the caller is indeed the current owner of the project. Upon validation, it updates the project's ownership to the new owner. This functionality is crucial for scenarios where project ownership needs to be reassigned. The transfer of ownership ensures continuity in project management and control.

Function transferLockOwnership

This function enables users to transfer the ownership of a specific lock to a new owner. Users provide the lock ID and the address of the new owner. The function checks that the caller is the current owner of the lock and then updates the lock's ownership to the new owner. It also updates the user lock IDs for both the current and new owners, ensuring that the lock is correctly associated with the new owner. This feature is particularly useful for

users who need to transfer their locked positions to another party. The ability to transfer lock ownership adds flexibility and utility to the locking mechanism, allowing for dynamic management of locked assets.

Roles

Owner

The owner has the authority to initialize the LockV2 contract. The owner is responsible for setting up various fee variables and excluding specific addresses from these fees.

The owner can interact with the following functions:

- function updateFee
- function excludeFromFee

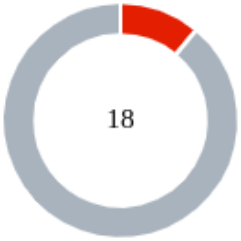
Users

The users can interact with the following functions:

- function multipleLock
- function multipleVestingLock
- function lockLpV3
- function unlock
- function unlockAllAvailable
- function withdrawableTokens
- function editLock
- function increaseLiquidityCurrentRange
- function decreaseLiquidityCurrentRange
- function collectFees
- function editLockDescription
- function editProjectTokenMetaData
- function transferProjectOwnerShip
- function transferLockOwnership
- function getTotalLockCount
- function getLockAt
- function allLpTokenLockedCount
- function allNormalTokenLockedCount
- function getCumulativeLpTokenLockInfoAt
- function getCumulativeNormalTokenLockInfoAt
- function lpLockCountForUser

- function lpLockForUserAtIndex
- function normalLockCountForUser
- function normalLockForUserAtIndex
- function totalLockCountForToken
- function getLocksForToken
- function getProject

Findings Breakdown



Critical	2
Medium	0
Minor / Informative	16

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	2	0	0	0
Medium	0	0	0	0
Minor / Informative	16	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Critical	LME	lockLpV3 Manipulation Exploit	Unresolved
Critical	VUI	V3 Unlock Inconsistency	Unresolved
Minor / Informative	ELFM	Exceeds Fees Limit	Unresolved
Minor / Informative	EIS	Excessively Integer Size	Unresolved
Minor / Informative	FLD	Fee Logic Duplication	Unresolved
Minor / Informative	MEM	Misleading Error Messages	Unresolved
Minor / Informative	MOA	Misleading Owner Assignment	Unresolved
Minor / Informative	MEE	Missing Events Emission	Unresolved
Minor / Informative	MVM	Missing ValidLock Modifier	Unresolved
Minor / Informative	MU	Modifiers Usage	Unresolved
Minor / Informative	PRE	Potential Reentrance Exploit	Unresolved
Minor / Informative	RCC	Redundant Condition Check	Unresolved
Minor / Informative	RRC	Redundant Require Checks	Unresolved
Minor / Informative	RSW	Redundant Storage Writes	Unresolved

●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L19	Stable Compiler Version	Unresolved

LME - lockLpV3 Manipulation Exploit

Criticality	Critical
Location	LockV2.sol#L238,935
Status	Unresolved

Description

The contract is currently designed to accept the `nftManager` address as a parameter in the `lockLpV3` function without performing any validation or checks. This lack of verification poses a significant security risk, as the provided `nftManager` address could potentially be a malicious contract. Consequently, this vulnerability could allow a malicious entity to create a lock through the `lockLpV3` function using a deceptive `nftManager`, which might mimic the behavior of a legitimate lock but with malicious intent. Furthermore, the `collectFees` function sends collected fees to the `msg.sender` and retrieves values based on the data returned from the `nftManager` contract. If the `nftManager` is malicious, it could potentially return maximum amounts, enabling the withdrawal of all funds from the contract, leading to substantial financial loss and contract manipulation.

Exploit Scenario:

Suppose that a malicious actor takes advantage of the lack of validation for the `nftManager` parameter in the `lockLpV3` function of the smart contract. Here's a step-by-step breakdown of the exploit:

1. The attacker calls the lockLpV3() function, passing in a custom contract address as the `nftManager`.
2. The custom `nftManager` contract's `positions(nftId)` function is designed to return USDT and USDC as the tokens involved in the liquidity pool, regardless of the actual tokens.
3. The `factory()` function of the malicious `nftManager` returns an address of another custom contract controlled by the attacker.
4. The `getPool()` function of the custom factory contract returns a mocked address, not an actual liquidity pool.

5. The contract proceeds to store the lock with the provided details, assuming the information is legitimate.
6. The `.safeTransferFrom` function transfers an NFT from the malicious contract to the main contract.
7. The attacker then calls the `collectFees()` function for the lock they created.
8. The `.collect` function of the malicious `nftManager` is programmed to return the maximum possible values for `amount0` and `amount1`, representing the collected fees.
9. The `amount0` and `amount1` variables are set to maximum values, which are interpreted as the fees collected by the malicious contract.
10. Since `amount0` and `amount1` can represent any value, all funds of the contract are transferred to the attacker's address, effectively draining the contract's assets.

