□ Title: Fee-On-Transfer Token Miscalculation

□ Description:

Some ERC20 tokens (like SafeMoon, RFI, etc.) have a fee-on-transfer mechanism — meaning if you send 100 tokens, only 90 (for example) might actually reach the recipient due to internal deductions.

If a smart contract assumes the full amount (e.g., amount sent by user) is received without checking the actual token balance before and after the transfer, it can:

Over-credit the sender (attacker gains)

Underfund the protocol

Cause accounting or logic errors

□ Where to Focus:

Look for patterns like:

```
token.transferFrom(msg.sender, address(this), amount);
doSomethingWith(amount);
This assumes amount was received, but if the token has transfer fees, address(this) may receive less than amount.
```

□ Fix:

Use balance-difference checks to verify how many tokens were actually received:

```
uint256 balanceBefore = token.balanceOf(address(this));
token.transferFrom(msg.sender, address(this), amount);
uint256 balanceAfter = token.balanceOf(address(this));
uint256 received = balanceAfter - balanceBefore;
Now use received for all further logic — not the original amount.
```

□ Title: Missing Reentrancy Guard for Concurrent Flash Loans per Token

□ Description:

The contract sets a flag s_currentlyFlashLoaning[token] = true; when a flash loan is active for a given token. But if the check is missing before setting this flag, the same token can be flash-loaned multiple times in the same transaction, potentially via reentrancy or callback-based exploits.

This can lead to:

Double loans for the same token

Exploits where loaned tokens are not paid back properly

Inconsistencies in accounting and logic

□ Where to Focus:

Check the logic before setting the flag:

```
s_currentlyFlashLoaning[token] = true;
// Flash loan logic...
s_currentlyFlashLoaning[token] = false;
If there's no check like:
```

```
require(!s_currentlyFlashLoaning[token], "Flash loan already in progress");
...then an attacker can initiate a second flash loan of the same token during the first loan's execution.
```

✅ Fix:

Add a require statement to block concurrent flash loans for the same token:

```
require(!s_currentlyFlashLoaning[token], "Reentrancy: flash loan already in progress");
s_currentlyFlashLoaning[token] = true;
...
s_currentlyFlashLoaning[token] = false;
Also consider using a reentrancy guard (nonReentrant) for added protection if reentrancy is possible.
```

🧾 Title: Free Loan Payback via Self-Deposit Loop (False Loan Repayment)

🔍 Description:

The code checks loan repayment using:

```
uint256 endingBalance = token.balanceOf(address(assetToken));
```

The logic assumes that if the contract assetToken holds enough of the loaned token at the end, the loan is repaid.

But this opens up a critical exploit: A malicious user can deposit the borrowed tokens back into the protocol, receive assetToken in return (like aLP tokens), and that deposit makes it seem like the loan is repaid, but they never really "returned" the funds — just locked it temporarily.

Even worse — they can then:

Use the assetToken to redeem again for original tokens

Take a loan, deposit it back, and loop to get free funds

🎯 Where to Focus:

Watch for patterns like:

```
// Assumes balance of token inside another contract proves loan repayment
uint256 endingBalance = token.balanceOf(address(assetToken));
```

✅ Fix:

Never rely on balance snapshots of another contract to verify loan repayment.

🧾 Title: Unsafe Deletion of Token Mapping Breaks Redeem Logic

🔍 Description:

The line:

```
delete s_tokenToAssetToken[token];
```

completely removes the mapping between the base token and its associated assetToken (like an aToken, cToken, etc.).

If this is done without checking whether users still have deposits, it results in:

Users losing the ability to redeem their asset tokens

Funds getting stuck permanently

Breaking the redeem/withdraw flow

This is especially dangerous in upgradable protocols or during admin-level actions.

