Code Security Audit Report

For

ThrustPad

8th Nov 2024

Table of Contents

Summary

Audit Result

Damocles

Appendix

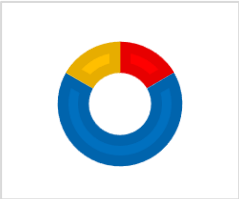
About

Damocles

Project Executive Summary

TYPES	MAINNET	AUDIT METHOD
Launchpad	EDU Chain	Cross-manual review; Static Analysis
DEVELOP LAUNGE	**CODEBASE**	**Commit ID**
Solidity	Contract Link	869cfded0a
Fixed codebase	Contract Link	5670dc459d

Vulnerability Summary



6	5	0	1
Total Findings	Solved	Pending	Acknowledged

SEVERITY	STATUS	DESCRIPTION
CRITICAL	**0**	Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risk
HIGH	**1 Solved**	High risks can include centralization issues and logical errors. Under specific circumstances, these high risks can lead to loss of funds and/or control of the project
MEDIUM	**4 Pending**	Medium risks may not pose a direct risk to user's funds, but they can affect the overall functioning of a platform
LOW	**1 Acknowledged**	Low risks can be any of the above, but on a smaller scale. They generally do not compromise the over all integrity of the project, but they may be less efficient than other solution

INFO	**0**	Info errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code

Audit Scope Summary

Code Repository: https://github.com/ThrustPad-ILO/contracts/tree/main/contracts/	
File Name	**Commit ID**
ThrustpadInstantAirdrop.sol	869cfded0a7aa3fa59591f1f01bb3efe5ab1088a
ThrustpadInstantAirdropFactory.sol	869cfded0a7aa3fa59591f1f01bb3efe5ab1088a
ThrustpadFairLaunch.sol	869cfded0a7aa3fa59591f1f01bb3efe5ab1088a
ThrustpadFairLaunchFactory.sol	869cfded0a7aa3fa59591f1f01bb3efe5ab1088a
ThrustpadLocker.sol	869cfded0a7aa3fa59591f1f01bb3efe5ab1088a
ThrustpadLockerFactory.sol	869cfded0a7aa3fa59591f1f01bb3efe5ab1088a
ThrustpadToken.sol	869cfded0a7aa3fa59591f1f01bb3efe5ab1088a
ThrustpadTokenFactory.sol	869cfded0a7aa3fa59591f1f01bb3efe5ab1088a
ThrustpadStaker.sol	869cfded0a7aa3fa59591f1f01bb3efe5ab1088a
ThrustpadStakerFactory.sol	869cfded0a7aa3fa59591f1f01bb3efe5ab1088a

Damocles

Audit Approach Summary

This report has been prepared **ThrustPad** to discover issues and vulnerabilities in the source code of the **Thrustpad Main Repo** as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Cross Manual Review and Static Analysis techniques

- The auditing process pays special attention to the following considerations:
- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders
- .Thorough line-by-line manual review of the entire codebase by industry experts
- Cross-audit mode of the current code by more than three security engineers.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live

Damocles

Audit Result

TPD-01(High): Improper configuration leading to the airdrop logic not being executed correctly

Category	Severity	Location	Status
Code Issue	High	ThrustpadInstantAirdrop.sol:37	Solved

