TECHFUND Audit Score

Smart contract Audit Report

SLABS Audit

Completed on 2022-06-25

Score **POSITIVE**

Risk level		
	Critical	0
	High	0
	Medium	4
	Low	5
	Note	3

Risk level detail

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

The tester arrives at the likelihood and impact estimates, they can now combine them to get a final severity rating for this risk. Note that if they have good business impact information, they should use that instead of the technical impact information.

https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

Vulnerability Review

Number of warnings

Compiler Version	1
State Variable Default Visibility	0
In-sufficient Event and Logging	2
Re-Entrancy	2
Integer Overflow / Underflow	0
Parity Multisig Bug	0
Un-Satisfactory address checks	2
In-Correct Estimation of amounts in Swap and during Burn	2
Unnecessary check: Always true	1
Costly Loops for external calls	1
Unassigned state variable: results in failed check	1

TECHFUND Audit Score

Smart contract Audit Report

Compiler Version

1

The codebase includes different Solidity versions for different smart contracts. Specifically, the "MerchantWearable" contract is defined with version "^0.8.11", while the "TicketLazyMint" contract uses version "^0.8.0".

It is highly recommended to use the same Solidity version throughout the codebase to maintain consistency and reduce the potential for version-specific vulnerabilities or compatibility issues.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.0;
3
4  // OpenZeppelin contracts
5  import "@openzeppelin/contracts/access/Ownable.sol";
6  import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
7  import "@openzeppelin/contracts/utils/cryptography/draft-EIP712.sol";
8  import "@openzeppelin/contracts/interfaces/IERC1271.sol";
9  import "@openzeppelin/contracts/utils/Address.sol";
10 import "@openzeppelin/contracts/metatx/ERC2771Context.sol";
11 import "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol";
12 import "./libraries/OrderTypes.sol";
13 import "./Interfaces/ICollection.sol";
```

report | graph (this) | graph | inheritance | parse | flatten | funcSigs | uml | draw.io