Where to Focus:

Any delete s_tokenToAssetToken[token]; or similar mapping clearing

Admin-only or governance functions

Protocol upgrade or shutdown flows

Redeem/withdraw logic that depends on this mapping to find the correct asset token

Fix:

Before deleting the mapping, check if users still have deposits:

```
require(assetToken.totalSupply() == 0, "Cannot delete: users still have deposits");
delete s_tokenToAssetToken[token];
```

Title: Price Oracle Manipulation Enables Flash Loan with Lower Fee

Description:

The fee calculation depends on the token's price in WETH:

```
uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) / s_feePrecision;
fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
```

If the getPriceInWeth() function is based on manipulatable sources like:

Uniswap TWAP (if misconfigured)

Instant DEX spot prices (most dangerous)

Low-liquidity pairs

...then the attacker can manipulate the token's price before taking the flash loan, making:

getPriceInWeth() return a much lower price

valueOfBorrowedToken seem cheaper than it actually is

So fee becomes much smaller than what it should be

This lets the attacker borrow a large amount with almost no fee, profiting significantly.

Where to Focus:

Look at how getPriceInWeth(address(token)) works (if it's based on manipulatable on-chain DEX prices, this is a red flag)

Check the liquidity and update frequency of price feeds

Ensure division happens after multiplication — but slither suppression (divide-before-multiply) shows dev may be ignoring known risks

Fix:

Use secure and manipulation-resistant oracles, like Chainlink:

```
price = chainlinkOracle.latestAnswer(); // Properly configured
```

Add sanity bounds:

Min price

Max deviation allowed

Oracle freshness checks (timestamps)

If relying on AMMs (like Uniswap):

Use TWAPs with >=30min windows

Or limit loan amounts based on liquidity

□ Title: Updating Fees Without Transparency or User Notification

□ Description:

The function:

```
function updateFlashLoanFee(uint256 newFee) external onlyOwner {
    if (newFee > s_feePrecision) {
        revert ThunderLoan__BadNewFee();
    }
    s_flashLoanFee = newFee;
}
```

Allows the owner to change flash loan fees arbitrarily and instantly without:

Informing users beforehand

Emitting an event to log the change

Enforcing any time delay or governance process

This can cause:

User distrust due to unexpected fee changes

Potential abuse by the owner to suddenly raise fees

No transparency in protocol behavior

□ Where to Focus:

Owner-only functions that update critical economic parameters

Lack of emit events after state changes

Absence of time locks or delay mechanisms before applying changes

□ Fix:

Emit an event whenever fees are updated:

```
event FlashLoanFeeUpdated(uint256 oldFee, uint256 newFee);
```

and inside the function:

```
emit FlashLoanFeeUpdated(s_flashLoanFee, newFee);
s_flashLoanFee = newFee;
```

Consider adding: A timelock or delay period before new fees take effect, so users can react or exit

□ Title: Potential Storage Layout Conflict Due to Constant and Variable Declaration Order

□ Description:

You have these declarations:

```
uint256 private s_flashLoanFee; // mutable fee variable
uint256 public constant FEE_PRECISION = 1e18; // constant
```

While constants don't consume storage slots, mixing state variables and constants without careful ordering can cause confusion in:

Proxy contracts (especially with upgradeable patterns like OpenZeppelin's Transparent Proxy)

Storage slot alignment

Future variable additions (if layout isn't properly managed)

If this contract is upgradeable, improper ordering or forgetting to reserve storage slots can lead to storage collision, causing:

Data corruption

Unexpected behavior in state variables

Where to Focus:

Contracts intended for upgradeability (using proxy patterns)

Storage variable order and gaps (uint256[50] private __gap;)

Constants vs. state variables — constants don't occupy storage but should be clearly separated

Fix:

For upgradeable contracts:

Use OpenZeppelin's Upgradeable contracts patterns

Add storage gaps to allow future variables

Keep state variables together, and constants separate (usually constants are declared at top or bottom)

Document storage layout clearly to avoid mistakes during upgrades.