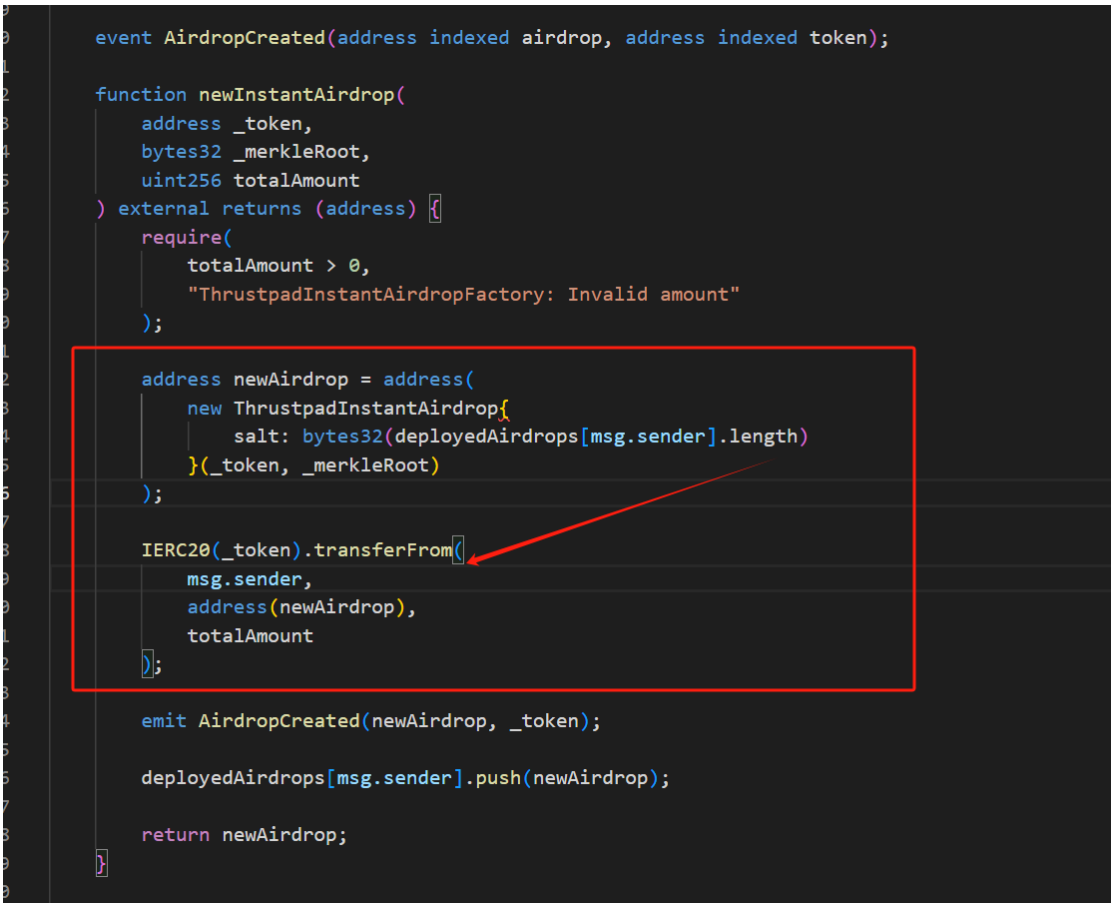
Description

When creating an Airdrop contract through the Factory contract, tokens are transferred to the newly created Airdrop contract. However, when the claim function is executed in the Airdrop contract, the target address for token transfer remains the Airdrop contract itself. This leads to two issues: users are unable to properly receive airdropped tokens, and the tokens intended for airdrop distribution are stuck in the Airdrop contract and cannot be transferred.

Vulnerability Analysis

1. When creating a new airdrop contract through the **ThrustpadInstantAirdropFactory's newInstantAirdrop** function, an instantiated Airdrop contract is created using the create2 method. Upon obtaining the contract address, the creator transfers a specified amount of tokens to the Airdrop

contract using the **transferFrom** function, which is a standard ERC20 transfer function.



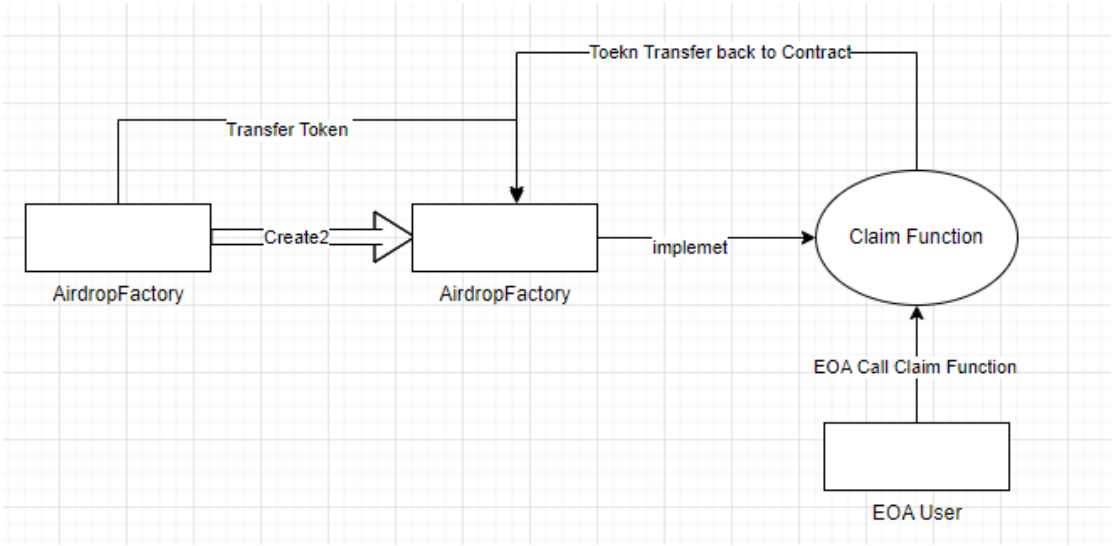
```
event AirdropCreated(address indexed airdrop, address indexed token);

function newInstantAirdrop(
    address _token,
    bytes32 _merkleRoot,
    uint256 totalAmount
) external returns (address) {
    require(
        totalAmount > 0,
        "ThrustpadInstantAirdropFactory: Invalid amount"
    );

    address newAirdrop = address(
        new ThrustpadInstantAirdrop{
            salt: bytes32(deployedAirdrops[msg.sender].length)
        }(_token, _merkleRoot)
    );

    IERC20(_token).transferFrom(
        msg.sender,
        address(newAirdrop),
        totalAmount
    );

    emit AirdropCreated(newAirdrop, _token);

    deployedAirdrops[msg.sender].push(newAirdrop);

    return newAirdrop;
}
```