This scenario highlights a critical vulnerability in the contract, where a lack of proper validation and verification mechanisms allows a malicious actor to manipulate the contract's functions and extract its funds. The exploit leverages the trust placed in the `nftManager` parameter and the associated functions to execute the attack.

```
function lockLpV3(
        address _owner,
        address nftManager,
        uint256 nftId,
        uint256 unlockDate,
        string memory description,
        string memory _metaData,
        address projectToken,
        address referrer
    ) external payable override returns (uint256 id) {
        {...
        (
            ,
            ,
            address token0,
            address token1,
            uint24 fee_,
            ,
            ,
            uint128 liquidity,
            ,
            ,
            ,

        ) = INonfungiblePositionManager(nftManager).positions(nftId);
        require(
            projectToken == token0 || projectToken == token1,
            "Invalid project token"
        );
        address factory =
INonfungiblePositionManager(nftManager).factory();
        address token = IUniswapV3Factory(factory).getPool(
            token0,
            token1,
            fee_
        );
        require(factory != address(0) && token != address(0), "Invalid
V3 LP");
        id = _locks.length;
        Lock memory newLock = Lock({
            id: id,
            token: token,
            owner: _owner,
            amount: liquidity,
            lockDate: block.timestamp,
            tgeDate: unlockDate,
            tgeBps: 1000000,
            cycle: 0,
            cycleBps: 0,
            unlockedAmount: 0,
```

```
            description: description,
            nftManager: nftManager,
            nftId: nftId
        });
        ...

        INonfungiblePositionManager(nftManager).safeTransferFrom(
            msg.sender,
            address(this),
            nftId
        );

        ...
        return id;
    }
```

```
  function collectFees(
        uint256 lockId
    ) external returns (uint256 amount0, uint256 amount1) {
        Lock storage userLock = _locks[lockId];
        require(userLock.nftManager != address(0), "No V3 LP lock");
        ...
        INonfungiblePositionManager.CollectParams
            memory params = INonfungiblePositionManager.CollectParams({
                tokenId: userLock.nftId,
                recipient: address(this),
                amount0Max: type(uint128).max,
                amount1Max: type(uint128).max
            });

        (amount0, amount1) =
INonfungiblePositionManager(userLock.nftManager)
            .collect(params);

        // send collected feed back to owner
        (
            ,
            ,
            address token0,
            address token1,
            ,
            ,
            ,
            ,
            ,
            ,
            ,
            ,

        ) = INonfungiblePositionManager(userLock.nftManager).positions(
                userLock.nftId
            );
        IERC20(token0).safeTransfer(userLock.owner, amount0);
        IERC20(token1).safeTransfer(userLock.owner, amount1);
    }
```

Recommendation

It is recommended to implement stringent validation checks for the `nftManager` address in the `lockLpV3` function to ensure it is a legitimate and trusted contract. This could involve verifying the `nftManager` against a list of approved managers or ensuring it adheres to specific interface standards. Additionally, consider implementing safeguards in the `collectFees` function to prevent excessive or unauthorized withdrawals. By

incorporating these security measures, the contract can significantly mitigate the risk of manipulation and protect its assets from potential exploits by malicious entities.

VUI - V3 Unlock Inconsistency

Criticality	Critical
Location	LockV2.sol#L298,574,588,611
Status	Unresolved

Description

The contract contains the `lockLpV3` function, which is designed to lock v3 liquidity pool positions. This function sets the `tgeBps` variable of the lock to `1000000`, a value greater than zero. The unlocking process for these locks is managed by the `unlock` and `unlockAllAvailable` functions, which call the internal function `_vestingUnlock` when `tgeBps` is greater than zero. However, there is a critical oversight in the implementation. The `_vestingUnlock` function, which is intended to handle the unlocking of v3 locks, does not contain any functionality to transfer the NFT position back to the user. In contrast, the transfer of the NFT position is handled within the `_normalUnlock` function, which is only called if `tgeBps` is equal to zero. This discrepancy means that when a v3 lock is unlocked, the NFT position is not properly returned to the user, leading to a significant functional issue in the contract.

```
function lockLpV3(
        address _owner,
        address nftManager,
        uint256 nftId,
        uint256 unlockDate,
        string memory description,
        string memory _metaData,
        address projectToken,
        address referrer
    ) external payable override returns (uint256 id) {
        {
        ...
        id = _locks.length;
        Lock memory newLock = Lock({
            id: id,
            token: token,
            owner: _owner,
            amount: liquidity,
            lockDate: block.timestamp,
            tgeDate: unlockDate,
            tgeBps: 1000000,
            cycle: 0,
            cycleBps: 0,
            unlockedAmount: 0,
            description: description,
            nftManager: nftManager,
            nftId: nftId
        });
        ...
        }
    function unlock(uint256 lockId) external override
validLock(lockId) {
        Lock storage userLock = _locks[lockId];
        require(
            userLock.owner == msg.sender,
            "You are not the owner of this lock"
        );

        if (userLock.tgeBps > 0) {
            _vestingUnlock(userLock, false);
        } else {
            _normalUnlock(userLock, false);
        }
    }

    function unlockAllAvailable() external {
        uint256 length = _userLpLockIds[msg.sender].length();
        for (uint256 i = 0; i < length; i++) {
            Lock storage userLock =
_locks[_userLpLockIds[msg.sender].at(i)];
```

```
            if (userLock.tgeBps > 0) {
                _vestingUnlock(userLock, true);
            } else {
                _normalUnlock(userLock, true);
            }
        }
        length = _userNormalLockIds[msg.sender].length();
        for (uint256 i = 0; i < length; i++) {
            Lock storage userLock = _locks[
                _userNormalLockIds[msg.sender].at(i)
            ];
            if (userLock.tgeBps > 0) {
                _vestingUnlock(userLock, true);
            } else {
                _normalUnlock(userLock, true);
            }
        }
    }
        function _normalUnlock(Lock storage userLock, bool _noRevert)
internal {
        ...
        _tokenToLockIds[userLock.token].remove(userLock.id);
        if (userLock.nftManager != address(0)) {

INonfungiblePositionManager(userLock.nftManager).safeTransferFrom(
                address(this),
                msg.sender,
                userLock.nftId
            );
        }
        ...
        }
```

