Protocol Audit Report

Version 1.0

io10

January 15, 2025

Protocol Audit Report

io10

15/01/2025

Prepared by: [io10]

Lead Auditors:

- io10

Table of Contents

ThunderLoan Protocol Summary

The ThunderLoan protocol is meant to do the following:

Give users a way to create flash loans 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.

Disclaimer

io10 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 me 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

- 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
- Tokens:
  - ERC20s:
    - USDC
    - DAI
    - LINK
    - WETH

Actors / 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

The audit of the ThunderLoan protocol revealed critical vulnerabilities that could compromise its functionality and user trust. Issues were identified in key areas such as fee calculations, collateral redemption, and flashloan repayments, exposing the protocol to potential exploitation. Additionally, upgrade inconsistencies and reliance on manipulable on-chain price oracles posed risks to the integrity and fairness of its operations.

To address these challenges, targeted mitigations were recommended, including refining fee and exchange rate mechanisms, enforcing stricter repayment validations, aligning storage structures across contract upgrades, and transitioning to more robust decentralized oracles. These changes aim to enhance security, maintain operational consistency, and protect user assets from exploitation.

Implementing the proposed solutions will bolster the protocol's reliability and ensure it meets the expectations of its users and stakeholders.

Issues found

Severity	Number Of Issues Found
High	3
Medium	2
Low	0
Informational	0
Total	5

Findings

High

[H-1] Reward Manipulation in deposit function breaks `Thunderloan::redeem` functionality which prevents depositors from redeeming their full collateral

Description: See 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;
        console.log("exchangeRate", exchangeRate);
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);
        uint256 calculatedFee = getCalculatedFee(token, amount); //bug
            why is a fee calculated when depositing? the amount for
            getcalculatedfee is the amount a user wants to borrow and
            not an amount a user deposits? When this happens, it means
            that the depositor can never redeem their full amount. See
            test_redeem in the test file
        assetToken.updateExchangeRate(calculatedFee); //bug comment out
            this line to allow depositor to redeem full amount or
            refactor updateexchangerate to allow for a deposit without
            updating the exchange rate DOS
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;}
```

The `Thunderloan::getCalculatedFee` is called in the deposit function when a user wants to deposit tokens into thunderloan. This is the wrong implementation because the following line is in the documentation 'These AssetTokens gain interest over time depending on how often people take out flash loans!' Fees are taken from users when they take out flash loans. There is no need for fees to be calculated or the `Thunderloan::updateExchangeRate` to be called in the `Thunderloan::deposit` function. As a result of this, when a user deposits and tries to redeem their full collateral, the exchange rate calculated in `Thunderloan::redeem` is wrong based on the logic used in the above function.

Impact: Exchange Rate calculated in `Thunderloan::redeem` is miscalculated and is higher than expected which stops users from redeeming their collateral.

Proof of Concept:

Code

```
function test_redeem() 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();
        vm.startPrank(liquidityProvider);
        /*address swapPoolOfToken = IPoolFactory(tokenA).getPool(token)
           ;
        uint256 price = ITSwapPool(swapPoolOfToken).
           getPriceOfOnePoolTokenInWeth(); */
        thunderLoan.redeem(tokenA, DEPOSIT_AMOUNT);
        vm.stopPrank();
        uint256 balance = tokenA.balanceOf(liquidityProvider);
        assert(balance > DEPOSIT_AMOUNT);
    }
```

Recommended Mitigation: Remove `Thunderloan::getCalculatedFee` and `Thunderloan::updateExchangeRate` from `Thunderloan::deposit` 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;
        console.log("exchangeRate", exchangeRate);
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);
-         uint256 calculatedFee = getCalculatedFee(token, amount); //bug
     why is a fee calculated when depositing? the amount for
    getcalculatedfee is the amount a user wants to borrow and not an
    amount a user deposits? When this happens, it means that the
    depositor can never redeem their full amount. See test_redeem in the
     test file
-        assetToken.updateExchangeRate(calculatedFee); //bug comment out
     this line to allow depositor to redeem full amount or refactor
    updateexchangerate to allow for a deposit without updating the
    exchange rate DOS
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;}
```