2. In the **claim** function of the airdrop instance contract, when eligible users claim tokens, the contract calls the **token.transfer(address(this), amount)** function. In theory, this function should transfer the tokens to the eligible user's address, i.e., msg.sender. However, it is incorrectly configured as address(this) here, causing the tokens to be frozen in the current contract and unable to be transferred correctly.

Damocles

```
function claim(
    uint256 amount,
    bytes32[] calldata proof
) external nonReentrant {
    require(
        !claimed[msg.sender],
        "ThrustpadInstantAirdrop: Account already claimed"
    );

    _verifyProof(proof, amount, msg.sender);

    claimed[msg.sender] = true;
    totalClaimed += amount;

    token.transfer(address(this), amount);

    emit Claimed(msg.sender, amount);
}
```

3. Token transfer flow direction.



Recommendation

1. Use msg.sender instead of address(this) in the code of **token.transfer(address(this), amount)** .

Damocles

TPD-02(Medium): Not limiting the number of purchases by users leads to centralization issues in token sales.

Category	Severity	Location	Status
Code Issue	Medium	ThrustpadFairLaunch.sol:66	Solved

Description

There is no restriction on the number of purchases by participating users in the **buyToken** function, allowing users to continuously call the function to buy tokens up to the maximum amount.

Vulnerability Analysis

1. In the buyToken function implemented in the Fiarlaunch contract, the sale contract restricts the current time, as well as the maximum and minimum purchase quantities. However, it does not limit the number of purchases. If a contract is used to buy tokens, the purchase quantity each time will be the maximum value allowed for a single purchase. This situation can result in tokens being concentrated in a single user's wallet.

Damocles

```
function buyToken() public payable {
    require(
        block.timestamp >= config.startDate,
        "ThrustpadFairLaunch: sale has not started yet"
    );
    require(
        block.timestamp <= config.endDate,
        "ThrustpadFairLaunch: sale has ended"
    );
    require(
        msg.value >= config.minimumBuy,
        "ThrustpadFairLaunch: amount is less than minimum buy"
    );
    require(
        msg.value <= config.maximumBuy,
        "ThrustpadFairLaunch: amount is more than maximum buy"
    );
    require(
        totalSold + msg.value <= config.hardCap,
        "ThrustpadFairLaunch: hard cap reached"
    );

    totalSold += msg.value;
    totalContributors += 1;
    purchaseHistory[msg.sender] += msg.value;

    emit TokenBought(msg.sender, config.token, msg.value);
}
```

Recommendation

Increase restrictions on the purchase quantity to avoid tokens being centralized in a single wallet address as much as possible.

Damocles

TPD-03(Medium): The liquidity deployment lacks permission control.

