Rubicon Finance Contracts: Mini Security Audit

Executive Summary

Type: DeFi

Auditor: Max Goodman, Security Researcher

Timeline: 1 day, June 21st, 2021 - June 22nd, 2021

Languages: Solidity

Methods: Computer-Aided Verification, Manual Review

Specifications: Rubicon Docs

Documentation Quality: Medium

Source Code:

Repository	Commit
RubiconDeFi/rubicon_protocol	361eca2b89689507c68096f4fba64c01a35d3e4d

Total Issues: 50

High Risk Issues: 0

Medium Risk Issues: 25

Low Risk Issues: 25

Findings (per Contract)

~~~~~RubiconMarket.sol~~~~~

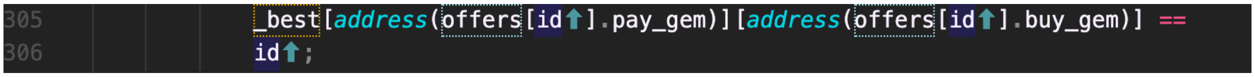
MG-1: Read of Persistent State Following External Call

Medium Risk

SWC-107

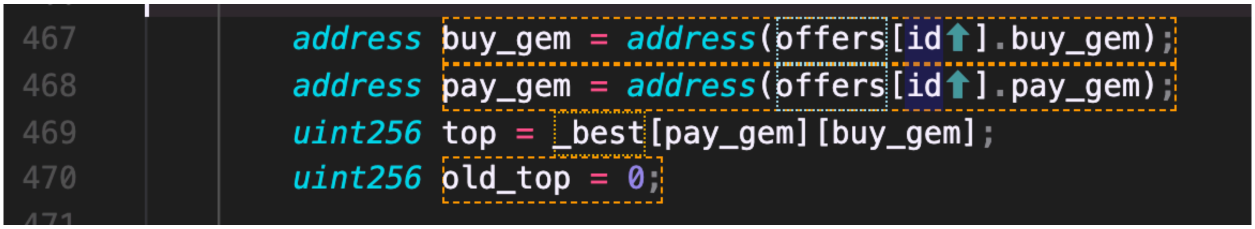
The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

First Instance:



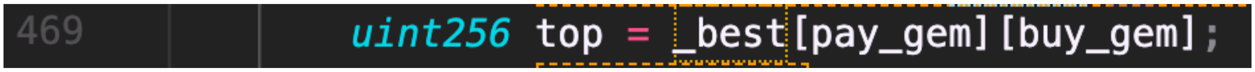
```
_best[address(offers[id].pay_gem)][address(offers[id].buy_gem)] ==
id;
```

Second Instance:



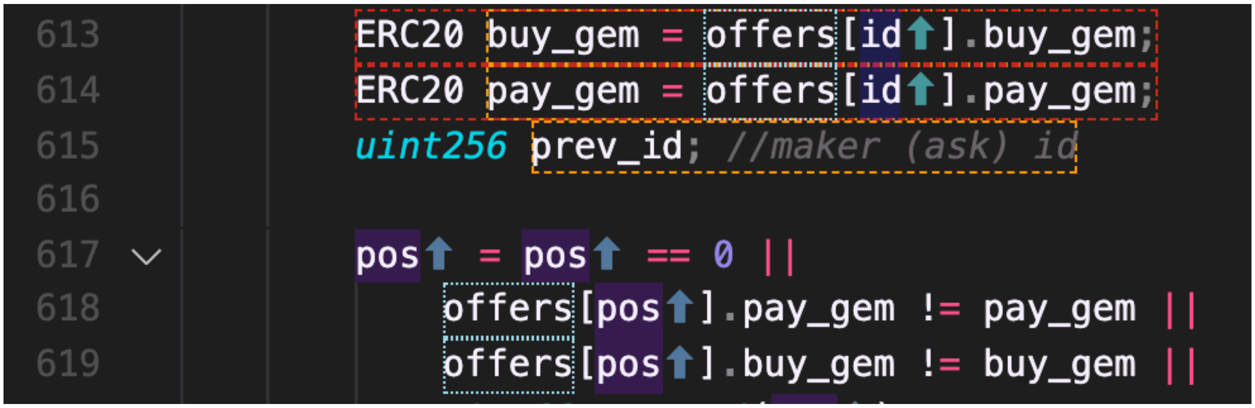
```
address buy_gem = address(offers[id].buy_gem);
address pay_gem = address(offers[id].pay_gem);
uint256 top = _best[pay_gem][buy_gem];
uint256 old_top = 0;
```

Third Instance:



```
uint256 top = _best[pay_gem][buy_gem];
```

Fourth Instance:



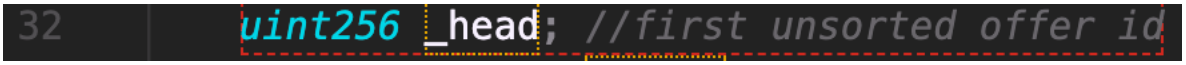
```
ERC20 buy_gem = offers[id].buy_gem;
ERC20 pay_gem = offers[id].pay_gem;
uint256 prev_id; //maker (ask) id

pos = pos == 0 ||
    offers[pos].pay_gem != pay_gem ||
    offers[pos].buy_gem != buy_gem ||
```

MG-2: State Variable Visibility Not Set

Low Risk

SWC-108

It is best practice to set the visibility of state variables explicitly. The default visibility for "_head" is internal. Other possible visibility settings are public and private.



