Mode Lock

Security Review

Cantina Managed review by:
Alex the Entreprenerd, Security Researcher

May 3, 2024

Contents

1 Introduction

1.1 Disclaimer

A security review a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While the review endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that a security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.2 Risk assessment

Severity	**Description**
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.2.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Mode is the Ethereum L2 that rewards users for growing the network via new economic mechanisms. Built on the OP Stack L2, designed for growth that incentivises and directly rewards developers, users and protocols to grow Mode and the Superchain ecosystem.

From May 3rd the security researchers conducted a review of mode-lock/src on commit hash e7d9e63b. A total of **8** issues in the following risk categories were identified:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 1
- Low Risk: 1
- Gas Optimizations: 0
- Informational: 6

3 Findings

3.1 Medium Risk

3.1.1 MED - Not using safeTransfer may cause sweep to fail for some tokens

Severity: Medium Risk

`ModeStaking` uses `IERC20` from `forge-std`:

```
import {IERC20} from "forge-std/interfaces/IERC20.sol";
```

The interface is:

```
function transfer(address to, uint256 amount) external returns (bool);
```

This will fail for tokens that do not return a boolean (e.g. USDT).

Recommendation: Use `safeTransfer`.

The Mode team fixed this issue by using `safeERC20`.

3.2 Low Risk

3.2.1 QA - Setter limits are not enforced in the constructor

Severity: Low Risk

The setter looks as follows:

```
function setCooldownDuration(uint96 duration) external onlyOwner {
    if (duration > MAX_COOLDOWN_DURATION) {
        revert InvalidCooldown();
    }

    uint96 previousDuration = cooldownDuration;
    cooldownDuration = duration;
    emit CooldownDurationUpdated(previousDuration, cooldownDuration);
}
```

Whereas the constructor doesn't perform a check on cooldownDuration:

```
cooldownDuration = _cooldown;
```

Recommendation: Apply the same validation for the `cooldownDuration` in the constructor.

This issue has been fixed by the Mode team by adding the same check as in `setCooldownDuration`.

3.3 Informational

3.3.1 Refactoring: Can use `uint32` for `cooldownDuration`

Severity: Informational

Since `UserPosition.lockEnd` is a `uint32` you could set `cooldownDuration` to also be a `uint32` as that way you won't have any additional concerns around casting types

It's worth noting that u32 can hold data until `Sun Feb 07 2106`, which should be enough for a points campaign

3.3.2 Analysis - Game theory on points

Severity: Informational

The following is speculative and implementation dependent.

By checking at daily point updates, smart depositors may figure out if you're tracking all amounts instead of only the locked amounts. If you elect to track deposits as `position.amount`, then the locking mechanism will be gameable as the contract doesn't reduce `amount` until a withdrawal has happened.

If you want to track only tokens that are deposited, and unlocked, you will have to compute: `position.amount - position.withdrawable`.

If you instead will want to track only tokens that are deposited and unvested, you will have to track: `position.amount` until `block.timestamp < position.lockEnd`, and then `position.amount - position.withdrawable`.

Recommendation: Track points on event changes (system accruals) (e.g. via theGraph). In my opinion, it's best to track `position.amount - position.withdrawable` at all times, which makes the code simple, and prevents any gaming.

This also reinforces a bonus for lockers, removing any advantage in "*being in cooldown*".

3.3.3 Informational - Lack of sweep for Mode

Severity: Informational

If people make the mistake of sending `mode`, that token will not be rescatable. Make sure to put a warning for end users.

A sweep for `mode` could be implemented by keeping track of a global amount.

Recommendation:

```
uint256 totalDeposited;
```

- Add a way to count all tokens deposited.
- On withdrawal reduce that count.
- That's the total `mode` tokens that the contract should hold.
- Allow withdrawing any amount of `mode` that is above that.

3.3.4 Informational - You can lock `u256`, but unlocks are `u224`

Severity: Informational

This means that in theory someone may have to perform multiple cooldowns in order to fully withdraw from the lock. This should not cause a loss of tokens, just a delay in unlocking them

Recommendation: I believe Mode will not have a supply above `u224` meaning this should not be a risk.

