PuppyRaffle Audit Report

Version 1.0

Jack

August 2, 2025

ThunderLoan Audit Report

Jack

July 30, 2025

Prepared by: Jack Lead Auditors: - Jack

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-1] Incorrect Fee Application in `ThunderLoan.deposit`
    * [H-2] Incorrect Denomination for Flash Loan Fee Calculation (Logic Bug + Funds Risk)
    * [H-3] Missing Flash-Loan Guard in deposit Allows Balance Manipulation (Missing Check + Funds Risk)
    * [H-4] Storage Collision Due to Variable Reordering in Upgrade (Storage Layout Mismatch + Protocol State Corruption)
  - Medium

    * [M-1] Dependency on USDC's Centralized Blacklist Freezes Protocol (Centralized Blacklist Dependency + Protocol Denial of Service)
    * [M-2] Reliance on Single Pool Spot Price in getPriceInWeth (Price Manipulation Risk + Inaccurate Oracle Data)
  - Low
    * [L-1] Unused Public Functions in ThunderLoan.sol (Public Interface Bloat + Minor Attack Surface Increase)
    * [L-2] Missing Event Emission in updateFlashLoanFee (Poor Transparency + Incomplete Audit Trail)
    * [L-3] Unprotected External Initializer in ThunderLoan::initialize (Missing Proxy Guard + Front-Running Risk)
  - Informational
    * [I-1] Redundant Storage Reads of s_exchangeRate in updateExchangeRate (Storage Access Inefficiency + Elevated Gas Costs)
    * [I-2] Unvalidated Assignment of s_poolFactory in __Oracle_init_unchained (Missing Input Validation + Potential Misconfiguration)
    * [I-3] Inconsistent Parameter Naming in initialize (Naming Inconsistency + Potential Misuse)
    * [I-4] Unused and Misspelled Custom Error ThunderLoan__ExhangeRateCanOnlyIncrease (Dead Code + Naming Inconsistency)
    * [I-5] s_feePrecision Not Declared as Constant (Storage Inefficiency + Accidental Modification Risk)

Protocol Summary

The `ThunderLoan` protocol is meant to do the following:

1. Give users a way to create flash loans
2. Give liquidity providers a way to earn money off their capital

Liquidity providers can `deposit` assets into `ThunderLoan` and be given `AssetTokens` in return. These `AssetTokens` gain interest over time depending on how often people take out flash loans!

What is a flash loan?

A flash loan is a loan that exists for exactly 1 transaction. A user can borrow any amount of assets from the protocol as long as they pay it back in the same transaction. If they don't pay it back, the transaction reverts and the loan is cancelled.

Users additionally have to pay a small fee to the protocol depending on how much money they borrow. To calculate the fee, we're using the famous on-chain TSwap price oracle.

We are planning to upgrade from the current `ThunderLoan` contract to the `ThunderLoanUpgraded` contract. Please include this upgrade in scope of a security review.

Disclaimer

The Jack team 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
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

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

Audit Details

Scope

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- In Scope:

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
```

```
4 |   #-- ITSwapPool.sol
5 |   #-- IThunderLoan.sol
6 #-- protocol
7 |   #-- AssetToken.sol
8 |   #-- OracleUpgradeable.sol
9 |   #-- ThunderLoan.sol
10 #-- upgradedProtocol
11     #-- ThunderLoanUpgraded.sol
```

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC (Proxy)
  - DAI
  - LINK
  - WETH

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.

Executive Summary

- 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.

Issues found

Severity	Number of issues found
High	4

Severity	Number of issues found
Medium	2
Low	3
Info	5
Total	14

Findings

High

[H-1] Incorrect Fee Application in `ThunderLoan.deposit`

Description:
In the `ThunderLoan.deposit` function, the contract calls `ThunderLoan.getCalculatedFee(token, amount)` and immediately invokes `ThunderLoan.assetToken.updateExchangeRate(calculatedFee)`. Depositing assets should not generate any fee revenue—fees are only realized when flash loans are repaid. By applying `updateExchangeRate` on every deposit, the exchange rate is artificially inflated, improperly attributing fee income to new depositors.

Impact:
• New depositors receive inflated interest-bearing tokens at a higher exchange rate, diluting the share of existing depositors.
• The exchange rate no longer correctly reflects only flash-loan fees, breaking `ThunderLoan.redeem` calculations and unfairly redistributing protocol revenue.
• Financial logic inconsistency can lead to unexpected losses for liquidity providers and undermine protocol integrity.

Proof of Concept:
The following test code (as shown in the screenshot) fails unless the two lines computing and applying fees in `ThunderLoan.deposit` are commented out. Only then does the redeem amount equal the original principal plus the flash-loan fee:

```
function testRedeemAfterLoan() public setAllowedToken hasDeposits {
    uint256 amountToBorrow = AMOUNT * 10;
    uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
        amountToBorrow);

    vm.startPrank(user);
```

```
6       tokenA.mint(address(mockFlashLoanReceiver), calculatedFee); // fee
7       thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
            amountToBorrow, "");
8       vm.stopPrank();
9
10      uint256 amountToRedeem = type(uint256).max;
11      vm.startPrank(liquidityProvider);
12      thunderLoan.redeem(tokenA, amountToRedeem);
13  }
```

With the following lines removed from `ThunderLoan.deposit`, the test passes and `ThunderLoan.redeem` returns exactly `AMOUNT + calculatedFee`:

```
1  -     uint256 calculatedFee = ThunderLoan.getCalculatedFee(token, amount
       );
2  -     ThunderLoan.assetToken.updateExchangeRate(calculatedFee);
```

Recommended Mitigation:

• Remove fee calculation and exchange-rate update from the `ThunderLoan.deposit` function.

• Centralize fee accrual to the flash-loan repayment logic in `ThunderLoan`: only invoke `ThunderLoan.assetToken.updateExchangeRate` when loans are repaid (e.g., within the `ThunderLoan.flashloan` callback or a dedicated repayment function).

• Add unit tests verifying that pure deposits do not alter the exchange rate.

• Document in code comments that fees are only realized on loan repayment.

[H-2] Incorrect Denomination for Flash Loan Fee Calculation (Logic Bug + Funds Risk)

Description:

In `ThunderLoan::getCalculatedFee` and `ThunderLoan::flashloan`, the fee is computed by converting the borrowed token amount into WETH value, then applying the fee rate, yielding a fee denominated in WETH:

```
1  function getCalculatedFee(IERC20 token, uint256 amount) public view
       returns (uint256 fee) {
2      uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(
           token))) / s_feePrecision;
3      fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
4  }
```

However, in `ThunderLoan::flashloan`, this fee (in WETH units) is compared against the increase in the token balance of the pool:

```
1  uint256 startingBalance = token.balanceOf(address(assetToken));
2  ...
3  uint256 fee = getCalculatedFee(token, amount);
```

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

The contract thus expects borrowers to repay `amount + fee` *in the borrowed token*, but `fee` represents an amount of WETH, not of the token. This mismatch allows attackers to underpay or misalign repayments.

Impact:

- Borrowers can satisfy the repayment check by returning only `amount + fee_in_WETH` *worth* of tokens at manipulated prices, effectively paying less than the intended fee or even pocketing part of the loan.
- If the token's price fluctuates between the flashloan and repayment, the pool may accept significantly less collateral than required, leading to loss of funds.
- An attacker could flashloan a token, deliberately manipulate its on-chain price oracle, and repay with tokens valued below the required WETH-denominated fee, draining the pool's reserves.

Recommended Mitigation:

1. Compute the fee directly in the borrowed token's units:

```
solidity function getCalculatedFee(IERC20 token, uint256 amount)
public view returns (uint256 fee){ //fee = amount * s_flashLoanFee
 / s_feePrecision fee = (amount * s_flashLoanFee)/ s_feePrecision; }
```

2. Remove any dependence on `getPriceInWeth` in fee logic, or if cross-currency fees are required, convert repayments to WETH within the pool or accept WETH directly.
3. Add unit tests and forked-chain scenarios to validate fee collection under varying price conditions and prevent manipulation.

[H-3] Missing Flash-Loan Guard in deposit Allows Balance Manipulation (Missing Check + Funds Risk)

Description:

In `ThunderLoan::deposit`, there is no check against `s_currentlyFlashLoaning[token]`, so deposits made during a flash-loan callback will increase the pool's reported balance:

```
function deposit(IERC20 token, uint256 amount) external {...
    AssetToken assetToken = s_tokenToAssetToken[token];
    //...
    emit Deposit(msg.sender, token, amount);
    assetToken.mint(msg.sender, mintAmount);
```

```
6       uint256 calculatedFee = getCalculatedFee(token, amount);
7       assetToken.updateExchangeRate(calculatedFee);
8       token.safeTransferFrom(msg.sender, address(assetToken), amount);
9   }
```

Meanwhile, in `ThunderLoan::flashloan`, after the borrower callback, the contract only verifies:

```
1 uint256 endingBalance = token.balanceOf(address(assetToken));
2 if (endingBalance < startingBalance + fee) {
3       revert ThunderLoan__NotPaidBack(startingBalance + fee,
            endingBalance);
4 }
```

An attacker can exploit this by calling `ThunderLoan::deposit` during their `executeOperation` callback, artificially inflating `endingBalance` so the repayment check passes without actually returning the borrowed principal.

Impact:

- Flash-loan borrowers can avoid repaying the loan principal.
- Attackers can mint AssetToken shares with zero economic cost (since they never truly repay), then later redeem those shares to drain funds from the pool.
- The protocol's reserves and user collateral are susceptible to theft.

Proof of Concept:

1. Attacker triggers `ThunderLoan::flashloan(token, amount, params)`.
2. In `IFlashLoanReceiver.executeOperation`, the attacker:

a. Calls `ThunderLoan::deposit(token, amount)`. This mints shares and transfers `amount` tokens into the pool.

b. Does *not* transfer any tokens back for the flash-loan repayment.

3. Control returns to `ThunderLoan::flashloan`. The `endingBalance` now equals `startingBalance + amount` (from deposit), so `endingBalance >=startingBalance + fee` and no revert occurs.
4. Flash-loan completes with no actual repayment of `amount`.
5. Attacker calls `AssetToken.redeem(...)` to withdraw `amount + accrued yield`, draining funds.

Recommended Mitigation:

- In `ThunderLoan::deposit`, block deposits during an active flash-loan:

```
1 function deposit(IERC20 token, uint256 amount) external revertIfZero(
      amount) revertIfNotAllowedToken(token) {
2       if (s_currentlyFlashLoaning[token]) {
3           revert ThunderLoan__FlashLoanInProgress();
4       }
```

```
    //...existing logic...
}
```

- Alternatively, add `revertIfCurrentlyFlashLoaning(token)` modifier to `deposit`.
- Ensure any state-changing functions that affect pool balance are disabled while `s_currentlyFlashLoaning[token]` is **`true`**.
- Add unit and forked-chain tests simulating deposits during flash-loan callbacks to verify the guard.

[H-4] Storage Collision Due to Variable Reordering in Upgrade (Storage Layout Mismatch + Protocol State Corruption)

Description:
The original `ThunderLoan.sol` defines its fee-related storage slots in this order:

```
uint256 private s_feePrecision;     // slot 0
uint256 private s_flashLoanFee;     // slot 1
```

In the upgraded `ThunderLoanUpgraded.sol`, the developer replaces `s_feePrecision` with a compile-time constant and moves `s_flashLoanFee` to the first declared slot:

```
uint256 private s_flashLoanFee;     // now declared at slot 0
uint256 public constant FEE_PRECISION = 1e18; // constant, no storage slot
```

Because constants do not consume storage slots, `s_flashLoanFee` in the upgraded contract will occupy slot 0, colliding with the original `s_feePrecision` slot. The legacy `s_flashLoanFee` value stored at slot 1 becomes orphaned.

Impact:
- Fee precision and flash-loan fee values are swapped or lost after upgrade, breaking all fee calculations.
- Borrowers may be over- or under-charged, enabling unintended free or excessively expensive flash loans.
- Inconsistent state can lead to revenue loss or denial-of-service if the contract reverts due to zero or malformed fee parameters.
- Permanent misconfiguration: since the storage layout is corrupted, no on-chain migration can restore correct values without another breaking upgrade.

Proof of Concept:
Test the following code.

testUpgradeBreaks

```
    function testUpgradeBreaks() public {
          uint256 feeBeforeUpgrade = thunderLoan.getFee();
          vm.startPrank(thunderLoan.owner());
          ThunderLoanUpgraded thunderLoanUpgraded = new
              ThunderLoanUpgraded();
          thunderLoan.upgradeToAndCall(address(thunderLoanUpgraded), ""
              );
          uint256 feeAfterUpgrade = thunderLoan.getFee();
          vm.stopPrank();
          console2.log("feeBeforeUpgrade", feeBeforeUpgrade);
          console2.log("feeAfterUpgrade", feeAfterUpgrade);
          assert(feeBeforeUpgrade != feeAfterUpgrade);
      }
```

You will find the test success means that feeBeforeUpgrade != feeAfterUpgrade.

Recommended Mitigation:

1. **Preserve Storage Layout:** Always keep the same ordering and types of state variables in upgrades. Do not remove or reorder existing storage declarations.

2. **Use Storage Gaps:** Reserve empty slots for future variables: `solidity uint256 private s_feePrecision; uint256 private s_flashLoanFee; uint256[50] private __gap;` 3. **Introduce New Variables After Gap:** Add new constants or variables after the reserved space: “‘solidity uint256 private s_feePrecision; uint256 private s_flashLoanFee; uint256[50] private __gap;

uint256 public constant FEE_PRECISION = 1e18; // safe, no slot used “4. **Immutable Over Constant:** If you need a deploy-time settable precision, prefer immutableinstead ofconstantso it still occupies a unique slot after the gap. 5. **Run Storage Layout Checks:** Use tools like OpenZeppelin' sopenzeppelin-upgrades‘ plugin to compare pre- and post-upgrade storage layouts and detect collisions before deployment.

Medium

[M-1] Dependency on USDC’s Centralized Blacklist Freezes Protocol (Centralized Blacklist Dependency + Protocol Denial of Service)

Description:

The function `AssetToken::transferUnderlyingTo(address to, uint256 amount)` invokes `i_underlying.safeTransfer(to, amount)` on the USDC token. USDC implements an on-chain blacklist controlled by its issuer. If the protocol’s contract address is added to USDC’s

blacklist, any call to `safeTransfer` will revert, making this function unusable.

Impact:
- All USDC transfers through this function will revert once the contract is blacklisted.
- Users will be unable to withdraw or repay USDC-denominated loans.
- Loan liquidations and other protocol operations depending on USDC transfer will be blocked.
- Protocol funds may be permanently locked, resulting in a denial of service and potential financial losses for users.

Proof of Concept:
1. Deploy a mock USDC token contract with a blacklist mapping and `safeTransfer` behavior identical to USDC's.
2. Deploy the loan protocol contract pointing `i_underlying` at the mock USDC.
3. As the token owner, call `blacklistAddress(protocolAddress)`.
4. Invoke `AssetToken::transferUnderlyingTo(user, 1e6)` via `onlyThunderLoan`.
5. Observe that `safeTransfer` reverts due to the blacklist, freezing withdrawals.

Recommended Mitigation:
- Use a non-permissioned stablecoin (e.g., DAI) that does not support centralized blacklisting.
- Implement an emergency withdrawal mechanism that can route around USDC (e.g., via a whitelisted multisig or a fallback token).
- At deployment, detect and reject tokens with blacklist functionality by querying known methods (`isBlacklisted`) or consulting an on-chain registry.
- Introduce a governance-controlled upgrade path or rescuer role to recover funds if the underlying token is blacklisted in the future.

[M-2] Reliance on Single Pool Spot Price in getPriceInWeth (Price Manipulation Risk + Inaccurate Oracle Data)

Description:
The function