```
    uint256 _head; //first unsorted offer id
```

~~~~~BathHouse.sol~~~~~

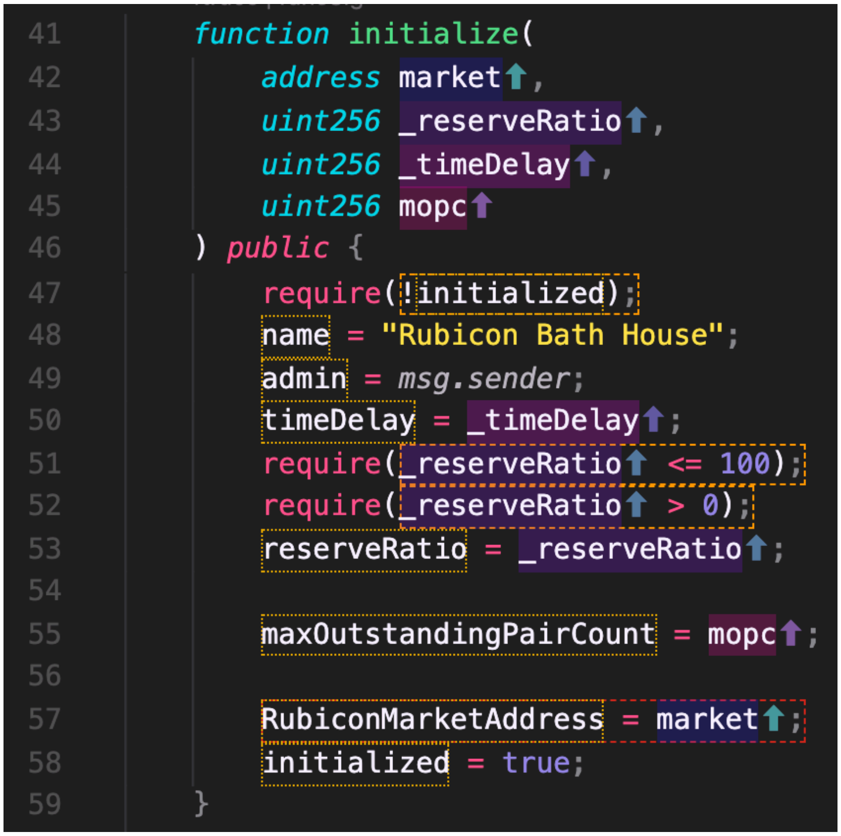
MG-1: Function could be marked as external

Medium Risk

SWC-000

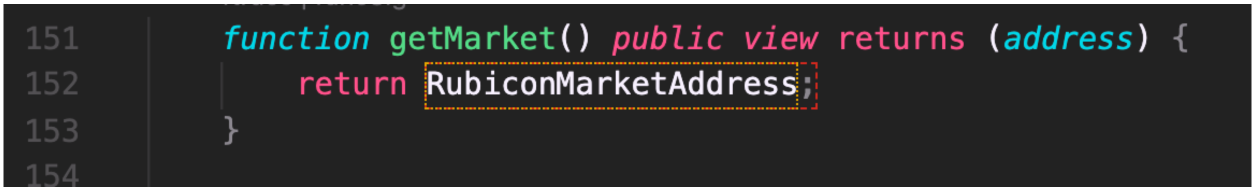
The function definition of each function below is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

First Instance:



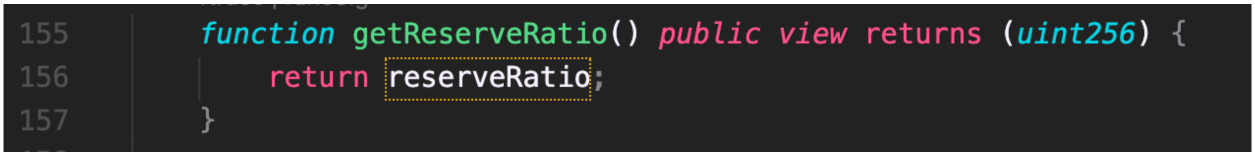
```
    function initialize(
        address market,
        uint256 _reserveRatio,
        uint256 _timeDelay,
        uint256 mopc
    ) public {
        require(!initialized);
        name = "Rubicon Bath House";
        admin = msg.sender;
        timeDelay = _timeDelay;
        require(_reserveRatio <= 100);
        require(_reserveRatio > 0);
        reserveRatio = _reserveRatio;

        maxOutstandingPairCount = mopc;

        RubiconMarketAddress = market;
        initialized = true;
    }
```

Second Instance:



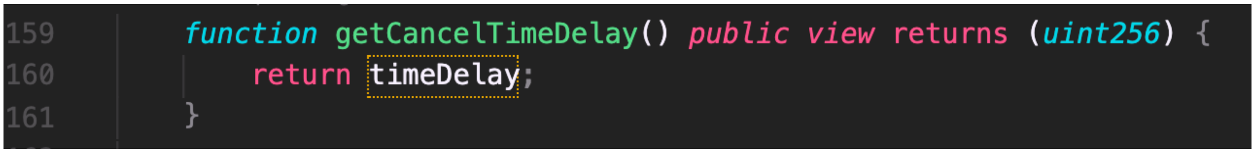
```
function getMarket() public view returns (address) {
    return RubiconMarketAddress;
}
```

Third Instance:



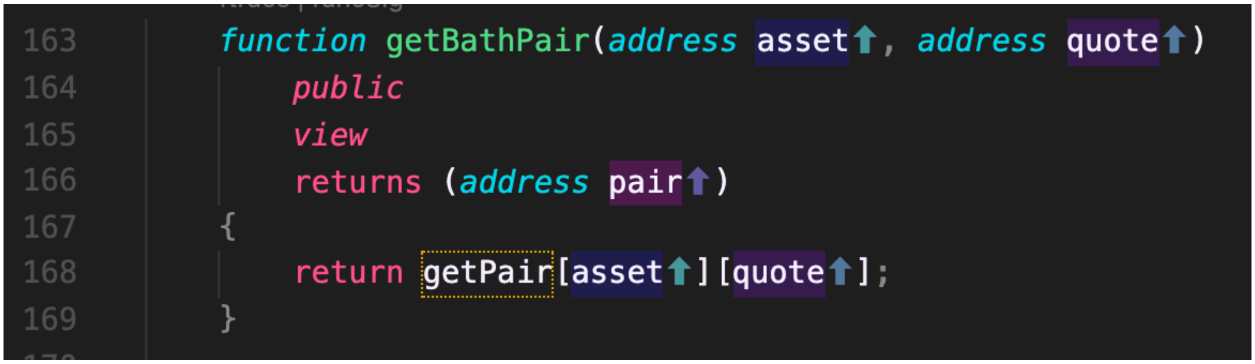
```
function getReserveRatio() public view returns (uint256) {
    return reserveRatio;
}
```

Fourth Instance:



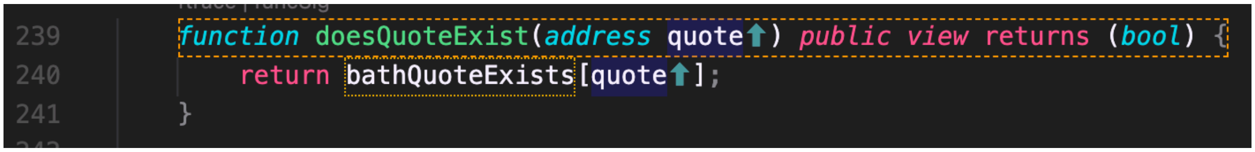
```
function getCancelTimeDelay() public view returns (uint256) {
    return timeDelay;
}
```

Fifth Instance:



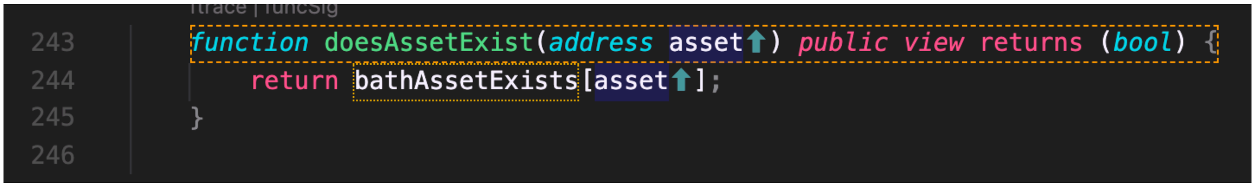
```
function getBathPair(address asset, address quote)
    public
    view
    returns (address pair)
{
    return getPair[asset][quote];
}
```

Sixth Instance:



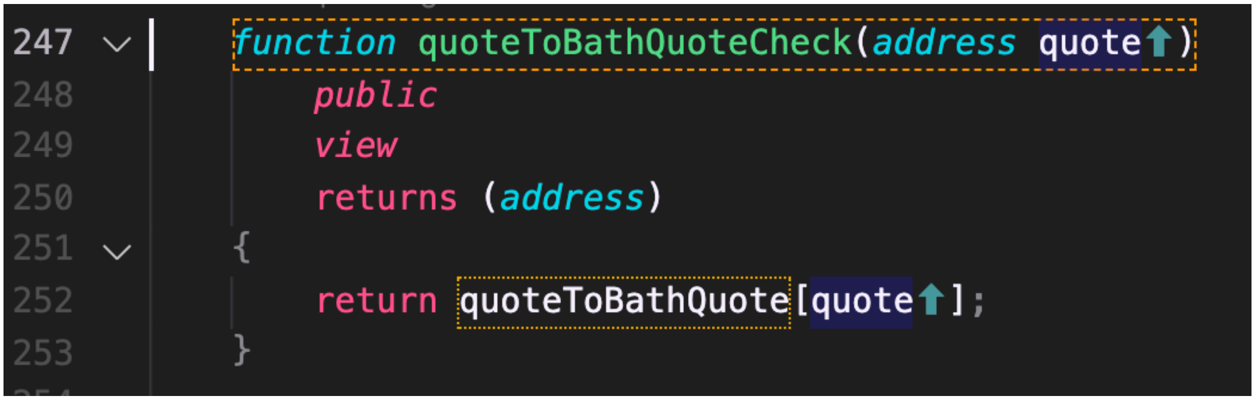
```
function doesQuoteExist(address quote) public view returns (bool) {
    return bathQuoteExists[quote];
}
```

Seventh Instance:



```
ftrace | funcSig
function doesAssetExist(address asset) public view returns (bool) {
    return bathAssetExists[asset];
}
```

Eighth Instance:



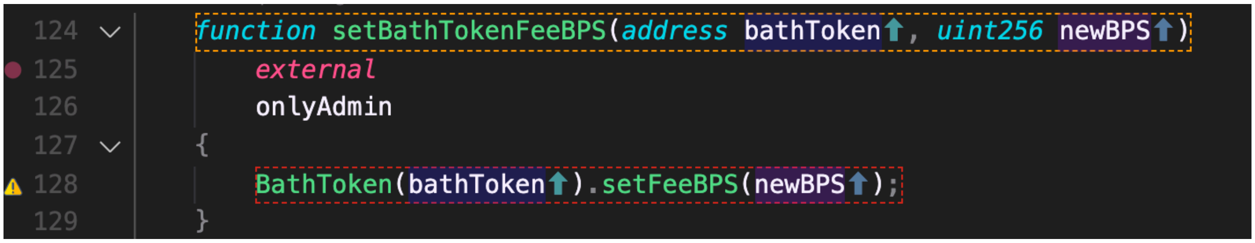
```
function quoteToBathQuoteCheck(address quote)
    public
    view
    returns (address)
{
    return quoteToBathQuote[quote];
}
```

MG-2: A call to a user-supplied address is executed

Low Risk

SWC-107

An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract. Reentering the contract in an intermediate state may lead to unexpected behavior. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.



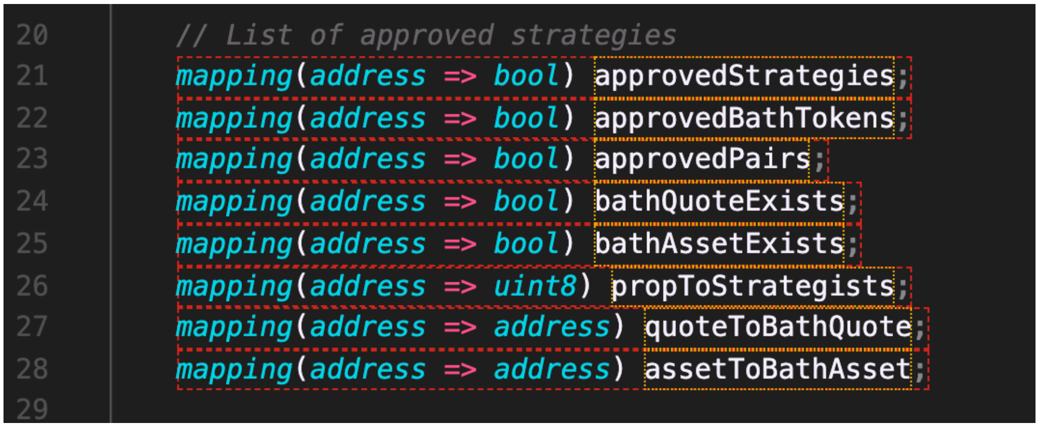
```
function setBathTokenFeeBPS(address bathToken, uint256 newBPS)
    external
    onlyAdmin
{
    BathToken(bathToken).setFeeBPS(newBPS);
}
```

