Inverse Finance
Junior Tranche

Security Assessment &
Formal Verification
Dec 16th, 2025

ryzhak.com

Inverse Finance Junior Tranche - Security Assessment & Formal Verification

Project Overview

Project Summary

Project Name	Sense Finance Point Tokenization Vault
Language	Solidity
Codebase	https://github.com/quulab/inverse-finance-junior-tranche

Project Description

Inverse Finance Junior Tranche smart contracts allow users to mint DBR tokens in exchange for DOLA tokens and deposit DOLA tokens in exchange for shares from the `JDola` ERC4626 vault. The more users queue withdrawals from the `JDola` ERC4626 vault the more the withdraw delay which is calculated by the `LinearInterpolationDelayModel` contract. In case some markets are undercollateralized the `FiRMSlashingModule` contract is able to slash DOLA tokens from the `JDola` ERC4626 vault.

Audit Overview

Audit Summary

Delivery Date	Dec 16, 2025
Audit Methodology	Manual Review, Static Analysis, Formal Verification
Author(s)	Vladimir Ryzhak
Final Commit	6aeda21778ad6b85752153dcb5f8d5936a72d9bc
Formal Verification Report	FiRMSlashingModule, JDola, LinearInterpolationDelayModel, WithdrawalEscrow

Audit Scope

Filename	URL
`FiRMSlashingModule.sol`	Link
`JDola.sol`	Link
`LinearInterpolationDelayModel.sol`	Link
`WithdrawalEscrow.sol`	Link

Severity Matrix

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - results in a considerable risk that may jeopardize the protocol's overall integrity, impacting all or the majority of users.
- **Medium** - results in a non-critical risk for the protocol, impacting either all users or a specific subset, yet remaining unequivocally unacceptable.
- **Low** - losses incurred will be within acceptable limits, attack vectors can be fixed with relative ease.

Likelihood

- **High** - highly probable, presenting significant financial opportunities for exploitation by malicious actors.
- **Medium** - still relatively probable, although contingent upon certain conditions.
- **Low** - requires a unique set of conditions and presents a cost of execution that does not yield a favorable ratio of rewards for the individual involved.

Findings Summary

Severity	Discovered
Critical	-
High	-
Medium	4
Low	4
Total	8

Findings

ID	Title	Severity
M-01	Rounding down of `dolaReserve` leads to `JDola` DoS and deplete of `DBR` reserves	Medium
M-02	No economic incentive to mint `DBR` tokens	Medium
M-03	`MIN_ASSETS` and `MIN_SHARES` checks in `JDola` can be circumvented	Medium
M-04	`FiRMSlashingModule` can slash all assets in the `JDola` vault	Medium
L-01	Max withdraw delay in `queueWithdrawal()` can be circumvented	Low
L-02	Withdrawal fees should be rounded up	Low
L-03	Redundant `DBR` check in the `sweep()` method	Low
L-04	Missed events	Low

Certora Formal Verification

Overview

Contract	Report URL
FiRMSlashingModule.sol	Report URL
LinearInterpolationDelayModel.sol	Report URL
WithdrawalEscrow.sol	Report URL
jDola.sol	Report URL

Properties (FiRMSlashingModule.sol)

Property Description	Type	Passed
Methods are called by expected roles	High	✅
`slash()` updates storage as expected	Unit	✅
`slash()` revets when expected	Unit	✅
`allowMarket()` updates storage as expected	Unit	✅
`allowMarket()` revets when expected	Unit	✅
`disallowMarket()` updates storage as expected	Unit	✅
`disallowMarket()` revets when expected	Unit	✅
`setMaxCollateralValue()` updates storage as expected	Unit	✅
`setMaxCollateralValue()` revets when expected	Unit	✅
`setMinDebt()` updates storage as expected	Unit	✅
`setMinDebt()` revets when expected	Unit	✅
`setActivationDelay()` updates storage as expected	Unit	✅
`setActivationDelay()` revets when expected	Unit	✅
`setPendingGov()` updates storage as expected	Unit	✅
`setPendingGov()` reverts when expected	Unit	✅