Recommendation

It is recommended to reconsider the code implementation for unlocking v3 liquidity pool positions. Since the `lockLpV3` function sets the `tgeBps` value, the unlocking process for these locks should be correctly implemented in the `_vestingUnlock` function rather than the `_normalUnlock` function. This change will ensure that the NFT position is appropriately transferred back to the user upon unlocking a v3 lock. It is crucial to align the unlocking logic with the locking parameters set by `lockLpV3` to maintain the intended functionality and integrity of the contract. Additionally, thoroughly testing the updated implementation to verify that it correctly handles both the unlocking process and the transfer of NFT positions is essential for ensuring the contract's reliability and user trust.

ELFM - Exceeds Fees Limit

Criticality	Minor / Informative
Location	LockV2.sol#L120
Status	Unresolved

Description

The contract is designed to allow the contract owner to update various fees through the `updateFee` function. This function accepts the five parameters, `projectCreationFee`, `lpTokenNormalLockFee`, `lpTokenVestingLockFee`, `normalTokenNormalLockFee`, and `normalTokenVestingLockFee`. The `onlyOwner` modifier ensures that only the contract owner can call this function. However, there are no checks or limitations on the values that can be set for these fees. This lack of restrictions means the contract owner has the authority to set the fees to any number, potentially leading to scenarios where the fees could be set to excessively high or arbitrary amounts. This could result in unreasonable costs for users interacting with the contract and may lead to trust issues or misuse of the contract.

```
    function updateFee(
         uint256 projectCreationFee,
         uint256 lpTokenNormalLockFee,
         uint256 lpTokenVestingLockFee,
         uint256 normalTokenNormalLockFee,
         uint256 normalTokenVestingLockFee
     ) external onlyOwner {
         fee = Fee({
             projectCreationFee: projectCreationFee,
             lpTokenNormalLockFee: lpTokenNormalLockFee,
             lpTokenVestingLockFee: lpTokenVestingLockFee,
             normalTokenNormalLockFee: normalTokenNormalLockFee,
             normalTokenVestingLockFee: normalTokenVestingLockFee
         });
     }
```

Recommendation

It is recommended to introduce a maximum limit for each fee parameter within the `updateFee` function. Implementing a cap on the fee values would prevent the contract owner from setting unreasonably high fees, thereby protecting users from potential exploitation. This can be achieved by adding require statements that check if the fee values are within a predefined acceptable range.

EIS - Excessively Integer Size

Criticality	Minor / Informative
Location	LockV2.sol#L213
Status	Unresolved

Description

The contract is using a bigger unsigned integer data type that the maximum size that is required. By using an unsigned integer data type larger than necessary, the smart contract consumes more storage space and requires additional computational resources for calculations and operations involving these variables. This can result in higher transaction costs, longer execution times, and potential scalability bottlenecks.

Specifically, the contract is currently utilizing `uint256` to store the `tgeBps` , `cycle` , and `cycleBps` variables in the `multipleVestingLock` function. However, the function includes `require` statements that restrict the maximum value of these variables to `1000000` . This constraint implies that the full range of uint256 (which can store values up to `2^256-1`) is not necessary for these variables. Since `uint256` is a 256-bit unsigned integer, using it for values that are known to be significantly smaller leads to inefficient gas usage. This inefficiency arises because the Ethereum Virtual Machine (EVM) allocates a full 256-bit storage slot for each uint256 variable, even if the actual value stored is much smaller.

```
function multipleVestingLock(
        ...
        uint256 tgeBps,
        uint256 cycle,
        uint256 cycleBps,
        ....
    ) external payable override returns (uint256[] memory) {
        ...
            require(cycle > 0, "Invalid cycle");
            require(tgeBps > 0 && tgeBps < 1000000, "Invalid bips for
TGE");
            require(
                cycleBps > 0 && cycleBps < 1000000,
                "Invalid bips for cycle"
            );
            require(
                tgeBps + cycleBps <= 1000000,
                "Sum of TGE bps and cycle should be less than 10000"
            );
        }
    ...
    }
```

Recommendation

To address the inefficiency associated with using an oversized unsigned integer data type, it is recommended to accurately determine the required size based on the range of values the variable needs to represent. It is recommended to use an appropriate uint type that matches the maximum possible value of these variables. Given that the maximum value for `tgeBps`, `cycle`, and `cycleBps` is capped at `1000000`, a smaller uint type, such as `uint32`, would be sufficient to store these values. Using a smaller uint type will reduce the gas cost associated with storing and manipulating these variables, especially when they are stored in the contract's state. This change will make the contract more gas-efficient while still ensuring that the variables can hold the required range of values. Additionally, using a smaller uint type can improve the clarity of the code by indicating the expected range of values for these variables.

