



Thunder Loan Audit Report

Version 1.0

0xBlockPay

October 16, 2024

Thunder Loan Audit Report

0xBlockPay

October 16, 2024

Prepared by: 0xBlockPay

Lead Security Researcher: - eth0x

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
 - Roles
- Executive Summary
 - Issues found
- Findings
 - High
 - Medium
 - Low
 - Informational
 - Gas

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.

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

Disclaimer

The 0xBlockPay team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

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

Audit Details

Scope

The findings describe in this document correspond the following repository and commit hash:

```
1 git clone https://github.com/Cyfrin/6-thunder-loan-audit
```

```
1 Commit Hash: 026da6e73fde0dd0a650d623d0411547e3188909
```

In Scope:

```
1  |-- interfaces
2  |    |-- IFlashLoanReceiver.sol
3  |    |-- IPoolFactory.sol
4  |    |-- ITSwapPool.sol
5  |    |-- IThunderLoan.sol
6  |-- protocol
7  |    |-- AssetToken.sol
8  |    |-- OracleUpgradeable.sol
9  |    |-- ThunderLoan.sol
10 |-- upgradedProtocol
11    |-- ThunderLoanUpgraded.sol
```

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

Roles

Roles

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

Executive Summary

*We spend 10 hours with 1 auditor using foundry, aderyn and slither tools.

Issues found

Severity	Number of issues found
High	2
Medium	2
Low	3
Info	1
Gas	2
Total	10

Findings

High

[H-1] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`

Description: `ThunderLoan.sol` has two variables in the following order:

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

However, the expected upgraded contract `ThunderLoanUpgraded.sol` has them in a different order.

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

Due to how Solidity storage works, after the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the positions of storage variables when working with upgradeable contracts.

Impact: After upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that users who take out flash loans right after an upgrade will be charged the wrong fee. Additionally the `s_currentlyFlashLoaning` mapping will start on the wrong storage slot.

Proof of Code:

Code Add the following code to the `ThunderLoanTest.t.sol` file.

```
1 // You'll need to import `ThunderLoanUpgraded` as well
```

```
2 import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/  
    ThunderLoanUpgraded.sol";  
3  
4 function testUpgradeBreaks() public {  
5     uint256 feeBeforeUpgrade = thunderLoan.getFee();  
6     vm.startPrank(thunderLoan.owner());  
7     ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();  
8     thunderLoan.upgradeTo(address(upgraded));  
9     uint256 feeAfterUpgrade = thunderLoan.getFee();  
10  
11     assert(feeBeforeUpgrade != feeAfterUpgrade);  
12 }
```

You can also see the storage layout difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`

Recommended Mitigation: Do not switch the positions of the storage variables on upgrade, and leave a blank if you're going to replace a storage variable with a constant. In `ThunderLoanUpgraded.sol`:

```
1 - uint256 private s_flashLoanFee; // 0.3% ETH fee  
2 - uint256 public constant FEE_PRECISION = 1e18;  
3 + uint256 private s_blank;  
4 + uint256 private s_flashLoanFee;  
5 + uint256 public constant FEE_PRECISION = 1e18;
```

[H-2] Unnecessary updateExchangeRate in deposit function incorrectly updates exchangeRate preventing withdraws and unfairly changing reward distribution

Description:

```
1 function deposit(IERC20 token, uint256 amount) external  
    revertIfZero(amount) revertIfNotAllowedToken(token) {  
2     AssetToken assetToken = s_tokenToAssetToken[token];  
3     uint256 exchangeRate = assetToken.getExchangeRate();  
4     uint256 mintAmount = (amount * assetToken.  
        EXCHANGE_RATE_PRECISION()) / exchangeRate;  
5     emit Deposit(msg.sender, token, amount);  
6     assetToken.mint(msg.sender, mintAmount);  
7     uint256 calculatedFee = getCalculatedFee(token, amount);  
8     assetToken.updateExchangeRate(calculatedFee);  
9     token.safeTransferFrom(msg.sender, address(assetToken), amount)  
        ;  
10 }
```

Impact:

Proof of Concept:

Recommended Mitigation:

[H-3] By calling a flashloan and then ThunderLoan::deposit instead of ThunderLoan::repay users can steal all funds from the protocol

[H-4] getPriceOfOnePoolTokenInWeth uses the TSwap price which doesn't account for decimals, also fee precision is 18 decimals

Medium**[M-1] Centralization risk for trusted owners**

Impact: Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

Instances (2):

```
1 File: src/protocol/ThunderLoan.sol
2
3 223:     function setAllowedToken(IERC20 token, bool allowed) external
         onlyOwner returns (AssetToken) {
4
5 261:     function _authorizeUpgrade(address newImplementation) internal
         override onlyOwner { }
```

Contralized owners can brick redemptions by disapproving of a specific token**[M-2] Using TSwap as price oracle leads to price and oracle manipulation attacks**

Description: The TSwap protocol is a constant product formula based AMM (automated market maker). The price of a token is determined by how many reserves are on either side of the pool. Because of this, it is easy for malicious users to manipulate the price of a token by buying or selling a large amount of the token in the same transaction, essentially ignoring protocol fees.