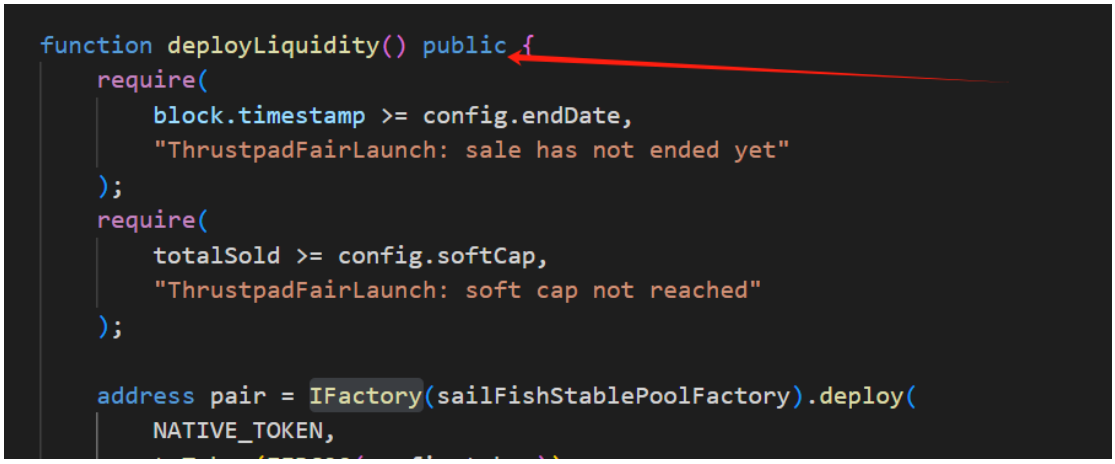
Category	Severity	Location	Status
Code Issue	Medium	ThrustpadFairLaunch.sol:159	Solved

Description

If **deployLiquidity**() lacks access control, all users can call this function after the sale ends. In this scenario, malicious users may deploy liquidity before the project team is ready, potentially leading to undesirable market conditions.

Vulnerability Analysis

At the time of declaring this function **deployLiquidity**, it was not set to exclusive permissions; currently, it has public access, meaning anyone can call it.



Recommendation

Add permission verification or use the onlyOwner authentication method.

TPD-04(Medium):Condition designed incorrectly, preventing users from smoothly executing the normal logic flow.

Category	Severity	Location	Status
Logic Issue	Medium	ThrustpadStaker.sol:149 158 ThrustpadStaker.sol:173 187	Solved

Description

When a user claims rewards, the system calculates the total rewards and subtracts the already claimed rewards to determine the current claimable amount. However, a check has been added during the reward claim process to compare the claimable amount with the already claimed amount. The issue arises because the claimable amount is recalculated from the beginning each time, while the claimed amount increases with each claim. This can lead to a situation where, if the user does not manage the claim intervals properly, there will be a period where the rewards cannot be continuously claimed due to this mismatch in calculation.

Vulnerability Analysis

1. When calculating rewards, the current lock-up period is first calculated, representing the total rewards from the beginning of the staking period until now. This total is then reduced by the rewards already claimed to determine the current reward amount.

```
function _claimableRewards(
    Stake memory stake,
    uint256 stakeIndex
) private view returns (ClaimableReward memory claimableReward) {
    uint256 mantissa = 1000;
    uint256 uptillNow = block.timestamp - stake.start;

    uint256 appliedEDUAPY = ((stake.eduAPY * mantissa * uptillNow) /
        365 days);
    uint256 appliedTokenAPY = ((stake.tokenAPY * mantissa * uptillNow)
        365 days);

    uint256 eduReward = (((stake.amount * appliedEDUAPY) / 100) /
        mantissa) / option.tokenToEDURate;

    uint256 tokenReward = (((stake.amount * appliedTokenAPY) / 100) /
        mantissa);

    return
        ClaimableReward({
            edu: eduReward - stake.claimedEDU,
            token: tokenReward - stake.claimedToken,
            index: stakeIndex
        });
}
```

2. After calculating the current claimable amount against the claimed rewards, the system checks if the current claimable quantity exceeds the previously claimed amount. Before transferring the rewards, the system updates the total claimed

rewards by adding the current claimed quantity to the past total. This results in a situation where the total claimed rewards continuously increase over time. As a result, a user must wait longer between reward claims, as the time required to claim the next reward becomes the staking time at the previous claim + 1 day

```
function claimAvailableRewards(
    uint256 stakeIndex
) public nonReentrant whenNotPaused {
    require(stakes[stakeIndex].owner == msg.sender, "Not owner");

    Stake storage stake = stakes[stakeIndex];

    require(!stake.unstaked, "Already unstaked");

    ClaimableReward memory claimableReward = _claimableRewards(
        stake,
        stakeIndex
    );

    require(
        claimableReward.edu > stake.claimedEDU ||
            claimableReward.token > stake.claimedToken,
        "No rewards to claim"
    );

    uint256 ethToClaim = claimableReward.edu;
    uint256 tokenToClaim = claimableReward.token;

    stake.claimedEDU += claimableReward.edu;
    stake.claimedToken += claimableReward.token;

    payable(msg.sender).transfer(ethToClaim);

    require(token.transfer(msg.sender, tokenToClaim), "Transfer failed

    emit ClaimedRewards(msg.sender, ethToClaim, tokenToClaim);
```