FLD - Fee Logic Duplication

Criticality	Minor / Informative
Location	LockV2.sol#L155,200,238
Status	Unresolved

Description

The contract contains the three functions, `multipleLock` , `multipleVestingLock` , and `lockLpV3` , each implementing a fee mechanism for users not excluded from fees. The fee calculation and charging process in these functions are very similar, with the primary differences being the specific fee values assigned. Specifically the `lpTokenNormalLockFee` , `normalTokenNormalLockFee` in the `multipleLock` function, the `lpTokenVestingLockFee` , `normalTokenVestingLockFee` in the `multipleVestingLock` and the `lpTokenNormalLockFee` in the `lockLpV3` function. This results in significant code duplication across these functions, as the same fee logic is repeated. Code duplication can lead to an unnecessarily large codebase, complicating maintenance and updates. It also increases the risk of inconsistencies and errors, particularly if future changes are made to the fee logic.

```
    function multipleLock(
        ...
    ) external payable override returns (uint256[] memory) {
        {
            if (!isExcludedFromFee[_msgSender()]) {
                uint256 _fee = isLpToken
                    ? fee.lpTokenNormalLockFee
                    : fee.normalTokenNormalLockFee;
                if (projects[projectToken].owner == address(0)) {
                    _fee += fee.projectCreationFee;
                }
                require(msg.value >= _fee, "Not enough funds for fees");
                (bool sent, ) = payable(owner()).call{value: _fee}("");
                require(sent, "Failed to charge fee");
            }
        ...
    }
function multipleVestingLock(
        ...
    ) external payable override returns (uint256[] memory) {
        if (!isExcludedFromFee[_msgSender()]) {
            uint256 _fee = isLpToken
                ? fee.lpTokenVestingLockFee
                : fee.normalTokenVestingLockFee;
            if (projects[projectToken].owner == address(0)) {
                _fee += fee.projectCreationFee;
            }
            require(msg.value >= _fee, "Not enough funds for fees");
            (bool sent, ) = payable(owner()).call{value: _fee}("");
            require(sent, "Failed to charge fee");
        }
    ...
    }

    function lockLpV3(
        ...
    ) external payable override returns (uint256 id) {
        {
            if (!isExcludedFromFee[_msgSender()]) {
                uint256 _fee = fee.lpTokenNormalLockFee;
                if (projects[projectToken].owner == address(0)) {
                    _fee += fee.projectCreationFee;
                }
                require(msg.value >= _fee, "Not enough funds for fees");
                (bool sent, ) = payable(owner()).call{value: _fee}("");
                require(sent, "Failed to charge fee");
            }
```

Recommendation

It is recommended to consider implementing a single function to handle the common fee logic across `multipleLock`, `multipleVestingLock`, and `lockLpV3`. This new function would accept parameters necessary to determine the appropriate fee and execute the fee charging process. Centralizing the fee logic into one function will reduce code duplication and simplify maintenance. This approach also facilitates easier implementation of future updates or changes to the fee logic, as modifications would only need to be made in one location. Furthermore, having a dedicated function for fee handling enhances code readability and organization, making the contract more understandable and manageable for developers.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	LockV2.sol#L887 contracts/Lock/LockV2.sol#L887
Status	Unresolved

Description

The contract is missing error messages. The are no error messages to accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(userLock.amount >= liquidity)
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

MOA - Misleading Owner Assignment

Criticality	Minor / Informative
Location	LockV2.sol#L311,499,540
Status	Unresolved

Description

The contract includes the `lockLpV3` , `_lockLpToken` , and `_lockNormalToken` functions, which are designed to handle different types of token locks. A common issue across these functions is the assignment of the project owner. The contract sets the `msg.sender` who initiates the first lock as the project owner. This approach assumes that the first user to lock tokens is the rightful project owner, which might not always be true. As a result, the contract could incorrectly assign project ownership, leading to potential misrepresentation and management issues. This misalignment between the actual project owner and the owner recorded in the contract could create confusion and governance problems.

```
    function lockLpV3(
        address _owner,
        address nftManager,
        uint256 nftId,
        uint256 unlockDate,
        string memory description,
        string memory _metaData,
        address projectToken,
        address referrer
    ) external payable override returns (uint256 id) {
        ...

        CumulativeLockInfo storage tokenInfo =
cumulativeLockInfo[token];
        if (tokenInfo.projectToken == address(0)) {
            tokenInfo.projectToken = projectToken;
            tokenInfo.factory = factory;
        } else {
            projectToken = tokenInfo.projectToken;
        }
        ...
        }

  function _lockLpToken(
        ...
    ) private returns (uint256 id) {
        ...
        if (project.owner == address(0)) {
            project.owner = msg.sender;
            project.metaData = _metaData;
        }
        project.lpLockedTokens.add(token);
    }

    function _lockNormalToken(
       ...
    ) private returns (uint256 id) {
        ...

        Project storage project = projects[token];
        if (project.owner == address(0)) {
            project.owner = msg.sender;
            project.metaData = _metaData;
        }
    }
```

Recommendation

It is recommended to revise the contract's approach to assigning project ownership in these functions. A more reliable and verifiable method should be implemented to ensure that project ownership is accurately assigned to the legitimate owner. This could involve a dedicated ownership claim process, where project owners explicitly assert their ownership, or a verification mechanism to confirm the legitimacy of the `msg.sender` as the project owner. By adopting a more precise and secure method for determining project ownership, the contract can avoid misleading representations and enhance its overall reliability and governance structure.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	LockV2.sol#L120,136
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
    function updateFee(
        uint256 projectCreationFee,
        uint256 lpTokenNormalLockFee,
        uint256 lpTokenVestingLockFee,
        uint256 normalTokenNormalLockFee,
        uint256 normalTokenVestingLockFee
    ) external onlyOwner {
        fee = Fee({
            projectCreationFee: projectCreationFee,
            lpTokenNormalLockFee: lpTokenNormalLockFee,
            lpTokenVestingLockFee: lpTokenVestingLockFee,
            normalTokenNormalLockFee: normalTokenNormalLockFee,
            normalTokenVestingLockFee: normalTokenVestingLockFee
        });
    }

    function excludeFromFee(
        address account,
        bool isExcluded
    ) external onlyOwner {
        isExcludedFromFee[account] = isExcluded;
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