MG-3: State variable visibility is not set

Low Risk

SWC-108

It is best practice to set the visibility of state variables explicitly. The default visibility for the mappings below is internal. Other possible visibility settings are public and private.



```
// List of approved strategies
mapping(address => bool) approvedStrategies;
mapping(address => bool) approvedBathTokens;
mapping(address => bool) approvedPairs;
mapping(address => bool) bathQuoteExists;
mapping(address => bool) bathAssetExists;
mapping(address => uint8) propToStrategists;
mapping(address => address) quoteToBathQuote;
mapping(address => address) assetToBathAsset;
```

~~~~~BathToken.sol~~~~~

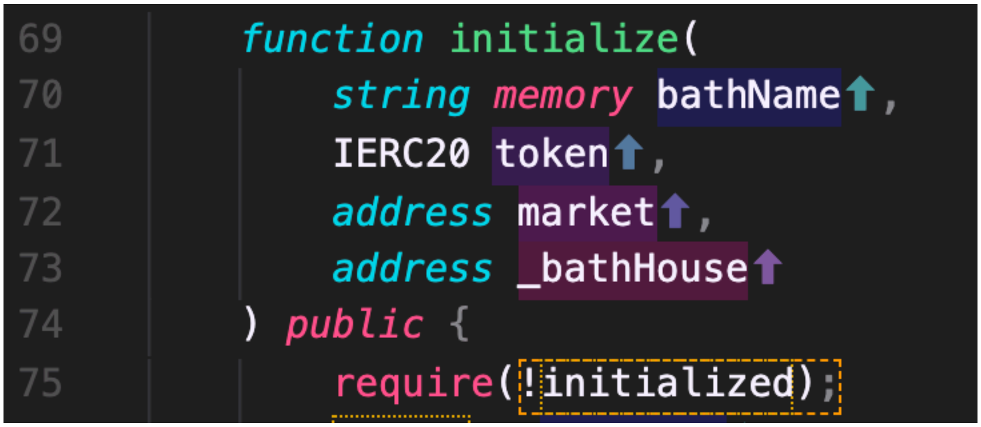
MG-1: Function could be marked as external

Medium Risk

SWC-000

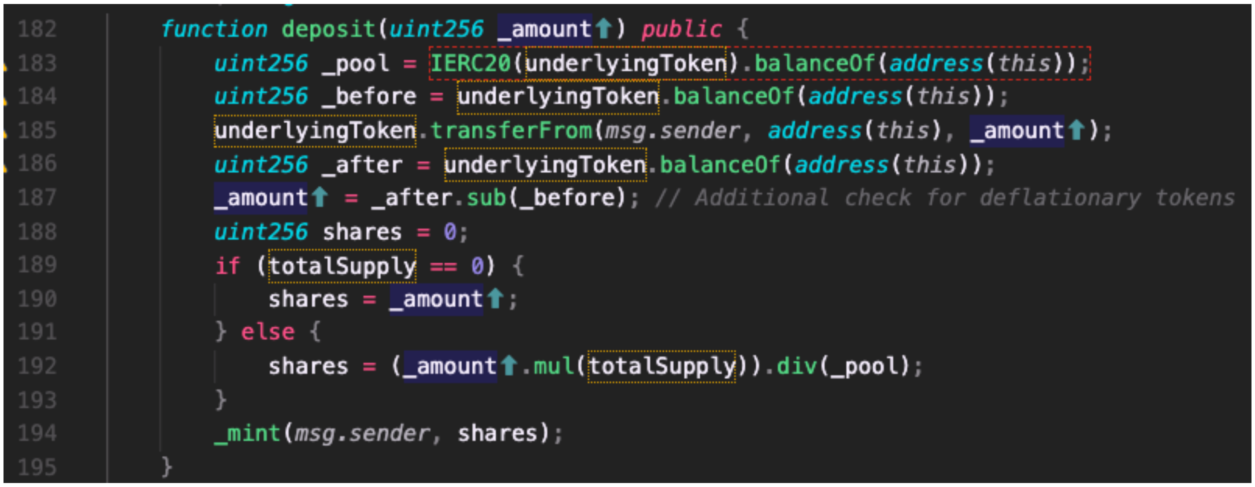
The function definition of functions below are marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as “external” instead.

First Instance:



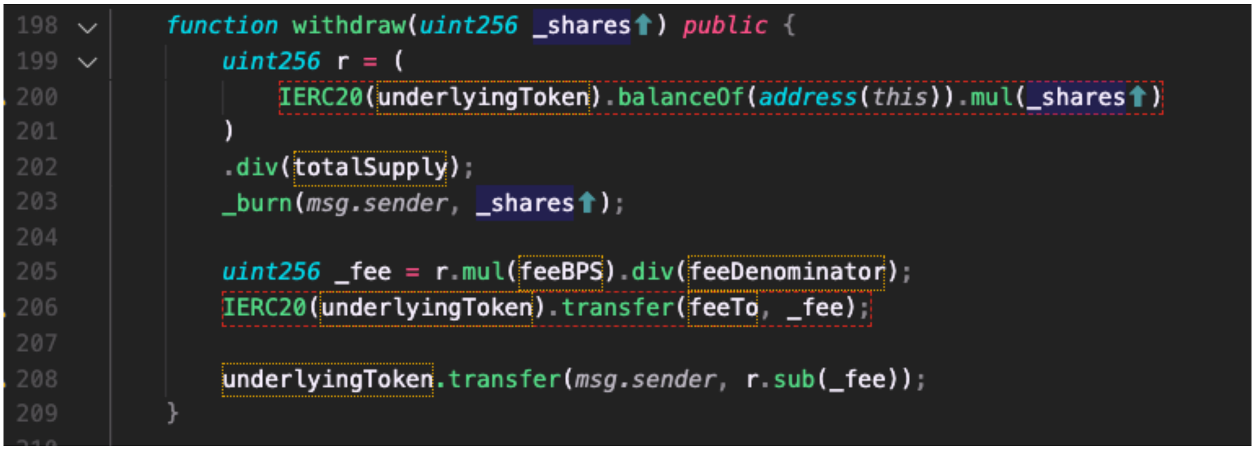
```
function initialize(
    string memory bathName,
    IERC20 token,
    address market,
    address _bathHouse
) public {
    require(!initialized);
```

Second Instance:



```
    function deposit(uint256 _amount) public {
        uint256 _pool = IERC20(underlyingToken).balanceOf(address(this));
        uint256 _before = underlyingToken.balanceOf(address(this));
        underlyingToken.transferFrom(msg.sender, address(this), _amount);
        uint256 _after = underlyingToken.balanceOf(address(this));
        _amount = _after.sub(_before); // Additional check for deflationary tokens
        uint256 shares = 0;
        if (totalSupply == 0) {
            shares = _amount;
        } else {
            shares = (_amount.mul(totalSupply)).div(_pool);
        }
        _mint(msg.sender, shares);
    }
```

Third Instance:



```
    function withdraw(uint256 _shares) public {
        uint256 r = (
            IERC20(underlyingToken).balanceOf(address(this)).mul(_shares)
        )
        .div(totalSupply);
        _burn(msg.sender, _shares);

        uint256 _fee = r.mul(feeBPS).div(feeDenominator);
        IERC20(underlyingToken).transfer(feeTo, _fee);

        underlyingToken.transfer(msg.sender, r.sub(_fee));
    }
```

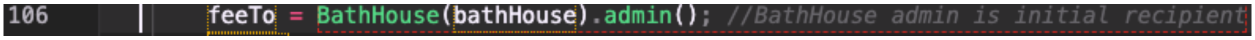
MG-2: Read of persistent state following external call

Medium Risk

SWC-107

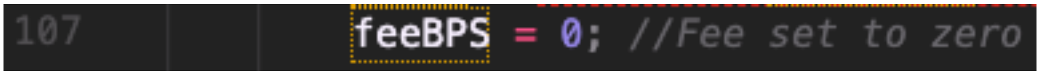
The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

First Instance:



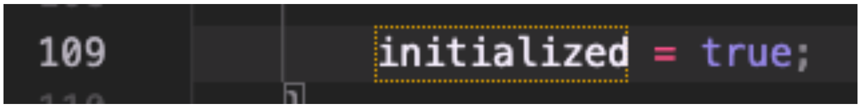
```
106 |     feeTo = BathHouse(bathHouse).admin(); //BathHouse admin is initial recipient
```

Second Instance:



```
feeBPS = 0; //Fee set to zero
```

Third Instance:



```
initialized = true;
```

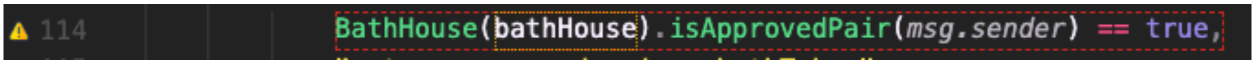
MG-3: Write to persistent state following external call

Medium Risk

SWC-107

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

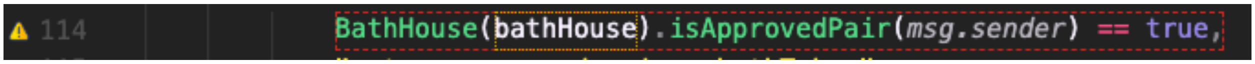
First Instance:



```
BathHouse(bathHouse).isApprovedPair(msg.sender) == true,
```

BathHouse ^

Second Instance:



```
BathHouse(bathHouse).isApprovedPair(msg.sender) == true,
```

msg.sender^

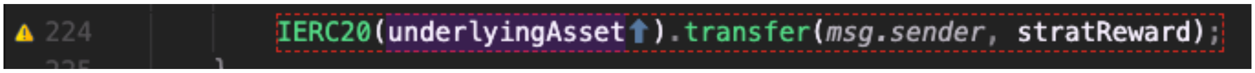
MG-4: A call to a user-supplied address is executed

Low Risk

SWC-107

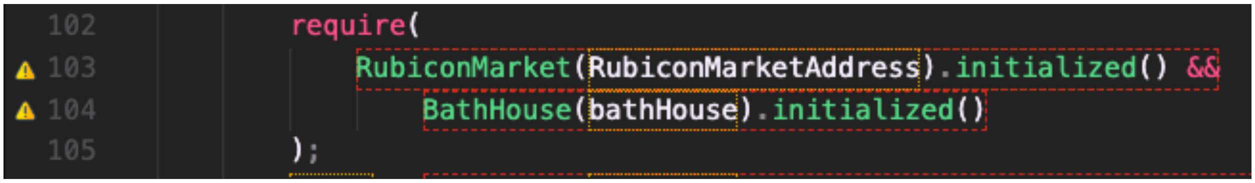
An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract. Reentering the contract in an intermediate state may lead to unexpected behavior. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.

First Instance:



```
        IERC20(underlyingAsset).transfer(msg.sender, stratReward);
```

Second Instance:



```
        require(
            RubiconMarket(RubiconMarketAddress).initialized() &&
                BathHouse(bathHouse).initialized()
        );
```

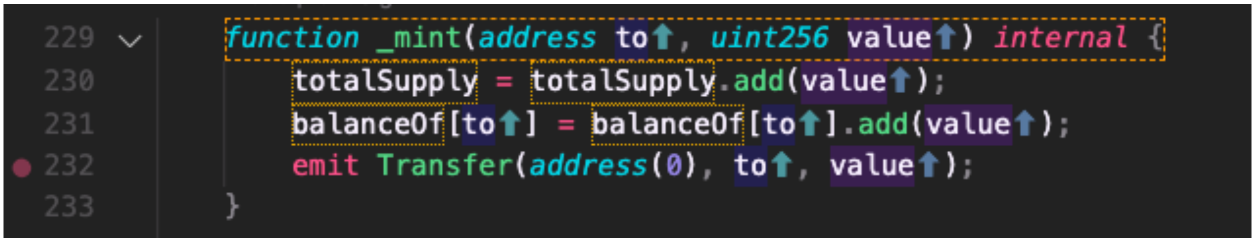
MG-5: Read of persistent state following external call

Low Risk

SWC-107

The contract account state is accessed after an external call to a fixed address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Line 232:



```
229     function _mint(address to, uint256 value) internal {
230         totalSupply = totalSupply.add(value);
231         balanceOf[to] = balanceOf[to].add(value);
232         emit Transfer(address(0), to, value);
233     }
```

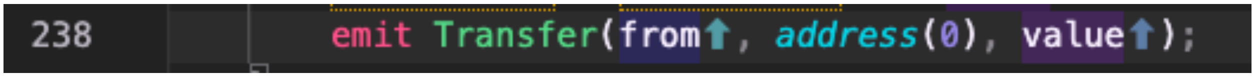
MG-6: Write to persistent state following external call

Low Risk

SWC-107

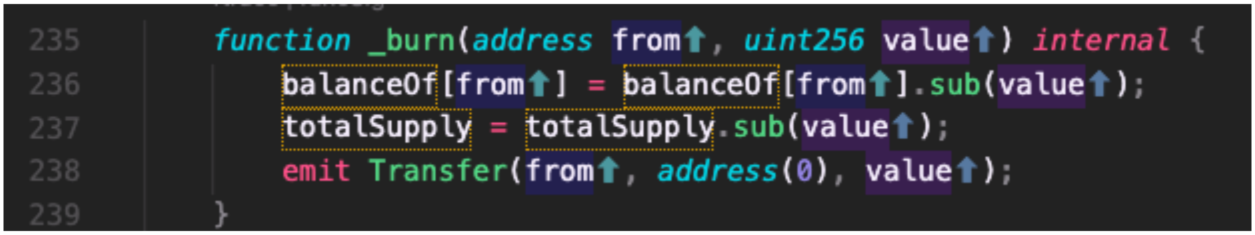
The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

First Instance:



```
        emit Transfer(from, address(0), value);
```

Second Instance:



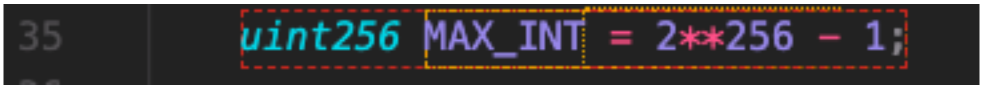
```
    function _burn(address from, uint256 value) internal {
        balanceOf[from] = balanceOf[from].sub(value);
        totalSupply = totalSupply.sub(value);
        emit Transfer(from, address(0), value);
    }
```

MG-7: State Variability is Not Set

Low Risk

SWC-108

It is best practice to set the visibility of state variables explicitly. The default visibility for "MAX_INT" is internal. Other possible visibility settings are public and private.



```
    uint256 MAX_INT = 2**256 - 1;
```

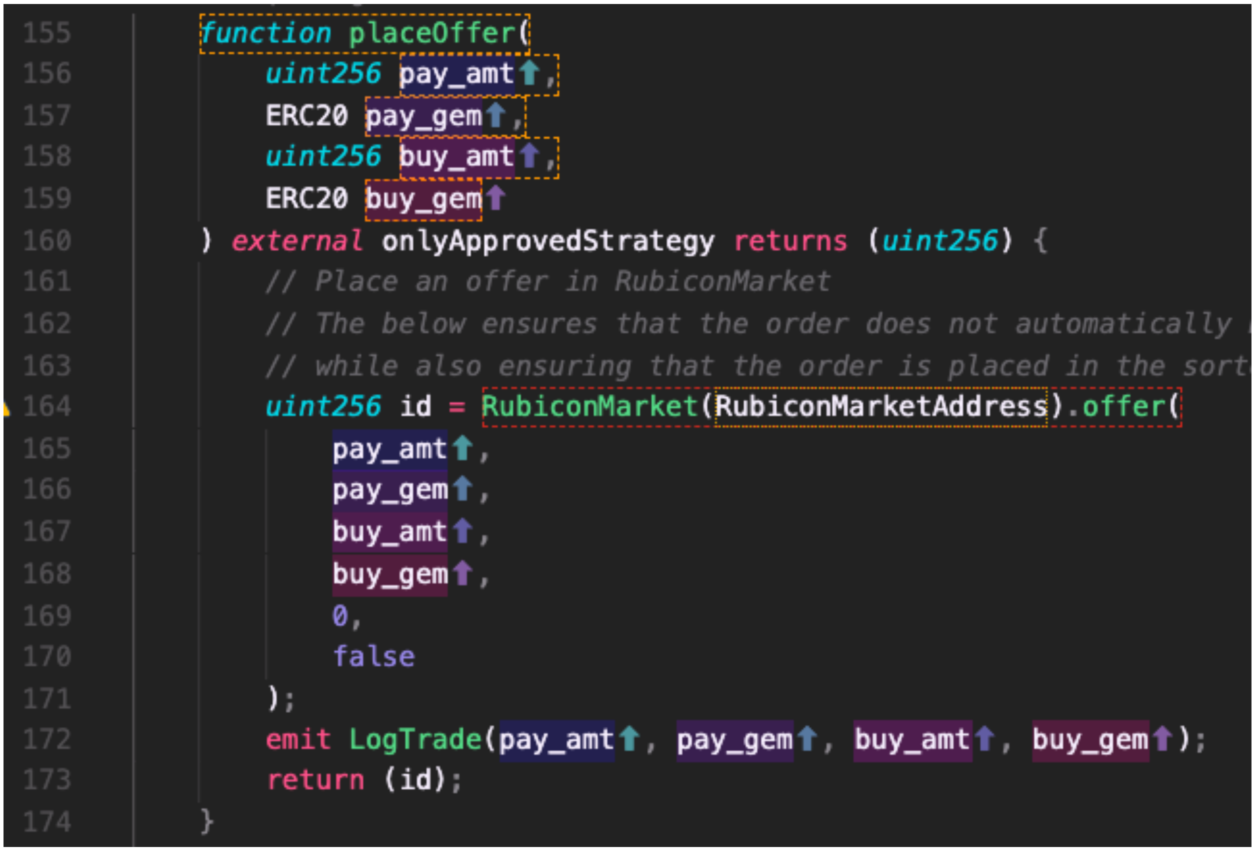
MG-8: Multiple calls are executed in the same transaction

Low Risk

SWC-113

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they’re part of your own codebase).

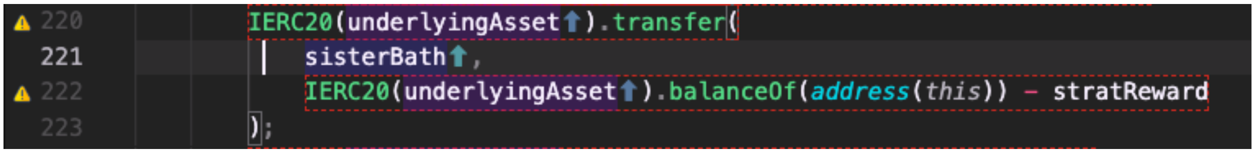
First Instance: placeOffer function declaration