Property Description	Type	Passed
`setGuardian()` updates storage as expected	Unit	✅
`setGuardian()` reverts when expected	Unit	✅
`acceptGov()` updates storage as expected	Unit	✅
`acceptGov()` reverts when expected	Unit	✅

Properties (LinearInterpolationDelayModel.sol)

Property Description	Type	Passed
Methods are called by expected roles	High	✅
`setMinDelay()` updates storage as expected	Unit	✅
`setMinDelay()` reverts when expected	Unit	✅
`setMaxDelay()` updates storage as expected	Unit	✅
`setMaxDelay()` reverts when expected	Unit	✅
`setMaxDelayThresholdBps()` updates storage as expected	Unit	✅
`setMaxDelayThresholdBps()` reverts when expected	Unit	✅
`setPendingGov()` updates storage as expected	Unit	✅
`setPendingGov()` reverts when expected	Unit	✅
`acceptGov()` updates storage as expected	Unit	✅
`acceptGov()` reverts when expected	Unit	✅

Properties (WithdrawalEscrow.sol)

Property Description	Type	Passed
Methods are called by expected roles	High	✅
`queueWithdrawal()` updates storage as expected	Unit	✅
`completeWithdraw()` updates storage as expected	Unit	✅
`completeWithdraw()` revets when expected	Unit	✅
`cancelWithdrawal()` updates storage as expected	Unit	✅

Property Description	Type	Passed
`cancelWithdrawal()` reverts when expected	Unit	✅
`initialize()` reverts when expected	Unit	✅
`setWithdrawDelayModel()` updates storage as expected	Unit	✅
`setWithdrawDelayModel()` reverts when expected	Unit	✅
`setWithdrawFee()` updates storage as expected	Unit	✅
`setWithdrawFee()` reverts when expected	Unit	✅
`setExitWindow()` updates storage as expected	Unit	✅
`setExitWindow()` reverts when expected	Unit	✅
`setGov()` updates storage as expected	Unit	✅
`setGov()` reverts when expected	Unit	✅
`acceptGov()` updates storage as expected	Unit	✅
`acceptGov()` reverts when expected	Unit	✅

Properties (jDola.sol)

Property Description	Type	Passed
K invariant always holds	High	✅
Methods are called by expected roles	High	✅
`initialize()` reverts when expected	Unit	✅
`buyDbr()` updates storage as expected	Unit	✅
`buyDbr()` reverts when expected	Unit	✅
`donate()` updates storage as expected	Unit	✅
`donate()` reverts when expected	Unit	✅
`slash()` updates storage as expected	Unit	✅
`slash()` reverts when expected	Unit	✅
`setDbrReserve()` updates storage as expected	Unit	✅
`setDbrReserve()` reverts when expected	Unit	✅

Property Description	Type	Passed
`setDolaReserve()` updates storage as expected	Unit	✅
`setDolaReserve()` reverts when expected	Unit	✅
`setMaxYearlyRewardBudget()` updates storage as expected	Unit	✅
`setMaxYearlyRewardBudget()` reverts when expected	Unit	✅
`setYearlyRewardBudget()` updates storage as expected	Unit	✅
`setYearlyRewardBudget()` reverts when expected	Unit	✅
`setSlashingModule()` updates storage as expected	Unit	✅
`setSlashingModule()` reverts when expected	Unit	✅
`setOperator()` updates storage as expected	Unit	✅
`setOperator()` reverts when expected	Unit	✅
`setPendingGov()` updates storage as expected	Unit	✅
`setPendingGov()` reverts when expected	Unit	✅
`acceptGov()` updates storage as expected	Unit	✅
`acceptGov()` reverts when expected	Unit	✅
`sweep()` updates storage as expected	Unit	✅
`sweep()` reverts when expected	Unit	✅

Findings

[M-01] Rounding down of `dolaReserve` leads to `JDola` DoS and deplete of `DBR` reserves

Description

There may be a case when `dolaReserve` rounds down to 0 leading to depleting of `DBR` reserves and DoS for most of the methods in the `JDola` contract.