```
1  // SPDX-License-Identifier: None
2  pragma solidity ^0.8.11;
3
4  import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
5  import "@openzeppelin/contracts/utils/structs/EnumerableSet.sol";
6  import "@openzeppelin/contracts/metatx/ERC2771Context.sol";
7  import "@openzeppelin/contracts/token/common/ERC2981.sol";
8  import "./utils/Whitelist.sol";
9  import "./utils/Authorizable.sol";
10 import "./MarketPlace/interfaces/IRoyaltyFeeRegistry.sol";
```

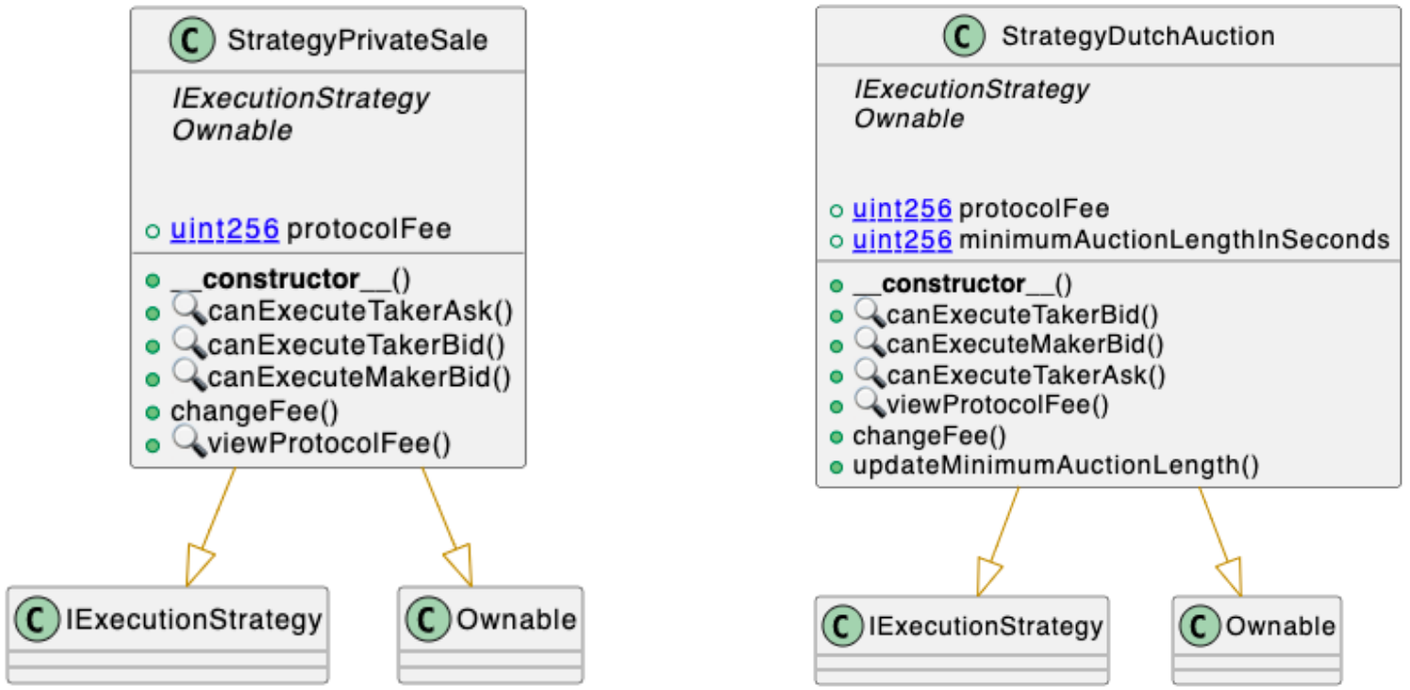
In-sufficient Event and Logging

2

In the contracts StrategyDutchAuction.sol (lines 111-114) and StrategyPrivateSale.sol (lines 72-75), there is a missing event emission after setting the value of protocolFee in the changeFee functions.

Events play a crucial role in recording significant occurrences and state changes, ensuring transparency and accountability on the blockchain. They provide real-time updates and notifications for external applications and user interfaces, allowing them to stay synchronized with the contract's activities.

By emitting an event after modifying the protocolFee value in the changeFee functions, the contract can effectively communicate this change to external entities and enable them to react accordingly. Emitting events also promotes contract composability and facilitates complex interactions between different contracts within the ecosystem.




```
function changeFee(uint256 _protocolFee) external override onlyOwner {
    require(protocolFee != _protocolFee, "Fee must be different");
    protocolFee = _protocolFee;
}
```

Re-Entry 2

A state variable is changed after a contract uses call.value. The attacker uses a fallback function—which is automatically executed after Ether is transferred from the targeted contract—to execute the vulnerable function again, before the state variable is changed.

That being said, not every reentrant behaviour can be called a bug. It can turn into a critical security issue if two conditions are met:
1) One contract calls another one while the former hasn't updated its state yet. This can be accidentally enabled by adding a function that makes an external call to another untrusted contract before the state changes get executed.
2) Once the malicious actor gains control over the untrusted contract, they can recursively call the original function over and over again, draining crypto wallets or sneaking into the code to make unsafe changes.

As a part of preventative techniques please ensure that all state changes happen before calling external contracts and use function modifiers that prevent re-entrancy. Although it is not a critical issue hence it has been mentioned as low level issue.

In the `create` function of `ERC1155Collection.sol` the state variable `_currentTokenID` is being set after external function calls to: _mint, onERC1155Received, updateRoyaltyInfoForNFT. See line 217.

Re-Entry

2

Similarly in create (ERC1155MusicCollection)
updateRoyaltyInfoForNFT
is finished & _currentTokenID is updated.

Since the state is getting updated after an external call, it is completely opposite to Ethereum guidelines.

We suggest that all state changes happen before calling external contracts and use function modifiers that prevent re-entrancy. Although it is not a critical issue hence it has been mentioned as low level issue.

Un-Satisfactory address checks

2

The initialize function in the Swap contract (swap.sol) lacks a zero-check on the _swapFundAddress variable before assigning its value to the swapFundAddress member.

A zero-check ensures that the provided address is not the zero address (address(0) or 0x0). By omitting this check, the contract allows the possibility of assigning the zero address to swapFundAddress,

The zero-check can be implemented by adding a simple condition before the assignment statement.

Similarly `updateLNQ` function in the SwapFund contract (SwapFund.sol) does not perform a zero-check on the `__newLnqAddress` variable before using it to initialize the new LNQ token.