[H-2] BalanceOf exploit allows attacker to drain all funds from contract

Description: `ThunderLoan::flashloan` constains the following starting and ending balance checks using balanceOf:

```
 function flashloan(
        address receiverAddress,
        IERC20 token,
        uint256 amount,
        bytes calldata params
    ) external revertIfZero(amount) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 startingBalance = IERC20(token).balanceOf(address(
            assetToken));

        if (amount > startingBalance) {
            revert ThunderLoan__NotEnoughTokenBalance(startingBalance,
                amount);
        }

        if (receiverAddress.code.length == 0) {
            revert ThunderLoan__CallerIsNotContract();
        }

        uint256 fee = getCalculatedFee(token, amount);
        // slither-disable-next-line reentrancy-vulnerabilities-2
           reentrancy-vulnerabilities-3
        assetToken.updateExchangeRate(fee);

        emit FlashLoan(receiverAddress, token, amount, fee, params);

        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
                )
            )
        );

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

Due to the balanceOf checks for the ending balance, it means thaat the repay function is not the only way to pay the token back to the assetcontract. An attacker can simply call the deposit function in the thunderloan contract and that function will change the balance of the asset token contract. If an attacker can call the deposit function and deposit the flashloan + fee back into the assettoken contract, this will pass all the checks to repay the flashloan. Since the attacker has now deposited the loan, they can then call the redeem function and take the deposited funds out of the protocol. So to summarise, take flash loan, pay back the loan by depositing the loan into the protocol, then redeem the funds deposited in the function.

Impact: Attacker can drain all funds from the contract

Proof of Concept:

POC

```
function test_oraclemanipulationv2() public setAllowedToken hasDeposits
     {
        uint256 amountToBorrow = AMOUNT * 10;
        vm.startPrank(user);
        tokenA.mint(address(implementedFlashLoanReceiver), AMOUNT * 10)
            ;
        uint256 balancebefore = tokenA.balanceOf(
            address(implementedFlashLoanReceiver)
        );
        thunderLoan.flashloan(
            address(implementedFlashLoanReceiver),
            tokenA,
            amountToBorrow,
            ""
        );
        vm.stopPrank();
        vm.startPrank(address(implementedFlashLoanReceiver));
        thunderLoan.redeem(tokenA, implementedFlashLoanReceiver.
           amountfee());
        vm.stopPrank();
        uint256 balanceafter = tokenA.balanceOf(
            address(implementedFlashLoanReceiver)
        );
        assert(balanceafter > balancebefore);
    }
```

The ImplentedFlashLoanContract can be located at /src/pools4flashloanexploit/ImplementedFlashLoanReceiver.sol.

Recommended Mitigation: To mitigate this, include a mapping in `ThunderLoan::repay` that is updated whenever the repay function is called and add a check in `ThunderLoan::flashloan` to make sure the mapping was updated for the receiver address. This would force every user to use the repay function to pay back their loan.

```

+ mapping (address => bool) private calledrepay;

function flashloan(
        address receiverAddress,
        IERC20 token,
        uint256 amount,
        bytes calldata params
    ) external revertIfZero(amount) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 startingBalance = IERC20(token).balanceOf(address(
           assetToken));

        if (amount > startingBalance) {
            revert ThunderLoan__NotEnoughTokenBalance(startingBalance,
                 amount);
        }

        if (receiverAddress.code.length == 0) {
            revert ThunderLoan__CallerIsNotContract();
        }

        uint256 fee = getCalculatedFee(token, amount);
        // slither-disable-next-line reentrancy-vulnerabilities-2
            reentrancy-vulnerabilities-3
        assetToken.updateExchangeRate(fee);

        emit FlashLoan(receiverAddress, token, amount, fee, params);

        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
                )
            )
        );

        uint256 endingBalance = token.balanceOf(address(assetToken));
        if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(
                startingBalance + fee,
                endingBalance
            );
+           if(!calledrepay[msg.sender]){
+              revert ThunderLoan__MustCallRepayFunction();
+            }
+        }
+        delete calledrepay[msg.sender];
        s_currentlyFlashLoaning[token] = false;
    }

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

```
            token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
        }
```

[H-3] Storage Slot Clashes messes up key functionality of the protocol including `ThunderLoan::getCalculatedFee`