MVM - Missing validLock Modifier

Criticality	Minor / Informative
Location	LockV2.sol#L801,873
Status	Unresolved

Description

The contract is equipped with `increaseLiquidityCurrentRange` and `decreaseLiquidityCurrentRange` functions to modify liquidity in v3 locks. However, these functions do not utilize the `validLock` modifier, which is essential for validating the lock ID passed as a parameter. Without this modifier, there is a risk that invalid or non-existent lock IDs could be processed, potentially leading to unintended behavior or errors in the contract's execution. The `validLock` modifier is crucial for ensuring that operations are performed on valid and existing locks, thereby maintaining the integrity and reliability of the contract.

```
    function increaseLiquidityCurrentRange(
        uint256 lockId,
        uint256 amount0ToAdd,
        uint256 amount1ToAdd
    ) external returns (uint128 liquidity, uint256 amount0, uint256
amount1) {
        ...
    }

    function decreaseLiquidityCurrentRange(
        uint256 lockId,
        uint128 liquidity
    ) external returns (uint256 amount0, uint256 amount1) {
       ...
    }
```

Recommendation

It is recommended to incorporate the `validLock` modifier in both the `increaseLiquidityCurrentRange` and `decreaseLiquidityCurrentRange` functions. This addition will enforce a check on the validity of the lock ID before proceeding with any modifications to the liquidity. By doing so, the contract can prevent erroneous

operations on invalid locks and uphold the robustness of its functionality. Implementing this change will enhance the security and reliability of the contract, ensuring that liquidity modifications are executed only on valid and intended locks.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	LockV2.sol#L576,620,769,772,807,809,812,878,880,888,939,941,984
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(
    userLock.owner == msg.sender,
    "You are not the owner of this lock"
);
...
require(userLock.unlockedAmount == 0, "Nothing to unlock");
...
require(userLock.nftManager != address(0), "No V3 LP lock");
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

PRE - Potential Reentrance Exploit

Criticality	Minor / Informative
Location	LockV2.sol#L648
Status	Unresolved

Description

The contract makes an external call to transfer funds to recipients using the transfer method. The recipient could be a malicious contract that has an untrusted code in its fallback function that makes a recursive call back to the original contract. The re-entrance exploit could be used by a malicious user to drain the contract's funds or to perform unauthorized actions. This could happen because the original contract does not update the state before sending funds.

Specifically, the contract is designed to transfer an NFT back to the user during the unlock process. However, if `msg.sender` is a malicious contract, it can exploit the order of operations. The function first calls `safeTransferFrom` to transfer the NFT, and only afterward updates the `userLock.unlockedAmount` state variable. A malicious contract can use its fallback function to re-enter the current contract before `userLock.unlockedAmount` is updated. This reentrancy vulnerability enables the malicious contract to execute any function within the current contract. For instance, by calling the `lockLpV3` function, it can initiate an additional lock on the NFT position, leading to an inflated representation of the position's amount. Consequently, the contract will erroneously assume it holds more assets than it actually does, compromising its accuracy in asset representation. Consequently, the contract inaccurately assumes that it holds more funds, potentially leading to unintended consequences, such as manipulating contract states. The vulnerability arises because the state update (`userLock.unlockedAmount = unlockAmount`) occurs after the external call (`safeTransferFrom`), which is a common pattern leading to reentrancy attacks.

```
    function _normalUnlock(Lock storage userLock, bool _noRevert)
internal {
        ...
        if (userLock.nftManager != address(0)) {

INonfungiblePositionManager(userLock.nftManager).safeTransferFrom(
                address(this),
                msg.sender,
                userLock.nftId
            );
        } else {
            IERC20(userLock.token).safeTransfer(
                msg.sender,
                unlockAmount
            );
        }
        userLock.unlockedAmount = unlockAmount;

        emit LockRemoved(userLock.id, userLock, unlockAmount,
block.timestamp);
    }
```

Recommendation

It is recommended to update the `userLock.unlockedAmount` state variable before making the external call to `safeTransferFrom`. This change adheres to the "checks-effects-interactions" pattern, which is a best practice in smart contract development to prevent reentrancy attacks. By updating all relevant state variables before performing external calls, the contract can ensure that its state is consistent even if a reentrant call is made. In this case, updating `userLock.unlockedAmount` before transferring the NFT will prevent a malicious contract from exploiting the reentrancy vulnerability. Additionally, consider using reentrancy guards or modifiers to further protect against such attacks. Implementing these changes will enhance the security of the contract and protect it from potential exploits.