3. For example, if the daily reward is 5 tokens, and a user stakes for two days, the claimable reward would be 10 tokens. If the user tries to claim rewards on the third day, the claimable reward would be 5 tokens, while the total claimed rewards would still be 10 tokens. In this case, the condition check would fail, and the user would need to wait until the fifth day when claimable rewards exceed the total claimed rewards before being able to claim rewards again.

In the third day:

- *eduReward = 15*
- *stake.claimedEDU = 10*
- *claimableReward.edu = 15 - 10 = 5*

the condition:

claimableReward.edu > stake.claimedEDU // 5 > 10 false

Recommendation

Only calculate new rewards accumulated since the last settlement, avoiding comparison with total claimed amounts, allowing users to continuously claim current rewards.

```
uint256 newPeriod = block.timestamp - lastCalculation;

uint256 mantissa = 1000;
uint256 appliedEDUAPY = ((stake.eduAPY * mantissa * newPeriod) / 365 days);
uint256 appliedTokenAPY = ((stake.tokenAPY * mantissa * newPeriod) / 365 days);

uint256 newEduReward = (((stake.amount * appliedEDUAPY) / 100) / mantissa) / option.tokenToEDURate;
uint256 newTokenReward = (((stake.amount * appliedTokenAPY) / 100) / mantissa);
```

```
ClaimableReward memory newRewards = _claimableRewards(stake, stakeIndex);

require(
    newRewards.edu > 0 || newRewards.token > 0,
    "No new rewards available"
);
```

TPD-05(Medium):Unchecked APY updates may lead to inability to properly distribute rewards in the future.

Category	Severity	Location	Status
Logic Issue	Medium	ThrustpadStaker.sol:301	Solved

Description

When updating the APY, there was no strict condition checking. If the current APY value is increased, there needs to be a corresponding increase in reward tokens stored in the contract. However, in the original code, only the APY value was modified.

Vulnerability Analysis

1. When creating the staker contract, it calculates the required amount of tokens and EDU, and transfers the corresponding amount of tokens to the Staker contract. At the same time, it requires that the current apyEDU >= 2 and apyToken >= 8

Damocles

```
function newStaker(
    stakeOption memory option
) public payable returns (address) {
    require(option.apyEdu >= 2, "APY EDU must be greater than 2");
    require(option.apyToken >= 8, "APY Token must be greater than 8");
    require(
        ERC20(option.token).decimals() == 18,
        "ThrustpadStakerFactory: token must have 18 decimals"
    );

    uint256 tokenNeeded = (option.hardCap * option.apyToken) / 100;
    uint256 eduNeeded = ((option.hardCap * option.apyEdu) / 100) /
        option.tokenToEDURate;

    require(
        msg.value >= eduNeeded,
        "EDU deposit to cover yields payout not enough"
    );
    require(
        tokenNeeded >= option.rewardPoolToken,
        "Deposit tokens to cover yields payout not enough"
    );

    address newStaking = address(
        new ThrustpadStaker{
            value: msg.value,
            salt: bytes32(deployedStakers[msg.sender].length)
        }(option)
    );
    deployedStakers[msg.sender].push(newStaking);

    IERC20(option.token).transferFrom(
        msg.sender,
        address(newStaking),
        option.rewardPoolToken
    );
```

2. However, when updating the APY, it only updates the APY value without validating its legitimacy. Additionally, it doesn't check whether the current token balance in the contract meets the required amount for rewards under the updated APY

Damocles

```
function updateAPY(
    uint256 _eduAPY,
    uint256 _tokenAPY
) external whenNotPaused onlyOwner {
    option.apyEdu = _eduAPY;
    option.apyToken = _tokenAPY;

    emit APYUpdated(_eduAPY, _tokenAPY);
}
```

Recommendation

Calculate the amount of tokens needed for distributing rewards under the current APY. If the current balance is less than the required amount of tokens, transfer the difference to the current contract.