Check the `setDbrReserve` method:

```
function setDbrReserve(uint _dbrReserve) external onlyGov updateReserves {
    require(_dbrReserve > 0, "dbr reserve cant be 0");
    require(_dbrReserve <= type(uint112).max, "dbr reserves can't exceed 2**112");
    dolaReserve = dolaReserve * _dbrReserve / dbrReserve;
    dbrReserve = _dbrReserve;
}
```

When `dbrReserve > dolaReserve * _dbrReserve` the `dolaReserve` rounds down to 0 which opens up an attack path for depleting `DBR` reserves and causing DoS for all of the methods that use the `updateReserves` modifier:

- `JDola.buyDbr()`
- `JDola.setDbrReserve()`
- `JDola.setDolaReserve()`
- `JDola.setMaxYearlyRewardBudget()`
- `JDola.setYearlyRewardBudget()`

Notice that ones the `dolaReserve` and `dbrReserve` are set to 0 they can no be recovered back to their working values.

Example:

1. Initially K invariant is highly skewed:

- `dbrReserve: 1_000_000_000_000 ether`
- `dolaReserve: 000000001 ether`

2. Government sets `DBR` reserve to 100 ether
3. Malicious user calls `buyDbr()` passing 0 `DOLA` tokens and minting 100 `DBR` tokens (full `DBR` reserves) for free

At this point both `dbrReserve` and `dolaReserve` equal to 0 and are unrecoverable because of division by 0.

Impact:

1. Attacker is able to mint `DBR` tokens for 0 `DOLA` tokens
2. DoS for most of the methods in the vault
3. Unrecoverable vault state when both `dbrReserve` and `dolaReserve` are 0