RCC - Redundant Condition Check

Criticality	Minor / Informative
Location	LockV2.sol#L747
Status	Unresolved

Description

The contract contains the `_withdrawableTokens` function which is designed to calculate the amount of tokens that can be withdrawn based on various conditions of the `userLock`. The function includes a series of if statements to check different conditions. Notably, there is the `if` condition which returns zero if the current block timestamp is less than the `userLock.tgeDate`. However, later in the function, there is a redundant check with the condition `if (block.timestamp >= userLock.tgeDate)`. This second condition is unnecessary because if the function execution has passed the first check, it implies that `block.timestamp` is already greater than or equal to `userLock.tgeDate`. Therefore, the second if statement is always true and does not serve any functional purpose, leading to redundant code execution.

```
function _withdrawableTokens(
        Lock memory userLock
    ) internal view returns (uint256) {
        ...
        if (block.timestamp < userLock.tgeDate) return 0;
        ...
        uint256 currentTotal = 0;
        if (block.timestamp >= userLock.tgeDate) {
            currentTotal =
                (((block.timestamp - userLock.tgeDate) /
userLock.cycle) *
                    cycleReleaseAmount) +
                tgeReleaseAmount; // Truncation is expected here
        }
        ...
    }
```

Recommendation

It is recommended to remove the second if check within the `_withdrawableTokens` function, as it will always evaluate to true given the earlier check. Eliminating this redundant condition will streamline the function, making the code more efficient and easier to read. Simplifying the function in this manner not only improves the overall quality of the code but also reduces the potential for confusion or errors in future modifications or updates to the contract. By ensuring that each part of the code serves a distinct and necessary purpose, the contract becomes more maintainable and robust.

RRC - Redundant Require Checks

Criticality	Minor / Informative
Location	LockV2.sol#L166,167,212
Status	Unresolved

Description

The contract is currently employing `require` statements in both the `multipleLock` and `multipleVestingLock` functions to validate certain conditions. These functions are declared as external, and both of them call the internal function `_multipleLock`. Notably, the `require` statements used in `multipleLock` and `multipleVestingLock` for checking conditions such as length mismatches and future date settings are duplicated in the `_multipleLock` function. This redundancy means that the same conditions are being checked twice. Once in the external functions and again in the internal function. Since `_multipleLock` is called within the external functions, the initial checks in `multipleLock` and `multipleVestingLock` do not add any functional value and only contribute to unnecessary gas consumption and code complexity.

```
    function multipleLock(
        ...
    ) external payable override returns (uint256[] memory) {
        {
         ...
            require(owners.length == amounts.length, "Length
mismatch");
            require(
                unlockDate > block.timestamp,
                "Unlock date needs to be set to future date"
            );
        }
        return
            _multipleLock(
                owners,
                amounts,
                token,
                isLpToken,
                [unlockDate, 0, 0, 0],
                description,
                _metaData,
                projectToken,
                referrer
            );
    }

    function multipleVestingLock(
        ...
    ) external payable override returns (uint256[] memory) {
        ...
        {
            require(token != address(0), "Invalid token");
            ...
        }
        return
            _multipleLock(
                owners,
                amounts,
                token,
                isLpToken,
                [tgeDate, tgeBps, cycle, cycleBps],
                description,
                _metaData,
                projectToken,
                referrer
            );
    }

    function _multipleLock(
        a...
```

```
    ) internal returns (uint256[] memory) {
        {
            require(owners.length == amounts.length, "Length
mismatch");
            require(
                vestingSettings[0] > block.timestamp,
                "TGE date should be set in the future"
            );
            require(token != address(0), "Invalid token");
        }
    ...
    }
```

Recommendation

It is recommended to remove the additional `require` statements from the `multipleLock` and `multipleVestingLock` functions and rely solely on the checks conducted within the internal function `_multipleLock`. This approach will streamline the contract by eliminating redundant operations, thereby reducing gas costs and simplifying the codebase. By centralizing these checks in `_multipleLock`, the contract will maintain its integrity and functionality while becoming more efficient and easier to maintain. Additionally, this change will make the contract more readable and reduce the potential for errors in future updates or modifications.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	LockV2.sol#L136
Status	Unresolved

Description

The contract modifies the state of the following variables without checking if their current value is the same as the one given as an argument. As a result, the contract performs redundant storage writes, when the provided parameter matches the current state of the variables, leading to unnecessary gas consumption and inefficiencies in contract execution.

```
    function excludeFromFee(
        address account,
        bool isExcluded
    ) external onlyOwner {
        isExcludedFromFee[account] = isExcluded;
    }
```

Recommendation

The team is advised to implement additional checks within to prevent redundant storage writes when the provided argument matches the current state of the variables. By incorporating statements to compare the new values with the existing values before proceeding with any state modification, the contract can avoid unnecessary storage operations, thereby optimizing gas usage.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	LockV2.sol#L150,196,239,244,993,1173
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
string memory _metaData
address _owner
address _projectToken
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	LockV2.sol#L748
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
currentTotal =
                (((block.timestamp - userLock.tgeDate) /
userLock.cycle) *
                    cycleReleaseAmount) +
                tgeReleaseAmount
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	LockV2.sol#L1143,1144
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
address possibleFactoryAddress
address factory
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	LockV2.sol#L2 interfaces/ILockV2.sol#L2
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.22;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
LockV2	Implementation	ILockV2, IERC721Receiver, Initializable, OwnableUpgradeable		
	initialize	Public	✓	initializer
	updateFee	External	✓	onlyOwner
	excludeFromFee	External	✓	onlyOwner
	multipleLock	External	Payable	-
	multipleVestingLock	External	Payable	-
	lockLpV3	External	Payable	-
	_multipleLock	Internal	✓	
	_sumAmount	Internal		
	_createLock	Internal	✓	
	_lockLpToken	Private	✓	
	_lockNormalToken	Private	✓	
	_registerLock	Private	✓	
	unlock	External	✓	validLock
	unlockAllAvailable	External	✓	-
	_normalUnlock	Internal	✓	