```
function updateLNQ(address _newLnqAddress↑) external onlyOwner {
    require(!finalUpdate, "already updated");
    require(lnqToken != ILNQToken(_newLnqAddress↑), "already added");
    lnqToken = ILNQToken(_newLnqAddress↑);
}
```

```
constructor(
    address _aggregationRouter↑,
    address _swapFundAddress↑,
    address _forwarderAddress↑
) ERC2771Context(_forwarderAddress↑) {
    require(
        _aggregationRouter↑ != address(0),
        "Swap: Enter a valid aggregation"
    );
    require(
        _forwarderAddress↑ != address(0),
        "Swap: Enter a valid forwarder"
    );
    aggregationRouter = _aggregationRouter↑;
    feesPercentage = 50;
    swapFundAddress = _swapFundAddress↑;
```

In-Correct Estimation of amounts in Swap and during Burn

1

Solidity integer division might truncate. As a result, performing multiplication before division can sometimes avoid loss of precision.
In the case of JavaScript, following issue happens due to floating point errors. In Solidity we don't have floating points, but we get a similar issue and we can get "rounding errors".
For example : In Solidity the second operation would actually yield 2999. By doing all our multiplications first, we mitigate rounding related issues as much as possible. Hence it is suggested to do all multiplication before division otherwise the output can be in-correct.

```
» console.log((30 * 100 * 13) / 13)
  3000
← undefined
» console.log((30 / 13) * 100 * 13)
  2999.9999999999995
← undefined
»
```

```
function simpleSwapCheck(
    address srcToken↑,
    address dstToken↑,
    uint256 amount↑
) internal view returns (uint256, uint256) {
    if (srcToken↑ == lnqToken) {
        require(dstToken↑ == stableToken, "invalid dst token");
        amount↑ = (amount↑ /
            (10 ** (18 - IERC20Metadata(stableToken).decimals())));
    }
    if (srcToken↑ == stableToken) {
        require(dstToken↑ == lnqToken, "invalid dst token");
    }
    uint256 fee = (amount↑ * feesPercentage) / 10000;
    return (amount↑, fee);
}
```

Because of the above issue in SimpleSwapCheck, the amount will not be correctly calculated. It is highly suggested to do the multiplications first and then proceed to do the division.

In-Correct Estimation of amounts in Swap and during Burn

1