PoC

```
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import {ERC20} from "lib/solmate/src/tokens/ERC4626.sol";
import {FiRMSlashingModule} from "../../src/FiRMSlashingModule.sol";
import {LinearInterpolationDelayModel} from
"../../src/LinearInterpolationDelayModel.sol";
import {WithdrawalEscrow} from "../../src/WithdrawalEscrow.sol";
import {JDola} from "../../src/jDola.sol";
import {MockDBR} from "./MockDBR.sol";
import {MockERC20} from "./MockERC20.sol";
import {MockMarket} from "./MockMarket.sol";

contract ProtocolTest is Test {
    LinearInterpolationDelayModel linearInterpolationDelayModel;
    WithdrawalEscrow withdrawalEscrow;
    JDola jDola;
    FiRMSlashingModule firmSlashingModule;
    MockDBR dbrToken;
    MockERC20 dolaToken;
    MockMarket mockMarket;

    address gov = makeAddr("gov");
    address newGov = makeAddr("newGov");
    address operator = makeAddr("operator");
    address slasher = makeAddr("slasher");
    address user = makeAddr("user");
    address user2 = makeAddr("user2");
    address user3 = makeAddr("user3");

    function setUp() public {
        linearInterpolationDelayModel = new LinearInterpolationDelayModel(
            1 days, // _minDelay
            60 days, // _maxDelay
            10_000, // _maxDelayThresholdBps (sharpens the curve)
```

```
    gov
);

withdrawalEscrow = new WithdrawalEscrow(
    gov,
    address(linearInterpolationDelayModel)
);

dbrToken = new MockDBR("DBR", "DBR");
dolaToken = new MockERC20("DOLA", "DOLA");

jDola = new JDola(
    gov,
    operator,
    address(withdrawalEscrow),
    address(dbrToken),
    dolaToken,
    "DOLA_VAULT", // name
    "DOLA_VAULT" // symbol
);

firmSlashingModule = new FiRMSlashingModule(
    address(jDola),
    address(dbrToken),
    address(dolaToken),
    gov
);

vm.startPrank(gov);
jDola.initialize(
    1_000_000_000_000 ether, // _dbrReserve
    0.000000001 ether // _dolaReserve
);
jDola.setSlashingModule(address(firmSlashingModule), true);
withdrawalEscrow.initialize(address(jDola));
vm.stopPrank();

mockMarket = new MockMarket(address(dbrToken), address(dolaToken));

dolaToken.mint(user, 1000 ether);
dolaToken.mint(user2, 1000 ether);
dolaToken.mint(user3, 1000 ether);

vm.startPrank(user);
dolaToken.approve(address(jDola), type(uint256).max);
jDola.approve(address(withdrawalEscrow), type(uint256).max);
```

```
        vm.stopPrank();

        vm.startPrank(user2);
        dolaToken.approve(address(jDola), type(uint256).max);
        jDola.approve(address(withdrawalEscrow), type(uint256).max);
        vm.stopPrank();

        vm.startPrank(user3);
        dolaToken.approve(address(jDola), type(uint256).max);
        jDola.approve(address(withdrawalEscrow), type(uint256).max);
        vm.stopPrank();
    }

    /**
     * Scenario:
     * 1. Government sets DBR reserve to 100 ether
     * 2. User mints 100 DBR for 0 DOLA
     */
    function test_kInvariantBroken() public {
        vm.prank(gov);
        jDola.setDbrReserve(100 ether);

        console.log("===before===");
        debugReserves();
        debugBalance(address(dbrToken), user, "DBR balance (user)");

        // buyDbr
        vm.prank(user);
        jDola.buyDbr(
            0, // exactDolaIn
            100 ether, // exactDbrOut
            user
        );

        console.log("===after===");
        debugReserves();
        debugBalance(address(dbrToken), user, "DBR balance (user)");

        vm.prank(gov);
        vm.expectRevert();
        jDola.setDolaReserve(1 ether);

        // Output:
        // ===before===
        // ===reserves===
        // DOLA reserve:  0
```

```
        // DBR reserve :  10000000000000000000
        // DBR balance (user) 0e0
        // ===after===
        // ===reserves===
        // DOLA reserve:  0
        // DBR reserve :  0
        // DBR balance (user) 1e20
    }

    function debugBalance(address token, address target, string memory memo) public
{
        console.log("%s %e", memo, ERC20(token).balanceOf(target));
    }

    function debugReserves() public {
        (uint dolaReserve, uint dbrReserve) = jDola.getReserves();
        console.log("===reserves===");
        console.log("DOLA reserve: ", dolaReserve);
        console.log("DBR reserve : ", dbrReserve);
    }
}
```

Recommendation

Disallow setting `dbrReserve` and `dolaReserve` to 0

[M-02] No economic incentive to mint `DBR` tokens

Description

There's no economic incentive for a target user to call `JDola.buyDbr()` if `DOLA/DBR` quote is better on a secondary market thus causing the protocol to lose possible donations.

Check the JDola.buyDbr() method:

```
function buyDbr(uint exactDolaIn, uint exactDbrOut, address to) external
updateReserves {
    require(to != address(0), "Zero address");
    uint K = dolaReserve * dbrReserve;
    dolaReserve += exactDolaIn;
    dbrReserve -= exactDbrOut;
    require(dolaReserve * dbrReserve >= K, "Invariant");
    donate(exactDolaIn);
    DBR.mint(to, exactDbrOut);
```

```
        emit Buy(msg.sender, to, exactDolaIn, exactDbrOut);
    }
```

Basically it swaps DOLA tokens for DBR.

The issue is that if `DOLA/DBR` quote is better on a secondary market then there's no economic sense for a user to mint `DBR` via `buyDbr()`.