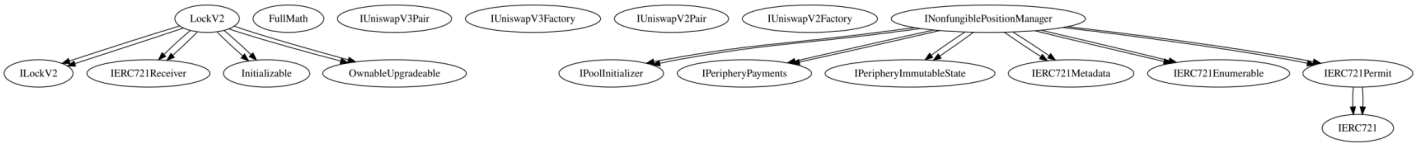
	_vestingUnlock	Internal	✓	
	withdrawableTokens	External		-
	_withdrawableTokens	Internal		
	editLock	External	✓	validLock
	increaseLiquidityCurrentRange	External	✓	-
	decreaseLiquidityCurrentRange	External	✓	-
	collectFees	External	✓	-
	editLockDescription	External	✓	validLock
	editProjectTokenMetaData	External	✓	-
	transferProjectOwnerShip	Public	✓	-
	transferLockOwnership	Public	✓	validLock
	_safeTransferFromEnsureExactAmount	Internal	✓	
	getTotalLockCount	External		-
	getLockAt	Public		-
	allLpTokenLockedCount	Public		-
	allNormalTokenLockedCount	Public		-
	getCumulativeLpTokenLockInfoAt	External		-
	getCumulativeNormalTokenLockInfoAt	External		-
	lpLockCountForUser	Public		-
	lpLockForUserAtIndex	External		-
	normalLockCountForUser	Public		-
	normalLockForUserAtIndex	External		-
	totalLockCountForToken	External		-

	getLocksForToken	Public		-
	_parseFactoryAddress	Internal		
	_isValidLpToken	Private		
	getProject	External		-
	onERC721Received	Public	✓	-
ILockV2	Interface			
	multipleLock	External	Payable	-
	multipleVestingLock	External	Payable	-
	unlock	External	✓	-
	editLock	External	✓	-
	lockLpV3	External	Payable	-
ILockV2	Interface			
	multipleLock	External	Payable	-
	multipleVestingLock	External	Payable	-
	unlock	External	✓	-
	editLock	External	✓	-
	lockLpV3	External	Payable	-
LockV2	Implementation	ILockV2, IERC721Receiver, Initializable, OwnableUpgradeable		
	initialize	Public	✓	initializer

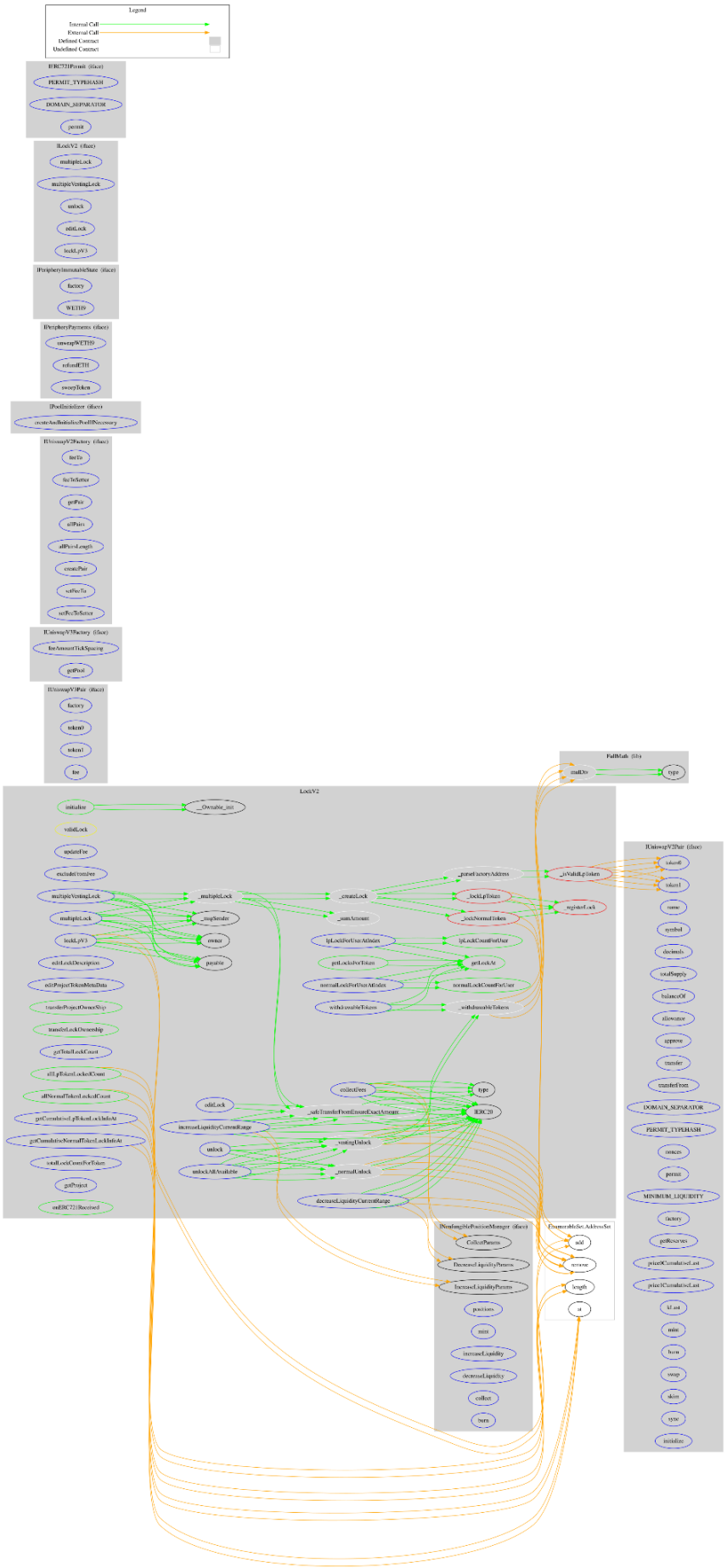
	updateFee	External	✓	onlyOwner
	excludeFromFee	External	✓	onlyOwner
	multipleLock	External	Payable	-
	multipleVestingLock	External	Payable	-
	lockLpV3	External	Payable	-
	_multipleLock	Internal	✓	
	_sumAmount	Internal		
	_createLock	Internal	✓	
	_lockLpToken	Private	✓	
	_lockNormalToken	Private	✓	
	_registerLock	Private	✓	
	unlock	External	✓	validLock
	unlockAllAvailable	External	✓	-
	_normalUnlock	Internal	✓	
	_vestingUnlock	Internal	✓	
	withdrawableTokens	External		-
	_withdrawableTokens	Internal		
	editLock	External	✓	validLock
	increaseLiquidityCurrentRange	External	✓	-
	decreaseLiquidityCurrentRange	External	✓	-
	collectFees	External	✓	-
	editLockDescription	External	✓	validLock
	editProjectTokenMetaData	External	✓	-