```
function burnAndClaim(
    uint256 _tokenId,
    address reciver,
    uint256 _ticketAmount
) external onlyLazyMint {
    require(reciver != address(0), "invalid address");
    EventInfo storage _event = events[_tokenId];
    require(
        _event.cancelStatus || _event.startTime > block.timestamp,
        "event is started and not canceled"
    );
    require(
        _event.ticketAmounts[reciver] >= _ticketAmount,
        "event ticket amount is less"
    );
    uint256 amount = (_event.userAmount[reciver] /
        _event.ticketAmounts[reciver]) * _ticketAmount;
    _event.totalTicketsAmount -= amount;
    _event.userAmount[reciver] -= amount;
    _event.ticketAmounts[reciver] -= _ticketAmount;
    IERC20(LNQToken).safeTransfer(reciver, amount);
    emit BurnAndClaim(_tokenId, reciver);
}
```

In the burnAndClaim function, the amount variable is calculated by dividing _event.userAmount[reciver] by _event.ticketAmounts[reciver] and then multiplying the result by _ticketAmount.
However, this calculation approach can produce unexpected results due to the order of operations in Solidity. In Solidity, when performing mathematical operations, it is crucial to consider the order of operations to achieve the desired outcome.

In this case, since the multiplication operation is performed after the division operation, the division result will be truncated to the nearest whole number before the multiplication occurs. This can result in inaccurate calculations and incorrect amounts being calculated for the burn and claim process.

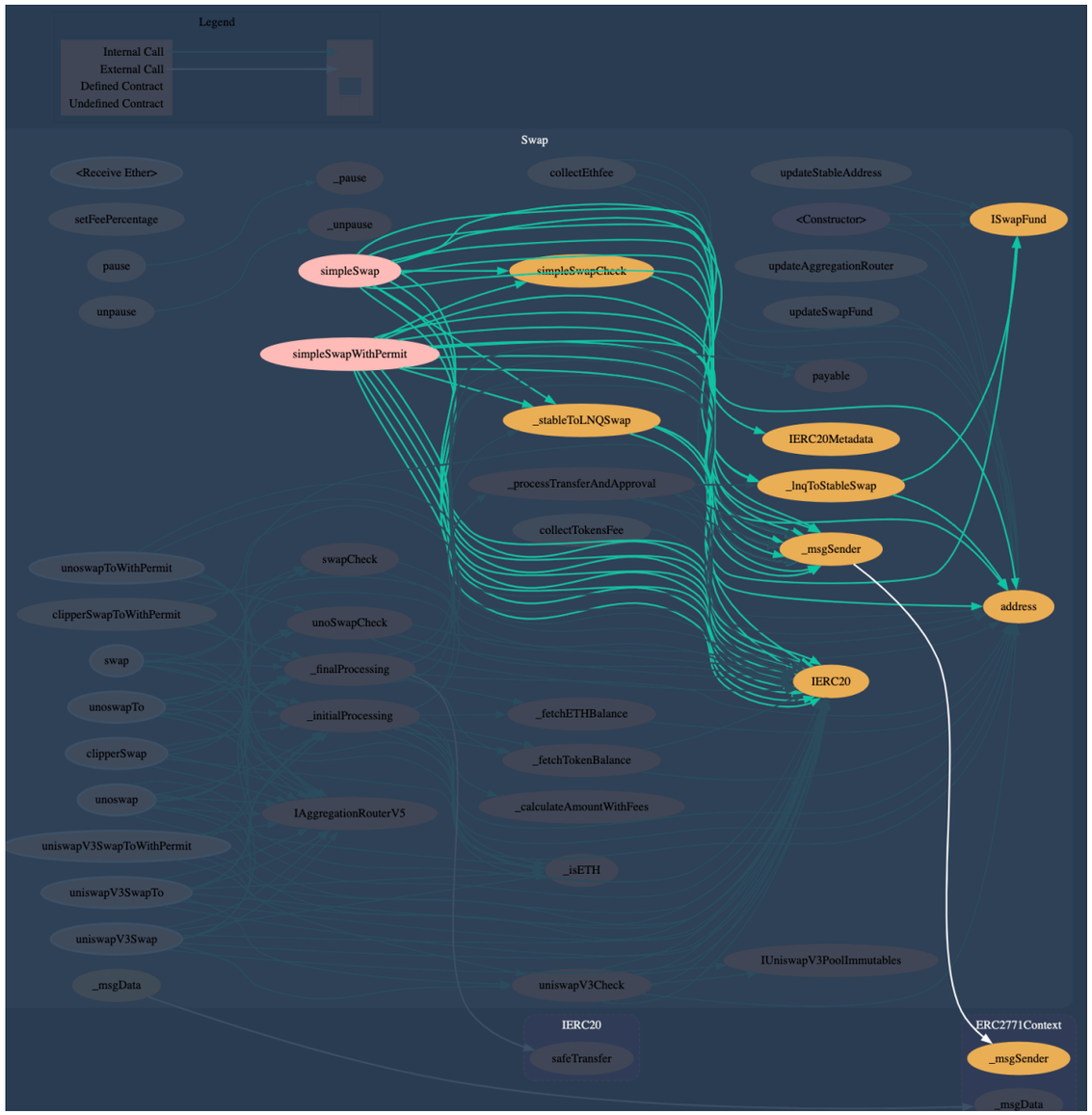
TECHFUND Audit Score

Smart contract Audit Report

Unnecessary check: Always true

1

In the simpleSwapWithPermit, the require statement checks if the amount argument is greater than or equal to zero.
However, since amount is declared as a uint256, an unsigned integer type, it is guaranteed to hold only non-negative values. The check for amount >= 0 is redundant and can be safely removed.

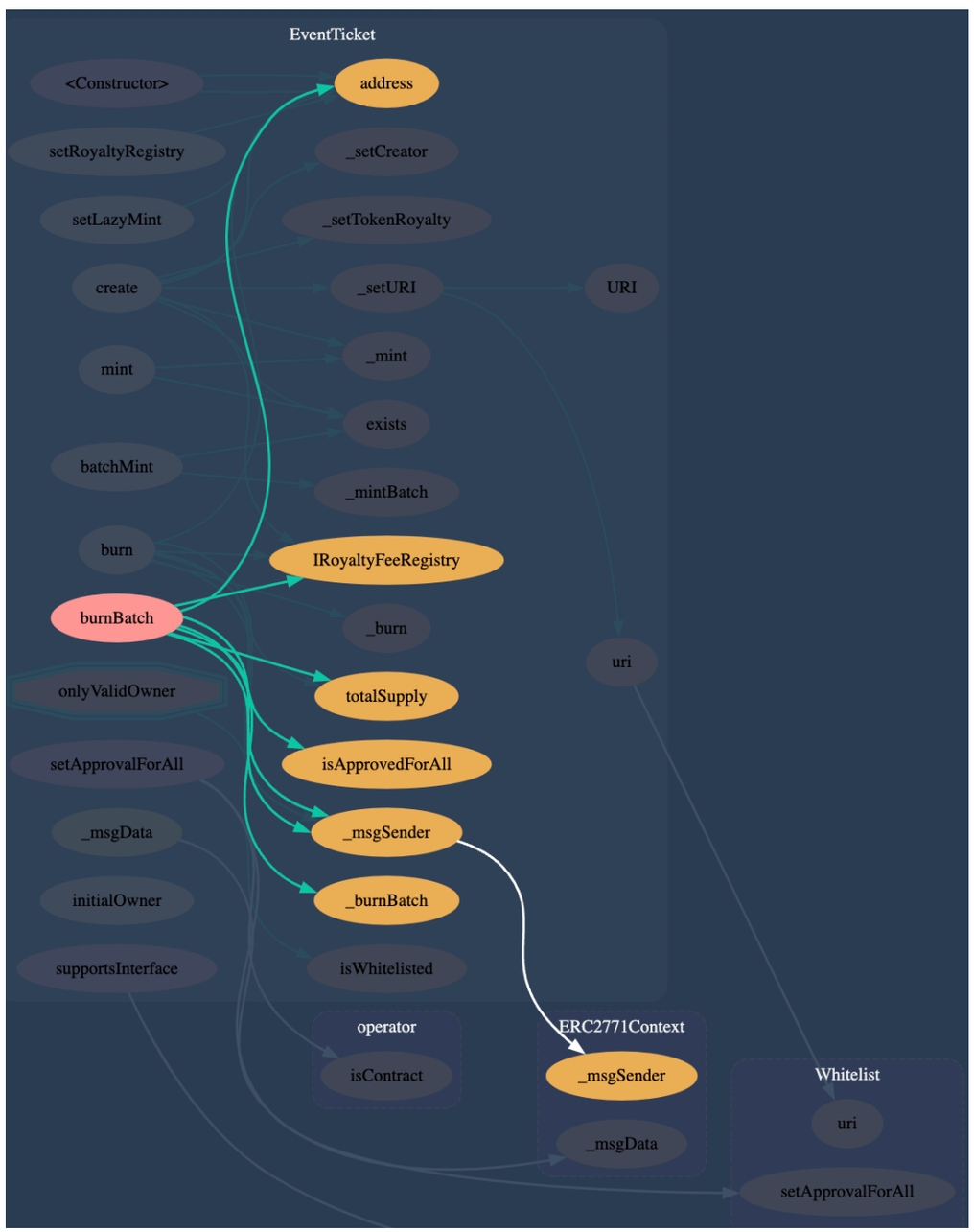