```
    function placeOffer(
        uint256 pay_amt,
        ERC20 pay_gem,
        uint256 buy_amt,
        ERC20 buy_gem
    ) external onlyApprovedStrategy returns (uint256) {
        // Place an offer in RubiconMarket
        // The below ensures that the order does not automatically
        // while also ensuring that the order is placed in the sort
        uint256 id = RubiconMarket(RubiconMarketAddress).offer(
            pay_amt,
            pay_gem,
            buy_amt,
            buy_gem,
            0,
            false
        );
        emit LogTrade(pay_amt, pay_gem, buy_amt, buy_gem);
        return (id);
    }
```

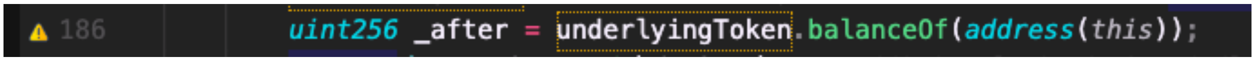
Second Instance: LogTrade (see above)

Third Instance:



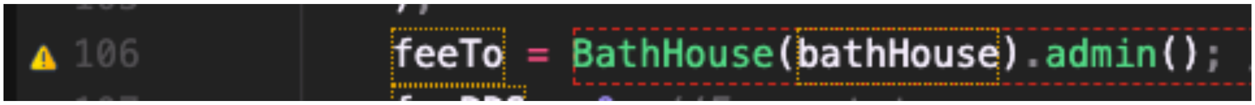
```
        IERC20(underlyingAsset).transfer(
            sisterBath,
            IERC20(underlyingAsset).balanceOf(address(this)) - stratReward
        );
```

Fourth Instance:



```
        uint256 _after = underlyingToken.balanceOf(address(this));
```

Fifth Instance:



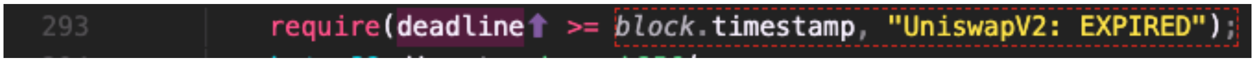
```
        feeTo = BathHouse(bathHouse).admin();
```

MG-9: A control flow decision is made based on the ‘block.timestamp’ environment variable

Low Risk

SWC-116

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.



```
        require(deadline >= block.timestamp, "UniswapV2: EXPIRED");
```

MG-10: Potentially unbounded data structure passed to builtin

Low Risk

SWC-128

Gas consumption in function "initialize" in contract "BathToken" depends on the size of data structures that may grow unboundedly. Specifically the "1-st" argument to builtin "keccak256" may be able to grow unboundedly causing the builtin to consume more gas than the block gas limit, effectively causing a denial-of-service condition.Consider that an attacker might attempt to cause this condition on purpose.

```
                address(this)
```

~~~~~BathPair.sol~~~~~

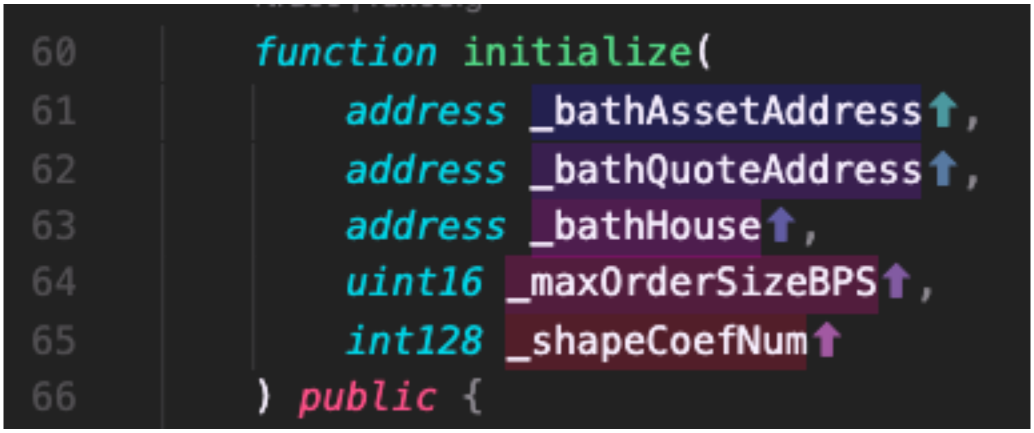
MG-1: Function could be marked external

Medium Risk

SWC-000

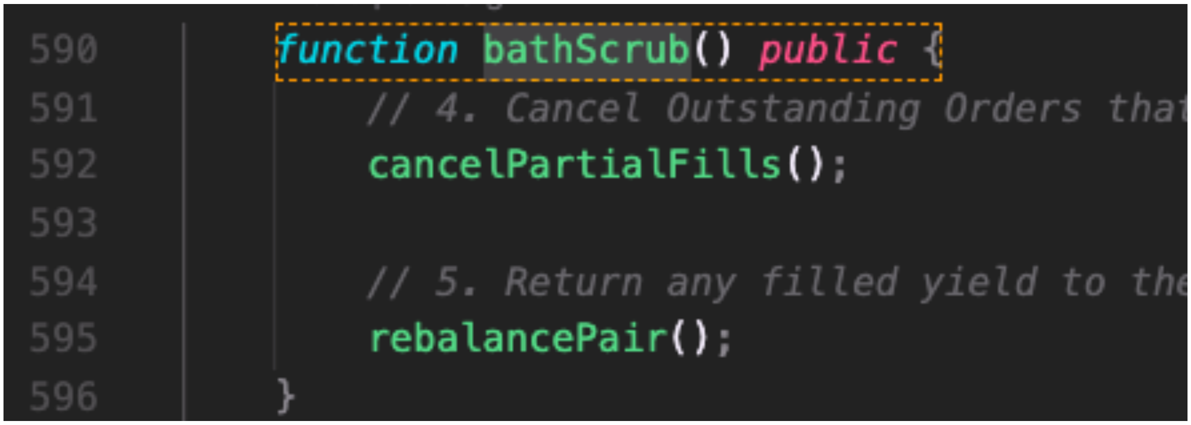
The function definition of the functions below are marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