Description: Running diff ./src/protocol/ThunderLoan.sol ./src/upgradedProtocol/ThunderLoanUpgraded.sol to see the differences between the lines of code in both files revealed the following:

< uint256 private s_feePrecision;

> ```
> uint256 public constant FEE_PRECISION = 1e18;
> ```

This showed that the `ThunderLoan::s_feePrecision` was replaced with a `ThunderLoanUpgraded::FEE_PRECISION` constant. Constants are not stored in storage and this leads to a storage slot being ommitted. Also, the storage slot for `ThunderLoan::s_flashLoanFee` was in the wrong position in `ThunderLoanUpgraded` which leads to a whole host of issues. One such issue is that when calculating the fee using `ThunderLoanUpgraded::getCalculatedFee`, the calculation was as follows:

```
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;
    }
```

Due to storage clash of `ThunderLoan::s_flashLoanFee` , `ThunderLoanUpgraded::s_flashLoanFee` is reset to 0 which means that the calculated fee will always be 0 which will keep the exchange rate stagnant and revert the function everytime.

Impact: Storage clashes breaks protocol functionality

Proof of Concept: details>

Code

```
function test_storageslotupgrademappingstorageslotclash() public {
        thunderLoan2 = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(thunderLoan2), "");
        thunderLoan2 = ThunderLoanUpgraded(address(proxy));
        vm.prank(thunderLoan.owner());
        thunderLoan2.setAllowedToken(tokenA, true);
```

```
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, DEPOSIT_AMOUNT);
        tokenA.approve(address(thunderLoan2), type(uint256).max);
        thunderLoan2.deposit(tokenA, DEPOSIT_AMOUNT);
        vm.stopPrank();
        uint256 amountToBorrow2 = AMOUNT * 10;
        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), AMOUNT * 10);
        thunderLoan.flashloan(
            address(mockFlashLoanReceiver),
            tokenA,
            amountToBorrow2,
            ""
        );
        vm.stopPrank();
    }
```

Looking at the logs for this test, you will see that when `ThunderLoanUpgraded::updateExchangeRate` is called during `ThunderLoanUpgraded::flashloan`, the value passed is always 0 due to the aforementioned issue.

Recommended Mitigation: Ensure that when upgrading contracts, all storage slots and their data types are kept consistent in the upgraded contract to avoid storage clashes.

Medium

[M-1] Oracle Manipulation in fee calculation allows users to reduce flash loan fees significantly

Description: There is an oracleupgradeable contract and it has the following function:

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

Thunderloan can get the value of the token in weth by quering how much that token is worth in weth in that pool right now. The reason thunderloan does this is that when calculating the fee, it gets the value of the token and does a calculation with it. This calculation also calculates the fee value wrong due to an order of operations bug which i will cover later. See below:

```
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: Liquidity providers on thunderloan dont get their expected amount of fees as user can manipulate the amount of fees they pay per flashloan.

Proof of Concept: The problem here is that thunderloan is using a single pool to get a price at the current block. This pool's price can change at any time with people buying and selling that pool on tswap. This means that the fee that thunderloan gets can be manipulated and will vary many times based on people buying and selling tokens on that pool. This means the fee that thunderloan takes for its flash loans will vary a lot based on the price of that pool. This is where the oracle manipulation attack can happen and I will detail how I did this using flash loans.

The idea on a high level would be that a user can reduce fees each time they take a flash loan by manipulating the tswap oracle used to get the price of the token the user is borrowing. To do this, the user takes a flash loan and once they get the loan, go into the pool that thunderloan is basing their prices off and swap a large amount of the token for weth in that pool which means there will me way more of token and way less weth. This will make the price of a pool token in weth much less than it was before based on the calculation in the function above which in turn reduces the fee amount. Then the user can go to another exchange and sell the tokens they just bought from the pool and get the token back and then pay back the loan to thunderloan. Once this is done, the fee is now significantly less than it was before. Now, the user can take a larger flash loan and pay less fees for that same amount of tokens. This is how oracle manipulation will work in this example.

In the BaseTest.t.sol, the original document used deployed using a MockPoolFactory and a MockTswapPool with a fixed price so this wouldnt be able to demonstrate the attack so i took the poolfactory and tswappool.sol contracts from the tswap protocol and imported them here. I initialised them like i did when testing tswap. I also created another exchange called bswap which pretty much uses the same code as the the tswappool contract but the idea was that bswap was going to be the exchange that I would sell the tokens I bought from the pool used as the thunderloan oracle.

I also created a FlashLoanReceiver contract that would do all the aforementioned swaps in the relevant pools and you can see the code for all of that in the contract which was easy enough. Note that the strategy I used in that contract was not profitable simply because the tswap oracle i created only had 100 tokens of token and weth in the pool so when i tried to swap 50 tokens for weth, due to slippage that comes from the swap trying to hold an x*y=k invariant which we have already learnt about, i dont get anywhere near 50 tokens back which was annoying. Dont focus on the profitability, this example is just to show that the oracle manipulation can be done.

From here, all I did was write a test to prove this which is in the thunderloantest.t.sol file but i will also paste it below:

Code

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

        uint256 amountToBorrow2 = AMOUNT * 10;
        uint256 calculatedFee2 = thunderLoan.getCalculatedFee(
            tokenA,
            amountToBorrow
        );
        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), AMOUNT * 10);
        thunderLoan.flashloan(
            address(mockFlashLoanReceiver),
            tokenA,
            amountToBorrow2,
            ""
        );
        vm.stopPrank();
        console.log(calculatedFee, calculatedFee2);
        assert(calculatedFee > calculatedFee2);
    }
```