```
function simpleSwap(
    address srcToken,
    address dstToken,
    uint256 amount
) external nonReentrant whenNotPaused {
    require(srcToken != dstToken, "token should be different");
    require(amount > 0, "amount should not be zero");
    uint256 fee;
    (amount, fee) = simpleSwapCheck(srcToken, dstToken, amount);

    if (srcToken == lnqToken) {
        _lnqToStableSwap(_msgSender(), amount);
    } else {
        IERC20(stableToken).safeTransferFrom(
            _msgSender(),
            address(this),
            amount
        );
    }
    if (dstToken == lnqToken) {
        _stableToLNQSwap(_msgSender(), (amount - fee));
    } else {
        IERC20(stableToken).safeTransfer(_msgSender(), (amount - fee));
    }
    emit FeesCollected(
        _msgSender(),
        IERC20(srcToken),
        IERC20(dstToken),
        amount,
        fee
```

Costly Loops for external calls

1



A peculiar case of excessive gas abuse can be seen in scenarios where there's a parameter that is directly used inside loops without any validation on the number of times for which the loop runs.
The functions should validate that the users can't control the variable length used inside the loop to traverse a large amount of data. If it can't be omitted, then there should be a limit on the length as per the code logic.
Whenever loops are used in Solidity, the developers should pay special attention to the actions happening inside the loop to make sure that the transaction does not consume excessive gas and does not go over the gas limit.

In the `burnBatch` function of Tickets.sol: Calling a remote function in a loop may have potential implications.

It is recommended to either set an upper limit on the loop iteration or follow a "to-from" pattern to avoid potential issues like gas limitations, out-of-gas errors, or inefficiencies.

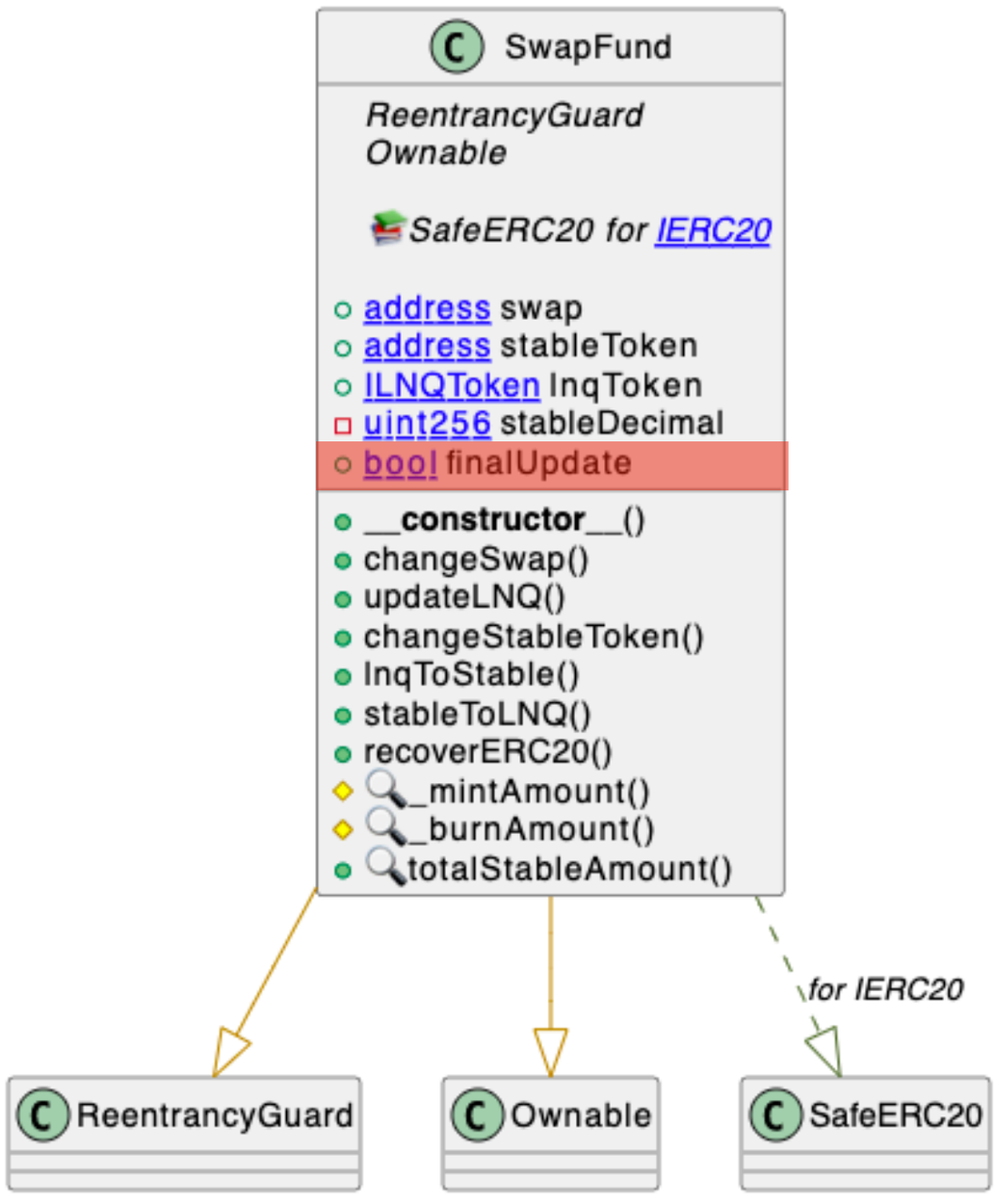
The "to-from" pattern refers to dividing the work into multiple transactions, where each transaction handles a subset of the loop iterations. For example, instead of looping through all the ids in a single transaction, you can split the work into multiple transactions, each processing a portion of the ids. This can help prevent gas limitations and ensure that the transaction does not exceed the gas limit set by the network.

Adding an upper limit to the loop iteration can also help mitigate potential issues. By limiting the number of iterations, you can ensure that the loop does not become too computationally expensive or consume excessive gas.

Costly Loops for external calls 1