```
1 function getPriceInWeth(address token) public view returns (uint256) {
2     address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token
          );
3     return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
4 }
```

fetches the price of a token in WETH directly from a single on-chain liquidity pool. There is no mechanism to guard against short-term price swings or manipulation, nor any fallback or aggregation from multiple sources.

Impact:
- An attacker can briefly distort the pool's spot price (for example via flash loans) to force the oracle to report a malicious price.
- Any dependent operations (borrowing, liquidation, margin calculations, collateral valuation) will use the manipulated price, leading to incorrect liquidations, fund loss, or unfair gains.
- Because the call is purely on-chain and immediate, there is no time-weighted or historical smoothing to resist short-lived price attacks.

Proof of Concept:
1. Attacker obtains a flash loan of the target token.
2. Attacker swaps a large amount of token for WETH (or vice versa) in the pool, pushing the spot price far from the true market price.
3. Within the same transaction, attacker calls `getPriceInWeth(token)` to read the distorted price.
4. Attacker uses the manipulated price to under-collateralize a borrow or trigger an unfavorable liquidation in the protocol.
5. Attacker unwinds the flash loan, repays it, and pockets the difference.

Recommended Mitigation:
- Integrate a time-weighted average price (TWAP) mechanism on the pool (e.g., accumulate price observations and compute an average over multiple blocks).
- Aggregate prices from multiple independent on-chain sources (e.g., other pools, DEXes) or external oracles (e.g., Chainlink).
- Implement sanity checks or bounds (e.g., reject prices that deviate more than X% from a trusted benchmark).
- Use forked-chain tests to simulate flash-loan manipulation scenarios and verify the oracle's resistance under attack conditions.

Low

[L-1] Unused Public Functions in ThunderLoan.sol (Public Interface Bloat + Minor Attack Surface Increase)

Description:
ThunderLoan.sol exposes several public functions that are not referenced or invoked by any other part of the contract's internal logic:

- `repay(IERC20 token, uint256 amount)`

- `isAllowedToken(IERC20 token)public view returns (bool)`
- `getAssetFromToken(IERC20 token)public view returns (AssetToken)`
- `isCurrentlyFlashLoaning(IERC20 token)public view returns (bool)`

Because these functions serve no internal purpose—no other contract code calls them—they exist solely to broaden the contract's external API surface.

Impact:
- Increases bytecode size and gas cost for deployment and verification.
- Expands the external attack surface (more entry points to analyze for misuse or unexpected behavior).
- May confuse integrators or auditors who expect every public function to serve a core protocol flow.

Proof of Concept:
1. Inspect the contract's internal call graph—none of the four functions above appear as internal targets.
2. Deploy the contract and verify via a static analysis tool (e.g. Slither) that these functions are never reached by any other function.

Recommended Mitigation:
- Remove any functions that are not needed by external integrators or downstream consumers.
- If read-only accessors are required for off-chain tooling (e.g. `isAllowedToken`, `getAssetFromToken`, `isCurrentlyFlashLoaning`), mark them as `external` to save a small amount of gas on call.
- If `repay` is not part of the intended public API, either convert it to `internal` or remove it entirely.
- Clean up the contract to expose only the minimal set of public functions necessary for its primary flash-loan functionality.

[L-2] Missing Event Emission in updateFlashLoanFee (Poor Transparency + Incomplete Audit Trail)

Description:
The `updateFlashLoanFee` function allows the contract owner to change the flash loan fee but does not emit any event to record this critical state change:

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

Without an event, off-chain services, indexers, or auditors have no easy way to detect when the fee was updated or what the previous and new values were.

Impact:
- Loss of transparency: external observers cannot track fee updates in the transaction logs.
- Reduced auditability: no on-chain record of when and by whom the fee was changed.
- Potential governance confusion: users cannot verify that the function executed as intended or review historical fee changes.

Proof of Concept:
1. Owner calls `updateFlashLoanFee(500)`.
2. No `FlashLoanFeeUpdated` event is emitted.
3. On-chain explorers and monitoring tools have no record of the change beyond a generic transaction entry.

Recommended Mitigation:
Introduce and emit a dedicated event to capture both the old and new fee:

```
contract ThunderLoan {
+    event FlashLoanFeeUpdated(uint256 indexed oldFee, uint256 indexed
    newFee);

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

- Define `FlashLoanFeeUpdated(uint256 oldFee, uint256 newFee)`.
- Capture the previous fee before assignment.
- Emit the event after updating.

This ensures clear, tamper-resistant on-chain records of every fee adjustment.

[L-3] Unprotected External Initializer in ThunderLoan::initialize (Missing Proxy Guard + Front-Running Risk)

Description:

In `src/protocol/ThunderLoan.sol`, the function

```
function initialize(address tswapAddress) external initializer {
    __Ownable_init(msg.sender);
    __UUPSUpgradeable_init();
    __Oracle_init(tswapAddress);
    s_feePrecision = 1e18;
    s_flashLoanFee = 3e15; // 0.3% ETH fee
}
```

is marked only with OpenZeppelin's `initializer` modifier. Until the initializer is called, any externally owned account can race to invoke it, seize ownership (`Ownable`), configure the oracle to a malicious address, and set arbitrary fee parameters.

Impact:

- Unauthorized ownership takeover: the first caller becomes owner and can upgrade, drain, or reconfigure the contract.
- Oracle misconfiguration: attacker can point `__Oracle_init` at a malicious price source.
- Fee manipulation: attacker can set `s_flashLoanFee` and `s_feePrecision` to values that harm legitimate users or protocols relying on this contract.

Proof of Concept:

1. Deploy the proxy pointing to the ThunderLoan implementation.
2. Before the legitimate deployment script calls `initialize`, an attacker calls `initialize(attackerOracle)`.
3. The attacker's EOA becomes owner, and `s_oracle`, `s_flashLoanFee`, and `s_feePrecision` are set to attacker-controlled values.
4. Any subsequent calls by the rightful deployer to `initialize` will revert due to the `initializer` guard, locking in the attacker's configuration.

Recommended Mitigation:

1. Disable direct initialization on the implementation contract by adding in the constructor:
`solidity constructor(){ _disableInitializers(); }`
2. Restrict `initialize` to proxy-only calls and validate inputs:
`solidity function initialize(address tswapAddress)external initializer onlyProxy { require(tswapAddress != address(0), "ThunderLoan: oracle is zero address"); __Ownable_init(msg.sender); __UUPSUpgradeable_init(); __Oracle_init(tswapAddress); s_feePrecision = 1e18; s_flashLoanFee = 3e15; }`

– `onlyProxy` ensures the initializer can only be invoked via `delegatecall` through a proxy.
3. In deployment scripts, call `initialize` within the same transaction as proxy creation to eliminate any uninitialized window.

Informational

[I-1] Redundant Storage Reads of s_exchangeRate in updateExchangeRate (Storage Access Inefficiency + Elevated Gas Costs)

Description:
The function `AssetToken::updateExchangeRate(uint256 fee)` in `AssetToken.sol` reads the storage variable `uint256 private s_exchangeRate;` multiple times.