3.3.5 Informational - Extreme misconfiguration could cause loss of tokens or early unlocks

Severity: Informational

```
position.lockEnd = uint32(block.timestamp + cooldownDuration);
```

Since the logic is: `uint32(block.timestamp + cooldownDuration)`, if `block.timestamp + cooldownDuration == u32.max + 1` then the value will overflow to 0 which will cause a loss of tokens.

If `block.timestamp + cooldownDuration > u32.max + 1`, then the stakers will instantly unlock. This requires `cooldowDuration` to be set to years, which can only happen due to a severe mistake and only in the constructor.

Recommendation: Ensure that the value set in the constructor is less than or equal to `90 days` in seconds.

3.3.6 Informational - Since Mode is a OZ token, using `transfer` is safe

Severity: Informational

I recommend adding this as a comment to avoid getting false positives in future reviews as well as in bug bounties, for example:

```
mode.transferFrom(msg.sender, address(this), amount); /// @audit-ok Safe because Mode is ERC20-OZ
```

4 Appendix

4.1 Invariants

The following properties where tested with Medusa on the in scope contract:

- You never get more tokens that you deposited | Global Property.
- Unlocks cannot happen before time X | Stateful Fuzz.
- Unlocks can always happen after the right time | Stateful Fuzz.
- No lock has a greater claimable than the amount (overflow protection) | Global Property.
- You can always unlock provided you have a balance, you queue the unlock, then wait the correct amount of time | Doomsday Invariant.

The full run logs, settings, corpus and result can be viewed here:

- Initial Run with basic properties log.
- Final Run with Doomsday properties log.

4.2 Invariant Tests

Invariant tests on mitigation review code.

I do not expect any property to break as the changes where to settings and external ERC20s.

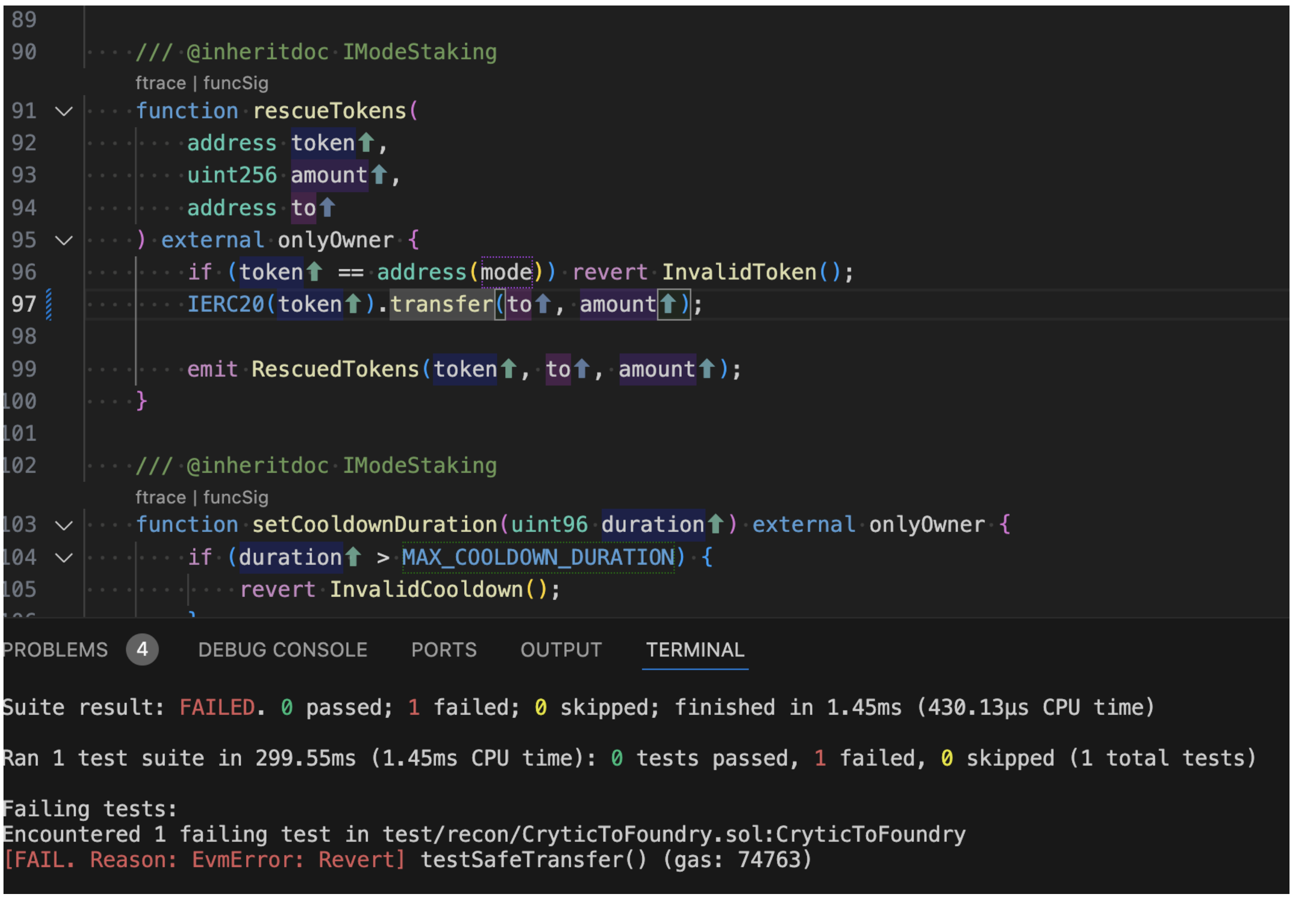
4.2.1 MED - Not using safeTransfer may cause sweep to fail for some tokens