Impact: Liquidity providers will drastically reduced fees for providing liquidity.

Proof of Concept:

The following all happens in 1 transaction.

1. User takes a flash loan from ThunderLoan for 1000 tokenA. They are charged the original fee fee1. During the flash loan, they do the following:

1. User sells 1000 `tokenA`, tanking the price.
2. Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`.
 1. Due to the fact that the way `ThunderLoan` calculates price based on the `TSwapPool` this second flash loan is substantially cheaper.

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

3. The user then repays the first flash loan, and then repays the second flash loan.

I have created a proof of code located in my `audit-data` folder. It is too large to include here.

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

[M-4] Fee on transfer, rebase, etc

Low

[L-1] Empty Function Body - Consider commenting why

Instances (1):

```
1 File: src/protocol/ThunderLoan.sol
2
3 261:     function _authorizeUpgrade(address newImplementation) internal
        override onlyOwner { }
```

[L-2] Initializers could be front-run

Initializers could be front-run, allowing an attacker to either set their own values, take ownership of the contract, and in the best case forcing a re-deployment

Instances (6):

```
1 File: src/protocol/OracleUpgradeable.sol
2
3 11:     function __Oracle_init(address poolFactoryAddress) internal
        onlyInitializing { }
```



```

1 File: src/protocol/ThunderLoan.sol
2
3 138:     function initialize(address tswapAddress) external initializer
4       {
5 138:     function initialize(address tswapAddress) external initializer
6       {
7 139:         __Ownable_init();
8
9 140:         __UUPSUpgradeable_init();
10
11 141:         __Oracle_init(tswapAddress);

```

[L-3] Missing critical event emissions

Description: When the `ThunderLoan::s_flashLoanFee` is updated, there is no event emitted.

Recommended Mitigation: Emit an event when the `ThunderLoan::s_flashLoanFee` is updated.

```

1 +     event FlashLoanFeeUpdated(uint256 newFee);
2 .
3 .
4 .
5     function updateFlashLoanFee(uint256 newFee) external onlyOwner {
6         if (newFee > s_feePrecision) {
7             revert ThunderLoan__BadNewFee();
8         }
9         s_flashLoanFee = newFee;
10 +     emit FlashLoanFeeUpdated(newFee);
11 }

```

Informational

[I-1] Poor Test Coverage

```

1 Running tests...
2 | File                                     | % Lines      | % Statements
3 | -----|-----|-----|
4 | src/protocol/AssetToken.sol             | 70.00% (7/10) | 76.92% (10/13)
5 |   | 50.00% (1/2) | 66.67% (4/6) |

```

5	<code>src/protocol/OracleUpgradeable.sol</code>	100.00% (6/6)	100.00% (9/9)
	100.00% (0/0) 80.00% (4/5)		
6	<code>src/protocol/ThunderLoan.sol</code>	64.52% (40/62)	68.35% (54/79)
	37.50% (6/16) 71.43% (10/14)		

[I-2] Not using `__gap` [50] for future storage collision mitigation**[I-3] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6****[I-4] Doesn't follow <https://eips.ethereum.org/EIPS/eip-3156>**

Recommended Mitigation: Aim to get test coverage up to over 90% for all files.

Gas**[G-1] Using bools for storage incurs overhead**

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. See source.

Instances (1):

```
1 File: src/protocol/ThunderLoan.sol
2
3 98:     mapping(IERC20 token => bool currentlyFlashLoaning) private
      s_currentlyFlashLoaning;
```

[G-2] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

Instances (3):

```
1 File: src/protocol/AssetToken.sol
2
3 25:     uint256 public constant EXCHANGE_RATE_PRECISION = 1e18;
```

```
1 File: src/protocol/ThunderLoan.sol
2
3 95:      uint256 public constant FLASH_LOAN_FEE = 3e15; // 0.3% ETH fee
4
5 96:      uint256 public constant FEE_PRECISION = 1e18;
```

[GAS-3] Unnecessary SLOAD when logging new exchange rate

In `AssetToken::updateExchangeRate`, after writing the `newExchangeRate` to storage, the function reads the value from storage again to log it in the `ExchangeRateUpdated` event.

To avoid the unnecessary SLOAD, you can log the value of `newExchangeRate`.

```
1 s_exchangeRate = newExchangeRate;
2 - emit ExchangeRateUpdated(s_exchangeRate);
3 + emit ExchangeRateUpdated(newExchangeRate);
```