This causes: - Up to four separate reads of `s_exchangeRate` (initial calculation, comparison, revert parameters, and event emission) - Repeated storage access increases gas consumption on every call, as each SLOAD operation costs additional gas.

Impact:
- Higher per-transaction gas cost whenever `updateExchangeRate` is invoked, reducing protocol efficiency.
- In high-gas environments (e.g., network congestion), the function could fail due to exceeding block gas limits, potentially disrupting fee accrual and exchange rate updates.

Proof of Concept:
1. Deploy `AssetToken` with a non-zero `s_exchangeRate` and `totalSupply`.
2. Call `updateExchangeRate` with a valid `fee` and record gas used (e.g., ~100 000 gas).
3. Modify the function to cache `s_exchangeRate` in memory (see mitigation) and re-measure gas (e.g., ~85 000 gas).
4. Observe a ~15% reduction in gas cost per call due to fewer SLOAD operations.

Recommended Mitigation:
Cache the storage variable in a memory local to minimize redundant reads:

```
 function updateExchangeRate(uint256 fee) external onlyThunderLoan {
+    // Cache storage in memory
+    uint256 oldRate = s_exchangeRate;
+
-    uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) / totalSupply();
+    uint256 newExchangeRate = oldRate * (totalSupply() + fee) / totalSupply();

-    if (newExchangeRate <= s_exchangeRate) {
```

```
-         revert AssetToken__ExhangeRateCanOnlyIncrease(s_exchangeRate, newExchangeRate);
+     if (newExchangeRate <= oldRate) {
+         revert AssetToken__ExhangeRateCanOnlyIncrease(oldRate, newExchangeRate);
      }
      s_exchangeRate = newExchangeRate;
-     emit ExchangeRateUpdated(s_exchangeRate);
+     emit ExchangeRateUpdated(newExchangeRate); // Use memory variable post-update
  }
```

- Read `s_exchangeRate` exactly once at the start.
- Use the memory variable (`oldRate`) for comparisons and revert parameters.
- For the event, emit the updated value directly from memory (`newExchangeRate`).

This reduces SLOAD operations and lowers gas costs while maintaining functionality.

[I-2] Unvalidated Assignment of s_poolFactory in __Oracle_init_unchained (Missing Input Validation + Potential Misconfiguration)

Description:
In OracleUpgradeable.sol, the initializer function

```
function __Oracle_init_unchained(address poolFactoryAddress) internal onlyInitializing {
    s_poolFactory = poolFactoryAddress;
}
```

assigns the state variable `s_poolFactory` directly from the caller's input without any validation. A malicious or malformed address (e.g. zero address or attacker-controlled) can be set, leading to incorrect behavior or loss of security guarantees.

Impact:
- If `poolFactoryAddress` is the zero address, any subsequent call to `s_poolFactory` will revert or return invalid data, effectively disabling Oracle functionality.
- If an attacker-controlled address is provided, the oracle may trust and forward calls or data from a malicious factory, leading to price manipulation or fund theft in dependent contracts.
- Since this is an unchained initializer, misconfiguration during deployment or upgrade cannot be corrected without a full contract redeployment.

Proof of Concept:

1. Deploy a proxy pointing to `OracleUpgradeable`.
2. Call the initializer with `poolFactoryAddress = address(0)` (or any attacker EOA).
3. Any function that queries `s_poolFactory` (e.g., retrieving price feeds) now fails or returns attacker-controlled data, demonstrating broken functionality or compromised trust.

Recommended Mitigation:
Add explicit validation to the initializer to ensure a non-zero, contract-type address is provided:

```
 function __Oracle_init_unchained(address poolFactoryAddress) internal
     onlyInitializing {
-    s_poolFactory = poolFactoryAddress;
+    require(poolFactoryAddress != address(0), "Oracle: poolFactory is
    zero address");
+    uint256 codeSize;
+    assembly { codeSize := extcodesize(poolFactoryAddress) }
+    require(codeSize > 0, "Oracle: poolFactory is not a contract");
+    s_poolFactory = poolFactoryAddress;
 }
```

- `require(poolFactoryAddress != address(0))` prevents zero-address assignment.
- Checking `extcodesize` ensures the provided address hosts deployed contract code.
- These guards eliminate misconfiguration risk and ensure `s_poolFactory` points to a valid factory contract.