Mitigated, see:

```
// SPDX-License-Identifier: GPL-2.0
pragma solidity ^0.8.0;

import {Test} from "forge-std/Test.sol";
import {console2} from "forge-std/console2.sol";
import {TargetFunctions} from "./TargetFunctions.sol";
import {FoundryAsserts} from "@chimera/FoundryAsserts.sol";

contract MockToken {
    function transfer(address, uint256) external {

    }
}

contract CryticToFoundry is Test, TargetFunctions, FoundryAsserts {
    function setUp() public {
        setup();
    }

    function testDemo() public {
        console2.log("initial bal", want.balanceOf(address(this)));
        lock(123123);
        cooldownAssets(123123);
        vm.warp(block.timestamp + theLock.cooldownDuration());
        console2.log("b4 unlock", want.balanceOf(address(this)));
        unlock();
        console2.log("final bal", want.balanceOf(address(this)));
    }
    function testSafeTransfer() public {
        MockToken fakeToken = new MockToken();
        theLock.rescueTokens(address(fakeToken), 123, address(this));
    }
}
```

This test passes only when using `safeTransfer`.

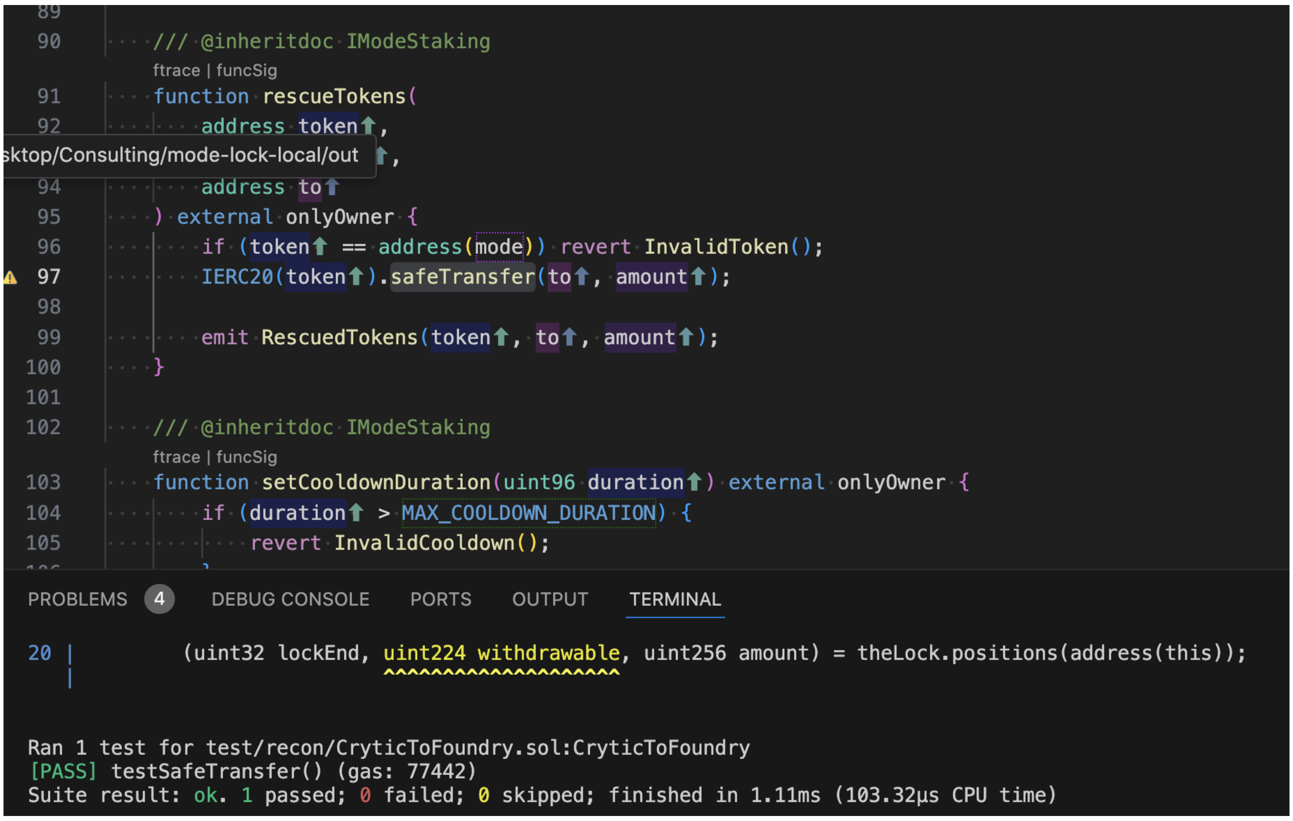
Without safeTransfer:



```
    /// @inheritdoc IModeStaking
    function rescueTokens(
        address token,
        uint256 amount,
        address to
    ) external onlyOwner {
        if (token == address(mode)) revert InvalidToken();
        IERC20(token).transfer(to, amount);

        emit RescuedTokens(token, to, amount);
    }

    /// @inheritdoc IModeStaking