```
function burnBatch(
    address account,
    uint256[] memory ids,
    uint256[] memory values
) public virtual {
    require(
        account == _msgSender() || isApprovedForAll(account, _msgSender()),
        "ERC1155: caller is not token owner or approved"
    );
    _burnBatch(account, ids, values);
    for (uint256 i = 0; i < ids.length; i++) {
        if (totalSupply(ids[i]) == 0) {
            IRoyaltyFeeRegistry(royaltyRegistry).removeRoyaltyInfoForNFT(
                address(this),
                ids[i]
            );
        }
    }
}
```

Unassigned state variable: results in failed check 1



In the `updateLNQ ` function in SwapFund.sol, the updateLNQ function is intended to update the LNQ token contract address.

However, the variable **finalUpdate** is not assigned a value anywhere in the contract, meaning it retains its default value of false. As a result, the condition !finalUpdate always evaluates to true, and the code execution proceeds without any checks.

Consequently, the statement lnqToken = ILNQToken(_newLnqAddress); will always be executed, allowing the LNQ token address to be updated without any restriction or validation.

To address this issue, you can consider adding the initialization of the finalUpdate variable in the contract's constructor or through a separate function. By properly setting the finalUpdate variable, you can introduce the desired logic and restrictions for updating the LNQ token contract address, ensuring that it is not updated multiple times or inappropriately.