Consider an example:

1. `DOLA` and `DBR` are traded 1 to 1 in the `JDola` contract, initial reserves are:

- `dbrReserve: 100`
- `dolaReserve: 100`

2. In a uniswap market the `DOLA/DBR` quote is 2, so for 1 `DOLA` token user can get 2 `DBR` tokens there.

It doesn't make sense for a user to buy 1 `DBR` for 1 `DOLA` in the `JDola` vault while he can buy 1 `DBR` for 0.5 `DOLA` in a secondary uniswap market.

The thing is that the protocol team will have to constantly update reserves on any radical `DOLA/DBR` price movement via the following methods:

- `JDola.setDbrReserve()`
- `JDola.setDolaReserve()`
- `JDola.setYearlyRewardBudget()`

And while the protocol team is updating the reserves to their market values the protocol itself loses possible donations.

Recommendation

Rethink the design, allow arbitragers to maintain market reserves.

[M-03] `MIN_ASSETS` and `MIN_SHARES` checks in `JDola` can be circumvented

Description

It is possible to withdraw all assets from the JDola vault circumventing the checks for min assets and min shares.

Check the `JDola.beforeWithdraw()` method:

```
function beforeWithdraw(uint256 assets, uint256 shares) internal override {
    require(msg.sender == withdrawEscrow, "Only withdraw escrow");
    require(totalAssets() >= assets + MIN_ASSETS || assets == totalAssets(), "Assets
below MIN_ASSETS");
    require(totalSupply - shares >= MIN_SHARES || shares == totalSupply, "Shares
below MIN_SHARES");
}
```

While the code suggests that users can not withdraw assets and shares less than there minimal constant values it is still possible to do so if withdraw amount equals to total supply of assets or shares.

Example 1 (expected error):

1. User deposits 1 `DOLA`
2. User queues 0.5 `DOLA` for withdrawal
3. On completing withdrawal tx reverts with "Assets below MIN_ASSETS"

Example 2 (total supply is withdrawn without min assets checks):

1. User deposits 1 `DOLA`
2. User queues 1 `DOLA` for withdrawal
3. Complete withdrawal succeeds leading to `assets < MIN_ASSETS`

Circumventing min assets / shares invariant opens up an attack vector for vault inflation attack.