    function setCooldownDuration(uint96 duration) external onlyOwner {
        if (duration > MAX_COOLDOWN_DURATION) {
            revert InvalidCooldown();
```

```
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 1.45ms (430.13µs CPU time)

Ran 1 test suite in 299.55ms (1.45ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/recon/CryticToFoundry.sol:CryticToFoundry
[FAIL. Reason: EvmError: Revert] testSafeTransfer() (gas: 74763)
```

With safeTransfer:



```
    /// @inheritdoc IModeStaking
    function rescueTokens(
        address token,
                      ,
        address to
    ) external onlyOwner {
        if (token == address(mode)) revert InvalidToken();
        IERC20(token).safeTransfer(to, amount);

        emit RescuedTokens(token, to, amount);
    }

    /// @inheritdoc IModeStaking
    function setCooldownDuration(uint96 duration) external onlyOwner {
        if (duration > MAX_COOLDOWN_DURATION) {
            revert InvalidCooldown();
```

```
20 |         (uint32 lockEnd, uint224 withdrawable, uint256 amount) = theLock.positions(address(this));
   |                          ^^^^^^^^^^^^^^^^^^^^^

Ran 1 test for test/recon/CryticToFoundry.sol:CryticToFoundry
[PASS] testSafeTransfer() (gas: 77442)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.11ms (103.32µs CPU time)
```

4.2.2 QA - Setter limits are not enforced in the constructor

Fixed by adding the same check as in `setCooldownDuration` in ModeStaking.sol#L31-L33:

```
if (_cooldown > MAX_COOLDOWN_DURATION) {
    revert InvalidCooldown();
}
```