	transferProjectOwnerShip	Public	✓	-
	transferLockOwnership	Public	✓	validLock
	_safeTransferFromEnsureExactAmount	Internal	✓	
	getTotalLockCount	External		-
	getLockAt	Public		-
	allLpTokenLockedCount	Public		-
	allNormalTokenLockedCount	Public		-
	getCumulativeLpTokenLockInfoAt	External		-
	getCumulativeNormalTokenLockInfoAt	External		-
	lpLockCountForUser	Public		-
	lpLockForUserAtIndex	External		-
	normalLockCountForUser	Public		-
	normalLockForUserAtIndex	External		-
	totalLockCountForToken	External		-
	getLocksForToken	Public		-
	_parseFactoryAddress	Internal		
	_isValidLpToken	Private		
	getProject	External		-
	onERC721Received	Public	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IERC721Permit (iface)
PERMIT_TYPEHASH
DOMAIN_SEPARATOR
permit
ILockV2 (iface)
multipleLock
multipleVestingLock
unlock
editLock
lockLpV3
IPeripheryImmutableState (iface)
factory
WETH9
IPeripheryPayments (iface)
unwrapWETH9
refundETH
sweepToken
IPoolInitializer (iface)
createAndInitializePoolIfNecessary
IUniswapV2Factory (iface)
feeTo
feeToSetter
getPair
allPairs
allPairsLength
createPair
setFeeTo
setFeeToSetter
IUniswapV3Factory (iface)
feeAmountTickSpacing
getPool
IUniswapV3Pair (iface)
factory
token0
token1
fee
FullMath (lib)
mulDiv
type
LockV2
initialize
_Ownable_init
validLock
updateFee
excludeFromFees
multipleVestingLock
multipleLock
lockLpV3
editLockDescription
editProjectTokenMetaData
transferProjectOwnership
transferLockOwnership
getTotalLockCount
allLpTokenLockedCount
allNormalTokenLockedCount
getCumulativeLpTokenLockInfoAt
getCumulativeNormalTokenLockInfoAt
totalLockCountForToken
getProject
onERC721Received
multipleLock
_msgSender
owner
paraSw
_createLock
_sumAmount
lpLockForUserAtIndex
getLockForToken
normalLockForUserAtIndex
withdrawableTokens
_parseFactoryAddress
_lockLpToken
_lockNormalToken
lpLockCountForUser
getLockAt
normalLockCountForUser
withdrawableTokens
_isValidLpToken
registerLock
editLock
increaseLiquidityCurrentRange
unlock
unlockAllAvailable
collectFees
_safeTransferFromEnsureExactAmount
_vestingUnlock
_normalUnlock
decreaseLiquidityCurrentRange
type
IERC20
INonfungiblePositionManager (iface)
CollectParams
DecreaseLiquidityParams
IncreaseLiquidityParams
positions
mint
increaseLiquidity
decreaseLiquidity
collect
burn
EnumerableSet.AddressSet
add
remove
length
at
IUniswapV2Pair (iface)
token0
token1
name
symbol
decimals
totalSupply
balanceOf
allowance
approve
transfer
transferFrom
DOMAIN_SEPARATOR
PERMIT_TYPEHASH
nonces
permit
MINIMUM_LIQUIDITY
factory
getReserves
price0CumulativeLast
price1CumulativeLast
kLast
mint
burn
swap
skim
sync
initialize

Summary

GemPad Lockers contract implements a locker mechanism. This audit investigates security issues, business logic concerns and potential improvements.

Initial Audit, 30 Jan 20224

At the time of the audit report, the contract with address 0x0bf94d51c7b525604D9e076BD875CA3468A75a6a is pointed out by the following proxy address: 0xFBe918da16bc7F4eBEa94D3Ca4184358A6e1c92A.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io