PoC

```
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import {ERC20} from "lib/solmate/src/tokens/ERC4626.sol";
import {FiRMSlashingModule} from "../../src/FiRMSlashingModule.sol";
import {LinearInterpolationDelayModel} from
"../../src/LinearInterpolationDelayModel.sol";
import {WithdrawalEscrow} from "../../src/WithdrawalEscrow.sol";
import {JDola} from "../../src/jDola.sol";
import {MockDBR} from "./MockDBR.sol";
import {MockERC20} from "./MockERC20.sol";
import {MockMarket} from "./MockMarket.sol";

contract ProtocolTest is Test {
    LinearInterpolationDelayModel linearInterpolationDelayModel;
    WithdrawalEscrow withdrawalEscrow;
    JDola jDola;
```

```
FiRMSlashingModule firmSlashingModule;
MockDBR dbrToken;
MockERC20 dolaToken;
MockMarket mockMarket;

address gov = makeAddr("gov");
address newGov = makeAddr("newGov");
address operator = makeAddr("operator");
address slasher = makeAddr("slasher");
address user = makeAddr("user");
address user2 = makeAddr("user2");
address user3 = makeAddr("user3");

function setUp() public {
    linearInterpolationDelayModel = new LinearInterpolationDelayModel(
        1 days, // _minDelay
        60 days, // _maxDelay
        10_000, // _maxDelayThresholdBps (sharpens the curve)
        gov
    );

    withdrawalEscrow = new WithdrawalEscrow(
        gov,
        address(linearInterpolationDelayModel)
    );

    dbrToken = new MockDBR("DBR", "DBR");
    dolaToken = new MockERC20("DOLA", "DOLA");

    jDola = new JDola(
        gov,
        operator,
        address(withdrawalEscrow),
        address(dbrToken),
        dolaToken,
        "DOLA_VAULT", // name
        "DOLA_VAULT" // symbol
    );

    firmSlashingModule = new FiRMSlashingModule(
        address(jDola),
        address(dbrToken),
        address(dolaToken),
        gov
    );
```

```
        vm.startPrank(gov);
        jDola.initialize(
            1_000_000_000_000 ether, // _dbrReserve
            0.000000001 ether // _dolaReserve
        );
        jDola.setSlashingModule(address(firmSlashingModule), true);
        withdrawalEscrow.initialize(address(jDola));
        vm.stopPrank();

        mockMarket = new MockMarket(address(dbrToken), address(dolaToken));

        dolaToken.mint(user, 1000 ether);
        dolaToken.mint(user2, 1000 ether);
        dolaToken.mint(user3, 1000 ether);

        vm.startPrank(user);
        dolaToken.approve(address(jDola), type(uint256).max);
        jDola.approve(address(withdrawalEscrow), type(uint256).max);
        vm.stopPrank();

        vm.startPrank(user2);
        dolaToken.approve(address(jDola), type(uint256).max);
        jDola.approve(address(withdrawalEscrow), type(uint256).max);
        vm.stopPrank();

        vm.startPrank(user3);
        dolaToken.approve(address(jDola), type(uint256).max);
        jDola.approve(address(withdrawalEscrow), type(uint256).max);
        vm.stopPrank();
    }

    /**
     * Scenario:
     * 1. User deposits 1 DOLA
     * 2. User queues 0.5 DOLA for withdrawal
     * 3. On completing withdrawal tx reverts with "Assets below MIN_ASSETS"
     */
    function test_exceedMinAssets_fail() public {
        skip(7 days);

        vm.prank(user);
        jDola.deposit(1 ether, user);

        vm.prank(user);
        withdrawalEscrow.queueWithdrawal(0.5 ether, type(uint256).max);
```

```
        skip(31 days);

        vm.prank(user);
        vm.expectRevert("Assets below MIN_ASSETS");
        withdrawalEscrow.completeWithdraw();
    }

    /**
     * Scenario:
     * 1. User deposits 1 DOLA
     * 2. User queues 1 DOLA for withdrawal
     * 3. Complete withdrawal succeeds leading to assets < MIN_ASSETS
     */
    function test_exceedMinAssets_success() public {
        skip(7 days);

        vm.prank(user);
        jDola.deposit(1 ether, user);

        vm.prank(user);
        withdrawalEscrow.queueWithdrawal(1 ether, type(uint256).max);

        skip(60 days);

        vm.prank(user);
        withdrawalEscrow.completeWithdraw();
    }
}
```

Recommendation

```
diff --git a/InverseFinance__JuniorDola/src/jDola.sol
b/InverseFinance__JuniorDola/src/jDola.sol
index ab7ce2c..b0babdf 100644
--- a/InverseFinance__JuniorDola/src/jDola.sol
+++ b/InverseFinance__JuniorDola/src/jDola.sol
@@ -110,8 +110,8 @@ contract JDola is ERC4626 {
      */
     function beforeWithdraw(uint256 assets, uint256 shares) internal override {
         require(msg.sender == withdrawEscrow, "Only withdraw escrow");
-        require(totalAssets() >= assets + MIN_ASSETS || assets == totalAssets(),
"Assets below MIN_ASSETS");
-        require(totalSupply - shares >= MIN_SHARES || shares == totalSupply,
"Shares below MIN_SHARES");
+        require(totalAssets() >= assets + MIN_ASSETS, "Assets below MIN_ASSETS");
```

```
+         require(totalSupply - shares >= MIN_SHARES, "Shares below MIN_SHARES");
      }
```

[M-04] `FiRMSlashingModule` can slash all assets in the `JDola` vault

Description

`FiRMSlashingModule` is able to slash `JDola` vault assets to cover potential bad debt from other markets.