This test shows that once the flashloan transaction happens, the calculated fee for the first flashloan and the fee for the second flash loan are very different even though the amounts borrowed in each loan are exactly the same, the fee for the second flash loan is less than half of what the first one is due to the orace manipulation. A user can do this everytime they want to take a flash loan which will hurt profits for the protocol and for liquidity providers. This is a very easy example of how oracle manipulation works and we are going to look at a lot more complicated examples of how this works so you will know how this can be exploited. Just keep in mind that if a protocol is using an onchain liquidity pool to get prices that are used for functions in its protocol, those functions can all be attacked using oracle manipulation and flashloans. Another way to do this would be to add functionality in the flashloanreceiver contract I created to call `Thunderloan::flashloan` again after manipulating the fees but add a check to make sure it doesnt make the swaps again in the reentrancy. It just does something else. In this case, i didnt do that. I jyst used the mockflashloanreceiver contract to call `Thunderloan::flashloan` again manually because the mockflashloanreceiver doesn't do the oracle manipulation. It just borrows the funds and pays them back and this shows that the fee when the second flashloan is taken is much lower than the first time. Both ways are acceptable but in reality, i would use the first method to make sure that everything is done in one transaction and no other transactions can happen in between.

Recommended Mitigation: To avoid this attack, use a decentralised price oracle like chainlink to get more accurate token prices that are aggregated from multiple sources. You can also use TWAP's which are time weighted average prices which still get prices from one pool but the prices are averaged over a number of blocks so even if a tries to reduce fees using oracle manipulation, the price that will be used will not be the price at the current block. It will be the price of the asset averaged over a number of blocks which means that this attack wont work. TWAP's have also been manipulated previously so this might not be the best solution. Best option is to use a chainlink price oracle.

[M-2] Misuse of initializer function in `ThunderLoanUpgraded::initialise` means that the function cannot be called

Description: `ThunderLoanUpgraded::initialise` uses an initialiser modifier which can only be called once on the proxy contract. Since it has already been called to initialise the `ThunderLoan` contract, when the implementation contract is to be changed to `ThunderLoanUpgraded`, calling initialise will not work as the intialiser modifier will identify that the version has been set to 1 and will throw an error.

Impact: All state changes made in initialiser function will not be updated on proxy contract

Proof of Concept:

```
function test_storageslotupgradecannotinitialize() public {
        thunderLoan2 = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(thunderLoan2), "");
        thunderLoan2 = ThunderLoanUpgraded(address(proxy));
        vm.expectRevert(InvalidInitialization.selector);
        thunderLoan2.initialize(address(poolFactory));
    }
```

Recommended Mitigation: Replace initialise modifier with reinitializer modifer using the version function. This allows a function to be reinitialised and its version updated. See the openzeppelin intializable contract at https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/proxy/utils/Initializable.sol .