TPD-06(Low): The inaccuracy of using TX.origin for ownership control.

Category	Severity	Location	Status
Logic Issue	Low	ThrustpadInstantAirdrop.sol:18 ThrustpadFairLaunch.sol 62 ThrustpadLocker.sol 34 ThrustpadStaker.sol 76 ThrustpadToken.sol 21	Acknowledged

Description

When using the Factory contract to create corresponding logic contracts, owner privileges are granted to the Tx.origin address. While this operation is secure when tx.origin is used as an EOA (Externally Owned Account) address, there are potential risks when contract-type users make the calls.

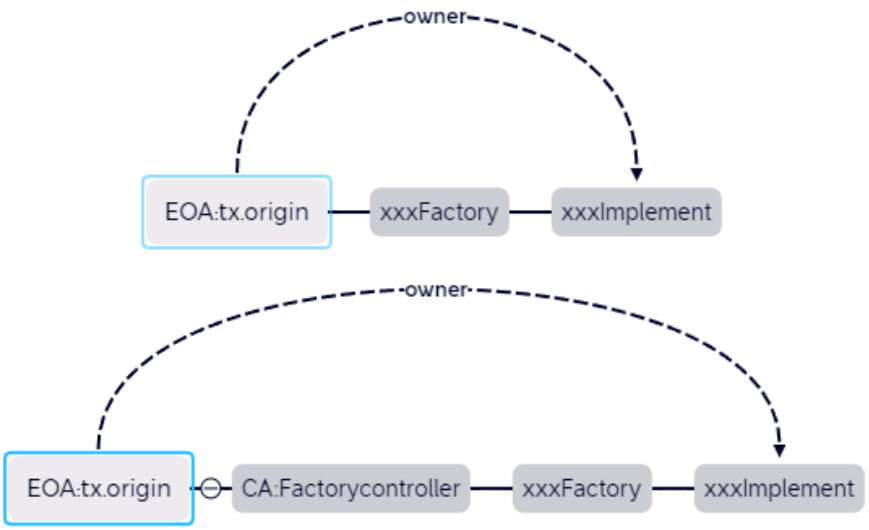
Vulnerability Analysis

1. For example, when creating a locker contract, if the caller uses a multi-signature contract account to call the Factory contract to create a Locker contract, the tokens from the multi-signature contract account are transferred to the locker contract during creation.

Damocles

```
function newLock(
    IERC20 token,
    uint256 lockTime,
    uint256 amount
) public returns (address) {
    address newLocker = address(
        new ThrustpadLocker{
            salt: bytes32(deployedLocks[msg.sender].length)
        }(token, lockTime, amount)
    );
    deployedLocks[msg.sender].push(newLocker);

    token.transferFrom(msg.sender, newLocker, amount);

    emit NewLock(msg.sender, newLocker, amount);

    return address(newLocker);
}
```

2. However, the Locker contract saves the original caller's address (the EOA that initiated the transaction through the multi-sig) as the beneficiary. As a result, when the release function is called, the tokens will be completely transferred to the initial EOA address that originated the transaction



Damocles

```
constructor(
    IERC20 _token,
    uint256 _lockTime,
    uint256 _amount
) Ownable(tx.origin) {
    require(_lockTime > 0, "TokenLock: lock time should greater than 0");
    token = _token;
    beneficiary = tx.origin;
    lockTime = _lockTime;
    startTime = block.timestamp;
    lockAmount = _amount;

    emit TokenLockStart(
        tx.origin,
        address(_token),
        block.timestamp,
        _lockTime
    );
}

function release() public onlyOwner {
    require(
        block.timestamp >= startTime + lockTime,
        "TokenLock: lock time not expired"
    );

    uint256 amount = token.balanceOf(address(this));
    require(amount > 0, "TokenLock: no tokens to release");

    token.transfer(beneficiary, amount);
    released = true;

    emit Release(msg.sender, address(token), block.timestamp, amount);
}
```

Recommendation

Configure specific ownership relationships based on different function requirements, and minimize using tx.origin directly as Owner.

APPENDIX

Audit Categories

Categories	Description
Code Issues	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities
Logic Issues	Logic Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralizedcontrols over the code

About

Damocles is a 2023 web3 security company specializing in online security services, including smart contract audit, Product audit, penetration testing, GameFi security audit and cheat detection.

Main Web: https://damocleslabs.com/

Twitter: https://twitter.com/DamoclesLabs

Email: support@damocleslabs.com