It may lead to the following scenario:

1. User deposits 100 `DOLA` tokens
2. `FiRMSlashingModule` slashes the `JDola` vault and transfers those 100 `DOLA` tokens to cover bad debt
3. User is unable to withdraw his deposit

Overall this leads to DoS for all of the withdrawal methods.

Recommendation

Rethink the design

[L-01] Max withdraw delay in `queueWithdrawal()` can be circumvented

Description

Check this `require` statement which checks whether max withdraw delay is exceeded.

The issue is that the maxWithdrawDelay parameter is user contolled which means that user can always pass `type(uint256).max` for the `maxWithdrawDelay` parameter thus making that `require` statement useless.

Recommendation

The purpose of the `maxWithdrawDelay` is not clear, either remove it either use a state variable set in the constructor.

[L-02] Withdrawal fees should be rounded up

Description

Check how the withdrawal fee is calculated.

Right now the withdrawal fee is rounded down causing the protocol to lose funds on each withdrawal.

Recommendation

Round up the withdrawal fee.

[L-03] Redundant `DBR` check in the `sweep()` method

Description

Check the `JDola.sweep()` method:

```
function sweep(address token, uint amount, address to) public onlyGov {
    require(address(DBR) != token, "Not authorized");
    require(address(asset) != token, "Not authorized");
    SafeTransferLib.safeTransfer(ERC20(token), to, amount);
}
```

The `sweep()` method doesn't allow `DBR` or `DOLA` tokens to be transfered from the vault.

The issue is that `DBR` tokens don't reside in the vault thus, if `DBR` tokens are sent mistakenly to the vault they will be stuck in the contract.

Recommendation

```
diff --git a/InverseFinance__JuniorDola/src/jDola.sol
b/InverseFinance__JuniorDola/src/jDola.sol
index a59366e..427d857 100644
--- a/InverseFinance__JuniorDola/src/jDola.sol
+++ b/InverseFinance__JuniorDola/src/jDola.sol
     function sweep(address token, uint amount, address to) public onlyGov {
-        require(address(DBR) != token, "Not authorized");
         require(address(asset) != token, "Not authorized");
```

```
        SafeTransferLib.safeTransfer(ERC20(token), to, amount);
    }
```

[L-04] Missed events

Description

There're missed events for the following state changing methods:

- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/FiRMSlashingModule.sol#L116
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/FiRMSlashingModule.sol#L126
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/FiRMSlashingModule.sol#L134
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/LinearInterpolationDelayModel.sol#L55
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/WithdrawalEscrow.sol#L136
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/WithdrawalEscrow.sol#L141
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/WithdrawalEscrow.sol#L145
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/WithdrawalEscrow.sol#L150
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/WithdrawalEscrow.sol#L156
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/WithdrawalEscrow.sol#L160

- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/jDola.sol#L118
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/jDola.sol#L165
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/jDola.sol#L208
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/jDola.sol#L218
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/jDola.sol#L265
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/jDola.sol#L273
- https://github.com/quulab/inverse-finance-junior-tranche/blob/6aeda21778ad6b85752153dcb5f8d5936a72d9bc/InverseFinance__JuniorDola/src/jDola.sol#L280

Recommendation

Emit events in state changing methods.

Disclaimer

This security review should not be interpreted as providing absolute assurance against potential hacks or exploits. Smart contracts represent a novel technological advancement, inherently associated with various known and unknown risks. The protocol for which this report is prepared indemnifies the author from any liability concerning potential misbehavior, bugs, or exploits affecting the audited code throughout the entirety of the project's life cycle. It is also crucial to recognize that any modifications made to the audited code, including remedial measures for the issues outlined in this report, may inadvertently introduce new complications and necessitate further auditing.

About the author

Prepared by ryzhak.com.

Perform a comprehensive smart contract audit using advanced formal verification tools that identify even the most elusive and intricate bugs within smart contracts and mathematically prove the absence of security issues. All formal verification properties will be integrated into your standard deployment CI/CD pipelines, which helps reduce the number of bugs in already audited code, thereby lowering costs for future security assessments.