First Instance:



```
    function initialize(
        address _bathAssetAddress,
        address _bathQuoteAddress,
        address _bathHouse,
        uint16 _maxOrderSizeBPS,
        int128 _shapeCoefNum
    ) public {
```

Second Instance:



```
    function bathScrub() public {
        // 4. Cancel Outstanding Orders that
        cancelPartialFills();

        // 5. Return any filled yield to the
        rebalancePair();
    }
```

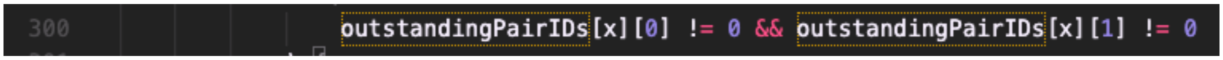
MG-2: Loop over unbounded data structure

Medium Risk

SWC-128

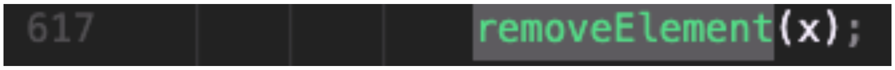
Gas consumption in these lines below depends on the size of data structures or values that may grow unboundedly. If the data structure grows too large, the gas required to execute the code will exceed the block gas limit, effectively causing a denial-of-service condition. Consider that an attacker might attempt to cause this condition on purpose.

First Instance:



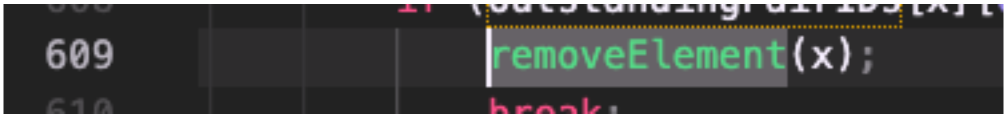
```
                outstandingPairIDs[x][0] != 0 && outstandingPairIDs[x][1] != 0
```

Second Instance:



```
                removeElement(x);
```

Third Instance:



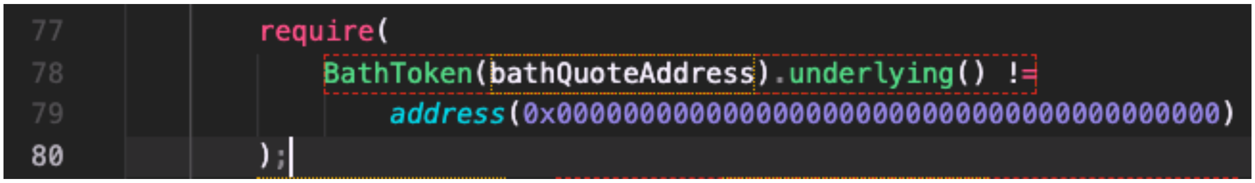
```
removeElement(x);
```

MG-3: Requirement Violation

Low Risk

SWC-123

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).



```
require(
    BathToken(bathQuoteAddress).underlying() !=
        address(0x0000000000000000000000000000000000000000)
);
```

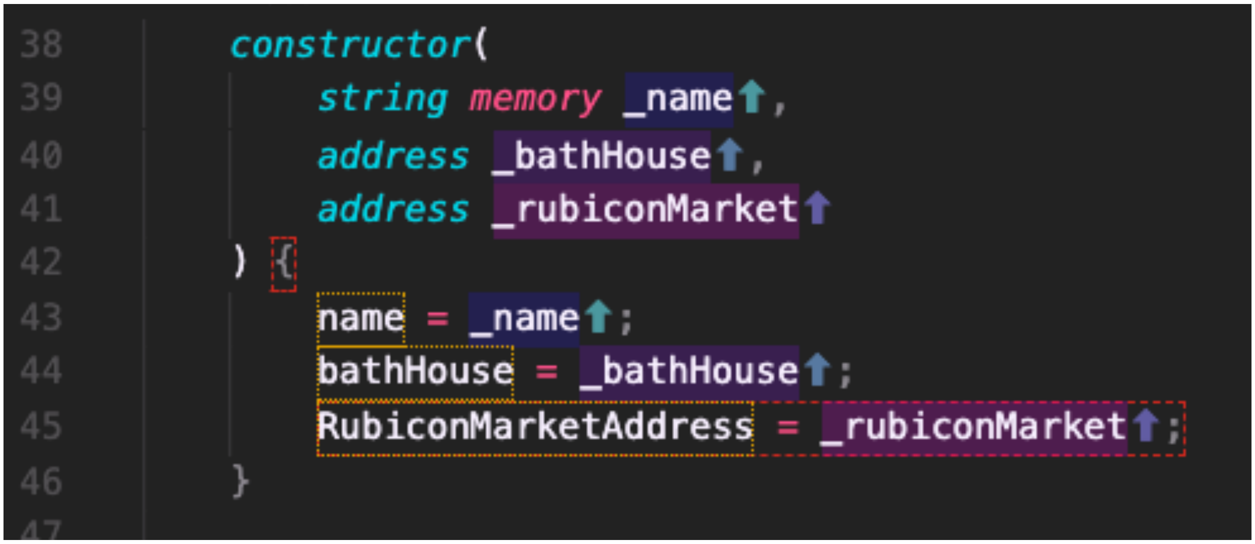
~~~~~PairsTrade.sol~~~~~

MG-1: Function visibility is not set.

Low Risk

SWC-100

The function definition of ‘constructor’ lacks a visibility specifier. Note that the compiler assumes "public" visibility by default. Function visibility should always be specified explicitly to assure correctness of the code and improve readability



```
constructor(
    string memory _name,
    address _bathHouse,
    address _rubiconMarket
) {
    name = _name;
    bathHouse = _bathHouse;
    RubiconMarketAddress = _rubiconMarket;
}
```

Conclusion

This mini audit was completed in one day from June 21, 2021-June 22nd, 2021. There were several medium risk issues revolving around potential re-entrancy attacks, unbounded data structures, etc. I recommend that the Rubicon team review the above issues in the respective solidity files and update their code accordingly with refactors. Overall, I recommend that fixes to these issues be implemented before production deployment.

Disclaimer: This audit does not guarantee against a hack. It is a snapshot in time of Rubicon Contracts according to the specific commit by one person. Any modifications to the code will require a new mini-audit. It is my recommendation that any production code go through a multi-person audit.