[I-3] Inconsistent Parameter Naming in initialize (Naming Inconsistency + Potential Misuse)

Description:
In `src/protocol/ThunderLoan.sol`, the `initialize` function declares its oracle factory address parameter as `tswapAddress`, but then passes it into `__Oracle_init`, which internally expects a parameter named `poolFactoryAddress`:

```
function initialize(address tswapAddress) external initializer {
    __Ownable_init(msg.sender);
    __UUPSUpgradeable_init();
    // e using tswap as a oracle
    __Oracle_init(tswapAddress);
   ...
}

function __Oracle_init(address poolFactoryAddress) internal
   onlyInitializing {
    __Oracle_init_unchained(poolFactoryAddress);
}
```

The mismatched names (“tswapAddress” vs. “poolFactoryAddress”) introduce unnecessary cognitive load and risk of passing an incorrect address to the oracle initializer.

Impact:
- Developer confusion: future maintainers may misunderstand which contract address is being supplied.
- Potential for misconfiguration: a developer might accidentally pass the wrong address parameter if naming is misleading.
- Documentation drift: comments referring to “tswap” may diverge from actual factory logic, hindering audits and integrations.

Proof of Concept:
- A code reviewer sees `initialize(address tswapAddress)` and assumes it expects a TSwap router or similar, then mistakenly supplies an unrelated contract address.
- At runtime, the wrong address flows into `__Oracle_init`, leading to a broken or malicious oracle configuration.

Recommended Mitigation:
- Rename the parameter in `initialize` to match the internal initializer signature:

```
-function initialize(address tswapAddress) external initializer {
+function initialize(address poolFactoryAddress) external initializer {
     __Ownable_init(msg.sender);
     __UUPSUpgradeable_init();
-    __Oracle_init(tswapAddress);
+    __Oracle_init(poolFactoryAddress);
    ...
 }
```

- Update inline comments to reference the “pool factory” or “oracle factory” consistently.
- Ensure all external documentation and deployment scripts use the updated parameter name.
-

[I-4] Unused and Misspelled Custom Error ThunderLoan__ExhangeRateCanOnlyIncrease (Dead Code + Naming Inconsistency)

Description:
The contract declares a custom error

```
error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

which is never referenced in any `revert` or `require` statement. Additionally, the identifier contains a typo: “ExhangeRate” instead of “ExchangeRate”.

Impact:
- Dead code increases bytecode size, raising deployment and verification gas costs.
- Unused definitions create confusion for developers and auditors who may search for missing error usages.
- Typos in error names propagate potential inconsistencies if later used, leading to mismatches in revert messages or error signatures.

Recommended Mitigation:
- Remove the unused error declaration entirely to reduce contract size and eliminate confusion.

[I-5] s_feePrecision Not Declared as Constant (Storage Inefficiency + Accidental Modification Risk)

Description:
In `src/protocol/ThunderLoan.sol`, the variable

```
1 uint256 private s_feePrecision;
```

is declared as a mutable storage slot but is only ever set once in `initialize` to `1e18` and never changed afterward. Such a value is inherently constant and should be declared accordingly.

Impact:
- Higher gas costs: Reading from and writing to a storage variable is more expensive than inlining a constant.
- Larger bytecode size: Unnecessary storage slots increase contract size.
- Risk of accidental reassignment: Future code changes might inadvertently modify what should be an immutable precision value, breaking fee calculations.

Recommended Mitigation:
1. Declare the precision as a constant:

```
1 uint256 private constant FEE_PRECISION = 1e18;
```

2. Replace all references to `s_feePrecision` with `FEE_PRECISION`.