

# Thunder Loan Audit Report

---

Prepared by: PAYAL Lead Auditors:

- [Payal](#)

Assisting Auditors:

- None

## Table of contents

---

► Details

Contents

- [Thunder Loan Audit Report](#)
- [Table of contents](#)
- [About YOUR\\_NAME\\_HERE](#)
- [Disclaimer](#)
- [Risk Classification](#)
- [Audit Details](#)
  - [Scope](#)
- [Protocol Summary](#)
  - [Roles](#)
- [Executive Summary](#)
  - [Issues found](#)
- [Findings](#)
  - [High](#)
    - [\[H-1\] Mixing up variable location causes storage collisions in `ThunderLoan::s\_flashLoanFee` and `ThunderLoan::s\_currentlyFlashLoaning`](#)
    - [\[H-2\] Unnecessary `updateExchangeRate` in `deposit` function incorrectly updates `exchangeRate` preventing withdraws and unfairly changing reward distribution](#)
    - [\[H-3\] 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:](#)
      - [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](#)
    - [\[M-4\] Fee on transfer, rebase, etc](#)
  - [Low](#)
    - [\[L-1\] Empty Function Body - Consider commenting why](#)
    - [\[L-2\] Initializers could be front-run](#)

- [L-3] Missing critical 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 YOUR\_NAME\_HERE

As a new security researcher exploring Web3 protocols, I'm committed to securing decentralized systems. This report highlights my efforts to improve security practices in digital ecosystems.

# 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 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	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:

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

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

```
uint256 private s_flashLoanFee; // 0.3% ETH fee  
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

```
// You'll need to import `ThunderLoanUpgraded` as well  
import { ThunderLoanUpgraded } from  
"../../src/upgradedProtocol/ThunderLoanUpgraded.sol";  
  
function testUpgradeBreaks() public {  
    uint256 feeBeforeUpgrade = thunderLoan.getFee();  
    vm.startPrank(thunderLoan.owner());  
    ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();  
    thunderLoan.upgradeTo(address(upgraded));  
    uint256 feeAfterUpgrade = thunderLoan.getFee();  
  
    assert(feeBeforeUpgrade != feeAfterUpgrade);  
}
```

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`:

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

```
+    uint256 private s_flashLoanFee;
+    uint256 public constant FEE_PRECISION = 1e18;
```

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

#### Description:

```
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);
}
```

#### 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):

File: `src/protocol/ThunderLoan.sol`

```
223:    function setAllowedToken(IERC20 token, bool allowed) external
onlyOwner returns (AssetToken) {
```

```
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.

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

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):*

```
File: src/protocol/ThunderLoan.sol
```

```
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):*

```
File: src/protocol/OracleUpgradeable.sol
```

```
11:     function __Oracle_init(address poolFactoryAddress) internal  
onlyInitializing { }
```

```
File: src/protocol/ThunderLoan.sol
```

```
138:     function initialize(address tswapAddress) external initializer {
```

```
138:     function initialize(address tswapAddress) external initializer {
```

```
139:         __Ownable_init();
```

```
140:         __UUPSUpgradeable_init();
```

```
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.

```
+     event FlashLoanFeeUpdated(uint256 newFee);  
.  
.  
.  
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {  
        if (newFee > s_feePrecision) {  
            revert ThunderLoan__BadNewFee();  
        }  
    }
```

```
        s_flashLoanFee = newFee;
+       emit FlashLoanFeeUpdated(newFee);
    }
```

Informational

[I-1] Poor Test Coverage

```
Running tests...
| File                                     | % Lines          | % Statements      | %
Branches      | % Funcs          |
| ----- | ----- | ----- | --
| ----- | ----- |
| src/protocol/AssetToken.sol           | 70.00% (7/10)    | 76.92% (10/13)    |
50.00% (1/2) | 66.67% (4/6)    |
| src/protocol/OracleUpgradeable.sol    | 100.00% (6/6)    | 100.00% (9/9)    |
100.00% (0/0) | 80.00% (4/5)    |
| src/protocol/ThunderLoan.sol          | 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

[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):*

```
File: src/protocol/ThunderLoan.sol

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):*

```
File: src/protocol/AssetToken.sol
```

```
25:      uint256 public constant EXCHANGE_RATE_PRECISION = 1e18;
```

```
File: src/protocol/ThunderLoan.sol
```

```
95:      uint256 public constant FLASH_LOAN_FEE = 3e15; // 0.3% ETH fee
```

```
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`.

```
s_exchangeRate = newExchangeRate;  
- emit ExchangeRateUpdated(s_exchangeRate);  
+ emit ExchangeRateUpdated(newExchangeRate);
```